RAPPORT DU PROJET 3D/MACHINE LEARNING/ REALITE VIRTUELLE : PARAPENTE AU DESSUS DE L'EVEREST

Valentin Marchand
61392@etu.he2b.be

TABLE DES MATIERES

PITCH

« Le plus grand, majestueux et imposant sommet du monde est en danger. Chaque année environ 1000 personnes tentent l'ascension. Ce qui devait être un exploit surhumain mêlant dangerosité et communion avec la nature, s'est transformé en activité touristique pour riche en recherche de sensation. C'est pour cela que vous êtes débauché ! Survolez le mont Everest avec votre parapente équipé d'un canon à boule de neige, pour empêcher cette frénésie des hommes en costards ! A vous de jouer !»

RÉSUMÉ

Dans cette unité d'enseignement dont le but est le développement de jeux vidéo nous avons réalisé un projet mêlant réalité virtuelle, environnement 3D et machine learning. L'objectif était donc de faire une simulation de parapente au-dessus du mont Everest et de le rendre le plus immersif possible grâce à la 3D ainsi que la réalité virtuelle. De plus le jeu intègre de petits personnages non-joueur animés par des algorithmes d'intelligence artificielle qui doivent gravir le sommet sans se faire avoir par le joueur qui pourra leur tirer dessus.

1. INTRODUCTION

Ce projet s'articule autour de la réalité virtuelle, la 3D et l'intelligence artificielle. Ces 3 principes ont pour but commun de proposer une expérience immersive où le joueur incarne les yeux d'un pilote de parapente, grâce à la VR, dans un environnement 3D autour de l'Everest et luttant contre des personnages non joueurs aux aspects humains aidés par l'intelligence artificielle.

1.1. Introduction à la réalité virtuelle du projet

Dans ce projet la réalité virtuelle nous permettra de simuler la présence physique de l'utilisateur dans un parapente évoluant dans un environnement complètement artificiel. Elle n'est pas à confondre avec la réalité augmentée qui ajoute des éléments virtuels dans un environnement physique et la réalité mixte, qui comme son nom l'indique, permet un mélange des deux comme visible en figure 1.

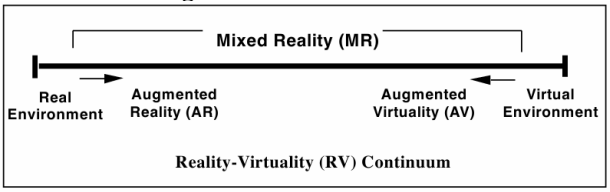

Figure 1. Schéma de la réalité mixte

Pour une expérience sensorielle accrue, nous utiliserons donc 4 éléments importants : un casque HTC VIVE (visible en figure 3), une structure pour simuler la sensation d'être assis dans un parapente (visible en figure 4), de ventilateurs et d'un ensemble Arduino + joystick pour l'élongation des commandes du parapente.

1.2. Introduction à la partie 3D / Moteur de jeu

Pour donner une expérience réaliste nous avons utilisé le moteur de jeu Unity. Il nous permettra de mettre en place un environnement 3D avec un paysage autour de l'Everest ainsi que de synchroniser les éléments utilisés pour la réalité virtuelle en partie 1.1 avec l'environnement et les PNJ grâce au langage C#. Ainsi, nous pourrons intégrer des interactions dynamiques, comme le comportement des PNJ et les actions du joueur, tout en garantissant une immersion totale grâce à l'optimisation du rendu 3D et des contrôles en VR.

1.3. Introduction à la partie Machine Learning

Les personnages non joueurs qui gravissent l'Everest seront représentés par le modèle 3D, que nous nommerons ici Dupont-de-Richeville (DdR). Le joueur devra faire face donc à une armée de DdR.

Figure 2. Modèle 3D de Dupont-de-Richeville

Les DdR's (PNJ) auront pour objectif d'atteindre le sommet de l'Everest sans se faire « éliminer » par le joueur dans son parapente.

Pour améliorer le réalisme et l'autonomie de ces personnages nous avons utilisé trois techniques d'intelligence artificielle orientée PNJ : Final State Machine, Système multi-agent et NavMesh.

- FSM : Méthode pour modéliser les comportements des PNJ avec l'utilisation d'états (courir, fuir etc…)
- NavMesh : Grille de navigation basée sur des maillages pour permettre aux PNJ de se déplacer de manière autonome. Elle utilise des algorithmes tels que A* pour le « pathfinding », un système de triangulation, flood fill (recherche en largeur avec Bred first search) et algorithmes de simplification.
- SMA (Système Multi Agent) : Ensemble d'agents autonomes qui interagissent dans un environnement commun.

1.4. Matériels et environnement de développement

Pour ce projet nous avons besoin :

- D'un HTC Vive pour la réalité virtuelle (figure 3).
- D'une structure pour simuler la sensation d'assise du joueur dans un parapente (figure 4)
- D'un Arduino pour récupérer les valeurs des joysticks (figure 5).
- De 2 joysticks pour récupérer l'élongation sur l'axe z (verticale) des commandes.
- De 2 ventilateurs commandables via DMX
- Du contrôleur USB DMXIS par ENTTEC (figure 6).
- Deux bases pour l'utilisation de l'HTC Vive

Figure 3 - Casque HTC VIVE

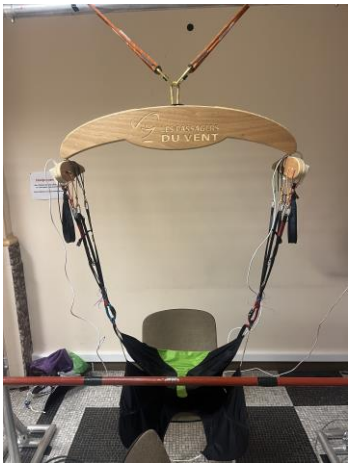

Figure 4. Structure de simulation de vol

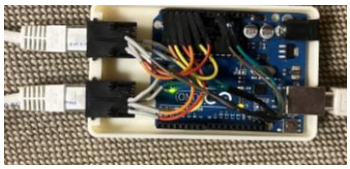

Figure 5. Arduino utilisé

Figure 6. Interface USB de contrôle DMX par ENTTEC

L'environnement de développement choisi est, comme précisé en partie 1.2, Unity choisi pour sa facilité de développement avec Visual Studio 2022 et le langage C#. C'est également un environnement gratuit qui possède également de nombreux avantages, assets pour la VR et 3D.

2. JEU ET MISE EN PLACE

2.1. Environnement 3D

L'objectif de cette partie est de mettre en place un environnement 3D captivant et proche de la réalité.

NB : le développement de l'animation 3D des personnages non-joueurs sera détaillée en partie 2.3.1.

2.1.1. Mont Everest et heightmap

Dans une optique de réalis le mont Everest le plus fidèle, nous avons choisi l'utilisation de heightmap. C'est une représentation bidimensionnelle pour stocker des informations sur la hauteur (altitude) d'une surface. Elle permet une simulation réaliste des surfaces comme l'Everest. Pour notre cas, la carte des hauteurs choisie est présente en figure 7.

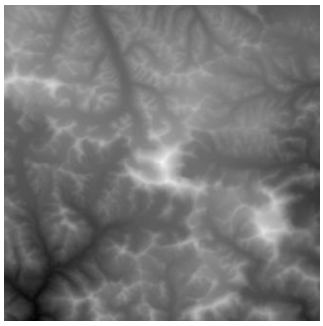

Figure 7. Heightmap du mont Everest

C'est une image en niveau de gris avec 0 qui représente les points les plus bas et 255 les plus hauts. Nous voyons en figure X que le pic de l'Everest (au centre) est blanc donc que nous pouvons aisément déterminer les points les plus hauts de la map.

Ensuite, la position (x, y) correspond à un emplacement sur le plan 2D, et la valeur associée (grayscale ou numérique) correspond à la hauteur en 3D.

A partir de cette map et des fonctionnalités, telles que TerrainTools 5.0.1 (lien disponible dans les références) de Unity, pour la génération de terrain, nous pouvons donner vie à un modèle 3D fiable du mont Everest comme visible en figure 8.

