THE TIMELESS CHRONICLES : EON'S LEGACY

Rapport de soutenance 1

Noah Matthieu Abi Chahla
Otto Debrie
Corentin del Pozo
Ilyann Gwinner
Nathan Hirth

Table des matières

1 Introduction

Dans ce rapport de soutenance, nous allons vous dévoiler le parcours innovant de "The Timeless Chronicles : Eon's Legacy", une aventure ambitieuse orchestrée par WarpGates Studio. Nous allons plonger au cœur des défis uniques auxquels nous avons été confrontés, à la fois sur le plan technique et créatif, tels que la conception de notre serveur, l'adaptation au client, la création de personnages distinctifs et l'élaboration de cartes détaillées. Nous avons aussi développé une intelligence artificielle pour enrichir l'antagonisme du jeu et générer des mondes d'une incroyable richesse.

Chaque aspect de notre projet a été scruté avec minutie, depuis l'optimisation des interactions réseau, cruciale pour une expérience de jeu fluide et réactive, jusqu'à la sophistication de l'intelligence artificielle, qui ne se contente pas de défier le joueur, mais l'invite également à explorer de nouvelles stratégies de jeu. La création de mondes générés aléatoirement à chaque début de partie a été d'une grande importance dans notre développement, permettant à chaque joueur de vivre une expérience unique, enrichie par la diversité et la complexité des environnements. Cette démarche, alliée à notre passion pour le détail et l'innovation, a façonné un projet qui procure une expérience ludique inoubliable. Ce rapport met donc en avant nos progrès techniques et créatifs sur ce projet.

2 Organisation

2.1 Logiciel utilisé

Nous avons opté pour l'utilisation du moteur Godot dans le cadre de notre projet. Ce choix s'est avéré judicieux en raison des multiples avantages qu'il offre. En effet, ce logiciel nous permet d'effectuer une diversité de modifications grâce à sa compatibilité avec le langage C#. De plus, nous avons décidé d'utiliser Discord en tant que plateforme de communication.Nous avons également créé un bot Discord entièrement codé en Python par notre équipe, qui sert d'outil pour gérer les tâches de manière similaire à des logiciels comme Notion. De plus, la création d'un répertoire (repository) GitHub nous a facilité la mise en commun du travail et nous a permis d'avoir une meilleure organisation sur le projet en lui-même.

L'utilisation de ce répertoire GitHub a été cruciale pour notre gestion de version, permettant à chaque membre de l'équipe de travailler sur des branches distinctes avant de fusionner leurs contributions au projet principal. Cela a non seulement amélioré notre flux de travail, mais a également réduit les risques de conflits de code, facilitant ainsi une collaboration efficace entre chaque membre de l'équipe.

En somme, l'intégration de ces outils et méthodologies a significativement augmenté notre efficacité et notre capacité à gérer des tâches complexes de développement de manière collaborative. Cela a non seulement renforcé la cohésion de l'équipe, mais a également accéléré le rythme de développement.

2.2 Répartition des taches

Dans le cadre de notre projet, nous avons adopté une approche stratégique dans la répartition des tâches en tenant compte des compétences spécifiques de chaque membre de l'équipe :

- Noah Abi Chahla, en tant que Responsable Graphisme avec un soutien en IA, a pris en charge la partie graphique.
- Otto Debrie, en tant que Responsable IA et soutien en personnage, a apporté une expertise précieuse dans le développement de l'intelligence artificielle.
- Corentin Del Pozo, en tant que Responsable Map avec un soutien en Réseau, s'est concentré sur la conception des cartes.
- De même, Ilyann Gwinner, Responsable Réseau avec un soutien en Map, a pris en charge la mise en place et la gestion du réseau.
- Enfin, Nathan Hirth, assumant le rôle de Responsable Personnages avec un soutien en Graphisme, a joué un rôle essentiel dans la création et la gestion des personnages.

Cette répartition des rôles a permis à chaque membre de l'équipe d'exploiter pleinement ses compétences et son expérience, garantissant ainsi une collaboration efficace tout au long du projet.

3 Serveur

Dans cette partie, nous allons aborder les protocoles utilisés sur le serveur, comment le jeu va transmettre les données, ainsi que le système coté client.

3.1 Protocoles

Dans ***The Timeless Chronicles : Eon's Legacy***, nous avons utilisé les protocoles TCP et UDP a différents endroits de notre jeu. Nous allons parler du premier protocole tout de suite.

3.1.1 TCP

Nous avons utilisé le premier protocole pour la partie connexion et la création de partie. Nous avons choisi ce protocole réseau pour sa fiabilité, car nous avons besoin d'être sûrs que toutes les informations sont bien arrivées à destinations. En effet, le protocole TCP se déroule en trois parties :

1. **Le SYN :** le SYN correspond à l'envoi d'une donnée (chaine de caractère, tableau de bits...) de l'ordinateur 1 à l'ordinateur 2.
2. **Le SYN ACK :** cette partie est envoyée de l'ordinateur 2 à l'ordinateur 1 pour dire à l'ordinateur 1 que les données sont bien arrivées.
3. **Le ACK :** c'est la confirmation que l'ordinateur 1 a reçu la confirmation.

   Si un packet n'est pas reçu, peu importe l'étape, il est renvoyé.

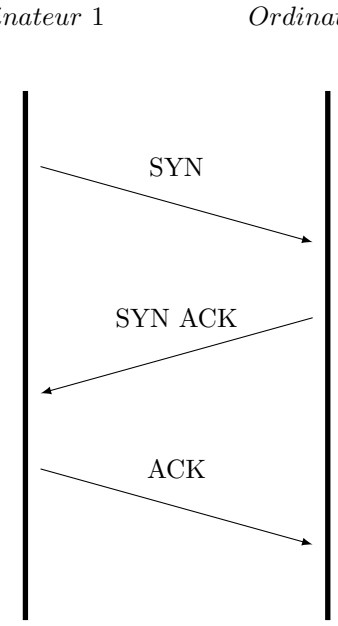

FIGURE 1 – Schéma du fonctionnement du protocole TCP

Nous utilisons le TCP, car ce serait problématique que le serveur ne puisse pas recevoir l'information qu'un client essaie de se connecter. Cependant, ce protocole est plutôt lent, car il faut envoyer/recevoir au moins 3 packets.

3.1.2 UDP

Ce second protocole nous sert pendant les parties. Contrairement au protocole TCP, le protocole UDP ne demande pas de confirmation *(cf. schéma ci-dessous)*. Comme ce protocole ne demande pas de confirmation, il est extrêmement rapide. Cependant, rien ne nous garantit que le packet va arriver et, s'il n'arrive pas, il est perdu et l'information n'arrivera jamais. Avec sa vitesse de transmission, nous pouvons obtenir le plus rapidement les coordonnées des autres joueurs.

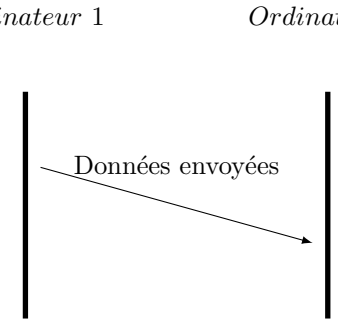

FIGURE 2 – Schéma du fonctionnement du protocole UDP

3.2 Les différents serveurs

Coté serveur, nous avons choisi une architecture comportant deux serveurs différents. Le premier, dit principal, et le second, dit de partie.

