Rapport de soutenance n°2

# Uzin

*Bart Entreprises*
*Mars 2025*

Tristan DRUART (*Chef d'équipe*)
Martin LEMEE
Cyril DEJOUHANET
Maxan FOURNIER

## Table des matières

# 1 Introduction

## 1.1 À propos de notre projet

Notre projet consiste à développer *Uzin*, un jeu vidéo où le joueur incarne un personnage se trouvant sur une planète inconnue. Le gameplay débute par la récolte manuelle de ressources dispersées sur la carte. Progressivement, le joueur pourra construire des infrastructures pour automatiser ces processus, tout en gérant une production énergétique évolutive. Les ressources, initialement simples, se complexifient avec le temps, exigeant la conception de machines avancées et une gestion stratégique accrue. Ce système progressif offre un mélange captivant de collecte, de construction et d'optimisation.

Cependant, la planète ne reste pas passive : l'environnement réagit à l'activité du joueur, mobilisant une faune hostile et une flore défensive. Inspiré du genre *tower defense*, le joueur devra construire des structures pour protéger ses infrastructures tout en explorant et en automatisant davantage. Grâce à cette combinaison de gestion, d'automatisation et de survie, *Uzin* promet une expérience stratégique immersive où chaque décision façonne le déroulement du jeu.

## 1.2 Le rôle de ce rapport de soutenance

Ce rapport de soutenance présente de manière détaillée l'ensemble des travaux réalisés depuis la validation du cahier des charges. Il retrace les avancées concrètes, les obstacles rencontrés et les solutions mises en œuvre pour respecter nos objectifs. Chaque section met en lumière le rôle de chaque membre de l'équipe, en décrivant précisément leurs contributions respectives, tout en analysant les éventuels écarts par rapport au planning initial. En somme, il s'agit d'une base essentielle pour évaluer le progrès de notre projet et ajuster nos priorités si nécessaire.

En plus du bilan des réalisations, notre rapport anticipe les étapes à venir en définissant clairement les tâches restantes et les responsabilités associées pour chaque membre de l'équipe. Il offre une vision structurée des objectifs à atteindre d'ici la prochaine soutenance, en tenant compte des leçons apprises et des ajustements requis pour maintenir une progression optimale. Ce document permet un suivi transparent et rigoureux de l'avancement du projet.

# 2 Découpage et avancement du projet

## 2.1 Répartition des tâches

Pour rappel, nous avons réparti la création du projet de manière équitable, en tenant compte des compétences spécifiques de chaque membre. Toutefois, chacun contribue à la programmation générale du jeu, de manière à ce que tout le monde puisse collaborer.

| | Tristan | Martin | Cyril | Maxan |
|---|---|---|---|---|
| Intelligence artificielle | Assistant | | | Responsable |
| Histoire et storytelling | | Responsable | Assistant | |
| Design | | Assistant | Responsable | |
| Game Design | Responsable | Assistant | | |
| Réseau | Responsable | | | Assistant |
| Site web | | | Assistant | Responsable |

## 2.2 Avancement du projet

| **Tâches** | **Soutenance 1** |
|---|---|
| Intelligence artificielle | 50% |
| Histoire et storytelling | 80% |
| Design | 70% |
| Game design | 80% |
| Réseau | 40% |
| Site web | 90% |

TABLE 1 – Le planning initial, présent dans le cahier des charges fonctionnel

Les objectifs fixés dans le cahier des charges initial ont globalement été respectés, à l'exception de l'intelligence artificielle. Ce retard s'explique principalement par notre volonté de concentrer les ressources sur le développement des mécanismes de jeu et du mode multijoueur. La mise en place d'un système multijoueur robuste a représenté un défi technique majeur qui a mobilisé l'essentiel de notre temps et de nos efforts. Ainsi, l'intégration de l'IA a été repoussée à une phase ultérieure, afin de pouvoir l'enrichir et l'ajuster en fonction des retours sur la jouabilité et de l'évolution des fonctionnalités multijoueur.

# 3 Petit rappel de l'histoire du jeu

## 3.1 L'importance de l'histoire dans Uzin

Le lore d'*Uzin* n'est pas un simple ajout esthétique ou narratif au jeu : il constitue le pillier de l'expérience immersive que nous souhaitons offrir. Dès le début du projet, nous avons compris que l'histoire du jeu devait servir de fil conducteur, donnant un sens à chaque aspect de gameplay et à chaque choix de design. Dans un jeu de gestion comme *Uzin*, où le joueur doit collecter, automatiser, et défendre ses installations, une trame narrative forte permet de renforcer son engagement et de lui offrir une perspective émotionnelle et intellectuelle sur ses actions.

L'histoire du jeu agit également comme un outil pour connecter les joueurs à l'univers que nous avons créé. Elle contextualise leurs efforts, justifie les défis qu'ils rencontrent, et propose une réflexion plus profonde sur des thèmes comme la survie, l'impact humain sur les environnements naturels, et le progrès technologique. Dans un jeu où la solitude et l'isolation jouent un rôle central, le lore vient enrichir cette ambiance en ajoutant une certaine profondeur philosophique. Voici comment nous avons développé l'univers et les thématiques centrales du jeu.

