INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE

Rapport de PAO

Développement d'un démonstrateur 3D sous Unity

Ambre DE CRESCENZO
Ryan JUTEAU
Marine RICHARD

A l'intention de
Maxime GUERIAU

Table des matières

1 Introduction

Ce projet est la suite du PAO du semestre précédent.

Le but du projet est le suivant : créer un jeu vidéo avec une IA qui utilise une technique d'apprentissage par renforcement.

Le projet sera effectué sur Unity, un logiciel qui nous permettra d'utiliser un environnement en 3D.

Pour atteindre notre objectif, nous allons devoir apprendre à coder en C#, comprendre comment utiliser Unity ainsi qu'une technique d'apprentissage par renforcement.

Dans ce rapport seront présentés les spécifications du projet, le journal de bord, les outils utilisés, la technique d'apprentissage par renforcement, ainsi que les problèmes et améliorations possibles du projet.

2 Spécifications

Le projet consiste en un jeu à la première personne. Le joueur se retrouvera dans un labyrinthe et son but sera de ne pas se faire tuer par un golem qui est notre IA.

2.1 Liste des fonctionnalités prévues

— Coder un algorithme de Q-learning pour le golem : le but est que le golem apprenne des actions du joueur pour pouvoir le trouver le plus rapidement possible.

— Séparer complètement la partie Q-learning de l'environnement, pour que Unity soit juste de la gestion d'environnement : pour cela, il y aura une personne qui sera entièrement dédiée à la partie Q-learning et une autre personne entièrement dédiée à la partie environnement.

— Créer un labyrinthe en 3D dans lequel le joueur et le golem évolueront : il faut que le joueur puisse se cacher et que le golem apprenne comment agir dans le cas où il voit le joueur ou non.

— Coder les actions du golem : il doit pouvoir explorer son environnement, voir que le joueur est proche de lui, lui infliger des dégâts et tuer le joueur pour mettre fin à la partie.

2.2 Objectifs finaux

— Séparer complètement la partie Q-learning de l'environnement, faire en sorte que Unity soit juste de la gestion d'environnement

— Réussir à ne pas faire du machine learning forcé et brut, c'est à dire n'avoir que 10 épisodes pour que l'agent apprenne.

3 Journal de Bord

3.1 Première semaine - 26/09

- **27/09 :** Réunion
- **Fait :**
  - La répartition des rôles de chacun
  - Le choix de la méthode d'apprentissage par renforcement : le Q-learning
  - Le choix de la date de la soutenance
  - Le choix des spécifications en fonction de ce qui a été fait le semestre précédent
  - Setup du premier GIT
  - Installation de Unity
  - Rédaction du brouillon de planning prévisionnel
  - Importation des assets
  - Début de tutoriels
- **A faire :**
  - Suivre des tutos "débutants" Unity du style : unity-tutorials-new-game-developers
  - Faire du reverse engineering sur les codes des Pauls (comprendre ce qu'ils ont fait et comment, et pourquoi)
  - Créer le gitlab (juste pour les scripts)
  - Créer/récupérer le dossier Nuage (pour les sauvegardes des scènes et des prefabs)
  - Commencer une feuille de route (planning avec semaines) pour essayer d'organiser semaine/semaine
  - Lire le rapport du projet du semestre précédente
  - Comprendre ce qu'est le Q-learning

3.2 Deuxième semaine - 03/10

- **04/10 :** Réunion
- **Fait :**
  - Présentation de ce qui a été compris du Q-learning
  - tuto microgames FPS avec altération de scripts et Lego OK
  - Début de recherches pour mieux coder les portes dans la scène.
- **A faire :**
  - Pour Ambre : Coder un problème qui utilise du Q-learning
  - Trouver un moyen d'utiliser Nuage de façon opti
  - Avancer voire terminer le tuto Unity de base.
  - Essayer de comprendre les scripts déjà codés.