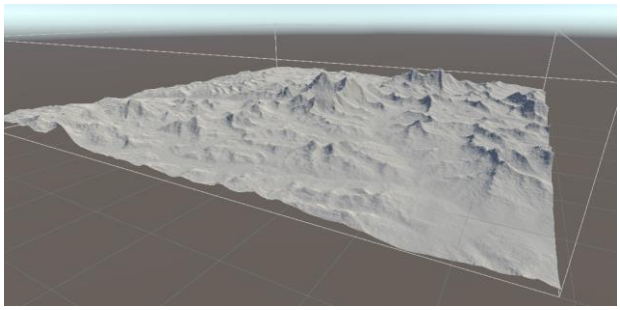

Figure 8 . Modèle 3D du mont Everest à partir de la heightmap

La création d'un terrain est importante notamment car elle permet de créer une surface navigable pour les PNJ dans le cadre de l'utilisation d'agent NavMesh mais aussi de pouvoir utiliser des outils pour le sculpter, d'avoir des performances optimisées etc…

2.1.2. Horizon et skybox

Pour optimiser la partie réalité virtuelle avec le HTC nous avons dû mettre en place un terrain d'une longueur, largeur et hauteur maximale de 100 000m x 100 000m x 10 000m. Ce qui complique donc l'ajout de terrain encore plus vaste (pour l'aperçu du joueur par exemple un effet de montagne lointaine) et donc un effet d'horizon difficile à mettre en place.

Pour parer à ce problème nous avons utilisé une skybox pour placer l'utilisateur dans les nuages (ce qui est logique en haute altitude) et d'autres terrains pour entourer le terrain principal en figure 8.
Une skybox est un environnement 3D simulé qui entoure la scène. Elle est à une distance infinie et bouge seulement avec la caméra. Ce qui nous permet d'avoir notre effet d'horizon lointain. Dans notre cas, nous utilisons une skybox à 6 faces (Cube Map) représentant les 6 faces d'un cube, donc si on regarde à gauche, à droite ou bien en bas avec le casque nous voyons un ciel nuagé.
Les résultats sont visibles en figure 9 (les liens des packages utilisés sont disponibles dans les références de l'article)

Figure 9 . Aperçu de la map avec le mont Everest, les montagnes qui l'entourent et la skybox

2.2. Intégration de la réalité virtuelle

2.2.1. Configuration de la VR avec OpenVR

La réalité virtuelle de ce projet s'articule autour de trois notions : le fonctionnement de l'HTC Vive, l'utilisation de steamVR et la bibliothèque OpenVR.

Le HTC Vive est un casque offrant une immersion dite stéréoscopique. Ce qui veut qu'il renvoie des images composées de deux vues (gauche et droite). L'effet permet de faire ressentir un effet de relief, accentué grâce à l'utilisation de la parallaxe. Il est également équipé de capteurs de mouvement (accéléromètres, gyroscopes, et capteurs de position). Cependant un aspect très important est l'utilisation de bases. Pour simplifier, les bases utilisent des faisceaux infrarouges pour suivre notre casque dans un espace 3D, appelés Lighthouse et reposent sur un système de triangulation des données des capteurs et récepteurs. Les stations sont donc placées dans des coins opposés pour constituer une zone de jeu, comme en figure 10.

Figure 10. Schéma d'illustration de la création d'une zone de jeu

Nous avons donc créé et configuré notre zone de jeu autour de la structure de parapente visible en figure 4.

Nous avons utilisé ensuite SteamVR qui est la couche logicielle, développée par Valve, qui permet de lier l'utilisation du casque avec le système d'exploitation. Ils exposent des API permettant de gérer les différents capteurs du casque VR.

Pour finir, OpenVR est un kit de développement, créé par Valve, qui fournit un accès à SteamVR. C'est une sorte de facilitateur (couche d'abstraction) qui nous permettra d'utiliser notre HTC sans se soucier des détails matériels.

Voici comment cela s'organise (figure 11).

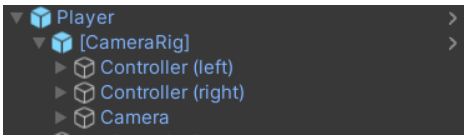

Figure 11. OpenVR dans le Game Object Player, pour l'utilisation de SteamVR

Nous glissons l'objet *[CameraRig]* dans le Game Object Player. Ce rig n'est autre que le système de suivi du joueur dans l'environnement virtuel. Il nous permettra de gérer la caméra, les contrôleurs (non utilisés dans ce projet) et le système de suivi des mouvements.
La caméra est intégrée et représente la vue du joueur. La position et l'orientation de la caméra sont mises à jour en temps réel en fonction des données du casque.

C'est à l'endroit où le rig est placé que la zone de jeu, formée précédemment grâce aux stations de base, est dessinée dans Unity (zone blanche en figure 12) et que les « yeux » du joueur (représenté par la caméra) seront, comme en figure 12.

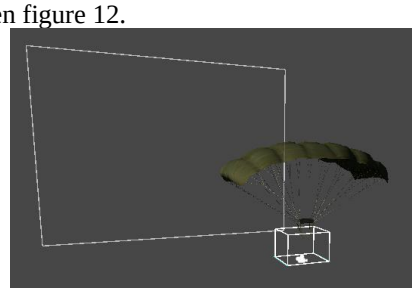

Figure 12. Zone de jeu et caméra du Player dans Unity

Le script SteamVR Play Area attaché au Caméra Rig définit et affiche la zone physique dans laquelle l'utilisateur peut se déplacer. Le script SteamVR Camera Helper facilite le suivi de la caméra du joueur, ajuste la position et l'orientation en fonction du casque.

2.2.2. Récupération de l'élongation des commandes du parapente sur l'Arduino

Pour les commandes des parapentes nous avons conçu un script C# (serialRead.cs) qui lit les données provenant du port série auquel est relié l'Arduino et qui effectue un traitement spécifique sur celle-ci.

Ce code contient 3 variables :

- _serial : Variable utilisée pour créer un objet SerialPort qui permet la communication via port série.
- z et z2 : Variables de type float pour récupérer l'élongation des commandes dans [0,1]

Nous avons la fonction ConnectToPort qui permet d'effectuer une connexion au port défini en figure 13:

```
public void ConnectToPort()
{
    // Port à lire
    string port = "COM3";
    try
    {
        // Configuration du port
        _serial = new SerialPort(port, 230400)
        {
            Encoding = System.Text.Encoding.UTF8,
            DtrEnable = true,
            ReadTimeout = 2000
        };

        // Ouverture du port
        _serial.Open();
        Debug.Log($"Connected to {port}");
    }
    // Exception
    catch (Exception e)
    {
        Debug.Log($"Not Connected to {e.Message}");
    }
}
```

Figure 13. Connexion au port Série

Si le port série est ouvert, alors dans la fonction update nous lisons les données et nous les traitons grâce au code en figure 14 (la fonction ExtractValue est disponible en Annexe) :

```
// Lire les données de l'Arduino
string data = _serial.ReadLine();

// Ségmenter les data avec '/'
string[] segments = data.Split('/');

foreach  (string segment in segments)
{
    // Si la valeur correspond à z
    if (segment.StartsWith("z") && segment[1] != '2')
    {
        // Extraire la valeur utile
        z = ExtractValue(segment,false);
    }
    // Si la valeur correspond à z2
    else if(segment.StartsWith("z2"))
    {
        // Extraire la valeur utile
        z2 = ExtractValue(segment, true);
    }
}
```

Figure 14. Récupération et traitement des données

Pour finir ce code, nous avons une fonction qui ferme le port série quand le joueur quitte l'application.

2.2.3. Commandes du parapente

Une fois que nous avons récupéré les valeurs d'élongation des commandes, nous pouvons passer au fonctionnement du parapente.

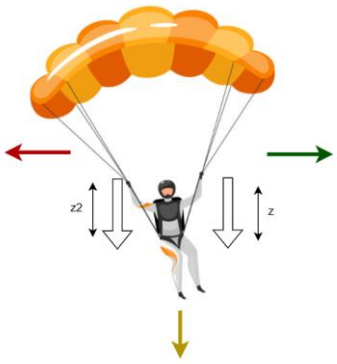

Figure 15. Schéma de fonctionnement du parapente

En regardant la figure 15 quand le joueur tire sur la manette droite, z2 diminue et le fait tourner à droite dans la direction de la flèche rouge et inversement avec la manette gauche et l'élongation z, il va dans la direction de la flèche verte. Quand les élongations de z2 et z sont tirées ensemble et donc diminuent en même temps, nous suivons la flèche jaune, le joueur tombe donc en chute libre.

