МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД «УНІВЕРСИТЕТ «КРОК»

Фаховий коледж Університету «КРОК»

ДИПЛОМНА РОБОТА

за темою

«Розробка гри на платформі Unity»

Студент 4 курсу групи ІПЗ-20к/2 **Керівник дипломної роботи**

_____

(посада керівника)

Кошелєв Дмитро Павлович — (прізвище, ім'я та по-батькові студента)

Добришин Юрій Євгенович — (прізвище, ім'я та по-батькові керівника)

До захисту

(резолюція «До захисту»)

_____ 11.06.2024 _____

(підпис студента) **(дата)** **(підпис викладача)**

Київ, 2024 рік

Анотація

Дипломна робота присвячена розробці гри на Unity та технології їх створення. В роботі досліджуються теоретичні основи створення та проектування ігор, аналізуються необхідні вимоги і навички для цього. В практичній частині представлено проектування архітектури гри, методи і алгоритми створення певних аспектів гри.

Робота має на меті показати унікальність і водночас складність створення ігор.

ЗМІСТ

ВСТУП 4
РОЗДІЛ 1 Архітектура проекту 5
РОЗДІЛ 2 Створення базових функцій гри 7
РОЗДІЛ 3 Створення просунутих функцій 19
РОЗДІЛ 4 Анімації 56
ВИСНОВКИ 61
Кадри з гри 62
СПИСОК ПОСИЛАНЬ 64

ВСТУП

У сучасному світі індустрія відеоігор є однією з найбільш динамічно розвиваючих галузей, яка пропонує значні можливості для творчої реалізації, технічних інновацій та економічного зростання. Розробка відеоігор вимагає поєднання різних аспектів, включаючи програмування, графічний дизайн, звукове оформлення та управління проектами. Одним із найпопулярніших інструментів для створення ігор є Unity — кросплатформенний рушій, який надає широкі можливості для розробників усіх рівнів.

Метою даної дипломної роботи є розробка гри на платформі Unity. Вибір саме цієї платформи зумовлений її універсальністю, потужними інструментами для створення тривимірної та двовимірної графіки, а також активною підтримкою з боку спільноти розробників. Unity дозволяє створювати високоякісні ігри для різних платформ, включаючи ПК, мобільні пристрої та консолі, що робить його ідеальним вибором для реалізації найрізноманітніших ідей.

Ця дипломна робота має на меті не тільки продемонструвати практичні навички розробника у створенні ігор, але й підкреслити важливість інтеграції різних технічних і творчих елементів для створення захоплюючого та якісного кінцевого продукту. Результатом роботи стане готова гра, яка може бути представлена на розгляд громадськості та використана як основа для подальших проектів.

РОЗДІЛ 1 Архітектура проекту

У процесі створення гри постійно виникають дилеми з приводу реалізації тих чи інших аспектів гри. Способів реалізації навіть одного з най простішого елементу гри є безліч. Тому важливо перед початком створенням проекту знайти чіткі відповіді на конкретні запитання (в залежності від самого проекту) наприклад:

1. Яка за розміром буде гра ?
2. На що саме буде спиратися гра ?
3. Що я хочу у ній бачити ?
4. Цільова аудиторія гри ?
5. І т.д.

На такі питання повинні бути відповіді навіть якщо ви працюєте один. Тому завжди перед початком роботи над грою потрібно створювати "Технічні документи" у яких є не тільки відповіді на ці самі питання а і технічна інформація з прогресом який у вас вже є.
Хоч це простий але і водночас потужний інструмент який не раз допоможе спроектувати майбутню гру і не загубитися по дорозі.

Також важливо зазначити що при створенні гри, як і будь якого повноцінного продукту, треба розуміти і знати головні принципи і правила проектування.

- Розуміння, що при створенні будь якого елементу гри треба не просто зробити так щоб воно працювало, а й так що створеній аспект легко було переносити та розширювати. Потрібно не виконувати конкретні задачі, а створити зручний і легко розширюваний інструментарій завдяки якому вже і створювати потрібні вам аспекти. Гра на етапі створення повинна бути легко розширюваною і це заощадить величезну кількість часу.

- Створення у першу чергу фундаментальних і основних елементів а потім вже поверх них додавати щось нове.

В загалі є багато правил але я виділив 2 найголовніших за моїм досвідом. Тому хоч і гра не велика за розміром усі продемонстровані у ній можливості будуть відповідати правилам і реалізовані більш складними шляхами. Для дотримання певної архітектури гри.

РОЗДІЛ 2 Створення базових функцій гри

Створення базових функцій гри є одним з найважливіших етапів у розробці будь-якого ігрового проекту. Цей процес включає в себе розробку основних механік, які визначатимуть, як гравці взаємодіють з ігровим світом, а також забезпечують фундамент для подальшого розвитку ігрових функціональностей. Основні функції гри охоплюють такі аспекти, як керування персонажем, взаємодія з об'єктами, обробка фізики, а також інтерфейс користувача.

1. Керування персонажем

Однією з перших задач у розробці гри є створення системи керування персонажем. Це включає в себе налаштування контролерів для руху, стрибків, атак та інших дій, які можуть виконуватися гравцем. В Unity цей процес зазвичай починається з налаштування Input Manager, де визначаються клавіші та кнопки, які відповідатимуть за певні дії.

2. Взаємодія з об'єктами

Важливим аспектом ігрового процесу є можливість взаємодії персонажа з різними об'єктами у світі гри. Це можуть бути двері, які можна відкрити, предмети, які можна підібрати, або вороги, з якими потрібно боротися. Для реалізації таких взаємодій використовуються колайдери та тригери, а також скрипти, що обробляють події взаємодії.

3. Обробка фізики

Unity надає потужний фізичний рушій, який дозволяє створювати реалістичні взаємодії між об'єктами в грі. Налаштування фізичних властивостей об'єктів, таких як маса, сила тертя та еластичність, дозволяє створювати різноманітні ефекти, від простих зіткнень до складних симуляцій руху. Використання Rigidbody та Physic Material є ключовими аспектами цього процесу.

4. Інтерфейс користувача

Для того щоб гравці могли ефективно взаємодіяти з грою, необхідно створити зручний та інтуїтивно зрозумілий інтерфейс користувача (UI). Це включає в себе створення меню, інформаційних панелей, індикаторів здоров'я та інших елементів, які допомагають гравцеві орієнтуватися у грі та приймати рішення. В Unity для створення UI використовується система Canvas, яка дозволяє розміщувати різноманітні елементи інтерфейсу у 2D просторі.

5. Збереження та завантаження даних

Ще однією важливою функцією є можливість збереження прогресу гравця та його завантаження у майбутньому. Це забезпечує можливість продовжувати гру з того місця, де гравець зупинився. Для цього використовуються системи збереження даних, такі як PlayerPrefs для простих даних, або більш складні рішення, як збереження в файли чи бази даних.

Розробка базових функцій гри є фундаментальним етапом, який закладає основу для подальшого розвитку проекту. Виконання цих задач вимагає глибоких знань у програмуванні, розуміння принципів роботи ігрових рушіїв та творчого підходу до вирішення технічних завдань. З правильно реалізованими базовими функціями, гра отримує міцний фундамент для подальшого розширення та вдосконалення, що дозволяє створити захоплюючий та якісний ігровий досвід для користувачів.

Приклад створення руху у 2D грі який легко налагодити під будь які потреби.

