Collège Lasalle Montréal

Projet Final

Pour le cours

420-J13-AS C9_STRUCTURE DE DONN_ENLIGNE

Thème :

Conception et développement d’une application jeu vidéo

Année : 2024-2025

Présenté par

AZIZCHERIF

Sous la direction de

ELISA SHAEFFER

Remerciements

Mes remerciements au corps professoral et administratif de Collège Lasalle de Montréal qui déploient de grands efforts pour nous assurer une formation très actualisée.

Je remercie également les membres du jury qui ont accepté d'évaluer mon travail.

J'exprime enfin ma profonde gratitude à tous ceux qui ont participé de près ou de loin au bon déroulement de ce travail.

Table de matières

Table des Figures

Introduction Générale

Le développement de jeux vidéo est un domaine multidisciplinaire en constante évolution, qui combine programmation, design graphique, intelligence artificielle, conception sonore et narration interactive. Il ne s'agit plus simplement de créer un divertissement, mais bien de bâtir une expérience immersive qui mobilise autant les compétences techniques que créatives. À travers les décennies, les jeux vidéo sont passés de simples pixels en mouvement à des mondes riches, dynamiques et connectés, grâce aux avancées technologiques et à l'essor des moteurs de jeu comme Unity, Unreal Engine ou Godot.

Dans un projet de développement de jeu, chaque élément est crucial. Le gameplay doit être fluide, les mécaniques bien définies, et l'interface intuitive. Pour y parvenir, les développeurs s'appuient sur des algorithmes efficaces et des structures de données adaptées à chaque fonctionnalité : gestion des collisions, pathfinding, inventaire, interactions entre objets, etc. Le développement d'un jeu est donc un excellent moyen d'appliquer des concepts informatiques concrets dans un environnement stimulant.

Au niveau pédagogique, le jeu vidéo représente un terrain d'apprentissage idéal pour les étudiants en programmation. Il permet de mettre en pratique des notions telles que la programmation orientée objet, les structures de données, la logique conditionnelle, et l'optimisation de code. Ce projet de fin de session s'inscrit dans cette logique d'apprentissage appliqué, en mettant en œuvre des niveaux progressifs de complexité afin de construire une base solide pour les futures initiatives en développement de jeux.

Chapitre1
Cadre général du projet

Introduction

Ce chapitre a pour objectif de situer le projet *PowerGame* dans son contexte général, en identifiant la problématique qui a inspiré sa création, en décrivant brièvement le concept du jeu, et en exposant les objectifs pédagogiques et techniques visés. Le projet s'inscrit dans le cadre d'un apprentissage progressif des structures de données et de leur intégration concrète dans un environnement de jeu interactif. En combinant les aspects visuels, interactifs et algorithmiques, le jeu constitue un excellent support pour illustrer des concepts tels que les listes, les listes chaînées, les arbres et les graphes dans un contexte motivant et stimulant.

Présentation du cadre de travail

Dans le cadre de la préparation d'un projet de fin de session pour le cours 420-J13-AS, dispensé au Collège Lasalle, nous avons été amenés à concevoir une application de type jeu vidéo appelée *PowerGame*. Ce jeu est développé sous Unity avec le langage C#, et vise à mettre en œuvre plusieurs structures de données réparties sur trois niveaux de difficulté croissante. Le jeu plonge le joueur dans un univers sombre et stratégique, où il doit naviguer entre différentes zones, collecter des objets, gérer ses ressources, et survivre face à des ennemis. À travers ce projet, l'objectif est non seulement de produire un prototype jouable, mais aussi de documenter toutes les étapes de développement, les choix techniques et les algorithmes intégrés.

Etude de l’existant

Tout développeur de jeu doit s’inspirer et s’informer sur les projets similaires existants afin de mieux concevoir son propre produit. L’étude de l’existant permet d’identifier les mécaniques de jeu populaires, les attentes des joueurs, ainsi que les bonnes pratiques en matière de conception. Dans notre cas, *PowerGame* s’inspire de jeux indépendants en 2D combinant survie, gestion de ressources et exploration, comme *Don’t Starve* ou *Darkwood*, tout en conservant une logique de gameplay simplifiée. Cette analyse du marché vidéoludique nous a permis de cibler les éléments essentiels à intégrer dans notre jeu, comme une interface intuitive, une gestion dynamique des objets et des transitions fluides entre zones. Elle nous a aussi permis d’éviter certaines erreurs courantes, telles que la surcharge de fonctionnalités inutiles ou une interface utilisateur peu claire. L’étude comparative a donc constitué une base solide pour construire une expérience de jeu cohérente et pédagogique, alignée avec les objectifs du cours.

Conclusion

Dans ce chapitre, nous avons commencé par la présentation du cadre général du projet, puis nous avons penché sur les détails qui constituent les solutions concurrentes sur le jeu de base afin de décortiquer leurs points faibles en vu de les éviter. Par la suite, nous avons exposé la que nous allons adopté.

Chapitre2
Analyse des besoins

1. Introduction

La réussite de tout projet dépend de la clarté des besoins. Dans le cas du projet PowerGame, l'étape de spécification
des besoins permet de définir précisément les fonctionnalités à intégrer dans le jeu, qu'elles soient liées à la
jouabilité, à l'interaction avec l'environnement ou à la gestion des ennemis et des objets. Ce chapitre présente
les besoins fonctionnels et non fonctionnels du projet, en identifiant les acteurs principaux (joueur, ennemi, système)
et en explicitant les différents cas d'usage du jeu.

2. **Analyse des besoins**

Dans cette section, nous allons identifier les acteurs ainsi que les besoins fonctionnels et non fonctionnels du jeu PowerGame.

2.1. Identification des acteurs

Le projet PowerGame fait intervenir les acteurs suivants :

• Le joueur : acteur principal qui se déplace, collecte des objets, évite ou affronte les ennemis, et tente de survivre dans un environnement sombre.

• L'ennemi : entité contrôlée par le système, effectue des déplacements automatiques selon un graphe, et interagit avec le joueur en cas de détection.

• Le système de jeu : moteur d'exécution qui gère les mécaniques de déplacement, collisions, état des objets, UI et logique de victoire/défaite.

2.2. Les besoins fonctionnels

Les besoins fonctionnels correspondent aux mécanismes de jeu visibles et interactifs :

• Déplacement du joueur : contrôle directionnel via le clavier (les flèches), mouvement fluide et géré par Rigidbody2D.

