



SATIS-KINGDOM

404 Social Life Not Found

Mars 2025

Sommaire

Introduction	3
Récapitulatif des tâches	4
1 Game design	6
1.1 Création des assets	6
1.2 Map design	9
1.2.1 Création d'une carte variée	9
1.2.2 Les différentes zones	9
1.2.3 Organisation des zones et définition des limites de la carte	10
1.2.4 Système de déblocage progressif	11
1.2.5 Détails des environnements	11
1.3 Tutoriel	12
2 Références	13
2.1 ScriptableObject	13
2.2 Prefab	13
2.3 Singleton Pattern	13
3 Système de Dialogue	14
3.1 UI	14
3.2 Editeur	16
3.3 Code	18
4 Gestion des Objets / Inventaire	20
4.1 Les Objets	21
4.1.1 Objet de base	22
4.1.2 Objet consommable	22
4.1.3 Objet d'équipements	23
4.2 Inventaire	25
4.2.1 Les fonctionnalités	25
4.3 UI	28
4.4 Implémentation	29
4.5 Objets ramassable	32
5 Multijoueur	34
5.1 UX	34
6 Ennemis	36
6.1 Implémentation	38

7	Site Web	39
7.1	Introduction	39
7.2	Approche de la création	39
7.3	Conception du site	40
7.4	Présentation du site	41
7.5	Conclusion :	43
8	FX	44
	Etat d'avancement	45
	Conclusion	47

Introduction

Dans le cadre de notre projet informatique du second semestre de notre année, notre équipe 404 Social Life Not found réalisons un jeu vidéo nommé Satis-Kingdom.

Satis-Kingdom est un jeu de type 2d RPG, s'inscrivant dans un style pixel art et avec une vue du dessus que qualifie de “Top-Down”. Le ou les Joueurs en fonction du mode de jeu solo ou multi-joueur incarne un personnage qui évolue dans un univers médiéval-fantastique. les joueurs seront amenés à suivre different quête qui les mèneront à explorer les different aspect du monde que nous avons creer. En plus de l'aspect d'aventure nous avons pensé un système de combat avec différents ennemis. Et enfin nous proposerons aux joueurs des mécanismes de récolte de ressources qui permettront de faire évoluer à la fois le personnage et le monde qui l'entour.

Ce rapport a pour objectif de vous présenter l'avancement global de notre projet. Afin de faire un état des lieux des différentes fonctionnalités du jeu qui sont implémentées, chaque partie sera aussi expliquée en détail, avec les difficultés rencontrées ou encore la manière dont l'implémentation a été réalisée.

Récapitulatif des tâches

Tâches	Tristan	Quentin R	Quentin S	Raphaël	Alexandre
Game Design					
Création des assets				R	S
Musiques / sons			S	S	R
Rédaction de l'histoire	R		S		
Map Design	R			S	
Programmation					
Système d'inventaire	S	R		S	
Système de sauvegarde		R		S	S
Déplacement / action Personnage		R		S	
Gestion des ennemis		S	R	S	
Système de dialogue		R		S	S
Tutoriel	S		S	R	
Multijoueur	R	S			
IA			S		R
Site Web					
Page d'accueil	S	S	R		
Téléchargement			R		S
Manuel Utilisateur			R	S	
Autres					
Mise en page Cdc		R	S		

Table 1: Répartition des tâches (**R** : Responsable, **S** : Suppléant)

Tâches	Soutenance 1	Soutenance 2	Soutenance 3
Game Design			
Création des assets	80 %	100 %	100 %
Musiques / sons	40 %	100 %	100 %
Rédaction de l'histoire	90 %	100 %	100 %
Map Design	20 %	100 %	100 %
Programmation			
Système d'inventaire	0 %	100 %	100 %
Système de sauvegarde	0 %	10 %	100 %
Déplacement / action Personnage	20 %	100 %	100 %
Gestion des ennemis	0 %	40 %	100 %
Système de dialogue	0 %	100 %	100 %
Tutoriel	0 %	20 %	100 %
Multijoueur	15 %	30 %	100 %
IA	0 %	40 %	100 %
Site Web			
Page d'accueil	15 %	100 %	100 %
Téléchargement	0 %	30 %	100 %
Manuel Utilisateur	0 %	30 %	100 %
Autres			
Mise en page Cdc	100 %	100 %	100 %

Table 2: Avancement et planification

Au cours de cette seconde phase de travail, nous avons concentré nos efforts sur la concrétisation de nos idées pour le jeu. Les principales fonctionnalités ont été développées, la carte a pris forme, et les animations ainsi que le site web ont bien progressé.

Pour les fonctionnalités implémentées, le plus important pour nous a été la modularité. Nous avons mis en place des systèmes solides qui permettront l'ajout facile de nouvelles fonctionnalités.

Grâce à ces avancées, notre équipe est en bonne voie pour finaliser un jeu captivant, avec des mécaniques engageantes et une histoire immersive.

Game design

1.1 Cration des assets

Depuis le debut du projet, des avances significatives ont et  realis es dans la conception et l'animation des ´el ments graphiques du jeu. A ce jour, l'ensemble des graphismes est achev  a 95 %, avec un travail approfondi sur l'animation des personnages, des cr atures ainsi que la cr ation des items et des ´el ments environnementaux. L'objectif principal a et  d'assurer une coh rence visuelle forte et une fluidit  d'animation optimale afin d'offrir une exp rience immersive et engageante pour les joueurs.

Les animations du personnage principal ont et  finalis es, garantissant des transitions naturelles entre les diff rentes actions, notamment les d placements et les attaques. Un soin particulier a et  apport  a l'animation des s quences de combat, assurant des mouvements r actifs et dynamiques pour renforcer l'impact visuel et le ressenti en jeu. Chaque cycle d'animation a et  optimis  afin de pr serves une harmonie avec le rythme du gameplay et la direction artistique d finie.

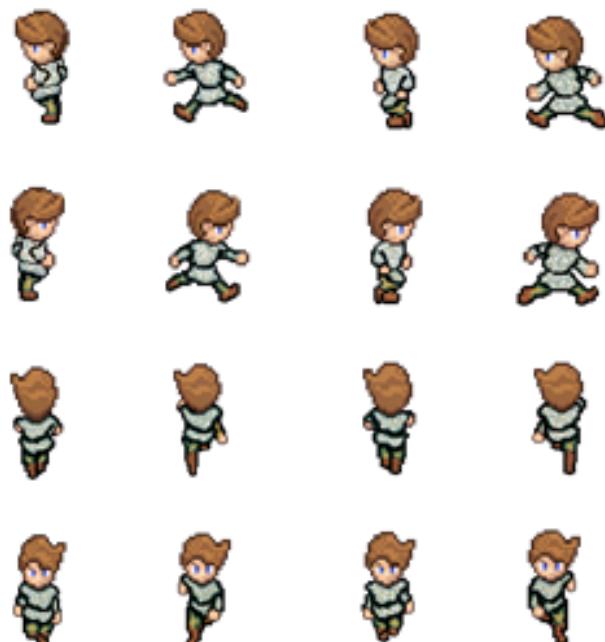


Figure 1.1: Animations du personnage principal

En parallèle, un travail approfondi a été effectué sur l'animation des créatures hostiles, chacune bénéficiant d'un traitement spécifique pour correspondre à son identité visuelle et comportementale. Le squelette, par exemple, a été animé avec des mouvements rigides et saccadés pour refléter sa nature, tandis que le golem adopte des déplacements lourds et puissants, accentuant sa force et son inertie. L'araignée, quant à elle, présente une animation fluide et rapide, mettant en avant son agilité et son imprévisibilité. L'ensemble de ces animations contribue à enrichir l'expérience de jeu en rendant chaque adversaire unique et en renforçant la diversité des combats.



Figure 1.2: Animation Forgeron

Les créatures passives ont également bénéficié d'un travail d'animation rigoureux, avec la finalisation des cycles de déplacement pour différentes espèces. Le cochon, la vache, la poule et le loup disposent désormais de mouvements adaptés à leur comportement respectif, garantissant une cohérence avec leur rôle dans l'univers du jeu. La diversité de ces animations participe à l'immersion globale en rendant les environnements plus vivants et dynamiques.



Figure 1.3: Animation vache

En complément du travail sur les personnages et les créatures, l'ensemble des items du jeu a été conçu et intégré en respectant scrupuleusement la direction artistique du projet. Ces objets interactifs, essentiels au gameplay, ont été réalisés avec un souci du détail afin d'assurer une harmonie visuelle avec les autres éléments du jeu et de renforcer l'identité esthétique du projet.



Figure 1.4: Items

Au fil du développement, de nouveaux besoins ont émergé, entraînant l'ajout de contenus graphiques supplémentaires. Des tiles environnementaux ont été créés afin d'enrichir les décors et d'apporter davantage de diversité aux paysages du jeu. Par ailleurs, une animation supplémentaire a été demandée et est en cours de réalisation. Ces ajouts ont eu un impact sur la charge de travail initialement prévue, expliquant pourquoi l'avancement actuel est de 90 % au lieu des 100 % initialement fixés.



Figure 1.5: Mage

Grâce aux efforts déployés, l'univers graphique du jeu est désormais bien établi, offrant des animations fluides et une forte cohérence artistique. Les derniers éléments en cours de production seront finalisés prochainement afin de compléter l'ensemble des graphismes et d'optimiser l'expérience visuelle du jeu. Une fois ces derniers ajustements réalisés, l'ensemble des éléments graphiques sera prêt pour les phases finales d'intégration et de test, garantissant ainsi un rendu visuel abouti et conforme aux attentes du projet.

1.2 Map design

1.2.1 Crédation d'une carte variée

La conception d'une carte variée est essentielle pour offrir une expérience immersive et captivante aux joueurs. Pour cela, il est crucial d'intégrer différentes zones possédantes chacune leur propre identité et fonction.

1.2.2 Les différentes zones

La carte doit être composée de plusieurs zones distinctes, apportant diversité et dynamisme à l'exploration:

- **Zones de combat de boss :**

Des environnements dédiés aux affrontements épiques contre des ennemis redoutables, marquant des évènements importants dans l'histoire et apportant challenge et intensité au game-play.



Figure 1.6: Zones de boss

- **Village abandonné :**

Un village mystérieux, vestige du passé, offrant différentes possibilités narratives et des opportunités de découverte.



Figure 1.7: Village abandonné

- **Zone de collecte de ressources :**

Un espace essentiel permettant aux joueurs de récolter des matériaux pour l'artisanat, l'amélioration de leur équipement et la réparation du village.

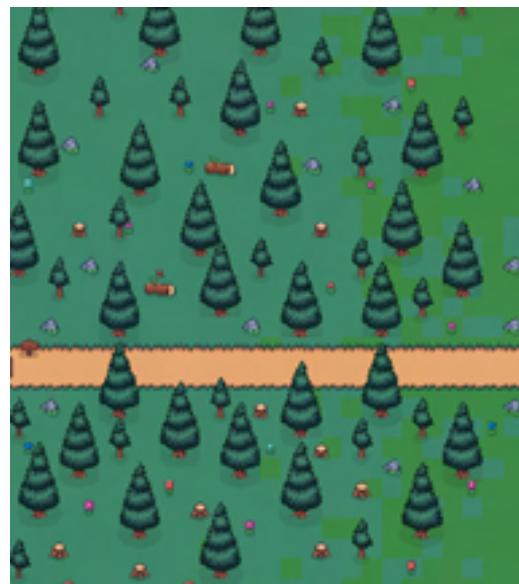


Figure 1.8: Zone de collecte de ressources

- **Zones d'exploration :**

Des paysages uniques remplis de secrets et de quêtes, incitant l'exploration et l'immersion complète dans l'univers du jeu.



Figure 1.9: Zones d'exploration

1.2.3 Organisation des zones et définition des limites de la carte

L'organisation des zones de collision est une étape nécessaire et importante dans la construction de la carte. Définir avec précision les limites des différentes zones permet d'assurer une transition fluide entre les environnements.

Les barrières naturelles et artificielles, telles que les montagnes, les rivières ou les ruines, serviront à structurer la carte et à guider les déplacements des joueurs. Elle permet de délimiter une zone précise et de s'assurer que le joueur ne se perde pas.

1.2.4 Système de déblocage progressif

La progression des joueurs doit être soigneusement gérée à travers un système de zones déblocables. Au fur et à mesure de l'aventure, certaines parties de la carte deviendront accessibles, favorisant un sentiment de progression et d'accomplissement. Le jeu offre tout de même une grande liberté au joueur le laissant explorer le maximum de zones.

1.2.5 Détails des environnements

Les différentes zones, qu'elles soient essentielles à l'histoire ou de simples lieux de passage, ont été conçues avec un soin particulier afin de rendre l'expérience de jeu la plus immersive et satisfaisante possible. Chaque environnement est décoré de manière à renforcer l'atmosphère qui lui est propre.

En combinant ces éléments, la carte offrira une expérience de jeu équilibrée, favorisant à la fois l'exploration, le défi et l'immersion dans un univers riche et varié.



Figure 1.10: Zone de collecte de ressources

1.3 Tutoriel

Dans le cadre du développement du jeu, un tutoriel a été prévu afin d'accompagner les joueurs dans la prise en main des mécaniques de jeu et de les immerger progressivement dans l'univers. L'objectif initial était de consacrer 20% du travail à cette phase d'apprentissage, garantissant ainsi une introduction efficace et intuitive pour les nouveaux joueurs.

