Projet : The Maze Runner

Présenté par : Badr Mesbahi
Sous l'ecadrement de : Vincent Limouzy

Responsables Projet : Mamadou Kanté
Responsable ISIMA : Murielle Mouzat

Projet de 60 heures

3 mars 2023

Remerciement

Je souhaiterais exprimer ma gratitude envers M. Vincent Limouzy pour sa précieuse assistance tout au long de ce projet. Grâce à son expertise en algorithmique, j'ai pu concevoir et mettre en œuvre ce projet.

De plus, je tiens à remercier sincèrement M. MAMADOU Kanté pour avoir accepté ce projet et pour nous avoir accordé sa confiance tout au long de sa réalisation.

Résumé

L'industrie des jeux 3D est dans une évolution constante et elle offre des jeux de plus en plus immersifs et réalistes, avec des graphismes et des mécaniques de jeu sophistiquées. Les avancées technologiques telles que la réalité virtuelle et la réalité augmentée ont également contribué à améliorer l'expérience de jeu en offrant une immersion encore plus profonde. De plus, l'industrie du jeu continue de repousser les limites de la virtualisation et de la construction de mondes, offrant des univers de plus en plus vastes et complexes à explorer. Tout cela contribue à faire des jeux 3D un divertissement populaire et captivant pour de nombreux joueurs à travers le monde. Dans ce sens, le projet THE MAZE RUNNER à pour but de créer un jeu immersif qui combine plusieurs notions mathématiques et des outils de développement des jeux plus adaptés pour générer un labyrinthe, permettre à l'utilisateur d'y naviguer et d'interagir avec ses objets. Afin de créer ce jeu, nous avons utilisé la plateforme de développement de jeux Unity3D. Cette plateforme offre des outils et des fonctionnalitées puissantes qui en font un choix idéal pour la création de jeux 3D, y compris la capacité d'importer des éléments et de créer des environnements interactifs.

Abstract

The 3D gaming industry is constantly evolving and offering increasingly immersive and realistic games with sophisticated graphics and gameplay mechanics. Technological advancements such as virtual reality and augmented reality have also contributed to emprove the gaming experience by providing even deeper immersion. Furthermore, the gaming industry continues to push the boundaries of storytelling and world-building, offering increasingly vast and complex universes to explore. All of this contributes of making 3D games a popular and captivating form of entertainment for many players around the world. In this regard, THE MAZE RUNNER project aims to create an immersive game that combines several mathematical concepts and more adapted game development tools to generate a maze, allow the user to navigate in it, and interact with its objects. In order to create this immersive game, we will be using the game development platform Unity3D. This platform offers powerful tools and features that make it an ideal choice for creating 3D games, including the ability to import assets and create interactive environments.

Table des matières

Table des figures

Introduction

La création de jeux immersifs est d'une grande importance dans l'industrie du jeu vidéo car cela offre une expérience de jeu réaliste et captivante pour les joueurs. Ces jeux permettent aux joueurs de plonger dans des mondes virtuels et de vivre des aventures excitantes. De plus, la création de jeux immersifs est un moteur pour l'innovation technologique, offrant de nouveaux défis pour les développeurs et de nouvelles opportunités pour les joueurs.

La théorie des graphes est particulièrement pertinente pour la création de jeux immersifs car elle peut aider à concevoir des environnements de jeu intéressants. En effet, les jeux immersifs peuvent être considérés comme des graphes, où les nœuds représentent les différents éléments du jeu tels que les personnages, les objets et les lieux, et les arêtes représentent les relations entre ces éléments. En utilisant la théorie des graphes, le développement de jeux arrive à concevoir des niveaux de jeu qui sont à la fois cohérents et complexes, tout en permettant une exploration libre pour le joueur.

De plus, la théorie des graphes peut être utilisée pour optimiser le gameplay en permettant aux joueurs de découvrir des chemins et des itinéraires cachés dans le jeu. En utilisant des algorithmes de parcours de graphes, on peut créer des environnements de jeu dynamiques qui offrent des défis uniques pour les joueurs.

Le projet THE MAZE RUNNER en question est un jeu de survie créé avec Unity3D. L'objectif est de concevoir un labyrinthe, en se basant sur la théorie des graphes, qui plonge l'utilisateur dans une expérience avec une caméra à la première personne et lui permettant de se déplacer librement. Le jeu est agrémenté d'un ennemi dont le but est de traquer l'utilisateur et de se rapprocher de lui. De plus, le jeu permet l'interaction entre l'utilisateur et l'ennemi afin augmenter la tension et la difficulté de l'expérience de jeu.

1 Contexte du projet

1.1 Outils

Pour le développement du projet, nous avons utilisé le moteur de jeu Unity3D. Unity est un moteur de jeu multiplateforme qui permet de développer des jeux en 2D ou en 3D pour une grande variété des apareils électroniques, telles que les ordinateurs, les consoles de jeux, les appareils mobiles... Unity propose un environnement de développement intégré (IDE) qui permet de créer des jeux en utilisant une combinaison de langages de programmation, notamment C#, JavaScript et Boo.

L'un des outils clés de la programmation avec Unity est la classe MonoBehaviour. MonoBehaviour est une classe de base dans Unity qui permet aux développeurs de créer des scripts personnalisés qui peuvent être attachés à des objets dans la scène du jeu. Les scripts MonoBehaviour permettent de contrôler le comportement des objets dans la scène, tels que les mouvements, les interactions avec le joueur, les effets sonores et visuels.

Les scripts MonoBehaviour sont écrits en utilisant le langage de programmation C#, qui est un langage orienté objet puissant. Les scripts peuvent être écrits directement dans l'IDE Unity, ou dans un environnement de développement tiers tel que Visual Studio. Ils sont ensuite attachés à des objets dans la scène du jeu en les faisant glisser et en les déposant sur l'objet concerné. Les objets , en Unity3D, se sont des formes possédant des propriétés physiques mises dans un espace muni d'une base absolue ($\vec{x}$, $\vec{y}$, $\vec{z}$), qui est une base orthonormée indirecte. Une base orthonormée indirecte est une base vectorielle dans laquelle chaque vecteur a une norme égale à 1 et le produit vectoriel de deux de ses vecteurs est égal au troisième vecteur de la base, mais orienté dans la direction opposée.

Lorsque le jeu est exécuté, les scripts MonoBehaviour attachés à chaque objet sont automatiquement activés et commencent à contrôler le comportement de l'objet. Nous pouvons également utiliser des événements et des fonctions prédéfinis dans Unity pour interagir avec d'autres objets dans la scène, tels que le joueur, les ennemis, les obstacles.

Sans oublier la théorie des graphes et l'algèbre linéaire, qui sont deux domaines des mathématiques qui ont une importance fondamentale dans la création de jeux vidéo en 3D. La théorie des graphes permet de modéliser et de représenter des objets 3D sous forme de graphes. En effet, les objets 3D peuvent être vus comme des ensembles de points reliés par des segments ou des polygones. Ces relations peuvent être représentées par des graphes, où les points sont des nœuds et les segments ou polygones sont des arêtes. La théorie des graphes permet également de modéliser des phénomènes de propagation, de déplacement et de comportement de personnages en jeu en utilisant des algorithmes de parcours de graphes. Cette modélisation permet de créer des environnements de jeu réalistes et interactifs.

L'algèbre linéaire permet de manipuler des matrices et des vecteurs, qui sont utilisés pour décrire des transformations géométriques telles que des translations, des rotations et des changements de bases. Ces transformations sont essentielles dans la création de modèles 3D et dans la gestion des mouvements et des interactions des personnages en jeu. Par exemple, les mouvements des personnages peuvent être décrits par des vecteurs, qui sont transformés par des matrices pour représenter les mouvements en 3D.

1.2 Objectifs

Le projet The Maze Runner a pour objectif de concevoir un jeu de survie hautement immersif, qui transportera le joueur dans un monde virtuel complexe. Plus précisément, ce jeu doit proposer un labyrinthe, conçu pour défier les compétences et les capacités de survie du joueur, tout en l'immergeant dans une expérience visuelle et sonore des plus intenses.

Pour atteindre cet objectif ambitieux, le projet vise à créer un ensemble de fonctionnalités sophistiquées, qui permettront à l'utilisateur de contrôler et de gérer ses interactions avec le jeu, en utilisant des interruptions extérieures déclenchées par des dispositifs tels que la souris ou le clavier.

En outre, le projet vise à intégrer un ennemi capable de suivre et interagir avec l'utilisateur dans le labyrinthe, ce qui ajoute une dimension supplémentaire au défi de survie du joueur et rend l'expérience de jeu encore captivante.

2 Développement du projet

2.1 Création de la scène

La scène d'un jeu immersif est un élément crucial pour offrir aux joueurs une meilleur expérience de jeu. Elle est souvent le premier élément que les joueurs voient et c'est ce qui leur donne leur première impression du jeu. Une scène bien conçue peut contribuer largement à l'immersion des joueurs dans le monde virtuel du jeu.

Pour cela, avant d'immerger l'utilisateur dans l'expérience de jeu, nous avons commencé par la création de la scène qui vérifie certaines conditions nécessaires. La scène est un labyrinthe connexe, en plus il doit être suffisamment rempli par des obstacles. Autrement dit, pour chaque paire de cases de la scène labyrinthe doit présenter au moins un chemin qui les relies, et pour chaque groupe de case 2x2 il doit présenter au minimum un mur.