• Collecte d'objets : les objets ramassables (comme Battery, Health) doivent pouvoir être activés ou désactivés via une liste dynamique.

• Activation de la lampe : appui sur le clic gauche de la souris pour activer la lumière, avec gestion de la batterie.

• Interaction avec RoomService : le joueur peut appeler un service ou donner une pièce via une touche spécifique.
• Patrouille ennemie : les ennemis se déplacent selon un graphe de nœuds générés automatiquement.
• Système de victoire/défaite : déclenché selon des événements (santé nulle, arrivée au point final).
• Interface utilisateur : affichage dynamique de la santé, de la batterie et de l'argent collecté.

2.3.Besoins non fonctionnels

3. Les besoins non fonctionnels définissent les exigences liées à l'expérience utilisateur :
   • L'ergonomie : interface claire avec des icônes simples, textes lisibles, interaction intuitive à la souris et au clavier.
   • La jouabilité : les contrôles doivent être réactifs et les mécaniques de jeu cohérentes avec le style survival/horror.
   • La compatibilité : le jeu doit être jouable sur un PC avec Windows 10, sans lags majeurs.
   • La stabilité : le système ne doit pas planter même si un objet est manquant ou mal référencé.
   • La modularité : possibilité d'ajouter facilement de nouveaux objets, ennemis ou zones sans casser l'ensemble du projet.

2.4 Spécification des besoins

Pour identifier les caractéristiques de notre application à développer, nous avons choisi d'utiliser le langage de modélisation unifié UML, c'est l'acronyme anglais pour « Unified Modeling Language ». On le traduit par « Langage de modélisation unifié ». La notation UML est un **langage visuel** constitué d'un ensemble de schémas, appelés des **diagrammes**, qui donnent chacun une vision différente du projet à traiter. UML nous fournit donc des diagrammes pour **représenter** le logiciel à développer : son fonctionnement, sa mise en route, les actions susceptibles d'être effectuées par le logiciel, etc.

Le langage UML ne préconise aucune démarche, ce n'est donc pas une méthode. Chacun est libre d'utiliser les types de diagramme qu'il souhaite, dans l'ordre qu'il veut.

Chapitre3
Etude conceptuelle

1. Introduction

Dans ce chapitre, nous allons présenter et identifier toutes les fonctionnalités de notre application jeu vidéo pour chaque type d'utilisateur : Administrateur, joueur et ennemies. Nous allons s'intéresser à la conception des cas d'utilisation que nous venons d'analyser dans le chapitre précédent en utilisant un langage de modélisation spécifique UML.

2. Spécification fonctionnelle

Les spécifications fonctionnelles ont pour objectif de décrire précisément l'ensemble des fonctions d'un logiciel ou d'une application. La spécification d'une fonction va donc décrire en détails les services qu'elle va fournir à l'application ou à l'utilisateur.

Nous allons traduire dans cette section les fonctionnalités offertes par notre site à travers un diagramme de cas d'utilisations d'UML comme vue d'utilisateur, la description textuelle et en vue comportementale on introduira les diagrammes de séquence.

2.1. Diagramme de cas d'utilisation détaillé

Le diagramme de cas d'utilisation permet de décrire l'interaction entre le système et les acteurs. Ce moyen sert à identifier les besoins et à représenter les fonctionnalités du jeu selon la perspective de chaque utilisateur (joueur, administrateur, ennemi). La figure ci-dessus offre une vue globale du comportement fonctionnel de PowerGame et de la manière dont les fonctionnalités sont réparties entre les différents acteurs.

Plus précisément, le joueur a pour rôle de ramasser des objets interactifs comme des batteries ou des trousses de soin, de se déplacer dans les différentes zones du jeu, de vaincre les ennemis, et de récupérer des clés permettant d'ouvrir des passages ou des coffres. Toutes ces actions sont rendues possibles via une interaction avec le système, notamment à l'aide des scripts comme `CollectibleManager`, qui gère l'état actif/inactif des objets affichés dans le jeu.

Les ennemis, quant à eux, effectuent une patrouille dynamique sur un graphe défini et tentent d'empêcher le joueur de progresser en le repérant ou en l'attaquant. Ces comportements sont automatisés via des coroutines de type `PatrolRoutine()` et une structure de graphe mise en place avec `PatrolNode`.

Enfin, l'administrateur (ou développeur) a accès à l'interface Unity pour gérer les objets, éditer les scènes, configurer les scripts et surveiller les interactions via la console. Il peut activer ou désactiver des objets, tester les patrouilles ennemies ou vérifier le bon fonctionnement des interfaces utilisateur.

Ainsi, PowerGame offre une structure d'interaction claire entre les entités du système, représentée à travers ses cas d'utilisation détaillés.

- **Diagramme de cas des ennemies**

Avant de devenir client, un internaute ne possède que la possibilité de consulter le catalogue des produits disponibles dans le stock du fournisseur et la possibilité de s'inscrire pour devenir client sur notre site web.