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

```
public class PMovement : MonoBehaviour
{

  private Rigidbody2D rb;
  private BoxCollider2D bc;
  private Animator animation;
  private enum MS {idle, running, jumping, falling, doubleJumping, wallJumping, wallSliding}

  [Header("For Movement")]
  [SerializeField] private float speed = 8f;
  [SerializeField] private float airspeed = 9f;
  private float inputHoririzontal;
  private bool facingRight = true;
  private bool isMoving;

  [Header("For Jumping")]
  [SerializeField] private float jumpForce = 16f;
  [SerializeField] private int jumplim = 2;
  [SerializeField] private LayerMask jg;
  [SerializeField] private int jumpc = 0;

  [Header("For WallJumping")]
  [SerializeField] float wallJumpDirection = -1;
  [SerializeField] Vector2 wallJumpAngle;

  [Header("For Wall")]
  [SerializeField] private LayerMask wallLayer;
  [SerializeField] private Transform wallCheckPoint;
  [SerializeField] private Vector2 wallCheckSize;
  [SerializeField] private float wallSlideSpeed = -3;
  private bool touchingwall;
  private bool WallSliding;
```

```
 void Start()
 {
    rb = gameObject.GetComponent<Rigidbody2D>();
    bc = GetComponent<BoxCollider2D>();
    wallJumpAngle.Normalize();
    animation = GetComponent<Animator>();
 }

 void Update()
 {
    if(Time.time >= PCombat.nextAttackTime && PCombat.blockMoment == false)
    {
       move();
       jump();
       WallJump();
       CheckWall();
       Wallslide();
       UpdateAnimation();
    }


}

void move()
{

   inputHoririzontal = Input.GetAxisRaw("Horizontal");
   if (Grounded())
   {
      rb.velocity = new Vector2(inputHoririzontal * speed, rb.velocity.y);
   }
   else if(!Grounded() && !WallSliding && inputHoririzontal != 0)
```

```
    {
        rb.AddForce(new Vector2(airspeed * inputHoririzontal, 0));
        if (Mathf.Abs(rb.velocity.x)> speed)
        {
            rb.velocity = new Vector2(inputHoririzontal * speed, rb.velocity.y);
        }
    }

    if (inputHoririzontal > 0 && !facingRight)
    {
        Flip();
    }

    if (inputHoririzontal < 0 && facingRight)
    {
        Flip();
    }
}

void jump()
{
    if (Input.GetButtonDown("Jump") && (Grounded() || (++jumpc < jumplim)))
    {
        rb.velocity = new Vector3(rb.velocity.x, jumpForce);
    }

    if (Grounded() )
    {
        jumpc = 0;
    }

}
```

```
 void Flip()
 {
    if (!WallSliding )
    {
       wallJumpDirection *= -1;
       facingRight = !facingRight;
       transform.Rotate(0, 180, 0);
    }


 }

 void Wallslide()
 {


  if (touchingwall && !Grounded() && rb.velocity.y < 0)
  {
      WallSliding = true;
  }
  else
  {
      WallSliding = false;
  }

  if (WallSliding)
  {
      rb.velocity = new Vector2(rb.velocity.x, wallSlideSpeed);
  }

}

void WallJump()
```

```
    {
        if((WallSliding || touchingwall ) && (++jumpc < jumplim) && Input.GetButtonDown("Jump"))
        {
            rb.AddForce(new Vector2(jumpForce * wallJumpDirection * wallJumpAngle.x, jumpForce *
wallJumpAngle.y), ForceMode2D.Impulse);



        }

        if(WallSliding || touchingwall)
        {
            jumpc = 0;
        }



    }

    private void UpdateAnimation()
    {

        MS state;

        if (inputHoririzontal > 0f || inputHoririzontal < 0f)
        {
            state = MS.running;
        }
        else
        {
            state = MS.idle;
        }

        if (rb.velocity.y >.1f)
```

```
        {
            state = MS.jumping;
        }
        else if(rb.velocity.y < -.1f)
        {
            state = MS.falling;
        }

        animation.SetInteger("state", (int)state);

    }


    private bool Grounded()
    {
        return Physics2D.BoxCast(bc.bounds.center, bc.bounds.size, 0f, Vector2.down, 0.2f, jg);
    }

    void CheckWall()
    {
        touchingwall = Physics2D.OverlapBox(wallCheckPoint.position, wallCheckSize, 0, wallLayer);
    }

    private void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.blue;
        Gizmos.DrawCube(wallCheckPoint.position, wallCheckSize);
    }
}
```

На цьому прикладі можливо побачити реалізацію 2D руху стрибків ковзання по стінам і т.д. з прив'язаним аніматором Unity.

Цей скрипт в Unity відповідає за рух персонажа, його анімації та взаємодію зі стінами. Ось коротке пояснення його основних частин:

Методи:

- `Start()`: Ініціалізація компонентів.

- `Update()`: Виклик методів для руху, стрибка, перевірки стін та оновлення анімацій у кожному кадрі.

- `move()`: Рух персонажа, як на землі, так і в повітрі.

- `jump()`: Стрибок персонажа з обмеженням на кількість стрибків.

- `Flip()`: Зміна напряму обличчя персонажа.

- `Wallslide()`: Персонаж може ковзати по стінах.

- `WallJump()`: Стрибок від стіни.

- `UpdateAnimation()`: Оновлення стану анімації в залежності від руху персонажа.

- `Grounded()`: Перевірка, чи персонаж на землі.

- `CheckWall()`: Перевірка, чи персонаж торкається стіни.

Додаткові функції:

- `OnDrawGizmosSelected()`: Відображення гізмо для перевірки контакту зі стіною в редакторі Unity.

Код пострілів гравця

```
using UnityEngine;

public class PlayerAttack : MonoBehaviour
{
  [SerializeField] private float attackCooldown;
```

```
    [SerializeField] private Transform firePoint;
    [SerializeField] private GameObject[] fireballs;
    [SerializeField] private AudioClip fireballSound;

    private Animator anim;
    private PlayerMovement playerMovement;
    private float cooldownTimer = Mathf.Infinity;

    private void Awake()
    {
        anim = GetComponent<Animator>();
        playerMovement = GetComponent<PlayerMovement>();
    }

    private void Update()
    {
        if (Input.GetMouseButton(0) && cooldownTimer > attackCooldown &&
 playerMovement.canAttack()
            && Time.timeScale > 0)
            Attack();

        cooldownTimer += Time.deltaTime;
    }

    private void Attack()
    {
        SoundManager.instance.PlaySound(fireballSound);
        anim.SetTrigger("attack");
        cooldownTimer = 0;

        fireballs[FindFireball()].transform.position = firePoint.position;

fireballs[FindFireball()].GetComponent<Projectile>().SetDirection(Mathf.Sign(transform.localScale.
x));
    }
    private int FindFireball()
    {
        for (int i = 0; i < fireballs.Length; i++)
        {
            if (!fireballs[i].activeInHierarchy)
                return i;
        }
        return 0;
    }
}
```

Методи :

`Awake()`

Метод, який викликається при ініціалізації об'єкта:

Він знаходить і зберігає посилання на компоненти Animator та PlayerMovement.

`Update()`

Метод, який викликається кожен кадр:

Цей метод перевіряє, чи натиснута ліва кнопка миші, чи минув достатній час з моменту останньої атаки, чи може гравець атакувати і чи не поставлена гра на паузу. Якщо всі умови виконані, викликається метод `Attack()`. Потім збільшується таймер кулдауну.

`Attack()`

Метод, який виконує атаку:

Цей метод виконує наступні дії:

1. Відтворює звук атаки.

2. Запускає анімацію атаки.

3. Скидає таймер кулдауну.

4. Знаходить вільну вогняну кулю, встановлює її позицію на `firePoint` і задає напрямок руху.

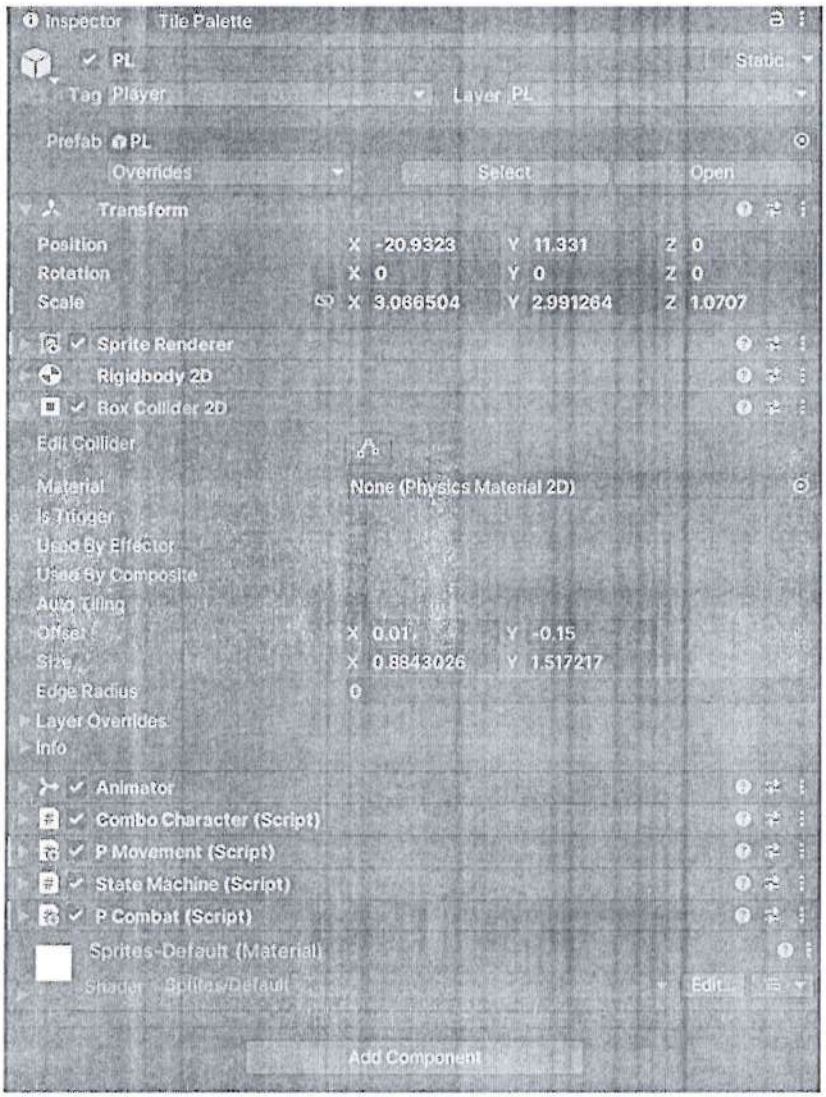
`FindFireball()`

Метод для пошуку неактивної вогняної кулі в масиві:

Цей метод перебирає масив `fireballs` і повертає індекс першої неактивної вогняної кулі. Якщо всі кулі активні, повертає 0.

