

Rapport de soutenance n°2

Mars 2025

BLACKOUT

Wassim ALOUINI

Brewen MERER

Loan MARIA-CHATELAIN

Fabien GUESSANT

Elliott MIEZE

Groupe : **Winter-Architect**

EPITA Promo 2029

Sommaire détaillé

1	Introduction	3
1.1	Répartition des tâches	3
1.2	Rappel du concept	3
2	Avancement	3
2.1	Interfaces graphiques	3
2.2	Modélisation des salles	6
2.3	Mécaniques de jeu	7
2.3.1	Génération de salles	7
2.3.2	Joueur 1	8
2.3.3	Joueur 2	9
2.4	Multijoueur	9
2.4.1	Problèmes identifiés et corrections	9
2.4.2	Fonctionnement actuel	10
2.4.3	Logique multijoueur	11
2.5	Intelligence Artificielle	14
2.5.1	Objectif de l'IA	14
2.5.2	Utilisation du Package AI Navigation	15
2.5.3	Création de NavMesh	15
2.5.4	Détection du Joueur	15
2.5.5	Machine à États Finis (State Machine)	16
2.5.6	Liste des États	16
2.5.7	Implémentation des IA	16
2.5.8	Types d'IA	17
2.5.9	Défis Techniques	17
2.5.10	Système de Gestion des Dégâts	18
2.5.11	Bilan de l'IA actuelle par rapport à la précédente	18
2.5.12	Les projets futurs	18
2.6	Level Design	19
2.6.1	Conception Des Enigmes	19
2.6.2	Bilan du Level Design	19
2.6.3	Projets Futurs	21
3	Conclusion	21
3.1	Bilan sur l'avancement	21
3.2	Perspectives et futur du projet	22
3.3	Réflexion et apprentissages	22

1 Introduction

Ce rapport est le second rapport de soutenance du jeu Blackout. Il a pour objectif de présenter l'avancée du projet depuis le dernier rapport.

1.1 Répartition des tâches

Référent	Suppléant	Tâche
Fabien	Wassim	Scénario
Loan	Brewen	Gameplay
Eliott	Wassim	Interfaces graphiques
Fabien	Eliott	Modélisation 3D
Wassim	Loan	Animation
Eliott	Wassim	Site Web
Brewen	Fabien	IA
Loan	Eliott	Level design

Table 1: Répartition des tâches

1.2 Rappel du concept

Blackout est un jeu vidéo coopératif de type *survival horror* centré sur l'exploration. L'histoire se déroule dans un laboratoire souterrain secret, appartenant à l'entreprise fictive *Winter Architect*, spécialisée dans l'expérimentation illégale sur des objets et entités aux propriétés anormales. Suite à un incident ayant provoqué la perte de contrôle du site, des mercenaires sont envoyés pour rétablir la situation, incarnés par les joueurs.

Le jeu propose une expérience immersive et dynamique avec une génération aléatoire des salles, obligeant les joueurs à adapter leurs stratégies en fonction des défis rencontrés. L'un des joueurs, l'**Agent**, se déplace librement et interagit physiquement avec l'environnement, tandis que l'autre, le **Support**, contrôle des dispositifs électroniques (caméras, portes, bras robotiques, ...) pour faciliter la progression.

2 Avancement

2.1 Interfaces graphiques

Nous avons terminé toutes les interfaces principales nécessaires au bon déroulement du jeu. Parmi elles, on retrouve :

- Un **terminal interactif** sur lequel les joueurs peuvent voir et écrire du texte, ainsi qu'interagir avec des boutons.

Pour concevoir ce terminal, nous avons utilisé un UIDocument, comme pour la plupart de nos interfaces. Côté script, nous avons implémenté diverses méthodes essentielles. Nous souhaitions un terminal réaliste permettant d'écrire des commandes, mais nous avons été confrontés à une problématique d'accessibilité. Pour y remédier, nous avons conclu qu'il était nécessaire d'ajouter quelques boutons principaux afin de faciliter la prise en main. Ainsi, le joueur dispose de deux moyens d'interaction : les commandes textuelles ou les boutons, permettant d'exécuter les actions suivantes :

- ouvrir la carte de la salle actuelle ('map')
- effacer les commandes ('clear')
- fermer le terminal ('exit')

Nous prévoyons d'en ajouter d'autres par la suite, ce qui est facilité par l'utilisation d'un **switch** dans le script gérant les commandes du terminal.

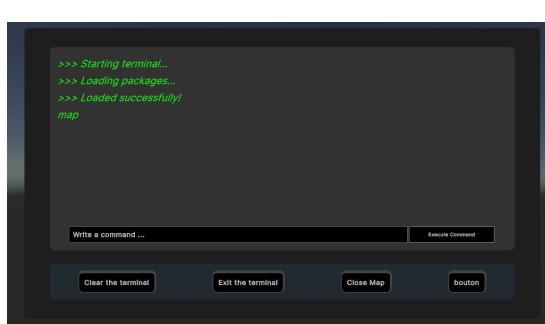


Figure 1: Terminal

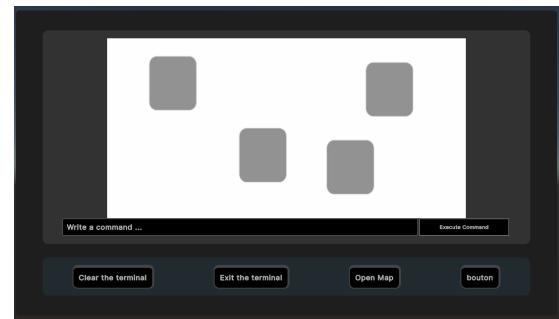


Figure 2: Affichage de la carte dans le terminal



```

void ExecuteCommand() {
    if (commandInput.value == "") return;
    AddMessageToTerminal(commandInput.text, true);
    switch (commandInput.value) {
        case "clear":
            ClearTerminal();
            break;
        case "exit":
            ExitTerminal();
            break;
        case "map":
            DisplayMap();
            break;
        default:
            AddMessageToTerminal("Command \\" + commandInput.value + "\\"
unknown");
            break;
    }
    commandInput.value = "";
}

```

Figure 3: Partie du code du terminal pour exécuter une commande

- Un **masque spécifique**, visible uniquement par le support, simulant une vision à travers une caméra et incluant un bouton permettant d'ouvrir l'interface du terminal.

