Universitatea Tehnică a Moldovei
Facultatea Calculatoare Informatică și Microelectronică
Departamentul Ingineria Software și Automatică

Proiect de an

Disciplina : Tehnici și mecanisme de proiectare a produselor program
Tema:”Zombie Shooter . Aplicatie Recreativa”

A efectuat: Mandis Nichita, TI-224

A verificat: Bîtca Ernest

Chişinău, 2025

Cuprins

Introducere

Acest proiect este un joc simplu de tip **top-down zombie shooter**, realizat în Unity ca exercițiu de învățare și dezvoltare personală. În ciuda resurselor limitate și a unui buget comparabil cu o pungă de biscuiți sau o cafea ieftină, am încercat să aplic principii solide de programare pentru a menține codul organizat, scalabil și ușor de întreținut.

Pentru a gestiona mai bine complexitatea jocului – chiar dacă este un proiect mic – am integrat câteva **șabloane de proiectare (design patterns)** cunoscute. Acestea m-au ajutat să evit scrierea codului redundant, să separ responsabilitățile între clase și să pot extinde mai ușor funcționalitățile jocului.

În cadrul acestui proiect am folosit următoarele cinci șabloane:

- **Factory Method** – pentru instanțierea zombi-lor într-un mod controlat;
- **Singleton** – pentru a asigura accesul global la GameManager și alți manageri esențiali;
- **Object Pooling (Flyweight)** – pentru gestionarea eficientă a gloanțelor și evitarea instanțierii excesive;
- **Component-Based Architecture (inspirație din Strategy/Decorator)** – pentru comportamentele inamicilor și modularitatea logici;
- **Prototype (implicit)** – folosirea prefabricatelor (Zombie Variant) în Unity implică de fapt o clonare rapidă a unui „șablon” de obiect, ceea ce se aliniază cu intenția pattern-ului Prototype.
- **Observer simplificat** – pentru a notifica diferite sisteme când se întâmplă evenimente importante (ex: moartea unui zombie).

În paginile următoare sunt prezentate aceste pattern-uri, modul în care au fost integrate în joc și ce probleme concrete au rezolvat.

Scopurile / obiectivele proiectului

Scopul acestui proiect este de a realiza un joc simplu de tip **zombie shooter**, care să demonstreze în mod practic aplicarea mai multor șabloane de proiectare esențiale în dezvoltarea de jocuri video, cu accent pe eficiență, modularitate și scalabilitate. Proiectul este realizat într-un context indie, cu resurse limitate, dar cu o abordare profesionistă în arhitectura codului.
Prin această implementare se urmărește evidențierea modului în care design patterns contribuie la reducerea complexității, separarea responsabilităților și îmbunătățirea mentenanței codului într-un proiect Unity.

Obiectivele proiectului:

- **Aplicarea șablonului Factory Method** pentru instanțierea zombilor printr-o clasă `ZombieFactory`, permițând crearea controlată și extensibilă a inamicilor.
- **Utilizarea șablonului Singleton** pentru gestionarea globală a managerilor principali (ex. `GameManager`, `ShotPool`), evitând instanțierea multiplă și facilitând accesul centralizat la funcționalități esențiale.
- **Implementarea Object Pooling (Flyweight)** pentru reutilizarea eficientă a obiectelor de tip glonț, reducând instanțierile și descărcările inutile de memorie în timpul jocului.
- **Utilizarea indirectă a șablonului Prototype** prin clonarea prefab-urilor Unity (ex. `Zombie Variant`), ceea ce permite generarea rapidă a unor entități complexe cu stări predefinite.
- **Folosirea unei arhitecturi bazate pe componente (inspirată din Strategy și Decorator)** pentru comportamentele zombilor, asigurând modularitate și flexibilitate în logica de gameplay.
- **Aplicarea unui Observer simplificat** pentru notificarea diferitelor subsisteme (UI, sistem de scor etc.) la evenimente relevante din joc, cum ar fi moartea zombilor.
- **Organizarea codului în clase cu responsabilități clare**, astfel încât să fie ușor de extins în viitor cu noi tipuri de inamici, arme sau mecanici de joc.

- **Documentarea clară a structurii proiectului** și a fiecărui șablon utilizat, pentru a sprijini înțelegerea și reutilizarea arhitecturii în alte proiecte Unity similare.

Proiectarea sistemului

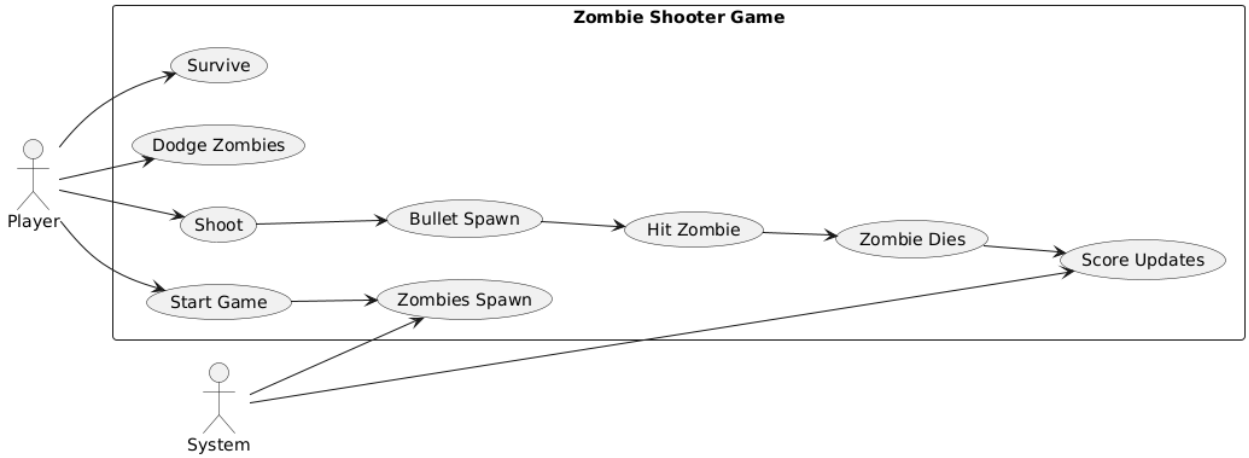

Figura 1 – Diagrama Use Case

User interacționează cu aplicația prin acțiuni de start, tragere, evitare a zombilor și supraviețuire.

Sistemul gestionează apariția zombilor și actualizarea scorului.

Când utilizatorul trage, se creează un glonț care poate lovi un zombie.