## 3.2 Le contexte narratif : un univers dystopique

### 3.2.1 Un monde en crise

L'histoire se déroule en l'an 2287, dans un contexte où l'humanité a atteint un sommet démographique sans précédent : 42,7 milliards d'habitants. Cette explosion de population a poussé la Terre au bord du gouffre, ses ressources s'épuisant à un rythme alarmant. Incapable de soutenir cette densité humaine, l'humanité a été contrainte de trouver des solutions radicales pour survivre.

C'est dans ce contexte que le programme d'exploration interstellaire a vu le jour. Inspiré des anciens services militaires obligatoires, ce programme impose à chaque individu, dès leur majorité, de quitter la Terre pour participer à des missions d'extraction de ressources sur des planètes lointaines. Cette mesure drastique illustre un monde où le progrès technologique cohabite avec des décisions impitoyables dictées par la nécessité.

### 3.2.2 Une découverte révolutionnaire

Au cœur de ce programme se trouve une matière organique exceptionnelle qui a révolutionné le voyage spatial. Cette substance, utilisée comme carburant

pour des réacteurs ultra-performants, permet d'envoyer des milliers d'explorateurs à travers l'espace. Cependant, cette avancée cache une réalité sombre : chaque explorateur reçoit uniquement de quoi financer un aller simple vers la planète d'extraction. Le retour sur Terre est conditionné à la réussite de leur mission, faisant de chaque voyage un pari risqué.

## 3.3 Thèmes centraux et ambiance narrative

### 3.3.1 Un environnement solitaire et brut

L'histoire d'*Uzin* met l'accent sur la solitude du joueur dans un monde hostile. Bien qu'un mode multijoueur soit prévu, les cartes et environnements sont conçus pour refléter une absence totale de vie humaine. Aucune ville abandonnée, aucun vestige de civilisation : tout est brut, naturel, et exempt d'intervention humaine, du moins au début du jeu.

Cette approche renforce l'idée que le joueur est un pionnier, seul face à une planète indomptée. La narration vient appuyer cette ambiance, en racontant l'histoire de ces jeunes envoyés seuls dans l'espace, sans garantie de retour. Cette solitude devient une composante clé du gameplay, influençant les choix et les émotions des joueurs.

### 3.3.2 L'impact humain sur l'environnement

Un thème central de l'histoire est la transformation des environnements naturels par l'action humaine. En commençant sur une planète préservée, les joueurs doivent exploiter les ressources et modifier le paysage pour survivre. Ce choix thématique soulève des questions morales : jusqu'où peut-on aller pour garantir sa propre survie ? Quelles sont les conséquences à long terme de ces actions sur un écosystème ?

## 3.4 Pour conclure sur la narration

L'histoire d'*Uzin* est bien plus qu'un simple cadre narratif : elle est le moteur émotionnel et philosophique du jeu. En mettant en lumière des thèmes comme la solitude, la survie, et l'impact de l'humanité sur la nature, elle enrichit l'expérience des joueurs tout en offrant une réflexion sur des enjeux universels. Ce lore, construit avec soin et en collaboration au sein de l'équipe, constitue une base solide pour soutenir le développement du jeu et captiver son public.

# 4 Design et graphismes

Nous n'avons pas apporté de changements majeurs à l'aspect design et graphismes depuis la dernière soutenance, car il nous a été recommandé de concentrer nos efforts sur le développement du multijoueur et des mécanismes de jeu. Néanmoins, nous tenons à réaliser entièrement nos graphismes en interne pour garantir une cohérence artistique et un rendu unique. C'est pour cette raison que nous avons décidé d'inclure ici quelques explications de notre vision graphique et des premières étapes de conception. Par ailleurs, nous avons designé un menu principal qui intègre déjà quelques options, et qui sera enrichi de fonctionnalités supplémentaires dans les prochaines itérations du projet.

## 4.1 Choix de l'équipement

Dans le cadre de la création graphique de *Uzin*, le choix du logiciel destiné à la réalisation du pixel art a constitué une étape déterminante. Il était essentiel d'opter pour un outil à la fois accessible, performant et compatible avec notre workflow. Nous avons ainsi évalué plusieurs critères tels que l'intuitivité de l'interface, la réactivité des outils de gestion des calques et la capacité d'intégration avec les périphériques tels que la tablette graphique. Après avoir testé plusieurs options, c'est **Pixel Studio** sur *Steam* qui a été retenu.

Nous avons tout d'abord envisagé *Aseprite*, reconnu pour sa spécialisation en pixel art. Bien que ce logiciel propose une large gamme d'outils spécifiques, son interface, bien que fonctionnelle, paraissait vieillissante et ne répondait pas pleinement aux exigences de modernité et de fluidité de notre processus créatif. De plus, le prix relativement élevé, justifié par sa spécialisation, représentait un investissement conséquent pour un projet nécessitant de nombreuses itérations graphiques.