Фізика та інші аспекти у Unity

Усі інші основні функції гри реалізуються у самому редакторі Unity. Усім елементам і об'єктам можливо призначити фізику, хітбокс, колайдер, спрайт, інші створенні ваші скрипти і т.д., та налагодити їх під свої потреби.



На прикладі можливо побачити інспектор у якому показані усі компоненти які належать персонажу гравця і їх параметри які можна змінювати.

РОЗДІЛ 3 Створення просунутих функцій

Game Manager

Game Manager – це тип файлу у Unity яких дозволяє керувати і слідкувати за глобальними процесами у грі. Такі елементи як: здоров'я гравця та ворогів, глобальні події для яких тригером є якась умова, іншу інформацію про деякі об'єкти до якої часто посилаються і т.д. краще і зручніше тригери цих подій розміщувати саме у цьому скрипті. (саме тригери а не реалізацію)

Приклад як реалізовано здоров'я гравця і ворогів:

Кусок коду з Game Manager який відповідає за здоров'я.

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

public class GameManager : MonoBehaviour
{

    public static GameManager gameManager { get; private set; }

    public UnitHealth playerHealth = new UnitHealth(100, 100);

    public UnitHealth crowHealth = new UnitHealth(60,60);

    public UnitHealth dogHealth = new UnitHealth(90, 90);

    void Awake()
    {
```

```
    public UnitHealth crowHealth = new UnitHealth(60,60);

    public UnitHealth dogHealth = new UnitHealth(90, 90);

    void Awake()
    {
        if (gameManager != null && gameManager != this)
        {
            Destroy(this);
        }
        else
        {
            gameManager = this;
        }
    }
```

Продовження реалізації здоров'я і нанесення пошкоджень в спеціальному для цього скрипту.

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

public class UnitHealth
{
    int currentHealth;
    int currentMaxHealth;

    public int Health
    {
```

```
    get
    {
        return currentHealth;
    }
    set
    {
        currentHealth = value;
    }
}

public int MaxHealth
{
    get
    {
        return currentMaxHealth;
    }
    set
    {
        currentMaxHealth = value;
    }
}

public UnitHealth(int health, int maxHealth)
{
    currentHealth = health;
    currentMaxHealth = maxHealth;
}

public void DmgUnit(int dmgAmount)
{
    if(currentHealth > 0)
    {
        currentHealth -= dmgAmount;
```

```
            Debug.Log(currentHealth);
        }
    }

    public void HealUnit(int healAmount)
    {
        if (currentHealth < currentMaxHealth)
        {
            currentHealth += healAmount;
        }
        if(currentHealth > currentMaxHealth)
        {
            currentHealth = currentMaxHealth;
        }


    }



}
```

Далі для кожного окремого об'єкта вже призначається його власна логіка пов'язана з здоров'ям.

Приклад для гравця:

```
using UnityEngine.SceneManagement;

public class PlayerBehaviour : MonoBehaviour
{
```

```
[SerializeField] HP_Bar healthbar;

private Rigidbody2D rb;
private Animator animation;



void Start()
{
    rb = GetComponent<Rigidbody2D>();
    animation = GetComponent<Animator>();
}



void Update()
{
    healthbar.SetHealth(GameManager.gameManager.playerHealth.Health);

    if (GameManager.gameManager.playerHealth.Health <= 0)
    {
        Die();
    }
}


public void PlayerTakeDmg(int dmg)
{
    GameManager.gameManager.playerHealth.DmgUnit(dmg);

}

private void PlayerHeal(int healing)
```

```
{
    GameManager.gameManager.playerHealth.HealUnit(healing);
    healthbar.SetHealth(GameManager.gameManager.playerHealth.Health);
}

private void Die()
{
    Debug.Log("PL died!");
    rb.bodyType = RigidbodyType2D.Static;
    animation.SetTrigger("death");
    RestartGame();
}
```

Модель State Machine

Модель State Machine (машина станів) — це концепція в програмуванні, яка використовується для управління поведінкою об'єктів, що можуть перебувати в різних станах і змінювати ці стани у відповідь на певні події.

Суть моделі полягає у контролі станів об’єкта. це модель, що використовується в програмуванні для управління станами об'єкта або системи. У State machine об'єкт може перебувати в одному з певного набору станів, причому перехід між станами відбувається відповідно до певних умов або подій.

State machine допомагає структурувати та управляти складними системами, особливо там, де об'єкт може перебувати в різних станах та поводитися по-різному в залежності від свого стану. Вона дозволяє реалізувати зручну та легко змінювану логіку переходів між станами.

У грі вона відповідає за:

- Контроль станів AI.
- Керування камерою.
- Відстежування порядку атак.

Код статусів

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

public abstract class State
{

  protected float time { get; set; }

  protected float fixedtime { get; set; }

  protected float latetime { get; set; }

  public StateMachine stateMachine;

  public virtual void OnEnter(StateMachine _stateMachine)
  {
     stateMachine = _stateMachine;
  }


  public virtual void OnUpdate()
  {
     time += Time.deltaTime;
     fixedtime += Time.deltaTime;
  }

  public virtual void OnFixedUpdate()
  {
    fixedtime += Time.deltaTime;
  }

  public virtual void OnLateUpdate()
  {
     latetime += Time.deltaTime;
  }
```

```
    public virtual void OnExit()
    {

    }

    #region Passthrought Methods

    protected static void Destroy(UnityEngine.Object obj)
    {
        UnityEngine.Object.Destroy(obj);
    }

    protected T GetComponent<T>() where T : Component { return
stateMachine.GetComponent<T>(); }

    protected Component GetComponent(System.Type type) { return
stateMachine.GetComponent(type); }

    protected Component GetComponent(string type) { return stateMachine.GetComponent(type);
}

    #endregion
}
```

Даний код представляє абстрактний клас `State`, який є одним із станів у State machine. Давайте розглянемо деякі елементи цього коду на прикладі методу `OnEnter`:

```
public virtual void OnEnter(StateMachine _stateMachine)

{

    stateMachine = _stateMachine;

}
```

У цьому методі `OnEnter` відбувається вхід у даний стан. Параметр `_stateMachine` представляє посилання на об'єкт `StateMachine`, який управляє State machine. При виклику цього методу об'єкт `stateMachine` отримує посилання на об'єкт `StateMachine`, що дозволяє взаємодіяти з іншими станами та контролювати процеси в State machine.

Наприклад, якщо потрібно здійснити перехід до іншого стану з методу `OnEnter`, можна скористатися методом `SetNextState` з об'єкта `stateMachine`, який буде встановлювати наступний стан для State machine:

```
public virtual void OnEnter(StateMachine _stateMachine)
{
    stateMachine = _stateMachine;
    stateMachine.SetNextState(new NextState());
}
```

Це лише один із способів використання State machine для управління станами та їх взаємодією. Код дозволяє реалізувати логіку станів та їх переходів відповідно до специфіки конкретної гри або програми.

Метод OnExit потрібен для виходу з стану і знешкодження його залишків.

Код у region потрібен для взаємодії з внутрішніми методами і компонентами у середовищі Unity.

Код самої машини

```
using UnityEngine;

public class StateMachine : MonoBehaviour
{
  public string customName;

  private State mainStateType;

  public State CurrentState {  get; private set; }
  private State nextState;


  void Update()
  {
```

```
        if (nextState != null)
        {
            SetState(nextState);
            nextState = null;
        }

        if(CurrentState != null)
            CurrentState.OnUpdate();
    }
    private void SetState(State _newState)
    {
        if(CurrentState != null)
        {
            CurrentState.OnExit();
        }
        CurrentState = _newState;
        CurrentState.OnEnter(this);
    }
    public void SetNextState(State _newState)
    {

        if(_newState != null)
        {
            nextState = _newState;
        }

    }
    public void SetNextStateToMain()
    {
        nextState = mainStateType;
    }

