Université Paris Nanterre
Département Informatique – Développement Jeu Vidéo

Rapport Technique de Projet

Développement complet d'un jeu style 2D avec Unity

Auteur :
Mohamed Boughmadi
Mohamet Thiam
Salah-Eddine Bekkari
Étudiants en 2e année
Email : 42014913@parisnanterre.fr
42014728@parisnanterre.fr
41019130@parisnanterre.fr

Encadrant :
Souheib Baarir
Enseignant référent
Université Paris Nanterre

Mai 2025

Table des matières

Liens importants

-[Vidéo Démonstrative] -[Github du projet]

Résumé

Ce rapport technique présente en détail le développement d'un projet de jeu vidéo 2D de style RPG (Role-Playing Game) réalisé avec le moteur Unity. Ce projet, développé dans un cadre pédagogique à l'Université Paris Nanterre, vise à démontrer les compétences acquises en programmation orientée objet, conception logicielle, gestion des interfaces utilisateurs et interactions en temps réel.

Le cœur du projet repose sur la création d'un système d'inventaire actif, couplé à une gestion complète des contrôles utilisateurs à travers le nouveau *Unity Input System*. L'objectif principal est de permettre une navigation fluide, intuitive et dynamique dans l'univers du jeu, tout en offrant une expérience utilisateur immersive et cohérente.

Le développement a suivi plusieurs étapes essentielles : analyse des besoins fonctionnels, modélisation de l'architecture du jeu, conception des interfaces et interactions, implémentation des scripts (notamment pour l'inventaire, les transitions UI et les contrôles utilisateur), ainsi que des phases rigoureuses de tests et validation.

Une attention particulière a été portée à l'optimisation des performances, à la modularité du code et à la maintenabilité du projet. Chaque composant a été conçu afin de favoriser la réutilisabilité, avec une séparation claire des responsabilités à travers des scripts spécialisés. Des extraits de code commentés sont présentés pour illustrer les mécanismes internes.

Enfin, le rapport aborde les perspectives d'évolution, telles que l'amélioration de l'interface utilisateur, l'ajout de nouvelles fonctionnalités liées à l'équipement du joueur, ainsi que l'enrichissement des effets visuels pour une meilleure immersion.

Chapitre 1

Introduction

1.1 Contexte du projet

Le projet présenté dans ce rapport s'inscrit dans le cadre d'un cursus universitaire en informatique. Il s'agit du développement complet d'un jeu vidéo de style RPG en 2D, réalisé avec le moteur Unity. Ce choix découle d'une volonté personnelle de relever un défi technique et créatif, en réalisant un projet ambitieux qui se démarque par son originalité et sa complexité.

L'idée principale était de concevoir un jeu proposant des mécaniques de combat exigeantes, avec des patterns d'ennemis difficiles, afin d'offrir une expérience stimulante pour le joueur. Le but n'était pas simplement de créer un jeu fonctionnel, mais également d'approfondir la maîtrise des concepts fondamentaux de la programmation orientée objet (POO) et des systèmes interactifs complexes, dans un contexte réel de développement.

Le projet se distingue par son caractère solo, contrairement aux projets multijoueurs souvent plus complexes à gérer en termes de réseau. Cette orientation permet de concentrer les efforts sur la qualité du gameplay, l'ergonomie de l'interface utilisateur, et la fluidité des interactions. De plus, le choix de la 2D s'inscrit dans une démarche esthétique et technique adaptée aux ressources disponibles et aux contraintes temporelles du projet.

1.2 Objectifs

L'objectif principal de ce projet était double : d'une part, créer un jeu vidéo solide, cohérent et plaisant à jouer, intégrant un système d'inventaire actif et une gestion des contrôles utilisateur efficaces, et d'autre part, renforcer et aiguiser mes compétences techniques, notamment en programmation orientée objet, en développement Unity, et en gestion de projets logiciels.

Ce projet représente un véritable défi technique puisque nous n'avions initialement aucune connaissance pratique de Unity. Apprendre à utiliser ce moteur de jeu, maîtriser le langage C#, comprendre le fonctionnement des systèmes d'entrée utilisateur (Unity Input System), et concevoir une architecture modulaire et évolutive ont constitué des objectifs pédagogiques essentiels.

Par ailleurs, l'accent a été mis sur la qualité du code, la maintenabilité, et la performance. Le système de combat devait proposer des patterns d'ennemis complexes afin d'assurer une difficulté progressive et un challenge intéressant, tandis que l'interface utilisateur devait être intuitive et responsive.

Enfin, ce projet a également pour but de développer une méthodologie rigoureuse de travail, incluant la conception UML, les tests et validations, ainsi que les optimisations nécessaires pour garantir une expérience utilisateur optimale.

1.3 Technologies utilisées

Le choix des technologies a été guidé par la volonté d'apprendre un outil professionnel largement utilisé dans l'industrie du jeu vidéo : Unity. Ce moteur de jeu, combiné au langage C#, offre une grande flexibilité pour la création de jeux 2D et 3D, ainsi qu'un large écosystème de ressources et de documentation.

Pour la gestion des entrées utilisateur, le système Unity Input System a été adopté, permettant une abstraction avancée des contrôles, une meilleure gestion des événements et une intégration facilitée avec différents dispositifs d'entrée.

Le projet a également utilisé divers outils annexes pour la modélisation (diagrammes UML réalisés avec des logiciels dédiés), le contrôle de version (Git), et la documentation (LaTeX), dans une approche professionnelle de développement logiciel.

Cette combinaison technologique a permis de répondre aux exigences fonctionnelles du projet tout en assurant un cadre propice à l'apprentissage et à l'expérimentation.

Chapitre 2

Analyse des besoins

2.1 Description fonctionnelle

Le projet consiste à développer un jeu vidéo de type RPG en 2D avec un système d'inventaire actif, une gestion dynamique des armes, et des interactions utilisateur intuitives.

Les fonctionnalités principales attendues sont :

— **Gestion de l'inventaire actif :** permettre au joueur de sélectionner et changer d'armes rapidement via un système de slots visibles à l'écran.

— **Système de combat dynamique :** intégration de différentes armes (arc, bâton magique, etc.) avec des mécaniques propres et animations spécifiques.

— **Contrôle utilisateur fluide :** utilisation du nouveau système Unity Input System pour gérer les entrées clavier, souris et autres périphériques.

— **Interface utilisateur ergonomique :** affichage clair de l'inventaire, effets visuels adaptés lors du changement d'armes et retour visuel lors des actions.

— **Gestion des états du joueur :** transitions entre états (idle, attaque, déplacement) et interactions avec l'environnement.

— **Extension possible :** architecture modulaire permettant l'ajout futur de nouveaux types d'armes, ennemis, et mécaniques.

Ces fonctionnalités sont destinées à offrir une expérience de jeu immersive, où le joueur peut interagir de manière intuitive avec l'interface et réagir rapidement lors des combats.

FIGURE 2.1 – Inventaire en jeu

Le système d'inventaire actif, cœur du gameplay, doit être réactif et permettre un changement d'arme immédiat, avec une indication visuelle claire du slot actif.
De plus, chaque arme dispose de caractéristiques propres (dégâts, portée, cooldown) qui influencent directement le gameplay.

Le contrôle utilisateur doit être précis, avec une gestion d'événements propre et efficace, évitant toute latence perceptible. L'interface doit non seulement informer le joueur mais aussi le guider dans ses choix grâce à des animations et effets visuels adaptés.

Ces exigences fonctionnelles ont été définies en s'appuyant sur une analyse approfondie des besoins des joueurs visés, ainsi que sur des références dans le domaine des jeux d'action-RPG 2D.