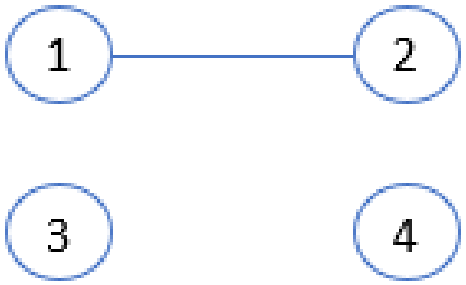


FIGURE 1 : Sommets 2x2

La méthode choisie pour créer un graphe comme décrit précédemment, débute par la construction d'une grille vide, c'est-à-dire sans arrêts, où chaque sommet de la grille représente une case dans la scène, et la présence d'un arc traduit la non-existence d'obstacle entre la paire de sommets qui les relies. Le rôle de l'algorithme générateur de labyrinthe, utilisé par la suite, est d'ajouter des arrêts à cette grille.

En matière de structure de données, nous avons choisi la représentation par listes d'adjacences pour représenter ce graphe. la représentation ,par listes d'adjacences d'un graphe G = (S, A), consiste en un tableau Adj de |S| listes, une pour chaque sommet de S. Pour chaque u $\in$ S, la liste d'adjacences Adj[u] est une liste des sommets v tels qu'il existe un arc (u, v) $\in$ A.

Pour notre cas, cette représentation des graphes est la plus adéquate puisque notre grille vérifie la condition suivante :

$|A| < |S|^2$ [1]

L'implémentation d'un graphe est faite suivant les trois classes suivantes :

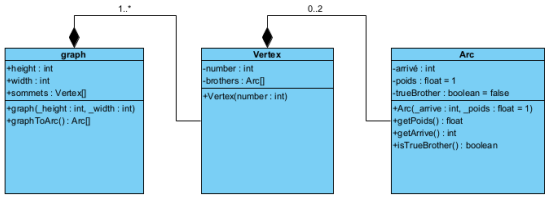


FIGURE 2 : Classe graphe

Afin de vérifier l'intégrité des scènes créer, nous avons implémenté la classe SceneCreator permettant de générer une scène dans l'espace de développement Unity à partir d'un graphe représenté sous la forme de listes d'adjacences. La classe SceneCreator est une classe attachée à un objet vide Scene (Empty Game Object) de Unity. Son rôle est de créer un plan sur lequel elle génère labyrinthe à l'aide de la méthode **printScene()**.

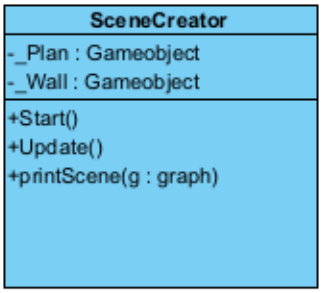


FIGURE 3 : Classe SceneCreator

La méthode **Start** est une méthode héritée de la classe MonoBehaviour, elle est la première méthode exécutée lorsque l'objet Scene est active. Cette méthode initialise la grille de départ, appelle une méthode extérieure de génération de labyrinthe et crée la scène à l'aide de la méthode printScene.

En passant la grille initiale comme paramètre à cette classe, nous obtenant le résultat suivant :

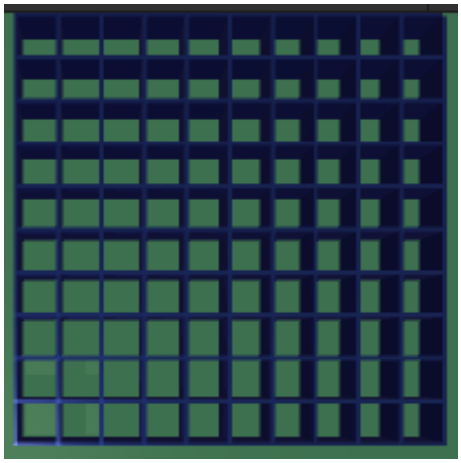


FIGURE 4 : Grille de départ

Une fois que le générateur de scènes est développé, nous pouvons procéder à la conception d'un algorithme pour la création du labyrinthe qui assure le respect des propriétés souhaitées.

2.1.1 Algorithme de Kruskal

Nous avons appliqué l'algorithme de Kruskal pour garantir la convexité du graphe généré, ce qui se traduit par la production d'un arbre couvrant en sortie.En ce qui concerne l'algorithme de Kruskal, il s'agit d'un algorithme de parcours de graphe qui permet de trouver un arbre couvrant de poids minimum dans un graphe non orienté et pondéré.

L'algorithme commence par trier les arêtes du graphe en ordre croissant selon leur poids. Ensuite, l'algorithme sélectionne l'arête de poids minimum et l'ajoute à l'arbre couvrant en construction. Si l'ajout de cette arête forme un cycle avec les arêtes déjà sélectionnées, alors elle est rejetée. L'algorithme continue ce processus jusqu'à ce que toutes les arêtes soient parcourues ou que l'arbre couvrant soit complet. En sortie, l'algorithme renvoie l'arbre couvrant de poids minimum.

Pour implémenter cet algorithme, nous avons utilisé des partitions. Le rôle des partitions dans l'algorithme de Kruskal est de suivre les composantes connectées du graphe en cours de traitement. Une partition est un ensemble de sommets du graphe qui sont connectés entre eux et qui peuvent être fusionnés avec d'autres partitions au fur et à mesure que l'algorithme de Kruskal progresse.

Au début de l'algorithme, chaque sommet du graphe appartient à une partition distincte. Lorsqu'une arête est traitée, l'algorithme vérifie dans quelles partitions se trouvent les sommets reliés par cette arête. Si les sommets se trouvent dans des partitions différentes, alors l'algorithme fusionne ces partitions en une seule partition en combinant tous les sommets de ces partitions en un seul ensemble.

En fusionnant les partitions, l'algorithme garantit que les sommets qui sont reliés

par l'arête sont maintenant dans la même partition, ce qui signifie qu'ils sont connectés. Cela permet également à l'algorithme de suivre les composantes connectées du graphe au fur et à mesure que les partitions sont fusionnées.

À la fin de l'algorithme, l'ensemble de partitions résultant représente l'arbre couvrant de poids minimum pour le graphe. En utilisant les partitions pour suivre les composantes connectées du graphe, l'algorithme de Kruskal garantit que les arêtes ajoutées à l'arbre couvrant ne forment jamais de cycle, ce qui est un aspect clé pour obtenir un arbre couvrant de poids minimum.

L'algorithme de Kruskal est implémenté à l'aide des partitions de la manière suivante :

1- Tout d'abord, on trie les arêtes du graphe en ordre croissant selon leur poids.
2- Ensuite, on crée une partition pour chaque sommet du graphe, où chaque partition ne contient initialement que le sommet correspondant.
3- On parcourt toutes les arêtes triées, de la plus légère à la plus lourde, et pour chaque arête, on effectue les étapes suivantes :

a. On détermine les partitions qui contiennent les deux sommets reliés par l'arête.

b. Si ces partitions sont différentes, on fusionne ces partitions en les combinant en une seule partition.

c. On ajoute l'arête à l'arbre couvrant en construction.

d. On continue ce processus jusqu'à ce que toutes les arêtes aient été parcourues ou que l'arbre couvrant soit complet.

Cette manière permet de garantir que les arêtes ajoutées à l'arbre couvrant ne forment jamais de cycle. Cette approche est efficace car elle ne nécessite pas de parcours exhaustif du graphe à chaque étape, mais plutôt une gestion intelligente des partitions.

En appliquant cet algorithme on obtient la sortie suivante à l'aide de l'objet Scene.

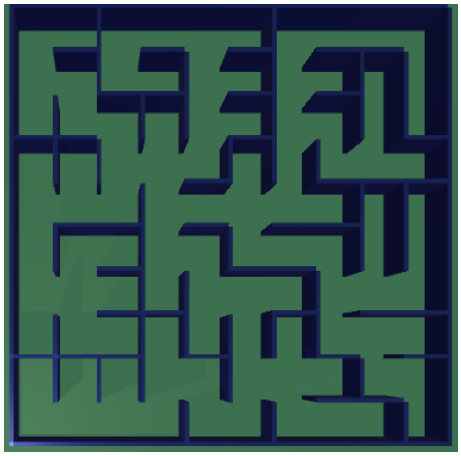


FIGURE 5 : Labyrinthe généré en utilisant l'algorithme kruskal

Le graphe de sortie présente un seul chemin entre deux cases de la scène, cela ne laisse pas assez de choix à l'utilisateur de s'échapper de son ennemi. En plus, en appliquant cet algorithme pour une deuxième fois sur le graphe résultant est loin d'être un labyrinthe.

FIGURE 6 : Resultat d'appliquant l'algorithme kruskal deux fois

En effet, le graphe résultant présente moins d'obstacle ce qui augmente la facilité du jeu. Pour cela, nous avons cherché une autre méthode qui permet de construire un labyrinthe plus réaliste, conformément aux conditions reliées à la convexité et la suffisance des obstacles. Ce qui nous mène à penser à une autre approche pour créer cette labyrinthe.

2.1.2 Algorithme de BackTracking

L'algorithme de BackTracking est une technique de résolution de problèmes qui consiste à explorer de manière exhaustive toutes les solutions possibles d'un problème en utilisant une méthode de recherche arborescente.

L'idée principale de l'algorithme de BackTracking est de construire progressivement une solution partielle du problème en faisant des choix à chaque étape. Si ces choix ne conduisent pas à une solution valide, l'algorithme revient en arrière (d'où le terme "BackTracking") et essaie une autre possibilité.

L'algorithme de BackTracking peut être appliqué à une grande variété de problèmes, tels que la recherche de chemins dans un graphe comme pour notre cas, la résolution de puzzles comme le Sudoku, ou encore la génération de permutations ou de combinaisons.

Le processus de BackTracking commence par construire une solution partielle en faisant un choix initial. Ensuite, l'algorithme explore tous les choix possibles à partir de cette solution partielle en utilisant une récursion. À chaque étape de la récursion, l'algorithme examine si la solution partielle actuelle peut être étendue en ajoutant

un nouvel élément. Si c'est le cas, il construit une nouvelle solution partielle qui inclut cet élément et poursuit la recherche en explorant toutes les possibilités à partir de cette nouvelle solution. Si à un moment donné aucune extension ne peut être effectuée, l'algorithme revient en arrière pour essayer une autre possibilité.