3.2.1 Le serveur principal

Le serveur principal occupe toute la partie connexion et création de partie. Il utilise le protocole réseau TCP pour éviter de perdre les informations. C'est ce serveur qui gérera la base de donnée des comptes. Il démarrera un serveur de partie et redirigera vers ce dernier tous les joueurs d'un lobby lorsqu'ils démarreront la partie. À ce moment-là, tous les joueurs présents dans le lobby se déconnecteront du serveur principal pour aller vers un serveur de partie.
Le serveur principal gère les clients au début sous forme d'ID puis sous forme de pseudo une fois connecté (aucun pseudo ne peut être doublé).

3.2.2 Le serveur de partie

Ce serveur est démarré par le serveur principal à chaque nouvelle partie sur un port non utilisé. Il restera allumé jusqu'à que tous les joueurs se déconnectent. Ce serveur utilise l'UDP, car il s'occupe des coordonnées et qu'il faut beaucoup de débit pour minimiser le lag.
Ce serveur gère les clients avec des IDs allant de zéro à trois, car les parties sont limitées à quatre joueurs.

3.3 Transmission des informations

Pour transmettre les informations, nous avons créé une sorte de langage pour savoir à quoi correspondent les informations reçues. En voici la liste.

La connexion : pour la connexion, nous commençons la ligne avec "conn :" puis nous écrivons l'identifiant ainsi que le mot de passe haché (avec sha) tous deux séparés par un point virgule (exemple : "conn :admin ;5e884898
da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8"). Le serveur répondra "connection success" si la connexion a réussi sinon il renverra "wrong password".

L'inscription : pour l'inscription, nous commençons par "insc :" puis nous continuons comme pour la connexion. Le serveur renverra "creation success" si l'inscription a pu aboutir, il renverra "already exist" sinon.

La création de partie : ici, le client enverra la phrase "newgame" au serveur lorsque l'utilisateur voudra en créer une. Le serveur répondra la même phrase suivie de l'ID de la partie (exemple : "newgame :AAAAAA").

La connexion à la partie : lorsque le joueur voudra rejoindre une partie déjà existante, il enverra au serveur "joingame" suivi de l'ID de la partie renseignée par l'utilisateur avec un espace devant (exemple : "joingame AAAAAA"). Le serveur lui répondra "join" si la partie peut être rejointe, sinon il renverra "ID inconnu".

Lancement de la partie : lorsque le joueur qui a créé la partie décide de la lancer, il envoie "start" au serveur qui enverra la ligne "newserv :" suivi du nouveau port du serveur à tous les joueurs de la partie.

Synchronisation des temps de chargements : pour synchroniser les temps de chargements, tous les joueurs doivent envoyer "load" au serveur à la fin du chargement. Une fois tous les joueurs chargés, le serveur leur envoie "start" pour pouvoir tous démarrer en même temps.

Synchronisation des emplacements : pour synchroniser les postions ainsi que les orientations, tous les joueurs envoient au serveur les coordonnées sous la forme : *"<ID>_in :co :<coordonnée x> ;<coordonnée y> ;<coordonnée z>/<rotation x> ;<rotation y> ;<rotation z>"*. Le serveur stockera cette ligne et renverra cette ligne concaténée aux informations des autres joueurs séparées pas un "|".

Synchronisations dites "oneshot" : ces synchronisations, dites "oneshot", sont des synchronisations d'événements (comme un changement de niveau ou un message dans le chat par exemple). Cette ligne sera sous la forme : "on :<ID>_<informations>". cette ligne sera renvoyée à l'identique à tous les joueurs.

Synchronisation des animations : cette section est comme celle du dessus sauf que nous remplaçons le "on" par "an" mais marche de la même façon. Le mot clé "an" existe uniquement pour éviter les conflits avec le chat par exemple.

Système de déconnexion en partie : pour finir, nous allons parler du système de déconnexion sur le serveur de partie. Ce système consiste à envoyer son ID ainsi que le mot "quit". Le serveur enverra cette phrase à tous les joueurs qui s'occuperont de faire disparaître le joueur déconnecté.

3.4 Côté client

Côté client, il y a un script nommé *GameManager* qui sert à structurer et centraliser tout ce qui se passe. C'est donc ici que ce fait la liaison avec le serveur.
Dans ce script, nous utilisons un système d'état qui nous permet de structurer une partie. Il en existe neuf que je vais vous lister.

- ***_Ready() :*** Au début, ce n'est pas un état, mais un *_Ready()* (fonction créée par Godot qui s'effectue au démarrage). Cette fonction effectue la connection avec le serveur.
- **State 0 :** Le premier état est appelé l'état 0. Il s'occupe de toute la partie connexion et inscription.
- **State 1 :** Le second état sert uniquement à passer au lobby.
- **State 2 :** L'état suivant sert à s'occuper du lobby. C'est-à-dire la possibilité de pouvoir créer et rejoindre une partie ainsi que de la démarrée.
- **State 3 :** Cet état s'occupe de se déconnecter du premier serveur et de se connecter au second.
- **State 4 :** Celui-ci affiche une barre de chargement ainsi que la synchronisation de celui-ci.
- **State 5 :** Le cinquième état charge les joueurs sur la carte.
- **State 6 :** Nous passons maintenant à l'état utilisé le plus longtemps. Il s'occupe d'envoyer toutes les coordonnées ainsi que les synchronisations oneshot.
- **State 7 :** Le dernier état s'occupe de changer de carte, de supprimer les joueurs et nous renvoie immédiatement à l'état numéro 4.
- ***_Notification() :*** Enfin, nous passons à la dernière étape qui n'est pas un état, c'est la déconnexion. Cette étape s'enclenche grâce au système de notification Godot (système qui créer un signal quand certaines actions s'effectuent, ici la fermeture du jeu). Une fois la notification reçue, nous déconnections le joueur du serveur avec le "système de déconnexion durant la partie" vue précédemment.

3.5 Les Seeds

Dans notre jeu, il existe 2 seeds, la seed de la map, et la seed des aléas. La seed des aléas est changé à chaque partie et elle correspond au fog *(cf. partie sur le fog)* ou au spawn des monstres.
La seconde seed est la seed de la map. Elle change uniquement toutes les 24h (à minuit). Pour avoir un aléatoire imprédictible, nous avons fait appel à différentes API. Tout d'abord, nous avons fait appel au site **Alpha Vantage** pour pouvoir avoir une API de bourse. Nous avons

choisi de prendre la bourse du célèbre studio français Ubisoft. Ensuite, nous avons pris une API de météo du site **Info Climat**. Pour le calcul de l'aléatoire, nous avons fait :

```
function RandomNumber()
    return ferme * volume + nivmer / (temp * pluie) * vent
```

Avec "ferme" qui est le prix de l'action Ubisoft lors de la fermeture du marché de la journée précédente, "volume" le volume d'action vendue cette même journée, "nivmer" le niveau de la mer prévue dans deux heures, "temp" la température a 2 mètres du sol au même moment, "pluie" le volume de pluie au sol et "vent" le vent moyen.

3.6 Ce qui reste à faire

Sur le serveur, il reste de nombreuses choses à faire, notamment à synchroniser l'intelligence artificielle des monstres et des boss. De plus, il reste de nombreuses optimisations au niveau du code et de la taille des données transférées.

4 Client

Après avoir vu tout le côté serveur de notre jeu, nous allons maintenant voir tous ceux qui concernent les personnages et les interfaces au niveau du client.

4.1 Joueur

Étant un jeu pouvant être joueur jusqu'à quatre joueurs, notre jeu contient en tout quatre types de classes de personnage différent : l'archer, le chevalier, le scientifique et enfin l'assassin. Chacune de ces quatre classes sont des classes de joueur, elles ont donc de nombreuse habilité en commun.