    private void Awake()
    {
        SetNextStateToMain();
    }

    private void OnValidate()
    {
        if (mainStateType == null)
        {
            if(customName == "Combat")
            {
                mainStateType = new IdleCombatState();
            }
        }
    }
}
```

Даний код реалізує просту State machine для управління станами об'єкта у грі:

1. Клас StateMachine: Цей клас відповідає за управлінням станами об'єкта. Він має поля `customName` (для вказівки спеціального імені стану) та `mainStateType` (основний стан).

2. Метод Update: Цей метод викликається кожен кадр гри і перевіряє, чи не було встановлено нового стану. Якщо було, то він викликає метод `SetState` для зміни поточного стану.

3. Метод SetState: Цей метод встановлює новий стан для State machine. Він викликає методи `OnExit` поточного стану, потім встановлює новий стан і викликає його метод `OnEnter`.

4. Метод SetNextState: Цей метод встановлює наступний стан для State machine. Він використовується для планування зміни стану на наступному кадрі гри.

5. Метод SetNextStateToMain: Цей метод встановлює основний стан як наступний стан для State machine.

6. Метод Awake: Цей метод викликається при створенні об'єкта. У ньому викликається метод `SetNextStateToMain`, щоб встановити основний стан за замовчуванням.

7. Метод OnValidate: Цей метод викликається в редакторі Unity, коли значення у скрипта налаштовуються або змінюються. У цьому випадку, якщо `mainStateType` не встановлено, то він встановлюється в залежності від значення `customName`.

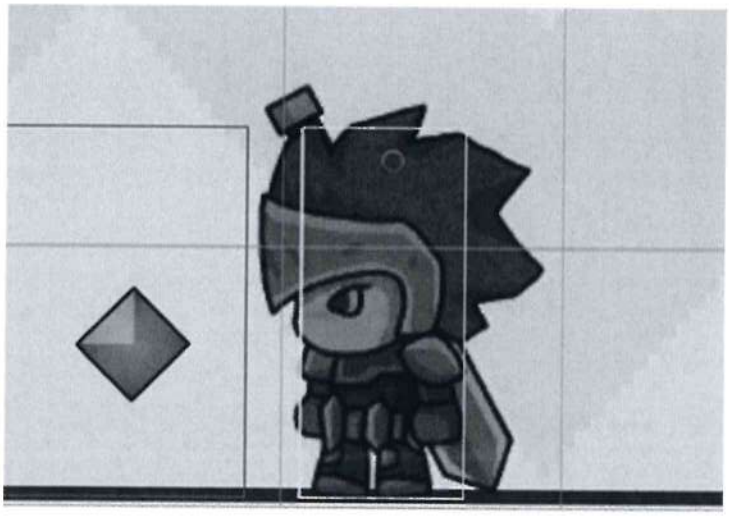
Код демонструє базову реалізацію State machine, яка може бути розширена та адаптована для конкретних потреб гри. Він дозволяє легко керувати станами об'єкта та здійснювати переходи між ними відповідно до логіки гри.

Ці 2 скрипта є основними компонентами які використовуються для зазначених вище функцій. Сама логіка і реалізація станів знаходиться вже безпосередньо у скриптах самих об'єктів.

Скрипти для AI

Не варто забувати що майже у кожній грі є вороги зі своїм власним інтелектом. У грі продемонстровано декілька видів ворогів 1 простий і інші 2 більш ускладнених.

Простий:



```
using UnityEngine;

public class RangedEnemy : MonoBehaviour
```

```
{
    [Header("Attack Parameters")]
    [SerializeField] private float attackCooldown;
    [SerializeField] private float range;
    [SerializeField] private int damage;

    [Header("Ranged Attack")]
    [SerializeField] private Transform firepoint;
    [SerializeField] private GameObject[] fireballs;

    [Header("Collider Parameters")]
    [SerializeField] private float colliderDistance;
    [SerializeField] private BoxCollider2D boxCollider;

    [Header("Player Layer")]
    [SerializeField] private LayerMask playerLayer;
    private float cooldownTimer = Mathf.Infinity;

    [Header("Fireball Sound")]
    [SerializeField] private AudioClip fireballSound;

    //References
    private Animator anim;
    private EnemyPatrol enemyPatrol;

    private void Awake()
    {
        anim = GetComponent<Animator>();
        enemyPatrol = GetComponentInParent<EnemyPatrol>();
    }

    private void Update()
    {
```

```
        cooldownTimer += Time.deltaTime;

        //Attack only when player in sight?
        if (PlayerInSight())
        {
            if (cooldownTimer >= attackCooldown)
            {
                cooldownTimer = 0;
                anim.SetTrigger("rangedAttack");
            }
        }

        if (enemyPatrol != null)
            enemyPatrol.enabled = !PlayerInSight();
    }

    private void RangedAttack()
    {
        SoundManager.instance.PlaySound(fireballSound);
        cooldownTimer = 0;
        fireballs[FindFireball()].transform.position = firepoint.position;
        fireballs[FindFireball()].GetComponent<EnemyProjectile>().ActivateProjectile();
    }
    private int FindFireball()
    {
        for (int i = 0; i < fireballs.Length; i++)
        {
            if (!fireballs[i].activeInHierarchy)
                return i;
        }
        return 0;
    }
```

```
    private bool PlayerInSight()
    {
        RaycastHit2D hit =
            Physics2D.BoxCast(boxCollider.bounds.center + transform.right * range *
transform.localScale.x * colliderDistance,
            new Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y,
boxCollider.bounds.size.z),
            0, Vector2.left, 0, playerLayer);

        return hit.collider != null;
    }
    private void OnDrawGizmos()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireCube(boxCollider.bounds.center + transform.right * range *
transform.localScale.x * colliderDistance,
            new Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y,
boxCollider.bounds.size.z));
    }
}
```

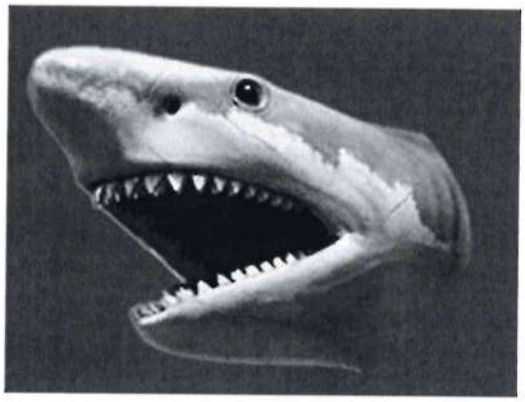
Суть цього супротивника у тому, що коли гравець підходить у певний радіус дії ворога то він починає по ньому стріляти.

Це показаний найпростіший спосіб створення загального скрипту для ворожого AI

Складний:

Інший спосіб передбачає використання State Machine і Pathfinding

Використання 1 зумовлено тим що через цей метод зручно переключати лінії поведінки AI в залежності від умови а використання 2 нам потрібно щоб знаходити гравця та вистроювати маршрути до гравця для AI.



Бійтесь його він не тільки літати вміє тай ще коробками стріляти

Кусок коду який показує використання скрипту для знаходження шляху.

```
public  Transform target;

public float speed = 400f;
public float nextWaypointDistance = 3f;
public float minimumDistance = 10;
public float RetreatDistance = 7;

[SerializeField] public  GameObject projectile;
[SerializeField] public GameObject projectile4;

public float timeBetweenShots = 2;
private float nextShootTime = 1;
private int shootRange = 12;
```

```
[SerializeField] private int attackCountCurent = 0;

Path path;
int currentWaypoint = 0;
bool reachedEndOfPath = false;

Seeker seeker;
Rigidbody2D rb;


void Start()
{
    seeker = GetComponent<Seeker>();
    rb = GetComponent<Rigidbody2D>();

    InvokeRepeating("UpdatePath", 0f, .5f);

}

void UpdatePath()
{
    if (seeker.IsDone())
    {
        seeker.StartPath(rb.position, target.position, OnPathComplete);
    }

}

void OnPathComplete(Path p)
{
    if (!p.error)
    {
```

```
        path = p;
        currentWaypoint = 0;
    }
}
```

Головна логіка AI

```
if (attackCountCurent == 4)
{
    attackCountCurent = 1;
}


if (Time.time > nextShootTime && shootRange > Vector2.Distance(rb.position, target.position))
{
    if (attackCountCurent == 3)
    {
        Instantiate(projectile4, transform.position, Quaternion.identity);
        nextShootTime = Time.time + timeBetweenShots;
        attackCountCurent++;
    }
    else
    {
        Instantiate(projectile, transform.position, Quaternion.identity);
        nextShootTime = Time.time + timeBetweenShots;
        attackCountCurent++;
    }

}
```

```
 if (Vector2.Distance(rb.position, target.position) > minimumDistance)
 {
    if (path == null)
       return;

    if (currentWaypoint >= path.vectorPath.Count)
    {
       reachedEndOfPath = true;
       return;
    }
    else
    {
       reachedEndOfPath = false;
    }

    Vector2 direction = ((Vector2)path.vectorPath[currentWaypoint] - rb.position).normalized;
    Vector2 force = direction * speed * Time.deltaTime;

    rb.AddForce(force);

    float distance = Vector2.Distance(rb.position, path.vectorPath[currentWaypoint]);

    if (distance < nextWaypointDistance)
    {
       currentWaypoint++;
    }
}
else if(Vector2.Distance(transform.position, target.position) < RetreatDistance)
{
   transform.position = Vector2.MoveTowards(rb.position, target.position, (-speed/100) *
Time.deltaTime);
}
```

Головний скрипт для знаходження шляху

```
using UnityEngine;
using System.Collections.Generic;
#if UNITY_5_5_OR_NEWER
using UnityEngine.Profiling;
#endif

namespace Pathfinding {

    [AddComponentMenu("Pathfinding/Seeker")]
    [HelpURL("http://arongranberg.com/astar/docs/class_pathfinding_1_1_seeker.php")]
    public class Seeker : VersionedMonoBehaviour {

        public bool drawGizmos = true;


        public bool detailedGizmos;


        [HideInInspector]
        public StartEndModifier startEndModifier = new StartEndModifier();


