

Objectif :

Le but de ce projet est de réaliser un jeu de tir 2D en suivant les spécifications fonctionnelles d'un game designer et des conseils de conception. Il permet de s'entraîner à concevoir une architecture propre d'un jeu et à l'appliquer dans Unity, notamment à travers son interface, des scripts C# et à l'aide de la documentation. Il illustre les concepts théoriques vus en cours sur les moteurs de jeu et le développement.

Pré-requis

Début du Cours JIN "[Intro. to Game Programming & Game Engines](#)"

Connaissances Unity de base (ex. [Introduction et TP ENSIIE](#), DJV, MOOC, projets personnels, stages...)

1. Organisation de la semaine

Mardi		Jeudi		Vendredi	
14h	9h	9h	14h	9h	14h
Cours I II Projet Part I		Cours III Projet Part I		Adrien Allard <i>Lead IA @ Amplitude</i> Conférence + Projet Part II	

2. Ressources

Logiciel : Unity 2021.3 LTS <http://unity3d.com/unity/>

Documentation et tutoriels :

<http://unity3d.com/support/documentation/> (en particulier Reference Manual, User Manual et Scripting Reference)

<http://unity3d.com/support/resources/>

3. Game Design

Maxence Voleau, Game Designer @ *Amplitude*, a rédigé une base de document régissant le fonctionnement du jeu. Essayer de faire au minimum les règles notées * pour vendredi 9h.

Avatar joueur

- Déplacement haut/bas/gauche/droite via flèches directionnelles et ZSQD (pour tout clavier)*
- Espace pour tirer*

Déplacement avancé :

- Déplacement à vitesse constante. Pas de ralentissement en cas de changement de direction. Le contrôle doit être fluide*
- Esquive d'une distance d (pixels) donnant invulnérabilité pendant x secondes si double tap dans une direction (deux entrées du même input en moins de y secondes)

Système de tir :

- Tabulation pour changer de type de tir parmi 3
 - continu et tout droit*
 - continu et dans les deux diagonales, à 45°
 - continu et en spirale
- Les projectiles ne touchent que les objets du camp opposé*
- Tir commence lorsque la touche est pressée, et se termine lorsque relâchée*

Énergie

- Chaque instant passé à tirer consomme de l'énergie*
 - Selon le type de tir : énergie et delta temps variable
 - L'énergie se recharge de x par seconde tant que le vaisseau ne tire pas*
 - Si énergie tombe à zéro, rechargement à 100% obligatoire et rechargement ralenti de 25%*
- Utiliser l'esquive consomme de l'énergie

Caméra

- Fixe*
- Avatar placé dans une bande représentant 10% de l'écran à gauche.

Structure du jeu et des niveaux

- 2 types de condition de victoire / défaite
 - Système de vie et vague fini à battre
 - Pas de vie juste gain de score en tuant et perte au hit + système de combo par tué sans hit
- Menu principal puis écran de sélection de niveau, un niveau est une série de vagues d'ennemis

Ennemis

- 2 types :
 - Avance tout droit et tir à intervalle régulier*
 - Avance en zigzag et tir à intervalle régulier
- Leur vitesse et l'intervalle de tir sont aléatoires entre deux bornes
- Chaque ennemi à peu de vie : besoin d'un hit et d'une explosion au minimum, au mieux un +x score à chaque mort

Collectibles

- Gain énergie courante / max selon le contexte
- Gain de vie / combo selon la condition de victoire
- Unlock de tir (3x ce collectible pour débloquent le prochain si contrainte de progression de l'avatar)
- Génération des collectibles aléatoires, contrôlés par évolution de la partie

4. Analyse et conception (cf. cours)

1. Identifier les éléments qui seront les plus importants dans votre jeu au regard du genre (rendu, commandes...)
2. Identifier les objets et composants nécessaires et leurs articulations (qui communique avec qui ?)
3. Proposer un diagramme de classes succinct (game objects, composants, quelques attributs et fonctions) pour représenter l'avatar du joueur et les ennemis (approche orientée composants)

NB : Prendre soin de factoriser et séparer les composants en les limitant à un ensemble de fonctionnalités cohérentes, notamment la gestion des commandes du joueur et leurs effets sur l'avatar.

