Міжнародний гуманітарний університет
Кафедра Комп’ютерної інженерії та інноваційних технологій

Лебедєва О.Ю.

ПРОГРАМУВАННЯ ТА ЗАХИСТ PLAY-ТЕХНОЛОГІЙ

конспект лекцій для здобувачів другого (магістерського) рівня,
які навчаються за спеціальністю 125 – Кібербезпека та захист інформації

Одеса 2024

Затверджено Вченою Радою Міжнародного гуманітарного університету (протокол № 13 від 16.08.2024 р.).

Лебедєва О.Ю.

Програмування та захист play-технологій: конспект лекцій для здобувачів другого (магістерського) 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, 2024. – 90 с.

Конспект лекцій для курсу «Програмування та захист play-технологій» призначені для здобувачів, що навчаються за другим (магістерським) рівнем за спеціальністю: 125 – Кібербезпека та захист інформації. Дисципліна «Програмування та захист play-технологій» 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 2D-ігор в середовищі Unity, написання скриптів для реалізації функціональності гри та захисту даних.

ЗМІСТ

Лекція 1. Базові конструкції в мові C#

1. Змінні та типи даних
2. Основні операції над типами даних.
3. Умовні конструкції.
4. Циклічні конструкції.

1. Змінні та типи даних

Перед використанням будь-яку змінну слід визначити. Синтаксис визначення змінної виглядає так:

```
тип ім'я_змінної;
```
або
```
тип ім’я_змінної = значення;
```

Як ім'я змінної може виступати будь-яка довільна назва, яка задовольняє наступним вимогам:

– ім'я може містити будь-які цифри, букви та символ підкреслення, при цьому перший символ в імені має бути буквою або символом підкреслення;
– в імені не повинно бути знаків пунктуації та пропусків (пробілів);
– ім'я не може бути ключовим словом мови C#.

Хоча ім'я зміною може бути будь-яким, але слід давати змінним описові імена, які будуть говорити про їхнє призначення.

C# є регістрозалежною мовою, тому два однакові визначення змінних, але записані різними строчними або заголовними літерами будуть представляти дві різні змінні. Наприклад, name та Name – це різні імена змінних.

Константи призначені для опису таких значень, які не повинні змінюватись у програмі.

Для визначення констант використовується ключове слово const, яке вказується перед типом константи:

```
const тип ім’я_константи = значення;
```

Як і в багатьох мовах програмування, C# є своя система типів даних, яка використовується для створення змінних.

Тип даних визначає внутрішнє представлення даних, безліч значень, які може приймати об'єкт, і навіть допустимі дії, які можна застосовувати над об'єктом.

Дійсні типи:

Ключевое слово или тип C#	Приблизительный диапазон значений	Точность	Размер	Тип .NET
`float`	От ±1,5 x 10^{-45} до ±3,4 x 10^{38}	6–9 цифр	4 байта	System.Single
`double`	от ±5,0 × 10^{-324} до ±1,7 × 10^{308}	15–17 цифр	8 байт	System.Double
`decimal`	от ±1,0 x 10^{-28} до ±7,9228 x 10^{28}	28-29 знаков	16 байт	System.Decimal

Цілочисельні типи:

Ключевое слово или тип C#	Диапазон	Размер	Тип .NET
`sbyte`	От -128 до 127	8-разрядное целое число со знаком	System.SByte
`byte`	От 0 до 255	8-разрядное целое число без знака	System.Byte
`short`	От -32 768 до 32 767	16-разрядное целое число со знаком	System.Int16
`ushort`	От 0 до 65 535	16-разрядное целое число без знака	System.UInt16
`int`	От -2 147 483 648 до 2 147 483 647	32-разрядное целое число со знаком	System.Int32
`uint`	От 0 до 4 294 967 295	32-разрядное целое число без знака	System.UInt32
`long`	От -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	64-разрядное целое число со знаком	System.Int64
`ulong`	От 0 до 18 446 744 073 709 551 615	64-разрядное целое число без знака	System.UInt64
`nint`	Зависит от платформы (вычисляется во время выполнения)	32- или 64-разрядное целое число со знаком	System.IntPtr
`nuint`	Зависит от платформы (вычисляется во время выполнения)	32- или 64-разрядное целое число без знака	System.UIntPtr

Інші типи:

- bool: зберігає значення true чи false. Представлений системним типом System.Boolean.
- char: зберігає одиночний символ у кодуванні Unicode і займає 2 байти. Представлений системним типом System.Char.
- string: зберігає набір символів Unicode. Представлений системним типом System.String.
- object: може зберігати значення будь-якого типу даних і займає 4 байти на 32-розрядній платформі та 8 байт на 64-розрядній платформі. Представлений системним типом System.Object, який є базовим для всіх інших типів та класів .NET.