L'algorithme de BackTracking peut être utilisé pour générer un labyrinthe en construisant progressivement un ensemble de chemins qui traversent le labyrinthe. Le principe de base est de commencer par une grille pleine sans murs comme précédemment, puis de retirer progressivement les murs pour créer un labyrinthe.

Le processus de génération du labyrinthe commence par choisir une cellule de départ et de la marquer comme visitée. Ensuite, on choisit une cellule adjacente non visitée au hasard et on crée un chemin entre ces deux cellules en retirant le mur qui les sépare. On marque ensuite la cellule nouvellement visitée comme visitée et on répète le processus en choisissant une nouvelle cellule adjacente non visitée.

Lorsqu'on arrive à une cellule qui n'a pas de cellules voisines non visitées, on revient en arrière en utilisant le BackTracking pour explorer les autres chemins possibles à partir des cellules précédentes jusqu'à trouver une cellule non visitée à partir de laquelle on peut continuer à explorer.

Le processus se termine lorsque toutes les cellules ont été visitées et un labyrinthe a été créé. À ce stade, il est possible de retirer certains murs supplémentaires pour créer des boucles dans le labyrinthe, ou pour ajouter des points d'entrée et de sortie.

Pour l'implémentation, nous avons choisie de créer une classe statique nommée BackTracking. Cette classe contient une méthode principale, nommée également BackTracking, qui modifie la grille de départ en respectant les étapes suivantes :

1) choisir une case de départ au hasard
2) tant qu'il reste au moins un choix de cas à faire :
    2.1) pour chaque direction possible (nord, est, sud, ouest) :
    2.1.1) si la case adjacente dans cette direction n'a pas été visitée :
    2.1.2) créer un arrêt entre la case actuelle et la case adjacente

La sortie obtenue ,dans la figure, reflet exactement le résultat souhaité. Plus précisément, un labyrinthe connexe avec circuit et pour chaque groupe de quatre sommets 2x2 il existe au moins une arête comme la figure ci-dessous 7.

Une fois que la scène a été créée sous forme de labyrinthe, il est primordial de se concentrer sur l'immersion de l'utilisateur pour offrir une expérience plus réaliste. Cela implique de mettre en place des éléments qui renforcent l'immersion, tels que des effets visuels, ainsi que de permettre à l'utilisateur de devenir un acteur actif dans la scène en lui permettant d'interagir et de se déplacer dans la scène.

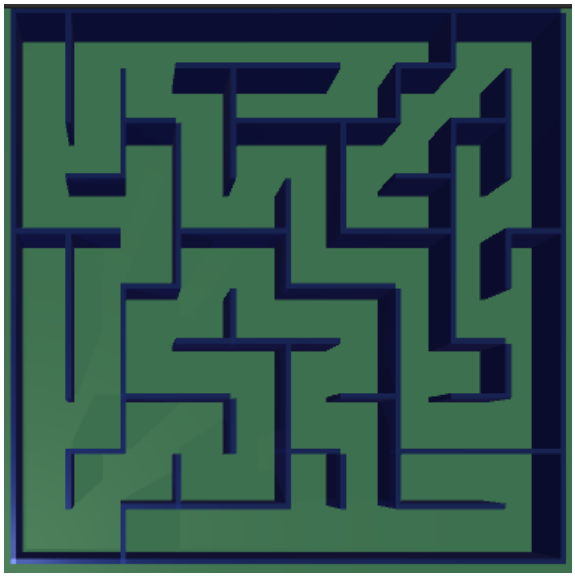


FIGURE 7 : labyrinthe généré en utilisant le BackTracking

2.2 Immersion de l'utilisateur

L'immersion de l'utilisateur reste un élement important dans la conception de ce jeu 3D, car elle permet au joueur de se sentir plongé dans le monde virtuel. Unity propose plusieurs fonctionnalités pour créer une expérience immersive, notamment l'utilisation d'une capsule pour gérer les collisions et d'une caméra pour fournir une vue à la première personne.

La capsule est un objet qui représente l'avatar du joueur dans la scène. Elle est utilisée pour gérer les collisions avec les autres objets. En définissant correctement la taille et la forme de la capsule, on peut s'assurer que le joueur ne peut pas traverser les murs ou les objets, ce qui contribue à renforcer l'illusion de la réalité.

La caméra est un autre élément essentiel pour créer une expérience immersive. Nous avons placé la caméra à l'intérieur d'une capsule, de manière à fournir une vue à la première personne. En ajustant la position, l'orientation et le champ de vision de la caméra, on peut créer une perspective qui permet aux joueurs de se sentir comme s'ils étaient réellement dans le monde virtuel.

En combinant la capsule et la caméra, nous pouvons créer une expérience immersive où le joueur peux se déplacer librement dans le monde virtuel, interagir avec les objets et l'ennemi, et avoir l'impression d'être réellement présents dans cette scène.

2.2.1 Caméra

La caméra à la première personne, ou *first-person camera*, utilisée est un type de caméra qui simule le point de vue d'un personnage en affichant l'image telle qu'elle serait vue par ses yeux. Cette technique est souvent utilisée dans les jeux vidéo pour donner au joueur une immersion plus réaliste dans le monde virtuel du jeu. En utilisant une caméra à la première personne, le joueur voit le monde à travers

les yeux du personnage qu'il contrôle, ce qui peut lui donner l'impression d'être physiquement présent dans l'environnement du jeu.

Dans notre contexte, la caméra est attachée à la capsule et sa direction précise la direction de l'utilisateur. En effet, un avancement de l'utilisateur se traduit par l'avancement suivant la direction de la caméra et non pas suivant la base absolue de l'espace Unity.

Le mouvement de la caméra est contrôlé par le script *CameraControler* relié à elle. Pour cela, nous utilisons la classe Input fourni par Unity qui permet de récupérer les entrées de l'utilisateur, telles que les touches du clavier et les mouvements de la souris.

```
Script Unity (1 référence de ressource) | 0 références
public class CameraControler : MonoBehaviour
{
    // Start is called before the first frame update
    public GameObject player;
    public float SenceX = 100f;
    public float SenceY = 100f;
    private float xRotation;
    private float yRotation;


    //public Transform playerbody;
    Message Unity | 0 références
    void Start()
    {
        Cursor.lockState = CursorLockMode.Locked ; // cursor in the midle of the screen
        Cursor.visible = false ; // cursor invisible
        xRotation = player.transform.rotation.x;
        yRotation = player.transform.rotation.y;

    }

    // Update is called once per frame
    Message Unity | 0 références
    void Update()
    {
        float mousX = Input.GetAxisRaw("Mouse X") * SenceX * Time.deltaTime;
        float mousY = Input.GetAxisRaw("Mouse Y") * SenceY * Time.deltaTime;


        yRotation += mousX;

        xRotation -= mousY;

        xRotation = Mathf.Clamp(xRotation, -90f, 90f);
        player.transform.rotation = Quaternion.Euler(xRotation, yRotation, 0);

    }
}
```

FIGURE 8 : Gestion de la rotation de la caméra

Lorsque la caméra est active, la méthode *Start* attache la souri au milieu de l'écran et il la rend invisible. Le mouvement de la caméra est controlé spécialement par la méthode *Update()*. Cette méthode, est une méthode intégrée dans Unity qui est appelée une fois à chaque frame ou image affichée à l'écran. Cela signifie que la méthode *Update()* est appelée environ 60 fois par seconde, par défaut, si la vitesse de rafraîchissement de l'affichage est de 60 Hz.

En utilisant la méthode *GetAxisRaw* , fourni par la classe *Input*, nous arrivons à détecté le mouvement de la souri suivant les axes X et Y de l'écran. Nous multipliant

cette grandeur par un constante qui représente la sensibilité souhaitée suivant chaque axe et par *Time.deltaTime*. Cette grandeur temporelle représente le temps écoulé en secondes depuis la dernière Frame rendue. Lorsque nous multiplions le mouvement (vitesse ou distance) par Time.deltaTime, nous adaptons le mouvement pour qu'il soit exécuté sur une durée équivalente à une frame, quelle que soit la vitesse à laquelle ces frames sont rendues. En d'autres termes, la multiplication par Time.deltaTime permet de rendre le mouvement indépendant de la fréquence d'images.

Les résultats mousX et mousY correspondent aux angles de rotation à appliquer autour des axes X et Y de la scène, ce qui permet d'obtenir une vue sur les objets qui l'entourent.

Une fois que l'on a la possibilité d'appliquer des rotations à la caméra, il est nécessaire de développer un mécanisme de détection de sa direction car celle-ci représente le point de vue de l'utilisateur. En effet, la direction de la caméra est cruciale pour les déplacements dans la scène et l'interaction avec les ennemis.

2.2.2 déplacement

A cette étape, l'utilisateur est capable de changer la direction de la caméra en appliquant des rotations autour de l'axe perpendiculaire au plan (l'axe y pour notre cas).Comme la figure 9 montre la base de la caméra $B_c = (\vec{x_c}, \vec{y_c}, \vec{z_c})$ effectue une rotation d'angle α_c autour de l'axe y de la base absolue $B_a = (\vec{x}, \vec{y}, \vec{z})$. Nous avons décidé de ne pas prendre en compte la rotation autour de l'axe X afin de ne pas impacter la vitesse de déplacement.