2.2 Contraintes techniques

Le projet a été développé dans un cadre pédagogique avec un temps et des ressources limités. Ces contraintes ont orienté plusieurs choix techniques :

— **Limitation du scope :** le jeu est un projet solo en 2D, évitant la complexité du réseau ou de la 3D afin de garantir une réalisation complète dans les délais impartis.
— **Performance :** Unity doit tourner de manière fluide sur des machines standards (PC moyen de gamme). Le code et les assets ont donc été optimisés pour limiter la consommation mémoire et CPU.
— **Simplicité des assets graphiques :** les graphismes sont volontairement minimalistes et en 2D, pour se concentrer sur la programmation et le gameplay plutôt que sur la création artistique poussée.
— **Compatibilité multiplateforme :** le projet cible principalement Windows, mais la structure du code et les ressources ont été pensées pour faciliter un portage futur vers d'autres plateformes.
— **Utilisation du Unity Input System :** la migration vers ce système moderne impose une architecture spécifique pour la gestion des entrées, notamment en termes d'événements et callbacks.
— **Modularité et maintenabilité :** le code doit être structuré selon les principes SOLID, facilitant les tests, les corrections et les évolutions.
— **Gestion des dépendances :** limitation à l'utilisation des packages Unity standards, évitant les dépendances externes complexes ou coûteuses.

Ces contraintes ont orienté la conception et le développement du projet, en veillant à respecter un équilibre entre ambitions fonctionnelles et réalisme technique.

2.3 Cahier des charges

Le cahier des charges formalise les attentes fonctionnelles et techniques du projet de développement du jeu vidéo 2D sous Unity. Il sert de référence tout au long du processus de conception et de réalisation, garantissant que le produit final respecte les objectifs définis.

Les exigences principales sont les suivantes :

Table 2.1 – Synthèse des exigences principales du cahier des charges

Catégorie	**Exigences**
Fonctionnalités principales	— Système d'inventaire actif avec sélection en temps réel — Gestion des contrôles via Unity Input System — Interface utilisateur intuitive avec effets visuels — Patterns d'ennemis progressifs en difficulté
Qualité et performance	— Fluidité d'exécution sur machines standard — Code modulaire, propre et documenté — Interface ergonomique et réactive
Contraintes techniques	— Utilisation de Unity et C# — Respect du Unity Input System — Limitation à la 2D
Livrables attendus	— Code source complet et commenté — Documentation technique et diagrammes UML — Rapport détaillé — Présentation orale

Ce cahier des charges a guidé le développement tout au long du projet, assurant une cohérence entre les besoins initiaux et les solutions techniques mises en œuvre. Les critères de qualité et de performance ont notamment orienté les choix d'architecture et d'implémentation, tandis que les contraintes techniques ont limité le périmètre fonctionnel à une version 2D maîtrisable dans le cadre temporel et humain disponible.

Une phase de validation des livrables a permis de vérifier la conformité avec ce cahier des charges, assurant que le produit final répond aux objectifs pédagogiques et techniques fixés.

Chapitre 3

Conception du projet

3.1 Architecture générale

L'architecture du projet a été pensée pour être modulaire, claire et extensible. Elle repose sur plusieurs modules principaux qui communiquent entre eux via des interfaces bien définies, facilitant ainsi la maintenance et l'ajout de fonctionnalités futures.

Le schéma ci-dessous illustre cette architecture générale, en mettant en évidence les principaux composants et leurs interactions :

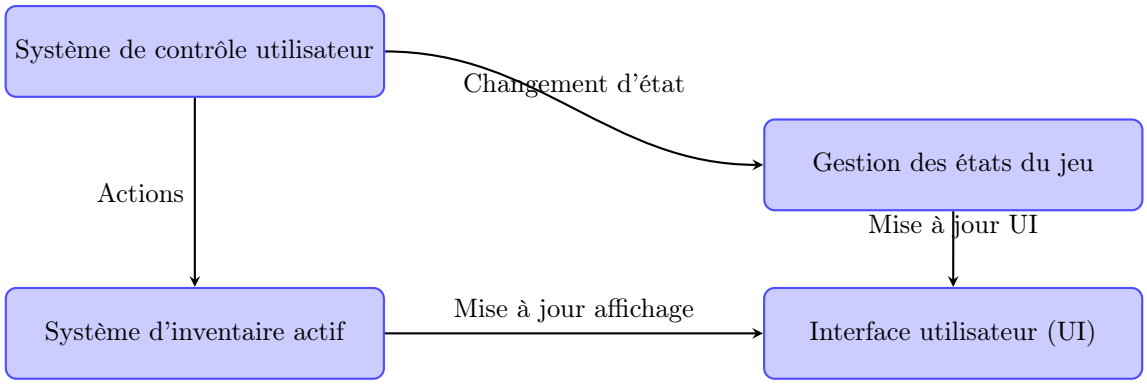

FIGURE 3.1 – Architecture simplifiée du projet

Ce modèle favorise une séparation claire des responsabilités :

— Le **système d'inventaire actif** gère la structure des données liées aux objets et armes, ainsi que les interactions utilisateur sur l'inventaire.
— L'**interface utilisateur (UI)** affiche les informations et propose des effets visuels dynamiques, notamment lors du changement d'armes ou de l'ouverture/fermeture de menus.
— Le **système de contrôle utilisateur** capture et interprète les entrées des joueurs via le Unity Input System, assurant une gestion fluide des commandes.
— La **gestion des états du jeu** contrôle les transitions de scènes, les modes de jeu (combat, exploration), et assure la cohérence des interactions.
— Le module **gestion des ennemis et IA** traite les comportements et patterns d'ennemis, avec une boucle d'exécution continue et adaptative.

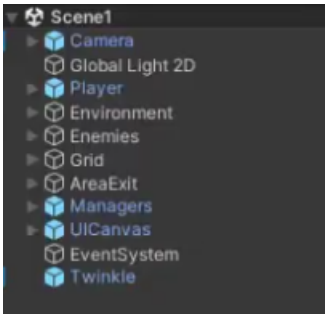

FIGURE 3.2 – Hiérarchie dans Unity

3.2 Diagrammes UML

Pour une meilleure lisibilité, le diagramme UML représentant l'architecture du projet a été divisé en deux parties complémentaires. Cette représentation permet d'observer les principales classes, leurs relations et responsabilités dans le système.