La lovirea zombicului, acesta moare, iar scorul este actualizat.

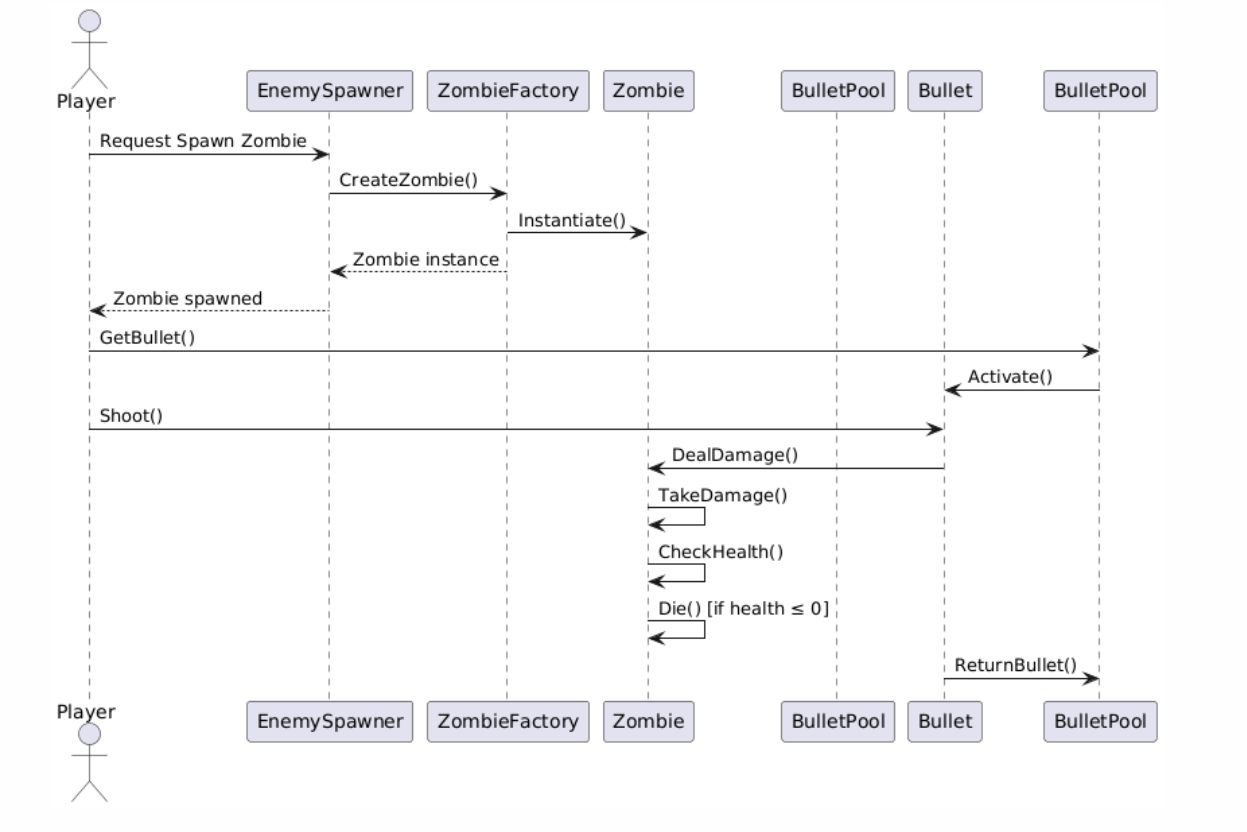

Figura 2 – Diagrama secvențială

Jucătorul cere spawner-ului să genereze un zombie.

Spawner-ul folosește Factory pentru a crea un zombie.

Zombie-ul este instanțiat și trimis înapoi spawner-ului.

Jucătorul cere un glonț din pool, îl activează și îl împușcă.

Glonțul provoacă daune zombie-ului.

Zombie-ul își verifică sănătatea și moare dacă este cazul.

Glonțul este returnat în pool pentru reutilizare.

