Documentation projet jeu vidéo

PAe 231 : 2014/2015

Année 2013-2014

15 septembre 2014

Sommaire

1 Codage du mini-jeu labyrinthe

1.1 Heuristique

Le labyrinthe considéré ne contient que des murs perpendiculaires entre eux placés aléatoirement. Le labyrinthe est de forme rectangulaire. L'idée est que le labyrinthe ait une entrée et une sortie, placées elles aussi aléatoirement sur les murs extérieurs de l'ensemble. Pour éviter au joueur d'utiliser des techniques basiques comme longer un mur, il devra récupérer une clé cachée à l'intérieur du labyrinthe. L'emplacement de la clé est lui aussi aléatoire, mais ne peut pas être dans les deux premières "bandes" de murs du labyrinthe, pour obliger le joueur à s'aventurer profondément dans l'édifice.
Les murs sont tous les mêmes : de forme parallélépipédique, avec pour largeur 1 mètre et pour longueur et hauteur 5 mètres, empêchant le joueur de les escalader. Pour empêcher toutefois qu'il puisse passer par dessus, un mur invisible est posé au dessus du labyrinthe. Les murs sont initialement placés tous parallèlement aux autres, puis, lors de la création du labyrinthe, sont tournés aléatoirement de 0, 90, 180 ou 270 degrés autour d'un axe vertical situé au milieu d'une de leurs longueurs. On obtient alors un labyrinthe dont les murs sont générés aléatoirement.

1.2 Codage du labyrinthe

1.2.1 Prefabs nécessaires

1. MazeWall
   Il faut tout d'abord le prefab d'un mur de taille 1x5x5. Il doit être placé de telle sorte que le joueur ne puisse pas passer au dessus ou en dessous.
2. MazeTrigger
   Un cube de taille 5x5x5 qui se place à chaque intersection. Les cubes ne se recoupent pas mais tapissent le labyrinthe. Ils vont servir à repérer la position du joueur au sein du labyrinthe.

1.2.2 Préparation

1. Créer le terrain. Provisoirement, lui donner une texture banale.
2. Créer un prefab du mur : MazeWall. Le placer idéalement juste au-dessus du plan du terrain pour donner l'idée de confinement.
3. Créer un nouveau script : MazeWallManager. Ce script va permettre au mur de tourner autour d'un axe positionné au centre d'une de ses largeurs. Lui donner un attribut qui va contenir tous les angles possibles desquels il peut tourner :

   ```
   private List<int> anglesPossibles;
   ```

4. Dans la fonction Start, on va définir les angles possibles, et demander au mur de tourner.

   ```
   anglesPossibles = new List<int> () {0,90,90,180,180,270,270};
   ChangerAngleHasard ();
   ```

5. Créer la fonction ChangerAngleHasard :

   ```
   void ChangerAngleHasard ()
   {
   int angleChoisi = Random.Range (0,anglesPossibles.Count - 1);
   //Fait tourner le mur selon un axe vertical situe au centre d'une de ses
   //largeurs
   transform.RotateAround(transform.position- new Vector3(2.5f,0.0f,0.0f),
           Vector3.up,anglesPossibles[angleChoisi]);
   }
   ```

On a maintenant le comportement d'un mur bien défini. On lui ajoute un BoxCollider à sa taille et la texture qu'on veut, puis on le transforme en prefab.
Tous ces murs doivent maintenant être générés aléatoirement, on doit les instancier un certain nombre de fois.

1. Créer un script : MazeWallManager. Lui donner 4 attributs :

```
private List<GameObject> lesMurs; //contient les murs
private int longueurMurs = 5; //la longueur est egale a la hauteur ici.
public int nombreMursParCote = 10; //taille du labyrinthe
public Transform unMur; //prefab du mur en tant que Transform
```

2. Dans sa fonction Start, le script va choisir une taille pour le labyrinthe. Il y aura donc $taille^2$ murs à placer. Ces murs sont succesivement placés dans une variable locale Transform mur déclarée au début de Start. Dans une boucle for établie en conséquence, on instancie tous les murs, et on les place. Enfin, on les rend enfant du GameObject qui détient le script.

```
mur=Instantiate(unMur) as Transform;
//lors de l'instantiation, le mur tourne
float positionXf=longueurMurs*i;
float positionYf=longueurMurs/2;
float positionZf=longueurMurs*j;
//Changer la position du mur
mur.position = transform.position +
        new Vector3(positionXf,positionYf,positionZf);
mur.transform.parent=this.transform; //le mur est enfant du GameObject
//qui tient ce script
```