3.3 Troisième semaine - 10/10

- **11/10 :** Réunion
- **Fait :**
  - Présentation de la V1 d'un problème qui utilise le Q-learning en Python
  - BoiteNoireV1.2 en Python. Tests avec des portes presque fonctionnelles. Apprentissage du système de collision.
- **A faire :**
  - Pour Ambre : Reprendre le code du Q-learning pour qu'il fonctionne, documenter le code, présenter une démonstration, gagner en abstraction au niveau du code
  - Gérer les portes

3.4 Quatrième semaine - 17/10

- **18/10 :** Réunion
- **Fait :**
  - Présentation de la V2 d'un problème qui utilise le Q-learning et qui fonctionne en Python
  - Tests avec les différentes vues car impossible de voir le GameObject porte, optimisations.
- **A faire :**
  - **Général :** Gérer les préfabs finales/bac à sable.
  - **Général :** Séparer environnement et QLearning.
  - **Game Design :** Décider du comportement Monstre/Joueur. Prendre les décisions définitives de GameDesign. Faire un labyrinthe cheum pour le monstre qui poursuit le joueur et montrer proprement l'apprentissage.
  - **Environnement :** Gérer et re-manager tous les objets de l'environnement avec lesquels le joueur interagit (verrou, porte ...)
  - **Environnement :** Gestion d'inventaire, tester les clefs, faire en sorte que les portes tournent sur les gonds, gérer les collider et trigger
  - **Environnement :** Faire en sorte de forcer au moins une première fois la rencontre entre le joueur et le monstre pour que le monstre commence à apprendre.
  - **QLearning :** Sauvegarder la QTable et plot les steps du dernier épisode où il est en full apprentissage (0 et 25000) pour voir ce qu'il apprend.
  - **QLearning :** Présentation avec des slides d'Ambre pour la partie QLearning

3.5 Cinquième semaine - 24/10

- **26/10 :** Réunion
- **Fait :**
  - Présentation par Ambre de la Qtable
  - Création du prefab PorteV3 qui est utilisable
- **A faire :**
  - Coder une Q-table sur Unity
  - Faire la liste des états et des actions, dans la limite d'une Q-table de 100 cases
  - Gagner en abstraction au niveau du code

3.6 Sixième semaine - 31/10

- **Vacances**
- **Fait :**
  - Création des classes Enemy, Food, Player sur Unity pour re-créer la même situation sur Python que sur Unity
  - Compréhension du C# grâce aux tutos Unity
  - Création de la liste des états et des actions, dans la limite d'une Q-table de 100 cases
- **A faire :**
  - Coder une Q-table sur Unity
  - Gagner en abstraction au niveau du code

3.7 Septième semaine - 07/11

- **08/11 :** Réunion
- **Fait :**
  - Remaniement de la liste des états et des actions, dans la limite d'une Q-table de 100 cases
  - Création de la fonction qui met à jour la Q-table
  - Début GameMaster
- **A faire :**
  - Coder une Q-table sur Unity
  - Gagner en abstraction au niveau du code

3.8 Huitième semaine - 14/11

- **Fait :**
  - Écriture des classes états et actions définitives
  - Création des classes états, actions sur Unity
  - Changement des clés de la Q-table qui deviennent des énumérations.
  - Code qui compile mais logique pas encore bonne pour le gameMaster, Joueur et Monstre
- **A faire :**
  - Coder une Q-table sur Unity
  - Gagner en abstraction au niveau du code

3.9 Neuvième semaine - 21/11

- **Fait :**
  - Liaison entre les scripts pour lancer la V1 de la Q-table.
  - Création des actions du Blob dans Unity
  - Rédaction des ordres. Le monstre peut suivre le joueur grâce au AI.NavMesh.
- **A faire :**
  - Coder une Q-table sur Unity
  - Gagner en abstraction au niveau du code

3.10 Dixième semaine - 28/11

- **Fait :**
  - V2 de la Q-table qui compile et qui n'a pas d'erreurs lorsqu'elle fonctionne
  - Création des actions du Blob dans Unity
  - Champ de vision V1 du monstre et début des fonctions procédures avec les dégâts.
- **A faire :**
  - Q-table qui ne se met à jour que lorsque le Blob a fini de faire son action
  - Adapter les états et actions à notre jeu et non plus à l'exemple jouet
  - Gagner en abstraction au niveau du code

