



***R5.A.06 – Programmation
Multimédia
Partie Unity – Rapport***

Version 1.0

Année universitaire 2025-2026

Sommaire

Sommaire.....	1
1 - L'idée du jeu.....	2
2 - Le menu du jeu.....	3
3 - La carte du jeu.....	6
4 - Le joueur.....	10
5 - Les coffres.....	18
6 - Les postes d'observations.....	21
7 - Les ennemis.....	25
8 - La Téléportation.....	29
8 - Le switch.....	30
9 - Le petit guide du jeu.....	31
10 - Conclusion.....	33

1 - L'idée du jeu

Pour ce projet, nous avons décidé de partir sur un jeu de labyrinthe. Le principe est simple : le joueur doit traverser un labyrinthe pour rechercher trois clés contenues dans des coffres, nécessaires pour ouvrir la porte de sortie et finir le jeu.

Pour autant, pour éviter que le jeu soit simple et banal, nous avons décidé de rajouter quelques spécificités à notre labyrinthe. Tout d'abord, le joueur ne peut pas accéder à toutes les zones du labyrinthe, l'empêchant ainsi d'accéder à tous les coffres pour rechercher les clés. Pour cela, nous avons décidé que, pour débloquer certains passages, le joueur doit modifier la temporalité : le jour, certains murs sont présents, le soir, d'autres murs s'ajoutent et certains disparaissent, et le soir, de même.

De plus, pour rajouter un peu de challenge, des fantômes parcourent le labyrinthe. Si le joueur se fait toucher par l'un d'entre eux, il retourne au début du labyrinthe, l'obligeant ainsi à tout parcourir de nouveau.

Ainsi d'éviter qu'il ne se perde trop dans le labyrinthe, des postes d'observations sont dissimulés un peu partout dans la carte, lui permettant de visualiser en hauteur les lieux et de se repérer.

Enfin, nous avons choisi comme nom *MazeRunning*, simplement pour rappeler l'esprit du labyrinthe et de la course dans ce dernier.

Ce dernier a été sauvegardé sur le dépôt GitHub suivant :
<https://github.com/VERHILLE-Manon-2326111mv/mazerunning>

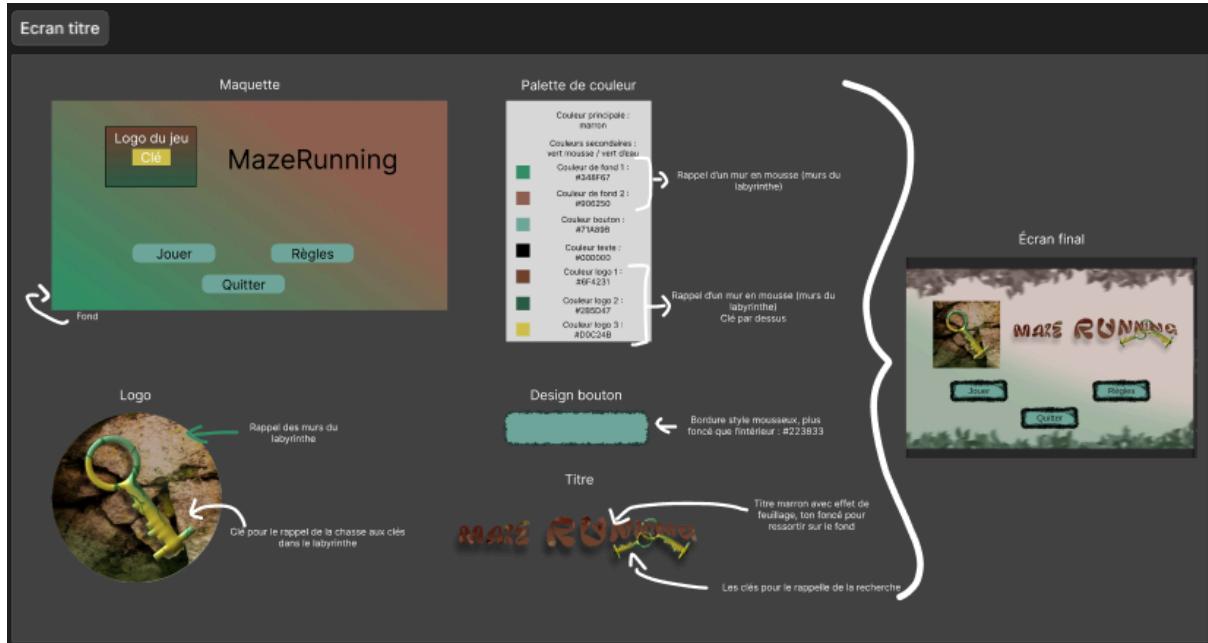
Afin de mener à bien ce projet, nous nous sommes répartis les tâches de la sorte :

- Ronan s'occupe de la création de la carte du jeu, des coffres, de la gestion des clés générées aléatoirement dans trois coffres ceux-ci une fois récupérés permettant de sortir du labyrinthe, l'implémentation du système de portal puis pour finir le système de switch de murs. ;
- Manon s'occupe du menu d'accueil du jeu et de la connexion avec la scène du jeu, aide sur la création de la map du jeu, met en place le personnage du joueur, ses scripts, ses contrôles et ses animations, des ennemis ainsi que des postes d'observations.

2 - Le menu du jeu

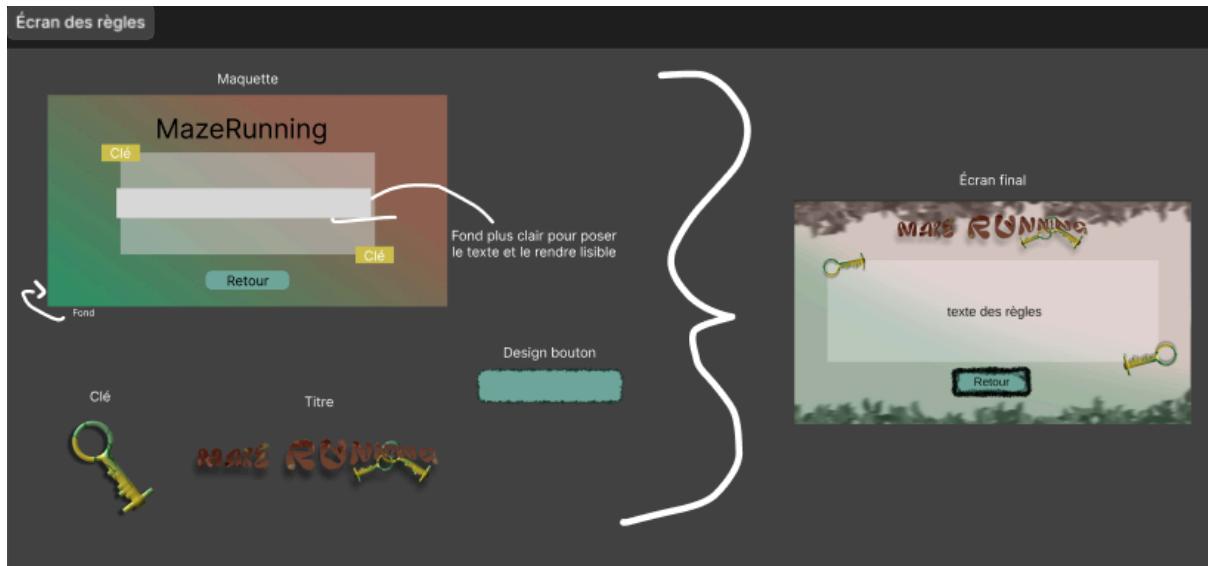
Avant d'élaborer notre jeu, nous avons imaginé le menu lors du lancement du jeu. Sur ce dernier, il est possible de visualiser les règles du jeu, de lancer une partie, ou bien de quitter ce dernier. Pour cela, nous avons réalisé une maquette afin de visualiser un peu plus le menu à implémenter dans Unity.

