Міжнародний гуманітарний університет
Кафедра Комп’ютерної інженерії та інноваційних технологій

Лебедєва О.Ю.

ПРОГРАМУВАННЯ ТА ЗАХИСТ ІГРОВИХ ЗАСТОСУНКІВ

методичні рекомендації для практичних занять
для здобувачів першого (бакалаврського) рівня,
які навчаються за спеціальністю 125 – Кібербезпека та захист інформації

Одеса 2025

Затверджено Вченою Радою Міжнародного гуманітарного університету (протокол № 12 від 23 червня 2025 р.).

Лебедєва О.Ю.

Програмування та захист ігрових застосунків: методичні рекомендації для практичних занять для здобувачів першого (бакалаврського) рівня, які навчаються за спеціальністю: 125 – Кібербезпека та захист інформації [Електронне видання] / О.Ю. Лебедєва, кафедра комп'ютерної інженерії та інноваційних технологій Міжнародного гуманітарного університету. Одеса, 2025. – 98 с.

Методичні рекомендації для практичних занять для курсу «Програмування та захист ігрових застосунків» призначені для здобувачів, що навчаються за першим (бакалаврським) рівнем за спеціальністю: 125 – Кібербезпека та захист інформації. Дисципліна «Програмування та захист ігрових застосунків» надає здобувачам знання в галузі комп'ютерних ігор та захисту інформації. На основі загальних понять в програмуванні та об'єктно-орієнтованого підходу дисципліна розглядає процес створення 3D-ігор в середовищі Unity, написання скриптів для реалізації функціональності гри, створення інтерфейсу користувача та захисту даних.

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:

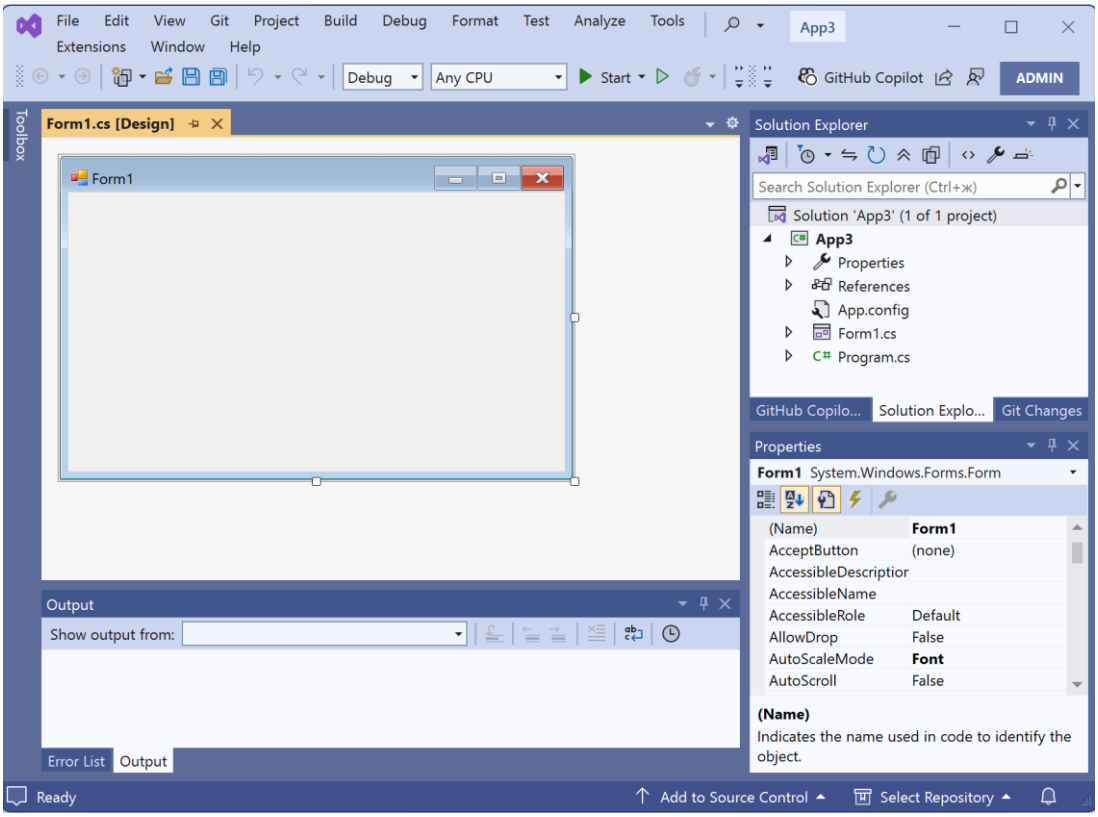
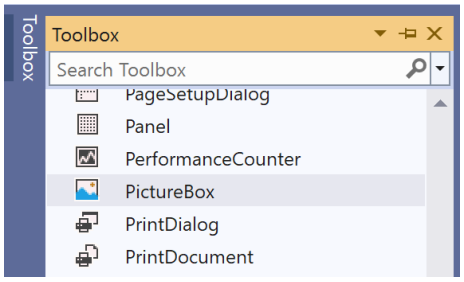
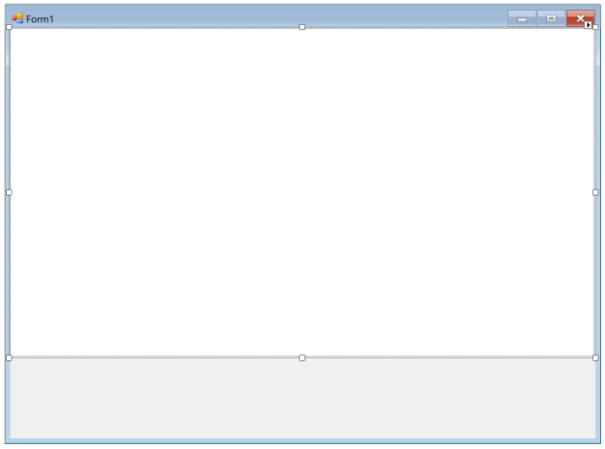


Рисунок 1.1 – Вид проекту Windows Forms App (.NET Framework)

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.

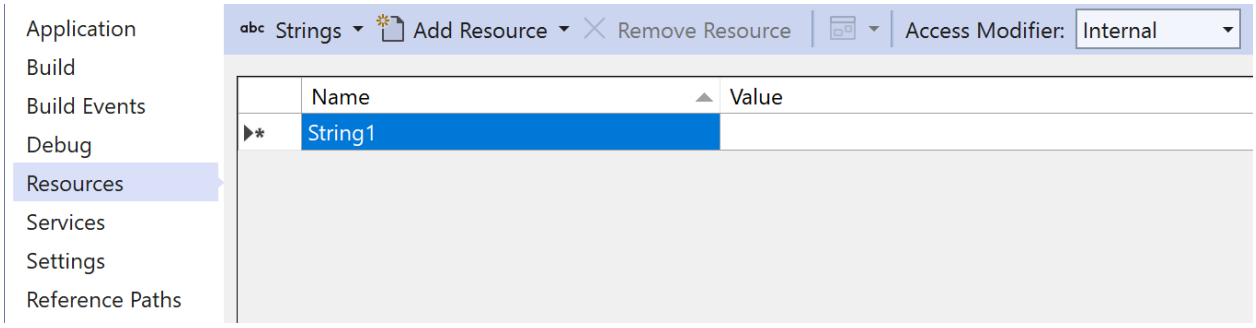


Виділимо PictureBox та в вікні Properties замінимо властивість форми BackColor на значення White:

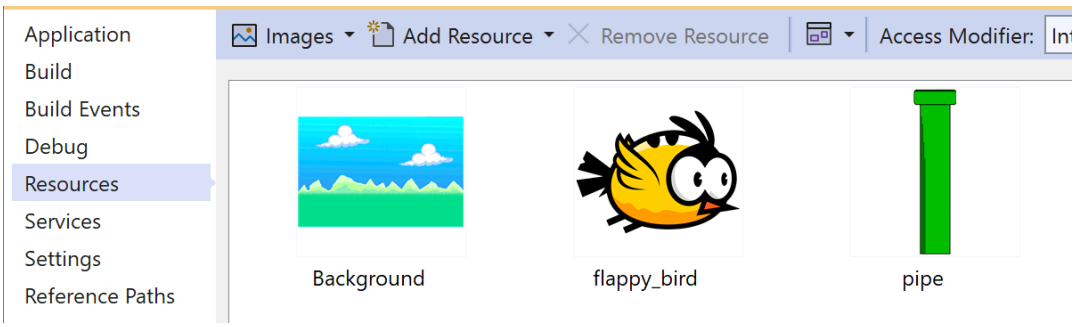


Створимо два нових класу для об’єкту Bird та Pipe. Для цього у вікні Solution Explorer виділяємо назву проекту та натискаємо праву кнопку миші. Обираємо Add → Class. У вікні, що з’явиться внизу напишемо назву класу, наприклад Bird. Аналогічно створюємо новий клас Pipe. Відкриємо ці класи. Для цього необхідно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

