

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ МОРСЬКИЙ УНІВЕРСИТЕТ

Кафедра кібербезпеки та захисту інформації

Методичні вказівки до практичних робіт

з освітнього компоненту (дисципліни)

РОЗРОБКА ТА ЗАХИСТ ІГРОВИХ ЗАСТОСУНКІВ

для здобувачів спеціальності F5 – Кібербезпека та захист інформації

галузі знань F – Інформаційні технології

другого (магістерського) рівня освіти

для денної та заочної форми навчання

ОДЕСА – 2025

Методичні вказівки до практичних робіт з освітнього компоненту **«Розробка та захист ігрових застосунків»** частина 1 для здобувачів спеціальності F5 – Кібербезпека та захист інформації галузі знань F – Інформаційні технології другого (магістерського) рівня освіти для денної та заочної форми навчання / Укл.: Лебедєва О.Ю. – Одеса, 2025. – 80 с.

Розробник: к.т.н., доцент Лебедєва Олена Юріївна, доцент кафедри кібербезпеки та захисту інформації.

Схвалено на засіданні кафедри «Кібербезпека та захист інформації» Протокол від №2 від 06.10.2025.

Схвалено на засіданні НМК ННІ ІТІП Протокол від №2 від 14.10.2025

Зміст

Вступ.....	4
Практична робота 1. Створення гри Space shooter на Unity.....	5
1. Теоретичний матеріал	5
2. Завдання до практичної роботи 1.....	20
Практична робота 2. 2D гра. Основні об'єкти.....	21
1. Теоретичний матеріал	21
2. Завдання до практичної роботи 2.....	26
Практична робота 3. 2D-гра. Анімація спрайтів	27
1. Теоретичний матеріал	27
2. Завдання до практичної роботи 3.....	34
Практична робота 4. 2D-гра. Рух персонажу	35
1. Теоретичний матеріал	35
2. Завдання до практичної роботи 4.....	43
Практична робота 5. 2D гра. Вороги	44
1. Теоретичний матеріал	44
2. Завдання до практичної роботи 5.....	50
Практична робота 6. 2D гра. Оточення	51
1. Теоретичний матеріал	51
2. Завдання до практичної роботи 6.....	56
Практична робота 7. 2D гра Clicker та UI	57
1. Теоретична частина	57
2. Завдання до практичної роботи 7.....	69
Практична робота 8. Методи захисту від злову в іграх.....	70
1. Теоретичний матеріал	70
2. Завдання до практичної роботи 8.....	79
Література.....	80

Вступ

Освітній компонент (навчальна дисципліна) «Розробка та захист ігрових застосунків» орієнтований на засвоєння здобувачами вищої освіти другого (магістерського) рівня методів розробки та захисту ігрових застосунків. Під час опанування освітнього компонента (навчальної дисципліни) здобувачі вивчатимуть основні питання, пов'язані зі створенням 2D-ігор та методами захисту від злому в іграх.

Метою освітнього компонента (навчальної дисципліни) є забезпечення розвитку фахових компетентностей майбутніх магістрів шляхом оволодіння сучасними підходами до розробки та захисту ігрових застосунків.

Завданнями, що мають бути вирішені у процесі викладання освітнього компонента (навчальної дисципліни), є набуття здобувачами компетентностей за освітньою програмою.

Інтегральна компетентність.

Здатність особи розв'язувати задачі дослідницького та/або інноваційного характеру у сфері інформаційної безпеки та/або кібербезпеки.

Загальні компетентності (КЗ)

КЗ-1. Здатність застосовувати знання у практичних ситуаціях

Фахові (спеціальні) компетентності (ФК/СК):

КФ1. Здатність обґрунтовано застосовувати, інтегрувати, розробляти та удосконалювати сучасні інформаційні технології, фізичні та математичні моделі, а також технології створення та використання прикладного і спеціалізованого програмного забезпечення для вирішення професійних задач у сфері інформаційної безпеки та/або кібербезпеки.

Вивчення освітнього компонента (навчальної дисципліни) передбачає здобуття таких *результатів навчання (РН)*:

ПРН4. Застосовувати, інтегрувати, розробляти, впроваджувати та удосконалювати сучасні інформаційні технології, фізичні та математичні методи і моделі в сфері інформаційної безпеки та/або кібербезпеки.

ПРН13. Досліджувати, розробляти, впроваджувати та використовувати методи та засоби криптографічного та технічного захисту інформації бізнес/операційних процесів, а також аналізувати і надавати оцінку ефективності їх використання в інформаційних системах, на об'єктах інформаційної діяльності та критичної інфраструктури.

Практична робота 1. Створення гри Space shooter на Unity

Мета роботи: познайомитися із програмою Unity. Навчитися створювати 2D-гру за допомогою вже існуючих асетів в програмі Unity.

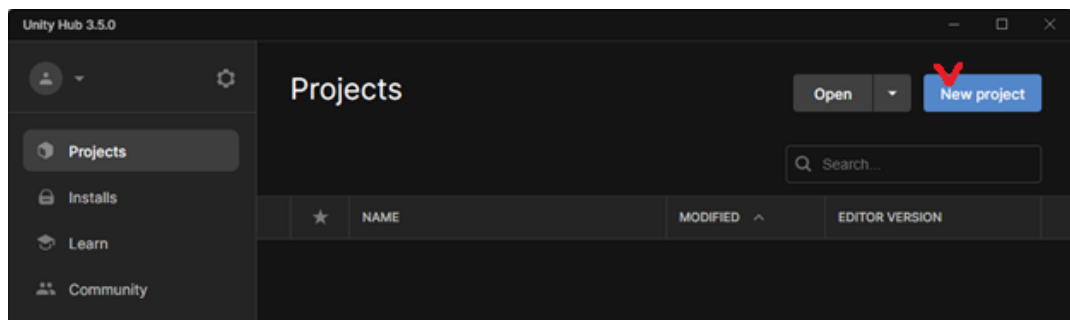
1. Теоретичний матеріал

Створимо гру, використовуючи такі кроки:

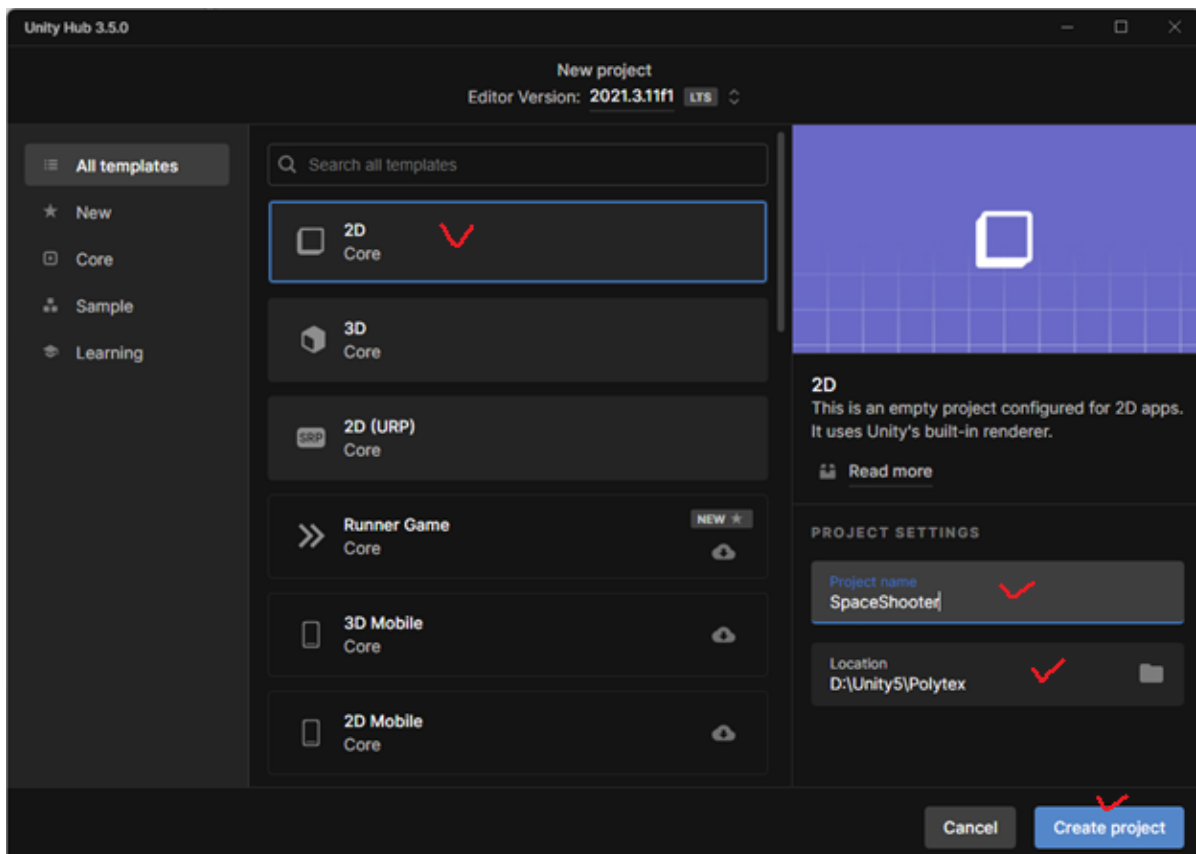
1. Створення порожнього проекту для гри 2D;
2. Створення нескінченного тла;
3. Додавання ігрового об'єкта;
4. Додавання ворогів

Крок 1. Створення порожнього проекту для гри 2D.

Запускаємо Unity Hub, в лівій панелі вікна переходимо до Projects та натискаємо кнопку New project



Вибираємо тип проекту, вводимо назву та місцезнаходження для гри, що створюємо:



Асет – це представлення будь-якого предмета, який можна використовувати у грі чи проєкті. Ресурс може бути отриманий з файлу, створеного поза Unity, наприклад, 3D-моделі, аудіофайлу, зображення або іншого типу файлу, який підтримує Unity. У Unity можна також створити деякі типи ресурсів, такі як Animator Controller, Audio Mixer або Render Texture.

Assets (активи), створені поза Unity, мають бути перенесені до Unity шляхом збереження файлу безпосередньо у папці «Assets» вашого проєкту чи копіювання до цієї папки. Для багатьох поширених форматів ви можете зберегти вихідний файл прямо в папці Assets вашого проєкту, і Unity зможе його прочитати. Unity помітить, коли ви збережете нові зміни у файлі, та повторно імпортує їх за потреби.

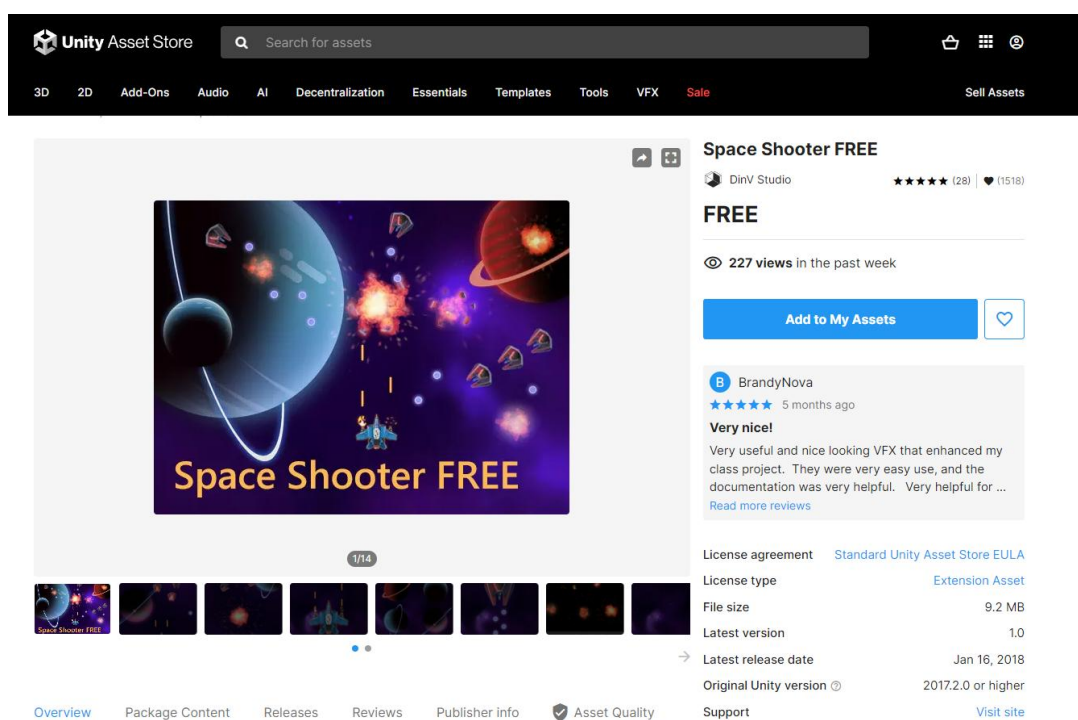
Якщо ви перетягнете файл у вікно проєкту Unity зі свого комп'ютера (наприклад, із провідника у Windows), він буде скопійований до папки Assets і з'явиться у вікні проєкту.

Найпростіший спосіб безпечно перемістити або перейменувати ваші активи завжди робити це з папки проєкту Unity. Таким чином, Unity автоматично перемістить або перейменує відповідний метафайл.

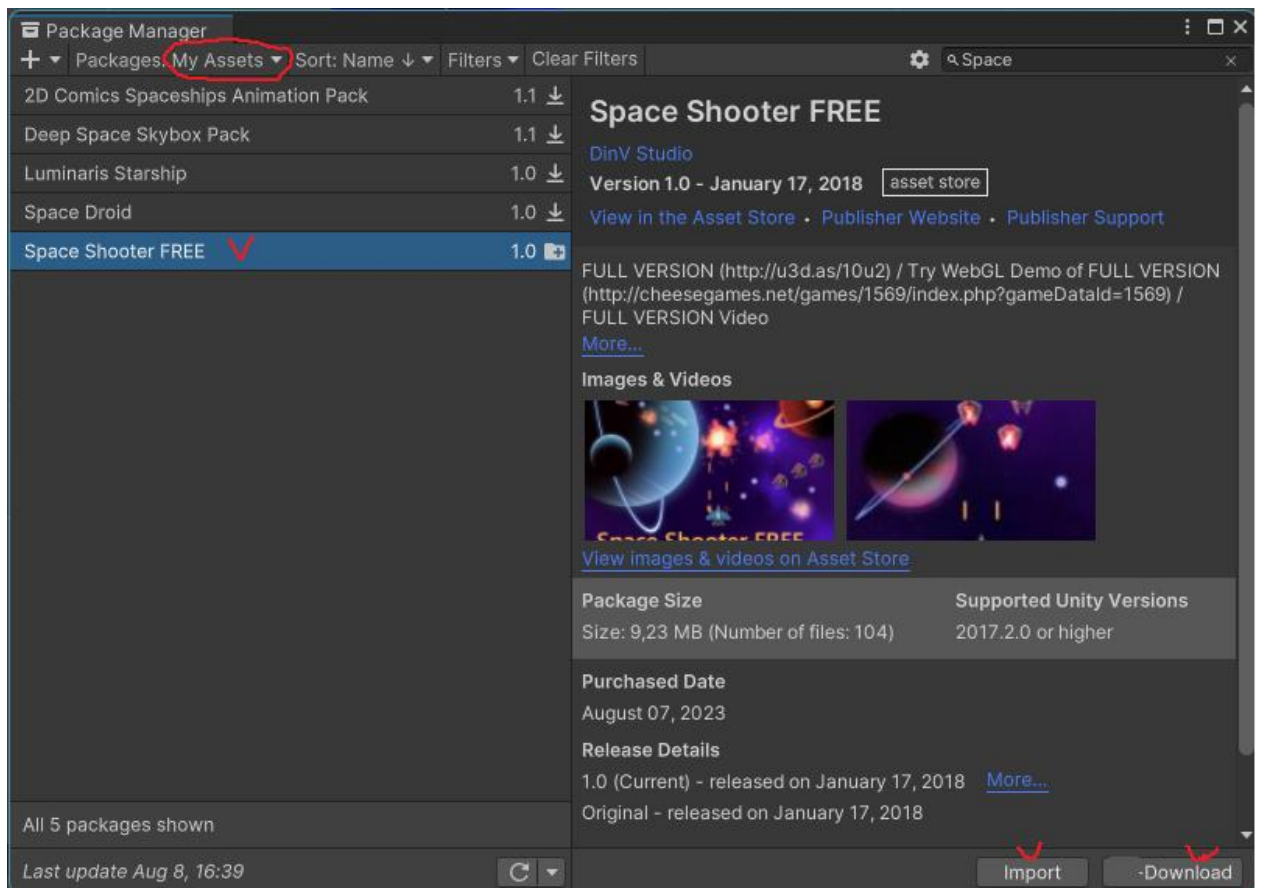
Unity Магазин (Asset Store) – це бібліотека, в якій зібрані безкоштовні та комерційні асети, створені як Unity Technologies, так і членами спільноти. Доступний великий вибір асетів — текстур, моделей та анімацій, прикладів проєктів, підручників та розширень для редактора. Асети доступні через простий інтерфейс, вбудований у редактор Unity. Через нього можна завантажити та імпортувати безпосередньо у ваш проєкт.

Відкриємо вікно Asset Store, вибравши в головному меню Window → Asset Store. При першому відвідуванні вам буде запропоновано зареєструвати безкоштовний обліковий запис, який буде використовуватись для доступу до магазину.

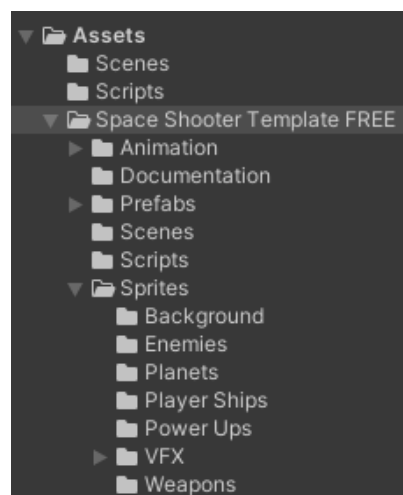
Додаймо до проєкту активи для створюваної нами гри. Для цього вибираємо Window → Asset Store та натискаємо кнопку Search Online. У вікні Unity Asset Store, що відкрилося в браузері, необхідно знайти Space Shooter FREE. Зайти в нього та натиснути на Add to My Assets. Вибраний асет завантажуватиметься у добірку асетів на вашому обліковому записі.



Переглянути та встановити потрібний асет можна за допомогою Window → Package Manager.

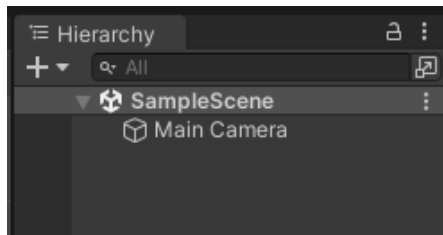


В результаті імпорту вибраного асета в папці Assets з'явиться нова папка Space Shooter Template FREE із набором асетів:



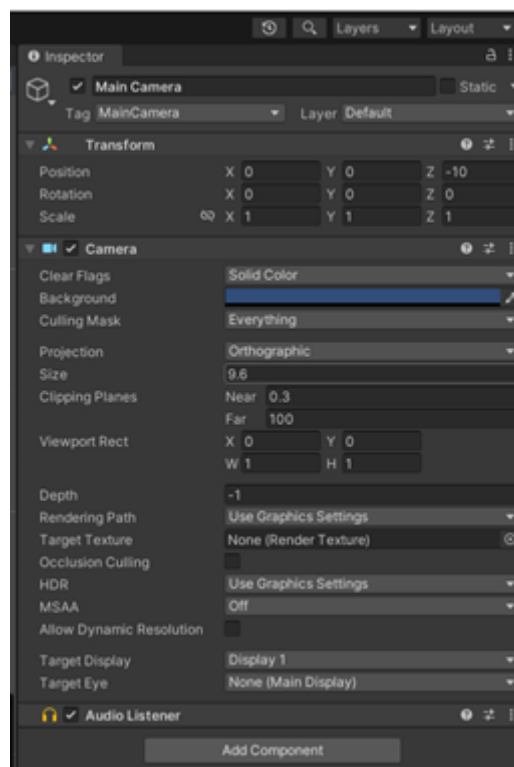
Ієрархія (Hierarchy) містить всі об'єкти (GameObject) у поточній сцені. Unity використовує концепцію під назвою Parenting. Щоб зробити будь-який GameObject дочірнім для іншого, перетягніть бажаний дочірній об'єкт на бажаний батьківський в ієрархії. Дочірній об'єкт успадковуватиме переміщення та повороти свого батька. Ви можете використовувати стрілку батьківського об'єкта, що розкриває, для того, щоб при необхідності показати або приховати його дітей.

При створенні проєкту 2D-гри в ієрархії створено об'єкт сцени, і до нього дочірньо створено камеру.



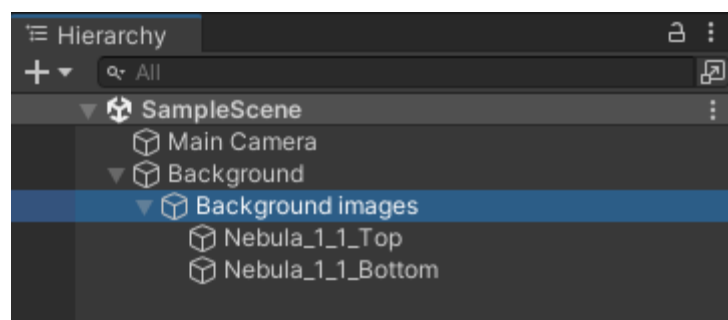
Камери є пристроями, які захоплюють та відображають світ гравцю. Можна мати необмежену кількість камер у сцені. Ви можете налаштувати рендеринг камерами в будь-якому порядку, на будь-якому місці екрана або лише в певних частинах екрану.

Встановимо камеру вертикально. Переходимо в режим Game вибираємо або створюємо 9:16 Aspect. Виділяємо Main Camera та у властивостях у полі Size встановлюємо значення 9,6.

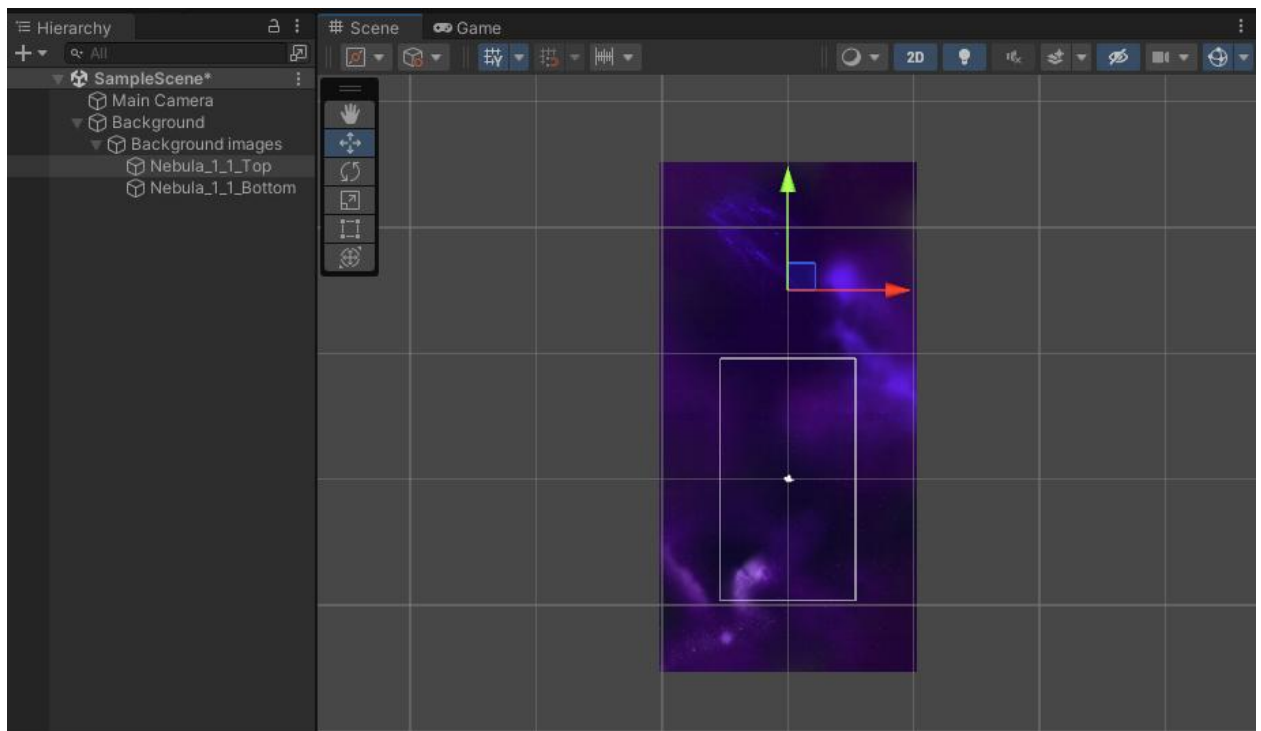


Крок 2. Створення нескінченного фону

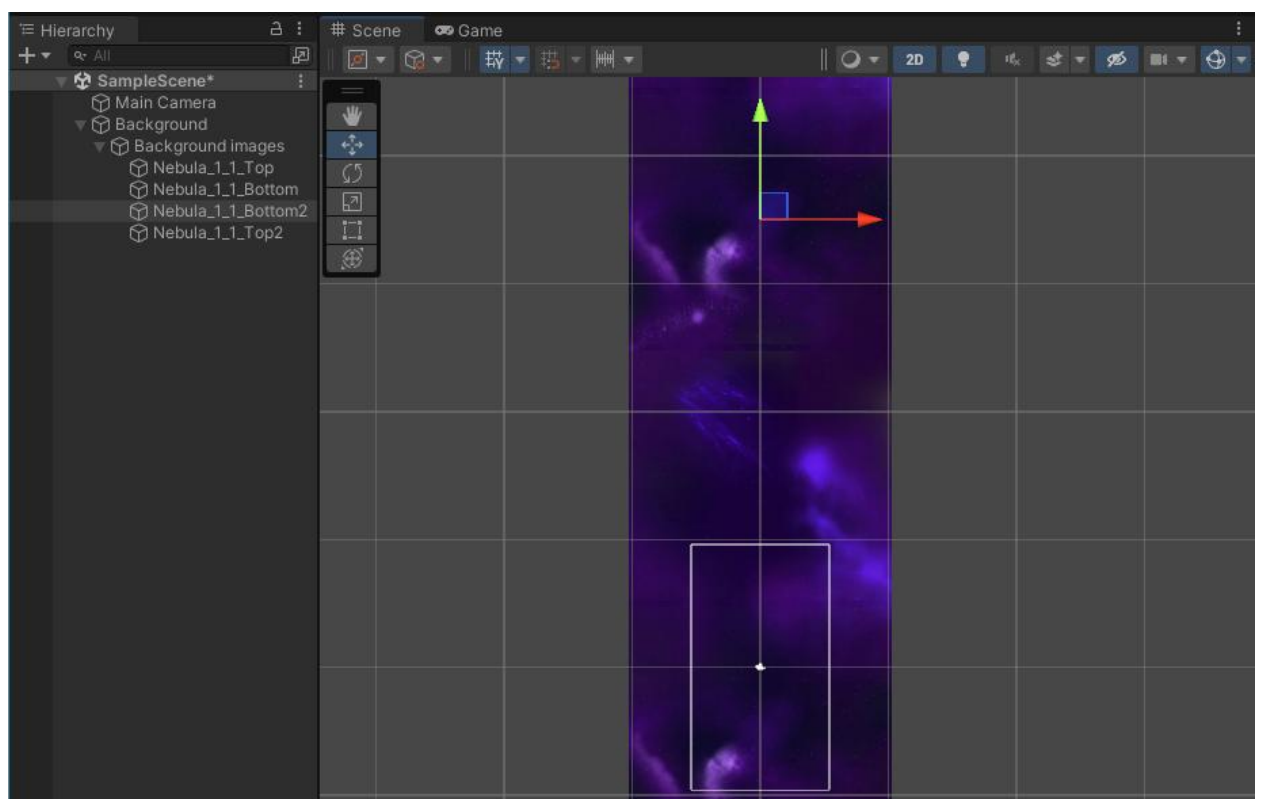
Створимо новий порожній об'єкт (Create Empty), назовемо його Background. У ньому створимо ще один порожній об'єкт Background images і помістимо спрайти з підвантаженого асета.