La première maquette est celle de l'écran-titre :

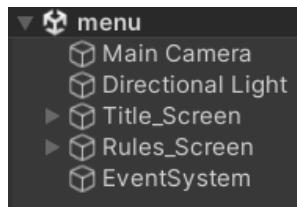


Sur cette dernière est représentée la maquette synthétique de notre menu, les différents éléments à modéliser, dessinés en amont, les codes couleurs à utiliser, ainsi qu'une image de la version finale à créer sur Unity.

De même, nous avons créé une maquette pour l'écran textuel des règles, accessible via le bouton "Règles", respectant les mêmes spécificités que l'écran-titre :



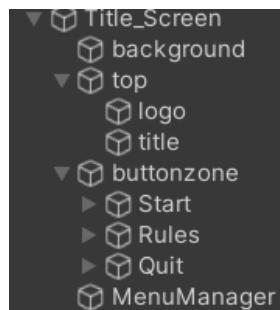
Une fois ces maquettes achevées, nous sommes passés à la création de ce menu sur Unity. Pour cela, nous avons implémenté, sur une branche dédiée sur le dépôt GitHub, ce menu. L'architecture générale est la suivante :



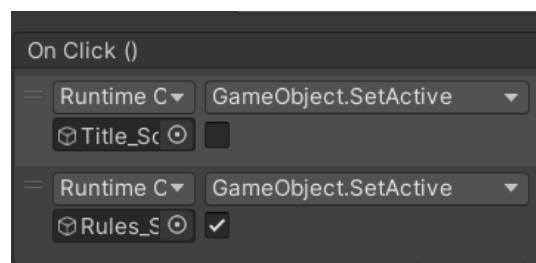
Dans une scène nommée “*menu*”, nous avons placé deux objets vides, **Title_Screen** et **Rules_Screen**, contenant chacun un écran. Avec cela, nous retrouvons une caméra, nécessaire pour visualiser la scène, une lumière directionnelle pour que l'interface soit toujours éclairée, et l'*EventSystem* qui permet le lien entre l'interface et les périphériques de l'utilisateur.

Dans le **Title_Screen**, nous retrouvons nos divers éléments présents dans la maquette, répartis dans les objets vides, ainsi que le fond d'écran nommé **background**. Dans l'objet **Top**, nous avons le titre et le logo du jeu, dans l'objet **buttonzone**, les boutons **Start**, **Rules** et **Quit**, et l'objet vide **MenuManager**, ayant chacun une action différente :

- **Start** permet, lorsque l'utilisateur clique dessus, de fermer le menu du jeu et d'ouvrir celui possédant la map du jeu ;
- **Rules** permet de mettre en invisible l'objet **Title_Screen** et ses enfants, et d'afficher à la place l'objet **Rules_Screen** ;
- **Quit** permet de fermer la fenêtre, et donc de fermer le jeu ;
- **MenuManager** est un objet lié à un script permettant l'action de certains boutons.



Seul le bouton **Rules** n'utilise pas de script. Ces différentes interactions sont mises en place directement dans l'Inspector du bouton, dans lequel il est possible de déclarer directement un ou plusieurs comportements simples sans utiliser de code.



Pour les boutons **Start** et **Rules**, un script **MainMenuManager.cs** a été créé. Ce dernier contient deux fonctions permettant d'attribuer un comportement à chaque bouton.

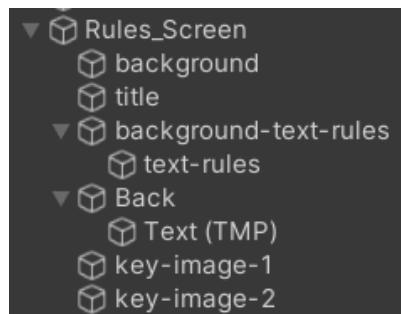
```
public class MainManager : MonoBehaviour
{
    [SerializeField] private string gameSceneName; // Nom de la scène principale à
    charger lors du démarrage du jeu

    /// <summary>
    /// Lance le jeu en chargeant la scène principale.
    /// </summary>
    public void StartGame()
    {
        Debug.Log("Start Game");
        // Charge la scène du jeu principal
        UnityEngine.SceneManagement.SceneManager.LoadScene(gameSceneName);
    }

    /// <summary>
    /// Quitte l'application.
    /// </summary>
    public void QuitGame()
    {
        Debug.Log("Quit Game");
        // Ferme l'application
        Application.Quit();
    }
}
```

Ici, la fonction **StartGame** permet ainsi d'ouvrir la scène du jeu. La fonction **QuitGame** ferme, de son côté, l'application.

Dans le **Rules_Screen**, nous retrouvons de nouveau le titre du jeu, un bouton **Return** permettant de revenir sur le menu principal, et un champ de texte sur lequel est exposé les règles du jeu. Le bouton est paramétré comme le bouton **Rules** de **Title_Screen** : dans l'inspector, au clic du bouton, **Rules_Screen** est mis en invisible et **Title_Screen** est affiché.



3 - La carte du jeu

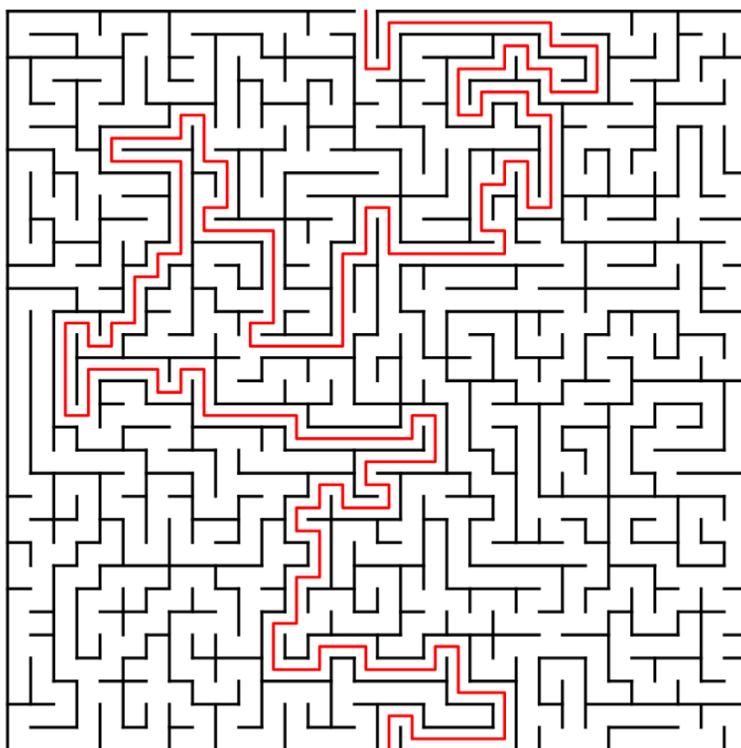
Afin de pouvoir jouer, il a fallu créer la carte “map” du jeu. Pour ce faire nous avons décidé de faire un choix entre trouver une map déjà faite ou la créer de toutes pièces. Nous avons donc choisi la deuxième option pour plusieurs raisons.