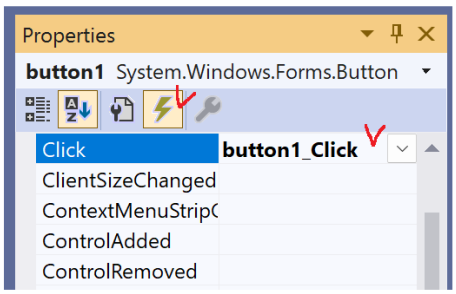
Метод PaintScene() малює картинки у PictureBox:

```
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 2. Інтерфейс Unity
Практична робота 2. Інтерфейс Unity

Мета роботи: познайомитися з програмою Unity Hub та редактором Unity. Навчитися створювати ігри з використанням 3D шаблону Unity, створювати нові сцени, додавати нові джерела світла.

1. Теоретичний матеріал

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

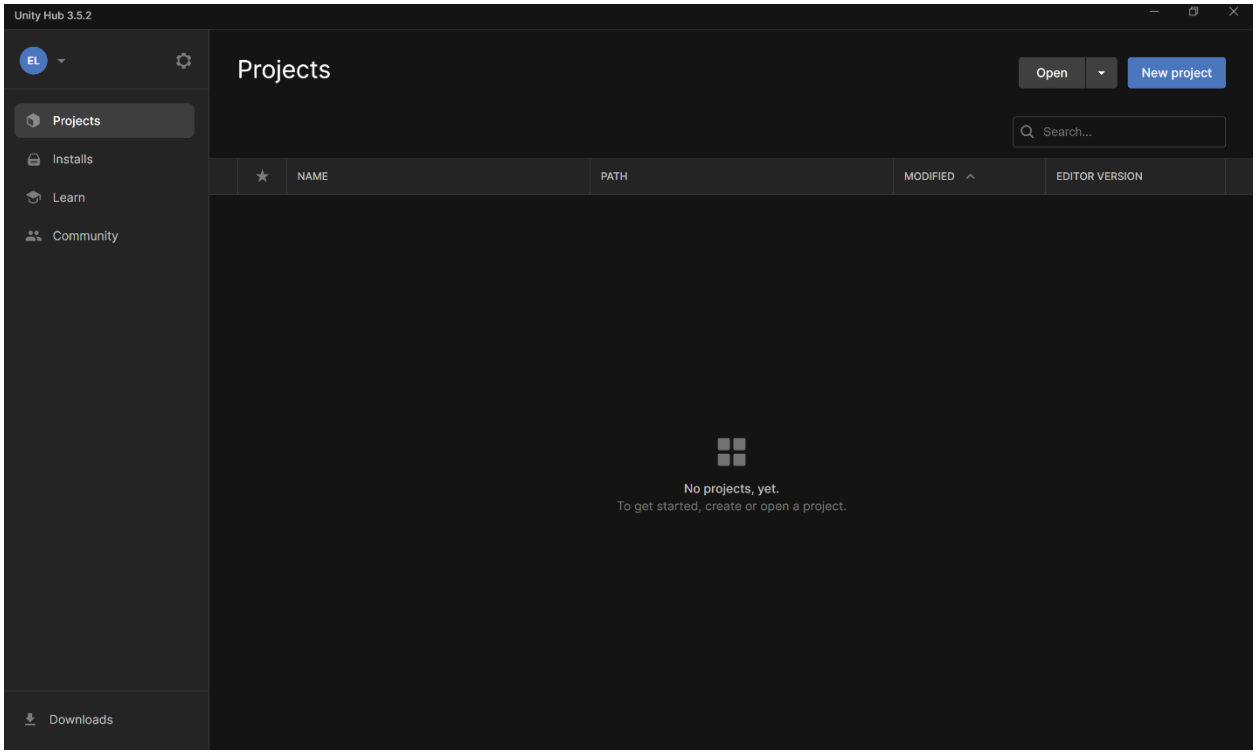


Рисунок 2.1 – Вікно Unity Hub

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

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

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

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

LTS (Long Term Support) у Unity означає довгострокову підтримку. Це версія Unity, призначена для розробників, яким потрібна максимальна стабільність та підтримка. LTS версії отримують виправлення помилок та патчі безпеки протягом двох років, що робить їх ідеальними для проектів, які потребують тривалого обслуговування та підтримки.

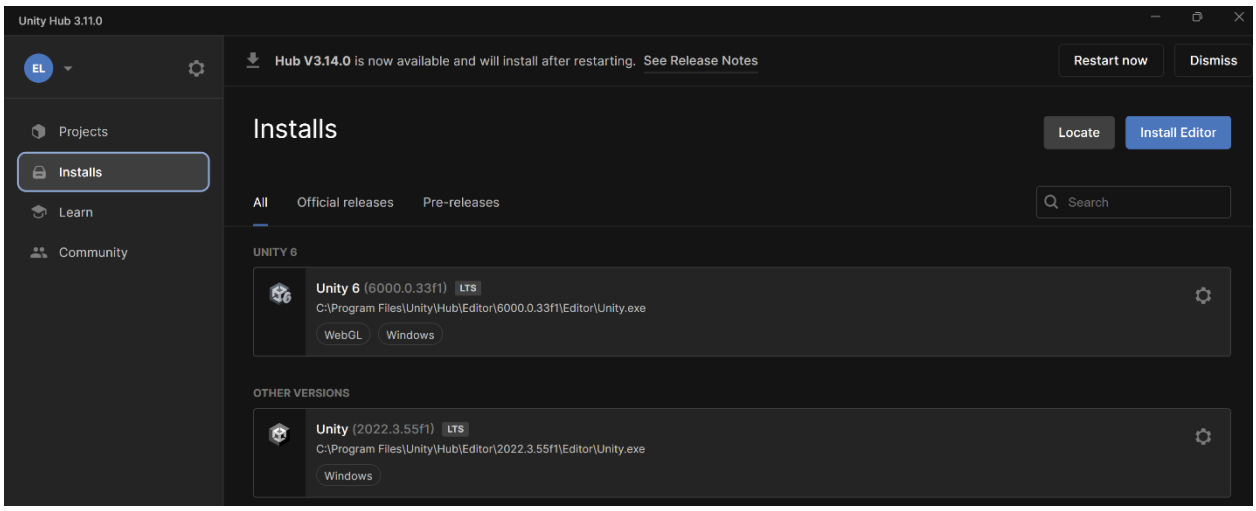


Рисунок 2.2 – Вікно Unity Hub Installs

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

Для створення порожнього 3D проекту зайдіть у Unity Hub та натисніть кнопку New project.

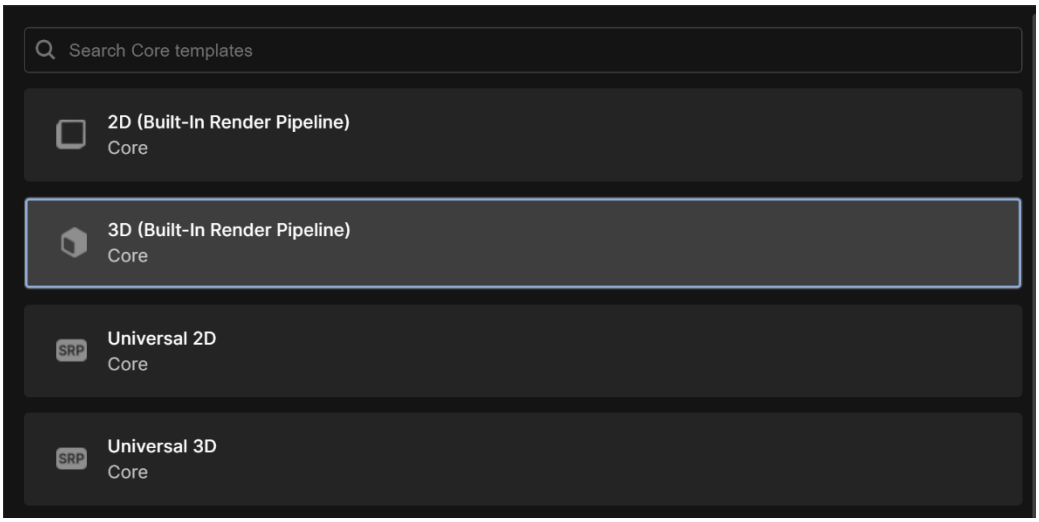


Рисунок 2.3 – Вікно для вибору шаблону проекту, що створюється

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.

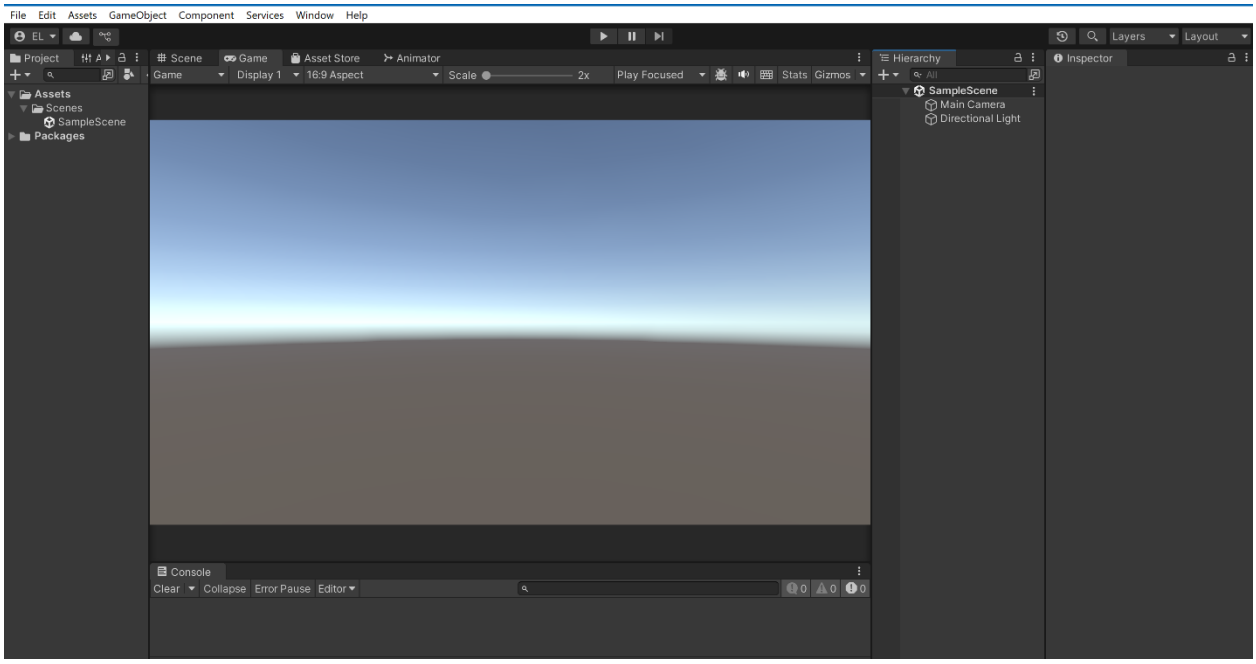
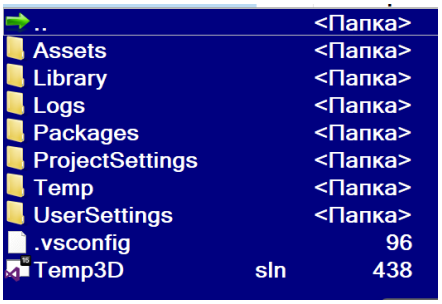


Рисунок 2.4 – Вікно проекту 3D

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

– Assets,
– ProjectSettings
– Library.



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

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

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

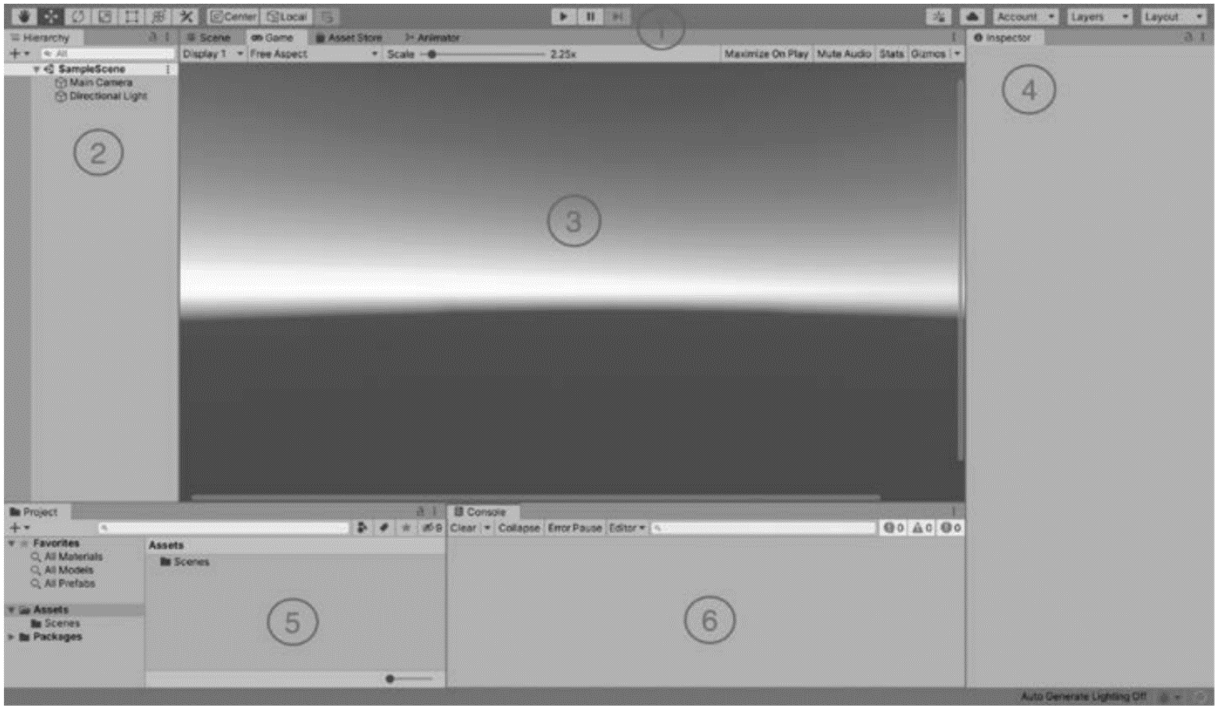
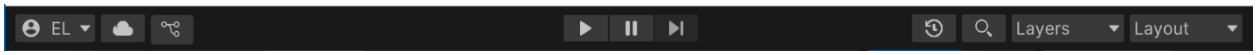
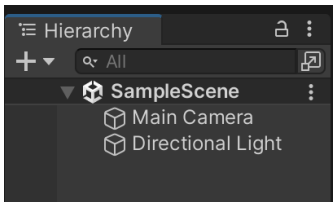


Рисунок 2.5 – Вікно редактору Unity

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



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



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

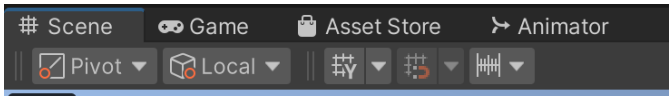
Ієрархія, як випливає з назви, дозволяє також бачити стосунки батьків-нащадок між об'єктами. У Unity об'єкти можуть містити інші об'єкти. Панель ієрархії дозволяє досліджувати такі дерева об'єктів. Також підтримується можливість переупорядковувати об'єкти списки, перетягуючи їх мишею.

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

Редактор Unity завжди знаходиться в одному з двох режимів: Edit Mode (Режим редагування) або Play Mode (Режим відтворення). У режимі редагування, що діє за умовчанням, можна створювати сцени, налаштовувати ігрові об'єкти та виконувати інші дії,

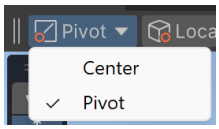
пов'язані зі створенням гри. У режимі відтворення ви можете грати в свою гру і взаємодіяти зі сценою.

На панелі Scene вгорі, поряд з інструментами, можна побачити ще кнопки.



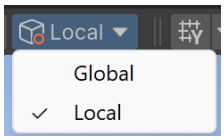
Зараз тут написано Pivot та Local. Ці кнопки це інструменти, це властивості інструментів.

Перша перемикається між пивот та центр. Ця властивість показує нам позицію наших інструментів, точніше центр, звідки виходять стрілки осей або напрямні обертання.



Пивот та центр однакові для куба та для більшості простих об'єктів вони збігатимуться. Але в майбутньому вам будуть зустрічатися об'єкти, в яких пивот не збігається, і якщо ви хочете обертати навколо пива або центру, це те, що ви перемикаєте.

Наступне властивість показує обертання інструментів. Нині воно стоїть локально.

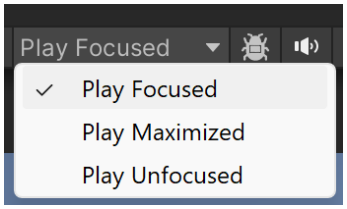


При повороті куба осі інструментів також повертається. Це тому зараз їх обертання локальні. Натиснувши кнопку і зробивши глобальним, можна побачити, що осі повертаються у вихідне становище.

Таким чином, ми можемо вибирати, чи є обертання нашого об'єкта також обертанням осей його інструменту.

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

Поведінка в режимі відтворення:



У Unity є кілька різних режимів запуску Play Mode у редакторі. Вони впливають на те, як вікно Game буде поводитися під час відтворення сцени.

Play Focused:

– При натисканні Play вікно Game автоматично отримує фокус.
– Тобто курсор і клавіатурні події спрямовуються одразу в Game View.
– Зручно, коли ви тестуєте управління (клавіатура, мишка, геймпад) і не хочете щоразу клікати у вікно гри.

– Недолік: якщо вам потрібно одночасно дивитися на Scene View чи інші панелі, вони втрачають фокус.

Play Maximized:

– Коли ви запускаєте Play Mode, Game View автоматично розгортається на весь простір редактора (в межах Unity).
– Це імітує гру у «повноекранному режимі», але без виходу за межі редактора.
– Зручно для перевірки UI, адаптивності інтерфейсу, візуальної якості сцени.
– Після зупинки Play Mode вікно повертається до попереднього розміру/розташування.

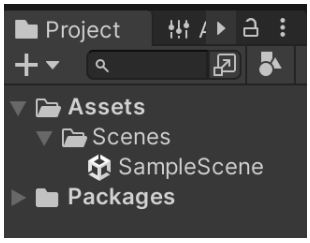
Play Unfocused:

– При запуску Play Mode вікно Game не отримує автоматично фокус.
– Це означає, що ввод із клавіатури/мишки може йти не в Game View, а в інші панелі (наприклад, Scene або Inspector).
– Зручно, коли ви хочете одночасно редагувати налаштування об’єктів у Inspector, змінювати параметри, не перемикаючись постійно між панелями.
– Використовується, коли під час тестів важливі не стільки управління в грі, скільки спостереження за змінами об’єктів у сцені.

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

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

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

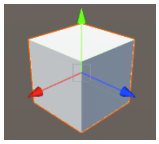


На *панелі Console* з'являться вихідні дані, які виводитимуть створені сценарії.

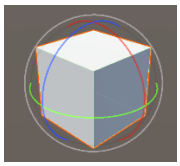
Подання сцени може бути в одному з п'яти різних режимів. Перемикач режимів, що знаходиться у вікні, керує режимом взаємодій з поданням сцени.

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

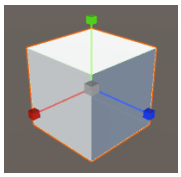
Режим переміщення  (Translation mode) – у цьому режимі можна переміщати виділені об'єкти.



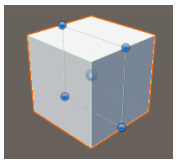
Режим обертання  (Rotation mode) – у цьому режимі можна повертати виділені об'єкти.



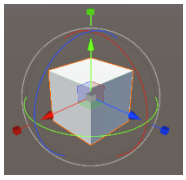
Режим масштабування  (Scale mode) – в цьому режимі можна змінювати розміри виділених об'єктів.



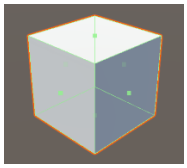
Режим прямокутника  (Rectangle mode) – у цьому режимі можна переміщати вибрані об'єкти та змінювати їх розміри, використовуючи двовимірні маркери. Це особливо зручно при формуванні двовимірних сцен або інтерфейсу користувача.



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.

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

Підтримується кілька способів навігації у поданні сцени:

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

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

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

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

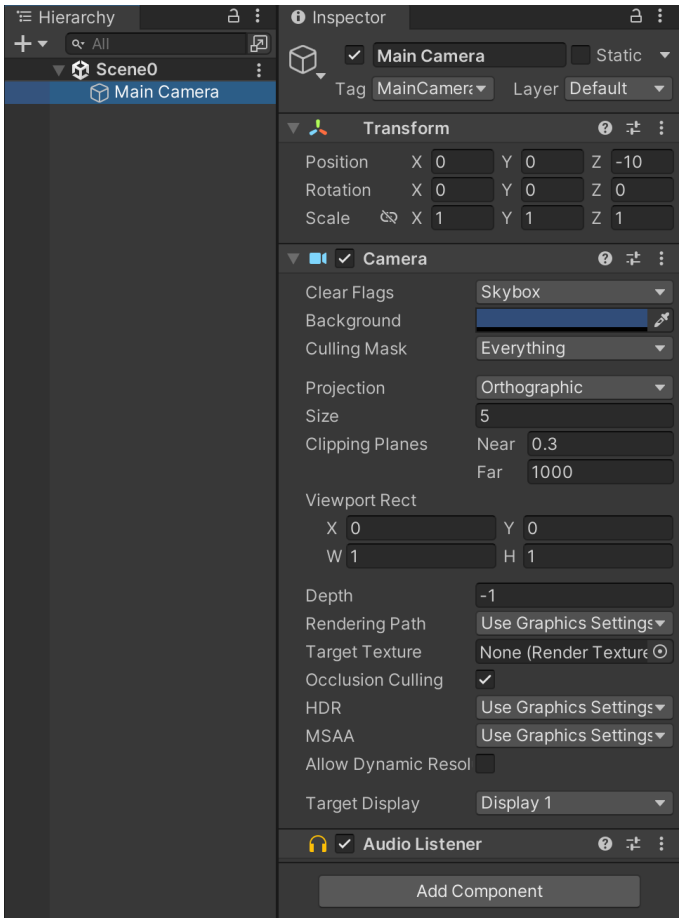


Рисунок 2.6 – Панель Hierarchy та панель Inspector для об’єкту MainCamera

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

Коли ви створюєте новий проект Unity, у поданні сцени відображатиметься нова сцена. Це сцена без назви та збереження. Сцена буде порожньою, за винятком об'єктів – камери та спрямованого світла.

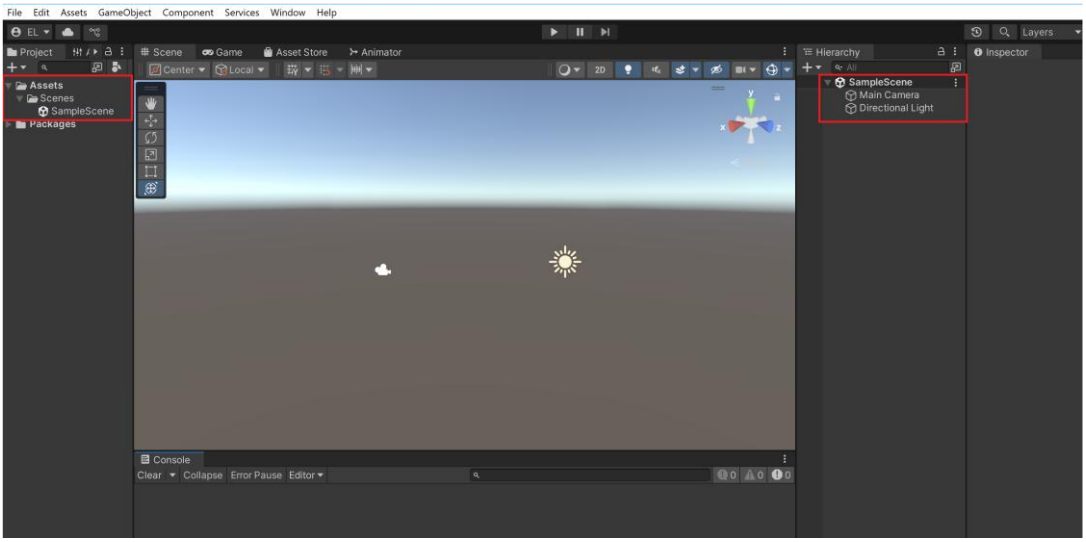


Рисунок 2.7 – Сцена в Unity

Використовуйте діалогове вікно «Нова сцена» для створення нових сцен із певних шаблонів сцен у вашому проекті. Ви також можете використовувати діалогове вікно «Нова сцена» для пошуку шаблонів сцен та керування ними.

За замовчуванням діалогове вікно «Нова сцена» відкривається під час створення нової сцени з меню (Файл → Нова сцена)

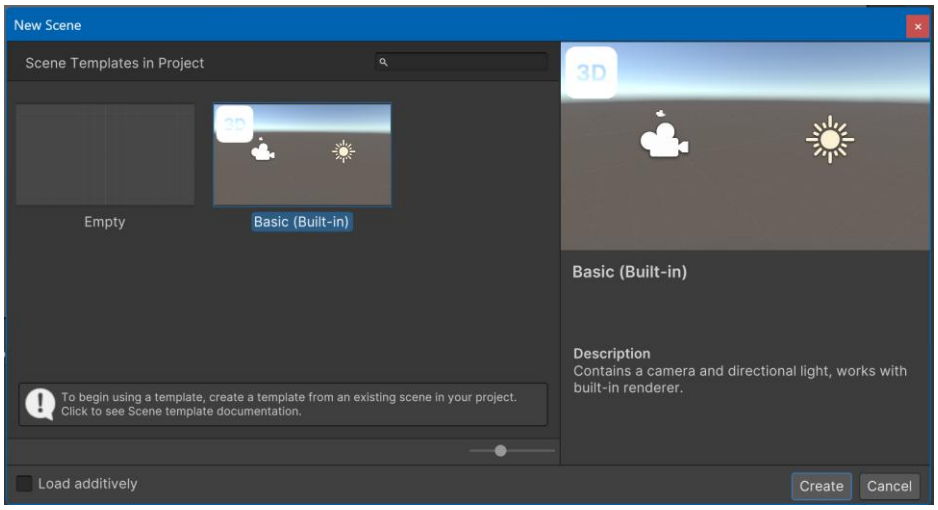
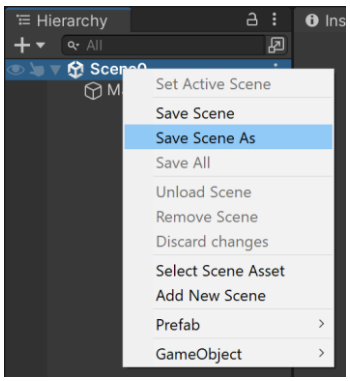


Рисунок 2.8 – Діалогове вікно «Нова сцена»

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



Щоб відкрити сцену, виконайте одну з таких дій:

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

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

Існує кілька способів переходу між сценами, а саме:

– зміна сцен за назвою;
– зміна сцен за індексами;
– зміна сцен за допомогою параметрів.

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

```
using UnityEngine;
using UnityEngine.SceneManagement;

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

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

}
```

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

```
using UnityEngine;
using UnityEngine.SceneManagement;

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

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

}
```

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

```
using UnityEngine;
using UnityEngine.SceneManagement;

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

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

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

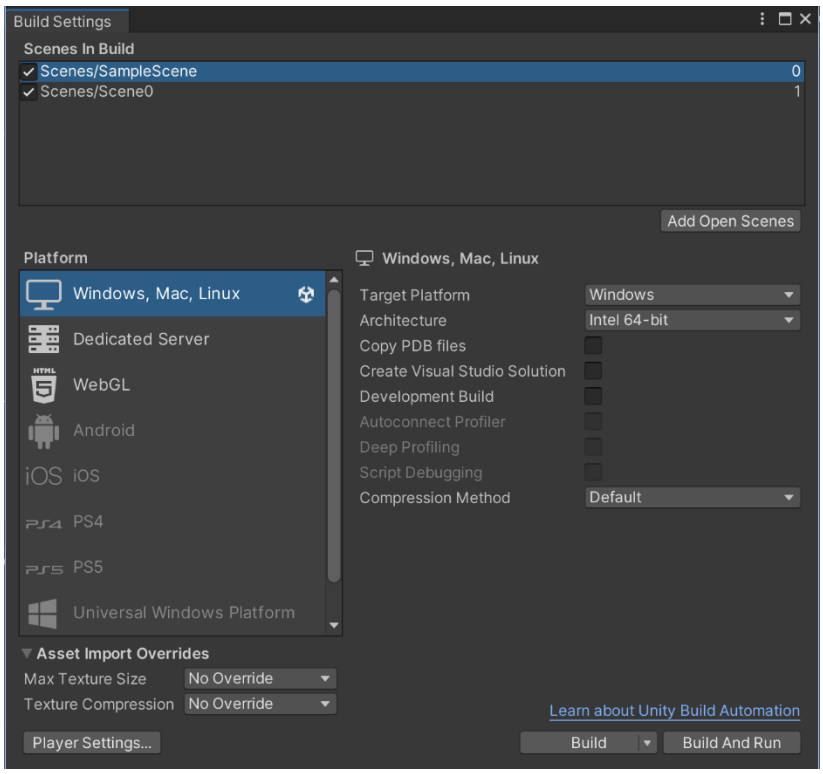


Рисунок 2.9 – Вікно «Build Settings»

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

```
using UnityEngine;
using UnityEngine.SceneManagement;

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

}
```

За замовчанням на сцені присутні такі об’єкти як камера та джерело світла.

Camera (камера) – так називають точку в ігровому просторі, з якою гравець бачить ігровий світ. З точки зору положення камери гри можна поділити на ігри від першої особи (камера розташована так, що реалізує вигляд ніби "з очей" персонажа), ігри з виглядом зліва-зверху (стратегії), ігри з виглядом ззаду – камера розташовується зазвичай позаду гравця і трохи вище за нього. Існують і ігри, де камера може приймати різні позиції.

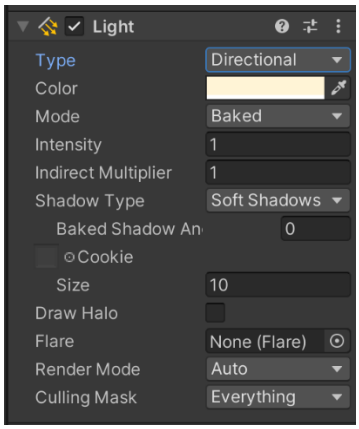
Камера показує нам те, що ми бачимо у нашій грі. У вікні сцени ми можемо бачити зону видимості камери.

Приклад параметрів камери**:**

- Clear Flags — що очищати перед рендером (небо, колір, нічого).
- Background — колір фону.
- Field of View (FOV) — кут огляду.
- Clipping Planes — ближня і дальня площина відсікання.

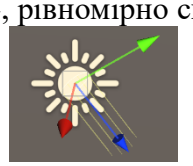
– Projection — перспектива чи ортографія.

Джерела світла – відповідають за освітлення сцени.



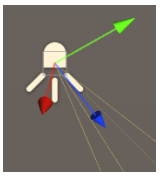
Типи:

– Directional Light — «сонце», рівномірно світить на всі об’єкти.

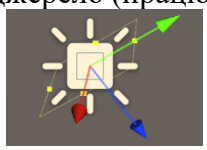


– Point Light — світиться від точки у всі сторони (як лампочка).

– Spot Light — прожектор з кутом світіння.



– Area Light — прямокутне джерело (працює лише з певними рендерами).



Параметри: Intensity — яскравість; Color — колір світла; Range — радіус дії (для Point і Spot); Shadows — увімкнені/вимкнені тіні, їхня якість.

2. Завдання до практичної роботи 2

Створіть 3D проект. В проекті створіть 4 сцени в грі. В кожный сцені зробіть свої налаштування камери. Налаштуйте порядок сцени у вікні «Build Settings». Додайте до сцені декілька додаткових джерел світла. Налаштуйте тип цих джерел світла та їх параметри.

