МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет “Києво-Могилянська академія”

Факультет інформатики
Кафедра інформатики

КУРСОВА РОБОТА

освітній ступінь – бакалавр

на тему: “**Розробка 2D платформної гри з використанням Unity**”

виконав студент третього року навчання
напряму підготовки 122 “Комп’ютерні науки”
Ракітенко Дмитрій Андрійович

Керівник: ст. викладач, кандидат технічних наук
Бучко О. А.

Кількість балів: _____

Члени комісії:

_____ _____
(підпис) (прізвища та ініціали)

_____ _____
(підпис) (прізвища та ініціали)

_____ _____
(підпис) (прізвища та ініціали)

Київ – 2023

Тема: Розробка 2D платформної гри з використанням Unity

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	листопад	
2.	Огляд літератури за темою роботи	листопад - грудень	
3.	Створення практичної частини роботи	січень - березень	
4.	Написання текстової частини до курсової роботи	березень - квітень	
5.	Надання роботи керівнику	кінець квітня	
6.	Коригування роботи	кінець квітня – початок травня	
7.	Подання роботи на кафедру для перевірки на плагіат	тиждень до захисту	
8.	Захист курсової роботи	кінець травня	

Студент Ракітенко Д. А.

Керівник Бучко О. А.

“_____”_____

ЗМІСТ

Перелік прийнятих скорочень

ШІ – штучний інтелект;

SOLID – п’ять базових принципів об’єктно орієнтованого програмування;

2D – двовимірний простір;

3D – тривимірний простір;

API – Application Programming Interface, прикладний програмний інтерфейс;

PNG – Portable Network Graphics, формат графічного файлу.

АНОТАЦІЯ

Робота присвячена розробці 2D – платформної гри з використанням рушія Unity та сучасних підходів до програмування ігор. Гра містить кілька механік, звукове та графічне оформлення та інтерфейс користувача.

Ключові слова: рушій Unity, платформер, 2D, скінченний автомат

ВСТУП

На сьогоднішній день актуальність ігрової індустрії складно переоцінити, а ігрова розробка – популярний напрям для розробників програмного забезпечення. Створення ігор значно спрощують ігрові рушії, а рушій Unity є одним з найпопулярніших та найдоступніших. Рушій дозволяє застосувати типові шаблони об’єктно - орієнтованого програмування та принципи SOLID для розробки логіки та поведінки для гри, використовуючи мову C#, при цьому аудіовізуальне оформлення гри та побудову рівнів виконувати прямо в редакторі рушія з графічним інтерфейсом. Отже, Unity є надзвичайно зручним засобом, що пришвидшує розробку ігрових продуктів.

Враховуючи актуальність ігрового програмного забезпечення та зручність і популярність рушія Unity, мета цієї роботи – розробити 2D – платформну гру виключно засобами рушія, при цьому застосувати шаблон станів для реалізації ші ворогів та принципи SOLID для створення програмного коду гри. Відповідно мета зумовила таку дослідницьку та творчу діяльність:

1) Визначити особливості рушія Unity, ознайомитися з принципами його роботи та навчитися виконувати проекти в ньому.
2) Визначити особливості жанру 2D – платформних ігор та основні ігрові механіки і системи, які доцільно реалізувати в проекті.
3) Реалізувати відповідні системи засобами рушія, намагаючись знаходити сучасні та оптимальні рішення.
4) Оформити гру візуально, дослідити теми графіки, анімації та звукового супроводу.
5) Розробити інтерфейс користувача.
6) Створити рівень для гравця, використовуючи можливості рушія для пришвидшення цього процесу.

Роботу поділено на три розділи. Перший розділ надає базові відомості про рушій Unity, описує актуальність та інструменти рушія. Також в цьому розділі проаналізовано його недоліки та переваги. У другому розділі проаналізовано

задачу створення 2D гри та основні механіки та системи, які треба реалізувати. В останньому розділі повністю описано процес розробки проекту, в ньому розкрито тему типових ігрових механік платформних ігор, типові підходи до організації 2D графіки, звуку, створення інтерфейсу користувача та діалогової системи. Також в цьому розділі описується шаблон стану та ші ворогів.

ОСНОВНА ЧАСТИНА

РОЗДІЛ 1: Знайомство з рушієм Unity

1.1 Загальний огляд

Рушій Unity – один з найпопулярніших рушіїв для створення ігор у світі. Він надає потужні можливості для розробки як 2д, так і 3д проектів. Unity – мульти - платформна система, яку часто обирають розробники -любителі за її обширний функціонал, низку доступних розширень, додаткових навчальних матеріалів, що допомагають поліпшити рівень володіння рушієм та факт присутності безкоштовної версії. Приблизно половина мобільних ігор сьогодення створена на Unity, що доводить його актуальність та популярність [1].

1.2 Деякі недоліки рушія

Основним недоліком рушія є те, що його версія “Professional Edition” вимагає платити деяку фіксовану суму грошей раз на місяць. Деякі з функцій, що наявні лише в цій версії, необхідні для розробки великих комерційних проектів, наприклад доступ до бета версій та командна ліцензія [1]. Також рушій вимагає знання мови програмування C# та передбачає розуміння певних принципів роботи ігор. Крім цього, більша частина вихідного коду гри, який автоматично генерує рушій, лишається закритою для розробника та відома лише інженерам Unity, а можливості з низькорівневого розширення можливостей рушія досить обмежені.

1.3 Основні складові частини Unity

Рушій пропонує наступний спосіб задання ігрових об’єктів: у вікні редактора присутня сцена(Scene) (D), що містить об’єкти типу GameObject. Взаємодія цих об’єктів в рамках сцени віддалено нагадує взаємодію екземплярів класів в об’єктно орієнтованій парадигмі програмування: кожен об’єкт може містити дітей – ігрові об’єкти, що підпорядковуються йому, а

також набір компонентів, що відповідають за різні складові частини та властивості ігрових об’єктів. Найважливішими компонентами є скрипти, а саме C# класи, що успадковуються від базового класу MonoBehaviour. Також розробник має доступ до файлового дерева проекту(G), де можна впорядкувати та відредагувати усі файли проекту, що включають: графічні файли, аудіофайли, C# скрипти, а також префаби(Prefabs). Ця система є однією з найзручніших функцій Unity та надає можливості багаторазового використання ігрових об’єктів. Prefab у Unity дозволяє повторно використовувати ігровий об'єкт, включаючи всі його компоненти, значення властивостей та дочірні ігрові об'єкти(дітей). Prefab виступає як шаблон, який можна кілька разів розмістити в межах однієї або кількох сцен. Це набагато краще та зручніше, ніж просто копіювати та вставляти ігровий об'єкт, тому що система Prefab дозволяє автоматично синхронізувати всі копії [2]. До інших елементів інтерфейсу належать: вікно ієрархії, що показує ієрархію об’єктів в межах певної сцени(B), вікно ігрової симуляції, що демонструє, як буде виглядати фінальний ігровий досвід гравця(C), основні інструменти для редагування вікна сцени та ігрових об'єктів всередині неї(E), вікно усіх компонентів обраного ігрового об’єкту(F), що може бути префабом або знаходитися в межах сцени, бібліотеку доступних ресурсів для використання в межах проекту(G), також є можливість імпортувати нові ресурси, рядок стану надає повідомлення про різноманітні процеси Unity(H) та панель інструментів надає доступ до облікового запису Unity та хмарних сервісів(A) [2].

