

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ МОРСЬКИЙ УНІВЕРСИТЕТ

Кафедра кібербезпеки та захисту інформації

Конспект лекцій

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

Конспект лекцій з освітнього компоненту **«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 / 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

Рецензент – Кобозєва Алла Анатоліївна, д.т.н., професор, зав. кафедри «Кібербезпека та захист інформації»

Зміст

Вступ.....	4
Лекція 1. Інтерфейс Unity. Сцена. Скрипти	5
1. Unity Hub та редактор Unity	5
2. Сцени	9
3. Асети.....	12
4. Скрипти	13
Лекція 2. 2D гра. Основні об'єкти	19
1. GameObject.....	19
2. Фізичний матеріал	24
Лекція 3. 2D гра. Анімація спрайтів	26
1. Спрайти.....	26
2. Анімація.....	29
Лекція 4. 2D гра. Рух персонажа	38
1. Основи руху персонажа	38
2. Переміщення персонажа	38
3. Рух по сходам.....	44
Лекція 5. 2D гра. Вороги.....	47
1. Перший тип ворога.....	47
2. Другий тип ворога	48
3. Третій тип ворога.....	50
Лекція 6. 2D гра. Оточення. Налаштування камери. Паралакс.....	54
1. Додавання оточення	54
2. Створення підвісного місту	54
3. Система частинок	57
4. Пульсуюча анімація предмету	58
5. Камера, що рухається за персонажем.....	59
6. Створення фону, що рухається	60
Лекція 7. Огляд популярних атак на комп'ютерні ігри та методи їх проведення	63
1. Основні визначення.....	63
2. Популярні атаки на комп'ютерні ігри.....	64
Лекція 8. Методи захисту від злому в іграх.....	68
1. Захист ігрових даних.....	68
2. Обфускація	76
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.

Лекція 1. Інтерфейс Unity. Сцена. Скрипти

1. Unity Hub та редактор Unity
2. Сцена
3. Асети
4. Скрипти

1. Unity Hub та редактор Unity

Unity Hub – це корисний інструмент для керування проєктами та встановленими версіями Unity. Використовуйте Hub для роботи з кількома версіями редактора Unity та пов'язаних з ними компонентів для створення нових або відкриття створених проєктів.

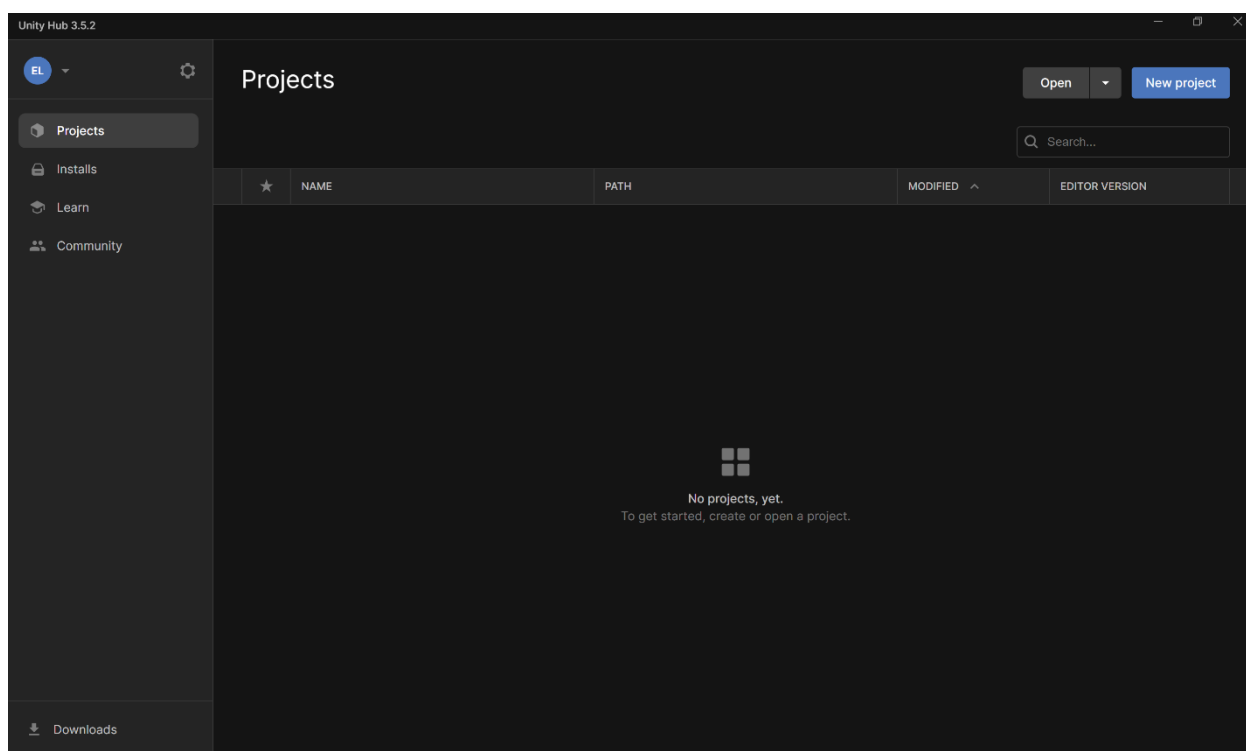


Рисунок 1.1 – Вікно Unity Hub

Unity Hub – це десктопна програма, спроектована для зручної роботи користувачів. З нього відбувається доступ до екосистеми ігрового двигуна Unity, робота з менеджером проєктів, створених у Unity, управління ліцензіями та встановлення додаткових компонентів.

Unity Hub є своєрідною “точкою старту”, в якій відбувається створення нових проєктів (вкладка Projects), встановлення різних версій Unity (вкладка Installs) тощо.

У центральній частині програми вказані проєкти (Projects), з якими ви працюєте. Якщо ви використовуєте Unity вперше, то це вікно має бути порожнім.

При натисканні на закладку Installs у лівому меню відкриється вікно вибору версій Unity для встановлення або буде показано перелік уже встановлених версій Unity.

Якщо надалі вам знадобляться інші версії середовища розробки Unity (наприклад, ви знайдете і захочете подивитися готові проєкти, зроблені під попередні версії середовища розробки), – то ви завжди зможете відкрити Unity Hub, перейти у вкладку Install і завантажити версії Unity і модулі.

Для створення порожнього 2D-проєкту зайдіть у Unity Hub та натисніть кнопку New project.

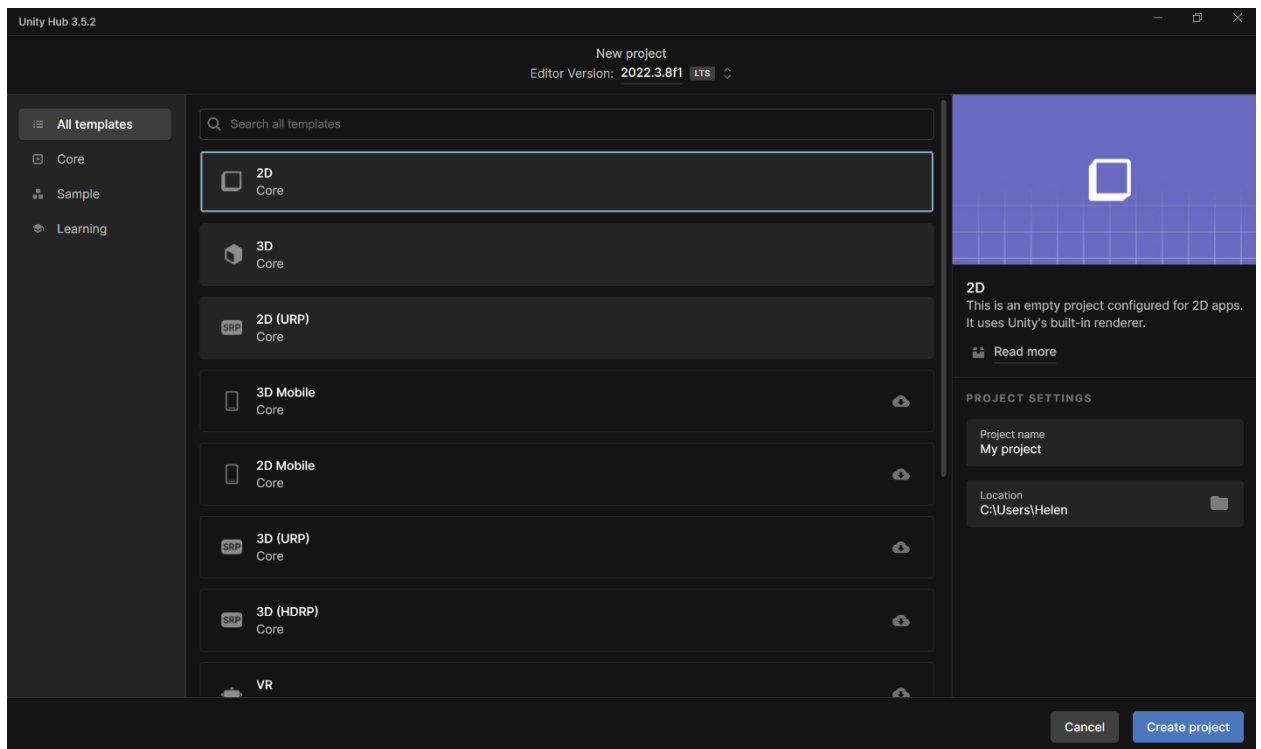
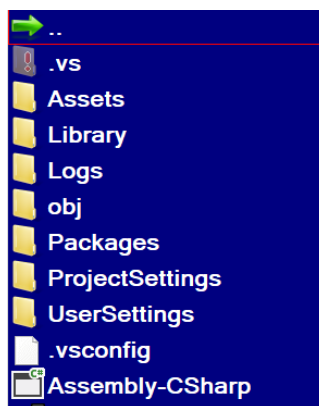


Рисунок 1.2 – Вікно для вибору шаблону для гри, що створюється

Вибираємо кнопку 2D, вказуємо ім'я проєкту та його місцезнаходження та натискаємо кнопку Create project. Коли новий проєкт ініціалізується, з'явиться редактор Unity.

Проект Unity – це не єдиний файл, а папка, що містить деяку кількість папок. Розглянемо три найважливіші підпапки:

- Assets,
- ProjectSettings
- Library.



Папка Assets містить усі файли, які використовуються грою: рівні, текстури, звукові ефекти та сценарії. Папка Library містить внутрішні дані для Unity, а папка ProjectSettings – файли з налаштуваннями проєкту. В основному не доведеться торкатися файлів у папках Library та ProjectSettings.

Інтерфейс Unity відображається як набір панелей. Кожна панель має вкладку ліворуч угорі, за яку панель можна перемістити і тим самим змінити макет програми. Також, переміщуючи панель за вкладку, її можна перетворити на самостійне вікно. Не всі панелі видимі за замовчуванням, і в міру розробки гри ви відкриватимете нові панелі через меню Window.

Інтерфейс Unity розбитий на кілька частин: вкладка Scene, вкладка Game, панель інструментів, вкладка Hierarchy, панель Inspector, вкладки Project та Console. Кожна частина має власне призначення, при цьому всі вони відіграють важливу роль у циклі створення гри.

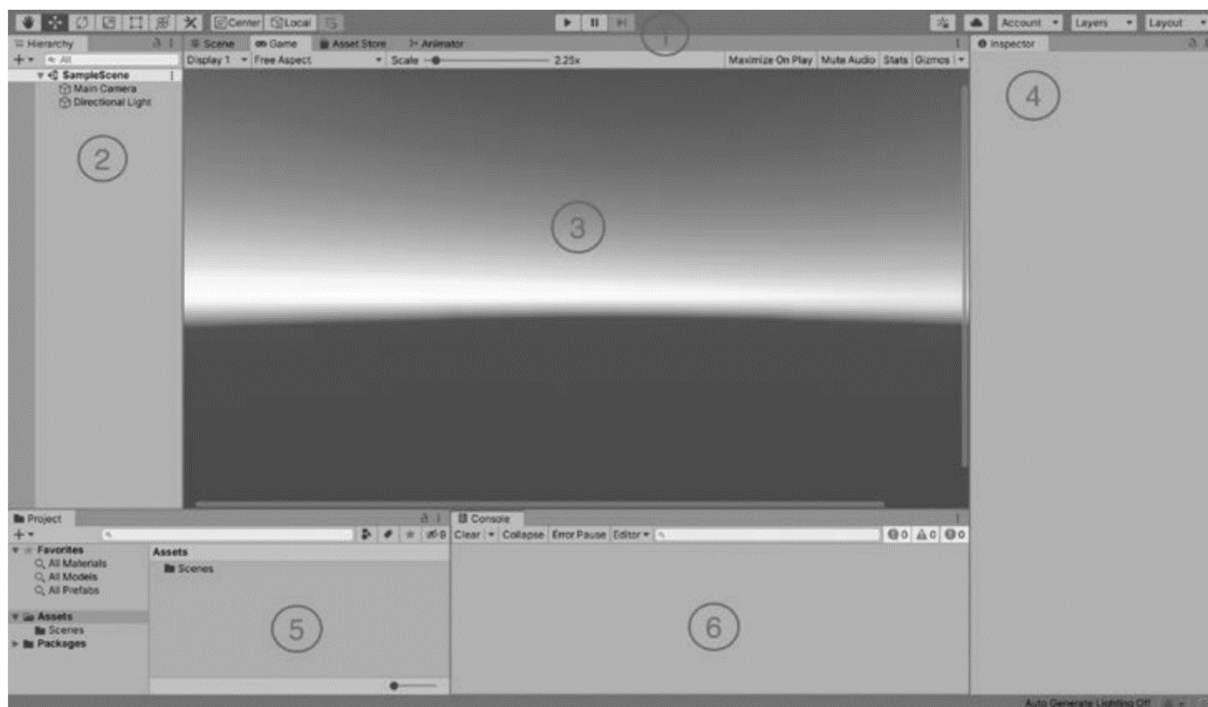


Рисунок 2.3 – Вікно редактору Unity

Панель Toolbar – найвища частина редактора Unity. На цій панелі можна керувати об'єктами (кнопки зліва), а також запускати та призупиняти гру (центральні кнопки). Праворуч розташовані кнопки сервісів Unity, шари-маски та опції макетів.

На панелі *Hierarchy* відображаються всі елементи, які зараз знаходяться на ігровій сцені. У новоствореному проєкті є камера та спрямоване джерело світла, але в міру створення гри ця панель почне наповнюватися об'єктами.

Панелі Game і Scene – найвізуальніші наочні частини редактора. Панель Scene – це робоча поверхня, на якій можна розміщувати та переміщувати 2D- та 3D-об'єкти. Коли ви натискаєте кнопку Play, з'являється вікно Game, в якому буде показана та сама сцена, але вже з робочими запрограмованими взаємодіями.

Редактор Unity завжди знаходиться в одному з двох режимів: Edit Mode (Режим редагування) або Play Mode (Режим відтворення). У режимі редагування, що діє за замовчуванням, можна створювати сцени, налаштовувати ігрові об'єкти та виконувати інші дії, пов'язані зі створенням гри. У режимі відтворення ви можете грати в свою гру і взаємодіяти зі сценою.

Щоб перейти в режим відтворення, клацніть по кнопці Play (Грати) у верхній частині вікна редактора – Unity запустить гру. Щоб залишити режим відтворення, натисніть кнопку Play (Грати) ще раз. Перебуваючи в режимі відтворення, можна зупинити гру, клацнувши піктограму Pause (Пауза) в центрі панелі керування режимом відтворення.

Проєкти в Unity поділені на сцени. Кожна сцена містить колекцію ігрових об'єктів. Сцену можна розглядати як один рівень у грі, але сцени також допомагають поділити гру на керовані фрагменти. Наприклад, головне меню гри зазвичай реалізується у вигляді окремої сцени, як і кожен її рівень.

Панель Inspector – узагальнений інструмент для перегляду та редагування властивостей ваших об'єктів. Вибравши, наприклад, компонент Main Camera, ви побачите, що він має кілька частин (в Unity вони називаються компонентами) і всі вони знаходяться саме тут.

У вікні *Project* представлені всі ресурси, які зараз знаходяться у вашому проєкті. Якщо простіше, то це всі файли та папки вашого проєкту. Додавати нові файли потрібно до папки *Assets*.

На панелі *Console* з'являться вихідні дані, які виводяться створеними сценаріями.

Подання сцени може бути в одному з п'яти різних режимів. Перемикач режимів, що знаходиться у вікні, керує режимом взаємодії з поданням сцени.



Режим захоплення (*Grab mode*) – у цьому режимі можна натиснути ліву кнопку миші та зрушити уявлення.

Режим переміщення (*Translation mode*) – у цьому режимі можна переміщати виділені об'єкти.

Режим обертання (*Rotation mode*) – у цьому режимі можна повертати виділені об'єкти.

Режим масштабування (*Scale mode*) – в цьому режимі можна змінювати розміри виділених об'єктів.

Режим прямокутника (*Rectangle mode*) – у цьому режимі можна переміщати вибрані об'єкти та змінювати їх розміри, використовуючи двовимірні маркери. Це особливо зручно при формуванні двовимірних сцен або інтерфейсу користувача.

Перемикання між режимами представлення сцени можна здійснювати за допомогою перемикача або клавіш Q, W, E, R та T.

Обирати об'єкти можна в будь-якому режимі, крім режиму захоплення.

Підтримується кілька способів навігації у поданні сцени:

- Клацнути по піктограмі із зображенням руки ліворуч угорі, щоб перейти в режим захоплення, потім натиснути ліву кнопку миші та зрушити сцену в потрібному напрямку.
- Утримуючи клавішу Alt, натиснути ліву кнопку миші та повернути сцену на потрібний кут.
- Вибрати об'єкт у сцені, клацнувши по ньому лівою кнопкою або вибравши його в панелі ієрархії, помістити вказівник миші у межі панелі з представленням сцени та натиснути F, щоб передати фокус введення вибраному об'єкту.
- Утримуючи праву кнопку, перемістити мишу, щоб озирнутися. Поки права кнопка миші залишається натиснутою, можна використовувати клавіші W, A, S і D, щоб змістити камеру вперед, ліворуч, назад та праворуч. Клавішами Q та E камеру можна зміщувати вгору та вниз. Якщо при цьому ще натиснути та утримувати клавішу Shift, переміщення відбуватимуться швидше.

Панель Hierarchy (Ієрархія) відображає список усіх об'єктів у сцені, відкритій на даний момент. Працюючи зі складними сценами, ієрархія дозволяє швидко знаходити потрібні об'єкти за іменами.

Ієрархія, як випливає з назви, дозволяє також бачити стосунки батьків-нащадків між об'єктами. У Unity об'єкти можуть містити інші об'єкти. Панель ієрархії дозволяє досліджувати такі дерева об'єктів. Також підтримується можливість переупорядковувати об'єкти списки, перетягуючи їх мишею.