Тема 3. 3D. GameObject та його налаштування
Практична робота 3. 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.

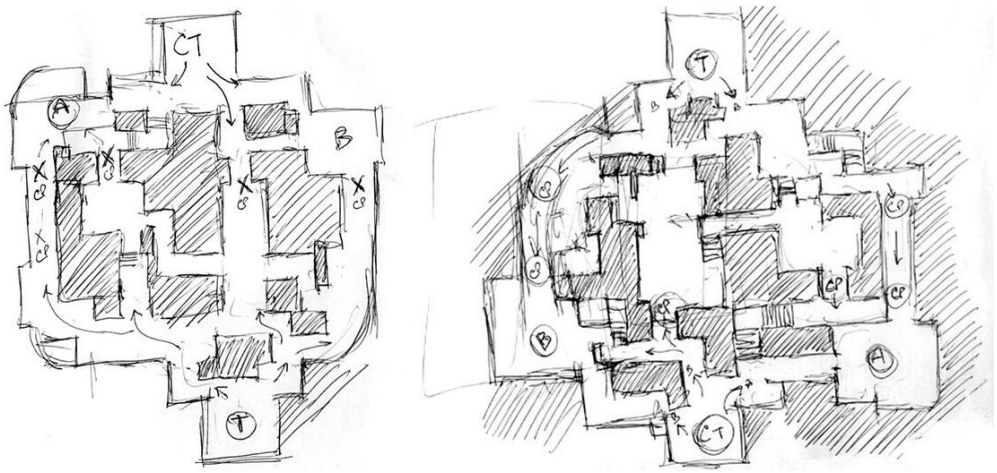


Рисунок 3.1 – Приклад ескізів рівнів

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.

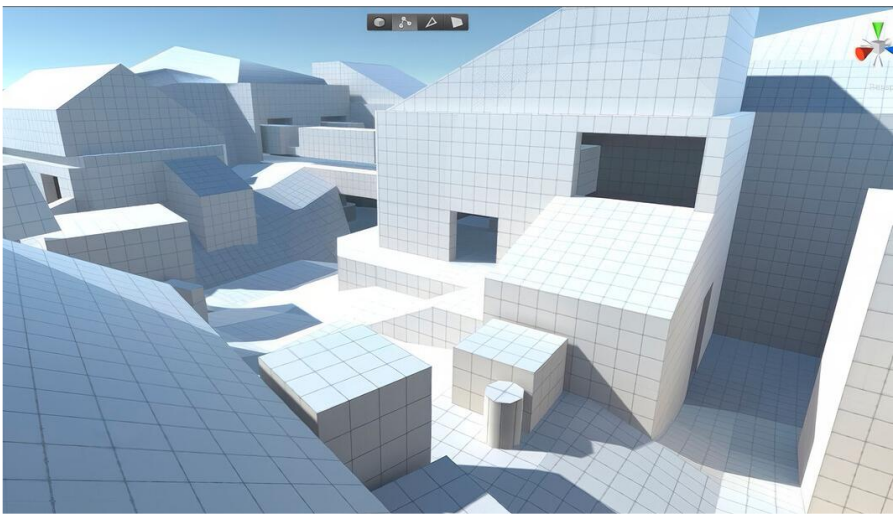


Рисунок 3.2 – Приклад рівня на етапі прототипування

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.

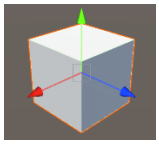
Інтерфейс Unity відображається як набір панелей. Кожна панель має вкладку ліворуч угорі, за яку панель можна перемістити і тим самим змінити макет програми. Також, переміщуючи панель за вкладку, її можна перетворити на самостійне вікно. Не всі панелі видимі за замовчуванням, і в міру розробки гри ви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

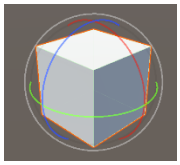
Подання сцени може бути в одному з п'яти різних режимів. Перемикач режимів, що знаходиться у вікні, керує режимом взаємодій з поданням сцени.

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

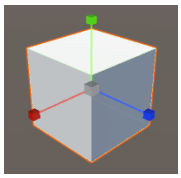
Режим переміщення  (Translation mode) – у цьому режимі можна переміщати виділені об'єкти.



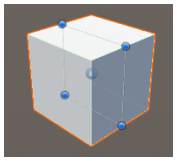
Режим обертання  (Rotation mode) – у цьому режимі можна повертати виділені об'єкти.



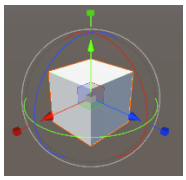
Режим масштабування  (Scale mode) – в цьому режимі можна змінювати розміри виділених об'єктів.



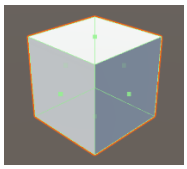
Режим прямокутника  (Rectangle mode) – у цьому режимі можна переміщати вибрані об'єкти та змінювати їх розміри, використовуючи двовимірні маркери. Це особливо зручно при формуванні двовимірних сцен або інтерфейсу користувача.



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.

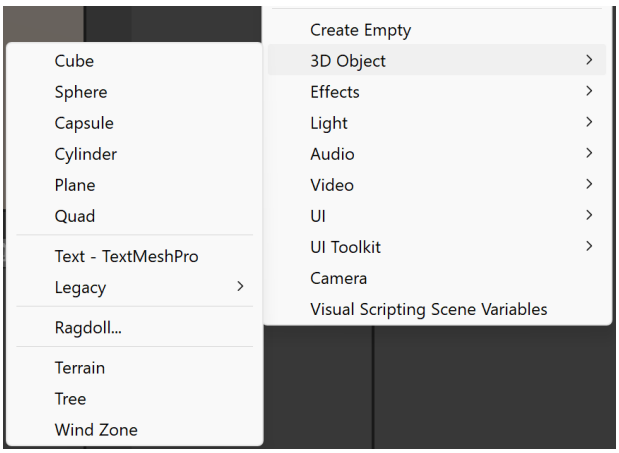


Рисунок 3.3 – Приклад GameObject

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

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

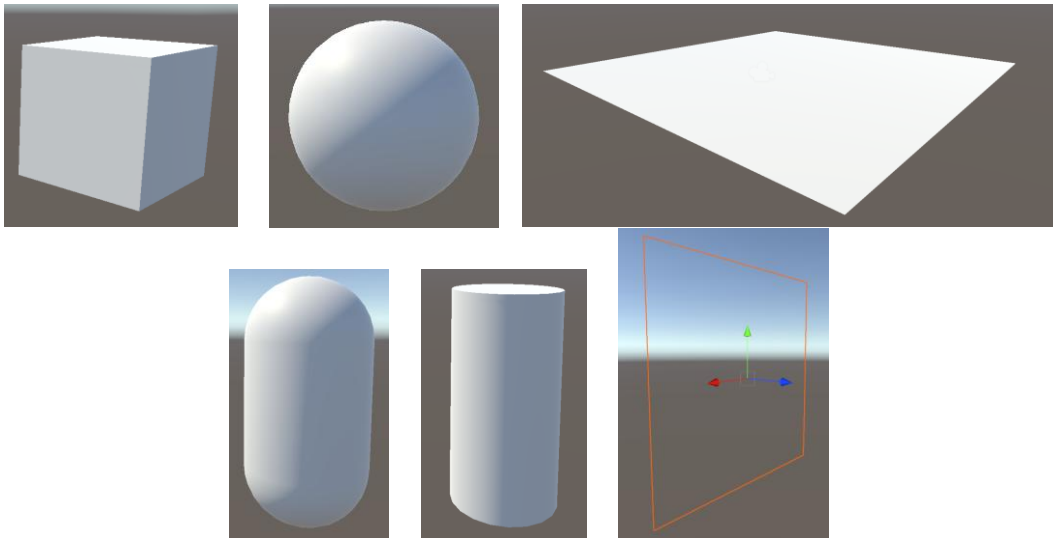


Рисунок 3.4 – Приклад базових 3D-об'єктів

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».

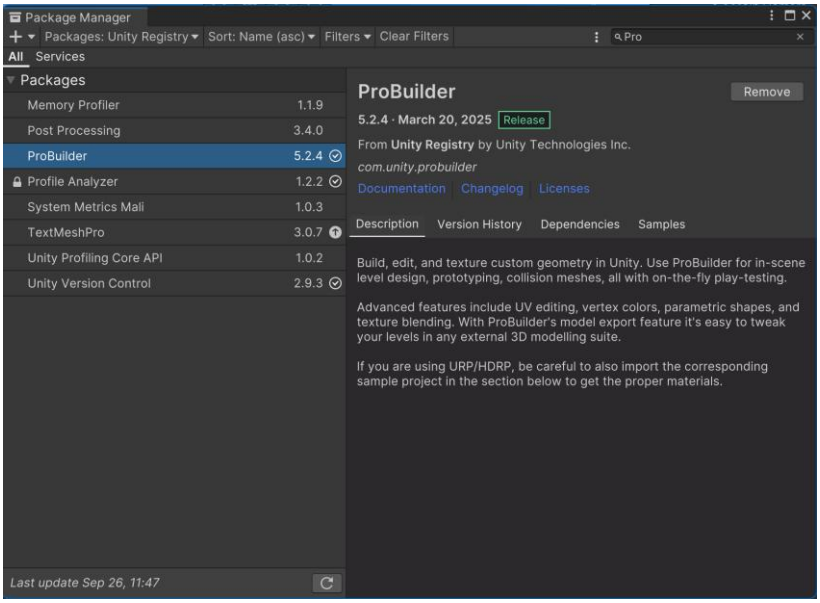


Рисунок 3.5 – Вікно Package Manager для установки ProBuilder

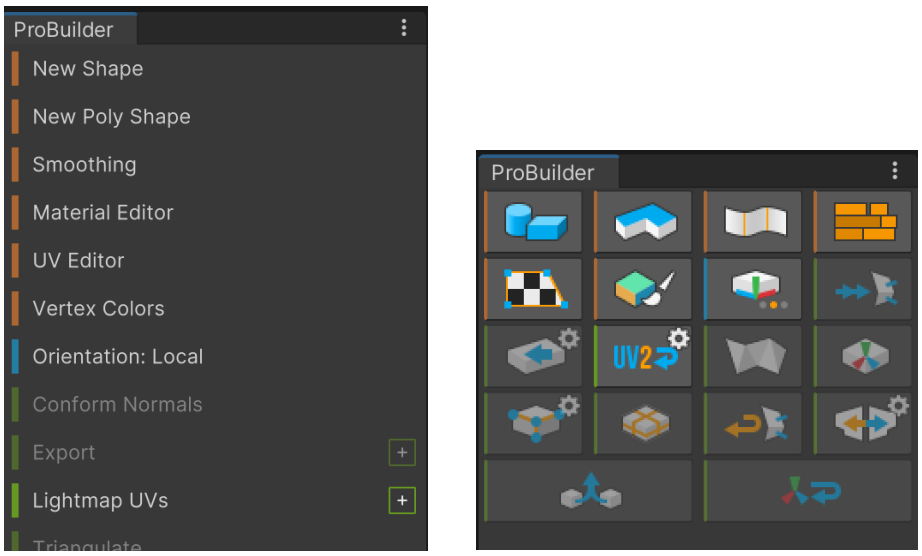
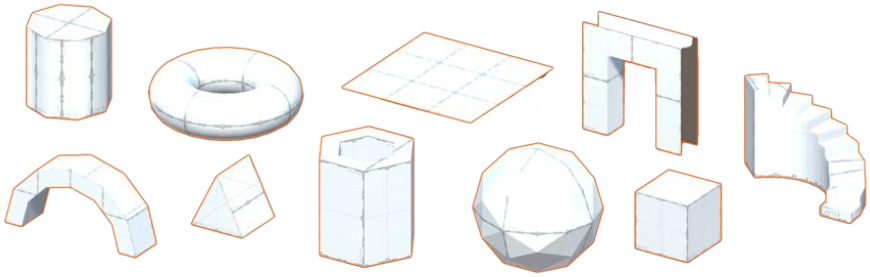
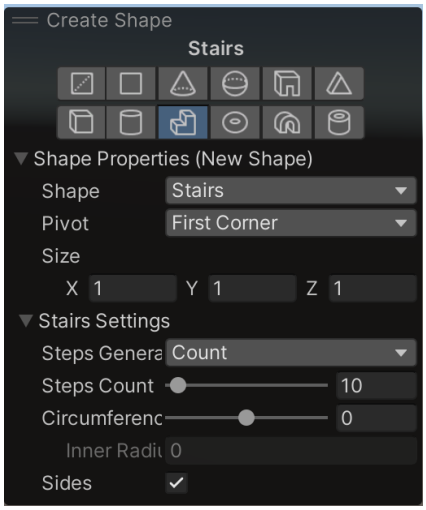


Рисунок 3.6 – Вікно ProBuilder з інструментами в режимі Text mode та Icon mode

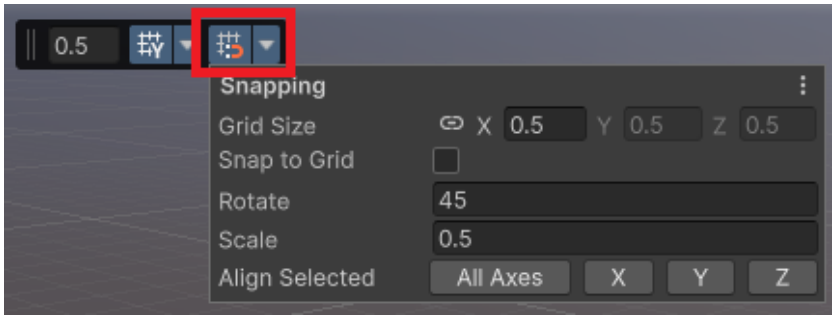
Найпоширеніший підхід полягає у створенні попередньо визначеної фігури за допомогою інструмента «Shape tool»  , який містить бібліотеку фігур. Ці попередньо визначені фігури включають: прості куби, призми, тори та інші прості геометричні елементи, які можна використовувати для створення будівель, транспортних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



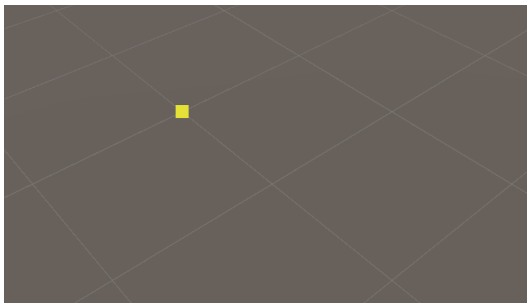
Щоб додати нову фігуру на сцену натискаємо на інструмент «Shape tool»  , з’явиться у вкладці Scene вікно в якому ми можемо обрати фігуру та зробити її налаштування.



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

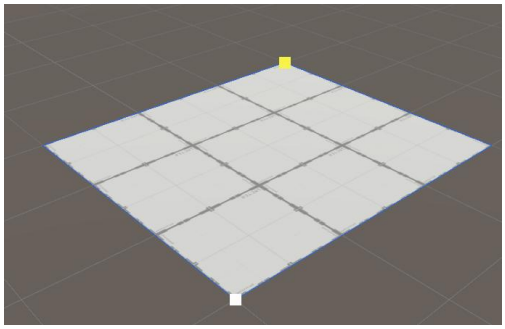


За замовчуванням інструмент «Snap tool» вимкнено. Щоб його ввімкнути, виберіть значок «Увімкнути/вимкнути прив'язку сітки», а потім наведіть курсор на вікно перегляду «Scene», щоб перевірити, чи його ввімкнено. Ви побачите жовтий квадрат, який слідує за вашим курсором, який прив'язується до вершин сітки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

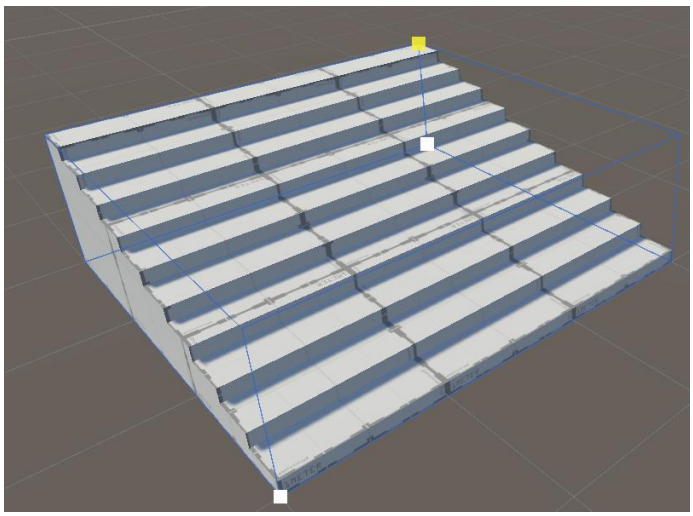


Щоб почати малювати фігури, виконайте такі інструкції:

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.



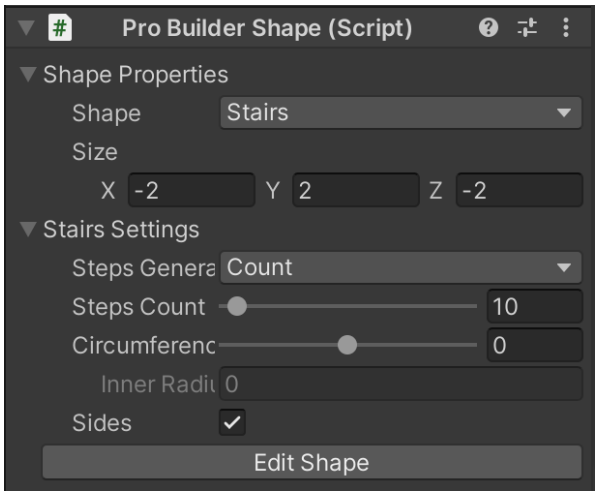
Після того, як ви намалювали фігуру, ви можете налаштувати додаткові властивості, щоб змінити її форму. Доступні властивості відрізняються залежно від вибраної вами початкової форми. Коли ви закінчите налаштування деталей, клацніть порожнє місце у вікні «Ієрархія», щоб завершити вибір та завершити створення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

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

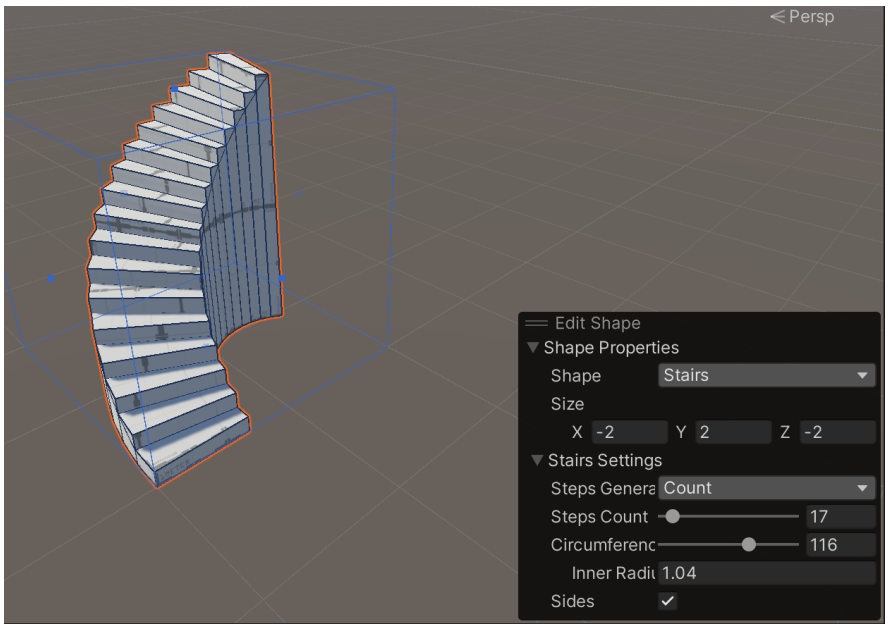
Щоб редагувати вже завершену фігуру, виконайте такі інструкції:

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 3

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 4. Скрипти в Unity
Практична робота 4. Скрипти в Unity

Мета роботи: познайомитися зі скриптами в Unity. Навчитися розробляти власні скрипти в Unity.

1. Теоретичний матеріал

Скриптинг – необхідна складова для всіх ігор. Навіть найпростіші ігри потребують скриптів для реакції на дії гравця та організації подій геймплею.