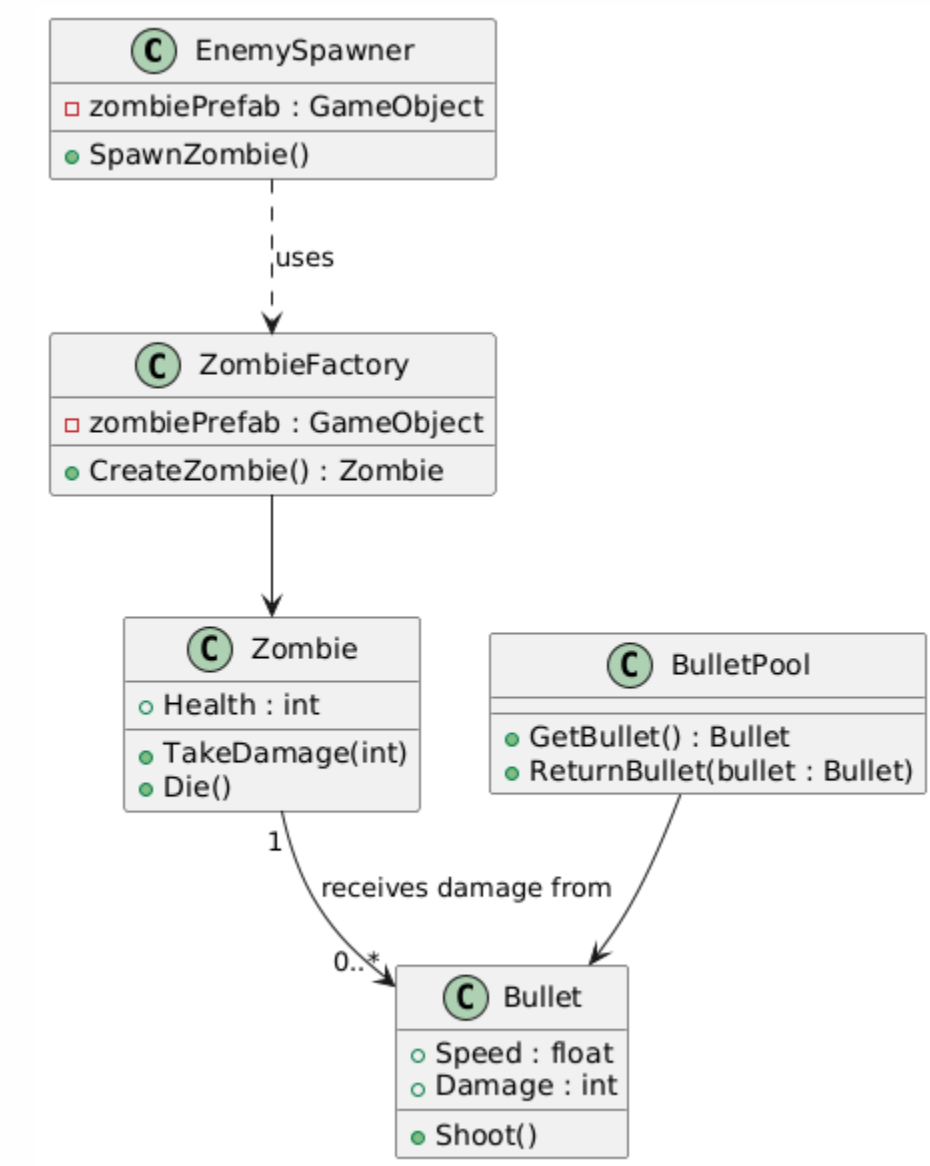

Figura 3 – Diagrama de clase

EnemySpawner folosește Factory pentru a crea zombi folosind prefab-ul.

ZombieFactory este responsabil pentru instanțierea zombilor.

BulletPool gestionează reutilizarea gloanțelor (Flyweight/Object Pooling).

Zombie primește daune de la gloanțe și poate muri.

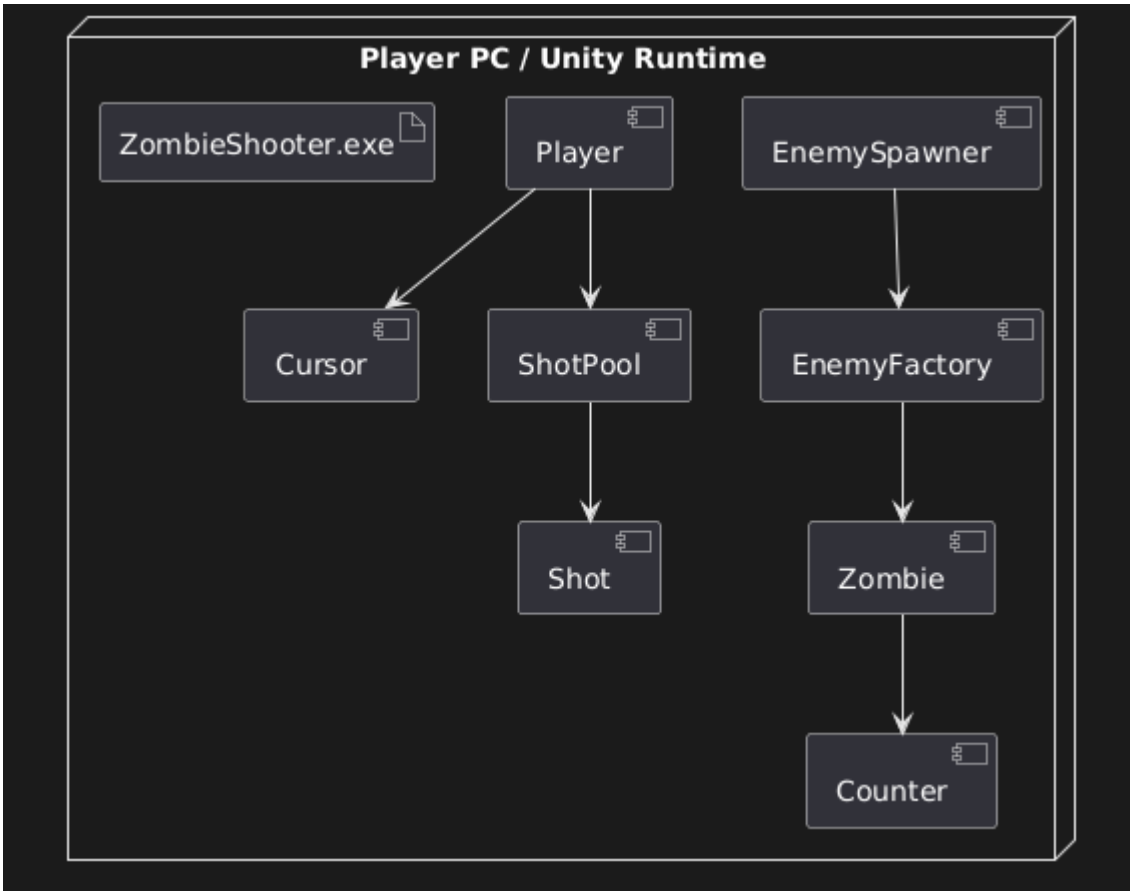

Figura 4 – Diagrama de deployment

Jocul rulează pe PC-ul jucătorului în cadrul motorului Unity.

Componentele principale precum Player, Cursor, EnemySpawner, EnemyFactory, ShotPool, Shot, Zombie și Counter sunt toate implementate în același runtime.

Player interacționează cu Cursor pentru poziționarea țintei.

EnemySpawner folosește EnemyFactory pentru a crea inamici (zombi).

Player utilizează ShotPool pentru gestionarea gloanțelor, care sunt reprezentate prin componentele Shot.

Zombi sunt create de EnemyFactory și, atunci când sunt uciși, actualizează Counter-ul pentru a ține scorul.

Toate aceste componente comunică intern în cadrul aceluiași nod, fără interacțiune cu un sistem extern de stocare a datelor.

Implementarea

Descriere implementare aplicație:

Aplicația este dezvoltată în C# folosind motorul de joc Unity, aplicând un design modular și bazat pe componente. Fiecare componentă este responsabilă pentru o parte specifică a funcționalității jocului.

- **Player** este o clasă singleton care controlează mișcarea jucătorului folosind NavMeshAgent, gestionează rotația către țintă și controlează atacul prin tragerea proiectilelor. De asemenea, Player interacționează cu obiectul **Cursor** pentru a determina poziția țintei pe teren.
- **Cursor** utilizează Raycast pentru a detecta poziția mouse-ului pe teren și actualizează poziția sa pentru a indica vizual ținta.
- **EnemySpawner** se ocupă cu generarea periodică a inamicilor (zombi) pe scena jocului, instanțiind obiectele Enemy la intervale reglate.
- **Zombie** reprezintă inamicul care se deplasează către jucător folosind NavMeshAgent și poate fi ucis prin loviturile proiectilelor. Clasa gestionează și animațiile și efectele vizuale asociate morții.
- **ShotPool** implementează un pool de obiecte pentru gestionarea eficientă a proiectilelor, reducând costurile legate de instanțiere și distrugere frecventă. Acest pool oferă și recuperează obiectele Shot.
- **Shot** reprezintă proiectilul tras de Player, cu reprezentare vizuală prin LineRenderer și logica de activare/dezactivare din pool.