3.11 Onzième semaine - 05/12

- **Fait :**
  - V3 de la Q-table qui fonctionne sur l'exemple jouet
  - Test sur plusieurs états différents pour voir si on peut diminuer la taille de la Q-table et s'il y a toujours convergence
  - Synchronisation entre la Q-table et les actions du Blob qui se font à la suite et non plus à chaque frame
  - Test sur les états et actions définis pour notre jeu pour vérifier la convergence
  - Abstraction de la fonction mise à jour de la Q-table
  - Dégâts qui fonctionnent. GameOver. Optimisations du NavMesh. Collisions du Monstre. Prefab Monstre. Début de vision V2. Son pour le monstre.
- **A faire :**
  - Écriture du rapport
  - Rendu des livrables

3.12 Douzième semaine - 12/12

- **15/12 :** Réunion
- **Fait :**
  - Soutenance : démonstration de ce qui a été fait au niveau du Q-learning, des actions du golem

4 Q-learning

4.1 Définition

Le Q-learning est une méthode d'apprentissage par renforcement. D'après la définition donnée par Wikipédia, l'apprentissage par renforcement consiste en un **agent** autonome qui doit apprendre des **actions** à effectuer, à partir d'expériences, à **un état** donné, de façon à optimiser **une récompense** quantitative au cours du temps.

On doit donc définir ce que sont nos **actions**, nos **états** et aussi les **récompenses** obtenues par l'**agent**.

4.2 Actions et états

Il n'existe pas de contraintes concernant le nombre d'actions et d'états. Si l'environnement n'impose également aucune action ou état particulier, le choix des actions et des états est à la liberté du développeur.

La seule contrainte à respecter est que pour passer d'un **état s** à un **état s+1**, l'**agent** doit effectuer une **action**.

4.3 Récompenses

Une **récompense** doit être attribué à **un état** en particulier. Plus une récompense est négative, plus l'agent comprendra que l'action qu'il a effectuée est mauvaise. Au contraire, plus une récompense est positive, plus l'agent comprendra que c'est la meilleure action à faire. Il est donc important de bien définir ses récompenses, auquel cas l'agent peut ne rien apprendre de ses actions.

4.4 La Q-table

La Q-table est l'outil qui va permettre à notre agent d'apprendre.

La Q-table se définit comme un tableau. Le nombre de colonnes est celui du nombre d'**actions** et le nombre de lignes celui des**états**.

Elle est initialisée de manière aléatoire.

C'est grâce aux **actions** et aux **états** de l'**agent** que les valeurs de la Q-table vont être définies.

Il est importante de vérifier, lors de la création de la Q-table, que les états et actions choisis permettent à la fonction de converger. Sans la convergence, l'agent ne pourra jamais atteindre l'état final.

4.5 Mise à jour de la Q-table

```
initialiser Q[s, a] pour tout état s, toute action a de façon arbitraire, mais Q(état terminal, a) =
0 pour toute action a

répéter
      //début d'un épisode
      initialiser l'état s

      répéter
               //étape d'un épisode
               choisir une action a depuis s en utilisant la politique spécifiée par Q (par exemple
ε-greedy)
               exécuter l'action a
               observer la récompense r et l'état s'

               Q[s, a] := Q[s, a] + α[r + γ max_a' Q(s', a') - Q(s, a)]

               s := s'
               a := a'
      jusqu'à ce que s soit l'état terminal
```

FIGURE 1 – Pseudo code de la mise à jour de la Q-table

Voici le pseudo code de la mise à jour de la Q-table.

Ici, on initialise la Q-table avec la valeur 0. Ce choix est arbitraire.

On va ensuite répéter les actions suivantes jusqu'à ce qu'on arrive à l'état final :

- Choix de la **meilleure action** par l'**agent** en fonction de son **état s** et de la politique spécifiée par Q.
- Exécution de l'**action** par l'**agent**.
- Calcul d' **une récompense**.
- Calcul du **nouvel état** et de la **nouvelle meilleure action**.
- Mise à jour de la valeur de la Q-table pour son ancien état et son ancienne action.

On peut voir dans la formule de mise à jour de la valeur de la Q-table plusieurs constantes : α et γ.

4.5.1 Le facteur d'apprentissage : alpha

α est le facteur d'apprentissage. C'est une valeur comprise entre 0 et 1.

Plus α a une valeur proche de 1, plus l'agent prendra en compte sa dernière action. Plus α a une valeur proche de 0 et moins l'agent apprendra.

4.5.2 Le facteur d'actualisation : gamma

γ est le facteur d'actualisation. C'est une valeur comprise entre 0 et 1.