Ensuite, *Photoshop* a été considéré pour sa puissance et sa renommée dans le domaine du graphisme. Toutefois, son caractère généraliste, sa complexité ainsi que le coût de son abonnement mensuel en ont fait une option moins adaptée aux besoins spécifiques du pixel art. La courbe d'apprentissage importante imposée par cet outil est également un frein dans un contexte où l'efficacité et la rapidité sont primordiales.

Quant à *Piskel*, sa gratuité et sa simplicité d'accès en font une option attrayante pour les débutants. Cependant, ses fonctionnalités limitées, notamment en ce qui concerne la gestion avancée des calques et la création d'animations fluides, l'ont rendu inadapté pour répondre aux exigences de qualité et de complexité attendues dans *Uzin*.

En définitive, **Pixel Studio** s'est distingué par sa modernité, sa facilité d'utilisation et ses nombreuses fonctionnalités spécifiques au pixel art. Son interface intuitive et ses outils de gestion des calques ainsi que des animations se sont révélés être des atouts majeurs, en particulier pour un usage optimisé sur tablette graphique. Le rapport qualité-prix attractif et la disponibilité sur *Steam* ont renforcé notre décision, s'inscrivant parfaitement dans notre workflow et nos exigences techniques.

### 4.2 Le design en soi

Le design et les graphismes de *Uzin* jouent un rôle central dans l'immersion du joueur. L'adoption d'une carte générée aléatoirement contribue à renouveler constamment l'expérience de jeu, favorisant ainsi l'exploration et offrant à chaque partie un environnement à la fois unique et varié. Cette approche dynamique permet de maintenir l'intérêt des joueurs sur le long terme, en offrant des situations de jeu imprévisibles et stimulantes.

Visuellement, *Uzin* se distingue par un style coloré et dynamique qui capte immédiatement l'attention. La carte est composée de cases modulaires travaillées avec soin, où le sol aux teintes roses vibrantes s'harmonise avec des lacs d'un mauve profond et un système de rebords dynamiques. Ces derniers s'ajustent automatiquement en fonction de la disposition des tuiles d'eau et de sol, garantissant ainsi une cohérence visuelle remarquable tout en préservant la structure modulaire essentielle au gameplay. L'attention portée aux détails permet d'offrir une expérience visuelle immersive, favorisant l'identification rapide des zones de jeu et la fluidité des interactions.

### 4.3 Le système de calques

Afin d'organiser efficacement l'ensemble des éléments visuels de la carte, un système de calques en trois niveaux a été mis en place. Cette approche hiérarchisée permet une gestion claire et structurée des données et facilite l'interaction entre les différentes composantes du jeu.

— **Le premier calque : le type de sol**
Ce calque définit la nature fondamentale du terrain (herbe, eau, roche, sable, etc.) et influe directement sur les possibilités de déplacement et de construction. Par exemple, une case d'eau ne permet pas le passage ni l'édification de certains bâtiments, alors qu'une case d'herbe peut accueillir des structures simples. Ce niveau sert de base à l'architecture de la carte, établissant les règles premières qui conditionnent le comportement des éléments interactifs.

— **Le deuxième calque : les éléments et ressources**
Ce calque regroupe les objets interactifs et les ressources exploitables, tels que les arbres, les minerais ou les plantations. Il inclut également des caractéristiques environnementales, comme les falaises, les montagnes ou les fosses, qui ajoutent une dimension stratégique au gameplay. En définissant précisément les interactions possibles avec la carte – qu'il s'agisse de l'extraction de ressources ou de la mise en place d'obstacles naturels – ce niveau enrichit l'expérience de jeu tout en offrant une profondeur tactique aux actions du joueur.

— **Le troisième calque : les bâtiments et constructions**
Ce niveau gère la superposition des structures construites sur la carte, en intégrant les informations issues des deux premiers calques. Par exemple, une case d'herbe dotée de ressources spécifiques (comme du bois) pourra accueillir un bâtiment qui exploite ces atouts, tandis qu'une surface rocheuse limitera les options de construction à des structures adaptées aux contraintes du terrain. Ce calque assure ainsi la cohérence et la logique de l'architecture du jeu, garantissant que chaque construction s'insère de manière harmonieuse dans l'environnement prédéfini.

L'intégration de ces trois calques permet de simplifier la gestion globale des éléments de la carte et d'assurer une meilleure lisibilité des interactions possibles :

— **Gestion des déplacements**
La différenciation nette entre les zones navigables et les zones restreintes permet d'interdire le passage sur l'eau ou dans certaines zones escarpées, assurant ainsi une réponse immédiate aux actions du joueur.

— **Logique de construction**
L'organisation en calques permet d'autoriser ou de restreindre la construction en fonction des ressources disponibles et du type de terrain, assurant une cohérence entre l'environnement et les structures édifiées.

— **Modularité**
La structure modulaire du système offre la possibilité d'ajouter facilement de nouveaux types de terrains, ressources ou bâtiments sans bouleverser l'architecture existante, garantissant ainsi une évolutivité continue du jeu.