Розташуємо їх на сцені в такий спосіб:



Продублюємо наявні спрайти і розташуємо вище початкових:

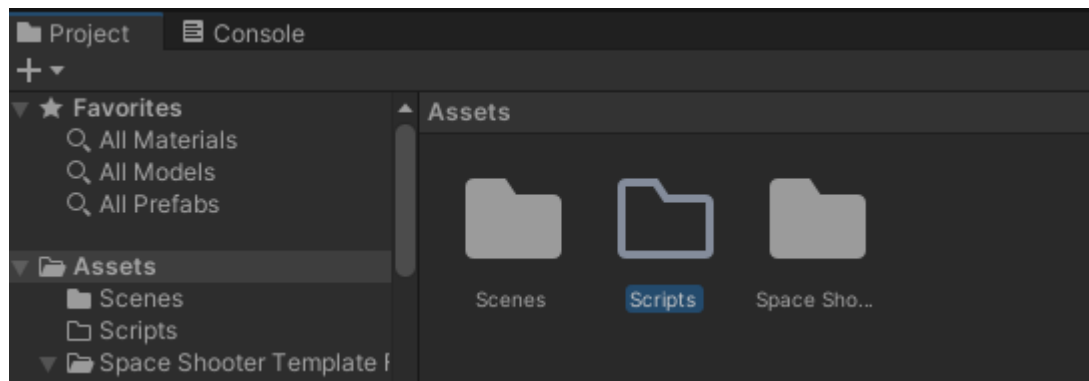


Скриптинг – необхідна складова для всіх ігор. Навіть найпростіші ігри потребують скриптів для реакції на дії гравця та організації подій геймплею. Крім того, скрипти можуть бути використані для створення графічних ефектів, управління фізичною поведінкою об'єктів або реалізації користувацької AI-системи для персонажів гри.

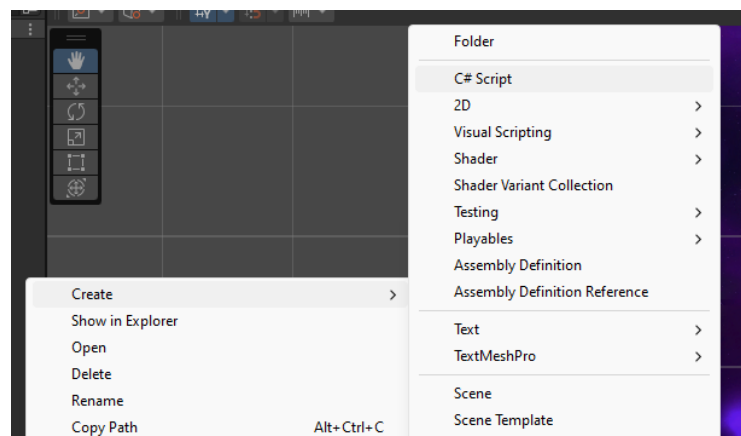
Поведінка ігрових об'єктів контролюється за допомогою компонентів (Components), які приєднуються до них. Незважаючи на те, що вбудовані компоненти Unity можуть бути дуже різнобічними, часто необхідно реалізувати власні особливості геймплею. Unity дозволяє створювати свої компоненти, використовуючи скрипти. Вони дозволяють активувати ігрові події, змінювати параметри компонентів, і відповідати на введення користувача в будь-який спосіб.

На відміну від інших ассетів, скрипти зазвичай створюються безпосередньо в Unity. Ви можете створити скрипт, використовуючи меню Create у лівому верхньому куті панелі Project або вибравши Assets → Create → C# Script у головному меню.

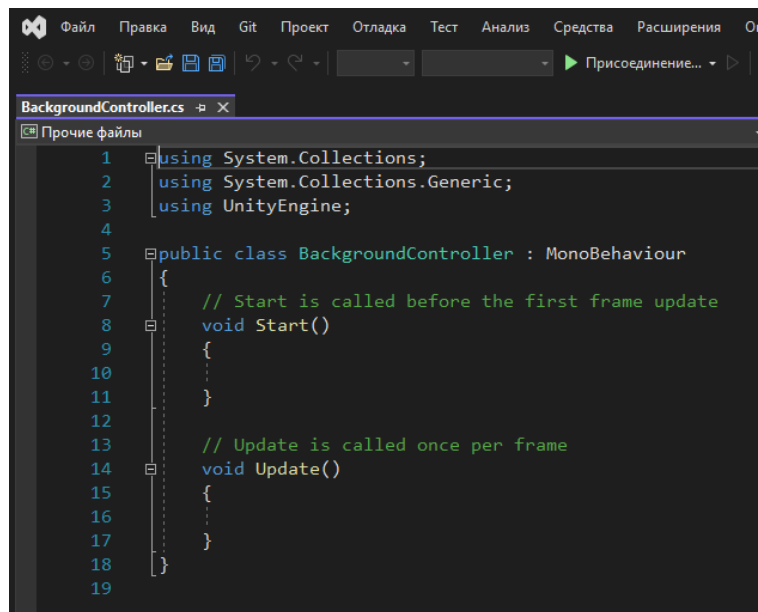
Створимо скрипт. Для цього в дереві проекту в папці Assets створимо нову папку Scripts



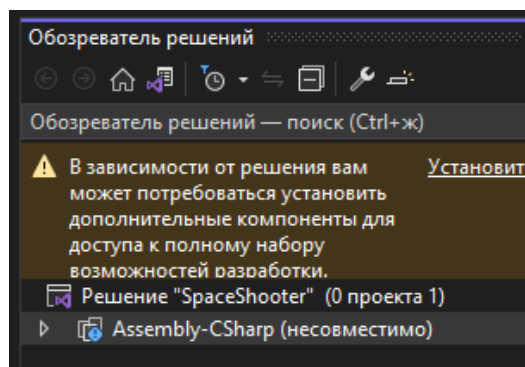
У папці Scripts створимо скрипт. Правою кнопкою миші всередині цієї папки вибираємо Create → C# Script. Вводимо назву BackgroundController.



Подвійне клацання миші по іконці створеного скрипту відкриває Visual Studio і в ньому шаблон створеного скрипту:



Перевірте, щоб у налаштуваннях Unity стояла Visual Studio: Edit→Preferences... →External tools→External Script Editor – вибрати Visual Studio, зберегти проект. Закрити Visual Studio та перезапустити Unity. Подвійне клацання миші по іконці скрипта, і він відкриється у Visual Studio. При відкритті Visual Studio може з'явитися повідомлення, що потрібно встановити додаткові компоненти. Встановлюємо їх.



Скрипт взаємодіє із внутрішніми механізмами Unity за рахунок створення класу, успадкованого від вбудованого класу, званого MonoBehaviour. Ви можете думати про клас як про свого роду план створення нового типу компонента, який може бути прикріплений до ігрового об'єкта. Щоразу, коли ви приєднуєте скриптовий компонент до ігрового об'єкта, створюється новий екземпляр об'єкта, визначений планом. Ім'я класу береться з імені, яке ви вказали під час створення файлу. Ім'я класу та ім'я файлу повинні бути однаковими для того, щоб скриптовий компонент міг бути приєднаний до ігрового об'єкта.

Функція Update – це місце для розміщення коду, який оброблятиме оновлення кадру для ігрового об'єкта. Це може бути рух, спрацювання дій і реакція на введення користувача, в основному все, що повинно бути оброблено з часом в ігровому процесі. Щоб дозволити функції Update виконувати свою роботу, часто буває корисно ініціалізувати змінні, рахувати властивості та здійснити зв'язок з іншими ігровими об'єктами до того, як будуть здійснені будь-які дії.

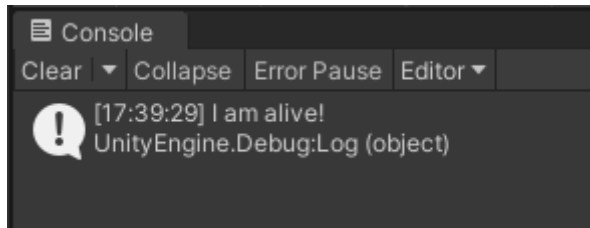
Функція Start буде викликана Unity до початку ігрового процесу (тобто до першого виклику функції Update), і це ідеальне місце для виконання ініціалізації.

Debug.Log – це команда, яка просто виводить повідомлення на консольне виведення Unity.

Наприклад, код:

```
void Start()
{
    Debug.Log("I am alive!");
}
```

Якщо ви натиснете зараз Play, ви побачите повідомлення внизу основного вікна редактора Unity у вікні Console (меню: Window > Console).



Скрипт Unity не схожий на традиційну ідею програми, де код працює постійно в циклі, поки не завершить своє завдання. Натомість Unity періодично передає керування скрипту при виклику певних оголошених у ньому функцій. Як тільки функція завершує виконання, керування повертається назад до Unity. Ці функції відомі як функції подій, тобто їх активує Unity у відповідь на події, що відбуваються у процесі гри. Unity використовує схему іменування, щоб визначити, яку функцію викликати певної події. Наприклад, ми вже розглянули функцію Update (викликається перед зміною кадру) та функцію Start (викликається перед першим кадром об'єкта). У Unity є набагато більше функцій подій.

Додаємо наступний код:

```
public class BackgroundController: MonoBehaviour
{
    [SerializeField]
    private SpriteRenderer sprite;

    private float speed = 2.5f;
    private float positionMin;
    private Vector2 positionRestart;

    // Start is called before the first frame update
    void Start()
    {
        positionRestart = transform.position;
        positionMin = sprite.bounds.size.y * 2 - positionRestart.y;
        //Debug.Log("I am alive!");
    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate(Vector3.down * speed * Time.deltaTime);
        if (transform.position.y <= -positionMin)
            transform.position = positionRestart;
    }
}
```

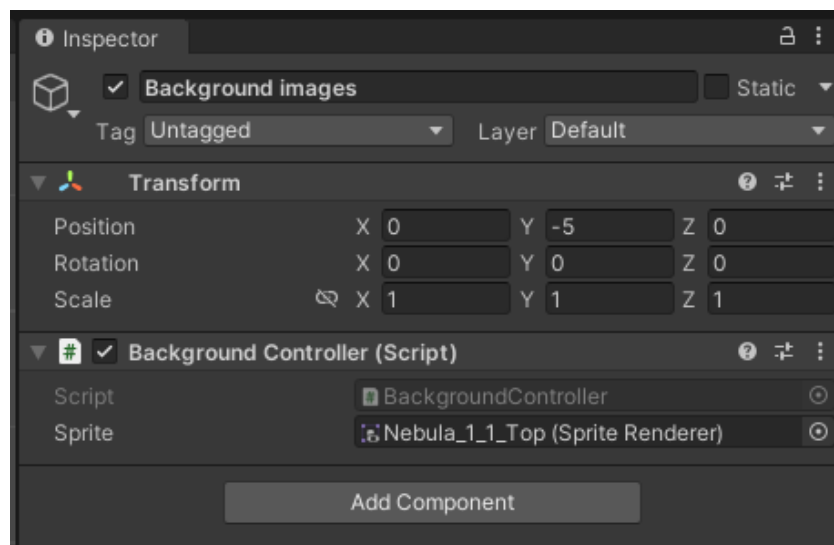
Unity дозволяє змінювати значення змінних скрипта у запусненій грі. Це дуже корисно, щоб побачити ефекти від змін відразу ж, без зупинки та перезапуску. Коли

програвання закінчується, значення змінних скидаються до стану, який вони мали до натискання кнопки Play.

Атрибути (Attributes) – це маркери, які можуть бути розміщені перед класом, властивостями або функціями в скрипті, щоб вказати особливу поведінку. Наприклад, атрибут `HideInInspector` може бути доданий перед оголошенням властивості для запобігання відображенню цієї властивості в інспекторі, навіть якщо вона публічна. У C# він міститься між квадратними дужками.

У C# ви повинні оголосити змінну загальнодоступною, щоб побачити її в інспекторі. Якщо ви також хочете, щоб Unity серіалізувала `private` поля (побачити їх в Інспекторі), ви можете додати до цих полів атрибут `SerializeField`.

Застосуємо створений скрипт на об'єкт `Background images` шляхом перетягування іконки скрипта до властивостей зазначеного об'єкта. Заповнимо значенням поле `Sprite`.



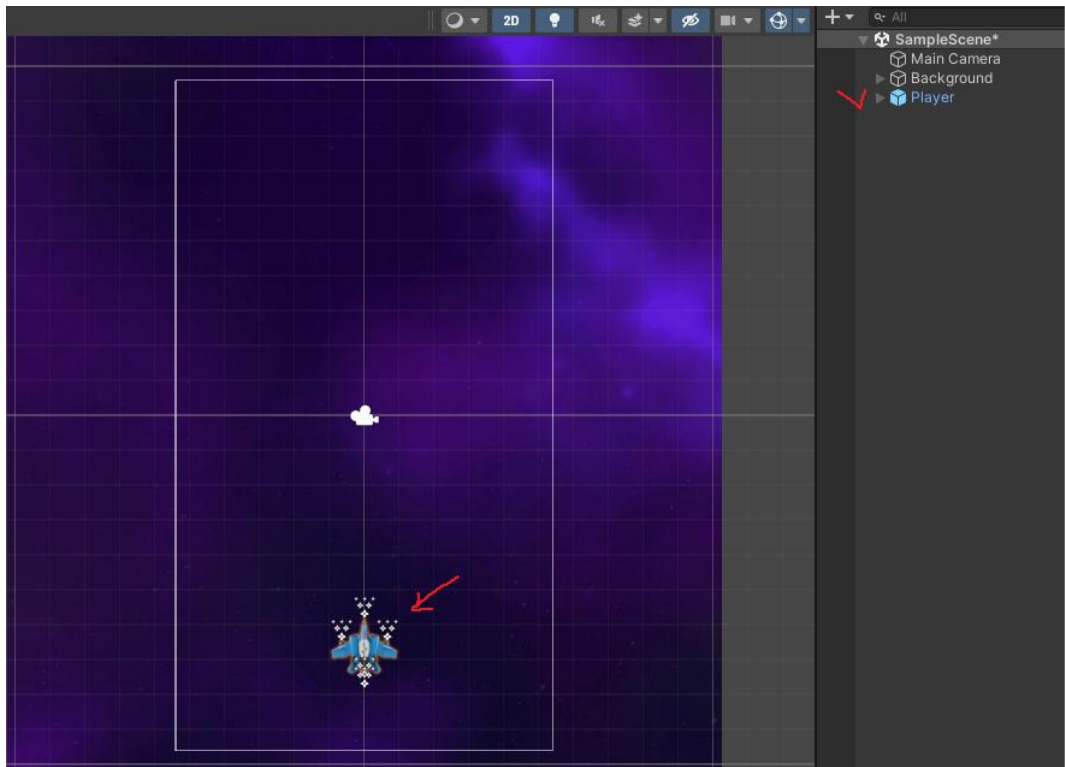
Запустимо проєкт гри. Тепер маємо нескінченне тло.

Крок 3. Додавання ігрового об'єкту

Додамо на сцену гравця як космічний корабель.

Система Prefab Unity дозволяє створювати, налаштовувати та зберігати `GameObject` у комплекті з усіма його компонентами, значеннями властивостей та дочірніми ігровими об'єктами як повторно використовуваний актив.

У нашому проєкті такий Prefab ми візьмемо із встановленого асету. Для цього в панелі Project заходимо до папки `Space Shooter Template FREE` → `Prefabs` та перетягуємо на сцену Prefab під назвою `Player`.



Prefab під назвою Player вже має ряд компонентів, які додав розробник асета, що використовується. Зокрема, на об'єкт Player навішено три скрипти.: Player, PlayerMoving та PlayerShooting.



У скрипті Player є параметр Destruction FX - префаб відеоефекту, що генерується після смерті гравця.

У скрипті PlayerMoving є параметр Borders, що налаштовується - відступ від кожної межі екрана, який гравець не може проходити.

У скрипті PlayerShooting є параметри Weapon Power – тип зброї гравця, що впливає на стрільянину (за замовчуванням від 1 до 4) і Guns – гармати і променеві об'єкти в ієрархії гравців.

Компонент Sprite Renderer візуалізує двовимірні графічні об'єкти Sprite та керує їхнім візуальним відображенням на сцені.

При створенні спрайту (GameObject → 2D Object → Sprite) Unity автоматично створює GameObject із прикріпленим компонентом Sprite Renderer. Ви також можете додати компонент до існуючого ігрового об'єкта через Компоненти (меню Component → Rendering → Sprite Renderer).

Розглянемо властивості компонента.

Властивість	Опис
Sprite	Визначте, яку текстуру Sprite має відображати компонент. Клацніть маленьку точку праворуч, щоб відкрити вікно вибору об'єктів, та виберіть зі списку доступних ресурсів спрайтів.

Color	Визначте колір вершин спрайту, який підфарбовує або перефарбовує зображення. Використовуйте палітру кольорів, щоб встановити колір вершин текстури, що візуалізується Sprite.
Flip	Перевертає текстуру Sprite вздовж зазначеної осі. Це не змінює позицію перетворення GameObject.
Material	Визначте Матеріал, який використовується для візуалізації текстури спрайту.
Draw Mode	Визначте, як масштабується спрайт при зміні розмірів. Виберіть один з наступних варіантів у списку, що розкривається. <i>Simple</i> – все зображення масштабується при зміні розмірів. Це опція за замовчуванням. <i>Sliced</i> – виберіть цей режим, якщо Sprite має 9-sliced (включає поділ зображення на дев'ять частин, щоб при зміні розміру Sprite, різні частини масштабуються різними способами, щоб спрайт залишався пропорційним). <i>Size</i> ('Sliced' or 'Tiled') – введіть нову ширину та висоту спрайту, щоб правильно масштабувати спрайт, що складається з 9 фрагментів. <i>Tiled</i> – за замовчуванням у цьому режимі середина спрайту з 9 фрагментами стає мозаїчною, а не масштабується за зміни його розмірів. Використовуйте мозаїчний режим, щоб керувати мозаїчною поведінкою спрайту. <i>Continuous</i> – це режим мозаїки за промовчанням. У безперервному режимі середня частина вирівнюється при зміні розмірів спрайту. <i>Adaptive</i> – в адаптивному режимі текстура Sprite розтягується при зміні її розмірів, як і простому режимі. Коли масштаб змінених розмірів збігатиметься зі значенням розтягування, середня частина почне викладатися мозаїкою. <i>Stretch Value</i> – використовуйте повзунок, щоб встановити значення від 0 до 1. Максимальне значення дорівнює 1, що відповідає подвійному масштабу вихідного спрайту.
Mask Interaction	Встановіть поведінку засобу візуалізації спрайтів при взаємодії з маскою спрайту <i>None</i> – засіб візуалізації спрайтів не взаємодіє з жодними масками спрайтів у сцені. Це опція за замовчуванням. <i>Visible Inside Mask</i> – спрайт видно там, де на нього накладається маска спрайту, але не за її межами. <i>Visible Outside Mask</i> – спрайт видно зовні маски спрайту, але не всередині неї. Маска спрайту приховує частини спрайту, які вона накладає.
Sprite Sort Point	Виберіть між центром спрайту або його точкою обертання при розрахунку відстані між спрайтом та камерою.
Material	Визначте Матеріал, який використовується для візуалізації текстури спрайту.
Sorting Layer	Встановіть шар сортування спрайту, який керує його пріоритетом під час рендерингу. Виберіть існуючий шар сортування зі списку або створіть новий шар сортування.
Order In Layer	Встановіть пріоритет рендерингу спрайту в межах його шару сортування. Спочатку візуалізуються спрайти з нижчими номерами, а спрайти з вищими номерами перекриваються спрайтами, розташованими нижче.

Компонент Box Collider 2D – це колайдер, який взаємодіє з системою 2D-фізики. Це прямокутник у формі з певним положенням, шириною та висотою в локальному просторі координат Sprite. Прямокутник вирівняний по осі, яка паралельна осям X чи Y локального простору.

Колайдер – невидима форма, що використовується для обробки фізичних зіткнень. Колайдер не обов'язково має бути саме тієї ж форми, що й сітка об'єкта.

Виявлення зіткнень – автоматичний процес, що виконується Unity, який визначає, чи GameObject з Rigidbody, що рухається, і колайдер увійшов в контакт з будь-яким іншим колайдером.

Система сценаріїв може визначати виникнення колізій та ініціювати дії за допомогою функції OnCollisionEnter. Однак також можна просто виявити, коли один колайдер входить

у простір іншого, не створюючи зіткнення. Колайдер, налаштований як тригер (використовуючи властивість `Is Trigger`), не веде себе як твердий об'єкт і просто пропускає інші колайдери. Коли колайдер входить у своє місце, тригер викликає функцію `OnTriggerEnter` для об'єкта тригера `scripts`.

`Composite Collider 2D` компонент – це колайдер, який взаємодіє із системою 2D-фізики. На відміну від більшості колайдерів, він визначає внутрішню форму. Натомість він поєднує форми будь-якого `Box Collider 2D` або `Polygon Collider 2D`, які ви налаштували для використання. `Composite Collider 2D` використовує вершини (геометрію) будь-якого з цих колайдерів та об'єднує їх разом у нову геометрію, керовану самим `Composite Collider 2D`.

Жорсткі тіла дозволяють вашим `GameObjects` діяти під контролем фізики. `Rigidbody` може сприймати сили та крутний момент, щоб ваші об'єкти рухалися реалістично. Будь-який `GameObject` повинен містити `Rigidbody`, щоб на нього впливала гравітація, щоб він діяв під дією додаткових сил за допомогою сценаріїв або взаємодіяв з іншими об'єктами.

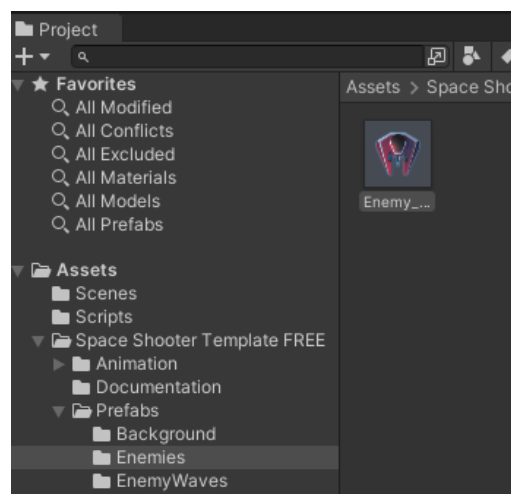
Компонент `Rigidbody 2D` поміщає об'єкт під управління фізичного двигуна. 2D об'єкти можуть рухатися тільки в площині `XY` і можуть обертатися навколо осі, перпендикулярної цій площині.

Зазвичай компонент `Transform` редактора Unity визначає, як `GameObject` (і його дочірні об'єкти `GameObject`) позиціонується, обертається та масштабується усередині Сцени. Коли він змінюється, він оновлює інші компоненти, які можуть оновлювати такі речі, як де вони відображаються або де колайдери.

Крок 4. Додавання ворогів

Наступним кроком буде додавання на сцену ворогів та інших об'єктів.

