Introduction à Unity

Réalisation d’un shoot’em up

Qu’est-ce qu’un shoot’em up?

D’après Wikipédia : “*Un **shoot 'em up** (aussi écrit **shoot them up** ou contracté en **shmup** ; littéralement « abattez-les tous ») est un genre de jeu vidéo dérivé du jeu d'action dans lequel le joueur dirige un véhicule ou un personnage devant détruire un grand nombre d'ennemis à l'aide d'armes de plus en plus puissantes, au fur et à mesure des niveaux, tout en esquivant les projectiles adverses pour rester en vie.*”

C’est un style de jeu relativement simple à développer, aux nombreux représentants du genre (vous pourrez trouver pléthore d’exemples sur itch.io, genre de Youtube des jeux indépendants), et qui nous servira ici de point de départ pour démarrer avec Unity.

Notre but sera pour ce TD de créer un shoot’em up en suivant les règles du genre : piloter un vaisseau pouvant lancer des projectiles sur des nuées d’ennemis afin de venir à bout de celles-ci pour atteindre le boss final, et ~~sauver la galaxie~~ terminer le niveau.

Voici les différentes étapes que nous allons suivre dans ce TD pour la production du jeu :

Cœur du jeu

- Entrées du joueur et contrôle du vaisseau
- Gestion d’ennemis simples et des projectiles
- Gestion du Game Over

Base de l’interface

- Informations du joueur (vie, score, etc….)
- Menu pause

Complétion du jeu

- Ecran titre / écran de fin
- Gestion de boss
- Vagues d’ennemis
- Bullet pattern
- Capacités spéciales et armes
- Gestion du son

Cœur du jeu

Entrées du joueur et contrôle du vaisseau

Pour commencer, faisons en sorte de pouvoir piloter un vaisseau. Dans la hiérarchie de la scène, ajoutez un nouvel objet 3D (un cube par ex.). Dans l'inspecteur, sélectionner l'objet et ajouter lui un nouveau composant, un nouveau script, de type "`Player`". Afin de pouvoir contrôler le vaisseau, et limiter ses mouvements au champ de la caméra, nous allons avoir besoin de certains paramètres.
Un champ `m_MainCamera`, qui récupérera la caméra,
Un champ `m_VerticalSpeed`, qui stockera la vitesse verticale du vaisseau,
Un champ `m_HorizontalSpeed`, qui stockera la vitesse horizontale du vaisseau.
Ces trois champs peuvent être marqués de l'attribut `[SerializeField]`, afin que malgré leur statut privé, ils puissent être visibles dans l'inspecteur.

Pour récupérer la valeur de la caméra, deux possibilités : soit la récupérer depuis la méthode `Awake()` via la classe `Camera`, soit la glisser-déposer dans l'inspecteur Unity.

Reste enfin à gérer les mouvements du vaisseau. Pour cela, créons une méthode `PlayerControl()` qui sera appelée depuis la méthode `Update()` du script. Dans `PlayerControl()`, commençons par vérifier si une clé d'entrée est pressée, via la méthode `GetKey()`, de la classe `Input`, et de l'énumération `KeyCode`. Si par exemple la clé `KeyCode.LeftArrow`, on voudra déplacer le vaisseau vers la gauche. En accédant à la position du vaisseau via la `transform` de celui-ci, ajoutez ou retirez un facteur de la direction voulu, et de la vitesse de l'axe considéré. Il est important, dans les calculs de prendre en compte `Time.deltaTime`, pour éviter d'avoir des différences de déplacement en fonction de votre machine. En retournant sur Unity, et en pressant Play, le vaisseau devrait se déplacer à l'écran, mais pourra outrepasser les limites de l'écran.

Pour limiter cela, pour chaque test sur la direction à suivre, nous ajouterons un test sur la position du vaisseau par rapport aux limites de l'écran. En comparant la position projetée du vaisseau sur l'écran via la méthode `WorldToScreenPoint()` de la classe `Camera` aux limites de taille de l'écran via la classe `Screen`, on pourra aisément limiter la position du vaisseau dans le cadre de l'écran.

Gestion d'ennemis simples et des projectiles

Projectiles

Commençons par ajouter à la scène un nouvel objet 3D (capsule par ex.), qui servira de projectile. Sur cet objet, ajoutons un nouveau script de type `Bullet`. Afin de convertir cet objet en `prefab`, et pouvoir l'instancier à la demande, sélectionnez l'objet dans la scène, et glissez le vers la fenêtre Project.