- **Counter** ține evidența numărului de inamici eliminați și actualizează interfața utilizatorului (UI).

Toate aceste componente interacționează în cadrul scenei Unity pentru a crea o experiență de joc fluidă, în care jucătorul se deplasează, țintește și doboară inamicii, iar progresul este reflectat prin UI.

Utilizarea șabloanelor de proiectare

Singleton (în clasa Player și ShotPool)

- **Descriere:**
  Pattern-ul Singleton asigură că o clasă are o singură instanță globală și oferă un punct de acces la acea instanță. În codul tău, `Player` și `ShotPool` folosesc acest pattern pentru a garanta că există o singură instanță accesibilă oriunde în joc.
- **+:**
  - Acces ușor și global la instanța unică.
  - Evită crearea mai multor instanțe inutile.
- **-:**
  - Dificultăți la testare unitară (mocking).
  - Poate crea dependențe ascunse greu de urmărit.
  - Riscul de probleme în context multi-threading (deși aici nu e cazul).

Prototype (în EnemySpawner folosind prefab-uri pentru clonarea zombilor)

- **Descriere:**
  Pattern-ul Prototype permite crearea obiectelor noi prin copierea (clonarea) unui obiect existent, în loc să fie creat de la zero. În Unity, folosirea prefab-urilor ca template-uri pentru instanțierea clonelor reprezintă o implementare a acestui pattern. `EnemySpawner` instanțiază clona prefab-ului `Enemy` pentru a crea zombii noi.
- **+:**

    - Eficiență în creare rapidă de obiecte similare.
    - Simplifică inițializarea, folosind un șablon gata configurat.
    - Ușor de modificat șablonul pentru toate clonele.
- **-:**
    - Uneori poate cauza dificultăți dacă clonarea trebuie să includă stări complexe sau referințe externe.
    - Dependență de prefab-uri configurate corect.

Object Pool (în ShotPool)

- **Descriere:**
  Pattern-ul Object Pool gestionează un set de obiecte reutilizabile pentru a evita costul ridicat al creării și distrugerii frecvente. `ShotPool` păstrează un pool de proiectile (Shot), dezactivează și reactivează aceste obiecte când sunt folosite.
- **+:**
    - Optimizează performanța prin reducerea instanțierilor/destructorilor.
    - Potrivit pentru obiecte folosite frecvent și pe termen scurt.
- **-:**
    - Implementarea pool-ului poate fi mai complexă.
    - Dacă pool-ul nu este bine dimensionat, poate apărea lipsă de obiecte disponibile sau memorie irosită.

Component (în MovementAnimator, Zombie, Player etc.)

- **Descriere:**
  Pattern-ul Component (folosit de Unity ca principiu) încurajează compunerea comportamentului unui obiect prin atașarea mai multor componente independente. De exemplu, `MovementAnimator` controlează animațiile, `Zombie` gestionează comportamentul inamicului, iar `Player` controlează jucătorul.

- **+:**
    - Modularitate și reutilizare.
    - Separare clară a responsabilităților.
    - Flexibilitate în combinarea comportamentelor.
- **-:**
    - Poate fi greu să urmărești interacțiunile complexe între componente.
    - Dificultăți în sincronizarea stărilor între componente.

Factory Method simplificat (EnemyFactory)

- **Descriere:**
  `EnemyFactory` este o clasă care creează obiecte inamice, ascunzând detaliile instanțierii față de client (în cazul tău, față de codul care cere inamici). Aceasta este o formă simplificată a pattern-ului Factory Method.
- **+:**
    - Separă logica de creare a obiectelor de restul codului.
    - Ușurează extinderea tipurilor de inamici fără modificări majore în codul client.
- **-:**
    - În codul tău actual, această clasă nu este folosită efectiv (EnemySpawner folosește direct Instantiate).
    - Fără o utilizare clară, poate complica codul inutil.

Importanța utilizării șabloanelor de proiectare în cod

În dezvoltarea jocului meu simplu top-down shooter cu zombi în Unity, folosirea șabloanelor de proiectare a fost esențială pentru menținerea clarității, modularității și scalabilității codului. Chiar și într-un proiect mic, complexitatea interacțiunilor între componente crește rapid, iar șabloanele oferă soluții clare, reutilizabile și ușor de extins.

Un exemplu concret este folosirea șablonului **Prototype** pentru clonarea inamicilor. În loc să creez manual fiecare zombi, am folosit un prefab care reprezintă un model de bază, iar apoi îl clonez la nevoie în joc. Astfel, pot adăuga rapid mai mulți inamici identici, iar dacă modific prefab-ul, toate clonele se actualizează implicit.

Pentru instanțierea efectivă a zombiilor, am aplicat o versiune simplificată a șablonului **Factory** prin clasa `EnemyFactory`. Acesta ascunde detaliile legate de cum se creează inamicul (prin clonarea prefab-ului) și oferă o metodă clară, `CreateEnemy()`, care primește poziția și rotația, iar apoi returnează noul obiect. Astfel, logica de creare este separată de restul codului, iar dacă vreau să schimb modul de generare (de exemplu, să adaug tipuri diferite de inamici), pot modifica doar fabrica, fără să afectez spawner-ul sau alte componente.

Acest lucru contribuie și la o arhitectură mai bună a jocului: `EnemySpawner(ZombiFabrica)` se ocupă doar de momentul și locul generării inamicilor, în timp ce `EnemyFactory` se ocupă de detaliile tehnice ale creației. Această separare de responsabilități face codul mai ușor de întreținut și extins. Spre exemplu, dacă în viitor voi dori să adaug un tip nou de zombi, pot crea o subclasă a fabricii sau un alt prefab fără să modific spawner-ul.

De asemenea, folosirea acestor șabloane face colaborarea și înțelegerea codului mai simpla, chiar dacă lucrez singur sau cu alți dezvoltatori. Termenii precum “Prefab”, “Factory” sau “Prototype” devin un limbaj comun, care reduce timpul petrecut cu explicarea funcționalităților și facilitează revizuirea codului.

În plus, design-ul modular creat cu ajutorul acestor șabloane permite testarea mai ușoară. Pot testa separat dacă fabrica creează corect inamicii, fără a rula întregul joc, iar schimbările ulterioare devin mai sigure.