Plus γ a une valeur proche de 1, plus l'agent prendra en compte les récompenses lointaines. Plus γ a une valeur proche de 0 et plus l'agent prendra en compte les récompenses proches.

4.6 Exemple

L'exemple qui va être expliqué ci-dessous se retrouve dans le fichier Q-table.ipynb dans les annexes.

4.6.1 But du jeu

Soit un plateau de jeu de dimension 5x5. Soit un agent, de la nourriture et un ennemi positionnés de manière aléatoire sur le plateau.

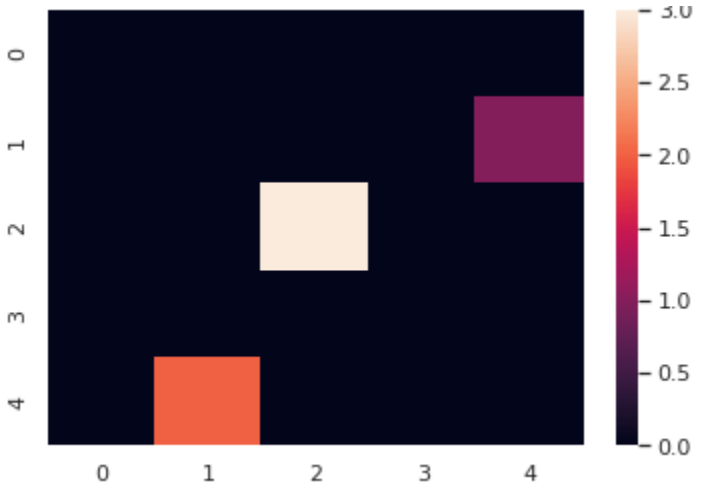

FIGURE 2 – Illustration du plateau de jeu avec en violet l'agent, en orange l'ennemi et en blanc la nourriture

Le but de l'agent est de trouver la nourriture et de ne pas trouver l'ennemi. La politique utilisée pour la Q-table est ϵ -greedy.

4.6.2 Définition des actions et des états

Ici, les **actions** de l'agent seront le déplacement dans 4 directions possibles : haut, bas, droite et gauche.

Les **états** seront la position relative de la nourriture et de l'ennemi par rapport à l'agent.

Les actions et états sont cohérents car pour passer d'un état s à un état s+1, il faut bien effectuer une des quatre actions de l'agent.

4.6.3 Définition des récompenses

Ici, il existe 3 types de récompenses :

— La récompense si l'agent atteint la nourriture.

— La récompense si l'agent atteint l'ennemi.

— La récompense si l'agent n'atteint rien.

4.6.4 Principe du epsilon -greedy

Le ϵ -greedy définit 2 modes de fonctionnement de l'agent au moment de choisir la prochaine action à choisir :

— Exploitation : l'agent va effectuer l'action qui sera définie comme la meilleure selon la Q-table.

— Exploration : l'agent va effectuer une action aléatoire sans prendre en compte les valeurs de la Q-table.

Normalement, l'agent est en mode exploitation. Le choix de basculer en mode exploration se fait à l'aide d'une variable aléatoire et de la constante ϵ fixée au préalable entre 0 et 1.

Si la variable aléatoire est supérieure à ϵ, on ajoute à ϵ la constante $\Delta\epsilon$, définie également au préalable et l'agent passe en exploration.

```
generator_random = np.random.default_rng() #Initialisation de la seed
random_action = generator_random.random() #Calcul de random_action pour savoir si on effectue une action au hasard
if (random_action > epsylon):#Si l'agent fait une action au pif
  epsylon += epsylon_delta #On ajoute epsylon_delta pour que l'agent fasse de moins en moins d'actions au hasard
  action = generator_random.integers(0,3)#On choisit parmi les 4 actions une action
else:#L'action est déterminée avec la Q-table
```

FIGURE 3 – Principe du ϵ-greedy en Python

Cette politique permet entre autre de forcer l'IA à explorer des choix qui sembleraient de prime abord peu rentable, par exemple s'il y a de l'aléatoire dans l'obtention de récompense.

4.6.5 Application

Appliquons notre algorithme à notre jeu.