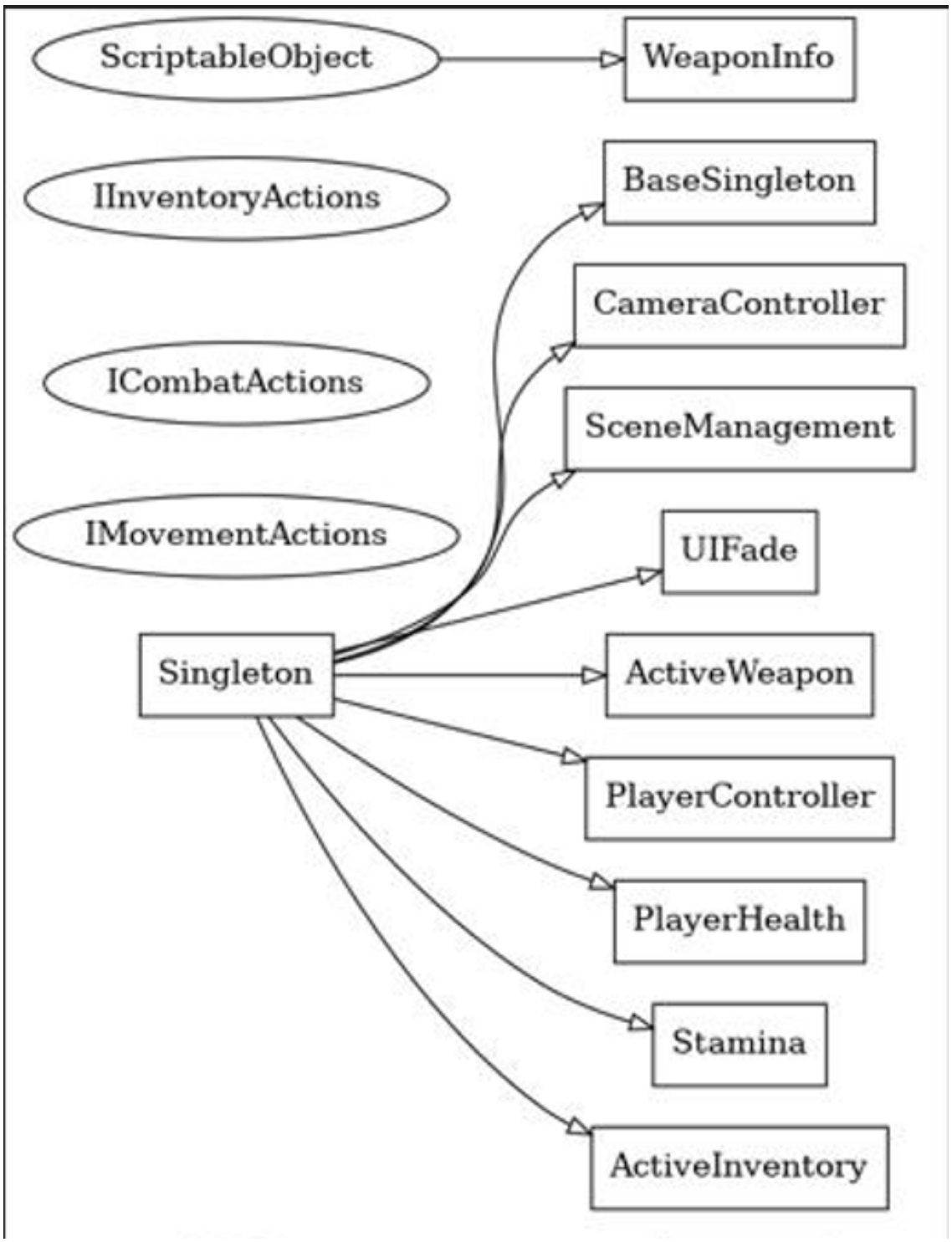

Figure 3.3 – Diagramme UML - Partie 1

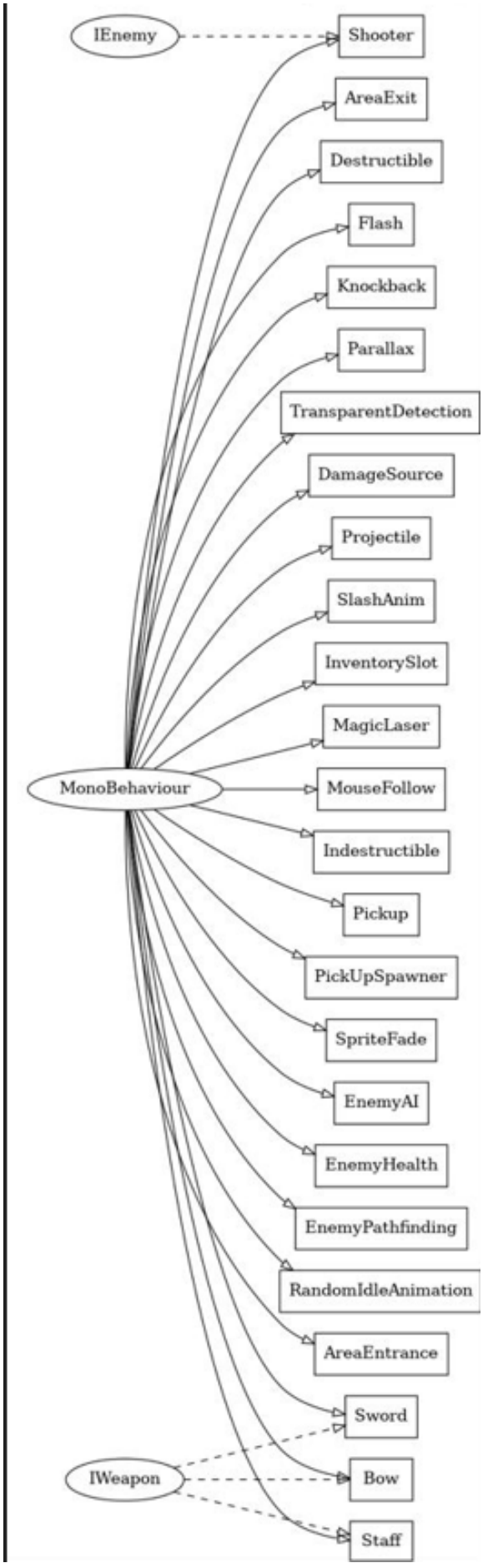

FIGURE 3.4 – Diagramme UML - Partie 2

3.3 Description des modules principaux

Le projet repose sur une architecture modulaire visant à séparer les responsabilités logicielles de manière cohérente et évolutive. Cette approche facilite la maintenance, les tests unitaires, ainsi que l'ajout futur de nouvelles fonctionnalités. Chaque module a été conçu pour être indépendant autant que possible, tout en collaborant efficacement avec les autres composants du système.

3.3.1 Module Joueur (PlayerController)

Ce module gère tous les aspects liés au joueur : mouvement, orientation, interaction avec l'environnement, et utilisation des objets. Il exploite l'Unity Input System pour une gestion fine des commandes et une compatibilité multiplateforme. Les fonctionnalités couvertes incluent :

— Déplacement du personnage via un vecteur de direction calculé à partir des axes d'entrée.
— Orientation dynamique du joueur en fonction de la position de la souris.
— Appel des actions de tir, d'interaction ou de changement d'arme via les événements d'entrée.
— Animation synchronisée avec l'état courant (idle, walking, shooting).

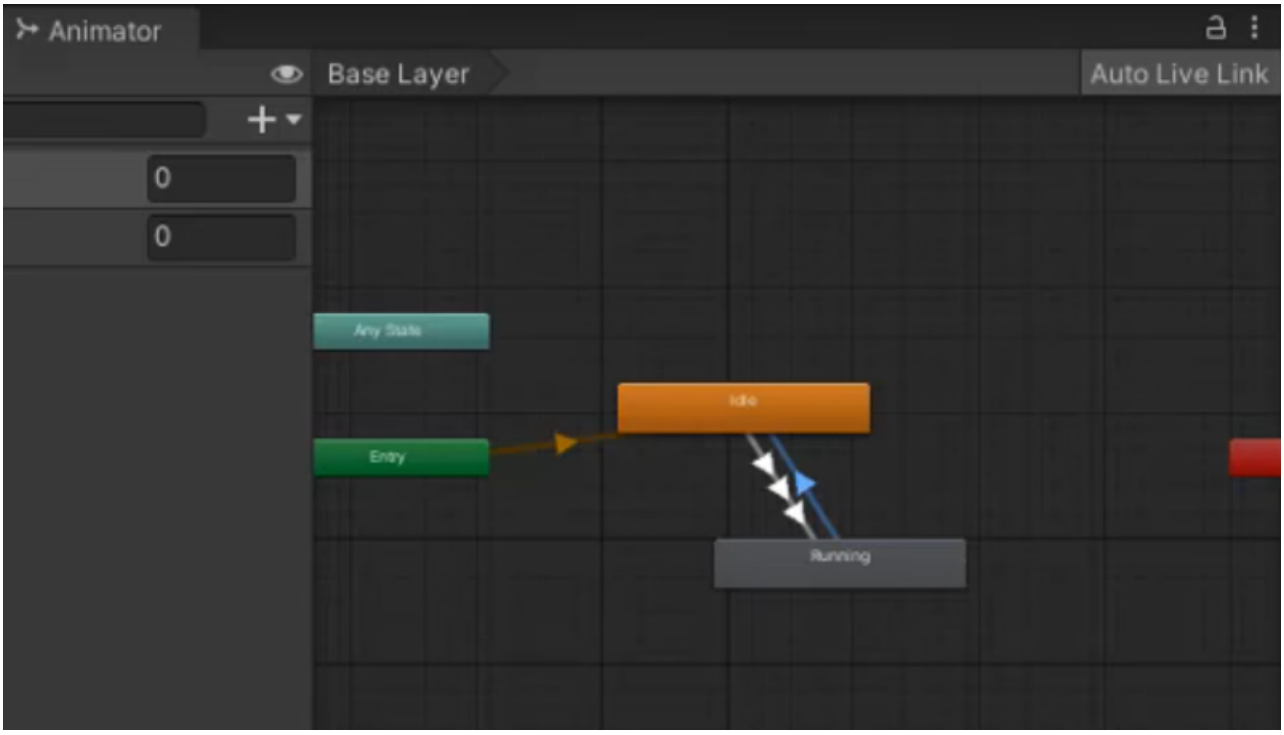

FIGURE 3.5 – Outil d'animation(Animator) dans Unity

3.3.2 Module Inventaire Actif (ActiveInventory)

Ce module implémente un système d'inventaire rapide permettant de stocker, afficher et basculer entre différentes armes ou objets d'usage rapide. Il est visuellement représenté à l'écran, associé à des raccourcis clavier (1-2-3...) configurables. Il contient :

— Une structure de données optimisée (liste indexée ou dictionnaire) pour gérer les slots d'armes.
— Des fonctions de navigation circulaire entre les éléments (suivant / précédent).
— Une liaison directe avec le système d'affichage UI et le joueur.
— La logique de changement d'arme déclenchée par le joueur ou l'environnement.