Après réflexion, il a été décidé que le tutoriel serait conçu sous la forme d'une petite carte dédiée, où le joueur fera son apparition dès le lancement de la partie. Cette zone spécifique servira à enseigner progressivement les différentes touches et commandes du jeu, en proposant une approche interactive et fluide. L'environnement de cette carte sera conçu de manière à introduire visuellement l'univers du jeu, tout en guidant le joueur dans ses premières interactions.

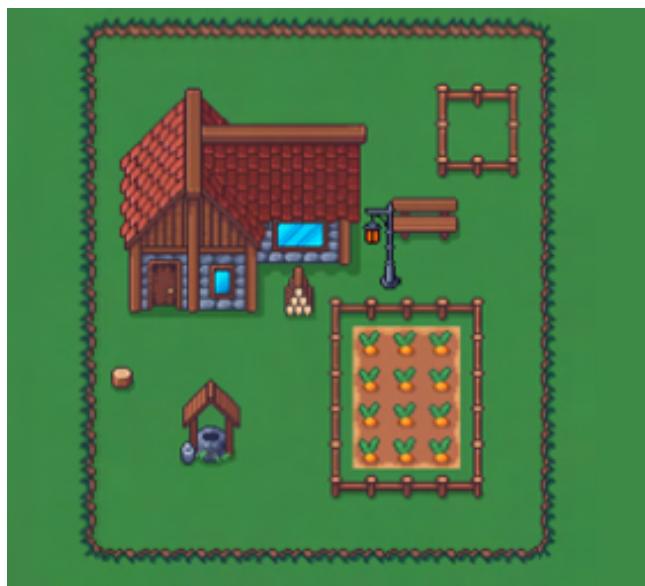


Figure 1.11: Tutoriel

Le tutoriel intégrera des indications claires et progressives, permettant au joueur d'apprendre les déplacements, les actions principales et les mécaniques essentielles du gameplay. Des éléments visuels et interactifs seront intégrés pour encourager l'expérimentation et assurer une assimilation naturelle des commandes. L'objectif est de proposer une introduction qui ne soit ni trop intrusive ni trop longue, afin de maintenir un bon équilibre entre apprentissage et immersion dans le jeu.

Grâce à cette approche, le joueur pourra rapidement comprendre les bases du jeu tout en découvrant son univers, ce qui facilitera sa progression et son engagement. Ce tutoriel servira également de première introduction à l'ambiance et aux mécaniques fondamentales, préparant ainsi efficacement le joueur à la suite de l'aventure.

Références

Dans les sous-parties suivantes, nous allons définir les différents principes et outils que nous utilisons pour développer notre jeu tout en assurant une bonne maintenabilité et modularité.

2.1 ScriptableObject

Les **ScriptableObjects** dans Unity sont des objets de données qui permettent de stocker des informations de manière centralisée, indépendante de toute instance d'un *GameObject*. Habituellement, les classes traditionnelles doivent être obligatoirement attachées à un *GameObject* dans la scène. Les **ScriptableObjects** sont des assets créés et gérés dans l'éditeur. Ils sont particulièrement utiles pour partager des données entre différentes scènes, pour améliorer l'organisation, la gestion des ressources et la modularité d'un système. Grâce à leur sérialisation, ils sont facilement réutilisables et peuvent être modifiés directement depuis l'éditeur Unity. Dans notre projet, nous les utilisons par exemple dans la gestion des objets, les dialogues et, dans le futur, pour les quêtes et peut-être de prochaines fonctionnalités.

2.2 Prefab

Les **Prefabs** dans Unity sont des modèles réutilisables de *GameObjects* qui peuvent inclure des composants, des scripts et des sous-objets. Ils permettent de créer des objets complexes une seule fois, puis de les réutiliser dans différentes scènes. Les Prefabs peuvent aussi être instanciés à la volée pendant l'exécution du jeu via des scripts, ce qui permet de créer dynamiquement des objets en réponse aux actions du joueur ou aux événements du jeu. Lorsqu'un Prefab est modifié, toutes ses instances dans le projet sont automatiquement mises à jour. Cela facilite la gestion et la maintenance des objets de jeu. En plus de leur utilisation dans l'éditeur, cela est particulièrement utile pour des éléments qui se répètent, comme les ennemis, les armes ou les décorations, tout en assurant une cohérence et une efficacité de travail.

2.3 Singleton Pattern

Le **Singleton Pattern** est un modèle de conception qui garantit qu'une classe ne peut avoir qu'une seule instance tout au long de l'exécution du programme et fournit un point d'accès global à cette instance. L'implémentation typique en C# consiste à créer une instance statique de la classe et à s'assurer qu'elle est unique, en l'assignant dans la méthode *Awake*. De plus, si on essaie de créer une nouvelle instance de la classe, celle-ci est détruite instantanément. Dans notre projet, nous utilisons ce modèle pour des éléments comme le *GameManager*, qui s'occupe des statistiques globales et de la gestion du jeu, ou le *SceneManager*, qui permet le changement de scène. Par défaut, le *NetworkManager* de Mirror suit ce même modèle.

Système de Dialogue

Notre jeu étant un RPG, nous avons la nécessité de créer un univers riche, cela se traduit notamment par de nombreux PNJs ou objets avec lesquels les joueurs peuvent interagir. Pour répondre à cette problématique, nous avons créé un système de dialogue modulaire et évolutif. Le but était de créer les outils nécessaires afin de créer des dialogues partout dans le monde et de les rendre fonctionnels de manière universelle. Cette approche permet de simplifier le développement en créant des scripts qui s'adaptent automatiquement et de ne pas réutiliser du code partout. De plus, cette approche modulaire nous permet de facilement modifier le comportement ou l'affichage dans le futur si nécessaire.

Nous allons d'abord vous montrer ce que permet notre système de dialogue d'un point de vue du joueur, puis de l'éditeur pour l'ensemble de notre équipe en tant que créateurs du jeu et enfin d'un point de vue de la programmation pour l'aspect technique de la mise en place de ce système.

3.1 UI

Nous pouvons voir ci-dessous un exemple de déclenchement d'une fenêtre de dialogue. Ici, nous avons simplement un panneau dont le dialogue sera là pour expliquer au joueur à quoi sert le bâtiment. Nous avons une zone de détection autour de ce panneau, puis, au moment où le joueur appuie sur la touche d'interaction, le dialogue correspondant se déclenche. Cette zone de détection, qui vérifie si un joueur est dans la zone et détecte si le joueur appuie sur la touche, est un Prefab. Comme expliqué plus tôt, cela nous permet de rendre le système modulaire : la partie détection est détachée de la partie visuelle. C'est ainsi que nous pouvons déclencher des dialogues avec des PNJs de la même manière qu'avec ce panneau.



Figure 3.1: Panneau dialogue

Une fois le dialogue activé, comme on peut le voir sur les images, le dialogue commence : le texte est affiché avec un effet machine à écrire où les caractères sont ajoutés un par un. Une fois le texte complètement affiché, le joueur peut cliquer sur la touche correspondante pour passer à la ligne de dialogue suivante.



Figure 3.2: Première ligne de dialogue



Figure 3.3: Deuxième ligne de dialogue

(Ici, les dialogues sont à titre d'exemple et n'ont pas de rapport avec le contexte actuel du panneau.)

3.2 Editeur

Nous allons à présent voir comment notre outil nous permet de créer des dialogues depuis l'éditeur, ce qui nous permet donc de façonner l'univers de notre jeu.

Pour assurer cette modularité, comme introduit précédemment, nous utilisons les **ScriptableObject**. Cela nous permet de créer un nouvel objet de type Dialogue directement depuis l'éditeur Unity. Cet objet est défini par un script que vous pourrez voir plus loin dans la partie Code.

La création de cet objet est la première étape pour créer un nouveau dialogue.

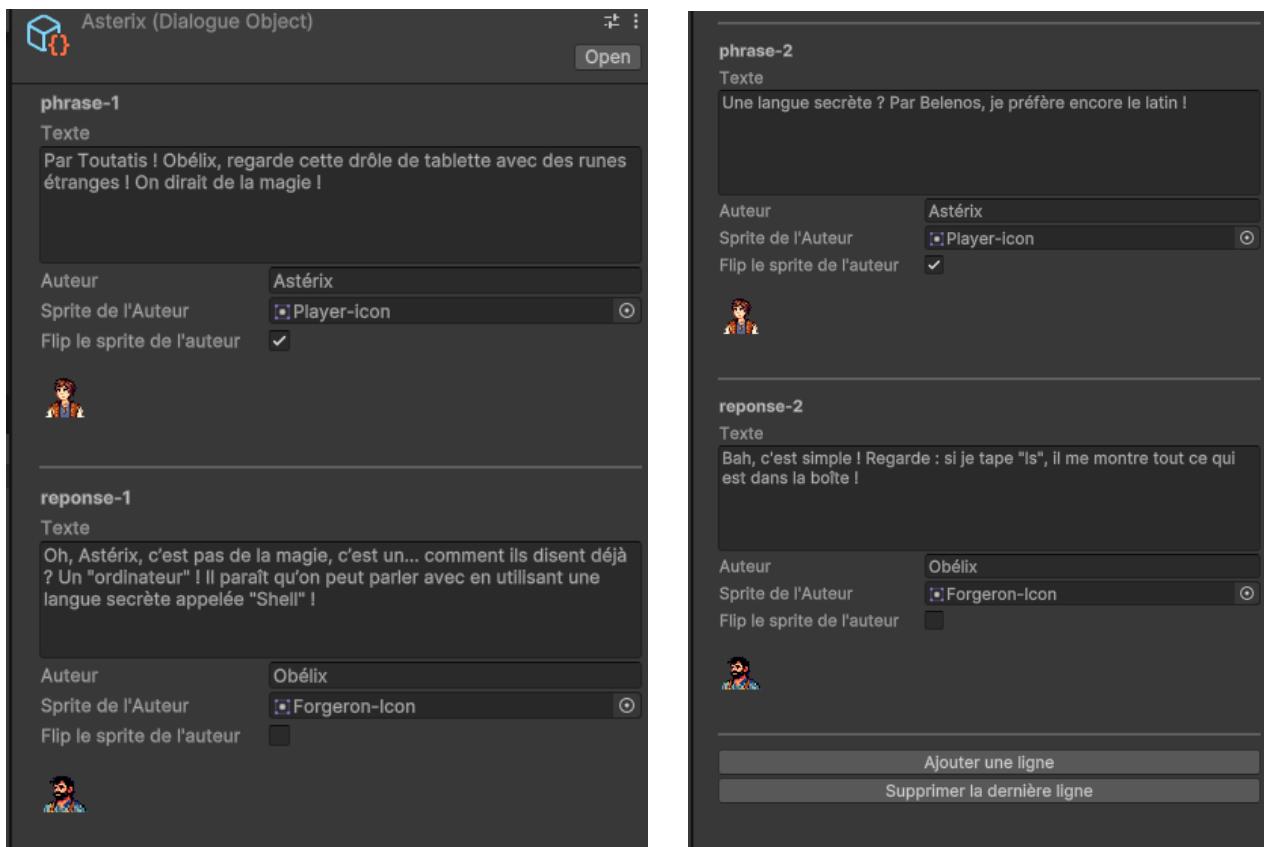


Figure 3.4: Exemple de Dialogue SO

Voici l'interface de l'éditeur de notre objet de type Dialogue. Comme on peut le voir au bas de l'image, nous avons deux boutons pour ajouter ou supprimer une ligne au dialogue. Chaque ligne est composée de quatre paramètres :

- **Texte** : le texte à afficher pour la ligne actuelle.
- **Auteur** : le nom à afficher au-dessus de la boîte de dialogue.
- **Sprite de l'auteur** : le sprite à afficher avec le nom au-dessus de la boîte de dialogue.
- **Flip** : permet de choisir si l'on inverse le sprite de l'auteur. Cela permet de faire en sorte que la personne qui parle en premier regarde vers la droite et celle qui répond, vers la gauche

On y retrouve l'ensemble du contenu qui est affiché dans l'exemple d'introduction du système de dialogue.

Dans un second temps, comme expliqué précédemment, nous utilisons un Prefab pour l'activation du dialogue. Le Prefab est l'objet que vous pouvez voir sur l'image suivante.