Premièrement on souhaitait avoir la maîtrise totale de la carte car nous savions que nous serions souvent amenés à la modifier et l'étudier. Il était donc impératif de créer de toute pièce pour s'assurer de son bon fonctionnement. Ensuite il y a la contrainte du jeu, en choisissant un jeu de labyrinthe nous savons qu'il serait impossible de trouver gratuitement une carte répondant à nos attentes en termes de design, difficultés... Le dernier élément ayant fait pencher la balance sur une map créée main est l'apprentissage. Bien que cela revienne à prendre un énorme temps de travail pour avoir un résultat satisfaisant, celui-ci nous apporte énormément de compétences et de connaissances dans la création de map.

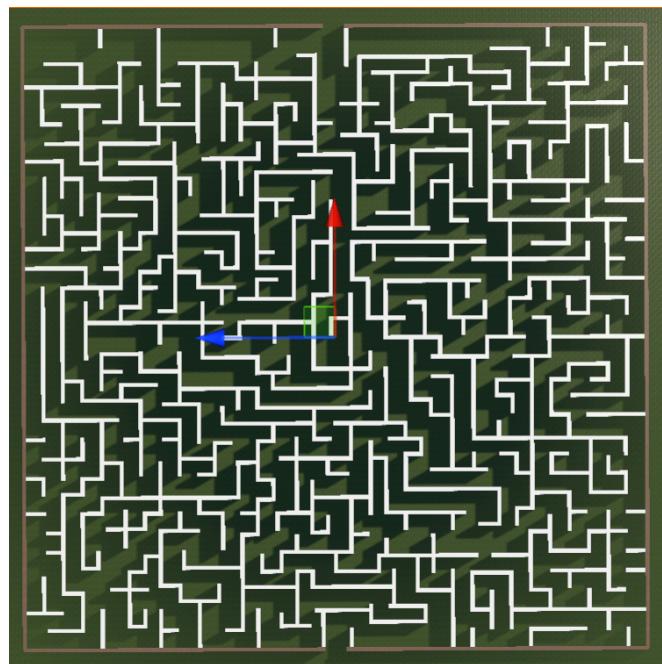
Processus de Création :

Nous allons désormais vous présenter le processus de création. Celui-ci a été fait en 4 grandes étapes.

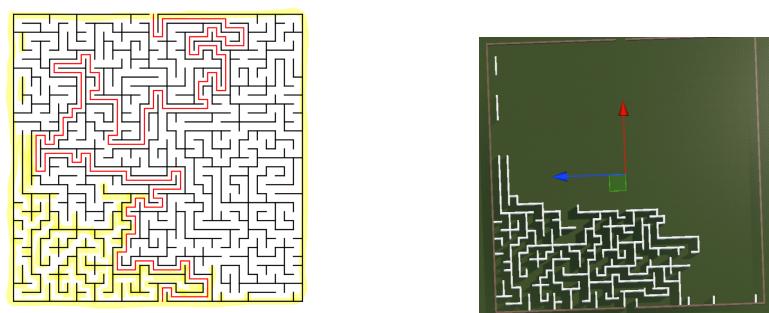
1. Le design du Labyrinthe. Lors de cette étape nous sommes allées sur ce site : <https://www.mazegenerator.net/> qui permet de générer un labyrinthe de taille définie par nous et de chemins aléatoires. C'est la meilleure façon de prendre un labyrinthe ayant un chemin unique mais faisable. Le faire soi-même pourrait donner des cartes trop facile ou bien impossible. Nous avons générés plusieurs prototypes jusqu'à choisir celui qui nous semblait le mieux. Une fois cette étape nous avons obtenu cette image ici présente. Le chemin en rouge indique la solution.



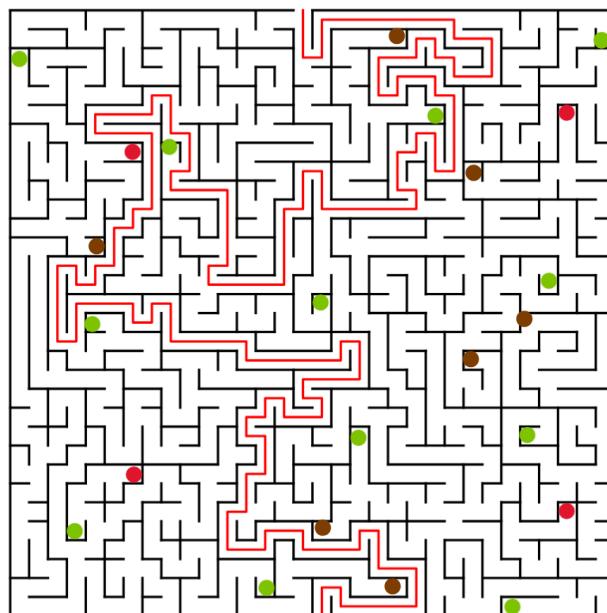
2. Réalisation de la carte. Ensuite il a fallu implémenter en 3d cette carte. Pour cela nous avons d'abord créé une plaine, vide avec des bordures faites avec des arbres. Cette partie ne sera presque pas visible par le joueur mais ça permet d'avoir une grande zone de sol pour y mettre le labyrinthe. Ensuite nous avons commencé à créer le labyrinthe. Il a d'abord fallu faire une échelle vis-à-vis du modèle. Celui-ci est donné comme 1 mur sur le dessin vaut une taille de 5. Ensuite nous avons donc passé un moment intense et long à placer chaque mur comme sur le modèle via des cubes, à chaque fois multiple de 5 avec des coordonnées ayant soit un nombre sans décimale soit avec une décimale à X,5. Il y a eu malgré les précautions plusieurs difficultés, parfois un décalage se crée quand on place des murs, un décalage de 1 selon si on commence le mur dans le mur ou à son extrémité. Nous avons donc fait quelques ajustements pour que même si certains espaces sont plus serrés que d'autres les chemins du labyrinthe sont parfaitement identiques au modèle. Lors de ce moment présent, la carte ressemble à cela.



Pour en arriver à ce résultat nous avons utilisé une technique, nous surlignons les murs placés au fur et à mesure pour exemple :



3. L'étape suivante est le placement des items, coffres, ennemis, point d'observation. Ces objets sont détaillés plus loin dans le rapport mais voici le processus d'intégration. Nous nous sommes posé la question de où les placer pour garantir un meilleur gameplay, puis avons tracé sur le modèle les lieux où nous souhaitions poser les items. Une fois cela fait, il ne reste plus qu'à les positionner exactement aux endroits souhaités. Le choix du placement est d'obliger le joueur à parcourir le plus possible de chemin. À la fin le modèle ressemble à cela :



Les points marrons sont les coffres, les verts ce sont les points d'observation et les rouges les 4 ennemis.

4. La dernière étape fut les finitions. Une fois le jeu fait, nous voulions apporter un petit plus à la map. Nous avons ajouté un début, une zone vide et une arrivée. L'arrivée est une petite salle remplie de fromage pouvant être mangé, correspondant à un lancement d'un petit bruit. Cette fin fait référence à notre personnage étant une souris et aux récompenses que mettent certains chercheurs après avoir mis une souris dans un labyrinthe. Ensuite nous avons rajouté un petit relief à la porte de sortie pour mettre en évidence la sortie. La dernière modification afin d'apporter un jeu plus agréable est l'ajout d'un "material" sur les murs pour rendre le jeu plus attractif. Cependant cette modification a posé une erreur. En effet, le material était très étiré sur les murs trop grands. En changeant le tiling qui permet de répéter le schéma c'était les petits murs qui devenaient incohérents. C'est pour cela que nous

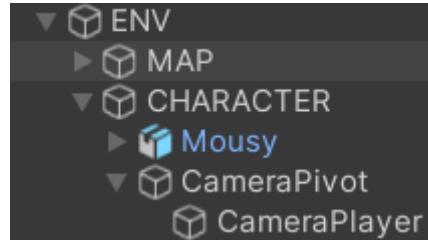
avons créé un script AutoTilling permettant au lancement du jeu de mettre le material correctement sur chaque mur. Voici des photos des modifications :



4 - Le joueur

Une fois la carte du jeu créée, nous sommes passés à la mise en place du joueur. Pour cela, nous avons réutilisé le modèle “*Mousy*” et ses animations proposés dans le cours.