3.3.3 Module Système d'Entrée (InputManager)

Ce module abstrait l'utilisation du système d'entrée d'Unity afin de séparer les actions utilisateur des traitements métier. Il comprend :

— La définition des cartes d'actions (Input Action Maps) pour chaque contexte (jeu, UI...).
— L'enregistrement des callbacks vers les fonctions cibles (tir, interaction, pause).
— Un contrôle du focus selon les états du jeu (par exemple, désactivation des mouvements durant une cinématique).

3.3.4 Module UI (UIManager)

Ce gestionnaire centralise les interactions avec tous les éléments de l'interface utilisateur. Il agit comme une façade entre les données de jeu et leur représentation visuelle. Ses rôles principaux sont :
— Affichage du HUD, des slots d'armes et des effets visuels.
— Synchronisation dynamique avec l'inventaire actif.
— Fondu d'éléments via des animations de type "Fade In / Fade Out".
— Adaptation automatique à différentes résolutions d'écran.

3.3.5 Module GameManager

Ce module assure le pilotage global du déroulement du jeu. Il orchestre les états du jeu (exploration, pause...) et centralise la logique de haut niveau. Il s'occupe notamment de :
— Initialiser les autres modules au démarrage.
— Maintenir l'état courant du jeu (enum GameState).
— Relayer les événements globaux (victoire, défaite, changement de scène).

3.4 Système d'inventaire actif

3.4.1 Structure des données

Le système d'inventaire actif repose sur une architecture légère et réactive, visant à permettre une gestion dynamique des armes ou objets à disposition du joueur. Chaque objet est représenté par une structure simple contenant les informations essentielles : un identifiant unique (`ID`), un nom, une icône (sprite), un type (arme, objet utilitaire, etc.), et une référence au prefab Unity correspondant.

L'inventaire lui-même est implémenté sous forme de tableau indexé fixe ou liste dynamique selon les besoins. Ce tableau est maintenu dans une classe centrale (`InventoryManager`) chargée de la gestion des objets possédés, du slot actif et de la communication avec l'interface utilisateur.

3.4.2 Gestion des slots

Le système de gestion des slots constitue l'un des piliers du système d'inventaire actif. Chaque slot représente un emplacement visuel et logique pour une arme ou un objet. Dans notre implémentation, les slots sont gérés via la classe `InventoryManager`, qui assure à la fois la gestion des données et la synchronisation avec l'interface utilisateur.

Les slots sont représentés par une liste d'objets de type `InventorySlot`, chacun correspondant à un élément de l'inventaire. Lorsqu'un joueur change d'arme, le slot actif est mis à jour, et l'état visuel reflète ce changement.

Voici un extrait du code responsable de la gestion des slots :

Listing 3.1 – Initialisation et mise à jour des slots

```
private void InitializeSlots()
{
    slots = new List<InventorySlot>();
    foreach (Transform slotTransform in slotParent)
    {
        InventorySlot slot = slotTransform.GetComponent<InventorySlot>();
        if (slot != null)
        {
            slots.Add(slot);
            slot.ClearSlot();
        }
    }
}
```

Chaque slot est lié à un élément graphique dans l'UI, ce qui permet de mettre à jour dynamiquement l'état visuel (par exemple, surbrillance du slot actif, affichage de l'icône d'arme, etc.).

Lorsqu'un changement de slot actif intervient, la méthode suivante est appelée pour effectuer la mise à jour :

Listing 3.2 – Changement de slot actif

```
public void SetActiveSlot(int index)
{
    if (index < 0 || index >= slots.Count) return;

    activeSlotIndex = index;
    for (int i = 0; i < slots.Count; i++)
    {
        slots[i].SetSelected(i == index);
    }
}
```

Cette méthode permet de gérer proprement la transition visuelle d'un slot à l'autre tout en maintenant la logique interne synchronisée avec l'interface.

3.4.3 Algorithmes de sélection et changement d'armes

Le changement d'arme repose sur une logique centralisée dans le script `InventoryManager.cs`, en interaction étroite avec le système d'entrée utilisateur (Input System) et l'interface utilisateur (UI).

Lorsqu'une action de changement d'arme est déclenchée par le joueur (via la molette ou une touche spécifique), l'inventaire évalue la validité de la nouvelle sélection, met à jour le slot actif, puis applique les changements visuellement et fonctionnellement dans le jeu.

Voici la méthode principale gérant la rotation des armes (par exemple lors d'un scroll) :

Listing 3.3 – Sélection d'une nouvelle arme via index

```
private void ChangeWeapon(int direction)
{
    if (slots.Count == 0) return;

    int nextIndex = activeSlotIndex + direction;

    if (nextIndex < 0) nextIndex = slots.Count - 1;
    else if (nextIndex >= slots.Count) nextIndex = 0;

    SetActiveSlot(nextIndex);
    UpdateWeapon(nextIndex);
}
```

Cette méthode assure une rotation circulaire dans l'inventaire, en utilisant un index relatif. Une fois le nouveau slot sélectionné, l'arme correspondante est chargée et affichée via la méthode `UpdateWeapon` :

Listing 3.4 – Application du changement d'arme

```
private void UpdateWeapon(int index)
{
    WeaponData newWeapon = slots[index].GetWeapon();
    if (newWeapon == null) return;

    equippedWeapon = newWeapon;
    player.SetWeapon(equippedWeapon);
    uiManager.UpdateWeaponDisplay(equippedWeapon);
}
```

Cette approche garantit une séparation claire entre les données, les effets de gameplay et la présentation graphique. Le code vérifie également que le slot sélectionné contient bien une arme avant d'effectuer le changement, évitant ainsi les erreurs d'accès null.

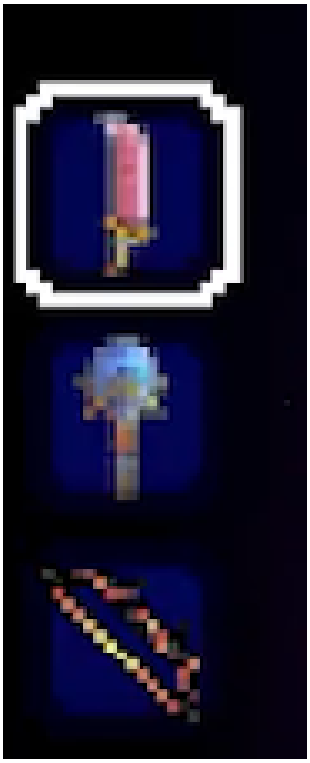

FIGURE 3.6 – Avant scroll

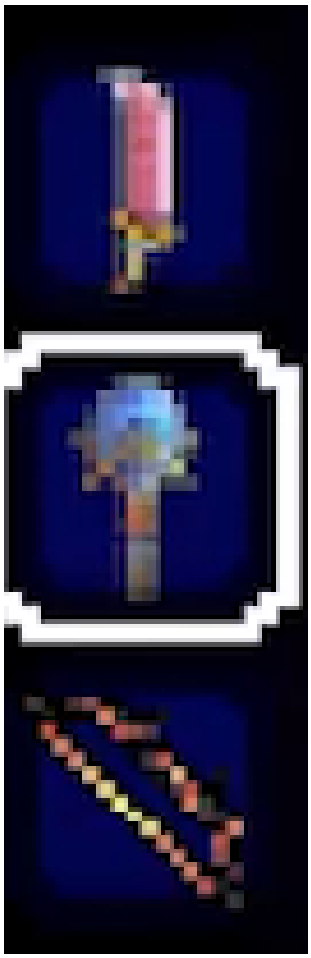

FIGURE 3.7 – Après scroll

3.5 Système UI et effets visuels

L'interface utilisateur joue un rôle fondamental dans l'expérience du joueur. Elle permet une lecture claire de l'état du personnage (santé, arme équipée, inventaire), tout en assurant une immersion visuelle fluide. Ce système se compose principalement de scripts dédiés à l'affichage, au fondu, et à la mise à jour dynamique des éléments.