Інспектор відображає інформацію про об'єкт, обраний в даний момент, і саме тут можна буде здійснювати налаштування своїх ігрових об'єктів. Інспектор відображає список усіх компонентів, підключених до вибраного об'єкта чи ресурсу. Для різних компонентів відображається різна інформація.

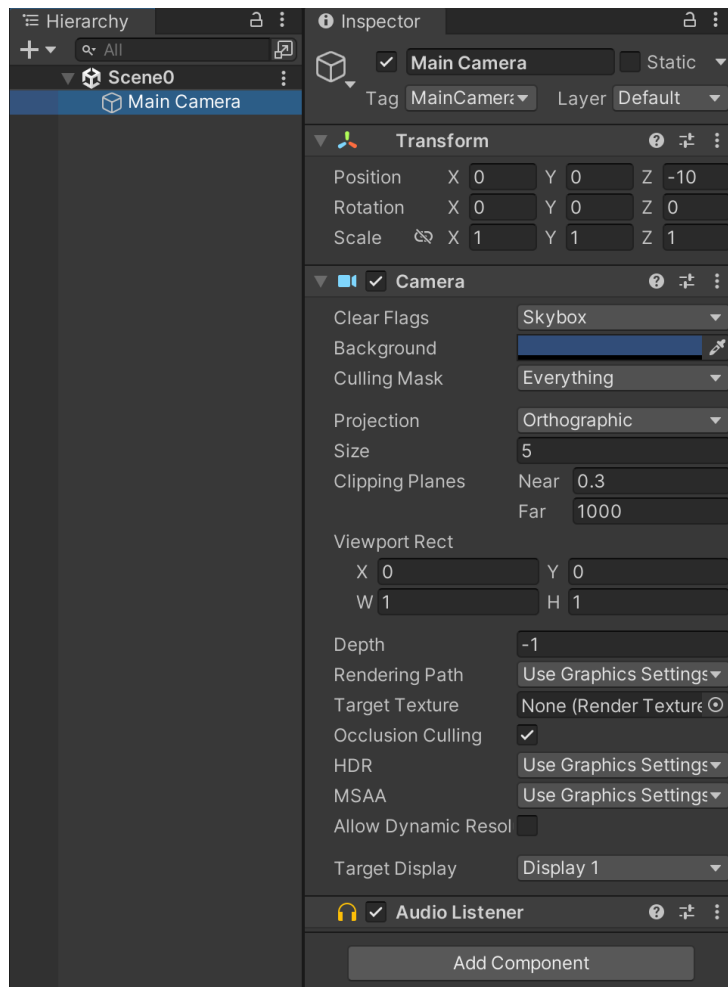


Рисунок 1.4 – Панель Hierarchy та панель Inspector для об'єкту MainCamera

2. Сцени

Сцени містять об'єкти гри. Вони можуть використовуватися для створення головного меню, окремих рівнів та інших цілей. Кожен файл сцени можна вважати окремим ігровим рівнем. У кожній сцені можна розмістити об'єкти оточення, загородження, декорації, шматочками створюючи дизайн і саму гру.

Коли ви створюєте новий проєкт Unity, у поданні сцени відображатиметься нова сцена. Це сцена без назви та збереження. Сцена буде порожньою, за винятком об'єктів – камер та спрямованого світла (тільки в 3D-проєктах).

Використовуйте діалогове вікно «Нова сцена» для створення нових сцен із певних шаблонів сцен у вашому проєкті. Ви також можете використовувати діалогове вікно «Нова сцена» для пошуку шаблонів сцен та керування ними. За замовчуванням діалогове вікно «Нова сцена» відкривається під час створення нової сцени з меню (Файл → Нова сцена).

Коли ви створюєте нову сцену з меню, Unity автоматично копіює базовий шаблон проєкту і додає нову сцену до будь-якої папки, відкритої на даний момент у вікні проєкту. Нову сцену потрібно зберегти.

Щоб відкрити сцену, виконайте одну з таких дій:

- У вікні «Проєкт» двічі клацніть об'єкт сцени.
- У меню виберіть Файл → Нова сцена.
- У меню виберіть Файл → Останні сцени → [назва сцени]

Якщо ваша поточна сцена містить незбережені зміни, Unity запропонує зберегти сцену або скасувати зміни.

Існує кілька способів переходу між сценами, а саме:

- Зміна сцен за назвою;

- Зміна сцен за індексами;
- Зміна сцен за допомогою параметрів.

Зміна сцен за назвою. Цей спосіб найпростіший, і зустрічається дуже часто. Він використовується тоді, коли нам потрібно перейти точно на певну сцену. Для цього створимо C# скрипт з назвою, наприклад, Scenes. І пропишемо в ньому наступний код:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Scenes : MonoBehaviour
{
    public void OpenMenu()
    {
        SceneManager.LoadScene("Menu");
    }

    public void OpenGame()
    {
        SceneManager.LoadScene("Game");
    }
}
```

Зміна сцен за індексами. Даний спосіб аналогічний попередньому, за винятком того, що ми замість назв сцен, використовуємо їх індекси. Цей спосіб корисний у тих випадках, коли нам необхідно перейти на наступну або попередню сцену.

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Scenes : MonoBehaviour
{
    public void OpenMenu()
    {
        SceneManager.LoadScene(0);
    }

    public void OpenGame()
    {
        SceneManager.LoadScene(1);
    }
}
```

Завдяки функціям відкриваються трохи більше можливостей для переходу на сцени. Наприклад, ми можемо перейти на наступну чи попередню.

```

using UnityEngine;
using UnityEngine.SceneManagement;

public class Scenes : MonoBehaviour
{
    public void NextLevel()
    {
        var index = SceneManager.GetActiveScene().buildIndex;
        SceneManager.LoadScene(index + 1);
    }

    public void PreviousLevel()
    {
        var index = SceneManager.GetActiveScene().buildIndex;
        SceneManager.LoadScene(index - 1);
    }
}

```

Перед тим, як працювати зі сценами за їх індексами, нам необхідно додати всі сцени до Build Settings. Для цього тиснемо на вкладку File → Build Settings. Далі перетягуємо всі свої сцени в область Scenes in Build і розставляємо їх по порядку. Після цього просто закриваємо це вікно.

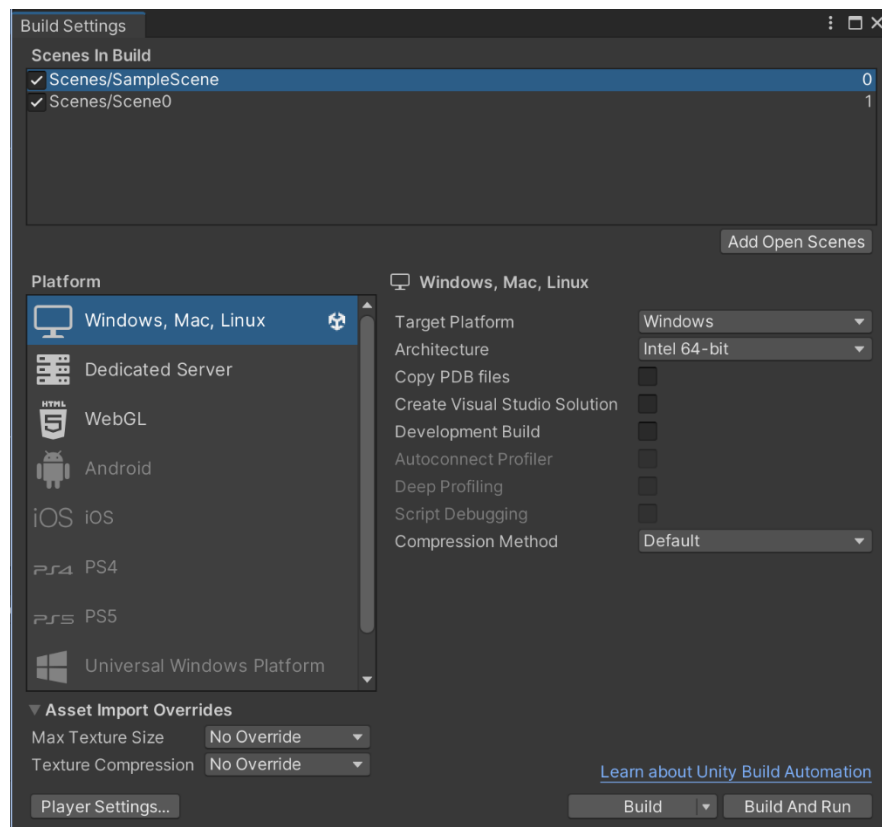


Рисунок 1.5 – Вікно Build Settings

Зміна сцен за допомогою параметрів. Даний спосіб аналогічний попередньому, тільки тут ми переходимо на певну сцену, виходячи з переданого аргументу в нашу функцію.

```

using UnityEngine;
using UnityEngine.SceneManagement;

public class Scenes : MonoBehaviour
{
    public void GoToLevel(int number)
    {
        SceneManager.LoadScene(number);
    }
}

```

3. Асети

Асети – це компоненти, які являють собою графіку, звуковий супровід або скрипти. Вони прикріплюються до об'єктів і становлять важливу частину гри.

Папка Assets – головна папка, в якій містяться всі асети, які можуть бути використані проектом у Unity. Вміст Project view відповідає вмісту папки Assets. Більшість функцій API припускають, що все міститься в папці Assets і не вимагають явної вказівки розташування. Рекомендується під кожен вид асетів створювати свою окрему папку в Assets.

Нижче наведено деякі з найбільш поширених типів асетів, які можна використовувати при роботі з Unity.

- файли 3D-моделей – Unity підтримує формат файлів FBX, що означає, що ви можете імпортувати дані з будь-якого програмного забезпечення для 3D-моделювання, що підтримує FBX. Файли 3D-моделей можуть містити багато типів активів, таких як сітки, анімація, матеріали та текстури.
- файли зображень – Unity імпортує файли зображень у вигляді текстур. Unity підтримує найпоширеніші типи файлів зображень, такі як BMP, TIF, TGA, JPG та PSD. Якщо ви зберігаєте багат шарові файли Photoshop (.psd) у папці, Unity імпортує їх як зведені зображення.
- аудіофайли – Unity підтримує безліч форматів аудіофайлів, наприклад, mp3, wav, flac та інші.
- текст, HTML, XML, JSON – Unity може імпортувати довільні дані із файлів, що дозволяє зберігати та використовувати дані із зовнішніх джерел.

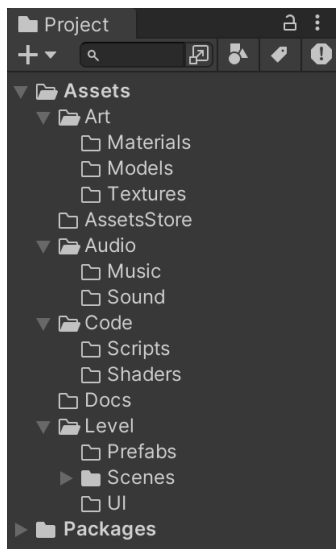
Texture – текстура. Зазвичай текстурами називають двовимірні зображення, які накладають на тривимірні моделі.

Background – тло. Так називається зображення, зазвичай відповідне розмірам ігрового поля, яке є фоном для інших зображень. Фон може бути нерухомим та рухомим.

Хоча єдиного способу організації проекту Unity не існує, кілька ключових рекомендацій:

- Задokumentуйте угоди про імена та структуру папок. Посібник із стилю та/або шаблон проекту спрощують пошук та організацію файлів.
- Не використовуйте пробіли в іменах файлів та папок. У інструментах командного рядка Unity виникають проблеми з іменами шляхів, що містять пробіли. Використовуйте CamelCase як альтернативу пробілам.
- Окремі зони тестування чи пісочниці. Створіть окрему папку для невиробничих сцен та експериментів. Підпапки з іменами користувачів можуть розділити вашу робочу область на учасників команди.
- Тримайте свої внутрішні ресурси окремо від сторонніх. Якщо ви використовуєте ресурси з Asset Store або інших плагінів, швидше за все, вони мають власну структуру проекту. Тримайте свої активи окремо.

Хоча заданої структури папок немає, розглянемо приклад того, як ви можете налаштувати свій проєкт Unity.



Unity генерує мета-файл для кожного файлу усередині проєкту. Хоча мета-файл створюється автоматично, він містить багато інформації про файл, з яким він пов'язаний. Коли Unity створює файл .meta для об'єкта, він записує ідентифікатор об'єкта у файл .meta і зберігає файл .meta у тому місці, що й файл ресурсу.

Мета-файли містять важливу інформацію про те, як актив використовується в проєкті, і вони повинні залишатися у файлі активу, до якого вони належать. Якщо ви переміщуєте або перейменовуєте ресурс у власному вікні проєкту Unity, Unity також автоматично переміщує або перейменовує відповідний файл .meta.

Однак якщо ви переміщуєте або перейменовуєте ресурс за межами Unity, ви повинні перемістити або перейменувати файл .meta, щоб він відповідав.

Якщо ресурс втрачає свій мета-файл (наприклад, якщо ви переміщуєте або перейменовуєте ресурс за межами Unity, але не переміщуєте або перейменовуєте відповідний файл .meta), будь-яке посилання на цей ресурс у вашому проєкті не працює.

У цій ситуації Unity зауважує, що ресурс не має відповідного мета-файлу, створює новий для переміщеного/перейменованого ресурсу, ніби це був абсолютно новий ресурс, і видаляє старий «безхазяйний» файл .meta.

Цей процес може спричинити серйозні проблеми у вашому проєкті. Наприклад:

- Якщо ресурс текстури втрачає свій файл .meta, будь-які матеріали, що використовують цю текстуру, втрачають посилання на цю текстуру. Щоб це виправити, вам потрібно буде вручну перепризначити цю текстуру всім матеріалам, для яких вона потрібна.
- Якщо ресурс сценарію втрачає свій файл. Щоб виправити це, вам потрібно буде вручну перепризначити цей скрипт для будь-яких ігрових об'єктів, яким він потрібен.

4. Скрипти

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.

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.

Скрипт взаємодіє із внутрішніми механізмами Unity за рахунок створення класу, успадкованого від вбудованого класу, званого MonoBehaviour. Клас MonoBehaviour в Unity - це базовий клас, від якого успадковуються всі скрипти користувача. Він надає доступ до основних функцій Unity, таких як керування ігровими об'єктами, обробка подій, взаємодія з фізикою, а також керування життєвим циклом об'єктів.

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 GameObject, 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.

При створенні нового скрипту на C#, Unity створює файл з розширенням cs та наступним вмістом:

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

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

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

Фактично можна змінити шаблон сценарію, який Unity використовує під час створення нового сценарію C#. Файл шаблону сценарію, який ми збираємося змінити, називається 81-C# Script-NewBehaviourScript.cs.txt; він знаходиться в папці C:\ProgramFiles\Unity\Hub\Editor\2022.2.1f1\Editor\Data\Resources\ScriptTemplates.

Щоб змінити шаблон, запустіть Visual Studio Community 2022 як адміністратор і відкрийте цей файл. Для цього запустіть Visual Studio та виберіть Continue without code (Продовжити без коду) внизу праворуч. Після запуску виберіть File → Open → File («Файл» → «Відкрити» → «Файл»), перейдіть до файлу, який потрібно редагувати, і двічі клацніть назву файлу.

Щоб зберегти зміни, вибираємо File → Save 81-C# Script-NewBehaviourScript.cs.txt As У вікні для збереження клацніть стрілку вниз праворуч від кнопки Save («Зберегти») у нижньому правому куті та виберіть Save with Encoding ... («Зберегти з кодуванням ...») У діалоговому вікні Advanced Save Options («Додаткові параметри збереження») змініть Line Endings на Windows (CR LF) і натисніть кнопку «ОК».

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.

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.

Розглянемо інші функції в Unity.

Функції	Опис
<u>Update</u>	викликається перед малюванням кадру та перед розрахунком анімації
<u>FixedUpdate</u>	викликається перед кожним оновленням фізичних даних. Т.к. оновлення фізики та кадру відбувається не з однаковою частотою, то ви отримаєте більш точні результати від коду фізики
<u>LateUpdate</u>	можливість внести додаткові зміни у момент, коли у всіх об'єктів у сцені відпрацювали функції <u>Update</u> та <u>FixedUpdate</u> та розрахувалися всі анімації.
<u>Start</u>	викликається до оновлення першого кадру чи фізики об'єкту.
<u>Awake</u>	викликається для кожного об'єкта в сцені під час завантаження сцени. Для різних об'єктів функції <u>Start</u> та <u>Awake</u> і викликаються в різному порядку, всі <u>Awake</u> будуть виконані до першого виклику <u>Start</u> .
<u>OnGUI</u>	У Unity є система для відтворення елементів керування GUI поверх того, що відбувається в сцені, і реагування на кліки по цих елементах. Цей код обробляється дещо інакше, ніж звичайне оновлення кадру, тому він повинен бути поміщений у функцію <u>OnGUI</u> , яка буде періодично викликатися.
<u>OnMouseOver</u> , <u>OnMouseDown</u>	дозволяє скрипту реагувати на дії з мишею користувача
<u>OnCollisionEnter</u> , <u>OnCollisionStay</u> , <u>OnCollisionExit</u>	будуть викликані на початок, продовження та завершення контакту (зіткненнях) з об'єктом
<u>OnTriggerEnter</u> , <u>OnTriggerStay</u> , <u>OnTriggerExit</u>	будуть викликані, коли колайдер об'єкта налаштований як <u>Trigger</u> (тобто цей колайдер просто визначає, що щось перетинає і не реагує фізично). Ці функції можуть бути викликані кілька разів поспіль, якщо виявлено більше одного контакту під час оновлення фізики, тому в функцію передається параметр, що надає додаткову інформацію про зіткнення (координати, "особистість" вхідного об'єкта і т.д.).

Unity дозволяє змінити значення змінних скриптів у запущеній грі. Це дуже корисно, щоб побачити ефекти від змін одразу ж, без встановлення та перезапуску. Коли програвання закінчується, значення змінних змінюються в цьому стані, яке вони мали до натискання кнопки Play.

Атрибути (Attributes) – це маркери, які можуть бути розміщені перед класом, властивостями або функціями в скрипті, щоб забезпечити особливу поведінку.

Наприклад, атрибут *HideInInspector* може бути доданий перед оголошенням властивостей для попереднього відображення цих властивостей в інспекторі, навіть якщо він публічний. В C# він поміщається 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*.

Unity створює мітку у вікні Inspector шляхом додавання пробілу скрізь, де в імені змінної зустрічається велика літера. Однак це виключно для відображення, і ви повинні завжди в коді використовувати ім'я змінної.

Атрибут Header, коли приєднується до поля, змушує Unity виводити напис над полем в інспекторі.