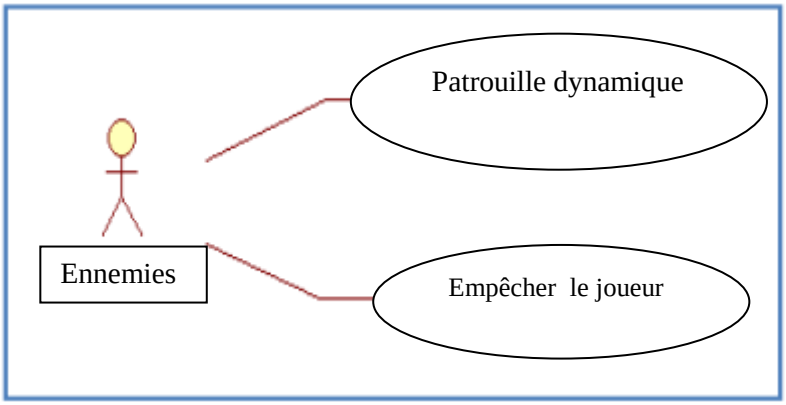


Figure 1 : Diagramme de cas d'utilisation des ennemies

- **Diagramme de cas d'un joueur**

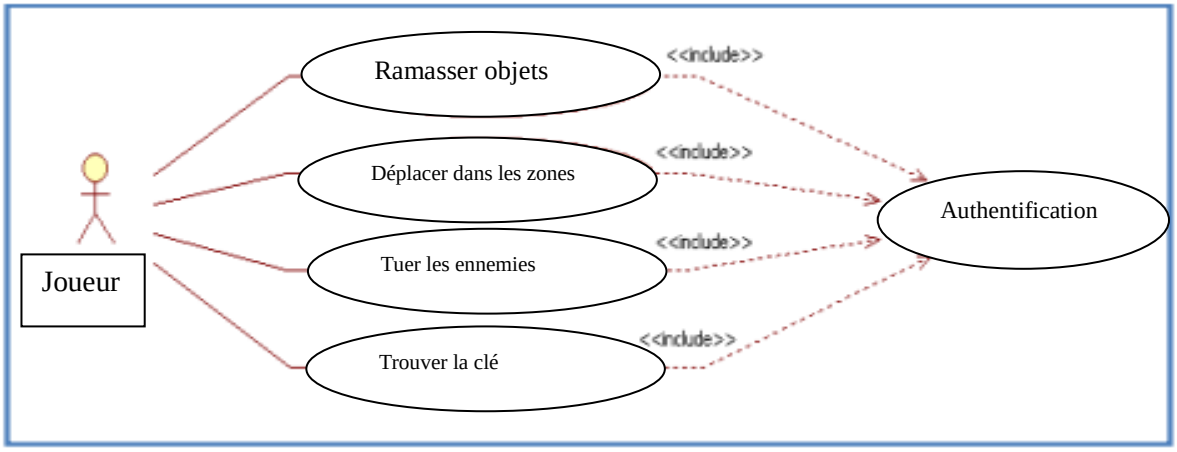


Figure 2: Diagramme de cas d'utilisation détaillé (joueur)

Après l'inscription, le visiteur devient client. Il est donc apte de continuer toute une procédure d'achat en ligne sur notre site (voir figure 8).

- **Diagramme de cas pour admin**

Le terme webmaster de site web désigne communément celui qui est chargé d'un site web. Il gère toute la mise en place technique et parfois la mission éditoriale, il doit gérer au jour le jour la technique et mettre à jour le contenu du site web.

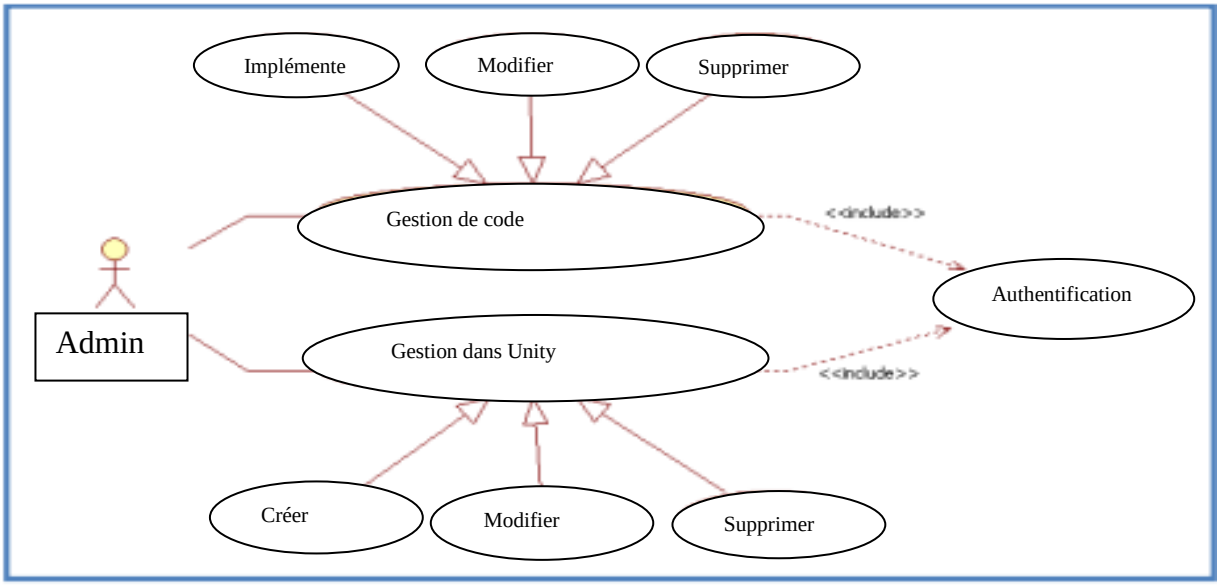


Figure 3: Diagramme de cas d'utilisation détaillé (admin)

3. Diagrammes et Fonctionnalités

3.1. Fonctionnalité du système de jeu

Ce diagramme de séquence illustre l'interaction entre plusieurs composants du projet **PowerGame** lors de l'exécution des fonctions de base : déplacement du joueur, gestion des objets ramassables et comportement des ennemis via un graphe de patrouille.

Le processus commence par l'appel de la méthode `Start()` depuis la classe `PlayerMovement`, qui déclenche l'initialisation du système via `CollectibleManager`. Ensuite, dans la méthode `Update()`, une vérification est effectuée à chaque frame pour détecter si le joueur appuie sur une touche spécifique. Si c'est le cas, la méthode `ShowCollectibles()` est invoquée, ce qui permet d'afficher ou de masquer dynamiquement les objets ramassables (comme les batteries ou les trousses de soins).

Parallèlement, la méthode `InitializePatrolGraph()` est appelée pour définir les points de patrouille des ennemis via la structure `PatrolGraph`. Cela conduit à l'exécution d'une coroutine `PatrolRoutine()` dans une boucle (`loop`), qui permet un déplacement automatique et fluide des ennemis à travers les différents nœuds du graphe, tant que l'ennemi n'est pas déclaré comme "mort" (`[not dead]`). Le composant `Transform` est aussi sollicité pour mettre à jour dynamiquement la position des ennemis dans la scène.

Ce diagramme permet ainsi de visualiser comment les différents éléments du jeu collaborent en temps réel pour gérer les déplacements, l'IA des ennemis et l'interaction avec les objets.