Afin de pouvoir fonctionner dans notre jeu, un personnage doit avoir plusieurs attributs et méthodes obligatoires. C'est pour cela que toutes nos classes de personnage héritent d'une classe abstraite mère afin de n'oublier ou de ne pas avoir à dupliquer ces attributs et ces méthodes.

4.1.1 Vue à la 3 personne

Au moment de lancer une partie, on peut déplacer sa souris afin de pouvoir voir librement l'environnement du jeu ainsi que son personnage. Pour faire cela, chaque personnage est équipé d'une *Camera3D* permettant à Godot d'afficher sur l'écran tous les objets présent devant la caméra. Cette caméra fonctionne grâce à un *SpringArm3D*, partant du joueur et allant vers l'arrière, qui empêche la caméra de pouvoir

voir à travers les murs. Le *SpringArm3D*, étant un rayon d'une certaine longueur qui détecte et récupère la position du point de contact avec un objet sur sa trajectoire, permet à la caméra de pouvoir se placer au point d'impact du rayon et donc d'éviter de se retrouver derrière un mur ou un objet.

Pour pouvoir bouger la caméra comme on le souhaite, nous utilisons la fonction *_Input()* de Godot afin de récupérer les mouvements de la caméra en récupérant toutes les actions de type *InputEventMouseMotion*. On calcule ensuite le mouvement de caméra en déplaçant la rotation en Y du *SpringArm3D* avec le mouvement de la souris sur l'axe X. Et en déplacement la rotation en X du *SpringArm3D* avec le mouvement de la souris sur l'axe Y.

La caméra offre également au joueur la possibilité de pouvoir zoomer et dézoomer. Pour faire cela, nous récupérons le mouvement de la molette de la souris pour pouvoir changer la valeur de l'attribut *FOV* de la *Camera3D* afin d'avancer ou de reculer l'angle de la caméra.

4.1.2 Mouvements et attaques

Dans notre jeu, les joueurs ont besoin de ce déplacé, il faut donc une méthode de déplacement. Cette fonction récupère les touches pressées par le joueur à un instant T, elle regarde si les touches pressées sont celle pour avancer (par défaut ZQSD). Ensuite pour chacune des touches, elle fait la différence entre la valeur des touches (1 si elle est activée et 0 si elle ne l'est pas) ce qui permet au mouvement de se compenser si on appuie à la fois sur la touche pour avancer et celle pour reculer. Une fois la valeur du mouvement vers l'avant et celle vers l'arrière, la fonction crée un vecteur à 3 dimensions composé :

— En X, du mouvement sur les côtés.
— En Y, d'une valeur de 0, car les joueurs ne peuvent pas sauter ou se déplacer vers le haut.
— Et en Z du mouvement vers l'avant ou l'arrière.

En connaissant ce vecteur direction, il suffit de le multiplier par la vitesse définie dans la classe (les vitesses sont différentes en fonction des classes, par exemple le chevalier a une vitesse moins élevée que l'assassin) pour pouvoir ensuite déplacer le joueur à l'aide du *MoveAndSlide()* de Godot qui permet de faire déplacer le joueur en fonction d'un vecteur *Velocity*. À noter que les joueuses ont la possibilité de courir, ce qui va changer la valeur de la vitesse pour permettre un mouvement plus rapide.

Malgré le fait que les joueurs ne peuvent pas sauter, ils sont quand

même soumis à la gravité. Si un joueur tombe d'un endroit élevé, il va être attiré par le sol grâce à un vecteur gravité qui se fait compenser par un autre vecteur quand le joueur touche le sol. Ce vecteur a une valeur de 9.8 qui correspond à peu près à la gravité terrestre.

Maintenant que nous avons vu comment les personnages peuvent se déplacer, nous allons voir comment ils peuvent combattre les ennemis. Suivant la classe choisie, les méthodes d'attaque sont différentes. Nous avons prévu de faire une attaque principale, qui se déclenche avec le clic gauche, ainsi qu'une habilité principale, qui se déclenche avec le clic droit, pour chaque classe. Toutes ces méthodes ont été faites à l'exception de celle de l'assassin.

Archer : Comme le sous-entend son nom, cette classe donne la possibilité au joueur de pouvoir tirer des flèches en tant qu'attaque principale. Pour cela, il devra utiliser sont habilités principal qui lui permet de viser grâce à un passage sur une vue à la première personne. Ce passage a la première personne se fait en mettant la longueur du *SpringArm3D* de la *Camera3D* à zéro ce qui permet à la caméra d'être au niveau du joueur et plus derrière lui. Pour que le rapprochement de la caméra soit progressif, nous avons utilisé la fonction d'animation de Godot pour réduire la longueur petit à petit sur une durée d'une seconde. Une fois que le joueur se retrouve en première personne, nous ajoutons au milieu de l'écran une croix afin de montrer au joueur la ou la flèche ira. Le joueur pourra utiliser le clic gauche pour viser, charger puis tirer une flèche. En maintenant le clic gauche, le joueur augmente la puissance de la flèche et la fait partir en le relâchant. La flèche est un *RigidBody3D*, elle possède donc une gravité ainsi d'une *LinearVelocity* gérer par Godot ce qui permet de créer le mouvement de la flèche grâce à une impulsion de départ représenté par l'attribut *LinearVelocity*. Cette vélocité de départ ainsi que la position de départ sont données par les formules suivantes :

```
PositionX = (CamV_PX + sin(CamH_RY) * 2 + Player_PX) / 2
PositionY = Player_PY + 1
PositionZ = (Player_PZ + cos(CamH_RY) * 2 + Player_PZ) / 2

LinearVelocityX = sin(CamH_RY) * 10
LinearVelocityY = -CamV_RX / 5
LinearVelocityZ = cos(CamH_RY) * 10 * ShootPower
```

Avec PX, PY, PZ qui correspond à la position X,Y,Z d'un objet et RX, RY, RZ la rotation X,Y,Z d'un objet.
CamV représente le *SpringArm3D* de la caméra et CamH représente le point H situé au début du *SpringArm3D*.

Chevalier : Le chevalier possède une épée ainsi qu'un boulier pour

pouvoir se battre. Son attaque principale est un coup d'épée. Pour cette attaque, une animation de coups d'épée est déclenché une fois que le joueur appuis sur le clic gauche. Afin de détecter si un ennemi est touché, nous allons mettre une *Area3D* qui détectera si un objet se trouvera devant le joueur dans sa zone d'attaque. Si cet objet est un ennemi, nous exécuterons une des méthodes de l'ennemi pour lui enlever des point de vie. Pour son habilité principal, le chevalier a la possibilité de bloquer les attaque grâce à son bouclier. Pour ce faire, nous détecterons si le clic droit est activé et nous annulerons ou réduirons les dommages de base si c'est le cas.

Scientifique : Les aptitudes du scientifique ressemblent à celle de l'archer. Ils ont la même habilité principale qui est la possibilité de viser. Cette habilité fonctionne de la même manière. Mais les attaques diffèrent, au lieu de lancer des flèches, le scientifique tire un rayon devant lui. Pour faire ce rayon, nous utilisons un *RayCast3D* afin de détecter si le rayon rencontre un obstacle et si c'est le cas, de pouvoir repère les coordonnées de l'impact. Pour afficher le rayon, nous utilisons un cylindre que nous plaçons à mi-chemin entre le joueur et le point d'impact du *RayCast3D*. Une fois placer, nous augmentons la taille du cylindre de la moitié de la longueur entre le joueur et le point d'impact. Avec ce système, si une entité rentre dans le rayon, sa taille sera automatiquement refaite afin qu'il s'arrête au niveau de l'entité.