Maintenant, on veut que les murs extérieurs forment un carré. Pour cela, imaginons que tous les murs sont placés verticalement, avec leur axe de rotation sur leur partie basse. Faisons tourner de 90 degrés vers la droite tous les murs du haut : on obtient un mur horizontal droit en haut. Le dernier mur, en haut à droite, "dépasse" ; on le fait de nouveau tourner de 90 degrés pour qu'il soit dirigé vers le bas. On fait de même pour tous les murs tout à droite : on a un mur vertical. De même, le mur en bas à droite dépasse : on le tourne de nouveau de 90 degrés, de façon à ce qu'il soit dirigé à gauche. Enfin, tous les murs tout à gauche sont dirigés vers le haut. On a bien une enceinte fermée.
Les murs changent directement de sens lors de leur instanciation. On ne peut donc pas savoir comment ils sont inclinés. Il faut changer le script MazeWallManager et ajouter un attribut contenant l'angle duquel le mur est tourné. La structure du script est donc changée :

```
int angleDeRotation = 0; //le mur est droit
void Start () {
anglesPossibles = new List<int> () {0,90,90,180,180,270,270};
int nombreRandom = Random.Range (0,anglesPossibles.Count - 1);
angleDeRotation = anglesPossibles[nombreRandom];
Tourner (angleDeRotation); //Tourne le mur de l'angle choisi
}

void Tourner (int angle) {
transform.RotateAround(transform.position-
new Vector3(2.5f,0.0f,0.0f),Vector3.up,angle-angleDeRotation);
        }
```

On peut maintenant imposer dans MazeWalls les orientations des murs sus-cités.
Problème : il faut accéder au composant MazeWallManager (mwm) de chaque mur. Et ensuite, pour faire tourner les nouveaux murs, il faut retrouver un point par lequel l'axe vertical doit passer, et c'est difficile. Il faut donc changer la structure du code pour que MazeWalls, une fois qu'il a placé les murs, leur demande de trouver leur point de rotation et de le mettre en attribut. Comme le mur ne peut pas avoir tourné avant ça, on ajoute la déclaration de anglesPossibles à ce moment. Cependant, anglesPossibles ne semble pas retenir ses valeurs. On a pour les orientations spéciales :

```
if(i==0){mwm.Tourner(270);}
if(j==nombreMursParCote-1){mwm.Tourner(0);}
if(i==nombreMursParCote-1){mwm.Tourner(90);}
if(j==0&&i!=0){mwm.Tourner(180);}
```

Ensuite, il faut créer l'entrée et la sortie du labyrinthe. On va donc choisir deux murs, opposés l'un de l'autre, dont on va dans un premier temps désactiver le MeshRenderer et le BoxCollider. Plus tard, on remplacera le MeshRenderer par quelque chose de différent (mur d'une autre couleur...) et on mettre le BoxCollider sur IsTrigger pour passer dedans et finalement changer de scène. On sépare ces deux actions, même si elles sont identiques, dans deux méthodes différentes pour anticiper sur le changement de scène.
Pour l'instant, il n'y a pas de recherche possible en fonction de i et j : on doit donc choisir le j d'entrée / sortie en dehors de la boucle.
Pour donc pouvoir repérer les murs, on ajoute dès maintenant des variables publiques cachées dans l'inspecteur pour récupérer les valeurs de i et j depuis la boucle. On crée une fonction setter dans MazeWallManager à cet effet.
On crée un prefab pour ces nouveaux cubes, car ils contiendront un script qui donnera au MazeWallManager l'information sur la position du fpc. On leur enlève le meshRenderer et le collider est mis sur isTrigger. On procède comme pour l'instantiation des murs, mais avec ces cubes. On les redimensionne pour qu'ils aient la taille d'un mur pour bien remplir les intersections. Leur positionnement se fait par rapport aux murs, mais décalés selon l'axe z d'une longueur de mur/2, plus une largeur de mur sur 2.
On s'aperçoit qu'on a trop de cubes, qui proviennent des rangs maximum de i et j ; on met tout le code lié aux triggers dans un if.
On ajoute un script, MazeTriggerManager, au prefab. On ajoute la fonction SetCoordinates précédente car on en a besoin.
On veut se donner la possibilité de faire apparaitre des objets, notamment le joueur et les zombies, à des intersections du labyrinthe. On ajoute donc des GameObjects vides placés au centre des cubes. Contrairement aux cubes, ils n'auront pas besoin de scripts attachés à eux : on peut les créer sans Instantiate. Comme ils sont placés au centre des cubes, on peut utiliser les cubes en eux-mêmes, sachant que leur centre est placé là où on aurait mis les dropPoints.
On va maintenant tester ce système, en utilisant le FirstPersonController. On le déclare comme variable interne (fpc) dans GameWalls. On le trouve au début de la fonction Start avec GameObject.Find. Pour tester, on le place successivement aux emplacements de chaque boite. Ca marche. On se dit donc qu'il serait possible qu'on souhaite régénérer le labyrinthe à un moment, sans forcément bouger le personnage. On sépare donc tout le code dans la fonction Start en deux sous-fonctions : GenererLabyrinthe et PlacerFPC. On a donc besoin d'ajouter les ordonnées des murs entrée/sortie en attributs pour qu'ils soient accessibles dans ces deux fonctions.
Pour pouvoir refaire le labyrinthe, on ajoute au début de GenererLabyrinthe un code pour détruire tous les objets contenus dans lesTriggers et lesMurs. Pour tester, on peut utiliser la fonction Update :

```
void Update () {
if(Input.GetKeyUp(KeyCode.R)){GenererLabyrinthe();}
}
```