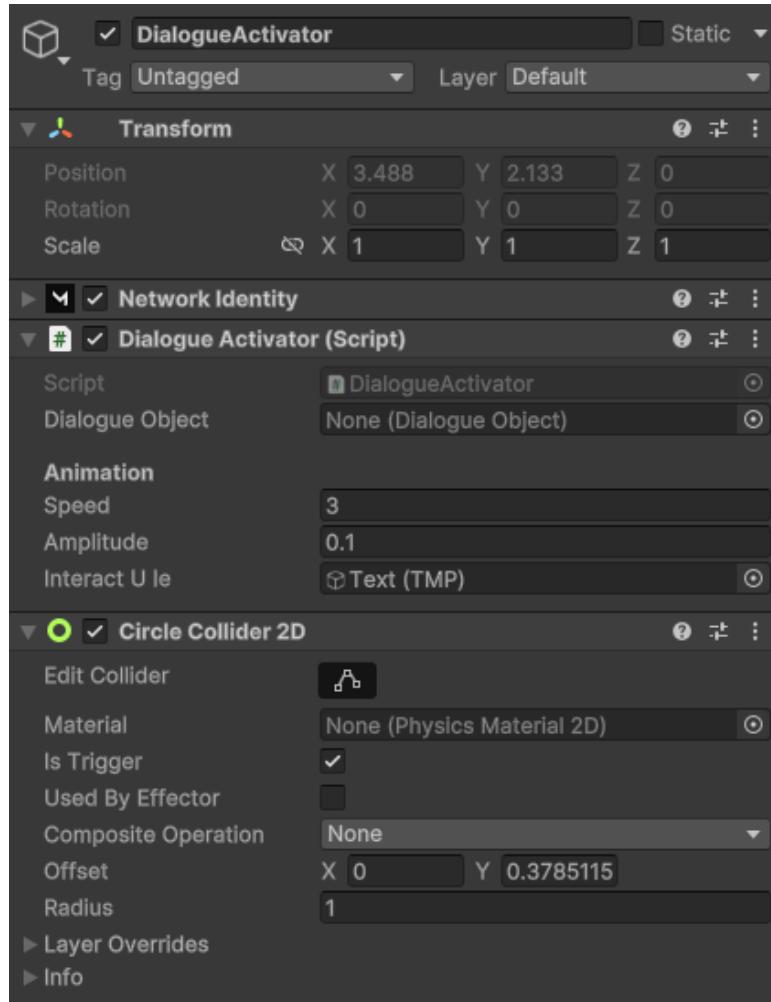


Figure 3.5: Prefab du dialogue Activator

On peut y voir notre script "Dialogue Activator", dans lequel on peut référencer le **ScriptableObject** créé précédemment. Les autres paramètres sont dédiés à l'animation permettant d'indiquer au joueur qu'il doit appuyer sur la touche d'interaction.

Le deuxième composant nécessaire est un Collider 2D qui va détecter les joueurs entrant dans la zone. Notre script fonctionne avec plusieurs types de Colliders. Ici, nous avons choisi un Collider en forme de cercle, mais un Collider en forme de rectangle fonctionnerait aussi.

D'un point de vue éditeur, ce sont les seules choses à faire. Une fois le Prefab posé au bon emplacement, notre script s'occupe dynamiquement du reste et affichera les dialogues en fonction des paramètres.

Pour que le dialogue s'affiche correctement, nous avons dans notre Prefab du joueur toute l'UI définie avec des textes placeholder, qui sont remplacés dynamiquement en fonction du dialogue. Sur cet objet, nous avons un autre script qui s'occupe de modifier l'UI. Ce script va donc être utilisé par le Dialogue Activator pour démarrer un dialogue.

3.3 Code

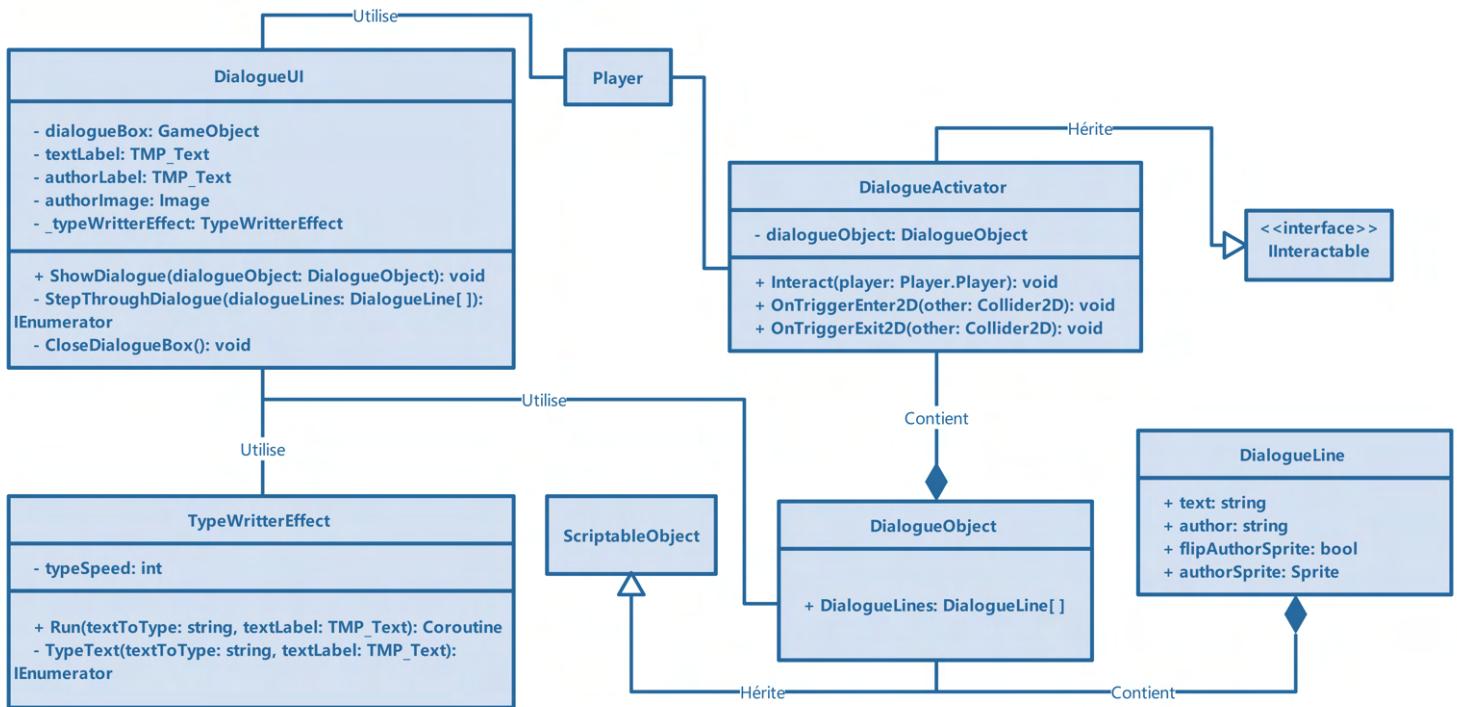


Figure 3.6: UML système de dialogue

Nous pouvons ici voir un diagramme UML de l'architecture de notre code pour la partie des dialogues.

- **DialogueLine** est une classe sérialisée avec les différents attributs pour une ligne de dialogue.
- **DialogueObjet**, qui est un **ScriptableObject**, contient une liste d'objets **DialogueLine**. Puisque la classe est sérialisée, depuis l'éditeur, sur les objets de type **DialogueObjet**, nous pouvons changer les valeurs des différents attributs.

Pour afficher cela de manière plus propre et ajouter les boutons pour ajouter et supprimer une ligne de dialogue, nous utilisons un script éditeur qui n'est pas présent dans ce schéma. Les scripts éditeurs sont exécutés uniquement dans l'éditeur et permettent de modifier la manière dont Unity affiche les éléments.

Ensuite, **DialogueActivator** s'occupe de détecter les entrées dans le Collider grâce aux deux méthodes que nous pouvons voir. Nous disposons d'une référence à un objet de type **DialogueObjet**, qui permet de savoir quel dialogue lancer. Enfin, cette classe hérite de l'interface **IInteractable**, c'est pour cela qu'elle implémente la méthode Interact. Nous utilisons des interfaces pour généraliser notre

code et ainsi faciliter l'ajout et la modification de fonctionnalités. Cette méthode Interact utilise donc la classe **Player**, qui permet d'accéder à la classe **DialogueUI** en utilisant la fonction *ShowDialogue*.

Pour terminer, une fois que la classe **DialogueUI** reçoit l'ordre d'afficher un dialogue, elle va utiliser la classe statique **TypeWriterEffect** ainsi que ses références, c'est-à-dire les Text et Sprite de l'UI, pour changer le texte et ajouter progressivement les caractères.

Enfin, **DialogueUI** s'occupe aussi, lors du dialogue, de gérer les entrées utilisateur pour passer à la ligne de dialogue suivante.

Gestion des Objets / Inventaire

Les parties qui vont suivre, c'est-à-dire, la gestion des objets, l'équipement et l'inventaire, sont toutes interdépendantes. Pour simplifier les choses, nous les aborderons une par une.

Pour un RPG, une chose très importante est la variété d'objets, que ce soit des objets pour accomplir des quêtes, des ressources, des objets consommables qui peuvent avoir des effets comme regagner de la vie, ou encore des équipements avec certaines statistiques.

L'ensemble des fonctionnalités plus avancées, comme le système de quêtes et la récolte de ressources, dépend fortement de l'implémentation des objets dans le jeu. C'est pourquoi nous avons passé beaucoup de temps sur cette partie et perfectionné beaucoup de détails afin de nous simplifier la tâche pour les futures fonctionnalités. Encore une fois, le but a été de réaliser un système modulaire et flexible qui, une fois implémenté, permet de créer facilement une grande quantité d'objets différents.

En plus de cela, comme nous le verrons dans la dernière de ces parties, pour mettre en pratique ces objets, il a été essentiel de créer un système d'inventaire afin que le joueur puisse gérer l'ensemble de ses objets, équipements, consommables, etc.

4.1 Les Objets

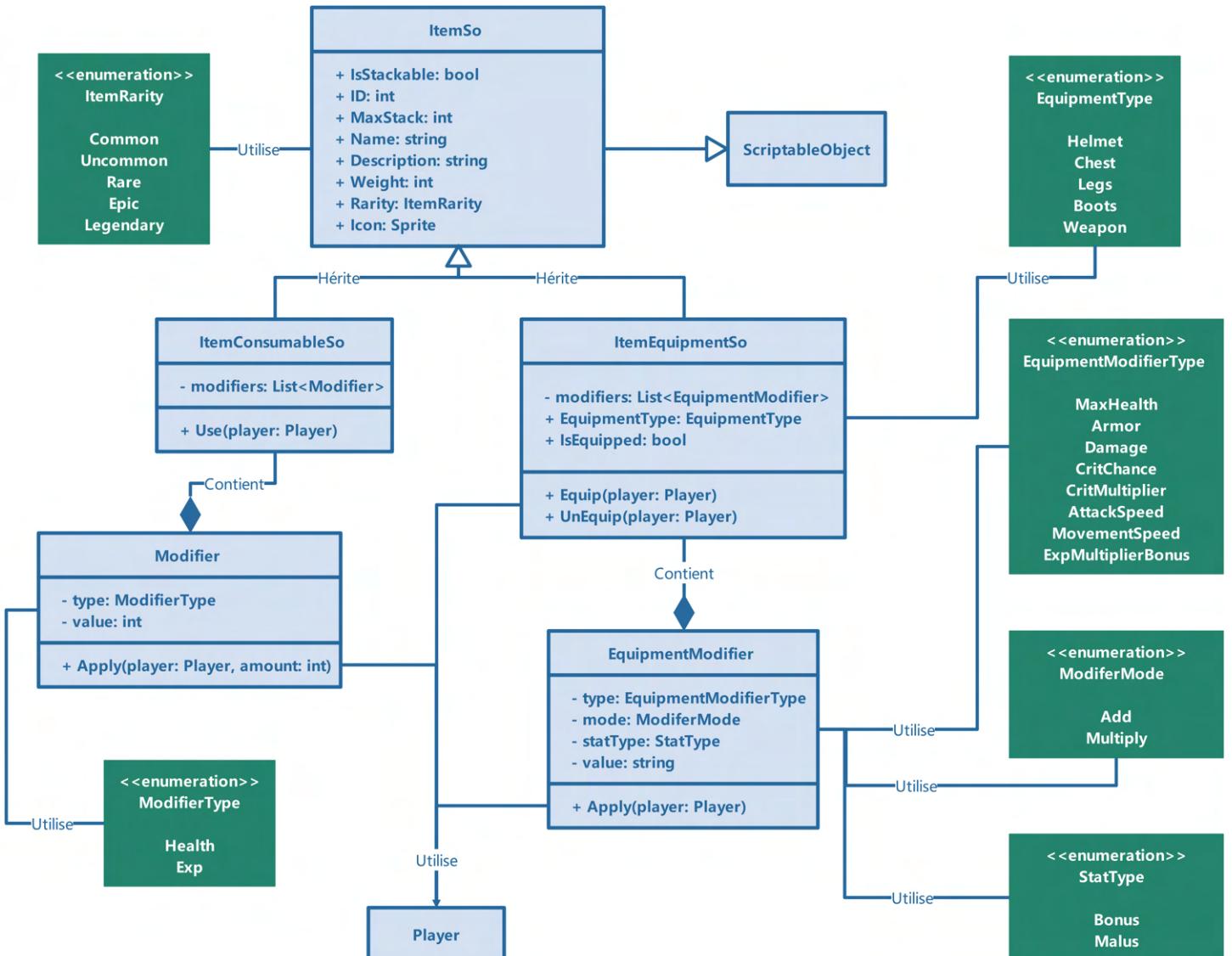


Figure 4.1: UML système d'objets

Encore une fois, comme pour les autres fonctionnalités, pour répondre au besoin de modularité, nous utilisons les **ScriptableObject**. De plus, les blocs verts sur le diagramme UML montrent les **Enums**, nous les utilisons beaucoup, car ils sont sérialisés de base par Unity et permettent un contrôle sur les différentes propriétés. Ainsi l'ajout ou la modification des objets peut être réalisé simplement via l'éditeur grâce à un système de menu déroulant qui permet de choisir entre les différentes valeurs des **Enums**.