Мал. 1.1 Інтерфейс редактору Unity

1.4 Поняття скриптів та MonoBehaviour

Саме у C# скриптах міститься основна логіка роботи гри, створеної на рушії. Скрипти прикріплюються до ігрових об’єктів та дозволяють отримувати доступ до різних компонентів об’єкту, таких як графічні зображення, текстури, інші скрипти, фізичне тіло об’єкта, звукові композиції. За замовчуванням кожен скрипт містить методи Start() та Update(). Метод Start автоматично викликається при активації скрипта, а метод Update викликається під час кожного кадру роботи гри. Кількість кадрів на секунду – кількість зображень, які показує гра за одну секунду. Також існують інші різновиди методу Update(), наприклад FixedUpdate() використовується для взаємодії з фізикою гри, бо частота оновлення зображення на екрані та роботи частини Unity, що відповідає за фізику, не збігаються та можуть відрізнятися у кілька разів [4]. Розуміння принципів роботи скриптів дуже важливе для розробки на рушії Unity, бо вони відрізняються від написання типових програм на C# за рахунок особливих типів даних, методів та взаємодії з компонентами, що надає Unity Scripting Api.

РОЗДІЛ 2: Короткий огляд місця 2D-платформних ігор в індустрії розваг

2.1 Коротка історична довідка

2D – платформні ігри – різновид, жанр комп’ютерних ігор, що був дуже популярний в епоху 2D, а виник десь на початку 1980 - х років. Згодом жанр здійснив перехід до 3D, але 2D ігри лишаються популярними й досі. Одними з найперших представників цього жанру є ігри серії “Super Mario Bros”, “Donkey Kong Country” та “Castlevania”. В цих іграх гравець дивився на персонажа збоку, а до типового набору ігрових механік зазвичай входили можливості стрибати, долати різні перешкоди, вступати у битви з ворогами. Трохи згодом у такі ігри додавалися деякі рольові елементи, наприклад підняття рівня гравця, можливість підбирати речі, спілкуватися з іншими персонажами [4].

Мал. 2.1 Приклади серії “Castlevania”

Усі ці механіки гарно відомі сучасним гравцям, а даний жанр все ще досить популярний, особливо серед невеликих груп розробників з обмеженим фінансуванням. Одними з найпопулярніших сучасних 2D - платформерів є “Hollow Knight” та “Ori and the Blind Forest”. Ці проекти вдосконалили ті механіки, які гірше демонструють їхні попередники, а також зробили акцент на зображенні високої якості та естетичній насолоді гравця.

2.2 2D - платформна гра з точки зору розробника

Розробка гри, що задовольнить сучасний ринок клієнтів, зазвичай потребує великого часового та людського ресурсу, а також кілька команд досвідчених фахівців, проте створення відносно простих ігор з обмеженим числом механік під силу й одному розробнику, враховуючи можливість сконцентруватися на програмуванні поведінки гри та створенню скриптів, використовуючи готовий графічний та музикальний супровід. З основних етапів створення типового 2D – платформера можна виділити: рух персонажа, налаштування камери, логіку стрибка, систему здоров'я, систему стрільби та атак на малій дистанції, нанесення пошкоджень, штучний інтелект ворогів, створення графічного представлення усіх елементів гри, графічний інтерфейс користувача, налаштування рівня, музикальний супровід. Виконання кожного з даних пунктів на рушії потребує знання C# та розуміння різних типових підходів до ігрового програмування.

РОЗДІЛ 3: Розробка власної 2D – платформної гри

3.1 План розробки гри

Зазвичай розробку гри поділяють на кілька послідовних етапів, що доповнюють одне одного. В рамках даної роботи створення платформера було поділено на відповідні частини:

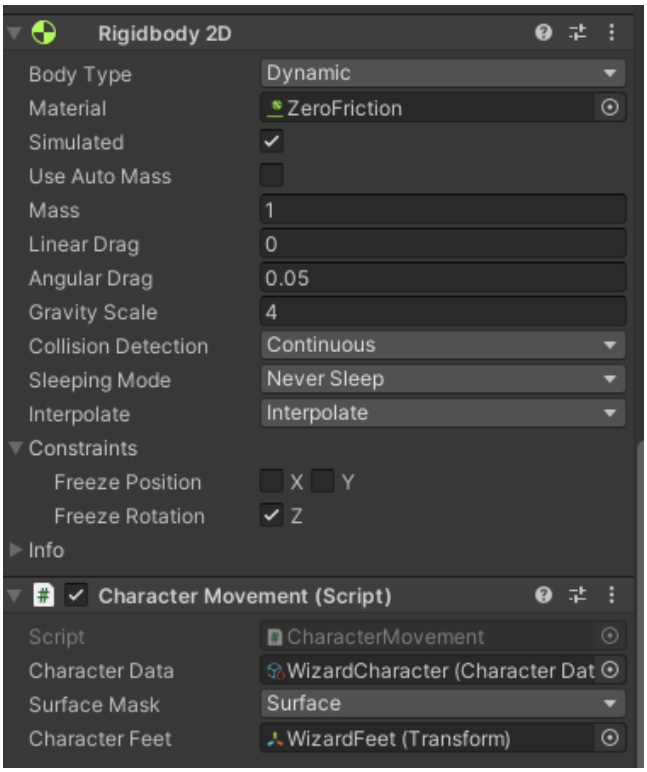
1) Створення базових механік руху персонажа, стрільби, універсального скрипта здоров'я, керування персонажем.
2) Розробка простої системи штучного інтелекту ворогів на основі теорії скінченних автоматів.
3) Підбір графічних та аудіоматеріалів в магазині Unity Asset Store. Цей магазин містить велику кількість навчальних та безкоштовних матеріалів, що дозволяє зекономити час на створенні власної графіки, що потребує специфічних навичків. В магазині доступні матеріали, які створюють учасники спільноти користувачів Unity [2].

4) Створення анімацій з використанням Animator та ефектів різноманітними способами, а також налаштування звуку.
5) Створення інтерфейсу користувача, початкового ігрового меню, меню смерті персонажа.
6) Створення діалогової системи, що забезпечує можливість спілкування персонажів.
7) Фінальна побудова ігрового рівня засобами Unity та оптимізація ігрового досвіду гравця.

3.2 Основні механіки та системи

3.2.1 Рух персонажа

Існує кілька типових підходів до реалізації переміщення ігрового об’єкта на сцені. Кожен об’єкт має компонент Transform з координатами x, y, z, проте третя координата під час 2D розробки не береться до уваги, тож робота переважно відбувається з координатами x та y. Найбільш очевидним підходом до реалізації є зміна відповідних координат об’єкта на сцені, проте цей метод не є оптимальним, бо рушій надає свої інструменти для роботи з фізичними об’єктами. Використаємо компонент Rigidbody2D, що симулює фізичні властивості певного об’єкта. У випадку з 2D об’єктом на його спрайт завдяки Rigidbody2D будуть впливати закони гравітації та прискорення, імпульси, вага тощо [3]. Відповідно для руху об’єкта будемо відстежувати, чи натискає користувач клавіші на клавіатурі, що відповідають за рух, та прискорювати об’єкт на заданий параметр швидкості. Таким чином, не треба змінювати координати GameObject вручну. Відповідні компоненти можна додати до об’єкту Wizard, основного персонажу у грі.