Maintenant, on fait en sorte que MazeWalls sache tout le temps où se trouve le fpc. Pour cela, on l'assigne en tant qu'attribut à chaque cube. Chaque cube, dans sa fonction OnTriggerEnter, renverra l'information du passage du fpc à MazeWalls. On met donc un setter dans MazeWalls, et on en profite pour initialiser les nouvelles variables iFpc et jFpc dans PlacerFPC. On ajoute d'avance les getters de iFpc et jFpc. On ajoute un attribut MazeWalls à MazeTriggerManager. On lui ajoute un setter, qu'on utilise dans la boucle de MazeWalls pour l'initialiser. On ajoute la fonction OnTriggerEnter dans MazeWallManager pour donner via le setter les nouvelles coordonnées de fpc si c'est bien lui qui entre dans le cube. Pour faire ce test, il faut aussi que les MazeTriggerManager sachent qui est le FPC. On rajoute un attribut et un setter, utilisé aussi dans MazeWalls. On teste en affichant publiques iFpc et jFpc, pour voir dans l'inspecteur si ça fonctionne. On a donc ici un labyrinthe généré aléatoirement, dont l'enceinte est bien délimitée. Il y a une entrée et une sortie, et le joueur est téléporté, à génération du labyrinthe, à l'entrée de celui-ci. MazeWalls récupère en temps réel la position du joueur dans le labyrinthe en fonction de coordonnées, définies en fonction des colonnes et lignes du labyrinthe. La prochaine étape va être de permettre à un personnage non joueur de se déplacer dans le labyrinthe. Le déplacement sera basique : le personnage ne pourra se déplacer qu'entre les centres des différents cubes, sans faire de courbes. Un meilleur déplacement pourra faire l'objet d'une prochaine amélioration. Il peut être judicieux d'à ce moment importer le GameObject qui contient MazeWalls comme prefab.