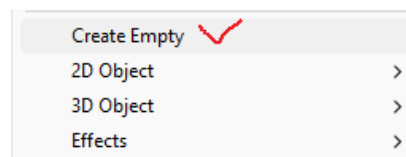
У встановленому асеті в папці `Prefab` можна знайти створений префаб ворога:



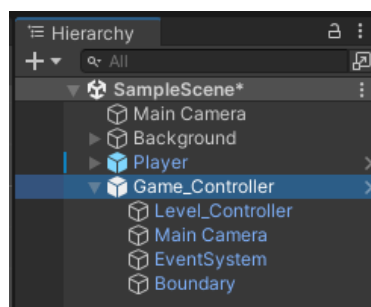
Якщо цей префаб просто перетягнути на сцену та запустити гру, то ворог перебуватиме там, де його поставили. Навіть якщо написати для нього скрипт, який його пересуватиме, то один ворог це мало.



В ідеалі нам потрібен потік таких ворогів, які з'являтимуться через деякий час у різних координатах. Для цього створюється порожній об'єкт GameObject і на нього навішується скрипт, який генеруватиме появу ворогів. Або створюється порожній об'єкт GameObject із дочірніми об'єктами, і на ці об'єкти додаються відповідні скрипти.



Так як у цій роботі ми використовуємо встановлений ассет, а він вже містить такий об'єкт, то створювати порожній об'єкт не будемо. Перетягнемо на сцену з папки Prefab об'єкт під назвою Game_Controller. У нашому випадку він міститиме кілька дочірніх об'єктів:

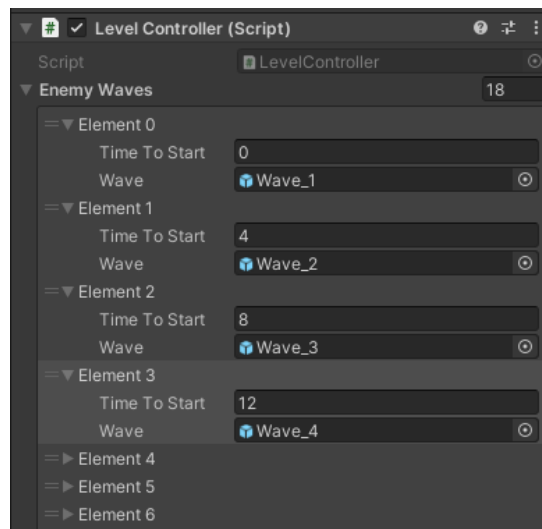


Boundary – об'єкт вказує на об'єкти, що перетинають лінію ігрового поля, знищує чи деактивує їх.

Level_Controller – контролює ігровий процес цього рівня, зокрема виробляє ворогів, бонуси та фонові об'єкти.

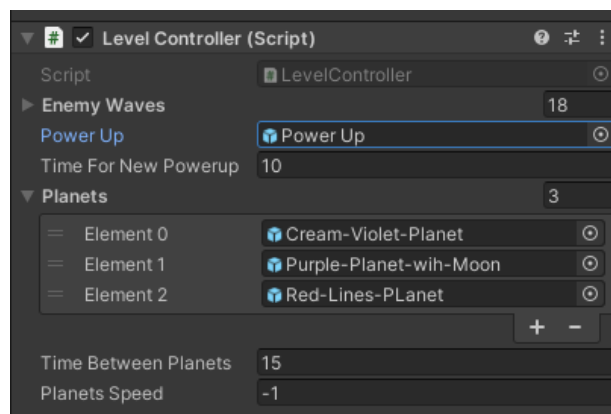
Для запуску рівня необхідно задати поля об'єкта Level_Controller, а саме Enemy Waves. У категорії «Enemy Waves» слід зазначити кількість ворожих хвиль, які з'являться на цьому рівні. Також потрібно вказати точку відліку від початку гри, коли з'являється

кожна хвиля (у секундах). Додаються заздалегідь підготовлені хвилі з папки Prefabs → EnemyWaves.

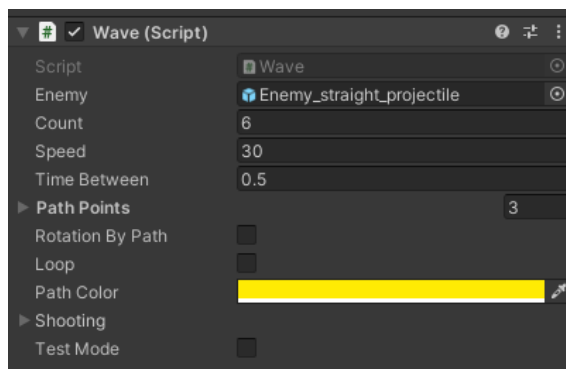


У Time For New Powerup встановіть, як часто повинен з'являтися бонус Power up.

У «Planets» налаштуйте планети-відьми, які ви збираєтеся використовувати, і як часто вони мають з'являтися.



Enemy Waves містить скрипт Wave, який генерує певну кількість ворогів протягом певного періоду часу і запускає їх по певному шляху.

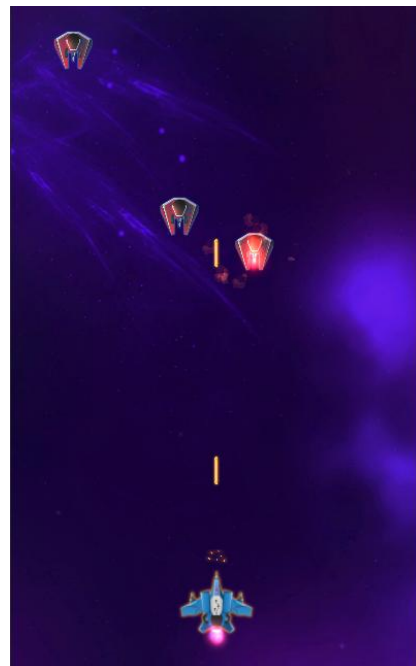
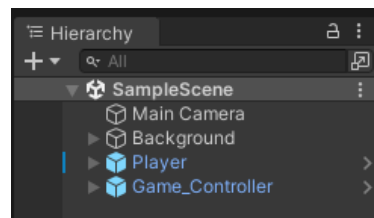


Поля скрипту «Wave»:

- «Enemy» – префаб ворога, який буде створений під час цієї хвилі;
- «Count» – кількість ворогів у цій хвилі;
- «Speed» – швидкість, з якою ворог проходить шлях;

- «Time Between» – період між появою ворогів;
- «Path Points» – точки, які пройде ворог під час проходження шляху;
- «Rotation Bypass» – чи обертатиметься ворог під час проходження шляху.
- «Loop» – якщо цикл активний, після завершення шляху ворог буде перегенерований у початковій точці;
- «Path Color» – колір, яким буде відзначений шлях у Редакторі;
- «Shooting»: «Shot chance» – ймовірність пострілу по противнику при проходженні шляху, а також мінімальний і максимальний час, коли противник спробує вистрілити.
- «Test Mode» – якщо тестовий режим активний, хвиля генеруватиметься постійно з інтервалом у 3 секунди. Ви можете використовувати тестовий режим для налаштування нової хвилі.

Тепер, коли на сцені ми розташували нескінченне тло, гравця та об'єкт, що відповідає за генерацію ворогів і планет можемо запустити гру на виконання:



2. Завдання до практичної роботи 1

Створити гру, описану в практичній роботі.

Практична робота 2. 2D гра. Основні об'єкти

Мета роботи: познайомитися з основними об'єктами на сцені в Unity. Навчитися додавати об'єкти та налаштовувати їх функціональність в програмі Unity.

1. Теоретичний матеріал

Клас `GameObject` в Unity використовується для представлення всього, що може існувати у сцені. Кожен об'єкт у вашій грі – це `GameObject`, від персонажів та колекційних предметів до джерел світла, камер та спеціальних ефектів.

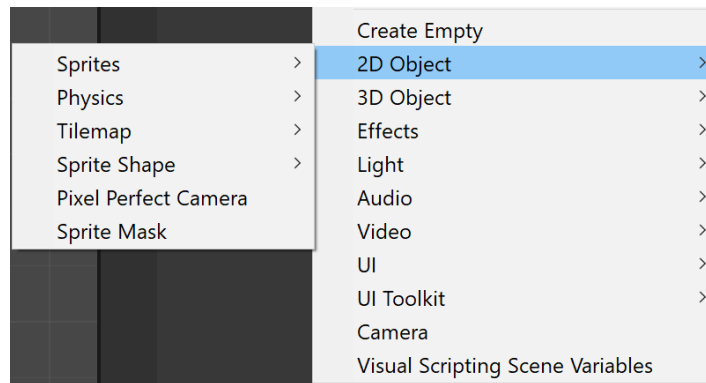
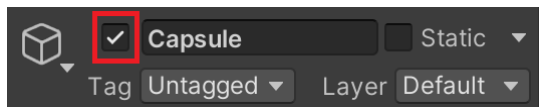


Рисунок 2.1 – Приклад `GameObject`

Ігрові об'єкти активні за замовчуванням, але їх можна деактивувати, що призведе до вимкнення всіх компонентів, прикріплених до ігрових об'єктів. Зазвичай це означає, що він стане невидимим і не отримуватиме звичайних зворотних викликів або подій, таких як `Update` або `FixedUpdate`. Статус ігрового об'єкта активний, представлений прапорцем ліворуч від імені об'єкта.



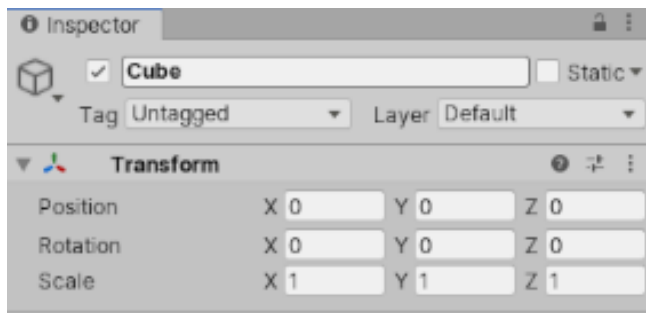
Тег – це довідкове слово, яке можна присвоїти одному або декільком `GameObjects`. Наприклад, ви можете визначити тег «Гравець» для персонажів, контрольованих гравцем, та тег «Ворог» для персонажів, які не контролюються гравцем. Ви можете визначити предмети, які гравець може збирати у Сцені з позначкою "Колекційний".

Теги допомагають ідентифікувати ігрові об'єкти для сценаріїв. Теги корисні для тригерів у колайдері, за допомогою яких можна з'ясувати, чи взаємодіє гравець, наприклад, з ворогом, реквізитом або предметом колекціонування.

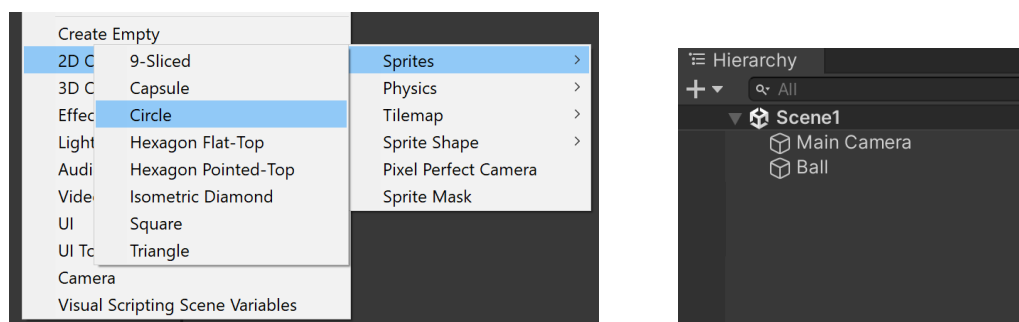
Самі по собі `GameObject` нічого не роблять, але діють як контейнери для компонентів, де реалізована ця функціональність.

Щоб додати ігровому об'єкту властивості, необхідні для того, щоб він став джерелом світла, деревом або камерою, потрібно додати до нього компоненти. Залежно від того, який об'єкт ви хочете створити, ви додаєте до `GameObject` різні комбінації компонентів.

До `GameObject` завжди приєднано компонент `Transform` (для подання положення та орієнтації), і видалити його неможливо. Інші компоненти, які надають об'єкту його функціональність, можуть бути додані з меню Компонент редактора або скрипта.



Створимо новий проєкт 2D-гри. У новому проєкті перейменуємо назву сцени на Scene1. Створимо на сцені простий GameObject у вигляді кола (2D Object → Sprites → Circle) та назвемо його, наприклад, Ball. Контекстне меню, де можна обрати вид GameObject, що створюється, можна викликати правою кнопкою миші в панелі Hierarchy або в меню GameObject.



Компонент Rigidbody – це основний компонент, що включає фізичну поведінку для об'єкта. З прикріпленим Rigidbody об'єкт негайно почне реагувати на гравітацію. Якщо додано один або кілька компонентів Collider, то при колізіях (зіткненнях) об'єкт пересуватиметься.

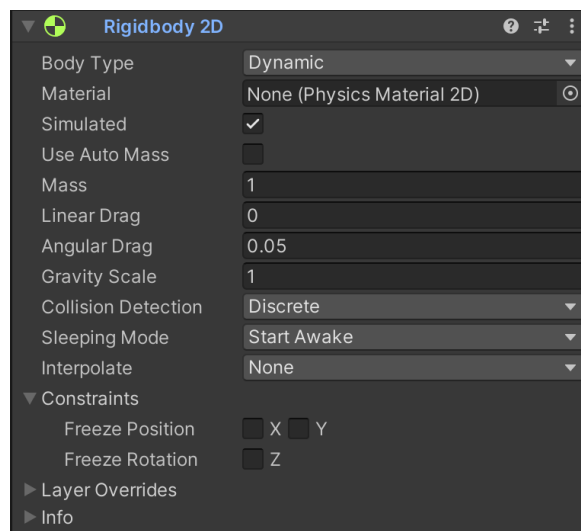


Рисунок 2.2 – Вікно компоненту Rigidbody

Для того, щоб об'єкт Ball став фізичним, додаємо до нього компонент Rigidbody. Для цього в панелі Inspector натискаємо кнопку Add Component та обираємо Rigidbody 2D.

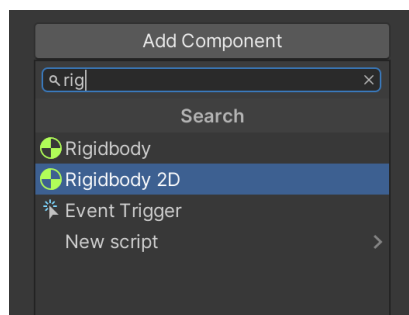


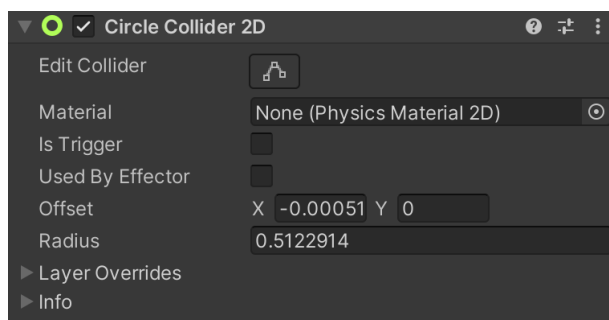
Рисунок 2.3 – Пошук у вікні компоненту Rigidbody

Якщо зараз запустити гру, об'єкт Ball почне падати вниз та вийде за межі камери, так як на нього впливає гравітація.

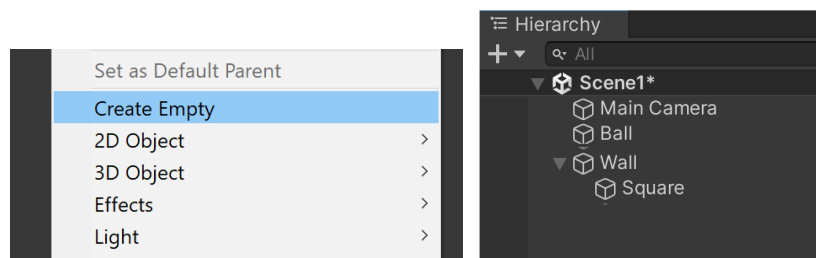
Компоненти Collider визначають форму GameObject для цілей фізичних зіткнень. Коллайдер не обов'язково має бути такої ж форми, як сітка GameObject. Приблизне наближення сітки часто є більш ефективним і нерозрізним у ігровому процесі.

Найпростіші коллайдери є примітивними типами коллайдерів. У 2D ви можете використовувати Box Collider 2D, Circle Collider 2D тощо. Ви можете додати будь-яку кількість їх до одного GameObject, щоб створити складені коллайдери.

Для того, щоб об'єкт Ball міг взаємодіяти з іншими об'єктами, додаємо до нього компонент Collider. Для цього в панелі Inspector натискаємо кнопку Add Component та обираємо Circle Collider 2D.

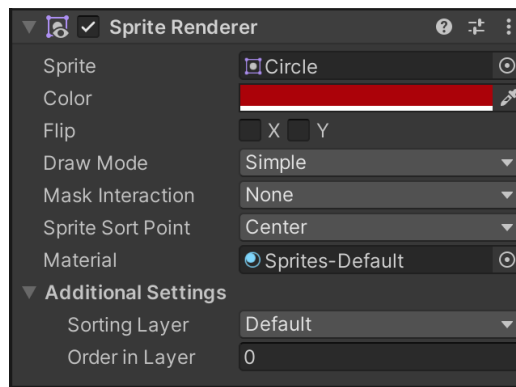


Щоб об'єкт Ball не летів у нескінченність вниз, додаємо на сцені новий об'єкт. Створимо пустий об'єкт GameObject → Create Empty (правою кнопкою миші в панелі Hierarchy). Назвемо його Wall. В ньому створимо ще один об'єкт 2D Object → Sprites → Square.

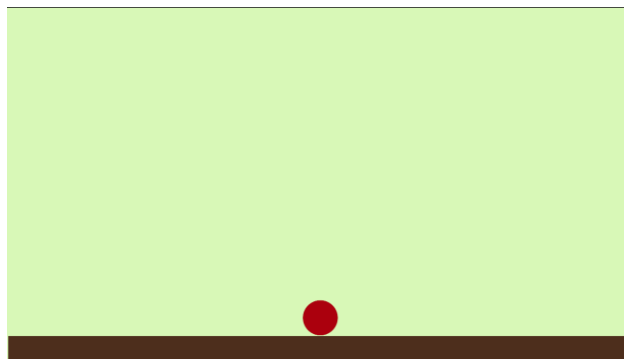


Додаємо до об'єкта Wall компонент Collider. Для цього в панелі Inspector натискаємо кнопку Add Component та обираємо Box Collider 2D.

Замінімо колір створених об'єктів Ball та Square. Для цього в панелі Inspector відкриваємо вже існуючий компонент Sprite Renderer та у властивості Color змінюємо колір на потрібний.



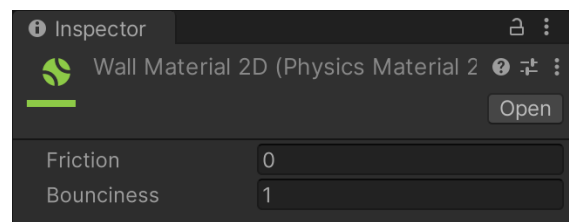
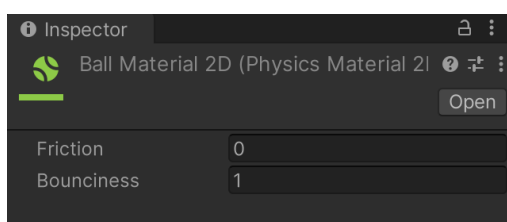
Запустимо гру. Тепер об'єкт Ball зупиниться на верхній поверхні об'єкта Wall.



Додаймо до об'єктів Ball та Wall можливість фізично взаємодіяти. Для цього в їхніх колайдерах необхідно додати фізичний матеріал.

Physics Material 2D використовується для налаштування тертя та відскакувань, які відбуваються між 2D фізичними об'єктами при їх зіткненні. Створити 2D фізичний матеріал можна через меню Assets (Assets → Create → Physics Material 2D).

Створимо фізичний матеріал для об'єктів Ball та Wall. З початку в папці Assets створимо нову папку Material. В цій папці створимо два матеріали Ball Material 2D та Wall Material 2D.



Фізичний матеріал має такі властивості:

- Friction – коефіцієнт тертя для даного колайдера.
- Bounciness – ступінь відскоку зіткнень від поверхні. Значення 0 означає, що немає відскоку, у той час як значення 1 означає хорошу відскакуваність без подальшої втрати енергії.

Для використання 2D фізичного матеріалу просто перетягніть його на компонент колайдера в інспекторі.

Якщо тепер запустити гру, об'єкт Ball буде відскакувати перпендикулярно об'єкту Wall. Несемо різноманітність. Створимо скрипт для об'єкта Ball з назвою Ball.

```

void Start()
{
    const float MinImpulse = 3f;
    const float MaxImpulse = 15f;
    float angle = Random.Range(0, 2*Mathf.PI);

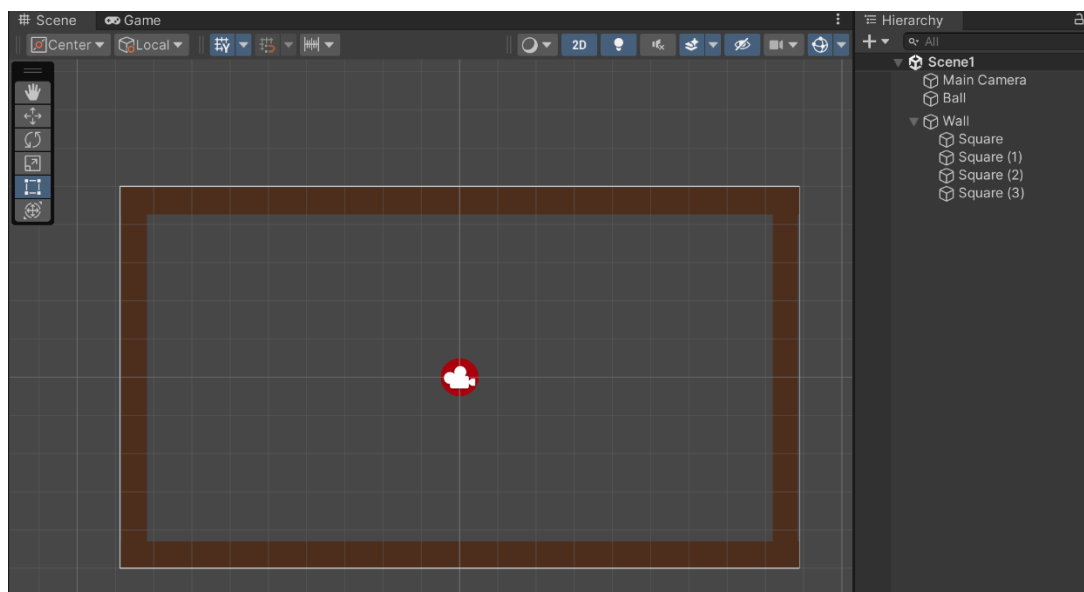
    Vector2 direction = new Vector2(Mathf.Cos(angle), Mathf.Sin(angle));
    float magnitude = Random.Range(MinImpulse, MaxImpulse);

    GetComponent<Rigidbody2D>().AddForce(direction* magnitude, ForceMode2D.Impulse);
}

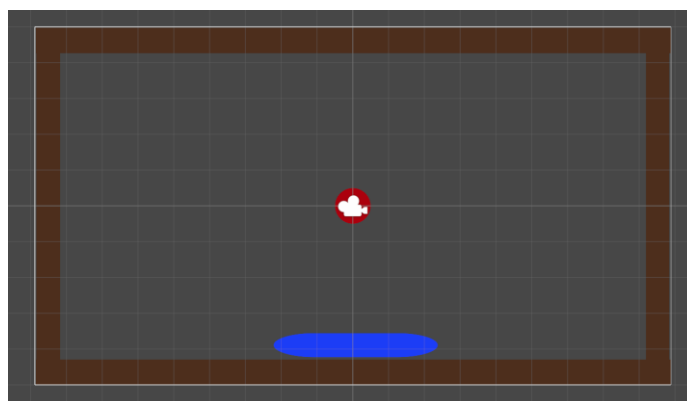
```

Якщо тепер запустити гру, об'єкт Ball буде відскакувати від об'єкта Wall з різною силою та під різними кутами.

Щоб Ball після відскоку не вилітав за ліву та праву границю камери, необхідно об'єкт Wall доповнити ліворуч, праворуч та нагорі об'єктами Square. Кожному створеному об'єкту Square додати компонент і фізичний матеріал Wall Material 2D.



Додаймо до гри платформу, яка буде рухатися вліво та вправо в нижній частині екрану. Створимо ще один об'єкт 2D Object → Sprites → Capsule та назвемо його Racket. Додаймо до нього компонент Capsule Collider 2D. Змінимо колір на інший. Створимо новий матеріал Racket Material 2D з такими ж властивостями, як і у попередніх. Додаймо матеріал до коллайдера.



Створимо новий скрипт для об'єкту Racket з такою ж назвою:

```
[SerializeField]
private float speed = 10f;

private float lastMouseX;

/// <summary>
/// Start is called before the first frame update
/// </summary>
void Start()
{
    lastMouseX = Input.mousePosition.x;
}

/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    var position = transform.position;

    Vector2 direction = Vector2.zero;

    if (Input.mousePosition.x > lastMouseX)
        direction = Vector2.right * speed;
    else if (Input.mousePosition.x < lastMouseX)
        direction = Vector2.left * speed;

    lastMouseX = Input.mousePosition.x;

    position.x += direction.x * Time.deltaTime;
    transform.position = position;
}
```

Запустимо гру. Об'єкт Racket буде рухатися в напрямку покажчика миші. Таким чином, користувач може управляти об'єктом.

Для складності гри можна додатково додати декілька об'єктів Ball різного розміру.

2. Завдання до практичної роботи 2

Створити 2D-гру, просту версію гри «Пінг-понг». Розібратися з налаштуваннями компонентів Rigidbody та Collider.