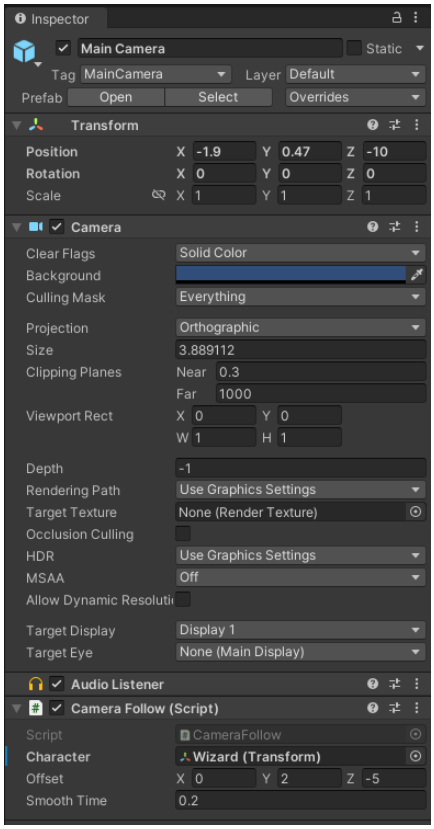


Мал. 3.1 Компоненти Rigidbody2D та CharacterMovement

Фізичне тіло дозволяє також імплементувати стрибок вгору, проте при прискоренні вже буде змінюватися координата y. Під час реалізації цієї функції виникає одна з типових проблем – персонаж не має стрибати у повітрі нескінченно, тож перед стрибком слід перевірити, що він знаходиться на землі. Для цього зручно використовувати механізм шарів Unity Layers, що дозволяє надавати різним ігровим об’єктам у сцені різні шари. Наприклад, для перевірки на стрибок можна пересвідчитися, що персонаж Wizard знаходиться на об’єкті з шару Surface, використовуючи Physics2D.OverlapCircle() що надається Scripting Api [3]. Під час ігрової розробки дуже важливо використовувати саме такі оптимальні рішення, бо вони сильно полегшують модифікацію та розширення скриптів. Рушій також надає вбудовану компоненту CharacterController, що має методи для керування гравцем, проте використання такого функціоналу зменшує рівень контролю розробником того, як саме обробляється рух [6].

3.2.2 Слідування камери за персонажем

Важливою частиною сцени є камера, що має слідувати за рухами персонажа. В Unity існує кілька типових способів реалізації такої функції. Найочевидніше рішення – зробити об’єкт камери підпорядкованим об’єкту гравця, та тоді координати у просторі її компоненти Transfrom будуть змінюватися паралельно з рухом гравця, проте це рішення не є оптимальним. Краще скористатися вбудованою компонентою Cinemachine для створення низки різних видів камер, або створити скрипт, що буде приймати ігровий об’єкт, за яким буде слідувати камера, а також додати деякі додаткові параметри, такі як зсув, що буде переміщувати камеру на певну відстань від персонажа, аби він не знаходився в центрі екрану під час гри. Для таких задач часто використовують методи, які надає Vector3 клас Unity [7]. Метод SmoothDamp дозволяє камері поступово рухатись за персонажем з певною затримкою [3]. Такий підхід створює приємний візуальний ефект слідування. Скрипт для слідування додається в якості компоненти до основної камери на сцені. Параметер smooth time відповідає за те, наскільки плавно камера “наздоганяє” персонажа.

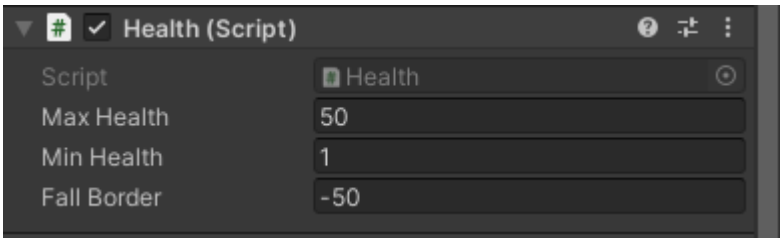


Мал. 3.2 Скрипт CameraFollow – компонент камери

3.2.3 Здоров’я та пошкодження, смерть

Здоров’я є однією з найважливіших систем в іграх, бо смерть мотивує гравця спробувати пройти рівень ще раз та вчитися на своїх помилках. Хоча і персонаж гравця, і вороги потребують цю систему, її доцільно створити у вигляді універсального скрипту, який можна використовувати для багатьох різних ігрових об’єктів, навіть для статичних перешкод. У скрипті в якості параметрів слід зберігати максимальний та мінімальний показники здоров’я об’єкта, а також в якості змінної цілочисельного типу – поточний рівень здоров’я. Скрипт здоров’я не відстежує, які об’єкти та яким чином можуть нанести пошкодження тому об’єкту, до якого компоненту здоров’я додано, проте надає відкритий метод отримання урону takeDamage, що дозволяє отримати доступ до цієї компоненти з іншого C# скрипту, а здоров’ю лишається тільки обробити нанесені пошкодження. Такий підхід часто застосовується в Unity для зменшення кількості зв’язків між компонентами. Копонент здоров’я також відповідає за смерть персонажа та має обробляти та розрізняти смерть ворога та смерть гравця. Виникає питання: як скрипт може визначити, до якого класу належить його ігровий об’єкт – батько. Для таких задач рушій надає систему тегів Tag, а аби виконувати певний програмний код тільки за умови, що ігровий об’єкт має певний тег, достатньо використати метод CompareTag, що порівнює тег об’єкту з заданим рядковим параметром [3]. Отже, гра завершується, якщо здоров’я гравця падає нижче мінімального рівня, а якщо помирає ворог, то його ігровий об’єкт знищується за допомогою методу Destroy. Destroy прибирає зі сцени заданий об’єкт миттєво або з заданою затримкою. Він є типовим та використовується для вирішення низки проблем програмування ігор на рушії Unity [3]. Однакові скрипти здоров’я додаються як до персонажа, так і до ворогів. Параметер fallBorder відповідає за координату y, де закінчується доступний ігровий простір, тобто забезпечує

смерть від падіння за межі ігрового простору. Такий параметер доцільний у платформних іграх, де гравець має долати перешкоди за допомогою стрибків.



Мал. 3.3 Скрипт Health з параметрами

3.2.4 Атака, нанесення пошкоджень

Механіка нанесення пошкоджень є однією з базових для різних ігрових жанрів. З точки зору розробника існує кілька підходів до її реалізації. Одним з найзручніших та популярних методів є використання Raycast, вбудованого в рушій Unity. Враховуючи 2D природу проекту, використовується саме 2D Physics2D.Raycast, що працює з координатами x та y у просторі. Концептуально цей функціонал рушію запускає промінь з вектору з початковими координатами у заданому напрямі та з заданою дистанцією [3]. Якщо ігровий об’єкт, якого торкнуся промінь, має компонент Collider, то зі змінної типу RaycastHit2D можна отримати до нього доступ та перевірити, чи містить він компонент Health, реалізація якого розглядалася раніше. Якщо так, за допомогою методу takeDamage цієї компоненти можна нанести урон ворогові. Так само ворог може знімати здоров’я гравця. Використовується також випадкова генерація напряму стрільби, щоб гравець не завжди стріляв

в одне й те саме місце.

```
private void DoShot()
{
    Vector3 shotDirection = CalculateShotDirection();
    GameObject shotTrailCurrent = Instantiate(shotTrail, _muzzlePoint.position, Quaternion.Euler(shotDirection));
    if (!_movement.IsFacingRight) shotTrailCurrent.GetComponent<SpriteRenderer>().flipX = false;
    ShotTrail shotTrailScript = shotTrailCurrent.GetComponent<ShotTrail>();
    Debug.DrawRay(_muzzlePoint.transform.position, shotDirection * _weaponData.range, Color.red, 1f);
    RaycastHit2D hit = Physics2D.Raycast(_muzzlePoint.position, shotDirection * _weaponData.range);
    if (hit && !hit.collider.CompareTag("MainCharacter"))
    {
        shotTrailScript.SetTarget(hit.point);
        GameObject impactEffect = Instantiate(impactEffects[Random.Range(0, impactEffects.Length - 1)], hit.point, Quaternion.identity);
        Destroy(impactEffect, _effectDestructDelay);
        AudioSource.PlayClipAtPoint(impactSound, new Vector3 (hit.point.x, hit.point.y, FindObjectOfType<Camera>().transform.position.z));
        var collider = hit.collider;
        Health health = collider.GetComponent<Health>();
        if (health != null) health.takeDamage(_weaponData.damage);
        if (_weaponData.knockback > 0)
        {
            Rigidbody2D rb = collider.GetComponent<Rigidbody2D>();

            if (rb != null)
            {
                rb.AddForce(Vector2.right * _weaponData.knockback, ForceMode2D.Impulse);
            }

        }
    }
    else shotTrailScript.SetTarget(_muzzlePoint.position + shotDirection * _weaponData.range);
    _isFiring = false;
}
```