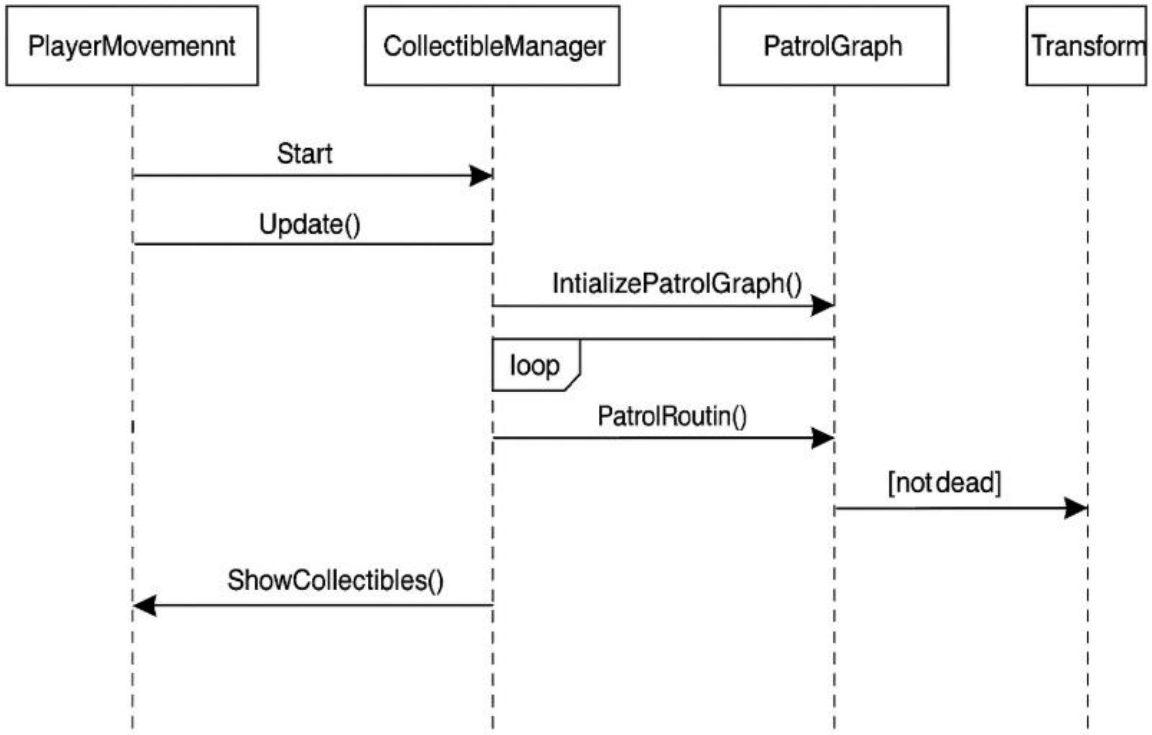


Figure 4 : Diagramme de séquence

Diagramme de classe

Ce diagramme de classe représente l'architecture objet principale du jeu *PowerGame*, en illustrant les relations et dépendances entre les différentes entités du système. La classe centrale est **Player**, représentant le joueur. Elle contient trois attributs : `health`, `money`, et `batteryLevel`, qui stockent respectivement les points de vie, l'argent accumulé, et le niveau de batterie du joueur. Elle expose également quatre méthodes : `UseTorch()`, `SummonStrive()`, `DisplayStats()`, et `NavigateZone()`, qui représentent différentes actions que le joueur peut effectuer pendant le jeu.

Le **Player** est associé à **Enemy** par une relation d'association multiple (0..*), indiquant qu'un joueur peut avoir plusieurs ennemis qui l'entourent. Ces ennemis utilisent la méthode `Patrol()` pour se déplacer selon une logique de patrouille.

Les textes **HealthText** et **MoneyText** sont des composants visuels liés au joueur, affichant respectivement la santé et l'argent sur l'interface utilisateur. Ils sont tous deux en association directe avec la classe **Player**.

La classe **ZoneGraphNode** représente un point de passage dans la carte du jeu et contient une liste de voisins (autres nœuds du graphe), modélisant une structure de navigation. Elle inclut la méthode `SetCurrent()` pour définir la position actuelle d'un ennemi ou du joueur. Cette classe est liée à **Enemy**, indiquant que chaque ennemi se déplace à travers ces nœuds.

RoomService et **BatteryMask** représentent des composants de la scène. `RoomService` active certains éléments dans la pièce courante, tandis que **BatteryMask** affiche un visuel de la batterie.

Ce diagramme montre une architecture bien structurée, orientée vers l'interaction en temps réel entre le joueur, les ennemis, et l'environnement du jeu.