Au niveau de l’architecture dans Unity, un objet vide **CHARACTER** a été créé au même niveau que l’objet vide **LABYRINTH**. Dans ce dernier se trouve l’asset de *Mousy*, ainsi qu’un objet vide **CameraPivot** contenant la caméra du joueur, nécessaire pour visualiser la scène pendant le contrôle du personnage.



Le personnage *Mousy* est constitué d’un Rigidbody, lui permettant de ne pas tomber lors de ses déplacements ou lors d’une collision, d’une Capsule Collider nécessaire pour la détection de collision, d’un Animator, permettant de relier l’avatar à un controller proposé dans le cours, et de son script qui sera détaillé plus tard.



Par la suite, nous avons ajouté un fichier permettant de relier des boutons spécifiques du clavier à des actions spécifiques du personnage, nommé **PlayerControls.inputactions**. Dedans est inscrit une action **Move**, défini comme un **Vector2**, définissant les boutons ZQSD pour chaque mouvement du personnage :

- Z pour avancer ;
- S pour reculer ;
- Q pour tourner à gauche ;
- D pour tourner à droite.

Ce fichier avec l’extension **.inputactions**, une fois sauvegardée, crée alors le fichier **PlayerControls.cs**, permettant d’avoir une passerelle entre les données de **PlayerControls.inputactions**, c’est-à-dire les touches configurées, et les scripts C#, qui permettront de mettre en place l’action lors de l’utilisation d’une touche.

Un script **PlayerScript.cs** est alors créé afin de gérer tous les inputs utilisateurs et son animation. Dans un premier temps, nous avons déclaré toutes les variables nécessaires à l'initialisation et au fonctionnement du personnage.

```
public class PlayerScript : MonoBehaviour
{
    [Header("Paramètre des mouvements du joueur")]
    [SerializeField] float moveSpeed = 2f; // Vitesse de déplacement du joueur
    [SerializeField] float rotationSpeed = 120f; // Vitesse de rotation du joueur
    [SerializeField] Transform playerStartPoint; // Point de départ du joueur

    private PlayerControls controls; // Référence au système d'input personnalisé
    private Vector2 moveInputs; // Stocke l'entrée de déplacement
    private Rigidbody rb; // Référence au Rigidbody du joueur
    public bool canMove = true; // Indique si le joueur peut se déplacer
    Animator animator; // Référence à l'Animator du joueur

    [...]
}
```

Par la suite, nous avons mis en place les différentes fonctions permettant le bon fonctionnement du joueur comme suit :

```
public class PlayerScript : MonoBehaviour
{
    [...]

    /// <summary>
    /// Initialise les contrôles et configure les callbacks d'entrée.
    /// </summary>
    void Awake()
    {
        controls = new PlayerControls();
        animator = GetComponent<Animator>();

        // Callback appelé quand le joueur bouge (input maintenu)
        controls.Player.Move.performed += ctx => {
            moveInput = ctx.ReadValue<Vector2>();
        };
    }
}
```

```
// Callback appelé quand le joueur arrête de bouger (input relaché)
controls.Player.Move.canceled += ctx => moveInput = Vector2.zero;
}

/// <summary>
/// Active les contrôles du joueur.
/// </summary>
void OnEnable()
{
    if (controls != null) controls.Player.Enable();
}

/// <summary>
/// Désactive les contrôles du joueur.
/// </summary>
void OnDisable()
{
    if (controls != null) controls.Player.Disable();
}

/// <summary>
/// Initialisation du Rigidbody et configuration des contraintes physiques.
/// </summary>
void Start()
{
    rb = GetComponent<Rigidbody>();
    if (rb == null)
        Debug.LogError("Le Player doit avoir un Rigidbody !");
    // Empêche le joueur de basculer sur les axes X et Z
    rb.constraints = RigidbodyConstraints.FreezeRotationX |
    RigidbodyConstraints.FreezeRotationZ;
    // Améliore la fluidité des mouvements physiques
    rb.interpolation = RigidbodyInterpolation.Interpolate;

    // Vérifie que le point de départ est assigné
    if (playerStartPoint == null) Debug.LogWarning("ATTENTION : Le point de départ
(playerStartPoint) n'est pas assigné !");

    // Place le joueur au point de départ
    transform.position = playerStartPoint.position;
}
```

```
/// <summary>
/// Appelé à chaque frame physique, gère le déplacement et la rotation du joueur.
/// </summary>
void FixedUpdate()
{
    // Si le joueur ne peut pas se déplacer, arrête tout mouvement
    if (!canMove)
    {
        animator.SetBool("Move", false);
        rb.velocity = Vector3.zero;
        return;
    }

    MoveAndRotate();
}

/// <summary>
/// Déplace et fait tourner le joueur selon les entrées utilisateur.
/// </summary>
private void MoveAndRotate()
{
    // Vérifie si le joueur se déplace réellement
    bool isMoving = moveInput.magnitude > 0.1f;
    animator.SetBool("Move", isMoving);

    // Déplacement avant/arrière (axe Y de moveInput)
    Vector3 move = transform.forward * moveInput.y * moveSpeed *
Time.fixedDeltaTime;
    rb.MovePosition(rb.position + move);

    // Rotation gauche/droite (axe X de moveInput)
    Quaternion rotation = Quaternion.Euler(0f, moveInput.x * rotationSpeed *
Time.fixedDeltaTime, 0f);
    rb.MoveRotation(rb.rotation * rotation);
}

/// <summary>
/// Empêche ou autorise le déplacement du joueur
/// </summary>
public void SetCanMove(bool status)
{
    canMove = status;
```

```

        if (!status)
        {
            moveInput = Vector2.zero;
            animator.SetBool("Move", false);
            rb.velocity = Vector3.zero;
        }
    }
}

```

Cependant, bien que le personnage puisse maintenant bouger et tourner sans souci, la caméra ne suit pas ce dernier, restant à un point statique sur la carte. Pour cela, un autre script a été créé, **CameraScript.cs**, ce dernier étant relié à la caméra CameraPlayer, ainsi qu'un autre contrôleur **CameraControls.inputactions**, ayant une action **Look** liée au clic gauche de la souris pour déplacer la caméra, et au bouton Tab du clavier pour changer de point de vu (passer de la vue à la première personne à la vue à la troisième personne et ainsi de suite). Le code de la caméra est le suivant :

```

public class CameraScript: MonoBehaviour
{
    [Header("Référence de l'objet à suivre")]
    [SerializeField] private Transform body;

    [Header("Paramètres de la camera")]
    [SerializeField] private float sensitivity = 100f;           // Sensibilité de la rotation
de la caméra
    [SerializeField] private float clampAngle = 80f;             // Angle maximum de rotation
verticale
    [SerializeField] private float distanceFromObject = 3f;     // Distance entre la caméra et
l'objet suivi
    [SerializeField] private float height = 1.6f;                // Hauteur de la caméra par
rapport à l'objet suivi

    private CameraControls controls;                            // Système de contrôles de la
caméra
    private Vector2 lookInput;                                // Valeur d'entrée de la
souris
    private float xRotation = 0f;                            // Rotation verticale actuelle
    private float yRotation = 0f;                            // Rotation horizontale
actuelle
    private bool isFirstPerson = false;                      // Indique si la camera est en
vue à la première personne (pour un joueur)

    /// <summary>

