Gladiator

Procédure du rendu

Start	Fin
Lundi 24/10/2021 – 13h	Lundi 21/11/2021 –19h

Nombre d'étudiants par groupe : 2
Rendu :

- Sources : tag GOLD sur git (sur la branche master)
- Ne pas versionner les dossiers suivants :
  - Binaries
  - Build
  - Saved
  - Intermediate
- Exe sur l'exercice, NOM_NOM.exe

=> -4 points si ce n'est pas respecté

Pénalités de retard:

- 5 minutes de retard = -1 points
- 15 minutes de retard = -2 points
- 30 minutes de retard = -4 points
- 1 heure de retard = -10 points
- Rendu le 24 = 0

Objectifs

Création d’un beat’em all, servant à mettre en pratique l’ensemble des compétences apprises sur le moteur.
Compte pour **40%** de la note du module Unreal.

Compétences

Le projet sera évalué sur les compétences suivantes:

Hard Skills:

- ➢ Comprendre la logique d'exécution du moteur
- ➢ Comprendre la représentation des données dans le moteur
- ➢ Maîtriser l'environnement du moteur

Features

But du jeu :
Le joueur devra combattre des ennemis à l'intérieur d’une arène et en ressortir vainqueur.

Caméra :

- Le joueur contrôle une third-person camera avec la souris.
- La caméra ne doit pas entrer dans le décor.
- 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 Z, S,Q, 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.

Player Défense :

- Lorsque le joueur appuie sur le bouton droit de la souris, son personnage défend et lève le bouclier tant que le bouton n'est pas relâché.
- En défense le personnage bloque toutes attaques frontales mais ne peut plus bouger, ni s'orienter.

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 mise à jour du HUD doit être effectuée à l'aide de delegates et d'events.

Bonus

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

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.

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 clic molette 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).
- HUD affiche la vie de l'ennemi ciblé.

Contraintes

Utiliser les BluePrint intelligemment , pas de “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 **DesertArenaBluerint** contient déjà les éléments suivants :
- Les meshs graphiques (l'arène, le sol, le plan d'eau, le ciel)
- Les collisions (sol et murs invisibles)
- 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.

Vous pouvez importez vos propres assets du moment que l'on y retrouve l'arène fermée et des personnages animés/utilisables

Liens utiles

http://romeroblueprints.blogspot.fr/

https://docs.unrealengine.com/en-US/Gameplay/Framework/index.html

Tips

- Appliquez vous sur le côté architecture et IA .
- N'oubliez pas de jouer à votre projet pour vous assurer de sa fluidité