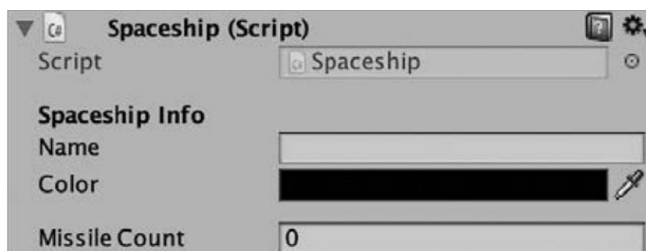
Атрибут Space діє аналогічно, але додає порожній простір. Обидва атрибути зручно використовувати для візуальної організації вмісту інспектора.

Наприклад:

```
public class Spaceship: MonoBehaviour
{
    [Header("Spaceship Info")]
    public string name;
    public Color color;

    [Space]

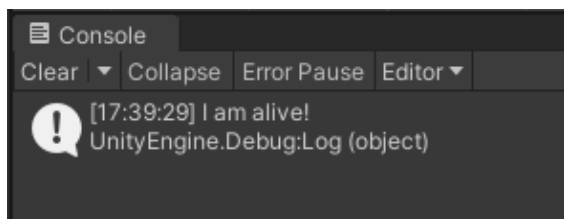
    public int missileCount;
}
```



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).



Найпростішим і найпоширенішим є випадок, коли скрипту необхідно звернутися до інших компонентів, приєднаних до того ж *GameObject*. Компонент насправді є екземпляром класу, тому першим кроком буде отримання посилання на екземпляр компонента, з яким

ви хочете працювати. Це робиться за допомогою функції *GetComponent*. Типово, об'єкт компонента зберігають в змінну, це робиться C# за допомогою наступного синтаксису:

```
void Start ()
{
    Rigidbody rb = GetComponent<Rigidbody>();
}
```

Як тільки ви маєте посилання на екземпляр компонента, ви можете встановлювати значення його властивостей, тих же, які ви можете змінити у вікні Inspector.

Немає причини, через яку ви не можете мати більше одного скрипта користувача, приєднаного до одного і того ж об'єкта.

Якщо вам потрібно звернутися до одного скрипту з іншого, ви можете використовувати, як завжди, *GetComponent*, використовуючи при цьому ім'я класу скрипту (або ім'я файлу), щоб вказати, який тип компонента вам потрібен.

Якщо ви спробуєте вибрати компонент, який не був доданий до ігрового об'єкта, *GetComponent* поверне null; виникне помилка порожнього посилання під час виконання (null reference error at runtime), якщо спробуєте змінити будь-які значення у порожньому об'єкті.

Скрипти можуть відстежувати інші об'єкти. Наприклад, ворог, що переслідує, повинен знати позицію гравця. Unity надає кілька шляхів отримання інших об'єктів.

Найпростіший спосіб знайти потрібний ігровий об'єкт – додати до скрипту змінну типу *GameObject* з рівнем доступу *public*. Змінну буде видно у вікні Inspector. Тепер можна перетягнути об'єкт зі сцени або з панелі Hierarchy в цю змінну, щоб призначити його.

Наприклад:

```
public class Enemy : MonoBehaviour
{
    public GameObject player;

    void Start()
    {
        // Start the enemy ten units behind the player character.
        transform.position = player.transform.position - Vector3.forward *
10f;
    }
}
```

Часто зручно знаходити об'єкти під час виконання, і Unity надає два основні способи зробити це.

Пошук дочірніх *GameObjects*. Іноді ігрова сцена використовує декілька ігрових об'єктів одного типу, таких як вороги, шляхові точки та перешкоди. Їх може знадобитися відстежувати за допомогою певного сценарію, який контролює або реагує на них (наприклад, усі шляхові точки можуть бути доступні для сценарію пошуку шляху).

Використання змінних для зв'язування цих об'єктів *GameObject* є можливістю, але це робить процес проектування стомлюючим, якщо кожен нову точку маршруту потрібно перетягувати до змінної в сценарії. Так само, якщо шляхову точку видалено, видаляти посилання змінної на відсутній об'єкт *GameObject* – це незручність.

У подібних випадках часто краще керувати набором *GameObjects*, роблячи їх усіма нащадками одного батьківського *GameObject*.

Дочірні *GameObjects* можна отримати за допомогою батьківського компонента *Transform* (оскільки всі *GameObjects* неявно мають *Transform*)

Наприклад:

```

using UnityEngine;

public class WaypointManager : MonoBehaviour {
    public Transform[] waypoints;

    void Start() {
        waypoints = new Transform[transform.childCount];
        int i = 0;

        foreach (Transform t in transform) {
            waypoints[i++] = t;
        }
    }
}

```

Ви також можете знайти заданий дочірній об'єкт за ім'ям, використовуючи функцію `Transform.Find`:

```
transform.Find("Gun");
```

Об'єкт або колекція об'єктів може бути знайдена за її тегом, використовуючи функції `GameObject.FindWithTag` і `GameObject.FindGameObjectsWithTag`.

```

void Start() {
    player = GameObject.FindWithTag("Player");
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
}

```

Деякі ігри мають постійну кількість об'єктів на сцені, проте зазвичай персонажі, скарби та інші об'єкти створюються та видаляються під час гри.

У Unity ігровий об'єкт (`GameObject`) може бути створений, використовуючи функцію *Instantiate*, яка робить копію існуючого об'єкта.

Об'єкт, з якого береться копія, не повинен бути присутнім на сцені. Набагато частіше використовується префаб, який був перетягнутий на відкриту змінну (`public variable`) із файлів проекту на панелі Project.

Також є функція *Destroy*, яка знищить об'єкт після завершення завантаження кадру або опціонально після короткої паузи.

Контрольні питання

1. Що таке Unity Hub та для чого він використовується?
2. Для чого використовується папка Assets?
3. Для чого використовується вкладка Hierarchy?
4. Для чого використовується панель Inspector?
5. В яких режимах може бути подання сцени?
6. Що таке сцена та для чого вона призначена?
7. З яких ассетів складається папка Assets?
8. Від якого класу спадкуються скрипти в Unity?
9. Коли виконується метод `Start()`?
10. Що робить команда `Debug.Log()`?
11. Коли виконується метод `Update()`?
12. Як у скрипті звернутись до позиції поточного об'єкта?
13. Як у скрипті звернутися до швидкості об'єкта?

Лекція 2. 2D гра. Основні об'єкти

1. GameObject
2. Фізичний матеріал

1. GameObject

Клас Unity GameObject 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.

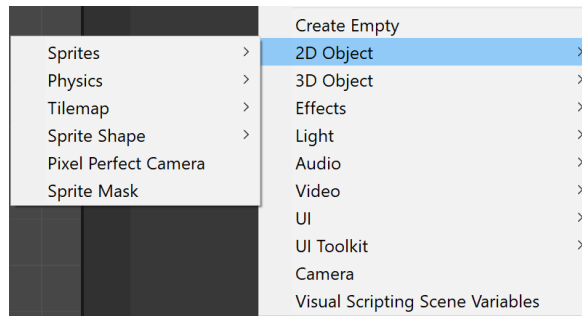


Рисунок 2.1 – Приклад GameObject

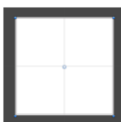
Види 2D примітивів GameObjects:

- Triangle
- Square
- Circle
- Capsule
- Isometric Diamond
- Hexagon Flat-Top
- Hexagon Point-Top
- 9-Sliced

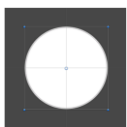
Примітив Triangle 2D – це білий рівнобедрений трикутник із основою, розмір якої становить одну одиницю. Ви можете використовувати його як заповнювач для різних елементів вашої сцени, таких як перешкоди або частини інтерфейсу користувача. Ви можете додати 2D-компонент Polygon Collider до примітиву, щоб він взаємодіяв з іншими об'єктами гри та двовимірною системою фізики.



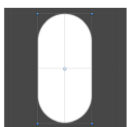
Примітив Square 2D – це білий квадрат розміром 1×1 одиниці. Ви можете використовувати його як заповнювач для різних елементів, наприклад, перешкод або платформ. Ви можете додати 2D-компонент Box Collider до примітиву, щоб він взаємодіяв з іншими об'єктами гри та двовимірною системою фізики.



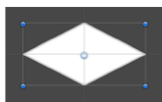
Примітив Circle 2D – це біле коло, діаметр якого становить одну одиницю Unity. Ви можете використовувати його як заповнювач для різних елементів сцени, наприклад, перешкод або реквізиту. Ви можете додати Circle Collider 2D до примітиву, щоб він взаємодіяв з іншими GameObject і системою 2D-фізики.



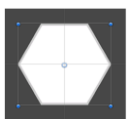
Примітив Capsule 2D – це біла капсула розміром 1×2 одиниці. Ви можете використовувати його як заповнювач для різних елементів у вашій сцені, наприклад, перешкоди, опори або заміни для персонажа. Ви можете додати Capsule Collider 2D до примітиву, щоб він взаємодіяв з іншими GameObject і системою 2D-фізики.



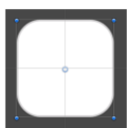
Примітив Isometric Diamond 2D – це білий спрайт у формі ромба розміром 1×0,5 одиниці. Цей спрайт часто використовується як заповнювач для плиток, розміщених на ізометричних картах плиток. Пікселі вгорі та внизу цього спрайту злегка обрізані, щоб покращити мозаїку.



Примітив Hexagon Flat-Top 2D – це правильний шестикутник шириною в одну одиницю; його плоскі сторони орієнтовані вгору та вниз. Цей спрайт часто використовується як заповнювач для плиток, розміщених на шестикутних картах плиток із плоским верхом. Пікселі ліворуч і праворуч від цього спрайту злегка обрізані для покращення мозаїки.



Примітив 9-Sliced 2D – це білий квадрат із заокругленими кутами розміром 1×1. Цей спрайт було розділено на 9 частин із межами розміром 64 пікселі з кожного боку. Цей спрайт в основному використовується з режимами малювання Sliced і Tiled компонента Sprite Renderer. Ви можете використовувати спрайт із 9 фрагментами як гнучкий заповнювач для різних елементів у вашій сцені та проєкті. Ви можете додати Box Collider 2D із увімкненою автомозаїкою, щоб спрайт взаємодіяв з іншими об'єктами та системою 2D-фізики.



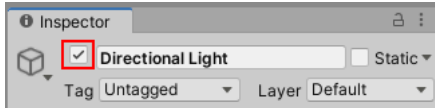
Метод `GameObject.CreatePrimitive()` створює 3D-ігровий об'єкт із примітивним рендерером сітки та відповідним коллайдером.

Наприклад:

```
GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);  
cube.transform.position = new Vector3(0, 0.5f, 0);
```

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, якщо він представлений прапорцем ліворуч від імені об'єкта. Ви можете керувати цим за допомогою `GameObject.SetActive`.



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.

Ви можете використовувати функцію `GameObject.FindWithTag()`, щоб знайти GameObject, налаштувавши її на пошук будь-якого об'єкта, що містить потрібний вам тег.

Наприклад:

```
GameObject respawn = GameObject.FindWithTag("Respawn");
```

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.

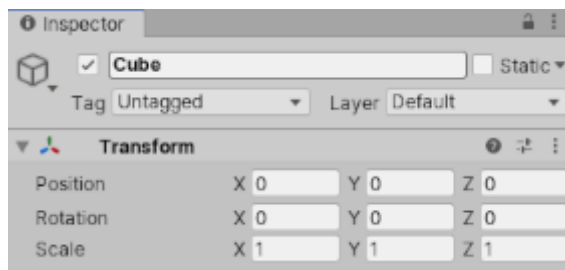


Рисунок 2.2 – Компонент Transform

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.

Оскільки компонент Rigidbody керує переміщенням об'єкта, до якого він прикріплений, вам не слід намагатися впливати на об'єкт із коду за допомогою зміни таких властивостей Transform, як position та rotation. Натомість вам слід застосовувати сили для того, щоб штовхати об'єкт і дозволити фізичному движку розрахувати результати.

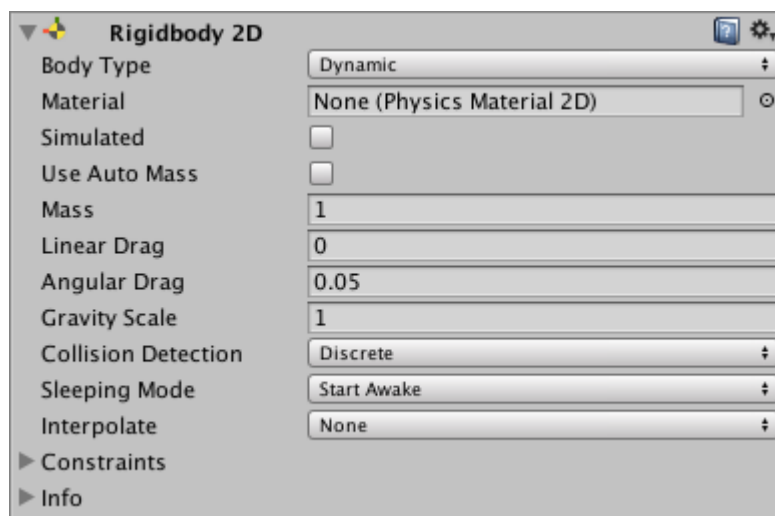


Рисунок 2.3 – Приклад компоненти Rigidbody 2D

У компоненті Rigidbody 2D вгорі є налаштування Body Type. Вибраний варіант впливає на інші параметри, доступні в компоненті. Існує три варіанти Body Type, кожен визначає загальну та фіксовану поведінку.

- Dynamic
- Kinematic
- Static

Dynamic Rigidbody 2D призначений для переміщення під час моделювання. Він має повний набір доступних йому властивостей, таких як кінцева маса і лобовий опір, і на нього впливають гравітація та сили. Динамічне тіло стикається з будь-яким іншим типом тіла і є найбільш інтерактивним з типів тіла.

Зміна положення Dynamic Rigidbody 2D відбувається відповідно до його швидкості. Можна змінити це безпосередньо за допомогою сил, прикладених до нього за допомогою скриптів або опосередковано через зіткнення та гравітацію.

Кінематичне 2D тверде тіло призначене для переміщення в умовах моделювання, але тільки під дуже явним контролем користувача. У той час як на Dynamic Rigidbody 2D діють гравітація та сили, на Kinematic Rigidbody 2D їх немає. Тому він швидкий і вимагає менше системних ресурсів, ніж Dynamic Rigidbody 2D.

Кінематична Rigidbody 2D призначена для явної зміни положення за допомогою Rigidbody2D.MovePosition або Rigidbody2D.MoveRotation

У деяких випадках вам може знадобитися, щоб GameObject мав Rigidbody без контролю його руху фізичним механізмом. Наприклад, ви можете керувати своїм персонажем безпосередньо з коду сценарію, але дозволити йому виявлятися за тригерами. Такий нефізичний рух, створений за допомогою сценарію, відомий як кінематичний рух.

Static Rigidbody 2D спроектований таким чином, щоб взагалі не рухатися під час моделювання; якщо з ним щось стикається, статичне тверде тіло 2D поводить себе як нерухомий об'єкт (якби він мав нескінченну масу). Статичне тіло стикається тільки з динамічним твердим тілом 2D. Зіткнення двох Static Rigidbody 2D не підтримується, оскільки вони не призначені для переміщення.

Коли тверде тіло переміщається зі швидкістю, меншою за певний мінімальний поріг, фізичний двигун припускає, що воно зупинилося і перебуває в спокої. При цьому об'єкт не

буде знову рухатися до тих пір, поки з ним не відбудеться зіткнення або до нього не застосують силу, так що він йде в "сплячий" режим.

Ця оптимізація означає, що на об'єкт не будуть витрачатися ресурси CPU, поки його знову не розбудять (тобто коли знову не приведуть в рух).

Розглянемо деякі властивості компонента Rigidbody 2D:

- Use Auto Mass – встановить прапорець, якщо ви хочете, щоб Rigidbody 2D автоматично визначав масу GameObject за його 2D-колайдером.
- Mass – задайте масу Rigidbody 2D. Це недоступно, якщо ви вибрали «Використовувати автомасу».
- Collision Detection – визначає, як виявляються зіткнення між Collider 2D.
 - Discrete – ігрові об'єкти з 2D-об'єктами Rigidbody та 2D-колайдерами можуть перекриватися або проходити крізь один одного під час оновлення фізики, якщо вони рухаються досить швидко. Контакти зіткнення генеруються лише у новій позиції.
 - Continuous – ігрові об'єкти з 2D-об'єктами Rigidbody та 2D-колайдерами не проходять один через одного під час оновлення. Натомість Unity обчислює першу точку зіткнення будь-якого з двовірних об'єктів Collider і переміщає туди GameObject. Це потребує більше процесорного часу, ніж Discrete.
- Freeze Position – вибірково зупиняє переміщення Rigidbody 2D світовими осями X та Y.
- Freeze Rotation – вибірково зупиняє обертання Rigidbody 2D навколо осі Z.

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

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

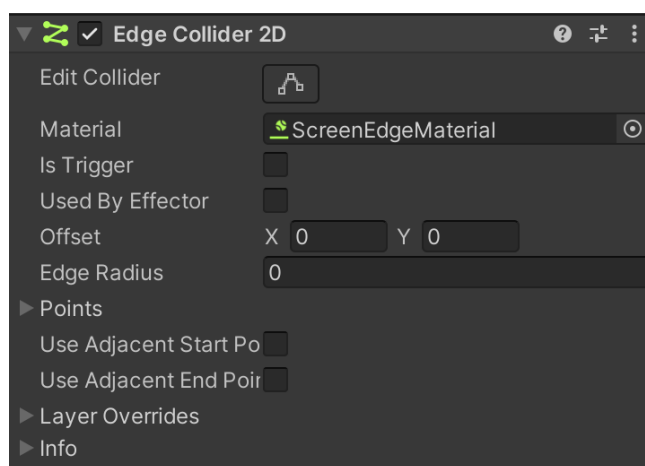


Рисунок 2.4 – Приклад компоненти Edge Collider 2D

Складені колайдери наближають форму GameObject, зберігаючи при цьому низькі витрати процесора. Щоб отримати додаткову гнучкість, ви можете додати додаткові колайдери до дочірніх GameObjects. Коли ви створюєте такий складний колайдер, вам слід використовувати лише один компонент Rigidbody, розміщений у кореновому GameObject в ієрархії.

Коли колайдери взаємодіють, їхні поверхні потрібно імітувати властивості матеріалу, з якого вони теоретично повинні складатися. Наприклад, шар льоду буде більш слизьким, у той час як резиновий м'яч буде пропонувати більше тренувань і буде дуже пружним. Хоча

форма колайдерів і не деформується під час зіткнення, їх тренування і пружність можна налаштувати за допомогою фізичних матеріалів (Physics Materials).

У разі зіткнення система скриптингу може виявити і виконати дії, зазначені у функції OnCollisionEnter. Однак ви також можете використовувати фізичний двигун просто для виявлення того, що один колайдер входить у простір іншого, без створення колізії.

Колайдер, налаштований як тригер (за допомогою властивості Is Trigger), не веде себе як твердий об'єкт, а просто пропускає інші колайдери крізь себе. Коли інший колайдер увійде «на територію» цього колайдера, тригер викличе функцію OnTriggerEnter у скриптах об'єкта, до якого приєднано тригер.

