Міністерство освіти і науки України
Міжнародний гуманітарний університет
Кафедра комп’ютерної інженерії та інноваційних технологій

Лебедєва О.Ю.

РОЗРОБКА ІГРОВИХ ЗАСТОСУНКІВ

Методичні рекомендації до практичних робіт
для здобувачів вищої освіти,
які навчаються за спеціальностями
галузі «Інформаційні технології»

Одеса 2025

Затверджено Вченою Радою Міжнародного гуманітарного університету.
Протокол № 5 від 20 січня 2025 р.

Лебедєва О.Ю Розробка ігрових застосунків. Методичні рекомендації до практичних робіт здобувачів вищої освіти, які навчаються за спеціальностями галузі «Інформаційні технології» [Електронне видання]. Кафедра комп'ютерної інженерії та інноваційних технологій Міжнародного гуманітарного університету. Одеса, 2025. 58 с.

Дисципліна «Розробка ігрових застосунків» спрямована на формування у здобувачів вищої освіти спеціальності Інженерія програмного забезпечення компетентностей, пов'язаних із проєктуванням та розробкою програмних застосунків ігрового призначення з використанням сучасних інструментальних середовищ створення інтерактивних систем. Вона орієнтована на формування здатності застосовувати технології програмної інженерії під час створення ігрових застосунків, розробляти програмні модулі ігрової логіки, організовувати взаємодію програмних компонентів та реалізовувати інтерактивні програмні системи.

ЗМІСТ

Тема 1. Введення в розробку ігрових застосунків
Практична робота 1. Створення гри на C#

Мета роботи: познайомитися з мовою програмування C# та принципами об'єктно-орієнтованого програмування. Навчитися створювати ігри за допомогою мови програмування C#, використовуючи об'єктно-орієнтований підхід.

1. Теоретичний матеріал

Створимо додаток, в якому реалізуємо варіант гри «Flappy Bird».

Flappy Bird – мобільна гра для платформ iOS і Android розроблена в'єтнамським розробником Донг Нгуєном (англ. Dong Nguyen). Гра була випущена 24 травня 2013.

Гравець керує польотом пташки, яка рухається між рядами перешкод у вигляді вертикальних зелених труб, що утворюють тунель складної форми. При зіткненні з ними гра завершується. Пташка летить вперед сама, управління здійснюється торканням до екрана — тоді пташка змахує крилами і робить невеликий ривок вгору. В іншому разі пташка не махає крилами і падає. За кожний успішний проліт між двома трубами зараховуються очки.

Створимо новий проект Windows Forms App (.NET Framework). Маємо такий вигляд проекту:

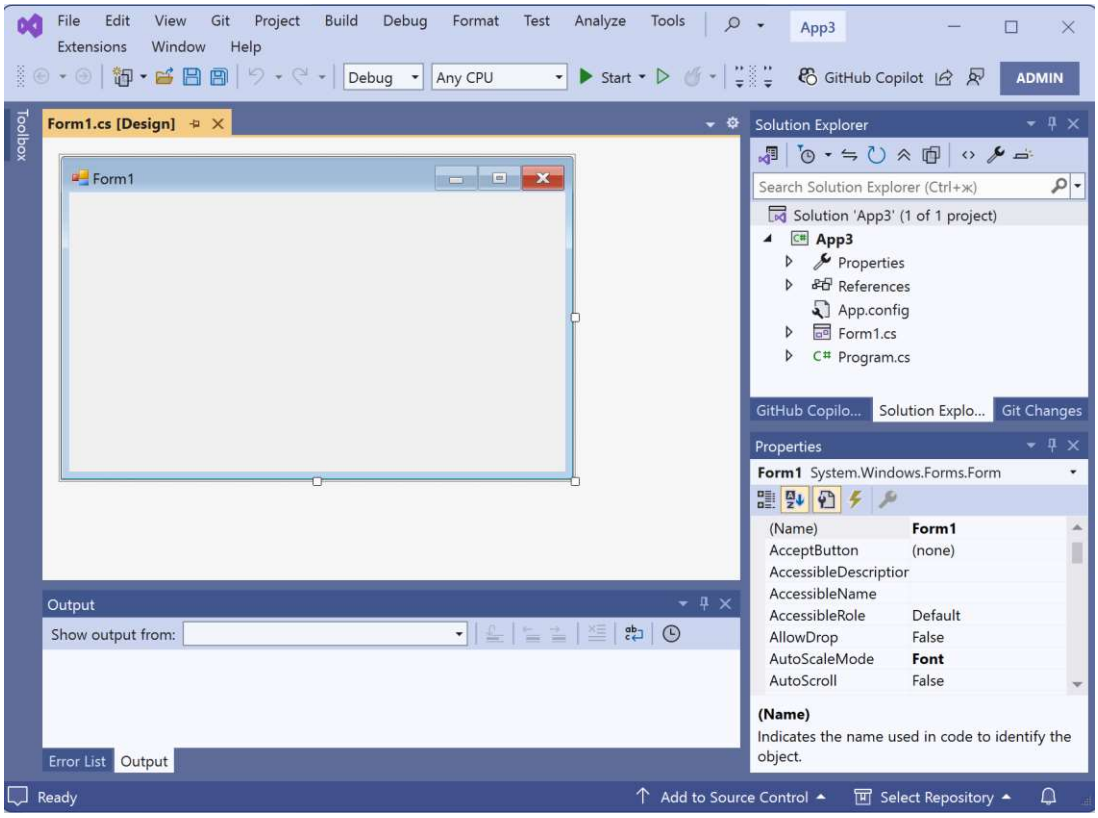
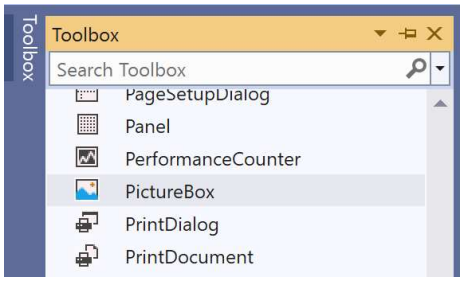
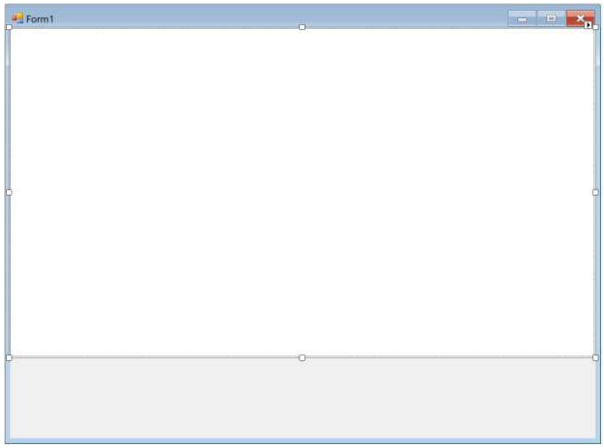


Рисунок 1.1 – Вид проекту Windows Forms App (.NET Framework)

Додаймо на форму такий компонент як PictureBox. Для цього натискаємо вкладку Toolbox та в вікні, що з'явиться обираємо PictureBox та перетягуємо на форму та надаємо потрібно розміру також змінимо розмір форми.

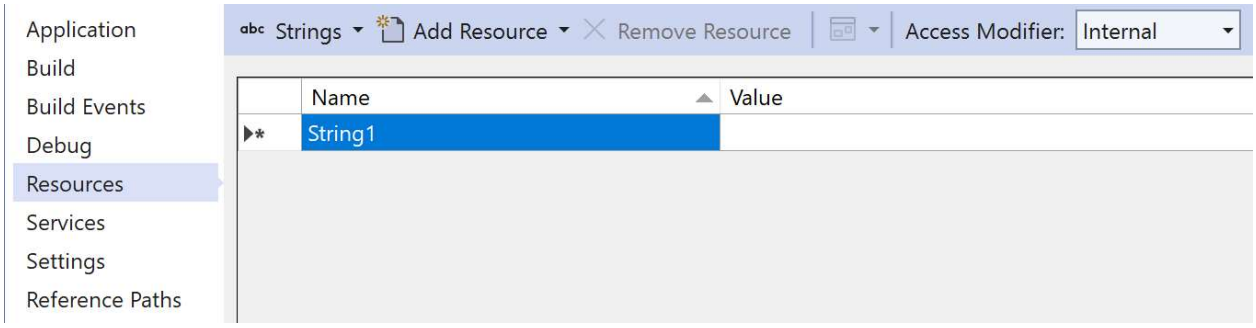


Виділимо PictureBox та в вікні Properties замінимо властивість форми BackColor на значення White:

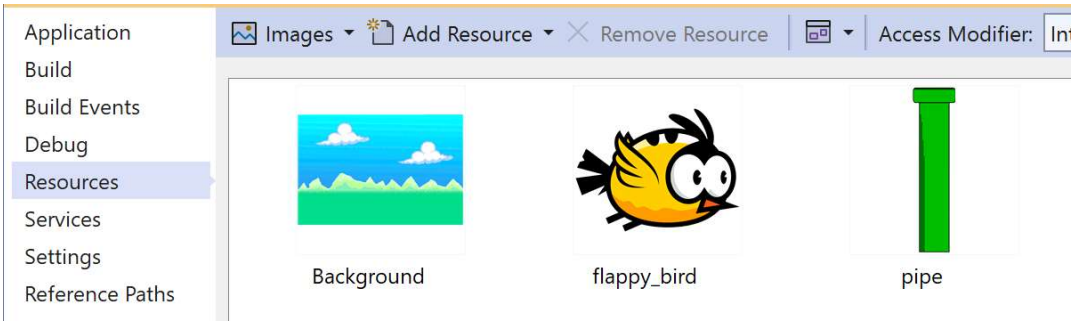


Створимо два нових класу для об’єкту Bird та Pipe. Для цього у вікні Solution Explorer виділяємо назву проекту та натискаємо праву кнопку миші. Обираємо Add → Class. У вікні, що з’явиться внизу напишемо назву класу, наприклад Bird. Аналогічно створюємо новий клас Pipe. Відкриємо ці класи. Для цього необхідно подвійно клацнути на ім’я класу в вікні Solution Explorer.

Додаймо картинки фону, птиці та труби як ресурси до нашого проекту. Для цього натискаємо на назву проекту правою кнопкою миші та обираємо Properties. З’явиться вікно в окремій вкладці. В цьому вікні переходимо до пункту Resources.



Далі визиваємо список для кнопки Add Resource та в цьому списку обраємо Add Existing File. Далі обираємо почергово картинки, які бажаємо додати. В результаті, наприклад, маємо такий вигляд вікна Resources:



Додаймо до класу Bird такий код:

```
internal class Bird
{
    public float x;
    public float y;

    public Image imgBird;
    public int size;

    public float gravityValue;
    public bool isAlive;

    public Bird(float x, float y)
    {
        this.x = x;
        this.y = y;

        size = 60;
        gravityValue = 0.125f;
        isAlive = true;

        imgBird = Properties.Resources.flappy_bird;
    }
}
```

Для роботи з класом Image в файлі Bird.cs та Pipe.cs підключить `using System.Drawing;`

Заповнимо інший клас Pipe:

```
internal class Pipe
{
    public float x;
    public float y;

    public Image imgPipe;
    public int sizeX;
    public int sizeY;

    public Pipe(float x, float y, bool isRotate = false)
    {
        this.x = x;
        this.y = y;

        sizeX = 80;
        sizeY = 250;

        imgPipe = Properties.Resources.pipe;
        if (isRotate)
            imgPipe.RotateFlip(RotateFlipType.Rotate180FlipX);
    }
}
```

Повертаємось до форми. Щоб получити файл коду форми, у вікні Solution Explorer виділяємо Form1.cs, натискаємо праву кнопку миші та обираємо View Code. Маємо:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

Перед конструктором Form1() створимо необхідні поля:

```
Bird bird;
Pipe pipe1, pipe2;

Bitmap bmp;
private Graphics myGraphics;
Image backgroundImage;

private float gravity;

Timer timer;
```

Додаймо до конструктора Form1() після виклика методу InitializeComponent() наступний код:

Вкажемо розміри вікна:

```
this.Size = new Size(650, 700);
```

Створимо картинку, на якій будемо малювати наші об’єкти:

```
bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);
myGraphics = Graphics.FromImage(bmp);
pictureBox1.Image = bmp;
```

Получимо картинку фону гри:

```
backgroundImage = Properties.Resources.Background;
```

Створимо таймер та визначим метод обробки події таймеру:

```
timer = new Timer();
timer.Interval = 7;
timer.Tick += Timer_Tick;
```

Зробимо виклик методу, який опишемо після конструктору:

```
Instantiate();
```

Метод Instantiate створює об’єкти та запускає гру:

```
private void Instantiate()
{
    bird = new Bird(200,200);
    pipe1 = new Pipe(400, -100, true);
    pipe2 = new Pipe(400, 300);

    gravity = 0;

    PaintScene();
    timer.Start();
}
```

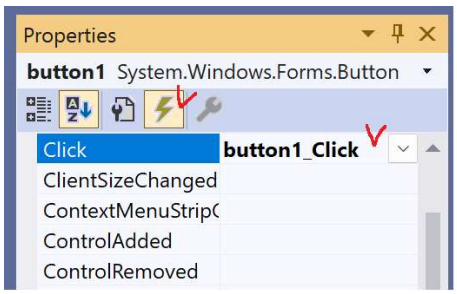
Метод PaintScene() малює картинки у PictureBox:

```
private void PaintScene()
{
    myGraphics.DrawImage(backgroundImage, 0, 0, pictureBox1.Width, pictureBox1.Height);

    myGraphics.DrawImage(bird.imgBird, bird.x, bird.y, bird.size, bird.size);
    myGraphics.DrawImage(pipe1.imgPipe, pipe1.x, pipe1.y, pipe1.sizeX, pipe1.sizeY);
    myGraphics.DrawImage(pipe2.imgPipe, pipe2.x, pipe2.y, pipe2.sizeX, pipe2.sizeY);

    pictureBox1.Refresh();
}
```

Якщо зараз запустити проект маємо таку картинку:

Далі додаймо рух птиці, труб та логіку гри. Для цього нам знадобиться метод Timer_Tick.