Крім того, скрипти можуть бути використані для створення графічних ефектів, управління фізичною поведінкою об'єктів або реалізації користувальницької AI системи для персонажів гри.

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

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

Ви можете думати про клас як про свого роду план створення нового типу компонента, який може бути прикріплений до ігрового об'єкта.

Щоразу, коли ви приєднуєте скриптовий компонент до ігрового об'єкта GameObject, створюється новий екземпляр об'єкта, визначений планом.

Ім'я класу береться з імені, яке ви вказали під час створення файлу. Ім'я класу та ім'я файлу повинні бути однаковими для того, щоб скриптовий компонент міг бути приєднаний до ігрового об'єкта.

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

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

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

    }

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

    }
}
```

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

Щоб змінити шаблон, запустить Visual Studio Community 2022 як адміністратор і відкрийте цей файл. Для цього запустіть Visual Studio та виберіть Continue without code

(Продовжити без коду) внизу праворуч. Після запуску виберіть File → Open → File («Файл» → «Відкрити» → «Файл»), перейдіть до файлу, який потрібно редагувати, і двічі клацніть назву файлу.

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

Скрипт Unity не схожий на традиційну ідею програми, де код працює постійно в циклі, поки не завершить своє завдання. Натомість Unity періодично передає керування скрипту при викликі певних оголошених у ньому функцій. Як тільки функція завершує виконання, керування повертається назад до Unity.

Ці функції відомі як функції подій, тобто їх активує Unity у відповідь на події, що відбуваються у процесі гри. Unity використовує схему іменування, щоб визначити, яку функцію викликати певної події.

Функція Update – це місце для розміщення коду, який оброблятиме оновлення кадру для ігрового об'єкта. Це може бути рух, спрацювання дій і реакція на введення користувача, в основному все, що повинно бути оброблено з часом в ігровому процесі.

Щоб дозволити функції Update виконувати свою роботу, часто буває корисно ініціалізувати змінні, рахувати властивості та здійснити зв'язок з іншими ігровими об'єктами до того, як будуть здійснені будь-які дії.

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

Розглянемо інші функції в Unity.

Функції	Опис
Update	викликається перед малюванням кадру та перед розрахунком анімації
FixedUpdate	викликається перед кожним оновленням фізичних даних. Т.к. оновлення фізики та кадру відбувається не з однаковою частотою, то ви отримаєте більш точні результати від коду фізики
LateUpdate	можливість внести додаткові зміни у момент, коли у всіх об'єктів у сцені відпрацювали функції Update та FixedUpdate та розрахувалися всі анімації.
Start	викликається до оновлення першого кадру чи фізики об'єкту.
Awake	викликається для кожного об'єкта в сцені під час завантаження сцени. Для різних об'єктів функції Start та Awake і викликаються в різному порядку, всі Awake будуть виконані до першого виклику Start.
OnGUI	У Unity є система для відтворення елементів керування GUI поверх того, що відбувається в сцені, і реагування на кліки по цих елементах. Цей код обробляється дещо інакше, ніж звичайне оновлення кадру, тому він повинен бути поміщений у функцію OnGUI, яка буде періодично викликатися.
OnMouseOver, OnMouseDown	дозволяє скрипту реагувати на дії з мишею користувача
OnCollisionEnter, OnCollisionStay OnCollisionExit	будуть викликані на початок, продовження та завершення контакту (зіткненнях) з об'єктом

OnTriggerEnter, OnTriggerStay OnTriggerExit	будуть викликані, коли колайдер об'єкта налаштований як Trigger (тобто цей колайдер просто визначає, що щось перетинає і не реагує фізично). Ці функції можуть бути викликані кілька разів поспіль, якщо виявлено більше одного контакту під час оновлення фізики, тому в функцію передається параметр, що надає додаткову інформацію про зіткнення (координати, "особистість" вхідного об'єкта і т.д.).

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

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

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

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

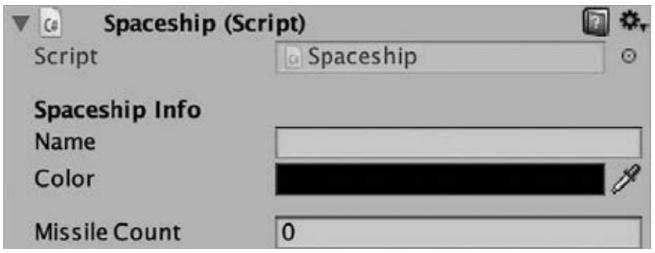
Unity створює мітку у вікні Inspector шляхом додавання пробілу скрізь, де в імені змінної зустрічається велика літера. Однак це виключно для відображення, і ви повинні завжди в коді використовувати ім'я змінної.

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

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

Наприклад:

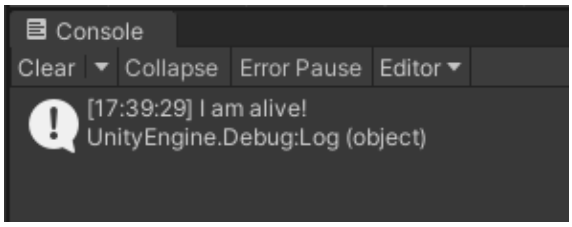
```
public class Spaceship : MonoBehaviour
{
[Header("Spaceship Info")]
public string name;
public Color color;

[Space]

public int missileCount;
}
```



Debug.Log це команда, яка просто виводить повідомлення на консольне виведення Unity. Наприклад, код:

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

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



Найпростішим і найпоширенішим є випадок, коли скрипту необхідно звернутися до інших компонентів, приєднаних до того ж *GameObject*. Компонент насправді є екземпляром класу, тому першим кроком буде отримання посилання на екземпляр компонента, з яким ви хочете працювати. Це робиться за допомогою функції *GetComponent*. Типово, об'єкт компонента зберігають в змінну, це робиться C# за допомогою наступного синтаксису:

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

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

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

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

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

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

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

Наприклад:

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

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

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

Пошук дочірніх GameObjects. Іноді ігрова сцена використовує декілька ігрових об’єктів одного типу, таких як вороги, шляхові точки та перешкоди. Їх може знадобитися

відстежувати за допомогою певного сценарію, який контролює або реагує на них (наприклад, усі шляхові точки можуть бути доступні для сценарію пошуку шляху).

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

У подібних випадках часто краще керувати набором GameObjects, роблячи їх усіма нащадками одного батьківського GameObject. Дочірні GameObjects можна отримати за допомогою батьківського компонента Transform (оскільки всі GameObjects неявно мають Transform)

Наприклад:

```
using UnityEngine;

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

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

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

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

Об'єкт або колекція об'єктів можуть бути знайдені за їх тегом, використовуючи функції GameObject.FindWithTag і GameObject.FindGameObjectsWithTag.

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

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

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

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

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

2. Завдання до практичної роботи 4

В проекті створеному у попередніх практичних роботах на одну зі сцен додайте пустий об’єкт та як дочірній об’єкт додайте джерело світло. Налаштуйте тип тп параметри джерела світла. Створіть новий скрипт та додайте його на пустий об’єкт. В скрипті напишіть код для пересування пустого об’єту с джерелом світла по сцені.

Тема 5. UI. Інтерфейс користувача в Unity
Практична робота 5. UI. Інтерфейс користувача в Unity

Мета роботи: познайомитися зі інтерфейсом користувача (UI), з об’єктом Canvas. Навчитися створювати інтерфейс користувача, використовувати різні елементи UI.

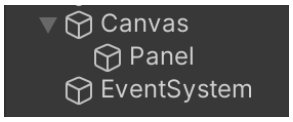
1. Теоретичний матеріал

Інтерфейс користувача – це сукупність засобів, за допомогою яких гравець взаємодіє з грою. Це не лише візуальні елементи, а й їхня функціональність. Хороший UI має бути інтуїтивно зрозумілим, легко сприйманим та забезпечувати зручний доступ до всіх необхідних функцій.

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

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

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



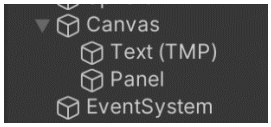
Область Canvas відображається у вигляді прямокутника у вікні Scene View. Це полегшує процес розташування елементів UI без потреби бачити ігрове вікно (Game View).

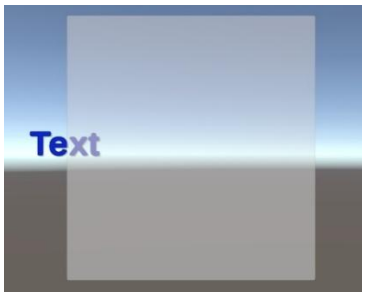
Разом із Canvas створюється Event System.

Event System (Система подій) – спосіб надсилання подій до об'єктів у програмі, заснований на введенні з клавіатури або миші; за допомогою дотиків або персональних пристроїв. Система складається з кількох компонентів, що працюють разом. Ця система за допомогою променів Unity визначає, що знаходиться під покажчиком.

Елементи UI на Canvas виникають у тому порядку, як вони розташовані в ієрархії. Перший дочірній елемент малюється першим, другий – за ним і таке інше. Якщо два елементи UI накладаються один на одного, доданий пізніше буде поверх того, що було додано раніше.

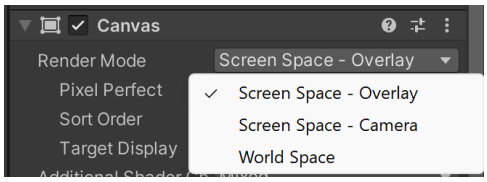
Наприклад:





Щоб змінити те, який елемент буде поверх інших, просто поміняйте місцями елементи в ієрархії шляхом перетягування. Порядком можна керувати за допомогою скриптингу, використовуючи такі методи компонента Transform: SetAsFirstSibling, SetAsLastSibling і SetSiblingIndex.

У полотна є *параметр Render Mode*, який визначає, де воно відображатиметься: у просторі екрана (screen space) чи ігрового світу (world space).



Режими відображення:

– Простір екрану – Перекриття (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.
– Простір екрану – Камера (Screen Space – Camera) – це схоже на режим «Простір екрана – Накладання», але в цьому режимі рендерингу полотно розміщується на заданій відстані перед вказаною камерою. Елементи інтерфейсу користувача рендеряться цією камерою, а це означає, що налаштування камери впливають на зовнішній вигляд інтерфейсу користувача.

Якщо для камери встановлено режим «Перспектива», елементи інтерфейсу користувача будуть рендеритися з перспективою, а ступінь спотворення перспективи можна контролювати за допомогою поля зору камери. Якщо розмір екрана змінюється, роздільна здатність або кут розрізу камери змінюється, полотно також автоматично змінює розмір відповідно.

– Простір ігрового світу (World Space) – при цьому режимі відображення Canvas поводиться так само, як і будь-який інший об'єкт на сцені.

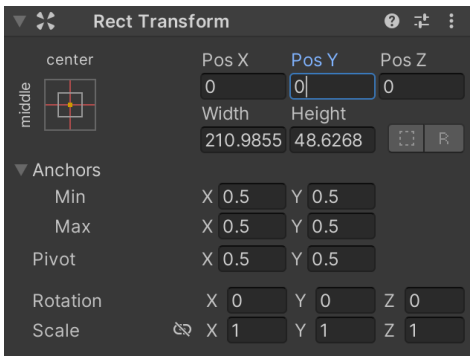
Розмір Canvas може бути заданий вручну за допомогою Rect Transform, а елементи інтерфейсу відображатимуться перед або за іншими об'єктами на сцені, залежно від їхнього тривимірного розташування.

Цей режим зручний для тих інтерфейсів, які передбачаються як частина ігрового світу.

Кожен елемент інтерфейсу користувача представлений у вигляді прямокутника для цілей макета. Цим прямокутником можна маніпулювати в режимі перегляду сцени за допомогою інструмента «Прямокутник» на панелі інструментів.

Інструмент «Прямокутник» використовується як для 2D-функцій Unity, так і для інтерфейсу користувача, і його можна використовувати навіть для 3D-об'єктів.

Rect Transform – це новий компонент перетворення, який використовується для всіх елементів інтерфейсу користувача замість звичайного компонента Transform.



Rect Transforms мають положення, обертання та масштаб, як і звичайні Transforms, але також мають ширину та висоту, які використовуються для визначення розмірів прямокутника.

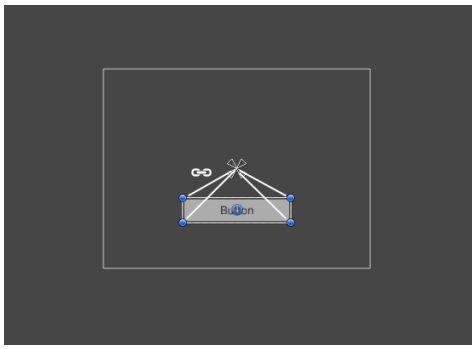
Зміни обертання, розміру та масштабу відбуваються навколо точки опори, тому положення точки опори впливає на результат обертання, зміни розміру або масштабування. Коли кнопка «Pivot» на панелі інструментів встановлено в режим «Pivot», точку опори прямокутного перетворення можна переміщувати в режимі Scene View.

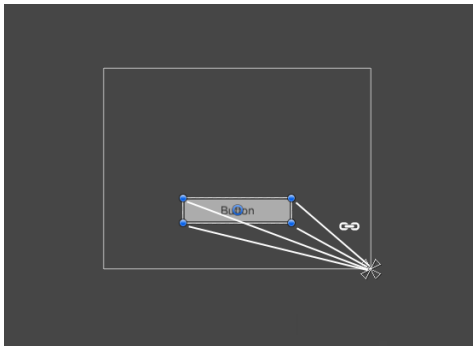
Rect Transforms включають концепцію макета, яка називається *прив'язками*. Прив'язки (Anchors) відображаються у вигляді чотирьох маленьких трикутних ручок у вікні перегляду сцени, а інформація про прив'язку також відображається в Інспекторі.

Якщо батьківський елемент прямокутного перетворення також є Rect Transform, дочірнє Rect Transform може бути прив'язане до батьківського Rect Transform різними способами.

Наприклад, дочірнє перетворення може бути прив'язане до центру батьківського елемента або до одного з кутів.

Наприклад:

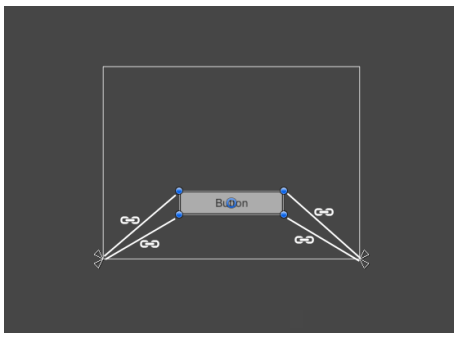




Прив'язка також дозволяє дочірньому об'єкту розтягуватися разом із шириною або висотою батьківського об'єкта.

Кожен кут прямокутника має фіксоване зміщення відносно відповідного йому прив'язувального елемента, тобто верхній лівий кут прямокутника має фіксоване зміщення відносно верхнього лівого прив'язувального елемента тощо.

Таким чином, різні кути прямокутника можуть бути прив'язані до різних точок батьківського прямокутника.



Положення якорів визначаються у частках (або відсотках) ширини та висоти батьківського прямокутника. 0,0 (0%) відповідає лівій або нижній стороні, 0,5 (50%) – середині, а 1,0 (100%) – правій або верхній стороні. Але якорі не обмежуються сторонами та серединою; їх можна прив’язати до будь-якої точки в межах батьківського прямокутника.

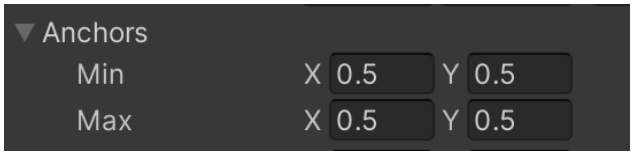
Ви можете перетягувати кожен з якорів окремо, або, якщо вони разом, ви можете перетягнути їх разом, клацнувши посередині між ними та перетягнувши. Якщо ви утримуєте клавішу Shift під час перетягування якоря, відповідний кут прямокутника переміститься разом з якорем.

В Інспекторі кнопку Anchor Preset button можна знайти у верхньому лівому куті компонента Rect Transform. Натискання цієї кнопки відкриває випадаюче меню «Початкові налаштування прив’язки». Звідси ви можете швидко вибрати один із найпоширеніших варіантів прив’язки.

Рисунок 5.1 – Приклад Anchor Preset

Ви можете прив’язати елемент інтерфейсу до боків або посередині батьківського елемента, або розтягнути його разом із розміром батьківського елемента. Горизонтальна та вертикальна прив’язка є незалежними.

Ви можете натиснути стрілку розгортання Anchors, щоб відобразити поля з номерами якорів, якщо вони ще не видно. Anchor Min відповідає нижньому лівому маркеру якоря в області перегляду сцени, а Anchor Max – верхньому правому маркеру.



Поля позиції прямокутника відображаються по-різному залежно від того, чи розташовані якорі разом (що створює фіксовану ширину та висоту), чи окремо (що призводить до розтягування прямокутника разом з батьківським прямокутником).

Сучасні ігри та програми часто потребують підтримки широкого спектру різних дозволів екрану, і особливо цієї можливості потребують інтерфейси цих ігор. Система створення інтерфейсів в Unity забезпечена рядом різних інструментів для реалізації цих можливостей, які можна комбінувати між собою масою різних способів.

Елементи інтерфейсу за замовчанням прив'язані до центру батьківського прямокутника. Це означає, що вони зберігають постійне зміщення щодо центру. Якщо з цією настройкою роздільна здатність була змінена під альбомне співвідношення сторін, кнопки можуть випасти зі своїх прямокутних областей, де вони спочатку мали бути розташовані.

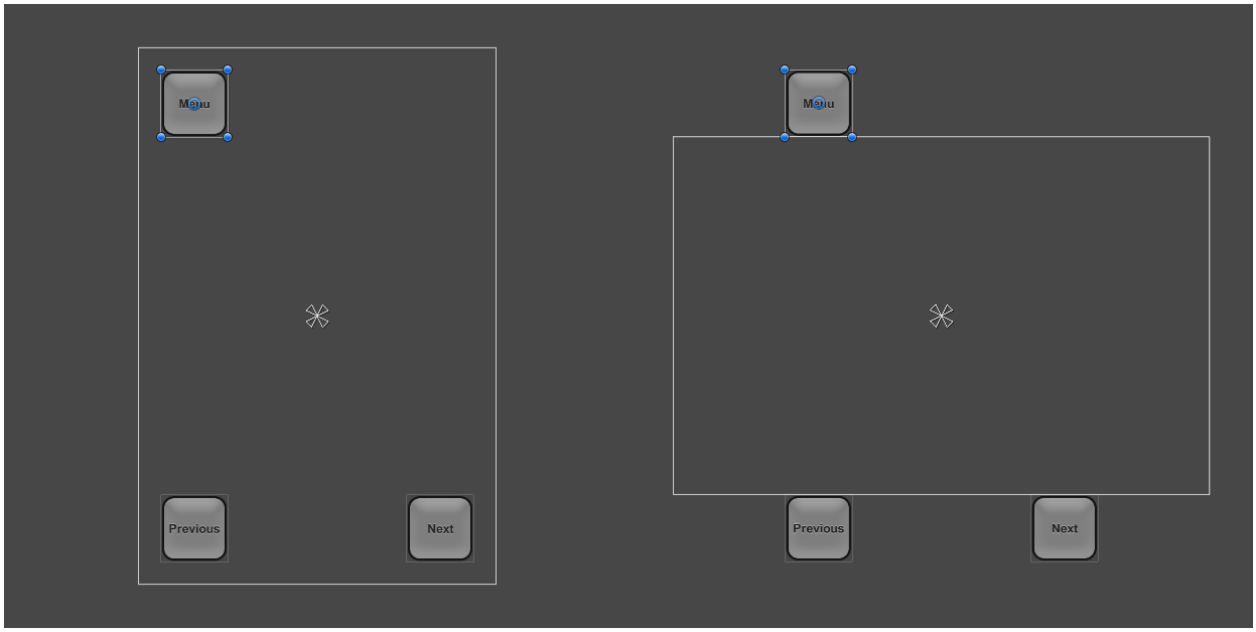


Рисунок 5.2 – Приклад зміни роздільної здатності

Одним із способів зберегти розташування кнопок в області екрану є зміна компонування таким чином, щоб місця розташування були пов'язані з їх відповідними кутами на екрані. Прив'язка лівої верхньої кнопки може бути також встановлена в лівому верхньому кутку при використанні в інспекторі списку Anchors Preset (набори прив'язок), або шляхом перетягування трикутних ручок прив'язок у видовому вікні сцени (Scene View). Так само наприклад, прив'язки для лівої нижньої і правої нижньої кнопок можуть бути виставлені в лівий нижній і правий нижній кут відповідно. Як тільки кнопки були прив'язані до своїх кутів, то при подальших змінах роздільної здатності екрану та співвідношень сторін вони зберігатимуть свою позицію щодо цих кутів.