```

```
/// Initialise les contrôles de la caméra et configure les callbacks d'entrée.  
/// </summary>  
private void Awake()  
{  
    controls = new CameraControls();  
    // Configure la détection du mouvement de la souris  
    controls.Camera.Look.performed += ctx => lookInput = ctx.ReadValue<Vector2>();  
    controls.Camera.Look.canceled += _ => lookInput = Vector2.zero;  
}  
  
// Active les contrôles de la caméra  
private void OnEnable()  
{  
    if (controls != null)  
        controls.Camera.Enable();  
}  
  
// Désactive les contrôles de la caméra  
private void OnDisable()  
{  
    if (controls != null)  
        controls.Camera.Disable();  
}  
  
/// <summary>  
/// Initialise la rotation de la caméra en fonction de sa position de départ par rapport au joueur.  
/// </summary>  
private void Start()  
{  
    Vector3 offset = transform.position - body.position;  
    // Calcule les angles de rotation initiaux à partir de la position relative  
    yRotation = Mathf.Atan2(offset.x, offset.z) * Mathf.Rad2Deg;  
    xRotation = Mathf.Asin(offset.y / offset.magnitude) * Mathf.Rad2Deg;  
}  
  
/// <summary>  
/// Met à jour la position et la rotation de la caméra à chaque frame après les mises à jour du joueur.  
/// </summary>  
private void LateUpdate()  
{  
    if (body == null) return;
```

```
if (controls.Camera.ToggleView.triggered && body.CompareTag("Player"))  
changeView();  
  
// Gère la rotation de la caméra uniquement lorsque le bouton gauche de la souris  
est maintenu  
else if (Mouse.current.leftButton.isPressed)  
{  
    // Calcule les rotations en fonction du mouvement de la souris  
    float mouseX = lookInput.x * sensitivity * Time.deltaTime;  
    float mouseY = lookInput.y * sensitivity * Time.deltaTime;  
  
    // Met à jour les angles de rotation  
    yRotation += mouseX;  
    xRotation -= mouseY;  
    // Limite la rotation verticale pour éviter le retournement de la caméra  
    xRotation = Mathf.Clamp(xRotation, -clampAngle, clampAngle);  
}  
  
// Calcule la nouvelle position de la caméra selon le mode de vue  
Quaternion rotation = Quaternion.Euler(xRotation, yRotation, 0);  
if (isFirstPerson)  
{  
    // En vue première personne, la caméra est directement é la position du joueur  
    transform.position = body.position + Vector3.up * height;  
    transform.rotation = rotation;  
}  
else  
{  
    // En vue troisième personne, garde le comportement existant  
    Vector3 offset = rotation * new Vector3(0, 0, -distanceFromObject);  
    transform.position = body.position + Vector3.up * height + offset;  
    transform.LookAt(body.position + Vector3.up * height);  
}  
}  
  
/// <summary>  
/// Passe de la vue de la première personne é la vue é la troisième personne et vice  
versa si la caméra est celle du joueur.  
/// </summary>  
private void changeView()  
{  
    isFirstPerson = !isFirstPerson;
```

```
if (isFirstPerson)
{
    distanceFromObject = 0f;
    height = 1f;
}
else
{
    distanceFromObject = 3f;
    height = 1.6f;
}

}
```

Ainsi, via ce code, la caméra suit obligatoirement le joueur, peut être bougé via le clic gauche de la souris et changer de vue via le bouton TAB.

5 - Les coffres

La prochaine étape est l'implémentation des coffres. L'idée est d'offrir une meilleure expérience de jeu en ajoutant une règle fondamentale, la sortie devient une porte fermée. Le but est donc non seulement de trouver une sortie mais de parcourir le labyrinthe pour fouiller des coffres. Dans certains de ces coffres se cachent des clés pouvant ouvrir la porte, il en faut un total de 3 afin que la porte s'ouvre.

Nous avons donc caché différents coffres dans la map. Le coffre est un prefab téléchargé depuis internet, celui-ci est donc placé dans le labyrinthe, avec pour chaque coffre le script "Chest.cs". Ce script a pour but de gérer les coffres. Il fonctionne de pair avec le script "ChestInteraction.cs" rattaché au joueur.



Initialisation des coffres :

Au lancement de la partie, nous avons créé un système de positionnement de clés de manière aléatoire. Via le script RandomKeyManager 3 clés se positionnent aléatoirement parmi les coffres présents sur la carte.

```
0 references | Unity Message
void Start()
{
    Chest[] foundChests = FindObjectsOfType<Chest>();
    allChests.AddRange(foundChests);
    DistributeKeys();
}
```

```
1 reference
void DistributeKeys()
{
    if (allChests.Count < keysRequired) return;

    List<Chest> chestPool = new List<Chest>(allChests);

    for (int i = 0; i < keysRequired; i++)
    {
        int randomIndex = Random.Range(0, chestPool.Count);
        chestPool[randomIndex].SetHasKey(true);
        chestPool.RemoveAt(randomIndex);
    }
}
```

Ce script permet aussi de gérer la gestion des clés avec notamment une méthode qui ajoute à l'inventaire du joueur les clés. C'est justement cette méthode qui est appelée lors de la récupération d'une clé. En effet dans le script Chest, il y a une méthode qui permet, si une clé est présente dans le coffre, de l'ajouter à l'inventaire du joueur.

```
1 reference
public void RetrieveItem()
{
    if (isRecupere) return;
    isRecupere = true;
    if (hasKey)
    {
        Debug.Log("BRAVO : Vous avez trouvé une clé !");
        RandomKeyManager.Instance.AddKey();
    }
    else
    {
        Debug.Log("Ce coffre est malheureusement vide...");
    }
}
```

Ceci est donc le fonctionnement d'un coffre, cependant pour ouvrir le coffre il faut donc créer une interaction avec le joueur. Il nous faut donc deux scripts importants, ChestInteraction pour créer une détection lorsqu'un joueur s'approche d'un coffre.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

private void DetectChest()
{
    Collider[] hits = Physics.OverlapSphere(transform.position, interactionRange);

    chestInRange = null;
    foreach (var hit in hits)
    {
        Chest chest = hit.GetComponent<Chest>();
        if (chest != null)
        {
            chestInRange = chest;
            break;
        }
    }
}
```

Mais il nous faut aussi et surtout une interface homme machine UiChest pour permettre l'interaction.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public void ToggleUi(bool show, Chest chest)
{
    currentChest = chest;
    chestPanel.SetActive(show);
    if (show && currentChest != null)
    {
        if (currentChest.IsEmpty())
        {
            slotImage.enabled = false;
            slotButton.interactable = false;
        }
        else if(currentChest.getHasKey())
        {
            slotImage.sprite = currentChest.itemSprite;
            slotImage.enabled = true;
            slotButton.interactable = true;
        }
        else
        {
            slotImage.enabled = false;
            slotButton.interactable = false;
        }
    }
}
```

C'est donc de cette manière que nous avons pu créer un système de coffre simple permettant non plus de simplement sortir du jeu mais aussi de parcourir la carte dans le but de fouiller des coffres, interagir et récupérer 3 clés pour gagner la partie. De plus le système de remplissage des coffres aléatoires nous permet de pouvoir lancer plusieurs fois la partie même si le chemin vers la porte est le même.