Практична робота 3. 2D-гра. Анімація спрайтів

Мета роботи: познайомитися зі створенням анімацій для 2D-гри на Unity. Навчитися додавати персонажів та інші ресурси на сцену, створювати анімацію спрайтів та персонажів.

1. Теоретичний матеріал

Робочий процес при створенні двовимірної та тривимірної графіки в Unity приблизно однаковий і включає імпорт графічних ресурсів, перетягування їх у сцену і написання сценаріїв, які потім приєднуються до об'єктів. Основний вид ресурсів, необхідний для створення двовимірної графіки, називається спрайтом.

Спрайти (sprites) являють собою двовимірні зображення, що відображаються безпосередньо на екрані, в той час як такі ж зображення, що відображаються на поверхні тривимірних моделей, називаються текстурами.

Створимо новий проєкт 2D-гри. У новому проєкті перейменуємо назву сцени на Scene1. Створимо папку Sprits, де будемо зберігати спрайти героїв гри. Додамо до цієї папки зображення з персонажем.

Іноді текстура спрайту містить лише один елемент графіки, але часто набагато зручніше об'єднати кілька зображень, пов'язаних один з одним, в одне зображення. Наприклад, зображення може містити складові персонажа, як-от колеса машини, які рухаються незалежно від корпусу.

У двомірних іграх повсюдно поширені анімовані спрайти. Вони створюються малюванням кожного кадру анімації та подальшим відтворенням отриманої послідовності кадрів у Unity.

Анімація персонажу може зберігатися в наступних видах:

- Листи спрайтів;
- Кожний кадр спрайту в окремому файлі.
- Створити анімацію за допомогою скрипту.

Кадри можна імпортувати у вигляді окремих зображень, але їх зазвичай викладають у вигляді однієї картинки, яка називається *листом спрайтів* (sprite sheet). Листи спрайтів можна автоматично генерувати в Unity. При імпорті аркуша у списку Sprite Mode налаштувань спрайту слід вибрати варіант Multiple.



Рисунок 3.1 – Приклад листу спрайтів

Для зміни розміру персонажа виділяємо в папці проєкту файл спрайта, в Inspector змінимо значення в полі Pixels per unit. Зменшення числа – збільшує спрайт та навпаки.

Якщо файл постачався без файлу meta, то для Unity це просто картинка, яку потрібно розбити на кадри. Щоб це зробити, виділяємо в папці відповідний файл та в Inspector

натискаємо кнопку Sprite Editor. Unity Sprite Editor дозволяє витягувати елементи складеного зображення.

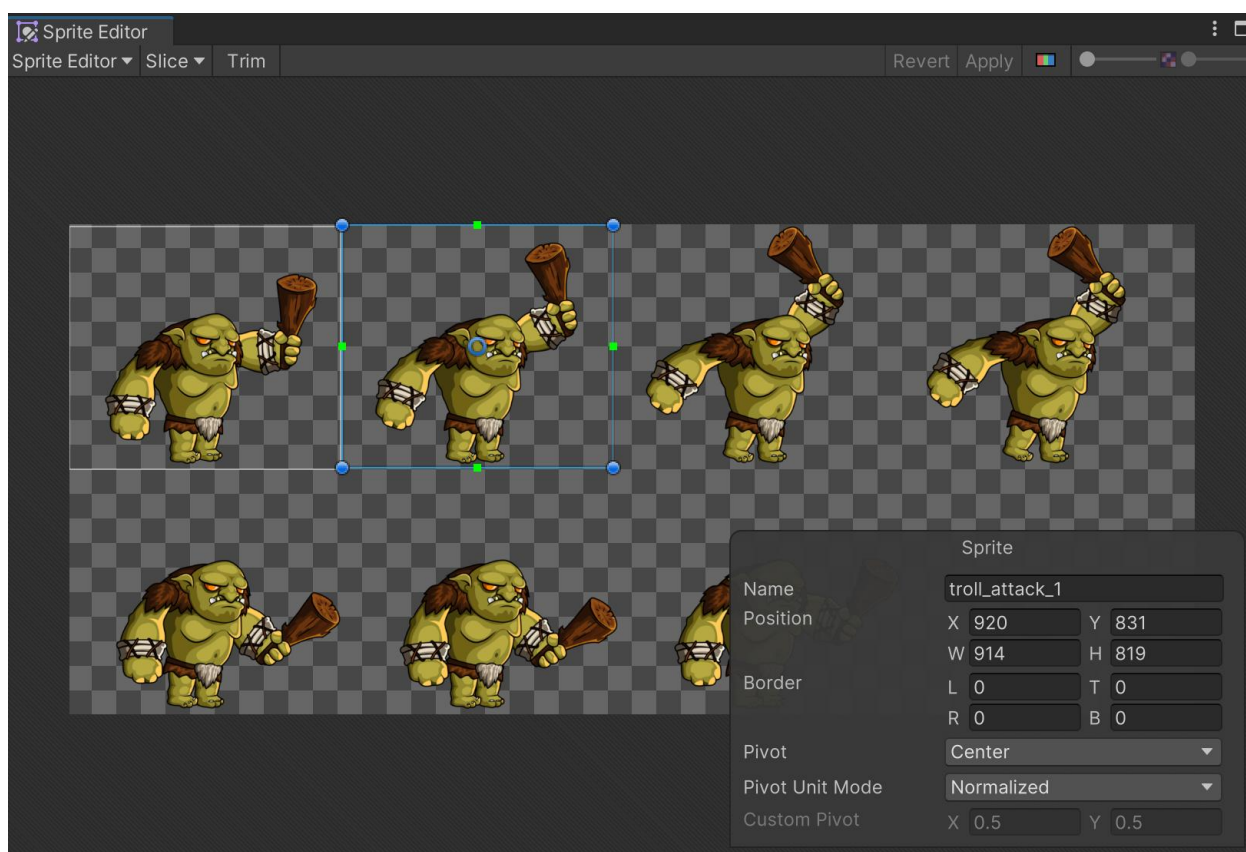


Рисунок 3.2 – Вікно Sprite Editor

Крім складеного зображення, ви побачите різні елементи керування у заголовку вікна редактора. Слайдер у правому верхньому куті керує наближенням, тоді як кнопка з кольоровими смужками ліворуч від нього перемикає режим відображення альфа-каналу та звичайний вигляд зображення.

Найважливіший елемент керування – це меню Slice у лівому верхньому кутку, яке надає опції для автоматичного нарізування елементів зображення.

Кнопки Apply та Revert дозволяють зберегти або скасувати зроблені зміни.

Найбільш прямим способом використання редактора є ідентифікація елементів вручну. Якщо натиснути на зображення, то з'явиться прямокутна область виділення з маркерами по кутах. Ви можете перетягнути маркери або краї прямокутника, щоб змінити його розмір навколо певного елемента.

Виділивши елемент, можна додати ще один, перетягнувши новий прямокутник в окрему частину зображення. Ви помітите, що коли ви виділите прямокутник, у нижньому правому куті вікна з'явиться панель:

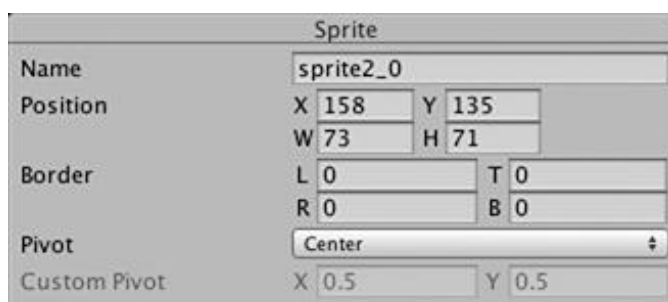


Рисунок 3.3 – Панель для ручного налаштування розміру кадру

Елементи керування на панелі дозволяють вибрати назву для графіки спрайтів та встановити положення та розмір прямокутника за його координатами. Ширина кордону для лівого, верхнього, правого та нижнього краю може бути вказана в пікселях. Також є налаштування для звороту спрайту, який Unity використовує як координату початку координат і головну «опорну точку» (Pivot) графіки.

Коли перетягуємо спрайт з анімацією вперше на сцену, то Unity запропонує зберегти анімацію. Зробимо це в створеній папці Animations. Назву файлу даємо як:

назваПерсонажа_типАнімації.anim

Наприклад: troll_walk.anim.

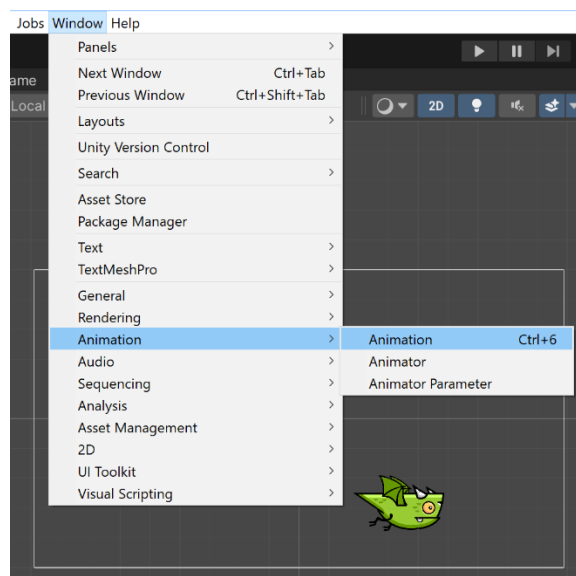
Аналогічно створимо всі необхідні анімації персонажа.

Розглянемо другий спосіб анімації, коли кадри анімації розташовуються кожний в окремому файлі.

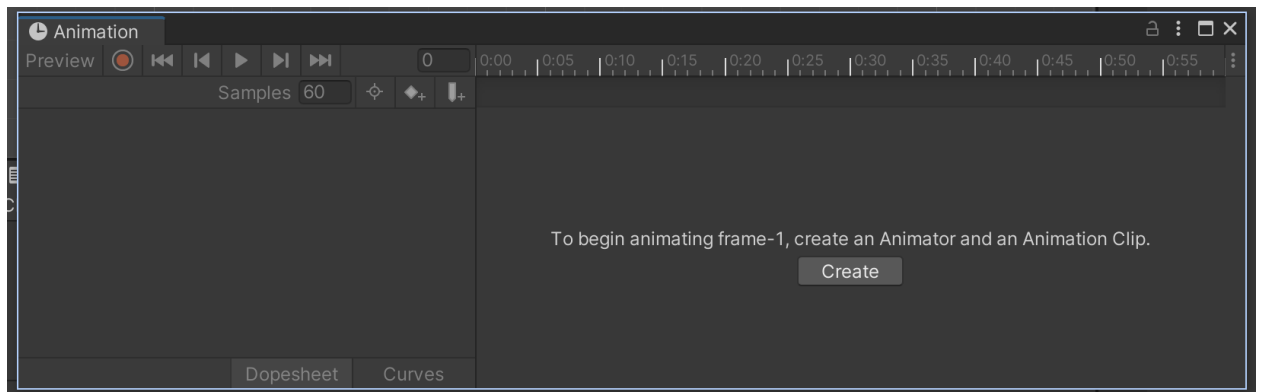
Наприклад маємо такі файли:



Перший кадр додаємо на сцену. Відкриваємо вікно Animation (Window → Animation → Animation).

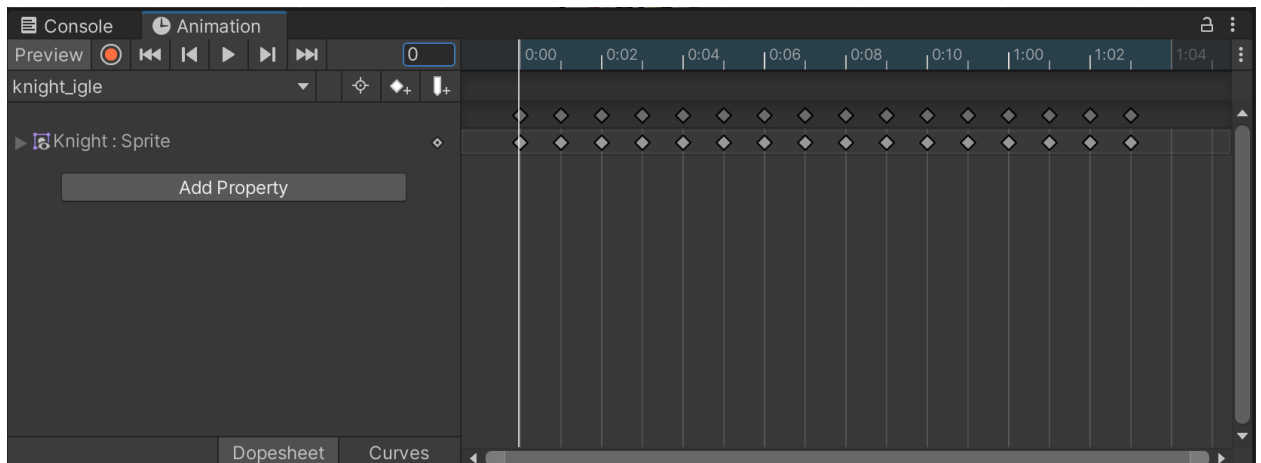


Вікно Animation має наступний вигляд:

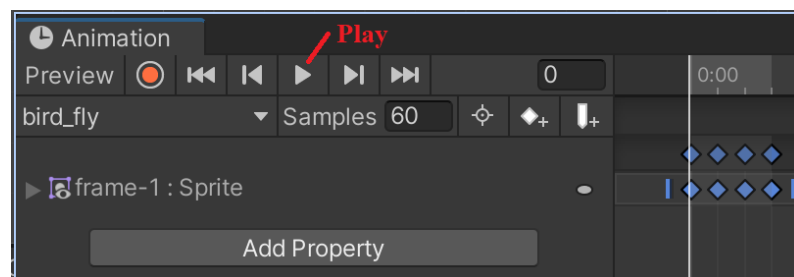


У вікні Animation натискаємо кнопку Create. Unity запропонує дати назву та зберегти файл анімації. Далі виділяємо всі спрайти, які є кадрами анімації, що створюється, та переносимо їх на шкалу timeline. Кожний кадр позначиться там окремим маркером.

Наприклад:



За замовчуванням кількість кадрів в секунду показується в полі Samples. Можна запустити анімацію, натиснувши кнопку Play в окні Animation.



Якщо анімація рухається надто швидко, її можна уповільнити, змінивши кількість кадрів на секунду, а саме зменшивши це число. Методом підбору знаходимо необхідну частоту кадрів, за якої анімація виглядає природною.

Наприклад маємо такі файли:



Для цього персонажа створена єдина анімація польоту у вигляді окремих спрайтів. Тоді можемо скористатися ще одним способом та створити анімацію з використанням скрипта.

Щоб перетворити спрайт на ігровий об'єкт, необхідно перетягнути перший спрайт анімації у вікно ієрархії, і тепер він доданий до сцени. Створимо для нього скрипт з наступним кодом:

```
public class Bird : MonoBehaviour
{
    [SerializeField]
    private Sprite[] sprites;

    private SpriteRenderer spriteRenderer;
    private int spriteIndex;

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

    void Start()
    {
        InvokeRepeating(nameof(AnimateSprite), 0.15f, 0.15f);
    }

    void Update()
    {
    }

    private void AnimateSprite()
    {
        spriteIndex++;

        if (spriteIndex >= sprites.Length)
        {
            spriteIndex = 0;
        }

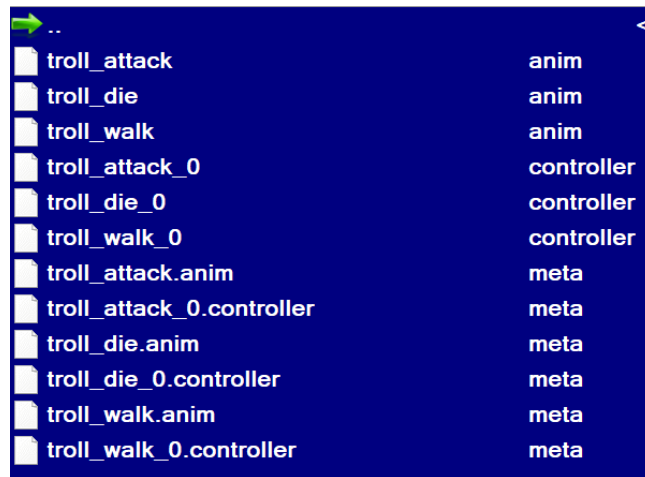
        if (spriteIndex < sprites.Length && spriteIndex >= 0)
        {
            spriteRenderer.sprite = sprites[spriteIndex];
        }
    }
}
```

Прикріпимо цей скрипт до доданого об'єкта та заповнимо масив `sprites` кадрами анімації. Якщо запустити проєкт, то буде програватися анімація в тому порядку, як кадри стоять в масиві `sprites`.

Метод `InvokeRepeating` викликає метод `methodName` після закінчення часу `time` секунд, потім повторює виклик кожні `repeatRate` секунд:

```
public void InvokeRepeating(string methodName, float time, float repeatRate);
```

При створенні анімації створилися файли контролери.



--	
troll_attack	anim
troll_die	anim
troll_walk	anim
troll_attack_0	controller
troll_die_0	controller
troll_walk_0	controller
troll_attack.anim	meta
troll_attack_0.controller	meta
troll_die.anim	meta
troll_die_0.controller	meta
troll_walk.anim	meta
troll_walk_0.controller	meta

Рисунок 3.4 – Приклад переліку файлів анімації персонажу

Залишемо один головний, а інші видалімо. Найчастіше головним буде контролер для анімації Idle (Стояти). Перейменуємо його на troll.controller. Відкриємо його в редакторі анімації.

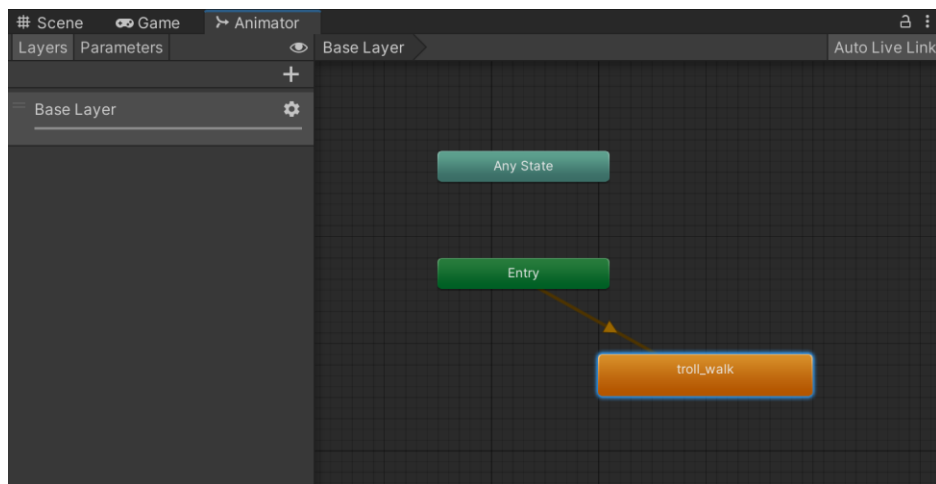


Рисунок 3.5 – Вікно Animator

В вікні редактору анімації – рух по полю: натискаємо колесо миші та рухаємо вправо, вліво, вгору, вниз.

Ресурс Animator Controller створюється в Unity і дозволяє впорядковувати та підтримувати набір анімацій для персонажа чи об'єкта. У більшості ситуацій нормально мати кілька анімацій і перемикається між ними, коли виникають певні умови гри.

Зазвичай персонаж має кілька анімацій, які відповідають різним діям, які можуть бути у грі. Наприклад, він може просто дихати і похитуватися в спокої, йти по команді і скидати руки в паніці при падінні з платформи. У дверей можуть бути анімації відкриття, закриття, заклинювання та зламу.

Однак, навіть якщо у вас є лише один анімаційний кліп, вам все одно потрібно розмістити його в контролері аніматора, щоб використовувати його на ігровому об'єкті.

Контролер має посилання на анімаційні кліпи, що використовуються в ньому, і керує різними станами анімації та переходами між ними за допомогою так званого кінцевого автомата, який можна розглядати як певну блок-схему або просту програму, написану мовою візуального програмування в Unity.

Основна ідея полягає в тому, що персонаж бере участь у якомусь конкретному виді дій у будь-який час. Типові дії включатимуть такі речі, як стійка на місці (idling), ходьба

(walking), біг (running), стрибок (jumping) тощо. Ці дії називають states, у тому сенсі, що персонаж перебуває у «стані», де він іде, стоїть чи робить ще щось.

Налаштування для наступного стану, в який гравець зможе перейти з поточного стану, називаються state transitions. Разом взяті набори станів, набори змінних та змінні пам'ятають поточну форму стану state machine.

Стани та переходи механізму станів можна уявити за допомогою графіка, де вузли будуть являти собою стани та дуги (стрілки між вузлами), що представляють переходи.

Вікно Animator складається з двох основних розділів: основної області макета з сіткою та лівої панелі «Layers & Parameters».

Ліва панель Parameters дозволяє створювати, переглядати та редагувати змінні. Ці змінні, які ви визначаєте, діють як вхідні дані в кінцевий автомат. Щоб додати параметр, клацніть значок «+» і виберіть тип параметра зі спливаючого меню. Щоб видалити параметр, виберіть параметр у списках і натисніть клавішу видалення.

Автомати стану складаються зі станів, переходів і подій, а менші автомати підстанів можна використовувати як компоненти у великих машинах.

Вузол Any State – це особливий стан, який існує завжди. Воно підходить для ситуацій, коли ви хочете перейти на певний стан, незалежно від поточного стану.

Вузол Entry використовується при переході в кінцевий автомат. Вузол входу буде оцінено та розгалужено до стану призначення відповідно до встановлених умов. Таким чином, вузол Entry може контролювати, в якому стані починається кінцевий автомат, оцінюючи стан ваших параметрів, коли починається кінцевий автомат.

Вузол Exit використовується для вказівки, що кінцевий автомат має завершити роботу.

Перетягнемо до вікна Animator анімації персонажа, що маємо. Додаймо змінні, які допоможуть перейти з одного стану в інший.

- Float speed;
- Trigger attack;
- Bool jump;

Щоб додати перехід на новий стан, правою кнопкою миші по поточному стані та обрати Make Transition та дотягнути лінію до нового стану. Далі потрібно вказати умову переходу. Для цього натискаємо на лінію переходу і в Inspector з'явиться наступне вікно:

Знімаємо прапорець з Has Exit Time, коли не потрібно, щоб попередня анімація спочатку програтися уся, а потім виконався перехід на наступну анімацію. Коли прапорець знято, то попередня анімація програється не до кінця, а як тільки виконується умова, переходить до іншої анімації. Натискаємо + в Conditions та додаємо умову для переходу.

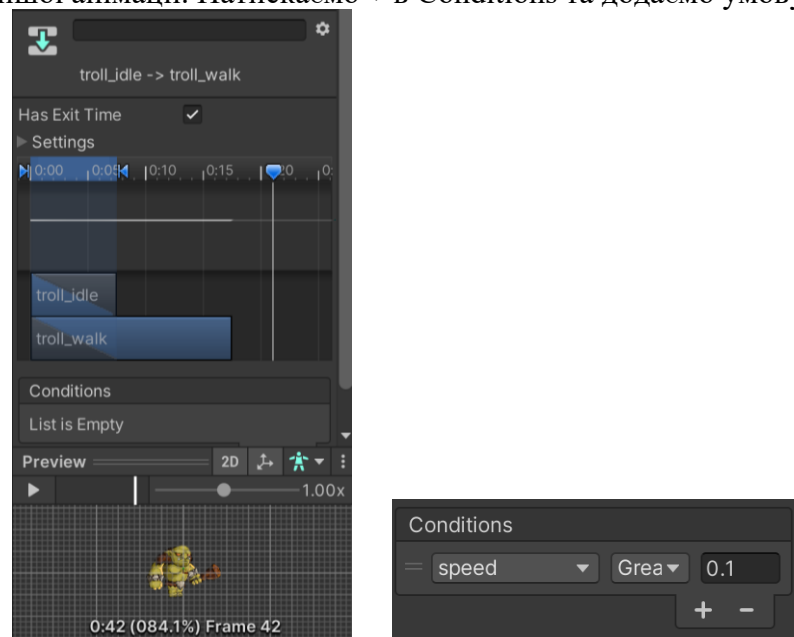


Рисунок 3.6 – Налаштування переходу з одного стану до іншого

Коли персонаж закінчить йти, потрібно повернутися в стан Idle. Тоді робимо новий перехід аналогічним чином до стану Idle.

В результаті налаштувань вікно Animator може мати наступний вигляд:

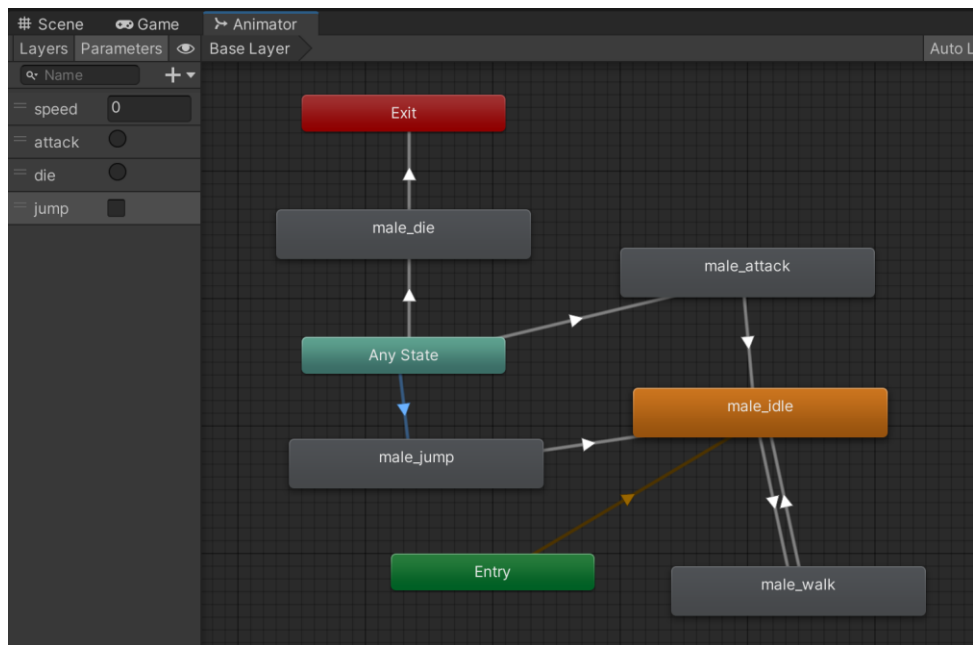


Рисунок 3.7 – Приклад вікна Animator для анімацій персонажу

2. Завдання до практичної роботи 3

Створити 2D-гру, додати персонажа та ворогів до сцени, створити анімацію для доданого персонажа та ворогів, налаштувати в Animator перехід між станами анімацій для персонажа та ворогів.