2. Фізичний матеріал

Коли ви граєте у відеогру, ви не просто взаємодієте з пікселями на екрані; ви входите в динамічний світ, керований правилами та законами, які імітують закони фізичного всесвіту. Від граційної дуги стрибка персонажа до вибухового зіткнення об'єктів, кожен рух і взаємодія в грі організовані за кулісним хореографом, відомим як фізика.

Фізичні рушії симулюють рух об'єктів у відповідь на сили, такі як гравітація, тертя та введення користувача. Це дозволяє розробникам створювати реалістичні анімації та правдоподібні взаємодії між елементами гри.

Фізичні рушії виявляють, коли об'єкти зіштовхуються один з одним, і обчислюють результуючі сили та реакції. Це дозволяє створювати динамічні середовища, де об'єкти можуть відскакувати, котитися або розбиватися при ударі.

Фізика в іграх не обмежується лише взаємодією об'єктів один з одним; вона також охоплює ефекти навколишнього середовища, такі як вітер, вода та вогонь. Ці елементи можуть впливати на ігровий процес і додавати глибини у світ гри.

Фізика в Unity – це вбудована система симуляції реальних фізичних законів у віртуальному ігровому світі. Вона дозволяє об'єктам взаємодіяти один з одним, підкорятися гравітації, стикатися та рухатися так, як це відбувалося б у реальному житті, що робить гру більш захоплюючою та правдоподібною.

Основні компоненти фізики в Unity:

- Rigidbody (тверде тіло) – компонент, який змушує об'єкт підкорятися фізичному рушію. З ним об'єкт набуває маси, гравітації та може реагувати на сили.
- Colliders (колайдери/Зіткнення) – форми (куб, сфера, капсула), які визначають фізичні межі об'єкта для обробки зіткнень.
- Forces (сили) – дозволяють додавати об'єктам прискорення, інерцію, вибухові хвилі тощо.
- Joints (з'єднання) – компоненти для з'єднання об'єктів між собою (наприклад, шарніри, мотузки).

Що робить фізика в Unity:

- Симулює гравітацію – об'єкти падають вниз, якщо на них діє Rigidbody;
- Обробляє колізії (зіткнення) – визначає, коли два об'єкти торкаються, щоб зупинити рух, спричинити відскок або руйнування;
- Імітує матеріали – можна налаштувати тертя або пружність поверхні (наприклад, лід буде ковзати, а гума — стрибати);
- Працює в 2D та 3D – у Unity є окремі фізичні рушії для 2D (Rigidbody2D, Collider2D) та 3D, що дозволяє працювати з різними типами ігор.

Physics Material 2D в Unity – це асет, що налаштовує тертя (friction) та пружність (bounciness) при зіткненні 2D-об'єктів. Створюється через Assets → Create → 2D → Physics Material 2D і перетягується на компонент колайдера (наприклад, BoxCollider2D). Значення варіюються від 0 до 1, де 0 – відсутність тертя (лід), а 1 – максимальне відскакування.

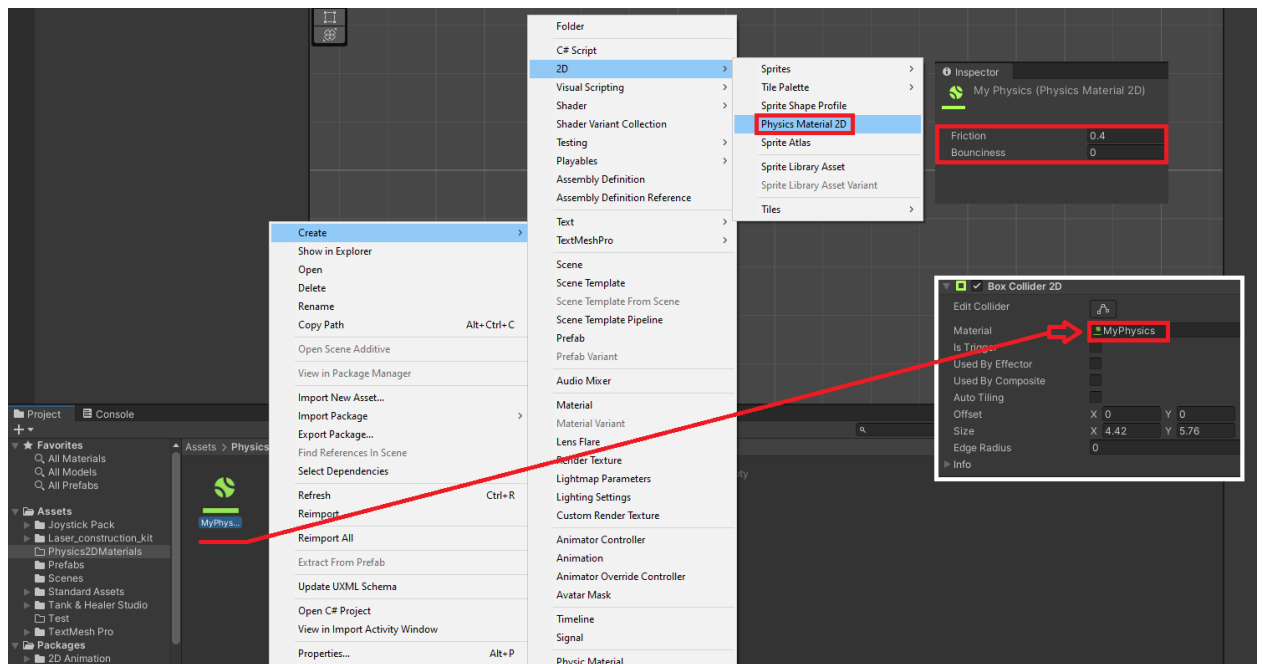


Рисунок 2.5 – Приклад створення фізичного матеріалу

Основні параметри Physics Material 2D:

- Friction (тертя) – визначає, наскільки легко об'єкт ковзає по поверхні;
- Bounciness (пружність) – визначає, наскільки високо об'єкт підстрибує під час удару.

Контрольні питання

1. Що таке GameObject?
2. Для чого використовується компоненти?
3. Що робить компонент Rigidbody?
4. Що робить компонент Collider?
5. Чим відрізняється динамічне тіло (Dynamic) від кінематичного (Kinematic)?
6. Для чого використовується матеріал?
7. Які основні параметри має Physics Material 2D?

Лекція 3. 2D гра. Анімація спрайтів

1. Спрайти
2. Анімація

1. Спрайти

Сцени Unity складаються з ігрових об'єктів. Самі по собі ці об'єкти невидимі і мають нічого, крім імені. Їхня поведінка визначається компонентами.

Компоненти – це будівельні блоки гри, і все, що ви бачите в інспекторі, це і є компоненти. Різні компоненти відповідають різним аспектам; наприклад, візуалізатори мішів (Mesh Renderers) відображають тривимірні міші (сітки), а джерела звуку (Audio Sources) відтворюють звуки. Сценарії (скрипти) також є компонентами.

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.

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.

Спрайти мають прямокутну форму, однак у комп'ютерних іграх часто трапляються непрямокутні об'єкти. Це досягається за рахунок задання прозорих областей при малюванні спрайт.

Unity підтримує безліч різних форматів, таких як PNG, JPEG тощо, для наших графічних зображень.

Є кілька способів додати спрайт до свого проєкту. Один із способів – просто скопіювати PNG у папку зі спрайтами за допомогою операційної системи, і тоді він стане частиною проєкту. Інший спосіб зробити це – перетягнути спрайт-файл з File Explorer в проєкт. Можна створити спрайт засобами Unity (GameObject → 2D Object → Sprite).

Щоб перетворити спрайт на ігровий об'єкт, необхідно перетягнути спрайт у вікно ієрархії, і тепер він доданий до сцени.

Якщо вибрати ігровий об'єкт у вигляді спрайту на сцені, можна побачити компоненти, які за замовчуванням прикріплені до нових ігрових об'єктів, доданих до сцени. Ми маємо засіб візуалізації спрайтів *Sprite Renderer*, що дозволяє візуалізувати або малювати спрайти, пов'язані з цим ігровим об'єктом. Якщо компонент Sprite Renderer не додався, Ви можете додати компонент до існуючого ігрового об'єкта через Компоненти (меню Component → Rendering → Sprite Renderer).

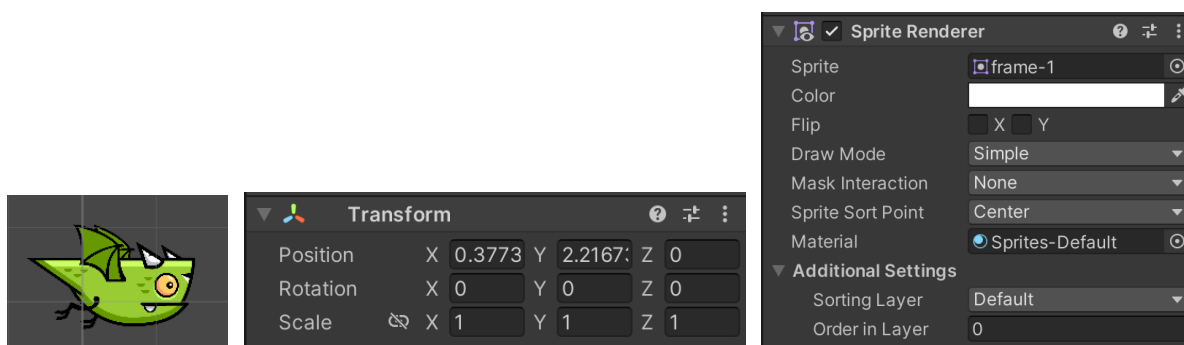


Рисунок 3.1 – Приклад спрайту на сцені та його компоненти Transform та Sprite Renderer

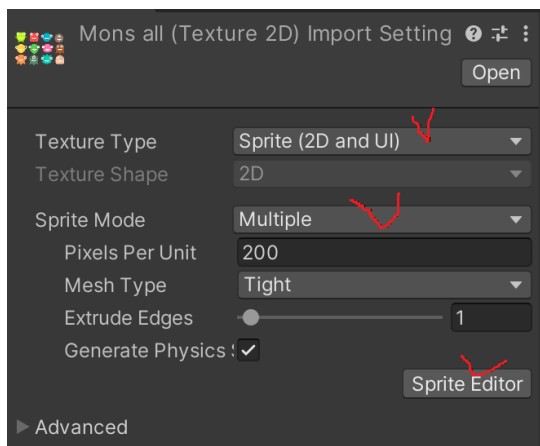
Розглянемо властивості Sprite Renderer.

Властивість	Функція
Sprite	Спрайт, який треба рендерувати. Спрайти можна створити з текстур, використовуючи налаштування Sprite в імпортері текстур.
Color	Колір меша, що рендериться.
Flip	Перевертає спрайт у площині X або Y.
Material	Матеріал, що використовується для рендеру спрайту.
Draw Mode	Визначає, як масштабується спрайт при зміні розмірів. Виберіть один з наступних варіантів у списку, що розкривається. Simple – всі зображення масштабуються при зміні розмірів. Це опція за замовчуванням Sliced – виберіть цей режим, якщо Sprite має 9-sliced (включає поділ зображення на дев'ять частин, щоб при зміні розміру Sprite різні частини масштабувалися різними способами, щоб спрайт залишався пропорційним). Size ('Sliced' or 'Tiled') – введіть нову ширину та висоту спрайту, щоб правильно масштабувати спрайт, що складається з 9 фрагментів. Tiled – 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. Continuous – це режим мозаїки за промовчанням. У безперервному режимі середня частина вирівнюється при зміні розмірів спрайту. Adaptive – в адаптивному режимі текстура Sprite розтягується при зміні її розмірів, як і у простому режимі. Коли масштаб змінених розмірів збігатиметься зі значенням розтягування, середня частина почне викладатися мозаїкою. Stretch Value – 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: None – засіб візуалізації спрайтів не взаємодіє з жодними масками спрайтів у сцені. Це опція за замовчуванням. Visible Inside Mask – спрайт видно там, де на нього накладається маска спрайту, але не за її межами. Visible Outside Mask – 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
Sorting Layer	Шар, що використовується для завдання пріоритету накладання під час рендерингу.
Order In Layer	Пріоритет накладання спрайту у межах його шару. Чим нижче число, тим раніше рендеруватиметься спрайт, а спрайти з числами вище будуть малюватись поверх тих, що нижче.

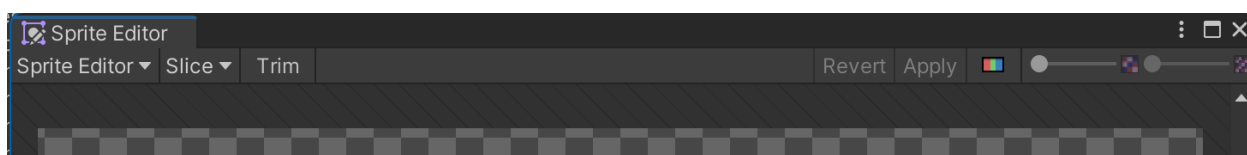
Для зміни розміру виділяємо в папці проєкту файл спрайту, в Inspector змінимо значення в полі Pixels per unit. Зменшення числа – збільшує спрайт та навпаки. Крім того, є інструмент Transform. Компонент Transform вказує положення ігрового об'єкта в сцені, його поворот навколо осей x, y і z, а також масштаб ігрового об'єкта.

У двовірних іграх повсюдно поширені анімовані спрайти. Вони створюються малюванням кожного кадру анімації та подальшим відтворенням отриманої послідовності кадрів у Unity. Кадри можна імпортувати у вигляді окремих зображень, але їх зазвичай викладають у вигляді однієї картинки, яка називається *листом спрайтів* (sprite sheet).

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 () 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.

Аркуш спрайтів з'являється на вкладці Project як один ресурс, але клацання на розташованій збоку стрілці розкриває його і дає можливість побачити окремі спрайти.



2. Анімація

Анімація персонажу може зберігатися в наступних видах:

- Листи спрайтів;
- Кожний кадр спрайту в окремому файлі.



Рисунок 3.2 – Приклад листу спрайтів

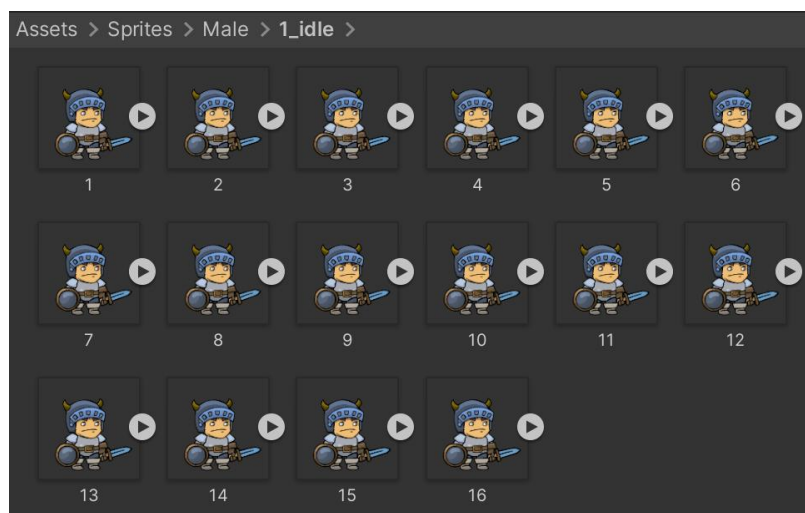


Рисунок 3.3 – Приклад анімації в окремих файлах

Листи спрайтів можна автоматично генерувати в Unity. При імпорті аркуша у списку Sprite Mode налаштувань спрайту слід вибрати варіант Multiple.

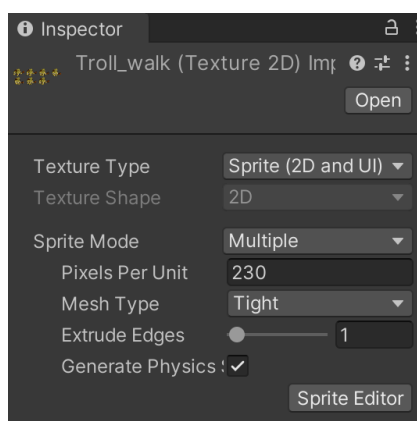


Рисунок 3.4 – Налаштування листу спрайтів в Inspector

Редактор Sprite Editor дозволяє витягувати елементи складеного зображення. Щоб викликати редактор Sprite Editor, виділяємо в Project спрайт, який складається з кадрів руху персонажа, та в Inspector натискаємо на кнопку Sprite Editor.

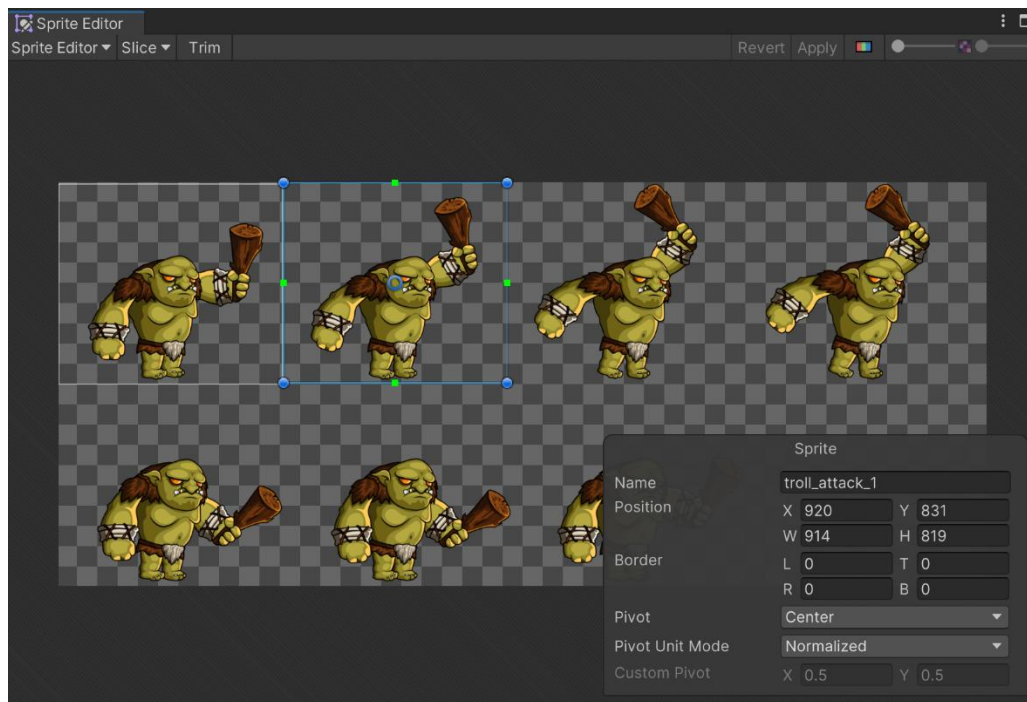


Рисунок 3.5 – Вікно редактору Sprite Editor

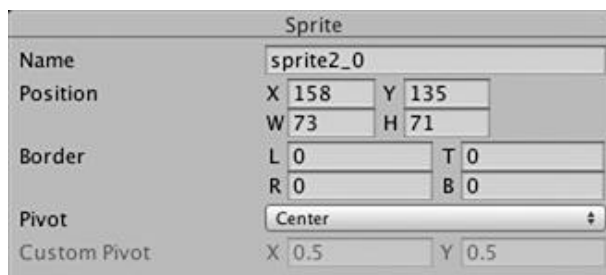
Крім складеного зображення, ви побачите різні елементи керування у заголовку вікна редактора.