### 4.4 Une expérience immersive

L'architecture graphique et technique de *Uzin* a été conçue pour offrir une immersion optimale dès les premiers instants de jeu. Chaque détail visuel, qu'il

s'agisse des textures, des couleurs ou de l'animation des éléments, a été pensé pour améliorer la compréhension du gameplay et faciliter la prise de décisions stratégiques. L'harmonie entre les graphismes et les mécaniques de jeu crée une expérience fluide, où l'utilisateur se trouve rapidement plongé dans un univers cohérent et captivant.

Le choix d'une direction artistique marquée, combiné à un système de calques structurant clairement l'univers du jeu, permet de garantir une excellente lisibilité des éléments et de la carte. Ce système offre également la flexibilité nécessaire pour intégrer de futures améliorations, sans compromettre l'équilibre visuel ou fonctionnel du jeu. Ainsi, l'architecture graphique répond à la fois aux exigences esthétiques et techniques, en assurant une navigation intuitive dans un environnement riche et détaillé.

Enfin, il est important de noter que les graphismes de *Uzin* se situent à un niveau de sophistication très élevé, voire parfois au-delà des autres aspects du jeu. Cette richesse visuelle, bien qu'extrêmement attrayante, implique que lors de la soutenance, il ne sera pas possible de présenter en détail l'ensemble des subtilités graphiques développées jusqu'à présent. Cependant, cette avancée témoigne d'une ambition de qualité, qui laisse présager des améliorations futures destinées à harmoniser l'ensemble des composantes du projet.

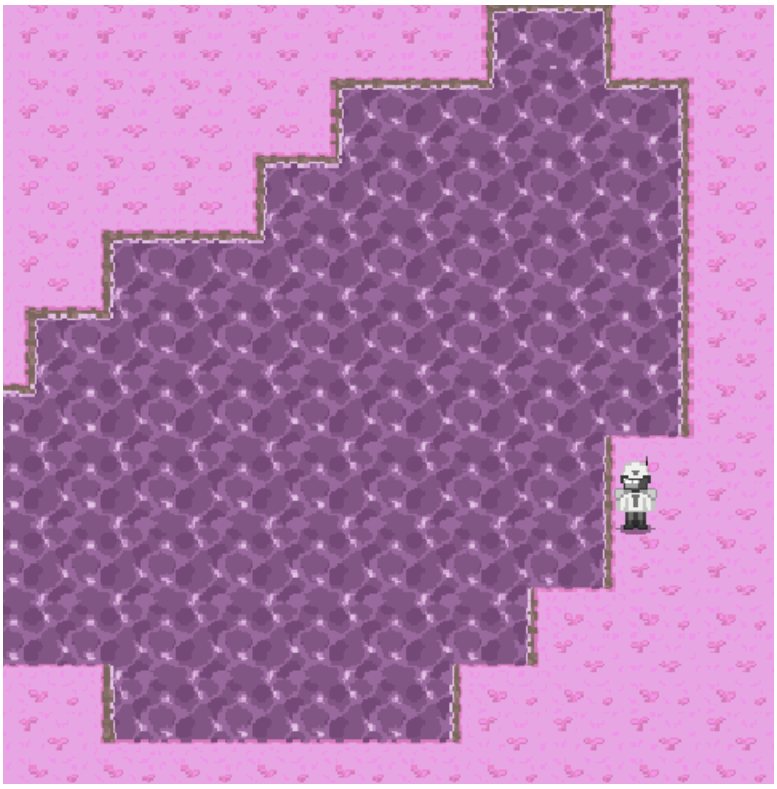

FIGURE 1 – Partie de map avec lac et personnage

# 5 Refonte de la génération de la map

La génération de la carte dans *Uzin* repose sur un système modulaire basé sur des chunks, permettant de charger dynamiquement des zones de la carte en fonction de la position du joueur. Ce procédé se divise en deux grandes étapes : la génération des données de chaque chunk à l'aide du bruit de Perlin et le rendu visuel via une Tilemap. Afin d'assurer une expérience fluide, cohérente et synchronisée en mode multijoueur, nous avons dû affiner et optimiser plusieurs aspects techniques du système.

## 5.1 Création des chunks et génération des données

Chaque chunk est défini par une position dans une grille en deux dimensions et contient un tableau de `TileData` de taille fixe (par exemple, 16 par 16 tuiles). La fonction `GenerateChunk` parcourt chaque cellule du chunk pour calculer, grâce au bruit de Perlin, une valeur d'altitude (ou « height ») qui permet ensuite de sélectionner la tuile correspondante.

Le bruit de Perlin est ajusté par un facteur d'échelle ainsi que par des offsets aléatoires. Ces paramètres garantissent une diversité dans les structures naturelles tout en assurant une certaine continuité visuelle entre les chunks adjacents. Pour renforcer l'immersion, nous avons travaillé à l'harmonisation des variations d'altitude de manière à éviter des ruptures brutales entre zones d'eau, de terre et de bordures. Ce processus itératif a impliqué de nombreux tests, permettant d'affiner les seuils de détection et de mieux simuler des reliefs naturels.