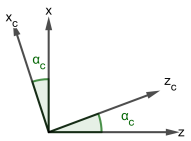


FIGURE 9 : rotation de B_c par rapport à B_a

Cela nous permet de définir une matrice de changement de base entre les deux bases, comme suivant, pour pouvoir préciser la direction de la caméra par rapport à la base absolue de l'espace du développement. Plus précisément, en multipliant cette matrice avec le vecteur de la direction choisie nous pouvant obtenir la direction absolue que l'utilisateur souhaite suivre.

$$\mathbf{M} = \begin{bmatrix} \cos(\alpha_c) & 0 & \sin(\alpha_c) \\ 0 & 1 & 0 \\ -\sin(\alpha_c) & 0 & \cos(\alpha_c) \end{bmatrix}$$

L'utilisateur a la possibilité de sélectionner la direction de son choix, parmi les directions suivantes, en utilisant les boutons fléchés du clavier, lui permettant ainsi de choisir parmi les quatre directions disponibles.

$$\tilde{\mathbf{v_1}} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \tilde{\mathbf{v_2}} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} \quad \tilde{\mathbf{v_3}} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad \tilde{\mathbf{v_4}} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

Où les vecteur $\vec{v_1}$ et $\vec{v_2}$ représentent les déplacements suivant $\vec{x}$, et les vecteur $\vec{v_3}$ et $\vec{v_4}$ représentent les déplacements suivant $\vec{z}$. Et cela nous permet de préciser la direction absolue choisie par l'utilisateur qui est représenté par le vecteur $\vec{D_i}$ tel que : $\vec{D_i} = \mathrm{M} \times \vec{v_i}$. Nous avons multiplier la direction par un coefficient modulable afin d'adapter la vitesse du déplacement.

Une fois que la direction souhaitée a été sélectionnée, il est nécessaire de prendre en compte les collisions avec les obstacles présents dans le labyrinthe. Pour cela, deux choix s'offrent à nous. Le premier choix consiste à déléguer le traitement des collisions à Unity, qui utilise des calcule physique. Le deuxième choix est d'utiliser la structure de graphe représentant le labyrinthe pour détecter la présence d'un obstacle. Le choix final dépendra de plusieurs facteurs, tels que la complexité du labyrinthe et les performances du système. Il est donc important de peser les avantages et les inconvénients de chaque approche avant de faire un choix.

Bien que déléguer le traitement des collisions à Unity puisse sembler être une solution simple et efficace, il présente également certains inconvénients. Tout d'abord, cela peut entraîner une surcharge de traitement, en particulier dans les labyrinthes complexes contenant de nombreux obstacles. En outre, la précision du traitement des collisions peut être affectée, car les colliders Unity ne sont pas toujours capables de détecter les collisions de manière fiable dans des situations complexes ou non conventionnelles. Enfin, la détection des collisions peut être moins flexible, car elle repose sur les fonctionnalités fournies par Unity, limitant ainsi la capacité de personnaliser et d'adapter la détection des collisions en fonction des besoins spécifiques de l'application.

En plus, utiliser la structure de graphe représentant le labyrinthe pour détecter les collisions offre plusieurs avantages. Tout d'abord, cela permet une plus grande flexibilité dans la détection des collisions, car la structure du graphe peut être adaptée pour prendre en compte les caractéristiques spécifiques du labyrinthe et les types d'obstacles présents. De plus, cela permet une meilleure précision dans la détection des collisions, car la méthode de détection est basée sur une représentation plus

précise et complète de la géométrie du labyrinthe. Enfin, cela peut entraîner une réduction de la charge de traitement, car la détection des collisions est effectuée de manière plus efficace et adaptée aux besoins spécifiques du déplacement.

Pour réaliser le premier choix,l a fonction Physics.OverlapSphere de la bibliothèque Unity peut être utilisée pour détecter la présence d'un objet en créant une sphère centrée sur l'objet avec un rayon suffisamment grand pour l'englober. Pour cela, nous appelons la fonction *Physics.OverlapSphere* en lui passant la position et le rayon de la sphère. Cette fonction renvoie un tableau d'objets *Collider* qui se trouvent dans la sphère.Nous pouvons alors parcourir ce tableau pour déterminer si le mur s'y trouve en vérifiant si son Collider est présent dans le tableau retourné. Il est important de noter que cette méthode de détection est basée sur les collisions entre les Collider des objets, il est donc important que les murs possède un Collider attaché pour qu'il soit détecté. Si la liste des objets trouvés ne contient pas un mur, nous permettant le déplacement vers la direction absolue choisie par l'utilisateur.

Pour la second option, nous pouvons détecter la présence d'un mur entre la position de l'utilisateur et le point où il souhaite se déplacer à l'aide du graphe. En fait, si la position de l'utilisateur et ce point existe dans la même case de labyrinthe nous permettons le déplacement, sinon nous accédons au graphe pour savoir s'il existe une arête entre les deux cases, si elle existe alors le déplacement est aussi autorisé. Nous interdirons le déplacement dans le cas échéant.

Selon ces deux manière, nous avons créé les deux fonctions goInDirection et goInDerction2, qui implémentent respectivement la première et la deuxième méthode. Pour notre labyrinthe, la deuxième méthode nous obtenons une latence quatre fois moins que la première, ce qui confirme l'analyse précèdent.

Après avoir géré les déplacements de l'utilisateur, il est nécessaire de mettre en place un moyen d'interagir avec les ennemis. Dans ce but, nous avons décidé d'équiper l'utilisateur d'une arme.

2.2.3 Arme

l'inclusion d'armes avec des effets de reflet visuel dans le jeu peut améliorer l'expérience de jeu globale et contribuer à rendre l'environnement plus immersifs et attrayants pour l'utilisateur.

Dans Unity, il est possible de construire une arme en combinant plusieurs Gameobjects. Pour cela, on peut utiliser un parent vide comme conteneur pour les différents Gameobjects qui composent l'arme.

Pour ce jeu, nous avons construit un rifle de chasse. Pour cela, nous avons créer un parent vide appelé "Rifle" et y ajouter les Gameobjects suivants en tant qu'enfants de ce parent :

- Le canon (représenté par un Gameobject avec une forme cylindrique et une

texture en métal)

- La crosse (représentée par un Gameobject avec une forme ergonomique et une texture en bois)
- Le chargeur (représenté par un Gameobject avec une forme rectangulaire et une texture en métal)
- Le viseur (représenté par un Gameobject avec une forme en croix et une texture en verre)

Une fois que les différents Gameobjects sont attachés à l'objet parent "Rifle", nous avons ajuster leur position, leur rotation et leur échelle pour former un rifle complet et fonctionnel.

Pour chaque Gameobject, nous avons appliqué des textures pour donner des couleurs et des reflet spécifique. Dans le contexte de Unity, les textures sont des images qui sont appliquées à des objets 3D pour leur donner l'apparence souhaitée. Les textures peuvent être utilisées pour simuler des matériaux tels que le bois, le métal, la pierre, le verre..., ou pour créer des effets tels que la transparence, les reflets et les ombres.

Unity fournit plusieurs types de textures [3] préfabriquées qui peuvent être utilisées dans les jeux, notamment :

- Texture2D : une texture de base qui peut être appliquée à des objets 3D pour leur donner une apparence plus réaliste.
- NormalMap : une texture qui est utilisée pour simuler les détails de la surface d'un objet 3D, tels que les bosses, les creux et les aspérités. Elle permet de donner l'impression que l'objet a une texture complexe, même si sa géométrie est simple.
- BumpMap : similaire à NormalMap, mais avec une approche différente pour simuler les détails de la surface.
- CubeMap : une texture qui est utilisée pour simuler les reflets de l'environnement sur les surfaces des objets 3D. Elle est souvent utilisée pour créer des effets de réflexion sur les surfaces des voitures, des bâtiments et des autres objets qui doivent refléter leur environnement.
- LightMap : une texture qui est utilisée pour stocker les informations d'éclairage statique d'une scène. Elle permet de créer des ombres et des reflets plus réalistes sur les surfaces des objets 3D.

En plus des textures préfabriquées, nous pouvons également créer nos propres textures en utilisant des logiciels de création d'images tels que Photoshop, GIMP ou Blender, puis les importer dans Unity pour les utiliser dans le jeu. Les textures

peuvent être appliquées à des objets en utilisant des matériaux, qui sont des configurations qui définissent l'apparence d'un objet 3D en combinant plusieurs textures et paramètres d'éclairage.

un matériau est une configuration qui définit l'apparence visuelle d'un objet 3D en combinant plusieurs textures, couleurs et paramètres d'éclairage. Les matériaux sont appliqués aux surfaces des objets 3D pour donner l'apparence souhaitée.

Il existe plusieurs types de matériaux dans Unity, chacun ayant des propriétés et des utilisations spécifiques :

- Standard Material : C'est le matériau le plus couramment utilisé dans Unity. Il permet de définir la couleur de base de l'objet, sa texture, sa brillance, sa métallicité, sa rugosité et d'autres paramètres.
- Unlit Material : C'est un matériau simple qui n'utilise pas d'éclairage. Il est souvent utilisé pour créer des objets émissifs (comme des écrans, des lumières, etc.) ou pour appliquer des textures qui ne nécessitent pas d'éclairage (comme des icônes, des textes, etc.).
- Terrain Material : C'est un matériau spécialement conçu pour être utilisé avec les terrains de Unity. Il permet de définir la texture du sol, sa rugosité, sa hauteur, etc.
- Particle Material : C'est un matériau utilisé pour définir l'apparence des particules (comme la fumée, le feu, l'eau, etc.). Il permet de définir la couleur, la transparence, la texture et d'autres paramètres spécifiques aux particules.
- Skybox Material : C'est un matériau utilisé pour définir l'apparence du ciel et de l'horizon dans une scène. Il permet de définir la texture du ciel, sa couleur, sa luminosité, etc.
- Sprite Material : C'est un matériau utilisé pour définir l'apparence des sprites (images 2D) dans Unity. Il permet de définir la texture, la couleur, la transparence et d'autres paramètres spécifiques aux sprites.
- Water Material : C'est un matériau utilisé pour définir l'apparence de l'eau dans Unity. Il permet de définir la couleur, la transparence, la réflexion, la réfraction et d'autres paramètres spécifiques à l'eau.