Перед тим як до нього переходити створимо кнопку (Button) на формі та змінимо його властивість Text на рядок Force. Створимо метод обробки натискання на цю кнопку. Для цього у вікні проекту Properties перейдемо на закладку Events та зробимо подвійний клік мишою в полі Click. В результаті в коді з’явиться метод button1_Click.



Запишемо наступний код в метод button1_Click:

```
private void button1_Click(object sender, EventArgs e)
{
    gravity = 0;
    bird.gravityValue = -0.125f;
}
```

Перейдемо до методу Timer_Tick:

```
private void Timer_Tick(object sender, EventArgs e)
{
    if (bird.y > 500)
    {
        bird.isAlive = false;
        timer.Stop();
    }

    if (Collide(bird, pipe1) || Collide(bird, pipe2))
    {
        bird.isAlive = true;
        timer.Stop();
    }

    if (bird.gravityValue != 0.1f)
        bird.gravityValue += 0.005f;

    gravity += bird.gravityValue;
    bird.y += gravity;

    MovePipes();
    PaintScene();
}
```

В методі Timer_Tick викликаються методи Collide та MovePipes, які необхідно створити.

Додаймо метод Collide, який перевіряє зіткнення птиці та труби:

```
private bool Collide(Bird bird, Pipe pipe)
{
    PointF delta = new PointF();
    delta.X = (bird.x + bird.size / 2) - (pipe.x + pipe.sizeX / 2);
    delta.Y = (bird.y + bird.size / 2) - (pipe.y + pipe.sizeY / 2);

    if (Math.Abs(delta.X) <= bird.size / 2 + pipe.sizeX / 2)
    {
        if (Math.Abs(delta.Y) <= bird.size / 2 + pipe.sizeY / 2)
            return true;
    }

    return false;
}
```

Метод MovePipes рухає труби справа на ліво:

```
private void MovePipes()
{
    pipe1.x -= 1;
    pipe2.x -= 1;

    CreatePipes();
}
```

В методі MovePipes викликається метод CreatePipes, який створює наві труби за новими координатами, коли попередні труби пройшли вліво за птицею.

Метод CreatePipes має такий код:

```
private void CreatePipes()
{
    if (pipe1.x < bird.x - 200)
    {
        Random r = new Random();
        int newY = r.Next(-100, 0);

        pipe1 = new Pipe(400, newY, true);
        pipe2 = new Pipe(400, newY+400);
    }
}
```

2. Завдання до практичної роботи 1

Запустити гру.

Додати до гри такий функціонал:

– Створити дві нові кнопки Start та Pause. При запуске програми маємо картинку з фоном, птицею та трубами, але гра не повинна працювати. Після натискання кнопки Start гра почне працювати. Кнопка Pause призупиняє гру.
– Додати силу птиці не через натискання кнопки Force (як зараз), а при натисканні кнопки миші по PictureBox.
– Додайте до форми мітку, в який буде показуватися кількість пройдених труб.
– Змініть код, щоб при проходженні певної кількості труб, труби почали швидше рухатися справа на ліво.

Тема 3. 3D. GameObject та його налаштування
Практична робота 2. Прототипування оточення на Unity

Мета роботи: познайомитися з прототипуванням оточення в іграх. Навчитися створювати ігри з використанням 3D шаблону Unity, створювати нові сцени та прототипування оточення використовуючи набір базових геометричних форм Unity та ProBuilder.

1. Теоретичний матеріал

Створимо додаток, в якому реалізуємо прототипування оточення на Unity.

Прототипування оточення в іграх – це процес створення швидких, чорнових версій локацій та рівнів гри для тестування ідей, механік та візуального дизайну до початку повномасштабної розробки. Мета такого прототипу — перевірити працездатність концепції, виявити потенційні проблеми на ранніх етапах та отримати зворотний зв'язок від команди та інвесторів, скоротивши витрати та час на розробку.

Перший етап розробки будь-якого рівня починається з простих ескізів та замальовок спільних ідей. Це допомагає візуалізувати і уявити собі майбутній рівень: яким буде шлях гравця, де розмістити основні об'єкти і як це все взаємодіятиме між собою. Прості начерки важливі, тому що їх легко обговорювати та коригувати на ходу.

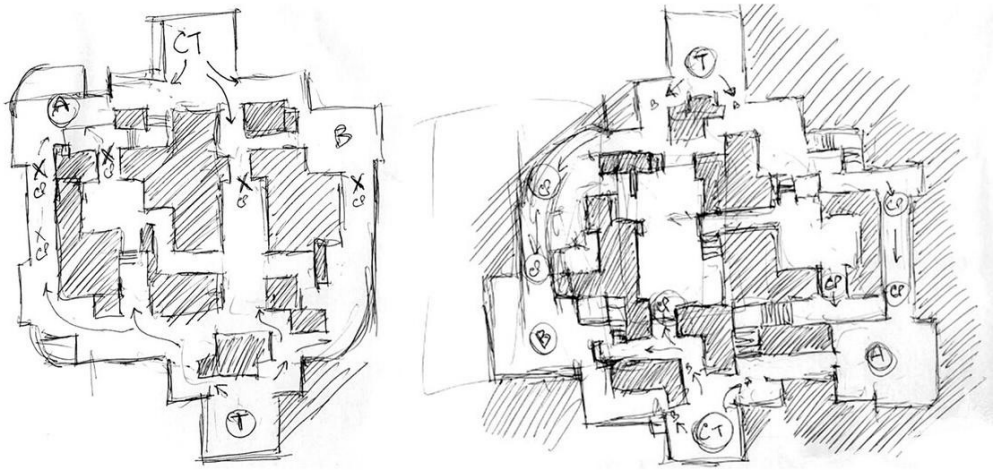


Рисунок 2.1 – Приклад ескізів рівнів

Метрики допомагають левел-дизайнерам створювати рівні за наперед визначеними параметрами, щоб він був зручним та інтуїтивно зрозумілим для гравців. Метрики включають розміри дверей, сходів, ширину коридорів, висоту уступів та інші характеристики, які повинні відповідати розміру персонажа та його фізичним можливостям.

Після того, як основні ідеї рівня та його розміри визначені, дизайнери переходять до створення макета рівня.

На цьому етапі не потрібна ідеальна графіка. Можна використовувати прості примітивні форми (куби, сфери) або спеціальні інструменти ігрових движків, наприклад Pro Builder в Unity, для швидкого будівництва рівнів.

Замість деталізованих моделей створюються базові об'єкти, які виконують свою функцію, але не вимагають багато часу на малювання.

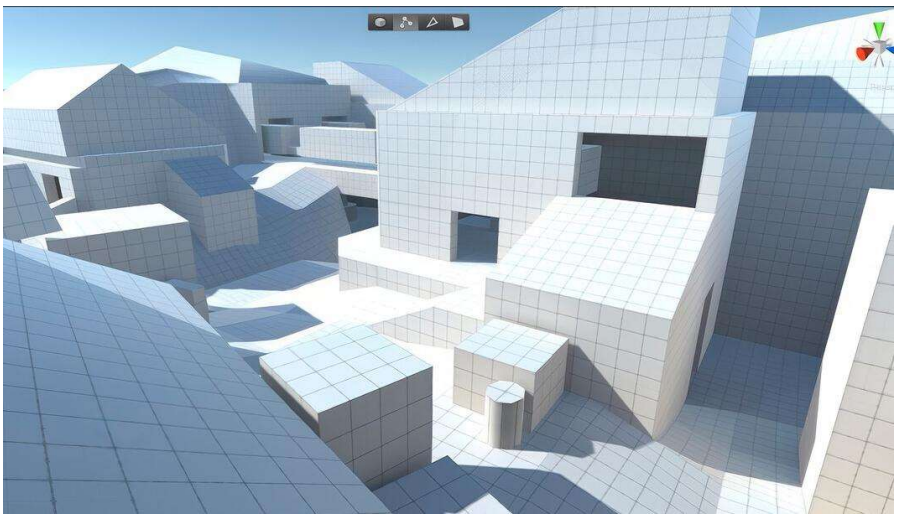


Рисунок 2.2 – Приклад рівня на етапі прототипування

Для створення порожнього 3D проекту зайдіть у Unity Hub та натисніть кнопку New project. Вибираємо кнопку 3D (Built-In Render Pipeline), вказуємо ім'я проекту та його місцезнаходження та натискаємо кнопку Create project. Коли новий проект ініціалізується, з'явиться редактор Unity.

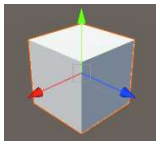
Інтерфейс Unity відображається як набір панелей. Кожна панель має вкладку ліворуч угорі, за яку панель можна перемістити і тим самим змінити макет програми. Також, переміщуючи панель за вкладку, її можна перетворити на самостійне вікно. Не всі панелі видимі за замовчуванням, і в міру розробки гри ви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

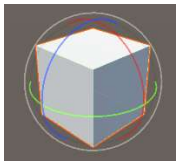
Подання сцени може бути в одному з п'яти різних режимів. Перемикач режимів, що знаходиться у вікні, керує режимом взаємодій з поданням сцени.

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

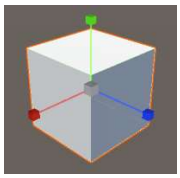
Режим переміщення  (Translation mode) – у цьому режимі можна переміщати виділені об'єкти.



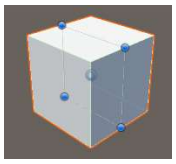
Режим обертання  (Rotation mode) – у цьому режимі можна повертати виділені об'єкти.



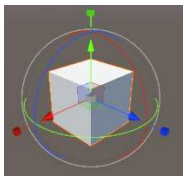
Режим масштабування  (Scale mode) – в цьому режимі можна змінювати розміри виділених об'єктів.



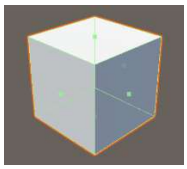
Режим прямокутника  (Rectangle mode) – у цьому режимі можна переміщати вибрані об'єкти та змінювати їх розміри, використовуючи двовимірні маркери. Це особливо зручно при формуванні двовимірних сцен або інтерфейсу користувача.



Наступний інструмент поєднує у собі функціональність всіх попередніх. За допомогою цього інструменту  ми можемо рухати об'єкт, збільшувати його розмір та обертати його.



Останнє  – це кастомні інструменти. Ці інструменти ми можемо встановити по-різному для кожного об'єкта. Для куба він ставатиме як Edit Box Collider.



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

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.

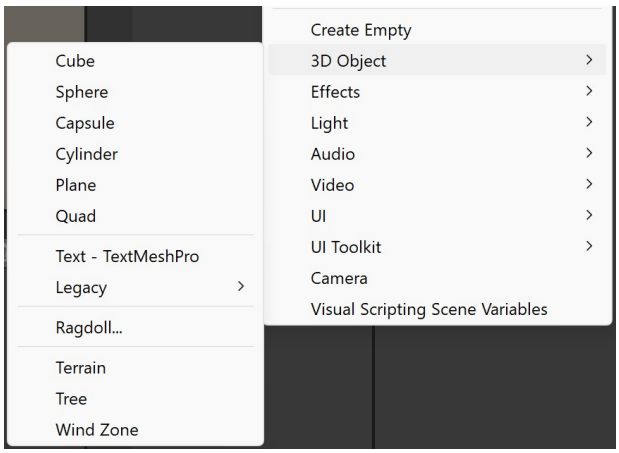


Рисунок 2.3 – Приклад GameObject

Unity надає набір базових геометричних форм, які можна створювати прямо в редакторі:

– Куб (Cube),
– Сфера (Sphere),
– Циліндр (Cylinder),
– Капсула (Capsule),
– Площина (Plane),
– Квад (Quad).

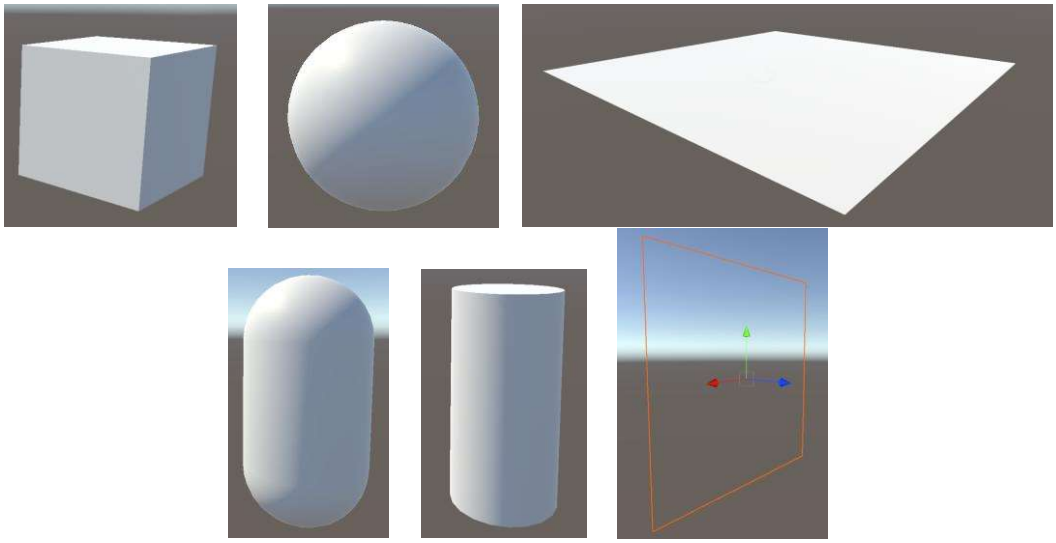


Рисунок 2.4 – Приклад базових 3D-об'єктів

Quad – це примітивний 3D-об'єкт, що є плоскою, двосторонньою поверхнею, створеною з двох трикутників, яка використовується для відображення зображень, текстур або створення простих елементів у 3D-сцені. На відміну від інших примітивів, таких як куби або сфери, Quad не має об'єму і служить лише як поверхня для чогось.