4.1.1 Objet de base

L'ensemble de nos objets se base sur le script **ItemSo** qui définit ce qu'est un objet. On y retrouve toutes les propriétés communes, peu importe le type de l'objet. C'est-à-dire :

- **ID** : un identifiant unique qui sert à différencier les objets et, comme nous le verrons plus tard, il nous est utile pour notre système d'inventaire et la synchronisation en ligne.
- **IsStackable** : qui nous permet de définir si l'objet peut être regroupé dans l'inventaire ou s'il doit être affiché de façon unitaire.
- **MaxStack** : qui complète la propriété précédente en définissant le nombre par lequel l'objet peut être regroupé s'il est regroupable.
- **Name** : le nom de l'objet, qui sert à identifier l'objet du point de vue du joueur. C'est notamment cette propriété qui est utile pour l'affichage.
- **Description** : tout comme **Name**, cela nous permet d'avoir une description pour transmettre au joueur via l'UI.
- **Weight** : le poids de l'objet, qui va nous être utile pour l'inventaire afin de limiter le nombre d'objets que le joueur peut transporter.
- **Rarity** : c'est un *enum* qui sert à définir la rareté de notre objet, ce qui pourra être utile dans des futures fonctionnalités.
- **Icon** : le Sprite qui permettra aussi au joueur d'identifier l'objet et sera utile pour l'UI.

4.1.2 Objet consommable

Les objets consommables sont aussi des **ScriptableObject** et ont les mêmes propriétés décrites plus tôt, car ils héritent de **ItemSo**.

En plus de tout ce qui est défini dans **ItemSo**, **ItemConsumableSo** rajoute une liste de **Modifier** et une méthode **Use**.

Ici, **Modifier** est un *struct* avec un champ de type **ModifierType** et **Value**.

- **ModifierType** sert à définir quel paramètre va être affecté par l'objet consommable. Pour le moment, on peut agir sur la vie et l'XP du joueur.
- **Value** sert à définir comment va agir l'effet. Il peut être négatif (pour perdre de la vie) ou positif (pour regagner de la vie ou de l'XP).

Enfin, **Apply**, en utilisant la classe **Player**, va modifier les statistiques du personnage en fonction des paramètres donnés.

Les **Modifiers** sont sous forme de liste, donc leur fonction **Apply** ne suffit pas à elle seule, car elle applique l'effet d'un seul **Modifier**. C'est là qu'intervient la méthode **Use** de **ItemConsumableSo**, qui va, pour chaque **Modifier** de l'objet, appliquer l'effet voulu.

4.1.3 Objet d'équipements

Tout comme les objets consommables, **ItemEquipmentSo** hérite de la classe **ItemSo**. Pour les équipements, nous avons 3 attributs en plus. D'abord **EquipmentType**, qui est un enum, il définit quel type d'équipement est l'objet, un casque, un plastron, une arme, etc.

Nous avons **IsEquipped**, qui permet de savoir si l'objet est actuellement équipé par le joueur. Enfin, le plus important est **modifiers**, qui est une liste de type **EquipmentModifier**, le but ici est de changer les statistiques du personnage en fonction de l'équipement qu'il équipe. **EquipmentModifier** est un struct qui comporte 4 attributs :

- **type** : un enum qui permet de choisir sur quelle statistique l'équipement va faire effet, ça peut être la santé max ou les dégâts, par exemple.
- **mode** : un enum pour choisir entre 2 modes pour appliquer les bonus/malus, soit Add, qui permet de simplement ajouter une valeur, ou Multiply, qui permet d'appliquer en pourcentage, par exemple +30% de dégâts.
- **statType** : un autre enum pour choisir si c'est un bonus ou un malus.
- **value** : la valeur du bonus/malus, sous forme de float ou int.

EquipmentModifier dispose aussi d'une méthode **Apply** qui, tout comme pour les items consommables, va appliquer les statistiques via l'intermédiaire de Player.

Enfin, la classe **ItemEquipmentSo** disposent de 2 méthodes pour équiper et déséquiper l'objet. Ces méthodes appliquent les changements sur le **Player** à l'aide de la méthode **Apply**.

Vous pouvez voir ci-dessous comment nous pouvons configurer chaque type d'objet. Tout comme montré pour le système de dialogue, ces **ScriptableObject** peuvent être créés depuis le menu de Unity et, une fois créés, nous pouvons configurer chaque item.

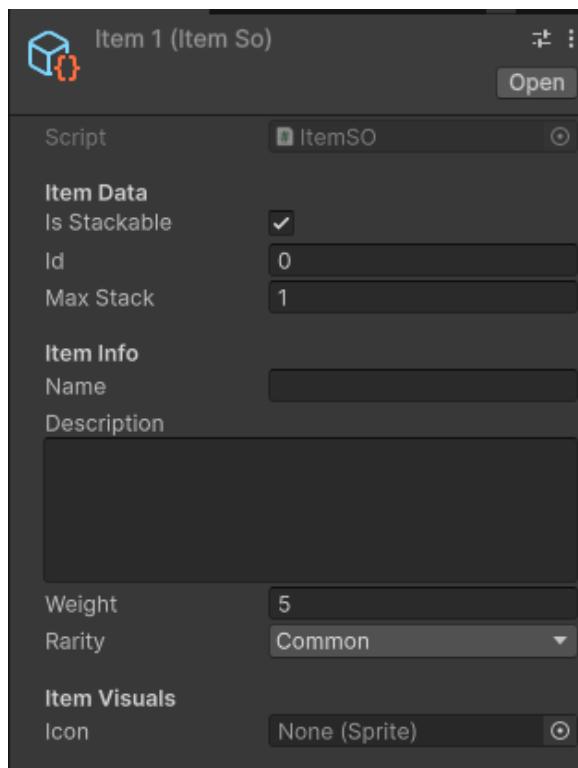


Figure 4.2: ItemSO basique

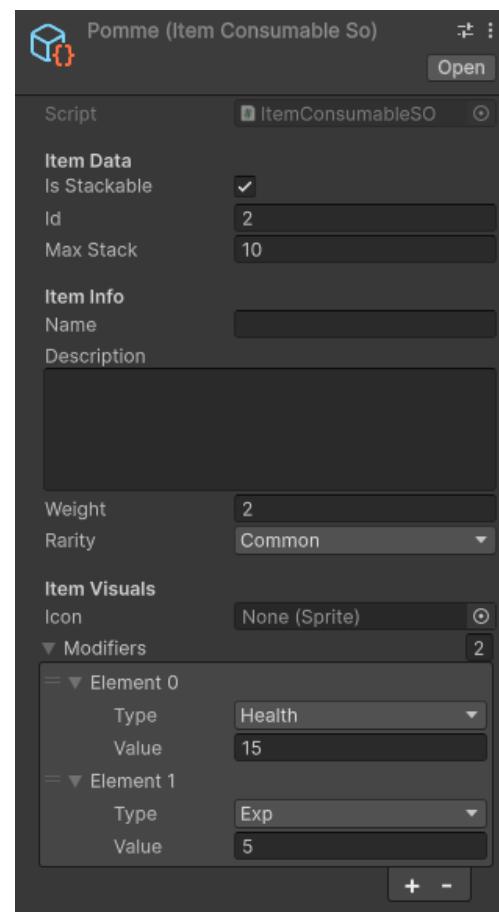


Figure 4.3: ItemSO consommables

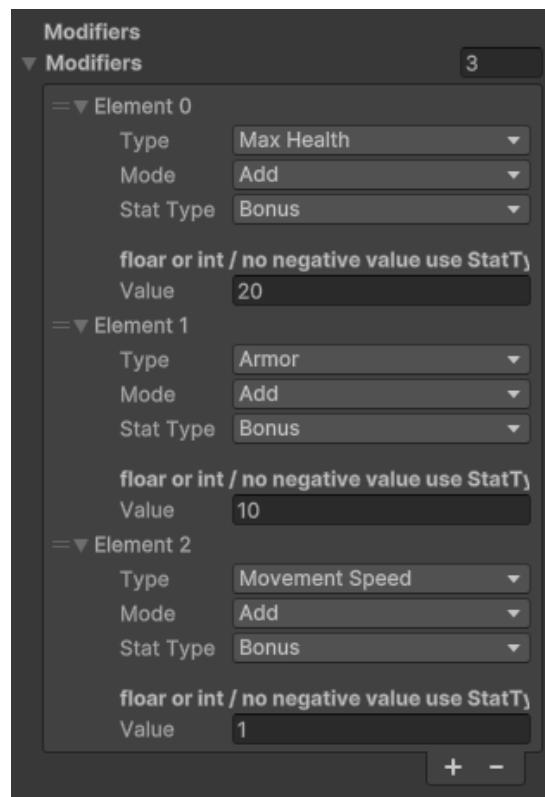
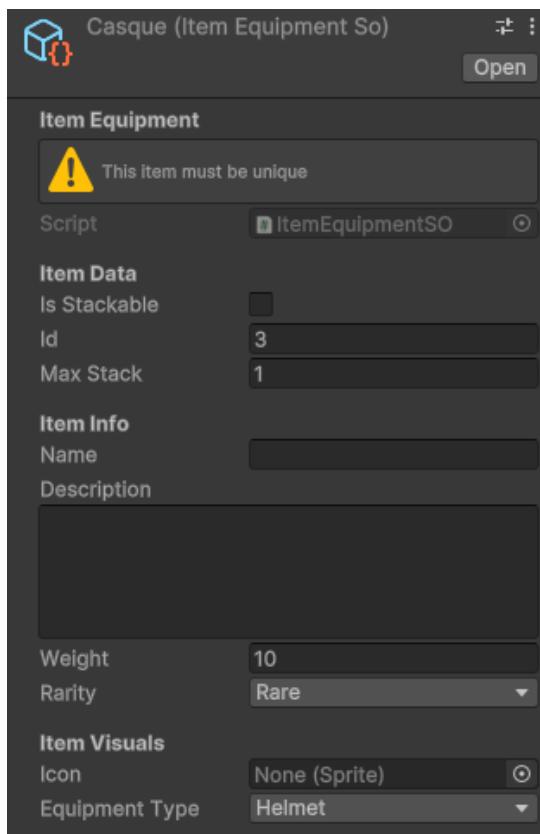


Figure 4.4: ItemSO équipements

4.2 Inventaire

Le plus grand défi lors de la création de l'inventaire a été la synchronisation pour le mode multi-joueur. Effectivement, synchroniser les objets que chaque joueur a en sa possession sera grandement utile pour les prochaines fonctionnalités, comme le système de quête.

Notre première approche a été d'essayer de synchroniser une liste d'une classe de type **Item**. Nous avons vite été confrontés à un problème, car Mirror ne gère pas de base la synchronisation de classes personnalisées. Nous avons profité de ce problème pour revoir complètement la structure de notre système d'inventaire avec une meilleure optimisation.

Nous avons donc choisi de synchroniser un dictionnaire d'entiers, où la clé est l'id de l'objet et la valeur le nombre d'items que possède le joueur. Ensuite, dans un second temps, nous avons une classe qui fait le travail d'une base de données et fait le lien entre l'id de l'objet et ses propriétés. Ce travail se fait seulement en local, ce qui permet une optimisation. En plus de cela, une structure plus complexe de la représentation de l'inventaire est générée en local elle aussi.

Avec cette approche, on synchronise seulement 2 entiers par item dans l'inventaire. Nous avons émis l'idée d'optimiser davantage cela en synchronisant seulement un seul entier par item, qui comporte à la fois l'id et la quantité grâce à des opérations binaires. Nous avons jugé que, pour la compréhension du code, il était préférable de garder notre système de dictionnaire.

Nous allons découper cette partie en 3 sous-parties :

1. **Les fonctionnalités**
2. **UI**
3. **Implémentation**

4.2.1 Les fonctionnalités

Notre inventaire doit pouvoir répondre à plusieurs exigences. Tout d'abord, comme nous pouvons le voir dans l'image, au milieu, les items sont sous forme de grilles où nous pouvons sélectionner chaque objet. Comme dit dans la partie sur **ItemSo**, chaque objet a un poids associé, et le joueur a un poids maximal pour transporter les objets. Ainsi, l'inventaire ne pourra plus recevoir de nouveaux objets si le poids maximal est atteint.

L'inventaire est créé dynamiquement en local en fonction des données synchronisées. Ici, nous pouvons voir que les objets apparaissent bien regroupés en fonction des paramètres donnés dans **ItemSo**.

On peut y voir la quantité de la pile ainsi que l'icône. Dans le cas où nous avons beaucoup d'objets, une barre de défilement apparaît et nous permet d'accéder à l'ensemble des objets.