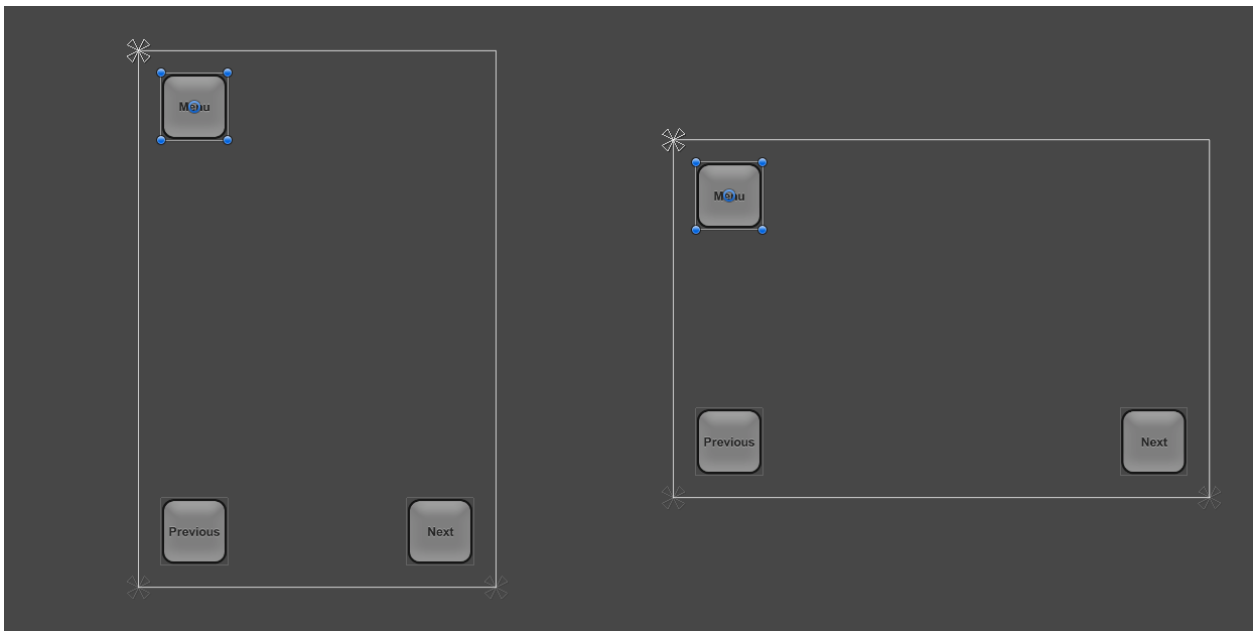


Рисунок 5.3 – Приклад зміни компонування

Коли роздільна здатність екрана змінюється на більшу і меншу щодо поточного, кнопки повинні зберігати своє початкове розташування щодо кутів, до яких вони прив'язані. Однак, зберігаючи свою оригінальну роздільну здатність, задану в пікселях, вони можуть ставати як більше так і менше, відповідаючи пропорціям поточної роздільної здатності екрана. Все це може бути, а може і не бажано, залежно від того, як ви хочете, щоб ваш інтерфейс реагував на зміну роздільної здатності екрану.

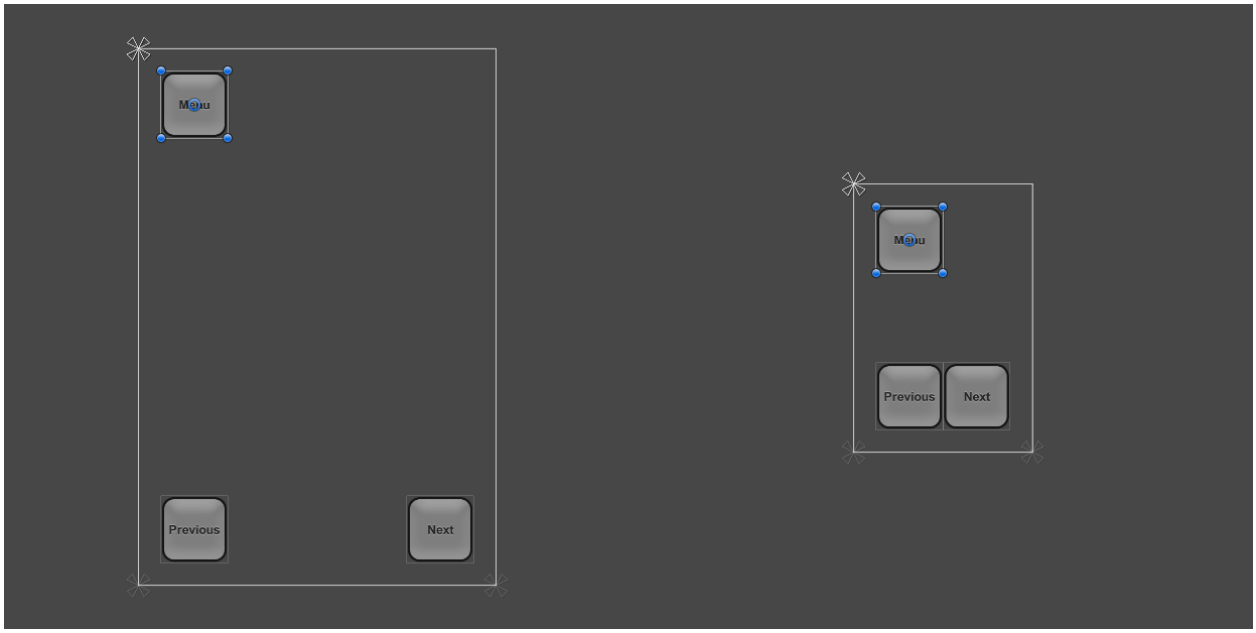


Рисунок 5.4 – Приклад зміни масштабу

У компоненті Canvas Scaler можна встановити його UI Scale Mode в Scale With Screen Size. У цьому режимі масштабування ви можете визначити, який дозвіл використовувати як базовий. Якщо поточна роздільна здатність більша або менша за базовий, фактор масштабування компонента Canvas встановлюється відповідно так, щоб всі елементи інтерфейсу масштабувалися у більшу або меншу сторону разом з роздільною здатністю екрана.

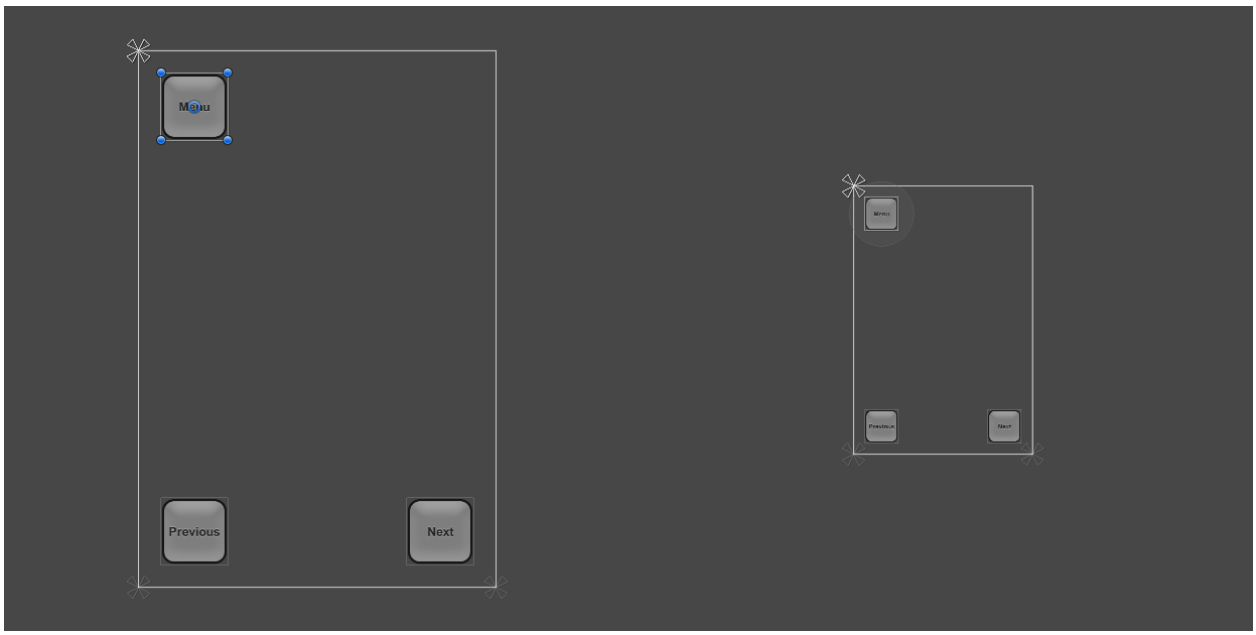


Рисунок 5.5 – Приклад роботи режиму Scale With Screen Size

Компонент Canvas Scaler має властивість під назвою Match, яка може прийняти значення, що дорівнює 0 (ширина), 1 (висота) або будь-яке значення, що лежить в межах між 0 і 1. За замовчуванням воно встановлено в 0, що означає те, що поточна ширина екрану відповідає базовій ширині базової роздільної здатності. Якщо властивість Match має значення не дорівнює 0.5, воно порівнюватиме поточну ширину з базовою шириною, поточну висоту з базовою висотою, і вибере коефіцієнт масштабу близький і до того і до іншого дозволу.

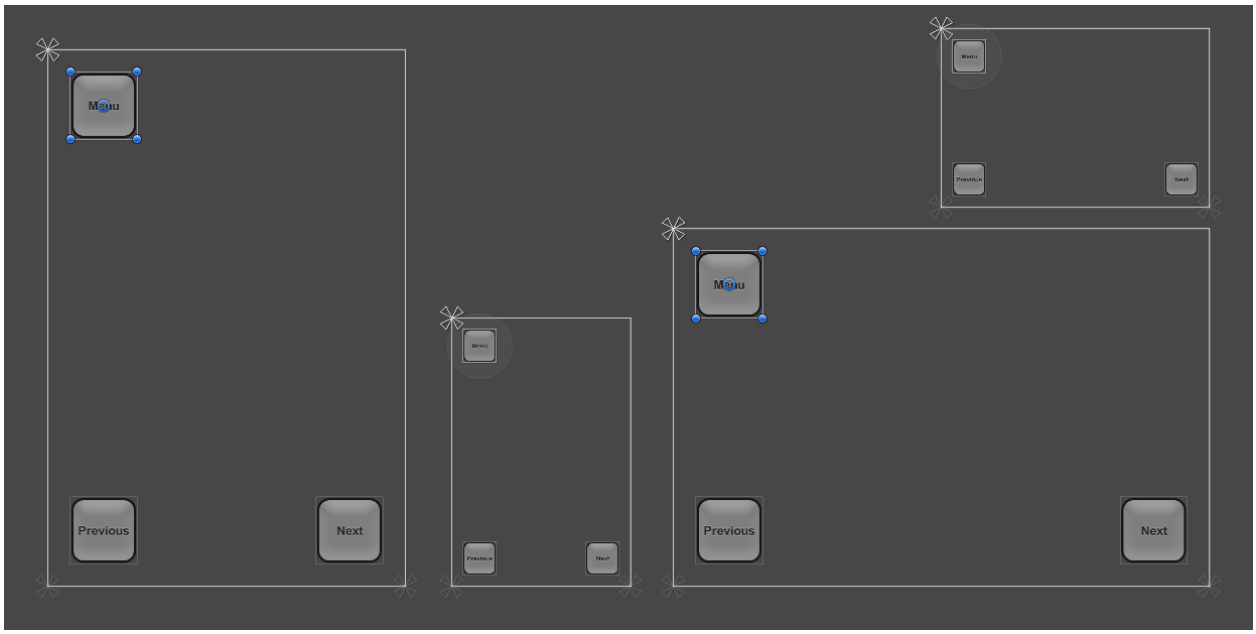


Рисунок 5.6 – Приклад роботи з властивістю Match

2. Завдання до практичної роботи 5

В проекті створеному у попередніх практичних роботах на одну зі сцен додайте інтерфейс користувача (UI). Додайте до UI різні елементи, налаштуйте їх параметри та додайте свої прив'язки. Налаштуйте створений UI до зміни роздільної здатності.

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

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

1. Теоретичний матеріал

У Unity, коли йдеться про візуальні компоненти в UI, мають на увазі ті елементи, які відповідають за відображення та роботу користувача на екрані. Всі вони працюють усередині Canvas і використовують Canvas Renderer для малювання.

Основні візуальні UI-компоненти:

– Panel – простий прямокутний контейнер із Image. Зазвичай використовується як тло для інших елементів;
– Image – показує спрайти, іконки, фонові елементи;
– Text (або TextMeshPro - TMP Text) - відображає текст;
– Raw Image – показує текстуру безпосередньо (наприклад, потік з камери чи відео);
– Button – складовий UI-елемент, зазвичай містить Image + Text;
– Slider – повзунок (для гучності, прогресу тощо). Складається з фону, заповненої частини та «бігунка»;
– Scrollbar – вертикальна або горизонтальна смуга прокручування;
– Toggle – перемикач;
– Dropdown – список, що випадає для вибору з декількох опцій;
– Input Field – дозволяє користувачеві вводити текст із клавіатури.

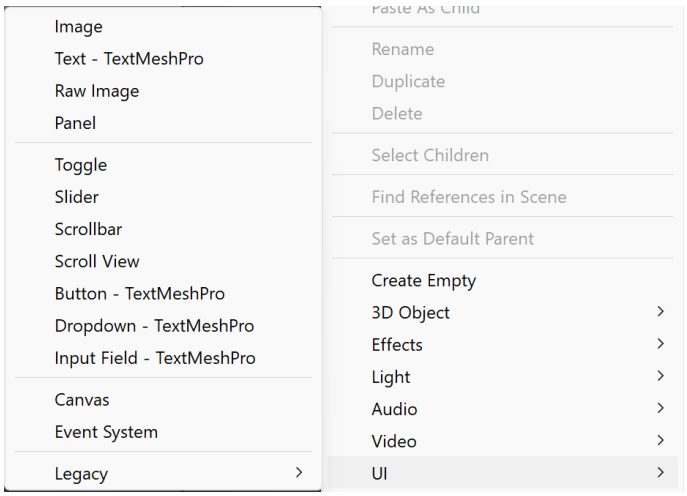


Рисунок 6.1 – Приклад меню для вибору візуальних UI-компонентів

Компонент Text, також відомий як Label, має область для введення тексту, який буде відображено. Є можливість задати шрифт, його стиль, розмір та здатність відображення Rich Text.

Існують опції для встановлення параметрів вирівнювання; налаштування горизонтального та вертикального переповнення, які керують поведінкою тексту, коли він не влазить по ширині або висоті у відведений йому прямокутник.

TextMeshPro – це потужний інструмент для роботи з текстом у Unity, який дозволяє покращити візуальну якість та гнучкість текстових елементів.

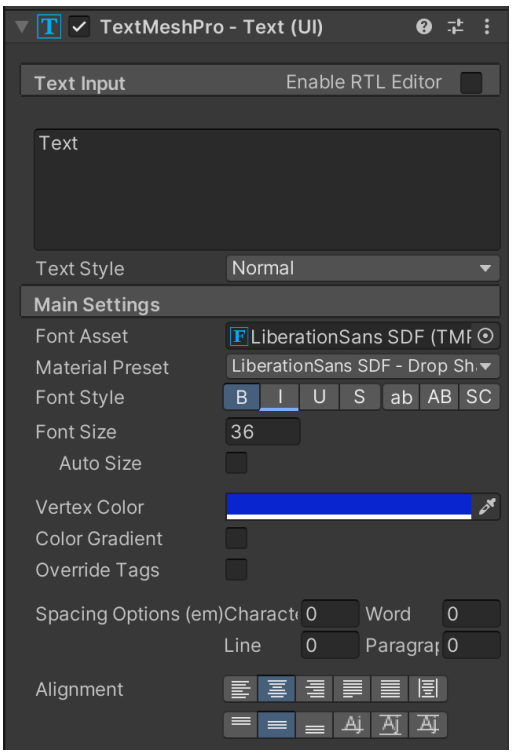


Рисунок 6.2 – Приклад компонента TextMeshPro

При першому використанні TextMeshPro у проекті Unity вам буде запропоновано імпортувати пакети TMP Essentials та Examples & Extras. Переконайтеся, що ви імпортували їх, щоб отримати доступ до всіх функцій.

Текст для елементів інтерфейсу користувача та текстових сіток може містити різні стилі та розміри шрифтів.

Форматований текст (Rich text) підтримується як для системи інтерфейсу користувача, так і для застарілої системи графічного інтерфейсу користувача. Класи Text, GUIStyle, GUIText та TextMesh мають налаштування Rich Text, яке вказує Unity шукати теги розмітки в тексті.

Функція Debug.Log також може використовувати ці теги розмітки для покращення звітів про помилки з коду. Теги не відображаються, але вказують на зміни стилю, які потрібно застосувати до тексту.

Система розмітки тексту в Unity була створена на основі HTML, однак сувора сумісність зі стандартним HTML не передбачається. Основна ідея полягає в тому, що фрагменти тексту можна укладати в пару тегів, що узгоджуються один з одним.

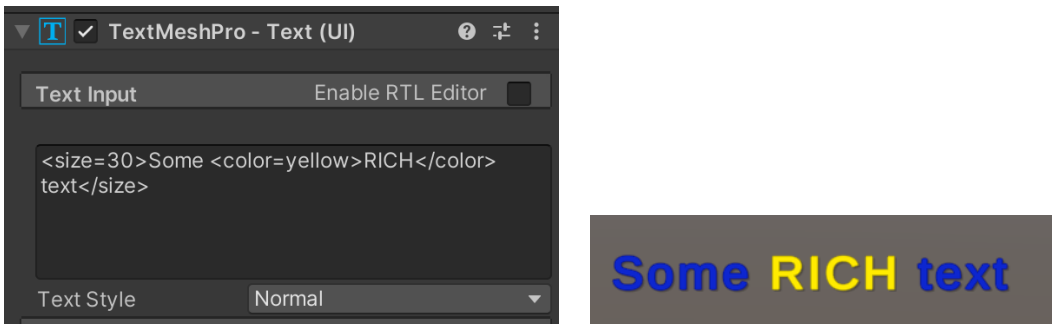
Тегами називаються фрагменти тексту, укладені в кутові дужки: <b>. Текст усередині дужок визначає ім'я тега (у разі b). Тег в кінці фрагмента (закриває) має те саме ім'я, що й тег на початку (відкриває), проте в ньому також міститься символ косої риси /.

```
We are <b>not</b> amused
We are not amused
```

Теги, які підтримуються:

– b – виділяє текст жирним шрифтом;
– i – виділяє текст курсивом;
– size – встановлює розмір тексту відповідно до значення параметра, заданого в пікселях;
– color – встановлює колір тексту відповідно до значення параметра.

Наприклад:

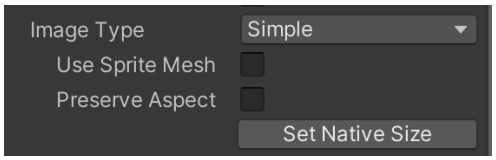


Компонент Image в Unity UI використовується для відображення картинки або кольорової області в інтерфейсі. Це один із найбільш базових візуальних елементів UI, і на його основі будуються багато інших елементів (кнопки, панелі, іконки).

Основні налаштування:

– Source Image – тут вибирається спрайт (наприклад, іконка, фон, картинка);
– Color – визначає колір або прозорість. Якщо є картинка, колір накладається як фільтр (Tint). Якщо картинки немає - малюється просто прямокутник потрібного кольору;
– Material – дозволяє призначити власний матеріал/шейдер;
– Image Type – визначає, яким чином доданий спрайт буде відображено, можливі варіанти:
  • Simple – масштабує весь спрайт рівноцінно.
  • Sliced – використовує поділ спрайту на 3x3 так, щоб масштабування не спотворювало кути, а розтягнута була лише центральна частина.
  • Tiled – схоже на Sliced, але замість розтягування центральної частини вона повторюється (заповнюється тайлами). Для спрайтів без будь-яких кордонів весь спрайт буде заповнений тайлами.
  • Filled – відображає спрайт так, як це робить Simple, за винятком того, що він заповнює спрайт, починаючи з певної точки у встановленому напрямку та кількості заданим методом.