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:



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.

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).

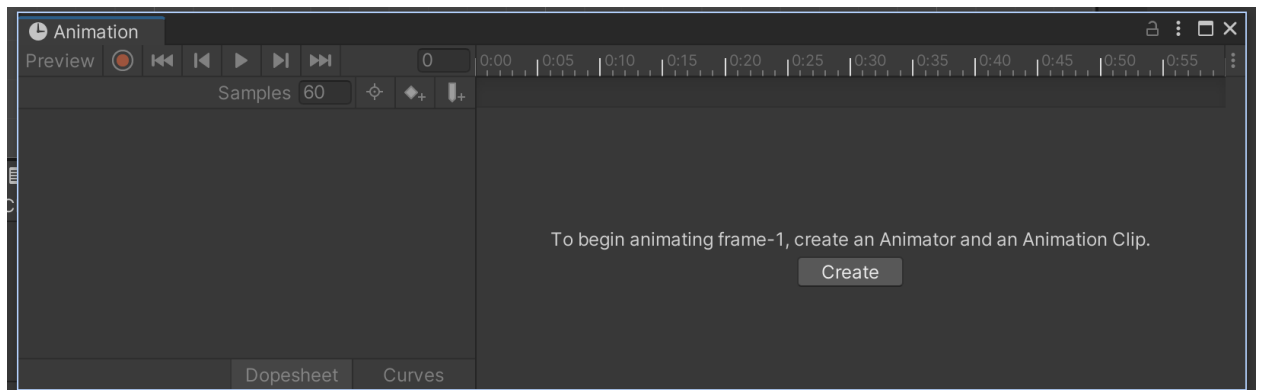


Рисунок 3.5 – Вікно Animation

Вікно *Animation* дозволяє створити та модифікувати кліпи анімації (Animation Clips) прямо всередині Unity. У вікні Animation знаходиться вікно *Timeline* (шкали часу), що дозволяє створювати кінематографічний вміст, ігрові послідовності, аудіоряди та складні ефекти частинок. Ви можете анімувати багато різних ігрових об'єктів в одній послідовності.

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 (синім ромбом).

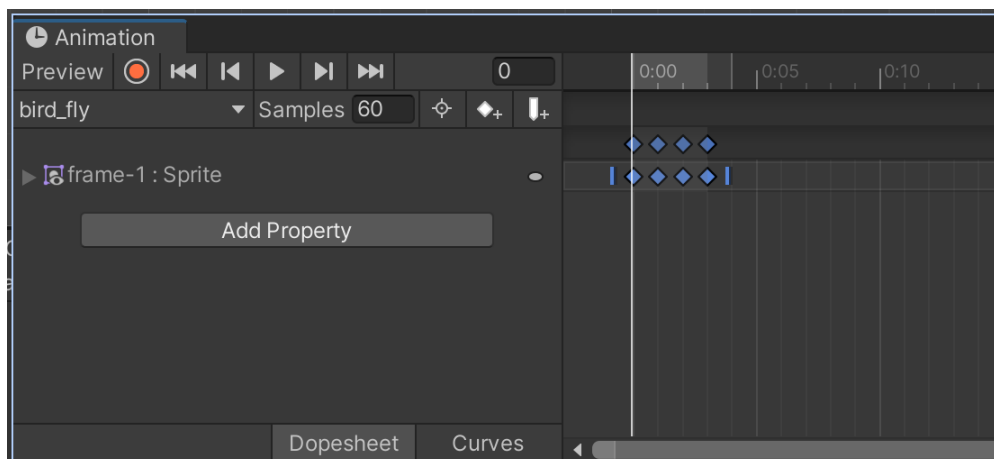


Рисунок 3.6 – Результат додавання кадрів анімації

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 її. Методом підбору знаходимо необхідну частоту кадрів, за якої анімація виглядає природною.

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.

При створенні анімації створилися файли контролери.

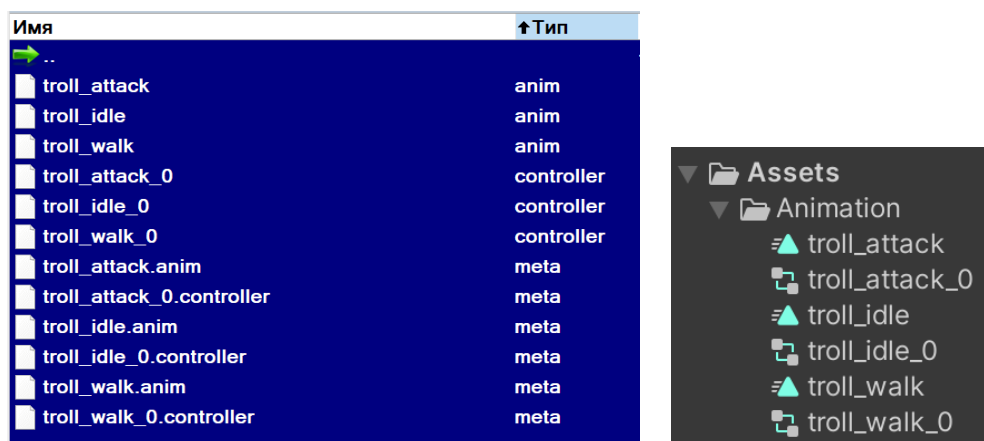


Рисунок 3.7 – Файли анімації та файли контролери

Залишимо один головний, а інші видалімо. Нехай головним буде troll_idle_0.controller. Перейменуємо його на troll.controller. Відкриємо його в редакторі анімації (подвійний клік миші на troll.controller).

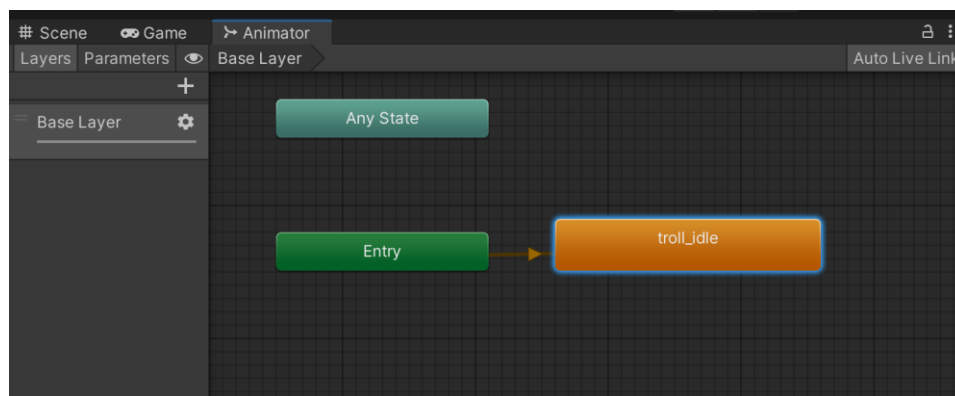


Рисунок 3.8 – Вікно Animator

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. Ви можете думати про поточний стан як про маркер або виділення, поміщене на один з вузлів, яке може перестрибнути на інший вузол по одній з цих стрілок.

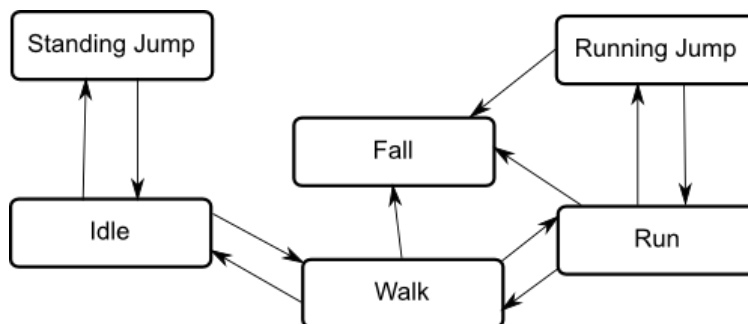


Рисунок 3.9 – Приклад станів та переходів

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);
- Тригер — логічний параметр, який скидається контролером під час переходу (позначається круглою кнопкою).

Параметрам можна надавати значення зі скрипта 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.

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

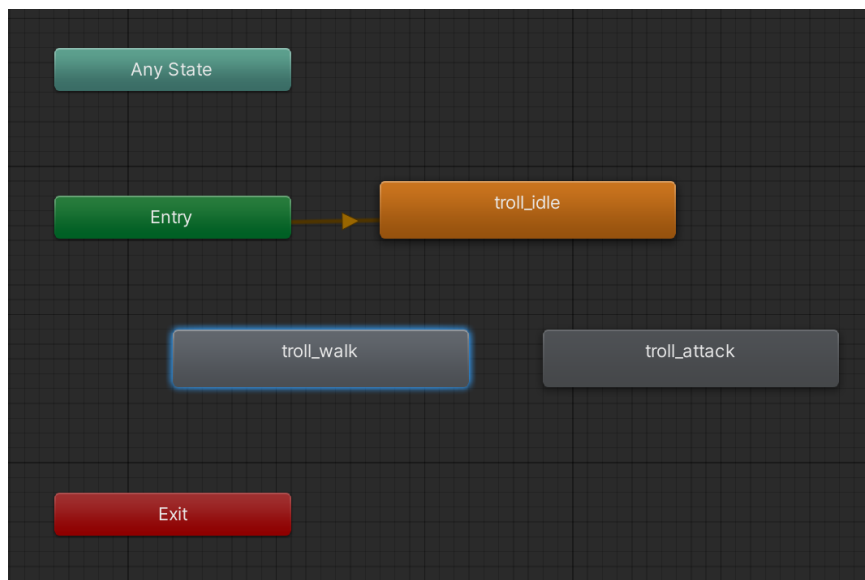
Вузол Entry використовується при переході в кінцевий автомат. Вузол входу буде оцінено та розгалужено до стану призначення відповідно до встановлених умов. Таким

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.

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.

Перетягнемо до вікна Animator анімації персонажа, що маємо.



Додаймо змінні, які допоможуть перейти з одного стану в інший.

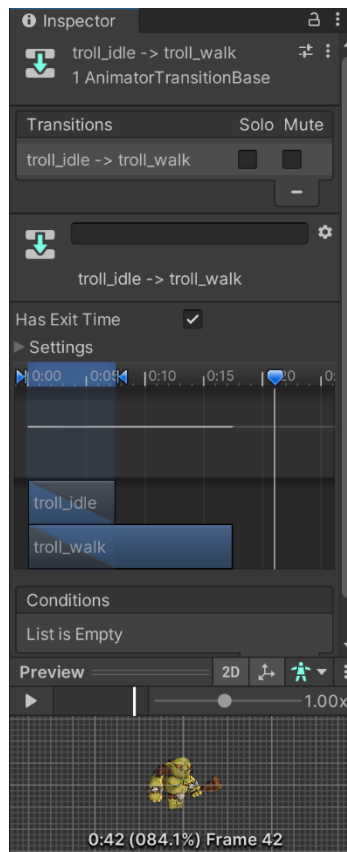
- Float speed;
- Trigger attack;
- Bool jump;

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.

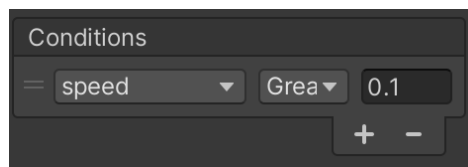
Перехід може мати одну умову, кілька умов або взагалі не мати умов. Якщо у вашого переходу немає умов, редактор Unity враховує лише час виходу, і перехід відбувається після досягнення часу виходу. Якщо ваш перехід має одну або кілька умов, всі умови повинні бути дотримані, перш ніж перехід буде активований.

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. 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. Встановіть цей прапорець, щоб перехід відбувався у певний час, вказаний у Exit Time.

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

Якщо об'єкт має анімацію, то автоматично додається компонент Animator.

Компонент *Animator* використовується для призначення анімації об'єкта гри у вашій сцені. Компонент *Animator* вимагає посилання на контролер аніматора, який визначає, які анімаційні кліпи використовувати, і керує, коли і як змішувати та переходити між ними.

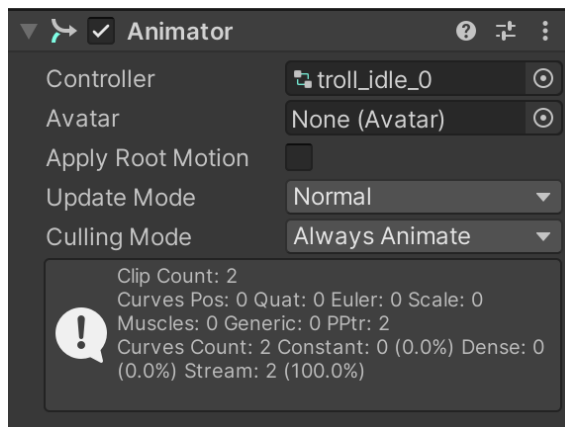


Рисунок 3.10 – Приклад компоненту Animator

У вікні Animator можна налаштувати швидкість програвання анімації певного стану (Animation State) у вашому контролері:

1. Відкрийте вікно Animator (Window → Animation → Animator).
2. Виберіть об'єкт, який має анімацію.
3. Натисніть на стан (state) анімації, швидкість якої потрібно змінити.
4. В інспекторі (Inspector) знайдіть параметр Speed та змініть значення:
 - 1 — стандартна швидкість.
 - 0.5 — у два рази повільніше.
 - 2 — у два рази швидше.
 - -1 — анімація буде програватися назад.

Контрольні питання

1. Що таке спрайти та для чого вона призначена?
2. Які бувають спрайти?
3. Що робить компонент Sprite Renderer?
4. Що дозволяє робити редактор Sprite Editor?
5. Що робить компонент Animator?
6. Що робить компонент Animation?
7. Як створити анімацію?
8. Що таке анімовані спрайти?
9. Що можна зробити за допомогою редактора анімації?

Лекція 4. 2D гра. Рух персонажа

1. Основи руху персонажа
2. Переміщення персонажа
3. Рух по сходам

1. Основи руху персонажа

Unity-двигжок є одним з найпопулярніших інструментів для створення відеоігор. Зокрема, в Unity 2D ви можете створювати ігри з унікальним сюжетом, цікавими персонажами та захоплюючими рівнями. Код руху персонажа – важлива частина ігрової механіки, і вміння його реалізовувати дозволить вам створювати реалістичну поведінку персонажів.

Спочатку необхідно визначити, яким чином буде управлятися персонаж. Це може бути клавіатура, миша, тач-скрін або контролер. У Unity 2D вбудовані засоби дозволяють легко обробляти введення та отримувати інформацію про поточний стан пристроїв введення.

Далі необхідно визначити, як персонаж буде рухатися у відповідь на введення. В основному, переміщення персонажа в Unity 2D здійснюється шляхом зміни його позиції. Це може бути зроблено за допомогою компонента Rigidbody2D або шляхом прямої зміни координат об'єкта.

Ще одним важливим аспектом руху персонажа в Unity 2D є контроль зіткнень. Персонаж може переміщатися по сцені і взаємодіяти з іншими об'єктами, тому необхідно передбачити перевірку зіткнень і можливість управління поведінкою персонажа у разі зіткнення.

Крім того, важливо враховувати фізичні властивості персонажа, такі як гравітація та тертя. Unity 2D надає механізми для роботи з фізичними ефектами, які можуть бути застосовані до персонажа і впливати на його рух.

В цілому, рух персонажа в Unity 2D ґрунтується на обробці введення, зміні позиції об'єкта, контролі зіткнень і обліку фізичних ефектів. Знання основних принципів руху допоможе вам створити реалістичну та керовану поведінку персонажів у ваших іграх.

2. Переміщення персонажа

Для початку необхідно задати швидкість переміщення персонажа. Швидкість може бути представлена числом, яке визначає, наскільки швидко персонаж буде рухатися. Потім необхідно отримати введення гравця для переміщення персонажа. Всі ці дії ми будемо обробляти в скрипті, який створимо та прикріпимо до 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.

Input Manager в Unity відповідає за налаштування відповідності між натисками кнопок та різними діями в грі. Він дозволяє використовувати різні контролери та пристрої, що робить управління грою більш гнучким та адаптивним. Для доступу до менеджера введення перейдіть до головного меню Unity, потім виберіть Edit → Project Settings → Input Manager.

У Input Manager ви визначаєте всі різні осі вхідних даних та ігрові дії для вашого проєкту. Осі (Axes) мають властивості. Розглянемо деякі з них.

- Name – назва осі у вікні лаунчера гри та скриптах;
- Negative Button – кнопка, яка відображатиме негативні значення на осі;
- Positive Button – кнопка, яка відображатиме позитивні значення на осі;

- Alt Negative Button – додаткова кнопка, яка відображатиме негативні значення на осі;
- Alt Positive Button – додаткова кнопка, яка відображатиме позитивні значення на осі;
- Sensitivity – у разі введення з клавіатури більше значення (на осі) відповідатиме швидшій реакції. Найменше значення відповідатиме більш згладженому. У разі миші значення буде збільшувати фактичне зміщення миші.

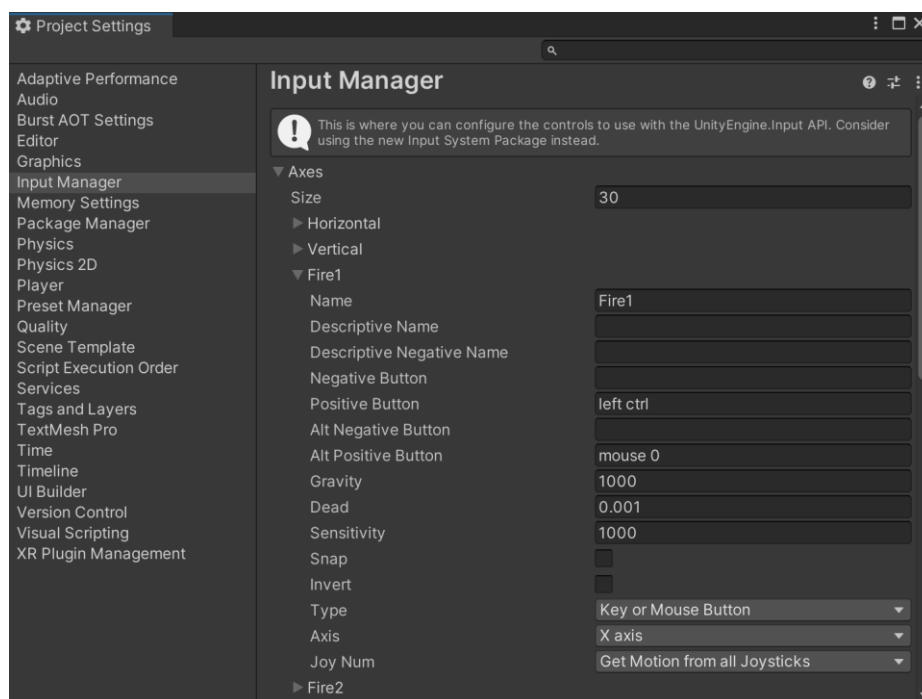


Рисунок 4.1 – Менеджер введення (Input Manager)

Усі осі, які ви налаштували в Input Manager, служать двом цілям:

- При скриптуванні це дозволить вам зчитувати значення з осі за допомогою її імені;
- Це дозволить гравцям вашої гри налаштувати управління на власний смак.