Chaque type de matériau a ses propres propriétés et paramètres, qui peuvent être ajustés pour créer des effets visuels réalistes et personnalisés dans une scène Unity. Les matériaux peuvent être appliqués aux objets en utilisant des composants tels que Renderer, MeshRenderer et SpriteRenderer.

Après avoir sélectionné les paramètres appropriés et effectué les ajustements nécessaires, nous avons réussi à créer une arme qui ressemble beaucoup à la réalité, comme indiqué ci-dessous [10].

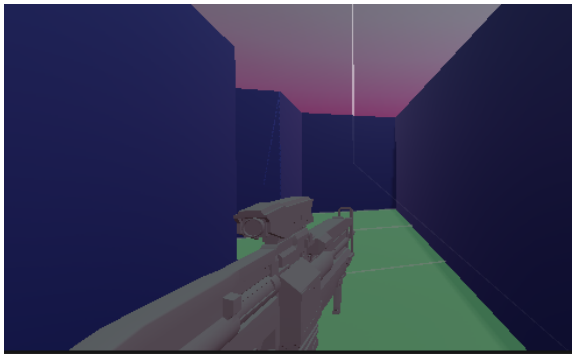


FIGURE 10 : La vue à la première personne de l'utilisateur sur la scène

Pour le tirage par l'arme sur l'ennemi, nous avons créé des reflet qui s'active en ce moment. l'entrée "Fire1" permet à activer ces reflets."Fire1" est un nom de bouton prédéfini dans Unity qui est utilisé pour détecter une entrée de tir du joueur. Il peut être utilisé pour créer des interactions entre le joueur et les objets dans le jeu, tels que des armes, comme pour notre cas, ou des objets interactifs.

Par défaut, "Fire1" est lié à la souris gauche et au bouton gauche du joystick sur les plateformes de jeu. Cela signifie que si le joueur clique sur le bouton gauche de la souris ou appuie sur le bouton gauche du joystick, l'entrée "Fire1" sera détectée et le script peut être configuré pour effectuer une action spécifique en réponse à cette entrée.

Pour notre jeu, les reflets sont créés à l'aide d'un système de particules [2]. Un système de particules est une technique d'animation en informatique graphique qui permet de simuler des effets visuels complexes tels que la fumée, le feu, la neige, la pluie, les explosions et bien d'autres encore.

Son principe de base est de générer et de manipuler un grand nombre d'objets graphiques (les particules) de manière dynamique en utilisant des algorithmes de simulation physiques ou artistiques. Les particules peuvent être configurées avec des propriétés telles que la masse, la vitesse, la direction, la taille, la couleur, la texture, la forme, la durée de vie et d'autres paramètres.

Les systèmes de particules sont couramment utilisés dans les jeux vidéo, les films d'animation, les publicités, les simulations scientifiques et les expériences interactives pour créer des effets visuels réalistes ou stylisés qui enrichissent l'expérience utilisateur.

Le système de particules que nous avons mis en place se compose de plusieurs points qui s'illuminent au moment du tir et s'éteignent après un certain délai. Pour l'implémenter dans Unity, nous avons procédé de la manière suivante :

- Création d'un nouveau système de particules : Dans l'onglet Hierarchy, à l'aide du boutton sur "Create" et nous avons sélectionné "Particle System" dans le menu déroulant pour créer un nouveau système de particules.

- Configuration du le système : Dans l'onglet Inspector, nous pouvons configurer les paramètres de notre système de particules pour qu'il crée l'effet de reflet souhaité. Pour cela, voici les étapes clés :
  - Sous "Start Lifetime", nous avons diminué la durée de vie de la particule pour que l'effet dure suffisamment court.
  - Sous "Shape", nous avons sélectionné "Sphere" pour créer un effet de particules sphérique.
  - Sous "Emission", nous avons augmenté le taux d'émission pour créer un plus grand nombre de particules.
  - Sous "Renderer", nous avons sélectionné "Mesh" pour pouvoir appliquer une texture de "muscle flash" aux particules.
- Ajout de la texture "muscle flash" : nous avons ajouter une texture de "muscle flash" pour donner l'apparence d'un flash de lumière intense lors du tir. Pour cela, nous avons créé une nouvelle texture que nous l'avons glissé sur l'onglet "Assets" pour l'importer dans notre projet Unity.
- Appliquation de la texture de "muscle flash" : Dans l'onglet Inspector, nous avons sélectionné "Add Component" et nous avons choisi "Mesh Renderer" pour ajouter un composant de rendu de maillage. Dans les paramètres du rendu, nous avons sélectionné la texture de "muscle flash" pour la texture de maillage.
- Animation du système : Pour animer le système de particules lors de l'utilisation de l'arme, nous avons utilisé un script pour activer et désactiver le système de particules en réponse à l'entrée du joueur.

En résumé, pour créer un effet de reflets au moment de tirer par l'arme, nous avons créé un système de particules, configuré ses paramètres pour créer l'effet souhaité, appliqué une texture de "muscle flash" et animé le système de particules en réponse à l'entrée "Fire1" du joueur.

En plus de créer un reflet, lors du tir, un Raycast est lancé pour détecter la présence d'un objet dans la ligne de mire de l'arme. Ce Raycast est une fonctionnalité fournie par Unity et se présente sous la forme d'une ligne droite où le point de départ du Raycast est placé à la position de l'arme et le point d'arrivée est déterminé en fonction de la direction de l'arme qui est parallèle à celle du caméra.

Jusqu'à présent, nous avons fourni au joueur les mises en œuvre nécessaires pour son interaction avec la scène grâce à une arme pointée en parallèle avec la caméra, qui produit également un effet visuel lors du tir. Nous allons maintenant passer à un autre élément important du jeu, à savoir l'ennemi.

2.3 Ennemi

L'ennemi est l'un des éléments important de notre jeu, étant donné que le but du joueur est de le vaincre dans ce jeu de survie. La création d'un ennemi implique

plusieurs étapes, et nous avons choisi d'utiliser une capsule pour représenter l'ennemi dans notre jeu. La capsule est un choix courant car elle permet une détection de collision plus précise avec le Raycast de l'arme et plus facilement manipulable. Une fois la capsule créée, nous avons travaillé sur son comportement, notamment en lui permettant de suivre l'utilisateur. Ceci est un élément crucial pour que le joueur se sent menacé et pour que le jeu soit plus immersif. Nous avons également travaillé sur les animations de l'ennemi pour qu'elles soient fluides et réalistes, contribuant ainsi à renforcer l'expérience de jeu. En résumé, la création d'un ennemi avec une capsule est une étape importante pour garantir une expérience de jeu réussie, et nous avons pris le temps de peaufiner son design et de son comportement pour offrir une expérience de jeu stimulante pour le joueur.

2.3.1 Déplacement

Comme évoqué auparavant, le déplacement de l'ennemi est important dans notre jeu. En effet, pour rendre le gameplay intéressant et dynamique, l'ennemi doit être en mesure de suivre l'utilisateur. Pour cela, nous avons mis en place un algorithme de Dijkstra, une méthode de recherche de chemin optimale dans un graphe, qui permet à l'ennemi de trouver la trajectoire la plus courte pour atteindre la cible. Cela implique que l'ennemi sera capable de naviguer entre les murs présents sur le labyrinthe pour atteindre l'utilisateur, ajoutant ainsi une dimension stratégique au jeu.

Nous avons créé une classe statique nommée "Dijkstra" dans le but d'implémenter l'algorithme de Dijkstra. Cette classe contient une méthode principale qui renvoie le chemin le plus court entre l'ennemi et l'utilisateur. Cette méthode est appelé après chaque déplacement de l'utilisateur afin de mettre à jour sa position. La méthode utilise l'algorithme de Dijkstra [1] suivant :

Algorithm 1 SOURCE-UNIQUE-INITIALISATION(G,s)

```
1: E ← ∅
2: F ← S[G]
3: while F ≠ ∅ do
4:     u ← EXTRAIRE-MIN(F)
5:     E ← E ∪ u
6:     for all vertex v ∈ Adj[u] do
7:         RELÂCHER(u, v, w)
8:     end for
9: end while
```

Ce code représente l'algorithme SOURCE-UNIQUE-INITIALISATION pour trouver le plus court chemin dans un graphe pondéré, à partir d'un nœud source s donné. Les variables E et F représentent respectivement l'ensemble des nœuds visités et l'ensemble des nœuds non visités. La fonction EXTRAIRE-MIN est utilisée

pour extraire le nœud avec la distance la plus courte depuis l'ensemble F. La fonction RELÂCHER met à jour les distances des nœuds adjacents à u si la distance actuelle est plus petite que la distance stockée dans la table de distances S.

L'idée de cet algorithme est de partir d'un sommet initial, la position de l'ennemi, et de calculer les distances les plus courtes à tous les autres sommets du graphe. Pour cela, on attribue une distance initiale infinie à tous les sommets sauf la source, à laquelle on attribue une distance initiale de zéro. Nous avons choisi une distance initiale de $|S| + 1$ pour les cas où la distance initiale est infinie, où $|S|$ représente le nombre de sommets. À chaque étape, on choisit le sommet non visité ayant la plus petite distance connue depuis la source, on le marque comme visité et on met à jour les distances des sommets adjacents non visités en utilisant la distance courante du sommet choisi et les poids des arcs sortant de celui-ci. L'algorithme s'arrête lorsque tous les sommets ont été visités ou que la destination souhaitée a été atteinte.