        [HideInInspector]
        public int traversableTags = -1;


        [HideInInspector]
        public int[] tagPenalties = new int[32];


        [HideInInspector]
```

```
public GraphMask graphMask = GraphMask.everything;

[UnityEngine.Serialization.FormerlySerializedAs("graphMask")]
int graphMaskCompatibility = -1;

public OnPathDelegate pathCallback;

public OnPathDelegate preProcessPath;

public OnPathDelegate postProcessPath;

[System.NonSerialized]
List<Vector3> lastCompletedVectorPath;

[System.NonSerialized]
List<GraphNode> lastCompletedNodePath;

[System.NonSerialized]
protected Path path;

[System.NonSerialized]
private Path prevPath;

private readonly OnPathDelegate onPathDelegate;
```

```
private OnPathDelegate tmpPathCallback;


protected uint lastPathID;


readonly List<IPathModifier> modifiers = new List<IPathModifier>();

public enum ModifierPass {
        PreProcess,

        PostProcess = 2,
}

public Seeker () {
        onPathDelegate = OnPathComplete;
}


protected override void Awake () {
        base.Awake();
        startEndModifier.Awake(this);
}


public Path GetCurrentPath () {
        return path;
}
```

```
        /// <param name="pool">If true then the path will be pooled when the pathfinding
system is done with it.</param>
        public void CancelCurrentPathRequest (bool pool = true) {
            if (!IsDone()) {
                path.FailWithError("Canceled by script
(Seeker.CancelCurrentPathRequest)");
                if (pool) {

                    path.Claim(path);
                    path.Release(path);
                }
            }
        }


        /// See: <see cref="ReleaseClaimedPath"/>
        /// See: <see cref="startEndModifier"/>

        public void OnDestroy () {
            ReleaseClaimedPath();
            startEndModifier.OnDestroy(this);
        }


        void ReleaseClaimedPath () {
            if (prevPath != null) {
                prevPath.Release(this, true);
                prevPath = null;
            }
        }


        public void RegisterModifier (IPathModifier modifier) {
```

```
            modifiers.Add(modifier);


            modifiers.Sort((a, b) => a.Order.CompareTo(b.Order));
        }


        public void DeregisterModifier (IPathModifier modifier) {
            modifiers.Remove(modifier);
        }


        public void PostProcess (Path path) {
            RunModifiers(ModifierPass.PostProcess, path);
        }


        public void RunModifiers (ModifierPass pass, Path path) {
            if (pass == ModifierPass.PreProcess) {
                if (preProcessPath != null) preProcessPath(path);

                for (int i = 0; i < modifiers.Count; i++) modifiers[i].PreProcess(path);
            } else if (pass == ModifierPass.PostProcess) {
                Profiler.BeginSample("Running Path Modifiers");

                if (postProcessPath != null) postProcessPath(path);


                for (int i = 0; i < modifiers.Count; i++) modifiers[i].Apply(path);
                Profiler.EndSample();
            }
        }
```

```
public bool IsDone () {
        return path == null || path.PipelineState >= PathState.Returned;
}


void OnPathComplete (Path path) {
        OnPathComplete(path, true, true);
}


void OnPathComplete (Path p, bool runModifiers, bool sendCallbacks) {
        if (p != null && p != path && sendCallbacks) {
                return;
        }

        if (this == null || p == null || p != path)
                return;

        if (!path.error && runModifiers) {

                RunModifiers(ModifierPass.PostProcess, path);
        }

        if (sendCallbacks) {
                p.Claim(this);

                lastCompletedNodePath = p.path;
                lastCompletedVectorPath = p.vectorPath;


                if (tmpPathCallback != null) {
                        tmpPathCallback(p);
```

```
                }


                if (pathCallback != null) {
                        pathCallback(p);
                }


                        prevPath.Release(this, true);
                }

                prevPath = p;
        }
}


[System.Obsolete("Use ABPath.Construct(start, end, null) instead")]
public ABPath GetNewPath (Vector3 start, Vector3 end) {
        // Construct a path with start and end points
        return ABPath.Construct(start, end, null);
}


/// <param name="start">The start point of the path</param>
/// <param name="end">The end point of the path</param>
public Path StartPath (Vector3 start, Vector3 end) {
        return StartPath(start, end, null);
}


/// <param name="start">The start point of the path</param>
/// <param name="end">The end point of the path</param>
```

```
        /// <param name="callback">The function to call when the path has been
calculated</param>
        public Path StartPath (Vector3 start, Vector3 end, OnPathDelegate callback) {
            return StartPath(ABPath.Construct(start, end, null), callback);
        }

        /
        /// <param name="start">The start point of the path</param>
        /// <param name="end">The end point of the path</param>
        /// <param name="callback">The function to call when the path has been
calculated</param>
        /// <param name="graphMask">Mask used to specify which graphs should be
searched for close nodes. See #Pathfinding.NNConstraint.graphMask. This will override
#graphMask for this path request.</param>
        public Path StartPath (Vector3 start, Vector3 end, OnPathDelegate callback,
GraphMask graphMask) {
            return StartPath(ABPath.Construct(start, end, null), callback, graphMask);
        }


        /// <param name="p">The path to start calculating</param>
        /// <param name="callback">The function to call when the path has been
calculated</param>
        public Path StartPath (Path p, OnPathDelegate callback = null) {

            if (p.nnConstraint.graphMask == -1) p.nnConstraint.graphMask = graphMask;
            StartPathInternal(p, callback);
            return p;
        }


        /// <param name="p">The path to start calculating</param>
```

```
        /// <param name="callback">The function to call when the path has been
calculated</param>
        /// <param name="graphMask">Mask used to specify which graphs should be
searched for close nodes. See #Pathfinding.GraphMask. This will override #graphMask for this
path request.</param>
        public Path StartPath (Path p, OnPathDelegate callback, GraphMask graphMask) {
            p.nnConstraint.graphMask = graphMask;
            StartPathInternal(p, callback);
            return p;
        }


            p.callback += onPathDelegate;

            p.enabledTags = traversableTags;
            p.tagPenalties = tagPenalties;


            if (path != null && path.PipelineState <= PathState.Processing &&
path.CompleteState != PathCompleteState.Error && lastPathID == path.pathID) {
                path.FailWithError("Canceled path because a new one was
requested.\n"+
                    "This happens when a new path is requested from the seeker
when one was already being calculated.\n" +
                    "For example if a unit got a new order, you might request a
new path directly instead of waiting for the now" +
                    " invalid path to be calculated. Which is probably what you
want.\n" +
                    "If you are getting this a lot, you might want to consider how
you are scheduling path requests.");

            }
```

```
            path = p;
            tmpPathCallback = callback;


            lastPathID = path.pathID;


            RunModifiers(ModifierPass.PreProcess, path);


            AstarPath.StartPath(path);
        }



        public void OnDrawGizmos () {
            if (lastCompletedNodePath == null || !drawGizmos) {
                return;
            }

            if (detailedGizmos) {
                Gizmos.color = new Color(0.7F, 0.5F, 0.1F, 0.5F);

                if (lastCompletedNodePath != null) {
                    for (int i = 0; i < lastCompletedNodePath.Count-1; i++) {

Gizmos.DrawLine((Vector3)lastCompletedNodePath[i].position,
(Vector3)lastCompletedNodePath[i+1].position);
                    }
                }
            }
```

```
                Gizmos.color = new Color(0, 1F, 0, 1F);

                if (lastCompletedVectorPath != null) {
                    for (int i = 0; i < lastCompletedVectorPath.Count-1; i++) {
                        Gizmos.DrawLine(lastCompletedVectorPath[i],
lastCompletedVectorPath[i+1]);
                    }
                }
            }