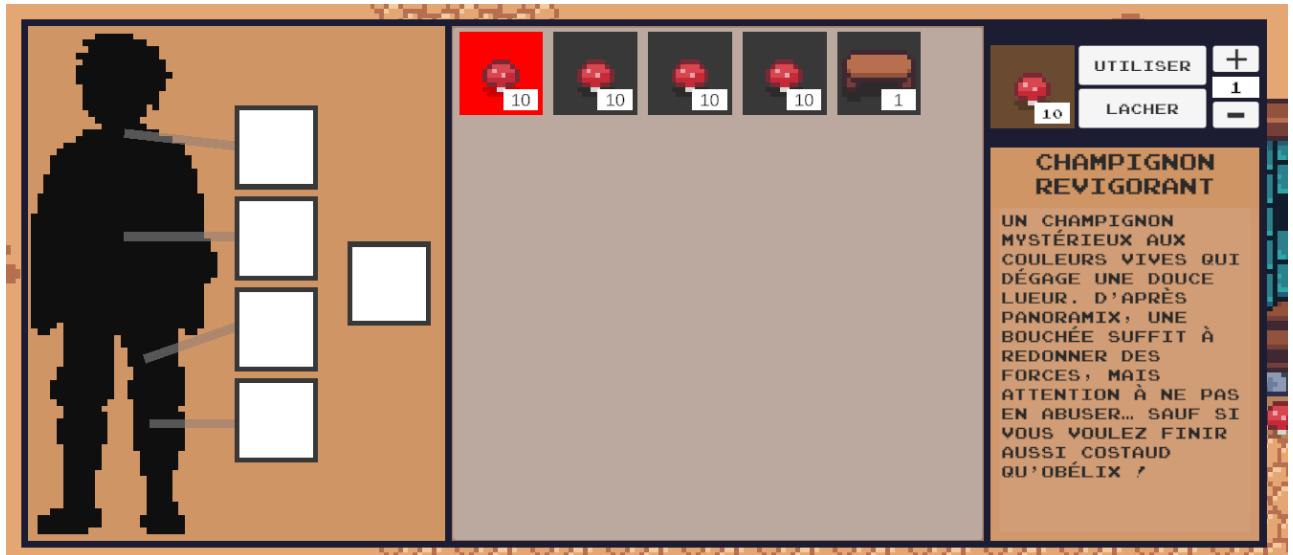


Figure 4.5: Inventaire

Dans la partie de droite, on y voit les informations liées à l'objet actuellement sélectionné. La description, le nom et l'icône sont ceux définis dans **ItemSo**, que nous avons vus dans la partie sur les objets. De même, si la description dépasse l'espace prévu, une barre de défilement apparaît et nous permet de tout lire.

Sur la partie haute, on retrouve l'icône et la quantité de la pile. À droite, les boutons sont générés en fonction du type d'objet. Si l'objet actuellement sélectionné est un objet basique, on aura seulement le bouton "Lâcher", qui, comme nous le verrons dans une prochaine partie, permet de poser au sol un objet qui pourra être ramassé par la suite ou par un autre joueur.

Ensuite, si l'objet est consommable, nous avons un bouton "Utiliser", qui va consommer l'objet comme défini plus tôt dans la partie sur les objets consommables. Ces 2 boutons sont accompagnés d'un sélecteur, qu'on peut voir à la droite des boutons. Le sélecteur est là pour améliorer l'expérience utilisateur et permet de consommer ou lâcher plusieurs objets en même temps.

Enfin, si l'objet est un équipement, un bouton "Équiper" ou "Déséquiper" apparaît en fonction de si l'objet est actuellement équipé ou non.

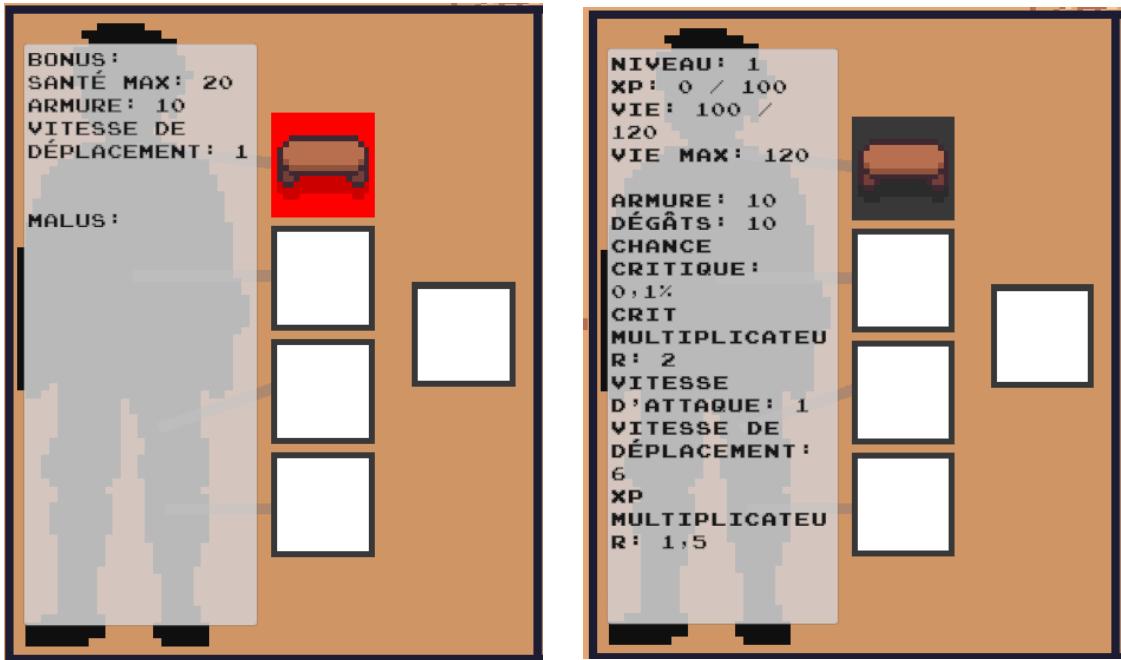


Figure 4.6: Inventaire - Equipements

Pour finir, sur la partie de gauche, on y voit le joueur avec différents emplacements. Ce sont les emplacements des équipements actuellement équipés. En passant la souris dessus, le joueur peut voir les bonus et malus que procure l'équipement. Et comme nous le voyons sur la deuxième image, si le joueur passe la souris sur le personnage, on peut y voir les statistiques complètes du personnage.

Quand le joueur décide de lâcher un objet, il apparaît au sol avec un indicateur de la quantité. N'importe quel joueur pourra alors le ramasser. Si le joueur n'a plus de place dans l'inventaire, il prendra le maximum et la quantité diminuera. Une fois la quantité à 0, l'objet se détruira. Le fonctionnement de cet objet sera expliqué plus en détail dans la partie technique.

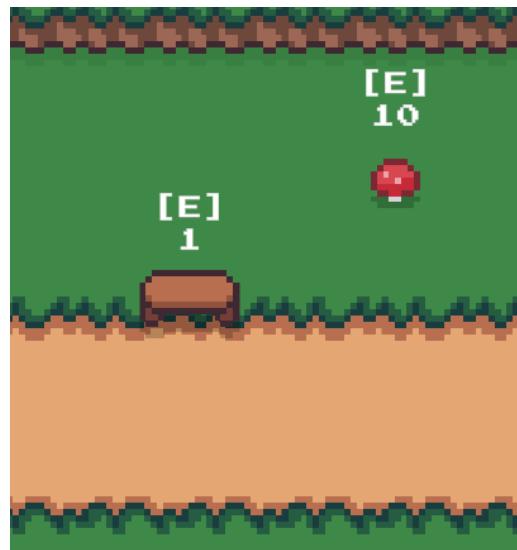


Figure 4.7: Objet à rammasser

4.3 UI

Pour l'UI, afin de bien agencer chaque partie de l'inventaire, nous avons beaucoup utilisé les **Vertical Layout Group** et **Horizontal Layout Group**. Ces composants permettent, par exemple, de séparer dynamiquement une zone en 3 parts égales. C'est aussi grâce à ces composants que nous pouvons afficher les boutons en fonction du type de l'objet. En effet, les boutons actifs vont alors changer de taille dynamiquement en fonction de l'espace disponible.

Pour l'élément central qui affiche les objets, nous utilisons un **Grid Layout Group** qui permet de positionner les objets enfants sur une grille définie à l'avance. Ensuite, nous instancions des **Prefab** pour remplir cette grille et le script sur le prefab se charge de changer le contenu en fonction de l'objet.

Pour récupérer les clics sur les objets de l'inventaire, nous utilisons le composant **Event Trigger** qui permet de choisir quel événement nous intéresse. Dans notre cas, ce sont **OnPointerEnter**, **OnPointerExit** et **OnPointerClick**. Pour chacun de ces événements, nous associons une fonction du script qui s'occupe de l'UI.

Le script qui s'occupe de l'UI a des références pour tous les composants visuels comme les textes, boutons, panels, etc. Cela nous permet de modifier chaque élément de l'UI depuis ce script.

4.4 Implémentation

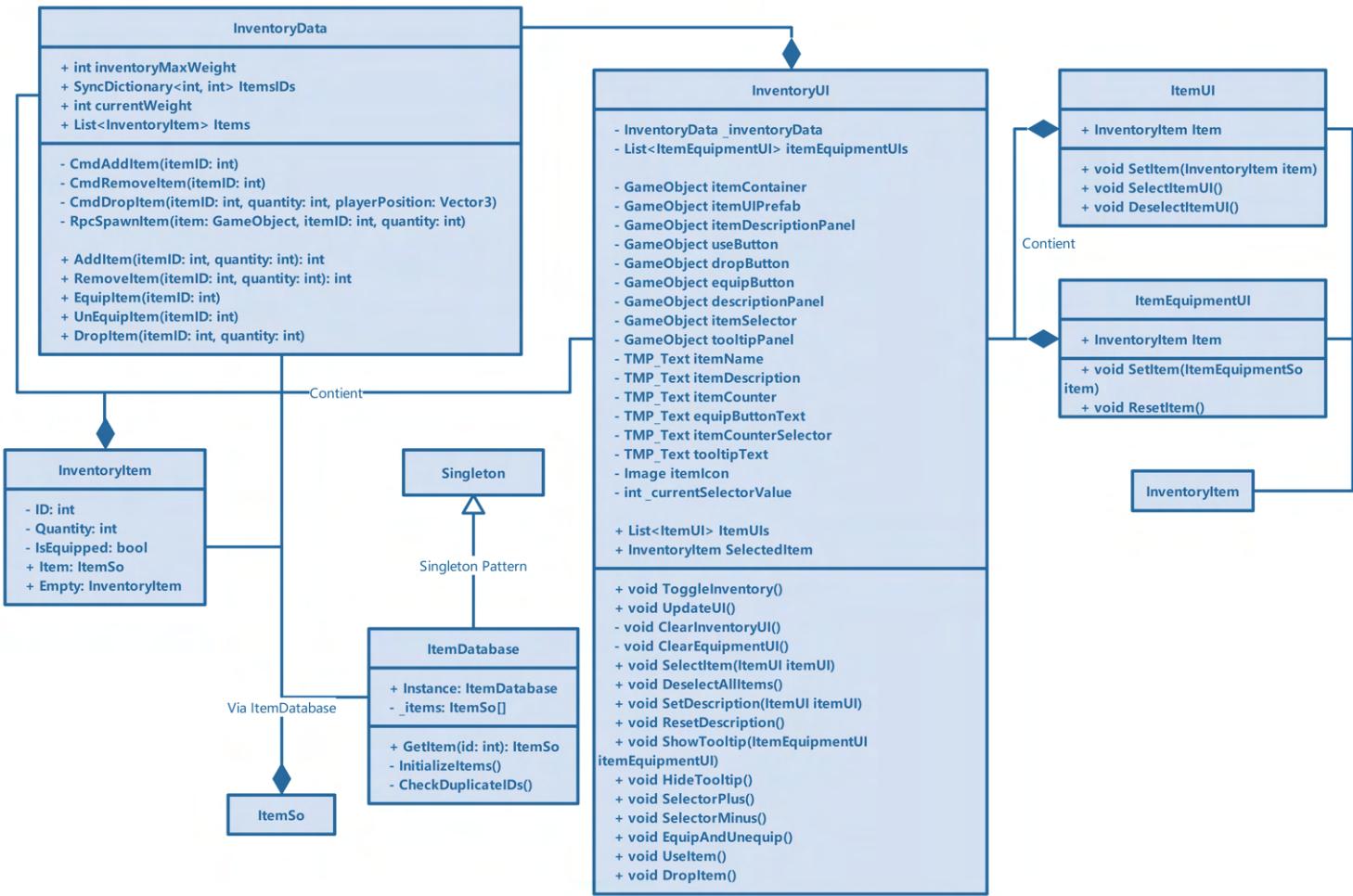


Figure 4.8: UML Inventaire

On peut voir ici le diagramme UML de notre système d'inventaire. Nous avons séparé ce système en deux scripts : un pour le côté technique qui s'occupe de synchroniser les données, ajouter/enlever des objets, déposer des objets au sol, etc., et un deuxième script qui s'occupe de l'UI.

Comme dit plus tôt, pour synchroniser les objets en multijoueur, nous utilisons un dictionnaire d'entiers qui met en mémoire l'ID de l'objet et sa quantité.

Pour faire le lien entre l'ID et l'objet, nous utilisons **ItemDatabase**, qui suit le singleton pattern expliqué dans une partie précédente.

Cette classe, utilisée comme base de données, va faire le lien entre les IDs qui sont synchronisées et les **ItemSo** pour récupérer toutes les données. Ensuite, à l'aide de cette base de données, nous allons créer des **InventoryItem** et les stocker dans une liste non synchronisée.

Cette classe va faire le lien entre les **ItemSo** obtenus grâce à la base de données et la quantité des objets qui sont synchronisés dans le dictionnaire. Cette liste d'objets de type **InventoryItem** nous sera utile pour l'affichage des objets, car cette liste s'occupe aussi de gérer les piles d'objets. Chaque élément de la liste va correspondre à un élément à afficher dans la grille.

Cette liste nous permet seulement de faire le lien entre le dictionnaire synchronisé et l'UI. Les opérations comme l'ajout et la suppression d'objets se feront directement sur le dictionnaire synchronisé, puis en local, la liste s'adapte dynamiquement. L'ensemble de ce processus nous permet un fonctionnement performant avec le mode multijoueur.