Вікно введення лаунчера гри відображається під час запуску гри:

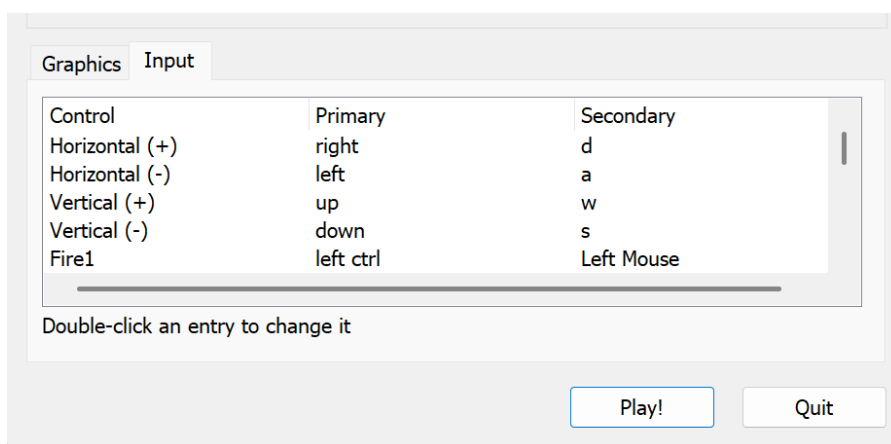


Рисунок 4.2 – Приклад вікна введення лаунчера гри

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.

У Unity це можна зробити за допомогою функції `Input.GetAxis`, яка дозволяє отримати значення осі введення. Наприклад, нам потрібно отримати значення осі x для переміщення вліво і вправо:

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

Отримане значення буде в діапазоні від -1 до 1, де -1 відповідає руху вліво, 1 – руху вправо, а 0 – відсутності руху.

Необхідно додати компонент `Rigidbody2D` та колайдери до персонажа. Компонент `Rigidbody2D` відповідає за фізичну поведінку об'єкта – гравітацію, колізії і рух під впливом сили.

Тепер, щоб перемістити персонажа вліво і вправо, ми можемо використовувати компонент `Rigidbody2D` і його властивість `velocity`

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

Властивість `velocity` компонента `Rigidbody` в Unity використовується для управління швидкістю об'єкта. Це вектор, що визначає напрямок і величину швидкості руху `Rigidbody` в світових координатах.

Використання `velocity` дозволяє напряму змінювати фізичний рух об'єкта, не використовуючи сили або інші параметри. Це особливо корисно для створення контрольованого руху, такого як польоти, плавання або простий платформер.

Під час зміни властивості `velocity` важливо враховувати, що це безпосередньо змінює розраховані значення фізичного руху. Таким чином, воно не враховує зіткнення, опір повітря або інші фізичні сили, що впливають на об'єкт.

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;
```

Щоб при пересуванні персонажа програвалася відповідна анімація, скористаємося параметрами, які створили в `Animator`, а саме:

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

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);
```

Оголосимо змінну типу `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 знаку:

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

Якщо запустити гру, то персонаж повинен ходити вліво-вправо, при цьому залишаючись обличчям у 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 для 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");
}

```

Щоб реалізувати стрибок, необхідно встановити значення для сили стрибка в скрипті управління персонажем. Це можна зробити за допомогою змінної:

```

[SerializeField]
private float jumpForce = 100.0f;

```

Щоб персонаж стрибав, до нього треба прикладати силу за допомогою виклику AddForce. Коли ми викликаємо метод AddForce(), ми передаємо вектор сили, що вказує напрямок і силу стрибка. При цьому потрібно визначати, чи перебуває персонаж на землі.

У Unity для компонента Rigidbody2D при використанні методу AddForce перерахування ForceMode2D має два основні значення, які визначають, як саме сила впливає на об'єкт:

1. ForceMode2D.Force (Сила)
2. ForceMode2D.Impulse (Імпульс)

ForceMode2D.Force додає постійну силу до об'єкта з огляду на його масу. Використовується для безперервної дії, наприклад, гравітації, вітру або постійного прискорення. Сила застосовується у кожному кадрі фізики (на секунду), тому її значення залежить від часу.

Приклад:

```

rb.AddForce(transform.up * speed, ForceMode2D.Force);

```

ForceMode2D.Impulse додає миттєвий імпульс сили до об'єкта, враховуючи його масу. Використовується для одноразових, різких впливів, наприклад, стрибка персонажа, вибуху або удару м'яча. Діє миттєво за один виклик.

Приклад:

```
rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
```

Створимо булеву змінну onGround і надамо їй значення true.

```
private bool onGround = true;
```

Щоб знати, чи є під персонажем земля, ми перевіримо, чи є відповідний колайдер на лінії між двома точками. Однією буде центр персонажа. Щоб встановити іншу – створимо порожній GameObject, назвемо його GroundCheck, зробимо його дочірнім об'єктом персонажа Knight. Розмістимо цей об'єкт трохи нижче ніж персонажа.



Створимо змінну groundCheck типу Transform і надаймо 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;
}
```

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);
    }
}
```

Тепер метод Update() має наступний вигляд:

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

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

3. Рух по сходам

В Unity 2D колізії і зіткнення обробляються за допомогою компонента Collider2D, який додається до ігрових об'єктів. Collider2D визначає межі об'єкта та його форму, які використовуються для обчислення зіткнень.

Коли два колайдери перетинаються, Unity 2D генерує подію зіткнення, яку можна обробити в коді для визначення подальших дій. Наприклад, коли персонаж стикається з перешкодою, можна зупинити його рух або змінити його напрямок.

При роботі зі зіткненнями важливо враховувати розмір і форму колайдерів, щоб об'єкти правильно взаємодіяли один з одним. Також варто пам'ятати про продуктивність, так як перевірка зіткнень може бути ресурсом-ємною операцією, особливо при великій кількості об'єктів.

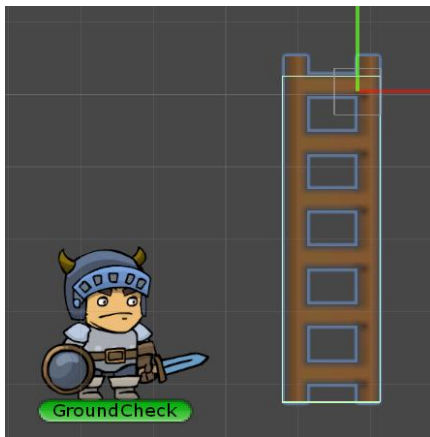
Для обробки зіткнень в Unity 2D використовується фізичний движок. В основі фізичного движка лежить система колайдерів і риджідбоді. Колайдери визначають межі об'єктів, а риджідбоді відповідає за їхню фізичну поведінку.

Після того, як колайдер доданий і налаштований, необхідно обробити зіткнення за допомогою скрипта. Для цього можна використовувати метод OnCollisionEnter2D() або OnTriggerEnter2D(). Перший метод викликається при зіткненні з іншим об'єктом, що має колайдер і риджідбоді, в той час як другий метод викликається при зіткненні з об'єктом, що має тільки колайдер. У середині методу можна вказати необхідні дії, такі як зміна швидкості персонажа, відтворення звуку або активація анімації.

Реакція на зіткнення – важливий момент при розробці ігор в Unity 2D. Вона дозволяє створити реалістичну і інтерактивну поведінку персонажа і об'єктів в ігровому світі.

Спочатку створимо сходи на сцені.

Якщо необхідно отримати сходи потрібної довжини, то створимо новий GameObject, назовемо його Stair, і 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;
    }
}

```

Контрольні питання

1. Для чого використовується Input Manager?
2. Що таке вісь в Input Manager?
3. Які властивості має кожна вісь?
4. Які компоненти потрібно додати до персонажу, щоб він міг керуватися клавішами?
5. Для чого використовуються методи `OnTriggerEnter2D` та `OnTriggerExit2D`?
6. З чого складається скрипт для управління персонажем?

Лекція 5. 2D гра. Вороги

1. Перший тип ворога
2. Другий тип ворога
3. Третій тип ворога

1. Перший тип ворога

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).

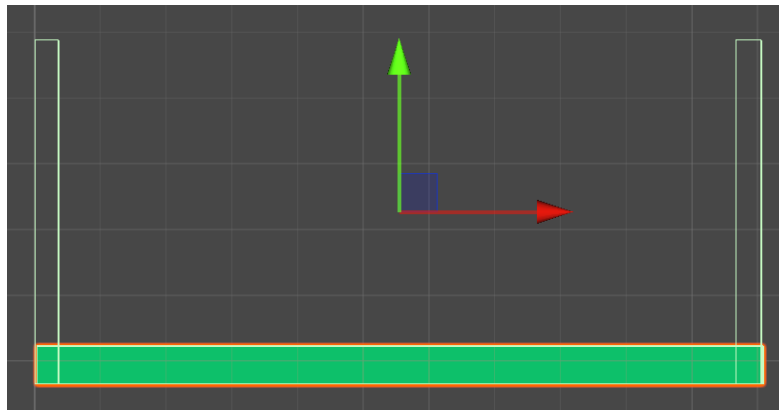
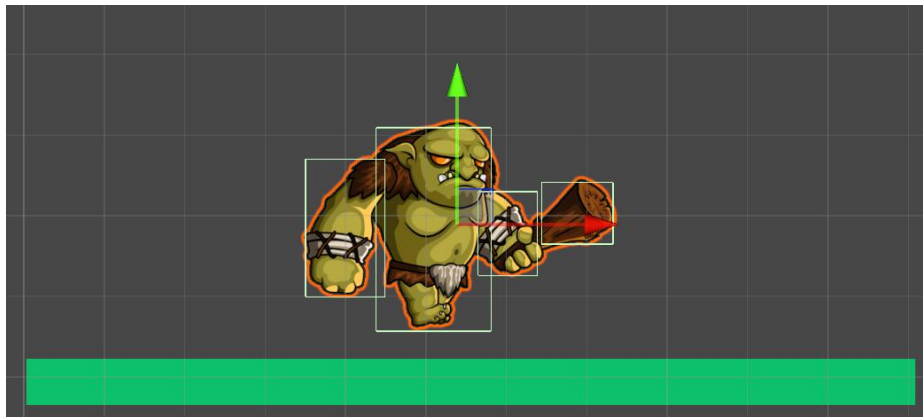


Рисунок 5.1 – Приклад платформи з колайдерами

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 Елементу, 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()
{
}

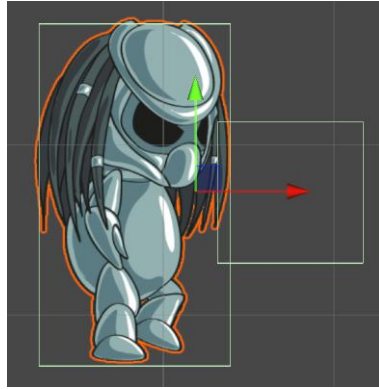
```

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

2. Другий тип ворога

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

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



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

```
[SerializeField]
private bool onToHero = false;
```

```
[SerializeField]
private Transform target;
```

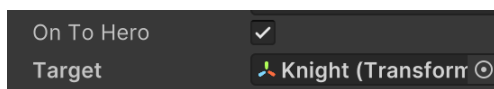
Відредагуємо методи Update та OnTriggerEnter2D:

```
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:



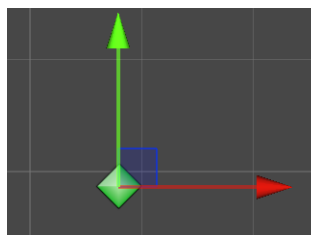
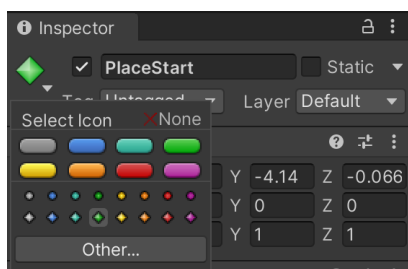
Тепер ворог ходить за персонажем.

3. Третій тип ворога

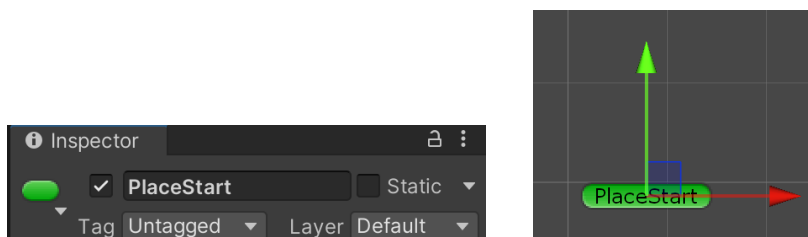
Розглянемо третій тип ворога, який при старті гри не знаходиться на сцені, але буде з'являтися в певній точці сцени через певний інтервал часу.

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 Мітка (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();

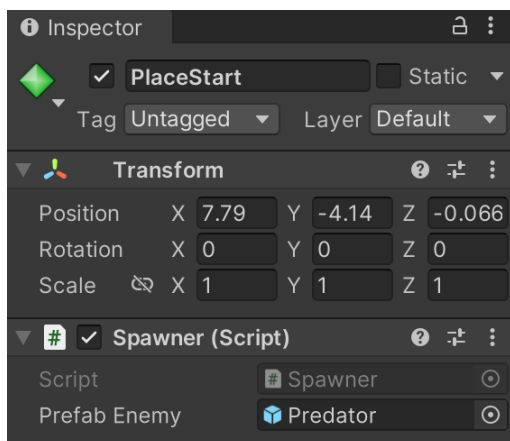
        spawnTimer.Duration = Random.Range(MinSpawnDelay, MaxSpawnDelay);
        spawnTimer.Run();
    }
}

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:



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

Контрольні питання

1. У чому відмінність роботи з ворогами?
2. Які типи ворогів ви знаєте?
3. Як зробити, щоб ворог ходив по деякому маршруту?
4. Що потрібно прописати в скрипті, щоб ворог йшов за персонажем?
5. Що потрібно прописати в скрипті, щоб ворог з'являвся через певний час з вказаного місця?

Лекція 6. 2D гра. Оточення. Налаштування камери. Паралакс

1. Додавання оточення
2. Створення підвісного місту
3. Система частинок
4. Пульсуюча анімація предмету
5. Камера, що рухається за персонажем
6. Створення фону, що рухається

1. Додавання оточення

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

Додаємо на сцену елементи сцени. Наприклад:

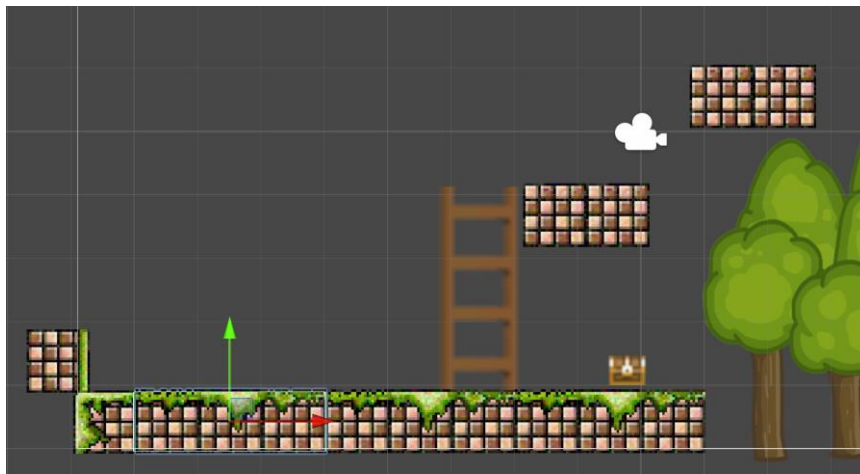


Рисунок 6.1 – Приклад сцени

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



2. Створення підвісного місту

Додаймо до сцени підвісний міст.

Створимо на сцені пустий GameObject та дамо йому назву Bridge. Помістимо всередину нього спрайт ділянки мосту.



До спрайту ділянки мосту додати Rigidbody2D та BoxCollider2D.

2D джоінти 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 джоінтів 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.

Існують різні типи Joint 2D:

Distance Joint 2D – з'єднує два ігрові об'єкти, керовані фізикою твердого тіла, і утримує їх на певній відстані один від одного.

Fixed Joint 2D – утримує два об'єкти в положенні один одного, тому об'єкти завжди зміщені в заданому положенні і під заданим кутом. Наприклад, об'єкти, які мають реагувати так, ніби вони жорстко пов'язані: вони не можуть віддалятися один від одного, не можуть зближуватися і не можуть обертатися один до одного. Ви також можете використовувати це з'єднання, щоб створити менш жорстке з'єднання, яке згинається.

Friction Joint 2D – зменшує як лінійну, так і кутову швидкість між двома ігровими об'єктами, керованими фізикою твердого тіла, до нуля (тобто уповільнює та зупиняє їх). Наприклад; платформа, яка обертається, але чинить опір цьому руху.

Hinge Joint 2D – дозволяє прикріпити ігровий об'єкт, керований фізикою твердого тіла, до точки у просторі, навколо якої він може обертатися. Наприклад; вісь на ножицях.

Relative Joint 2D – дозволяє двом ігровим об'єктам, керованим фізикою твердого тіла, зберігати позицію залежно від розташування один одного. Використовуйте це з'єднання, щоб перемістити два об'єкти відносно один одного. Наприклад, космічний шутер, в якому гравець має додаткові батареї, які йдуть за ним.

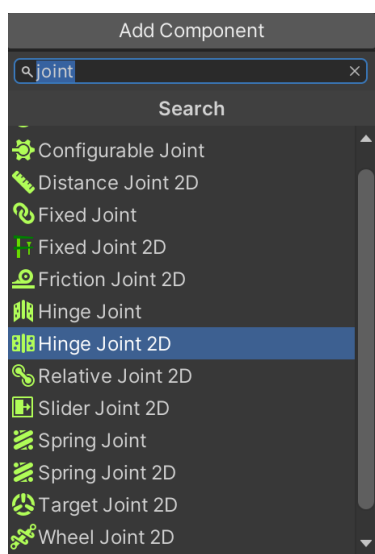
Slider Joint 2D – дозволяє ігровому об'єкту, керованому фізикою твердого тіла, ковзати вздовж лінії у просторі, наприклад, як розсувні двері.