            protected override int OnUpgradeSerializedData (int version, bool unityThread) {
                if (graphMaskCompatibility != -1) {
                    Debug.Log("Loaded " + graphMaskCompatibility + " " +
graphMask.value);
                    graphMask = graphMaskCompatibility;
                    graphMaskCompatibility = -1;
                }
                return base.OnUpgradeSerializedData(version, unityThread);
            }
        }
}
```

Цей код представляє собою клас `Seeker`, який є частиною системи пошуку шляхів для ігрового рушія Unity. `Seeker` відповідає за пошук оптимальних шляхів для персонажів або об'єктів у грі. Ось короткий опис роботи ключових частин коду:

Основні частини коду:

1. Класи і простори імен:

- Клас `Seeker` належить до простору імен `Pathfinding` і використовує Unity-специфічні простори імен, такі як `UnityEngine`.

2. Поля і властивості:

- `drawGizmos` і `detailedGizmos`: Булеві змінні для контролю відображення Gizmos у редакторі Unity.

- `startEndModifier`: Модифікатор для налаштування початкової і кінцевої точок шляху.

- `traversableTags`, `tagPenalties`, `graphMask`: Параметри, що визначають, які теги і графи будуть враховуватись при пошуку шляху.

- `pathCallback`, `preProcessPath`, `postProcessPath`: Делегати для виклику функцій після завершення або під час обробки шляху.

- `lastCompletedVectorPath`, `lastCompletedNodePath`: Зберігають останній завершений шлях у вигляді векторів і вузлів.

- `path`, `prevPath`: Поточний і попередній шляхи, які обробляються.

- `onPathDelegate`: Делегат для обробки завершеного шляху.

- `modifiers`: Список модифікаторів для обробки шляху.

3. Методи:

- `Awake()`: Ініціалізація `Seeker` при запуску.

- `GetCurrentPath()`: Повертає поточний шлях.

- `CancelCurrentPathRequest()`: Скасовує поточний запит на пошук шляху.

- `OnDestroy()`: Виконується при знищенні об'єкта, звільняє ресурси.

- `RegisterModifier()`, `DeregisterModifier()`: Додають або видаляють модифікатори зі списку.

- `PostProcess()`, `RunModifiers()`: Виконують обробку шляху за допомогою модифікаторів.

- `IsDone()`: Перевіряє, чи завершено обробку поточного шляху.

- `OnPathComplete()`: Обробляє завершення шляху.

- `GetNewPath()`, `StartPath()`: Створюють і запускають новий шлях.

- `OnDrawGizmos()`: Малює Gizmos для відображення шляху у редакторі Unity.

Основний процес роботи:

1. Ініціалізація: При створенні об'єкта `Seeker`, він ініціалізується і налаштовується для обробки шляхів.

2. Запит на пошук шляху: Коли потрібно знайти шлях, викликається метод `StartPath()`, який створює новий шлях і запускає його обробку.

3. Обробка шляху: Під час обробки шляху використовуються модифікатори і делегати для налаштування і корекції шляху.

4. Завершення шляху: Після завершення обробки шляху викликається `OnPathComplete()`, що виконує необхідні дії, такі як виклик колбеків і звільнення ресурсів.

5. Відображення шляху: Якщо увімкнено `drawGizmos`, шлях відображається у редакторі Unity за допомогою `OnDrawGizmos()`.

Цей клас є частиною більшої системи для роботи з пошуком шляхів і забезпечує основні функціональні можливості для керування і обробки шляхів у ігровому середовищі Unity.

Скрипти інтерфейсу і інших зв'язних функцій

Скрипт для меню паузы

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class UIManager : MonoBehaviour
{
  [Header ("Game Over")]
  [SerializeField] private GameObject gameOverScreen;
  [SerializeField] private AudioClip gameOverSound;

  [Header("Pause")]
  [SerializeField] private GameObject pauseScreen;

  private void Awake()
  {
    gameOverScreen.SetActive(false);
    pauseScreen.SetActive(false);
  }
  private void Update()
  {
    if (Input.GetKeyDown(KeyCode.Escape))
    {

      PauseGame(!pauseScreen.activeInHierarchy);
    }
  }

  #region Game Over

  public void GameOver()
  {
    gameOverScreen.SetActive(true);
    SoundManager.instance.PlaySound(gameOverSound);
  }


  public void Restart()
  {
```

```
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }


    public void MainMenu()
    {
        SceneManager.LoadScene(0);
    }


    public void Quit()
    {
        Application.Quit();

#if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false; //Exits play mode (will only be executed in the
editor)
#endif
    }
    #endregion

    #region Pause
    public void PauseGame(bool status)
    {

        pauseScreen.SetActive(status);


        if (status)
            Time.timeScale = 0;
        else
            Time.timeScale = 1;
    }
    public void SoundVolume()
    {
        SoundManager.instance.ChangeSoundVolume(0.2f);
    }
    public void MusicVolume()
    {
        SoundManager.instance.ChangeMusicVolume(0.2f);
    }
    #endregion
}
```

Цей скрипт `UIManager` керує інтерфейсом користувача в грі, зокрема екранами завершення гри та паузи, а також обробляє звуки.

Основний процес роботи:

1. Ініціалізація: В методі `Awake()` екрани завершення гри та паузи деактивуються, щоб вони не були видимими при запуску гри.

2. Перевірка паузи: В методі `Update()` перевіряється натискання клавіші Escape, щоб перемикати стан паузи.

3. Завершення гри: Виклик методу `GameOver()` активує екран завершення гри та відтворює відповідний звук.

4. Перезапуск гри: Метод `Restart()` перезавантажує поточну сцену.

5. Перехід до головного меню: Метод `MainMenu()` завантажує сцену з індексом 0, яка є головним меню.

6. Вихід з гри: Метод `Quit()` завершує додаток або виходить з режиму відтворення в редакторі Unity.

7. Управління паузою: Метод `PauseGame()` активує або деактивує екран паузи та зупиняє або відновлює час у грі за допомогою `Time.timeScale`.

8. Зміна гучності: Методи `SoundVolume()` і `MusicVolume()` використовують `SoundManager` для зміни гучності звукових ефектів та музики.

Налаштування звуку

```
using UnityEngine;
using UnityEngine.UI;

public class VolumeText : MonoBehaviour
{
  [SerializeField] private string volumeName;
  [SerializeField] private string textIntro; //Sound:  or Music:
  private Text txt;

  private void Awake()
  {
    txt = GetComponent<Text>();
```

```
    }
    private void Update()
    {
        UpdateVolume();
    }
    private void UpdateVolume()
    {
        float volumeValue = PlayerPrefs.GetFloat(volumeName) * 100;
        txt.text = textIntro + volumeValue.ToString();
    }
}
```

Графічний курсор у меню

```
using UnityEngine;
using UnityEngine.UI;

public class SelectionArrow : MonoBehaviour
{
    [SerializeField] private RectTransform[] buttons;
    [SerializeField] private AudioClip changeSound;
    [SerializeField] private AudioClip interactSound;
    private RectTransform arrow;
    private int currentPosition;

    private void Awake()
    {
        arrow = GetComponent<RectTransform>();
    }
    private void OnEnable()
    {
        currentPosition = 0;
        ChangePosition(0);
    }
    private void Update()
    {

        if (Input.GetKeyDown(KeyCode.UpArrow) || Input.GetKeyDown(KeyCode.W))
            ChangePosition(-1);
        if (Input.GetKeyDown(KeyCode.DownArrow) || Input.GetKeyDown(KeyCode.S))
            ChangePosition(1);


        if (Input.GetKeyDown(KeyCode.KeypadEnter) || Input.GetKeyDown(KeyCode.E))
            Interact();
    }
```

```
    private void ChangePosition(int _change)
    {
        currentPosition += _change;

        if (_change != 0)
            SoundManager.instance.PlaySound(changeSound);

        if (currentPosition < 0)
            currentPosition = buttons.Length - 1;
        else if (currentPosition > buttons.Length - 1)
            currentPosition = 0;

        AssignPosition();
    }
    private void AssignPosition()
    {

        arrow.position = new Vector3(arrow.position.x, buttons[currentPosition].position.y);
    }
    private void Interact()
    {
        SoundManager.instance.PlaySound(interactSound);


        buttons[currentPosition].GetComponent<Button>().onClick.Invoke();
    }
}
```

Основний процес роботи:

1. Ініціалізація: В методі `Awake()` ініціалізується компонент `RectTransform`, що представляє стрілку.

2. Активація: В методі `OnEnable()` стрілка встановлюється на початкову позицію.

3. Оновлення: В методі `Update()` обробляються натискання клавіш для переміщення стрілки та взаємодії з кнопкою.

4. Зміна позиції: Метод `ChangePosition()` змінює поточну позицію стрілки, відтворює звук зміни позиції та оновлює її на екрані.

5. Взаємодія: Метод `Interact()` відтворює звук взаємодії та викликає метод `onClick` вибраної кнопки.

Цей скрипт забезпечує управління стрілкою вибору в меню, дозволяючи гравцеві переміщатися між кнопками та взаємодіяти з ними за допомогою клавіатури.