Літерали можна передавати змінним як значення. Літерали бувають логічними, цілими, речовими, символьними і малими. І окремий літерал є ключовим словом null.

Наприклад:

```
bool alive = true;
int b = -101;
int a = 0b11;       // 3 у двійковій формі
int a = 0xFF;     // 255 у шістнадцятковій формі
uint a = 10U;
long b = 20L;
ulong c = 30UL;
float b = 30.6f;
double a = 10.003;
decimal c = 1005.8M;
char g = '1';
char k = '\x78';     // x в ASCII
char l = '\u0420';   // Р в Unicode
string m = "hello word";
```

```
string n = " Компанія \"Роги та копита\"";
string p = "Hello \nWorld!!!";
```

2. Основні операції над типами даних

Приклади арифметичних операцій:

```
int x = 10;
int z = x + 12; // 22
int y = x - 6;  // 4
int m = x * 5; // 50
double b = 3;
double c = x / b; //3.33333333
double z = 10 / 4; //результат дорівнює 2
double z = 10.0 / 4.0; //результат равен 2.5
```

Операція % отримання залишку від цілісного поділу двох чисел.

Наприклад:

```
double x = 10.0;
double y = x % 4; //результат равен 2
```

Операція інкременту.

Інкремент буває префіксним: ++x – спочатку значення змінної x збільшується на 1, а потім її значення повертається як результат операції.

І також існує постфіксний інкремент: x++ - спочатку значення змінної x повертається як результат операції, а потім до нього додається 1.

Наприклад:

```
int x1 = 5;
int z1 = ++x1; // z1=6; x1=6
int x2 = 5;
int z2 = x2++; // z2=5; x2=6
```

Операція декременту чи зменшення значення на одиницю. Також існує префіксна форма декременту (--x) та постфіксна (x--).

Наприклад:

```
int x1 = 5;
int z1 = --x1; // z1=4; x1=4
int x2 = 5;
int z2 = x2--; // z2=5; x2=4
```

Усі арифметичні оператори є лівоасоціативними, тобто виконуються зліва направо. Тому вираз 10/5*2 необхідно трактувати як (10/5)*2, тобто результатом буде 4.

Крім базової операції присвоєння в C# є ще ряд комбінованих операцій:

+= присвоєння після додавання. Надає лівому операнду суму лівого та правого операндів: вираз A += B рівнозначний виразу A = A + B

-= присвоєння після віднімання. Надає лівого операнду різниця лівого і правого операндів: A -= B еквівалентно A = A - B

*= присвоєння після множення. Надає лівому операнду добуток лівого та правого операндів: A *= B еквівалентно A = A * B

/= присвоєння після розподілу. Надає лівому операнду приватне лівого та правого операндів: A /= B еквівалентно A = A / B

%= присвоєння після поділу по модулю. Надає лівому операнду залишок від цілого розподілу лівого операнду на правий: A %= B еквівалентно A = A % B.

Наприклад:

```
int a = 10;
a += 10;         // 20
a -= 4;          // 16
a *= 2;          // 32
a /= 8;          // 4
```

Операції присвоєння є правоасоціативними, тобто виконуються праворуч наліво.

Наприклад:

```
int a = 8;
int b = 6;
int c = a += b -= 5;     // 9
```

Розглянемо приклад:

```
byte a = 4;
byte b = a + 70;  // помилка
```

При спробі скомпілювати програму ми отримаємо помилку на етапі компіляції.

Справа в тому, що операція додавання (та й віднімання) повертає значення типу int, якщо в операції беруть участь цілі численні типи даних з розрядністю менше або дорівнює int (тобто типи byte, short, int). Тому результатом операції a + 70 буде об'єкт, який має довжину в пам'яті 4 байти. Потім цей об'єкт ми намагаємось присвоїти змінній b, яка має тип byte і в пам'яті займає 1 байт.

І щоб вийти із цієї ситуації, необхідно застосувати операцію перетворення типів.

Операція перетворення типів передбачає вказівку у дужках того типу, якого треба перетворити значення:

```
(тип_даних_в_який треба перетворити) значення_для_перетворення;
```

Змінимо попередній приклад, застосувавши операцію перетворення типів:

```
byte a = 4;
byte b = (byte)(a + 70);
```

Перетворення можуть бути звужувальні (narrowing) та розширюючі (widening). Розширювальні перетворення розширюють розмір об'єкта пам'яті. Звужують перетворення, навпаки, звужують значення типу меншої роз'ядності.