3.5.1 Gestion du fondu (UIFade)

Le script `UIFade.cs` permet d'effectuer des transitions visuelles douces entre différents états de l'interface, en gérant l'opacité des éléments via leur `CanvasGroup`.

Voici la méthode clé permettant de déclencher un fondu enchaîné :

Listing 3.5 – Fondu d'un élément UI

```
public void Fade(float targetAlpha, float duration)
{
    StartCoroutine(FadeCoroutine(targetAlpha, duration));
}

private IEnumerator FadeCoroutine(float targetAlpha, float duration)
{
    float startAlpha = canvasGroup.alpha;
    float time = 0f;

```

```
11      while (time < duration)
12      {
13          canvasGroup.alpha = Mathf.Lerp(startAlpha, targetAlpha, time /
                duration);
14          time += Time.deltaTime;
15          yield return null;
16      }
17
18      canvasGroup.alpha = targetAlpha;
19  }
```

Cette approche permet un contrôle précis des transitions d'éléments UI (apparition, disparition, surbrillance, etc.) de manière non-bloquante pour le gameplay.

FIGURE 3.8 – Transition fade lors d'un changement de salle

FIGURE 3.9 – Transition finie

3.5.2 Design UX / UI

Le design de l'interface a été pensé pour être fonctionnel, minimaliste et cohérent avec le style visuel du jeu. Les éléments principaux sont regroupés en HUD non-intrusif :

— Slot d'arme active (avec icône dynamique)
— Barre de vie
— Indicateurs de changements contextuels (changement d'arme, interaction)

Tous les éléments UI sont disposés avec des ancrages adaptés aux résolutions d'écran, permettant une adaptation fluide sans altérer la lisibilité.

FIGURE 3.10 – Interface expliquée

3.5.3 Implémentation et optimisation

L'implémentation UI repose sur une hiérarchie claire d'objets dans la scène Unity. La séparation logique entre gestionnaires de données et composants visuels est assurée via des références croisée dans les scripts (ex : `UIManager.cs`).

Voici un extrait clé du script `UIManager.cs`, chargé de mettre à jour l'arme affichée :

Listing 3.6 – Mise à jour de l'arme UI

```
public void UpdateWeaponDisplay(WeaponData weapon)
{
    weaponIcon.sprite = weapon.icon;
    weaponNameText.text = weapon.weaponName;
}
```

Cette méthode est appelée automatiquement par le système d'inventaire à chaque changement d'arme, garantissant une synchronisation visuelle immédiate.

D'un point de vue optimisation, l'interface :

— Utilise un système de pooling pour éviter les instanciations répétées.
— Exploite les événements Unity pour déclencher les updates uniquement lorsque nécessaire.
— Évite toute opération UI dans la boucle `Update()`.

FIGURE 3.11 – Hiérarchie des objets UI

3.6 Système UI et effets visuels

L'interface utilisateur (UI) joue un rôle fondamental dans la clarté et l'accessibilité du jeu. Dans notre projet, l'UI a été conçue de manière à rester épurée, intuitive et cohérente avec l'esthétique générale du jeu. Cette section présente la gestion des éléments visuels à l'écran, ainsi que les effets de transition pour améliorer l'expérience utilisateur.

3.6.1 Gestion du fondu (UIFade)

Le système de fondu est utilisé pour assurer une transition douce entre les scènes (par exemple, lors de la connexion ou du retour au menu principal). Il repose sur une simple animation d'alpha (opacité) appliquée à une image couvrant tout l'écran.

— L'élément principal est une image noire en surcouche.
— Une animation de fondu entre 0 et 1 d'opacité est déclenchée par script via `CanvasGroup.alpha`.
— Le script gérant cet effet est `UIFade.cs`.

Extrait de code tiré du script `UIFade.cs` :

Listing 3.7 – Fondu en entrée/sortie via CanvasGroup

```
public IEnumerator FadeOut(float duration) {
    float time = 0;
    while (time < duration) {
        time += Time.deltaTime;
        canvasGroup.alpha = 1 - (time / duration);
        yield return null;
    }
    canvasGroup.alpha = 0;
    canvasGroup.interactable = false;
}
```

Ce système permet une immersion plus fluide en évitant les coupures brutales à l'écran.

3.6.2 Design UX / UI

L'UX a été pensée pour maximiser la lisibilité et la simplicité :

— Les éléments d'interface (inventaire, barre de sélection d'arme, messages système) sont placés en zones périphériques pour ne pas gêner la visibilité de l'action.
— Une taille de police et un contraste adaptés à une lecture rapide ont été utilisés.
— La cohérence graphique est assurée par des assets visuels homogènes.

3.6.3 Implémentation et optimisation

Du point de vue technique, l'ensemble des composants UI est contenu dans un `Canvas` Unity dédié à l'interface, en mode `Screen Space - Overlay`.

Plusieurs optimisations ont été mises en place :

— Le nombre d'objets dans le canvas est limité pour éviter des recalculs constants de layout.
— Les textes UI sont regroupés sous un seul `TextMeshProUGUI` manager pour éviter les surcharges.
— Le code UI est découplé du code logique via des événements (`UnityEvent`, `C Actions`).

Ce système a permis de maintenir une performance stable même lors de transitions rapides ou de l'ouverture/fermeture de fenêtres UI.

3.7 Système de contrôle utilisateur

Le système de contrôle utilisateur constitue un pilier central de l'interaction dans un jeu vidéo. Dans ce projet, l'ensemble des entrées clavier et manette est géré via le **Unity Input System**, une solution moderne et flexible permettant une meilleure organisation des commandes et une gestion multi-device. Ce système permet au joueur d'interagir avec son personnage, d'utiliser l'inventaire ou encore de naviguer dans les menus.

3.7.1 Unity Input System

Nous avons fait le choix d'utiliser le **nouveau système d'entrée de Unity** pour plusieurs raisons :

— Support natif des manettes, claviers et dispositifs multiples.
— Possibilité de mapper plusieurs actions à une même commande.
— Interface graphique intégrée pour configurer les inputs.

Le système repose sur la création d'un `InputActionAsset`, dans lequel les différentes actions sont définies et regroupées en `Action Maps`.

3.7.2 Définition des actions et maps

Les actions définies sont regroupées dans un fichier d'asset nommé `Controls.inputactions`. On y retrouve notamment :

— **Move** : Déplacement du joueur.
— **ChangeWeapon** : Changement d'arme dans l'inventaire.
— **Interact** : Utilisation ou interaction avec un objet.

Chaque action est associée à une ou plusieurs touches ou boutons. Par exemple, `ChangeWeapon` peut être déclenchée via `E` au clavier ou `RB` sur manette.

3.7.3 Gestion des événements et callbacks

Une fois les actions définies, elles sont associées à des méthodes spécifiques via des événements. Le script `InputManager.cs` centralise cette logique.

Extrait tiré de `InputManager.cs` :

Listing 3.8 – Réception de l'événement de changement d'arme

```
1 private void OnEnable() {
2     controls.Player.ChangeWeapon.performed += ctx => inventory.CycleWeapon
          ();
3 }
```

Cet exemple montre l'écoute de l'événement `ChangeWeapon`, qui appelle la méthode `CycleWeapon()` de l'inventaire. Cela garantit une séparation claire entre logique d'entrée et logique de gameplay.

3.7.4 Exemples d'intégration

Dans le script du joueur, les mouvements sont aussi contrôlés via l'Input System :

Listing 3.9 – Déplacement du joueur via les entrées

```
void Update() {
    Vector2 moveInput = controls.Player.Move.ReadValue<Vector2>();
    Vector3 move = new Vector3(moveInput.x, 0, moveInput.y);
    transform.Translate(move * moveSpeed * Time.deltaTime);
}
```

[Voir la vidéo (au tout début) pour voir les mouvements du joueur.].

L'approche modulaire retenue permet une meilleure maintenabilité, notamment pour l'ajout de nouvelles actions ou la personnalisation du contrôle utilisateur selon les préférences du joueur.