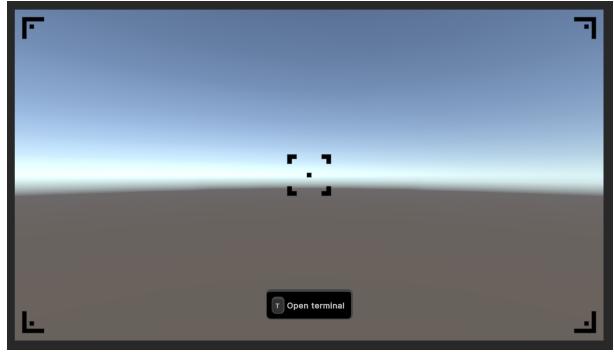


Figure 4: HUD de caméra pour le support

- Un **inventaire** sous forme de roue à six emplacements, accessible uniquement par l'agent. Celui-ci peut l'ouvrir avec la touche **TAB** et collecter des objets au sol via la touche **E**, ces objets étant ajoutés directement à l'inventaire.

Pour concevoir cet inventaire, nous avons d'abord créé un objet **Item** contenant les attributs nécessaires à tout objet du jeu : *id*, *nom*, *description*, *icone* et *prefab*.

Ensuite, nous avons conçu l'interface de l'inventaire en utilisant un **Canvas** et une image réalisée sur **GIMP**, que nous avons dupliquée six fois avant de l'intégrer au script de l'inventaire.

Le fonctionnement repose sur un **dictionnaire** associant chaque item à l'ID de la case d'inventaire dans laquelle il se trouve, ainsi qu'une **liste d'ID disponibles** pour permettre l'ajout de nouveaux objets. Lorsqu'un bouton est cliqué, il met à jour l'attribut **selectedItem** avec l'item correspondant et équipe ce dernier pour l'agent. Ainsi, il est très simple d'ajouter un item via un autre script.

Enfin, l'ouverture de l'inventaire est gérée via un **Animation Controller** util-

isant un booléen **isInventoryOpen**. Lorsque ce booléen est sur **true**, la taille de l'inventaire passe de 0 à 1, et inversement lorsqu'il est sur **false**.

Pour gérer l'inventaire, nous utilisons une liste de six objets (Item) directement intégrée au script principal de l'agent. L'ouverture de l'inventaire est animée grâce à un **Animation Controller** et un booléen **isInventoryOpen**.

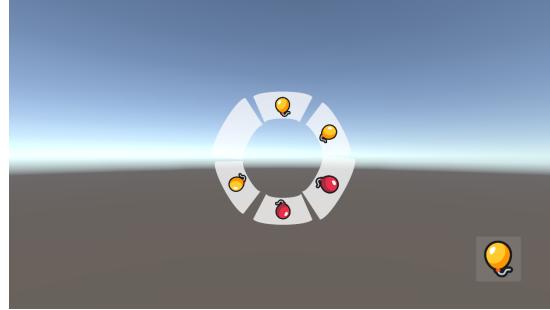


Figure 5: Inventaire

Cependant, il nous manque encore quelques détails tels que :

- L'ajout d'une fonctionnalité dans le lobby permettant de **sélectionner son rôle**, car actuellement, le premier connecté est l'agent et le second le support.
- L'extension des **paramètres** avec la possibilité de modifier les contrôles du jeu.

2.2 Modélisation des salles

Nous avons pu produire un certain nombre de modèles de salles, de meubles et de décors afin de permettre au joueur l'immersion dans notre univers graphique. Voici donc quelques exemples de ces modèles :



Figure 6: corridor



Figure 7: Escalier



Figure 8: hangar



Figure 9: Silo

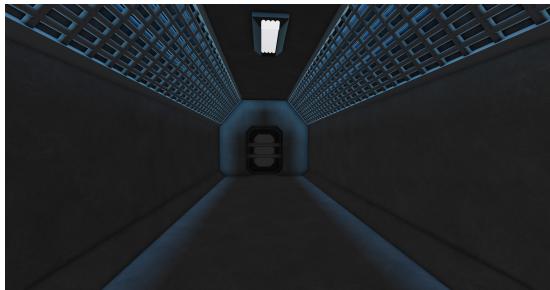


Figure 10: couloir de maintenance

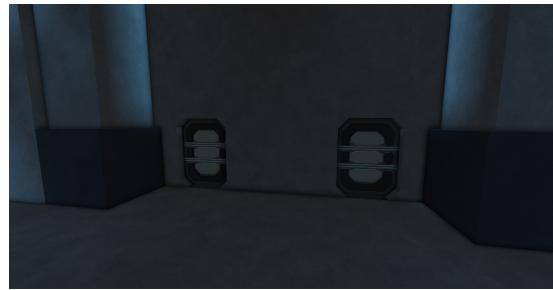


Figure 11: salles supplémentaire

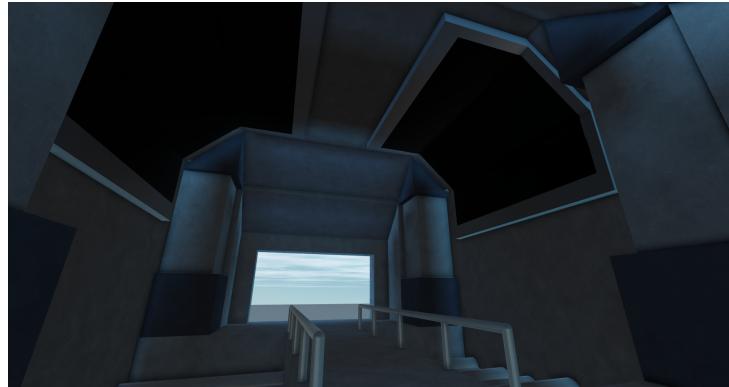


Figure 12: salle de transition

2.3 Mécaniques de jeu

2.3.1 Génération de salles

Les joueurs peuvent se déplacer dans des salles générées pseudo-aléatoirement que nous avons commencé à créer. Pour réaliser un tel système, nous avons choisi d'opter pour la solution de coller les portes d'entrée et de sortie de chaque salle. Cependant, en faisant ceci, nous avons rencontré un problème majeur : Le premier et principal étant qu'ayant des salles tournantes, il arrivait qu'elles fassent une boucle se générant ainsi l'une dans l'autre. Une solution à ce problème a été d'enregistrer le "statut" de la salle précédente (si elle tournait à droite ou à gauche). Ainsi, les salles se mettent toujours dans le bon sens et ne font pas de boucles.