La complexité de l'algorithme de Dijkstra dépend de la façon dont les sommets sont stockés et triés. En utilisant une liste de sommets non visités et en cherchant le minimum à chaque itération, la complexité est O($|S|^2$) pour un graphe de $|S|$ sommets. Cependant, en utilisant une structure de données appelée tas binaire [1] pour stocker les sommets non visités et pour trouver rapidement le minimum, la complexité de l'algorithme est réduite à $O((|A| + |S|)\ log|S|)$, où $|A|$ est le nombre d'arêtes dans le graphe.

Les tas binaires sont des arbres binaires qui respectent une propriété appelée *tas* ou *heap*. Dans un tas binaire, chaque nœud a une valeur inférieure ou égale à ses enfants (pour un tas binaire minimum, c'est-à-dire le plus petit élément est en racine). Cette propriété permet de trouver rapidement le minimum du tas en O(1) et de supprimer un élément du tas en $O(log(n))$ où n est le nombre d'éléments dans le tas. Lors de l'exécution de l'algorithme de Dijkstra, chaque sommet est ajouté au tas avec sa distance courante comme clé. À chaque étape, le sommet avec la plus petite distance courante est extrait du tas et ses voisins sont mis à jour en insérant ou en diminuant leur distance courante dans le tas.

L'algorithme déployé pour trouvé le plus court chemin entre l'ennemi et l'utilisateur renvoie une table de sommet (ou case pour le labyrinthe) à parcourir pour atteindre le sommet où l'utilisateur se trouve. La méthode de parcourir ce chemin est aussi importante pour une première approche nous pouvons parcourir le chemin suivant un trajectoire rectiligne en un sommet et le suivant.

Afin que la situation soit plus claire, nous proposons de traiter le problème suivant (11). Supposons que nous disposions d'un labyrinthe 4x4 et l'ennemi se trouve à la position 0 et l'utilisateur à la position 11.

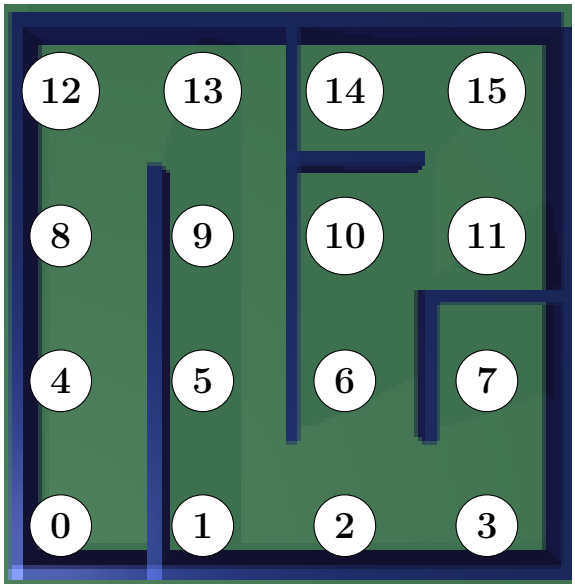


FIGURE 11 : Exemple d'un labyrinthe 4x4

Dans ce cas, l'algorithme de Dijkstra retourne le chamin suivant : (0) → (4) → (8) → (12) → (13) → (9) → (5) → (1) → (2) → (6) → (10) → (11).

Si l'ennemi suit des lignes droites, le mouvement résultant a une trajectoire continue, mais il peut sembler peu réaliste au moment où l'ennemi change de direction [12].

Ce problème survient lorsque la trajectoire n'est pas différentiable en chaque point. Plus précisément, au point de changement de direction, la dérivée à droite diffère de la dérivée à gauche en ce point, ce qui explique le mouvement peu réaliste, sur le plan mécanique, la composante perpendiculaire à la trajectoire de l'accélération ne varie pas d'une manière continue lors de la déviation. Afin d'obtenir un mouvement plus réaliste de l'ennemi, il est donc nécessaire d'améliorer la trajectoire afin satisfaire cette condition de continuité de la dérivée en chaque point.

2.3.2 Amélioration du chemin

Cette partie n'est pas encore implémenté au moment de l'écriture du rapport mais il représente une idée intéressante en matière de développement du chemin de l'utilisateur.

Pour mieux traiter le problème, nous allons nous intéressé qu'aux points de déviation. En effet, nous allons isolé que les trois pour qui définissent la trajectoire au moment du déviation, par exemple les point 8, 12 et 13 de la figure précédente [11].

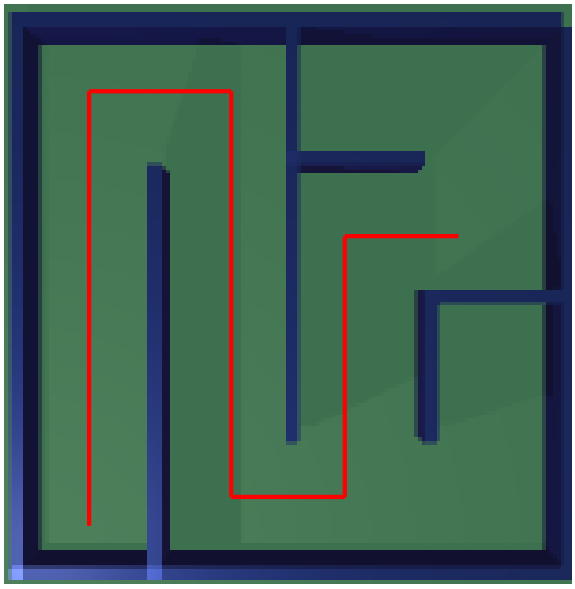


FIGURE 12 : Le chemin entre les positions 0 et 11

Nous allons essayé de crée un fonction d'interpolation entre ces point afin de les reliés par une courbe de classe $\mathcal{C}^1$, une fonction qui représente une portion de cercle peut réalisé la tâche. Mais avant de faire cela, il faut penser au type de trajectoire adéquate à ce type de mouvement. Autrement dit, il faut trouver la trajectoire qu'un objet soumit à une force de déviation parcours.

Pour cette partie, nous allons réduire l'ennemi de masse m à son centre de masse G. En plus, nous allons considéré que l'ennemi est soumit à une force motrice de déviation $\vec{F_m}$ orienté vers le sens de la déviation (l'axe S) comme ci-dessous [13]. La base $(\vec{s}, \vec{t})$ est reliée à un repère fixe $(c_{12}, \vec{s}, \vec{t})$ d'origine confondu avec le centre de la case 12 , qu'on le note c_{12}, $\vec{s}$ est orienté vers le centre de la case 9 et $\vec{t}$ prend la direction du vecteur perpendiculaire à $\vec{s}$ et qui appartient au plan du mouvement.

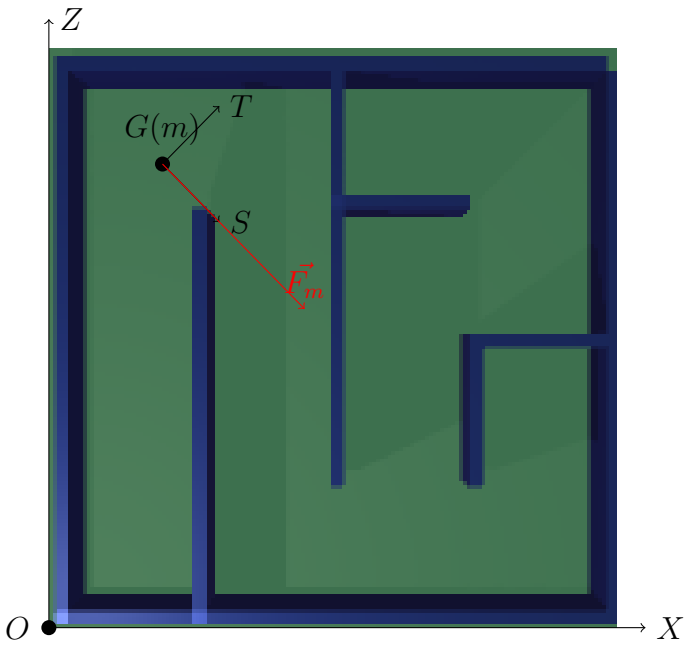


FIGURE 13 : Image avec repère x-z

Nous allons considéré également que l'ennemi ,avant d'entrer en déviation, a été en mouvement avec une vitesse $\vec{v_0}$ parallèle à l'axe Z . En appliquant la deuxième loi de Newton sur ce système nous obtenons :

$$\vec{F_m} + \vec{P} = m \times \vec{a_G} \tag{1}$$

où $\vec{P}$ est le poids de l'ennemi et $\vec{a_G}$ est l'accélération du point G.

Après la projection sur l'axe X et Z nous obtenons :

$$F_m = m \times a_{s_G} \tag{2}$$

$$0 = a_{t_G} \tag{3}$$

En primitivement les deux relations et en utilisant les conditions initiales nous pouvons montrer que les coordonnés s_G et t_G sont relié par l'équation de la trajectoire suivante :

$$s_G = \frac{F_m}{2 \times m} \times (\frac{t_G}{v_0 \times \sin \alpha})^2 \tag{4}$$

où α est l'angle entre l'axe X et l'axe T qui est égale à $\frac{\pi}{4}$ pour ce ces. En général, $\alpha \in \{ \frac{\pi}{4} , \frac{-\pi}{4}, \frac{3\pi}{4} , \frac{-3\pi}{4} \}$.

La relation $s_G = \mathrm{f}(t_G)$ est une fonction polynomiale de degré deux, ce qui implique que la solution optimale pour ce problème est une trajectoire en forme de parabole et non pas une portion de cercle. Il faut mentionner que la solution trouvée n'est

pas adapté au contexte du jeu parce que le repère (c_{12} , $\vec{t}$, $\vec{s}$) ne représente pas la base absolue de l'espace du développement.

