

# Gladiator

---

3C	Gold
Lundi 5 Novembre – 23h59	Mercredi 14 Novembre - 23h59

**Nombre d'étudiants par groupe : 2**

**GIT:**

git clone sh://[git@git.isartintra.com:2424/2018/MTL\\_GP2\\_2021\\_Unreal/{nom\\_du\\_groupe}.git](https://git@git.isartintra.com:2424/2018/MTL_GP2_2021_Unreal/{nom_du_groupe}.git)

**Rendu :**

- Il est attendu un binaire pour le push final.
- Dans un dossier différent de l'exercice Platformer.
- Ne pas versionner les dossiers suivants :
  - Binaries
  - Build
  - Saved
  - Intermediate
- Un bref document technique, expliquant de manière concise :
  - l'architecture générale du jeu
  - les features implémentées et la répartition des tâches
  - les limitations et bugs connus
  - ce qu'il faut savoir pour une meilleure expérience de jeu

## But

---

Le but du projet consiste à créer un petit jeu afin de synthétiser les connaissances acquises sur Unreal Engine.

# Sujet

---

## Sommaire :

Le but est de créer un jeu de combat de mêlée simple en 3D, dans lequel le joueur contrôle un nain gladiateur à la troisième personne et affronte 5 autres gladiateurs dans une arène.

La version final devra contenir cette **liste de features obligatoires**:

### Caméra :

- Le joueur contrôle une third-person camera avec la souris.
- La caméra ne doit pas rentrer dans le décors.
- La caméra ne doit effectuer aucun test de collision avec des ennemis (cela aurait pour effet de la rapprocher de très près du joueur lorsqu'un ennemi attaque, ce qui n'est pas pratique du point de vue du jeu).

### Player :

- le joueur peut déplacer son personnage dans l'environnement en utilisant W, A, S, D (qui déplace le personnage du joueur en avant, à gauche, en arrière, à droite dans la référence de la caméra).
- Le personnage du joueur doit toujours s'orienter dans la direction dans laquelle il se déplace, il ne doit jamais strafe, ni reculer.
- Le joueur commence avec 5 points de la vie (cette valeur devrait être facilement ajustable dans l'éditeur).

### Player attaque :

- Lorsque le joueur appuie sur le bouton gauche de la souris, son personnage attaque et l'animation d'attaque est jouée.
- Pendant l'attaque, le joueur ne peut pas se déplacer.
- L'attaque doit vérifier s'il y a collision pour voir si un ennemi est touché (ce test doit être effectué au bon moment de l'animation de l'attaque pour que le joueur se sente bien dans le jeu).

- Si l'ennemi est touché, un effet visuel doit apparaître pour indiquer clairement au joueur que son attaque a bien touchée. (Cela peut être une particule ou une couleur de matériau temporaire qui vire au rouge, cela n'a aucune importance tant que le retour est évident pour le joueur).
- Dans ce cas, l'ennemi touché perd un point de vie.

#### **Ennemis :**

- Quand le jeu commence, il y a 5 ennemis dans l'arène.
- Ils ont tous 3 points de vie.

#### **IA - Position des Ennemis:**

- En tant que groupe, ils ne restent pas trop loin du joueur, mais pas trop près pour des raisons évidentes de sécurité (pour éviter d'être facilement attaqué par le joueur).
- Chaque ennemi doit avoir une vue non obstruée du joueur, c'est-à-dire que pour chaque ennemi, la ligne de mire entre cet ennemi et le joueur ne doit pas se croiser avec d'autres ennemis.
- Sauf pendant l'attaque, l'ennemi doit toujours faire face au joueur, à tout moment. Par conséquent, les ennemis sont autorisés à se dégager et à reculer.