Nous avons aussi souhaité avoir certaines salles plus présentes que d'autres. Pour ce faire, une solution a été d'ajouter un "poids" à chaque salle afin de déterminer sa rareté. Moins ce dernier est élevé, plus la salle est rare.

De plus, certaines salles montent ou descendent également. Pour éviter de donner l'impression au joueur de toujours monter ou descendre, une solution similaire à celle des salles tournantes a été mise en place. La valeur indiquant si la salle précédente changeait de niveau est enregistrée. Si c'est le cas, la nouvelle salle générée ne peut pas être sélectionnée, et une autre est choisie à la place.

Pour développer cette fonctionnalité, nous avons d'abord créé de petites salles de test qui nous ont permis d'implémenter toutes les fonctionnalités sans avoir à concevoir des salles définitives. Une fois l'algorithme de génération terminé et les problèmes résolus, nous avons pu intégrer les salles finales.

Avec ces dernières, nous avons ajouté des portes qui, pour certaines salles, s'ouvrent à l'approche du joueur, et pour d'autres, à la saisie d'un code sur un clavier ou au passage d'une clé.

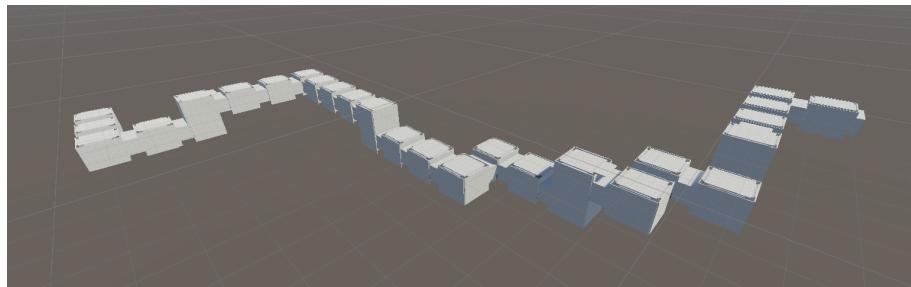


Figure 13: Exemple de génération de 25 salles

2.3.2 Joueur 1

Certaines fonctionnalités du joueur 1 ont été retravaillées afin d'améliorer son comportement et son interaction avec l'environnement. Les principales améliorations incluent :

- **Animations :** Les animations du personnage ont été rendues plus fluides et complètes. L'utilisation des **Blend Trees** permet une transition plus naturelle entre les différents états d'animation
- **Mouvement et physique :** Le système de déplacement a été optimisé : Les sauts sont plus intuitifs et immédiats à l'Input.
- **Objets interactifs :** Le joueur peut désormais collecter divers objets, les sélectionner depuis l'inventaire et les utiliser en fonction du contexte du jeu. Le seul objet pour l'instant disponible étant un grappin qui donne une nouvelle possibilité de déplacement au joueur.

2.3.3 Joueur 2

Le code a été repensé afin de permettre au joueur 2 de contrôler facilement plusieurs types d'objets, en mettant en place une architecture plus flexible et scalable.

- **Système de contrôle généralisé :** Une nouvelle classe abstraite, `Controllable`, a été introduite. Tous les objets pouvant être contrôlés par le joueur 2 héritent désormais de cette classe, garantissant une gestion uniforme des entrées et des interactions.
- **Exemples d'objets contrôlables :** Grâce à cette refonte, le joueur peut prendre le contrôle de différents types d'objets, comme :
 - **La caméra :** Permettant au joueur d'avoir une vue dynamique et ajustable sur l'environnement.
 - **Le drone terrestre :** Un véhicule télécommandé que le joueur peut déplacer et utiliser pour explorer des zones inaccessibles.
- **Flexibilité et évolutivité :** Ce système facilite l'ajout de nouveaux objets contrôlables à l'avenir, en assurant une intégration simplifiée au sein de l'architecture du jeu.

2.4 Multijoueur

La première soutenance avait pour point focal **l'implémentation du multijoueur en ligne**. Le fonctionnement était le suivant :

- Un **joueur hôte** initie une session, se connectant à un **serveur relais via Unity Relay Service**, accédant ainsi à la scène de **Lobby**.
- Un **deuxième joueur** pouvait rejoindre la session en utilisant le **code à 6 caractères** affiché sur l'interface du joueur hôte.
- Un **troisième joueur optionnel**, en tant que **spectateur**, pouvait également se connecter de la même manière.
- Une fois **tous les joueurs déclarés prêts**, la partie peut être lancée et les transporte vers la **scène de jeu principale**.

2.4.1 Problèmes identifiés et corrections

- La scène de lobby ne supportait pas la déconnexion des joueurs.
→ Ajout d'une gestion de la déconnexion dans la scène Lobby.

- La scène de menu principal ne supportait pas l'entrée de code erroné.
→ Implémentation d'une logique permettant de réessayer après entrée du code erroné.
- Pas de possibilité de jouer en local (LAN).
→ Ajout du mode de connexion LAN pour une expérience plus fluide.
- L'interface du menu et du lobby étant répartie sur 2 scènes manque de fluidité et n'est pas intuitive à utiliser.
→ Réunion des deux scènes de connexion en une seule (`StartMenu.scene`) avec une meilleure logique sur les interfaces graphiques.
- Bug : le joueur 2 prenait beaucoup de temps à se connecter (lié à la fréquence d'envoi de paquets).
→ Correction du bug en ajustant la fréquence d'envoi des paquets.
- Autres bugs mineurs liés à la synchronisation en jeu, aux changements de scène, et à la logique côté serveur.
→ Corrections et optimisations diverses.

2.4.2 Fonctionnement actuel

La scène de départ `StartMenu` propose désormais deux options de connexion : **LAN** et **Online**.