6 - Les postes d'observations

Comme cité précédemment, afin d'éviter au joueur de se perdre, nous avons décidé de mettre en place des zones où il est possible de visualiser une partie de la carte, nommée chez nous des postes d'observation.

Pour cela, nous avons créé un prefab “*ObservationPoint*” constitué d'un plane et d'une caméra. Ce plane est une zone de détection. Lorsque le joueur est sur ce dernier, il a la possibilité de changer de vue en appuyant sur un bouton, sa caméra devenant celle du poste. Ce plane possède un Box Collider, nécessaire pour capter la collision avec le joueur, ainsi qu'un script, **ObservationScript.cs**. Dans ce dernier, nous définissons la caméra du joueur ainsi que la caméra du poste, en tant que *SerializeField*, ainsi qu'une touche d'interaction, ici la touche E. À cela est ajouté trois variables privées : deux booléens pour gérer l'état des deux caméras, et une référence au script du joueur.

```
public class ObservationScript : MonoBehaviour
{
    [Header("Références des Caméras")]
    [SerializeField] private GameObject playerCamera;
    [SerializeField] private GameObject observationCamera;

    [Header("Touche d'interaction avec la zone d'observation")]
    [SerializeField] private KeyCode interactionKey = KeyCode.E;

    // Etats des caméras
    private bool isPlayerOnPoint = false;
    private bool isObserving = false;
    private PlayerScript playerScriptReference;

    [...]
}
```

Dès l'initialisation, via la fonction **Start()**, nous nous assurons d'un état cohérent : la caméra du joueur est activée par défaut, tandis que celle du poste d'observation est désactivée pour éviter tout conflit visuel.

Pour gérer l'autorisation d'interagir, nous utilisons les événements physiques d'Unity. Lorsqu'un objet entre dans la zone avec **OnTriggerEnter()**, nous vérifions via son **Layer** s'il s'agit bien du “*Player*”, ici notre **Character** et son asset *Mousy*. Si c'est le cas, nous stockons une référence à son script *PlayerScript* (nécessaire pour bloquer ses mouvements plus tard) et autorisons l'interaction.

À l'inverse, lorsque le joueur quitte la zone, avec **OnTriggerExit()**, une sécurité est mise en place : si le joueur était encore en train d'observer la carte, nous forçons le retour à la vue

normale pour éviter qu'il ne reste bloqué sur la caméra aérienne tout en contrôlant son personnage au sol.

Enfin, la logique principale réside dans la méthode ***ToggleView()***, appelée lorsque le joueur appuie sur **E**. Cette méthode effectue trois actions simultanées :

1. Elle inverse l'état d'observation avec le booléen ***isObserving***.
2. Elle intervertit l'activation des deux caméras via ***SetActive()***.
3. Elle appelle la méthode publique ***SetCanMove()*** du joueur pour geler ou dégeler ses déplacements. Ce point est essentiel pour l'expérience utilisateur, empêchant le joueur de tomber dans un piège pendant qu'il observe la carte.

```
public class ObservationScript : MonoBehaviour
{
    [...]
    /// <summary>
    /// Initialisation de l'état des caméras au démarrage.
    /// </summary>
    private void Start()
    {
        // On s'assure que la vue commence sur le joueur
        playerCamera.SetActive(true);
        observationCamera.SetActive(false);
    }

    /// <summary>
    /// Vérifie l'appui sur la touche d'interaction à chaque frame.
    /// </summary>
    private void Update()
    {
        // Si le joueur est dans la zone, que le script est valide et qu'il appuie sur
        la touche
        if (isPlayerOnPoint && playerScriptReference != null &&
Input.GetKeyDown(interactionKey))
        {
            ToggleView();
        }
    }

    /// <summary>
    /// Déetecte l'entrée du joueur dans la zone d'observation.
    /// </summary>
    /// <param name="other">Le collider qui entre dans la zone.</param>
```

```

private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerOnPoint = true;
        Debug.Log("Le joueur est entré dans la zone.");

        // Récupération du script pour pouvoir geler les mouvements plus tard
        playerScriptReference = other.GetComponent<PlayerScript>();
    }
}

/// <summary>
/// Déetecte la sortie du joueur de la zone et réinitialise la vue si nécessaire.
/// </summary>
/// <param name="other">Le collider qui sort de la zone.</param>
private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerOnPoint = false;
        Debug.Log("Le joueur a quitté la zone/";

        // Sécurité : Si le joueur part en laissant la caméra active, on coupe tout
        if (isObserving)
        {
            ToggleView();
        }

        playerScriptReference = null;
    }
}

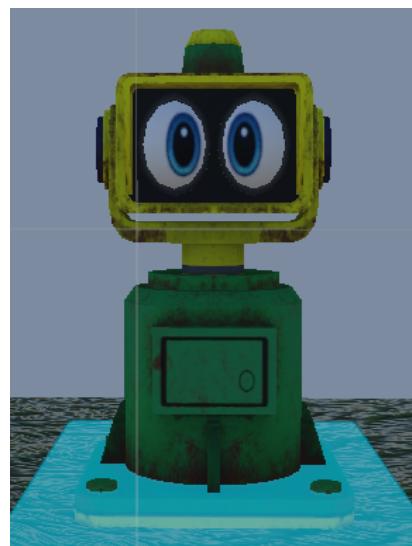
/// <summary>
/// Inverse l'état d'observation, change les caméras actives et gère le blocage du
joueur.
/// </summary>
private void ToggleView()
{
    // Inverse l'état
    isObserving = !isObserving;

    // Active/Désactive les caméras en fonction du nouvel état
}

```

```
playerCamera.SetActive(!isObserving);  
observationCamera.SetActive(isObserving);  
  
// Bloque ou débloque les mouvements du joueur via son script  
if (playerScriptReference != null)  
{  
    playerScriptReference.SetCanMove(!isObserving);  
}  
}  
}
```

Une fois la logique du point d'observation appliquée via ce script, le plane a été mis en invisible en décochant le Mesh, et un asset a été ajouté pour plus de visuel dans le labyrinthe. Le poste d'observation final est le suivant :

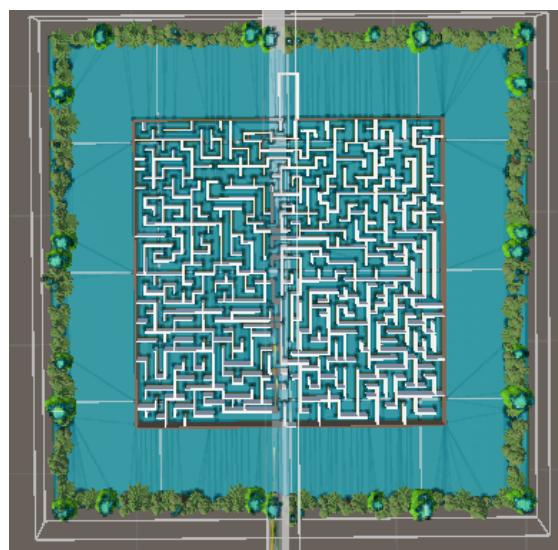


7 - Les ennemis

L'objectif de cette partie était d'intégrer des entités hostiles capables de patrouiller de manière autonome dans le labyrinthe. Ces bots devaient répondre à deux contraintes majeures : naviguer fluidement sans se bloquer dans les murs, et détecter une collision avec le joueur pour déclencher une téléportation forcée au début de la map. Pour cela, nous avons créé un nouveau prefab “*Ennemy*” contenant un cube comme objet, ce dernier supprimé à la fin dans le jeu (et non pas dans le prefab) pour être remplacé par les assets.