Assassin : L'assassin est le seul personnage à ne pas avoir l'entièreté de ces capacités implémentées. Néanmoins, il possède quand même son habilité principale. Cette habilité est un dash permettant au joueur d'avancer rapidement dans une direction afin d'esquiver des attaques ou des ennemis.

4.1.3 Design et Animation

Concernant les graphismes, nous avons trouvé des modèles sur internet afin d'illustrer nos personnages. Nous avons également trouvé des animations pour les personnages sur le site de Mixamo. Pour passe les animations de Mixamo a Godot, nous avons utilisé Blender.

4.2 Autres Joueurs

Étant un jeu en réseau, il est nécessaire de pouvoir voir et interagir avec les autres joueurs que se trouve sur la même partie. Pour ce faire, lors du début de la partie, nous créons une entité de classe Player et un nombre d'entités de la classe OtherPlayer allant de 0 à 3 en fonction du nombre de joueurs présent dans la partie. La classe OtherPlayer a pour unique fonction de reproduire les mouvements et les déplacements des joueurs. Pour ce faire, elle va récupérer les données envoyer par le serveur concernant le joueur qu'elle représente. Une fois ces données

récupérées, elle va les interpréter, par exemple pour le déplacement et la rotation des joueurs, elle va s'orienter et se placer aux valeurs que reçu. Pour les animations que les joueurs peuvent faire, elle va jouer la même animation pour que tous les joueurs puissent la voir.

4.3 Interface utilisateur

Afin de pouvoir se connecter et paramètre son jeu, il est nécessaire d'avoir des interfaces. Ces interfaces apparaissent peuvent être appliqué avant le lancement d'une partie pour la connexion et le paramétrage de la partie ou bien durant une partie pour mettre en pause le jeu ou pour parler avec les autres joueurs via un chat.

4.3.1 Le Redimensionnement

Les interfaces ont besoin de s'adapter en fonction de la taille de l'écran. Godot permet d'adapter automatiquement la taille des boutons et des images, mais il ne change pas la taille du texte. Pour résoudre ce problème, nous avons utilisé le fait de pouvoir détecter le changement de la taille de l'écran pour appliquer un redimensionnement au texte. Voici la formule de ce dernier :

```
Size = DefaultSize * (ScreenSize / ScreenDefaultSize)
```

Avec "Size" qui représente la taille du texte que l'on redimensionne, "DefaultSize" qui représente la taille du texte par défaut, "ScreenSize" qui représente la taille actuelle de l'écran et "ScreenDefaultSize" qui représente la taille par défaut de l'écran.

4.3.2 La Connexion

L'interface de connexion est la base d'un jeu en multijoueur. Cette interface est divisée en plusieurs parties :

La première consiste à se connecter à son compte ou à en créer un si le joueur n'en a pas encore. Cette création de compte permet de distinguer les joueurs en jeu grâce au pseudo donné lors de la création du compte. Afin de pouvoir se connecter à son compte, le joueur doit rentrer son pseudo et son mot de passe. Pour faciliter la connexion, le joueur a la possibilité d'enregistrer son mot de passe en local, il pourra donc se connecter directement en appuyant sur un bouton de connexion rapide. Le stockage du mot de passe en local étant moins sécurisé, nous avons ajouté un message de prévention. Nous offrons également sur cette page de connexion la possibilité au joueur de changer la langue du jeu (Par défaut, le jeu est en anglais).

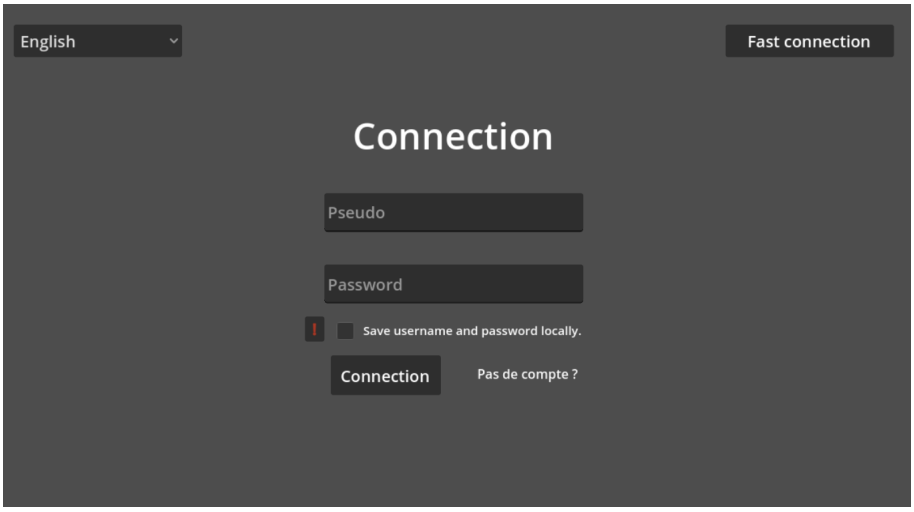

FIGURE 3 – Menu de connection

La seconde est pour créer et rejoindre une partie. Si le joueur décide de créer une partie, il sera envoyé vers une page dans laquelle il aura accès au code de la partie que le serveur a envoyé et il aura aussi la possibilité de commencer la partie. Le code de la partie permet à d'autre joueur de rejoindre une partie. Pour cela, il devra aller dans le menu pour rejoindre une partie et entrer le code de la partie. Si le code est valide, il sera envoyé sur la même page que la page de celui qui a créé la partie, mais sans la possibilité de commencer la partie. Dans ce menu, la liste des joueurs est affiché et s'actualise quand un joueur rejoint.

La dernière, est celle qui apparait une fois la partie démarrée. Ce menu permet aux joueurs de choisir leur classe. Ils ont le choix entre les quatre différentes classe. Une fois choisie, ils doivent cliquer sur un bouton pour valider leur choix et attendre que leur coéquipier ait validé leur classe. Une fois que tout le monde à faire sont choix, un chargement commence afin de charger la carte et les joueurs. Une fois ce chargement fini, la partie commence.

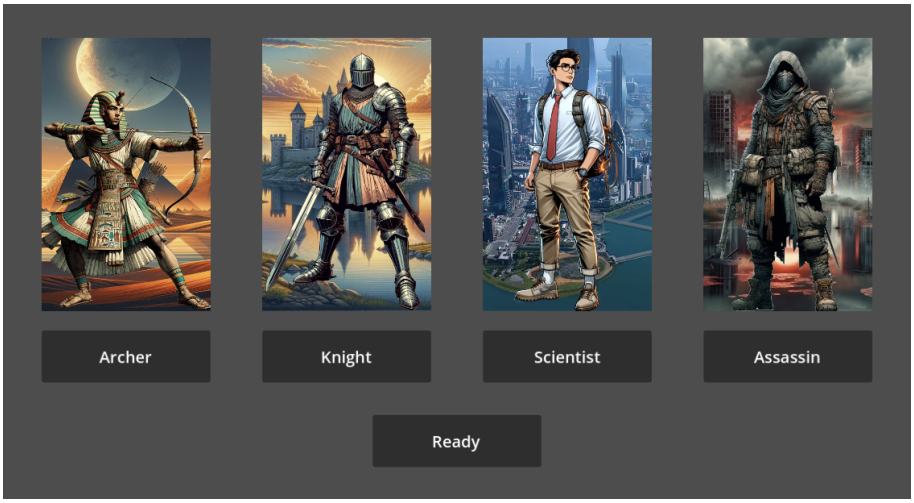

FIGURE 4 – Menu du choix des classes

4.3.3 Le Menu Pause

Le menu pause est très important, il permet à la fois de quitter le jeu, mais aussi de pouvoir changer ses paramètres. Afin d'afficher le menu Pause le joueur devra appuyer sur la touche de pause (par défaut Echap). Le menu va ensuite être placé sur la scène actuelle afin de proposer plusieurs boutons cliquable. Le premier est un bouton pour reprendre le jeu, le deuxième est pour aller dans l'interface de paramètre et le dernier est pour quitter le jeu.
Pour que le joueur ne quitte pas le jeu par inadvertence, une confirmation apparaît lorsque le joueur veut le quitter.
Il y a quatre sections différentes dans l'interface de paramètre, mais pour le moment seuls deux d'entre elle peuvent être utilisées :