У разі розширення перетвореннями компілятор за нас виконував всі перетворення даних, тобто перетворення були неявними (implicit conversion). Такі перетворення не викликають якихось труднощів.

При явних перетвореннях (explicit conversion) ми повинні застосувати операцію перетворення.

Перетворення, що розширюють, від типу з меншою розрядністю до типу з більшою розрядністю компілятор проводить неявно.

byte -> short -> int -> long -> decimal

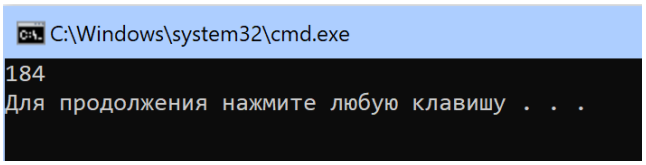
int -> double

short -> float -> double

char -> int

Наприклад:

```
byte a = 30;
byte b = 100;
byte c = (byte)(a * b);//    3000 % 256 = 184
Console.WriteLine(c);
```



Для виконання різних математичних операцій у бібліотеці класів .NET призначено клас Math (using System;).

Математичні операції можна поділити на наступні категорії:

– Округлення – Round, Truncate, Floor, Ceiling;
– Максимум та мінімум – Max, Min;
– Абсолютне значення та знак – Abs, Sign;
– Квадратний корінь – Sqrt;
– Зведення в ступінь – Pow, Exp;
– Логарифм – Log, Log10;
– Тригонометричні функції – Sin, Cos, Tan, Sinh, Cosh, Tanh, Asin, Acos, Atan.

Обчислення синуса, косинуса обчислюється у радіанах. Тому, якщо потрібні градуси, потрібно їх конвертувати в радіани.

Наприклад:

```
int gradus = 30;
double radian = gradus * Math.PI / 180;
```

Розглянемо оператори відносин:

Оператор¤	Значение¤
==¤	Рівно¤
!·=·¤	Не·дорівнює¤
>·¤	Більше¤
<·¤	Менше¤
>=·¤	Більше·або·дорівнює¤
<=·¤	Менше·або·дорівнює¤

Розглянемо логічні операції:

Оператор¤	Значення¤
&·¤	І¤
|·¤	АБО¤
^¤	Виключне·АБО¤
&&·¤	Укорочене·І¤
||·¤	Укорочене·АБО¤
!·¤	НЕ¤

Результатом виконання оператора відносин чи логічного оператора є логічне значення типу bool.

Тип даних повідомляє нам дійсні значення та операції для змінних і констант цього типу даних. Маємо наступні операції:

Операція	Типи даних
+	byte, short, int, long, float, double, decimal
-	byte, short, int, long, float, double, decimal
*	byte, short, int, long, float, double, decimal
/	byte, short, int, long, float, double, decimal
%	byte, short, int, long
++, --	byte, short, int, long
==, !=	byte, short, int, long, float, double, decimal, char, bool
<, >	byte, short, int, long, float, double, decimal, char, bool
<=, >=	byte, short, int, long, float, double, decimal, char, bool
&&, ||, !	bool

3. Умовні конструкції

Умовні конструкції – один із базових компонентів багатьох мов програмування, які спрямовують роботу програми по одному зі шляхів залежно від певних умов.

Однією з таких конструкцій у мові програмування C# є конструкція if.

Конструкція if/else перевіряє істинність певної умови та залежно від результатів перевірки виконує певний код.

Її найпростіша форма складається з блоку if:

```
if(умова)
{
     виконувані інструкції
}
```

Якщо при недотриманні умови також виконувались якісь дії, то в цьому випадку можна додати блок else:

```
if(умова)
{
      інструкції_1
}
else
{
      інструкції_2
}
```

Якщо при недотриманні умови необхідно перевірити ще одну умову та виконати будь-які дії, то в цьому випадку можна додати блок else if:

Синтаксис else if:

```
if(умова1)
{
      інструкції_1
}
else if(умова2)
{
      інструкції_2
}
else
{
      інструкції_3
}
```

Конструкція switch–case це інструмент управління вибором, що дозволяє перевіряти змінну на рівність

Синтаксис оператора switch C#:

```
switch (expression)
{
    case constantA:
         Instructions1;
         break;
     case constantB:
          Instructions2;
          break;
    default:
          Instructions3;
          break;
}
```

Значення виразу може мати такі типи: типи чисел, тип bool, char, string.