En complément, une attention particulière a été portée sur la gestion de la mémoire et la rapidité d'exécution. La création d'un chunk implique non seulement le calcul de ses tuiles, mais aussi la mise en place d'une structure de données qui permet de les retrouver rapidement lors du rendu ou du déchargement. Cette phase de génération se déroule en arrière-plan, ce qui contribue à éviter des blocages pendant le jeu.

## 5.2 Gestion des transitions et sélection des tuiles

La fonction `GenerateTile` joue un rôle central dans la détermination de l'apparence finale de chaque tuile. Elle ne se contente pas d'évaluer la valeur de bruit à la position courante, mais prend également en compte celles des tuiles voisines, tant orthogonales que diagonales.

Ce système de vérification complexe permet de gérer avec précision les transitions entre différents types de terrains, notamment entre l'eau et la terre.

Par exemple, lorsque la valeur de bruit est inférieure à un seuil critique (ici, 0.2f), la tuile est considérée comme de l'eau. Toutefois, pour éviter des interfaces trop tranchées, des conditions supplémentaires sont appliquées pour sélectionner des tuiles de bord adaptées aux configurations particulières (coins internes, bords hybrides, coins extérieurs, etc.).

L'optimisation de cette fonction a nécessité plusieurs itérations de tests et d'ajustements fins. Chaque nouveau cas testé a permis de peaufiner les conditions de transition, rendant les interfaces entre les zones d'eau et de terre plus naturelles et esthétiques. Ces efforts garantissent que le rendu final est en phase avec l'identité visuelle recherchée pour *Uzin*.

### 5.3 Rendu dynamique sur la Tilemap et synchronisation réseau

Une fois les données d'un chunk générées, la fonction `RenderChunk` se charge de positionner chaque tuile sur la Tilemap en traduisant les coordonnées du chunk vers des coordonnées mondiales. Ce processus assure un rendu dynamique qui s'adapte en temps réel aux déplacements du joueur.

Dans un contexte multijoueur, la cohérence de la carte est primordiale. Pour ce faire, la synchronisation des seeds (via `NetworkVariable`) sur le serveur garantit que tous les joueurs disposent d'une carte identique, même si le chargement des chunks se fait de manière asynchrone. Ce mécanisme de synchronisation repose sur la centralisation de la génération des seeds sur le serveur, évitant ainsi toute divergence dans les valeurs aléatoires qui pourraient autrement provoquer des incohérences visuelles entre les clients.

Le rendu dynamique est également optimisé par un système de mise en cache et de gestion efficace des ressources. En limitant le nombre d'opérations par frame et en actualisant uniquement les zones modifiées, nous avons réussi à minimiser l'impact sur les performances, même lors de déplacements rapides ou dans des environnements très étendus.

### 5.4 Problèmes rencontrés et solutions apportées

#### 5.4.1 Chargement et déchargement des chunks

Initialement, nous avons constaté que certains chunks restaient en mémoire même lorsque le joueur s'en éloignait, ce qui entraînait une surcharge et des ralentissements notables.

Pour résoudre ce problème, nous avons instauré une vérification continue de la position du joueur. Grâce à l'utilisation de la fonction `Vector2Int.Distance`,

un seuil précis a été défini, permettant de décharger automatiquement les chunks situés au-delà de la distance d'affichage définie (*render distance* + 1).

Cette approche a nécessité la mise en place d'une logique de suivi très fine de la position du joueur, ainsi que des tests de performance pour déterminer le seuil optimal. En conséquence, le système de chargement/déchargement s'est révélé être à la fois réactif et efficace, optimisant l'utilisation de la mémoire et garantissant un jeu fluide.

#### 5.4.2 Incohérences dans les transitions de tuiles

La sélection des tuiles pour les transitions entre zones d'eau et de terre posait problème. Dans certaines configurations, la logique initiale ne suffisait pas à choisir la tuile la plus adaptée, ce qui entraînait des bords irréguliers et visuellement décevants.

Après plusieurs tests et retours d'expérience, nous avons affiné la fonction `GenerateTile`. Nous avons étendu l'analyse pour inclure non seulement les voisins directs mais aussi les voisins diagonaux. Cela a permis d'ajuster avec précision les transitions et de sélectionner des tuiles de bord spécifiques pour chaque configuration.

En parallèle, nous avons intégré un système de logs et de visualisation interne, permettant de repérer rapidement les incohérences et de tester de nouveaux seuils. Ces ajustements successifs ont permis d'obtenir des transitions naturelles et harmonieuses, améliorant significativement la qualité visuelle de la carte.

#### 5.4.3 Synchronisation des seeds en mode multijoueur

En mode réseau, une divergence dans les valeurs aléatoires utilisées pour générer le bruit de Perlin (offsets et seeds) pouvait entraîner des cartes différentes entre le serveur et les clients.

Pour pallier ce problème, la génération des seeds a été centralisée sur le serveur. En utilisant des `NetworkVariable` pour `mapSeedX` et `mapSeedY`, le serveur génère les valeurs aléatoires et les propage à tous les clients de manière transparente.

Cette solution assure que chaque client dispose des mêmes paramètres de génération, éliminant ainsi toute possibilité de divergence. Des tests en environnement multijoueur ont confirmé que cette approche permettait une synchronisation parfaite, garantissant une expérience de jeu uniforme pour tous les participants.