La première est la section jeu. Cette section permet de changer la langue du jeu ainsi que plusieurs autres paramètres tels que la sensibilité de la souris ou bien la taille du chat. Pour pouvoir gérer et sauvegarder ces paramètres, nous utilisons un fichier .txt. Nous récupérons ensuite les données de ce fichier dans un dictionnaire contenant en clé de string et en valeur des int. À chaque fois que le joueur change un paramètre et qu'il appuie sur le bouton Save du menu, on change la valeur dans le dictionnaire et on réécrit dans le fichier des paramètres la nouvelle valeur pour la sauvegarder. À chaque fois qu'on utilise une valeur des paramètres, on appelle le dictionnaire avec le nom de ce paramètre.
La traduction du jeu a été fait au préalable. Nous nous sommes chargés de traduire le jeu en français et en anglais et nous avons traduit les autres langues avec une intelligence artificielle (GitHub Copilot). Les traductions sont sous forme d'une liste de dictionnaire. Chaque dictionnaire de cette liste représente une langue. La valeur du paramètre des langues dans le dictionnaire des paramètres correspond donc à l'indice

de la langue souhaité dans la liste.

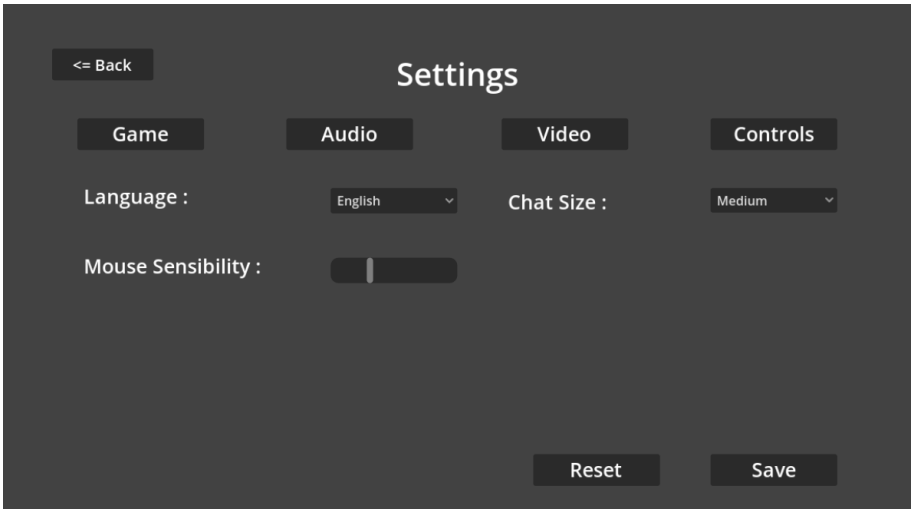

FIGURE 5 – Menu des paramètres du jeu

La seconde section fonctionnelle est celle du changement des touches. Afin que les joueurs puissent adapter leurs touches à leurs habitudes. Cette section fonctionne aussi avec un système de fichier en .txt pour stocker le nom des touches avec leur valeur en int. Après avoir récupéré les touches et leur valeur dans le fichier, nous les stockons dans une liste de tuple avec le nom et la clé des touches afin de pouvoir les afficher dans la section correspondante. Pour les afficher, on utilise un *ItemList* qui affiche une liste d'objet ordonné grâce à un id. Pour changer les touches, on récupère la touche pressée une fois que le joueur clique sur un des items de l'*ItemList* et on modifie la liste de touches ainsi que l'*ItemList* avec la nouvelle touche.

4.4 Ce qui reste à faire

Concernant le Client, il reste beaucoup de chose à faire même si la base des personnages et des UI est faite. Voici les choses qu'il reste à faire :

1. La vie, le mana, l'argent et l'expérience des personnages.
2. Les dommages causés avec la détection de l'ennemi qui est touché.
3. La mécanique de mort des joueurs.
4. Les capacités spéciales des personnages.
5. La mise en place de l'HUD avec la vie, le mana, l'argent et l'expérience.
6. Le reste des sections dans l'UI des paramètres.

5 Intelligence Artificielle

Les combats entre les joueurs et des "mobs" contrôlés par intelligence artificielle correspondent à l'un des aspects primordiaux du jeu ***The Timeless Chronicles : Eon's Legacy***.

5.1 Le comportement théorique

Les mobs doivent suivre un comportement spécifique, suivant différentes étapes :

1. Le mob attend d'être à une certaine distance du joueur pour se mettre à effectuer tous les calculs. Cela permet d'économiser énormément de puissance de calcul en ignorant les mobs inutiles.
2. Lorsqu'il est assez proche d'un joueur, et est capable de le voir directement, le mob commence alors à l'approcher.
3. S'il est assez proche, il commence à l'attaquer.
4. Lorsque le joueur n'est plus à portée de vue, une variable "*agro*", initialisée à 100, se met à diminuer jusqu'à atteindre 0.
5. Si le joueur est de nouveau visible, cette variable est remise à 100 et le mob se remet à le suivre.
6. Quand *agro* atteint 0, le mob retourne à sa position initiale (sauf s'il revoit le joueur entre temps).

En d'autres termes, le mob doit suivre l'algorithme suivant :

```
if distance(joueur,mob) < distance_of_view then
    lookat(joueur)
    if player is visible then
        agro = 100
        follow player
        if distance(joueur,mob) < value then
            attack
        end if
    else
        if agro vaut = 0 then
            while not mob is at his initial position
            and player is not visible
                go towards initial position
            end while
        else
            continue follow player
            agro = agro - 1
        end if
    end if
end if
```

5.2 L'implémentation

5.2.1 Le pathfinding

Ces intelligences artificielles doivent être capables de se repérer dans l'espace. Il est donc nécessaire d'implémenter un algorithme de pathfinding, ici l'algorithme A-star.

L'algorithme maintient deux ensembles de nœuds : un ensemble de nœuds à évaluer et un ensemble de nœuds déjà évalués. À chaque itération, A-star choisit le nœud à évaluer en fonction d'une fonction heuristique qui estime le coût restant pour atteindre la destination. Il évalue ensuite tous les successeurs de ce nœud, en mettant à jour leurs coûts estimés. L'algorithme s'arrête lorsque le nœud d'arrivée est évalué, fournissant ainsi le chemin optimal.

Godot permet une implémentation simple de l'algorithme, via certaines classes créés spécialement pour. L'utilisation d'un type *NavigationRegion* permet de définir tous les points dont l'accès est possible. Le type *NavigationMesh*, lui, permet alors aux mobs d'accéder au *NavigationRegion* et de déterminer les points qu'ils peuvent accéder. Le *NavigationMesh* peut alors se voir attribuer un type *Vector3* correspondant aux coordonnées de la cible du mob, afin que celui-ci puisse déterminer le chemin à suivre, puis avancer en fonction de ce chemin.