Ключові особливості Quad:

– Двостороння – на відміну від інших об'єктів, які можуть бути односторонніми, Quad має дві сторони, що робить його зручним для відображення зображень з обох сторін;
– Плотська геометрія – він є плоским об'єктом, що відрізняє його від більш об'ємних примітивів, таких як куб.

Ці об'єкти служать для швидкого прототипування, так і для створення базових елементів сцени, наприклад, площину можна використовувати як поверхню землі.

У Unity не можна було створювати тривимірні моделі. Тобто. нативних засобів та інструментів для редагування тривимірних моделей у Unity немає, є кілька вбудованих об'єктів (куб, сфера тощо). Вбудованими засобами редактора не можна виконувати такі маніпуляції, як додавання (переміщення, обертання, видалення) вершин, ребер, полігонів, не можна накладати текстури на окремі полігони, не можна створювати фаски та виконувати витягування полігонів.

Починаючи з версії Unity 2018, у редактор були інтегровані інструменти ProBuilder. *ProBuilder* – це унікальний набір засобів для побудови двовимірних та тривимірних геометричних форм з можливістю їх подальшого редагування та текстурування.

Щоб встановити ProBuilder зайдемо в меню Windows → Package Manager. В Unity Registry в полі знайти набрати ProBuilder.

Після встановлення в головному меню програми Unity з'явиться новий пункт Tools з одним підпунктом ProBuilder, навивши на який отримаємо список можливостей. Виберемо пункт ProBuilder Window, після чого на екрані з'явиться панель, що плаває, з інструментами. Панель з інструментами може знаходитися в двох режимах: Icon mode та Text mode. Щоб переключитися між режимами натискаємо праву кнопку миші на вільній ділянці панелі та в меню, що з’явиться обираємо необхідний режим.

Перш ніж почати малювати фігуру, скористайтеся інструментом, який значно полегшить налаштування її розміру пізніше: інструментом «Snap tool».

За замовчуванням інструмент «Snap tool» вимкнено. Щоб увімкнути його, виберіть значок «Увімкнути/вимкнути прив’язку сітки», а потім наведіть курсор на режим перегляду «Сцена», щоб перевірити, чи його ввімкнено. Ви побачите жовтий квадрат, який слідує за вашим курсором, який прив’язується до вершин сітки в режимі перегляду «Сцена».

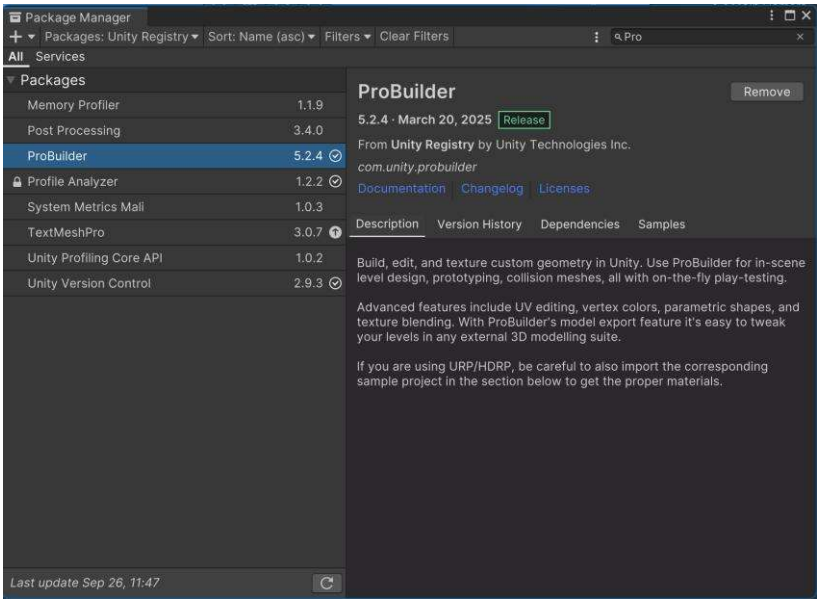


Рисунок 2.5 – Вікно Package Manager для установки ProBuilder

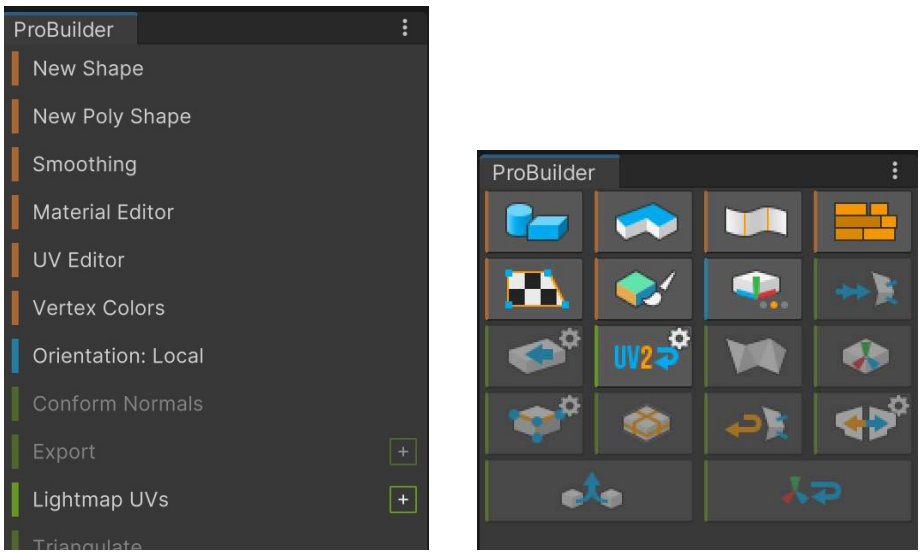
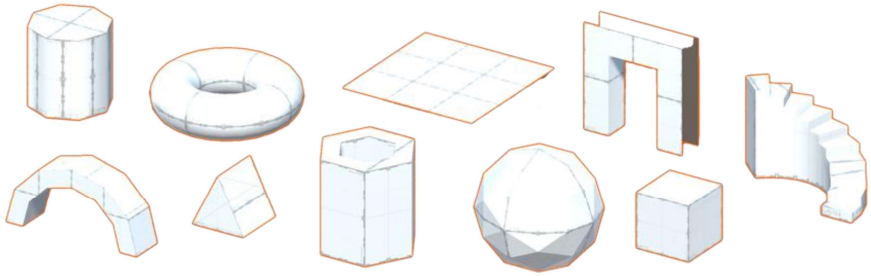
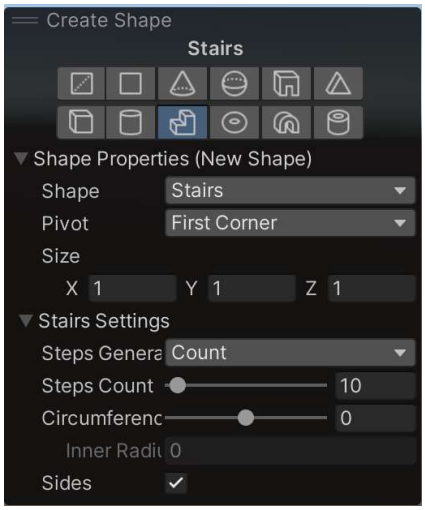


Рисунок 2.6 – Вікно ProBuilder з інструментами в режимі Text mode та Icon mode

Найпоширеніший підхід полягає у створенні попередньо визначеної фігури за допомогою інструмента «Shape tool»  , який містить бібліотеку фігур. Ці попередньо визначені фігури включають: прості куби, призми, тори та інші прості геометричні елементи, які можна використовувати для створення будівель, транспортних засобів та інших об'єктів. Попередньо визначені фігури, які зазвичай зустрічаються в будівлях, такі як сходи, арки та двері.

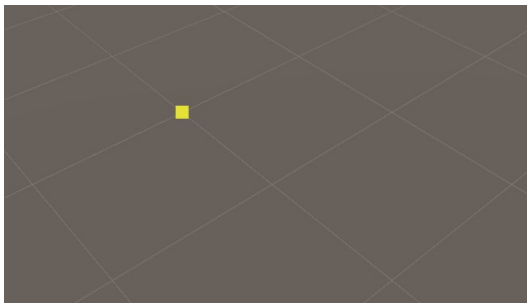


Щоб додати нову фігуру на сцену натискаємо на інструмент «Shape tool»  , з’явиться у вкладці Scene вікно в якому ми можемо обрати фігуру та зробити її налаштування.



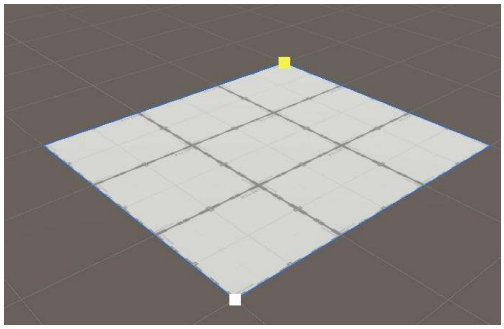
Перш ніж почати малювати фігуру, є інструмент, який значно спростить налаштування розміру фігури пізніше: інструмент «Snap tool».

За замовчуванням інструмент «Snap tool» вимкнено. Щоб його ввімкнути, виберіть значок «Увімкнути/вимкнути прив'язку сітки», а потім наведіть курсор на вікно перегляду «Scene», щоб перевірити, чи його ввімкнено. Ви побачите жовтий квадрат, який слідує за вашим курсором, який прив'язується до вершин сітки в вікні перегляду «Scene».

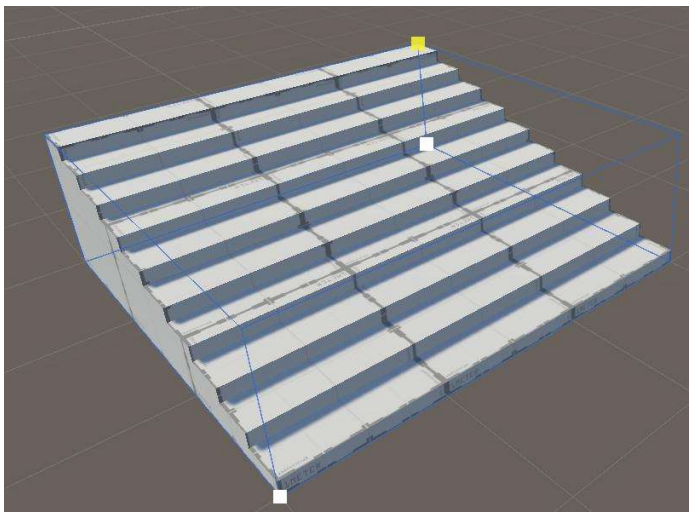


Щоб почати малювати фігури, виконайте такі інструкції:

1. Клацніть у вікні перегляду «Scene» точку, з якої ви хочете розмістити фігуру, а потім, утримуючи ліву кнопку миші, перетягніть курсор по сітці, щоб встановити ширину фігури. Нарешті, відпустіть ліву кнопку миші.



2. Для 3D-фігур виберіть один із кутів, виділених жовтим кольором, і перемістіть курсор вгору, утримуючи ліву кнопку миші, щоб встановити висоту фігури. Знову натисніть ліву кнопку миші, коли досягнете потрібного розміру.



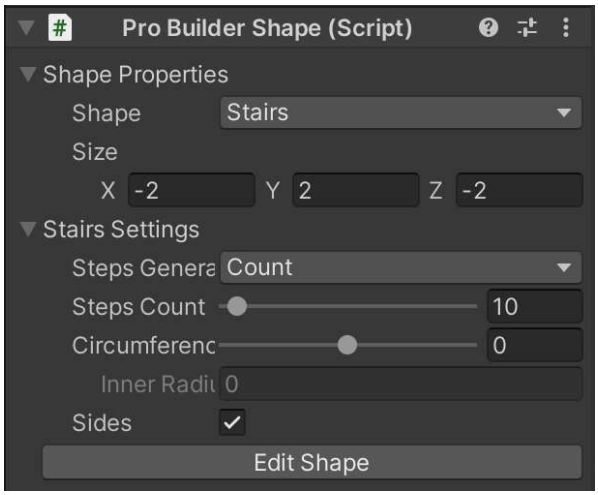
Після того, як ви намалювали фігуру, ви можете налаштувати додаткові властивості, щоб змінити її форму. Доступні властивості відрізняються залежно від вибраної вами початкової форми. Коли ви закінчите налаштування деталей, клацніть порожнє місце у вікні «Ієрархія», щоб завершити вибір та завершити створення сітки ProBuilder.

Вже завершені фігури не мають інтерфейсів обертання чи зміни розміру, а також не мають вікна «Створити фігуру», як показано раніше.