Опция Set Native Size, доступная, когда выбран Simple или Filled, устанавливает изображению изначальный размер спрайта.



Елемент керування Button реагує на натискання користувача та використовується для ініціювання або підтвердження дії. Склададється з:

Розглянемо основні властивості кнопки:

– Interactable – чи прийматиме цей компонент вхідні дані?
– Transition – властивості, що визначають візуальну реакцію елемента керування на дії користувача.
– Navigation – властивості, що визначають послідовність елементів керування.

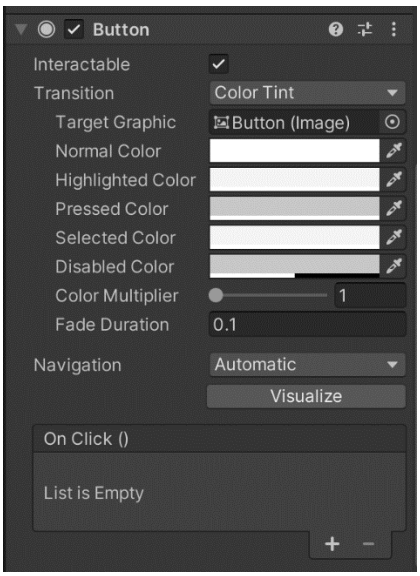
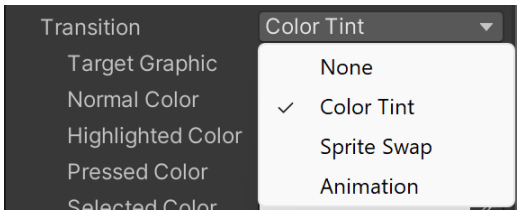


Рисунок 6.3 – Приклад компонента Button

Властивість Transition може приймати такі значення:

– None – цей параметр дозволяє кнопці взагалі не впливати на стан.
– Color Tint – змінює колір кнопки залежно від її стану. Можна вибрати колір для кожного окремого стану. Також можна встановити тривалість переходу між різними станами. Чим більше число, тим повільніше буде переходити між кольорами.
– Sprite Swap – дозволяє відображати різні спрайти залежно від поточного стану кнопки, спрайти можна налаштовувати.
– Animation – дозволяє створювати анімацію залежно від стану кнопки. Для використання переходу анімації має існувати компонент-аніматор.

Важливо переконатися, що рух кореня вимкнено. Щоб створити контролер анімації, натисніть «Згенерувати анімацію» (або створіть свій власний) і переконайтеся, що контролер анімації додано до компонента-аніматора кнопки.



При виборі значення Color Tint маємо такі властивості:

– Target Graphic – графіка, що використовується для компонента взаємодії.
– Normal Color – звичайний колір елемента керування.
– Highlighted Color – колір елемента керування, коли він виділений.
– Pressed Color – колір елемента керування, коли він натиснутий.
– Disabled Color – колір елемента керування, коли він вимкнений.
– Color Multiplier – це множить колір відтінку для кожного переходу на його значення. За допомогою цього ви можете створювати кольори більше 1, щоб зробити кольори (або альфа-канал) на графічних елементах, базовий колір яких менший за білий (або менший за повну альфа-канал).
– Fade Duration – час у секундах, необхідний для плавного переходу з одного стану в інший.

Властивість Navigation може приймати такі значення:

– Navigation – параметри навігації стосуються способу керування навігацією елементів інтерфейсу користувача в режимі відтворення.
– None – навігація за допомогою клавіатури відсутня. Також гарантує, що фокус не переходить у разі натискання/натискання на нього.
– Horizontal – горизонтальна навігація.
– Vertical – вертикальна навігація.
– Automatic – автоматична навігація.
– Explicit – у цьому режимі ви можете явно вказати, куди переміщується елемент керування для різних клавіш зі стрілками.
– Visualize – вибір опції «Візуалізація» надає візуальне представлення навігації, яку ви налаштували у вікні сцени.

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.

Основні функції:

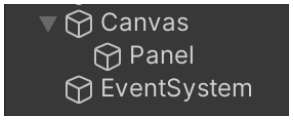
– Установки – дозволяє змінювати настройки гри, такі як мова, гучність, керування або налаштування графіки.
– Вибір гри – дозволяє розпочати нову одиночну або розраховану на багато користувачів гру, приєднатися до існуючого сервера або вибрати рівень.
– Пауза – тимчасова зупинка гри для виконання дій, не перериваючи ігровий процес.
– Інші опції – залежно від гри, може включати доступ до розділів з сюжетом, інвентарем персонажа, місіями або іншими можливостями.

Види ігрових меню:

– Головне меню – відображається під час запуску гри та містить основні опції, такі як початок гри, налаштування, вихід тощо.
– Меню паузи – з'являється під час гри і дозволяє поставити її на паузу, щоб змінити налаштування, зберегти гру або вийти.
– Внутрішньоігрове меню – може відображатися безпосередньо в ігровому просторі та містити специфічну інформацію або команди, наприклад меню карти або інвентарю.

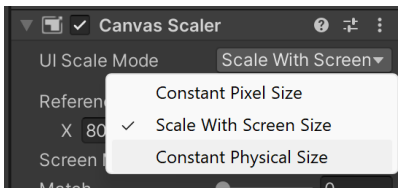
Для створення головного меню гри створимо нову сцену в проекті.

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.

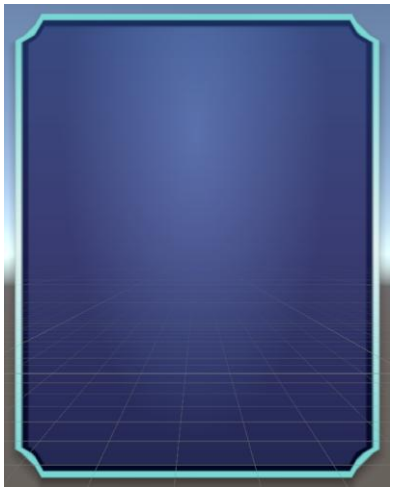
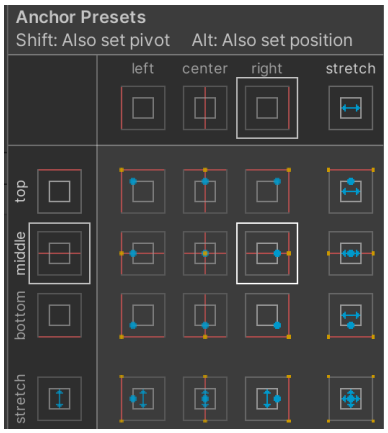


Рисунок 6.4 – Приклад фону головного меню

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

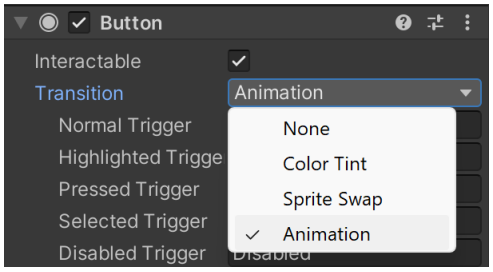


Створюємо дочірнім об’єктом до панелі MainMenu компонент Button. Компонент Button складається з компонента Image та Text. Додаємо до компонента Image зображення для представлення кнопки, в компоненту Text – запишемо текст Start та змінимо колір тексту та зробимо інші налаштунки. Змінимо назву кнопки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

Рисунок 6.5 – 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. Color Tint, це коли вибираємо один спрайт для кнопки, але змінюємо лише кольори. Тобто колірний перехід має кожен стан. Можна налаштувати кольори для нормального стану, при наведенні курсору, при натисканні, для виділення та для неактивного стану у відповідних полях. Замінимо це значення на Animation. Значення Animation дозволяє створити повноцінний контролер анімації та встановити окремі анімації для різних станів.



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.

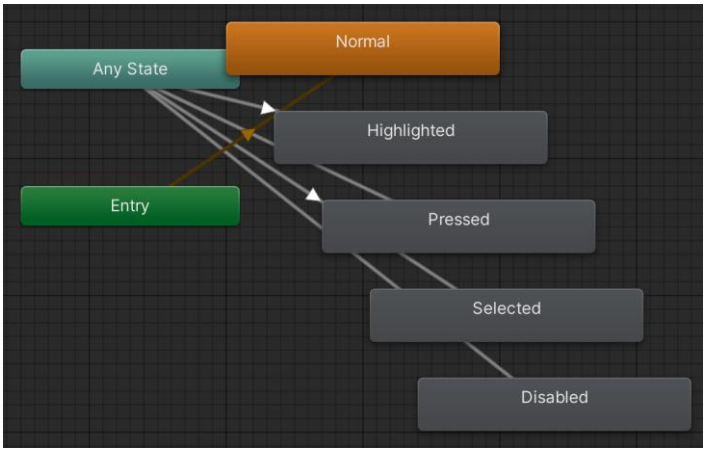
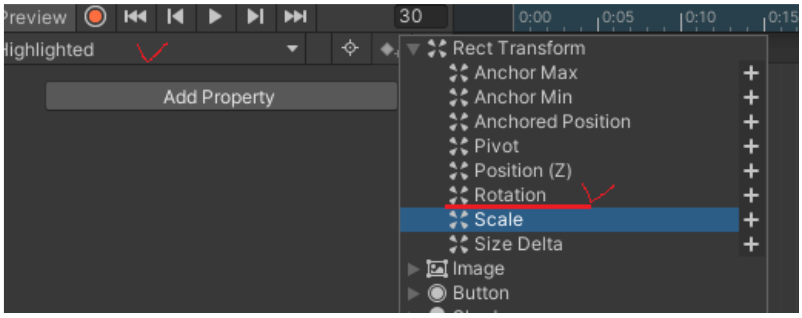


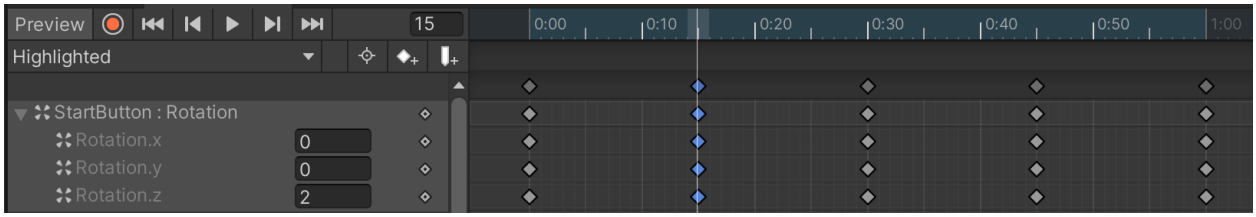
Рисунок 6.6 – Приклад анімаційного контролера у вікні Animator

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

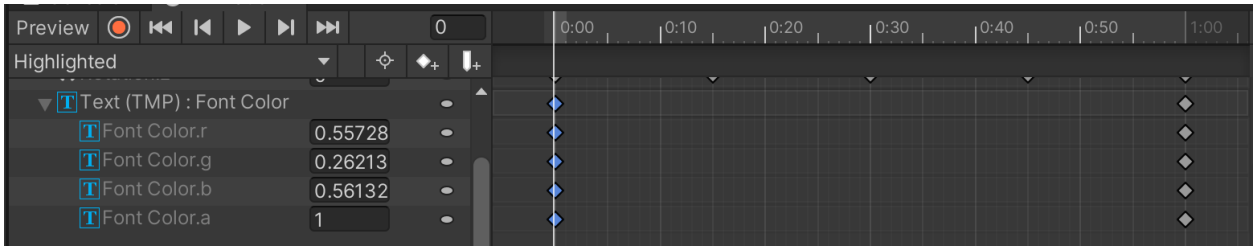
Обираємо Highlighted, натискаємо кнопку Add Property та обираємо Rect Transform → Rotation.



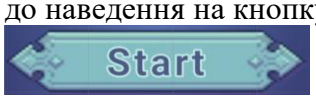
Розставляємо 5 keyframes, за часом 0:00, 0:15, 0:30, 0:45, 1:00. На час 0:00, 0:30 та 1:00 залишаємо Rotation.x, Rotation.y, Rotation.z рівними нулю, а на час 0:15 напишемо Rotation.z = 2, на час 0:45 напишемо Rotation.z = -2.



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

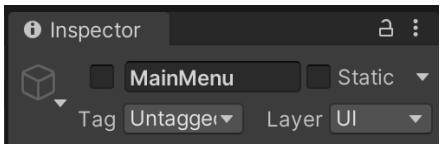
Зробимо 2 копії кнопки, що вийшла та розставимо по вертикалі на панелі. Змінимо текст на кнопках. В результаті отримаємо:

Рисунок 6.7 – Приклад початкового варіанта головного меню

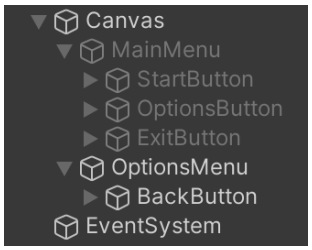
Отже маємо три кнопки: Start – при натисканні якої повинна починатися гра, Options – при натисканні якої повинна з’являтися додаткове меню для налаштувань та Exit – для виходу з програми.

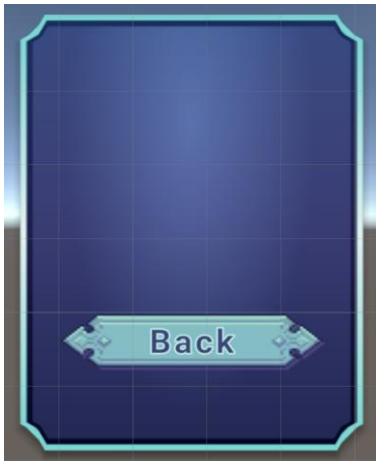
Створимо додаткове меню для налаштувань. В нашій версії воно буде мати тільки одну кнопку Back. Для цього виділяємо елемент MainMenu в ієрархії та робимо його

дублікат. Змінюємо ім’я дубліката на OptionsMenu. Для зручності тимчасово зробимо елемент MainMenu не активним.



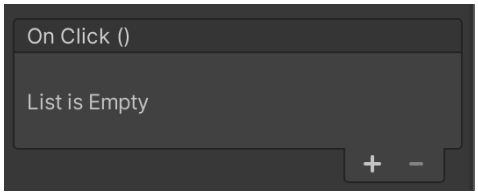
Якщо в ієрархії розкрити елемент OptionsMenu, то він має у підпорядкуванні три кнопки. Видалимо перші дві, а в останній кнопці змінимо текст на Back:

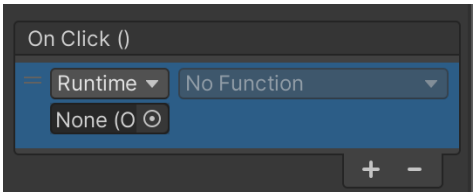




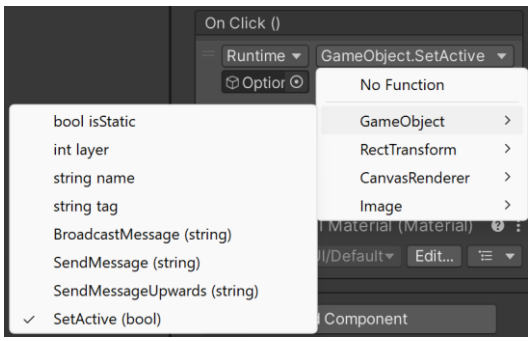
Отже, додаткове меню для налаштувань готово. Необхідно додати функціональність оскільки при натисканні кнопок нічого не відбувається.

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.



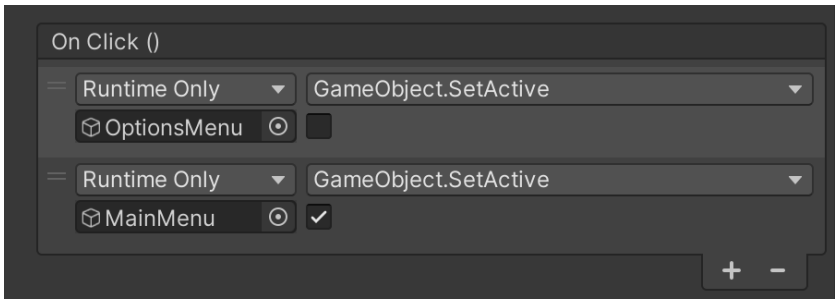


У нижне вікно перетягуємо об’єкт OptionsMenu, тоді справа в полі відкриються доступні функції над цим об’єктом. Обираємо GameObject → SetActive(bool)



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.

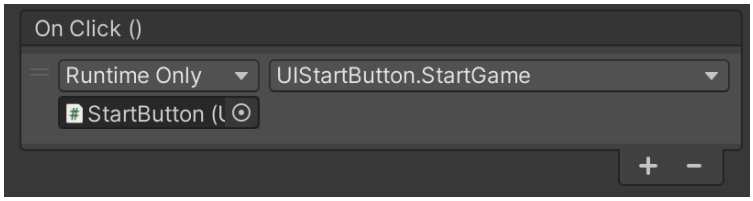
Якщо запустити проект, при натисканні кнопки Options з'явиться додаткове меню з однією кнопкою Back при натисканні на яку, повертається головне меню.

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 6

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

Тема 7. UI. Вікно інвентаря. Зовнішній вигляд вікна
Практична робота 7. UI. Вікно інвентаря. Зовнішній вигляд вікна

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

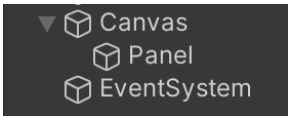
Інвентар – це спливаюче меню, яке гравець використовує для керування предметами, які він має при собі.

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 7.1 – Приклад вікна інвентаря в іграх

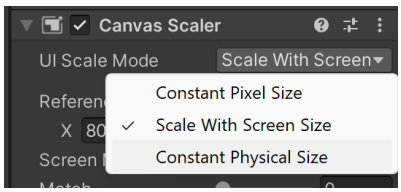
Для створення вікна інвентаря гри створимо на сцені гри новий Canvas.

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.

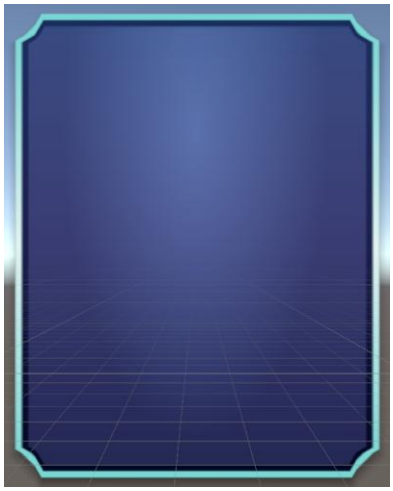
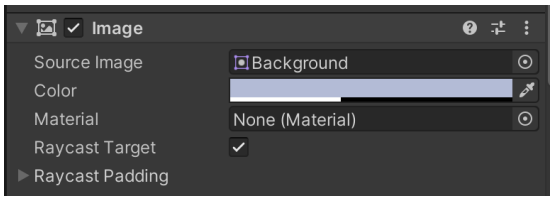
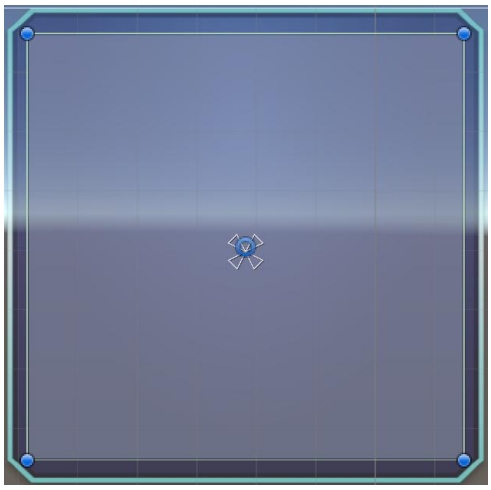


Рисунок 7.2 – Приклад фону вікна інвентаря

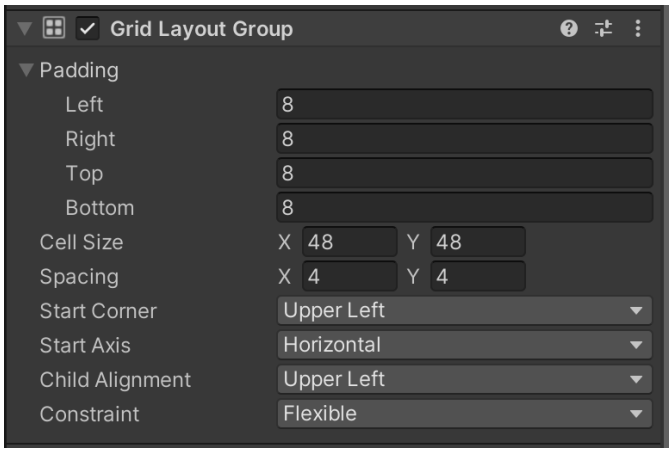
Створюємо дочірнім об’єктом до панелі InventiryWindow компонент Panel та назвемо його ContentView. Саме цю панель будемо розбивати на осередки, в яких будемо показувати картинки об’єктів які підібрали у грі. В компоненті Image змінимо колір фону та зробимо його частково прозорим.