Totuși, este important să nu forțez aplicarea unui șablon când nu e cazul, pentru că ar putea complica inutil codul(Ceea ce s-a facut in unele locuri =)). În proiectul meu, am început cu o implementare simplă a fabricii, iar pe măsură ce jocul crește, pot transforma aceasta într-un Factory Method complet, cu clase și metode abstracte, dacă va fi nevoie să gestionez mai multe tipuri de inamici.

Interacțiunea cu utilizatorul în proiectul Unity

În proiectul Unity, interacțiunea cu utilizatorul este realizată printr-un sistem simplu și direct de input de la tastatură și mouse, gestionat în principal în clasa **Player**:

- Jucătorul controlează personajul folosind tastele **W, A, S, D** pentru mișcare în scenă.
- Tragerea cu arma se realizează prin apăsarea butonului stâng al mouse-ului, iar lovirea inamicului este detectată printr-un **Raycast**.
- Poziția cursorului pe teren este actualizată de componenta **Cursor**, care citește poziția mouse-ului și mută cursorul pe scena de joc.
- Starea și orientarea personajului sunt actualizate în timp real, urmărind poziția cursorului.

Avantajele acestei abordări sunt:

- Separarea clară a responsabilităților — mișcarea, animația, logica inamicilor și UI sunt gestionate în componente independente.
- Ușurința extinderii — se pot adăuga noi tipuri de inamici sau comportamente prin modificarea componentelor corespunzătoare.
- Eficiență — pool-ul de obiecte reduce costurile performanței la efectuarea frecventă a focurilor, iar navigația se bazează pe NavMeshAgent.

Limitări:

- Interfața simplă poate să nu ofere o experiență vizuală completă fără un UI mai complex.

Secvențe de cod

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

public class Player : MonoBehaviour
{
    public static Player Instance { get; private set; }

    private Shot shot;
    private Cursor cursor;
    private NavMeshAgent _navMeshAgent;

    public float moveSpeed;
    public Transform gunBarrel;

    void Awake()
    {
        // Singleton
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
            return;
        }

        Instance = this;

        DontDestroyOnLoad(gameObject);
    }

    void Start()
    {
        cursor = FindObjectOfType<Cursor>();
        shot = FindObjectOfType<Shot>();
        _navMeshAgent = GetComponent<NavMeshAgent>();
        _navMeshAgent.updateRotation = false;
    }
```

```
    void Update()
    {
        Vector3 dir = Vector3.zero;
        if (Input.GetKey(KeyCode.A)) dir.z = -1.0f;
        if (Input.GetKey(KeyCode.D)) dir.z = 1.0f;
        if (Input.GetKey(KeyCode.S)) dir.x = 1.0f;
        if (Input.GetKey(KeyCode.W)) dir.x = -1.0f;
        _navMeshAgent.velocity = dir.normalized * moveSpeed;

        if (Input.GetMouseButtonDown(0))
        {
            var from = gunBarrel.position;
            var target = cursor.transform.position;
            var to = new Vector3(target.x, from.y, target.z);
            var direction = (to - from).normalized;

            RaycastHit hit;
            if (Physics.Raycast(from, direction, out hit, 100))
            {
                if (hit.transform != null)
                {
                    var zombie = hit.transform.GetComponent<Zombie>();
                    if (zombie != null)
                        zombie.Kill();
                }

                to = new Vector3(hit.point.x, from.y, hit.point.z);
            }
            else
            {
                to = from + direction * 100;
            }

            var shotInstance = ShotPool.Instance.GetShot();
            shotInstance.Show(from, to);
        }

        Vector3 forward = cursor.transform.position - transform.position;
        transform.rotation = Quaternion.LookRotation(new Vector3(forward.x, 0, forward.z));
    }
}
```

Secventa 1 - Singleton

Script: Player

Pattern-ul Singleton asigură existența unei singure instanțe a clasei Player și oferă un punct global de acces prin proprietatea statică `Instance`. Astfel, se previne crearea mai multor obiecte Player în scenă și se păstrează o instanță unică pe durata jocului.

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

public class ShotPool : MonoBehaviour
{
    public static ShotPool Instance;

    public GameObject shotPrefab;
    public int poolSize = 10;

    private Queue<Shot> pool = new Queue<Shot>();

    void Awake()
    {
        Instance = this;
        for (int i = 0; i < poolSize; i++)
        {
            var go = Instantiate(shotPrefab);
            go.SetActive(false);
            pool.Enqueue(go.GetComponent<Shot>());
        }
    }

    public Shot GetShot()
    {
        if (pool.Count > 0)
        {
            Shot shot = pool.Dequeue();
            shot.gameObject.SetActive(true);
```

```
            return shot;
        }
        else
        {
            var go = Instantiate(shotPrefab);
            return go.GetComponent<Shot>();
        }
    }

    public void ReturnShot(Shot shot)
    {
        shot.gameObject.SetActive(false);
        pool.Enqueue(shot);
    }
}
```

Secventa 2 - Pool de Obiecte (Object Pool)

Script: ShotPool

Pattern-ul Pool de Obiecte gestionează o colecție de obiecte reutilizabile, reducând costurile instanțierii și distrugerii frecvente. În acest script, se menține un pool de proiectile (Shot), care sunt activate și dezactivate pentru a fi reutilizate în loc să fie recreate, ceea ce îmbunătățește performanța.

```
using UnityEngine;

public class EnemyFactory
{
    private GameObject _enemyPrefab;

    public EnemyFactory(GameObject enemyPrefab)
    {
        _enemyPrefab = enemyPrefab;
    }

    public GameObject CreateEnemy(Vector3 position, Quaternion rotation)
    {
        return GameObject.Instantiate(_enemyPrefab, position, rotation);
    }
```

```
}
```

Secventa 3 - Simple Factory Pattern

Script: EnemyFactory (ZombiFabrica)

Pattern-ul Fabrica oferă o metodă simplificată și centralizată pentru crearea obiectelor (în acest caz, inamici). Metoda `CreateEnemy` instanțiază inamici folosind prefab-uri, separând procesul de creare a obiectelor de restul codului.

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

public class MovementAnimator : MonoBehaviour
{
    private NavMeshAgent _navMeshAgent;

    private Animator _animator;

    // Start is called before the first frame update
    void Start()
    {
        _navMeshAgent = GetComponent<NavMeshAgent>();
        _animator = GetComponentInChildren<Animator>();
    }

    // Update is called once per frame
    void Update()
    {
        _animator.SetFloat("speed", _navMeshAgent.velocity.magnitude);
    }
}
```

Secventa 4 - Component

Script: MovementAnimator

Pattern-ul Component permite adăugarea modulară a funcționalităților unui obiect. În

MovementAnimator, această componentă gestionează animațiile în funcție de viteza agentului NavMesh, separând logica mișcării de cea a animației.