#### 5.4.4 Problèmes de performance lors de la génération des chunks

La génération en temps réel de la carte, notamment le calcul du bruit de Perlin pour chaque tuile, représente une charge de calcul non négligeable. Le chargement simultané de plusieurs chunks pouvait ainsi entraîner des baisses de performance, affectant l'expérience utilisateur.

Pour améliorer les performances, plusieurs optimisations ont été mises en œuvre :

— **Réduction des appels redondants :** Les boucles de génération ont été optimisées pour éviter les calculs inutiles. Les paramètres tels que la taille des chunks et les offsets sont calculés une seule fois par chunk, puis réutilisés.

— **Mise en cache des résultats :** Certaines valeurs issues du calcul du bruit de Perlin sont mises en cache, notamment pour les tuiles ayant des voisins identiques ou pour des configurations répétitives. Cela permet de réduire significativement le nombre d'appels à `Mathf.PerlinNoise`.

— **Gestion efficace des ressources :** L'utilisation d'un dictionnaire pour stocker les chunks chargés permet de limiter les opérations de recherche et de mise à jour aux seuls chunks actifs. Par ailleurs, des techniques de pooling d'objets ont été envisagées pour réutiliser les chunks déchargés, évitant ainsi la création et la destruction répétées d'objets.

Ces optimisations ont été validées par des tests de charge et de performance, permettant de maintenir une fluidité acceptable même dans des environnements de grande taille ou lors de mouvements rapides du joueur.

En résumé, la mise en place du système de génération de la carte dans *Uzin* a nécessité un travail d'optimisation et de réglage fin afin d'assurer une expérience fluide et immersive. La gestion dynamique des chunks, combinée à des techniques avancées de sélection de tuiles via le bruit de Perlin, permet de créer un environnement de jeu riche et cohérent, tant en mode solo qu'en multijoueur. Chaque étape, de la génération des données à leur rendu sur la Tilemap, a été minutieusement étudiée et optimisée pour répondre aux exigences de qualité et de performance du projet.

# 6 Multijoueur

Nous avons décidé de prioriser l'implémentation du mode multijoueur dès les premières phases de développement, afin d'éviter de devoir refondre ou adapter ultérieurement l'ensemble du système. Cette approche précoce nous a permis de prendre en compte les exigences réseau dès le début, influençant ainsi la conception globale du jeu.

## 6.1 Choix de l'architecture réseau

Après une étude comparative des différentes architectures disponibles, deux solutions principales s'offraient à nous :

— **Serveur centralisé :** Dans ce modèle, un serveur dédié gère toutes les données du jeu et tous les joueurs se connectent à ce serveur. Bien que cette approche garantisse une synchronisation quasi instantanée et soit idéale pour les jeux compétitifs nécessitant une faible latence, elle présente l'inconvénient de nécessiter une infrastructure serveur payante et plus complexe à maintenir.

— **Peer-to-peer (P2P) :** Dans ce modèle, l'un des joueurs agit en tant qu'hôte, servant de serveur pour les autres clients. Cette solution est souvent privilégiée pour les jeux coopératifs ou les parties à effectif réduit, car elle permet de réduire les coûts et la complexité de la gestion des serveurs.

Nous avons opté pour l'architecture peer-to-peer, adaptée à notre jeu coopératif. Ainsi, chaque joueur se connecte au PC de l'hôte qui centralise la gestion des données de la partie. Cette décision a permis d'éviter des coûts supplémentaires liés à l'hébergement d'un serveur dédié, tout en offrant une expérience de jeu fluide pour un nombre limité de joueurs.

## 6.2 Choix de l'outil réseau

Une fois l'architecture choisie, nous avons évalué plusieurs frameworks permettant de gérer le multijoueur sous Unity. Les trois candidats principaux étaient :

— **Mirror,**

— **Photon,**

— **Unity NetCode for GameObjects.**

Étant donné que `Unity NetCode for GameObjects` est déjà intégré dans Unity et offre une prise en main relativement simple, nous avons décidé de l'utiliser. Ce framework nous a permis de bénéficier d'une intégration native, de fonctionnalités telles que les `NetworkVariables` et les `Server RPC`, et d'une communauté active pour le support technique.

### 6.3 Création et gestion des sessions multijoueur

Nous avons développé un menu dédié qui permet au joueur de créer ou de rejoindre un salon multijoueur. Lorsqu'un joueur crée une partie, son PC se comporte comme l'hôte, initialisant ainsi un serveur local. Dès qu'un autre joueur se connecte, un nouveau modèle préfabriqué (prefab) représentant le joueur est instancié sur la scène, assurant ainsi une représentation synchronisée de tous les participants. Ce mécanisme a nécessité quelques adaptations, notamment pour :

— La gestion des caméras, puisque le jeu initial ne comportait qu'une seule caméra. Lorsqu'un second joueur se connectait, il voyait la même vue que l'hôte. Pour résoudre ce problème, nous avons implémenté une solution où chaque joueur se voit attribuer une caméra dédiée qui suit ses propres déplacements.