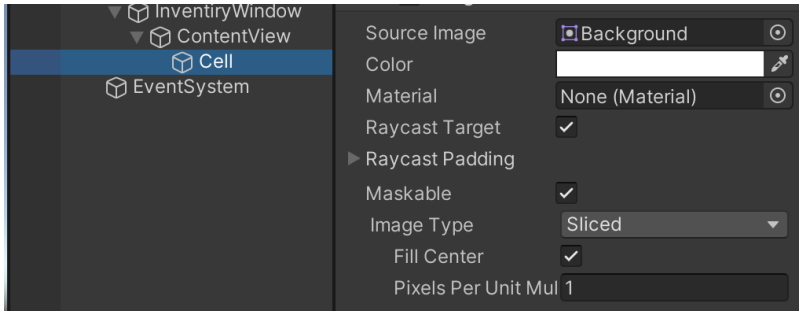


Додаймо до панелі ContentView компонент Grid Layout Group.

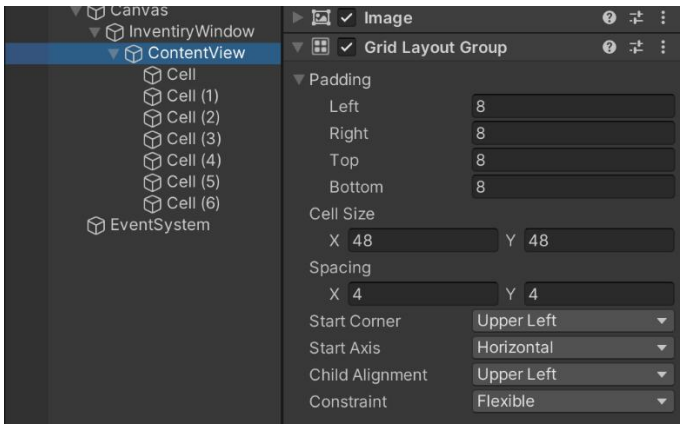
Grid Layout Group в Unity – це компонент, який використовується для автоматичного розміщення дочірніх компонентів в рівномірну мережу, де всі клітинки мають однаковий розмір. Розмір та міжклітинкова різниця контролюються компонентом Grid Layout Group самостійно, а дочірні компоненти не впливають на їхні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

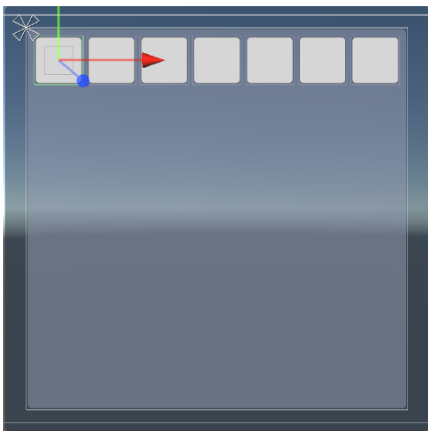


Створюємо дочірнім об’єктом до панелі ContentView компонент Image та назвемо його Cell.

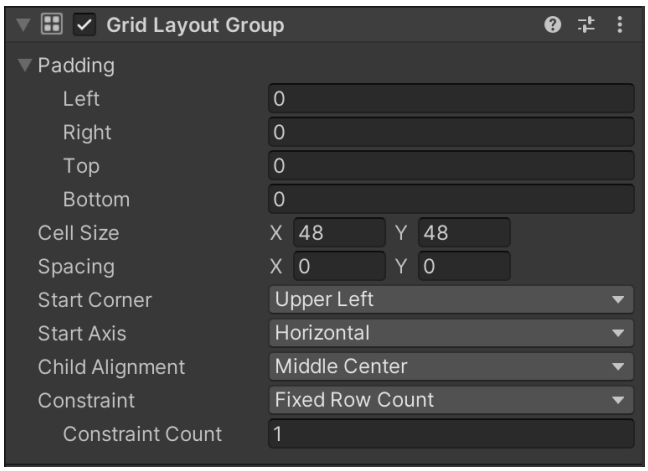


Підберемо значення Padding, Cell Size та Spacing в компоненті Grid Layout Group, щоб в панелі рівномірно розмістилось ціле значення осередків. Для цього тимчасово створіть декілька об’єктів Cell.

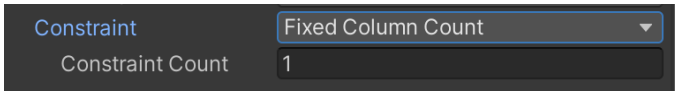




У подальшому до Cell будемо додавати картинку предмету, тому додаймо до об’єкту Cell компонент Grid Layout Group для найкращого розміщення картинки. В компоненті Grid Layout Group зробимо такі налаштування:



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.

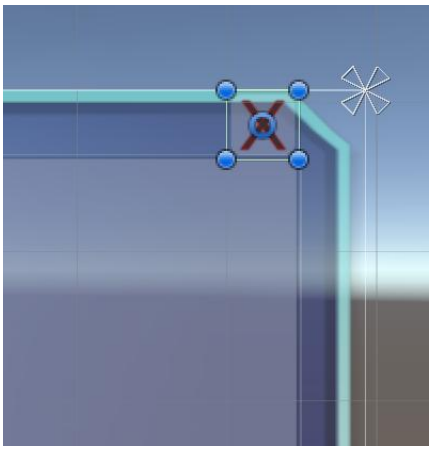
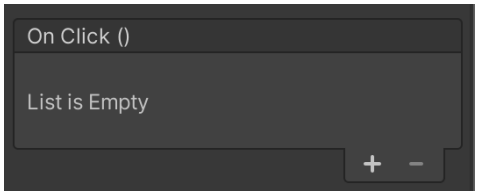
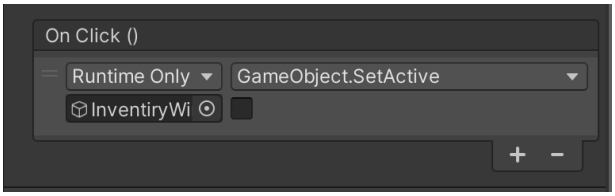


Рисунок 7.3 – Приклад кнопки закриття вікна інвентарю

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.





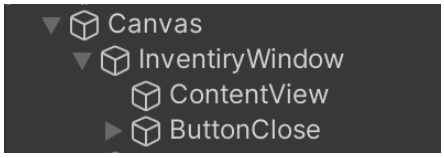
У нижне вікно перетягуємо об'єкт InventiryWindow, тоді справа в полі відкриються доступні функції над цим об'єктом. Обираємо GameObject → SetActive(bool). Рядом з полем з об'єктом InventiryWindow створиться прапорець. Якщо обрати прапорець, то вікно буде

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.



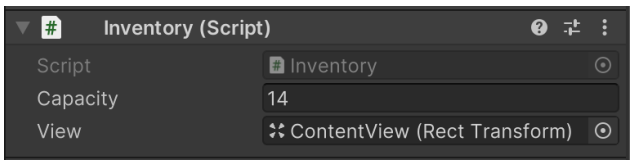
Створюємо скрипт Inventory та додаємо його як компонент до об’єкту InventiryWindow. Для створення необхідної кількості осередків у ContentView створимо поле capacity та поле view для посилання на ContentView. Масив content зберігатиме створені осередки.

```
[SerializeField]
private int capacity;

[SerializeField]
private Transform view;

public static Cell[] content;
```

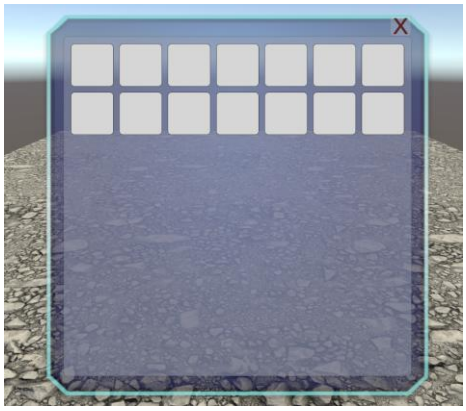
Створимо осередки та заповнимо масив content:

```
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:



2. Завдання до практичної роботи 7
Створити новий проект та сцену. Створити вікно інвентаря.

Тема 8. UI. Вікно інвентаря. Зберігання предметів
Практична робота 8. UI. Вікно інвентаря. Зберігання предметів

Мета роботи: познайомитися зі вікном інвентаря в іграх, та способом зберігання об’єктів у ньому. Навчитися створювати вікна інвентаря в іграх.

1. Теоретичний матеріал

У попередній лекції було створено вікно інвентаря з додаванням пустих осередків. Додаймо можливість показувати в осередках зображення предметів. Нехай маємо такі зображення:



Створимо в Assets папку Sprites та додаймо ці зображення 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.

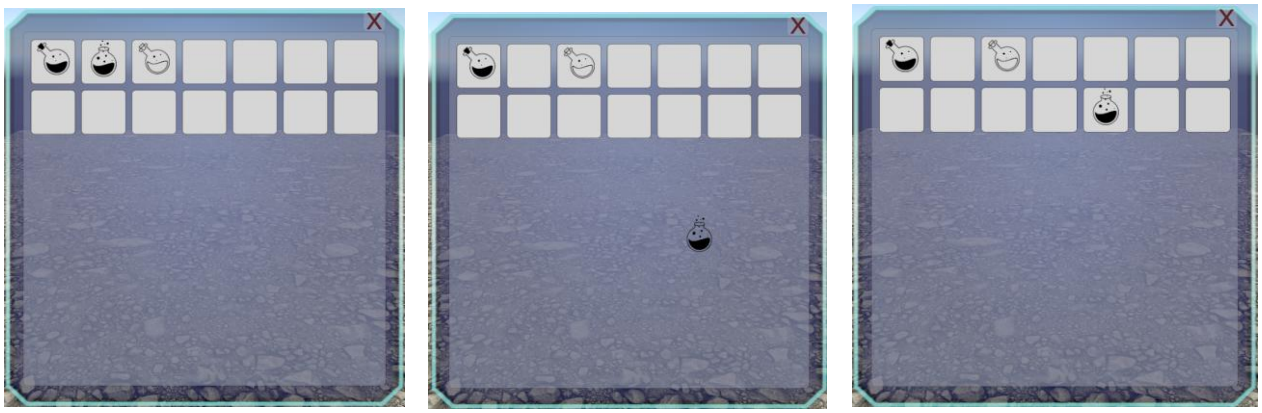


Рисунок 8.1 – Приклад вигляду вікна інвентарю при переміщенні картинок

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.

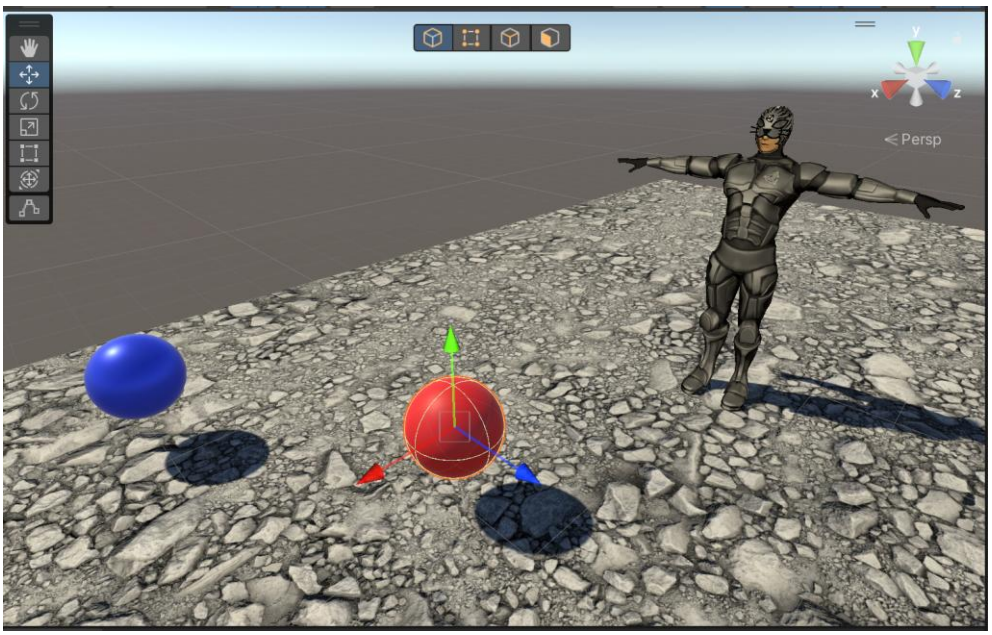


Рисунок 8.2 – Приклад вигляду сцени

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 8

В проекті, що створили на попередній практичній роботі, створити на сцені персонаж та скрипт для його руху, об’єкти, які може підібрати персонаж та написати скрипти для поміщення їх до вікна інвентарю.

Тема 9. UI. Додаткові ігрові вікна
Практична робота 9. UI. Додаткові ігрові вікна

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

Ігри – це захоплююче заняття, яке допомагає нам відволіктися від реальності та поринути у світ фантазії та пригод. Але коли ми запускаємо гру на комп'ютері або мобільному пристрої, виникає безліч різних ігрових вікон, які можуть плутати або викликати питання. Розглянемо як називаються ігрові вікна і навіщо вони призначені.

Назви ігрових вікон можуть відрізнятися в різних іграх, але в основному вони виконують одні й ті самі функції – забезпечують кращу ігрову платформу, зручність керування та навігації, а також допомагають взаємодіяти з гравцями по всьому світу. Крім основних вікон, в іграх є додаткові вікна, які надають додаткові можливості та інформацію для більш повного та цікавого ігрового процесу.

Основні ігрові вікна:

- Головне вікно гри – це головне вікно гри, в якому відображається весь ігровий процес. Тут ви керуєте своїм персонажем або об'єктом, взаємодієте з іншими гравцями та виконуєте різноманітні ігрові завдання.
- Вікно налаштувань – тут можна налаштувати графіку гри, управління, звукові ефекти та інші параметри, щоб плавно і комфортно грати.
- Інвентарне вікно – у цьому вікні ви можете переглядати та керувати своїм інвентарем, тобто набором предметів та ресурсів, які ви зібрали у грі. Тут ви можете вибрати потрібні речі для використання чи продажу.
- Вікно завдань – якщо у грі передбачені завдання або квести, то вони відображаються у спеціальному вікні, де ви можете побачити поточні завдання, їх опис та прогрес виконання.
- Чат – багато онлайн-ігор мають чат, в якому ви можете спілкуватися з іншими гравцями, ставити питання, узгоджувати стратегію і просто насолоджуватися спілкуванням.

Додаткові ігрові вікна:

- Вікно магазину – якщо гра має внутрішньоігрову валюту або магазин, ви можете відкрити спеціальне вікно, де можна придбати предмети, поліпшення або додатковий контент.
- Вікно карти – деякі ігри надають карту ігрового світу, в якій ви можете переглядати своє становище, можливості переміщення та цікаві місця.
- Вікно досягнень – тут відображаються всі важливі досягнення та нагороди, які ви здобули у грі, щоб ви могли похвалитися своїми успіхами.
- Вікно лідерів – у режимах гри змагання можна переглянути таблицю лідерів, де відображається рейтинг гравців та їх досягнення.
- Вікно довідки – якщо у грі виникли труднощі або питання, ви можете відкрити вікно довідки, де знайдете відповіді на різні питання та рекомендації щодо проходження.
- Вікно програшу або перемоги – вікно в якому повідомляється, що ви виграли або програли і надає подальші дії в ситуації, що склалася.

Різновиди ігрових вікон:

- Вікно діалогу – це один із найпоширеніших типів ігрових вікон. Воно відкривається при взаємодії з персонажами гри та дозволяє гравцеві вибирати діалогові варіанти, ставити запитання чи отримувати інформацію.

- Меню — це вікно гри, яке містить різні настройки та параметри гри. Тут гравець може змінювати керування, налаштовувати графіку, вибирати рівень складності тощо.
- Інвентар — це вікно, де відображається список предметів, які гравець зібрав протягом гри. Тут він може керувати своїм інструментом — вибирати, використовувати, продавати чи купувати предмети.
- Карта - це ігрове вікно, яке дозволяє гравцеві переглядати ігровий світ та його локації. Тут він може відзначати цікаві місця, знаходити шлях до наступної мети чи дослідити довкілля.

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.

Еволюція інтерфейсу користувача з часом призвела нас до кількох гілок його розвитку. Зі зростанням популярності та складності ігор інтерфейс ставав їх важливою частиною. Поява нових жанрів народжувало і нові типи інтерфейсу: потрібно відображати більше інформації в рамках нових ігрових механік, тому старого доброго лічильника очок і таймера поверх геймплею вже не вистачало.

Згодом ці галузі розвитку були систематизовані в кілька типів інтерфейсів, які сьогодні ми можемо спостерігати в тих чи інших іграх. Залежно від того, чи є інтерфейс частиною оповідання, а також чи є він частиною ігрового простору, ігрові інтерфейси поділяються на такі типи:

- Дієгетичні – коли, наприклад, швидкість автомобіля передається через спідометр, що у кабіні вантажівки. Інтерфейс, який бачить та з яким може взаємодіяти персонаж гравця. Він дозволяє отримати більш наративний досвід у рамках ігрового світу;
- Мета – наприклад, краплі крові та води на «візорі». По суті – постпроцеси, що дозволяють трансформувати досвід персонажа у досвід гравця через ще один шар взаємодії;
- Просторові – це різні виділені області, інтерактивні об'єкти у світі, траєкторії польоту гранат, проекційні повідомлення тощо, створені для гравця, але з персонажа. Для персонажа їх немає і він не може з ними взаємодіяти;
- Недієгетичні – той же витратомір/спідометр і показники цілісності вантажівки – по суті віджети, розташовані поверх геймплей. Дозволяє просто та швидко зчитувати інформацію, але ламає наратив і дуже рідко пояснюється через ігрову історію.

Для HUD цей поділ більш актуальний, тому що в інших інтерфейсах гри використовується в основному недієгетичний тип інтерфейсу. При цьому саме HUD може активно використовувати 3-4 типи одночасно.

Рисунок 9.1 – Приклад трьох типів інтерфейсу на одному екрані, але на різних фічах (1 – недієгетичний, 2 – дієгетичний, 3 – просторовий)

Інформація повинна знаходитись у доступності від точки/області основного фокусу уваги гравця під час ігрового процесу. Для розуміння – розіб'ємо екран на базові блоки HUD:

Рисунок 9.2 – Приклад базових блоків HUD

Тут ми бачимо поділ екрана на такі блоки:

– Головну область фокусу (MFA) – область екрану з найвищою видимістю для гравця, де концентрується його увага під час ігрового процесу. Центр цієї області це центр екрану, точка концентрації основного геймплею.
– Області навколо – з більш низькою видимістю за рахунок свого віддалення від точки максимальної видимості, які найчастіше використовуються для відображення UI.

Якщо раптом ви граєте у щось швидше і когнітивно навантажене вам необхідно отримувати інформацію якнайшвидше та зручніше, для того, щоб бути успішним, наприклад, у шутерах. Саме тому такі елементи, як покажчик напряму атаки, з'являються в момент влучення, знаходяться в головній області фокусування і мають великий розмір.

При розміщенні інтерфейсів в областях навколо MFA вони менш відволікатимуть на себе увагу і при цьому будуть перебувати в достатній близькості від точки фокусування уваги гравця.

Розташування навколо MFA дозволить гравцеві просто і швидко використовувати інформацію, що відображається, витрачаючи на це невелику кількість часу і зусиль. Однак варто зауважити, що видимість інтерфейсів в областях навколо MFA все ж таки буде значно знижена в порівнянні з інтерфейсами всередині MFA.

Зниження видимості інтерфейсу в залежності від його віддалення від точки концентрації уваги та розміру описується законом Фіттса.

Час, необхідне для швидкого переміщення до цільової області (до областей навколо MFA), є функцією відношення між відстанню до мети (зоровий шлях) і розміром мети (цільового інтерфейсу). Якщо простіше: чим далі наш інтерфейс знаходиться від центру екрана, чим він менший, тим менш помітним він стає для уваги гравця, а значить він витрачатиме більше когнітивних зусиль на те, щоб з ним взаємодіяти.

Щоб нівелювати зниження видимості та зменшити необхідні зусилля, UI у цих областях має ініціювати зміщення уваги користувача на себе. Зробити це може через ефекти, анімації та інші можливі показники свого зміненого стану. Це необхідно, щоб гравець не пропустив критично важливу інформацію, і йому було простіше взаємодіяти з інтерфейсом.

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.

2. Завдання до практичної роботи 9

Створити новий проект. В Assets Store Unity підібрати та встановити у своєму проекті безоплатні ассети для створення HUD та показу додаткових вікон.

Тема 10. UI. HUD та вікно перемоги
Практична робота 10. UI. HUD та вікно 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

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.

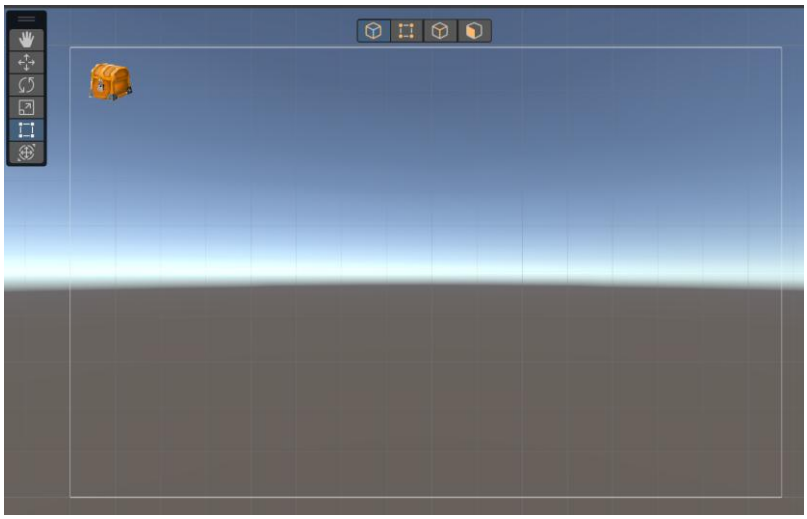
Основними елементи HUD є індикатори здоров'я, кількість боєприпасів, рівень та досвід, прогрес у грі, інвентар тощо.

На попередніх лекціях 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.

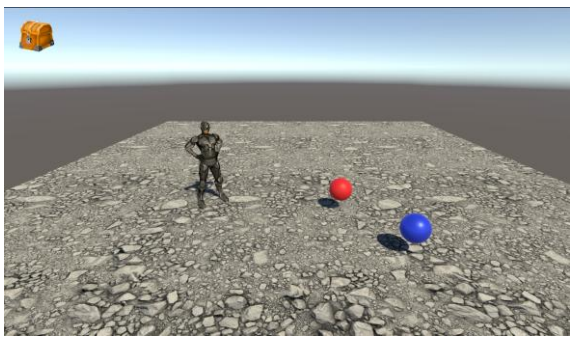
Створимо у Canvas кнопку, наприклад з назвою OpenInventory. В компоненті Image в поле Source Image додаймо зображення, наприклад якоїсь скрині. В компоненті Text (TMP) видалимо надпис.

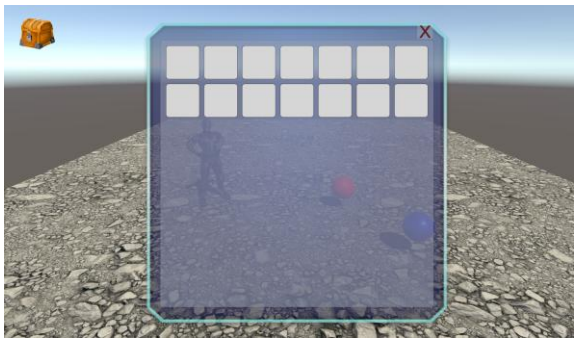
Прикріпимо кнопку до лівого верхнього куту Canvas:



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:



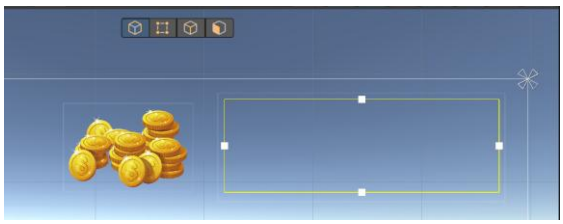


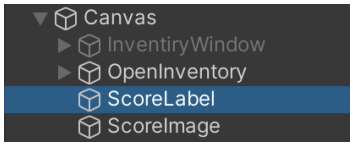
Додаймо у наш інтерфейс можливість бачити прогрес у грі. В якості процесу у грі будемо розуміти інформацію про кількість очок або грошей, які має користувач. Інформація складатиметься з двох частин: картинки, що показує тип інформації та сама кількість. Позиціонувати цю інформацію будемо у правому верхньому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

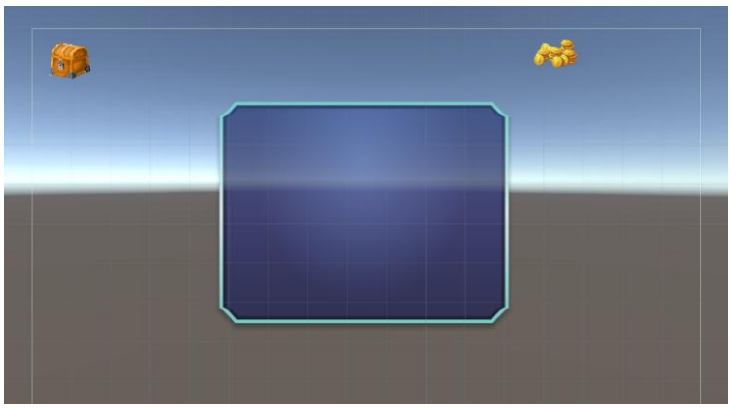
```
void CheckPickUp()
{
    if (Vector3.Distance(transform.position, MoveHelper.player.position) < 1.0F)
    {
        //Debug.Log("Near");
        if (Inventory.AddItem(item))
        {
            GameController.Instance.SetScore(5);
            Destroy(gameObject);
        }