En principe un parapente est toujours en chute selon une proportion de 10 m en avant et 1m en chute sans toucher aux contrôles (à moins de rencontrer un courant d'air chaud comme en partie 2.2.3). Nous avons donc essayé de respecter cela avec le script ParapenteController.cs qui :

1- Récupère les valeurs d'élongation z et z2.
2- Calcule le déplacement en fonction des valeurs d'élongation, de vitesse et de chute du joueur en figure 16.
3- Et finalement, applique les mouvements de translation et rotation au Rigidbody du joueur en figure 17.

```
// Accéder aux valeurs de z et z2 renvoyées par SerialRead.cs (Arduino)
float rightJoystickVal = serialRead.z;
float leftJoystickVal = serialRead.z2;

float difference = leftJoystickVal - rightJoystickVal;
elongation_value = 0;

// Si les valeurs des joysticks sont proches, calculer l'élongation
if (difference < 0.05 && difference > -0.05)
{
    mean_joystick_value = (leftJoystickVal + rightJoystickVal) / 2;
    elongation_value = (float)0.96 - mean_joystick_value;
}
// Calculer le déplacement en fonction de la vitesse avant et de la chute
forwardMovement = transform.forward * (forward_speed - (elongation_value * 300)) * Time.deltaTime;

if (!inStorm)
{
    downwardMovement = Vector3.down * (falling_speed + (elongation_value * 2000)) * Time.deltaTime;
}
```

Figure 16. Calcul du déplacement du parapente

```
// Appliquer le mouvement au Rigidbody pour gérer les collisions
Vector3 movement = forwardMovement + downwardMovement;
rb.MovePosition(rb.position + movement);

// Gestion de la rotation
rb.MoveRotation(rb.rotation * Quaternion.Euler(0, difference * speed_rotation, 0));
```

Figure 17. Application du mouvement au Rigidbody du joueur

2.2.4. Remontée des courant d'air chaud

Les remontées sont visibles, par le joueur, grâce aux effets de particules de tornade blanche en figure 18 :

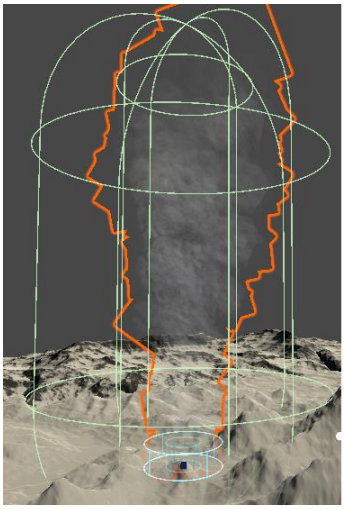

Figure 18. Courant d'air chaud pour la remontée

Pour les courants d'air chaud nous avons deux colliders qui permettent :

1- De détecter si le joueur est proche de la tornade pour « échauffer » les ventilateurs à atteindre une grande puissance.
2- De détecter si le joueur est au cœur du courant pour qu'il remonte et déclenche à pleine puissance les ventilateurs.

Dans le code ParapenteController.cs, nous avons donc une vérification qui nous permet de voir si le joueur est dans le cœur de la tornade. Si oui, nous le faisons remonter vers le haut en inversant le vecteur du mouvement vers le bas jusqu'à une limite fixée à une hauteur de 180000. Nous pouvons voir cela dans la figure 19.

```
else
{
    if (rb.transform.position.y < 18000)
    {
        downwardMovement = Vector3.up * 100 * Time.deltaTime;
    }
    else
    {
        downwardMovement = Vector3.up * 0 * Time.deltaTime;
    }
}
```

Figure 19. Code pour la remontée du parapente

2.2.5. Mise en place des ventilateurs

Le protocole utilisé par les ventilateurs pour communiquer est appelé DMX. C'est un protocole standardisé utilisé principalement dans le domaine de l'éclairage pour contrôler des équipements de types scéniques. Celui utilisé par les ventilateurs est le DMX 512, qui permet de contrôler un maximum de 512 canaux (2 seulement pour le projet) câblés en série, donc sur une seule ligne. Nous utilisons le contrôleur maître DMXIS par ENTTEC pour contrôler nos ventilateurs en utilisant une communication en série avec l'OS. Chaque canal peut prendre une valeur entre 0 et 255 pour contrôler l'intensité.

Nous pouvons voir en figure 20 la trame d'un signal DMX512 :

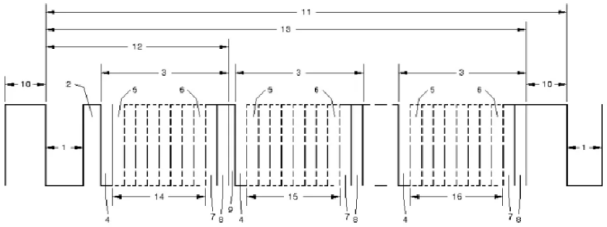

Figure 20. Trame DMX512

Grâce à ce schéma nous avons pu définir la trame en figure 21 qui est expliquée dans le tableau 1 :

```
// Initialisation de la trame DMX
dmxData[0] = 0x7E; // Start Code
dmxData[1] = 0x06; // Label
dmxData[2] = 0x00; // Data Length LSB
dmxData[3] = 0x02; // Data Length MSB (512 canaux)
dmxData[517] = 0xE7; // End Code
```

Figure 21. Initialisation de la trame DMX

Elément	Description	Position
Start Code	Début de la trame (0x7E).	dmxData[0]
Label	Code d'identification pour les données DMX (0x06 ici).	dmxData[1]
Data Length	Longueur des données envoyées (2 octets).	dmxData[2] et dmxData[3]
Data	Valeurs des canaux DMX (512 octets).	dmxData[4] à dmxData[515]
End Code	Fin de la trame (0xE7).	dmxData[517]

Tableau 1. Tableau descriptif de la trame DMX512

Le code pour contrôler les ventilateurs, ControllFan.cs, est disponible en annexe mais nous pouvons fournir un descriptif de ses principales fonctionnalités ici :

1- Le constructeur DMXController initialise le contrôleur avec un port spécifié, configure la trame par défaut et lance un thread pour envoyer les données périodiquement.
2- SetChannelValue(int channel, byte value) permet d'émettre une valeur pour un canal précis.

3- SendInitialDMXFrame() envoie une trame initiale en fixant tous les canaux à 0 pour activer le réseau de ventilateurs.
4- SendToDMX() est une fonction exécutée dans un thread. Elle envoie les trames DMX périodiquement (toutes les 25 ms).
5- Les fonctions Stop() et Dispose() arrêtent le thread et ferment le port série.

Cette classe est ensuite utilisée pour contrôler les ventilateurs grâce à la classe WindController.cs, disponible en annexes, en fonction de la vitesse de chute du joueur et de son entrée dans un courant d'air chaud.

2.3. Développement des PNJ avec IA

2.3.1. Animation des personnages 3D

Pour l'animation du modèle 3D visible en figure 2 nous avons utilisé Mixamo qui est une plateforme d'Adobe qui permet d'ajouter des animations réalistes et rigging (armature) automatique à des modèles 3D.

Nous avons donc sélectionné quatre animations qui sont :

- Idle : pour un état d'attente
- Walk : pour faire marcher le PNJ
- Run : pour faire courir le PNJ
- Chicken Dance : danse pour humilier le joueur quand il n'a pas su stopper le Dupont-de-Richeville (pour rappel c'est le nom donné aux personnages)

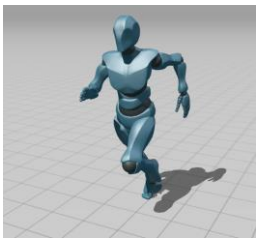

Figure 22. Exemple de Y-bot qui court sur Mixamo

Mixamo donne accès à un fichier FBX qui contient le modèle 3D de l'exemple, le squelette d'animation, les textures et matériaux, mais surtout l'animation qui va nous intéresser.
Il faut donc rendre le modèle FBX humanoïde et extraire l'animation comme visible en figure 23.

Figure 23. Extraction de l'animation

Nous attachons à notre modèle 3D de DdR un Animator pour contrôler ces animations, et nous les glissons dedans.

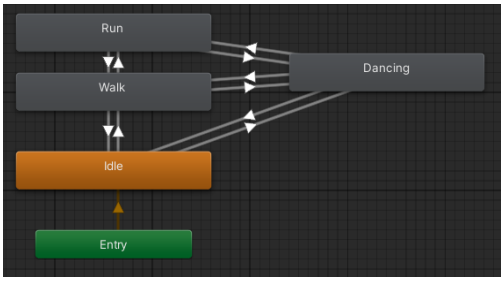

Figure 24. Animations dans l'Animator de Unity

L'Animator dans notre cas fonctionne comme une Final State Machine. Pour passer d'un état à l'autre, nous devons remplir des conditions, ici booléennes. Une fois remplies, celles-ci permettent des transitions (flèches blanches dans la figure 24) pour passer d'une animation à une autre.
Ici nous avons dix transitions qui nous permettent de naviguer entre nos quatre animations.

2.3.2. NavMesh pour la navigation

Le NavMesh est une zone navigable sous forme de maillage (mesh) où un agent peut se déplacer. Il repose sur des algorithmes de pathfinding et gestion de maillage pour naviguer de manière optimale dans un environnement 3D.