Initialisons notre Q-table dans un premier temps. Ici, le choix a été fait de n'avoir que des valeurs négatives.

```
def Q_table_initialisation():
  generator_random = np.random.default_rng() #Initialisation de la seed
  Q_table = {} #Initialisation du dictionnaire Q-table
  for x1 in range(-n+1,n): #Parcours des clés de Q-table
    for y1 in range(-p+1,p):
      for x2 in range(-n+1,n):
        for y2 in range(-p+1,p):
          for action in range(4):
              Q_table[(x1,y1),(x2,y2),action] = np.random.uniform(-5,0)
              #Valeur aléatoire attribuée à chaque clé de Q-table
  return Q_table
Q_table =  Q_table_initialisation() #Initialisation de Q-table
print(Q_table)#Vérification des valeurs de Q-table
```

```
{((-4, -4), (-4, -4), 0): -1.4263558991949776, ((-4, -4), (-4, -4), 1): -0.3602424967642657,
```

FIGURE 4 – Initialisation de la Q-table

Initialisons les constantes suivantes :

- $\epsilon = 0.9$
- $\alpha = 0.9$
- $\gamma = 0.9$
- Récompense pour avoir atteint la nourriture = 0
- Récompense pour avoir atteint l'ennemi = -1000
- Récompense pour n'avoir rien atteint = -1

On définit l'état final comme étant soit l'agent ayant atteint la nourriture ou soit l'agent ayant atteint l'ennemi.

On regarde si l'agent est en mode exploration ou exploitation. Dans notre exemple, on obtient $randomAction = 0.4613002406049078$. Comme randomAction est inférieur à ϵ, l'agent est en mode exploitation.

On va donc chercher les actions possibles à l'état courant. Soit l'état courant définit par $obsFood = (-2, -1)$ et $obsEnemy = (-3, -3)$. Dans la Q-table, on va donc regarder à l'état(-2,-1)(-3,-3) quelle action a la plus grande valeur.

Imaginons par exemple que nous avons les valeurs ci-dessous.

Droite	Gauche	Haut	Bas
-2.6	-0.5	-4.3	-1.2

— La plus grande valeur est attribuée à l'action gauche. L'agent va donc bouger à gauche.
— Le nouvel état de l'agent est (-1,-1)(-2,-3).
— On va ensuite calculer la récompense. Comme l'agent n'a rien atteint, la récompense est -1.
— On met à jour la valeur de la Q-table pour l'état (-2,-1)(-3,-3) et on recommence jusqu'à atteindre l'état final.
— On peut vérifier si l'agent est plus efficace en regardant la récompense moyenne qu'il obtient.

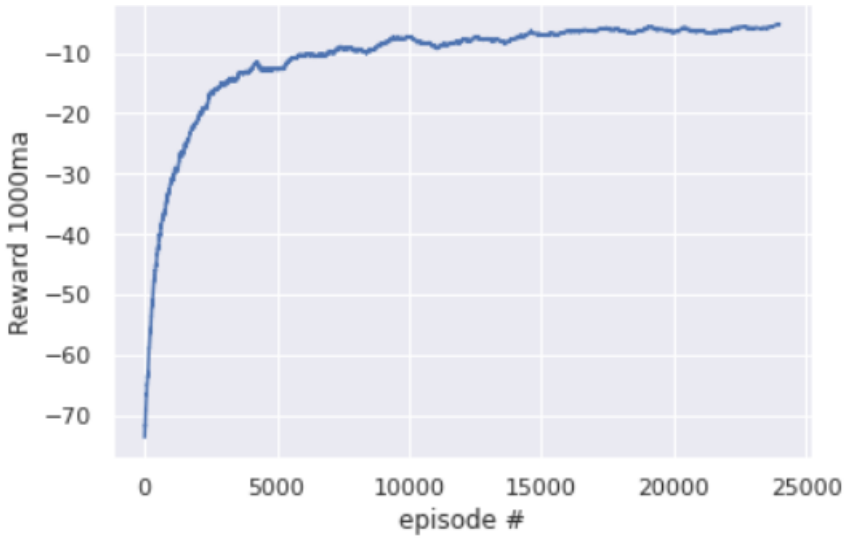

FIGURE 5 – Graphe des récompenses moyennes obtenues sur 25000 épisodes

On peut donc en déduire si notre Q-table est fonctionnelle et efficace. Si la récompense est grande, alors ça veut dire que l'agent apprend de ses erreurs. Si le nombre d'épisodes est petit, ça veut dire que l'agent apprend vite.