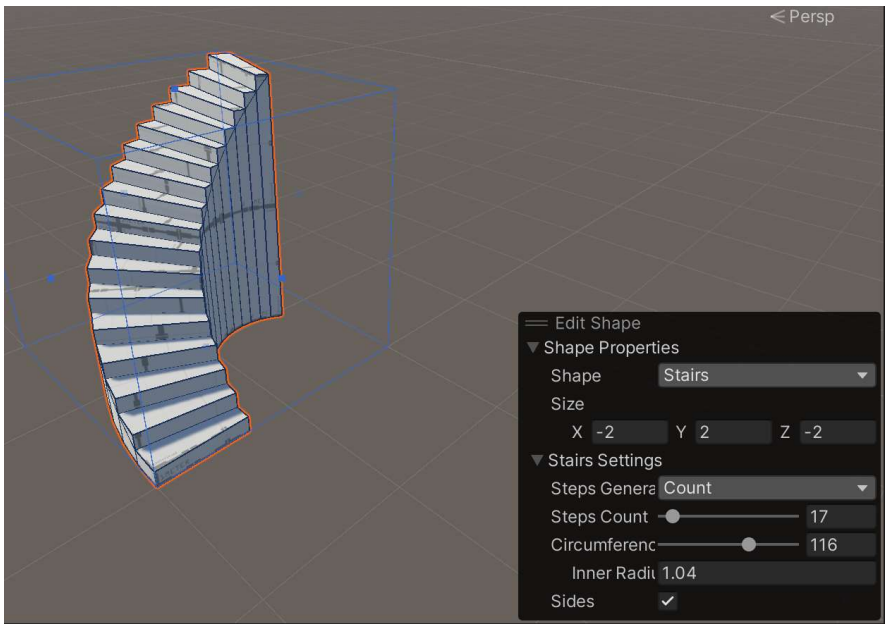
Щоб редагувати вже завершену фігуру, виконайте такі інструкції:

1. Виберіть компонент ProBuilder Shape у вікні Інспектора.
2. Натисніть кнопку «Edit Shape».

Відкриється вікно «Редагувати фігуру».



Ви можете налаштувати різні властивості готової фігури у вікні «Редагувати фігуру».



Коли вікно ProBuilder відкрито в Редакторі Unity, у верхній частині вікна Сцена з’являються чотири кнопки-перемикачі.

Вони поділені одиницями вибору (об'єкт, вершина, лінія та поверхня) для редагування 3D-моделі. Інші інструменти 3D-моделювання також мають подібні функції.

Гізми, які з'являються під час вибору GameObject, також з'являються під час вибору вершин, ліній або поверхонь. Це означає, що ви також можете вибирати та розташовувати вершини, лінії та поверхні.

Гізми відповідають основним інструментам сцени, тому використовуйте інструменти, щоб розташувати їх так, як вам потрібно. Для детального розташування вимкніть попередньо ввімкнену прив’язку або утримуйте клавішу Ctrl під час роботи, щоб тимчасово вимкнути її.

Більш детальну інформацію про роботу та редагування об’єктів можна знайти на:

1. https://learn.unity.com/tutorial/build-basic-shapes?language=en&projectId=6629dd69edbc2a0e64f34a4e#
2. Interacting with ProBuilder | ProBuilder | 5.0.7

2. Завдання до практичної роботи 2

Створіть 2 сцени в грі. В кожну сцену додай прототип рівня. В перший використовуючи тільки набір базових геометричних форм вбудованих в Unity. На другій сцені створить прототип рівня використовуючи можливості ProBuilder.

Тема 6. UI. Створення головного меню гри
Практична робота 3. Створення головного меню гри

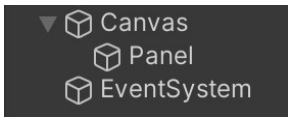
Мета роботи: познайомитися зі створення ігрового меню гри, з об’єктом Canvas, компонентами Panel та Button. Навчитися створювати головне меню гри.

1. Теоретичний матеріал

Ігрове меню – це інтерфейс, який дозволяє керувати грою, роблячи доступними такі функції, як налаштування, запуск, пауза або вихід з гри, а також вибір режимів гри, керування персонажем та інші опції. Воно може бути основним (головним), коли гра тільки запущена, або меню паузи, яке з'являється під час ігрового процесу, щоб дати гравцеві можливість вибрати, продовжити гру, змінити налаштування або вийти.

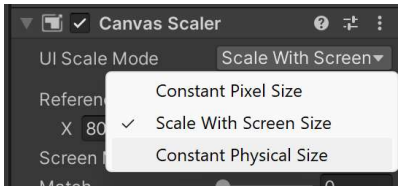
Для створення головного меню гри створимо нову сцену в проекті.

Додаймо на сцену Canvas (Полотно). Полотно – це ігровий об'єкт з доданим до нього компонентом Canvas. Всі елементи UI повинні бути дочірніми цьому Canvas.



В поле Render Mode об’єкту Canvas встановлюємо значення Screen Space – Overlay. Цей режим відображення поміщає елементи інтерфейсу на екран поверх сцени. Якщо змінюється розмір екрана або його роздільна здатність, полотно автоматично прийме потрібний розмір разом із ним.

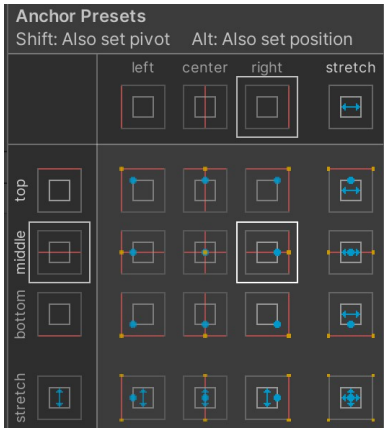
За замовчанням в компоненті Canvas Scaler в полі UI Scale Mode стоїть значення Constant Pixel Size. Якщо зміниться розмір екрана, то всі елементи що входять в Canvas не змінять свого розміру. Замінимо це значення на Scale with Screen Size.



Додаймо в Canvas компонент Panel. Panel це прямокутний контейнер із Image. Назвемо його MainMenu. Налаштуємо розмір компоненти, зробимо його не на весь екран. В компонент Image в поле Source image додаємо картинку фону.

Рисунок 3.1 – Приклад фону головного меню

Змінимо прив’язку панелі MainMenu відносно Canvas. Перетягнемо панель ближче до правої межі Canvas та виберемо Anchor Presets в значення middle-right.

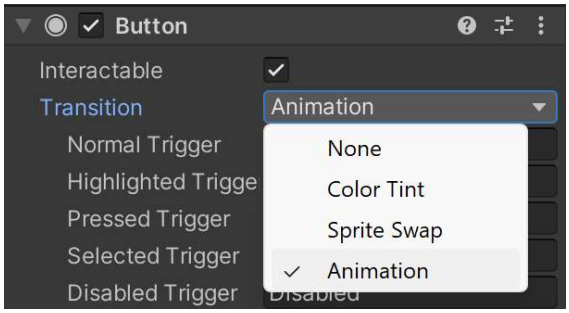


Створюємо дочірнім об’єктом до панелі MainMenu компонент Button. Компонент Button складається з компонента Image та Text. Додаємо до компонента Image зображення для представлення кнопки, в компоненту Text – запишемо текст Start та змінимо колір тексту та зробимо інші налаштунки. Змінимо назву кнопки в Canvas на StartButton.

Рисунок 3.2 – Приклад кнопки в головному меню

Щоб головне виглядало цікавіше додаймо анімацію при наведенні курсору на кнопку.

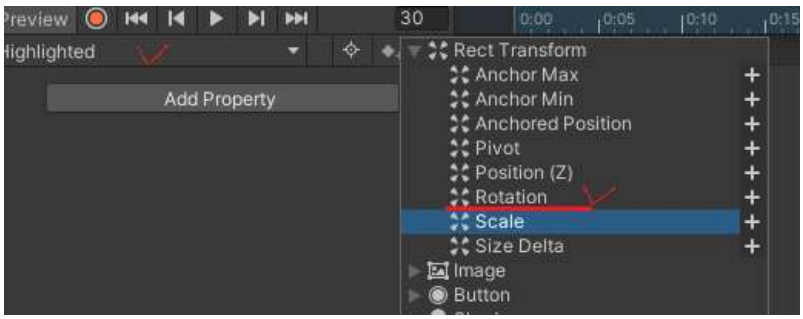
Поле Transition – це переходи для відтворення кнопки в різних станах. Може мати три варіанти: Color Tint, Sprite Swap та Animation. За замовчанням в компоненті Button в полі Trasition стоїть значення Color Tint. Замінимо це значення на Animation.



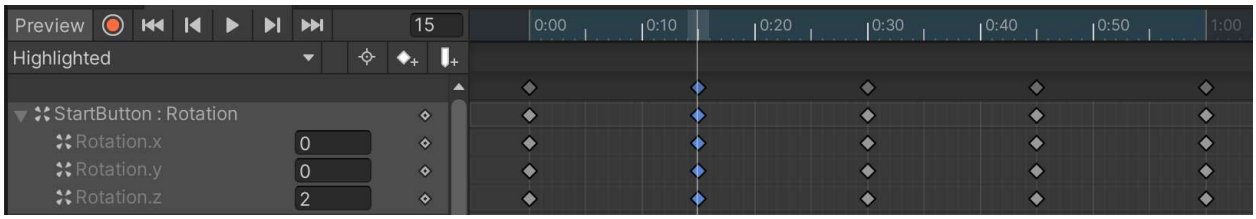
При значення Animation натискаємо кнопку Auto Generate Animation та зберігаємо файл анімації до нової папки Animations. Якщо натиснутий на створений контроллер, він відкриється у вікні Animator.

Додаймо анімацію під час наведення на кнопку мишкою. Нехай трохи похитуватиметься. Виділяємо кнопку та відкриваємо віконце анімації Window → Animation → Animation.

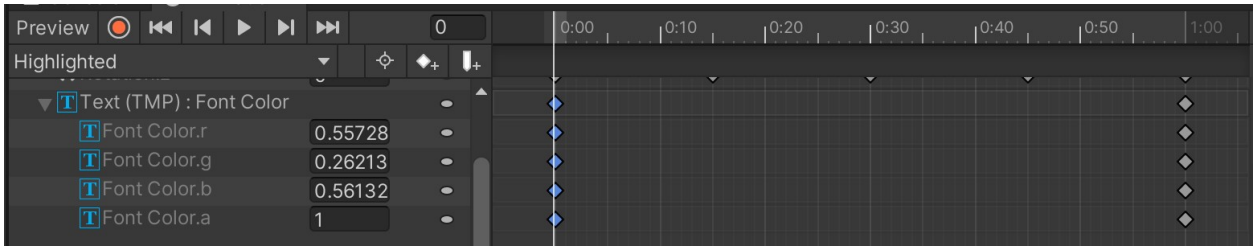
Обираємо Highlighted, натискаємо кнопку Add Property та обираємо Rect Transform → Rotation.



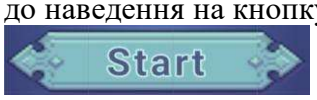
Розставляємо 5 keyframes, за часом 0:00, 0:15, 0:30, 0:45, 1:00. На час 0:00, 0:30 та 1:00 залишаємо Rotation.x, Rotation.y, Rotation.z рівними нулю, а на час 0:15 напишемо Rotation.z = 2, на час 0:45 напишемо Rotation.z = -2.



Також можна додати анімацію на колір тексту. В кнопці встановлюємо колір на текст, а у вікні анімації будемо його змінювати:



до наведення на кнопку мишкою



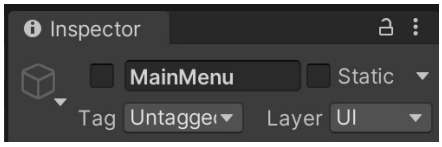
після наведення

Зробимо 2 копії кнопки, що вийшла та розставимо по вертикалі на панелі. Змінимо текст на кнопках. В результаті отримаємо:

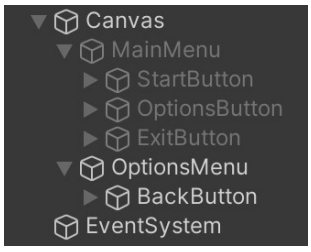
Рисунок 3.3 – Приклад початкового варіанта головного меню

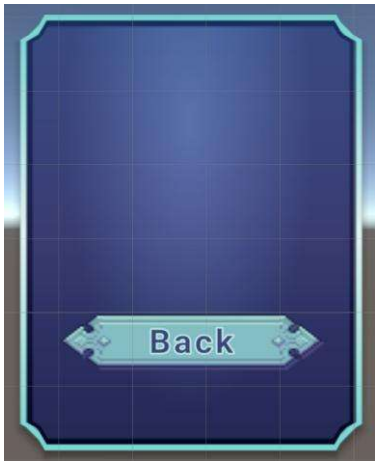
Отже маємо три кнопки: Start – при натисканні якої повинна починатися гра, Options – при натисканні якої повинна з’являтися додаткове меню для налаштувань та Exit – для виходу з програми.

Створимо додаткове меню для налаштувань. В нашій версії воно буде мати тільки одну кнопку Back. Для цього виділяємо елемент MainMenu в ієрархії та робимо його дублікат. Змінюємо ім’я дубліката на OptionsMenu. Для зручності тимчасово зробимо елемент MainMenu не активним.