5. Construction des éléments du projet

Concepts/Classes/méthodes utilisés : Editor, Assets, Project, Hierarchy, Inspector, GameObject, Component, Transform, Collider/Rigidbody, Prefab...

- Créer un nouveau projet (= dossier) (**avec réglages 2D**)
- Positionner les onglets à votre convenance ou utiliser un layout prédéfini (scrolling horizontal)
- Dans la fenêtre Projet, créer les dossiers "Prefabs", "Scenes", "Scripts", "Sounds" et "Textures".
- Enregistrer cette 1^{ère} scène vide (= .unity)
- Télécharger les ressources du projet sur la [page du cours](#) et copier toutes les images dans le dossier Textures ¹
- Insérer le sprite du Player
 - o Diminuer l'échelle (ex. 0.2 0.2 1)
 - o Ajouter un composant Box Collider 2D et modifier sa taille en 10 10
 - o Ajouter un Rigid Body 2D
 - o Créer un prefab
- Processus similaire pour un Enemy

6. Déplacements du Player et des Enemies

+ Input, Project Settings, Script, MonoBehaviour, Start, Update, Transform, Time, Debug, Random, Vector2, GetComponent, Destroy, héritage...

Principe :

- la caméra est fixe
 - l'avatar du joueur est fixe lorsqu'il n'y a pas de commande et se déplace lors des commandes (dans une zone rectangulaire)
 - les ennemis se déplacent
 - pas de décor (sauf background de la caméra)
1. Créer le composant abstrait BaseAvatar. Ajouter une property éditable float MaxSpeed (voir [Annexe](#)). Créer les composants fils PlayerAvatar et EnemyAvatar.
 2. Implémenter le composant Engines qui gère le déplacement de l'avatar :
 - propriétés Vector2 Speed et Position (= celle du transform)
 - calcul à chaque frame de la nouvelle position en fonction de la MaxSpeed de l'avatar (et du Frame Delta)
 3. Implémenter le composant InputController qui contrôle la Speed des Engines (qu'il faut donc récupérer...) à l'aide des Input Axis
 4. Compléter les prefab Player et Enemy et tester. Paramétrer MaxSpeed et l'Input Manager de Unity pour améliorer la réactivité des commandes.
 5. Créer le composant AIEnemyBasicEngine² pour contrôler le déplacement horizontal simple des ennemis via leur Engine
 6. Ajouter dans un des composants la destruction de l'ennemi lorsqu'il sort de la zone visible de la caméra (faire test très simple)

¹ Tutoriel et ressources originales par <http://pixelnest.io/> vous pouvez bien sûr en prendre d'autres

² On pourra par la suite créer une hiérarchie de différents AIEnemyEngines pour varier leurs déplacements

7. Tirs du Player et des Enemies

+ instantiate, foreach, OnTriggerer, layers...

Principe : Player/Enemy contient un (ou plusieurs) BulletGun qui possède des caractéristiques de tir et peut tirer (c'est-à-dire instancier des Bullets) sous certaines conditions.

1. Créer les prefabs PlayerBullet et EnemyBullet à partir des sprites correspondants et d'un collider de type trigger
2. Implémenter le tir du Player :
 - créer le composant abstrait Bullet
 - propriétés : float Damage, Vector2 Speed, Vector2 Position
 - fonction d'initialisation des propriétés virtual Init(...)
 - fonction de mise à jour de la position virtual UpdatePosition()
 - gestion de la destruction quand invisible
 - créer le composant concret SimpleBullet pour des PlayerBullets horizontales et tester
 - créer le composant BulletGun du Player :
 - propriétés : float Damage, Vector2 Speed, float Cooldown
 - Fire() instancie une bullet si c'est possible
 - compléter InputController
3. Implémenter la gestion des dégâts :
 - compléter BaseAvatar : Health, MaxHealth, TakeDamage(float), Die()...
 - compléter Bullet : OnTriggererEnter2D(...)
4. Ajouter/modifier les composants nécessaires pour gérer les tirs ennemis :
 - AIBasicBulletGun : tir automatique avec cooldown
 - enum BulletType (Player ou EnemyBullet) à prendre en compte pour les collisions
5. Ajouter un layer pour les Bullets et paramétrer la matrice de collision pour limiter les calculs (Project Settings)

8. GameManager

+ singleton, InvokeRepeating, Random...