4.7 Application dans Unity

Nous n'avons malheureusement pas eu le temps de relier l'algorithme de la Q-table au golem ; cependant, nous avons pu tester si la fonction que nous proposions convergeait ou non.

L'agent est notre golem.

Nous avons choisi les états suivants :

- Ne voit pas le joueur/N'a pas infligé de dégâts
- Voit le joueur
- A infligé des dégâts au joueur
- Suit le joueur
- Suit la dernière position du joueur
- A portée du joueur
- A tué le joueur

Nous avons choisi les actions suivantes :

- Attendre
- Explorer la carte
- Suivre le joueur
- Attaquer

L'action suivre le joueur n'est possible que si le golem voit le joueur.

Nous avons choisi les récompenses suivantes :

- Ne voit pas le joueur/N'a pas infligé de dégâts = -20
- Voit le joueur = -10
- A infligé des dégâts au joueur = -5
- A tué le joueur = 0

Une des contraintes qui s'est posée lors du choix des actions et états est la taille de la Q-table qui ne devait pas être trop grande. Autrement, cela aurait demandé beaucoup de puissance de calcul et aurait potentiellement rendu le jeu moins fluide.

En testant sur un exemple simplifié de notre jeu, avec un joueur statique et notre agent qui peut se déplacer, nous avons observé que tant que l'agent ne voit pas le joueur, il n'apprend rien.

Il serait donc intéressant d'initialiser au préalable la Q-table avec une situation où le golem verrait forcément le joueur et apprendrait ensuite qu'il faut tuer le joueur.

Un autre souci qui s'est posé est la synchronisation entre les mouvements du golem et la mise à jour de la Q-table. Nous avons dû mettre des fonctions timers pour laisser le golem faire son action avant qu'il n'en choisisse une autre. Sinon, le golem n'effectue chaque action que le temps d'une frame, ce qui ne lui permet pas d'apprendre.

5 Environnement

5.1 Fonctionnalités pour le monde

5.1.1 Etat

Cette classe est très simple, elle sert à référencer deux énumérés avec les noms des actions et états de la QTable.

5.1.2 GameMaster

Le GameMaster est le gameObject qui gère la liaison entre la QTable et les états et actions du monstre. A la base, le GameMaster devait constituer une structure avec les différents états et informations du joueur et monstre. Cependant, la QTable n'interagit qu'avec le monstre. Cette structure est donc présente mais non utilisée. Le GameMaster gère par contre le Game Over quand le joueur n'a plus de points de vie. Enfin, c'est lui qui a le dictionnaire des récompenses en fonction de l'état qui permet d'optimiser la QTable.

5.1.3 Portes

Les portes n'étaient pas encore parfaitement dynamiques. La première fonction permettait de rendre la porte "molle" telle une ragdoll, mais toujours retenue par des charnières. Grâce à un tutoriel trouvé sur internet, la PorteV2 permettait d'effectuer une translation du gameObject en appuyant sur une touche du clavier et en affichant un prompt à l'écran. La PorteV3 développée ensuite permet de relier une clé à une porte et donc de verrouiller, déverrouiller et ouvrir une porte.

5.2 Fonctionnalités pour le joueur

Le Joueur s'est vu attribuer quelques fonctions procédures

- SetVisibilite pour être détectable ou non par le monstre
- getter setter pour les points de vie
- estTouche pour afficher à l'écran que l'on a été frappé et créer une période d'invulnérabilité.
- estVivant qui permet de savoir si le joueur est vivant ou non.

5.3 Fonctionnalités pour le monstre

Il est possible grâce à setVisibilite de rendre le monstre invisible. Cela sera utile plus tard pour qu'il apprenne de façon secrète lors des premiers rounds.

5.3.1 Vision

L'implémentation de la vision du monstre n'est pas encore au point mais il reste qu'un FOV à implémenter. En effet, la vision du monstre se fait en deux étapes grâce au gameObject EyeSight. En premier, le box collider vérifie que le joueur actionne son trigger. Souci, le monstre voit à travers les murs. Afin de régler ça rapidement, un RayCast est généré au niveau des yeux du monstre et de cette manière, en cas de collision entre le rayon et le

collider du joueur, alors le monstre voit le joueur. Cependant, le monstre n'envoie qu'un seul rayon donc quand le joueur est légèrement décalé et pas pile au milieu, il ne voit rien. A terme, il faudra coder un Raycast qui forme un FOV en cône et enlever le box collider. Ou alors, se servir des dimensions pour faire un raycast qui suit les bordures et la portée du box collider.