Dans le script `Bullet`, un nouveau champ sérialisé sera nécessaire, afin de gérer la vitesse de déplacement des projectiles. Également, une méthode de gestion du mouvement du projectile, appelée chaque frame depuis `Update()`, sera nécessaire. Pour déplacer le projectile, utilisez un moyen similaire à celui utilisé dans le `Player`. Enfin, pour limiter les déplacements du projectile à l'écran, ajouter une gestion du projectile similaire à celle du vaisseau sur la hauteur de l'écran, et détruire le projectile via la méthode `Destroy()` si celui-ci est hors champ.

Afin de contrôler le lancement des projectiles, deux nouveaux champs sérialisés seront à ajouter dans `Player` : un flottant pour gérer la cadence de tir et un `GameObject` dans lequel nous mettrons le `prefab` des projectiles, depuis l'inspecteur du vaisseau. Enfin, il faudra créer un champ de type `Stopwatch` de la classe `System.Diagnostics` pour minuter le lancement des projectiles.
Dans la méthode `Awake()` du `Player`, instanciez la `Stopwatch`, et démarrez la.
Dans la méthode `PlayerControl`, il faudra ajouter une nouvelle section qui réagira à l'entrée de la commande de tir (ex. `KeyCode.Space`) et vérifier en parallèle si le dernier tir a bien été effectué au-delà de la cadence de tir. Si ces deux conditions sont respectées, on peut créer une instance du projectile via la méthode `Instantiate()`, modifier la position du projectile pour que celui-ci apparaisse au même endroit que le vaisseau, et enfin redémarrer le minuteur de la `Stopwatch`, afin de permettre un prochain tir.

Ennemis

Pour la gestion des ennemis, deux nouveaux éléments seront nécessaires dans la scène :

- un objet vide, qui servira de logique pour l'apparition des ennemis,
- une sphère qui servira de `prefab` pour les ennemis.

Sur l'objet vide, créez un script `EnemiesManager`, et un script `Enemy` sur le `prefab`.

Dans `EnemiesManager`, deux champs sérialisés seront nécessaires : un pour récupérer le prefab des ennemis, et un pour gérer le délai d'apparitions entre deux ennemis.
Pour gérer l'apparition aléatoire des ennemis, nous allons utiliser une `coroutine` (pensez à la démarrer dans la méthode `Start()`, via la méthode `StartCoroutine()`). Dans cette coroutine, tant que l'application est active (`Application.isPlaying`) faites apparaître un ennemi en haut de l'écran (`Camera.ScreenToWorldPoint` pourra vous y aider). Attention ici à prendre en compte la position de la caméra en profondeur, pour éviter de faire apparaître les ennemis dans le plan de la caméra (mieux vaut les faire apparaître dans le plan du vaisseau). Une fois l'ennemi instancié, attendre le délai défini plus tôt avant l'apparition du prochain ennemi.

Pour le comportement des ennemis, dans le script `Enemy`, un seul champ sérialisé sera nécessaire, pour gérer la vitesse des ennemis. Leurs déplacements seront très simples pour commencer : à l'inverse des missiles, ils devront descendre le long de l'écran, et être détruits s'ils touchent le bas de celui-ci.

A présent que notre vaisseau se déplace, peut tirer des projectiles, et possède des ennemis sur lesquels faire feu, reste à implémenter la logique des collisions.

Gestion du Game Over

La première chose qui sera nécessaire à la gestion du Game Over et du score du joueur, sera l'ajout d'un `Rigibody` à chaque objet qui va devoir interagir avec un autre via une collision (vaisseau, projectile et ennemi). Un `Rigibody` permettant la gestion d'un objet via le moteur physique de Unity, il est important de contraindre la gestion de la position et de la rotation de chaque `Rigidbody` : dans la section "Constraints", sélectionnez toutes les cases pour bloquer le calcul de ces axes par le moteur physique, puisque que nous les gérons nous même dans le code.
Depuis l'inspecteur de ces éléments, il faudra également définir leur `tag` afin de simplifier la reconnaissance des objets entre eux au moment de la collision. Définissez deux nouveaux tags "Enemy" et "Bullet" ("Player" étant un des tags par défaut de Unity), et associez-les à leur objets respectifs.