5.2.2 Le RayCast

Il est nécessaire que le mob ne se mettre à suivre un joueur que lorsque celui-ci est à portée de vue. Pour cela, la meilleure option est d'utilisé la classe implémentée par Godt : le *RayCast3D*. Cette classe se présente comme un rayon partant du mob et s'oriantant vers une cible définie (ici le joueur le plus proche).

L'utilitée de cette classe réside dans sa méthode "*iscolliding()*", qui renvoie *True* lorsqu'un élément se situe entre le départ du *RayCast* et sa cible. Ainsi, lorsue cette méthode retourne *False*, le mob peut lancer son algorithme de pathfiding pour se diriger vers le joueur le plus proche. Dans le cas contraire, la valeur "*agro*" (discutée précédement) diminura jusqu'à atteindre 0, où le mob reviendra à sa position de départ.

5.3 Ce qu'il reste à faire

Bien que la base des mobs est déjà implémentée, beaucoup reste à faire. Tout d'abord, la classe *mob* devra à l'avenir implémenter un système de points de vie. Ce dernier sera géré par une variable *HP* et une méthode *_AddHP(hp)* qui ajoutera le nombre hp à la variable HP (si la somme ne dépasse pas une valeur maximum prédéfinie) et supprime le mob de la partie si celui-ci voit sa variable *HP* inférieur ou égale à 0 (représentant la mort de l'ennemi).

Les mobs devront aussi implémenter un système d'attaque différant pur chaque type de mobs, d'où l'utilitée de faire de *mob* une classe

abstraite. Le pathfiding, raycast et le système de points de vie resteront implémentés dans la classe abstrète *mob*, là où le système d'attaque et des valeurs comme la vitesse, la force et le nombre maximum de points de vie seront implémentés dans des classes correspondantes à chaque type de mobs.

6 Map

Le jeu intitulé ***The Timeless Chronicles : Eon's Legacy*** est composé de quatre niveaux distincts, chacun situé dans des périodes temporelles différentes. Chaque niveau se compose de deux cartes (Maps) :

La première est celle où les joueurs apparaissent initialement. Cette carte est générée aléatoirement et varie quotidiennement, permettant ainsi aux joueurs de la redécouvrir à plusieurs reprises. Dans cette première carte, les joueurs doivent explorer tout en affrontant des monstres pour trouver la sortie qui les conduira directement vers la seconde carte.

Cette dernière est la salle du boss, qui reste généralement inchangée et contient uniquement le boss à vaincre pour progresser vers le niveau suivant. De plus, au début du jeu et entre chaque niveau, les joueurs passeront par la carte du marchand, qui demeure également inchangée.

En ce qui concerne les algorithmes utilisés pour la génération des cartes, ils sont tous différents ou présentent des variations par rapport à ceux utilisés ailleurs dans le jeu. Cela permet aux joueurs de ressentir une sensation de découverte constante, simulant ainsi l'exploration d'un nouveau monde à chaque niveau. Actuellement, seuls les trois premiers niveaux sont dotés d'une carte et seul le premier niveau dispose d'une salle de boss.

6.1 Niveau 1

La première étape se déroule au cœur du désert de l'ancienne Égypte, à l'intérieur d'un temple mystérieux dont le but est d'atteindre le centre de la map. Pour cette phase du jeu, nous avons opté pour l'utilisation d'un algorithme de génération de donjons unique. Cet algorithme, inspiré du jeu TinyKeep, a été présenté en détail par ses développeurs dans un article instructif sur gamedeveloper.com. En nous appuyant sur ces ressources, nous avons adapté cet algorithme pour créer le premier niveau du jeu. La conception de la carte s'effectue en plusieurs étapes.

Étape 1 : Nous commençons par générer un ensemble défini de salles situées dans un petit cercle central de la carte. Les coordonnées de ces salles sont obtenues grâce à l'algorithme ci-dessous :

```
function getRandomPointInCircle(rayon)
    local theta = 2 * math.pi*math.random()
    local randomsum = math.random()+math.random()
    local randomradius = nul
    if randomsum > 1 then
        randomradius = 2 - randomsum
    else
        randomradius = randomsum
    end if
    return ( rayon * randomradius * math.cos(theta),
             rayon*randomradius * math.sin(theta) )
end
```

Cet algorithme utilise des calculs trigonométriques pour sélectionner aléatoirement des points à l'intérieur du cercle défini par le rayon spécifié.

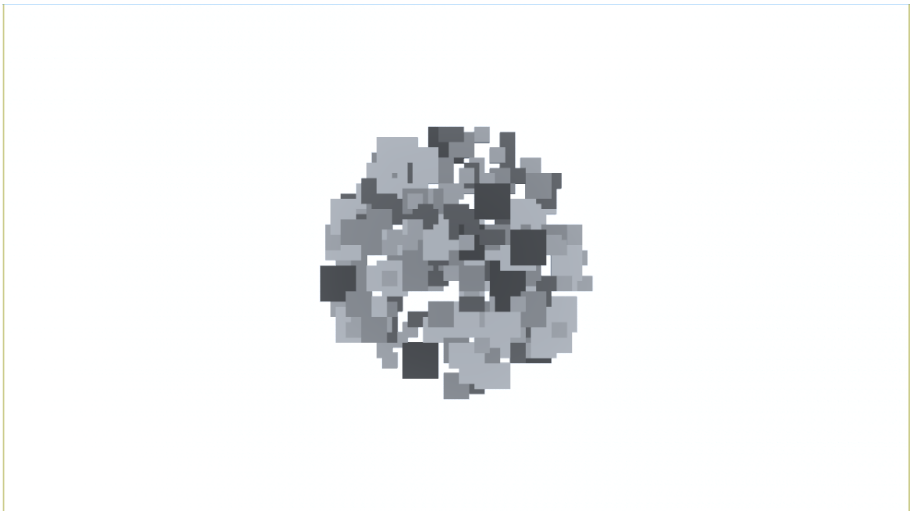

FIGURE 6 – Map avant le calcul de la physique

Étape 2 : Chaque salle est représentée par un objet *RigidBody3D*, qui est soumis aux lois physiques du moteur Godot. Comme les salles se superposent initialement, elles vont naturellement chercher à s'éloigner les unes des autres afin d'éviter cette superposition. Ainsi, les salles vont progressivement se déplacer tout en conservant leur cohésion jusqu'à ce que toutes les interactions physiques se stabilisent et que les salles passent en mode *Sleep*. À ce stade, nous pouvons passer à la suite de l'algorithme.