#### **IA - Ennemis attaque:**

- Toutes les secondes (cette période devrait être une valeur ajustable dans l'éditeur), un ennemi est choisi pour attaquer le joueur.
- L'algorithme qui choisit l'ennemi pour attaquer appartient aux développeurs, dans la mesure où il rend le jeu aussi amusant que possible (choisissez un ennemi aléatoire, choisissez l'ennemi le plus proche, celui situé à l'arrière du joueur, le dernier attaquant récent, une combinaison des règles précédentes...)
- L'attaquant choisi doit se déplacer vers le joueur, suffisamment près pour que l'attaque puisse l'atteindre. Il devrait alors déclencher l'attaque.
- Comme le joueur, l'ennemi ne peut pas se déplacer lorsqu'il attaque et une vérification de la collision doit être effectuée au bon moment pour détecter si le joueur est touché.
- Si le joueur est touché, il devra perdre un point de vie et un FX devra être joué afin de lui donner un retour évident qu'il a été attaqué.
- Une fois l'attaque terminée, l'ennemi doit revenir à sa position par défaut (pas trop près / pas trop loin du joueur).

#### **Mort :**

- Si un personnage (joueur ou ennemi) perd toute sa vie, l'animation de mort est jouée.
- Une fois l'animation terminée, le personnage reste sur la dernière image (couché au sol), les collisions et la logique sont désactivées (le personnage devient simplement un élément graphique d'arrière-plan).

### Game state :

- Lorsque tous les ennemis sont vaincus et que le joueur est toujours en vie, le jeu doit informer le joueur qu'il a gagné (affiche un texte «You Win», par exemple), et le jeu devrait être réinitialisé (joueur et ennemis respawn).
- Lorsque le joueur perd toute sa vie, le jeu doit l'informer qu'il a perdu et le jeu doit être réinitialisé.

### HUD :

- Affichez, à votre guise, la santé du joueur (texte, jauge...)
- La santé des ennemis doit être affichée sur eux, dans un espace 3D, et toujours visible par la caméra du joueur.
- La mise à jour du HUD doit être effectuée à l'aide de delegates et d'événements.

### Cycle jour-nuit :

- Le jeu doit passer du jour au soir et du jour au soir automatiquement (**peut être implémenté en Blueprint**).

### Le son (toutes les fonctionnalités de sons peuvent être développées dans blueprint, à vous de choisir)

- Un son aléatoire doit être joué lorsque le joueur ou l'ennemi attaque (son de voix).
- Lorsque vous touchez le sol pendant une attaque, vous devez jouer un son de métal.
- Quand un personnage marche, un son différent doit être joué en fonction de la surface sur laquelle il marche.
- Il devra y avoir une musique ambiante qui change en fonction de l'heure de la journée.
- Il devrait y avoir un jingle lorsque le jeu se termine (gagner ou perdre).

## Bonus

---

**Si les fonctionnalités principales sont correctement implémentées, vous pouvez ajouter certaines des fonctionnalités supplémentaires suivantes, dans n'importe quel ordre.**

### Pluie:

- De temps en temps, de la pluie peut tomber sur l'arène. Vous pouvez implémenter cela dans blueprint. Elle peut être aussi laide que possible, le seul but de cette fonctionnalité est de permettre l'intégration des sons de pluie. Vous pouvez également utiliser un son de pas de pied différent quand il pleut.

### **Drop d'Équipement :**

- Les personnages peuvent laisser tomber leur bouclier ou leur marteau au hasard lorsqu'ils reçoivent un coup.
- L'équipement largué s'envolerait à quelques mètres de la position du personnage (il doit être physiquement transformé en conséquence).
- Sans son bouclier, un personnage subit deux fois plus de dégâts.
- Sans son marteau, un personnage ne peut pas attaquer.
- Une fois qu'il aura perdu son bouclier ou son marteau, le joueur devra s'approcher de lui et appuyer sur le bouton d'attaque pour récupérer l'équipement perdu.
- Quand un IA perd son bouclier ou son marteau, il doit passer à l'équipement perdu et le récupérer une fois qu'il en est proche. Dans cette situation, l'intelligence artificielle ne fera pas face au joueur mais à l'équipement vers lequel il se dirige.

### **Effet de Caméra :**

- Une animation de caméra réalisée en **Matinee** peut être lue lorsque le joueur commence le jeu (introduction de quelques secondes).
- De la même manière, une simple animation de caméra peut être lue lorsque le joueur meurt.