Comme nous pouvons le voir dans les méthodes de la classe **InventoryData**, nous avons des méthodes privées et publiques. Le but ici est d'utiliser un niveau d'abstraction sur les méthodes pour ajouter, supprimer, lâcher un objet. En effet, ces méthodes sont un peu complexes, car elles nécessitent une synchronisation multijoueurs. Pour cela, nous utilisons des attributs comme **Command** et **ClientRPC**. Les méthodes avec ces attributs correspondent aux méthodes privées. Les méthodes publiques, quant à elles, vont être utilisées dans les autres scripts, comme par exemple celui pour ramasser des objets ou par le script de l'UI. Ces méthodes publiques utilisent les méthodes privées et peuvent faire des tests supplémentaires en local.

L'attribut **Command**, utilisé pour ajouter et supprimer un objet, est là pour que la fonction puisse être appelée depuis un client et exécutée par le serveur. Ici, pour ajouter un objet, le client demande d'abord au serveur d'ajouter un nouvel objet. Ensuite, le serveur fait les modifications nécessaires et ajoute l'objet. Ce processus permet un contrôle sur les actions du joueur et est utile pour la synchronisation.

L'attribut **ClientRPC** est utilisé pour faire apparaître un nouvel objet sur la scène. Il permet à la méthode d'être appelée depuis le client et exécutée ensuite sur tous les autres clients. Dans notre cas, si nous instancions l'objet sans **ClientRPC**, il sera seulement présent pour le joueur local, mais pas pour les autres joueurs. Ici, cette méthode, étant exécutée sur tous les clients, va bien instancier l'objet sur toutes les instances du jeu.

Le second script, **InventoryUI**, s'occupe de faire le lien entre **InventoryData** et le joueur ainsi que ses actions. Pour cela, **InventoryUI** utilise une liste de type **ItemUI** et **ItemEquipmentUI**, qui est générée grâce aux données récupérées dans **InventoryData**. Ces classes permettent de faire le lien avec les actions du joueur. En effet, ce sont elles qui s'occupent de détecter quand le joueur clique sur un objet pour le sélectionner ou le désélectionner.

À l'aide de toutes ces données, **InventoryUI** instancie des **prefabs** dans la grille vue précédemment. Ensuite, le script possède des références vers tous les **GameObjects**, textes et images dont les valeurs doivent changer dynamiquement.

Enfin, le script comporte des méthodes déclenchables par l'utilisateur, comme celles permettant d'équiper et de déséquiper des objets, d'utiliser un objet consommable ou encore de lâcher des objets. Toutes ces méthodes utilisent les fonctions complexes de **InventoryData** vues précédemment.

4.5 Objets ramassable

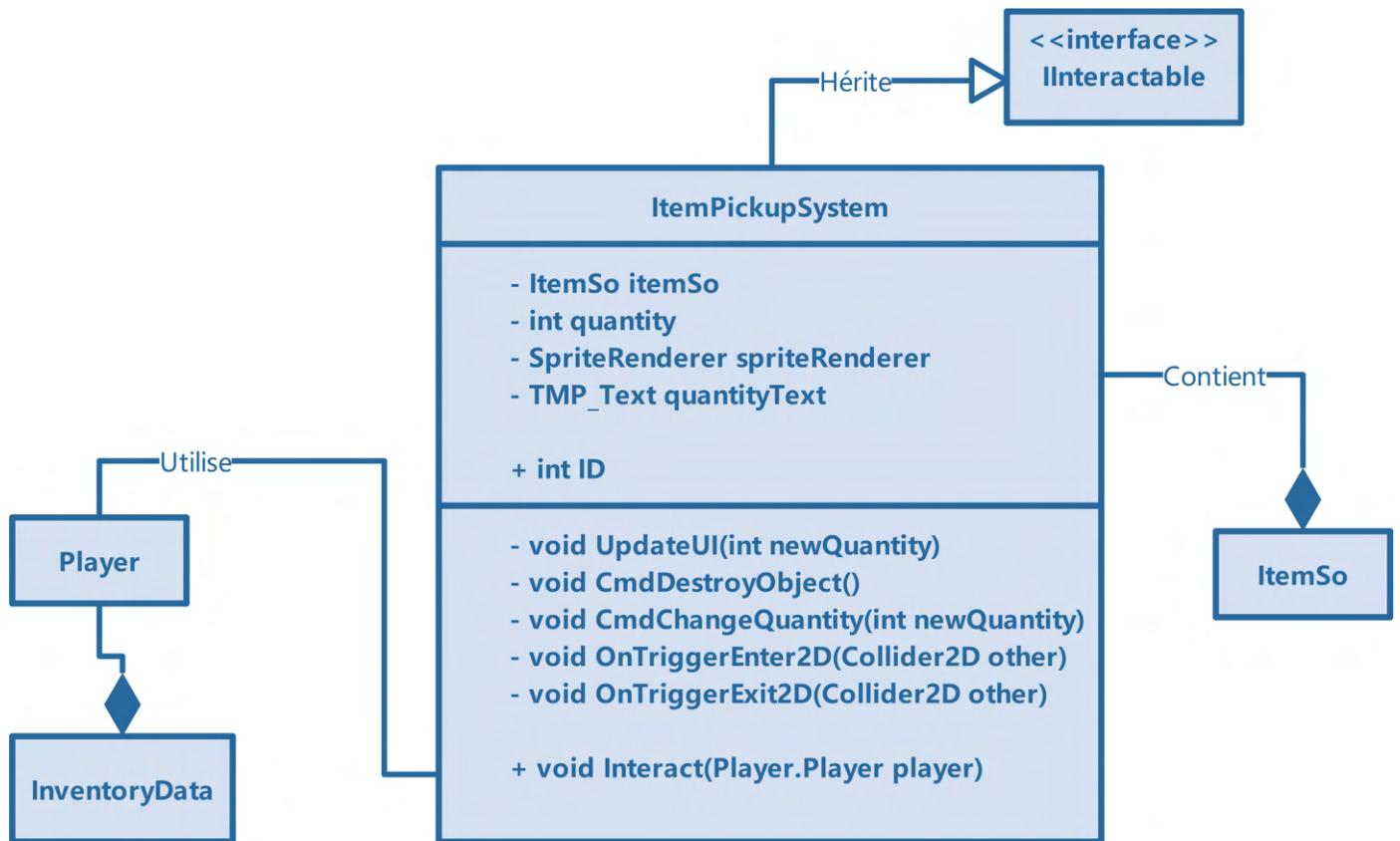


Figure 4.9: UML Item Pickup

Voici la représentation du système d'objets ramassables. Ce système repose aussi sur le principe de **Prefab**. Nous avons créé un **Prefab** utilisant cette classe. Ce **Prefab** dispose d'un sprite pour afficher l'objet actuel à ramasser. Il possède également un texte pour représenter la quantité d'objets à ramasser ainsi qu'un dernier texte pour aider l'utilisateur en affichant la touche à utiliser pour ramasser l'objet.

La classe dispose d'attributs pour stocker des données, comme l'objet actuel représenté par **ItemSO** et un entier pour représenter la quantité actuelle disponible.

La difficulté ici a été de synchroniser les différents objets. En effet, le **Prefab** peut être instancié en jeu lorsqu'un joueur décide, par exemple, de lâcher un objet ou encore en obtenant des récompenses. Pour cela, il faut d'abord ajouter ce **Prefab** à la liste des Prefabs présents dans le **Network Manager** pour que Mirror puisse bien le gérer. Lors de l'instanciation d'un **Prefab**, nous utilisons **ClientRPC** pour exécuter le code sur tous les clients et ainsi instancier l'objet sur toutes les instances du jeu. De

plus, pour détruire l'objet, nous utilisons une commande mise à disposition par **Mirror**, *NetworkServer.Destroy*. À l'instar de **ClientRPC**, cette méthode permet de supprimer l'objet pour tous les clients.

La classe **ItemPickupSystem** dispose également d'une méthode pour changer la quantité disponible. En effet, si un joueur ramasse des objets mais n'a pas assez de place, il faut simplement ajouter le maximum possible et réduire la quantité de l'objet actuel. De même, lorsque un joueur lâche un objet, notre script vérifie d'abord si un objet de type **ItemPickupSystem** est dans les environs et stocke le même type de **ItemSO**. Si c'est le cas, nous ne voulons pas instancier un nouvel objet, mais simplement augmenter la quantité dans **ItemPickupSystem**.

Comme pour les objets de type **DialogueActivator**, notre classe hérite de l'interface **IInteractable**. Cette interface impose donc l'implémentation de la méthode **Interact**, qui utilise **Player**. Ici, cette méthode permet, via **Player**, d'accéder à **InventoryData** et de demander l'ajout d'objets.

Tout comme les autres objets utilisant cette interface, notre **Prefab** possède un composant de type **Collider** et deux méthodes qui permettent de vérifier si un joueur entre dans la zone, afin de lui permettre d'interagir ou non.

Multijoueur

5.1 UX

Afin d'améliorer l'expérience du joueur, il a fallu revoir le système de base de Mirror. En effet, par défaut, avec Mirror, pour créer et rejoindre une partie multijoueur, il faut spécifier manuellement l'adresse IP et le port du serveur que l'on veut héberger ou rejoindre. Ce processus est bien trop complexe et technique pour le type de joueur visé.

L'objectif ici était donc de créer un menu principal, que nous pouvons voir ci-dessous. On y retrouve des boutons pour lancer ou rejoindre une partie.



Figure 5.1: Menu

Pour simplifier la connexion, nous voulions mettre en place un système de code généré par celui qui crée la partie. Les joueurs souhaitant rejoindre n'auraient alors qu'à copier-coller ce code. Pour cela, nous avons implémenté une conversion d'entier vers une base62. La base62 est composée de l'ensemble des lettres majuscules, minuscules et des chiffres.

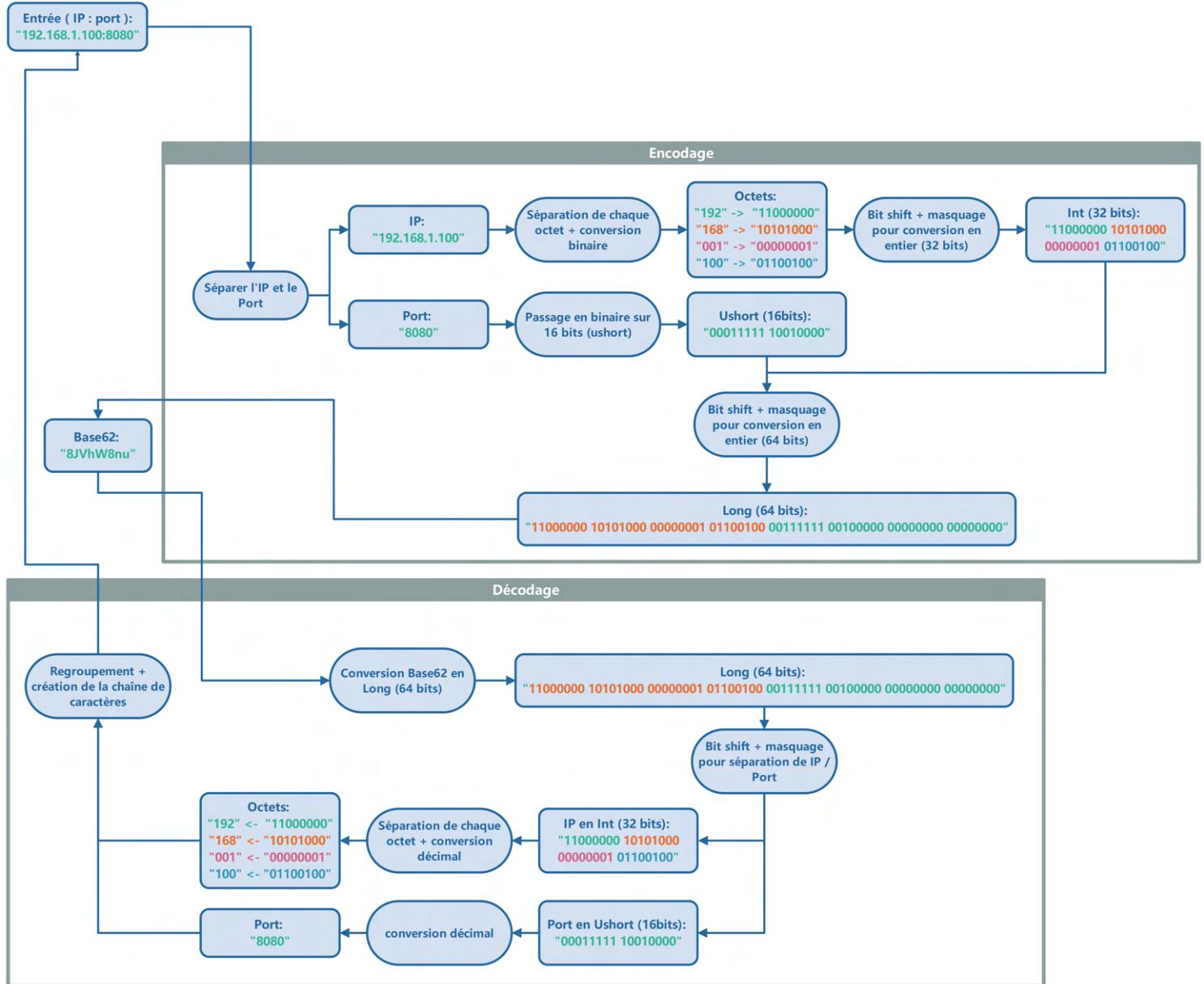


Figure 5.2: Schémas encodage/décodage base62

Le schéma ci-dessus montre le fonctionnement de notre système d'encodage/décodage de l'adresse IP et du port vers un code en base62. Cette manière de procéder nous permet d'assurer un code unique pour chaque adresse IP et chaque port. Nous avons essayé d'optimiser cela au maximum, c'est pourquoi nous avons élaboré notre système à l'aide d'opérations binaires qui nous permettent de stocker toutes les informations sur un seul entier, qui sera ensuite converti en base62.