NavMesh utilise le très performant A*, qui est un algorithme de pathfinding qui permet aux agents de trouver le chemin optimal. Sur Unity, chaque nœud à explorer représente une zone navigable. A* utilise donc une fonction heuristique pour partir d'un point initial, calculer un score, choisir le nœud avec le f(n) le plus faible jusqu' à atteindre le nœud final et reconstruire le chemin optimal. Le pseudo-code est détaillé en annexe en partie 5.1.1.

Il utilise également des algorithmes de triangulation pour diviser en triangles les zones navigables de la manière la plus optimale (Triangulation de Delaunay, détaillée en annexe en partie 5.1.2). Le flood file, pour identifier les zones navigables connectées en partant d'un point et en explorant toutes les surfaces atteignables à la manière de l'algorithme Breadth First Search. Et en simplifiant et supprimant les points inutiles.
Le résultat offre une représentation de polygones, idéale pour le pathfinding.

Pour éviter les contacts entre agents ou avec obstacles, Unity utilise le Reciprocal Velocity Obstacle Algorithm. Il est utilisé pour prédire des collisions potentielles et modifie la trajectoire de notre agent. Ce qui peut être utile dans le cas de système multi-agents comme le nôtre.

Dans un premier temps, nous avons appliqué le NavMesh sur toute la surface dite navigable, comme visible en figure 25 Cela crée un composant NavMeshSurface que l'on configure pour nos PNJ comme en figure 26.

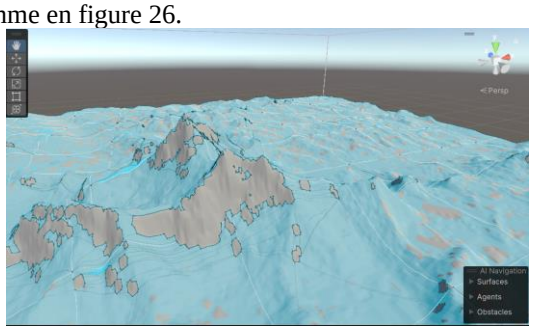

Figure 25. NavMesh appliqué sur le terrain principal

Les zones bleues représentent les surfaces navigables où le personnage peut naviguer.

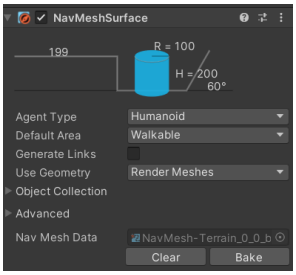

Figure 26. Configuration de la Surface NavMesh

Il suffit maintenant d'assigner un composant NavMeshAgent aux PNJ comme visible en figure 27.

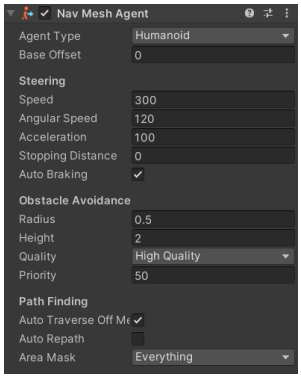

Figure 27. Configuration de l'agent NavMesh

Il faut aussi assigner un Transform au sommet du mont Everest qui sera le point de destination, d'ajuster l'accélération ainsi que la vitesse comme dans le code en figure 28.

```
ennemy.speed = 300;
ennemy.acceleration = 100;
ennemy.SetDestination(target.position); //Lance la logique run to target
```

Figure 28. Réglage de la destination du NavMeshAgent

Une première observation, c'est que le terrain est tellement grand (pour rappel 100_000m x 100_000m x 10_000m) que si nous fixons le point target en haut de l'Everest alors la recherche du chemin optimal prend beaucoup trop de temps et le personnage tourne en rond.

Nous avons donc mis en place un système de NavMesh Local visible en figure 29

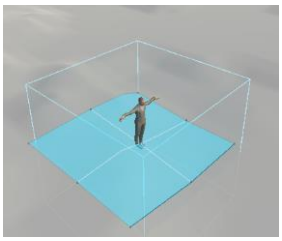

Figure 29. NavMesh "local" du PNJ

Le NavMesh est simplement généré autour du joueur toutes les 5 secondes. Cependant comme le point de destination final n'est pas toujours proche du joueur, nous définissons donc un point intermédiaire le plus proche possible du final et dans la zone navigable. Voici comment cela fonctionne de manière schématisée en figure 30.

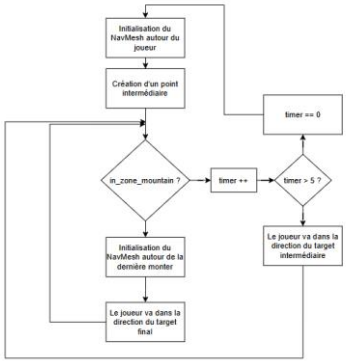

Figure 30. Schéma simplifié du fonctionnement avec NavMesh Local

Cette seconde solution a dû être malheureusement abandonnée pour revenir à la première car la génération de NavMesh dit local crée des bugs dans le jeu en général quand les PNJ sont trop nombreux. Ce choix à été fait dans le but d'améliorer les performances du jeu.

2.3.3. Final State Machine

Une FSM ou machine à états finis, est un modèle utilisé pour décrire un système qui peut se trouver dans un ensemble limité d'états prédéfinis. Elle peut être dans un état ou dans l'autre en fonction d'un certain input.

Le code de la Final State Machine est complet en annexe, mais voici, en figure 31 un schéma pour résumer son fonctionnement.

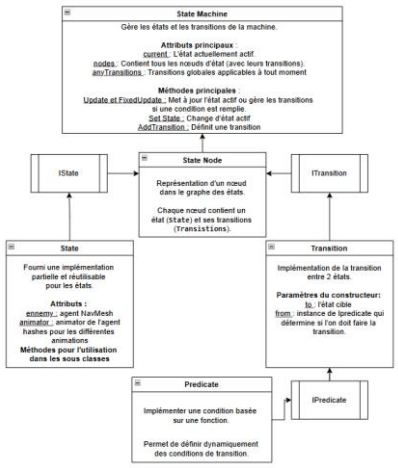

Figure 31. Schéma de fonctionnement du code utilisé pour la FSM

L'interface :

- IPredicate : sert à définir une condition qui retourne un booléen pour décider si une transition doit être activée.
- IState : sert à définir le comportement de base de tout état.
- ITransition : sert à définir une transition entre deux états.

Grâce à ce code, nous pouvons définir nos états selon le diagramme simplifié présent en figure 32.

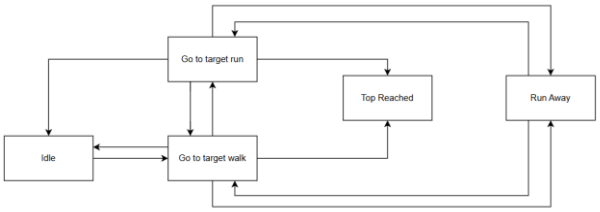

Figure 32. Diagrammes des états d'un PNJ

Dans ce schéma, nous voyons cinq états :

- Idle : Etat d'attente du PNJ pour réfléchir au chemin.
- Top Reached : Etat du PNJ quand il arrive au sommet et effectue sa danse de la victoire.
- Go To Target Walk : Le personnage marche vers le sommet en toute impunité.
- Go To Target Run : Le personnage cours en direction du sommet car le joueur est proche.
- Run Away : Le PNJ s'enfuit vers l'un des points de fuite le plus proche.

Avec la définition du fonctionnement des états et de la final state machine nous pouvons passer aux codes (entièrement disponibles en annexes).
Tout d'abord nous initialisons nos états (Classes IdleState, TopReachState, RunAway, GoToTargetWalk et GoToTargetRun disponibles en annexe), nous les implémentons dans le script EnnemyControllerFinal.cs (figure 34) et nous créons nos transitions (figure 35).

```
idleState = new IdleState(navMeshAgent, animator);//Init Idle State
goToTargetStateWalk = new GoToTargetWalk(navMeshAgent, animator,target);// Init GoToTarget State
goToTargetStateRun = new GoToTargetRun(navMeshAgent, animator,target); //Init GoToTarget State
topReachState = new TopReachState(navMeshAgent, animator);//Init TopReach State
runAway = new RunAway(navMeshAgent, animator, target, runAwayTarget);//Init RunAway State
```

Figure 33. Initialisation des états dans le code EnnemyController.cs

```
stateMachine.AddTransition(idleState, goToTargetStateWalk, new FuncPredicate(() => IsGameStarted)); //Transition idleState --> To target Walk
stateMachine.AddTransition(goToTargetStateWalk, idleState, new FuncPredicate(() => IsGameStarted)); //Transition idleState --> To target Walk
stateMachine.AddTransition(goToTargetStateWalk, goToTargetStateRun, new FuncPredicate(() => playerIsNear)); // Transition Walk -> Run
stateMachine.AddTransition(goToTargetStateRun, goToTargetStateWalk, new FuncPredicate(() => !playerIsNear));// Transition Run -> Walk
stateMachine.AddTransition(goToTargetStateWalk, topReachState, new FuncPredicate(() => playerInTop));//Transition Walk --> Dancing
stateMachine.AddTransition(goToTargetStateRun, topReachState, new FuncPredicate(() => playerInTop));//Transition Run --> Dancing
stateMachine.AddTransition(goToTargetStateWalk, runAway, new FuncPredicate(() => newDirection));//Transition Walk --> RunAway
stateMachine.AddTransition(goToTargetStateRun, runAway, new FuncPredicate(() => newDirection));//Transition Run --> RunAway
stateMachine.AddTransition(runAway, goToTargetStateWalk, new FuncPredicate(() => !playerInZoneScript.playerInZone));//Transition RunAway --> Walk
stateMachine.AddTransition(runAway, goToTargetStateRun, new FuncPredicate(() => !playerInZoneScript.playerInZone));//Transition RunAway --> Run
```

Figure 34. Initialisation des transitions dans le code EnnemyController.cs

Le code EnnemyController.cs est disponible en annexes, mais voici un pseudo-code pour le décrire :

Classe EnnemyController :

Initialisation des variables nécessaires (spawner, stateMachine, navMeshAgent, etc.)