Практична робота 4. 2D-гра. Рух персонажу

Мета роботи: познайомитися зі створенням руху персонажа для 2D-гри на Unity. Навчитися підключати анімацію персонажа та виконувати його рухи.

1. Теоретичний матеріал

Ресурс Animator Controller створюється в Unity і дозволяє впорядковувати та підтримувати набір анімацій для персонажа чи об'єкта. У більшості ситуацій нормально мати кілька анімацій і перемикатися між ними, коли виникають певні умови гри.

Зазвичай персонаж має кілька анімацій, які відповідають різним діям, які можуть бути у грі. Наприклад, він може просто дихати і похитуватися в спокої, йти по команді і скидати руки в паніці при падінні з платформи. У дверей можуть бути анімації відкриття, закриття, заклинювання та зламу.

Однак навіть якщо у вас є лише один анімаційний кліп, вам все одно потрібно розмістити його в контролері аніматора, щоб використовувати його на ігровому об'єкті.

Контролер має посилання на анімаційні кліпи, що використовуються в ньому, і керує різними станами анімації та переходами між ними за допомогою так званого кінцевого автомата, який можна розглядати як певну блок-схему або просту програму, написану мовою візуального програмування в Unity.

Основна ідея полягає в тому, що персонаж бере участь у якомусь конкретному виді дій у будь-який час. Типові дії включатимуть такі речі, як стійка на місці (idling), ходьба (walking), біг (running), стрибок (jumping) тощо. Ці дії називають states, у тому сенсі, що персонаж перебуває у «стані», де він іде, стоїть чи робить ще щось.

Налаштування для наступного стану, в який гравець зможе перейти з поточного стану, називаються state transitions. Разом взяті набори станів, набори змінних та змінні пам'ятають поточну форму стану state machine.

Стани та переходи механізму станів можна уявити за допомогою графіка, де вузли будуть являти собою стани та дуги (стрілки між вузлами), що представляють переходи.

Вікно Animator складається з двох основних розділів: основної області макета з сіткою та лівої панелі «Layers & Parameters».

Ліва панель Parameters дозволяє створювати, переглядати та редагувати змінні. Ці змінні, які ви визначаєте, діють як вхідні дані в кінцевий автомат. Щоб додати параметр, клацніть значок «+» і виберіть тип параметра зі спливаючого меню. Щоб видалити параметр, виберіть параметр у списках і натисніть клавішу видалення.

Параметри анімації – це змінні, визначені в контролері анімації, до яких можна отримати доступ та присвоїти значення зі скрипти. Ось як сценарій може контролювати чи впливати на потік машини станів/

Параметри анімації можуть бути чотирьох основних типів:

- Int – ціле число;
- Float – число з дробовою частиною;
- Bool – справжнє чи хибне значення (представлено прапорцем);
- Тригер — логічний параметр, який скидається контролером під час переходу (позначається круглою кнопкою).

Параметрам можна надавати значення зі скрипту за допомогою функцій класу Animator: SetFloat, SetInteger, SetBool, SetTrigger та ResetTrigger.

Використовуйте SetFloat() у сценарії для надсилання дійсних значень до аніматора для активації переходів.

```
SetFloat (string name, float value);
```

Використовуйте `SetInteger()` у сценарії для надсилання цілих значень до аніматора для активації переходів.

```
SetInteger (string name, int value);
```

Використовуйте `SetBool()` у сценарії для надсилання булевих значень до аніматора для активації переходів.

```
SetBool (string name, bool value);
```

Метод `SetTrigger()` дозволяє встановити (тобто активувати) тригер анімації, щоб викликати зміну потоку в кінцевому автоматі аніматора контролера.

```
SetTrigger (int id)
```

Наприклад:

```
[SerializeField]
private Animator animator;
...
animator.SetBool("jump", false);
```

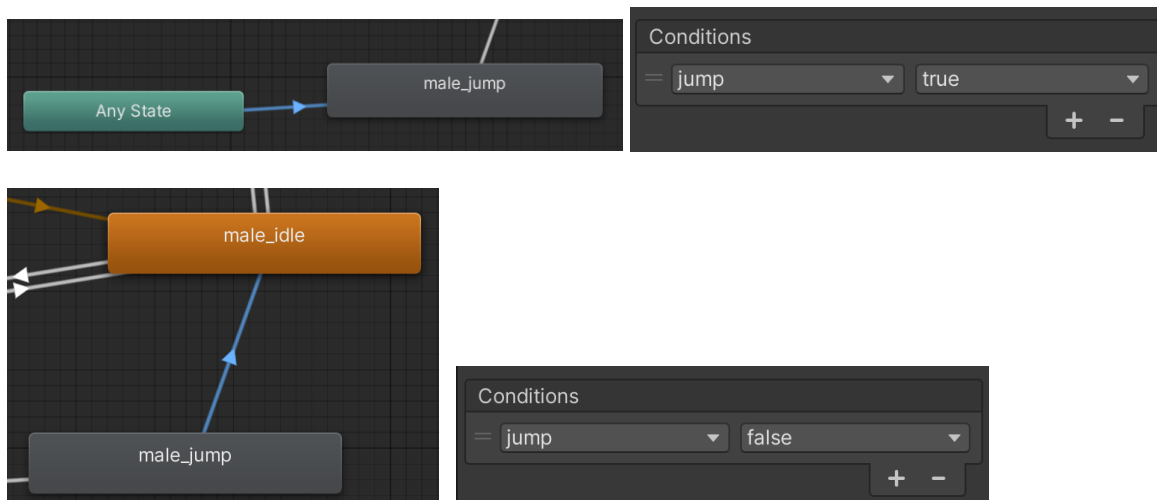
Перетягнемо до вікна `Animator` анімації персонажа, що маємо. Додаймо змінні, які допоможуть перейти з одного стану в інший.

- `Float speed;`
- `Trigger attack;`
- `Bool jump;`

В переходах на новий стан додаємо умови, в яких використовуються створені змінні. Наприклад, для переходів між станом `idle` та `walk`:



Наприклад для стрибку:



Наприклад для атаки:



Додаймо управління до нашого персонажа. Для цього створимо скрипт, наприклад, Него, та прикріпимо його до головного персонажа.

Клас Input у Unity надає доступ до введення на мобільних пристроях, таких як натискання на екран, акселерометр та географічні/локаційні дані. Доступ до клавіатури на мобільних пристроях забезпечується через keyboard iOS. Клас Input також використовується для визначення осей введення та пов'язаних з ними дій у вашому проєкті.

Для доступу до менеджера введення перейдіть до головного меню Unity, потім виберіть Edit > Project Settings > Input Manager.

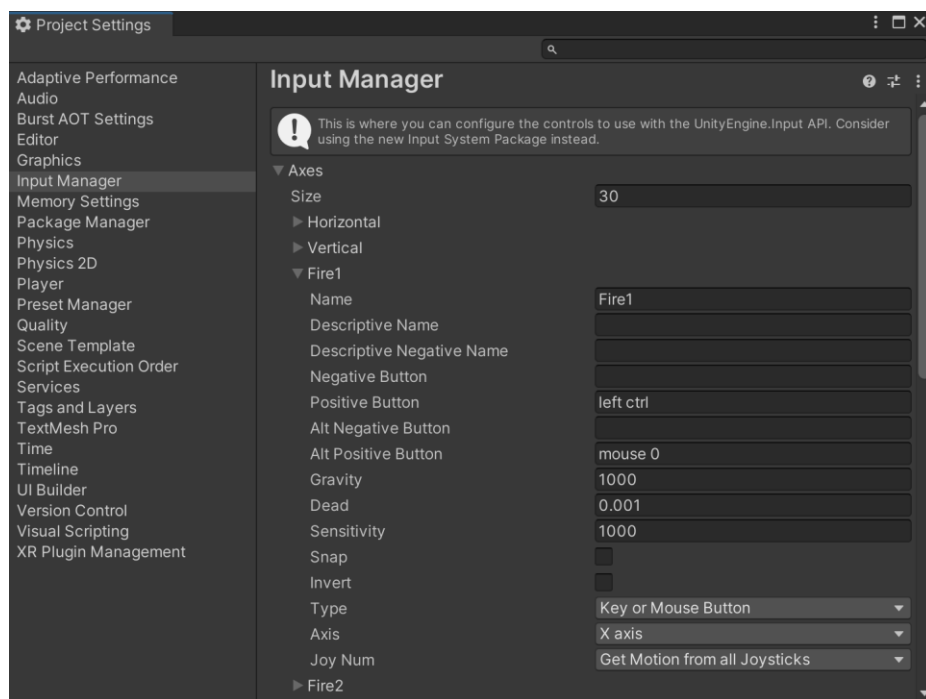


Рисунок 4.1 – Менеджер введення (Input Manager)

Кожен проєкт, який ви створюєте, має низку вхідних осей, створених за замовчуванням. Ці осі дозволяють відразу використовувати введення з клавіатури, миші та джойстика у проєкті.

Наприклад, щоб запросити поточне значення горизонтальної осі та зберегти його у змінній, можна використовувати Input.GetAxis ось так:

```
float horizontalInput = Input.GetAxis("Horizontal");
```

Для осей, які описують подію, а не рух (наприклад, стрілянина зі зброї у грі), використовуйте Input.GetButtonDown замість цього.

Перед тим як додати код до створеного скрипту необхідно персонажам додати такі компоненти:

- Rigidbody2D
- Коллайдер (CircleCollider2D, BoxCollider2D)

Додаємо до персонажа компонент Rigidbody 2D та 2 колайдери. Налаштуйте розміри та місцезнаходження колайдерів за допомогою кнопки EditCollider.

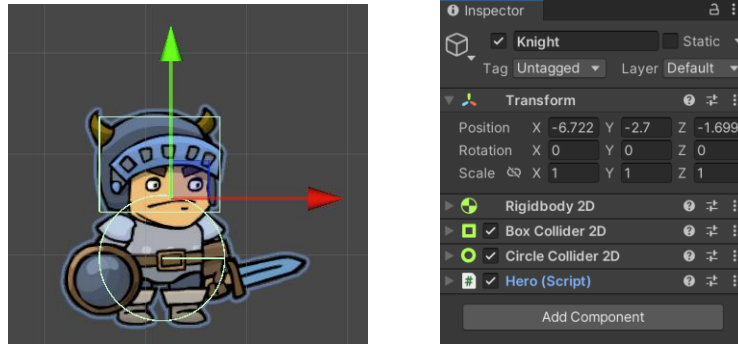


Рисунок 4.2 – Приклад персонажу с Rigidbody 2D та колайдерами

У Rigidbody2D поставте галочку Constraints -> Freeze Rotation -> Z. Відкрийте створений скрипт для персонажа.

Оголосимо змінну типу Animator (назвіть її animator) і ще одну типу Rigidbody2D.

```
[SerializeField]
private Animator animator;

[SerializeField]
private Rigidbody2D rigidbody;
```

Аніматор налаштований так, щоб включати анімацію ходьби тоді, коли швидкість більша за нуль. Якщо швидкість менша за нуль, персонаж стоїть на місці. Передаватимемо не швидкість безпосередньо, а модуль швидкості – її абсолютне значення, незалежно від знака:

```
animator.SetFloat("speed", Mathf.Abs(Input.GetAxis("Horizontal")));
```

Далі додаймо код, який рухатиме персонажа по горизонталі. Робитимемо це, задаючи значення швидкості Rigidbody2D вручну.

```
Vector2 velocity = rigidbody.velocity;
velocity.x = Input.GetAxis("Horizontal") * speed;
rigidbody.velocity = velocity;
```

Множник speed – це змінна, яку треба оголосити у класі, та вивести в інспектор. Змінюючи її, контролюємо швидкість переміщення персонажа.

```
[SerializeField]
private float speed = 2.0f;
```

Якщо запустити гру, то персонаж повинен ходити вліво-вправо, при цьому залишаючись обличчям в один і той же бік. Щоб персонаж повертався обличчям у потрібну сторону, додаймо наступний код:

```

if (transform.localScale.x < 0)
{
    if (Input.GetAxis("Horizontal") > 0)
        transform.localScale = Vector3.one;
}
else
{
    if (Input.GetAxis("Horizontal") < 0)
        transform.localScale = new Vector3(-1, 1, 1);
}

```

Об'єднаємо увесь цей код у метод Walk(), а метод будемо викликати з методу Update():

```

private void Walk()
{
    animator.SetFloat("speed", Mathf.Abs(Input.GetAxis("Horizontal")));

    Vector2 velocity = rigidbody.velocity;
    velocity.x = Input.GetAxis("Horizontal") * speed;
    rigidbody.velocity = velocity;

    if (transform.localScale.x < 0)
    {
        if (Input.GetAxis("Horizontal") > 0)
            transform.localScale = Vector3.one;
    }
    else
    {
        if (Input.GetAxis("Horizontal") < 0)
            transform.localScale = new Vector3(-1, 1, 1);
    }
}

```

Тепер маємо повну функціональність щодо руху персонажа вліво-вправо.

Додаймо функціональність виконувати атаку у вигляді метода Attack(), який також будемо викликати з методу Update():

```

private void Attack()
{
    if (Input.GetButtonDown("Fire1"))
        animator.SetTrigger("attack");
}

```

Залишається додати функціональність для виконання стрибка.

Щоб персонаж стрибав, до нього треба прикладати силу за допомогою виклику AddForce. При цьому потрібно визначати, чи перебуває персонаж на землі.

Створимо булеву змінну onGround і надамо їй значення true. Також створимо і виведемо в інспектор змінну jumpForce – сила стрибка. Потрібно підібрати їй значення для вашого персонажа.

```
[SerializeField]
private float jumpForce = 100.0f;

bool onGround = true;
```

Додаймо функціональність виконувати стрибок у вигляді метода Jump(), який також будемо викликати з методу Update():

```
private void Jump()
{
    animator.SetBool("jump", false);

    if (Input.GetButtonDown("Jump") && onGround)
    {
        animator.SetBool("jump", onGround);
        rigidbody.AddForce(Vector2.up * jumpForce);
    }
}
```

Залишається написати код, який буде визначати перебуває персонаж на землі чи ні. Щоб знати, чи є під персонажем земля, ми перевірятьимемо, чи є відповідний колайдер на лінії між двома точками. Однією буде центр персонажа. Щоб встановити іншу – створимо порожній GameObject, назовемо його GroundCheck, зробимо його дочірнім об'єктом персонажа Knight. Розмістимо цей об'єкт трохи нижче ніж персонажа.

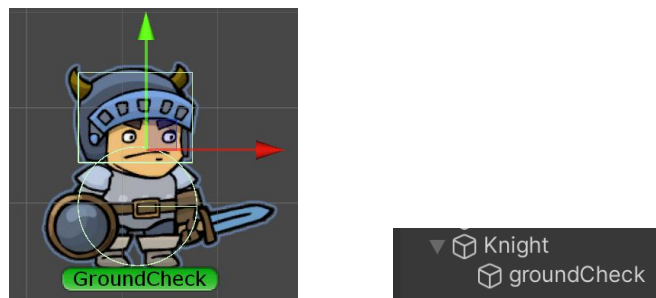
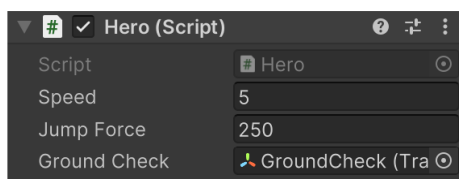


Рисунок 4.3 – Реструктуризація персонажу

Створимо змінну groundCheck типу Transform і надамо їй посилання на цей об'єкт в інспекторі.

```
[SerializeField]
private Transform groundCheck;
```



Додаймо метод CheckGround(), який повертатиме булеве значення, залежно від того, є земля під персонажем чи ні:

```
private bool CheckGround()
{
    RaycastHit2D[] hits = Physics2D.LinecastAll(transform.position, groundCheck.position);

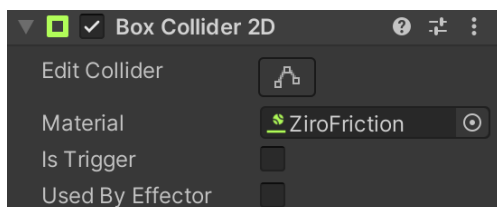
    for(int i=0; i<hits.Length; i++)
    {
        if (!GameObject.Equals(hits[i].collider.gameObject, gameObject))
        {
            return true;
        }
    }
    return false;
}
```

Тепер метод Update() має наступний вигляд:

```
void Update()
{
    onGround = CheckGround();

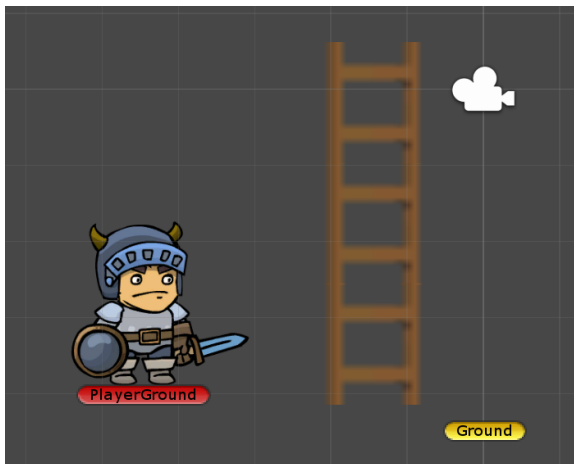
    Walk();
    Jump();
    Attack();
}
```

Якщо персонаж у стрибку притиснеться до стіни, він зупиниться і висітиме на місці доти, доки гравець не відпускає кнопку ходьби. Щоб цього не було, створіть новий Physics Material 2D, назвіть його ZeroFriction, задайте йому значення Friction (тертя) рівним нулю, і призначте цей матеріал як фізичний матеріал колайдера персонажа.



Розглянемо рух по сходам.

Спочатку потрібно створити сходи на сцені. Якщо необхідно отримати сходи потрібної довжини, то створимо новий GameObject, назвемо його Stair, додаймо як підлеглий об'єкт спрайт сходів. За потреби спрайт сходів розмножимо стільки разів, скільки потрібно, щоб отримати сходи потрібної довжини. Додаймо до об'єкта Stair BoxCollider2D. Налаштуємо розміри та положення колайдера, і поставимо галочку Is Trigger.



Потрібно, щоб, коли персонаж входить до зони тригера сходів, у нього включався режим «на сходах», а коли виходить із зони – вимикався. Для цього є два стандартні методи MonoBehaviour: `OnTriggerEnter2D` та `OnTriggerExit2D`. Як аргумент у них автоматично передається посилання на колайдер, який увійшов/вийшов у/з зони тригера.

У скрипт персонажа додаймо булеву змінну `onStair`, і створимо для неї public властивість:

```
private bool onStair = false;

public bool OnStair
{
    get
    {
        return onStair;
    }
    set
    {
        if (value == true)
            rigidbody.gravityScale = 0;
        else
            rigidbody.gravityScale = 1;

        onStair = value;
    }
}
```

Щоб персонаж не сповзав зі сходів під дією гравітації, у властивості `OnStair` додали код включення/вимкнення гравітації.

Створимо новий скрипт `Stair` та приєднаємо його до об'єкта `Stair`. Додаймо до цього скрипту методи `OnTriggerEnter2D` та `OnTriggerExit2D` з наступним кодом:

```

public class Stair : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
        Hero knight = collision.gameObject.GetComponent<Hero>();
        if (knight != null )
            knight.OnStair = true;
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        Hero knight = collision.gameObject.GetComponent<Hero>();
        if (knight != null)
            knight.OnStair = false;
    }
}

```

У скрипт персонажа додаймо змінну `stairSpeed`, яка визначатиме швидкість руху сходами.

Для того, щоб персонаж міг пересуватися вгору-вниз, знадобиться наступний код:

```

Vector2 velocity = rigidbody.velocity;
velocity.y = Input.GetAxis("Vertical") * stairSpeed;
rigidbody.velocity = velocity;

```

Створимо метод `Stairs()`, в якому, якщо персонаж перетнув тригер сходів, буде виконуватися код пересування вгору-вниз. Виклик цього методу зробимо в `Update()`.

```

private void Stairs()
{
    if (OnStair)
    {
        Vector2 velocity = rigidbody.velocity;
        velocity.y = Input.GetAxis("Vertical") * stairSpeed;
        rigidbody.velocity = velocity;
    }
}

```

2. Завдання до практичної роботи 4

Створити 2D-гру. Налаштувати в Animator перехід між станами анімацій для персонажа та реалізувати рух персонажа (йти вліво та вправо, пригати, атакувати, йти по сходам).

Практична робота 5. 2D гра. Вороги

Мета роботи: познайомитися зі створенням 2D-гри на Unity. Навчитися додавати анімацію ворогів, створювати ворогів в програмі Unity.

1. Теоретичний матеріал

Для ворогів нам будуть потрібні анімаційні спрайти walk та attack. Зробіть анімації за алгоритмом, який використовували для створення анімації персонажа. Залишимо головним контролером контролер для анімації walk (Йти). Перейменуємо його на назваВорога.controller. Відкриємо його в редакторі анімації та налаштуємо наступним чином:

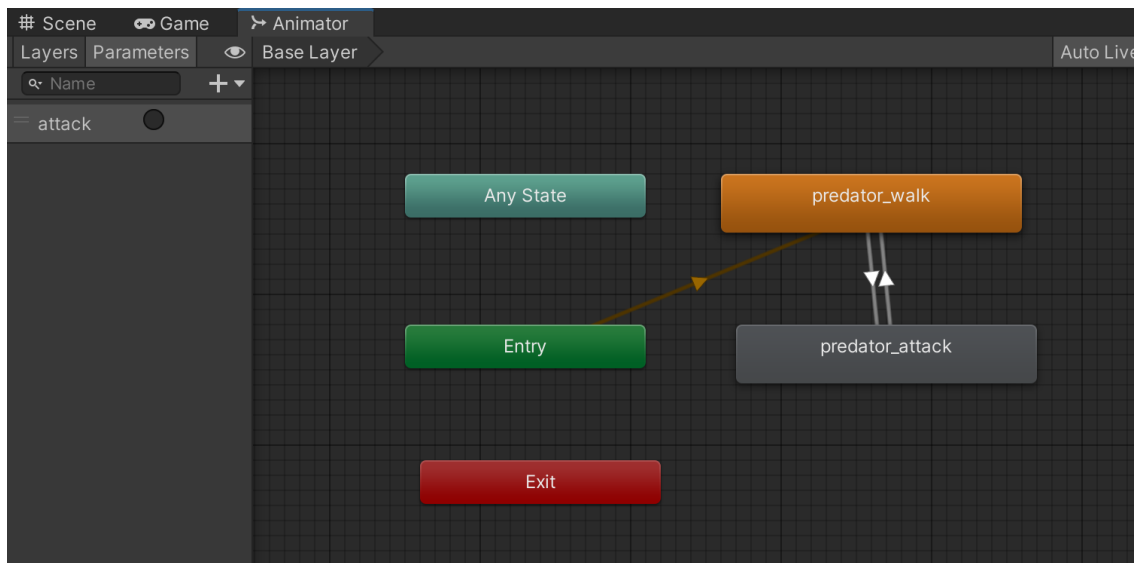


Рисунок 5.1 – Вікно Animator

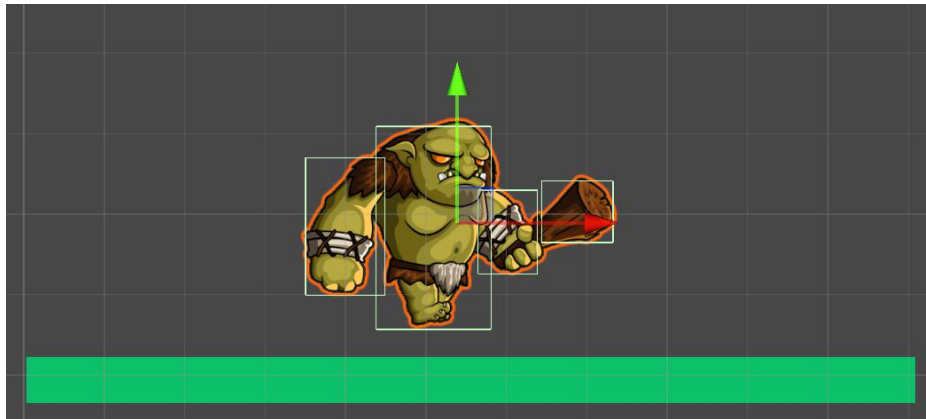
Рух ворогів не залежить від дій користувача. Розглянемо три способи задати рух для ворогів.

Перший спосіб – це ворог, який рухається між двома межами. Додаймо до сцени платформу, де цей ворог буде ходити. Додати до платформи три колайдери, один для визначення форми платформи, два для створення меж (з галочкою Is Trigger).