### **Visual FX :**

- Spawn particules de poussière pour chaque étape de pied.
- Spawn particules de sang quand il y a un coup.

### **Lock :**

- Le joueur devrait pouvoir verrouiller un ennemi en utilisant le bouton droit de la souris.
- Lorsque le joueur appuie sur le bouton droit de la souris, il appartient au développeur de choisir quel ennemi doit être verrouillé (le plus proche, celui qui fait le plus face au joueur...)
- En utilisant la molette de la souris, le joueur peut alterner entre les ennemis et changer sa cible actuelle
- Lorsque le joueur verrouille une cible, il peut cliquer à nouveau sur le bouton droit de la souris pour quitter le mode de verrouillage.
- Verrouiller un ennemi signifie que le personnage du joueur fait toujours face à l'ennemi et qu'il se déplace autour de l'ennemi en cercle (comme dans les jeux 3D Zelda ou Dark Souls).

# Contraintes

---

**Utilisé les BluePrint intelligemment , pas de code “code spaghetti” (Utilise-les pour de la logique simple, des valeurs par défauts, l'animation et les VFX). Toute logique avancée et toute tâche comportementale complexe doivent être écrites en C ++.**

**Conseils et contraintes supplémentaires:**

**Les personnages** doivent porter un bouclier et un marteau par défaut.

**Les contrôleurs IA** doivent utiliser un Behavior Tree.

**Un directeur d'IA** n'est pas nécessaire, mais vivement conseillé. Il s'agit d'un seul acteur capable de calculer les positions idéales de tous les ennemis et de décider quelle IA devrait attaquer.

Les variables suivantes doivent être tweakables:

- Points de santé du joueur et des ennemis
- Période entre deux attaques par IA

# Assets

---

Pour commencer le projet avec les assets graphiques :

- Créer un nouveau projet vide Unreal appelé **GladiatorGame**.
- Copier le dossier **GladiatorAssets** situé dans votre partage vers un répertoire sur votre disque.
- Ouvrez votre copie de **GladiatorAssets** (c'est un projet Unreal)
- Sélectionnez tous les dossiers et migrez les vers votre projet **GladiatorGame** (indiquez le chemin du dossier Contents de ce projet). Si l'option n'est pas disponible, migrer les dossiers un par un.

De cette façon, lorsque de nouveaux assets seront ajoutés plus tard dans **GladiatorAssets**, vous pourrez les importer sans avoir besoin de recréer un nouveau projet

#### Arène :

- L'arène se situe dans le répertoire Game/Arena du projet **GladiatorAssets**
- Migrez le fichier **DesertArenaBlueprint** vers votre projet (toutes les dépendances seront ainsi importées)
- Ensuite vous pouvez insérer un **DesertArenaBlueprint** dans votre map

Le **DesertArenaBlueprint** contient déjà les éléments suivants :

- Les meshes graphiques (l'arène, le sol, le plan d'eau, le ciel)
- Les collisions (sol et murs invisibles)
- Les Spawnpoints pour les IA (5 targets points)
- Le Player Start actor
- Des lumières afin que tout soit visible à l'écran

En revanche il ne contient pas :

- Un navmesh bounds volume
- Pas de matériaux physiques

Par conséquent, pour créer un niveau jouable from scratch, il suffit de :

- Créer un empty level sous Unreal (niveau noir sans lumière ni autres actors)
- Ajouter un **DesertArenaBlueprint** dans le niveau
- Ajouter un Navmesh bounds et vérifier que le navmesh est valide
- Ajouter votre AIDirectors (ou vos IA) et vos SpawnPoints
- Changer le gamemode par le votre pour y spawner votre joueur

Vous devriez ainsi avoir un niveau entièrement jouable. Vous pouvez bien sûr ajouter le **DesertArenaBlueprint** à votre niveau déjà existant.

#### La soutenance évaluera :

- Si les features obligatoires sont présentes.
- Sur la documentation rendue.
- Si le code est propre, bien architecturé et aussi performant que possible.

## Liens utiles

---

<https://docs.unrealengine.com>