Comme et les ennemis et le joueur doivent posséder des points de vie (PV), nous allons créer une nouvelle classe `Entity`, dont `Player` et `Enemy` hériteront. Cette classe devra posséder un champ sérialisé pour définir les PVs max de l'entité, et d'une propriété pour les PVs actuelles de l'entité, accessible publiquement en lecture, mais protégée en écriture.
Également, la méthode `Awake()` pourra être virtualisée afin d'initialiser la valeur courante des PVs depuis `Entity`, et devra donc être surchargée dans `Player`.

Chacune des trois classes Player, Bullet et Enemy va maintenant devoir implémenter la méthode `OnCollisionEnter()` afin de gérer les contacts entre elles. Note importante : cette méthode peut être implémentée par n'importe quelle `MonoBehaviour`, mais elle ne sera appelée que si au moins l'un des objets concernés par la collision possède un `Rigibody` et que celui-ci n'est pas défini comme étant Kinematic.

Dans `Player`, `OnCollisionEnter()` devra vérifier si l'objet en collision est un ennemi (via son tag), réduire les PVs du joueur en fonction, et vérifier que les PVs du joueur ne sont pas tombés sous la valeur fatidique de zéro. Si c'est le cas, détruisez le vaisseau, et informez le joueur de sa défaite.
Dans `Enemy`, si l'objet en contact est le joueur, détruisez l'ennemi, et si c'est un projectile, gérez les PVs de l'ennemi en fonction.
Enfin, dans `Bullet`, avant d'implémenter `OnCollisionEnter()`, il sera nécessaire de créer une action à effectuer quand le projectile touche un ennemi, afin de remonter l'information au joueur (`public event Action`). Dès que le projectile entre en contact avec cette action, invoquez là, puis détruisez le projectile. Pour que ce délégué soit fonctionnel, dans Player, ajoutez une méthode `OnBulletHit()`, qui augmentera le score du joueur à chaque appel. A l'endroit où est instancié le projectile, abonné la méthode `OnBulletHit` au délégué `OnHit` de `Bullet`.

En retournant dans Unity et en pressant Play, vous devriez être en mesure de pouvoir jouer à un petit shoot'em up : Félicitations !

Base de l'interface

Informations du joueur

Afin de faire un retour à l'utilisateur sur sa barre de vie et son score, il va nous falloir créer une interface. Commencez par ajouter un `Canvas` dans la scène, et y ajouter un nouveau script `UserInterface`. Dans ce `Canvas`, nous allons ajouter un `Text` pour l'affichage du score, et un `Slider` pour l'affichage des PV. Le `Slider` contient de base une poignée qui ne nous sera pas nécessaire, désactivez-la parmi les enfants du `Slider` (Handle Slide Area). Également, la poignée couvrant normalement une partie du remplissage, celui-ci ne remplit pas le fond entièrement quand la barre est pleine : dans le `RectTransform` de Fill Area, définissez la valeur Right à 5.
Placez la barre de vie dans un coin de l'écran, et le score de l'autre.
Pour finir avec l'interface, il faudra également prévoir un affichage du Game Over, et un moyen de relancer le jeu. Dans le `Canvas`, ajoutez en enfant un `Text` qui affichera "Game Over", et un bouton pour relancer le niveau.
Enfin, rajoutez dans la scène un objet qui servira de gestionnaire de jeu : sur un nouvel objet vide, à la racine de la scène, ajouter un nouveau script `GameManager`.

Du côté des scripts, commençons par adapter `Player`. Afin de retourner les informations du joueur à l'interface, seulement quand nécessaire, nous allons créer deux délégués (comme pour `OnBulletHit`) : `OnHPChange`, et `OnScoreChange`. Il sera également nécessaire de créer une propriété publique qui permet de récupérer la valeur max des PVs du vaisseau. `OnHPChange` devra être invoquée à chaque fois que les PVs sont modifiés, en précisant leur valeur, et de même pour `OnScoreChange` avec le score.

Dans `UserInterface`, nous allons commencer par implémenter cinq champs sérialisés : un pour récupérer le `GameObject` représentant le Game Over, un pour le `Button` de relance du niveau, un pour le `Text` du score, un pour le `Slider` de la barre de vie, et enfin un pour récupérer le `Player`.
Deux méthodes seront ici nécessaires : une qui s'abonne à l'événement de changement de score du `Player`, et qui changera la valeur afficher en fonction (l'usage de chaîne interpolée pour ici vous être utile) et une qui s'abonne aux changements des PVs, qui changera la valeur du `Slider` en fonction, et qui activerait l'objet Game Over si nécessaire.
Dans `Awake()` de `UserInterface`, faites les abonnements nécessaires .
Dans la méthode `Start()` de `UserInterface`, mettez à jour la valeur max du `Slider` représentant les PVs, et de même pour la valeur courante.