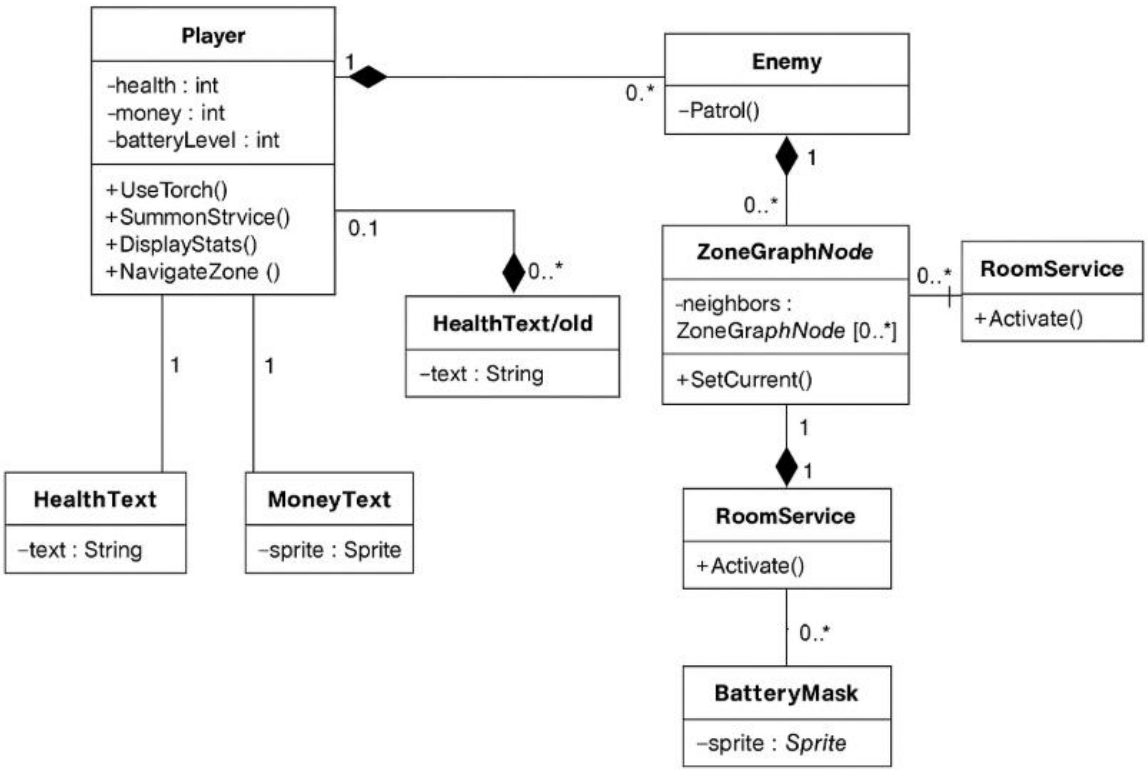


Figure 5:1 Diagramme de classe

1. Conclusion

Dans ce chapitre, nous avons expliqué le mode de fonctionnement de notre application jeu vidéo , en présentant les diagrammes correspondants.

Chapitre4 :
Réalisation

1. Environnement de travail

Dans ce paragraphe, nous allons présenter l'environnement matériel et logiciel de développement de l'application jeu vidéo que nous avons utilisé.

1.1. Environnement matériel

Pour développer l'application, nous avons utilisé comme environnement matériel un ordinateur portable qui possèdent les caractéristiques suivantes :

- Un Processeur Intel(R) Core (TM)i7-6700HQ CPU @ 2.60 GHz 2.59 GHz
- Type du Système : SE :64 bits, Processeur x64
- Une mémoire vive 8 Go
- Un système d'exploitation Microsoft Windows 11.

1.2. Environnement logiciel

Afin de garantir le succès de la réalisation de toute application, il faut choisir l'environnement de travail adéquat comportant le maximum de caractéristiques qui répondent aux besoins du programmeur avec le minium de difficultés. Pour implémenter notre application, nous avons choisis de travailler avec les outils suivants :

❖ **Visual Studio Code**

Visual studio est l'environnement de développement intégré (IDE) utilisé tout au long du projet *PowerGame*. Développé par Microsoft, il permet d'écrire, compiler et déboguer efficacement du code C#, qui est le langage principal utilisé avec le moteur de jeu Unity

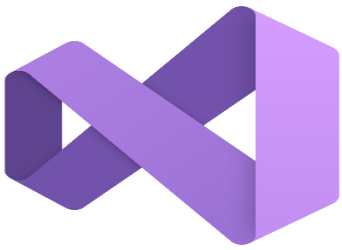


Figure 6 : Visual studio code

❖ **Unity**

Unity est le moteur de jeu utilisé pour le développement du projet *PowerGame*. Il s'agit d'une plateforme multiplateforme puissante et largement utilisée dans l'industrie du jeu vidéo pour créer des jeux en 2D, 3D, en réalité virtuelle et en réalité augmentée. Unity offre un

environnement de travail visuel intuitif, combiné à un éditeur de scènes interactif et un système de scripts basé sur le langage C#. Dans notre projet, Unity a permis de concevoir l'ensemble de l'univers de jeu : déplacement du joueur, interface utilisateur, objets interactifs, intelligence artificielle des ennemis, et transitions entre les zones.

Figure 7: Unity

1.3. Langages utilisés

- **C#**

C# (C-Sharp) est le langage de programmation principal utilisé dans le projet *PowerGame*. Développé par Microsoft, C# est un langage moderne, orienté objet, qui combine la puissance de langages comme C++ avec la simplicité de Java. Il est parfaitement intégré dans le moteur Unity, ce qui en fait un choix naturel pour développer des jeux interactifs

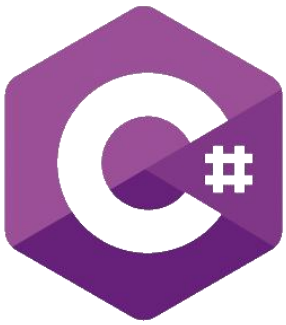
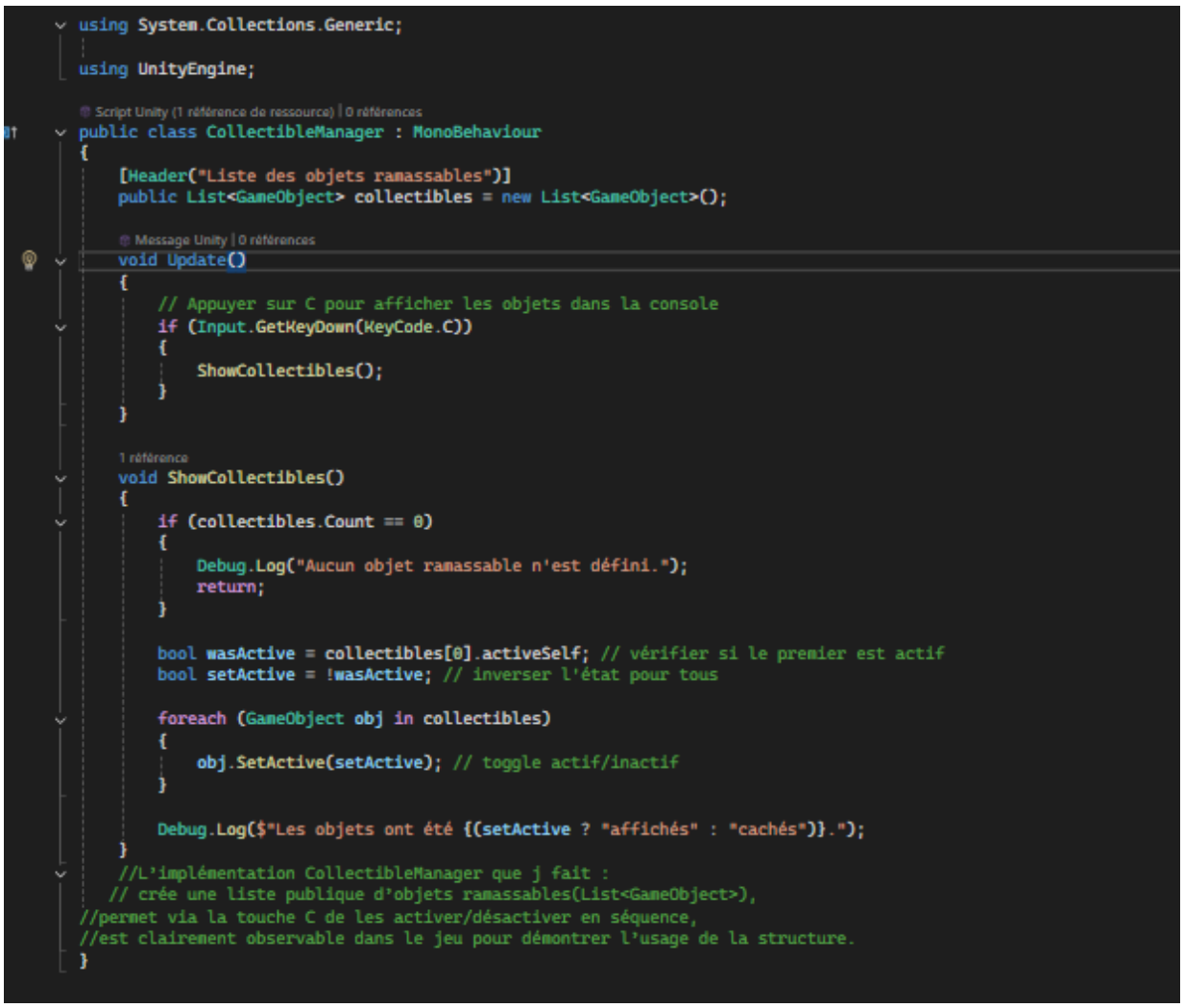