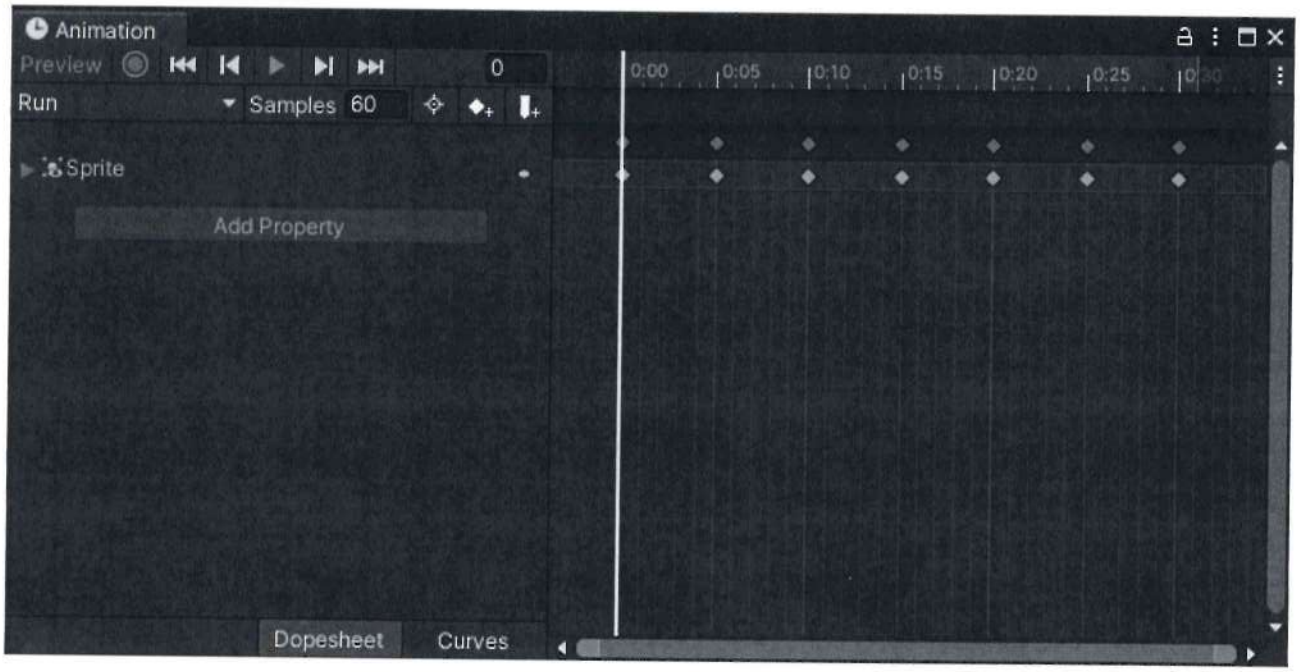
РОЗДІЛ 4 Анімації

Анімації є невід'ємною частиною розробки ігор, оскільки вони додають динаміки та жвавості персонажам і об'єктам. Unity пропонує потужні інструменти для створення та контролю анімацій, що дозволяє розробникам реалізувати свої ідеї з максимальною точністю.

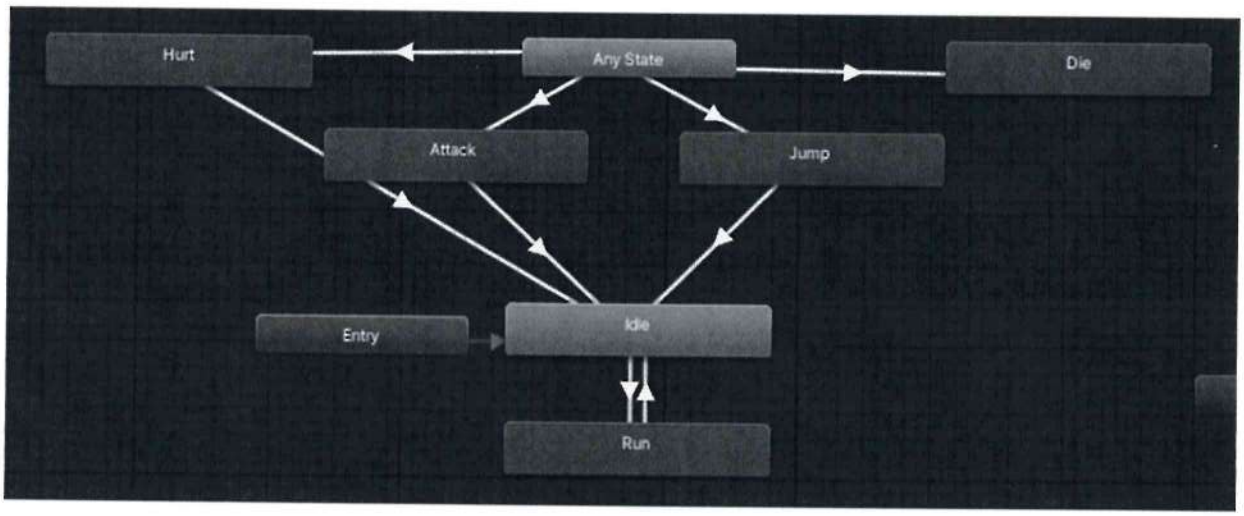
Основні Компоненти Анімацій у Unity

1. Animator Controller: це основний компонент для управління анімаціями в Unity. Він дозволяє створювати стан машини для анімацій, визначати переходи між анімаціями та встановлювати умови для цих переходів.

2. Animation Clips: це окремі анімації, які можна використовувати в Animator Controller. Кожен кліп представляє собою набір ключових кадрів, які визначають зміну параметрів об'єкта у часі (позиція, ротація, масштаб, інші властивості).



3. Animator: компонент, який додається до об'єкта і відповідає за відтворення анімацій, визначених у Animator Controller. Animator прив'язується до конкретного Animator Controller і дозволяє керувати анімаціями через скрипти.



Анімації для гри були як і створені власноруч так і запозиченні з безкоштовних комплектів а потім завдяки коду і інструментам Unity були запроваджені у гру.

РОЗДІЛ 5 Аналіз складності створення ігор і особливості цього напрямку

Ігрова індустрія є однією з найскладніших та най найдинамічніших, яка вимагає від розробників широкого спектру знань, навичок та ресурсів. Створення гри є багатогранним процесом, що включає в себе технічні, художні та управлінські аспекти. Аналіз складності створення ігор та особливостей цього напрямку допомагає краще зрозуміти виклики, з якими стикаються розробники, а також визначити ключові фактори успіху в цій сфері.

Технічні аспекти

Однією з найбільших складнощів у створенні ігор є технічна сторона процесу. Розробники повинні володіти знаннями в таких галузях, як програмування, математика, фізика та комп'ютерна графіка. Використання ігрових рушіїв, таких як Unity або Unreal Engine, вимагає глибокого розуміння їхніх можливостей та обмежень.

Програмування

Програмування є основою будь-якої гри. Розробники використовують різні мови програмування, такі як C#, C++ або Python, для створення ігрових механік, обробки вводу гравця, управління об'єктами та реалізації штучного інтелекту. Важливою частиною є також оптимізація коду для забезпечення плавного ігрового процесу на різних платформах.

Комп'ютерна графіка

Комп'ютерна графіка відіграє ключову роль у створенні візуального досвіду гри. Це включає в себе моделювання 3D об'єктів, створення текстур, анімацію персонажів та ефектів. Розробники використовують спеціалізовані інструменти, такі як Blender, Maya або Substance Painter, для створення високоякісних графічних ресурсів.

Художні аспекти

Художній бік гри є не менш важливим, ніж технічний. Він включає в себе розробку концептуального мистецтва, дизайну персонажів, створення атмосфери гри та звукового оформлення.

Дизайн

Дизайн гри визначає її вигляд, стиль та відчуття. Це включає в себе роботу над концептуальним мистецтвом, створенням ігрових локацій, розробкою персонажів та їх костюмів. Дизайнери також працюють над інтерфейсом користувача, щоб забезпечити зручне та інтуїтивне керування грою.

Звукове оформлення

Звук є важливою складовою гри, що створює атмосферу та підсилює емоційне сприйняття. Це включає в себе створення музики, звукових ефектів та голосової озвучки. Розробники співпрацюють з композиторами та звуковими дизайнерами для створення якісного звукового оформлення.

Управлінські аспекти

Управління проектом є ключовим елементом успіху в розробці ігор. Це включає в себе планування, розподіл ресурсів, координацію команди та контроль якості.

Планування

Ефективне планування дозволяє розподілити завдання між членами команди та забезпечити своєчасне виконання проекту. Використання методологій управління проектами, таких як Agile або Scrum, допомагає структурувати процес розробки та адаптуватися до змін.

Контроль якості

Забезпечення високої якості гри є важливим аспектом. Це включає в себе тестування гри на різних етапах розробки, виявлення та виправлення помилок,

а також оптимізацію продуктивності. Тестування може включати автоматизовані тести, а також ручне тестування на різних пристроях.

Створення ігор є складним та багатогранним процесом, який вимагає від розробників високого рівня знань та навичок у різних галузях. Технічні, художні та управлінські аспекти проекту є взаємопов'язаними та вимагають комплексного підходу для досягнення успіху. Розуміння складнощів та особливостей цього напрямку дозволяє краще підготуватися до викликів, що постають перед розробниками, та створити якісний продукт, який буде задовольняти потреби гравців та ринку.