Fonction Awake :
Trouve et initialise les objets nécessaires (NavMeshMount, NavMeshAgent, Animator)

Fonction Start :
Initialise l'état de départ et les composants (StateMachine, PlayerInZone, localNavMeshController)
Défini les états et transitions entre états

Fonction Update :
Si le joueur n'est plus dans la zone :
Désactive la nouvelle direction
Réinitialise la cible finale
Si la fuite est activée :
Définie la cible finale comme le point de fuite
Met à jour la cible des états
Met à jour la machine d'états

Fonction OnTriggerEnter (détection des collisions) :
Si collision avec une montagne :
Active inZoneMountain
Si collision avec un ennemi détecteur :
Active playerIsNear
Si collision avec la destination finale :
Active playerInTop

Fonction OnTriggerExit (sortie de zone) :
Si sortie de la montagne :
Désactive l'état inZoneMountain
Si sortie de la zone de détection de l'ennemi :
Désactive l'état playerIsNear

Fonction OnDestroy :
Notifie les ennemis proches de la destruction
Informe le spawner de la destruction
Déclenche une explosion visuelle

Fonction NotifyNearbyEnemies :
Parcourt les ennemis proches dans un rayon donné
Envoie un message pour activer le mode fuite

Fonction ActivateFleeMode :
Si le mode fuite doit être activé :
Défini une nouvelle direction

Nous voyons également que quand un personnage meurt, il transmet un message, dans un rayon défini, aux autres personnages non-joueurs. Lui-même peut recevoir un message ainsi. Si ce message est reçu, alors cela enclenche le mode fuite du personnage qui a peur de terminer comme son collègue.

2.4. UI et jeu

2.4.1. Tutoriel et fin de jeu

Un tutoriel expliquant le fonctionnement du jeu a été créé grâce au Game Manager. Nous avons créé un Canvas auquel nous avons lié plusieurs Game Object listés de 1 à 8 (8 ne faisant pas partie du tutoriel car c'est un écran de fin). Le Canvas est lié au Player et placé devant l'emplacement de la caméra pour que le joueur le voit correctement comme en figure 35.

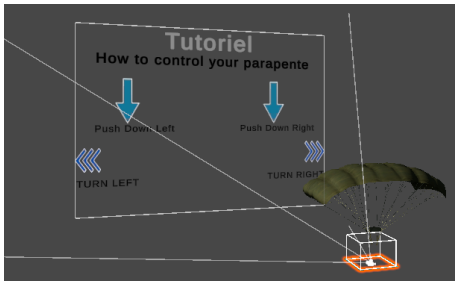

Figure 35. Vue du tutoriel

Les Game Object UI_x, avec x dans [1-7], s'enchainent toutes les cinq secondes pour donner un premier aperçu de comment jouer au jeu.

Quand le Collider du joueur touche le Mesh Collider du terrain, cela enclenche la fin de partie qui replace le joueur au point de départ et lui affiche un message de Game Over.

Toutes ses explications sont résumées avec la figure 36 qui reprend le Game Manager du jeu en schéma.

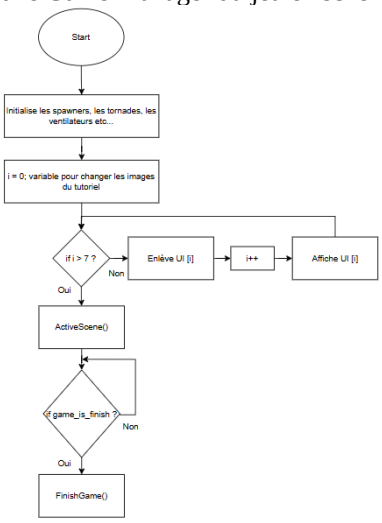

Figure 36. Schéma simplifié du Game Manager

La fonction ActiveScene(), nous sert simplement à activer tout ce qui a été initialisé dans le start ainsi que les scripts C# pour activer les PNJ, le parapente et les tornades.
La fonction FinishGame() replace le joueur à la position de départ, stop tout et affiche le UI de fin de jeu.

2.4.2. Logique de tir sur les ennemis

Pour empêcher les PNJ de gravir la montagne le joueur doit se rapprocher d'eux, les avoir dans sa ligne de mire (qu'il soit dans l'angle de la Caméra) et attendre un délai défini.

```
0 Message Unity | 0 références
void Update()
{

    if (ennemyInZone && enemiesInZone.Count > 0)
    {

        foreach (var ennemy in enemiesInZone)
        {
            if (ennemy != null)
            {
                UpdateLineRenderer(this.transform.position, ennemy.transform.position, ennemy);
                try
                {
                    if (IsObjectVisible(ennemy))
                    {
                        shoot = true;
                    }
                    else
                    {
                        shoot = false;
                    }
                }
                catch (NullReferenceException) { }
            }
        }

    }
}
```

Figure 37. Code Trigger Script pour lancer le script Shoot.cs sur l'ennemi

Dès qu'un ennemi rentre dans la zone du joueur, il est ajouté à la liste *ennemyInZone*. Ensuite le joueur parcourt cette liste et trace une ligne en direction de l'ennemi qui est mis à jour tous les updates pour suivre le PNJ qui bouge dans la Scène. S'il est dans le champ de la caméra alors cela déclenche le script Shoot.cs qui charge pendant un delay avant de tirer sur le PNJ selon une logique « FIFS » (First In First Shoot).

3. CONCLUSION

Le projet « Parapente au-dessus de l'Everest » est une réalisation mélangeant Réalité Virtuelle, modélisation 3D et intelligence artificielle. Il avait pour but de créer une expérience amusante et immersive en intégrant des éléments interactifs, comme la simulation réaliste de parapente et les comportements autonomes des personnages non joueurs.

Grâce à l'utilisation de Unity, de l'HTC Vive et d'algorithmes comme les FSM, NavMesh et SMA, ce projet a atteint son objectif principal : immerger le joueur dans un parapente pour jouer contre des PNJ qui grimpent jusqu'au sommet.

En conclusion, ce projet n'est pas seulement un projet d'école, mais une opportunité pour réfléchir aux impacts environnementaux et à notre responsabilité collective face aux merveilles de ce monde telles que l'Everest. Nous terminerons par spécifier qu'à travers cette simulation, l'objectif réel n'était pas simplement de divertir, mais aussi sensibiliser à la préservation de notre planète.

4. RÉFÉRENCES

Cours de Réalité Augmenté, LEVAILLANT Gwendal, 17 août 2021.

Cours de Machine Learning, DEGEEST Alexandra

Cours d'Intelligence artificielle, DEGEEST Alexandra.