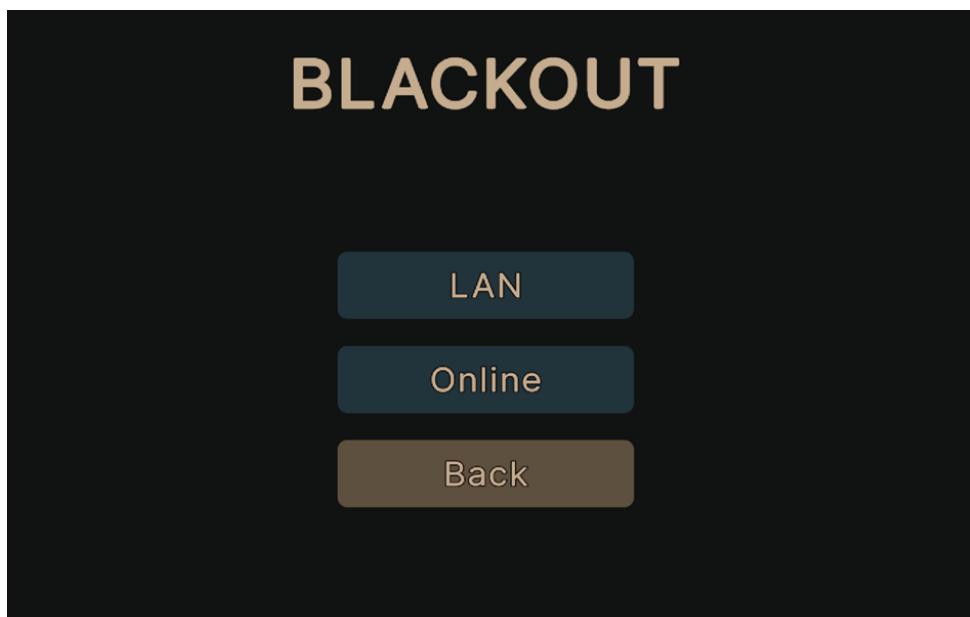


Figure 14: Interface du menu de connexion

L'option **LAN** permet la connexion grâce à une **adresse IP**, tandis que **Online** requiert le **code de connexion au relais**. Une fois la connexion établie, l'interface est

identique pour les deux modes : un **Lobby** où les joueurs peuvent se déclarer prêts et où l'hôte de la session peut lancer la partie.

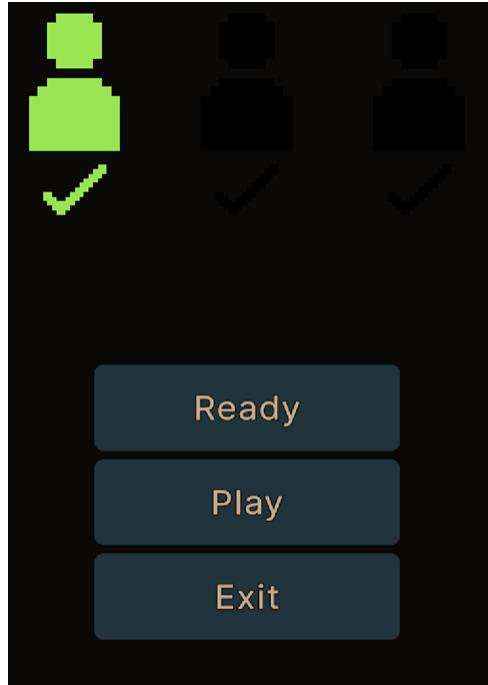


Figure 15: Note : Le jeu ne permet le lancement de la partie qu'avec au moins 2 joueurs.

2.4.3 Logique multijoueur

La synchronisation des événements en jeu est assurée de deux manières :

Le Network Object : Il définit un objet synchronisé sur le serveur. Celui-ci est sous l'autorité d'un client, et les modifications qu'il subit pendant le jeu sont répliquées sur le serveur et donc synchronisées sur tous les autres clients.

Le **Network Object** permet de définir des variables de certains types côté serveur (`NetworkVariable`), dont certaines sont prédéfinies par `Netcode for GameObjects` comme :

- `NetworkTransform` : synchronisation de la position.
- `NetworkAnimator` : synchronisation des animations.
- `NetworkRigidbody` : synchronisation de la physique des objets.

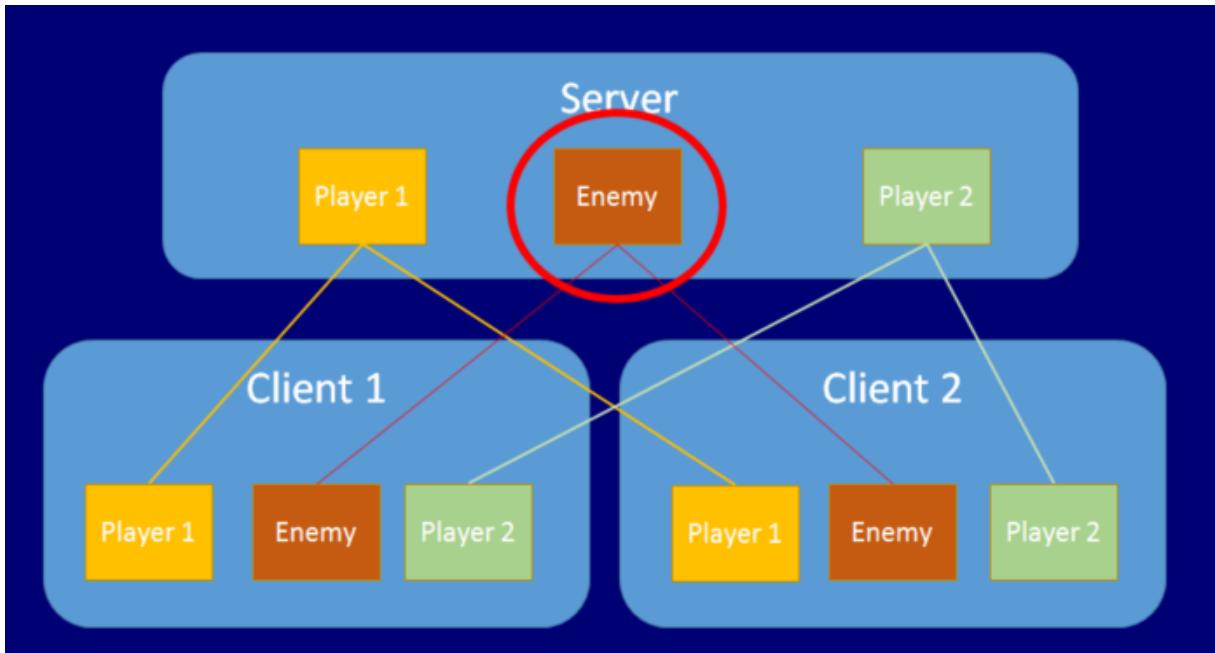


Figure 16: NetworkObjects et Autorité

Autorité : Si l'autorité d'un NetworkObject est laissée par défaut, il est **Server Authoritative**, à savoir que son état ne peut changer que sur Serveur. On peut réassigner l'autorité sur un objet à un client pour que l'état considéré dans la synchronisation soit celui de l'instance de l'objet sur le client en question.