```
using System;
using UnityEngine;
using UnityEngine.UI;

public class Counter : MonoBehaviour
{
    public event Action<int> OnKillCountChanged;

    private int kills;
    [SerializeField] private Text killsText;

    public void AddKill()
    {
        kills++;
        killsText.text = kills.ToString();
        OnKillCountChanged?.Invoke(kills);
    }
}
```

Secventa 5 - Observer (Observator)

Script: Counter

Pattern-ul Observer permite ca un obiect să notifice alte obiecte despre schimbări de stare fără să fie strâns legat de ele. În acest script, clasa Counter emite un eveniment OnKillCountChanged de fiecare dată când numărul de ucideri se schimbă, permițând altor componente să se aboneze și să reacționeze la aceste schimbări fără a avea o dependență directă.

Documentarea Produsului

Figura 5 – Texture pack pe fundal de SkyBox

Texture Pack din Unity Assets Store folosit in proiect.

Figura 6 – Playerul

Așa arată personajul principal (controlat cu WASD).

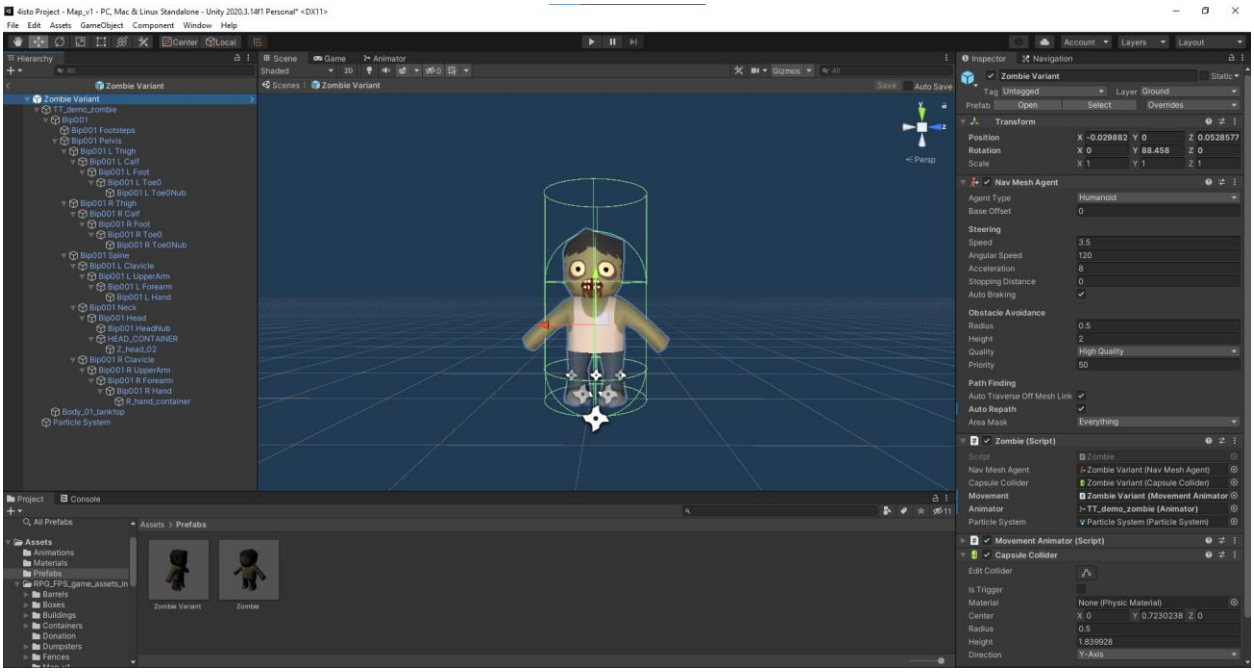

Figura 7 – Prefabul la Zombie

Prefab-ul Zombie Variant este utilizat pe baza zombiului de bază (cu toți parametrii lui), este folosit în Enemy Spawner / Zombie Factory

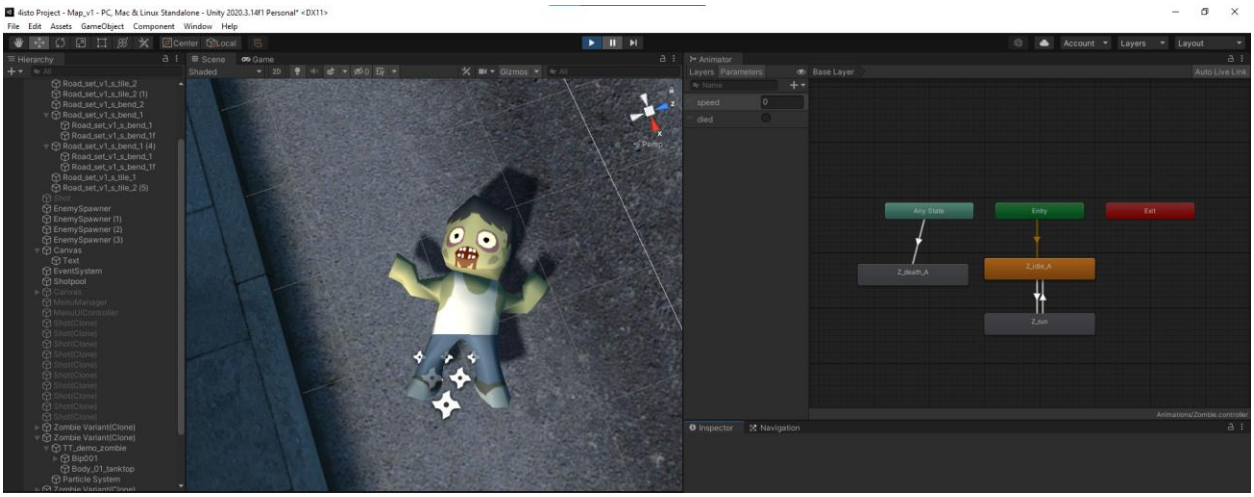

Figura 8 – Zombie_Animation

În această captură de ecran vedem cadrul final al animațiilor morții zombiului nostru și logica controlerului nostru de animație.

Figura 8 – Zombie_Particles

In aceasta imagine observam effectul de primire a „damage-ului” , ca urmare vedem semne de

„singe” verde (Caracteristic pentru un zombie :D).

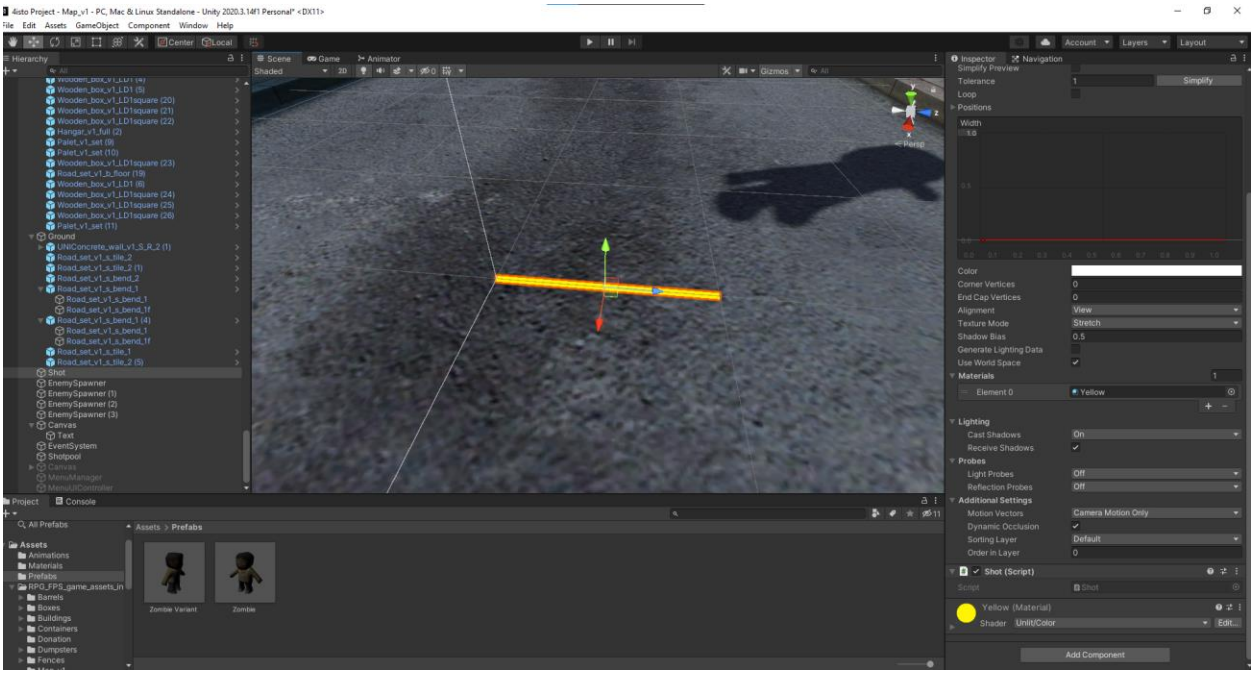

Figura 9 – Shot (using RayCast)

Aici vedem cum arată Tragerea(Impuscarea) noastra

Concluzie

Acest proiect a fost dezvoltat cu scopul de a crea un joc simplu de tip shooter cu zombii, în care jucătorul are libertatea de a se deplasa liber pe hartă, de a ținti și elimina inamicii care apar constant. Pe parcursul procesului de dezvoltare, am învățat să integrez diverse componente esențiale în Unity, precum sistemul de navigație inteligentă cu ajutorul NavMeshAgent, gestionarea interacțiunii prin raycast-uri, și afișarea elementelor vizuale, cum ar fi linia proiectilelor (Shot) și numărătoarea eliminărilor (Counter).

Proiectul nu s-a limitat doar la implementarea funcționalităților de bază, ci am acordat o atenție specială organizării codului folosind **pattern-uri de design**. Aceste pattern-uri au un rol fundamental în dezvoltarea software-ului, mai ales în jocuri, deoarece oferă soluții clare, reutilizabile și testate pentru probleme frecvente. De exemplu, pattern-ul **Singleton** aplicat în clasa `Player` garantează existența unei singure instanțe a jucătorului, evitând conflicte și facilitând accesul global la aceasta. Pattern-ul **Object Pool** din clasa `ShotPool` optimizează performanța prin reutilizarea obiectelor în loc să le creeze și distrugă constant, reducând astfel costurile de procesare și alocare de memorie.