Dans un premier temps, nous avons tenté une approche basée sur des lancers de rayons nommés **Raycasts** sur Unity. L'ennemi avançait tout droit et changeait de direction aléatoirement lorsqu'un "laser" détectait un mur. Cependant, cette méthode s'est révélée inefficace pour un labyrinthe complexe : les ennemis restaient souvent coincés dans des boucles ou des culs-de-sac, et mettre des déplacements aléatoires sur des intervalles de temps précises ou de rajouter des Raycasts pour forcer un déplacement si aucun mur n'est détecté ne faisait que bloquer davantage l'ennemi.

Suite à ces limitations, et avec l'assistance d'une IA générative (Gemini) pour le débogage et l'optimisation de l'architecture, nous avons migré vers le système de navigation natif d'Unity : le **NavMesh** (Navigation Mesh). Cette approche transforme le problème : au lieu de simplement "fuir les murs", l'ennemi calcule désormais un chemin valide sur une surface navigable. Le sol et les murs du labyrinthe ont été marqués comme statiques. Un composant **NavMeshSurface** calcule ensuite les zones navigables (affichées en bleu ci-dessous), créant une carte vectorielle que les agents peuvent comprendre. Nous avons ajusté le rayon de l'agent, nommé **Radius** dans les paramètres de ce dernier, à 0.8 pour correspondre à la taille de nos ennemis et éviter qu'ils ne traversent visuellement les murs dans les virages.



Le script **EnnemyScript.cs**, attaché à l'ennemi, utilise le composant **NavMeshAgent**. L'algorithme de patrouille définit une destination aléatoire valide sur la carte. Une fois cette destination atteinte, un nouveau point est calculé.

```
public class EnnemyScript : MonoBehaviour
{
    [Header("Réglages Navigation")]
    [Tooltip("Rayon de recherche pour la destination (Mets 50 ou 100 pour couvrir tout le labyrinthe)")]
    [SerializeField] float range = 50f; // Rayon de recherche pour la destination aléatoire
    [Header("Réglages Interaction")]
    [SerializeField] Transform playerStartPoint; // Point de réapparition du joueur après collision

    private NavMeshAgent agent; // Référence au composant de navigation

    /// <summary>
    /// Initialisation de l'agent et configuration physique.
    /// </summary>
    void Start()
    {
        agent = GetComponent<NavMeshAgent>();

        // On s'assure que la physique ne perturbe pas le NavMesh
        Rigidbody rb = GetComponent<Rigidbody>();
        if (rb != null) rb.isKinematic = true;

        // On lance la première destination dès le début
        SetNewRandomDestination();
    }

    /// <summary>
    /// Vérifie à chaque frame si l'agent a atteint sa destination.
    /// </summary>
    void Update()
    {
        // On vérifie si l'agent a fini de calculer ET s'il est arrivé
        if (!agent.pathPending && agent.remainingDistance <= agent.stoppingDistance)
        {
            // Il est arrivé, on lui donne une nouvelle destination
            SetNewRandomDestination();
        }
    }
}
```

```
/// <summary>
/// Trouve un point valide sur la carte bleue (NavMesh) et y va.
/// </summary>
void SetNewRandomDestination()
{
    // On cherche un point au hasard dans une sphère autour de l'ennemi
    Vector3 randomPoint = transform.position + Random.insideUnitSphere * range;

    NavMeshHit hit;

    // On vérifie si ce point tombe bien sur la zone bleue (Walkable)
    // SamplePosition cherche le point le plus proche sur le NavMesh dans un rayon
    de 10 unités
    if (NavMesh.SamplePosition(randomPoint, out hit, 10.0f, NavMesh.AllAreas))
    {
        agent.SetDestination(hit.position);
    }
}

/// <summary>
/// Gestion de la collision avec le joueur.
/// </summary>
private void OnCollisionEnter(Collision collision)
{
    // Si on tape le joueur
    if (collision.gameObject.layer == LayerMask.NameToLayer("Player"))
    {
        Debug.Log("Joueur touché !");

        // Téléportation au point de départ
        if (playerStartPoint != null)
            collision.transform.position = playerStartPoint.position;

        // Réinitialisation de la physique du joueur (pour stopper l'inertie)
        Rigidbody playerRb = collision.gameObject.GetComponent<Rigidbody>();
        if (playerRb != null)
        {
            playerRb.velocity = Vector3.zero;
            playerRb.angularVelocity = Vector3.zero;
        }
    }
}
```

L'intégration de l'asset a soulevé un défi technique concernant la hiérarchie Unity. L'asset importé possédait son propre Rigidbody, ce qui créait un conflit avec le NavMeshAgent du parent, notre objet vide **Ennemy** : le parent guidait le mouvement, mais l'asset, l'enfant de **Ennemy**, tombait ou se désynchronisait à cause de la gravité.

Pour résoudre cela, nous avons mis en place la structure suivante :

- Le **Parent** porte le Script, le NavMeshAgent et un Rigidbody configuré en **Is Kinematic**. Il agit comme le "cerveau" et le corps physique, recevant les collisions mais n'étant pas affecté par la gravité.
- L'**Enfant**, l'asset **LittleGhost_LP**, ne porte que le CapsuleCollider (la forme) et l'Animator (le visuel), sans Rigidbody.



Cette séparation permet une navigation fluide gérée par le NavMesh, tout en garantissant que les collisions avec le joueur soient correctement remontées au script principal pour déclencher la défaite.

Pour cette partie spécifique, l'assistant Gemini a été utilisé comme partenaire technique . Il a permis d'identifier rapidement les limitations du système par Raycast initial et a aidé à la transition vers le NavMesh, notamment en résolvant les conflits de hiérarchie physique entre le parent, notre objet vide et le modèle 3D importé.

8 - La Téléportation

Il était important de pouvoir ajouter de nouvelles possibilités de jeu notamment avec un système de téléportation. Cet ajout va permettre de voir le jeu différemment avec la possibilité de trouver encore plus de plaisir à jouer. Le principe est simple, le joueur part équipé d'un portail sur lui. À n'importe quel moment et s'il le souhaite il peut poser le portail représenté par un rond bleu au sol, cette action est irréversible : le lieu du portail reste inchangé durant le restant de la partie. Pour effectuer cette action il lui suffit d'appuyer sur la touche "P". Ensuite, à n'importe quel moment il peut réappuyer sur le même bouton pour se téléporter instantanément vers le portail.

Ceci amène donc à de nouvelles stratégies, la question permanente du lieu le plus intéressant afin de poser le portail, ou bien de la fuite des monstres.