    }
}
```

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:

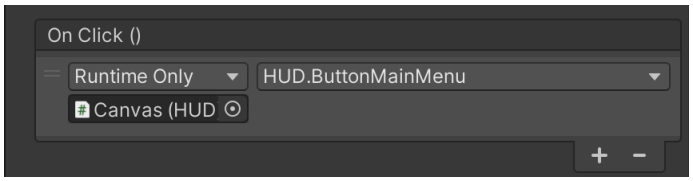
```
[SerializeField]
private GameObject LevelWonWindow;


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 10

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

Тема 11. Огляд популярних атак на комп'ютерні ігри та методи їх проведення
Практична робота 11. Популярні атаки на комп'ютерні ігри

Мета роботи: познайомитися з атаками на комп'ютерні ігри та методами їх проведення. Навчитися протидіяти атаками на комп'ютерні ігри.

1. Теоретичний матеріал

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

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

Ігри є привабливою метою атак з наступних причин:

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

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

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

Усі атаки умовно поділяються на кілька груп:

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

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

Як проводиться атака:

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

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

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

свою чергу, дозволяє змінювати будь-які параметри, але має складний інтерфейс, який може бути непридатним для новачків.

Атака шкідливі моди та «безплатні чіти» – чимало гравців прагнуть модифікувати ігровий досвід: встановлюють нові карти, візуальні покращення, чіти. Саме на цьому й грають кіберзлочинці: вони створюють моди, які містять троянські програми, майнери чи кейлогери. Гравець запускає мод, а разом із ним активується шкідливий процес, що викрадає паролі або використовує ресурси ПК.

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

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

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

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

Причини вразливості – відсутність шифрування та відсутність підпису цілісності.

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

Механізм атаки:

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

Ефекти:

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

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

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

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

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

2. Завдання до практичної роботи 11

Створити презентацію по комп'ютерним іграм та атакам, яким вони піддавалися.

Тема 12. Зберігання даних
Практична робота 12. 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

Слабоструктуровані дані, такі як XML та JSON, використовуються для зберігання та передачі даних, але мають різні характеристики та застосування.

XML (Extensible Markup Language):

- Структура: XML використовує теги для визначення структури даних. Це робить його дуже гнучким, але також може бути громіздким.
- Читабельність: XML легко читається як людьми, так і машинами, але може бути складним для обробки через велику кількість тегів.
- Використання: Часто використовується в веб-сервісах, конфігураційних файлах та для обміну даними між різними системами

JSON (JavaScript Object Notation):

- Структура: JSON використовує пари ключ-значення, що робить його легшим та компактнішим у порівнянні з XML.
- Читабельність: JSON легко читається та обробляється, особливо в JavaScript, що робить його популярним у веб-розробці.
- Використання: Широко використовується в веб-додатках для передачі даних між клієнтом та сервером.

Обидва формати мають свої переваги та недоліки, і вибір між ними залежить від конкретних вимог проекту.

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>
```

За тегами XML зрозуміло, які дані ви переглядаєте. Наприклад, ясно, що це дані про кота, і можна легко визначити його ім'я, вік тощо. Завдяки можливості створювати теги, що визначають майже будь-яку структуру даних, мова XML розширюється.

Правильно сформований XML-файл має відповідати дуже строгим правилам. Якщо він не відповідає цим правилам, не працює XML.

Наприклад, у попередньому прикладі кожен тег, що відкриває, має відповідний тег, що закриває, тому в даному прикладі дотримано одне з правил правильно сформованого XML-файлу.

Базовый синтаксис XML:

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

Декларація (оголошення) XML містить відомості, які готують процесор XML до синтаксичного аналізу документа XML.

Наступний синтаксис показує оголошення XML:

```
<?xml
   version = "version_number"
   encoding = "encoding_declaration"
   standalone = "standalone_status"
?>
```

Декларація XML складається з версії XML, кодування символів та/або автономного статусу. Декларація є необов'язковою. *Якщо декларація XML присутня, вона має бути першою*.

Таблиця 12.1 Атрибути та їх значення в декларації XML

Параметр	Parameter_value	Parameter_description
Версія	1.0	Задає версію стандарту XML.
Кодування	UTF-8, UTF-16, ISO-10646-UCS-2, ISO-10646-UCS-4, ISO-8859-1 до ISO-8859-9, ISO-2022-JP, Shift_JIS, EUC-JP	Він визначає кодування символів у документі. UTF-8 – кодування за замовчуванням.
Автономний	Так чи ні	Він повідомляє синтаксичному аналізатору, чи залежить документ від інформації із зовнішнього джерела, такого як визначення типу зовнішнього документа (DTD) для свого вмісту. Значення за замовчуванням – ні. Встановлення цього значення yes повідомляє процесору, що для синтаксичного аналізу документа не потрібно ніяких зовнішніх оголошень.

Наприклад:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

UTF-8 використовує 8 біт для представлення символів. UTF-16 для представлення символів використовує 16 біт.

Оголошення типу XML-документа, широко відоме як DTD, дозволяє точно описати мову XML. DTD перевіряють словниковий запас та правильність структури XML-

документів на відповідність граматичним правилам відповідної мови XML. XML DTD може бути вказаний усередині документа або може зберігатися в окремому документі.

Приклад внутрішнього DTD:

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "yes" ?>
<!DOCTYPE address [
      <!ELEMENT address (name,company,phone)>
      <!ELEMENT name (#PCDATA)>
      <!ELEMENT company (#PCDATA)>
      <!ELEMENT phone (#PCDATA)>
]>

<address>
      <name> ... </name>
      <company> ... </company>
      <phone> ... </phone>
</address>
```

Коментарі не є обов'язковими. Додавання коментарів допомагає зрозуміти зміст документа. Починається з <!- і закінчується ->.

Наприклад:

```
<!-- Add your comment here -->
```

Теги працюють парами, крім оголошень. Кожна пара тегів складається з тега, що відкриває (початковий тег) і тега, що закриває (кінцевий тег). Імена тегів розташовуються у <>. Для конкретної пари тегів початковий та кінцевий теги мають бути ідентичними, за винятком того, що кінцевий тег має / після <.

Наприклад:

```
<name>...</name>
```

Теги чутливі до регістру. Приклад невірного тегу:

```
<name>...</Name>
```

Все, що знаходиться між тегами, що відкриває і закриває, називається *зміст*.

Відкриваючий тег, контент і тег, що закриває, в цілому називається *елемент*.

Елементи можуть містити *атрибути*.

Наприклад маємо:

```
<age>20</age>
```

age – це ім'я елемента (або елемент); 20 – зміст;

Якщо між тегами немає контенту, це називається *порожні теги*.

Наприклад:

```
<result></result>
```
або
```
<result/>
```

XML – документ повинен містити рівно один кореневий елемент.

Теги мають бути розташовані строго одне всередині одного.

Наприклад:

```
<b><u>This text is bold and italic</u></b>
```

Атрибут елемента розміщується після імені тега у початковому тегу. Можна додати більше одного атрибута до одного елемента з різними іменами атрибутів. Значення атрибутів мають бути поміщені в лапки. Елемент не може містити декілька атрибутів з однаковим назвою.

Атрибут XML має наступний синтаксис

```
<element-name attribute1 attribute2 > ....content.. < /element-name>
```

де attribute1 та attribute2 має наступну форму:

```
name = "value"
```

Наприклад:

```
<teacher subject="English">
   <name>Mr. John</name>.
   <qualification>Graduate </qualification>
</teacher>
```

Ще один приклад:

```
<garden>
   <plants category = "flowers" />
   <plants category = "shrubs"> </plants>
</garden>
```

XML-сутності – це спосіб представлення спеціальних символів. Деякі символи (наприклад, & та < тощо) зарезервовані в XML. Їх називають *спеціальні символи* і вони не можуть бути використані безпосередньо для інших цілей.

Символ	Опис	Ім'я сутності
"	Кавичка (double цитата)	&quot;
&	Амперсант	&amp;
'	Апостроф (одинарна лапка)	&apos;
<	Знак менший	&lt;
>	Знак більше	&gt;

Наприклад:

```
<friend>
    <name>My friends are Alice &amp; Jane</name>
</friend>
```

На додаток до правильно сформованих даних з тегами XML-системи зазвичай використовують два додаткові компоненти:

- схеми;
- перетворення.

Схема XML відома як *XML Schema Definition (XSD)*. Він використовується для опису та перевірки структури та змісту XML-даних. Схема XML визначає елементи, атрибути та типи даних. Елемент схеми підтримує простір імен. Це схоже на схему бази даних, яка описує дані у базі даних.

Основна ідея схем XML полягає в тому, що вони описують допустимий формат, який може приймати документ XML.

Приклад схеми:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
   <xs:element name = "contact">
      <xs:complexType>
         <xs:sequence>
            <xs:element name = "name" type = "xs:string" />
            <xs:element name = "company" type = "xs:string" />
            <xs:element name = "phone" type = "xs:int" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Обумовлені прості типи:

```
xs: integer, xs: boolean, xs: string, xs: date.
```

Складний тип – це контейнер для інших визначень елементів. Це дозволяє вказати, які дочірні елементи може містити елемент, та надати деяку структуру:

```
<xs:element name = "Address">
   <xs:complexType>
   </xs:complexType>
</xs:element>
```

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

XML дозволяє ефективно використовувати та повторно використовувати дані. Механізм повторного використання даних називається *перетворенням XSLT* (або просто перетворенням).

Припустимо, що в базі даних А дані про продаж зберігаються в таблиці, зручній для відділу продажу. У базі даних Б зберігаються дані про доходи та витрати у таблиці, спеціально розробленої для бухгалтерії. База Б може використовувати перетворення, щоб прийняти дані від бази даних А і помістити їх у відповідні таблиці.

Поєднання файлу даних, схеми та перетворення утворює базову систему XML. Файл даних перевіряється на відповідність до правил схеми, а потім передається будь-яким придатним способом для перетворення.

XSLT найчастіше використовується для перетворення даних між різними схемами XML або перетворення даних XML на веб-сторінки або документи PDF.

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();
}
```

JSON (JavaScript Object Notation) – це простий, популярний і легкий для читання формат обміну даними. Він був розроблений з метою забезпечення зручного способу передачі інформації між додатками.

Історія JSON бере свій початок 2001 року, коли Дуглас Крокфорд представив цей формат, щоб полегшити роботу з даними в Ajax-додатках.

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 12

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 13. 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
dWxldCwgS2V5IiwNCiAgICAiQXJ0aWZhY3RzIjogWw0KICAgICAgIlJpbmciLA0KICAgICAgIkFtdWxldCIsDQogICAgICAiS2V5Ig0KICAgIF0sDQogICAgICAiTnVt
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));
}
```

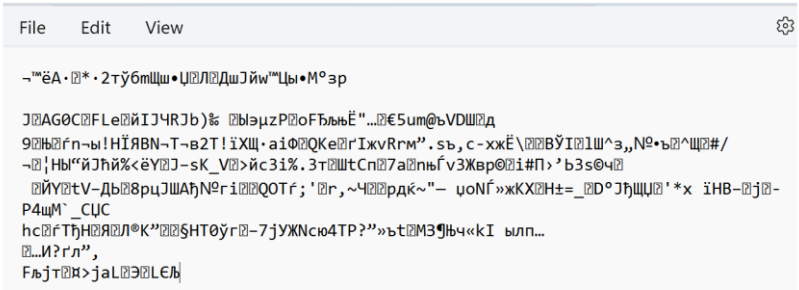
//шифрування даних

```
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:



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 13

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: CRC Press, 2022. - 414 p. URL: https://www.google.com.ua/books/edition/3D_Game_Development_with_Unity/zl5cEAAAQBAJ?hl=ru&gbpv=0
2. Jackson S. Unity 3D UI Essentials: Packt Publishing, 2015. – 280 p. URL: https://www.google.com.ua/books/edition/Unity_3D_UI_Essentials/yxH5rQEACAAJ?hl=ru
3. Gupta K. Unity3d Tutorial For Beginners: FutureZenGroup, 2021. – 81 p. URL: https://www.google.com.ua/books/edition/Unity3d_Tutorial_For_Beginners_By_Kartik/HUIvEAAAQBAJ?hl=ru&gbpv=0
4. Creighton R. Unity 3D Game Development by Example: Packt Publishing Ltd, 2010. – 364 p. URL: https://play.google.com/store/books/details/Ryan_Henson_Creighton_Unity_3D_Game_Development_by?id=vFMggPqVnhcC

Допоміжна

5. Nakov S. Programming Basics with C# / S. Nakov, Nakov's Team: SoftUni, 2019. – 408 p. URL: https://www.google.com.ua/books/edition/Programming_Basics_with_C/vzKyDwAAQBAJ?hl=ru&gbpv=0
6. Norton T. Learning C# by Developing Games with Unity 3D: Packt Publishing Ltd, 2013. – 292 p. URL: https://play.google.com/store/books/details/Terry_Norton_Learning_C_by_Developing_Games_with_U?id=7Zf9AAAAQBAJ
7. Miller R. C# for Artists. The Art, Philosophy, and Science of Object-Oriented Programming: Pulp Free Press, 2008. – 750 p. URL: https://www.google.com.ua/books/edition/C_for_Artists/K0ICo6r29W4C?hl=ru&gbpv=0
8. Zoubir Z. Mammeri. Cryptography: Algorithms, Protocols, and Standards for Computer Security : John Wiley & Sons, 2024. – 624 p. URL: https://www.google.com.ua/books/edition/Cryptography/ecb0EAAAQBAJ?hl=ru&gbpv=1&dq=Cryptography&printsec=frontcover
9. Paar C. Understanding Cryptography: A Textbook for Students and Practitioners / C. Paar, J. Pelzl, B. Preneel: Springer Berlin Heidelberg, 2010. – 372 p. URL: https://www.google.com.ua/books/edition/Understanding_Cryptography/f24wFELSzkoC?hl=ru&gbpv=0

Інформаційні ресурси

10. Офіційний сайт Unity [Електронний ресурс] URL: https://unity.com/ (Дата звернення: 01.09.2025)
11. Офіційний сайт Visual Studio Community [Електронний ресурс] URL: https://visualstudio.microsoft.com/ru/vs/community/ (Дата звернення: 01.09.2025)
12. Офіційний сайт програми CrypTool 2 [Електронний ресурс] URL: https://www.cryptool.org/en/ct2/downloads/ (Дата звернення: 01.09.2025)