Рисунок 5.1 – Приклад платформи з колайдерами

Додамо на сцену ворога, який буде рухатися по платформі. Персонажу додати колайдери для визначення форми ворога та колайдер в режимі тригера на зброю.



Створимо новий скрипт Елему, в який додамо наступну функціональність.

```
private Animator animator;
private Rigidbody2D rigidbody;

[SerializeField]
private float speed = 2.0f;

private void Awake()
{
    animator = gameObject.GetComponent<Animator>();
    rigidbody = gameObject.GetComponent<Rigidbody2D>();
}

void Update()
{
    Vector2 velocity = rigidbody.velocity;
    velocity.x = transform.localScale.x * speed;
    rigidbody.velocity = velocity;
}

private void OnTriggerEnter2D(Collider2D collision)
{
    Knight knight = collision.gameObject.GetComponent<Knight>();
    if (knight != null)
    {
        animator.SetTrigger("attack");
        Attack();
    }
    else
    {
        ChangeDirection();
    }
}
```

```

private void ChangeDirection()
{
    if (transform.localScale.x < 0)
        transform.localScale = Vector3.one;
    else
        transform.localScale = new Vector3(-1, 1, 1);
}

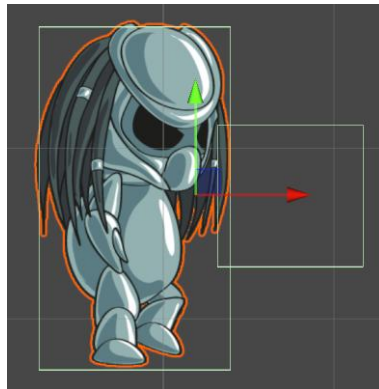
public void Attack()
{
}

```

Додаймо скрипт Епему до ворога. Запустити гру. Тепер ворог ходить по платформі туди-сюди, та при колізії колайдеров персонажа та ворога у ворога програється анімація атаки.

Розглянемо другий спосіб, де ворог буде йти за головним персонажем.

Додаймо до сцени ворога та колайдери для визначення форми ворога та колайдер в режимі тригера на зброю.



Відредагуємо скрипт Епему для функціонування другого типу ворога. Додамо можливість відрізнати ворога, який ходить туди-сюди, та який ходить за персонажем. Для цього оголосимо дві змінні. Одна для визначення типу ворога, інша для вказівки мети переслідування:

```

[SerializeField]
private bool onToHero = false;

[SerializeField]
private Transform target;

```

Відредагуємо методи Update та OnTriggerEnter2D:

```

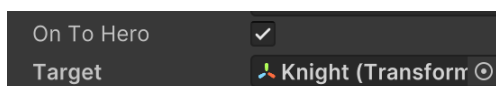
void Update()
{
    if (onToHero)
    {
        if (target == null)
            return;

        Vector2 vectorToPlayer = target.transform.position - transform.position;
        vectorToPlayer.y = 0;
        transform.rotation = Quaternion.FromToRotation(Vector2.right, vectorToPlayer);
        rigidbody.velocity = transform.right * speed;
    }
    else
    {
        Vector2 velocity = rigidbody.velocity;
        velocity.x = transform.localScale.x * speed;
        rigidbody.velocity = velocity;
    }
}

private void OnTriggerEnter2D(Collider2D collision)
{
    Knight knight = collision.gameObject.GetComponent<Knight>();
    if (knight != null)
    {
        animator.SetTrigger("attack");
        Attack();
    }
    else
    {
        if (!onToHero)
            ChangeDirection();
    }
}

```

Додаймо скрипт Енему до нового ворога. Не забуваємо заповнити нові поля:

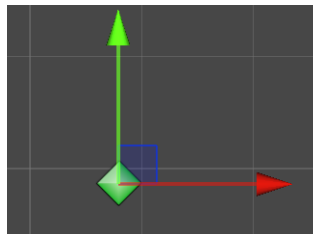
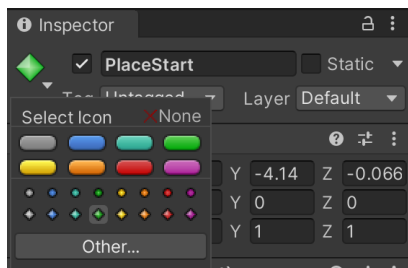


Запустити гру. Тепер ворог ходить за персонажем.

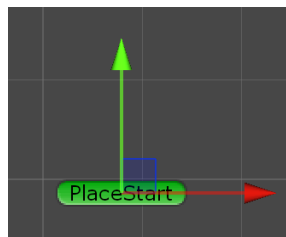
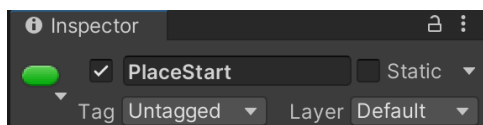
Розглянемо третій тип ворога, який при старті гри не знаходиться на сцені, але буде з'являтися в певній точці сцени через певний інтервал часу.

На початку створимо пустий об'єкт на сцені (Create Empty). Це буде наша точка появи нових ворогів на сцені. Дамо назву, наприклад, PlaceStart. Перенесемо його на те місце, де повинні з'являтися нові вороги.

Можна встановити іконки для об'єктів у сцені та інспекторі. Для зміни іконок для об'єкта гри просто натисніть на його іконку в інспекторі.



У селекторі ікон є спеціальний сорт іконок, званих мітками (Label Icon). Цей тип іконок відображається у сцені як текстова позначка, використовуючи ім'я об'єкта.



Створимо два нові скрипти. Перший Timer з наступним кодом:

```
float totalSeconds = 0;
float elapsedSeconds = 0;
bool running = false;
bool started = false;

public float Duration
{
    set
    {
        if (!running)
            totalSeconds = value;
    }
}

public bool Finished
{
    get
    {
        return started && !running;
    }
}

public bool Running
{
    get
    {
        return running;
    }
}
```

```

void Update()
{
    if (running)
    {
        elapsedSeconds += Time.deltaTime;
        if (elapsedSeconds >= totalSeconds)
            running = false;
    }
}

public void Run()
{
    if (totalSeconds > 0)
    {
        running = true;
        elapsedSeconds = 0;
        started = true;
    }
}

```

Другий скрипт Spawner, який навішаємо на створений об'єкт PlaceStart.

```

[SerializeField]
GameObject prefabEnemy;

Timer spawnTimer;

const float MinSpawnDelay = 1;
const float MaxSpawnDelay = 10;

void Start()
{
    spawnTimer = gameObject.AddComponent<Timer>();
    spawnTimer.Duration = Random.Range(MinSpawnDelay, MaxSpawnDelay);
    spawnTimer.Run();
}

void Update()
{
    if (spawnTimer.Finished)
    {
        SpawnEnemy();

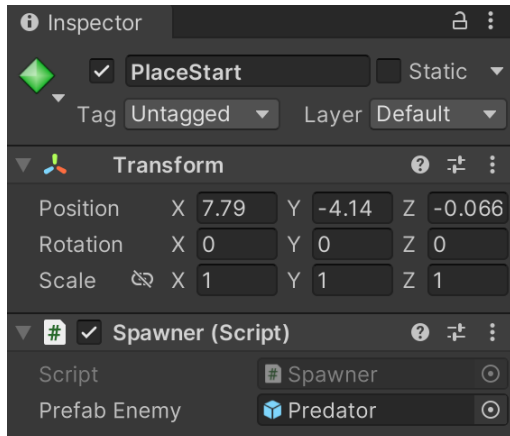
        spawnTimer.Duration = Random.Range(MinSpawnDelay, MaxSpawnDelay);
        spawnTimer.Run();
    }
}

```

```
private void SpawnEnemy()
{
    GameObject enemy = Instantiate(prefabEnemy) as GameObject;
    enemy.transform.position = transform.position;
}
```

Створіть префаб ворога, який буде з'являтися через деякі проміжки часу у вказаному місці PlaceStart. Щоб зробити префаб, перетягніть об'єкт ворога з вкладки Hierarchy на вкладку Project у папку Prefabs.

Перетягніть створений префаб в поле скрипта Spawner в Inspector. Наприклад:



Запустити гру. Тепер через проміжки часу від MinSpawnDelay до MaxSpawnDelay (константи в скрипті Spawner) будуть з'являтися нові вороги.

2. Завдання до практичної роботи 5

Створити 2D-гру, додати ворогів до сцени, створити анімацію ворогів. Реалізувати всі три типи ворогів, які розглядалися в практичній роботі.

Посилання на ворогів: <https://assetstore.unity.com/packages/2d/characters/dark-fantasy-popular-enemies-free-sample-327389>

Практична робота 6. 2D гра. Оточення

Мета роботи: познайомитися зі створенням 2D-гри на Unity. Навчитися додавати оточення, працювати з компонентами Joint, створювати камеру, що рухається за персонажем, та паралакс-ефект в програмі Unity.

1. Теоретичний матеріал

Tile – невелике зображення, яке використовується для конструювання рівнів у іграх. Tile можна перекласти як «мозаїка» або «черепиця».

Додаємо на сцену елементи сцени. Наприклад:

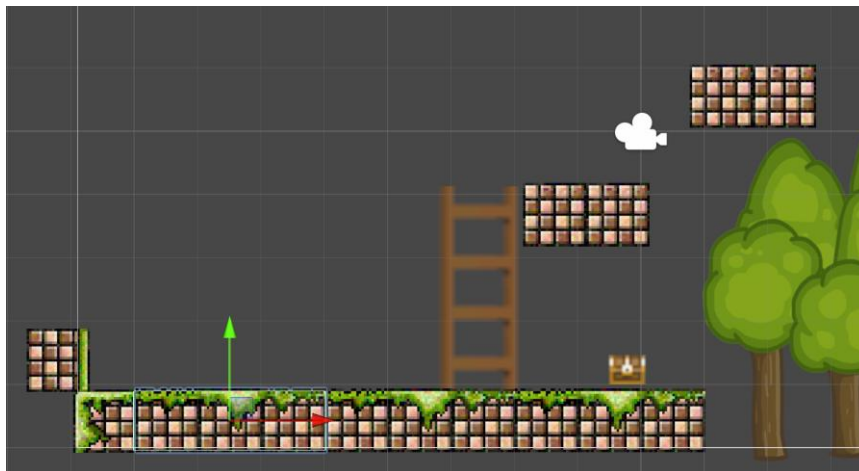


Рисунок 6.1 – Приклад сцени

Щоб елементи оточення щільно притискалися один до одного, натискаємо на клавіатурі кнопку **v** та тягнемо елемент до відповідного кута.



Додаймо до сцени підвісний міст.

Створимо на сцені порожній GameObject та дамо йому назву Bridge. Створимо стовпці для мосту з тайлів, додамо до них BoxCollider2D та зробимо дочірні до об'єкта Bridge. Помістимо всередину об'єкта Bridge спрайт ділянки мосту.



До спрайту ділянки мосту додати Rigidbody2D та BoxCollider2D.

2D джоїнти являють собою звичайні фізичні об'єкти, яким можна надавати силу, переміщати, кидати, зтиснути тощо, загалом – робити все те, що ми зазвичай робимо з фізичними об'єктами в Unity.

Особливість же джоїнтів в тому, що з їх допомогою можна створити зв'язок з іншими фізичними об'єктами на сцені. В цьому випадку всі дії, які ми здійснюємо над одним об'єктом, будуть також впливати на інші – пов'язані з ним об'єкти.

Hinge Joint 2D (або шарнірна петля) – це компонент, що контролюється фізикою Rigidbody, який дозволяє приєднати Sprite до точки в просторі, навколо якої цей об'єкт може обертатися. Обертання може бути пасивним (наприклад, у відповідь на зіткнення) або активним, коли двигун приводить його в рух. Цей компонент корисний, наприклад, для створення дверних петель, коліс, маятників та інших об'єктів, які мають обертатися навколо певної точки.

Ви можете використовувати це з'єднання, щоб дві точки перекривалися. Ці дві точки можуть бути двома компонентами Rigidbody 2D або компонентом Rigidbody2D і фіксованим положенням у світі. З'єднайте шарнірне з'єднання 2D із фіксованою позицією в світі, встановивши для параметра Connected Rigidbody значення None. З'єднання застосовує лінійну силу до обох з'єднаних 2D-ігрових об'єктів Rigidbody.

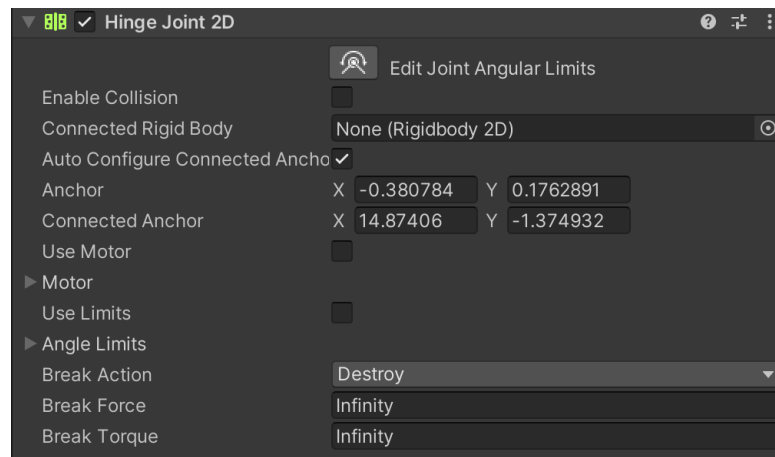


Рисунок 6.2 – Приклад компоненти Hinge Joint 2D

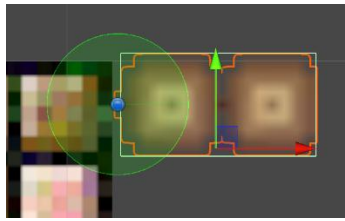
Налаштування компонента:

- Поле Connected Rigid Body – в цьому полі можна помістити будь-який фізичний об'єкт, з яким необхідно встановити зв'язок.
- Поле Enable Collision, яке включає та відключає зіткнення між джоінтом і приєднаним об'єктом.
- Поле Break Force вказує силу, при перевищенні якої відбувається розрив з'єднання між джоінтом і об'єктом. За замовчуванням стоїть значення Infinity, при якому розриву з'єднань не відбувається.
- Поле Break Torque, де можна вказати силу, при якій відбувається розрив з'єднань при повороті. Також, як і поле Break Force, за замовчуванням має значення Infinity.

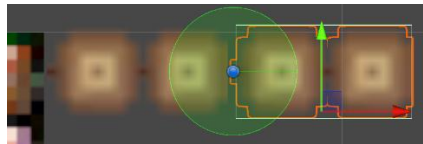
Hinge Joint 2D дозволяє застосувати силу для вдосконалення повороту об'єкта, а також ввести обмеження на кут повороту. Для цього в компоненті є поле Motor, де вказані дві змінні – Motor Speed та Maximum Motor Force, для швидкості обертання та максимальної сили, що застосовуються для вдосконалення обертання.

Обмежувати кут повороту можна задати в полі Angle Limits, де для цього введені дві змінні – Lower i Upper Angle – нижня і верхня межа кута повороту.

Після додавання компоненту Hinge Joint 2D на спрайті з'явиться мітка у вигляді кола. Центр цієї мітки переносимо до місця, де деталь буде примикати до іншої деталі.



Створимо копію спрайту ділянки мосту зі всіма компонентами. В копії в полі Connected Rigid Body додаймо посилання на Rigidbody2D попередньої ділянки мосту. Так, продовжуємо, поки не будемо мати необхідну довжину мосту.



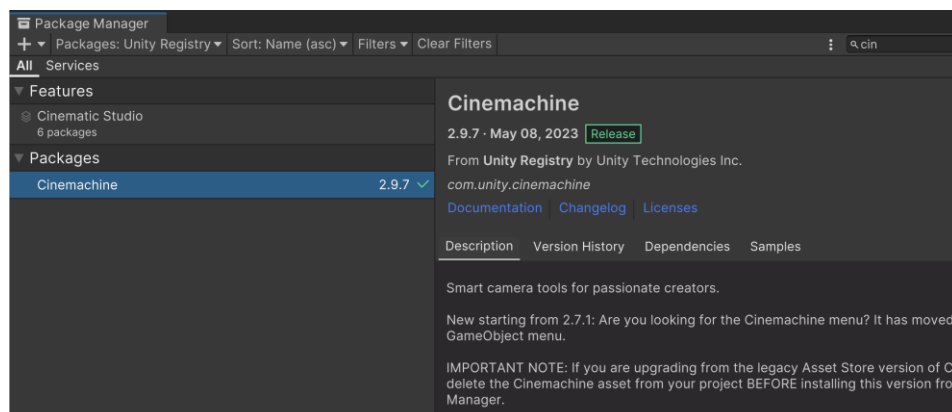
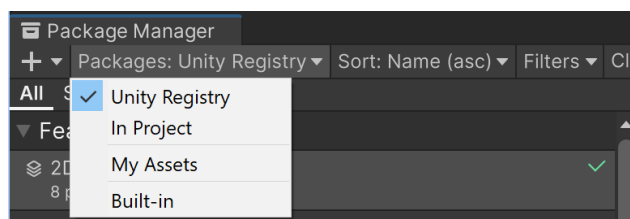
В останній спрайт ділянки мосту додаймо ще один компонент Hinge Joint 2D та в полі Connected Rigid Body нічого не додаємо.



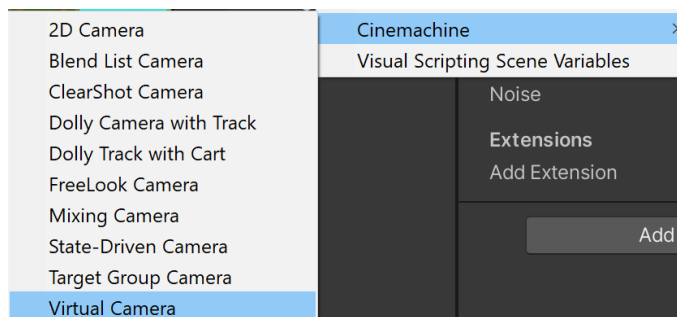
Рисунок 6.3 – Приклад місту, створеного за допомогою Hinge Joint 2D

Можна додати камеру як дочірній об'єкт до персонажа. Тим самим зробивши, щоб камера рухалася за персонажем. Но це не дуже гарне рішення.

Додаймо до проекту Cinemachine (Window → Package Manager). Вибрати Unity Registry.

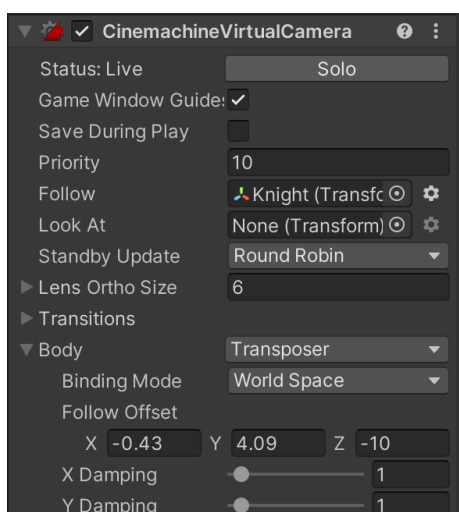


Далі створюємо на сцені новий об'єкт типу Virtual Camera:



Зробимо налаштування камери, що додалася:

- В полі Follow додаємо персонажа за ким слідувати
- В групі полів Body (поля Follow Offset: X, Y) налаштуємо, щоб межі камери не виходили за межі сцени.



Автоматично до Main Camera додався компонент CinemachineBrain. Якщо він автоматично не додався, то потрібно його додати вручну.

Тепер камера слідує за головним персонажем.

Паралакс – це ефект, який створюється за зміни кута зору на близький об'єкт щодо далекого. Завдяки цьому ефекту ми бачимо предмети обсягом, сприймаємо глибину і розуміємо, що знаходиться ближче, а що далі. В іграх паралакс може використовуватись для створення ілюзії об'єму у двовимірній грі.

Основна ідея паралаксу полягає у можливості відображення безлічі шарів з різними глибинами на одному екрані. Різні шари рухаються з різною швидкістю, створюючи ефект глибини та об'ємності. Це дозволяє зробити ігровий світ більш реалістичним та занурюючим, а також додає нові можливості для дизайнерів та розробників ігор.

Створимо шари фону. Для цього додаймо до сцени спрайти з зображенням фону, озера та кущів.



Рисунок 6.4 – Приклад спрайтів з зображенням фону

Далеким фоном буде блакитний екран, кущі будуть передні, та між ними буде озеро.

Для кожного спрайту, перед тим, як додати на сцену, робимо такі налаштування.

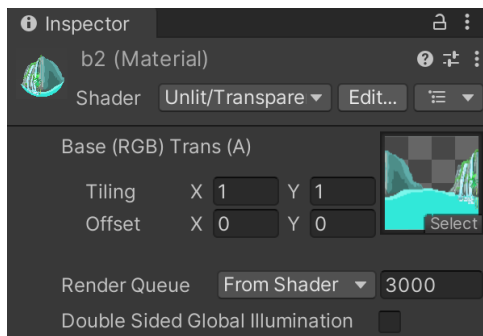
У полі Texture Type вибираємо тип Sprite (2D and UI), у полі Wrap Mode відзначаємо Repeat, у полі Mesh Type – Full Rect. У полі Pixels per unit встановлюємо таке значення, щоб спрайт закривав увесь екран.

Створимо новий скрипт SimpleMove:

```
1 public class SimpleMove : MonoBehaviour
1 {
1     private float speed = 3.0f;
1
1     /// <summary>
1     /// Update is called once per frame
1     /// </summary>
1     void Update()
1     {
1         if (Input.GetAxis("Horizontal") != 0f)
1         {
1             transform.Translate(new Vector2(Input.GetAxis("Horizontal") *
1                                             Time.deltaTime * speed, 0f));
1         }
1     }
1 }
```

Приєднуємо цей скрипт до Main Camera.

Додаймо як дочірні об'єкти до Main Camera спрайти з фоном. Далі для кожного дочірнього об'єкта потрібно створити свій матеріал. У створеного матеріалу змінити значення в полі Shader на Unlit – Transparent. Та вказати спрайт в полі Select.



В дочірніх об'єктах в компоненті Sprite Renderer, в полі Material вказати відповідний матеріал. Крім того, в полі Order in Layer поставити значення -1, -5 тощо. Чим далі об'єкт, тим більше по модулю буде число.

Створимо паралакс за рахунок зміщення offsets на матеріалі. Зміна зміщення для різних спрайтів фону залежить від параметра scale. Чим ближче спрайт фону, тим параметр матиме найменше значення, наприклад, 1, а найдаальший, наприклад, 8.

Створимо новий скрипт Parallax:

```

public class Parallax : MonoBehaviour
{
    [SerializeField]
    float scale = 1.0f;

    Transform target;
    Material material;
    Vector2 offset = Vector2.zero;

    /// <summary>
    /// Start is called before the first frame update
    /// </summary>
    void Start()
    {
        target = transform.root;
        material = GetComponent<SpriteRenderer>().material;
    }

    /// <summary>
    /// Update is called once per frame
    /// </summary>
    void Update()
    {
        offset = new Vector2(target.position.x/100f/scale, 0f);
        material.mainTextureOffset = offset;
    }
}

```

Приєднуємо цей скрипт до кожного спрайту фону та встановлюємо відповідне значення параметра scale.

2. Завдання до практичної роботи 6

Створити оточення для 2D-гри: додати тайли та створити з них платформи для переміщення персонажа та ворогів. Створити, щоб камера рухалася за персонажем. Створити об'єкт з використанням компоненти Joint 2D. Реалізувати паралах ефект.

Практична робота 7. 2D гра Clicker та UI

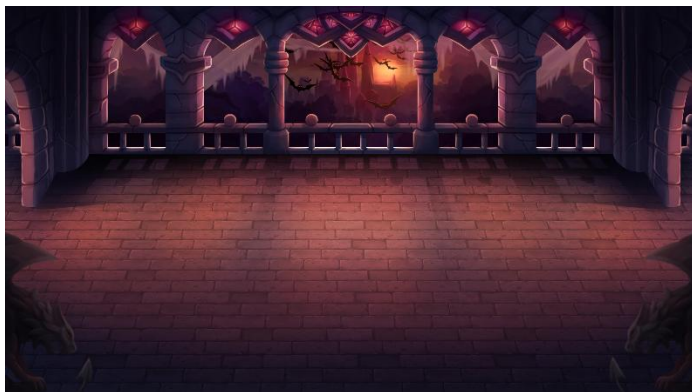
Мета роботи:

Познайомитися з розробкою гри Clicker та UI. Навчитися створювати 2D-ігри та анімацію спрайтів, писати скрипти, створювати HUD в іграх, зберігати інформацію про кількість очок або грошей, які має користувач, створювати вікна перемоги.

1. Теоретична частина

Створимо новий 2D-проект. В папці асетів створимо папки Sprites, Prefabs, Scripts, Animation. До папки Sprites скопіюємо файл для фону.

Наприклад такого вигляду:

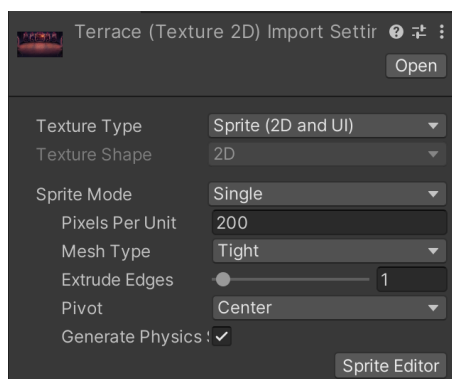


Виділимо ресурс фону в Unity на вкладці Project. Перетягнемо фон на сцену. Фон може бути замалим чи зavelиким. Цю проблему ліпше вирішити, не змінюючи масштаб спрайту на сцені, а скориставшись закладкою Inspector. Для цього видаляємо зі сцени спрайт та виділяємо його в закладці Project. На вкладці Inspector нас буде цікавити властивість Pixels per Unit.