La classe `GameManager` servira pour l'instant à simplement recharger le niveau courant, mais pourra être utile à l'avenir pour la gestion de plusieurs niveaux, d'un menu pause, d'un écran titre, etc… Afin de rendre cette classe facilement accessible par l'ensemble des scripts, nous allons la faire hériter de la classe `MonoBehaviourSingleton<T>`, qui permettra d'utiliser le patron de conception singleton (cette classe sera fournie en base code, si vous voulez des explications à son sujet, contactez moi !).

Pour l'instant, `GameManager` implémentera simplement une méthode `ResetLevel()`, qui utilisera la classe `SceneManager` de Unity afin de charger le niveau courant (`SceneManager.LoadScene()` et `SceneManager.GetActiveScene()`).

De retour dans UserInterface, dans `Awake()`, sur le bouton de relance du niveau, ajoutez un nouveau listener sur la méthode `onClick` du bouton, et appelez la méthode `ResetLevel()` du `GameManager`, en passant par son singleton.

Plus qu'à renseigner les champs sérialisés dans Unity, et vous aurez une interface fonctionnel pour votre shoot'em up !

Menu pause

Dernière étape avant d'améliorer le gameplay de base : réalisons un système de pause.
Dans le `Canvas` déjà présent dans la scène, ajoutez un `Panel` qui servira de fond pour le menu pause. Sur ce `Panel`, ajoutez un text qui indique que le jeu est en pause.

Dans le script `UserInterface`, ajoutez un champ sérialisé qui permettra de récupérer le `GameObject` du `Panel` que l'on vient de créer. Ajoutez ensuite une méthode `TogglePausePanel(bool pause)`, qui permettra d'activer ou non le `Panel`. Enfin, faites hériter UserInterface de `MonoBehaviourSingleton` (attention, il faudra penser à modifier `Awake()` pour ne pas écraser la méthode `Awake()` de `MonoBehaviourSingleton`).

Dans le script `GameManager`, ajoutez un champ public à l'écriture privée, qui servira à la récupération du statut "pause" du jeu et qui sera par défaut à `true`. Cette variable pourra être utile à l'avenir.
Créez ensuite une méthode `Play(bool play)`, qui permettra d'appeler `TogglePlayPause()` de `UserInterface`, qui changera le statut "pause" du jeu, et qui changera le `timeScale` du jeu, afin de bloquer les différents calculs basés sur le temps, et donc mettre le jeu en pause.

Il ne reste plus qu'à renseigner le `GameObject` du `Panel` de pause dans l'éditeur Unity, et la base du jeu est terminée !

Complétion du jeu

Cette section du TD sera beaucoup moins guidée que les précédentes. Le but sera ici de vous suggérer des voies d'amélioration du jeu réalisé jusqu'à présent, et d'évaluer votre maîtrise de Unity.

Ecran titre, écran de fin et écran des scores

Pour cette partie, il vous faudra réaliser un écran titre permettant d'afficher le titre du jeu, puis de charger le jeu via le système de gestion de scène. Pour l'écran de fin du jeu, il faudra afficher un texte de Game Over ou de victoire. Enfin, faites en sorte de sauvegarder le score du joueur via le système de `PlayerPrefs` et créer un écran des meilleurs scores.

Vagues d'ennemis

Les ennemis ont pour l'instant un comportement très simple, le but de ce jalon sera d'améliorer cela. Faites en sorte à présent que les ennemis se déplacent en formation, et non plus individuellement. Sinus, cosinus, ligne, formation en V, libre à vous d'imaginer différents déplacements. Également, faites en sorte que les ennemis arrivent par vagues régulières (et vous pourrez, à l'occasion, ajouter un texte à l'écran informant le joueur de l'arrivée d'une vague).