3.8 Autres scripts utiles

Dans cette section, nous présentons plusieurs scripts complémentaires essentiels au fonctionnement du système d'armes et d'inventaire du jeu. Chaque script est détaillé avec son rôle principal et son code source.

3.8.1 ActiveInventory.cs

Ce script gère l'inventaire actif du joueur, permettant de changer d'arme via les raccourcis clavier, d'équiper une arme de départ et d'instancier dynamiquement l'arme sélectionnée.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ActiveInventory : Singleton<ActiveInventory>
{
    private int activeSlotIndexNum = 0;

    private PlayerControls playerControls;

    protected override void Awake() {
        base.Awake();

        playerControls = new PlayerControls();
    }

    private void Start() {
        playerControls.Inventory.Keyboard.performed += ctx =>
            ToggleActiveSlot((int)ctx.ReadValue<float>());
    }

    private void OnEnable() {
        playerControls.Enable();
    }

    public void EquipStartingWeapon() {
        ToggleActiveHighlight(0);
    }

    private void ToggleActiveSlot(int numValue) {
```

```
30          ToggleActiveHighlight(numValue - 1);
31      }
32
33      private void ToggleActiveHighlight(int indexNum) {
34          activeSlotIndexNum = indexNum;
35
36          foreach (Transform inventorySlot in this.transform)
37          {
38              inventorySlot.GetChild(0).gameObject.SetActive(false);
39          }
40
41          this.transform.GetChild(indexNum).GetChild(0).gameObject.SetActive
                (true);
42
43          ChangeActiveWeapon();
44      }
45
46      private void ChangeActiveWeapon() {
47
48          if (ActiveWeapon.Instance.CurrentActiveWeapon != null) {
49              Destroy(ActiveWeapon.Instance.CurrentActiveWeapon.gameObject);
50          }
51
52          Transform childTransform = transform.GetChild(activeSlotIndexNum);
53          InventorySlot inventorySlot = childTransform.
                GetComponentInChildren<InventorySlot>();
54          WeaponInfo weaponInfo = inventorySlot.GetWeaponInfo();
55          GameObject weaponToSpawn = weaponInfo.weaponPrefab;
56
57          if (weaponInfo == null) {
58              ActiveWeapon.Instance.WeaponNull();
59              return;
60          }
61
62
63          GameObject newWeapon = Instantiate(weaponToSpawn, ActiveWeapon.
                Instance.transform);
64
65          //ActiveWeapon.Instance.transform.rotation = Quaternion.Euler(0,
                0, 0);
66          //newWeapon.transform.parent = ActiveWeapon.Instance.transform;
67
68          ActiveWeapon.Instance.NewWeapon(newWeapon.GetComponent<
                MonoBehaviour>());
69      }
70  }
```

3.8.2 Bow.cs

Ce script implémente le comportement de l'arme arc, une arme projectile. Il suit l'interface `IWeapon` pour garantir une gestion uniforme des attaques.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Bow : MonoBehaviour, IWeapon
```

```
{
    [SerializeField] private WeaponInfo weaponInfo;
    [SerializeField] private GameObject arrowPrefab;
    [SerializeField] private Transform arrowSpawnPoint;

    readonly int FIRE_HASH = Animator.StringToHash("Fire");

    private Animator myAnimator;

    private void Awake()
    {
        myAnimator = GetComponent<Animator>();
    }

    public void Attack()
    {
        myAnimator.SetTrigger(FIRE_HASH);
        GameObject newArrow = Instantiate(arrowPrefab, arrowSpawnPoint.
            position, ActiveWeapon.Instance.transform.rotation);
        newArrow.GetComponent<Projectile>().UpdateProjectileRange(
            weaponInfo.weaponRange);
    }

    public WeaponInfo GetWeaponInfo()
    {
        return weaponInfo;
    }
}
```

3.8.3 IWeapon.cs

Interface définissant les méthodes que doivent implémenter toutes les armes.

```
interface IWeapon {
    public void Attack();
    public WeaponInfo GetWeaponInfo();
}
```

3.8.4 InventorySlot.cs

Ce script représente un emplacement d'inventaire qui stocke un objet `WeaponInfo` et permet d'y accéder.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class InventorySlot : MonoBehaviour
{
    [SerializeField] private WeaponInfo weaponInfo;

    public WeaponInfo GetWeaponInfo() {
        return weaponInfo;
    }
}
```

3.8.5 MagicLaser.cs

Ce script gère un projectile laser magique qui grandit progressivement en taille jusqu'à atteindre une portée définie ou rencontrer un obstacle.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MagicLaser : MonoBehaviour
{
    [SerializeField] private float laserGrowTime = 2f;

    private bool isGrowing = true;
    private float laserRange;
    private SpriteRenderer spriteRenderer;
    private CapsuleCollider2D capsuleCollider2D;

    private void Awake() {
        spriteRenderer = GetComponent<SpriteRenderer>();
        capsuleCollider2D = GetComponent<CapsuleCollider2D>();
    }

    private void Start() {
        LaserFaceMouse();
    }

    private void OnTriggerEnter2D(Collider2D other) {
        if (other.gameObject.GetComponent<Indestructible>() && !other.
            isTrigger) {
            isGrowing = false;
        }
    }

    public void UpdateLaserRange(float laserRange) {
        this.laserRange = laserRange;
        StartCoroutine(IncreaseLaserLengthRoutine());
    }

    private IEnumerator IncreaseLaserLengthRoutine() {
        float timePassed = 0f;

        while (spriteRenderer.size.x < laserRange && isGrowing)
        {
            timePassed += Time.deltaTime;
            float linearT = timePassed / laserGrowTime;

            // sprite
            spriteRenderer.size = new Vector2(Mathf.Lerp(1f, laserRange,
                linearT), 1f);

            // collider
            capsuleCollider2D.size = new Vector2(Mathf.Lerp(1f, laserRange
                , linearT), capsuleCollider2D.size.y);
            capsuleCollider2D.offset = new Vector2((Mathf.Lerp(1f,
                laserRange, linearT)) / 2, capsuleCollider2D.offset.y);

            yield return null;
```

```
        }

        StartCoroutine(GetComponent<SpriteFade>().SlowFadeRoutine());
    }

    private void LaserFaceMouse() {
        Vector3 mousePosition = Input.mousePosition;
        mousePosition = Camera.main.ScreenToWorldPoint(mousePosition);
        Vector2 direction = transform.position - mousePosition;
        transform.right = -direction;
    }
}
```

3.8.6 MouseFollow.cs

Script utilitaire qui oriente un objet pour qu'il fasse face à la position de la souris.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MouseFollow : MonoBehaviour
{
    private void Update() {
        FaceMouse();
    }

    private void FaceMouse() {
        Vector3 mousePosition = Input.mousePosition;
        mousePosition = Camera.main.ScreenToWorldPoint(mousePosition);

        Vector2 direction = transform.position - mousePosition;

        transform.right = -direction;
    }
}
```

3.8.7 Staff.cs

Implémentation d'une arme magique utilisant un bâton (staff). Le script suit l'interface `IWeapon` et gère l'animation d'attaque ainsi que la génération d'un projectile laser.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Staff : MonoBehaviour, IWeapon
{
    [SerializeField] private WeaponInfo weaponInfo;
    [SerializeField] private GameObject magicLaser;
    [SerializeField] private Transform magicLaserSpawnPoint;

    private Animator myAnimator;

    readonly int ATTACK_HASH = Animator.StringToHash("Attack");
```

```
14
15      private void Awake() {
16          myAnimator = GetComponent<Animator>();
17      }
18
19      private void Update() {
20          MouseFollowWithOffset();
21      }
22
23
24      public void Attack() {
25          myAnimator.SetTrigger(ATTACK_HASH);
26      }
27
28      public void SpawnStaffProjectileAnimEvent() {
29          GameObject newLaser = Instantiate(magicLaser, magicLaserSpawnPoint
                .position, Quaternion.identity);
30          newLaser.GetComponent<MagicLaser>().UpdateLaserRange(weaponInfo.
                weaponRange);
31      }
32
33      public WeaponInfo GetWeaponInfo()
34      {
35          return weaponInfo;
36      }
37
38      private void MouseFollowWithOffset()
39      {
40          Vector3 mousePos = Input.mousePosition;
41          Vector3 playerScreenPoint = Camera.main.WorldToScreenPoint(
                PlayerController.Instance.transform.position);
42
43          float angle = Mathf.Atan2(mousePos.y, mousePos.x) * Mathf.Rad2Deg;
44
45          if (mousePos.x < playerScreenPoint.x)
46          {
47              ActiveWeapon.Instance.transform.rotation = Quaternion.Euler(0,
                    -180, angle);
48          }
49          else
50          {
51              ActiveWeapon.Instance.transform.rotation = Quaternion.Euler(0,
                    0, angle);
52          }
53      }
54  }
```