Figure 8: C#

2. Etapes de développement et Interfaces graphiques

2.1. CollectibleManager.cs



```
using System.Collections.Generic;

using UnityEngine;

Script Unity (1 référence de ressource) | 0 références
public class CollectibleManager : MonoBehaviour
{
    [Header("Liste des objets ramassables")]
    public List<GameObject> collectibles = new List<GameObject>();

    Message Unity | 0 références
    void Update()
    {
        // Appuyer sur C pour afficher les objets dans la console
        if (Input.GetKeyDown(KeyCode.C))
        {
            ShowCollectibles();
        }
    }

    1 référence
    void ShowCollectibles()
    {
        if (collectibles.Count == 0)
        {
            Debug.Log("Aucun objet ramassable n'est défini.");
            return;
        }

        bool wasActive = collectibles[0].activeSelf; // vérifier si le premier est actif
        bool setActive = !wasActive; // inverser l'état pour tous

        foreach (GameObject obj in collectibles)
        {
            obj.SetActive(setActive); // toggle actif/inactif
        }

        Debug.Log($"Les objets ont été {(setActive ? "affichés" : "cachés")}.");
    }
    //L'implémentation CollectibleManager que j fait :
   // crée une liste publique d'objets ramassables(List<GameObject>),
//permet via la touche C de les activer/désactiver en séquence,
//est clairement observable dans le jeu pour démontrer l'usage de la structure.
}
```

Figure 9 : ***processus de Liste d'objets ramassables***

Ce script permet de gérer dynamiquement l'affichage ou la disparition d'objets ramassables dans la scène.

Il utilise une structure de données de type liste (List<GameObject>), qui permet d'y stocker tous les objets

de type "collectibles" présents dans le jeu. Le script vérifie si une touche (ici la touche C) est pressée.

Si c'est le cas, il appelle la méthode ShowCollectibles().

Dans cette méthode, s'il n'y a aucun objet dans la liste, un message est affiché dans la console

et le processus
s’arrête. Sinon, le script vérifie l’état actif (activeSelf) du premier objet de la liste. Cela permet de déterminer
s’ils sont affichés ou non. Il inverse ensuite cet état pour tous les objets de la liste grâce à une boucle foreach.
L’appel à SetActive() rend l’objet visible ou invisible dans la scène.

Cette structure est un exemple parfait d’utilisation de données dynamiques dans un contexte de jeu. Elle permet au
joueur ou au développeur de tester ou activer un mécanisme de collecte d’objets, et pourrait facilement être étendue
pour intégrer une logique d’inventaire. Elle sert également de démonstration visuelle pour illustrer l’utilisation
de la structure List<>, en cohérence avec les objectifs pédagogiques du niveau 1 (structure linéaire + algorithme
simple d'activation en boucle).