Modèle 3D Suit Character Pack - https://assetstore.unity.com/packages/3d/characters/humanoids/suit-character-pack-generic-16772

Documentation de Terrain Tools : https://docs.unity3d.com/Packages/com.unity.terrain-tools@5.0/manual/index.html

Package des terrains pour l'horizon : https://assetstore.unity.com/packages/tools/terrain/terrain-tools-64852

Package utilisés pour la skybox : https://assetstore.unity.com/packages/2d/textures-materials/sky/skybox-series-free-103633

Lien vers Mixamo pour les animations : https://mixamo.com/#/

Documentation pour le NavMesh : https://docs.unity3d.com/2020.1/Documentation/Manual/nav-BuildingNavMesh.html

Triangulation de Delaunay pour NavMesh : https://fr.wikipedia.org/wiki/Triangulation_de_Delaunay

Documentation du HTC Vive : https://www.vive.com/fr/support/vive/category_howto/choosing-your-play-area.html

Explication du protocole DMX : https://www.interfacedmx.fr/quest-ce-que-le-protocole-dmx-512/

Final State Machine : https://www.youtube.com/watch?v=NnH6ZK5jt7o&t=140s

5. ANNEXES

5.1. Navigation Mesh

*5.1.1. Algorithme A**

```
Structure noud = {
   x, y: Nombre
   cout, heuristique: Nombre
}

depart = Noud(x=_, y=_, cout=0, heuristique=0)

Fonction compareParHeuristique(n1:Noud, n2:Noud)
    si n1.heuristique < n2.heuristique
        retourner 1
    ou si n1.heuristique == n2.heuristique
        retourner 0
    sinon
        retourner -1

Fonction cheminPlusCourt(g:Graphe, objectif:Noud, depart:Noud)
    closedList = File()
    openList = FilePrioritaire(comparateur = compareParHeuristique)
    openList.ajouter(depart)
    tant que openList n'est pas vide
        u = openList.defiler()
        si u.x == objectif.x et u.y == objectif.y
            reconstituerChemin(u)
            terminer le programme
        pour chaque voisin v de u dans g
            si non(    v existe dans closedList
                       ou v existe dans openList avec un coût inférieur)
                v.cout = u.cout +1
                v.heuristique = v.cout + distance([v.x, v.y], [objectif.x, objectif.y])
                openList.ajouter(v)
        closedList.ajouter(u)
    terminer le programme (avec erreur)
```

Figure 38. Pseudo Code de A*

5.1.2. Triangulation de Delaunay

Une triangulation de Delaunay dans un ensemble de points P est un réseau de triangle qui respectent les points suivants :

Le cercles circonscrit de chaque triangle ne contient aucun autre point de P **à l'intérieur de lui.**

Donc pour chaque triangle de la triangulation, les sommets de ce triangle sont les seuls points à se trouver sur ou à l'intérieur du cercle qui passe par ces trois sommets.

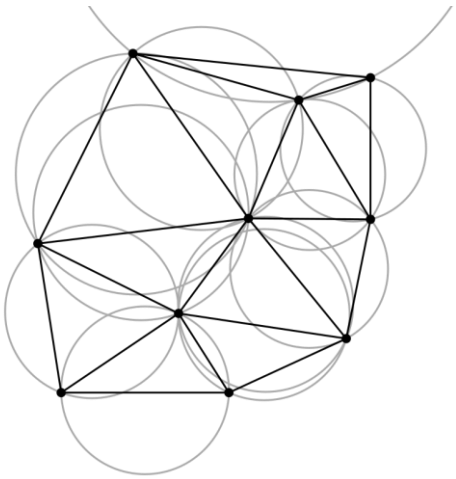

Figure 39. Exemple de la triangulation de Delaunay

5.1.3. Fonctionnement de l'algorithme RVO

1- Chaque agent détecte les autres agents et les obstacles dans un rayon défini comme étant sa perception.
2- Pour chaque agent détecté ou obstacle, un VO (Velocity Obstacle) est calculé pour avoir les vitesses à éviter.
3- Les VO sont combinées réciproquement pour générer des zones dites RVO où l'agent peut se déplacer sans risque de collision.

Ensuite chaque agent choisit la vitesse optimale en dehors de la zone de collision et la met à jour quand c'est nécessaire

5.2. Code

5.2.1. Serial Read.cs

```
private float ExtractValue (string segment, bool plus_minus)
{
    string valuePart;
    float value;
    // On enlève les informations inutiles
    if (plus_minus)
    {
        valuePart = segment.Substring(3); // Ignorer le premier caractère (z ou z2)
    }
    else
    {
        valuePart = segment.Substring(1); // Ignorer le premier caractère (z ou z2)
    }

    // Récupération de la valeur en excluant la partie voulue
    if (float.TryParse(valuePart, out value))
    {
        return value;
    }
    else
    {
        Debug.LogError($"Failed to parse value from segment: {segment}");
        return 0f; // Valeur par défaut en cas d'échec
    }
}
```

5.2.2. ControllFan.cs

```
public class DMXController : IDisposable
{
    private SerialPort serialPort;
    private byte[] dmxData = new byte[518]; // StartCode (1) + Label (1) + Data Length (2) + Data (512) + EndCode (1)
    private Thread dmxThread;
    private bool isRunning;
    1 référence
    public DMXController(string portName, int baudRate = 250000)
    {
        // Initialisation de la trame DMX
        dmxData[0] = 0x7E; // Start Code
        dmxData[1] = 0x06; // Label
        dmxData[2] = 0x00; // Data Length LSB
        dmxData[3] = 0x02; // Data Length MSB (512 canaux)
        dmxData[517] = 0xE7; // End Code
        try
        {
            // Initialisation du port série
            serialPort = new SerialPort(portName, baudRate);
            serialPort.Open();

            if (serialPort.IsOpen)
            {
                SendInitialDMXFrame();
                isRunning = true;
                dmxThread = new Thread(SendToDMX);
                dmxThread.Start();
            }
            else
            {
                throw new InvalidOperationException("Le port série n'a pas pu être ouvert.");
            }
        }
        catch (Exception e)
        {
            throw new InvalidOperationException($"Erreur lors de l'ouverture du port série : {e.Message}", e);
        }
    }
```

```
1 référence
private void SendInitialDMXFrame()
{
    try
    {
        for (int i = 1; i <= 512; i++)
        {
            SetChannelValue(i, 0);
        }
        serialPort.Write(dmxData, 0, dmxData.Length);
        Thread.Sleep(100);
    }
    catch (Exception e)
    {
        Console.Error.WriteLine($"Erreur lors de l'envoi de la trame initiale : {e.Message}");
    }
}

1 référence
private void SendToDMX()
{
    while (isRunning)
    {
        try
        {
            if (serialPort != null && serialPort.IsOpen)
            {
                // Envoyer la trame DMX
                serialPort.Write(dmxData, 0, dmxData.Length);

                // Pause pour respecter le timing DMX
                Thread.Sleep(25);
            }
        }
        catch (Exception e)
        {
            Console.Error.WriteLine($"Erreur lors de l'envoi au DMX : {e.Message}");
        }
    }
}
public void SetChannelValue(int channel, byte value)
{
    if (channel < 0 || channel > 512)
        throw new ArgumentOutOfRangeException(nameof(channel), "Le canal doit être entre 1 et 512.");

    dmxData[4 + channel] = value; // Les canaux DMX commencent à l'indice 4 dans la trame
}
1 référence
public void Stop()
{
    if (serialPort != null && serialPort.IsOpen)
    {
        serialPort.Close();
    }
    isRunning = false;
}

2 références
public void Dispose()
{
    try
    {
        SetChannelValue(1, 0);
        SetChannelValue(2, 0);
        serialPort.Write(dmxData, 0, dmxData.Length);

        Stop();

        if (dmxThread != null && dmxThread.IsAlive)
        {
            dmxThread.Join();
        }

        serialPort?.Dispose();
    } catch (Exception) {}

}
```

5.2.3. *WindController.cs*

```
public class WindController : MonoBehaviour
{
    private DMXController DMXController;
    public Transform playerTransform;
    public float speed;
    private Vector3 lastPosition;
    private float timeElapsed;

    private ParapenteController player;

    private bool inStormZone;
    private bool inStormHearth;

    private bool[] states = new bool[5];

    Message Unity | 0 références
    void Awake()
    {
        for (int i = 0; i < states.Length; i++)
        {
            states[i] = false;
        }
        player = FindObjectOfType<ParapenteController>();
        try
        {
            DMXController = new DMXController( "COM4");
        }
        catch (System.Exception e)
        {
            Debug.LogError($"Erreur lors de l'initialisation du DMX Controller : {e.Message}");
        }
        lastPosition = playerTransform.position;
        timeElapsed = 0f;
```

```
void Update()
{
    if (finishTest && !finish)
    {
        finish = true;
        StopDmx();
        DMXController.Dispose();
    }
    else if (!finishTest)
    {
        if (!inStormZone || !inStormHearth)
        {
            positionWindController();
        }
    }
}

Message Unity | 0 références
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("StormZone"))
    {
        inStormZone = true;
        DMXController.SetChannelValue(1, 150);
        DMXController.SetChannelValue(2, 150);
    }

    if (other.CompareTag("StormHearth"))
    {
        player.inStorm = true;
        inStormHearth = true;
        DMXController.SetChannelValue(1, 200);
        DMXController.SetChannelValue(2, 200);
    }
}
```