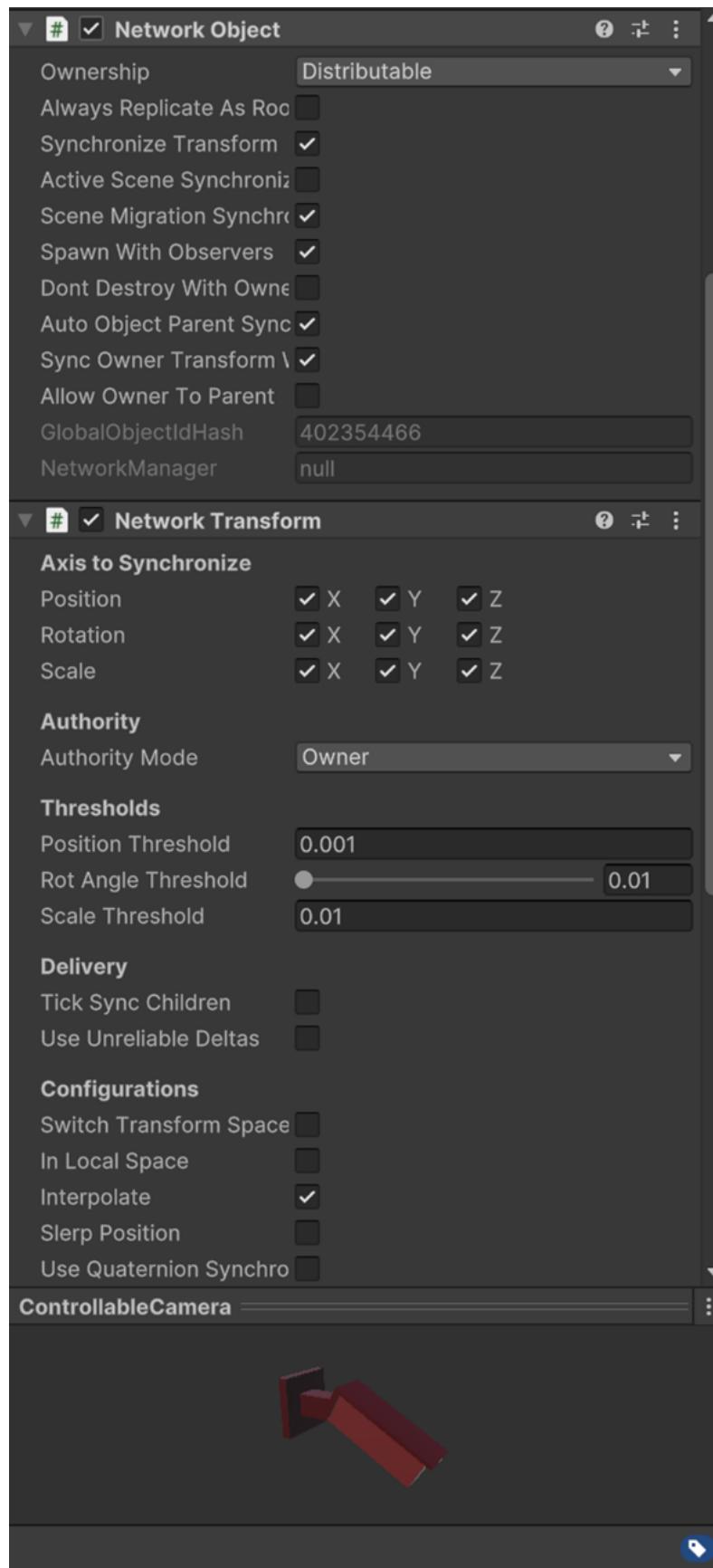


Figure 17: Exemple d'un Network Object et d'un Network Transform

Objet de la caméra contrôlable par le joueur 2 : Le composant NetworkObject

permet de contraindre son utilisation au joueur 2 et le `NetworkTransform` synchronise sa position et sa rotation.

Les RPC (Remote Procedure Call) : Il s'agit de méthodes pouvant être appelées par un client (`ServerRPC`) et exécutées sur le serveur, ou vice versa (`ClientRPC`). Elles permettent par exemple d'instancier des objets sur tous les clients sans avoir à les transmettre directement via le serveur (un `GameObject` n'étant pas transférable sur le serveur pour des raisons de performances).