Microsoft випустила C# 7.0 у 2017 році. Одна з еволюційних функцій цієї версії, пов'язаних з оператором switch і можливість вказувати умову в блоці case за допомогою ключового слова when, при якому обробляється необхідний діапазон значень

Наприклад:

```
double x;
Console.WriteLine("Input number: ");
CultureInfo uk = CultureInfo.GetCultureInfo("en-GB");
x = Double.Parse(Console.ReadLine(), uk);

switch (x)
{
    case double i when i >= 1:
        Console.WriteLine("The number greater than or equal to 1"); break;
    case double i when i < 0:
        Console.WriteLine("The number is negative"); break;
    case double i when i >= 0 && i < 1:
        Console.WriteLine("The number between 0 and 1"); break;
}
```

```
Input number:
0.4
The number between 0 and 1
```

4. Циклічні конструкції

Цикли while багаторазово виконують код у своєму тілі до тих пір, поки результатом виразу типу bool є true. Вираз перевіряється перед виконанням тіла циклу.

```
while (умова)
{
    тіло циклу
}
```

Цикли do-while відрізняються за функціональністю від циклів while тільки тим, що вираз у них перевіряється після виконання блоку операторів (гарантуючи виконання тіла циклу щонайменше один раз).

```
do
{
    тіло циклу
}
while (умова);
```

Цикли for схожі на цикли while, але мають спеціальні конструкції для ініціалізації та ітерації змінної циклу.

Цикл for містить три частини:

```
for ([дії_до_виконання_циклу]; [умова]; [дії_після_виконання])
{
    тіло циклу
}
```

Оператор foreach забезпечує прохід по всіх елементах в об'єкті, що перераховується. Більшість типів .NET, які представляють набір або список елементів, є переліченими.

Прикладами перерахованих типів можуть бути масиви і рядки.

Наприклад:

```
foreach (char c in "programmer")
    Console.WriteLine(c);
```

```
p
r
o
g
r
a
m
m
e
r
```

Іноді виникає ситуація, коли потрібно вийти з циклу, не чекаючи на його завершення. У цьому випадку можна скористатись оператором break.

Оператор continue ігнорує оператори, що залишилися, в циклі і починає наступну ітерацію.

Наприклад:

```
for (int i = 0; i < 9; i++)
{
    if (i == 5)
        continue;
    Console.WriteLine(i);
}
```

Оператор return використовується для виходу з циклу та методу.

Віднесення його до категорії операторів переходу обумовлено тим, що він змушує програму перейти до точки виклику методу.

Оператор return може мати асоційоване з ним значення, тоді при виконанні даного оператора це значення повертається як значення функції.

```
return вираз;
```

У функціях типу void використовується оператор return без значення.

Всередині методу може бути довільна кількість операторів return. Вихід із методу відбувається тоді, коли зустрічається один із них.

Контрольні питання

1. Які правила створення змінних в C#?
2. Які типи даних існують в C#?
3. Які основні операції можна проводити на цілочисельними даними в C#?
4. Для чого використовуються умовні конструкції?
5. Коли потрібно застосовувати вітку else при створенні умовних конструкцій?
6. Для чого використовуються циклічні конструкції?
7. Які оператори використовуються для створення циклічної конструкції в C#?

Лекція 2. Структури даних в мові C#

1. Одновимірні та двовимірні масиви
2. Ступінчасті масиви
3. Текстові дані C#
4. Створення власних функцій

1. Одновимірні та двовимірні масиви

Для оголошення масиву в мові C# використовується наступний синтаксис:

```
тип[] ім'я_масиву;
ім'я_масиву = new тип[кількість];
```

або

```
тип[] ім'я_масиву = new тип[кількість];
```

При такому визначенні всі елементи набувають значення за замовчанням, яке передбачено для їх типу.

Ініціалізація являє собою набір початкових значень елементів масиву, розділених комами.

```
тип[] ім'я_масиву = new тип[кількість]{ініціалізація};
```

Розглянемо приклад:

```
int[] nums2 = new int[4] { 1, 2, 3, 5 };
int[] nums3 = new int[] { 1, 2, 3, 5 };
int[] nums4 = new[] { 1, 2, 3, 5 };
int[] nums5 = { 1, 2, 3, 5 };
string[] people = { "Tom", "Sam", "Bob" };
```

Для звернення до елементів масиву застосовуються індекси.

Індекс представляє номер елемента в масиві, при цьому нумерація починається з нуля, тому індекс першого елемента дорівнюватиме 0.

```
ім'я_масиву[індекс] = значення;
```

або

```
змінна = ім'я_масиву[індекс];
```

Для оголошення двовимірного масиву в мові C# використовується наступний синтаксис:

```
тип[,] ім'я_масиву;
ім'я_масиву = new тип[к-ть рядків, к-ть стовпців];
```

або

```
тип[,] ім'я_масиву = new тип[к-ть рядків, к-ть стовпців];
```

При такому визначенні всі елементи набувають значення за замовчанням, яке передбачено для їх типу.