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("StormZone"))
    {
        inStormZone = true;
        DMXController.SetChannelValue(1, 150);
        DMXController.SetChannelValue(2, 150);
    }

    if (other.CompareTag("StormHearth"))
    {
        player.inStorm = true;
        inStormHearth = true;
        DMXController.SetChannelValue(1, 200);
        DMXController.SetChannelValue(2, 200);
    }
}
Message Unity | 0 références
private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("StormZone"))
    {

        inStormZone = false;
    }
    if (other.CompareTag("StormHearth"))
    {
        player.inStorm = false;
        inStormHearth = false;
        DMXController.SetChannelValue(1, 100);
        DMXController.SetChannelValue(2, 100);
    }
}
```

```
private void positionWindController()
{
    if (player.downwardMovement.y > -3 && !states[0])
    {
        states[0] = true;
        ReinitStates(0);
        DMXController.SetChannelValue(1, 50);
        DMXController.SetChannelValue(2, 80);
    }
    if (player.downwardMovement.y < -3 && !states[1])
    {
        states[1] = true;
        ReinitStates(1);
        DMXController.SetChannelValue(1, 90);
        DMXController.SetChannelValue(2, 80);
    }

    if (player.downwardMovement.y < -5 && !states[2])
    {
        states[2] = true;
        ReinitStates(2);
        DMXController.SetChannelValue(1, 130);
        DMXController.SetChannelValue(2, 80);
    }
    if (player.downwardMovement.y < -8 && !states[3])
    {
        states[3] = true;
        ReinitStates(3);
        DMXController.SetChannelValue(1, 170);
        DMXController.SetChannelValue(2, 80);
    }

    if (player.downwardMovement.y < -12 && !states[4])
    {
        states[4] = true;
        ReinitStates(4);
        DMXController.SetChannelValue(1, 210);
        DMXController.SetChannelValue(2, 80);
    }
}
```

```
5 références
private void ReinitStates(int exceptNumber)
{
    for (int i=0; i<states.Length; i++)
    {
        if(i != exceptNumber)
        {
            states[i] = false;
        }
    }
}

2 références
public void StopDmx()
{
    if (DMXController != null)
    {
        DMXController.SetChannelValue(1, 0);
        DMXController.SetChannelValue(2, 0);
    }

}

Message Unity | 0 références
private void OnApplicationQuit()
{
    if (DMXController != null)
    {
        DMXController.Dispose();
    }
}
```

5.3. Codes pour la Final State Machine

5.3.1. Classe State Machine

```
public class StateMachine
{
    StateNode current;
    Dictionary<Type, StateNode> nodes = new
Dictionary<Type, StateNode>();
    HashSet<ITransistion> anyTransitions = new
HashSet<ITransistion>();

    public void Update()
    {
        var transition = GetTransition();
        if (transition != null)
            ChangeState(transition.To);

        current.State?.Update();
    }

    public void FixedUpdate()
    {
        current.State?.FixedUpdate();
    }

    public void SetState(IState state)
    {
        current = nodes[state.GetType()];
        current.State?.onEnter();
    }

    void ChangeState(IState state)
    {
        if (state == current.State) return;

        var previousState = current.State;
        var nextState =
nodes[state.GetType()].State;

        previousState?.onExit();
        nextState?.onEnter();
        current = nodes[state.GetType()];
    }
    ITransistion GetTransition()
    {
        foreach (var transition in
anyTransitions)
        {
            if (transition.Condition.Evaluate())
            {
                return transition;
            }
        }
        foreach (var transition in
current.Transistions)
        {
            if(transition.Condition.Evaluate()
){
                return transition;
            }
        }
        return null;
    }
    public void AddTransition(IState from,
IState to ,IPredicate condition)
    {

GetOrAddNode(from).AddTransition(GetOrAddNode(to
).State, condition);
    }

    public void AddAnyTransition(IState
to,IPredicate condition)
    {
        anyTransitions.Add(new Transition(to,
condition));
    }

    StateNode GetOrAddNode(IState state)
    {
        var node =
nodes.GetValueOrDefault(state.GetType());

        if (node == null)
        {
            node = new StateNode(state);
            nodes.Add(state.GetType(), node);
        }

        return node;
    }
```

5.3.2. Classe StateNode

```
class StateNode
    {
        public IState State { get;}
        public HashSet<ITransistion>
Transistions { get; }

        public StateNode(IState state)
        {
            State = state;
            Transistions = new
HashSet<ITransistion>();

        }

        public void AddTransition(IState to ,
IPredicate condition)
        {
            Transistions.Add(new Transition(to,
condition));
        }
    }
```

```
}
```

5.3.3. *Interface IState*

```
public interface IState
{

    //Définition des méthodes de la classe
State, il peut y en avoir plusieurs
    void onEnter(); //Quand on rentre dans
l'état
    void Update(); // Update quand l'état est
activé
    void FixedUpdate(); //Update pour gérer la
physique
    void onExit();//Quand on sort de l'état
}
```

5.3.4. *Classe State*

```
public abstract class BaseState: IState
{
    //Toutes les State doivent au moins
connaitre qui est l'ennemy et son animator
    protected readonly NavMeshAgent ennemy;
    protected readonly Animator animator;

    protected static readonly int runHash =
Animator.StringToHash("isRunning");
    protected static readonly int walkHash =
Animator.StringToHash("isWalking");
    protected static readonly int dancingHash =
Animator.StringToHash("isDancing");

    protected BaseState(NavMeshAgent ennemy,
Animator animator)
    {
        this.ennemy = ennemy;
        this.animator = animator;
    }
    public virtual void FixedUpdate()
    {
        //noop
    }

    public virtual void onEnter()
    {
        //noop
    }

    public virtual void onExit()
    {
        //noop
    }

    public virtual void Update()
    {
        //noop
    }
}
```

5.3.5. *Classe Transition*

```
public class Transition : ITransistion
{
    public IState To { get; }
    public IPredicate Condition { get; }

    public Transition(IState to ,IPredicate
condition)
    {
        To = to;
        Condition = condition;
    }
}
```

5.3.6. *Interface ITransition*

```
public interface ITransistion
{
    IState To { get; }
    IPredicate Condition { get; }
}
```

5.3.7. *Classe Predicate*

```
public class FuncPredicate : IPredicate
{
    readonly Func<bool> func;

    public FuncPredicate(Func<bool> func)
    {
        this.func = func;
    }
    public bool Evaluate() => func.Invoke();
}
```

5.3.8. *Interface IPredicate*

```
public interface IPredicate
{
    bool Evaluate();
}
```

5.3.9. *State IdleState*

```
public class IdleState : BaseState
{

    public IdleState(NavMeshAgent ennemy,
Animator animator ) : base(ennemy, animator) { }

    public override void onEnter()
    {
        ennemy.isStopped = true;
        Debug.Log("Enter IdleState");
    }

    public override void onExit()
    {
        Debug.Log("Exit IdleState");
    }

}
```

5.3.10. *State GoToTargetWalk*

```
public class GoToTargetWalk : BaseState
{
    public Transform target;
    public GoToTargetWalk(NavMeshAgent ennemy,
Animator animator, Transform target) :
base(ennemy, animator)
    {
        this.target = target;
    }

    public override void onEnter()
```

```
    {
        animator.SetTrigger(walkHash);//Active
l'animation walkHash
        ennemy.isStopped = false;
        ennemy.speed = 100;
        ennemy.acceleration = 25;
        ennemy.SetDestination(target.position);
//Lance la logique run to target

        Debug.Log("Enter GoToTarget");
    }

    public override void onExit()
    {
        Debug.Log("Exit GoToTarget");
        animator.ResetTrigger(walkHash);//Reset
le walkHash
    }

    public override void Update()
    {
        ennemy.SetDestination(target.position);
// Met à jour la destination en permanence
    }
```

5.3.11. *State GoToTargetRun*

```
public class GoToTargetRun : BaseState
{
    public Transform target;
    public GoToTargetRun(NavMeshAgent ennemy,
Animator animator, Transform target) :
base(ennemy, animator)
    {
        this.target = target;
    }

    public override void onEnter()
    {
        animator.SetTrigger(runHash);//Active
l'animation walkHash
        ennemy.isStopped = false;
        ennemy.speed = 300;
        ennemy.acceleration = 100;
        ennemy.SetDestination(target.position);
//Lance la logique run to target

        Debug.Log("Enter GoToTarget");
    }

    public override void onExit()
    {
        Debug.Log("Exit GoToTarget");
        animator.ResetTrigger(runHash);//Reset
le walkHash
    }

    public override void Update()
    {
        ennemy.SetDestination(target.position);
// Met à jour la destination en permanence
    }
}
```

5.3.12. *State RunAway*