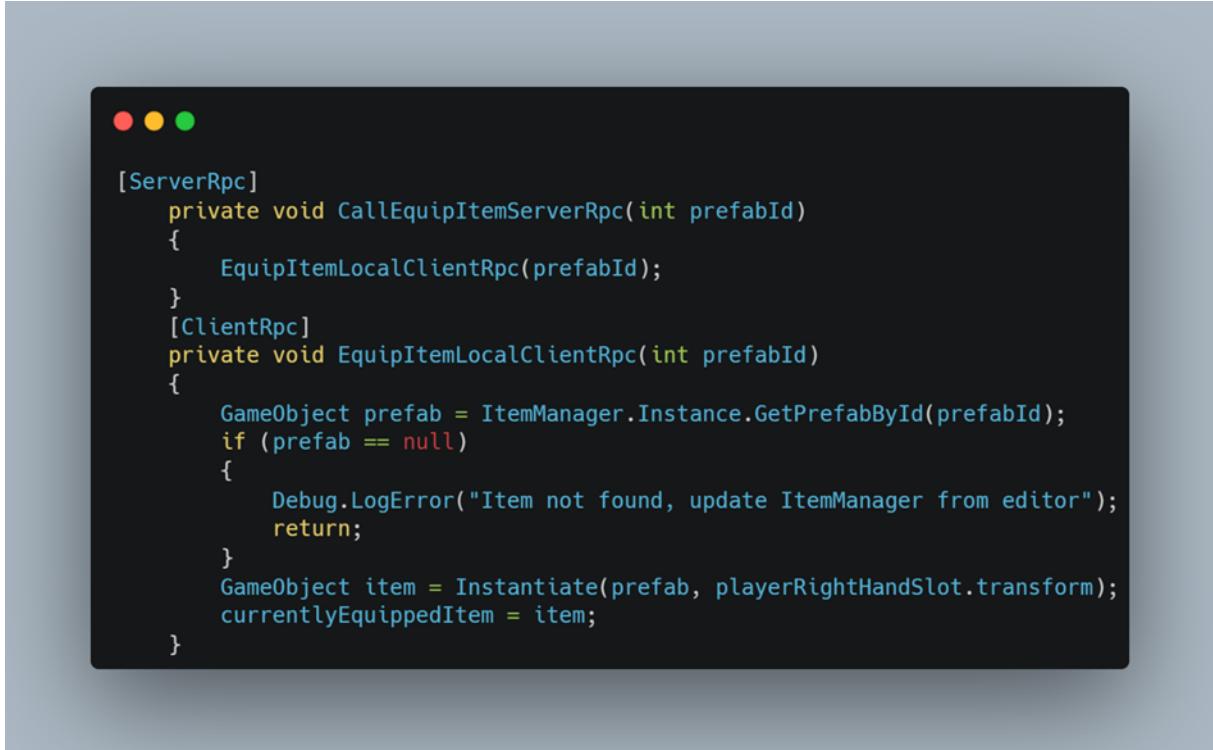


Figure 18: Utilisation des RPC pour synchroniser l'équipement

Exemple d'utilisation d'une ServerRPC faisant appel à une ClientRPC pour assurer que l'objet équipé par le joueur 1 apparaisse également sur l'affichage du joueur 2. Ici, `prefabId` permet d'identifier un certain `GameObject` défini à la compilation dans le singleton `ItemManager`. La méthode `EquipItemLocalClientRPC(int prefabId)` permet d'instancier l'objet sur tous les clients.

2.5 Intelligence Artificielle

2.5.1 Objectif de l'IA

Le but de nos intelligences artificielles est d'apporter une complexité qui dépasse la simple réflexion nécessaire à la résolution des énigmes. Nous souhaitons que notre jeu ne se limite pas à un enchaînement de puzzles, mais qu'il propose également des phases de tension et de poursuite, créant ainsi une expérience plus immersive et plus impactante

émotionnellement.

En combinant des épreuves de réflexion avec des séquences de traque, nous pouvons générer des situations angoissantes qui altèrent la capacité des joueurs à réfléchir sereinement. Cette pression psychologique amplifie la difficulté des énigmes, même les plus simples, en introduisant une dynamique où chaque hésitation peut s'avérer fatale.

Une telle approche permet de plonger le joueur en immersion, où la peur de l'échec ou de la capture intensifie les émotions, enrichissant ainsi l'expérience globale du jeu.

2.5.2 Utilisation du Package AI Navigation.

Concernant l'implémentation de l'intelligence artificielle, nous nous servons du package AI Navigation qui permet l'implémentation d'un système de *pathfinding* de manière efficace. Ceci est réalisé en construisant (à l'aide de ce package) une surface servant à déterminer où peuvent se déplacer les IA ennemis.

2.5.3 Création de NavMesh

Pour créer une *NavMesh*, qui définit les zones de déplacement des intelligences artificielles (IA) dans Unity, le principe de base repose sur l'utilisation d'une surface navigable et d'obstacles. Ce fonctionnement est relativement simple tant que les déplacements se font sur un plan horizontal.

Toutefois, la complexité augmente lorsqu'il s'agit de permettre aux agents de se déplacer sur des murs ou même des plafonds. Dans ce cas, il est nécessaire d'adapter l'orientation de la surface sur laquelle la *NavMesh* est générée. De plus, pour assurer la transition entre différentes zones de navigation (que ce soit du sol vers un mur ou d'un mur vers un plafond), il est nécessaire de se servir des *NavMesh Links*, des liens jouant le rôle de passerelles.

2.5.4 Détection du Joueur

L'implémentation de la détection repose sur deux systèmes complémentaires :

- **Zone de vision** : définie par un cône de vision basé sur des vecteurs, un angle et un rayon. Un *Raycast* est utilisé pour détecter les obstacles et assurer une détection réaliste.
- **Zone de détection de déplacement** : une sphère détecte les mouvements du joueur. Son rayon varie en fonction de la vitesse du joueur (marche, course, accroupissement). Si le joueur reste hors de portée suffisamment longtemps, l'ennemi l'oublie progressivement.

Ce double système empêche le joueur d'exploiter une seule faille et oblige à une approche plus stratégique.

Un problème rencontré concernait la zone de détection initialement représentée par un disque, limitant la détection aux surfaces planes. Une sphère a été adoptée pour une meilleure gestion des interactions, notamment pour les attaques verticales.

2.5.5 Machine à États Finis (State Machine)

Nous avons implémenté notre propre *State Machine* afin d'en améliorer la maîtrise et d'en adapter précisément le comportement.

2.5.6 Liste des États

- **Patrol** : état par défaut, l'IA suit un parcours défini.
- **Hunt Down** : activation lorsque le joueur est détecté, déclenchant une poursuite.
- **Investigate** : après perte de vue du joueur, l'IA cherche activement dans la zone.
- **Attack** : si le joueur est suffisamment proche et accessible.
- **Run Away** : l'IA prend la fuite si sa programmation l'impose (exemple : fin d'attaque).
- **Ambush** : l'IA prépare une attaque surprise basée sur un pattern spécifique.

Les états les plus couramment rencontrés sont la patrouille et la recherche du joueur après une détection incomplète.

```
void Start()
{
    target = null;

    _currentNode = _nodes[_currentNodeIndex];
    var patrolState = new EnemyPatrolState(this, animator);
    var huntDownState = new EnemyHuntDownState(this, animator);
    var investigateState = new EnemyInvestigateState(this, animator);

    At(patrolState, huntDownState, new FuncPredicate(()=>fieldOfView.Spotted));
    At(huntDownState, patrolState, new FuncPredicate(()=>!fieldOfView.Spotted && lastPlayerPositionVisited));
    At(huntDownState, investigateState, new FuncPredicate(()=>fieldOfView.Spotted && !isHeard && (!agent.hasPath || agent.velocity.sqrMagnitude == 0f));
    At(investigateState, patrolState, new FuncPredicate(()=>!isInvestigating));
    At(investigateState, huntDownState, new FuncPredicate(()=>fieldOfView.Spotted || isHeard));

    stateMachine.SetState(patrolState);
}
```

Figure 19: Exemple d'utilisation de la State Machine

2.5.7 Implémentation des IA

Toutes les IA héritent d'une classe abstraite `Enemy`. Il s'agit d'une base commune pour tous les ennemis du jeu qui définit leurs comportements généraux comme la patrouille, la chasse ou l'attaque, tout en intégrant le système d'état pour gérer leurs transitions. Chacune des IA possède des attributs spécifiques pour la vision et la détection de mouvement.