Мал. 3.4 Реалізація методу DoShot() з Raycast

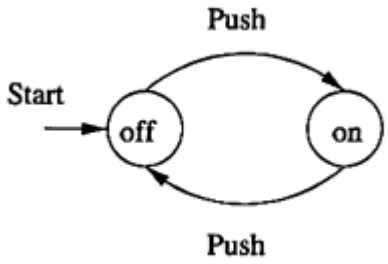
Інформація про зброю зберігається у ScriptableObject – контейнері для даних, що дозволяє незалежно зберігати велику кількість різних даних. На відміну від MonoBehaviour, такі об’єкти не містять скриптів та використовуються зазвичай для більш зручного збереження даних, параметрів, характеристик тощо [2]. Такий підхід дозволяє створювати кілька різних конфігурацій зброї з різними параметрами, при цьому в параметрах скрипта зброї змінюється лише одне поле.

Мал. 3.5 WeaponData SriptableObject з параметрами

3.3 Штучний інтелект ворогів

3.3.1 Поняття скінченного автомату

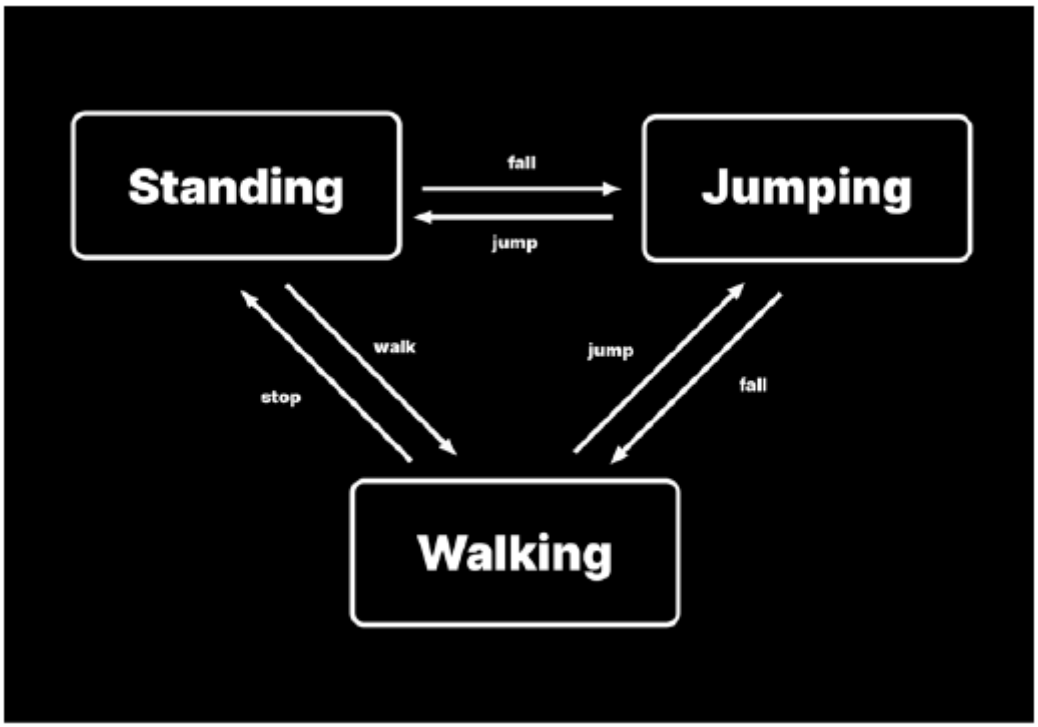
Скінченний автомат – дуже важлива та базова модель багатьох програмних та апаратних продуктів. Концептуально ідея автомату полягає в тому, що він постійно знаходиться в одному з заданих заздалегідь станів, відповідно кількість станів обмежена та визначена, що спрощує створення моделі та її обслуговування. Одним з найпростіших прикладів скінченного автомату є перемикач з положеннями увімкнено та вимкнено. Така система має два стани та два переходи, та відповідно в кожному зі станів можливий лише один перехід, зовнішньою умовою для переходу є натискання на перемикач [8, с. 17 - 18]. Слід зазначити, що часто інформація, що зберігається в станах, досить нетривіальна та скінченний автомат використовується для багатьох задач, в тому числі і для моделювання деяких процесів в іграх.



Мал. 3.6 Скінченний автомат - перемикач

3.3.2 Місце скінченного автомата в ігровій розробці

В ігровій розробці скніченний автомат як модель часто використовується для відстеження та зміни стану персонажа або об’єкту на ігровій сцені [9]. Самі розробники Unity у своїх рекомендаціях щодо використання шаблонів проектування в іграх називають такий шаблон State Pattern (шаблон станів). Схематично діаграма станів, в яких може знаходитися персонаж, виглядає наступним чином(за умови, що персонаж не має інших станів):

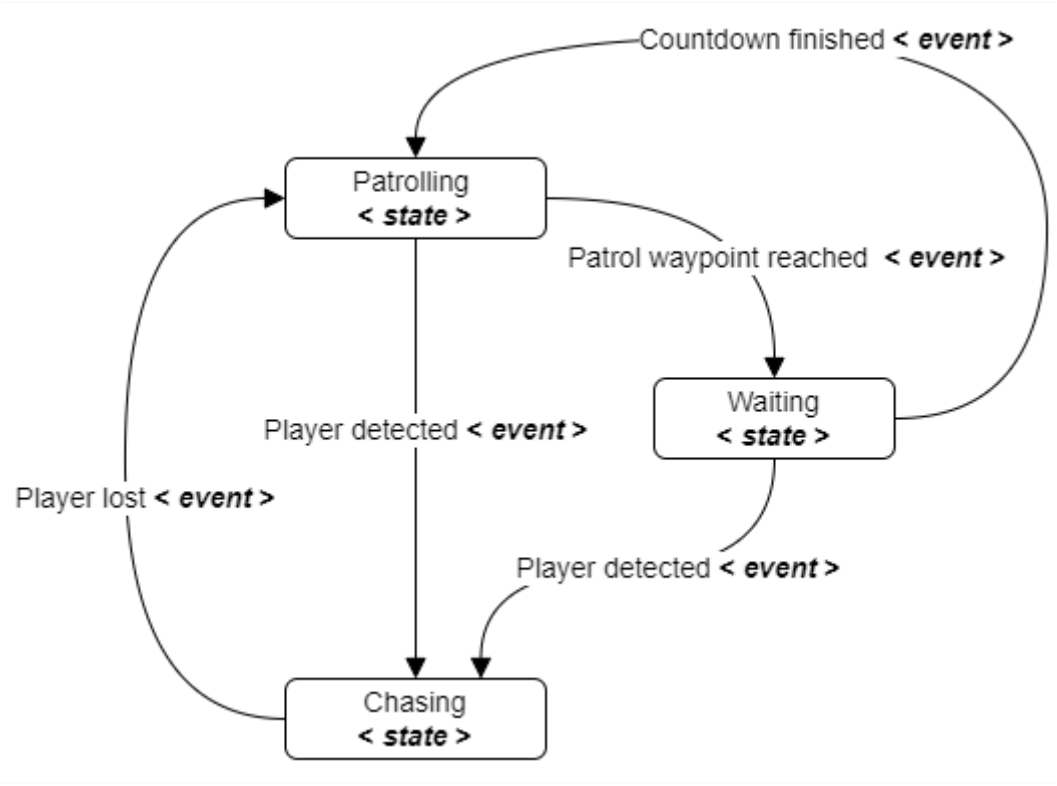


Мал. 3.7 Стани персонажа

Такий підхід є надзвичайно корисним, бо в динамічних іграх система постійно отримує команди від гравця та стан ігрової сцени змінюється. Наприклад, у проекті, що розглядається в цій роботі, основний персонаж має принаймні 4 стани: стан нерухомості, стан руху, стан атаки та стан знаходження в повітрі(стрибка вгору та падіння вниз). Він не може знаходитися в двох станах одночасно, а також мають бути визначені умови переходів між станами. Проте через те, що персонажем керує гравець, використання шаблону станів для реалізації його поведінки є надлишковим та достатньо використовувати умовні вирази та змінні булевого типу для контролю станів. Коли ж мова йде про ворогів, цілком доцільно для зручності скористатися шаблоном стану для створення штучного інтелекту. Ворог теж має такі стани як гравець, проте не стрибає, а також можемо визначити певні умови переходів між ними.