— La gestion de la génération de la carte. Initialement, la carte se générait autour du joueur local. Cependant, en mode multijoueur, chaque client devait recevoir les mêmes informations concernant les "chunks" (morceaux de carte) pour garantir la cohérence de l'environnement de jeu. Nous avons donc recours aux `Server RPC` pour envoyer des requêtes depuis les clients vers l'hôte, qui se charge de générer et de renvoyer les données appropriées. Ainsi, la carte se génère de manière dynamique et personnalisée sur l'écran de chaque joueur, tout en conservant une structure globale cohérente.

### 6.4 Problèmes rencontrés et solutions apportées

#### 6.4.1 Synchronisation des spawn Points et des caméras

Lors des premiers tests, nous avons constaté que lorsqu'un nouveau joueur rejoignait la partie, il utilisait la même caméra que l'hôte, entraînant ainsi un suivi inapproprié et une vision partagée qui nuisaient à l'immersion et à la jouabilité. Ce problème posait également des difficultés en termes de positionnement, le système de spawn n'étant pas suffisamment robuste pour garantir un point d'apparition sûr pour chaque joueur.

Pour résoudre ce problème, nous avons modifié le système d'instanciation afin que chaque joueur se voie attribuer une caméra dédiée, créée lors de la connexion via le prefab du joueur. Par ailleurs, la fonction `FindSafeSpawn` a été améliorée pour rechercher et centrer chaque joueur sur une tuile sûre, garantissant ainsi un spawn cohérent et sécurisé pour tous.

#### 6.4.2 Génération de la carte en Mode multijoueur

Initialement, la génération de la carte était centrée sur la position du joueur local, ce qui fonctionnait bien en mode solo mais posait problème en multijoueur. Chaque client recevait une partie de la carte basée sur sa propre position, entraînant des incohérences dans le rendu global et des difficultés pour synchroniser l'environnement de jeu entre les différents joueurs.

Pour pallier cette difficulté, nous avons implémenté des `Server RPC` qui permettent aux clients d'envoyer des requêtes à l'hôte pour générer les "chunks" de carte autour de leur position. Ainsi, l'hôte centralise la génération et la distribution des segments de carte, assurant que chaque joueur visualise une carte cohérente et synchronisée, tout en optimisant les performances en ne générant que les portions nécessaires.

#### 6.4.3 Incohérence de la "seed" de la carte

Nous avons remarqué que, dans la version initiale, la seed de génération de la carte était générée aléatoirement à chaque lancement du jeu, ce qui pouvait conduire à ce que deux joueurs dans la même partie se retrouvent sur des cartes différentes. Cette divergence entraînait des désynchronisations majeures, compromettant l'intégrité de l'expérience multijoueur.

La solution apportée a consisté à centraliser la génération de la seed sur l'hôte en utilisant les `NetworkVariables` de `Unity NetCode for GameObjects`. En diffusant la même seed à tous les clients, nous garantissons que l'environnement de jeu reste identique pour tous les participants, assurant ainsi une synchronisation parfaite de la carte.

#### 6.4.4 Problèmes de latence et fluidité des mouvements

Des tests en conditions réelles ont révélé que les mouvements des joueurs étaient parfois saccadés ou décalés, en particulier dans des situations de latence réseau élevée. Ces saccades affectaient non seulement l'expérience de jeu, mais pouvaient également induire des désynchronisations temporaires entre les actions des joueurs et leur représentation visuelle.

Pour remédier à ce problème, nous avons optimisé le traitement des entrées en normalisant les vecteurs de déplacement et en appliquant les mises à jour dans `FixedUpdate` pour assurer une régularité accrue. Par ailleurs, nous avons implémenté une interpolation des positions des joueurs distants, permettant d'adoucir les transitions et d'offrir une expérience visuelle fluide même en cas de fluctuations du réseau.

# 7 Inventaire

La mise en place du système d'inventaire dans *Uzin* était cruciale pour permettre aux joueurs de gérer, échanger et utiliser leurs objets dans un environnement multijoueur dynamique. Après une analyse approfondie des solutions disponibles, nous avons, ici aussi, opté pour `Unity NetCode for GameObjects`. Ce choix s'est imposé grâce à son intégration native dans Unity, facilitant la synchronisation des objets et des données entre les clients et le serveur, sans nécessiter de refonte majeure de l'architecture existante.

## 7.1 Choix de l'outil

L'utilisation de `Unity NetCode for GameObjects` a permis de transformer rapidement chaque objet d'inventaire en entité réseau à l'aide de composants tels que `NetworkObject` et `NetworkBehaviour`. Cette intégration native simplifie la mise en place d'un système d'inventaire multijoueur en assurant la synchronisation en temps réel des actions (ajout, retrait, utilisation) sur tous les clients. De plus, la robustesse et la flexibilité de NetCode offrent des performances optimisées, minimisant la latence et évitant la surcharge du réseau grâce à des mécanismes de réplication efficaces.