Le code du portail est relativement simple, nous avons un script [PortalControl.cs](#) qui gère complètement l'utilisation du portail. Il sait détecter si c'est la première fois que l'on appuie sur P et par conséquent que l'on pose le portail exactement sur nos pas.

```
1 reference
void SpawnPortal()
{
    RaycastHit hit;
    Vector3 spawnPos = transform.position;

    if (Physics.Raycast(transform.position + Vector3.up, Vector3.down, out hit, 5f))
    {
        spawnPos = hit.point + new Vector3(0, 0.02f, 0);
    }
    else
    {
        spawnPos = new Vector3(transform.position.x, 0.02f, transform.position.z);
    }

    currentPortal = Instantiate(portal, spawnPos, Quaternion.identity);
    Debug.Log("Portail posé !");
}
```

Puis si ce n'est pas la première fois alors il ne faut pas poser un portail mais y accéder en transférant les coordonnées du joueur vers celles du portail.

```
1 reference
IEnumerator TeleportSequence()
{
    isTeleporting = true;
    Debug.Log("► Téléportation imminente...");

    if (playerMovement != null) playerMovement.SetCanMove(false);

    if (teleportEffect != null) teleportEffect.Play();

    yield return new WaitForSeconds(teleportDelay);

    rb.velocity = Vector3.zero;
    Vector3 targetPos = new Vector3(currentPortal.transform.position.x, transform.position.y, currentPortal.transform.position.z);
    transform.position = targetPos;

    Debug.Log("► Téléportation réussie !");

    if (playerMovement != null) playerMovement.SetCanMove(true);
    isTeleporting = false;
}
```

8 - Le switch

Le switch est la dernière mécanique de jeu apportée. Celle-ci vient rajouter une réflexion importante pour pouvoir parcourir le labyrinthe. Désormais le labyrinthe ne possède à la base pas de réel sortie, il est complètement impossible de sortir du jeu. Pour cause celui-ci est divisé en 4, ce sont deux murs géants traversant la totalité de la carte, un à l'horizontale et le second à la verticale, tous deux passant par le centre, qui empêchent le joueur de passer. En revanche, les deux ne sont jamais présents ensemble, il y a donc deux configurations possibles. La première est celle avec le mur vertical présent et la seconde celle avec le mur cette fois-ci horizontal de présent. Il faut donc réussir à avancer dans le labyrinthe en alternant continuellement entre les deux configurations en appuyant sur "I".

Pour cette fonctionnalité il suffisait juste de rajouter deux murs géants en leur attachant un script qui venait détecter avec le clavier du joueur quelle configuration il souhaite. Puis par conséquent désactiver ou non un mur.

```
void ToggleWalls()
{
    if (horizontalWall == null || verticalWall == null) return;

    bool isHorizActive = horizontalWall.activeSelf;

    horizontalWall.SetActive(!isHorizActive);

    verticalWall.SetActive(isHorizActive);

    Debug.Log("SWITCH ! Labyrinthe modifié.");
}
```

9 - Le petit guide du jeu

Afin de faciliter la prise en main et l'évaluation du projet *MazeRunning*, ce guide récapitule les objectifs, les contrôles ainsi que les commandes de débogage (considérées comme des cheats code) implémentées pour tester et terminer plus rapidement le jeu.

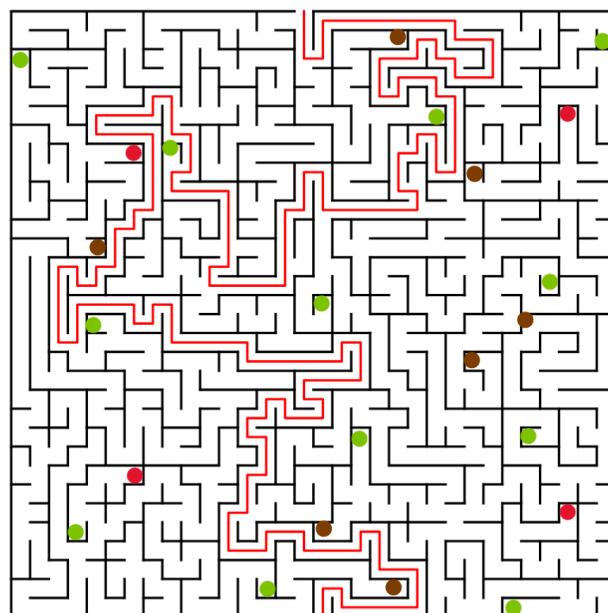
1. Objectif du jeu

Le joueur incarne une souris ("Mousy") piégée dans un labyrinthe évolutif.

- **But principal** : Trouver et ouvrir les coffres pour récupérer **3 clés**.
- **Condition de victoire** : Une fois les 3 clés en poche, la porte de sortie s'ouvre.
- **Condition de défaite** : Si le joueur entre en contact avec un **Fantôme (Ennemi)**, il est capturé et téléporté instantanément à son point de départ. Les ennemis patrouillent de manière autonome grâce à une IA (NavMesh).

2. La carte

Pour se repérer dans le jeu, voici la carte. Le trait rouge est le chemin de sortie (pour correction désactiver les murs switch afin de traverser librement). Les points marron sont les coffres, les points rouges les ennemis et les points verts les points d'observation.



3. Contrôles du Personnage (Clavier & Souris)

Action	Touche	Description
Se Déplacer	Z Q S D	Mouvements directionnels (Avancer, Gauche, Reculer, Droite).
Caméra	Souris	Orienter la vue (regarder autour de soi).

Changer de Vue	TAB	Basculer entre la vue 1ère personne (FPS) et 3ème personne (TPS).
Interagir	E	Action contextuelle : - Ouvrir un coffre. - Activer un poste d'observation (vue aérienne).
Portail	P	Système de téléportation : - 1er appui : Pose un portail au sol. - 2ème appui : Téléporte le joueur au portail posé.
Structuration	I	Modifie la structure du labyrinthe (fait apparaître/disparaître certains murs).

4. Cheat Code (Pour l'évaluation)

Afin de permettre une vérification rapide de toutes les mécaniques de jeu sans avoir à parcourir l'intégralité du labyrinthe, des commandes spécifiques ont été laissées actives dans cette version ("Cheat Codes") :

- **Touche [K] - "Give All Keys"** : Donne instantanément les 3 clés au joueur (simule la récupération des coffres). Utile pour tester la condition de victoire sans chercher les coffres, et ainsi ouvrir la porte.
- **Touche [J] - "Clear Walls"** : Détruit les murs destructibles du labyrinthe. Utile pour naviguer librement et rejoindre la sortie ou un point d'intérêt sans restriction.

Conseils de Survie

- **Les Ennemis** : Ils ne peuvent pas traverser les murs, mais ils changent de direction s'ils sont bloqués. Observez leur ronde avant de traverser un couloir.
- **L'Orientation** : Le labyrinthe est vaste. Utilisez les **Postes d'Observation (Touche E)**, placés dans des culs de sac, pour prendre de la hauteur et repérer les coffres ou la sortie.

10 - Conclusion

Ce projet nous a permis de prendre en main une nouvelle façon de coder dans un domaine amusant. Nous avons créé un labyrinthe avec de multiples fonctionnalités apportant un véritable gameplay. Nous avons donc appris à gérer un environnement 3D et à apporter du code via des scripts C#.

Malgré tout, de nombreuses difficultés sont apparues notamment la gestion d'objets 3D et leur interaction.

C'est avec cet ensemble de difficultés et de travail que nous avons pu produire ce petit jeu qui nous a plu de développer.

UTILISATION DE l'IA :

Lors de ce projet nous avons eu recours à l'ia pour deux choses principales. Tout d'abord il est important de dire que toute production de l'ia a été longuement étudiée afin d'être comprise et utilisée de bonne manière. Toute production est donc entièrement faite par nous.

Mais l'ia nous a permis premièrement de nous aider avec l'interface Unity, notamment pour les problèmes rencontrés par exemple avec la map, ou l'ia nous donnait des conseils pour mieux prendre en main unity.

Elle nous a aussi été utile dans les scripts pour comprendre comment mieux organiser le code.