1.2.3 Mouvement du PNJ

Commençons par créer un nouveau script : MazeTravel. Le mouvement total d'un PNJ sera constitué de plusieurs fois un déplacement entre deux intersections. Codons tout d'abord ce mouvement. Il s'agit, depuis la position initiale, d'aller se placer à une nouvelle position dont on connait les coordonnées (ce sont celles du cube). Mais pour connaître les coordonnées, il faut avoir accès à MazeWalls : il doit être en attribut de ce script. De même, comme MazeWalls est le grand manager du labyrinthe, il devrait avoir accès à tous les MazeTravel de la scène.
Deux possibilités sont envisageables. Soit les PNJ sont placés par nos soins dans la scène, à la manière du FPC : auquel cas, il vaut mieux opter pour une liste de GameObject dans MazeWalls qu'on remplira manuellement dans l'inspecteur. Soit MazeWalls crée les instances des PNJ à partir de prefab. On utilisera ici la première solution. On ajoute donc un nouvel attribut public dans MazeWalls qui contiendra tous les PNJ sous forme de MazeTravel. La fonction PlacerFPC devient alors PlacerPersonnages. On impose que les nouveaux personnages soient relativement éloignés du FPC avec i>3. Pour réduire la complexité du positionnement des personnages, on laisse tomber le parcours de la liste des triggers, et on utilise directement les formules explicites de placement des objets. On teste avec des GameObject au hasard pour vérifier leur bon positionnement aléatoire.
L'objet à déplacer a besoin de sa position. On lui donne donc les attributs correspondants, et un setter utilisé dans MazeWalls : :PlacerPersonnages, ainsi que dans MazeTriggerManager. MazeTriggerManager a donc besoin de la liste des PNJ, qu'on récupère à l'aide d'un setter. Maintenant, le PNJ saura toujours sa position à l'aide des Triggers.
On peut enfin s'intéresser aux déplacements basiques du PNJ. La première chose à faire est de savoir quand appeler cette fonction 'de base'. En effet, cette fonction va dire au PNJ de se déplacer entre deux points de l'espace pendant une durée constante. Il ne faudrait donc pas relancer cette même action au milieu d'un déplacement : il ne bougerait plus. Ca ne pose pas de problème tant que le joueur ne bouge pas, ce qui n'arrivera pas. Commençons toutefois par faire une fonction basique. L'algorithme de déplacement sera codé dans MazeTravel. L'algorithme aura besoin de savoir où aller ; il a donc besoin de savoir la position du GameObject qui tient GameWalls pour transformer une position i,j en Vector3. On lui donne un attribut qu'on initialise dans MazeWalls.Start. Au passage,

on aura aussi besoin de la longueur des murs, qu'on récupère de même.
Juste faire un Lerp ne marchera pas : l'objet a besoin de traverser les Triggers. Comme c'est la seule façon de faire, on laisse tomber le suivi des PNJ par trigger. Comme on sait leur position initiale, et qu'on sait quels mouvements ils font, on saura toujours où ils sont donc ça ne posera pas problème. On retire toutes les fonctions qui étaient là pour ça.
Ensuite, on va dans MazeTravel : :Update : c'est là que se feront tous les déplacements. On va donc devoir utiliser des flags. Si on est en déplacement, continuer le Translate. Si on est arrivés, ou assez près, on s'arrête (dans un premier temps). Le flag pour déclencher le déplacement est dans MoveTo. C'est lui aussi qui dit dans quelle direction aller et où. La direction, c'est la normalisation de l'arrivée moins la position actuelle.
Pour aller plus vite, on va directement déclarer la liste qui va contenir tous les mouvements à faire. Celle-ci sera constitué d'une succession de i et de j correspondant aux cases à parcourir successivement. On l'initialise dans Start. Tous les mouvements se feront dans Update car on se sert de deltaTime. On se servira de Translate, qui a besoin d'un vecteur de translation, colinéaire donc au vecteur séparant la position initiale de la position du prochain déplacement. Celui-ci ne change pas pendant le déplacement, on ne va pas le calculer à chaque Update : on le met en attribut. On l'initialisera en demandant le déplacement. Modifions donc la fonction MoveTo pour l'initialiser. MoveTo désormais ne servira qu'à créer le vecteur déplacement et à dire la position finale du mouvement. Comme il ne s'occupe pas de déplacer le personnage, on ne va pas changer i et j dans cette fonction. MoveTo donnera donc leurs valeurs à ces variables. La position finale dépend du i et j finaux, et de la position du GameObject qui tient MazeWalls, d'où la nécessité de bien avoir mis comme attribut cette valeur.
Il convient maintenant de parler de Update. Comme la fonction est appelée au moins à chaque frame, il faut savoir en temps réel si le PNJ est en train de bouger, donc si le script a déjà en mémoire la position suivante, ou si ce n'est pas le cas. On met donc un nouvel attribut qui servira à savoir si le PNJ bouge : isMoving. Dans le premier cas, si le PNJ ne bouge pas, on va lui dire d'avancer, donc appeler MoveTo. Cet appel ne pourra se faire que si la liste n'est pas vide, donc si elle a au moins deux éléments (un i et un j). Si c'est le cas, on appelle MoveTo, et on met isMoving à true car il va bouger à la prochaine frame.
Si le PNJ bouge, il faut regarder s'il est arrivé à la position suivante. Le PNJ bouge par pas de vitesse*Time.deltaTime : c'est à cette valeur qu'on doit comparer la distance entre la position actuelle et la position voulue. Si la distance (obtenue par Vector3.magnitude) est inférieure à ce pas, continuer à translater va nous éloigner de la position voulue. On dit donc au PNJ de s'arrêter, en mettant isMoving à false, on lui met ses nouvelles coordonnées i et j à l'aide de la liste des mouvements suivants, et on retire les deux premiers éléments de la liste donc. Si le PNJ est encore trop loin, on continue simplement la translation.
A ce stade, on peut manuellement demander aux PNJ d'avancer, en ajoutant des commandes dans la fonction Update en fonction des touches appuyées. Utiliser Start est dangereux car le script peut ne pas avoir le temps de s'initialiser. A noter que les objets vont allègrement passer dans les murs. En effet, la liste sera remplie de telle sorte que ces mouvements impossibles ne soient pas pris en compte. Le code à cette étape est disponible dans le package LabyrintheSansIA.
NOTE : La méthode Translate ne serait pas du tout viable avec un "vrai" PNJ animé via Mecanim. Il faudrait utiliser ses animations pour le faire avancer. Ca ne changerait théoriquement que la ligne "Translate", les autres ne changeant pas. De plus, si le PNJ avance réellement, il passerait parmouvementsAFaire.Add(i-1) les MazeTrigger, ce qui rendraient inutiles les modifications de i et j en ajoutant quelques lignes de code à MazeWallTrigger (comme pour le FPC).
NOTE : Il semble que cliquer trop vite, ou à une certaine fréquence, sur l'une des touches,