2.2 CollectibleManager.cs (unity)

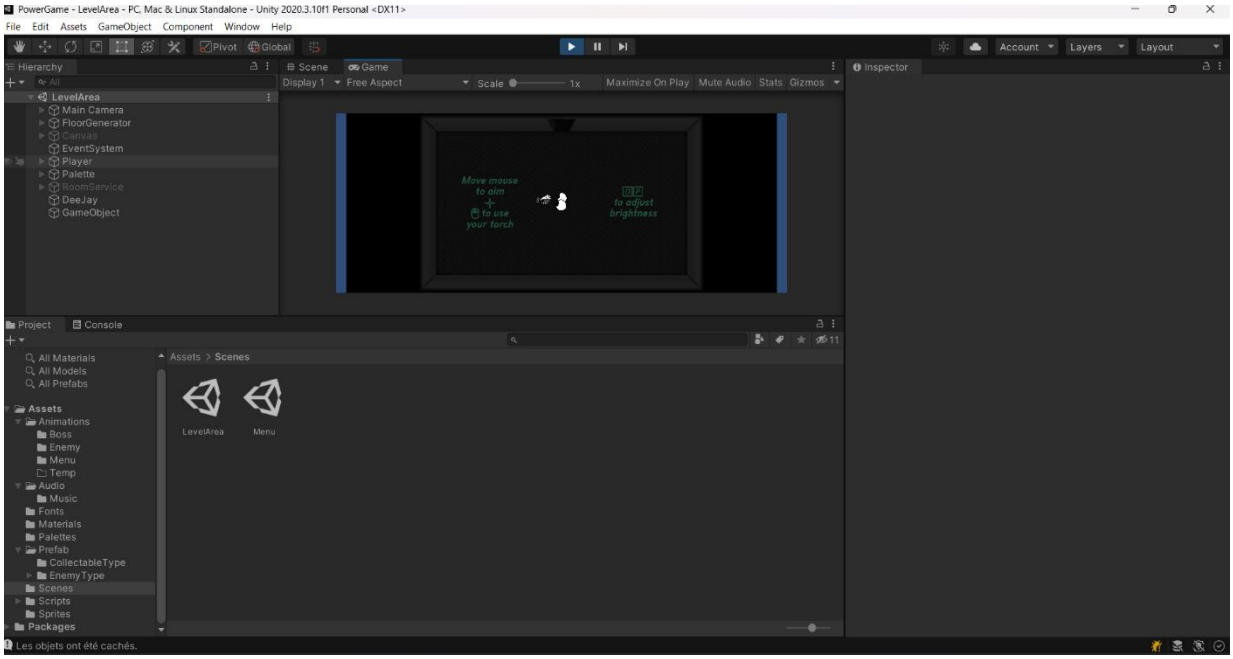


Figure 10: ***objet caché***

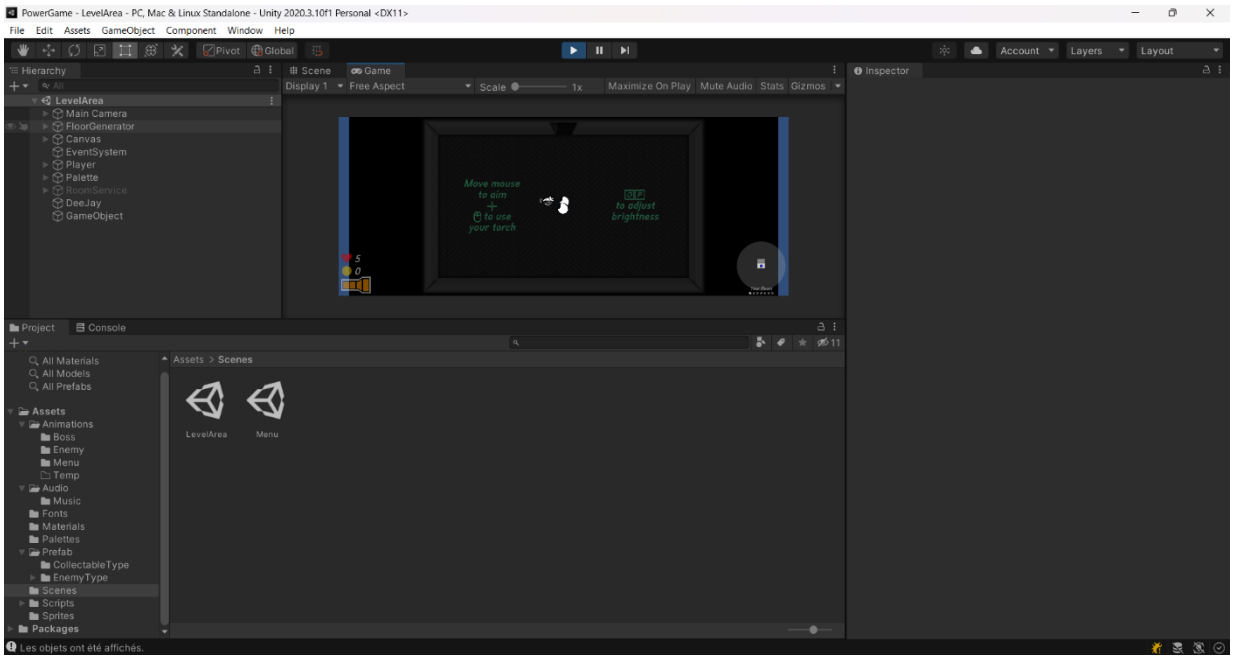


Figure 11: ***affichage d'objet***

Le script `CollectibleManager`, visible sur l'image, gère l'affichage ou la désactivation des objets ramassables dans le jeu (appelés *collectibles*). Ce système est lié aux objets comme "Battery", "Health", et "Canvas" qu'on voit dans l'éditeur Unity (sous le composant `CollectibleManager` dans l'Inspector). La capture Unity montre clairement que ces objets ont été liés au GameObject contenant ce script.

Dans le code, une liste publique de type `List<GameObject>` nommée `collectibles` permet de rassembler les objets ramassables. Lors de l'exécution du jeu, si le joueur appuie sur la touche **C**, la méthode `ShowCollectibles()` est appelée grâce à la condition dans la méthode `Update()`. Cette méthode vérifie si la liste est vide. Si c'est le cas, un message s'affiche dans la console :
`"Aucun objet ramassable n'est défini."`

Sinon, le script vérifie l'état du premier objet de la liste (`collectibles[0].activeSelf`) et applique le **même état inversé à tous les autres**. Cela signifie que si les objets étaient visibles (actifs), ils seront cachés (désactivés), et inversement. C'est un effet de **toggle** (basculement).

Un message est ensuite affiché dans la console pour indiquer ce qui s'est passé :

- `"Les objets ont été affichés"` si `setActive` est `true`,
- ou `"Les objets ont été cachés"` si `setActive` est `false`.

Cette fonctionnalité est utile en démonstration ou en test pour montrer comment **la structure de données (liste)** peut être utilisée dans Unity pour manipuler dynamiquement plusieurs objets dans une scène de jeu. On observe le résultat dans Unity : à l'exécution, les objets visibles ou invisibles changent en réponse à la touche "C", confirmant la bonne intégration du script.

2.3 InitializePatrolGraph

```
    // === Initialisation des points de patrouille === // LIGNE AJOUTÉE
    // Cette méthode crée trois points de patrouille : un au centre(position de départ), un à droite, un à gauche.
//Elle relie ces points entre eux pour former un graphe bidirectionnel, puis démarre la coroutine de déplacement.
    1 référence
    void InitializePatrolGraph() // LIGNE AJOUTÉE
    {
        // Exemple de 3 points de patrouille proches
        var nodeA = new PatrolNode(transform.position); // point initial
        var nodeB = new PatrolNode(transform.position + new Vector3(2f, 0, 0));
        var nodeC = new PatrolNode(transform.position + new Vector3(-2f, 0, 0));

        nodeA.neighbors.Add(nodeB);
        nodeA.neighbors.Add(nodeC);
        nodeB.neighbors.Add(nodeA);
        nodeC.neighbors.Add(nodeA);

        patrolGraph.AddRange(new[] { nodeA, nodeB, nodeC });
        currentPatrolNode = nodeA;

        StartCoroutine(PatrolRoutine());
    }
```