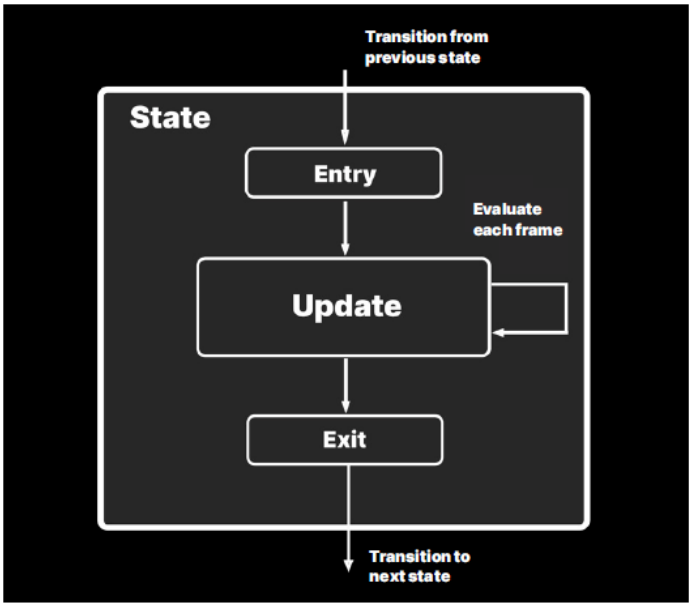
3.3.3 Створення власного скінченного автомату для штучного інтелекту ворогів

Аби створити таку систему, доцільно скористатися абстрактними класами, що будуть надавати інтерфейс для конкретних станів та переходів у автоматі. Також доцільно використати об’єкти типу Scriptable Object, що згадувався раніше, аби для зміни деяких параметрів ші ворогів було достатньо змінити саме екзмепляр Scriptable Object [10]. Схематично наша модель буде мати такий вигляд, проте стан Chase передбачає також атакувати гравця, коли ворог на достатній для цього відстані.



Мал. 3.8 Схема скінченного автомату для ворогів

Маємо кілька станів та події – умови переходів між ними. Ворог ходить між кількома місцями на ігровій карті, поки не бачить гравця. Коли ворог доходить до заданої точки, він чекає деякий час та йде до іншої точки зі списку заданих. Якщо ворог бачить гравця, він починає слідувати за ним та атакувати його, якщо перестає бачити гравця – знову переходить у стан патрулювання [10]. Отже, маємо скінченний автомат поведінки ворога та можемо задавати точки патрулювання. Для реалізації стану зазвичай використовується базовий шаблон наступного вигляду:



Мал. 3.9 Структура стану

У випадку використання такого інтерфейсу кожен стан має визначати наступні методи:

1) Enter – виконується при входженні у стан.
2) Execute – виконується кожного кадру в методі Update з Unity.
3) Exit – виконується при виході зі стану та відповідно переході в інший стан.

Отже, всі стани будуть працювати за такою схемою [9].

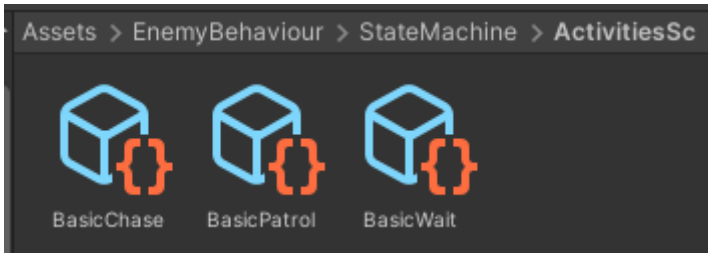
Загалом можемо поділити автомат на такі частини:

1) Стани – об’єкти Scriptable Object, що реалізують базовий шаблон, що визначений вище. Кожен стан містить список дій, які має виконувати ворог в цьому стані та список переходів, які може здійснити ворог. Клас BaseStateMachine запускає та виконує автомат, також визначає початковий стан машини.
2) Дії – те, що ворог виконує в межах певного стану. В нашому випадку кожному стану відповідає певна дія. Кожна дія містить методи Enter, Execute та Exit відповідно шаблону стану та в цих методах

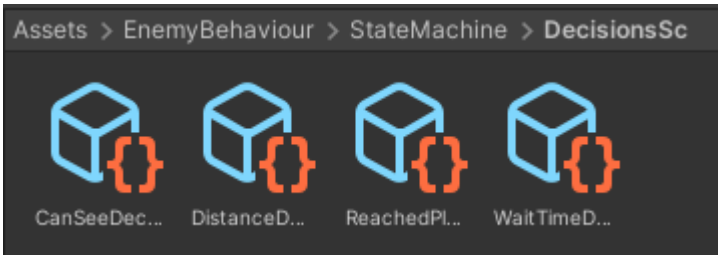
визначається поведінка ворога. Конкретні дії є об’єктами Scriptable Object, як і стани. Під час одного стану може виконуватись кілька дій.

3) Рішення – кожне рішення реалізує метод Decide та має вирішити, в який стан переходить автомат в залежності від результатів виконання цього методу. В нашому випадку необхідні рішення: чи може ворог бачити гравця, чи надто далеко гравець від ворога, чи достатньо ворог очікує на місці патрулювання та чи дійшов ворог до відповідного місця.
4) Переходи – визначення переходів з одного стану в інший. Кожен окремий перехід також є SriptableObject та містить рішення та два стани – стан, в який має перейти автомат, якщо було прийняте позитивне рішення, та відповідно стан – наслідок прийняття негативного рішення. Інколи стан не треба змінювати при отриманні негативного результату методу Decide, тоді автомат лишається в тому стані, в якому був.