Folosirea acestor pattern-uri este esențială pentru a menține codul curat, modular și ușor de extins. Ele permit dezvoltatorilor să adauge funcționalități noi fără să afecteze alte părți ale aplicației, ceea ce este vital pentru proiectele mari sau pentru munca în echipă. Mai mult, pattern-urile ajută la reducerea erorilor și fac codul mai ușor de înțeles și de întreținut pe termen lung.

În concluzie, prin acest proiect am reușit să aplic atât conceptele tehnice de bază din Unity, cât și principii avansate de arhitectură software, construind o fundație solidă pentru dezvoltarea unor jocuri mai complexe și mai bine structurate în viitor.

Anexa 1

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using static System.Net.Mime.MediaTypeNames;



public class Counter : MonoBehaviour
{
    public int Count = 0;
    Text _text;
    // Start is called before the first frame update
    void Start()
    {
        _text = GetComponent<Text>();
    }

    // Update is called once per frame
    void Update()
    {

    }

    public void AddKill()
    {
        Count = Count + 1;
        _text.text = $"{Count}";
    }
}

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

public class Cursor : MonoBehaviour
{
    private SpriteRenderer _spriteRenderer;

    private int layerMask;
    // Start is called before the first frame update
    void Start()
    {
        _spriteRenderer = GetComponent<SpriteRenderer>();
        layerMask = LayerMask.GetMask("Ground");
    }

    // Update is called once per frame
    void Update()
    {
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

        RaycastHit hit;
        if (Physics.Raycast(ray, out hit, 1000, layerMask))
        {
            transform.position = new Vector3(hit.point.x, transform.position.y, hit.point.z);
            _spriteRenderer.enabled = true;
```

```
        }
        else
        {
            _spriteRenderer.enabled = false;
        }
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnemySpawner : MonoBehaviour
{
    public float Period;

    public GameObject Enemy;
    private float _TimeUntilNextSpawn;


    // Start is called before the first frame update
    void Start()
    {
        _TimeUntilNextSpawn = Random.Range(0, Period);
    }

    // Update is called once per frame
    void Update()
    {
        _TimeUntilNextSpawn -= Time.deltaTime;
        if (_TimeUntilNextSpawn <= 0.0f)
        {
            _TimeUntilNextSpawn = Period;
            Instantiate(Enemy, transform.position, transform.rotation);
        }
    }
}

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class MovementAnimator : MonoBehaviour
{
    private NavMeshAgent _navMeshAgent;

    private Animator _animator;

    // Start is called before the first frame update
    void Start()
    {
        _navMeshAgent = GetComponent<NavMeshAgent>();
        _animator = GetComponentInChildren<Animator>();
    }

    // Update is called once per frame
    void Update()
    {
        _animator.SetFloat("speed", _navMeshAgent.velocity.magnitude);
    }
}
```