3.8.8 WeaponInfo.cs

ScriptableObject contenant les données d'une arme, telles que son prefab, son cooldown, ses dégâts, et sa portée.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  [CreateAssetMenu(menuName = "New Weapon")]
```

```
6  public class WeaponInfo : ScriptableObject
7  {
8      public GameObject weaponPrefab;
9      public float weaponCooldown;
10     public int weaponDamage;
11     public float weaponRange;
12 }
```

Chapitre 4

Tests et validation

4.1 Méthodes de test utilisées

Pour garantir la stabilité, la jouabilité et la fluidité de notre jeu RPG 2D développé sous Unity, une stratégie de tests systématique a été appliquée à chaque étape du développement. L'objectif était de valider à la fois le bon fonctionnement des scripts, la cohérence des comportements en jeu, ainsi que l'expérience utilisateur. Voici les principales méthodes de test utilisées :

- **Tests manuels** : Des tests en conditions réelles ont été réalisés fréquemment pour s'assurer que les différentes fonctionnalités du jeu (combat, déplacement, interaction avec l'interface utilisateur, utilisation des armes, etc.) fonctionnaient correctement.
- **Tests fonctionnels unitaires** : Bien que non automatisés, ces tests consistaient à vérifier individuellement le bon comportement des scripts clés tels que `ActiveInventory`, `Bow`, `Staff`, ou encore `MagicLaser` en utilisant des objets de test dans des scènes spécifiques.
- **Débogage via la console Unity** : L'utilisation de commandes comme `Debug.Log()`, `Debug.DrawRay()`, et `Debug.Break()` a permis d'observer en temps réel les états internes du jeu (par exemple, changement d'arme, détection de collisions, déclenchement d'animations).
- **Tests d'intégration progressive** : À mesure que les nouvelles fonctionnalités étaient intégrées (UI, inventaire, système d'attaque, rotation vers la souris), elles étaient testées en combinaison avec les fonctionnalités existantes afin de détecter d'éventuelles régressions.
- **Tests de l'interface utilisateur (UI)** : Les menus (inventaire, interface de jeu) ont été testés pour garantir leur accessibilité, leur bon affichage, ainsi que leur adaptabilité en fonction des actions du joueur (changement d'arme, attaque, etc.).
- **Utilisation du Profiler Unity** : Cet outil a été utilisé pour identifier les éventuels problèmes de performance tels que les pics d'utilisation CPU ou GPU, les fuites de mémoire, ou les appels coûteux.
- **Tests sur différents matériels** : Bien que le jeu soit destiné à une utilisation locale, des essais ont été réalisés sur plusieurs configurations matérielles afin de s'assurer d'une bonne compatibilité et fluidité.

Ces tests ont permis de construire une base solide et d'assurer une expérience de jeu stable, sans blocage ni comportement incohérent.

4.2 Scénarios testés

Dans le cadre de la validation fonctionnelle du jeu, plusieurs scénarios réalistes ont été définis pour vérifier le bon comportement des principales mécaniques du jeu. Le joueur commence directement dans une salle d'exploration, sans menu d'accueil. Les tests suivants ont été réalisés :

Scénario 1 : Initialisation du jeu

- Lancement direct du jeu dans une salle fermée.
- Apparition immédiate du personnage jouable.
- Chargement correct des éléments de la scène (joueur, armes, interface).

Scénario 2 : Contrôle du joueur

- Déplacement fluide dans toutes les directions via les touches directionnelles.
- Comportement du joueur face aux murs : vérification du système de collision.
- Réactivité aux entrées clavier sans latence visible.

Scénario 3 : Système de combat

- Changement d'arme à l'aide des touches numériques (1 à 3).

— Animation d'attaque déclenchée à chaque action du joueur.
— Appel correct des scripts d'instanciation de projectiles (flèches, lasers).
— Détection de collision entre les projectiles et les éléments du décor.
— Rotation dynamique de l'arme selon la position de la souris.

Scénario 4 : Interface d'inventaire

— Mise en surbrillance de l'arme active dans l'UI.
— Affichage correct des slots disponibles dans l'inventaire.
— Test de basculement rapide entre plusieurs armes.

Scénario 5 : Effets visuels et ambiance

— Animation de croissance du rayon laser lors de son tir.
— Rotation fluide du rayon laser vers la souris.
— Activation du système de fade à la fin du tir laser.

Scénario 6 : Robustesse et stabilité

— Changement rapide d'armes en boucle sans crash.
— Lancement de multiples attaques consécutives sans bug graphique.
— Comportement du système en cas de slot vide (grâce à la gestion d'arme nulle).

Ces tests ont été réalisés à la fois dans l'environnement de développement Unity et à partir d'une build autonome du jeu, garantissant une cohérence fonctionnelle entre les deux versions.

4.3 Résultats

Les différents scénarios de test décrits précédemment ont permis de valider la majorité des fonctionnalités attendues. Voici un résumé des résultats observés :

Comportement général

Le jeu démarre correctement et le joueur est immédiatement placé dans une salle jouable. L'environnement initial est stable et aucun bug bloquant n'a été détecté lors du lancement.

Déplacements et contrôles

Les déplacements du joueur sont fluides et précis. Les collisions avec les murs fonctionnent correctement, empêchant tout dépassement non autorisé des limites de la carte. Les actions répondent instantanément aux commandes clavier et souris.

Système d'attaque

Le changement d'arme à la volée est parfaitement opérationnel. Chaque type d'arme (arc, bâton magique) déclenche l'attaque attendue, avec les bons effets visuels et collisions.
— Les flèches atteignent leur cible et se détruisent proprement.
— Le laser magique grandit en direction de la souris, puis disparaît via une animation de fondu.

UI d'inventaire

L'interface d'inventaire fonctionne comme prévu. La mise en surbrillance de l'arme sélectionnée est bien visible, et le changement de slot entraîne l'activation du bon objet.

Robustesse

Des tests intensifs de changement d'armes, de tirs en boucle, et d'interaction entre systèmes ont été menés :

— Aucun crash observé.

— Le système gère les cas d'absence d'arme dans un slot sans erreur.

Performance

Le jeu tourne à une fréquence d'images constante (60 FPS sur machine de test), avec une consommation mémoire stable. Aucun ralentissement ni fuite mémoire n'a été constaté pendant les sessions de test prolongées.

Les résultats globaux indiquent que le jeu est stable, réactif, et que les mécaniques principales sont fonctionnelles dans l'état actuel du développement.

4.4 Correctifs apportés

Au cours du développement, plusieurs anomalies et comportements indésirables ont été identifiés à l'aide des tests manuels. Divers correctifs ont été implémentés pour garantir une expérience de jeu fluide et sans erreur. Cette section détaille les principaux problèmes rencontrés ainsi que les solutions apportées.

1. Activation incorrecte des slots d'inventaire

Problème : Lors de l'utilisation des touches de sélection rapide, certains slots d'inventaire n'activaient pas correctement l'arme associée ou laissaient plusieurs surbrillances actives.
Correctif : Le script `ActiveInventory.cs` a été modifié pour désactiver toutes les surbrillances avant d'activer uniquement celle du slot sélectionné. La méthode `ToggleActiveHighlight()` a été recentrée sur une logique de nettoyage complète de l'état visuel.

2. Erreur lors de l'attaque sans arme

Problème : Si aucun objet n'était associé au slot actif, le jeu tentait de lancer une attaque et provoquait une erreur de référence null.
Correctif : Un test conditionnel a été ajouté dans la méthode `ChangeActiveWeapon()` pour vérifier la validité du `WeaponInfo` avant toute instanciation. Le système déclenche désormais une fonction de gestion de nullité via `ActiveWeapon.Instance.WeaponNull()`.

3. Mauvais alignement du laser magique

Problème : Le rayon laser du bâton ne se dirigeait pas précisément vers la souris, créant une incohérence visuelle.
Correctif : La méthode `LaserFaceMouse()` dans `MagicLaser.cs` a été ajustée pour recalculer correctement le vecteur directionnel à partir des coordonnées écran du curseur, assurant une orientation juste du sprite et du collider.

4. Problèmes de persistance du laser après collision

Problème : Le laser magique ne disparaissait pas correctement après avoir touché un objet `Indestructible`.
Correctif : Le booléen `isGrowing` est désormais modifié à l'impact et une coroutine de disparition lente est appelée via le composant `SpriteFade`, garantissant une fin d'effet propre.

5. Instanciation excessive d'armes

Problème : Chaque changement de slot entraînait une instanciation sans destruction de l'arme précédente, entraînant une accumulation invisible.
Correctif : Une vérification et destruction systématique de l'objet arme en cours a été ajoutée dans la méthode `ChangeActiveWeapon()`, grâce à :