```
public class RunAway : BaseState
{
    List<Transform> runAwayTarget;
    public Transform escapePosition;
    public Transform target;

    public RunAway(NavMeshAgent ennemy, Animator
animator, Transform target,List<Transform>
runAwayTarget) : base(ennemy, animator) {
        this.runAwayTarget = runAwayTarget;
        this.target = target;
            }

    private Transform FindClosestTarget()
    {
        Transform closestTarget = null;
        float closestDistance = Mathf.Infinity;

        foreach (Transform target in
runAwayTarget)
        {
            float distance =
Vector3.Distance(ennemy.transform.position,
target.position);
            if (distance < closestDistance)
            {
                closestDistance = distance;
                closestTarget = target;
            }
        }

        return closestTarget;
    }

    public override void onEnter()
    {
        Debug.Log("RunAway Enter");

        Transform closestTarget =
FindClosestTarget();
        escapePosition = closestTarget;
    }

    public override void Update()
    {
        animator.SetTrigger(runHash);
        ennemy.speed = 450;
        ennemy.acceleration = 160;
        ennemy.SetDestination(target.position);
    }

    public override void onExit()
    {
        Debug.Log("RunAway Exit");
    }

}
```

5.3.13. *State TopReachState*

```
public class TopReachState : BaseState
{
    private float timer = 5f;
    public TopReachState(NavMeshAgent ennemy,
Animator animator) : base(ennemy, animator) { }

    public override void onEnter()
    {
        animator.SetTrigger(dancingHash);
        ennemy.isStopped = true;
        Debug.Log("Enter TopReachState");
        timer = 5f;
    }

    public override void onExit()
    {
        animator.ResetTrigger(dancingHash);
        Debug.Log("Exit TopReachState");
    }
```

```
    public override void Update()
    {
        base.Update();
        timer -= Time.deltaTime;

        if (timer <= 0f)
        {

UnityEngine.Object.Destroy(ennemy.gameObject);
        }
    }

}
```

5.4. Script EnnemyControllerFinal.cs

```
public class EnnemyController : MonoBehaviour
{
    public EnnemySpawner spawner;
    private StateMachine stateMachine;
    private NavMeshAgent navMeshAgent;
    private Animator animator;
    private localNavMeshController
localNavMeshController;

    public GameObject explosionOnHitEnnemy;
    private NavMeshSurface navMeshMountain;
    private NavMeshSurface localNavMeshSurface;
    public GameObject terrainGameObject;
    public NavMeshSurface navMeshSurfaceModel;
    private bool inZoneMountain = false;
    private float timer = 0f;
    private bool playerIsNear = false;
    private bool playerInTop = false;

    public Transform finalTarget;
    private Transform tempFinalTarget;
    private Transform target;
    public List<Transform> runAwayTarget = new
List<Transform>();
    private bool newDirection = false;
    private PlayerInZone playerInZoneScript;

    IdleState idleState;
    GoToTargetWalk goToTargetStateWalk;
    GoToTargetRun goToTargetStateRun;
    TopReachState topReachState;
    RunAway runAway;

    private bool IsGameStarted = false;
    private void Awake()
    {
        GameObject navMeshMountObject =
GameObject.Find("NavMeshMount");
        navMeshMountain =
navMeshMountObject.GetComponent<NavMeshSurface>(
);
        navMeshAgent =
GetComponent<NavMeshAgent>();//Init NavMeshAgent
        animator = GetComponent<Animator>();
//Init Animator
    }
    private void Start()
    {
        //Check du commencement de la partie
        IsGameStarted= true;
        // Assigné la valeur de fin à la target
        tempFinalTarget = finalTarget;


        stateMachine = new StateMachine();//Init
StateMachine

        playerInZoneScript =
GetComponentInChildren<PlayerInZone>();
        Collider agentCollider =
GetComponent<Collider>();

        if (agentCollider != null)
        {
            agentCollider.isTrigger = true;
        }
        target = finalTarget;

        localNavMeshController = new
localNavMeshController(navMeshSurfaceModel,
this.gameObject);//Init localNavMeshController

        target =
localNavMeshController.UpdateIntermediateDestina
tion2(tempFinalTarget,
navMeshAgent.transform.position);

        idleState = new IdleState(navMeshAgent,
animator);//Init Idle State
        goToTargetStateWalk = new
GoToTargetWalk(navMeshAgent, animator,target);//
Init GoToTarget State
        goToTargetStateRun = new
GoToTargetRun(navMeshAgent, animator,target);
//Init GoToTarget State
        topReachState = new
TopReachState(navMeshAgent, animator);//Init
TopReach State
        runAway = new RunAway(navMeshAgent,
animator, target, runAwayTarget);//Init RunAway
State



        stateMachine.AddTransition(idleState,
goToTargetStateWalk, new FuncPredicate(() =>
IsGameStarted)); //Transition idleState --> To
target Walk

stateMachine.AddTransition(goToTargetStateWalk,
idleState, new FuncPredicate(() =>
IsGameStarted)); //Transition idleState --> To
target Walk

stateMachine.AddTransition(goToTargetStateWalk,
goToTargetStateRun, new FuncPredicate(() =>
playerIsNear)); // Transition Walk -> Run

stateMachine.AddTransition(goToTargetStateRun,
goToTargetStateWalk, new FuncPredicate(() =>
!playerIsNear));// Transition Run -> Walk

stateMachine.AddTransition(goToTargetStateWalk,
topReachState, new FuncPredicate(() =>
playerInTop));//Transition Walk --> Dancing

stateMachine.AddTransition(goToTargetStateRun,
topReachState, new FuncPredicate(() =>
playerInTop));//Transition Run --> Dancing

stateMachine.AddTransition(goToTargetStateWalk,
runAway, new FuncPredicate(() =>
newDirection));//Transition Walk --> RunAway

stateMachine.AddTransition(goToTargetStateRun,
runAway, new FuncPredicate(() =>
newDirection));//Transition Run --> RunAway
        stateMachine.AddTransition(runAway,
goToTargetStateWalk, new FuncPredicate(() =>
!playerInZoneScript.playerInZone));//Transition
RunAway --> Walk
```

```
        stateMachine.AddTransition(runAway,
goToTargetStateRun, new FuncPredicate(() =>
!playerInZoneScript.playerInZone));//Transition
RunAway --> Run



        stateMachine.SetState(idleState);
    }


    // Update is called once per frame
    void Update()
    {
        //Si le joueur est plus dans la zone du
player
        if (!playerInZoneScript.playerInZone)
        {
            newDirection = false;
            finalTarget = tempFinalTarget;
        }

        //Si le mode fuite est activer
        if (newDirection)
        {
            if (runAway.escapePosition != null)
            {
                //La target deviens un point de
fuite
                finalTarget =
runAway.escapePosition;
            }
        }

        goToTargetStateWalk.target =
finalTarget;
        goToTargetStateRun.target = finalTarget;

        stateMachine.Update();
    }
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Mountain"))
        {
            Collider[] colliders =
GetComponents<Collider>();
            foreach (Collider collider in
colliders)
            {
                if
(collider.bounds.Intersects(other.bounds))
                {
                    if (!(collider is
SphereCollider))
                    {
                        inZoneMountain = true;
                    }
                }

            }
        }
        if (other.CompareTag("EnnemyDetect"))
        {
            playerIsNear = true;
        }
        if
(other.CompareTag("targetDestination"))
        {
            Collider[] colliders =
GetComponents<Collider>();
            foreach (Collider collider in
colliders)
            {
                if
(collider.bounds.Intersects(other.bounds))
                {
                    if (!(collider is
SphereCollider))
                    {
                        playerInTop = true;
                    }
                }

            }
        }
    }
    private void OnTriggerExit(Collider other)
    {

        if (other.CompareTag("Mountain"))
        {
            inZoneMountain = false;
        }
        if (other.CompareTag("EnnemyDetect"))
        {
            playerIsNear = false;
        }
    }

    //OnDestroy du NPC
    private void OnDestroy()
    {
        //Notifie ses amis qu'il est mort
        NotifyNearbyEnemies();

        //Indique au spawner qu'il est mort
        if (spawner != null)
        {
            spawner.OnEnnemyDestroyed();
        }

        //VFX explosion de mort
        if (explosionOnHitEnnemy != null)
        {
            Vector3 newPosition =
transform.position + new Vector3(0, 300, 0);

            Instantiate(explosionOnHitEnnemy,
newPosition, Quaternion.identity);
        }
    }
    private void NotifyNearbyEnemies()
    {
        //Radius de détection
        float detectionRadius = 8000f;
        Collider[] colliders =
Physics.OverlapSphere(transform.position,
detectionRadius);

        foreach (Collider collider in colliders)
        {
            if (collider.CompareTag("Ennemy") &&
collider.gameObject != gameObject)
            {

collider.gameObject.SendMessage("ActivateFleeMod
e", true,
SendMessageOptions.DontRequireReceiver);
            }
        }
    }
    public void ActivateFleeMode(bool
shouldFlee)
    {
        if (shouldFlee)
        {
            newDirection = true;
        }
    }
  }
```