fait partir les objets très très loin. Surement parce qu'on essaie de dire aux objets de retourner à la même position qu'avant. Mettre un test au cas où. Après test, c'est pas ça. Ca peut venir du fait que les trucs dans Update mettent trop de temps à se faire. FixedUpdate ?

1.2.4 Implémentation de l'intelligence artificielle

Avec les deux "PNJ" précédents, on se rend compte qu'on peut imposer des mouvements via la fonction Update, mais que les deux PNJ devraient agir en même temps. On va donc devoir ajouter une nouvelle fonction, FindPath, qui va utiliser un algorithme à essais successifs pour remplir la liste des mouvements à faire pour atteindre le FPC. Lors de l'appel à cet fonction, la liste va être modifiée souvent (par le principe de l'algorithme), il ne faut donc pas que Update fasse changer la position du PNJ au risque de résultats faux. On ajoutera dès maintenant un nouveau flag pour ne pas faire démarrer le mouvement si l'algorithme calcule un chemin. En revanche, comme le FPC va finir par bouger, l'algo pourra être appelé pendant le mouvement du PNJ : il ne doit pas s'arrêter et attendre, mais continuer son mouvement. Un problème pourra survenir si l'algorithme est trop lent, et commence donc pendant un déplacement et que celui-ci se finit avant la fin de l'algorithme. Le PNJ devra s'arrêter. Il faut aussi être certain que l'algorithme ne sera pas appelé deux fois de suite ; ce flag devra désactiver la fonction en elle-même. A noter que la fonction permettra de plus de savoir s'il existe un chemin entre deux positions, ce qui est idéal pour savoir si la sortie est bien accessible, ou si un personnage est coincé. Il suffit de vérifier que chaque PNJ peut accéder au joueur, et que l'un des PNJ peut accéder à la sortie. Lors de ces vérifications, il faudra prendre garde à ce que personne ne bouge, car une réponse négative à une des vérifications entraînera la reconstruction du labyrinthe.
On va créer une matrice carrée de taille nombresDeMurs-1 qui contiendra des vecteurs de positions accessibles depuis la case de coordonnées liées à la matrice. Par exemple, prenons l'intersection (5,6) qui a un seul mur à proximité, vers le haut. On se rappelle que les i sont selon l'axe des abscisses x et les j selon l'axe des ordonnées z. L'élément (5,6) de la matrice sera [4,6,4,7,5,5]. Comme ça, depuis la position (5,6), le premier essai sera (4,6), etc. Pour savoir s'il y a des murs aux alentours, on utilisera des rayons RayCast. Il faut donc qu'aucun objet autre que les murs ne soit en place. Il faut aussi prendre garde à ce que le PNJ ne puisse pas sortir du labyrinthe, en empêchant les nombres de la matrice d'être en dehors de l'intervalle acceptable [0,NombreMursParCote-2]. Le problème est que le FPC peut bloquer ces rayons. Il faudrait donc remplir la matrice sans que le PNJ, et donc sans que les objets/PNJ, ne soient présents dans le labyrinthe. Si c'est faisable au début (on remplit la matrice dans MazeWalls avant l'ajout des personnages), si on veut reconstruire le labyrinthe en cours de route, ça devient plus dur. Une solution serait de mettre tout le monde dans le layer "IgnoreRaycast". Mais alors le joueur ne pourrait plus être vu directement par l'ennemi, ce qui peut être problématique. Plus simplement, on peut les ajouter tous à un layer spécifique qui ignorera un certain type de Raycast, celui qui sert à remplir la matrice.
La liste obtenue est une liste de liste de liste. Elle sera remplie à l'aide de trois boucles. Il faut faire attention au positionnement des déclarations des listes pour remplir la matrice correctement. Pour éviter que les personnages ne perdent un éventuel Layer qu'ils auraient au départ, on le leur place dans des variables locales, on leur assigne le layer 8 (le nouveau), et à la fin de la fonction de remplissage, on leur remet leur ancien layer.
On commence par mettre le code pour changer le layer, qu'on teste. Ensuite, on met le code des deux boucles, sans le Raycast. On devrait terminer par un Set de la nouvelle matrice. En effet, lorsque cette matrice change, elle devra changer dans tous les MazeTravel. On doit donc ajouter une fonction setter qui va faire tout ça. Son code changera la matrice de MazeWalls et de tous les MazeTravel, auxquels on rajoute cette matrice