```
if (ActiveWeapon.Instance.CurrentActiveWeapon != null) {
    Destroy(ActiveWeapon.Instance.CurrentActiveWeapon.gameObject);
}
```

Ces correctifs ont amélioré significativement la stabilité, la robustesse et la jouabilité du prototype, tout en préparant le terrain pour les futures extensions fonctionnelles.

Chapitre 5

Optimisations et perspectives

Ce chapitre présente les principales limites rencontrées durant le développement, les optimisations réalisées afin d'y remédier, ainsi que les axes d'amélioration et les fonctionnalités futures envisagées pour enrichir le jeu.

5.1 Limites rencontrées

1. Gestion rudimentaire de l'inventaire

La structure de l'inventaire, bien que fonctionnelle, repose sur une hiérarchie d'objets dans la scène. Cela implique une dépendance forte avec l'UI, rendant complexe son évolution ou sa réutilisation dans d'autres scènes.

2. Absence de sauvegarde

Aucune mécanique de sauvegarde n'a été implémentée. Par conséquent, l'état du jeu (équipement, position du joueur, santé, etc.) est perdu entre deux sessions.

3. Contrôle d'entrée rigide

Le système d'entrée repose sur un mapping fixe via `PlayerControls`, ce qui ne permet pas de personnalisation côté joueur. De plus, certaines touches (sélections numériques) n'étaient pas toujours correctement captées sur tous les claviers.

4. Manque de feedback utilisateur

L'absence de HUD, de menus, ou d'indicateurs visuels (comme les points de vie, les effets d'impact ou de cooldown) nuit à la clarté de l'expérience utilisateur, surtout pour un joueur découvrant le jeu sans instructions.

5.2 Optimisations réalisées

1. Simplification du système d'armes

Une interface `IWeapon` a été définie pour normaliser les comportements des armes (`Attack()`, `GetWeaponInfo()`). Cela a permis d'uniformiser les appels et de réduire la complexité lors des changements d'arme.

2. Nettoyage et instanciation contrôlée

Des vérifications systématiques ont été ajoutées pour éviter les instanciations inutiles ou multiples des objets d'arme. Ce contrôle améliore les performances mémoire et réduit le risque de bugs visuels ou logiques.

3. Utilisation de coroutines pour les effets visuels

Les effets progressifs (comme le laser magique) ont été implémentés via des coroutines asynchrones, ce qui permet une animation fluide sans blocage du fil principal de jeu.

4. Architecture modulaire

Le découpage en composants spécialisés (projectiles, inventaire, armes) facilite la lecture, la maintenance et la réutilisation du code. Cette approche modulaire rend également plus aisée l'ajout de futures fonctionnalités.

5.3 Fonctionnalités futures envisagées

1. Ajout d'un système de sauvegarde

Permettre au joueur de sauvegarder sa progression (armes équipées, niveau atteint, état du joueur). Cette fonctionnalité pourrait reposer sur un système de fichiers JSON ou le PlayerPrefs de Unity.

2. Affichage d'un HUD contextuel

Un affichage minimaliste à l'écran (état du joueur, arme active, cooldowns) apporterait une meilleure compréhension de l'état du jeu, surtout pour les nouveaux joueurs.

3. Ajout de compétences spéciales

Développement de compétences actives ou passives propres à chaque arme (ralentissement, explosion de zone, vol de vie) qui enrichiraient le gameplay et la stratégie.

4. Système d'ennemis et de progression

L'intégration d'ennemis, d'obstacles et d'un système de progression (XP, niveaux, loot) donnerait plus de profondeur à l'expérience RPG, pour sortir du cadre purement technique et démonstratif.

5. Modularité du système d'inventaire

Refactorisation du système d'inventaire en le rendant indépendant de l'UI Unity, via un modèle orienté données (inventory model + inventory view), pour un meilleur découplage et une plus grande flexibilité.

Ce projet, bien qu'abouti dans son état actuel, constitue une base solide pour de multiples évolutions techniques et créatives. Les perspectives ouvertes permettent d'envisager un développement plus poussé du jeu dans un cadre futur académique ou personnel.

Chapitre 6

Conclusion

Le projet présenté dans ce rapport visait à concevoir et implémenter un jeu vidéo 2D de type RPG à l'aide du moteur Unity, en mettant l'accent sur la maîtrise des aspects fondamentaux du développement de jeu : gestion des entrées, systèmes d'inventaire, mécanique d'attaque, architecture modulaire du code, et animation.

Au fil du développement, de nombreuses compétences techniques ont été mobilisées, allant de la manipulation de l'environnement Unity à la structuration du code en C#, en passant par l'implémentation de systèmes interactifs complexes. Le projet s'est construit autour d'une interface simple, où le joueur contrôle un personnage armé, capable de manier différentes armes, chaque arme apportant un gameplay distinct.

Les objectifs initiaux ont été atteints :
- Une structure d'inventaire interactive permet la sélection dynamique des armes.
- Chaque arme est associée à une logique spécifique d'attaque, instanciée dynamiquement.
- Un système d'interface abstraite (`IWeapon`) assure la cohérence de l'architecture.
- Des effets visuels et animations accompagnent les actions de combat.

Au-delà de la simple exécution, ce projet a permis de découvrir les défis pratiques du développement de jeux :
- Maintenir un code clair, modulaire et évolutif.
- Comprendre les limites de performance d'un moteur comme Unity.
- Gérer la synchronisation entre logique de jeu et rendu graphique.
- Prévoir les évolutions futures dès la conception initiale.

Ce travail constitue donc non seulement une démonstration de compétences acquises, mais aussi un socle pertinent pour un développement ultérieur. Que ce soit dans un cadre académique ou personnel, ce prototype pourra être enrichi par des éléments de gameplay, une narration, ou encore une architecture plus poussée (états de jeu, IA, etc.).

Enfin, ce projet a été l'occasion de développer un regard critique sur ses propres pratiques de programmation, de planification et de test, dans une démarche à la fois technique et créative.

Chapitre A

Annexes

A.1 Assets utilisés

Ce projet utilise plusieurs ressources graphiques (sprites) provenant de plateformes telles que l'Unity Asset Store et Itch.io. Voici une liste non exhaustive des assets utilisés, classés par catégories :

- **Sprites du personnage** :
  `https://assetstore.unity.com/packages/2d/characters/warrior-free-asset-195707`
- **Sprites de la carte (tileset)** :
  `https://lukepolice.itch.io/pixelariumgrasslands`
- **Sprites du Boss** :
  `https://assetstore.unity.com/packages/2d/characters/bringer-of-death-free-195719`
- **Sprite de l'arc** :
  `https://assetstore.unity.com/packages/2d/environments/free-pixel-gear-113561`
- **Sprites des arbres (animés)** :
  `https://assetstore.unity.com/packages/2d/environments/pixel-art-spruce-tree-pack-animated`
- **Autres assets potentiellement oubliés** :
  `https://assetsstore.unity.com`

Tous les assets utilisés sont sous des licences libres ou gratuites compatibles avec les projets étudiants et non commerciaux.