Gestion de boss

Une fois que le joueur aura affronté un certain nombre d'ennemis, il sera temps pour lui de se mesurer à un ennemi plus musclé. Le boss sera un ennemi avec beaucoup de points de vie (mais pas trop non plus, soyez sympa avec ceux qui testeront votre jeu), et qui ne sortira pas de l'écran tant qu'il ne sera pas mort (pensez à implémenter des mouvements pour le boss, pour éviter que le combat ne soit trop simple). Vous pourrez aussi faire en sorte que l'ennemi possède deux type de collider, un type qui fera que si le joueur entre en contact avec, il perde des PVs, et un autre qui servira de cible à toucher avec les projectils pour baisser la barre de vie du boss (des points faibles sur le boss).

Bullet pattern

Pour améliorer les phases de boss, et pour développer l'un des éléments les plus emblématiques des shoot'em up, il faudra ici implémenter des motifs de projectiles, ou bullet pattern. Le principe est de faire en sorte que lorsqu'un ennemi tire, il lance plusieurs projectiles qui suivront un schéma spécifique (les mathématiques seront encore une fois votre allié) et qui suivront un système de vagues de patterns (exemple dans le jeu Touhou Project).

Gestion du son

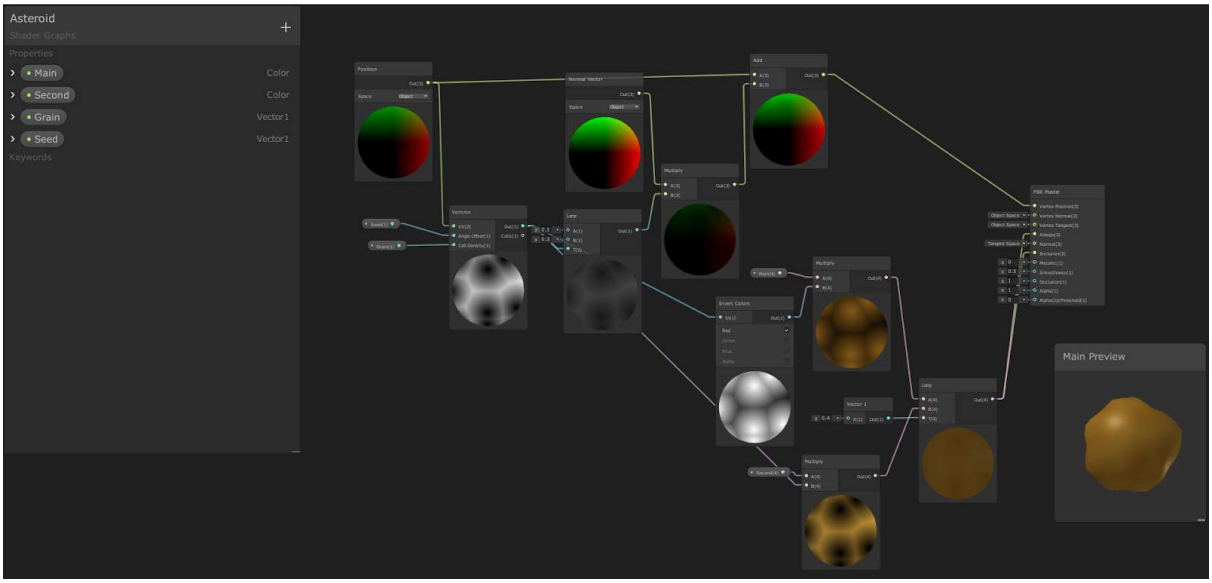
Afin d'apporter plus d'immersion dans le jeu, il est temps d'y ajouter du son : pour les projectiles, les collisions, la musique de fond, etc… Pour gérer le son dans Unity, deux composants sont nécessaires, un `AudioListener` (normalement déjà présent sur la caméra de la scène), et des `AudioSources`, qu'il vous faudra ajouter sur les objets qui selon vous doivent émettre un son. Si vous utilisez des sons que vous n'avez pas produits vous même, il vous faudra les lister, ainsi que leurs créateurs, dans un écran des crédits (par exemple accessible via l'écran titre du jeu). Egalement, pensez à utiliser des sons et des musiques libres de droit, ce qui pourra vous permettre de publier le jeu si vous le désirez, une fois terminé.

Capacités spéciales et armes

Le joueur ne possède pour l'instant qu'une petite arme simple pour affronter des vagues d'ennemis et des boss. Afin d'améliorer les conditions de jeu, faites en sorte que certains ennemis fassent apparaître (aléatoirement ou non), des bonus qui permettront au joueur de profiter de nouvelles capacités si il les récupère (cadence de tir plus élevées, projectiles plus puissants, plus gros, tirs en cônes, bouclier, etc…). Pensez à rajouter un indicateur visuel dans l'interface, pour indiquer au joueur les bonus actifs.