Spring Joint 2D – дозволяє з'єднувати два ігрові об'єкти, керовані фізикою твердого тіла, начебто за допомогою пружини.

Target Joint 2D – з'єднується із зазначеною метою, а не з іншим твердим об'єктом, як це роблять інші з'єднання. Це з'єднання пружинного типу, яке можна використовувати, наприклад, для підйому та переміщення об'єкта, що діє під дією сили тяжіння.

Wheel Joint 2D – моделює колеса та підвіску.

Всі компоненти джоінтів можна знайти у вкладці Component → Physics 2D.



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.

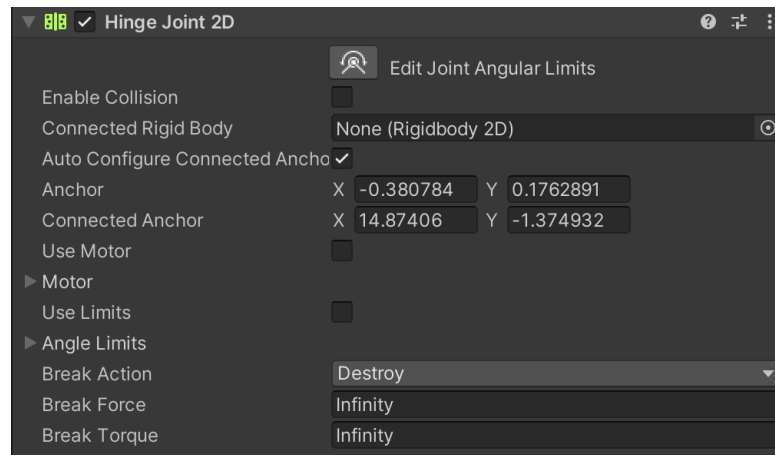


Рисунок 6.2 – Приклад компоненти Hinge Joint 2D

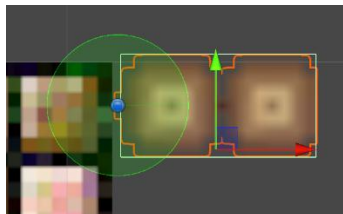
Налаштування компонента:

- 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 умовчанням стоїть значення Infinity, при якому розрива 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 умовчанням 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 і 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

3. Система частинок

Компонент Particle System (Система частинок) імітує поточні об'єкти, такі як рідина, хмара та пламя, створюючи та анімуючи велику кількість невеликих 2D-образів у сцені.

Particle System	Effects	>
Particle System Force Field	Light	>
Trail	Audio	>
Line	Video	>

Панель Ефект частинок у Сцені містить деякі додаткові параметри для попереднього перегляду частинок системи.



Властивість	Функції
Simulate Layers	Дозволяє попередньо переглянути частки системи, які не вибрані. За замовчуванням у представленій сцені створюються тільки вибрані системи частинок. Однак, якщо для параметра «Імітувати шар» задано значення, відмінне від Нічого, ефекти, відповідні Маски шару відтворюється автоматично, без необхідності їх вибору. Це особливо корисно для попереднього перегляду ефектів навколишнього середовища.
Resimulate	Коли це властивість включено, система частинок негайно застосовує зміни властивості до вже створених частинок. При відключенні системи частки залишають існуючі частки такими, які вони є, і застосовує зміни властивості тільки до нових частинок.
Show Bounds	Коли це властивість включено, Unity показує обмежувальний обсяг інформації про частки вибраних систем. Ці межі визначають, чи знаходяться частки системи в даний момент на екрані чи ні.
Show Only Selected	Коли ця властивість включена, Unity відкриває всі невибрані системи часток, що дозволяє вам зосередитися на створенні одного ефекту.

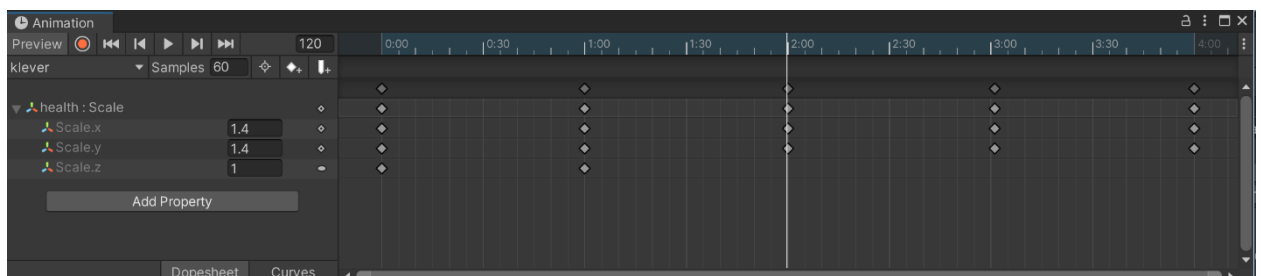
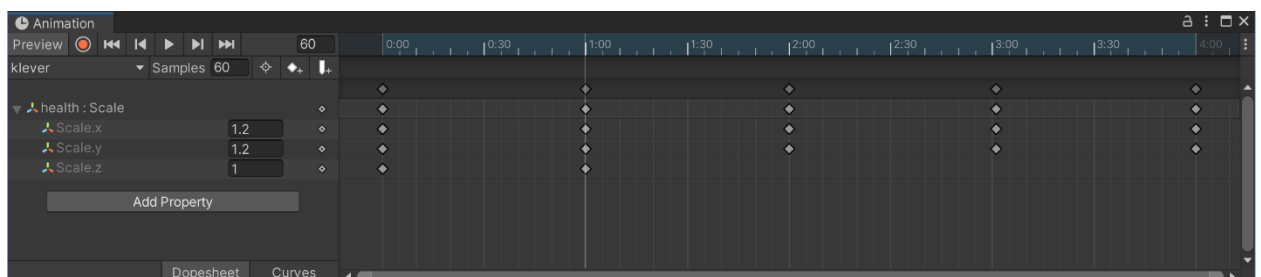
4. Пульсуюча анімація предмету

Обираємо об'єкт, який бажаємо зробити пульсуючим. Додаємо до нього компонент Collider2D. Викликаємо вікно Animation для цього об'єкта.

Для створення нового кліпу натискаємо кнопку Create та вказуємо назву анімації, що створюється. Для запису анімації натискаємо кнопку 

У вікні Animation додаємо властивості, що будуть змінюватися. Наприклад, місцеположення, масштаб тощо. Для цього натискаємо кнопку Add property.

Встановлюємо моменти часу на timeline. У кожному моменту часу змінюємо обрані властивості.



Можна на об'єкти додати можливість їх переносити по сцені.

Для цього створимо новий скрипт DragDrop. Додаймо до нього наступний код:

```

private Vector3 offset;

private void OnMouseDown()
{
    offset = gameObject.transform.position -
        Camera.main.ScreenToWorldPoint(new Vector3(Input.mousePosition.x, Input.mousePosition.y, 10.0f));
}

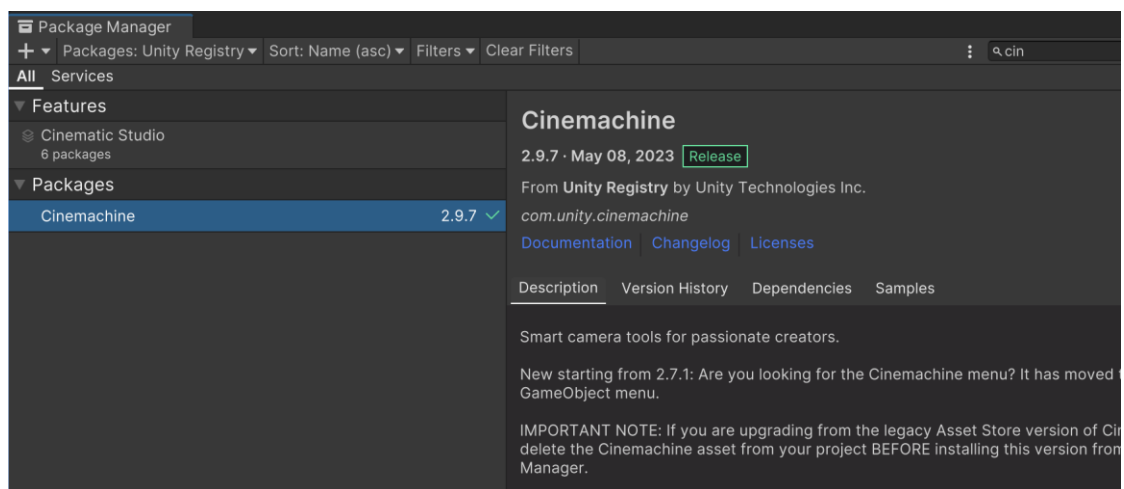
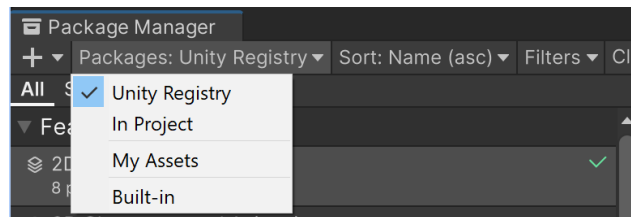
private void OnMouseDown()
{
    Vector3 newPosition = new Vector3(Input.mousePosition.x, Input.mousePosition.y, 10.0f);
    transform.position = Camera.main.ScreenToWorldPoint(newPosition) + offset;
}

```

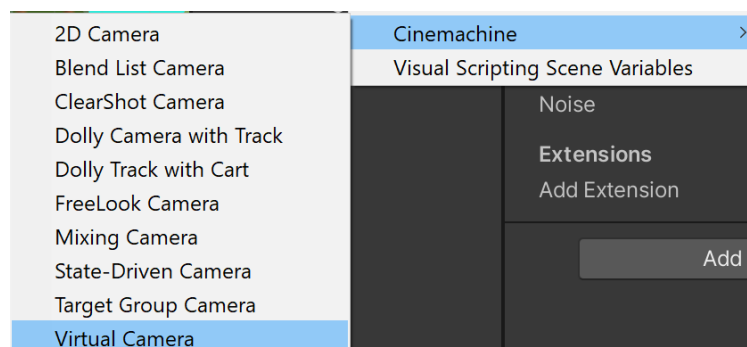
5. Камера, що рухається за персонажем

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.

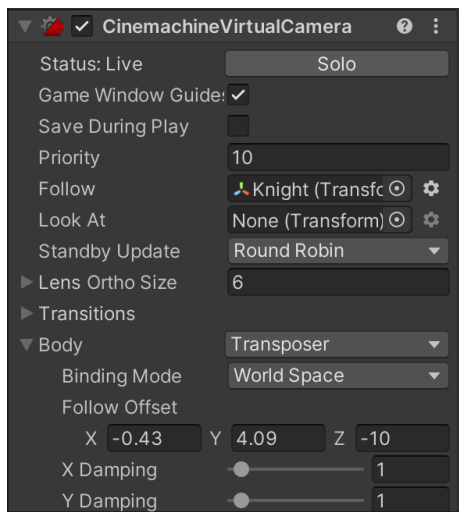


Далі створюємо на сцені новий об'єкт типу Virtual Camera:



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.

6. Створення фону, що рухається

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.

Особливості візуального ефекту паралаксу в іграх:

- Шаровість: Візуальний ефект паралаксу включає використання декількох шарів фону, кожен з яких рухається з різною швидкістю. Це створює ілюзію глибини та додає у гру деяку реалістичність.
- Різні швидкості руху: Кожен шар фону рухається із різною швидкістю, щоб створити ілюзію руху. Шари, які знаходяться ближче до спостерігача, рухаються швидше, а ті, що знаходяться далі, рухаються повільніше. Це додає глибину та реалістичність ігрового світу.
- Ефект плавності: Паралакс-ефект також може створити враження плавного руху, особливо під час використання різних переходів між шарами фону. Це дозволяє зробити ігровий світ більш живим та привабливим для гравця.

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 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.