5.3.2 Portée de coup

le gameObject Reach est une capsule collider au niveau des bras du monstre qui permet de s'assurer que le monstre est à portée du joueur et donc de pouvoir le frapper s'il le décide. Un testeur de dégâts intégré au script permet de blesser le joueur dès qu'il est à portée de coups.

5.3.3 Déplacement

Le script Monstre contient la majorité des fonctions et procédures qui permettent de gérer le mouvement. La fonctionnalité NavMesh Agent de Unity est très pratique car elle permet de créer automatiquement un pathfinding par rapport à la carte existante. En fonction des ordres et de ce qui a été détecté, il suffit de fixer une cible à l'agent pour qu'il se dirige tout seul jusqu'à elle. C'est pour cette raison qu'il existe deux Waypoints qui servent à marquer les positions où le monstre doit aller. L'un sert à l'exploration et l'autre à suivre la dernière position du joueur. Quand il s'agit d'attendre, il suffit de dire au monstre de se suivre lui-même, ce qui l'immobilise. Cependant, ce système a une limite. En effet, quand le monstre est arrivé au Waypoint, il commence à tourner de manière aléatoire et finit par s'arrêter. Il faut probablement mieux spécifier les paramètres du NavMesh Agent avec les distances d'arrivée par rapport à la cible. Concernant l'exploration, le gameObject ExploRange est une sphère dont le rayon réglable est récupéré par la fonction de nouvelle destination. Le Waypoint est donc forcément placé à l'intérieur de la sphère. Il y a aussi une petite fonction qui règle la vitesse du monstre afin de l'accélérer en poursuite par rapport à l'exploration.

5.3.4 Gestion des ordres et états

Chaque ordre spécifié a une procédure affiliée permettant d'exécuter l'ordre. Ensuite des Getter / Setter des états ont été créés pour simplifier les appels externes. L'ordre est rappelé en permanence au monstre via FixedUpdate. Une fois la QTable reliée, il faudra mettre dans le gameMaster un timer pour limiter l'actualisation de la QTable, tel un timer de 10 secondes.

6 Map Design

Le labyrinthe a été pensé en 3 zones :

— Une première pièce vide où le monstre pourra tuer le joueur en boucle afin de remplir sa Q-table.

— La deuxième zone est un labyrinthe basique pour que le monstre apprenne à suivre le joueur en prenant des décisions sur l'endroit où aller.

— La troisième zone est un labyrinthe constitué de décors et de cachettes pour le joueur afin de "challenger" le monstre.

Il reste encore plusieurs choses à définir telles que la vitesse de déplacement du monstre par rapport à celle du joueur, comment l'accès aux autres zones se fera (clefs, temps, nombre de morts, état de la Q-table...)

Le labyrinthe est volontairement peu poussé car il s'agit d'une zone d'apprentissage et de vérification pour les développeurs et non la zone finale de jeu.

FIGURE 6 – Design du labyrinthe

7 Ouverture

Certaines fonctionnalités ont été commencées mais ignorées pour raison de temps. En effet, il faudra implémenter :

- Le lien QTable - GameMaster
- La nouvelle map labyrinthe
- Permettre au monstre d'interagir avec les portes ou au minimum de ne pas être bloqué par celles-ci
- Coder une lampe qui stun et se recharge
- Créer un vrai gameOver qui permet de relancer la partie sans devoir relancer le simulateur
- Intégrer et améliorer la fonctionnalité de cachette si le gameplay s'y prête

Les connaissances acquises pendant ce premier semestre nous sont très utiles car elles nous permettront d'aller encore plus vite dans l'implémentation de nouvelles fonctionnalités.

Tous nos objectifs n'ont pas pu être réalisés, mais au vu de la quantité d'information et de connaissances à acquérir, nous avons fait un bon et long chemin qui nous servira de tremplin pour le second semestre.

8 Sources

8.1 Vidéos

— Chaîne Sentdex
— Tutos Unity

8.2 Ressources

— Page wikipédia sur le Q-learning
— Git