Ініціалізація являє собою набір початкових значень елементів масиву, розділених комами.

```
тип[,] ім'я_масиву = new тип[к-ть рядків, к-ть стовпців] {ініціалізація};
```

Наприклад:

```
int[,] nums3 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };
int[,] nums4 = new int[,] { { 0, 1, 2 }, { 3, 4, 5 } };
int[,] nums5 = new[,] { { 0, 1, 2 }, { 3, 4, 5 } };
int[,] nums6 = { { 0, 1, 2 }, { 3, 4, 5 } };
char[,] people = { {'T', 'o', 'm'}, {'B', 'o', 'b'}};
```

Для звернення до елементів масиву застосовуються два індекси.

Перший індекс представляє номер рядка в масиві, при цьому нумерація починається з нуля. Другий індекс – номер стовпця, нумерація також з нуля.

```
ім'я_масиву[рядок, стовпець] = значення;
```

або

```
змінна = ім'я_масиву[рядок, стовпець];
```

Метод `GetUpperBound(номер_розмірності)`, який повертає індекс останнього елемента у певній розмірності. У двомірному масиві перша розмірність (з індексом 0) насправді і є таблиця. І за допомогою виразу масив.`GetUpperBound(0) + 1` можна одержати кількість рядків таблиці, представленої двовимірним масивом.

Через `масив.Length / кількість_рядків` можна одержати кількість елементів у кожному рядку, тобто, кількість стовпців. Або `масив.GetUpperBound(1) + 1` – дозволяє обчислити також кількість стовпців.

2. Ступінчасті масиви

Ступінчасті масиви оголошуються за допомогою ряду квадратних дужок, у яких вказується їх розмірність.

Наприклад, для оголошення двовимірного ступінчастого масиву служить наступна загальна форма:

```
тип[] [] ім'я_масиву = new тип [розмір] [];
```

де розмір позначає кількість рядків у масиві.

Розглянемо приклад оголошення ступінчастого масиву:

```
int[][] jagged = new int[3][];
jagged[0] = new int[4];
jagged[1] = new int[3];
jagged[2] = new int[5];
```

Після створення ступінчастого масиву доступ до його елементів здійснюється за індексом, що вказується в окремих квадратних дужках.

Наприклад, у наступному рядку коду

```
jagged[2][1] = 10;
```

елементу масиву jagged, що знаходиться на позиції з координатами (2,1), надається значення 10.

У разі ступінчастого масиву за допомогою властивості Length можна отримати довжину кожного масиву, що становить ступінчастий масив.

Розглянемо приклад роботи з ступінчастим масивом:

```
int[][] nums = new int[3][];
nums[0] = new int[] { 1, 2 };
nums[1] = new int[] { 1, 2, 3 };
nums[2] = new int[] { 1, 2, 3, 4, 5 };

for (int i = 0; i < nums.Length; i++)
{
    for (int j = 0; j < nums[i].Length; j++)
        Console.Write("{0,2}  ", nums[i][j]);

    Console.WriteLine();
}
```

```
1   2
1   2   3
1   2   3   4   5
```

Для створення масиву можна використовувати ще клас Array. Цей клас визначає ряд властивостей та методів, які ми можемо використовувати під час роботи з масивами. Створення екземплярів масивів можливо за допомогою методу CreateInstance().

Наприклад:

```
Array strs = Array.CreateInstance(typeof(string), 3);
strs.SetValue("Brand", 0);
strs.SetValue("Model", 1);
strs.SetValue("Goods", 2);

Console.WriteLine((string)strs.GetValue(1));
```

```
Model
```

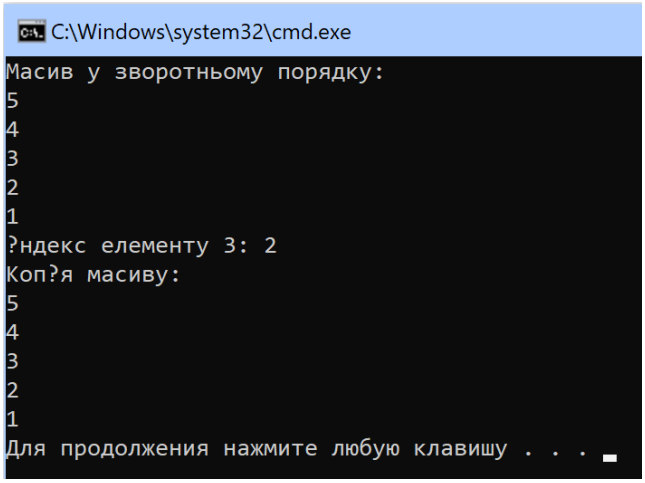
Основні властивості та методи:

- Властивість Length повертає довжину масиву
- Властивість Rank повертає розмірність масиву
- Clear (Array array) очищає масив, встановлюючи для всіх його елементів значення за замовчуванням
- void Copy (Array sourceArray, int sourceIndex, Array destinationArray, int destinationIndex, int length) копіює з масиву sourceArray починаючи з індекс sourceIndex length елементів в масив destinationArray починаючи з індексу destinationIndex
- int IndexOf (Array array, object value) повертає індекс першого входження елемента в масив

– int LastIndexOf (Array array, object value) повертає індекс останнього входження елемента в масив
– void Reverse (Array array) має елементи масиву у зворотному порядку
– void Sort (Array array) сортує елементи одновимірного масиву

Наприклад:

```
int[] numbers = { 1, 2, 3, 4, 5 };

Array.Reverse(numbers);
Console.WriteLine("Масив у зворотньому порядку: ");
foreach (int num in numbers)
    Console.WriteLine(num);

int index = Array.IndexOf(numbers, 3);
Console.WriteLine("Індекс елементу 3: " + index);

int[] copy = new int[numbers.Length];
Array.Copy(numbers, copy, numbers.Length);
Console.WriteLine("Копія масиву: " );
foreach (int num in copy)
    Console.WriteLine(num);
```



3. Текстові дані C#

Текстові дані C# представлені типами System.String (або просто string) або System.Char (char). Обидва типи зберігають дані в кодуванні Unicode. Перший тип даних дозволяє зберігати рядок, другий – тільки один символ.

Наприклад:

```
string s1 = "hello";
string s2 = new String('a', 6);
string s3 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' });
string s4 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' }, 1, 3);

Console.WriteLine(s1);  // hello
Console.WriteLine(s2);  // aaaaaaa
Console.WriteLine(s3);  // world
Console.WriteLine(s4);  // orl
```

Для створення порожнього рядка можна скористатися наступним кодом:

```
string message2 = null;
```

або

```
string message3 = System.String.Empty;
```

4. Створення власних функцій

Загальне визначення методів виглядає так:

```
[модифікатори] тип значення, що повертається назва методу ([параметри])
{
      // Тіло методу
}
```

Перед назвою методу йде тип даних, що повертається. Тип void, який вказує на те, що фактично нічого не повертає, він просто робить деякі дії. Після назви методу у дужках йде перелік параметрів. Якщо дужки порожні, це означає, що метод не приймає жодних параметрів.

Параметри дозволяють передати до методу деякі вхідні дані. Параметри визначаються через кому в дужках після назви методу у вигляді:

```
тип_методу ім'я_методу (тип_параметра1 параметр1, тип_параметра2 параметр2,
...)
{
     // дії методу
}
```

Щоб використати метод, треба його викликати. Для виклику методу вказується його ім'я, після якого у дужках йдуть значення його параметрів (якщо метод приймає параметри).

```
назва методу (значення параметрів методу);
```

Наприклад:

```
static void sum(double x, double y)
{
    double res =  x + y;
    Console.WriteLine("{0} + {1} = {2}", x, y, res);
}

static void Main(string[] args)
{
    Console.WriteLine("Input a first number: ");
    double number1 = Convert.ToDouble(Console.ReadLine());

    Console.WriteLine("Input a second number: ");
    double number2 = Convert.ToDouble(Console.ReadLine());

    sum(number1, number2);

    Console.ReadKey();
}
```

```
Input a first number:
20
Input a second number:
30
20 + 30 = 50
```

За замовчуванням під час виклику методу необхідно надати значення для всіх параметрів.

Але C# дозволяє використовувати *необов'язкові параметри*. Для таких параметрів нам необхідно оголосити значення за промовчанням. Також слід враховувати, що після необов'язкових параметрів усі наступні параметри також мають бути необов'язковими.

Наприклад:

```
static void PrintPerson(string name, int age = 1, string company = "Undefined")
{
    Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");
}

static void Main(string[] args)
{
    PrintPerson("Tom", 37, "Microsoft");
    PrintPerson("Tom", 37);
    PrintPerson("Tom");

    Console.ReadKey();
}
```

```
Name: Tom  Age: 37  Company: Microsoft
Name: Tom  Age: 37  Company: Undefined
Name: Tom  Age: 1  Company: Undefined
```

Існує два способи передачі параметрів метод у мові C#:

– за значенням;
– за посиланням.

Найбільш простий спосіб передачі параметрів є передача *за значенням*, по суті це звичайний спосіб передачі параметрів.