2.5.8 Types d'IA

Nous avons créé plusieurs intelligences artificielles faisant office d'ennemis et recherchant le joueur :

- **Le Rodeur** : Suit un schéma de déplacement prédéfini à l'aide de noeuds qui déterminent ses destinations. Il traque le joueur dès qu'il le détecte, que ce soit par la vision ou par une zone de détection des mouvements. Si le joueur disparaît de son champ de vision, il effectue une recherche autour de lui pour vérifier s'il ne s'est pas simplement caché contre un mur. Si la traque aboutit, il attaque immédiatement. Dans le cas contraire, il reprend sa patrouille.
- **Le Gardien** : Une tourelle immobile qui tire sur le joueur lorsqu'il entre dans son champ de vision. Une fois le joueur hors de vue, la tourelle reste momentanément immobile, prête à tirer si celui-ci tente de se montrer à nouveau trop rapidement.
- **Garlaz** : Une autre variante de tourelle qui tire un laser en continu tant que le joueur est à portée d'attaque. Son fonctionnement est similaire à celui du *Gardien*, mais au lieu de tirer des projectiles individuels, il maintient un faisceau laser constant.
- **Rotten Slime** : Un ennemi initialement immobile, suspendu au plafond, attendant que le joueur passe en dessous pour lui tomber dessus et l'attaquer. Si la traque échoue et que le joueur devient inatteignable, il adopte un comportement de patrouille aléatoire afin de tenter de le retrouver.
- **Spikey** : Un ennemi qui se déplace sur les plafonds à la recherche du joueur, à l'image du *Rodeur*. Lorsqu'il détecte sa cible, il lui fonce dessus en lui sautant dessus. Après l'impact, il reste immobilisé au sol pendant deux secondes avant de repartir vers le plafond et reprendre son comportement initial.

2.5.9 Défis Techniques

Gestion des Missiles : Les projectiles traversaient initialement les obstacles. L'ajout d'un *Rigidbody* a corrigé ce problème, mais a introduit un effet de rebond indésirable. Des ajustements de physique ont permis d'obtenir un comportement réaliste.

Attaques Aériennes : L'utilisation de **NavMesh** pour le déplacement posait problème dans l'air. Désactiver le **NavMesh** temporairement causait des erreurs ou permettait à l'ennemi de traverser le sol. Un calcul d'altitude optimisé a été nécessaire.

State Machine Avoir de très nombreux états facilitent la manière de gérer notre IA, en revanche en avoir de nombreux implique de devoir bien naviguer entre eux. Ainsi, plusieurs heures ont été perdu à comprendre pourquoi un état était déclenché en priorité par rapport à d'autres.

Création de NavMesh : Pour notre jeu, la création de surfaces de NavMesh directement dans la scène est une possibilité qui s'avère impossible. En effet, celui-ci repose sur un principe de génération aléatoire de salles, ce qui empêche d'avoir une scène contenant toutes les salles avant même le lancement du jeu. Il a donc été nécessaire de trouver un moyen de construire la NavMesh dynamiquement en jeu.

Pour ce faire, nous créons séparément les surfaces de NavMesh pour chaque salle. Ensuite, lors de la génération de la carte en jeu, nous les activons. Si ce principe semble relativement simple en théorie, sa mise en œuvre est bien plus complexe. Si aucun problème n'est rencontré du côté des surfaces, il en va autrement pour les NavMesh Links. Bien qu'ils soient présents dans la scène, ils ne sont pas activés.

Egalement, la surface de beaucoup de salles n'étant pas planes, les surfaces de NavMesh ne les lie pas entre elles. Il faut donc rajouter manuelle un considérable nombre de NavMesh Links pour avoir une surface praticable, ce qui demande un temps important.

Les différentes solutions testées pour résoudre ce problème ont conduit à des complications inattendues, telles que la désactivation involontaire des surfaces ou la disparition du sol.

2.5.10 Système de Gestion des Dégâts

Deux interfaces ont été définies :

- Une interface pour les objets pouvant infliger des dégâts.
- Une interface pour les objets pouvant en subir.

Cela permet une gestion flexible des interactions entre projectiles, joueurs et ennemis.

2.5.11 Bilan de l'IA actuelle par rapport à la précédente

Auparavant, il nous manquait ce principe de State Machine avancé qui nous permet un travail organiser en ce qui concerne nos IA. Notre système de détection a été amélioré en prenant en considération le déplacement du joueur lorsqu'il se déplace, ce qui correspond au rayon d'écoute de notre IA. Egalement quatre autres IA ont été rajouté permettant ainsi une plus grande variété d'ennemis.

2.5.12 Les projets futurs

En ce qui concerne le travail, le plus gros a été fait. Il est nécessaire de s'assurer du bon fonctionnement de tout et de régler le souci lié à la construction en jeu des salles. L'ajout d'une ou deux intelligences artificielles supplémentaires est une option envisageable, bien que non essentielle. Cela pourrait enrichir l'expérience de jeu, mais notre priorité reste d'optimiser celles déjà en place. Désormais, il vaut mieux se concentrer sur la relation entre

l'IA et le joueur, plutôt que sur l'IA elle-même, qui est déjà suffisamment aboutie pour un jeu de notre genre. Cela implique de modifier le comportement du joueur lorsqu'il meurt, est repoussé par une attaque ou subit d'autres événements similaires. Enfin, l'intégration de modèles 3D détaillés et d'animations fluides sera essentiel pour donner vie à nos intelligences artificielles. Cela permettra non seulement de rendre nos ennemis plus immersifs, mais aussi de susciter des émotions plus intenses chez nos joueurs, renforçant ainsi l'expérience et l'immersion lors de leurs parties.

2.6 Level Design

2.6.1 Conception Des Enigmes

Beaucoup d'énigmes ont et auront pour but de retrouver un code, permettant ainsi de déverrouiller la porte vers la salle suivante. Il existe plusieurs implémentations possibles, telles que d'afficher des morceaux de code sur différents éléments de la salle, ou encore inscrire le code sur une feuille que le joueur pourrait récupérer, ce qui est rendu possible grâce à l'ajout du système d'**inventaire**. Il faut donc un moyen d'afficher du texte sur une surface, cela est rendu possible grâce aux **TextMeshPro** ou TMP, que nous utiliserons pour afficher du texte en 2D sur une surface 3D (voir figure ci-contre).



Figure 20: Affichage de texte sur une surface via TextMeshPro

2.6.2 Bilan du Level Design

Le travail de Level Design a grandement évolué :

- D'une part, l'ajout d'un nombre conséquent d'IA et de fonctionnalités telles que l'inventaire nous offre de plus en plus de possibilités quant à la conception d'énigmes et de salles. En effet, au début du développement de Blackout, le Level Design était en grande partie (si ce n'est entièrement) **théorique** : les salles et ennemis étaient simplement représentés schématiquement sans jamais vraiment avoir la certitude

qu'ils seraient implémentés un jour. À présent, nous disposons de nombreux outils que nous pouvons utiliser et implémenter à notre guise.