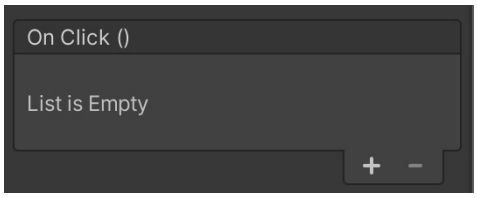
Якщо в ієрархії розкрити елемент OptionsMenu, то він має у підпорядкуванні три кнопки. Видалимо перші дві, а в останній кнопці змінимо текст на Back:

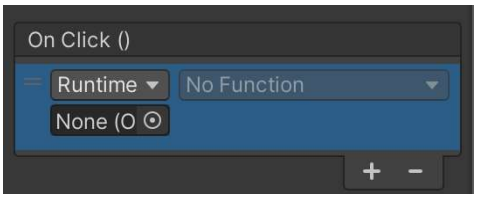




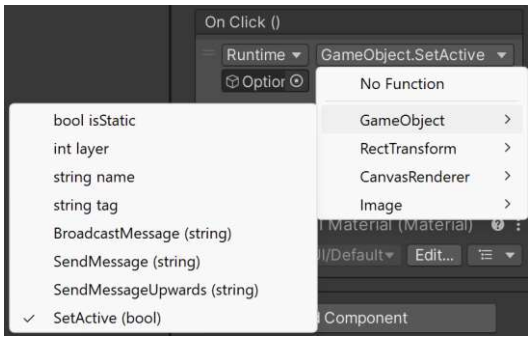
Отже, додаткове меню для налаштувань готово. Необхідно додати функціональність оскільки при натисканні кнопок нічого не відбувається.

Додаймо функціонал при натисканні кнопки Back, а саме елемент OptionsMenu повинен стати не активним, а елемент MainMenu активним. Для цього виберіть із компонента Button розділ «On Click()» і натисніть на плюс.



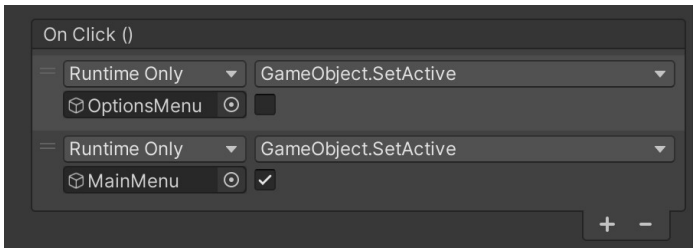


У нижне вікно перетягуємо об’єкт OptionsMenu, тоді справа в полі відкриються доступні функції над цим об’єктом. Обираємо GameObject → SetActive(bool)

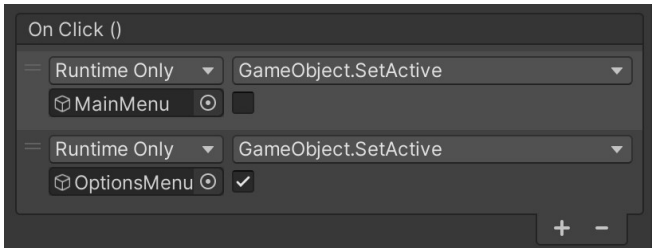


Рядом з полем з об’єктом OptionsMenu створиться прапорець. Якщо обрати прапорець, то вікно буде активно (видимо), якщо ні – не активно. Для об’єкта OptionsMenu прапорець не обираємо.

В розділі «On Click()» можна створювати декілька дій одразу. В нашому випадку нам потрібно ще одна дія для показу вікна MainMenu. Зробимо цю дію аналогічно попердій. Тоді в розділі «On Click()» будемо мати наступне:



Повертаємось до головного вікна MainMenu. Виділяємо елемент OptionsMenu в ієрархії та робимо його не активним. Виділяємо елемент MainMenu в ієрархії та робимо його активним. Пропишемо схожі дії для кнопки OptionsButton.



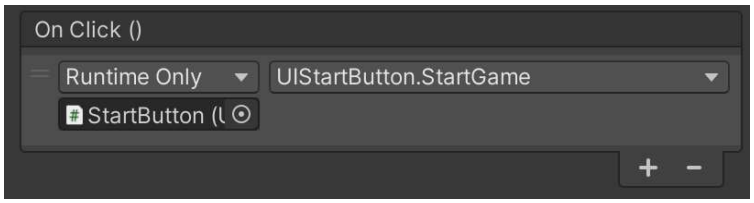
Якщо запустити проект, при натисканні кнопки Options з’явиться додаткове меню з однією кнопкою Back при натисканні на яку, повертається головне меню.

Залишається додати функціональність до кнопки Start. Створимо новий скрипт, наприклад UIStartButton, який має тільки один метод:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class UIStartButton : MonoBehaviour
{
    public void StartGame()
    {
        SceneManager.LoadScene("Scene1");
    }
}
```

В метод LoadScene необхідно передати назву сцени з грою.

Додіммо цей скрипт до кнопки Start. Переходимо в розділ розділі «On Click()» кнопки Start. У нижне вікно перетягуємо кнопку Start тому що до неї прикреплен скрипт з необхідною функцією. В поле функцій обираємо UIStartButton → StartGame().



Тепер при натисканні на кнопку Start програма перейде та запустить сцену, яку прописали в функції LoadScene().

2. Завдання до практичної роботи 3

Створіть сцену та розробити головне меню програми з можливістю переходу до додаткового меню та іншої сцени. Додайте анімацію до кнопок головного меню.

Тема 8. UI. Вікно інвентаря. Зберігання предметів
Практична робота 4. Створення вікна інвентаря

Мета роботи: познайомитися зі створенням вікна інвентаря в іграх, з компонентом Grid Layout Group, з інтерфейсами IDragHandler, IEndDragHandler та методи OnDrag, OnEndDrag. Навчитися створювати вікна інвентаря в іграх.

1. Теоретичний матеріал

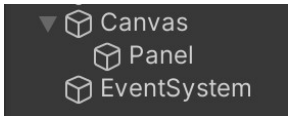
Інвентар – це спливаюче меню, яке гравець використовує для керування предметами, які він має при собі.

Вікно інвентаря в іграх – це екран, який показує предмети, що їх має персонаж, розділяючи їх на те, що він носить, і те, що зберігає в рюкзаку.

Рисунок 4.1 – Приклад вікна інвентаря в іграх

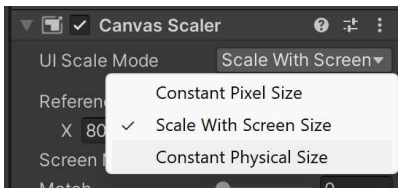
Для створення вікна інвентаря гри створимо на сцені гри новий Canvas.

Додаймо на сцену Canvas (Полотно). Полотно – це ігровий об'єкт з доданим до нього компонентом Canvas. Всі елементи вікна інвентаря повинні бути дочірніми цьому Canvas. Додаймо в Canvas компонент Panel.



В поле Render Mode об’єкту Canvas встановлюємо значення Screen Space – Overlay. Цей режим відображення поміщає елементи інтерфейсу на екран поверх сцени. Якщо змінюється розмір екрана або його роздільна здатність, полотно автоматично прийме потрібний розмір разом із ним.

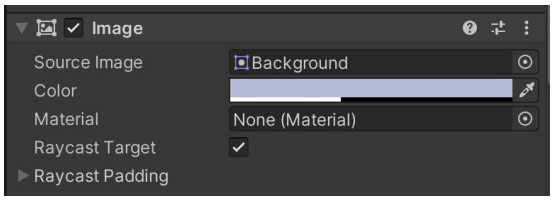
За замовчанням в компоненті Canvas Scaler в полі UI Scale Mode стоїть значення Constant Pixel Size. Якщо зміниться розмір екрана, то всі елементи що входять в Canvas не змінять свого розміру. Замінимо це значення на Scale with Screen Size.



Panel це прямокутний контейнер із Image. Назвемо його InventiryWindow. Налаштуємо розмір компоненти, зробимо його не на весь екран. В компонент Image в поле Source image додаємо картинку фону.

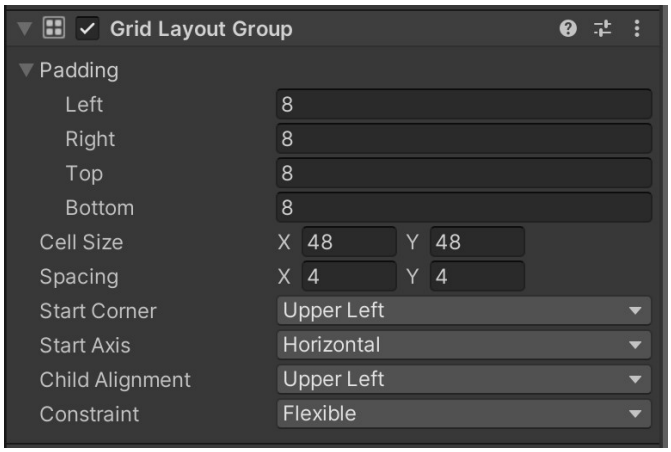
Рисунок 4.2 – Приклад фону вікна інвентаря

Створюємо дочірнім об’єктом до панелі InventiryWindow компонент Panel та назвемо його ContentView. Саме цю панель будемо розбивати на осередки, в яких будемо показувати картинки об’єктів які підібрали у грі. В компоненті Image змінимо колір фону та зробимо його частково прозорим.

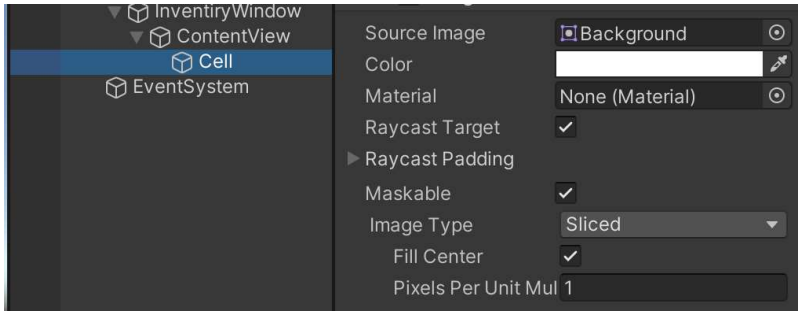


Додаймо до панелі ContentView компонент Grid Layout Group.

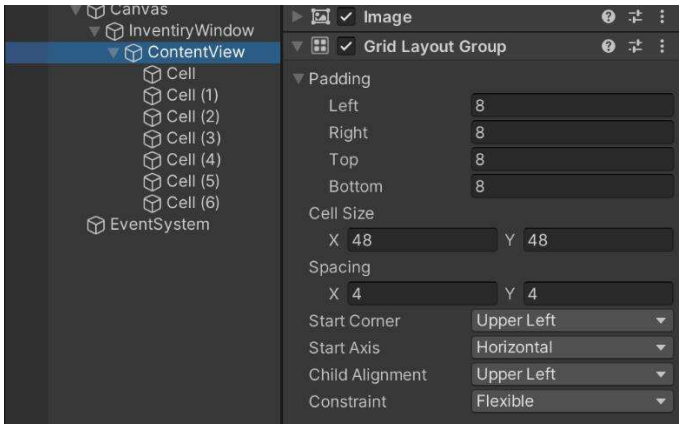
Grid Layout Group в Unity – це компонент, який використовується для автоматичного розміщення дочірніх компонентів в рівномірну мережу, де всі клітинки мають однаковий розмір. Розмір та міжклітинкова різниця контролюються компонентом Grid Layout Group самостійно, а дочірні компоненти не впливають на їхні розміри.

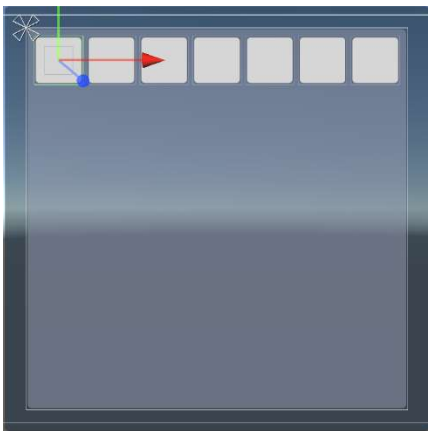


Створюємо дочірнім об’єктом до панелі ContentView компонент Image та назвемо його Cell.

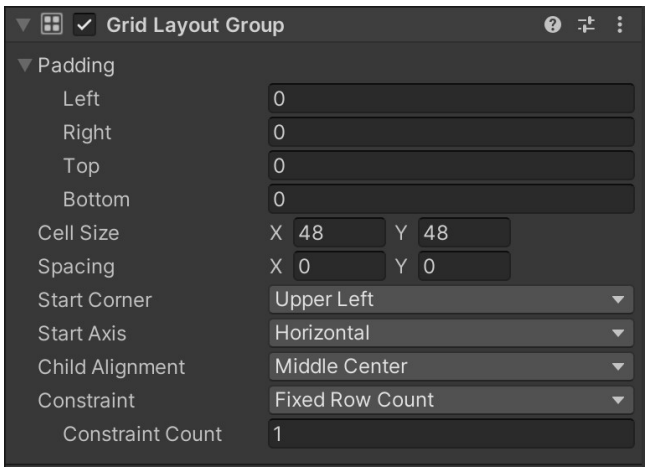


Підберемо значення Padding, Cell Size та Spacing в компоненті Grid Layout Group, щоб в панелі рівномірно розмістилось ціле значення осередків. Для цього тимчасово створіть декілька об’єктів Cell.

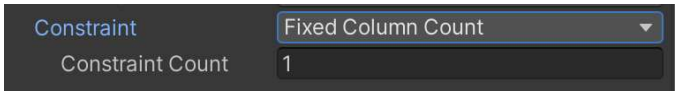




У подальшому до Cell будемо додавати картинку предмету, тому додаймо до об’єкту Cell компонент Grid Layout Group для найкращого розміщення картинки. В компоненті Grid Layout Group зробимо такі налаштування:



Також перевіримо, щоб во властивості Constraint при виборі значення Fixed Column Count було значення 1.



Для надання можливості користувачеві закрити вікно інвентаря, створюємо дочірнім об'єктом до панелі InventiryWindow компонент Button. Компонент Button складається з компонента Image та Text. Додаємо в компоненту Text текст «X» та змінимо колір тексту та зробимо інші налаштунки. Змінимо назву кнопки в Canvas на ButtonClose.

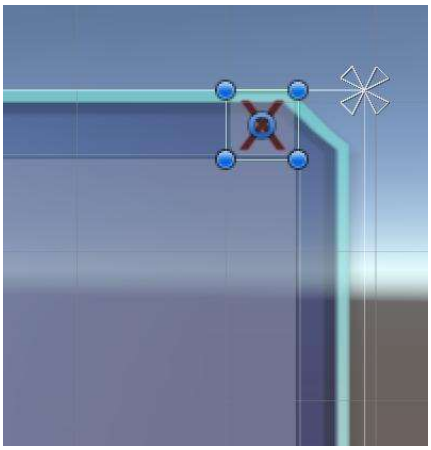
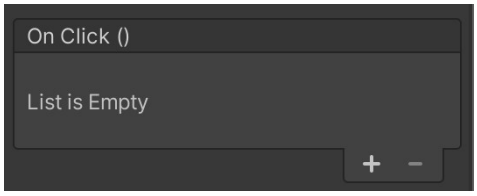
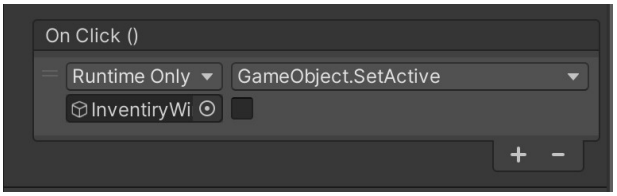


Рисунок 4.3 – Приклад кнопки закриття вікна інвентарю

Додаймо функціонал при натисканні кнопки ButtonClose, а саме елемент InventiryWindow повинен стати не активним. Для цього виберіть із компонента Button розділ «On Click()» і натисніть на плюс.





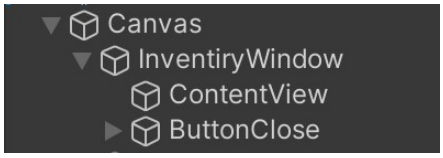
У нижне вікно перетягуємо об'єкт InventiryWindow, тоді справа в полі відкриються доступні функції над цим об'єктом. Обираємо GameObject → SetActive(bool). Рядом з полем з об'єктом InventiryWindow створиться прапорець. Якщо обрати прапорець, то вікно буде

активно (видимо), якщо ні – не активно. Для об’єкта InventiryWindow прапорець не обираємо.

Якщо запустити проект, при натисканні кнопки «X» вікно інвентарю закриється.

Залишається додати функціональність. Створимо нові скрипти.

Створюємо скрипт Cell та додаємо його як компонент до об’єкту Cell. Тепер об’єкт Cell не потрібен на сцені, тож зробимо з нього префаб та видалимо зі сцени. Створимо в Assets папку Resources. Тут будуть зберігатися prefabs, тих об'єктів, які необхідно буде створювати динамічна, тобто в процесі гри.



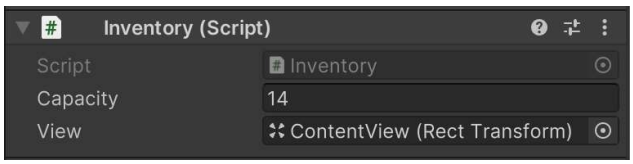
Створюємо скрипт Inventory та додаємо його як компонент до об’єкту InventiryWindow. Для створення необхідної кількості осередків у ContentView створимо поле capacity та поле view для посилання на ContentView. Масив content зберігатиме створені осередки.