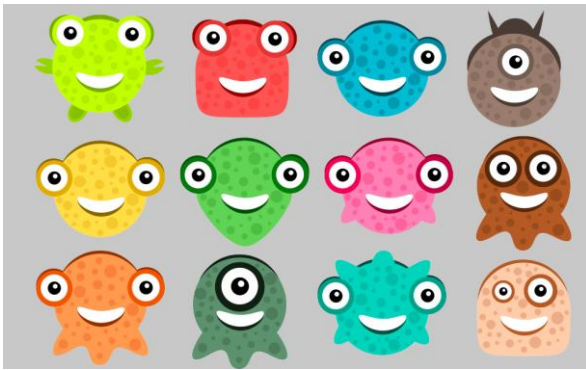
Властивість Pixels Per Unit (PPU) у Unity визначає, скільки пікселів зображення (спрайту) буде вміщено в одну одиницю виміру (1 unit) ігрового простору. Вона масштабує спрайт при імпорті: вищий PPU робить спрайт меншим, а нижчий PPU – більшим. За замовчуванням у Unity 100 PPU, що зазвичай відповідає 100 пікселям = 1 метр (unit).

Якщо спрайт має розмір 100x100 пікселів, то при Pixels Per Unit = 100 його розмір у грі буде 1 x 1 unit. Якщо встановити PPU = 50, той самий спрайт стане 2 x 2 units.

Наприклад, в нашому випадку спрайт зavelикий, зменшимо його, встановивши Pixels Per Unit у 200.

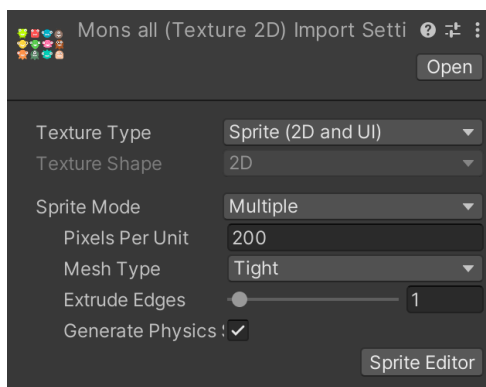


Добавимо до проекту спрайт з зображеннями монстрів, наприклад такого виду:

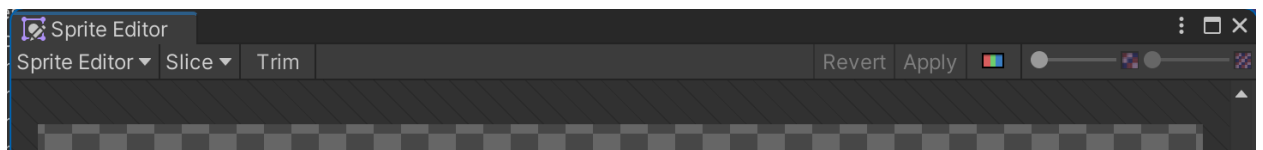


Спрайт із кількома зображеннями в Unity називається *спрайтом із складовими елементами*. Це може бути зображення, яке містить кілька маленьких картинок, які можуть бути переміщені або анімовані разом.

Для роботи зі спрайтом з кількома зображеннями в Unity необхідно використовувати редактор спрайтів (Sprite Editor). Переконайтеся, що для параметра Texture Type встановлено значення Sprite (2D and UI) для зображення, яке потрібно відредагувати. Встановіть для параметра Sprite Mode значення Multiple в Inspector імпорту текстур, якщо зображення складається з декількох елементів. Щоб відредагувати спрайт, виберіть зображення у вкладці Project, натисніть кнопку Sprite Editor в Inspector.



Відкриється вікно Sprite Editor. Разом із складеним зображенням ви побачите ряд елементів керування на панелі у верхній частині вікна.



Повзунок у правому верхньому куті керує масштабуванням, а кнопка кольорової смуги зліва від нього вибирає, чи ви переглядаєте саме зображення або його альфа-рівні. Крайній правий повзунок управляє пікселізацією (mipmap) текстури. Переміщення повзунка вліво зменшує роздільну здатність текстури спрайту. Найважливішим елементом керування є меню Slice у лівому верхньому кутку, яке дає можливість автоматично розділяти елементи зображення. Нарешті, кнопки Apply та Revert дозволяють зберегти або скасувати будь-які внесені зміни.

Найпростіший спосіб використання редактора – це ідентифікувати елементи вручну. Якщо клацнути зображення, з'явиться область вибору з ручками по кутах. Ви можете перетягувати ручки або краї прямокутника, щоб змінити розмір навколо певного елемента. Виділивши елемент, можна додати інший, перетягнувши новий прямокутник в окрему

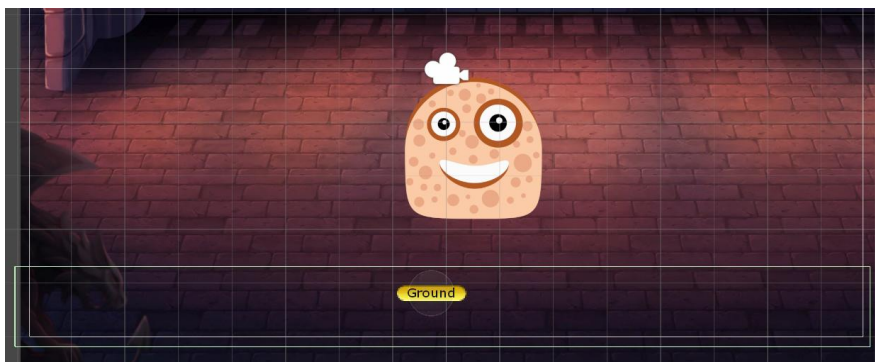
частину зображення. Ви помітите, що при виборі прямокутника в нижньому правому куті вікна з'являється панель:



Елементи керування на панелі дозволяють вибрати ім'я для зображення спрайту та вказати положення та розмір прямокутника за його координатами. Ширина межі зліва, зверху, праворуч і знизу може бути вказана у пікселях. Також є налаштування для осі Sprite, яку Unity використовує як початок координат (0) та основну «точку прив'язки» графіки. Ви можете вибрати одну з позицій щодо прямокутника за промовчанням (наприклад, по центру, зверху праворуч тощо) або використовувати власні координати.

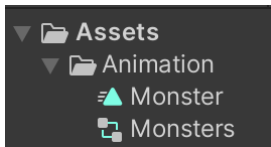
Додаймо на сцену, наприклад, першого монстра. Перед перетягуванням його на сцену, при необхідності, змініть значення в полі Pixels per unit. Щоб у подальшому не було проблем з відображенням картинки монстра та фону, поставимо в об'єкті монстра в Inspector в полі Order in layer, наприклад, значення 5. Додаймо також такі компоненти, як Rigidbody 2D та Box Collider 2D.

Додаймо на сцену пустий об'єкт, перетягнімо його до низу екрана та назовемо його Ground. Додати до нього компонент Box Collider 2D та розширити його форму на усю ширину екрана:

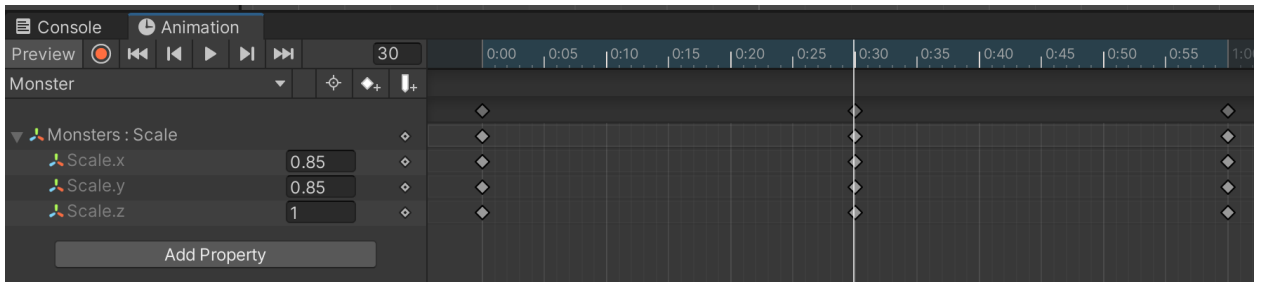


Картинка з монстром, яку ми обрали, не має анімації. Придумаємо власну анімацію, яка буде запускатися при натисканні користувачем лівої кнопки миші по цьому об'єкту. Для створення анімації виділимо на сцені в Hierarchy об'єкт монстра, назовемо його Monster. Для виділеного об'єкта визиваємо вікно Animation (Window → Animation → Animation).

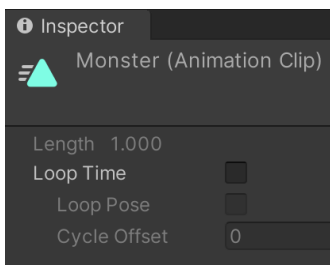
У вікні Animation натискаємо кнопку Create. Відкриється вікно для створення файлу з розширенням anim. Даємо назву анімації та записуємо її у папку Animation (якщо вона ще не створена, створіть її). В папці маємо два об'єкта: Animation clip () та Animator Controller ()



У вікні Animation натискаємо кнопку Add Property та обираємо Transform → Scale. Виділяємо на лінії часу 0:00 всі значення параметрів масштабу, встановлені в 1. Обираємо час 0:30, встановлюємо Scale.x та Scale.y на 0,85. Обираємо час 1:00 – всі значення параметрів масштабу встановлюємо в 1.



На вкладці Project виділимо анімаційний кліп Monster та на вкладці Inspector знімемо галочку з Loop Time.



Animator у Unity – це компонент, який керує анімацією об'єктів у сцені. Animator дозволяє розробникам створювати анімації та переходи між ними, що спрощує керування поведінкою персонажів чи об'єктів. Контролер анімації дозволяє перемикатися між анімаціями під час виконання певних умов. Для відкриття вікна Animator скористаємося меню Unity, а саме Window → Animation → Animator.

Вікно Animator складається з двох основних розділів: основної області макета з сіткою та лівої панелі «Layers & Parameters».

Ліва панель Parameters дозволяє створювати, переглядати та редагувати змінні. Ці змінні, які ви визначаєте, діють як вхідні дані в кінцевий автомат. Щоб додати параметр, клацніть значок «+» і виберіть тип параметра зі спливаючого меню. Щоб видалити параметр, виберіть параметр у списках і натисніть клавішу видалення.

Параметри анімації – це змінні, визначені в контролері анімації, до яких можна отримати доступ та присвоїти значення зі скрипта. Ось як сценарій може контролювати чи впливати на потік машини станів/

Параметри анімації можуть бути чотирьох основних типів:

- Int – ціле число;
- Float – число з дробовою частиною;
- Bool – справжнє чи хибне значення (представлено прапорцем);
- Trigger — логічний параметр, який скидається контролером під час переходу (позначається круглою кнопкою).

Параметрам можна надавати значення зі скрипту за допомогою функцій класу Animator: SetFloat, SetInteger, SetBool, SetTrigger та ResetTrigger.

Наприклад:

```
[SerializeField]
private Animator animator;
...
animator.SetBool("jump", false);
```

Автомати стану складаються зі станів, переходів і подій, а менші автомати підстанів можна використовувати як компоненти у великих машинах.

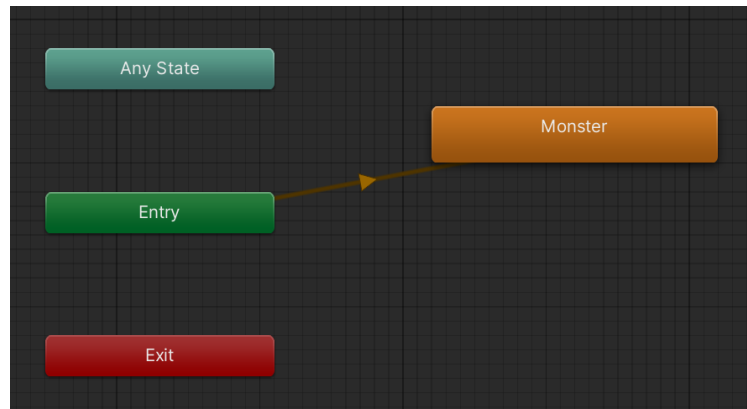


Рисунок 7.1 – Приклад вікна Animator

Any State – це особливий стан, який існує завжди. Воно підходить для ситуацій, коли ви хочете перейти до певного стану, незалежно від поточного.

Вузол Entry використовується при переході в кінцевий автомат. Вузол входу буде оцінено та розгалужено до стану призначення відповідно до встановлених умов. Таким чином, вузол Entry може контролювати, в якому стані починається кінцевий автомат, оцінюючи стан ваших параметрів, коли починається кінцевий автомат.

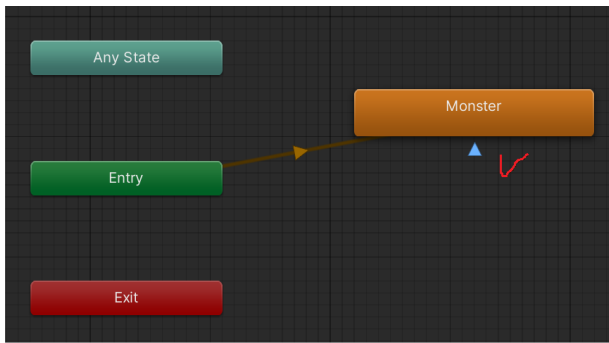
Вузол Exit використовується для вказівки, що кінцевий автомат має завершити роботу.

Можна змішувати переходи кінцевого автомата зі звичайними переходами станів, тому можна переходити від стану до стану, від стану до кінцевого автомата та від одного кінцевого автомата безпосередньо до іншого кінцевого автомата.

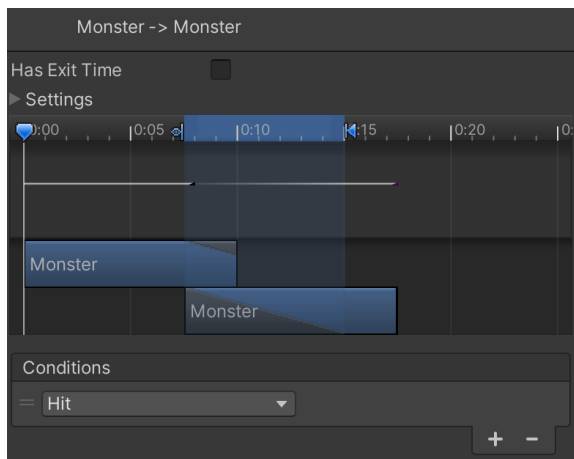
Створимо новий параметр анімації типу Trigger та назвемо його hit.

Переходи анімації (Animation Transitions) визначають, що саме відбувається, коли ви перемикаєте один стан анімації (Animation State) на інший. У будь-який час може бути активний лише один перехід. Перехід можна налаштувати лише за певної умови, яка виконується, і ці умови базуються на значеннях параметрів, встановлених у контролері аніматора.

Щоб додати перехід на новий стан, правою кнопкою миші по поточному стані та обрати Make Transition та дотягнути лінію до нового стану. Створимо перехід зі стану Monster до себе ж.

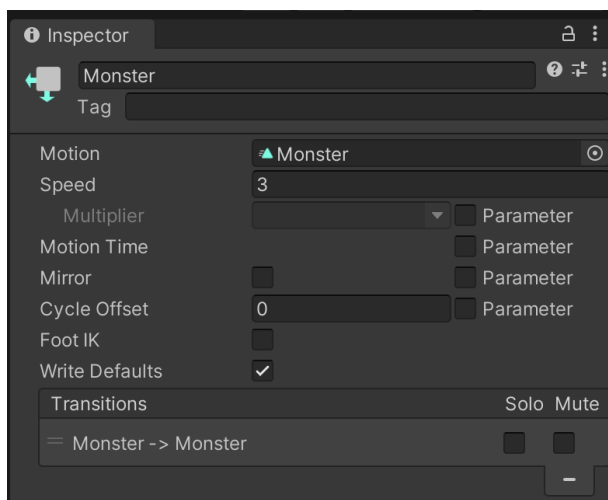


Далі потрібно вказати умову переходу. Для цього натискаємо на лінію переходу, і в Inspector з'явиться вікно, в якому, по-перше, знімаємо прапорець з Has Exit Time. А далі натискаємо + в Conditions та додаємо умову для переходу.



Has Exit Time – визначає, чи може умова переходу набути чинності у будь-який час або лише під час виходу зі стану.

Якщо виділити стан Monster, то в Inspector з'явиться вікно, в якому можна редагувати швидкість анімації (поле Speed):



З налаштуванням анімації закінчили, перейдемо до створення функціональності гри.

Створимо скрипт HitMonster, в якому додаймо код для запуску анімації при натисканні кнопки миші по монстру.

```

public class HitMonster : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
    }

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

    private void OnMouseDown()
    {
        GetComponent<Animator>().SetTrigger("Hit");
    }
}

```

Додаємо скрипт до об'єкта Monsters на сцені. Якщо зараз запустити гру, то при натисканні лівою кнопкою миші по монстру буде програватися створена анімація.

Створимо ще один скрипт HealthMonster, в якому додаймо код, який дозволить закріпити за монстром кількість його життя та скільки балів ми отримаємо, коли його знешкодимо. Додаймо цей скрипт також до об'єкта Monsters на сцені.

Спочатку оголосимо необхідні змінні:

```

[SerializeField]
private int health = 100;

[SerializeField]
private int maxCoins = 90;

```

Створимо метод GetHit(), в якому будемо зменшувати кількість життя монстра на величину вхідного параметра damage.

```

public void GetHit(int damage)
{
    int currHealth = health - damage;
    if (currHealth <= 0)
    {
        Destroy(gameObject);
    }
    health = currHealth;
}

```

Створимо виклик цього методу в класі HitMonster в методі OnMouseDown():

```
private void OnMouseDown()
{
    GetComponent<Animator>().SetTrigger("Hit");
    GetComponent<HealthMonster>().GetHit(10);
}
```

Тепер при кожному кліку по монстру кількість його життя буде зменшуватися на деяку величину, і наприкінці об'єкт Monsters видалиться зі сцени.

Далі нам потрібно реалізувати показ наступного монстра з його кількістю життя та балами, які ми отримуємо, коли його знешкодимо.

Для цього спочатку створимо префаб з об'єкта Monsters на сцені. Перетягнемо об'єкт Monsters у папку Prefabs. Створимо ще один скрипт GameController.

Оголосимо в новому скрипті змінні:

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

[SerializeField]
private GameObject monstersPrefab;

[SerializeField]
private Transform startPoint;
```

Змінна coins – поточна кількість балів, які ми набрали. Змінна monstersPrefab – префаб монстра, який буде показано замість знешкодженого монстра. Змінна startPoint – точка, де з'явиться наступний монстр.

Створимо нові методи TakeCoins() та SpawnMonster():

```
public void TakeCoins(int numCoins)
{
    coins += numCoins;

    SpawnMonster();
}

private void SpawnMonster()
{
    int index = 0;

    GameObject newMonster = Instantiate(monstersPrefab) as GameObject;
    newMonster.transform.position = startPoint.position;
}
```

Створимо на сцені пустий об'єкт GameController та прикріпимо до нього скрипт GameController.

Додаймо виклик методу TakeCoins() у скрипт HealthMonster:

```

GameManager gameController;

// Start is called before the first frame update
void Start()
{
    gameController = GameObject.FindObjectOfType<GameManager>();
}

public void GetHit(int damage)
{
    int currHealth = health - damage;
    if (currHealth <= 0)
    {
        gameController.TakeCoins(maxCoins);
        Destroy(gameObject);
    }
    health = currHealth;
}

```

Якщо запустити гру, то після того, як знешкодимо першого монстра, з'явиться новий, якого також потрібно знешкодити та отримати за це бали.

Інтерфейс користувача – це сукупність засобів, за допомогою яких гравець взаємодіє з грою. Це не лише візуальні елементи, а й їхня функціональність.

Всі UI-елементи (картинки, кнопки, текст тощо) створюються в Unity всередині спеціального об'єкта – Canvas.

Canvas (полотно) – це область, всередині якої знаходяться всі елементи UI (інтерфейсу користувача). Полотно – це ігровий об'єкт (Game Object) з доданим до нього компонентом Canvas. Всі елементи UI повинні бути дочірніми цього Canvas.

Коли ви створюєте новий елемент UI, такий, як, наприклад, панель, використовуючи меню GameObject → UI → Panel, разом з ним автоматично створюється і Canvas, якщо до цього на сцені його ще не було. Елемент UI створюється дочірнім Canvas.

HUD (Hheads-up Display) – це набір графічних інтерфейсів та декоративних елементів, розташованих на передньому плані та/або в ігровому світі, які покликані явно та швидко доносити до гравця необхідну інформацію від гри у певний момент часу, не заважаючи отриманню ігрового досвіду.

Основними елементами HUD є індикатори здоров'я, кількість боєприпасів, рівень та досвід, прогрес у грі, інвентар тощо.

Додаймо у наш інтерфейс можливість бачити прогрес у грі. Як процес у грі будемо розуміти інформацію про кількість очок або грошей, які має користувач. Інформація складатиметься з двох частин: картинка, що показує тип інформації, та сама кількість. Позиціонувати цю інформацію будемо у правому верхньому кутку.

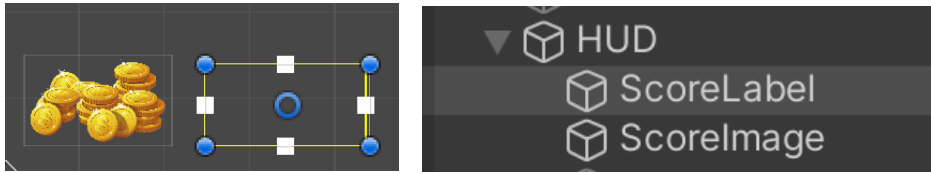
Доробимо гру Clicker та додаємо в інтерфейс можливість бачити кількість накопичених балів.

Створимо у Canvas з назвою HUD картинку та в компоненті Image в поле Source Image додаймо зображення, наприклад, купки монет, та назовемо ScoreImage.



Також створимо Text та назовемо його ScoreLabel. Налаштуємо шрифт та колір тексту та видалимо текст.

Нові елементи позиціонуємо до правого верхнього кута.



Створимо скрипт HUD та прикріпимо його до об'єкта HUD. В скрипті створимо посилання на екземпляр класу:

```
static private HUD _instance;
public static HUD Instance { get { return _instance; } }

private void Awake()
{
    _instance = this;
}
```

Створимо поле для посилання на об'єкт Text та метод який буде заповнювати цей об'єкт інформацією:

```
[SerializeField]
private TMP_Text scoreLabel;

public void SetScore(string scoreValue)
{
    scoreLabel.text = scoreValue;
}
```

Скористаємося вже створеним скриптом GameController. В цьому скрипті було створено поле coins, де зберігається поточна кількість накопичених балів. В скрипті HealthMonster в методі GetHit() є рядок передачі кількості очок до GameController:

```
public void GetHit(int damage)
{
    int currHealth = health - damage;
    if (currHealth <= 0)
    {
        gameController.TakeCoins(maxCoins);
        Destroy(gameObject);
    }
    health = currHealth;
}
```

Відредагуємо в скрипті GameController метод TakeCoins:

```
public void TakeCoins(int numCoins)
{
    coins += numCoins;

    HUD.Instance.SetScore(coins.ToString());

    SpawnMonster();
}
```

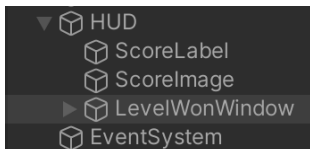
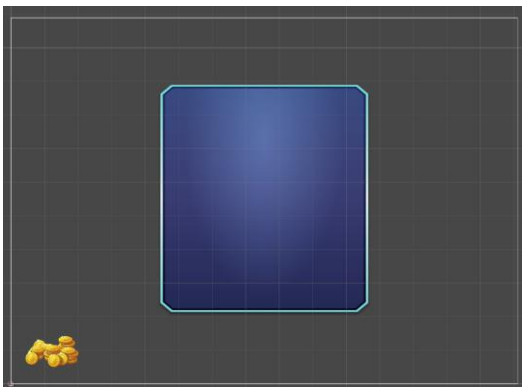
Якщо запустити проєкт, після знищення монстра на екрані з'явиться кількість зароблених балів.



Нехай, коли користувач накопичить певну кількість очок, гра припиняється і з'являється вікно перемоги. Розглянемо кроки для реалізації такої задачі.

По-перше, створимо вікно перемоги.

Створимо у Canvas нову Panel та назвемо її LevelWonWindow. В компоненті Image в поле Source Image додаймо зображення фону вікна та налаштуємо його розмір:



Додаймо до панелі LevelWonWindow такі об'єкти, як Text та Button. Налаштуємо їх наступним образом:



Необхідно додати функціональність появи цього вікна та те, що буде відбуватися при натисканні кнопки у вікні.

У скрипт HUD додаймо поле для зберігання посилання на вікно виграшу та методи показу цього вікна та переходу на іншу сцену:

```
[SerializeField]
private GameObject LevelWonWindow;

public void ShowLevelWonWindow()
{
    LevelWonWindow.SetActive(true);
}

public void ButtonMainMenu()
{
    SceneManager.LoadScene("MainMenu");
}
```

В скрипт GameController до методу TakeCoins() додаймо перевірку кількості набраних очок:

```
public void TakeCoins(int numCoins)
{
    coins += numCoins;

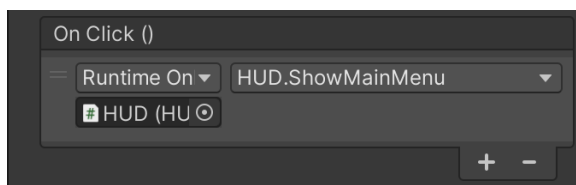
    HUD.Instance.SetScore(coins.ToString());

    if (coins > 100)
        HUD.Instance.ShowLevelWonWindow();