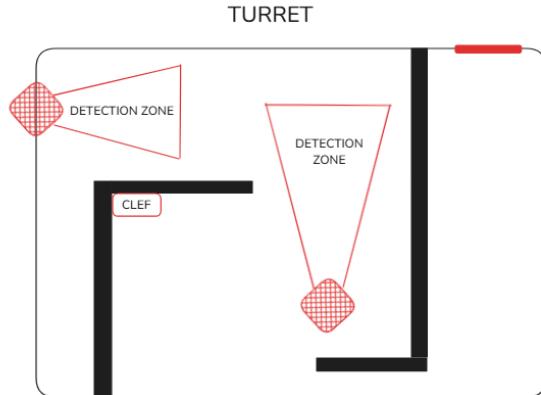


Figure 21: Schéma d'éénigme réalisé au début du développement du jeu.

- Cependant, tous les projets issus du développement initial du jeu n'ont pas forcément pu être implémentés, soit parce qu'ils étaient trop ambitieux, difficilement réalisables, ou bien parce que leur apport au jeu serait moindre.

Un des exemples principaux est le labyrinthe (The Maze) sur la figure ci-contre. Les portes rouges représentent les entrées et sorties du labyrinthe, tandis que les portes noires représentent les portes par lesquelles il est possible de relier chaque salle entre elles. Les 9 salles seraient dans le désordre, et le Support aurait eu pour but de les réorganiser afin que l'Agent puisse se frayer un chemin. L'implémentation d'une telle énigme aurait été trop complexe : comment faire fonctionner le système de déplacement des salles ? comment s'assurer que l'entièreté des objets et ennemis des salles intermédiaires soient bel et bien déplacés eux aussi ? que se passerait-il si l'on déplaçait une salle dans laquelle l'Agent se situe ?

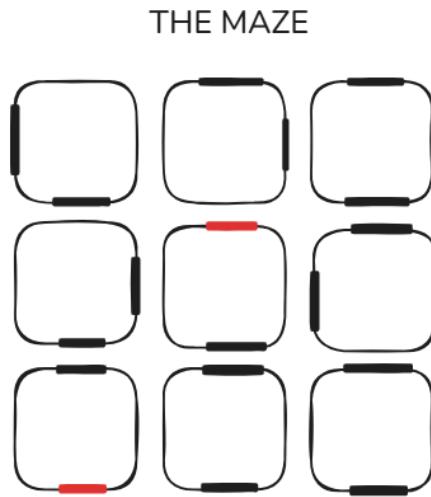


Figure 22: Schéma de l'énigme "The Maze", dont l'idée a été abandonnée.

2.6.3 Projets Futurs

À l'avenir, notre objectif principal sera d'implémenter un nombre conséquent de salles, sachant que chacune d'entre elles devra représenter une expérience la plus unique possible pour les joueurs, c'est-à-dire que chaque défi proposé au joueur se doit d'être différent d'une manière ou d'une autre. C'est un des aspects du jeu sur lequel nous mettons un point d'honneur à respecter.

Par exemple, supposons que l'Agent ait pour objectif de chercher un élément quelconque (clé, code...) dans la salle :

- Si l'on ne souhaite que **partiellement** ralentir le joueur, il serait possible d'ajouter une tourelle dans la salle, forçant ainsi le joueur à ne pas rentrer dans son champ de vision. Cela aurait pour conséquence de simplement **limiter** les déplacements du joueur.
- Si l'on souhaite freiner la progression du joueur **entièrement**, il serait judicieux de placer un ennemi dont le but est de poursuivre le joueur s'il le détecte. Le joueur devrait alors **constamment** rester sur ses gardes afin de s'assurer que les ennemis en patrouille ne le repèrent pas.

3 Conclusion

3.1 Bilan sur l'avancement

Depuis la première soutenance, nous avons bien avancé sur le développement du jeu. Nous avons finalisé plusieurs interfaces clés comme l'inventaire et le terminal interac-

tif, amélioré la gestion du multijoueur et rendu le système de génération de salles plus fluide. L'intégration des IA a aussi beaucoup progressé, rendant le gameplay bien plus **dynamique**.

Parmi les principales avancées, nous pouvons citer :

- La finalisation des interfaces essentielles pour une meilleure expérience utilisateur.
- L'amélioration du système de génération procédurale des salles et la gestion des déplacements des IA.
- L'optimisation du multijoueur avec un mode LAN et une synchronisation plus fluide.
- L'ajout d'intelligences artificielles avec des comportements plus travaillés.

Ces progrès nous ont permis de rendre le jeu plus stable et jouable, même si certains aspects restent à ajuster.

3.2 Perspectives et futur du projet

Par la suite, notre but est d'avoir une version jouable plus aboutie avec une expérience fluide et immersive. Les prochaines étapes importantes sont :

- L'ajout de salles et d'énigmes afin d'**enrichir** davantage le gameplay.
- Travailler sur les interactions entre les IA et les joueurs pour les rendre plus **fluides**.
- Optimiser encore les performances et corriger les bugs restants.
- Faire tester le jeu à des joueurs extérieurs pour avoir des retours concrets.

Pour y arriver, nous comptons nous organiser avec un planning plus précis afin d'éviter de perdre du temps sur des détails annexes et se concentrer sur les aspects clés du jeu.

3.3 Réflexion et apprentissages

Cette phase de développement nous a permis d'apprendre beaucoup sur la gestion de projet, le développement en équipe et la résolution de problèmes techniques. Nous avons observé l'importance d'une bonne organisation et d'une communication efficace pour avancer plus rapidement.

Nous avons par ailleurs pris conscience que l'anticipation des difficultés techniques est essentielle, surtout sur des aspects tels que la synchronisation en multijoueur et la génération de salles.

En conclusion, malgré les défis rencontrés, nous sommes fiers du chemin parcouru et motivés pour la suite. On a maintenant une base solide et les idées claires pour mener ce projet jusqu'au bout et proposer une expérience de jeu réussie.

En conclusion, bien que cette première phase ait présenté des défis, elle constitue un **socle prometteur** pour la suite. Nous sommes désormais mieux préparés à mener ce projet à son terme et à livrer un jeu captivant et de qualité, fidèle à notre vision initiale.