```
[SerializeField]
private int capacity;

[SerializeField]
private Transform view;

public static Cell[] content;
```

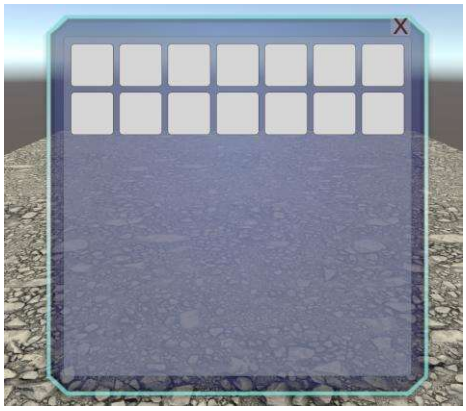
Створимо осередки та заповнимо масив content:

```
void Awake()
{
    content = new Cell[capacity];
    CreateCells();
}


void CreateCells()
{
    for (int i = 0; i < capacity; i++)
    {
        GameObject cell = Instantiate(Resources.Load<GameObject>("Cell"));

        cell.transform.SetParent(view);
        cell.transform.localScale = Vector2.one;
        cell.name = string.Format("Cell [{0}]", i);

        content[i] = cell.GetComponent<Cell>();

    }
}
```

В Inspector заповнюємо поля певними значеннями, наприклад, створимо 14 осередків:



Якщо запустити гру, маємо вікно інвентаря з 14 пустими осередками:



Додаймо можливість показувати в осередках зображення предметів. Нехай маємо такі зображення:

Створимо в Assets папку Sprites та додаймо ці зображення (або якісь інші) до цієї папки. Виділить в закладці Project перше зображення та в панелі Inspector в поле Texture Type вибрати значення Sprite (2D and UI) та зберегти зміни. Тип Sprite (2D and UI) форматує ресурс таким чином, щоб його можна було використовувати в 3D-додатках як спрайт.

Створимо в панелі ContentView тимчасовий об’єкт типу Image. На панелі Inspector в компоненте Image в поле Source Image перетягуємо, наприклад, перше зображення з папки Sprites. Створимо скрипт Item та приєднуємо його до створеного зображення типу Image. Збережемо об'єкт, що вийшов, як префаб, наприклад, в папці Resources. Перейменуємо, наприклад, в Poison01.

В папці Resources створимо ще дві копії префаба Poison01, назвемо їх Poison02 та Poison03. В компоненті Image в поле Source Image для нових префабів встановимо інші картинки.

В скрипте Item створимо такі поля та їх ініціалізацію:

```
[HideInInspector]
public Cell cell;
private Transform canvas;

void Start()
{
    cell = GetComponentInParent<Cell>();
    canvas = GameObject.Find("Canvas").transform;
}
```

Необхідно додати можливість користувачеві перетягувати картинки у вікні інвентаря між осередками. Для цього зробимо клас Item, що реалізує два інтерфейси IDragHandler, IEndDragHandler та реалізуємо два методи: OnDrag та OnEndDrag.

```
public class Item : MonoBehaviour, IDragHandler, IEndDragHandler
    ...

public void OnDrag(PointerEventData eventData)
{
    transform.SetParent(canvas);
    transform.position = eventData.position;
}
public void OnEndDrag(PointerEventData eventData)
{
    float distance = float.MaxValue;
    Cell newCell = cell;

    for (int i = 0; i < Inventory.content.Length; i++)
    {
        float temp = Vector2.Distance(transform.position, Inventory.content[i].transform.position);

        if (temp < distance)
        {
            if (Inventory.content[i].transform.childCount > 0)
                continue;

            distance = temp;
            newCell = Inventory.content[i];
        }
    }

    cell = newCell;

    transform.SetParent(cell.transform);
    transform.position = cell.transform.position;
    transform.localScale = Vector2.one;
}
```

Методи OnDrag та OnEndDrag реалізують переміщення картинок по осередкам. Якщо для картинки вказати не конкретний осередок, то програма знайде найближчий не зайнятий та перенесе картинку до нього.

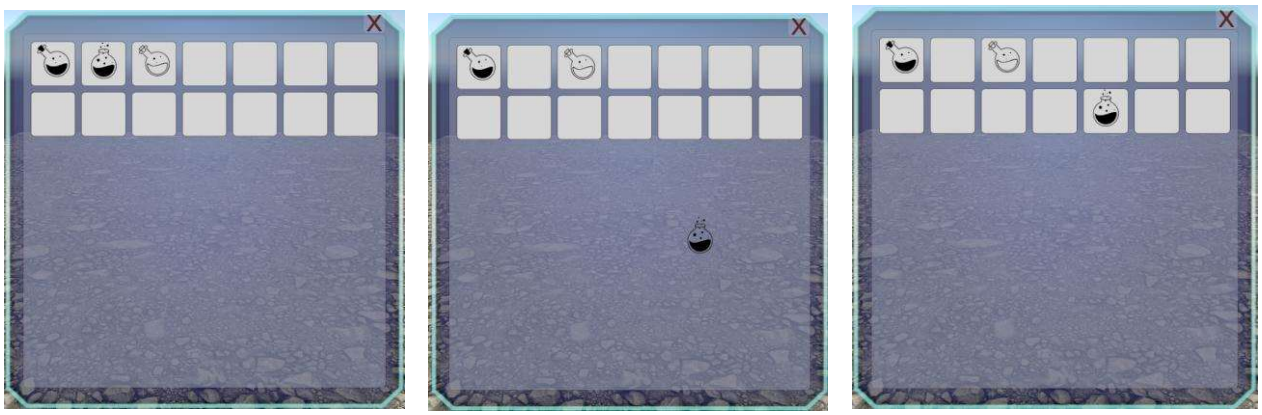


Рисунок 4.4 – Приклад вигляду вікна інвентарю при переміщенні картинок

Додаймо до сцени площину та персонаж зі скриптом, який керує його переміщенням.

Створимо на сцені декілька об’єктів яких персонаж буде підбирати та зберігати у вікні інвентаря. Для простоти візьмемо в якості таких об’єктів сфери. Створимо скрипт ItemObejct та приєднаємо до сфер.

Рисунок 4.5 – Приклад вигляду сцени

Розглянемо структуру скрипту ItemObejct.

Створимо поле для зберігання картинки з префабів, яка додається до вікна інвентарю у випадку підбирання об’єкту персонажем.

```
public GameObject item;
```

В скрипті Inventory створити метод, який додає картинку до інвентарю:

```
public static bool AddItem(GameObject item)
{
    for (int i = 0; i < content.Length; i++)
    {
        if (content[i].transform.childCount == 0)
        {
            GameObject newItem = Instantiate(item);

            newItem.transform.SetParent(content[i].transform);
            newItem.transform.localScale = Vector3.one;
            newItem.transform.localPosition = Vector3.zero;
            newItem.transform.position = newItem.transform.parent.position;
            newItem.GetComponent<Item>().cell = content[i];

            return true;
        }
    }

    return false;
}
```

Метод AddItem получає на вході об’єкт який необхідно додати. В методі шукається пустий осередок у вікні інвентарю та коли такий знаходиться до нього додається картинка.

В скрипті ItemObejct створимо метод який визиває метод AddItem:

```
void Update()
{
    CheckPickUp();
}

void CheckPickUp()
{
    if (Vector3.Distance(transform.position, MoveHelper.player.position) < 1.0F)
    {
        //Debug.Log("Near");
        if (Inventory.AddItem(item))
        {