    SpawnMonster();
}
```

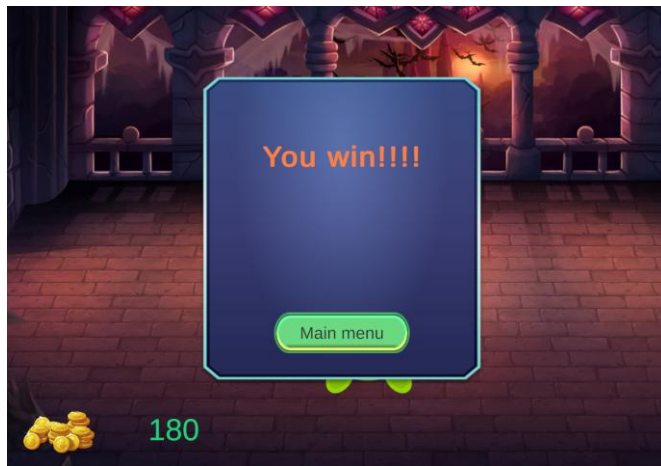
У подальшому в скрипт GameController можна ввести поле для зберігання кількості очок, при якій завершується гра, та замінити число 100 на цю змінну.

І в завершенні для кнопки на панелі LevelWonWindow додати дію при натисканні:



Знімаємо прапорець активності для вікна перемоги LevelWonWindow.

Запустимо проєкт, знищимо декілька монстрів. В результаті з'явиться вікно перемоги:



2. Завдання до практичної роботи 7

Створіть новий 2D-проект та реалізуйте гру Clicker. Зробіть модифікації: після знищення монстра на сцені повинен з'явитися новий, який буде обраний випадковим чином із масиву монстрів. Всім монстрам у масиві присвоїти різні рівні життя та кількості балів, які отримаємо після їхнього знищення.

Додати UI у вигляді кількості виграних балів, кількості життя, що залишилося у монстрів, та створити вікно перемоги.

Практична робота 8. Методи захисту від злому в іграх

Мета роботи: познайомитися з методами захисту від злому в іграх. Навчитися створювати ігри та реалізовувати методи захисту від злому.

1. Теоретичний матеріал

Якщо Вам потрібна справжня безпека, то найкращий варіант – зберігати дані на сервері, де користувачі не можуть їх змінити. Щоб це працювало, програма не повинна надсилати дані безпосередньо на сервер, оскільки користувачі все одно можуть маніпулювати ними. Замість цього програма може лише надсилати команди на сервер, дозволяти серверу змінювати дані, а потім надсилати результати назад до програми.

Розглянемо простий спосіб збереження даних, тобто локально, та можливі методи захисту.

Використання простих JSON-файлів у папці збережень дозволяє користувачам легко маніпулювати характеристиками персонажа. Зловмисник може з легкістю маніпулювати даними свого акаунту, використовуючи внутрішні дані, завантажені на пристрій. Це перша і основна проблема, яку потрібно вирішити.

Шифрування внутрішньоігрових даних важливе для забезпечення безпеки та унеможливлення несанкціонованого доступу до конфіденційної інформації. Цей процес забезпечує захист від можливих атак та витоків даних, а також дозволяє зберігати важливу інформацію, таку як паролі або ігровий прогрес, у безпечному стані. Шифрування внутрішньоігрових даних включає в себе застосування різноманітних шифрувальних алгоритмів та практик, щоб забезпечити конфіденційність та цілісність інформації в межах гри.

Base64-перетворення в іграх використовується для безпечної передачі бінарних даних (зображень, збережень, мережевих пакетів) через текстові протоколи, перетворюючи їх на набір із 64 ASCII-символів. Це забезпечує цілісність даних під час обміну, захищаючи від пошкоджень, але збільшує розмір файлів приблизно на 33%.

Base64 – це метод кодування двійкових (binary) даних у текстовий формат (ASCII), який використовує лише 64 безпечні друковані символи (A-Z, a-z, 0-9, +, /). Суть полягає у перетворенні 3 байтів (24 біти) даних у 4 символи, що дозволяє передавати файли або зображення через текстові протоколи (email, JSON, HTTP) без пошкодження даних.

Алгоритм кодування Base64 (покроково):

- Розбиття на байти – вихідні дані діляться на групи по 3 байти ($3 \times 8 = 24$ біти).
- Регруповання в 6 бітів – кожен 24-бітний блок розбивається на 4 блоки по 6 бітів ($4 \times 6 = 24$ біта).
- Індксація – отримані 6-бітові значення (0 – 63) перетворюються на відповідні символи ASCII-таблиці (A – Z, a – z, 0 – 9, +, /).
- Доповнення (Padding) – якщо вихідний блок містить менше 3 байтів, використовується символ = для заповнення, щоб підсумковий рядок був кратний 4 символам.

Наприклад, маємо слово "Man" (3 байта). Запишемо слово в бітах: М (77), а (97), н (112) → 01001101 01100001 01101110 (24 біти). Розбивка по 6 бітів: 010011 (19), 010110 (22), 000101 (5), 101110 (46). Результат: TWFu.

Розглянемо роботу з Base64 на C#:

```
static void CodeToBase64(string fileName)
{
    byte[] bytes = File.ReadAllBytes(fileName);
    string base64 = Convert.ToBase64String(bytes);

    File.WriteAllText("data_b64.txt", base64);
}

static void DecodeFromBase64(string fileName)
{
    string base64 = File.ReadAllText(fileName);
    byte[] bytes = Convert.FromBase64String(base64);

    File.WriteAllBytes("settingsBase64.json", bytes);
}

...

string fileName = "settingsSer.json";
CodeToBase64(fileName);

fileName = "data_b64.txt";
DecodeFromBase64(fileName);
```

```
File Edit View
```

```
[
  {
    "Name": "Olena",
    "Life": 100,
    "Money": 250,
    "ArtifactsCsv": "Ring, Amulet, Key",
    "Artifacts": [
      "Ring",
      "Amulet",
      "Key"
    ],
    "Num": 3
  },
  {
    "Name": "Ivan",
    "Life": 80,
    "Money": 120,
    "ArtifactsCsv": "Map, Potion",
    "Artifacts": [
      "Map",
      "Potion"
    ],
    "Num": 2
  }
]
```

File Edit View

hW0KICB7DQogICAgIk5hbWUoiOiA1T2x1bmEiLA0KICAgICJMaWZlIjogMTAwLA0KICAgICJNb25leSI6IDI1MCwNCiAgICA1QXJ0awZhY3RzQ3N2IjogIlJpbmcsIEFTdWxldCwgS2V5IiwNCiAgICA1QXJ0awZhY3RzIjogWw0KICAgICAgIlJpbmciLA0KICAgICAgIkFtdWxldC1TsDQogICAgICA1S2V5Ilg0KICAgIF0sDQogICAgIk51bSI6IDMNCiAgFswNCiAgew0KICAgICJOYWllIjogIk1k1Yw4iLA0KICAgICJMaWZlIjogODAsDQogICAgIk1vbW5IjogMTIwLA0KICAgICJBCnRpZmZjHNDc3YiOiA1TWFWLCBQb3Rpb24iLA0KICAgICJBCnRpZmZjHMiOiBbDQogICAgICA1TWFiIiwNCiAgICAgICQb3Rpb24iDQogICAgSxswNCiAgICA1TnVtIjogMg0KICB9DQpd

Rijndael є блоковим шифром з довжиною блоку n та довжиною ключа n_k , які можуть приймати одне з трьох значень: 128, 192, 256 бітів. Алгоритм складається з n_r раундів, де параметр n_r визначається за числами n та n_k згідно з таблицею 8.1.

Таблиця 8.1. Кількість раундів в алгоритмі шифрування Rijndael

n_r	$n=128$	$n=192$	$n=256$
$n_k = 128$	10	12	14
$n_k = 192$	12	12	14
$n_k = 256$	14	14	14

В Rijndael використовуються чотири базові перетворення, означення яких наведено нижче:

- 1) K (AddRoundKey) – додавання до вхідного повідомлення раундового ключа;
- 2) S (ByteSub) – застосування нелінійних вузлів заміни;
- 3) R (ShiftRow) – циклічний зсув рядків;
- 4) M (MixColumn) – перемішування стовпців.

Зашифрування відкритого тексту полягає у виконанні таких дій:

- 1) обчисленні (за допомогою окремої процедури – розкладу ключів) $n_r + 1$ раундових ключів за ключем шифрування;
- 2) додаванні першого раундового ключа до відкритого тексту;
- 3) виконанні $n_r - 1$ проміжних раундів;
- 4) виконанні останнього раунду.

Кожен проміжний раунд полягає в послідовному застосуванні до вхідного повідомлення перетворень S, R, M, K; в останньому раунді здійснюються перетворення S, R, K.

Для реалізації шифрування AES у C# використовується простір імен *System.Security.Cryptography* та клас Aes. Основний підхід полягає у створенні екземпляра Aes, використанні ключа (Key) та вектора ініціалізації (IV) для створення ICryptoTransform через CreateEncryptor або CreateDecryptor, а потім обробці даних за допомогою CryptoStream.

Вектор ініціалізації (Initialization Vector, IV) – це випадкова або псевдовипадкова послідовність байтів, яка використовується разом із секретним ключем у симетричному шифруванні для гарантування унікальності кожного зашифрованого блоку даних. Він запобігає появі однакових шаблонів у зашифрованому тексті, роблячи шифр стійким до атак.

Без IV однаковий текст, зашифрований одним і тим самим ключем, завжди виглядатиме однаково. Це дозволяє хакерам знаходити закономірності (наприклад, розпізнати стандартні заголовки повідомлень).

В алгоритмах AES використання IV залежить від конкретного режиму роботи. Використовуються такі режими роботи:

- AES-CBC (Cipher Block Chaining) – у цьому режимі кожен блок даних перед шифруванням «змішується» з попереднім зашифрованим блоком. Оскільки для першого блоку ще немає «попереднього», IV виступає в ролі цього першого віртуального блоку. Перший блок тексту проходить операцію XOR з IV, і лише після цього результат шифрується ключем AES.
- AES-CTR (Counter Mode) – цей режим перетворює AES на потоковий шифр. Тут IV – число, що використовується один раз і є частиною «лічильника». AES шифрує не сам текст, а комбінацію «IV + лічильник» (який збільшується для кожного блоку). Отриманий «випадковий» потік даних потім змішується (XOR) з текстом.
- AES-GCM (Galois/Counter Mode) – найсучасніший режим, який не тільки шифрує, а й перевіряє цілісність даних. Працює схоже на CTR, але використовує

IV (зазвичай 96 біт) для створення унікального ключа для кожного повідомлення. Якщо дані будуть змінені під час передачі, алгоритм це виявить завдяки «тегу аутентифікації», який генерується разом з IV.

Aes – це абстрактний клас, від якого успадковуються всі реалізації алгоритму AES. У свою чергу, клас Aes сам є спадкоємцем іншого абстрактного класу – SymmetricAlgorithm, який є, як впливає з назви, симетричним алгоритмом шифрування. У .NET всі класи, похідні від SymmetricAlgorithm класу, використовують режим зчеплення блоків тексту (англ. Cipher Block Chaining, CBC). Режим CBC використовується за замовчуванням, але за бажанням його можна змінити.

Щоб отримати об'єкт для шифрування за заданим алгоритмом, достатньо викликати метод Create(). Наприклад, скористаємося цим методом і створимо об'єкт для шифрування даних з використанням AES та подивимося на його налаштування за замовчуванням та деякі інші властивості:

```
Aes aes = Aes.Create();
Console.WriteLine($"{aes.GetType()}");
Console.WriteLine($"Размір ключа: {aes.KeySize}");
Console.WriteLine($"Размір блоку: {aes.BlockSize}");
Console.WriteLine($"Режим роботи: {aes.Mode}");
Console.WriteLine("Розміри ключів, що підтримуються симетричним алгоритмом:");
foreach (var keySize in aes.LegalKeySizes)
{
    Console.WriteLine($"{keySize.MinSize} - {keySize.MaxSize} бит (с шагом {keySize.SkipSize})");
}
Console.WriteLine("Розміри блоків, що підтримуються симетричним алгоритмом:");
foreach (var blockSize in aes.LegalBlockSizes)
{
    Console.WriteLine($"{blockSize.MinSize} - {blockSize.MaxSize} бит");
}
Console.WriteLine($"Вектор ініціалізації: {Convert.ToString(aes.IV)}");
Console.WriteLine($"Ключ: {Convert.ToString(aes.Key)}");
```

```
System.Security.Cryptography.AesImplementation
Размір ключа: 256
Размір блоку: 128
Режим роботи: CBC
Розміри ключів, що підтримуються симетричним алгоритмом:
128 - 256 бит (с шагом 64)
Розміри блоків, що підтримуються симетричним алгоритмом:
128 - 128 бит
Вектор ініціалізації: 4D7AE664F78D13C1548E61FC303BA2BD
Ключ: A43EB10A60FE6A4D7F088AE499F3457669A5066946DB061C9FAE4865EEC01656
```

Як видно з виведення консолі, для AES ми можемо використовувати ключі розміром 128 бітів, 192 біти та 256 бітів. За замовчуванням, в отриманому об'єкті вже створено ключ розміром 256 бітів і вектор ініціалізації в 128 бітів. Але за бажанням ми можемо змінити ці значення. Зазвичай беруть ключ 32 байта при використанні AES-256, 16 байт – AES-128.

Класи симетричного шифрування (Aes, Des тощо) використовуються зі спеціальним класом потоку – CryptoStream, який шифрує дані, раховані в потік. Клас CryptoStream ініціалізується за допомогою класу керованого потоку та класу, що реалізує інтерфейс ICryptoTransform, а також із зазначенням значення перерахування CryptoStreamMode, яке вказує тип доступу, дозволеного для CryptoStream.

Клас CryptoStream можна ініціалізувати за допомогою будь-якого класу, який є похідним від класу Stream, включаючи FileStream, MemoryStream тощо.

Наприклад, щоб створити об'єкт типу CryptoStream для шифрування даних, можна використовувати такий конструктор:

```

Aes aes = Aes.Create();

string sourceFileName = "settingsSer.json";
string outputFileName = "settingsjson.enc";

FileStream inStream = new FileStream(sourceFileName, FileMode.Open);
FileStream outStream = new FileStream(outputFileName, FileMode.Create);

CryptoStream encStream = new CryptoStream(outStream,
                                           aes.CreateEncryptor(aes.Key, aes.IV),
                                           CryptoStreamMode.Write);

```

Ось як може виглядати процес шифрування та дешифрування файлу з використанням алгоритму AES:

```

//виклик функцій
string sourceFileName = "settingsSer.json";
string outputFileName = "settingsjson.enc";
string destFile = "settings.json";

string key = "secret key";
string ivSecret = "vector";

EcryptFile(sourceFileName, outputFileName, key, ivSecret);
DecryptFile(outputFileName, destFile, key, ivSecret);

//перетворення ключа та вектору
private static byte[] GetIV(string ivSecret)
{
    MD5 md5 = MD5.Create();
    return md5.ComputeHash(Encoding.UTF8.GetBytes(ivSecret));
}

private static byte[] GetKey(string key)
{
    SHA256 sha256 = SHA256.Create();
    return sha256.ComputeHash(Encoding.UTF8.GetBytes(key));
}

//шифрування даних

```

```

static void EcryptFile(string sourceFileName, string outputFileName,
                      string key, string ivSecret)
{
    Aes aes = Aes.Create();

    aes.IV = GetIV(ivSecret);
    aes.Key = GetKey(key);

    FileStream inStream = new FileStream(sourceFileName, FileMode.Open);
    FileStream outStream = new FileStream(outputFileName, FileMode.Create);

    CryptoStream encStream = new CryptoStream(outStream,
                                              aes.CreateEncryptor(aes.Key, aes.IV),
                                              CryptoStreamMode.Write);

    long readTotal = 0;

    int len;
    int tempSize = 100; //розмір тимчасового сховища
    byte[] bin = new byte[tempSize]; //тимчасове Сховище для зашифрованої інформації
    while (readTotal < inStream.Length)
    {
        len = inStream.Read(bin, 0, tempSize);
        encStream.Write(bin, 0, len);
        readTotal = readTotal + len;
        Console.WriteLine($"{readTotal} байт оброблено");
    }

    encStream.Close();
    outStream.Close();
    inStream.Close();
}

```

//дешифрування даних

```

private static void DecryptFile(string sourceFile, string destFile,
                                string key, string ivSecret)
{
    using FileStream fileStream = new(sourceFile, FileMode.Open);
    using Aes aes = Aes.Create();

    aes.IV = GetIV(ivSecret);
    aes.Key = GetKey(key);

    using CryptoStream cryptoStream = new(fileStream,
                                            aes.CreateDecryptor(aes.Key, aes.IV),
                                            CryptoStreamMode.Read);

    using FileStream outputStream = new FileStream(destFile, FileMode.Create);

    using BinaryReader decryptReader = new(cryptoStream);
    int tempSize = 10;
    byte[] data;
    while (true)
    {
        data = decryptReader.ReadBytes(tempSize);
        if (data.Length == 0)
            break;
        outputStream.Write(data, 0, data.Length);
    }
}

```

Результат:

File	Edit	View
[	{	"Name": "Olena",
	"Life": 100,	"Money": 250,
	"ArtifactsCsv": "Ring, Amulet, Key",	"Artifacts": [
	"Ring",	"Amulet",
	"Key"	],
	"Num": 3	},
	{	"Name": "Ivan",
	"Life": 80,	"Money": 120,
	"ArtifactsCsv": "Map, Potion",	"Artifacts": [
	"Map",	"Potion"
	],	"Num": 2
	}	}
]		

Обфускація – це процес ускладнення коду програми або даних з метою утруднення реверс-інжинірингу, тобто забезпечення труднощів у розумінні або відтворенні вихідного коду. Це стандартна практика у сфері програмування та розробки програмного забезпечення для підвищення безпеки та унеможливлення несанкціонованого доступу чи модифікації програмного коду.

Основні аспекти обфускації включають:

- Перейменування – зміна імен змінних, функцій і класів на більш неочевидні та складні для розуміння;

- Заміна констант – заміна числових та рядкових констант на інші значення для утруднення розуміння логіки програми;
- Вставлення непотрібного коду – додавання непотрібного коду, який не впливає на функціональність, але збільшує обсяг та ускладнює аналіз;
- Абстракція умов – заміна зрозумілих умов та логічних виразів більш складними або вигаданими;
- Шифрування рядків – шифрування текстових рядків, таких як рядки, що містять повідомлення про помилки чи ключі;
- Видалення зайвої мета-інформації – вилучення зайвої мета-інформації, яка може бути корисною для реверс-інжинірингу, наприклад, видалення імен методів чи класів.

Методи обфускації можна застосовувати для захисту пам'яті програми.

Є різні методи заміни значень змінних. Розглянемо один із них з використанням випадкових значень.

Випадкове значення генерується на початку, потім з нього віднімається реальне значення (згодом воно приховується), а потім, коли необхідно, приховане значення віднімається зі згенерованого випадкового значення, при цьому різниця є вихідним числом. Ключ полягає в тому, щоб значення, яке відображається на екрані, мало значення, яке повністю відрізняється від значення змінної, що призведе хакерів до абсолютно неправильного шляху під час сканування.

Наприклад, в попередніх лекціях в скрипті GameController зберігаємо накопичену кількість балів від об'єктів, що піднімаємо, тобто:

```
private int score;
private int Score { get { return score; } set { score = value; } }

private void Awake()
{
    _instance = this;
}

private void Start()
{
    Score = 0;
    HUD.Instance.SetScore(Score.ToString());
}

public void SetScore(int add)
{
    Score += add;
    HUD.Instance.SetScore(Score.ToString());

    if (Score >= 10)
        HUD.Instance.ShowLevelWonWindow();
}
```

Доробимо наш скрипт для використання заміни значень змінних з використанням випадкових значень. Для цього створимо новий скрипт ObfuscateSettings.

```

public class ObfuscateSettings : MonoBehaviour
{
    static int random;

    public static void Initialize()
    {
        random = UnityEngine.Random.Range(10000, 99999);
    }

    public static int Obfuscate(int originalValue)
    {
        return random - originalValue;
    }

    public static int Deobfuscate(int obfuscatedValue)
    {
        return random - obfuscatedValue;
    }
}

```

Зробимо змінив в скрипті GameController:

```

private void Awake()
{
    _instance = this;

    ObfuscateSettings.Initialize();
    Score = ObfuscateSettings.Obfuscate(0);
}

private void Start()
{
    //Score = 0;
    //HUD.Instance.SetScore(Score.ToString());

    HUD.Instance.SetScore(ObfuscateSettings.Deobfuscate(Score).ToString());
}

public void SetScore(int add)
{
    //Score += add;
    //HUD.Instance.SetScore(Score.ToString());

    Score = ObfuscateSettings.Obfuscate(ObfuscateSettings.Deobfuscate(Score) + add);
    HUD.Instance.SetScore(ObfuscateSettings.Deobfuscate(Score).ToString());

    //if (Score >= 10)
    if (ObfuscateSettings.Deobfuscate(Score) >= 10)
        HUD.Instance.ShowLevelWonWindow();
}

```

Замість безпосередньої ініціалізації змінної очок ми ініціалізуємо її на початку в void Awake(). Перш ніж призначати значення за допомогою Obfuscate(value), обов'язково викликайте Initialize().

Тоді під час відображення значень ми деобфускуємо їх на льоту, викликавши Deobfuscate(value), таким чином, відображаючи справжні значення. Хакер спробував би знайти 0 або 10 або 20 тощо, але не зміг би їх знайти, оскільки реальні значення зовсім інші.

Honeypot – це вид кібербезпеки, що представляє собою фальшивий об'єкт чи систему, призначений для виявлення, вивчення чи відстеження кіберзлочинців. Honeypot може бути як програмним, так і апаратним засобом, розташованим в мережі з метою привертання уваги потенційних зловмисників.

Основні характеристики honeypot включають:

- Фальшиві дані – honeypot намагається видавати себе за реальну систему чи ресурс, ізолюючи його від решти реальних активів мережі;
- Захоплення даних – honeypot може записувати та аналізувати всю взаємодію зловмисників, зокрема, їхні спроби атак та методи;
- Виявлення загроз – за допомогою honeypot можна виявляти нові та еволюціонуючі загрози, адже вони привертають атаки, які можуть вказувати на нові та невідомі види атак.
- Навчання та аналіз – використання honeypot дозволяє вивчати методи та тактику зловмисників, а також покращувати стратегії кіберзахисту на основі отриманих даних;
- Попередження перед атакою – honeypot може слугувати попередженням перед реальними атаками, дозволяючи вчасно реагувати та захищати реальні системи.

2. Завдання до практичної роботи 8

Створити файли для зберігання та читання введених даних у форматах XML чи JSON. Зберігати інформацію в них у зашифрованому вигляді. Спочатку до них застосувати шифрування даних, а потім Base64-перетворення.

Реалізувати метод заміни значень змінних, що зберігаються в пам'яті комп'ютера, з використанням випадкових значень.

Ввести в дані, які зберігаються, непотрібні параметри. Додати на сцену об'єкти, які будуть змінювати ці параметри. Самі об'єкти не беруть участь в грі.

Література

1. Lanzinger F. 2D Game Development with Unity: CRC Press, 2020. – 444 p. URL: https://www.google.com.ua/books/edition/2D_Game_Development_with_Unity/hjwDEAAQBAJ?hl=ru&gbpv=0
2. Felicia P. The Ultimate Guide to 2D Games with Unity: Amazon Digital Services LLC – Kdp, 2020. – 488 p. URL: https://www.google.com.ua/books/edition/The_Ultimate_Guide_to_2D_games_with_Unit/9AYBEAAAQBAJ?hl=ru&gbpv=0
3. Halpern J. Developing 2D Games with Unity. Independent Game Programming with C#: Apress, 2018. – 383 p. URL: https://www.google.com.ua/books/edition/Developing_2D_Games_with_Unity/loF8DwAAQBAJ?hl=ru&gbpv=0
4. Murray Jeff W. 2D Unity. Your First Game from Start to Finish: No Starch Press, 2016. – 312 p. URL: https://www.google.com.ua/books/edition/2D_Unity/0-8cswEACAAJ?hl=ru
5. Nakov S. Programming Basics with C# / S. Nakov, Nakov's Team: SoftUni, 2019. – 408 p. URL: https://www.google.com.ua/books/edition/Programming_Basics_with_C/vzKyDwAAQBAJ?hl=ru&gbpv=0
6. Norton T. Learning C# by Developing Games with Unity 3D: Packt Publishing Ltd, 2013. – 292 p. URL: https://play.google.com/store/books/details/Terry_Norton_Learning_C_by_Developing_Games_with_U?id=7Zf9AAAAQBAJ
7. Zoubir Z. Mammeri. Cryptography: Algorithms, Protocols, and Standards for Computer Security: John Wiley & Sons, 2024. – 624 p. URL: <https://www.google.com.ua/books/edition/Cryptography/ecb0EAAAQBAJ?hl=ru&gbpv=1&dq=Cryptography&printsec=frontcover>
8. Paar C. Understanding Cryptography: A Textbook for Students and Practitioners / C. Paar, J. Pelzl, B. Preneel: Springer Berlin Heidelberg, 2010. – 372 p. URL: https://www.google.com.ua/books/edition/Understanding_Cryptography/f24wFELSzkoC?hl=ru&gbpv=0

Інтернет ресурси

1. Офіційний сайт Unity [Електронний ресурс] URL: <https://unity.com/>
2. Офіційний сайт Visual Studio Community [Електронний ресурс] URL: <https://visualstudio.microsoft.com/ru/vs/community/>