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

public class Player : MonoBehaviour
{
    public static Player Instance { get; private set; }

    private Shot shot;
    private Cursor cursor;
    private NavMeshAgent _navMeshAgent;

    public float moveSpeed;
    public Transform gunBarrel;

    void Awake()
    {
        // Singleton
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
            return;
        }

        Instance = this;

        DontDestroyOnLoad(gameObject);
    }

    void Start()
    {
        cursor = FindObjectOfType<Cursor>();
        shot = FindObjectOfType<Shot>();
        _navMeshAgent = GetComponent<NavMeshAgent>();
        _navMeshAgent.updateRotation = false;
    }

    void Update()
    {
        Vector3 dir = Vector3.zero;
        if (Input.GetKey(KeyCode.A)) dir.z = -1.0f;
        if (Input.GetKey(KeyCode.D)) dir.z = 1.0f;
        if (Input.GetKey(KeyCode.S)) dir.x = 1.0f;
        if (Input.GetKey(KeyCode.W)) dir.x = -1.0f;
        _navMeshAgent.velocity = dir.normalized * moveSpeed;

        if (Input.GetMouseButtonDown(0))
        {
            var from = gunBarrel.position;
            var target = cursor.transform.position;
            var to = new Vector3(target.x, from.y, target.z);
            var direction = (to - from).normalized;

            RaycastHit hit;
            if (Physics.Raycast(from, direction, out hit, 100))
            {
                if (hit.transform != null)
                {
                    var zombie = hit.transform.GetComponent<Zombie>();
```

```
                    if (zombie != null)
                        zombie.Kill();
                }

                to = new Vector3(hit.point.x, from.y, hit.point.z);
            }
            else
            {
                to = from + direction * 100;
            }

            var shotInstance = ShotPool.Instance.GetShot();
            shotInstance.Show(from, to);
        }

        Vector3 forward = cursor.transform.position - transform.position;
        transform.rotation = Quaternion.LookRotation(new Vector3(forward.x, 0, forward.z));
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerCamera : MonoBehaviour
{
    private Player player;

    private Vector3 offset;
    // Start is called before the first frame update
    void Start()
    {
        player = FindObjectOfType<Player>();
        offset = transform.position - player.transform.position;


    }

    // Update is called once per frame
    void Update()
    {
        transform.position = player.transform.position + offset;
    }
}

using UnityEngine;

public class Shot : MonoBehaviour
{
    private LineRenderer _lineRenderer;
    private bool visible;

    void Awake()
    {
        _lineRenderer = GetComponent<LineRenderer>();
    }

    void OnEnable()
    {

        _lineRenderer.enabled = true;
        visible = true;
```

```
    }

    void FixedUpdate()
    {
        if (visible)
        {
            visible = false;
        }
        else
        {

            ShotPool.Instance.ReturnShot(this);
        }
    }

    public void Show(Vector3 from, Vector3 to)
    {
        if (!_lineRenderer) _lineRenderer = GetComponent<LineRenderer>();

        _lineRenderer.SetPositions(new Vector3[] { from, to });
        _lineRenderer.enabled = true;
        visible = true;
        gameObject.SetActive(true);
    }

    void OnDisable()
    {

        if (_lineRenderer != null)
            _lineRenderer.enabled = false;
    }
}
using System.Collections.Generic;
using UnityEngine;

public class ShotPool : MonoBehaviour
{
    public static ShotPool Instance;

    public GameObject shotPrefab;
    public int poolSize = 10;

    private Queue<Shot> pool = new Queue<Shot>();

    void Awake()
    {
        Instance = this;
        for (int i = 0; i < poolSize; i++)
        {
            var go = Instantiate(shotPrefab);
            go.SetActive(false);
            pool.Enqueue(go.GetComponent<Shot>());
        }
    }

    public Shot GetShot()
    {
        if (pool.Count > 0)
        {
            Shot shot = pool.Dequeue();
            shot.gameObject.SetActive(true);
```

```
            return shot;
        }
        else
        {
            var go = Instantiate(shotPrefab);
            return go.GetComponent<Shot>();
        }
    }

    public void ReturnShot(Shot shot)
    {
        shot.gameObject.SetActive(false);
        pool.Enqueue(shot);
    }
}

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class Zombie : MonoBehaviour
{
    [SerializeField] private NavMeshAgent _navMeshAgent;
    [SerializeField] private CapsuleCollider _capsuleCollider;
    [SerializeField] private MovementAnimator _movement;
    [SerializeField] private Animator _animator;
    [SerializeField] private ParticleSystem _particleSystem;

    private Player _player;
    private Counter _counter;
    private bool dead;

    void Start()
    {
        _player = FindObjectOfType<Player>();
        _counter = FindObjectOfType<Counter>();


        _navMeshAgent.updateRotation = false;
    }

    void Update()
    {
        if (dead)
            return;

        _navMeshAgent.SetDestination(_player.transform.position);

        if (_navMeshAgent.velocity.sqrMagnitude > 0.1f)
        {

            Vector3 moveDirection = _navMeshAgent.velocity.normalized;
            moveDirection.y = 0;


            transform.rotation = Quaternion.Slerp(transform.rotation,
Quaternion.LookRotation(moveDirection), Time.deltaTime * 5f);
        }
    }
```

```
    public void Kill()
    {
        if (!dead)
        {
            dead = true;
            _particleSystem.Play();
            _counter.AddKill();
            _capsuleCollider.enabled = false;
            _movement.enabled = false;
            _navMeshAgent.enabled = false;
            Destroy(gameObject, 7.0f);
            _animator.SetTrigger("died");
        }
    }
}
```

```
using UnityEngine;

public class EnemyFactory
{
    private GameObject _enemyPrefab;

    public EnemyFactory(GameObject enemyPrefab)
    {
        _enemyPrefab = enemyPrefab;
    }

    public GameObject CreateEnemy(Vector3 position, Quaternion rotation)
    {
        return GameObject.Instantiate(_enemyPrefab, position, rotation);
    }
}
```

Anexa 2

https://github.com/Akerius682/TMPP_Project cu aplicația creată însoțită de un ReadME file.