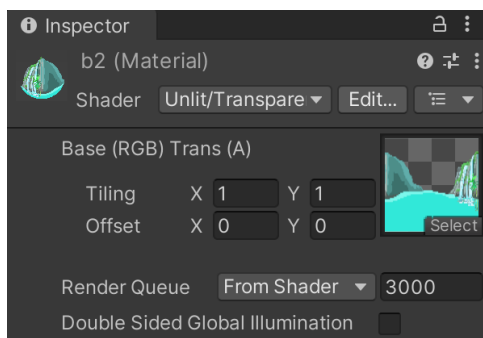
Створимо новий скрипт SimpleMove:

```
public class SimpleMove : MonoBehaviour
{
    private float speed = 3.0f;

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

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 найдаліший, 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.

Контрольні питання

1. Що таке оточення в грі?
2. Що таке Tile?
3. Яку функціональність потрібно додати персонажу, щоб він міг ходити сходами?
4. Який компоненти потрібно додати об'єкті, щоб він виконував функції підвісного мосту?
5. Для чого використовуються компоненти Joint 2D?
6. Що таке система частинок? Для чого вони використовуються?
7. Для чого використовується компонент Particle System?
8. Як створити пульсуючу анімацію для об'єкта?
9. Які є способи, що камера слідувала за персонажем?
10. Що таке паралакс-ефект?

Лекція 7. Огляд популярних атак на комп'ютерні ігри та методи їх проведення

1. Основні визначення
2. Популярні атаки на комп'ютерні ігри

1. Основні визначення

Кібератака – це навмисна спроба окремої особи або групи осіб зламати, пошкодити або порушити роботу комп'ютерних систем, мереж або цифрових пристроїв, часто з метою зловмисного використання, таких як крадіжка даних, шпигунство, фінансове шахрайство або саботаж системи.

У період з 2019 по 2025 роки експерти з кібербезпеки виявили тривожне зростання кількості спроб кібератак, пов'язаних із відеоіграми. Ця тенденція впливає як на найпопулярніші ігри, так і на звички ігрової спільноти щодо завантажень, особливо серед молоді. У цей період було зафіксовано понад 2025 мільйонів спроб кібератак із використанням файлів, замаскованих під відеоігри.

Зловмисники користуються довірою та цікавістю геймерів, які часто шукають неофіційні моди, чити чи версії на неофіційних сайтах. Наслідки цих атак не обмежуються зараженням пристроїв. Один із найсерйозніших наслідків — крадіжка облікового запису відеоігри. Багато з цих облікових записів, які можуть містити скіни, прогрес та цінні предмети, зрештою, перепродаються на підпільних ринках.

- Сучасні комп'ютерні ігри – це складні програмні системи, що включають:
- Клієнтську частину – гра на ПК, консолях, мобільних пристроях;
- Серверну частину – онлайн-логіка, бази даних, авторизація;
- Мережеві протоколи обміну даними;
- Економіку – внутрішньоігрова валюта, предмети, донат.

Ігри є привабливою метою атак з наступних причин:

- Фінансова мотивація – продаж читів, ботів, зламаних акаунтів;
- Змагальний фактор – бажання отримати перевагу;
- Низький рівень захисту клієнтської логіки;
- Велика аудиторія та слабка цифрова грамотність користувачів.

Важливо розуміти, що ігрові атаки не відрізняються принципово від атак на звичайні програми, але використовують особливості ігрової логіки.

Статистика кібератак, пов'язаних з відеоіграми, показує, що це явище зачіпає не тільки молодь, але й будь-який гравець, що взаємодіє з іншими гравцями за межами офіційних каналів, може наражатися на ризик втрати даних або поставити під загрозу свою конфіденційність. Найкращим захистом залишаються обережність та правильне використання засобів безпеки, щоб насолоджуватися грою без побоювань.

Розглянемо деякі специфічні терміни, які використовуються в геймплеї.

Перші чит-коди у іграх додавали самі розробники й тестувальники. Спочатку це були вбудовані інструменти, які девелопери використовували, щоб легше знайти баги. Чит-коди дозволяли тестувати функції, швидко перевіряти рівні та налаштовувати параметри.

З часом розробники виявили, що чит-коди можуть зацікавити гравців. Так, ігрові студії почали залишати деякі чит-коди в іграх. Чит-коди були приховані від геймерів, а деякі можна було розкрити через послідовності дій. Серед яскравих прикладів — відеогра Doom (1993), де можна знайти чит-код після вибору місії та введення коду.

Чит-коди стали популярними елементами у геймдев-культурі. Гравці обмінювалися відомими кодами, а ігрові журнали публікували їх для зацікавлення читачів. Спільноти гравців досі активно вивчають ігри, шукаючи нові чит-коди або неочікувані можливості, щоб розширити свій геймплей.

Щоб геймери не використовували чит-коди, розробники вдаються до античит-систем. Вони виявляють надмірне використання певних функцій (це часто вказує, що тут були чити)

та вживають заходів для їхнього припинення. Серед методів, якими розробники карають за читерство, — карантин, блокування гравців і тимчасові виключення.

Чит-коди створюють заздалегідь, і їхній функціонал спеціально заклали в гру розробники. А чит — це сторонній софт, який знаходить вразливість у відеогрі та дозволяє «змахлювати».

Мод (від англійського "modification" – модифікація) – це доповнення гри, яке змінює її зміст, геймплей, графіку, аудіо, або інші аспекти. Модифікації дозволяють гравцям створювати власні варіанти гри, змінюючи її під свої вподобання або додаючи нові функції.

Моди можуть бути створені спільнотою гравців або розробниками гри. Вони можуть бути досить простими, такими як зміна вигляду персонажів або об'єктів у грі, або складними, такими як додавання повноцінних історій, місій, нових персонажів, режимів гри або навіть створення зовсім нових світів ігрового процесу. Завдяки модам гравці можуть насолоджуватися більш різноманітним геймплеєм і продовжити ігровий досвід після того, як пройшли оригінальну гру.

Моддери – це ті, хто створює моди. Вони можуть брати за основу код гри та інші ресурси, що надаються розробниками, і змінювати їх згідно зі своїми ідеями та вміннями.

Важливо зазначити, що не всі ігри дозволяють модифікації або вони можуть бути обмежені правами або заборонені політикою розробника чи видавця гри. Крім того, встановлення модів може бути пов'язане з ризиком, так як неправильні модифікації можуть порушити стабільність гри або спричинити проблеми з її функціонуванням.

2. Популярні атаки на комп'ютерні ігри

Геймінг – це мільярдний ринок, де обертаються великі гроші, і саме тому геймери є бажаною цілью для кіберзлочинців. Адже навіть у грі користувач може втратити персональні дані, гроші або доступ до акаунтів.

Усі атаки умовно поділяються на кілька груп:

- Атаки на клієнтську частину – модифікація пам'яті, підміна логіки, ін'єкції коду;
- Атаки на ігрові дані – підміна збережень, зміна конфігураційних файлів;
- Мережеві атаки – перехоплення пакетів, заміна запитів;
- Атаки на серверну частину – SQL-ін'єкції, логічні уразливості;
- Атаки на ігрову економіку та акаунти – боти, фішинг, перебір паролів.

Memory Hack (атаки на пам'ять) – це зміна значень оперативної пам'яті гри під час її виконання.

Як проводиться атака:

- Гра завантажується на пам'ять (RAM);
- Зловмисник використовує інструменти: Cheat Engine, ArtMoney, Process Hacker;
- Виконується пошук значень (наприклад, кількість здоров'я = 100);
- Значення змінюється (наприклад, 9999);
- Гра продовжує використовувати підмінене значення.

Cheat Engine, ArtMoney та Process Hacker – усі ці програми використовуються для модифікації пам'яті ігор, дозволяючи змінювати різноманітні ігрові параметри, такі як гроші, здоров'я, боєприпаси тощо. Проте їх функціонал обмежений.

Cheat Engine та ArtMoney вимагають певних навичок та знань для ефективного використання, на відміну від платних читів, які часто пропонують більш розширений функціонал. ArtMoney має інтуїтивно зрозумілий інтерфейс, який легко використовують навіть новачки, але використання програми може призвести до бана облікового запису або інших наслідків у розрахованих на багато користувачів або онлайн-іграх. Cheat Engine, у свою чергу, дозволяє змінювати будь-які параметри, але має складний інтерфейс, який може бути непридатним для новачків.

Атака шкідливих модів та «безплатні чити» – чимало гравців прагнуть модифікувати ігровий досвід: встановлюють нові карти, візуальні покращення, чити. Саме на цьому й

грають кіберзлочинці: вони створюють моди, які містять троянські програми, майнери чи кейлогери. Гравець запускає мод, а разом із ним активується шкідливий процес, що викрадає паролі або використовує ресурси ПК.

Кейлогери (keyloggers) – це програми, які призначені для записування та збереження всіх натискань клавіш на клавіатурі комп'ютера або смартфона. Кейлогери можуть використовуватися для злочинних цілей, таких як злам банківських рахунків, крадіжка паролів та інших конфіденційних даних, шпигунство за користувачами, відстеження дій працівників та інше.

Майнери – це програми, що використовують потужність вашого комп'ютера для видобутку криптовалют без вашого відома. У контексті ігор це особливо небезпечно, оскільки вони можуть: значно навантажувати процесор та графічну карту, сповільнюючи роботу ігор і системи; викликати перегрів компонентів та зношування обладнання; встановлюватися приховано, наприклад, через модифіковані ігрові файли або шкідливі завантаження. Користувач може навіть не помітити встановлення майнера, оскільки він працює у фоновому режимі, споживаючи ресурси без прямого втручання.

Атаки на збереження (Save Game Hack) – редагування файлів збережень (JSON, XML, Binary).

Як проводиться:

- Пошук папки збережень;
- Відкриття файлу в текстовому чи hex-редакторі;
- Зміна значень (гроші, рівні, предмети).

Причини вразливості – відсутність шифрування та відсутність підпису цілісності.

Speed Hack (атаки на якийсь час) — зміна швидкості перебігу ігрового часу.

Механізм атаки:

- перехоплення системних викликів часу (Time.time, deltaTime);
- уповільнення чи прискорення таймера гри;
- підміна часових коефіцієнтів.

Ефекти:

- прискорення персонажа;
- прискорений видобуток ресурсів;
- швидке прокачування.

Причини можливості атаки – прив'язка логіки до локального часу клієнта; відсутність серверної синхронізації.

Фишингові атаки – наймасовіший тип шахрайства в геймінгу. Хакери створюють копії відомих сайтів (Steam, Battle.net, Xbox Live тощо), надсилають гравцеві посилання через чат, email або Discord, пропонуючи безплатні скіни, бонуси або участь у розіграші. Після переходу на фейковий сайт гравець вводить логін і пароль – і за секунди втрачає контроль над акаунтом. Часто облікові записи з цінними ігровими предметами одразу перепродаються на чорному ринку.

Мережева атака в комп'ютерних іграх – це спеціальна дія, спрямована на несанкціонований доступ, спотворення або блокування роботи ігрового сервісу або клієнта гри. Мета таких атак зазвичай полягає у контролі над грою, отриманні переваги над іншими гравцями, вимкненні серверів чи перехопленні конфіденційних даних.

Мережеві атаки та Packet Manipulation – перехоплення мережевого трафіку, заміна пакетів, повтор запитів (Replay Attack) тощо.

Інструменти: Wireshark; Fiddler; Burp Suite.

Типові цілі:

- підміна нагород;
- фальшиві запити на купівлю;
- підміна координат.

Wireshark використовується для захоплення та аналізу мережевого трафіку, допомагаючи розуміти роботу мережі і діагностувати проблеми з підключенням та безпекою.

Основні функції Wireshark:

- Захоплення мережевого трафіку – Wireshark дозволяє перехоплювати пакети даних, що передаються через комп'ютерну мережу, у режимі реального часу. Це може бути трафік Ethernet, Wi-Fi, Bluetooth, TCP/IP та інших протоколів.
- Аналіз протоколів – програма розпізнає сотні мережевих протоколів і дозволяє переглядати деталі кожного пакета — адресу відправника та одержувача, заголовки протоколів і вміст переданих даних. Це корисно для налагодження та навчання.
- Виявлення проблем і помилок – Wireshark допомагає знаходити проблеми з мережею, наприклад, втрачені пакети, затримки у з'єднанні або неправильну конфігурацію маршрутизаторів та серверів.
- Безпека та аудит мережі – завдяки аналізу трафіку можна виявляти підозрілу активність, спроби несанкціонованого доступу або атаки на мережу. Wireshark застосовують у тестуванні мереж та аудиті безпеки.
- Фільтри та звітність – програма дозволяє застосовувати фільтри для відбору конкретних протоколів, IP-адрес чи портів, що значно спрощує роботу з великим обсягом даних. Захоплені пакети можна зберігати для подальшого аналізу та створення звітів.

Fiddler – це інструмент для перехоплення, аналізу та модифікації HTTP- і HTTPS-трафіку, який допомагає розробникам та тестувальникам відстежувати та налагоджувати запити до серверів. Він дозволяє бачити всі HTTP- та HTTPS-запити та відповіді, включно із заголовками, тілом запитів, параметрами та куки. Це допомагає виявляти помилки, вразливості та проблеми зі швидкістю завантаження веб-ресурсів.

Burp Suite – це інтегрована платформа для тестування безпеки веб-додатків, яка дозволяє перехоплювати, аналізувати та модифікувати HTTP/HTTPS-трафік, автоматично і вручну виявляти вразливості, а також проводити різноманітні атаки для оцінки безпеки веб-сайтів чи API.

Burp Suite використовується для тестування на проникнення вебдодатків. Вона встановлюється між браузером користувача і сервером так, щоб перехоплювати всі запити і відповіді, дозволяючи досліджувати веб-трафік, шукати уразливості, модифікувати запити або вводити автоматизовані сценарії атак.

DDoS-атаки спрямовані на перевантаження ігрових серверів, щоб вивести їх з ладу. Це не тільки шкодить розробникам, але й гравцям – у рейтингових матчах втрата з'єднання може призвести до бану чи втрати досягнень.

IP-спуфінг і перехоплення пакетів – зловмисник підмінює власну IP-адресу на адресу легітимного користувача. Перехоплення або маніпуляція ігровими пакетами дозволяє змінювати стан гри або отримувати конфіденційні дані гравців.

Трояни та мережеві хробаки – програми, що заражають комп'ютери користувачів і автоматично передають управління атакуючому. Можуть викрадати облікові дані чи змінювати поведінку клієнта гри.

Sniffing і перехоплення трафіку LAN – у локальних мережах зловмисник може отримати доступ до всього обміну даними між комп'ютерами. Це дозволяє читати або змінювати пакети гри безпосередньо на маршрутах передачі.

Мережева атака в іграх – це форма віддаленого втручання, яка може порушити конфіденційність, цілісність або доступність ігрових ресурсів. Відмінною особливістю саме ігрових атак є їх орієнтація на ігрову логіку, latency та взаємодію між клієнтом і сервером, що відрізняє їх від загальноприйнятих кібернетичних атак на корпоративні системи. Правильна організація захисту та використання сучасних методів шифрування і аналізу трафіку дозволяє мінімізувати ризик таких атак.

Не всі кібератаки – це технічні трюки. Соціальна інженерія передбачає маніпуляції, коли зловмисник налагоджує «довірливі» стосунки з гравцем або навіть співробітником студії.

Кіберзагрози в геймінгу – це реальність, з якою стикаються як розробники, так і гравці. Щоб убезпечити себе, дотримуйтесь кількох правил:

- Використовуйте двофакторну автентифікацію.
- Не переходьте за підозрілими посиланнями та не вводьте свої дані на незнайомих сайтах.
- Регулярно оновлюйте антивірусне програмне забезпечення та операційну систему.
- Будьте обережні з модами та сторонніми програмами – завантажуйте їх лише з перевірених джерел.

Цікаві факти про кібербезпеку в іграх:

- За даними дослідження, понад 50% гравців стикалися зі спробами зламу своїх акаунтів.
- У 2020 році кількість атак на ігрові платформи збільшилася на 300% порівняно з попередніми роками.
- Згідно зі статистикою, 80% користувачів використовують один і той же пароль для кількох облікових записів.
- Двофакторна автентифікація може знизити ризик зламу облікового запису на 99,9%.
- Деякі ігри використовують систему блокування IP-адрес для запобігання шахрайству.
- Шкідливі програми можуть ховатися під виглядом популярних ігор чи модифікацій.
- Соціальна інженерія є однією з найпоширеніших тактик шахраїв у ігровій індустрії.
- Багато великих ігрових компаній інвестують мільйони доларів у кібербезпеку для захисту своїх користувачів.

Контрольні питання

1. Що розуміється під терміном кібератака?
2. Які причини привабливості ігор для атак?
3. Що таке чіт-коди в іграх? Для чого використовуються?
4. Що таке моди?
5. Як проводиться атака на пам'ять?
6. Як проводиться атака шкідливих модів та «безплатні чіти»?
7. У чому суть атаки на файли збереження?
8. Як проводиться атака на ігровий час?
9. Що таке фішингові атаки?
10. У чому суть DDoS-атак?

Лекція 8. Методи захисту від злому в іграх

1. Захист ігрових даних
2. Обфускація

1. Захист ігрових даних

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.

Основні методи захисту JSON в іграх:

- Шифрування (Encryption) – перетворення JSON-тексту в нечитабельний вигляд перед записом на диск;
- Бінарна серіалізація – збереження даних у бінарному форматі (наприклад, MessagePack, Protobuf), що ускладнює пряме редагування;
- Контрольні суми (Checksums/Hashing) – створення унікального хешу файлу. При завантаженні гра перевіряє, чи не був файл змінений (hash mismatch), і відхиляє його;
- Обфускація – заплутування ключів та значень у JSON, щоб зробити їх менш зрозумілими для людини;
- Серверне зберігання – найбільш надійний метод зберігання критичних даних (рівень, валюта) на сервері, а не на пристрої користувача.

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.

Ключові аспекти Base64 в ігровій індустрії:

- Кодування збережень – рядки збережень (save strings) часто кодуються в Base64 для використання у хмарних сховищах або обміну між гравцями, що також є елементарною формою «обфускації» (утруднення читання) даних.
- Мережевий обмін – дозволяє передавати бінарні дані, такі як аватари гравців або ігрові об'єкти, через системи, які підтримують лише текст.

- Впровадження ресурсів – маленькі зображення або текстури можуть бути вбудовані прямо в код або JSON-файли за допомогою Base64, що зменшує кількість запитів до сервера.
- Безпека (URL-safe) – у браузерних або онлайн-іграх використовується Base64URL (із заміною символів + та / на - та _), що запобігає помилкам при передачі даних в URL-адресах.

Алгоритм кодування Base64 (покроково):

- Розбиття на байти – вихідні дані діляться на групи по 3 байти ($3 \times 8 = 24$ біти).
- Перегрупування 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);
```

Файл settingsSer.json:

```
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
  }
]
```

Файл data_b64.txt:

```
File Edit View

Їw0KICB7DQogICAgIk5hbWUiOiAiT2xlbmEiLA0KICAgICJMaWZlIjogMTAwLA0KICAgICJNb25leSI6IDI1MCwNCiAgICAIQXJ0aWZhY3RzQ3N2IjogIlJpbmcsIEFtdWxldCwgS2V5IiwNCiAgICAIQXJ0aWZhY3RzIjogWw0KICAgICAgIlJpbmciLA0KICAgICAgIkFtdWxldCIsDQogICAgICAIIS2V5IG0KICAgIF0sDQogICAgIk51bSI6IDMNCiAgfSwNCiAgew0KICAgICJ0YW1lIjogIkl2YW4iLA0KICAgICJMaWZlIjogODAsDQogICAgIk1vbWV5IjogMTIwLA0KICAgICJBcnRpZmFjdHNdC3YiOiAiTWFwLCBQb3Rpb24iLA0KICAgICJBcnRpZmFjdHMiOiBbDQogICAgICAIWFwIiwNCiAgICAgICJQb3Rpb24iDQogICAgXSswNCiAgICAIbnVtIjogMg0KICB9DQpd
```

Файл settingsBase64.json:

```
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
  }
]
```

Хоча Base64 зручний, він не підходить для великих даних через зростання розміру файлу.

Дані, закодовані в Base64, легко декодувати назад, тому що це просто інший формат подання, а не приховування інформації. Для приховування інформації використовують шифрування даних.

Сучасні методи шифрування поділяються на потокові та блочні.

Характерною особливістю блокових криптоалгоритмів є той факт, що в ході своєї роботи вони роблять перетворення блоку вхідної інформації фіксованої довжини і отримують результуючий блок того ж обсягу, але недоступний для прочитання стороннім особам, які не мають ключа.

Таким чином, схему роботи блокового шифру можна описати функціями

$$Z = \text{EnCrypt}(X, \text{Key})$$

та

$$X = \text{DeCrypt}(Z, \text{Key})$$

Ключ (Key) є параметром блокового криптоалгоритму і є деяким блоком двійкової інформації фіксованого розміру. Вхідний (X) та зашифрований (Z) блоки даних також мають фіксовану розрядність, рівну між собою, але необов'язково рівну довжині ключа.

Основним принципом побудови сучасних блокових шифрів є умова їхньої практичної стійкості до усіх відомих атак поряд з можливістю ефективної реалізації алгоритмів шифрування на різних платформах.

Стійкість блокового шифру визначається властивостями його раундових функцій, типом ключового суматора, довжиною блоку, довжиною ключа та кількістю раундів шифрування;

Стандарт AES (Advanced Encryption Standard) є стандартом шифрування Сполучених Штатів Америки та включає в себе алгоритм Rijndael.

Алгоритм Rijndael розроблений двома фахівцями з криптографії з Бельгії Rijmen та Daemen, вимовляється як «рейндал». Він є нетрадиційним блоковим шифром, оскільки не використовує мережу Фейштеля для криптоперетворень.

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).

Вектор ініціалізації IV змішується з першим блоком даних (зазвичай через операцію XOR) перед початком шифрування. У сучасних режимах (наприклад, CBC) результат шифрування першого блоку впливає на наступний, створюючи «ланцюжок».

IV не обов'язково тримати в секреті – його часто передають разом із зашифрованим повідомленням у відкритому вигляді. Для кожної нової сесії шифрування потрібно генерувати новий IV. Повторне використання однієї й тієї ж комбінації «ключ + IV» критично знижує безпеку.

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 бітів) 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 класу *SymmetricAlgorithm*, 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($"Вектор ініціалізації: {Convert.ToHexString(aes.IV)}");
Console.WriteLine($"Ключ: {Convert.ToHexString(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
}
]
```

```
File Edit View
~"eA·~*·2тУ6mШш•ЦЛЛДшДйw"Цы•М°эр

JAG0CFLеЙIЧRjb)% ьlэmзPоFльё"...ёSum@ьVDшд
9ьнfn~ы!HІЯВN~Т~в2Т!іХЩ·аіФQKeRІжvRrm".сб,с-хжЕ\ьВҮIш^э,,N°•ььЩЩ# /
~!Ны"йДЖй%<ёYД-сK_V>йс3і%.3тШтCн7аьнфv3Жвр@и#П,'b3с@ч
ЙYтV-Дь8рцШАН°гiQOTf;'г,~чрдк~"-uoNf»жKXН±=_D°ЖЩЩ'*x iHB-Дж-
P4шM`_CJC
hcгfTђНЯЛ°К"ШШHTӨґг-7jУЖNсю4TP?"»ьтMЗґьч«kI ьлп...
...И?rл",
ФьjtH>jaLЭЭLEб
```

2. Обфускація

Сканування значень – це процес пошуку відповідних значень у виділеній пам'яті гри та перезапису їх іншими значеннями. Найчастіше використовується для збільшення здоров'я гравця, боезапасу зброї чи будь-якої іншої цінності, яка могла б дати хакеру несправедливу перевагу у грі.

Як правило, хакери-початківці націлюються лише на значення, які відображаються на екрані, та знають, для чого вони використовуються (наприклад, здоров'я гравця, боеприпаси тощо).

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.

Обфускація не забезпечує абсолютної безпеки, і великі обфусковані програми все ще можуть бути розібрані. Однак вона створює додаткові труднощі для тих, хто намагається аналізувати або модифікувати вихідний код.

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 зломисників.

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 зломисників, 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 зломисників, 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.

Gameplay-honeypot (ігрові пастки) – це вбудовані у геймплей елементи, які недоступні звичайному гравцю та доступні лише через чити/хаки. Наприклад, предмет за стіною (його «бачать» лише WallHack) або скриня з нульовою нагородою, яку «відкривають» лише боти

або невидимий NPC, з яким взаємодіють лише скрипти. Легальний гравець ніколи не активує honeypot.

Data-honeypot (пастки у даних) – навмисно створені фейкові або контрольні дані. Наприклад, поле в JSON/XML-файлі, збереженні дані, яке не використовується грою, але змінюється читерами. Наприклад, створення «фейкових» змінних (player_gold_shadow, debug_speed_value): якщо значення змінене, це говорить про 100% індикатор злому.

AI / Behavior honeypot – пастки на основі поведінки. Наприклад, ресурс у зоні, до якої неможливо дістатися без teleport hack або ціль, що потребує нелюдської точності реакції або фейкові події, які бачать лише ESP-чити.

Honeypot у іграх – це інтелектуальна пастка, яка дозволяє не боротися з читерами напряму, а змусити їх видати себе самих.

Контрольні питання

1. Як захистити ігрові дані?
2. Для чого використовується шифрування?
3. Що робить Base64-перетворення?
4. Який сучасний стандарт шифрування?
5. Який алгоритм лежить в основі сучасного стандарту шифрування?
6. Що таке режими роботи AES? Для чого використовуються?
7. Що таке обфускація?
8. Які методи обфускації для захисту ігор ви знаєте?
9. Що таке honeypot?

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_Unity/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