            Destroy(gameObject);
        }

    }
}
```

MoveHelper – це скрипт який навішаний на персонажа і в якому поле player зберігає посилання на персонаж на сцені.

Коли персонаж підбере об’єкт на сцені, до інвентаря додається префаб картинки зарезервований за цим об’єктом. У подальшому об’єкт на сцені прибирається.

2. Завдання до практичної роботи 4

Створити сцену, додати персонаж та скрипти для його управління. Створити вікно інвентаря. Створити на сцені об’єкти, які може підібрати персонаж та поміщення їх до вікна інвентарю.

Тема 10. UI. HUD та вікно перемоги
Практична робота 5. Створення HUD та вікна перемоги

Мета роботи: познайомитися з HUD в іграх. Навчитися створювати HUD в іграх, зберігати інформацію про кількість очок або грошей, які має користувач, створювати вікна перемоги.

1. Теоретичний матеріал

В іграх існує величезна кількість інтерфейсів: інвентар, діалоги, меню крафту та торгівлі, лобі, карта, дерева прокачування персонажа та його екіпірування та багато інших. Всі вони дозволяють гравцям взаємодіяти із представленими через інтерфейс механіками, які творці гри заклали у свій продукт.

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

Основні елементи HUD:

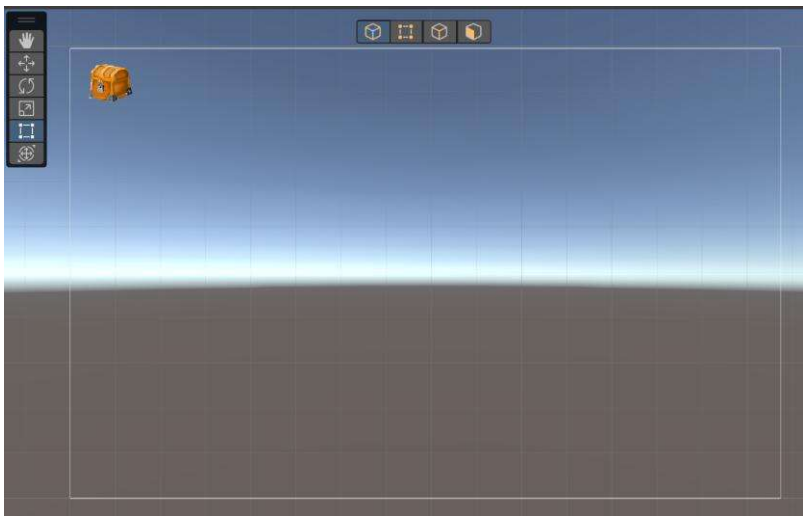
- Індикатори здоров'я – показують стан здоров'я гравця, часто у вигляді шкали чи чисел. При отриманні пошкоджень індикатор може змінювати колір або мигати, щоб вказати на небезпеку;
- Кількість боєприпасів – відображає, скільки набоїв залишилося у зброї, а також типи доступних боєприпасів;
- Рівень та досвід – наприклад у RPG-іграх, HUD може показувати рівень персонажа та кількість досвіду, необхідного для підвищення рівня;
- Прогрес у грі – інформація про поточний рівень, завдання, кількість очок або грошей, а також відсоток завершення завдань;
- Інвентар – деякі ігри мають панель, що дозволяє швидко використовувати предмети, такі як медичні набори або зілля.

На попередній практичній роботі розглядали створення вікна інвентарю. Вікно інвентарю показується не весь час на екрані, а з’являється при потребі користувача. Доповнимо нашу програму кнопкою для виклика вікна інвентарю.

По-перше, вимкнемо вікно інвентарю. Для цього в Canvas зробимо неактивним панель InventiryWindow.

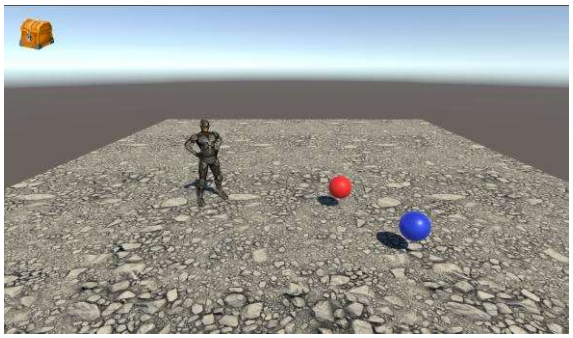
Створимо у Canvas кнопку, наприклад з назвою OpenInventory. В компоненті Image в поле Source Image додаймо зображення, наприклад якоїсь скрині. В компоненті Text (TMP) видалимо надпис.

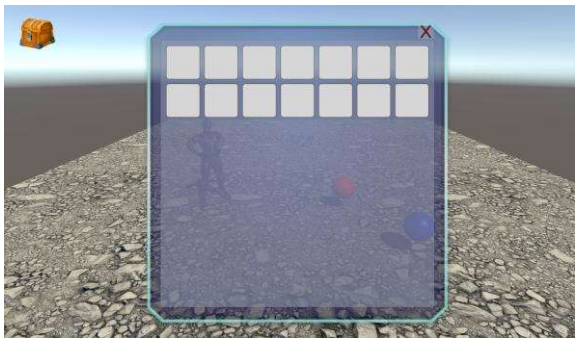
Прикріпимо кнопку до лівого верхнього куту Canvas:



Наприкінці, обробимо подію натискання на кнопку. А саме повинно з’явитися вікно інвентарю, тобто воно повинно стати активним.

Якщо запустити проект, то при натисканні кнопки з скринею з’явиться вікно інвентарю:



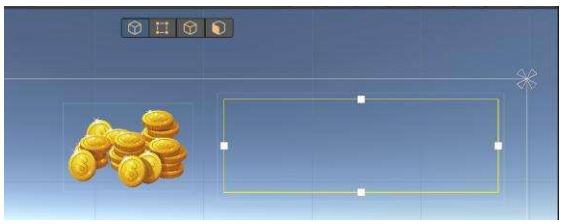


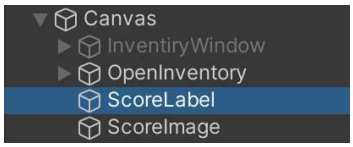
Додаймо у наш інтерфейс можливість бачити прогрес у грі. В якості процесу у грі будемо розуміти інформацію про кількість очок або грошей, які має користувач. Інформація складатиметься з двох частин: картинки, що показує тип інформації та сама кількість. Позиціонувати цю інформацію будемо у правому верхньому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

Створимо у Canvas картинку та в компоненті Image в поле Source Image додаймо зображення, наприклад купки монет та назвемо ScoreImage.

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

Нові елементи позиціонуємо до правого верхнього кутку.





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

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

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

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

```
[SerializeField]
private TMP_Text scoreLabel;

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

Далі нам потрібен скрипт який буде керувати подіями на сцені. Для цього створюємо пустий об'єкт на сцені назвемо його GameController. Створимо скрипт з такою ж назвою та прикріпим його до цього об'єкту.

В скрипті GameController створимо посилання на екземпляр класу:

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

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

Створимо поле score, в якому зберігатися поточна кількість очок або грошей.

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

В методі Start() ініціалізуємо цю змінну та визиваємо метод для показу поточної кількості в HUD, а також створюємо метод SetScore() для зміни значення.

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

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

}
```

В скрипті ItemObejct в метод CheckPickUp() додаємо рядок передачі кількості очок до GameController:

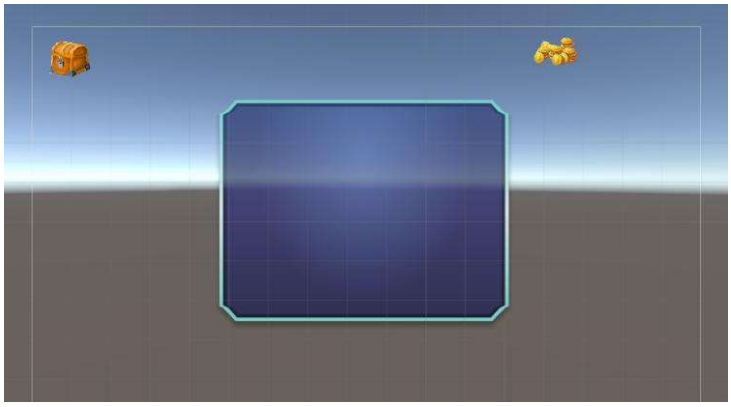
```
void CheckPickUp()
{
    if (Vector3.Distance(transform.position, MoveHelper.player.position) < 1.0F)
    {
        //Debug.Log("Near");
        if (Inventory.AddItem(item))
        {
            GameController.Instance.SetScore(5);
            Destroy(gameObject);
        }

    }
}
```

В нашому випадку коли персонаж підбере об’єкт на сцені, до інвентаря додається префаб картинки зарезервований за цим об’єктом, поточна кількість очок збільшиться на 5. У подальшому об’єкт на сцені прибирається. Можна ускладнити програму, зробивши для кожного об’єкту на сцені свою кількість балів які отримає користувач, якщо підбере їх на сцені.

Нехай коли користувач накопить певну кількість очок гра припиняється і з’являється вікно перемоги. Розглянемо кроки як реалізувати таку задачу.

По-перше створимо вікно перемоги.

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



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

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

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

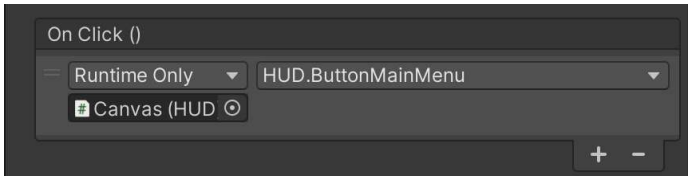
```
[SerializeField]
private GameObject LevelWonWindow;


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

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

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

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

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

}
```

У подальшому в скрипт GameController можна ввести поле для зберігання кількості очок при якому завершується гра та замінити число 10 на цю змінну.

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



Запустимо проект, підберемо два об’єкти та бачимо вікно виграшу.

2. Завдання до практичної роботи 5

До попереднього проекту додати HUD та створити вікно перемоги чи вікно закінчення гри.

Тема 12. Зберігання даних
Практична робота 6. Зберігання даних в іграх

Мета роботи: познайомитися зі формати для зберігання даних XML та JSON. Навчитися створювати та працювати з файлами даних формату XML та JSON.

1. Теоретичний матеріал

XML з'явився ще 1996 року і вперше був опублікований 1998 року. WWW Consortium (W3C) є розробником XML, і це стало Рекомендація W3C у 1998 році. XML виглядає дуже схожим на HTML, але XML є структурованішою мовою.

Документи у форматах HTML і XML містять дані, укладені в теги, але подібність між двома мовами закінчується. У форматі HTML теги *визначають оформлення даних* – розташування заголовків, початок абзацу тощо. У форматі XML теги *визначають структуру та зміст даних* – те, чим вони є.

Можна використовувати одну систему для генерації даних та позначки їх тегами у форматі XML, а потім обробляти ці дані в будь-яких інших системах незалежно від клієнтської платформи або операційної системи. Завдяки такій сумісності XML є основою однієї з найпопулярніших *технологій обміну даними*.

Правила XML дозволяють створювати будь-які теги, необхідні опису даних та його структури. Припустимо, що вам необхідно зберігати та спільно використовувати відомості про домашніх тварин. Для цього можна створити наступний XML-код:

```
<?xml version="1.0"?>
<CAT>
    <NAME>Izzy</NAME>
    <BREED>Siamese</BREED>
    <AGE>6</AGE>
    <OWNER>Colin Wilcox</OWNER>
</CAT>
```

Базовый синтаксис XML:

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<root>
    <child>
        <subchild>.....</subchild>
    </child>
</root>
```

Для перевірки XML-файлів можна скористатися онлайн-додатком, наприклад https://www.freeformatter.com/xml-validator-xsd.html.

Розглянемо приклад створення XML-файлу з результатами гри. Наприклад, в грі в інтерфейсі показується кількість життя гравця у процентах, сума накопичених грошей, кількість підібраних артефактів та назви артефактів. Розробимо функції для створення файлу з такими показниками та отримання даних з файлу.

Створення XML-файлу з результатами гри:

```
static void CreateXMLFile(string xmlName, string name,
                          int life, int money, string[] artifacts)
{
    XmlDocument xDoc = new XmlDocument();
    XmlElement xRoot = xDoc.CreateElement("users");
    xDoc.AppendChild(xRoot);

    XmlElement userNode = xDoc.CreateElement("user");

    XmlElement nameNode = xDoc.CreateElement("name");
    nameNode.InnerText = name;
    userNode.AppendChild(nameNode);

    XmlElement lifeNode = xDoc.CreateElement("life");
    lifeNode.InnerText = life.ToString();
    userNode.AppendChild(lifeNode);

    XmlElement moneyNode = xDoc.CreateElement("money");
    moneyNode.InnerText = money.ToString();
    userNode.AppendChild(moneyNode);

    XmlElement numNode = xDoc.CreateElement("num");
    numNode.InnerText = artifacts.Length.ToString();
    userNode.AppendChild(numNode);

    XmlElement artifactsNode = xDoc.CreateElement("artifacts");
    artifactsNode.InnerText = string.Join(",", artifacts);
    userNode.AppendChild(artifactsNode);

    xRoot.AppendChild(userNode);

    xDoc.Save(xmlName);
}
```

Якщо потрібно доповнювати інформацію про інших гравців, то можна скористатися наступною функцією:

```
static void AddToXMLFile(string xmlName, string name,
                         int life, int money, string[] artifacts)
{
    XmlDocument xDoc = new XmlDocument();
    xDoc.Load(xmlName);
    XmlElement xRoot = xDoc.DocumentElement;

    XmlElement userNode = xDoc.CreateElement("user");

    XmlElement nameNode = xDoc.CreateElement("name");
    nameNode.InnerText = name;
    userNode.AppendChild(nameNode);

    XmlElement lifeNode = xDoc.CreateElement("life");
    lifeNode.InnerText = life.ToString();
    userNode.AppendChild(lifeNode);

    XmlElement moneyNode = xDoc.CreateElement("money");
    moneyNode.InnerText = money.ToString();
    userNode.AppendChild(moneyNode);
```

```
            XmlElement numNode = xDoc.CreateElement("num");
            numNode.InnerText = artifacts.Length.ToString();
            userNode.AppendChild(numNode);

            XmlElement artifactsNode = xDoc.CreateElement("artifacts");
            artifactsNode.InnerText = string.Join(",", artifacts);
            userNode.AppendChild(artifactsNode);

            xRoot.AppendChild(userNode);

            xDoc.Save(xmlName);
        }
```

Читання XML-файлу з результатами гри:

```
static void ReadFromXMLFile(string xmlName)
{
    var doc = new XmlDocument();
    doc.Load(xmlName);

    XmlNodeList userNodes = doc.SelectNodes("/users/user");
    if (userNodes == null || userNodes.Count == 0)
    {
        Console.WriteLine("No nodes found for <user>");
        return;
    }

    foreach (XmlNode userNode in userNodes)
    {
        string name = userNode.SelectSingleNode("name").InnerText;
        string life = userNode.SelectSingleNode("life").InnerText;
        string money = userNode.SelectSingleNode("money").InnerText;
        string num = userNode.SelectSingleNode("num").InnerText;
        string artifacts = userNode.SelectSingleNode("artifacts").InnerText;

        Console.WriteLine($"name={name}, life={life}%, money={money}, artifacts={artifacts}");
    }
}
```

Серіалізація в XML у C# – це перетворення об’єкта (або списку об’єктів) у XML-файл так, щоб його можна було зберегти, передати мережею або прочитати іншою програмою. Зворотній процес називається *десеріалізація* – відновлення об’єкта з XML.

Серіалізація в XML C# виконується за допомогою класу XmlSerializer з простору імен System.Xml.Serialization.

Для цього потрібно створити клас, який описує дані для XML-файлу. Клас має бути public, і зазвичай потрібен порожній конструктор (або авто-ініціалізація). Серіалізуються public властивості/поля (можна керувати атрибутами XmlRoot, XmlElement, XmlAttribute, XmlIgnore).