en attribut. En effet, ce sont les MazeTravel qui vont déterminer comment aller voir le fpc. Il est donc normal que ça soit eux qui aient la matrice qui dit comment on peut se déplacer.
Le problème qu'on rencontre est que le Raycast prend en compte les colliders des murs... qui sont invisibles et en isTrigger ! Donc on met les murs dans le layer "IgnoreRaycast". Ensuite, tout devrait fonctionner. Pour tester, on va laisser les objets se balader un peu dans le labyrinthe. On change la fonction Update de MazeTravel.
A cette étape, nous avons donc nos PNJ qui se baladent à leur vitesse dans la labyrinthe. Notons qu'ici, on ne traitera pas de ce qui pourrait se passer une fois le FPC atteint. En effet, le code qui indique ce qu'il se passera est intrinsèque au PNJ. Parfois, les PNJ n'en font qu'à leur tête et ne s'arrêtent pas à la position suivante, et continuent donc leur course infernale en dehors du labyrinthe. Pour empêcher ça, il suffit de rajouter un morceau de code dans Update pour être certain que le PNJ ne s'éloigne pas trop de la position de départ. Le code est disponible dans le Prefab LabyrinthePNJLibres.
On passe donc maintenant à la dernière étape : l'implémentation de l'algorithme à essais successifs AES. On commence par mettre dans MazeTravel un nouveau flag pour empêcher Update d'aller chercher dans mouvementAFaire des mouvements pas encore enregistrés. La fonction va créer la liste des positions successives à adopter pour atteindre le FPC. Il est possible que le joueur soit hors d'atteinte. Il faut prendre ce cas en compte. L'algorithme renverra donc true si la liste est correctement remplie, false sinon. Le premier souci qu'on aura est l'endroit où on change les coordonnées i et j du PNJ. Comme l'algo peut être lancé pendant que le PNJ bouge, il faudra que l'algorithme commence depuis cette position. On déplace donc le code de changement de i et j avant l'appel du mouvement élémentaire. La fonction Prometteur de l'AES doit prendre en compte le fait que le PNJ est passé ou non par la case sur laquelle il tente d'accéder. On ajoute donc une matrice pour vérifier si c'est le cas ou non.
L'implémentation du code a donc été faite sans heuristique aucune. Les PNJ vont donc atteindre le PNJ, mais non sans accrocs. Tout d'abord, on va rajouter une contrainte dans la fonction de DeplacementElementaire pour que la case suivante demandée soit bien juste à côté de l'actuelle. Cette contrainte supplémentaire demande que DeplacementElementaire renvoie désormais un booléan : si true, Update devra effectivement changer les valeurs de i et j, et activer isMoving, sinon ne rien faire. En rajoutant ces conditions, les PNJs vont se déplacer comme il faut, en continu, mais toujours avec une trajectoire horrible, ce qui peut compenser le fait qu'ils savent exactement où est le joueur.
A cette étape, il reste à exploiter le boolean qu'on obtient dans MazeWalls, qui dit si le PNJ peut atteindre le joueur, et l'extrapoler pour savoir s'il y a un chemin menant à la sortie pour le joueur. Cette extrapolation n'est pas réellement nécessaire, sachant que pour des valeurs relativement grandes du labyrinthe, il sera quasiment impossible de bloquer le joueur. Il reste ensuite à améliorer l'IA des PNJ si souhaité. Le code à cet instant est dans LabyrintheIASimple. Rq : pour 3PNJ, 20 murs de côté est trop. 15 semble être un bon compromis sans heuristique.
Dans MazeWalls, on repère l'appel de MoveTo pour ajouter, si la méthode renvoie false, un redémarrage du labyrinthe. En effet, si le PNJ voit le FPC au départ, il le fera tout le long ; le false ne pourra arriver qu'au tout début. Pour améliorer légèrement l'algorithme, on peut faire en sorte que le FPC soit une sorte d'aimant : le PNJ va aller dans la direction du FPC en priorité. Le code est ajouté dans AES, pour modifier l'ordre des éléments de la liste des positions accessibles. Ces modifications sont disponibles dans le prefab LabyrintheIA.
Ensuite, on voudra gérer la mort du joueur. Lorsque le joueur est attrapé par un PNJ, il revient au début. Le labyrinthe n'est pas changé pour permettre au joueur de se repérer. En revanche, il est nécessaire de replacer les PNJ, notamment si le joueur est attrapé proche de l'entrée : il ne pourrait plus s'échapper. Pour cela, on ajoute une fonction