Nous sommes assez modulaires sur le format de sortie. En effet, notre algorithme peut gérer une base modifiable. Nous pourrions, si nous le souhaitions, utiliser une base contenant uniquement des lettres majuscules ou des chiffres. Bien que cette approche soit possible, nous avons conservé la base62, car elle nous permet d'obtenir un code d'une longueur moyenne de 8 caractères. Une base avec moins de caractères aurait augmenté la taille finale du code.

Ennemis

Un autre point important pour un jeu RPG est le système de combat et donc les ennemis. Pour le moment, nous avons implanté deux types d'ennemis : le premier attaque au corps à corps et le second attaque à distance. Lors de cette période de travail, nous avons seulement implémenté la porte logique et technique. Nos ennemis sont amenés à encore évoluer, tout comme la partie graphique, qui n'a pas encore été implémentée.

Pour la partie du déplacement des ennemis, nous utilisons du pathfinding. C'est pour cela que nous avons cartographié notre carte à l'aide de NavMesh. Cela nous permet de définir les zones où les ennemis peuvent aller ou non.



Figure 6.1: Map navMesh

Ici, les zones bleues représentent les zones que l'ennemi peut parcourir.

Pour comprendre comment sont implantés nos ennemis, nous avons délimité plusieurs zones que nous allons détailler. L'affichage de ces zones sont activables/désactivables depuis l'éiteur et nous aident lors de nos tests. La taille de toutes les zones est configurable pour chaque ennemi. Dans l'exemple donné, il s'agit d'un ennemi au corps à corps, mais le fonctionnement est similaire pour les ennemis à distance, à l'exception de la zone de dégâts.



Figure 6.2: Ennemie

- **Zone bleue** : la zone bleue représente le rayon d'attaque. Une fois dans cette zone, l'ennemi attaque le joueur et inflige des dégâts. Pour un ennemi à distance, cette zone est bien plus grande et représente le moment à partir duquel l'ennemi peut lancer un projectile.
- **Zone jaune** : quand l'ennemi n'est pas en combat et qu'il n'y a pas d'ennemis proches, il va faire des rondes. Ce cercle, contrairement aux autres, est fixe et ne bouge pas en même temps que le joueur. Toutes les 2 secondes, un point aléatoire dans le cercle est choisi, et l'ennemi s'y rend. Cela permet d'ajouter de la vie à nos ennemis et de rendre leur comportement plus naturel.
- **Zone rouge** : cette zone représente la zone de détection. Dès qu'un joueur franchit cette zone, il est poursuivi par l'ennemi.
- **Zone verte** : cette zone est seulement utile quand l'ennemi est en train de poursuivre le joueur. En effet, elle sert à délimiter le moment où notre ennemi perd la trace du joueur. Une fois le joueur poursuivi sorti de cette zone, l'ennemi ne va plus le poursuivre.

Pour ajouter du dynamisme, une fois que le joueur poursuivi sort de la zone verte, l'ennemi effectuera une ronde à son dernier emplacement. La zone jaune se déplacera alors temporairement. Cette ronde dure 10 secondes et, si après ce délai l'ennemi n'a pas retrouvé le joueur, il commencera à retourner à son emplacement initial. Une fois arrivé, la zone jaune retrouvera sa position d'origine. Bien sûr, la détection reste active même lorsque l'ennemi est en train de revenir à sa position initiale.

6.1 Implémentation

Notre implémentation repose sur un système d'états avec un *enum* représentant les différents états possibles : **Ronde**, **Chasse**, **Attaque**, **Retour au point d'origine**. Ainsi, à intervalles réguliers, dans la méthode Update, nous appelons la fonction appropriée en fonction de l'état de l'ennemi.

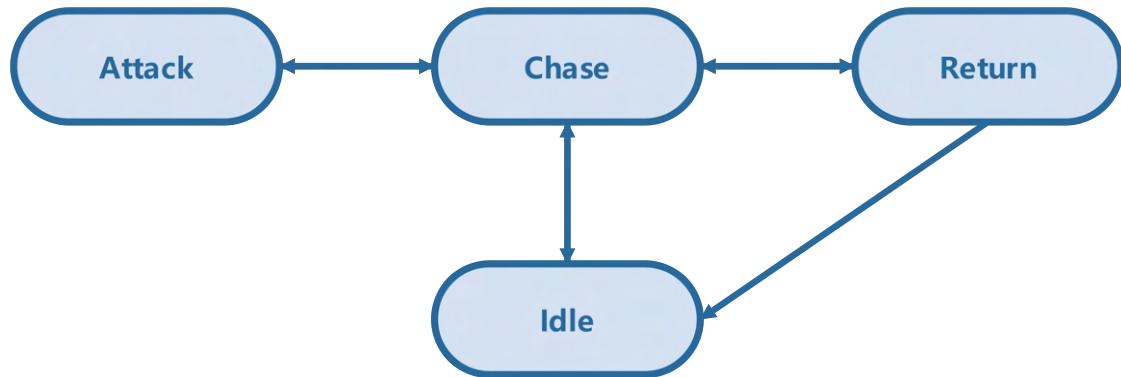


Figure 6.3: Comportement enemie

Voici les changements d'état possibles : l'ennemi commence toujours dans l'état **Ronde**. Il peut ensuite poursuivre le joueur (**Chasse**), puis soit l'attaquer (**Attaque**), soit revenir en ronde, soit retourner à sa position initiale (**Retour au point d'origine**).

Site Web

7.1 Introduction

Afin de promouvoir au maximum un jeu vidéo, le site associé à celui-ci se doit d'être attractif, car dans la plupart des cas, c'est ce que les personnes vont voir en premier lorsqu'elles voudront se renseigner sur le jeu ou lorsqu'elles effectueront une recherche sur Internet avec le nom du jeu. C'est pourquoi, Satis-Kingdom se doit d'avoir un site attractif pour donner une bonne première impression. Un autre objectif pour notre site web est d'informer le visiteur sur deux grands points. Le premier est le fonctionnement du jeu. Le joueur pourra trouver diverses informations telles que l'histoire du jeu, un manuel pour l'installation du jeu contenant aussi les contrôles ainsi que les différentes mécaniques du jeu. Ces informations ne sont cependant pas essentielles à la progression du joueur dans le jeu : le joueur peut directement se lancer dans le jeu sans consulter l'intégralité du site web. Les commandes du jeu seront aussi apprises lors du tutoriel au cas où le joueur ne souhaiterait pas consulter le manuel et apprendre en jouant. Ces informations sont consultables à tout moment de l'aventure s'il souhaite en savoir plus sur le jeu. Le second point concerne l'équipe et la création du jeu dans sa globalité. La page "À propos" est entièrement consacrée à cet aspect. Ici, nous trouverons des informations sur les tâches principales effectuées par chacun des membres ainsi que l'avancée en fonction du temps du projet Satis-Kingdom. Les différents rapports permettant d'en apprendre plus sur le développement du jeu se trouvent dans la partie "Installer".

7.2 Approche de la création

À la suite de différentes réunions avec la totalité de l'équipe et du cahier des charges fourni, le site web connaît maintenant toutes les différentes pages dont il sera composé (FIGURE 7.1). Les autres pages resteront dans le thème de la page d'accueil présentée lors de la première soutenance, car elle a plu à l'ensemble de l'équipe et à la suite des différents retours reçus. Cette page représente bien l'univers du jeu, ce qui est essentiel pour l'équipe afin d'informer, dès le chargement du site, le monde dans lequel l'utilisateur va se plonger.



Figure 7.1

7.3 Conception du site

La page d'accueil du site web a connu beaucoup de modifications. Notre équipe, n'ayant pas d'idée particulière en tête, a dû beaucoup tâtonner pour aboutir à un résultat convenable. C'est pourquoi, pour la suite des pages des schémas globaux, ont été effectués préliminairement à la programmation de la page correspondante. Le thème général étant aussi posé, cela facilite à la visualisation du type de page que nous devons obtenir.

Exemple du croquis puis de la page correspondante :

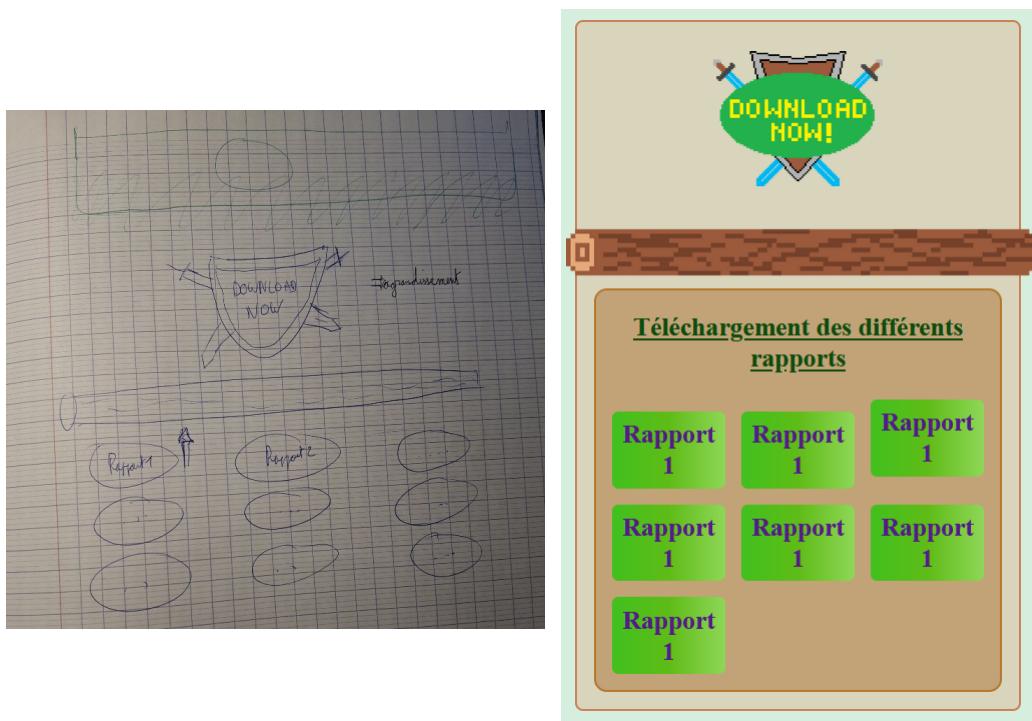


Figure 7.2

Malgré le croquis, certaines modifications sont toujours présentes lors du développement du site pour s'harmoniser au mieux avec le thème. Ces changements restent cependant mineurs et n'affectent pas l'objectif global de la page, mais demeurent néanmoins essentiels pour la bonne lisibilité et l'esthétique de la page, car sur papier, cela ne rend pas de la même façon que sur un écran. Par exemple, la bûche permettant de marquer une séparation sur la page n'était pas présente sur les premiers croquis cependant, à la suite du développement, cette page donnait une impression étrange à notre équipe, d'où l'idée de placer une bûche, ce qui nous permet de rester dans notre thème naturel tout en marquant une séparation.



Figure 7.3

7.4 Présentation du site

La page d'accueil présentera un bref trailer du jeu ainsi que des slides présentant les différentes mécaniques du jeu. Aucune modification n'a été effectuée depuis le premier rapport.

La seconde page sera consacrée à l'histoire du jeu. Cette page sera entièrement optionnelle pour la progression du jeu mais permettra au joueur d'en apprendre plus sur le monde dans lequel il joue ainsi que comment son personnage en est arrivé là. Des illustrations des personnages vont être ajoutées au texte afin de rendre cette page moins monotone.



Figure 7.4

La troisième page se concentrera, comme mentionné précédemment, sur notre équipe, ses membres et les étapes de la conception du jeu.



Figure 7.5

La quatrième page permettra d'installer le jeu ainsi que les différents cahiers des charges et rapports que nous vous avons fournis.



Figure 7.6

Le 5ème onglet permettra de télécharger un PDF correspondant au manuel d'installation, de désinstallation ainsi que les commandes du jeu. Ce manuel doit cependant encore être rédigé.

La dernière page donnera à l'utilisateur accès à nos différents réseaux pour nous contacter afin de nous envoyer des messages, repérer des bugs ou bien partager son expérience avec le reste de la communauté de Satis-Kingdom.