Для прикладу, який розглянуто вище створимо окремий клас User:

```
using System.Xml.Serialization;
```

```
[XmlRoot("user")]
public class User
{
    [XmlElement("name")]
    public string Name { get; set; } = "";

    [XmlElement("life")]
    public int Life { get; set; } = 0;

    [XmlElement("money")]
    public int Money { get; set; } = 0;

    [XmlElement("artifacts")]
    public string ArtifactsCsv { get; set; } = "";

    [XmlIgnore]
    public string[] Artifacts
    {
        get => string.IsNullOrWhiteSpace(ArtifactsCsv)
            ? Array.Empty<string>()
            : ArtifactsCsv
                .Split(',', (char)StringSplitOptions.RemoveEmptyEntries)
                .Select(x => x.Trim())
                .Where(x => x.Length > 0)
                .ToArray();

        set => ArtifactsCsv = (value == null || value.Length == 0)
            ? ""
            : string.Join(", ", value.Select(x => (x ?? "").Trim()).Where(x => x.Length > 0));
    }

    [XmlElement("num")]
    public int Num
    {
        get => Artifacts.Length;
        set { }
    }
}
```

Серіалізація списку об'єктів:

```
string fileName = "settingsSer.xml";
var players = new List<User>
{
    new User { Name="Olena", Life=100, Money=250, Artifacts = new[] {"Ring","Amulet","Key"} },
    new User { Name="Ivan",  Life=80,  Money=120, Artifacts = new[] {"Map","Potion"} }
};

var serializer = new XmlSerializer(typeof(List<User>));

var writer = XmlWriter.Create(fileName, new XmlWriterSettings { Indent = true });
serializer.Serialize(writer, players);
```

В результаті маємо наступний файл:

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfUser xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <User>
    <name>Olena</name>
    <life>100</life>
    <money>250</money>
    <artifacts>Ring, Amulet, Key</artifacts>
    <num>3</num>
  </User>
  <User>
    <name>Ivan</name>
    <life>80</life>
    <money>120</money>
    <artifacts>Map, Potion</artifacts>
    <num>2</num>
  </User>
</ArrayOfUser>
```

Десеріалізація списку об'єктів:

```
var serializer = new XmlSerializer(typeof(List<User>));

var fs = File.OpenRead(fileName);
List<User> players = (List<User>)serializer.Deserialize(fs);

foreach (var p in players)
{
    Console.WriteLine($"Name={p.Name}, Life={p.Life}, Money={p.Money}, Num={p.Num}");
    Console.WriteLine("Artifacts: " + string.Join(" - ", p.Artifacts));
    Console.WriteLine();
}
```

Структура JSON – вся інформація зберігається у вигляді *пар ключ-значення*.

Ключі в JSON це рядки, укладені в подвійні лапки. Ключі відіграють роль ідентифікаторів для значень і допомагають структурувати дані. Вони є унікальними в межах одного об’єкта JSON, що означає, що кожен ключ в об’єкті має бути унікальним.

Значення в JSON можуть бути різними типами даних, включно з рядками (текст), числами, логічними значеннями (true або false), null (порожнє значення), масивами або вкладеними об’єктами. Значення можуть бути укладені в подвійні лапки (для рядків), без лапок (для чисел, логічних значень і null) або в квадратні дужки чи фігурні дужки (для масивів і об’єктів відповідно).

Приклад JSON-файлу:

```
{
     "name": "John",       // Строкове значення
     "age": 30,            // Числове значення
     "isStudent": true     // Логічне значення
     "address": null       // значення null (пусто)
}
```

де name, age, isStudent та address – це ключі, а їхні значення відповідно «John», 30 і true.

Масиви в JSON являють собою впорядковані списки елементів. Вони укладаються в квадратні дужки і можуть містити значення різних типів даних, включно з іншими масивами та об’єктами.

Наприклад:

```
{
  "fruits": ["apple", "banana", "orange"]
}
```

де fruits – це ключ, а його значення – масив з елементами apple, banana і orange.

Об'єкти являють собою невпорядковані набори ключів і значень. Вони містяться у фігурних дужках і дають змогу структурувати дані складніше, вкладаючи об'єкти один в одного.

Наприклад:

```
{
      "person":
      {
            "name": "Alice",
            "age": 25
      }
}
```

person – ключ, а його значення являє собою вкладений об'єкт із ключами name і age.

Серіалізація і десеріалізація – це два важливі процеси, пов'язані з JSON, які дають змогу перетворити дані у формат JSON і назад.

Серіалізація JSON – це процес перетворення даних з об'єктів і структур даних у рядок JSON. Коли ми серіалізуємо дані, ми робимо їх придатними для передавання мережею або збереження на диску в текстовому форматі.

Десеріалізація JSON – це зворотний процес, під час якого рядок JSON перетворюється назад в об'єкти і структури даних. Коли ми отримуємо дані у форматі JSON з мережі або файлу, нам потрібно перетворити їх назад на об'єкти, щоб використовувати їх у програмі.

Розглянемо зберігання даних, описаних вище, у JSON-файлу. Розглянемо процес серіалізації і десеріалізації. Клас даних описаний вище.

Потрібно в проекті встановити System.Text.Json.

Серіалізація списку об'єктів:

```
string fileName = "settingsSer.json";
var users = new List<User>
{
    new User { Name="Olena", Life=100, Money=250, Artifacts = new[] {"Ring","Amulet","Key"} },
    new User { Name="Ivan",  Life=80,  Money=120, Artifacts = new[] {"Map","Potion"} }
};
//string jsonString = JsonSerializer.Serialize(users);
var options = new JsonSerializerOptions
{
    WriteIndented = true,
    Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping
};

string json = JsonSerializer.Serialize(users, options);
File.WriteAllText(fileName, json);
```

В результаті маємо наступний файл:

```
[
  {
    "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
  }
]
```

Десеріалізація списку об'єктів:

```
string json = File.ReadAllText(fileName);
List<User> users = JsonSerializer.Deserialize<List<User>>(json);

if (users == null)
{
    Console.WriteLine("No data in JSON.");
    return;
}

foreach (var p in users)
{
    Console.WriteLine($"Name={p.Name}, Life={p.Life}, Money={p.Money}, Num={p.Num}");
    Console.WriteLine("Artifacts: " + string.Join(" - ", p.Artifacts));
    Console.WriteLine();
}
```

2. Завдання до практичної роботи 6

Введіть в гру 3 – 5 параметрів, які необхідно зберігати та зчитувати при запуске гри. Додайте можливість збергігання та читання введених даних з формату XML чи JSON.

Тема 13. Методи захисту від злому в іграх
Практична робота 7. Методи захисту від злому в іграх

Мета роботи: познайомитися з мовою програмування C# та принципами об'єктно-орієнтованого програмування. Навчитися створювати ігри за допомогою мови програмування C#, використовуючи об'єктно-орієнтований підхід.

1. Теоретичний матеріал

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

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

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

Шифрування внутрішньо-ігрових 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 внутрішньо-ігрових даних включає в себе застосування різноманітних шифрувальних алгоритмів та практик, щоб забезпечити конфіденційність та цілісність інформації в межах гри.

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

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

Алгоритм кодування Base64 (покроково):

- Розбиття на байти – вихідні дані діляться групи по 3 байти (3 x 8=24 біта).
- Перегрупування в 6 біт – кожен 24-бітний блок розбивається на 4 блоки по 6 біт (4 x 6=24 біта).
- Індексація – 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 байт, використовується символ = для заповнення, щоб підсумковий рядок був кратний 4 символам.

Наприклад, маємо слово "Man" (3 байта). Запишемо слово в бітах: M (77), a (97), n (112) → 01001101 01100001 01101110 (24 біта). Розбивка по 6 біт: 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


```
Ww0KICB7DQogICAgIk5hbWUiOiAiT2xlbmEiLA0KICAgICJMaWZlIjogMTAwLA0KICAgICJNb25leSI6IDI1MCwNCiAgICAiQXJ0aWZhY3RzQ3N2IjogIlJpbmcsIEFt
dWxldCwgS2V5IiwNCiAgICAiQXJ0aWZhY3RzIjogWw0KICAgICAgIlJpbmciLA0KICAgICAgIkFtdWxldCIsDQogICAgICAiS2V5Ig0KICAgIF0sDQogICAgIk51bSI6
IDMNCiAgfSwNCiAgew0KICAgICJOYW1lIjogIkl2YW4iLA0KICAgICJMaWZlIjogODAsDQogICAgIk1vbmV5IjogMTIwLA0KICAgICJBcnRpZmFjdHNDc3YiOiAiTWFw
LCBQb3Rpb24iLA0KICAgICJBcnRpZmFjdHMiOiBbDQogICAgICAiTWFwIiwNCiAgICAgICJQb3Rpb24iDQogICAgXSwNCiAgICAiTnVtIjogMg0KICB9DQpd
```


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

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 таблиці 13.1.

Таблиця 7.1. 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 раундо вого 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 вектору ініціалізації (IV) для створення ICryptoTransform через CreateEncryptor або CreateDecryptor, а потім обробку 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 же ключем, завжди виглядатиме однаково. Це дозволяє хакерам знаходити закономірності (наприклад, розпізнати стандартні заголовки повідомлень).

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

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

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

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

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

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

```
System.Security.Cryptography.AesImplementation
Разм?р ключа: 256
Разм?р блоку: 128
Режим роботи: CBC
Розм?ри ключ?в, що п?дтримуються симетричним алгоритмом:
128 - 256 бит (с шагом 64)
Розм?ри блок?в, що п?дтримуються симетричним алгоритмом:
128 - 128 бит
Вектор ?н?ц?ал?зац?ї: 4D7AE664F78D13C1548E61FC303BA2BD
Ключ: A43EB10A60FE6A4D7F088AE499F3457669A5066946DB061C9FAE4865EEC01656
```

Як видно з виведення консолі, для AES ми можемо використовувати ключі розміром 128 біт, 192 біти та 256 біт. За замовчуванням, в отриманому об'єкті вже створено ключ розміром 256 біт і вектор ініціалізації в 128 біт. Але, за бажання, ми можемо змінити ці значення. Зазвичай беруть ключ 32 байти 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 Stream від класу, 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 outStream = 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;
        outStream.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

```
¬™ёA·□*·2тўбтЩш•Ц□Л□ДшЈйw™Цы•М°эр

J□AG0C□FLe□йIJЧRJb)‰ □ЫэµzP□oFЂьњЁ"…□€5um@ъVDШ□д
9□њ□ґn¬ы!НЇЯBN¬Т¬в2Т!їХЩ·aiФ□QKe□ґIжvRґм".sъ,c-хжЁ\□□ВЎI□1Ш^э„№•ъ□^Щ□#/
¬□¦НЫ“йЈй%<ёY□J–sK_V□>йc3i%.3т□ШtCп□7a□nњЃv3Жвр0□i#П›’b3s©ч□
 □ЙУ□tV–ДЬ□8рцJШАђ№ri□□QOTґ;'□r,~Ч□□рдќ~"— цoNЃ»жKX□Н±=_□D°JђЩЦ□'*x їНВ–□j□-
Р4щМ`_СЏС
hc□ґТђН□Я□Л®К”□□§НТ0ўг□–7jУЖNсю4ТР?”»ъt□М3¶Њч«kI ылп…
□…И?ґл”,
Fљjт□¤>jaL□Э□LЄљ
```

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 зайвого мета-інформації – вилучення зайвої мета-інформації, яка може бути корисною для реверс-інжинірингу, наприклад, видалення імен методів чи класів.

Методи обфускації можна застосовувати для захисту пам'яті програми.

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

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

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

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

private void Awake()
{
    _instance = this;
}

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

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

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

}
```

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

```
public class ObfuscateSettings : MonoBehaviour
{
    static int random;

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

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

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

Зробимо змінив в скрипті GameController:

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

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

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

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

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

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

}
```

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

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

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

Основні характеристики honeypot включають:

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

2. Завдання до практичної роботи 7

Створенні файли для збергігання та читання введених даних у формату XML чи JSON зберігати у зашифрованому вигляді. Спочатку до них застосувати шифрування даних, а потім Base64-перетворення.

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

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

Рекомендована література

Основна

1. Lanzinger F. 3D Game Development with Unity. Boca Raton: CRC Press, 2022. 414 p.
2. Jackson S. Unity 3D UI Essentials. Birmingham: Packt Publishing, 2015. 280 p.
3. Gupta K. Unity3D Tutorial for Beginners. FutureZenGroup, 2021. 81 p.
4. Creighton R. Unity 3D Game Development by Example. Birmingham: Packt Publishing, 2010. 364 p.

Допоміжна

5. Nakov S., Nakov's Team. Programming Basics with C#. Sofia: SoftUni, 2019. 408 p.
6. Norton T. Learning C# by Developing Games with Unity 3D. Birmingham: Packt Publishing, 2013. 292 p.
7. Miller R. C# for Artists: The Art, Philosophy, and Science of Object-Oriented Programming. Seattle: Pulp Free Press, 2008. 750 p.
8. Coggan A. Unity Game Audio Implementation: A Practical Guide for Beginners. Boca Raton: Taylor & Francis, 2021. 434 p.
9. Sinclair J.-L. Principles of Game Audio and Sound Design. Boca Raton: Taylor & Francis, 2020. 295 p.

Інформаційні ресурси

10. Unity: офіційний вебсайт. URL: https://unity.com/
11. Visual Studio Community: офіційний вебсайт. URL: https://visualstudio.microsoft.com/vs/community/

Unity Asset Store: вебсайт. URL: https://assetstore.unity.com/