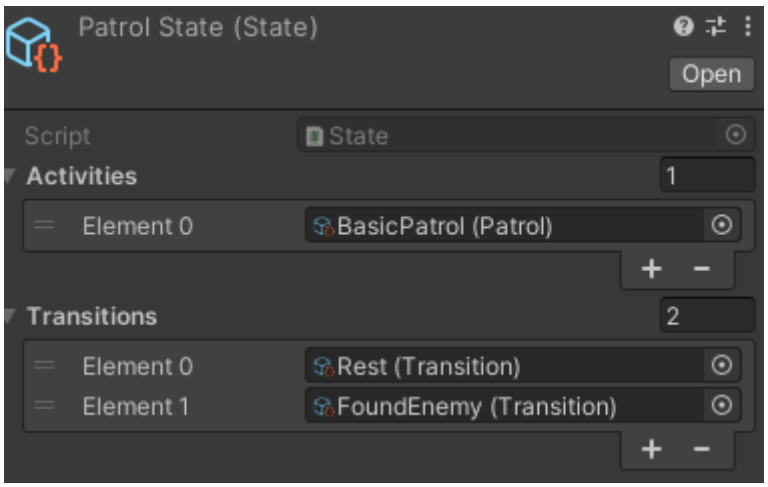
Отже, для використання автомату достатньо створити об’єкти для станів, рішень, переходів та дій відповідно до схеми ші ворога, а також запрограмувати всю необхідну логіку. Наведемо приклади об’єктів:



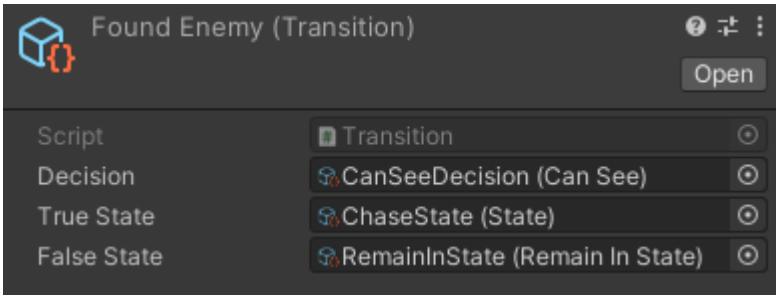
Мал. 3.10 Елементи ScriptableObject для дій



Мал. 3.11 Елементи ScriptableObject для рішень



Мал. 3.12 Стан патрулювання



Мал. 3.13 Перехід “Ворога знайдено”

Наступні елементи автомату визначаються аналогічно. Після закінчення модифікацій автомату достатньо до ігрового об’єкту ворога додати скрипт BaseStateMachine та визначити точки патрулювання і початковий стан ворога.

3.4 Графічне та звукове оформлення гри

3.4.1 Звукове оформлення гри

Unity надає низку можливостей обробки аудіофайлів та використання їх у грі. Рушій підтримує низку форматів, а для роботи зі звуком використовується AudioSource компонент. Він надає низку методів та параметрів для коректного відтворення звуку, наприклад, чи програвати мелодію одразу після ініціалізіції компонента та чи починати програвати її спочатку після завершення. Цей компонент зручно використовувати для програвання фонової музики, а для програвання звуків у грі, наприклад вибухів, часто використовується метод AudioSource PlayClipAtPoint, що програє

мелодію в заданих координатах на ігровій сцені, використовуючи Vector3. Часто створюється додатковий скрипт музикального програвача, аби розширити базовий функціонал Audio Source. Наприклад, у грі використовується поступова заміна однієї композиції на іншу з затримкою, яку доцільно реалізувати в окремому класі.

```
public static IEnumerator FadeOut(AudioSource audioSource, float fadeTime)
{
    float startVolume = audioSource.volume;

    while (audioSource.volume > 0)
    {
        audioSource.volume -= startVolume * Time.deltaTime / fadeTime;
        yield return null;
    }

    audioSource.Stop();
    audioSource.volume = startVolume;
}
```

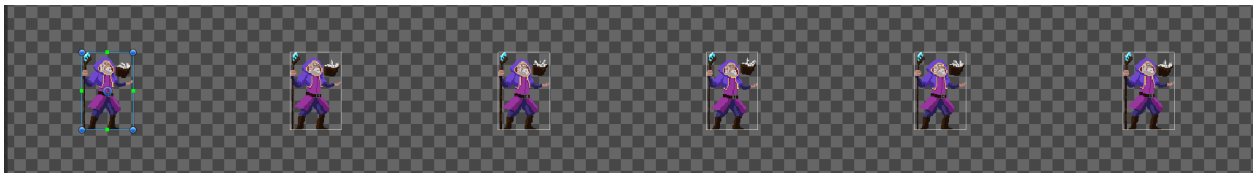
Мал. 3.14 Метод поступового зменшення гучності музики

FadeOut поступово зменшує гучність композиції а згодом повністю зупиняє її. Використовується мехпнізм Unity Coroutines. Він дозволяє створити певну затримку та часто використовується для створення функцій, що мають працювати поступово, а не в реальному часі. Yield дозволяє зупинити виконання програмного коду та повернутися туди, де його було зупинено. Важливо розуміти, що Coroutines не є потоками та виконуються в основному потоці, хоча чимось й схожі на потоки за принципом роботи [2].

3.4.2 Графічне оформлення гри

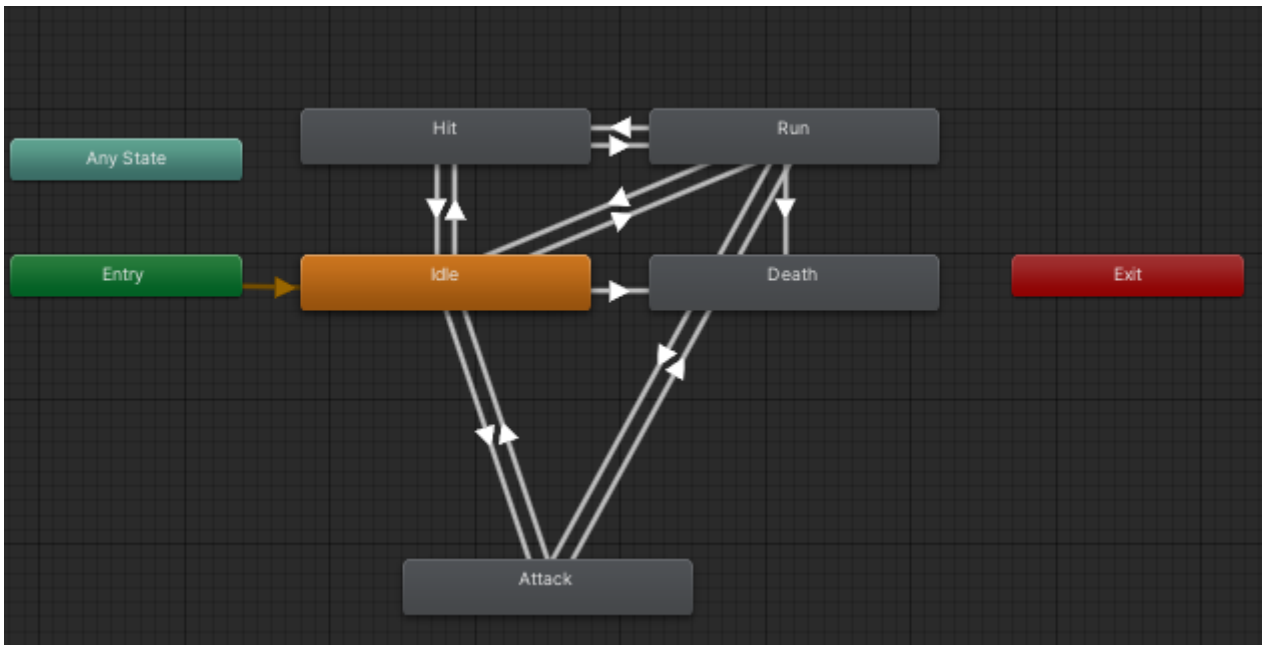
Для створення 2д графіки зазвичай використовуються спрайти. Очевидно, що спрайт – двовимірний графічний об’єкт, який зазвичай являє собою картинку або серію картинок, з яких можна створити покадрову анімацію. Саме зручність створення анімації є найбільшою перевагою спрайтів. Їх можна обертати та всіляко змінювати в межах ігрової сцени, а також їх часто використовують для анімування дій, що відповідають станам персонажів, наприклад стану нерухомості, стрибку, атаці чи ходьбі, бігу [11]. Анімувати

також можна і ігрове оточення, наприклад лампу, запалений факел чи прапор під дією вітру. У рушії Unity для роботи зі спрайтами використовується зручна компонента SpriteRenderer, що приймає спрайт у форматі картинки як параметер. Наприклад, розглянемо як зберігається анімація нерухомості головного героя проекту. Маємо один png файл з кадрами анімації, який розділено автоматично засобами рушію.