Figure 12 :création d'un graphe de patrouille

Ce bloc de code initialise une structure de données de type graphe non orienté, utilisée pour gérer la patrouille
automatique d'un ennemi. L'idée est de créer plusieurs nœuds de déplacement (ici nodeA, nodeB, nodeC) qui représentent
des positions fixes dans l'espace. Ces nœuds sont ensuite reliés entre eux grâce à la liste neighbors de chaque PatrolNode.
L'algorithme crée donc trois points autour de la position de départ du personnage ou de l'ennemi : un à droite, un à gauche
et un au centre. Ensuite, chaque point est connecté aux autres pour former un graphe bidirectionnel. Ces connexions permettent
à l'ennemi de se déplacer librement entre ces zones de manière fluide et logique. Cette structure est très utile dans les jeux vidéo pour modéliser des zones de déplacement réalistes et contrôlées, sans être linéaires.
Le graphe est ensuite stocké dans patrolGraph, et le point de départ est défini par currentPatrolNode. Enfin, une coroutine
PatrolRoutine() est lancée pour animer le déplacement entre ces points. Ce modèle est extrêmement flexible : on peut y ajouter
ou retirer des points facilement, ce qui le rend adaptable pour de futurs niveaux, des IA plus complexes ou même des systèmes de quête.

2.4. PatrolRoutine

```
// === Mouvement automatique entre les points du graphe === // LIGNE AJOUTÉE
// lee mouvement va etre dynamique et aleatoire et non pas lineaire comme dans le code de base grace a cette structre
1 référence
IEnumerator PatrolRoutine() // LIGNE AJOUTÉE
{
    while (!deadNow)
    {
        if (currentPatrolNode.neighbors.Count > 0)
        {
            int i = Random.Range(0, currentPatrolNode.neighbors.Count);
            PatrolNode nextNode = currentPatrolNode.neighbors[i];

            float elapsed = 0f;
            Vector3 start = transform.position;
            Vector3 end = nextNode.position;

            while (elapsed < 1f)
            {
                transform.position = Vector3.Lerp(start, end, elapsed);
                elapsed += Time.deltaTime * patrolSpeed;
                yield return null;
            }
            transform.position = end;
            currentPatrolNode = nextNode;

            yield return new WaitForSeconds(2f); // pause entre déplacements
            // ce code en general Cree coroutine permet à l'ennemi de se déplacer automatiquement entre les nœuds voisins du graphe.
        }
    }
}
```

Figure 13 : ***Mouvement automatique entre les points du graphe***

Cette coroutine, nommée PatrolRoutine(), permet à un ennemi de se déplacer automatiquement entre les points du graphe créés
dans InitializePatrolGraph. Son exécution commence par une boucle while (!deadNow), ce qui signifie que l'ennemi continue de
patrouiller tant qu'il est en vie.

L'algorithme commence par vérifier si le nœud courant (currentPatrolNode) possède des voisins. Si oui, il choisit un voisin
aléatoire à l'aide de Random.Range(). Ensuite, il calcule une interpolation entre la position actuelle et la position cible
à l'aide de Vector3.Lerp() : cette méthode crée un mouvement fluide entre deux positions. La variable elapsed augmente
progressivement avec le deltaTime, ce qui permet d'atteindre une animation douce et contrôlée.

Une fois arrivé au nouveau point, la coroutine attend un délai (WaitForSeconds(2f)) avant de recommencer le déplacement
vers un nouveau voisin. Cette structure rend le comportement de l'ennemi dynamique, non linéaire et imprévisible, ce qui
augmente le réalisme et la difficulté du jeu. Il ne suit pas un chemin rigide mais se déplace

selon un graphe, rendant le gameplay plus immersif.

Ce code est un excellent exemple de l'application d'un algorithme de déplacement sur graphe, et met en œuvre des concepts
comme les coroutines Unity, la gestion du temps, et l'animation de position avec interpolation. Il répond parfaitement aux exigences du niveau 3 (graphe + algorithme complexe) du projet.

1. Conclusion

Le projet PowerGame m'a permis d'approfondir mes compétences en structures de données, conception de gameplay, et collaboration en Git. Il constitue une base solide pour mes futurs projets en programmation de jeux.

Conclusion Générale

Les jeux vidéo interactifs sont devenus une composante essentielle du paysage numérique moderne. Leur développement nécessite la maîtrise de plusieurs concepts techniques, allant de la conception d'interfaces à la mise en œuvre d'algorithmes et de structures de données complexes. Dans le cadre de notre projet de fin de session, nous avons conçu et développé un jeu vidéo éducatif nommé **PowerGame**, intégrant plusieurs structures de données (liste, liste chaînée, graphe) dans des scénarios de gameplay concrets.

Ce rapport retrace toutes les étapes de la conception du jeu, en passant par l'analyse des besoins, le choix des outils (Unity, Visual Studio, C#), la modélisation UML (diagrammes de classes, séquence et déploiement), l'implémentation technique ainsi que les tests fonctionnels. Le projet a été mené selon une démarche itérative et incrémentale, chaque semaine étant dédiée à l'ajout d'un nouveau niveau de complexité (structure + algorithme).

Cette expérience a été enrichissante à plusieurs niveaux. Sur le plan humain, elle nous a permis de travailler en autonomie, de collaborer à distance via GitHub et de structurer notre temps. Sur le plan technique, nous avons approfondi notre compréhension des **structures linéaires et non linéaires**, tout en les intégrant dans un environnement réel (le moteur Unity). La création d'un comportement d'ennemi basé sur un graphe et d'une interface de gestion d'objets ramassables via une liste dynamique sont des exemples concrets de cette maîtrise.

Enfin, PowerGame reste une base évolutive. Il pourrait être amélioré à l'avenir par l'ajout de fonctionnalités supplémentaires telles que la sauvegarde de progression, un système de quêtes, ou même un mode multijoueur. Ce projet constitue donc une première étape prometteuse dans notre parcours de développement de jeux vidéo et de programmation orientée objet.

néographie

[1] https://en.wikipedia.org/wiki/C_Sharp_(programming_language)

[2]https://en.wikipedia.org/wiki/Visual_Studio#:~:text=Visual%20Studio%20is%20an%20integrated,web%20services%20and%20mobile%20apps.

[3] https://en.wikipedia.org/wiki/Unity_(game_engine)