Principe : Le GameManager et la caméra seront les seuls objets présents dans la scène au démarrage de l'application. Le GameManager est responsable de la création des autres objets.

1. Créer l'objet vide [GameManager], implémenter son composant GameManager et tester :
 - pattern Singleton
 - instantiation d'un Player
 - instantiation répétée d'un Enemy à une hauteur aléatoire³

9. Build

Build settings

- Générer une version WebGL du projet et tester
- Générer une version du projet pour votre OS et tester

³ Vous ferez une gestion des objets plus aboutie en dernière séance

10. UI et Scene Manager

Canvas, UnityEngine.SceneManagement, SceneManager.LoadScene(...)

1. Créer une nouvelle scène Menu qui sera le point d'entrée du jeu
2. Insérer un canvas, un texte et 2 boutons Start et Quit
3. Créer un script MenuManager avec 2 fonctions associées aux boutons (start lance la scène principale faite précédemment)
4. Mettre à jour un des builds précédents

11. Energie

Implémenter la gestion de l'énergie dans les composants précédents les plus appropriés:

1. Caractéristiques, vérification et baisse à chaque tir
2. Ajout de la régénération hors tirs
3. Ajout de l'efficacité de régénération

Pour visualiser simplement la vie et l'énergie du Player, nous allons utiliser les sliders de l'UI Unity.

4. Insérer un canvas nommé UIManager
5. Insérer et paramétrer 2 sliders HealthBar et EnergyBar
6. Créer le script UIManager qui met à jour les sliders selon les caractéristiques du Player (mécanisme de "polling")

12. Evénements

Mettre en place l'envoi et la réception d'un événement OnDeathEvent qui signale la mort du player ou des ennemis à l'UI et au GameManager (à vous d'imaginer le contenu des handlers)

Ajouter d'autres événements de votre choix.

Annexe : Indications de programmation Unity/C#

Pour ajouter des attributs à nos scripts nous avons choisi la méthode suivante :

- déclaration d'un champ private
- ajout du tag [SerializeField] si l'on souhaite pouvoir éditer ce champ dans l'inspecteur
- définition d'une *propriété* C# associée (qui, elle, ne peut pas être accessible dans l'inspecteur). Elle est public, possède des méthodes d'accès équivalentes aux getter et setter habituels mais s'utilise ensuite comme une variable classique (=).

Exemple :

```
[SerializeField]
private float maximumHealthPoint;

public float MaximumHealthPoint {
    get {
        return this.maximumHealthPoint;
    }
    private set {
        this.maximumHealthPoint = value;
    }
}
...
MaximumHealthPoint = 50f; //appel implicite au set
float m = MaximumHealthPoint; //appel implicite au get
```

Les *propriétés* peuvent bien sûr être utilisées sans lien avec un champ, et les get/set peuvent être personnalisés.

Héritage d'une classe abstraite en C# :

```
public abstract class A : MonoBehaviour {
    protected virtual void Func() {
        ...
    }
    ...
}

public class B : A {
    ...
    protected override void Func () {
        base.Func ();
    }
}
```

Annexe : Démarrage rapide avec Git

1. S'inscrire sur une plateforme d'hébergement git
 - a. Github, gitlab, bitbucket, forge ensiie ou tsp...
 - b. Compléter la procédure nécessaire aux connexions sécurisées ssh
2. Installer un client graphique git (et le shell git s'il n'est pas inclus)
 - a. Github desktop, sourcetree...
3. Créer le projet unity
4. Initialiser le dépôt git dans le dossier du projet via le client (« create »)
5. Paramétrer le .gitignore (ex. fourni dans les ressources du projet)
6. Commit
7. Paramétrer le repository distant
8. Push