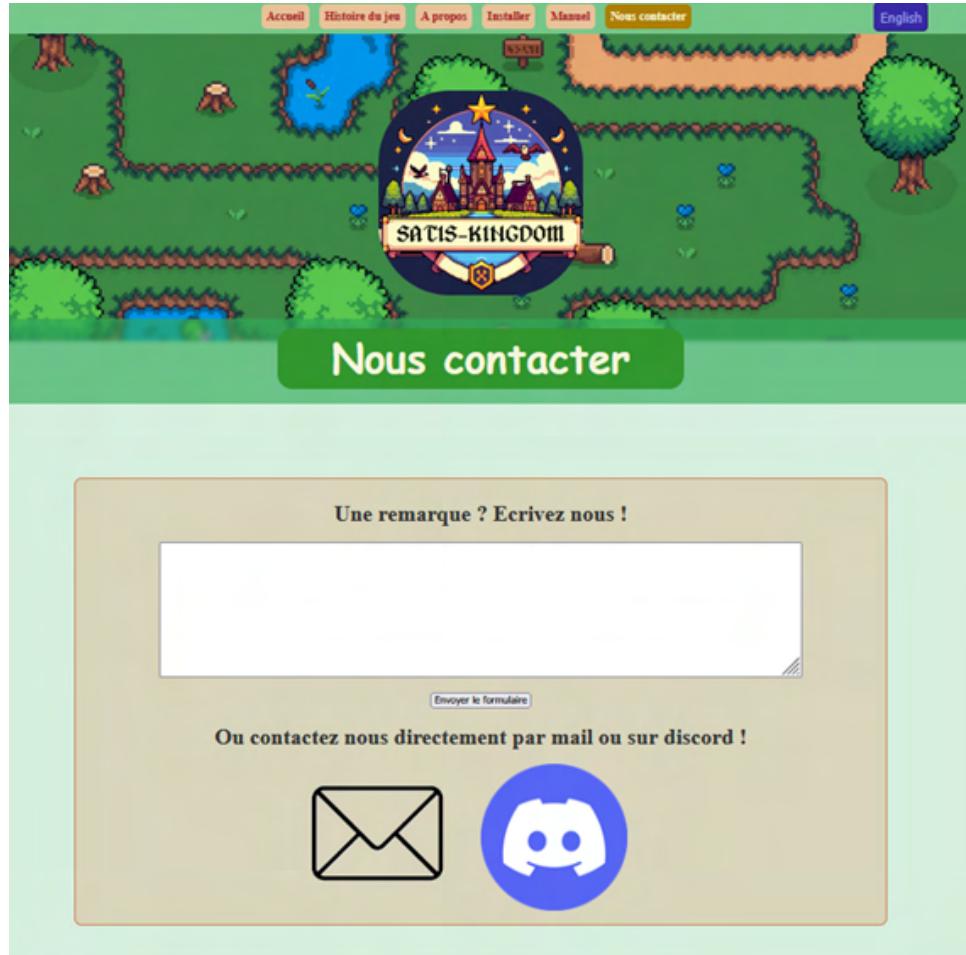


Figure 7.7

7.5 Conclusion :

Concernant le site web de Satis-Kingdom, l'entièreté de la forme des différentes pages a été implémentée, ce qui constitue la partie majeure du site web. Les différentes parties textuelles comme l'histoire et le manuel doivent cependant encore être ajoutées. Des modifications mineures peuvent toujours apparaître après avoir pris du recul sur ce qu'il y a déjà été réalisé, mais cela ne représentera qu'une faible partie du travail qu'il reste à faire sur le site web.

FX

Les effets sonores jouent un rôle essentiel dans le feedback et l'interactivité de Satis-Kingdom. Chaque action du joueur, chaque interaction avec l'environnement et chaque événement clé sont accompagnés de sons soigneusement sélectionnés, assurant ainsi une clarté sonore et renforçant l'immersion. Les FX incluent une large gamme de sons, tels que les bruits de pas, de saut, de mort, les sons d'attaque des ennemis, les interactions avec les objets et les échanges, ainsi que les sons de l'interface utilisateur (UI). Tous ces effets ont été pensés pour correspondre parfaitement aux actions et événements du jeu.

Pour concevoir l'ambiance sonore de Satis-Kingdom, nous avons utilisé des effets sonores libres de droit issus de sites comme UNIVERSAL-SOUNDBANK.COM ou encore LASONOTHEQUE.ORG, que nous avons ensuite modifiés et personnalisés via Audacity afin qu'ils s'intègrent parfaitement à l'univers du jeu.

Cependant, nous avons rencontré plusieurs difficultés, notamment des effets sonores ne correspondant pas toujours à l'ambiance recherchée, la présence de blancs ou de coupures audio, ainsi que des bruits parasites altérant la qualité sonore. Pour pallier ces problèmes, nous avons appliqué différentes techniques d'édition et de montage afin d'obtenir un rendu final cohérent et immersif.

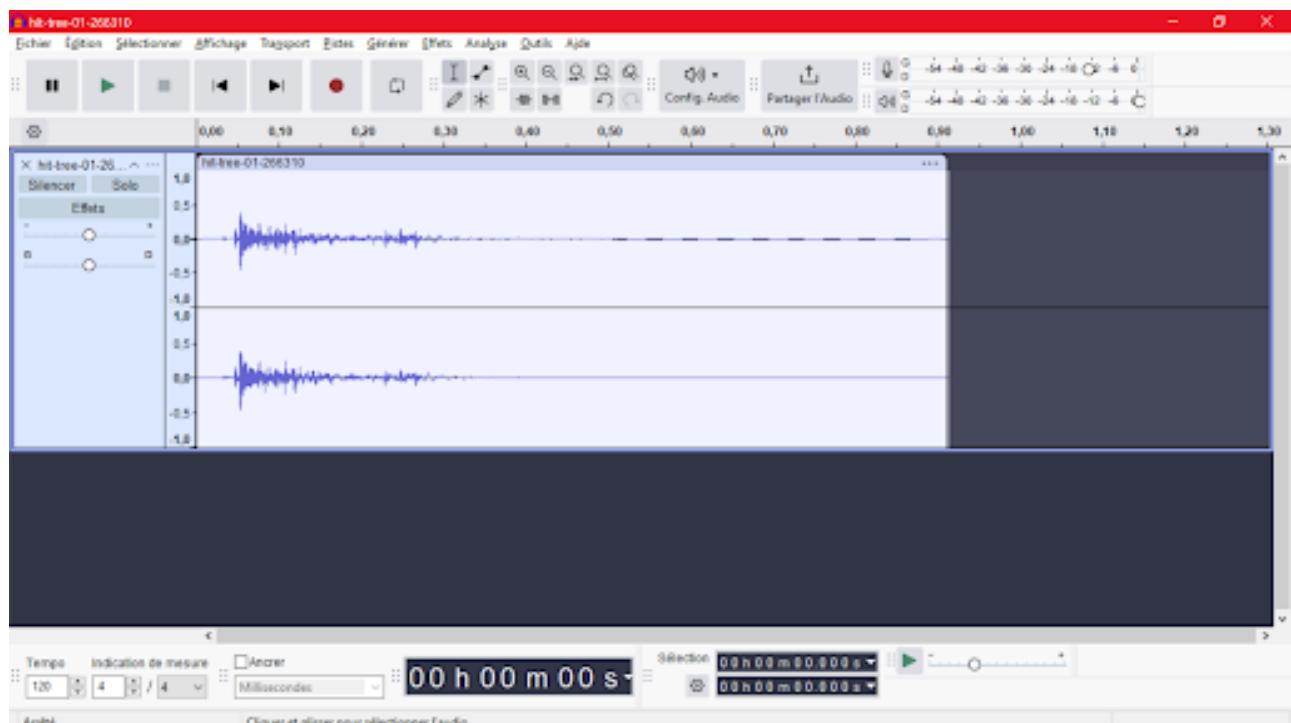


Figure 8.1: Audacity : exemple d'effet sonore

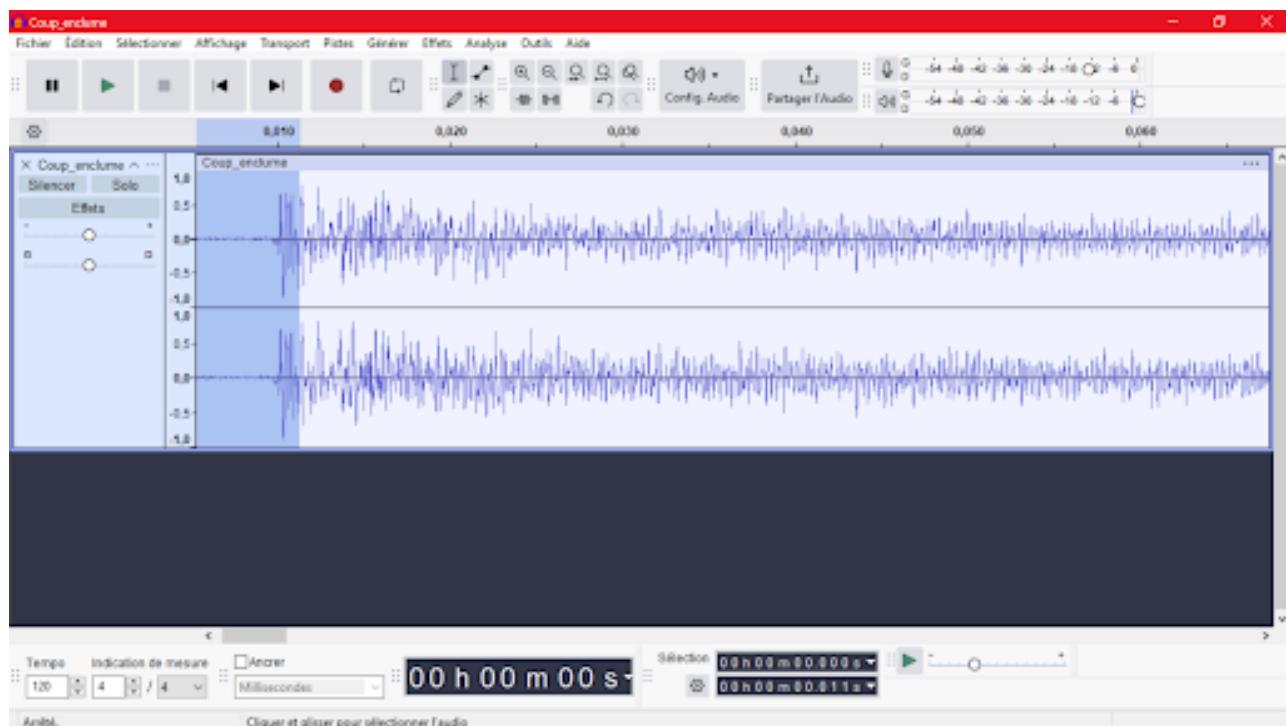


Figure 8.2: Audacity : exemple de bruit parasite

Etat d'avancement

Tâches	Prévisions	Avancements
Game Design		
Création des assets	100 %	95 %
Musiques / sons	100 %	95 %
Rédaction de l'histoire	100 %	100 %
Map Design	100 %	90 %
Programmation		
Système d'inventaire	100 %	100 %
Système de sauvegarde	10 %	10 %
Déplacement / action Personnage	100 %	90 %
Gestion des ennemis	40 %	40 %
Système de dialogue	100 %	100 %
Tutoriel	20 %	20 %
Multijoueur	30 %	60 %
IA	40 %	40 %
Site Web		
Page d'accueil	100 %	100 %
Téléchargement	30 %	30 %
Manuel Utilisateur	30 %	30 %
Autres		
Mise en page Cdc	100 %	100 %

Table 8.1: Etat d'avancement

Globalement, nous avons réussi à avancer sur les tâches que nous voulions. Certaines tâches demandent un peu plus de travail que prévu. C'est pour cela que la partie game design est légèrement en retard, mais cela s'explique principalement par les modifications nécessaires que nous rencontrons au fil du temps et des nouvelles fonctionnalités.

De même, pour la partie actions du personnage, il ne nous était pas possible de la terminer à 100 %, car certaines actions dépendent de fonctionnalités futures qui ne sont pas encore implémentées.

Le multijoueur, quant à lui, est plus avancé que prévu. En effet, nous nous sommes rendu compte que son développement devait se réaliser en même temps que celui des autres fonctionnalités. C'est pour cela qu'il est plus avancé que prévu.

Nous avons aussi eu le temps de travailler sur des fonctionnalités non présentes dans le rapport, comme par exemple un système de connexion multijoueur via un code de connexion.

Conclusion

En conclusion, nous sommes satisfaits des avancées réalisées sur Satis-Kingdom. Malgré certains défis, comme l'optimisation du multijoueur, l'ajustement des effets sonores et l'amélioration de l'IA des ennemis, nous avons réussi à structurer un jeu cohérent et immersif.

À ce stade, nous avons un game design abouti, avec une création d'assets presque finalisée, une map design structuré offrant une carte variée et immersive, ainsi qu'un système de progression bien défini. Le système de dialogue est opérationnel et facilement modifiable, tandis que l'inventaire fonctionne parfaitement, avec une gestion avancée des objets et une UI intuitive.

Nous avons également un mode multijoueur fonctionnel, avec un système de connexion simplifié et des synchronisations optimisées. La gestion des ennemis est bien avancée, avec un système de détection et de poursuite intelligent. L'interface utilisateur a été travaillée pour assurer une navigation fluide et intuitive, et le site web du projet est en grande partie finalisé.

Pour la prochaine soutenance, nous allons nous concentrer sur l'amélioration des mécaniques de combat et de l'exploration, l'optimisation du multijoueur, ainsi que le perfectionnement du sound design afin de renforcer l'immersion. Nous finaliserons aussi les animations et l'interface utilisateur pour améliorer encore l'expérience de jeu.

Nous avons réussi à nous adapter et travailler efficacement pour mener ce projet à son terme. Nous sommes déterminés à mettre tout en œuvre pour finaliser Satis-Kingdom et offrir une expérience de jeu fluide et complète dans sa version finale.