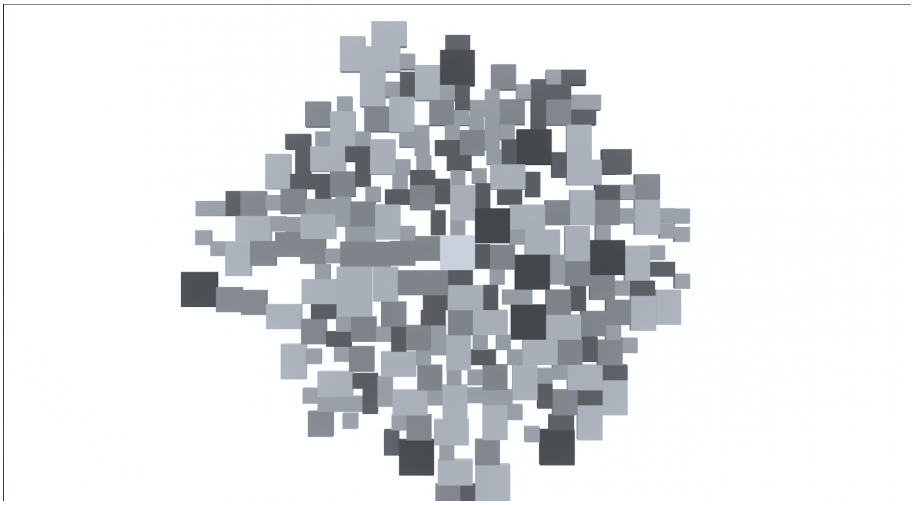

FIGURE 7 – Map après le calcule de la physique

Étape 3 : Après que toutes les salles se sont stabilisées et sont immobiles, nous procéderons à l'ajustement de leur position en les arrondissant. Chaque salle est constituée d'un certain nombre de tuiles de taille fixe, mesurant 6 par 6 mètres. Ainsi, nous allons arrondir leurs positions à des multiples de 6 mètres, assurant ainsi un alignement correct des portes lors de l'ouverture des salles.

```
function roundm(n, m)
    // n l'entier et m la taille d'un coté de la tuile
    return math.floor(((n + m - 1)/m))*m
end
```

Une fois que toutes les positions sont arrondies, nous supprimerons chaque salle pour la remplacer par une véritable salle équivalente, comprenant toutes les collisions et les éléments graphiques associés. Chaque type de salle présente des variations de décor intérieur qui sont choisies aléatoirement.

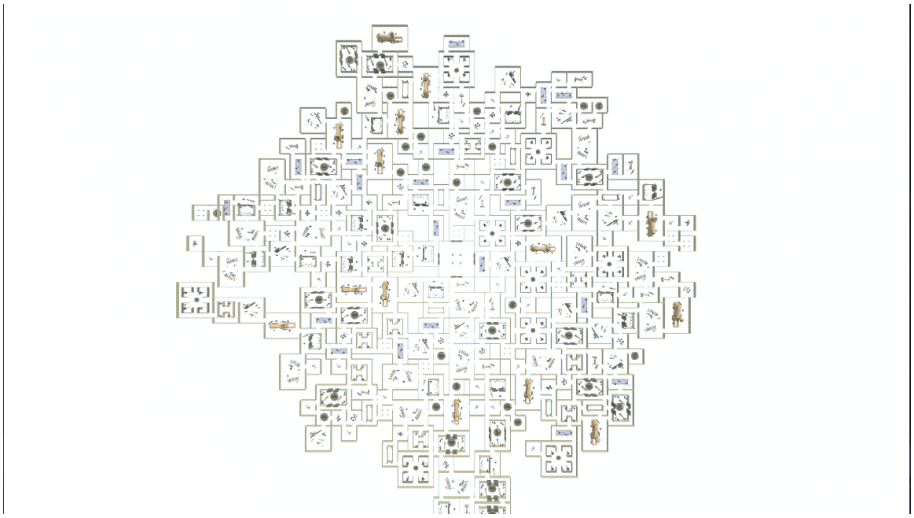

FIGURE 8 – Map une fois les salles remplacées

Étape 4 : Pour la dernière étape, nous parcourons toutes les salles et vérifions celles qui sont adjacentes à chaque salle en utilisant un calcul de distance, dans le but de repérer les murs en communs. Ensuite, nous choisissons aléatoirement un mur commun entre deux salles pour le transformer en porte. De cette manière, nous nous assurons que chaque salle est reliée à une autre et qu'aucune salle n'est coupée du labyrinthe. Au centre de la carte se trouve la salle principale, qui reste statique et immuable.

Après avoir accompli ces quatre étapes, nous sommes prêts à passer à la génération de la carte, notamment à l'apparition des différents monstres ou des joueurs.
Pour les monstres, chaque salle possède une liste de points d'apparition. Pour les faire apparaître, on parcourt cette liste et on fait apparaître aléatoirement un monstre sur ce point d'apparition.
Pour les joueurs, c'est presque la même chose, sauf qu'ils apparaîtront dans la salle la plus éloignée du centre.

Le Fog : Au premier niveau, pour ralentir les joueurs dans leur quête, une tempête de sable peut apparaître aléatoirement. Cette tempête est représentée par un brouillard couleur sable qui se lève progressivement pour brouiller la vue des joueurs et perturber leur repère visuel. Pendant cette phase, certains monstres qui n'étaient pas présents en temps normal peuvent apparaître.

Pour générer ce brouillard, on génère une durée aléatoire commune à tous les participants pendant laquelle rien n'est effectué. Une fois cette durée écoulée, le brouillard apparaît progressivement, puis une nouvelle durée aléatoire est générée pour définir la durée du brouillard.

Une fois cette durée écoulée, le brouillard disparaît progressivement, et le processus recommence.

6.2 Niveau 2

Le second niveau se déroule dans l'Europe médiévale, et la carte correspondante représente une vaste forêt dense. L'objectif est de localiser la porte du donjon après avoir récupéré quatre clés dispersées dans cette forêt.

Pour générer cette carte, trois éléments doivent être pris en compte :

1. Les arbres
2. Les clés
3. Les points d'apparition des monstres

Dans ce niveau, la carte a la forme d'un cercle entouré de falaises. Nous devons placer ces trois éléments à des emplacements stratégiques dans le cercle, en veillant à éviter toute superposition. Pour cela, nous allons réutiliser l'algorithme utilisé pour le premier niveau, tout en l'adaptant à cette nouvelle configuration.

Contrairement au premier niveau, il est préférable que les arbres soient espacés les uns des autres et ne se chevauchent pas. Par conséquent, nous agrandirons le cercle initial pour qu'il englobe toute la zone de jeu. Ensuite, comme pour la première carte, nous laisserons le moteur physique de Godot gérer les *RigidBody3D* afin d'éviter les collisions et les chevauchements.

Enfin, nous remplacerons tous les objets par des arbres, des clés ou des points d'apparition selon leur type, tel que défini au début. Chaque objet initial possède une taille différente selon son type. Toutes les étapes intermédiaires du processus se déroulent exactement de la même manière que dans l'algorithme utilisé pour le premier niveau.

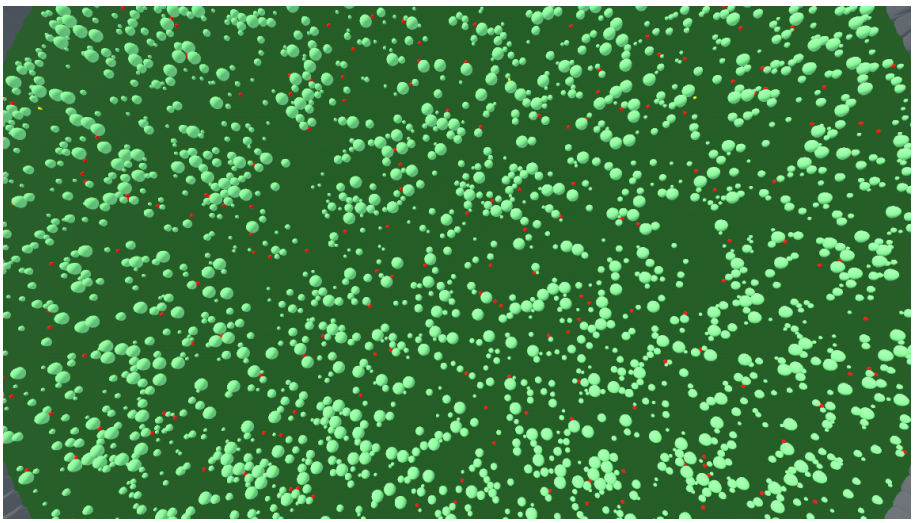

FIGURE 9 – Répartition des objets sur la Map du niveau 2
Rouge : Point d'apparition, **Vert :** Arbre, **Jaune :** Clé

6.3 Niveau 3

Le niveau 3, l'avant-dernier du jeu, se déroule au cœur d'une grande ville, dans un futur proche. Cette ville est composée exclusivement de gratte-ciel émergeant des nuages. Le jeu se déroule sur les toits de ces immeubles. Pour concevoir cette carte, nous utilisons un algorithme très différent : nous la créons à l'aide d'une fonction récursive et d'une matrice représentant la carte du jeu sous forme de grille, chaque cellule représentant un gratte-ciel. La création de cette carte se déroule donc en plusieurs étapes.