Champ d'astéroïdes *(optionnel)*

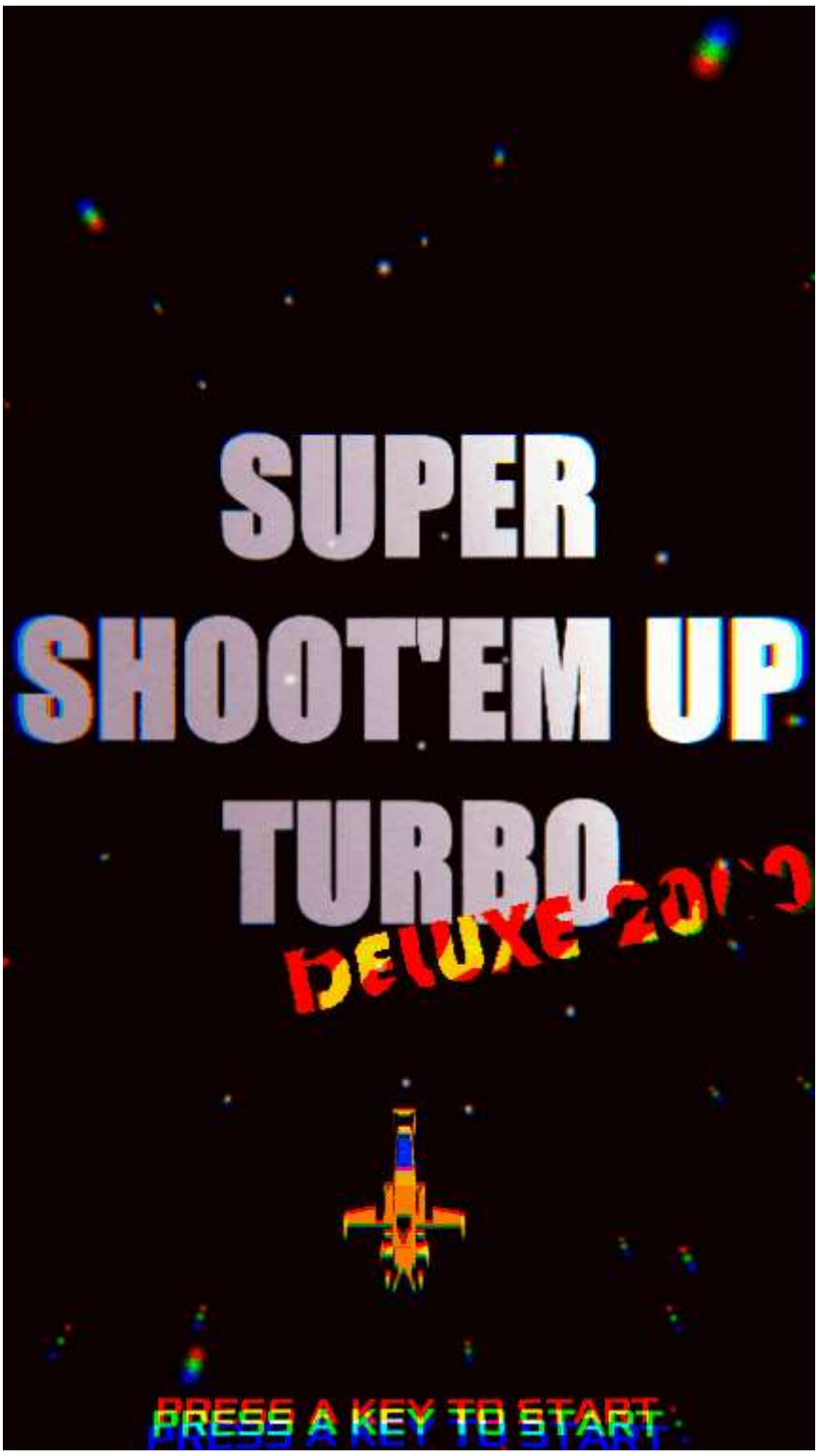
Afin de rendre le fond du jeu plus vivant, il serait intéressant de faire apparaître régulièrement des astéroïdes à l'écran. Il vous faudra dans un premier temps créer un modèle d'astéroïdes (une sphère fera l'affaire), sur laquelle vous appliquerez un shader qui lui donnera l'aspect d'un caillou de l'espace. En utilisant le `ShaderGraph`, vous pouvez très rapidement arriver à ce résultat (une capture de mon graph pour ce résultat):



Pour les instancier à l'écran, une simple classe basée sur celle gérant l'apparition des ennemis, fortement simplifiée, fera l'affaire (pensez à placer les astéroïdes dans un plan plus profond que celui du vaisseau et des ennemis pour éviter toutes obstructions visuelles).