Наприклад:

```
static void Increment(int n)
{
    n++;
}

static void Main(string[] args)
{
    Console.WriteLine("Input a number: ");
    int number1 = Convert.ToInt32(Console.ReadLine());

    Increment(number1);
    Console.WriteLine("Number after method: {0}", number1);

    Console.ReadKey();
}
```

```
Input a number:
10
Number after method: 10
```

Під час передачі значень параметрів *за посиланням* метод отримує адресу змінної пам'яті. І, таким чином, якщо в методі змінюється значення параметра, що передається за посиланням, то також змінюється значення змінної, яка передається на його місце. Під час передачі параметрів за посиланням перед параметрами використовується модифікатор ref:

Наприклад:

```
static void Increment(ref int n)
{
    n++;
}

static void Main(string[] args)
{
    Console.WriteLine("Input a number: ");
    int number1 = Convert.ToInt32(Console.ReadLine());

    Increment(ref number1);
    Console.WriteLine("Number after method: {0}", number1);

    Console.ReadKey();
}
```

```
Input a number:
10
Number after method: 11
```

У попередньому прикладі використовувалися вхідні параметри. Але параметри можуть бути вихідними. Щоб зробити вихідним параметром, перед ним ставиться модифікатор out. Використання подібних параметрів полягає в тому, що ми можемо повернути з методу не одне значення, а декілька значень.

Наприклад:

```
static void GetRectangleData(int width, int height, out int rectArea, out int rectPerimetr)
{
    rectArea = width * height;
    rectPerimetr = (width + height) * 2;
}

static void Main(string[] args)
{
    int area;
    int perimetr;

    GetRectangleData(10, 20, out area, out perimetr);

    Console.WriteLine($"Площадь прямоугольника: {area}");
    Console.WriteLine($"Периметр прямоугольника: {perimetr}");

    Console.ReadKey();
}
```

```
Площадь прямоугольника: 200
Периметр прямоугольника: 60
```

У попередніх прикладах використовувалося постійне число параметрів. Але використовуючи ключове слово params, можна передавати невизначену кількість параметрів. Сам параметр з ключовим словом params щодо методу повинен представляти одномірний масив того типу, дані якого збираємося використовувати.

При виклику методу на місце параметру з модифікатором params ми можемо передати як окремі значення, і масив значень, або взагалі не передавати параметри. Якщо нам треба передати якісь інші параметри, то вони повинні вказуватися до параметра з ключовим словом params.

Наприклад:

```
static void Sum(int initialValue, params int[] numbers)
{
    int result = initialValue;
    foreach (var n in numbers)
    {
        result += n;
    }
    Console.WriteLine(result);
}

static void Main(string[] args)
{
    int[] nums = { 1, 2, 3, 4, 5 };
    Sum(0, nums);
    Sum(0, 2, 3);

    Console.ReadKey();
}
```

```
15
5
```

Контрольні питання

1. Що таке масиви?
2. Які масиви існують в C#?
3. В чому особливість ступенчастих масивів в C#?
4. Як створити рядок з тексту в C#?
5. Коли потрібно створювати власні функції?
6. Для чого використовується ключове слово params в C#?
7. Для чого використовуються модифікатор ref та out в C#?

Лекція 3. Використання мови C# в Unity

1. Поняття класу та конструктору в ООП
2. Інкапсуляція.
3. Спадкування.
4. Скрипти в Unity.

1. Поняття класу та конструктору в ООП

Вся функціональність класу представлена його членами:

- полями (полями називаються змінні класу)
- властивостями
- методами
- подіями

Поле – так називається член-змінна, що містить деяке значення. В ООП поля іноді називають даними об'єкта. До полю можна застосовувати кілька модифікаторів в залежності від того, як ви збираєтеся це поле використовувати. У число модифікаторів входять static, readonly і const.

Константи – це поле, значення якого змінити не можна.

Метод – це код, що впливає на дані об'єкта (або поля).

Подія викликає виконання деякого фрагмента коду. Події – невід'ємна частина програмування для Microsoft Windows. Наприклад, події виникають при русі миші, кліку або зміні розмірів вікна.

Властивості – їх іноді називають "розумними" полями (smart fields), так як вони насправді є методами, які клієнти класу сприймають як поля. Це блоки коду з set і get;

Для створення класу використовується ключове слово `class`:

```
class NameOfClass
{

}
```

Можна визначати спеціальні методи, що викликаються щоразу при створенні екземпляра класу. Ці методи називаються *конструкторами*.

Конструктор найчастіше використовується для ініціалізації об'єкта. У конструкторів має бути те ж ім'я, що і у самого класу. Значення конструктори не повертають.