## 7.2 Processus de gestion de l'inventaire

Le système d'inventaire repose sur plusieurs étapes essentielles :

— **Conversion en entités réseau :** Chaque objet d'inventaire est transformé en une entité réseau via `NetworkObject`, ce qui permet de gérer et de synchroniser son état sur tous les clients. Cette étape garantit que toute modification apportée à un objet est immédiatement répercutée dans l'ensemble de la partie.

— **Synchronisation des actions :** Les interactions des joueurs, telles que l'ajout, le retrait ou l'utilisation d'un objet, déclenchent des `Remote Procedure Calls` (RPC). Ces RPC mettent à jour l'état de l'inventaire sur le serveur, lequel réplique ensuite les changements à tous les clients, assurant ainsi une cohérence parfaite.

— **Gestion des propriétés et des effets :** Outre la synchronisation des états dynamiques, le système gère également des données plus statiques, telles que les attributs intrinsèques des objets (par exemple, leur rareté, leurs statistiques ou leurs effets spéciaux). Cette approche permet d'intégrer facilement des fonctionnalités avancées, comme des inventaires partagés ou des objets interactifs avec des effets uniques.

## 7.3 Problèmes rencontrés et solutions apportées

### 7.3.1 Synchronisation des objets d'inventaire

Lors des premiers tests, nous avons constaté que les actions de modification de l'inventaire (ajout ou retrait d'objets) n'étaient pas toujours correctement synchronisées entre les clients. Parfois, un objet ajouté par un joueur n'apparaissait pas immédiatement sur les autres clients, créant ainsi des incohérences dans l'état global de l'inventaire.

Pour résoudre ce problème, nous avons utilisé les `NetworkVariable` de `Unity NetCode for GameObjects` afin de centraliser et répliquer l'état de l'inventaire sur tous les clients. Chaque objet devient une entité réseau dont les modifications sont validées sur le serveur avant d'être diffusées, garantissant ainsi une synchronisation en temps réel.

### 7.3.2 Latence et réactivité des mises à jour

En situation de latence élevée, les mises à jour de l'inventaire, telles que l'ajout ou le retrait d'un objet, pouvaient être perçues avec un léger décalage, affectant la réactivité du système.

Pour pallier ce problème, nous avons optimisé le processus de réplication en réduisant la quantité de données transmises lors des mises à jour et en implémentant des mécanismes d'interpolation côté client. Cette approche permet d'adoucir les transitions visuelles et d'offrir une expérience fluide, même en présence de fluctuations du réseau.

### 7.3.3 Gestion des interactions et des échanges

Les échanges d'objets entre joueurs représentent une fonctionnalité complexe, pouvant entraîner des conflits lorsque plusieurs transactions se produisent simultanément. Dans certaines situations, des objets étaient dupliqués ou supprimés par inadvertance, compromettant ainsi l'intégrité de l'inventaire.

Pour résoudre ces problèmes, nous avons mis en place un mécanisme transactionnel basé sur des RPC atomiques. Chaque échange d'objet est validé par le serveur, qui s'assure que l'opération se déroule de manière cohérente pour tous les participants. Ce système de verrouillage empêche les conflits et garantit que l'état de l'inventaire reste synchronisé et fiable.

# 8 Conclusion

En conclusion, ce deuxième rapport de soutenance démontre que le projet *Uzin* a considérablement progressé dans plusieurs domaines clés, notamment la mise en place d'un système multijoueur robuste, l'optimisation de la génération de la carte et la mise en œuvre d'un système d'inventaire synchronisé. Les choix technologiques, tels que l'utilisation d'`Unity NetCode for GameObjects`, ont permis de répondre efficacement aux défis techniques, en assurant une synchronisation en temps réel et une répartition optimisée des ressources réseau. Ces avancées constituent une base solide pour offrir aux joueurs une expérience immersive et fluide.

Toutefois, il est important de noter que certaines fonctionnalités, notamment l'intégration de l'intelligence artificielle, ont été repoussées afin de concentrer les efforts sur le développement des mécanismes de jeu et du multijoueur. Ce choix stratégique, bien que responsable de quelques retards dans l'enrichissement de l'IA, nous a permis d'assurer la stabilité et la cohérence globale du projet. De plus, l'aspect design et graphismes, bien que moins évolué depuis la première soutenance, continue d'être développé en interne afin de garantir une identité artistique forte et unique pour Uzin, comme en témoigne la conception d'un menu principal évolutif.

En somme, malgré quelques retards et défis techniques, l'équipe reste pleinement engagée et confiante quant à la poursuite du développement du projet. Les solutions mises en œuvre pour la synchronisation, l'optimisation des performances et la gestion des interactions multijoueur ouvrent la voie à de futures améliorations. Nous envisageons d'enrichir prochainement le gameplay avec l'intégration d'une intelligence artificielle plus avancée et l'ajout de fonctionnalités supplémentaires, tout en consolidant les bases techniques déjà établies.

Ce rapport témoigne non seulement des progrès réalisés, mais aussi de notre capacité à identifier et résoudre les problèmes au fur et à mesure qu'ils se présentent. Nous sommes déterminés à finaliser Uzin en respectant nos objectifs et en assurant une expérience de jeu cohérente, immersive et innovante pour l'ensemble des joueurs.