Restart au MazeTravel pour réinitialiser tous les flags et matrices utilisés dans les algorithmes. On appelle cette fonction Restart dans PlacerPersonnages, qui est la fonction qui sera appelée à chaque fois que le joueur est attrapé.
On veut que les ennemis fassent face au joueur en toute circonstance pour rendre le tout plus plausible. Cette fonctionnalité s'ajoute avec un autre script, qui ne va gérer que la rotation des PNJ. Il est à noter qu'il faut rajouter, dans chaque Translate fait auparavant, le fait que les axes à choisir sont les axes globaux et non locaux. Ca n'a pas posé problème jusqu'à présent car les deux étaient confondus, ce qui n'est plus le cas en imposant une rotation selon y.
On peut également ajouter un AudioSource aux PNJ pour qu'ils s'annoncent. On prend alors un son qui sera maximal lorsque le PNJ est à distance suffisante du joueur pour l'attraper. On ajoute également un son d'ambiance au GameObject Maze. On peut également ajouter du brouillard et une lampe au joueur. A noter qu'avec ceci, la taille maximale du labyrinthe est péniblement 15.
Maintenant, on va ajouter au script qui fait tourner les PNJ le code pour enclencher le retour au début du joueur s'il s'approche trop près. On utilise de nouveau les Raycast pour cette fois regarder quel objet est touché. Si c'est le personnage, on active la caméra qui contiendra le screamer, ainsi que son clip audio. Sinon, on ne fait rien. Pour que l'image reste assez longtemps à l'écran, on utilise la commande yield, qui doit être appelée dans une coroutine. On peut alors changer la position des personnages pendant que l'image du screamer est affichée. Pour cela, il faut que le script ait accès à MazeWalls. On rajoute le code nécessaire dans ces deux scripts. Dans MazeWalls, au même endroit que pour le script MazeTravel, on attache MazeWalls aux LookAtPlayer.
Pour améliorer le temps de calcul et accéder à des tailles de labyrinthe plus élevées, on limite dans AES le nombre de mouvements max que les PNJ doivent faire. Une taille acceptable est 5 fois le nombre de murs de côté. On limitera à 100.
Après coup, on voit une amélioration du temps de calcul. Cependant, l'algorithme peut être changé. En effet, l'AES n'a besoin de regarder qu'une seule fois chaque case du labyrinthe pour savoir comme accéder à la cible : on enlève la remise à false dans l'AES pour éviter de regarder plusieurs fois des mêmes cases.
Maintenant, on s'intéresse à un problème graphique qui survient dans le labyrinthe : quand plusieurs murs se superposent, il y a un problème de textures. On rajoute donc dans MazeWalls une fonction qui détruit les murs qui sont "en double". Pour cela, on regarde les transform.position de chacun par rapport aux autres. Cette fonction est en $O(n^4)$ avec n le nombre de murs par côté, mais n'est appelée qu'une fois.

1.2.5 Implémentation des événements dans le labyrinthe

Tout d'abord, pour que le labyrinthe soit sombre et que la lampe serve à quelque chose, on va dans Project>Render Settings, et on met dans Ambiant Light du noir. Le fog n'a plus d'intérêt, sauf peut-être pour un événement. A noter que la lampe ne peut pas éclairer le fog. Pour éviter la monotonie, on ajoute donc différents événements qui peuvent s'enclencher dans le labyrinthe. On les ajoute dans MazeTriggerManager.