La question qu'on peut poser maintenant porte sur la valeur de la force F_m et v_0 parce que si nous trouvons leur valeur nous pouvons avoir l'équation qui décrit le mouvement dans ce repère. La réponse est que nous ne pouvons pas trouver cette force directement mais à ce stade là, nous avons une information très puissante sur la nature du mouvement que nous avons déduit à partir des lois de la physique classique. Une fois que nous avons la nature de la trajectoire avec les trois point à partir de lesquels elle passe nous pouvons trouver facilement cette équation.

L'équation de la trajectoire est de la forme f(t) = s où f(t) = $a.t^2$, d'après l'équation 4, qui est vérifiée par les point c_8, c_{12} et c_{13} les centres des cases 8, 12 et 13 respectivement, de coordonnés respective ($\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}$), (0,0) et ($\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}$) dans le repère ($c_{12}$, $\vec{t}$, $\vec{s}$). D'où, l'équation du mouvement indépendante de la valeur de α est :

$$s = \sqrt{2} \times t^2 \tag{5}$$

Une fois que la trajectoire optimale et la fonction d'interpolation entre ces trois points est déterminée, nous pouvons alors passer à sa conception en cherchant une fonction paramétrée qui relie x_G et z_G à un paramètre, tout en garantissant que la trajectoire reste la même.

Pour cela nous allons considérer un point M de la trajectoire, M est donc de cordonnés (t_M, f(t_M)) dans la repère (c_{12}, $\vec{t}$, $\vec{s}$). Pour cette déviation en particulier, $\vec{t}$ = $\frac{\vec{x}+\vec{z}}{\sqrt{2}}$ et $\vec{s} = \frac{\vec{x}-\vec{z}}{\sqrt{2}}$. afin de préciser les coordonnés du point M dans la base absolue nous allons nous intéressé au cordonnées du vecteur $\vec{OM}$. On a :

$$\vec{OM} = \vec{OC}_{12} + \vec{C_{12}M} = \vec{OC}_{12} + t_M \times \vec{t} + f(t_M) \times \vec{s} \tag{6}$$

En remplaçant les vecteurs $\vec{t}$ et $\vec{s}$ par leurs coordonnés dans la base absolue, nous obtenons :

$$\vec{OM} = \vec{OC}_{12} + \frac{t_M + f(t_M)}{\sqrt{2}}.\vec{x} + \frac{t_M - f(t_M)}{\sqrt{2}}.\vec{z} \tag{7}$$

Ainsi, pour parcourir la courbe parabolique d'équation $s = \sqrt{2} \times t^2$ dans le repère absolu, il est simplement nécessaire de faire varier t dans l'intervalle $\left[-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right]$. En d'autres termes, nous avons trouvé une courbe paramétrée $\gamma(t)$ qui associe chaque valeur de t à un point de la courbe parabolique à parcourir de manière à ce que :

$$\gamma\colon \left[-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right] \to \mathbb{R}^2,\ t \mapsto \left(x_{c_{12}} + \frac{t + f(t)}{\sqrt{2}}, z_{c_{12}} + \frac{t - f(t)}{\sqrt{2}}\right)$$

Passons maintenant au cas générale où le centre de déviation est un point quelconque

qu'on note c_i avec un angle $\alpha = (\widehat{\vec{s}, \vec{x}}) \in \{\frac{\pi}{4}, \frac{-\pi}{4}, \frac{3\pi}{4}, \frac{-3\pi}{4}\}$

En utilisant le même raisonnement précèdent, nous pouvons démontrer que la courbe paramétrée générale à toutes les déviations γ_α est de la forme :

$$\gamma_\alpha : \left[-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right] \to \mathbb{R}^2,\ t \mapsto \left(x_{c_i} + \frac{t\cos\alpha + f(t)\sin\alpha}{\sqrt{2}}, z_{c_i} + \frac{t\sin\alpha - f(t)\cos\alpha}{\sqrt{2}}\right)$$

Grâce à cette relation, nous pouvons parcourir la courbe en utilisant le paramètre t pour toutes les déviations possibles et nous pouvons détecté la présence d'une déviation lorsque la croissance de numéro attribué à la case varie.

Après l'étude du chemin de l'ennemi vers l'utilisateur, nous pouvons passer à l'étape suivante, qui consiste à traiter l'interaction entre l'ennemi et l'utilisateur lorsqu'ils sont visibles l'un par l'autre.

2.3.3 Interaction entre l'ennemi et l'utilisateur

Lorsqu'un ennemi se trouve en face de l'utilisateur sans aucun obstacle entre eux, il peut attaquer l'utilisateur et diminuer son score. Cependant, l'utilisateur dispose d'une arme qu'il peut utiliser pour tirer sur l'ennemi et le vaincre. Afin de rendre le jeu plus stimulant, l'utilisateur peut abattre l'ennemi en utilisant seulement trois tirs. Si l'utilisateur réussit avant que son score n'atteigne zéro, son score augmente et un nouvel ennemi apparaîtra dans une position différente pour continuer le défi. Lorsque l'utilisateur atteint un score de 1000 points, un objet apparaît dans la scène qui symbolise la clé d'évasion. Le but principal du jeu est de collecter cet objet.

Lorsque le joueur tire sur l'ennemi, le système du jeu détecte le tir ,à l'aide du Raycast, et calcule les dégâts infligés à ce dernier. Si le collider de l'ennemi est touché par le rayon Raycast, il subit des dégâts qui affectent son score.

Pour déclencher cette action, une fonction "Damaged" est appelée en cemoment. Cette fonction est définie dans le script attaché à l'ennemi, et elle est appelée en utilisant le SendMessage de Unity.

SendMessage est une méthode dans Unity qui permet à un objet de transmettre un message à tous les scripts attachés aux autres objets dans la scène. Lorsqu'un SendMessage est appelé, Unity va parcourir l'ensemble de la hiérarchie des objets pour rechercher les scripts qui répondent au nom de la méthode appelée.

La fonction "Damaged()" peut ensuite mettre à jour les statistiques de l'ennemi, telle que son score, en fonction des dégâts subis. Si le score de l'ennemi est égale zéro ou en dessous, l'ennemi peut être détruit à l'aide de la fonction "Destroy()".

La fonction "Destroy" dans Unity est une méthode qui permet de détruire un objet de la scène. Elle peut être utilisée pour supprimer des objets qui ne sont plus

nécessaires, tels que des ennemis vaincus, des objets collectibles ou des éléments de l'interface utilisateur.

La méthode "Destroy" peut être appelée à partir de n'importe quel script attaché à un objet de la scène. Lorsqu'elle est appelée, l'objet est supprimé de la scène, ce qui signifie qu'il ne peut plus être vu ni manipulé par les joueurs ou les autres scripts.

L'utilisateur dispose également de la fonction "Damaged()", qui diminue son score s'il est attaqué par un ennemi. Si son score atteint zéro, l'écran de fin de partie est affiché.

2.4 Prévisions et différences

Lors de la première réunion avec mes tuteurs, nous avons pu établir une liste d'objectifs , correspondante à un volume horaire de 60 heures, qui a donné lieu au diagramme de Gantt prévisionnel suivant :

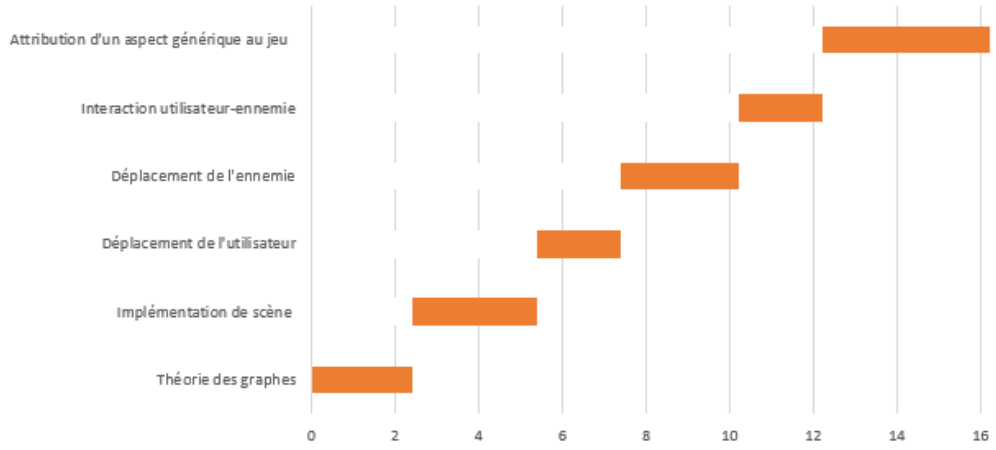


FIGURE 14 : Diagramme de Gantt prévisionnel

Grâce à cette planification, nous disposions d'un temps suffisant pour assimiler les différentes aspects théoriques du jeu à créer et pour concevoir une scène de base de bonne qualité.

Le diagramme final a été différent pour plusieurs raisons, en raison de la diversité des solutions possibles, qui a nécessité des tests pour trouver la solution la plus appropriée. En plus, en cours de développement du projet quelque objectifs ont été modifié pour améliorer l'expérience du jeu.