Étape 1 : Pour générer la carte, nous commençons par créer une matrice carrée de taille fixe remplie de zéros. Ensuite, nous exécutons une fonction récursive. Cette fonction a trois paramètres : n, un indice qui décroît à chaque appel pour savoir quand arrêter la fonction, i et j,

les coordonnées de la cellule de la matrice où se trouve la fonction. À chaque appel, la fonction remplace la cellule actuelle par un 1, puis vérifie les 4 cases autour de la cellule actuelle. Cependant, elle ne vérifie pas les cases adjacentes, mais celles situées une cellule plus loin. comme dans l'exemple ci-dessous :

$$\begin{pmatrix} 0 & 0 & X & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ X & 0 & 1 & 0 & X \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & X & 0 & 0 \end{pmatrix}$$

1 : représente la case actuelle où se situe la fonction,
0 : représente les cases libres,
X : représente les cellules que la fonction va tester pour savoir si elles sont libres

Laisser une case libre entre la cellule testée et la cellule actuelle est utile pour indiquer où les ponts seront positionnés et dans quelle orientation. Une fois la liste des cellules libres autour de notre cellule établie, la fonction en choisit aléatoirement deux, s'il y en a au moins deux, sinon elle prend la seule disponible. Si aucune cellule n'est disponible, elle s'arrête prématurément.

Ensuite, la fonction se rappelle aux deux prochaines coordonnées et change la valeur de la cellule entre la cellule de départ et la cellule d'arrivée par un -1 ou un -2 selon l'orientation du pont souhaitée. Si la variable n atteint 0, la fonction s'arrête également.

Un exemple de la matrice après exécution du programme :

$$\begin{pmatrix} 0 & 0 & \# & 0 & \# & 0 & \# & 0 & 0 \\ 0 & 0 & | & 0 & | & 0 & | & 0 & 0 \\ 0 & 0 & \# & 0 & \# & 0 & \# & 0 & 0 \\ 0 & 0 & | & 0 & | & 0 & | & 0 & 0 \\ 0 & 0 & \# & - & \# & - & \# & 0 & 0 \\ 0 & 0 & | & 0 & | & 0 & 0 & 0 & 0 \\ \# & - & \# & 0 & \# & - & \# & - & \# \\ 0 & 0 & | & 0 & | & 0 & 0 & 0 & 0 \\ 0 & 0 & \# & 0 & \# & - & \# & 0 & 0 \end{pmatrix}$$

Dans cette matrice :
représente 1 (les immeubles),
| représente -1 (les ponts verticaux),
- représente -2 (les ponts horizontaux).

Étape 2 : Une fois la matrice créée, nous la parcourons et pour chaque cellule, nous reportons sa position dans la matrice à une position sur la carte et faisons apparaître un bâtiment selon son numéro :

- **1 :** un immeuble avec un toit possédant une collision
- **-1 :** un pont vertical
- **-2 :** un pont horizontal

— **0 :** un immeuble de taille aléatoire, mais supérieure à la taille de l'immeuble de base

Voici un exemple du résultat obtenu (sans afficher les cellules contenant un 0) :

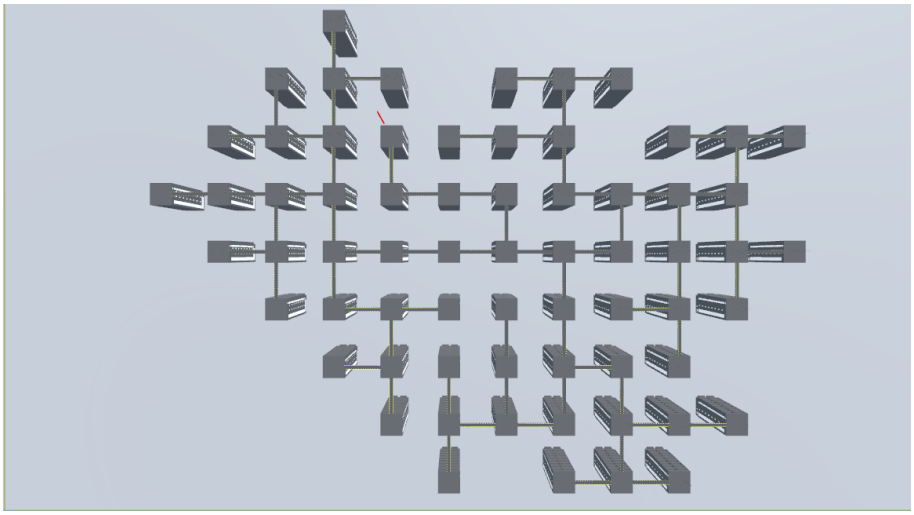

FIGURE 10 – Visualisation du cheminement créé par la fonction récursive

Étape 3 : Une fois ces étapes accomplies, il ne reste plus qu'à faire apparaître les monstres et les joueurs. Pour cette carte, l'apparition des joueurs et des monstres est similaire à celle du niveau 1, ainsi que pour l'objectif défini qui est d'atteindre le centre de la carte.

Une fois cela fait, comme dans les autres niveaux, on arrive dans la salle du boss qui n'est pas aléatoire, puis on retourne dans la salle du marchand.

6.4 Ce qu'il reste à faire

Concernant les Map la plupart des features sont terminés, il reste quelques étapes pour les finaliser à 100%, les voici :

1. La Map du niveau 4
2. Les salles du boss 2, 3 et 4
3. La salle du marchand
4. l'implémentation propre et optimiser du *NavigationMesh* (ce qui permet aux IA de savoir où elles peuvent se déplacer)
5. Les sorties de chaque niveau qui permet de passer à la salle du boss
6. l'implémentation du portail qui permet de rentrer dans chaque niveau

7 Conclusion

En somme, notre projet de jeu progresse de manière très encourageante. Actuellement, nous disposons de personnages opérationnels avec trois niveaux fonctionnels. Nos intelligences artificielles sont opérationnelles, mais uniquement en local. Le serveur a réussi à synchroniser toutes les actions, ce qui constitue une étape cruciale. Cependant, il reste un chemin à parcourir pour atteindre nos objectifs finaux. Parmi les tâches à accomplir, la synchronisation des IA sur le serveur, la conception de trois cartes de boss ainsi que l'élaboration du niveau 4 figurent en tête de liste. De plus, il reste à mettre en place l'IA des boss, à finaliser le sound design, à faire les dialogues et à assurer une transition fluide entre les différents niveaux. Ces étapes représentent autant de défis passionnants à relever pour concrétiser pleinement notre vision du jeu.