1. Explosion
   On ajoute une explosion devant le personnage. On ajoute donc dans les MazeTrigger un prefab quelconque d'explosion. Faire attention à bien ajouter le prefab d'explosion dans le prefab de MazeTrigger. On ajoute aussi un son.
2. Clignoter la lampe
   On utilise une coroutine pour faire ça. On désactive/active la lampe pour le clignotement, on lance un son pour dire que la lampe change d'état, et on demande un yield.

3. La lampe change de couleur
   On utilise un script, appelé Couleurs, pour y écrire les RGB de chaque couleur pour facilement les appeler. On en crée une instance dans la méthode qui va changer la lampe de couleur, et on sélectionne une couleur au hasard parmi une sélection. On effectue un Lerp pour faire changer graduellement la lampe de couleur.
4. Changer la portée de la lampe
   On peut par script changer la distance à laquelle le joueur peut éclairer la zone avec sa lampe, ainsi que l'angle du cône qui englobe la lumière de la lampe.
5. Le gouffre
   On utilise la méthode de : http ://answers.unity3d.com/questions/11093/modifying-terrain-height-under-a-gameobject-at-run.html ainsi que cette vidéo : http ://www.youtube.com/watcl
   Il est donc trop compliqué et trop coûteux de remettre en place le terrain après chaque modification. On choisit de mettre un gouffre au hasard sur le terrain, en i=12 et j=10. On met ces valeurs dans MazeManager au cas où on en ait besoin.
6. Bruit oscillant derrière le personnage
   On utilisera un gameObject vierge avec juste une AudioSource, qu'on fera bouger par script derrière le joueur. Comme ça, quand il courra, on pourra ajouter un battement de coeur par exemple.

1.2.6 Screamers et sons du labyrinthe

On avait avant une camera et un sprite devant elle. On activait la caméra et on jouait le son qui y était affiché, et c'est tout. Si on veut plusieurs screamers/sons, il faut changer ça. On ajoute un script sur la camera, MazeCameraScreamer. On y mettra les différents sprites et sons qu'on voudra ajouter. Chaque sprite aura un son pour aller avec. On crée donc le nouveau script deux listes publiques, mazeSounds et mazeScreamers. On les remplit, on crée une méthode qui va lancer un mazeSound et un mazeScreamer au hasard. On fait attention à ne pas déclencher deux screamers à la fois. On a donc un int, initialisé à -1, qui contient le rang dans la liste mazeScreamers de l'affichage en cours. S'il n'est pas à -1, c'est que la caméra est déjà enclenchée ; on ne fait rien.

1.2.7 Ajouter des objets aléatoires

Pour ajouter des objets aléatoirement dans le labyrinthe, on crée une nouvelle liste lesObjets. On crée une méthode PlacerObjets, appelée uniquement dans Start, dans le script MazeManager. Cette méthode est la même que pour les PNJ, sauf qu'on n'a pas besoin de dire aux objets où ils sont, ce ne sont que du décor. En revanche, il faut s'occuper de leurs layers dans la méthode SetMatriceDeplacementsPossibles. On copie juste le code pour les PNJ.
Il faut prendre garde à ce que tout le code pour les événements dans le labyrinthe (dans MazeWallTrigger) soit dans un if, qui vérifie que c'est bien le FPC qui rentre dedans. En effet, les objets ajoutés seront pourvus de rigidbody pour pouvoir un peu les éjecter dans le labyrinthe.

1.2.8 Correction de bugs

Pour réinitialiser le labyrinthe, il faut détruire les murs, placer les personnages et demander à retranscrire la matrice des déplacements possibles. Or, après l'initialisation, on se rend compte que juste copier la fonction Start de MazeManager ne suffit pas. On doit faire la matrice à la fin de la fonction PlacerPersonnages, pour être sur que tous les murs et les personnages sont en place.
Faire attention à ce que le FPC soit en DEHORS du labyrinthe lorsqu'on clique sur Play. Sinon, sa simple présence sur une case va déclencher les MazeTrigger pour rien et

provoquer peut-être des événements.
AJOUT DE LA COURSE : dans FPSInputController, rajouter dans le vecteur directionVector un multiplicateur par $(1 + Input.GetButton(\text{"}course\text{"}))$.