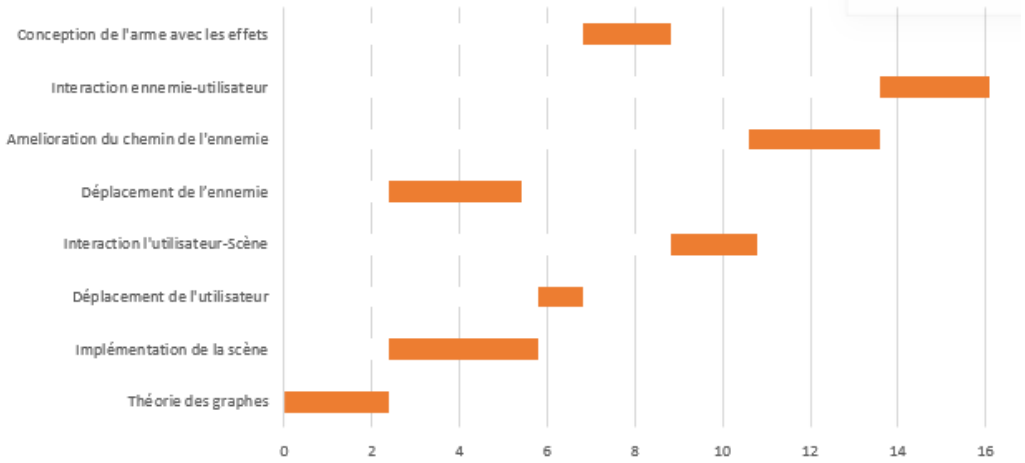


FIGURE 15 : Diagramme de Gantt réel

3 Résultats

À la fin de ce projet, nous disposons d'un jeu immersif qui comprend une scène, un utilisateur et un ennemi. Lorsque l'utilisateur appuie sur le bouton "play" de l'écran d'accueil, le jeu se génère avec ces trois parties.

La scène peut être créée en fonction du niveau de difficulté choisi. Pour une difficulté faible, nous utilisons l'algorithme de Kruskal deux fois pour créer une scène qui comporte moins de murs. Pour une difficulté moyenne, la scène est créée en utilisant l'algorithme du backtracking, qui génère un labyrinthe connexe. Pour une difficulté élevée, nous utilisons l'algorithme de Kruskal une fois pour créer un labyrinthe avec une unicité de chemin entre deux cases quelconques.

Pour l'utilisateur, le jeu permet de se déplacer dans les directions de la caméra avec une latence réduite grâce à la gestion des collisions qui utilise le graphe de la scène. Il permet également à l'utilisateur de viser et de tirer sur les objets environnants avec un effet visuel de tir, offrant ainsi un moyen d'interaction avec l'ennemi.

Pour l'ennemi, il est capable de suivre l'utilisateur en utilisant l'algorithme de Dijkstra et de suivre une trajectoire parabolique lorsqu'il doit dévier, ce qui rend ses déplacements très réalistes. Il est aussi capable de porter du damage sur l'utilisateur près l'atteindre.

Au début du jeu, l'utilisateur commence avec un score de 100 points et chaque ennemi a une santé de 60 points. Lorsqu'un ennemi est éliminé, le joueur gagne 20 points et un nouvel ennemi apparaît. Pour gagner la partie, le joueur doit atteindre un score de 1000 points et récupérer la clé, ce qui déclenche l'affichage d'un bouton de rejouer pour relancer le jeu. En cas de score nul, un écran de fin de partie s'affiche.

4 Conclusion

Le projet THE MAZE RUNNER est un projet de création d'un jeu de survie immersive. Il consiste à immerger l'utilisateur ,par le biais d'une caméra à la première personne, dans un labyrinthe conçue de tel sort qu'elle crée un obstacle devant ses mission. La mission principale du joueur est de collecter un score précis à fin de quitter cette endroit ce qu'il amène à essaye d'abattre ses ennemis puisque c'est son seul moyen de sortir. Les ennemis sont moins puissant que l'utilisateur mais il peut lui créer des problème à cause de leurs capacité de le suivre.

Lors du développement du projet THE MAZE RUNNER, plusieurs difficultés techniques ont été rencontrées en utilisant Unity comme la plateforme de développement. En effet, elle utilise un seule système de coordonnés qui crée une des principales difficultés en matière de changement de base. En effet, Unity utilise un système de coordonnées droitier (x, y, z) tandis que la caméra de notre jeu devait être définie en coordonnées gaucher (x, y, -z). Ce changement a posé des problèmes pour la gestion des mouvements de la caméra.

Un autre problème rencontré a été la trajectoire des ennemis. Pour créer un effet réaliste de poursuite, les ennemis devaient être capables de suivre le joueur à travers le labyrinthe. Cependant, la gestion de la trajectoire des ennemis a été complexe en raison de la présence de nombreux obstacles dans le labyrinthe et de la présentation de la courbe qui ne reste plus une fonction dans la base absolue mais une courbe paramétrée. Nous avons finalement opté pour un système de trajectoire en utilisant les coordonnées de l'ennemi dan un repère où la trajectoire peut se représenter par une fonction et de modifier son expression afin d'utiliser le système coordonnées absolue.

En outre, Unity est une plateforme de développement très riche en fonctionnalités, mais elle peut être difficile à maîtriser. La création d'un jeu immersif comme THE MAZE RUNNER nécessite une connaissance approfondie de la plateforme pour optimiser les performances ,en évitant les solutions données par la plateforme s'il sont très coûteuses, et obtenir un résultat final de qualité.

Malgré ces difficultés, le projet THE MAZE RUNNER a permis d'explorer les possibilités offertes par Unity pour la création de jeux immersifs. Des améliorations pourraient être apportées en ajoutant des effets sonores pour renforcer l'immersion du joueur ainsi qu'une meilleurs exploitation des systèmes des particules pour créer des effets visuels intéressantes. En fin de compte, le développement de ce projet m'a donné l'occasion de découvrir les différentes étapes de la création d'un jeu interactif et peut constituer une première étape vers des projets de plus grande envergure dans l'avenir.

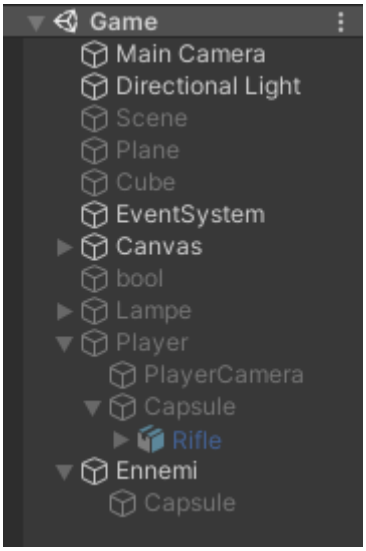
Références

[1] Thomas H Cormen, Charles Eric Leiserson, Ronald L Rivest, Philippe Chrétienne, and Xavier Cazin. *Introduction à l'algorithmique*, volume 6. Dunod, 1994. https://www.mcours.net/cours/pdf/hasclic3/hassclic995.pdf.

[2] Unity Technologies. Documantation sur les systèmes de particules, 2022. https://docs.unity3d.com/ScriptReference/ParticleSystem.html,Consulté le 1 mars 2023.

[3] Unity Technologies. Documentation sur les textures, 2022. https://docs.unity3d.com/Manual/Textures.html , Consulté le 26 février 2023.

Annexes



hiérarchie des objets

```
void printScene(Graph g)
{
    createPlan(height, width);
    for (int i = 0; i < height * width; i++)
    {
        foreach (Arc frere in g.sommets[i].brothers)
        {
            if ((!frere.trueBrother) && (frere.numVertex == (g.sommets[i].number + 1)) )
            {
                putCube(_Wall, (i / height), (i % width) + 1, 1.0f, 0.1f);
            }
            if ((!frere.trueBrother) && (frere.numVertex == (g.sommets[i].number + width))  )
            {
                putCube(cube: _Wall, (i / height) + 1, i % width, 0.1f, 1.0f);
            }
        }
        if(g.sommets[i].contientBool )
        {
            GameObject newBool = Instantiate(_bool,offset + new Vector3( (float) ((int)i/width)  , 0.5f,
                (float) i%width), Quaternion.Euler(0.0f, 0.0f, 0.0f));

        }
    }

    for(int i = 0; i < height; i++)
    {
        putCube(_Wall, i, 0, 1.0f, 0.1f);
        putCube(_Wall, i, width, 1.0f, 0.1f);

    }

    for (int i = 0; i < width; i++)
    {
        putCube(_Wall, 0, i, 0.1f, 1.0f);
        putCube(_Wall, height, i, 0.1f, 1.0f);

    }
    CreatPersonnage();
    CreatHunter();
    _Score.SetActive(true);
}
```

Créateur de la scène

```
private void goInDirection(Vector3 direction)
{
    Collider[] colliders = Physics.OverlapSphere(transform.position + direction, CircelRaduis);

    bool seDeplacer = true;

    foreach (Collider collider in colliders)
    {
        if (collider.gameObject.name != "Capsule" && collider.gameObject.name != "Rifle")
        {
            Debug.Log("Detected " + collider.gameObject.name + " within the sphere");
            seDeplacer = false;
        }
    }
    if (seDeplacer)
    {
        transform.position = transform.position + direction;
    }
}
```

Mouvement dans la direction de la caméra

```
private Vector3[] ChangementDeBase(float Rotation)
{
    Vector3[] Mat_B_Bc = new Vector3[3];

    Mat_B_Bc[0] = new Vector3( Mathf.Cos(Rotation * Mathf.Deg2Rad), 0, Mathf.Sin(Rotation * Mathf.Deg2Rad) );
    Mat_B_Bc[1] = new Vector3(0, 1, 0);
    Mat_B_Bc[2] = new Vector3(-Mathf.Sin(Rotation * Mathf.Deg2Rad), 0, Mathf.Cos(Rotation * Mathf.Deg2Rad));

    return Mat_B_Bc;
}
3 références
private float Mul_vect_vect(Vector3 vect1, Vector3 vect2)
{
    return vect1.x * vect2.x + vect1.y * vect2.y + vect1.z * vect2.z;
}
4 références
private Vector3 Mult_Mat_Vect(Vector3[] Mat, Vector3 vec)
{
    Vector3 result = new Vector3( Mul_vect_vect( Mat[0],vec), Mul_vect_vect(Mat[1], vec), Mul_vect_vect(Mat[2], vec));
    return result;
}
```

Fonction qui traite les changement de base