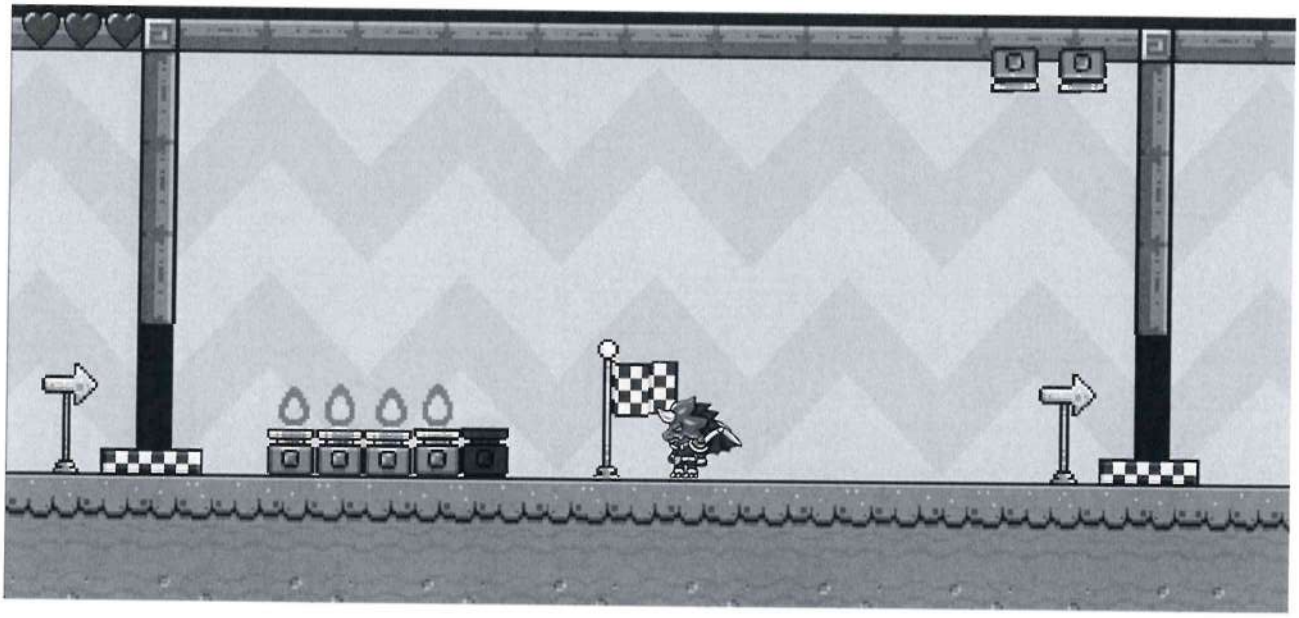
ВИСНОВКИ

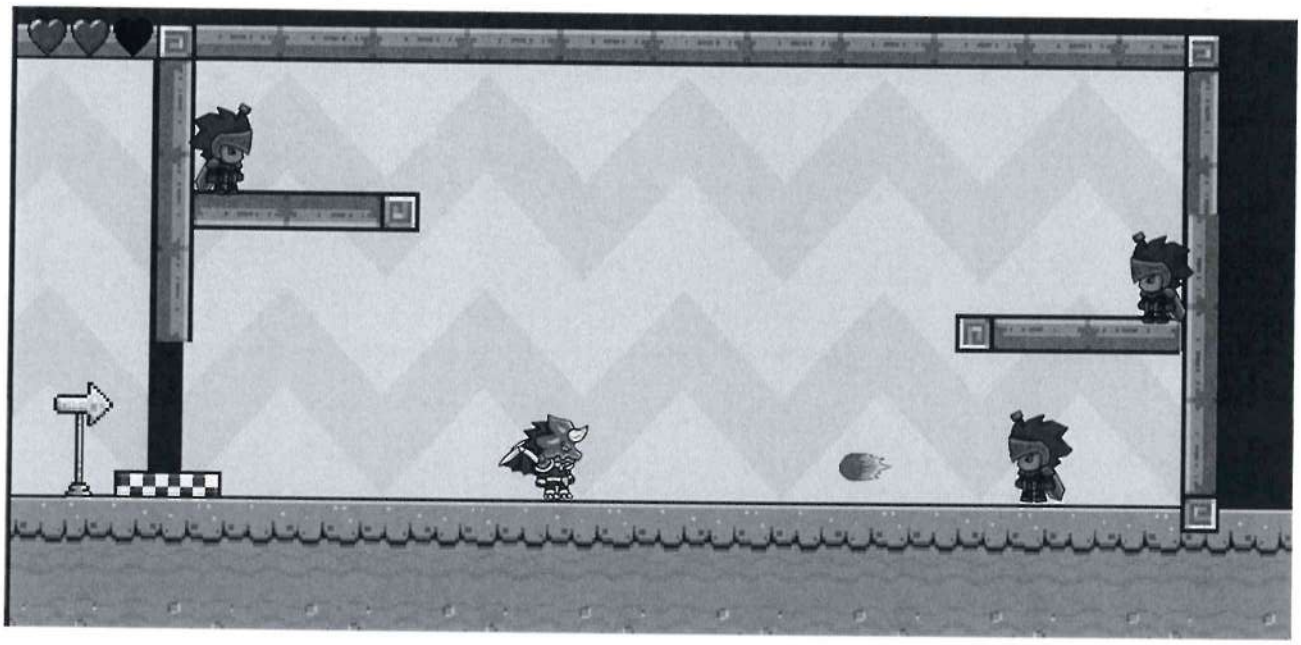
Висновком є створена гра яка демонструє у собі знання і навички того чого що я навчився за весь період і мій досвід який я отримав у результаті. Навіть якщо сама по собі гра не виглядає великою за обсягом кількість роботи на те щоб створити щось з 0 при відсутності досвіду у цьому напрямку була значною.

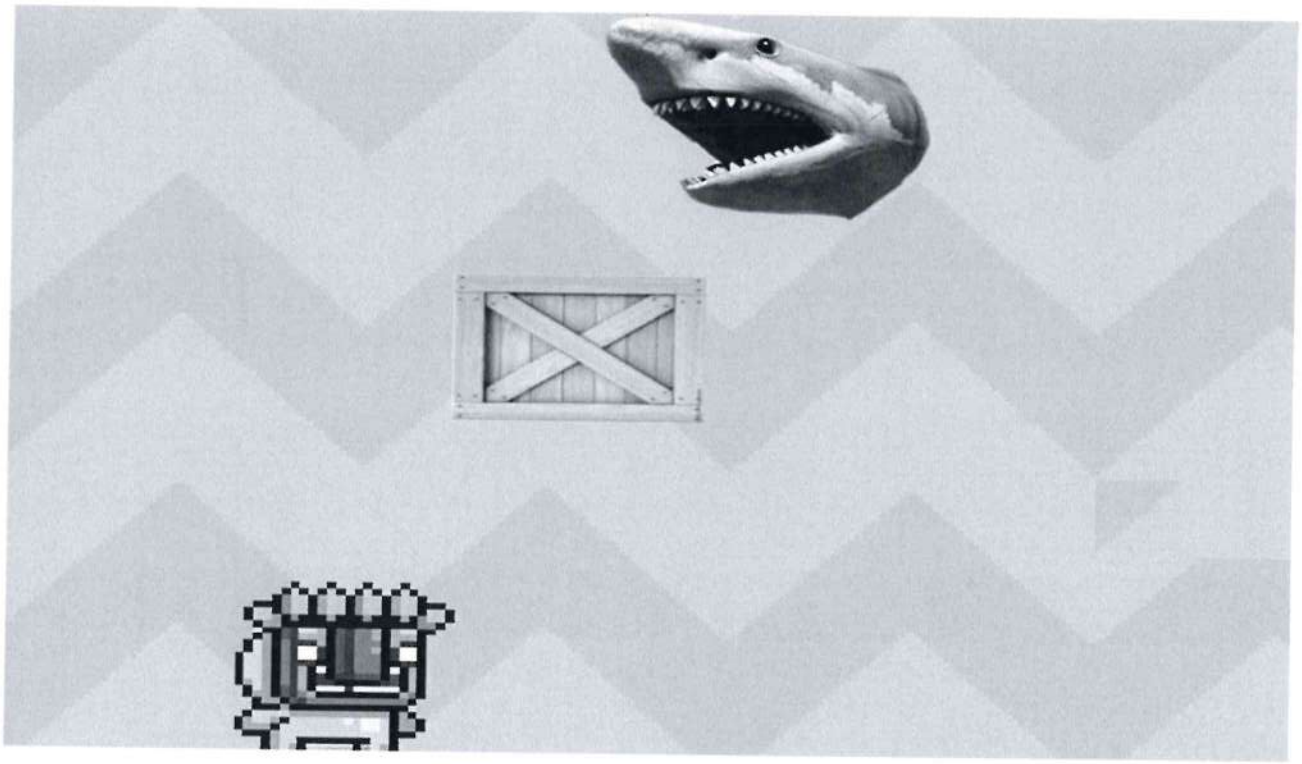
При створенні гри ви не тільки програмуєте а й ще проектувати свій власний продукт. Планування де саме буде ворог, який повинен бути рівень щоб гравцям не було нудно або дискомфортно, музика, графіка, анімації, історія – все це є невід'ємними компонентами будь якої гри.

І хоч це складні процеси, особливо коли робиш гру у невеликій команді або самотужки, неможливо не зазначити що це дуже унікальне й творче направлення і я не проти продовжити свій рух по даному вектору.

Кадри з гри







СПИСОК ПОСИЛАНЬ

1. Джозеф Албахарі, Бен Албахарі — «Довідник C#. Опис мови програмування» / Джозеф Албахарі, Бен Албахарі — США, 2020 р. — 1024 с.
2. Технічний посібник Unity https://docs.unity.com
3. Dev. Log різних розробників https://www.gamedev.net/tag/dev/
4. Канал на платформі YouTube для загального вивчення Unity https://www.youtube.com/@Dani_Krossing