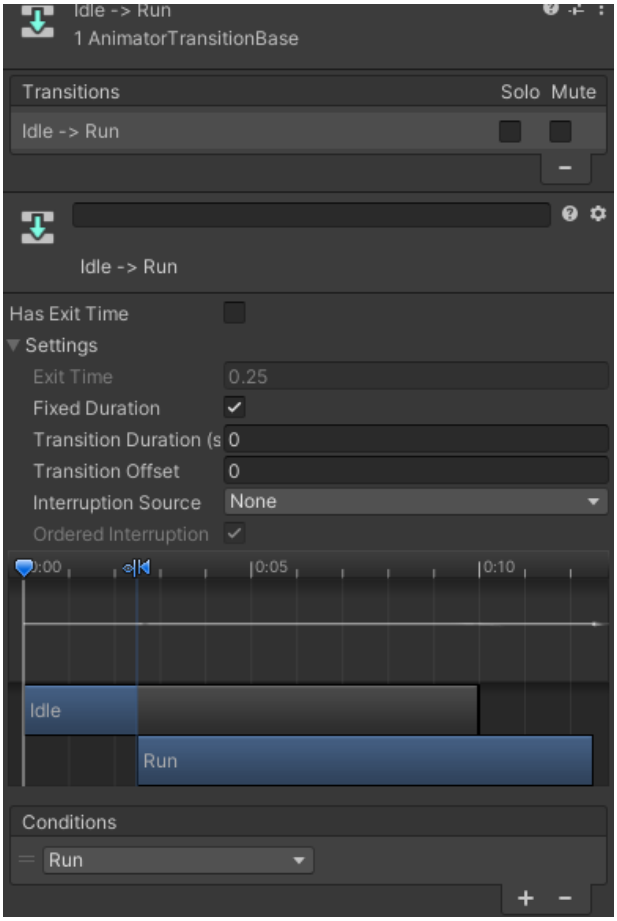
Мал. 3.15 Спрайти для анімації головного героя

Аби перетворити цей набір зображень на анімацію, використовують компоненту Animation, а, аби налаштувати різні анімації для ігрового об’єкту, використовують AnimationController. Цей компонент дозволяє оформити переходи між анімаціями у вигляді діаграми, а також встановити певні умови, при яких анімації будуть змінюватися. Такі умови можна використати в коді інших скриптів за допомогою методу SetTrigger(). Наприклад, для вище розглянутого ші ворогів можна, коли ворог знаходиться у стані патрулювання, встановити його анімацію через компоненту Animator, використавши SetTrigger(“Run”), що відповідає стану ходьби ворога. Загалом діаграма станів анімації ворога виглядає таким чином:



Мал. 3.16 Діаграма станів ворога для анімації

Налаштування окремого переходу з визначенням умови переходу доступно для розробника в окремому вікні:



Мал. 3.17 Перехід з анімації Idle до анімації Run

Наприклад, перехід ворога з нерухомого стану у стан ходьби відбувається при встановленні відповідної заздалегідь створеної умови Run, що зберігається у вигляді рядку символів “Run”. Таким чином можна налаштувати низку анімацій для різних об’єктів, як ігрового персонажа, так і ворогів.

3.4.3 Ефекти

Окремим аспектом візуального оформлення гри є ефекти. Часто створюють ефекти стрільби, вибухів, ударів для того, щоб гравець краще та приємніше відчував свої дії під час гри. На відміну від анімацій, ефекти часто використовуються у вигляді готових об’єктів типу Prefab, та їх достатньо створити у заданому місці у разі потреби та видалити через деякий час, використовуючи метод Destroy. У проекті присутній ефект вибуху під час атаки головного героя. Такий ефект має програватися у місці попадання у ворога, отже використати його доцільно у скрипті Weapon у разі позитивного відпрацювання Raycast. Також часто генерують один з кількох доступних ефектів випадковим чином, як і в нашому проекті, де генерується один з елементів списку ігрових об’єктів impactEffects:

```
if (hit && !hit.collider.CompareTag("MainCharacter"))
{
    shotTrailScript.SetTarget(hit.point);
    GameObject impactEffect = Instantiate(impactEffects[Random.Range(0, impactEffects.Length - 1)], hit.point, Quaternion.identity);
    Destroy(impactEffect, _effectDestructDelay);
```

Мал. 3.18 Створення ефекту в місці попадання

Сам ефект використовує ParticleSystem з Unity. Це компонент, що використовується для низки ефектів, таких як кров, дощ, чи вибухи, як в цьому прикладі. Робота з такими системами досить складна, тому часто розробники - початківці використовують безкоштовні шаблони для відображення ефектів.

3.5 Елементи інтерфейсу користувача

3.5.1 Основне меню

Для створення елементів інтерфейсу в Unity присутній окремий функціонал, а саме компонент Canvas. На цьому компоненті можна

розміщувати картинки, текстові поля, різноманітні слайдери, кнопки. За допомогою цього компоненту легко створити елементи основного меню. Воно складається з картинки заднього фону, двох кнопок – вийти та почати гру, а також з оформленої назви гри: “Elderly Wizard Routine”. Для створення кнопок, тексту та картинок використовують вбудовані компоненти Unity Button, Text та Image. Unity має різноманітні налаштування кнопок, а при натисканні на кнопку зазвичай виконується скрипт або якась функція певного компоненту на сцені. Рушій також надає різні налаштування: оформлення, колір, прозорість полів, шрифти. Також присутній зручний спосіб імпортування сторонніх шрифтів формату TrueType.

Мал. 3.19 Зовнішній вигляд основного меню

Наприклад, так виглядає налаштування компоненти кнопки “Start Game”:

Мал. 3.20 Компонент Button

Кнопка змінює колір в залежності від позиції курсору гравця, а також програє звук натискання та виконує скрипт, що відкриває основну ігрову сцену.

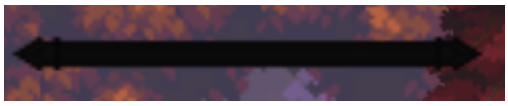
3.5.2 Меню смерті

Аналогічно до початкового меню легко створити меню смерті. Після смерті гравцю пропонується спробувати ще раз або завершити гру. Як і у випадку з головним меню, використовується Canvas та дві кнопки. Загалом, Unity має зручну систему для створення об’єктів інтерфейсу.

Мал. 3.21 Зовнішній вигляд екрану смерті

3.5.3 Візуалізація здоров’я

Canvas дозволяє створити шкалу здоров’я за допомогою елементу Slider. Цей компонент можна налаштувати таким чином, що він буде відображати поточне здоров’я гравця. Після цього достатньо створити скрипт HealthVisualizer для відстеження здоров’я персонажа через скрипт Health. Якщо число здоров’я змінюється, оновлюється відповідне значення шкали здоров’я. Отже, отримуємо інтерактивний слайдер здоров’я персонажа, який використовує створений раніше скрипт Health головного персонажа. Коригування параметру value слайдера заповнює шкалу здоров’я на задане число, як показано на малюнках нижче.