Post processing *(optionnel)*

Pour polir le visuel de votre application, n'hésitez pas à jouer avec le post processing. On peut très rapidement et simplement obtenir des effets très stylisés. Le shoot'em up étant un très vieux style de jeu, il peut être de bon goût d'y ajouter un style visuel approprié.



Design

Enfin, pour le dernier coup de polish sur votre jeu : le design. Il est question dans ce jalon de faire en sorte de rendre votre jeu visuellement attractif : vous pourrez au choix dessiner et modéliser vous même les différents assets du jeu, ou bien utiliser les sources d'assets disponibles en ligne. Vous pouvez par exemple utiliser la fenêtre Asset Store de Unity, ou toutes autres sources, comme par exemple Kenney Assets. Si vous désirez créer vous même les assets, il existe de très nombreuses solutions, gratuites ou payantes : Blender, Krita, Aseprite, Asset Forge, Photoshop, etc… Encore une fois, si vous utilisez des assets qui ne sont pas les vôtres, pensez à citer les sources dans un écran des crédits.

Nettoyage du projet

Comme nous l'avions vu en cours, il est important d'avoir un projet bien rangé. A la fois dans ses dossiers et fichiers, et depuis l'inspecteur Unity et le code. Faites donc en sorte de bien ranger vos fichiers, commenter votre code, et d'utiliser les attributs de customisation des inspecteurs. Voici une liste de quelques attributs qui pourront vous être utiles :
`[Header]` pour créer un titre de section dans un inspecteur,
`[Tooltip]` pour créer un champ d'information au survol de la souris,
`[Range]` pour créer un slider permettant de saisir une valeur entre deux limites,
`[HideInInspector]` pour masquer un champ normalement visible,
`[ContextMenu]` pour créer une option au clic droit sur l'inspecteur du script.

Il existe beaucoup d'autres attributs dans Unity, vous permettant de développer des options plus détaillées si vous le désirez. Vous trouverez leur liste ici.

Enfin, vous pouvez également créer des inspecteurs personnalisés pour vos scripts, basé sur un fonctionnement similaire aux scripts éditeurs vu plus haut. Plus d'informations ici.

Aussi, pour faciliter le nettoyage de votre code, vous pouvez utiliser le plug-in CodeMaid, pour Visual Studio.

Rendu du projet

La date du rendu du projet est le 19 décembre 2021 à 23h59.
Pour pouvoir vous évaluer sur ce module, vous devrez publier votre jeu sur itch.io, et l'inscrire à la game jam (https://itch.io/jam/projet-shootemup-promo-2023), en précisant lors de l'inscription les membres de votre groupe et en indiquant le lien vers votre repo git.
Votre jeu devra pouvoir être builder pour les plateformes suivantes (au choix) :
PC, ou web player (web player étant le plus pratique des deux).

Ce projet servira à vous évaluer pour ce module. Vous pourrez le rendre en groupe de deux. Afin de faciliter votre travail d'équipe et de pouvoir efficacement nous transmettre le projet, merci de mettre en place un git pour gérer les sources de votre jeu (si vous n'avez pas l'habitude, il existe un très grand nombre de clients pour git, comme TortoiseGit, SourceTree, Fork Git, etc...). Pour l'utilisation de Git, afin d'éviter les fichiers Unity qui ne sont pas nécessaires lors du partage des sources, vous pourrez utiliser un fichier .gitignore comme celui-ci.
Votre code devra être commenté, et les fichiers devront être ordonnés proprement dans le projet Unity. Si vous utilisez des sources qui ne sont pas les vôtres, pensez à les créditer.

sources externes non créditées = points perdus sur le projet

Si vous avez des questions sur certains points du projet, n'hésitez pas à nous contacter par email ou par Discord :
Antoine Cherel : ac[at]imsimity.de / antoine_ch#5465
Kévin Gallien : gallien.kevin[at]outlook.com / Pikevchu#3797