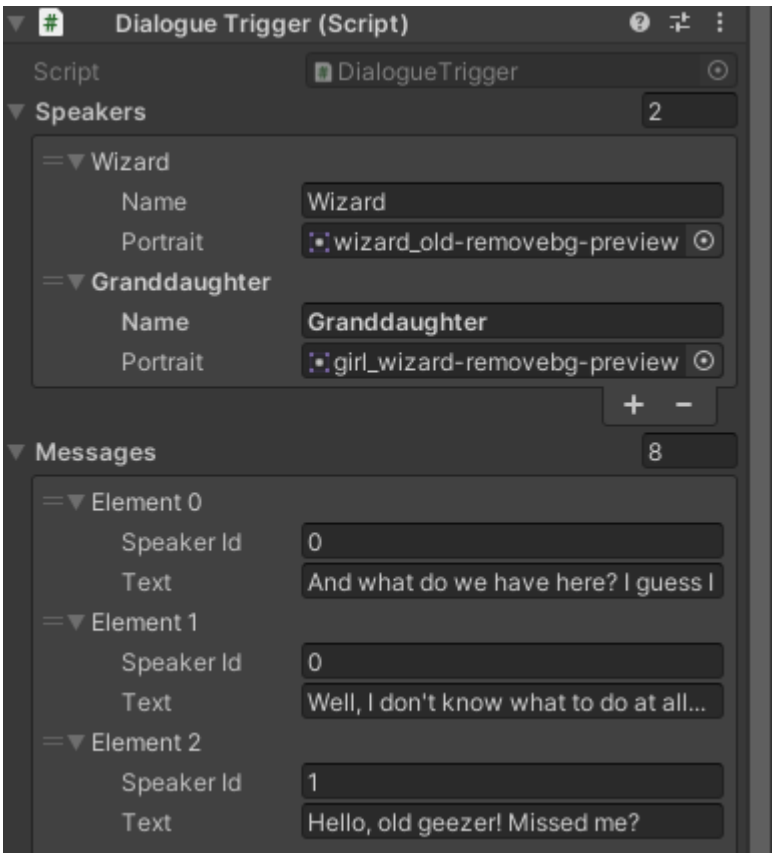


Мал. 3.22 Здоров’я, значення слайдеру value 0

Мал. 3.23 Здоров’я, частково заповнена шкала

3.6 Діалогова система

Для передачі деяких сюжетних аспектів гри та пояснення гравцеві основ управління персонажем використовується проста діалогова система. Вона також є компонентом Canvas, а репліки героїв зберігаються у списку рядків. Також використовується шрифт, імпортований з формату TrueType. Аби кілька героїв могли розмовляти між собою, для кожної репліки визначається ім’я та зображення того, хто її говорить. Основна логіка обробки діалогу відбувається у скрипті DialogManager, а скрипт DialogueTrigger використовує менеджер, аби запускати довільні діалоги за умови дотримання певних запрограмованих дій, використовуючи відкритий метод StartDialogue. Наприклад, один діалог запускається на початку гри, інший – коли гравець натрапляє на нову, невідому для себе перешкоду.



Мал. 3.24 Приклад використання компонента DialogueTrigger

Візуально діалог оформлюють різними способами. У випадку проекту літери з’являються поступово з програванням характерного звуку, для цього використовуються Coroutines.

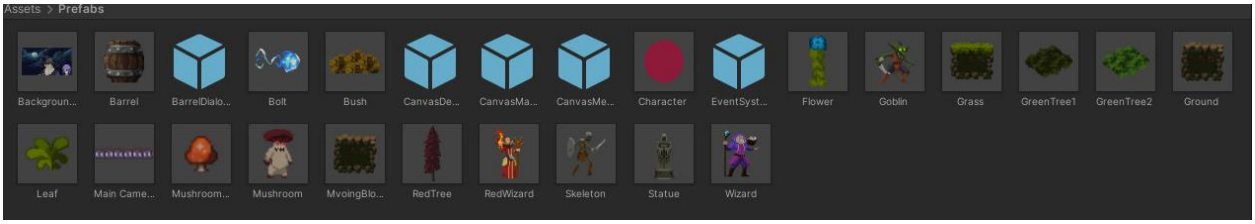
Мал. 3.25 Візуальне оформлення діалогового вікна

Отже, маємо зручну діалогову систему, що використовується для безлічі різних ситуацій та є універсальним способом донести певну інформацію до гравця.

3.7 Побудова ігрового рівня

Фінальний етап розробки – побудова ігрового рівня для тестування усіх розроблених наробок. Для створення рівня:

1) Виділимо певну кількість об’єктів Prefab, які можна зручно розміщувати на сцені. Слід намагатися максимально перевикористовувати ігрові об’єкти та створювати шаблони. Таким чином, створення рівня нагадує гру з конструктором.



Мал. 3.26 Створення Prefab

Таким чином для створення рівня маємо: кілька типів ворогів, кілька типів поверхні, перешкоду, блок, який може рухатись, та різні допоміжні декорації і тестові елементи.

2) Після створення заготовок слід продумати дизайн рівня, подумати про складність проходження для гравця та про те, на яких етапах рівня треба підключати певні реалізовані механіки. Наприклад, недоцільно

одразу запускати на гравця багато ворогів, поки він не звик рухатись та керувати базовими діями персонажа.

3) Після підготовчих етапів можна приступити до базового оформлення рівня. При використанні заготовок цей процес є досить творчим. В ігровій розробці слід враховувати, що навіть з обмеженою кількістю графічних елементів можна створювати різні рівні, розміщуючи ці графічні елементи по – різному.

Мал. 3.27 Рівень. Початок гри

Мал. 3.28 Рівень. Перешкода

Мал. 3.29 Рівень. Зустріч з ворогом

ВИСНОВОК

Отже, було реалізовано просту 2D – платформну гру з використанням рушія Unity. Гра містить кілька основних механік, простий інтерфейс користувача, ворогів, графіку та звуковий супровід. Ігрова розробка дуже різнопланова та потребує розуміння багатьох галузей, тож для реалізації проекту було проведено певну дослідницьку діяльність з різних тем. Для реалізації ші ворогів було використано шаблон стану та скінченний автомат. Було виконано поставлені базові завдання та план розробки.

Розробка ігрових продуктів є дуже кропіткою та поступовою діяльністю, проте дозволяє розширити знання з низки галузей інформатики та поліпшити розуміння роботи програмних продуктів, що працюють в реальному часі та постійно оновлюють екран. Також ігри часто використовують поняття та алгоритми, шаблони з теорії ші, тому при розробці також треба було досліджувати і цю важливу тему.

Список використаних джерел

1. How to Choose the Best Video Game Engine [Електронний ресурс] – Режим доступу до ресурсу: https://www.gamedesigning.org/career/video-game-engines/.
2. Unity User Manual 2021.3 [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Manual.
3. Unity Script Reference [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/ScriptReference.
4. 2D platform video games [Електронний ресурс] – Режим доступу до ресурсу: https://gamicus.fandom.com/wiki/2D_platform_video_games.
5. 10 Best Castlevania Games [Електронний ресурс] – Режим доступу до ресурсу: https://www.dualshockers.com/best-castlevania-games/.
6. Creating a Physics Based Character Controller in Unity [Електронний ресурс] – Режим доступу до ресурсу: https://levelup.gitconnected.com/creating-a-physics-based-character-controller-in-unity-61b3830dc042.
7. Make a Follow Camera in Unity [Електронний ресурс] – Режим доступу до ресурсу: https://gamedevbeginner.com/how-to-follow-the-player-with-a-camera-in-unity-with-or-without-cinemachine/.
8. Хопкрофт Д. Introduction to Automata Theory, Languages and Computation Second Edition / Д. Хопкрофт, Р. Мотвані, Д. Ульман., 2001. – 535 с.
9. Level up your code with game programming patterns [Електронний ресурс] / Unity. – 2021. – Режим доступу до ресурсу: https://resources.unity.com/games/level-up-your-code-with-game-programming-patterns.

10. Finite State Machine for AI Enemy Controller in 2D [Електронний ресурс] – Режим доступу до ресурсу: https://pavcreations.com/finite-state-machine-for-ai-enemy-controller-in-2d/.
11. Löw A. What is a sprite? [Електронний ресурс] / Andreas Löw – Режим доступу до ресурсу: https://www.codeandweb.com/knowledgebase/what-is-a-sprite.