Створення екземплярів об'єктів робиться за допомогою ключового слова new:

```
НазваКласу НазваОб’єкту = new НазваКласу (аргументи)
```

Наприклад, наступний рядок створює екземпляр класу MyClass.

```
MyClass myClass = new MyClass ();
```

У класі може бути більш одного конструктора. Розглянемо приклад.

```
namespace ConsoleApplication1
{
    class Person
    {
        public string name; // имя
        public int age;     // возраст

        public Person()
        {
            name = "Неизвестно";
            age = 0;
        }
        public Person(string n)
        {
            name = n;
            age = 0;
        }
        public Person(string name, int age)
        {
            this.name = name;
            this.age = age;
        }

        public void GetInfo()
        {
            Console.WriteLine($" Имя: {name}  Возраст: {age}");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine();

            Person person2 = new Person();
            person2.GetInfo();
            Person person1 = new Person("Richard", 25);
            person1.GetInfo();
            Person person3 = new Person("Alex");
            person3.GetInfo();

            Console.ReadLine();
        }
    }
}
```

```
Имя: Неизвестно  Возраст: 0
Имя: Richard  Возраст: 25
Имя: Alex  Возраст: 0
```

2. Інкапсуляція

Концепцію інкапсуляції визнають одним з основоположних принципів об'єктно-орієнтованого програмування.

Принцип *інкапсуляції* полягає в тому, що в класі можна вказати рівень доступності для звернення до кожного з його членів з коду, розташованого поза цим класу або структури.

У C# є чотири основні модифікатори доступу:

– public – змінна доступна для будь-якого сценарію без обмежень;
– private — змінна доступна лише у класі, де створена;
– protected — доступна з класу, що містить, або похідних від його типів;
– internal — доступна лише у поточній збірці.

Якщо явного модифікатора доступу немає, то використовується private.

Крім звичайних методів в мові C# передбачені спеціальні методи доступу, які називають *властивості*. Вони забезпечують простий доступ до полів класу, щоб дізнатися їх значення або виконати їх установку.

Стандартне опис властивості має наступний синтаксис :

```
[модиф_доступу] поверт_тип назва
{
    // код властивості
}
```

Розглянемо приклад:

```
class Person
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }
}


class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine();

        Person p = new Person();

        // Устанавливаем свойство - срабатывает блок Set
        // значение "Tom" и есть передаваемое в свойство value
        p.Name = "Tom";

        // Получаем значение свойства и присваиваем его переменной
        //срабатывает блок Get
        string personName = p.Name;
        Console.WriteLine(" "+ personName);

        Console.ReadLine();
    }
}
```

```
Tom
```

Поле, оголошене з модифікатором static, є *статичним*.

Статичне поле визначає строго одне місце зберігання. Незалежно від того, скільки буде створено реалізації відповідного класу, існує тільки одна копія статичного поля.

Якщо клас має *статичні методи*, то, щоб отримати до них доступ, необов'язково створювати об'єкт цього класу. Доступ до статичних методів здійснюється через клас. Наприклад, методи Write() і WriteLine() класу Console мають модифікатор static і звернення до цих методів здійснюється через ім'я класу: Console.WriteLine();

Наприлад:

```
public class Room
{
    string type;
    float area;
    int num;

    public static int count = 0;

    public Room(string type, float area)
    {
        ChangeNumRooms();
        this.type = type;
        this.area = area;
        this.num = count;
    }

    public Room(string type):this(type, 0)
    {

    }

    private static void ChangeNumRooms()
    {
        count++;
    }

    public static void ShowNumberOfObjects()
    {
        Console.WriteLine(count.ToString());
    }
}


internal class Program
{
    static void Main(string[] args)
    {
        Room.ShowNumberOfObjects();

        Room room1 = new Room("Bedroom", 56.7f);
        Room room2 = new Room("Kitchen");
        Room room3 = new Room("Living room", 87.9f);

        Room.ShowNumberOfObjects();
    }
}
```

3. Спадкування

Спадкування дозволяє створювати нові класи, які повторно використовують, розширюють і змінюють поведінку, визначену в іншому класі. Клас, члени якого успадковуються, називається *базовим класом*, а клас, який успадковує ці члени, називається *похідним класом*.

Похідний клас може мати тільки один прямий базовий клас. Однак успадкування є *транзитивним*. Якщо клас ClassC є похідним від ClassB, і ClassB є похідним від ClassA, ClassC успадковує члени, оголошені в ClassB і ClassA.

У похідному класі можна додати більше членів. Таким чином, похідний клас розширює функціональність базового класу.

Приклад:

```
class ім'я похідного класу : ім'я базового класу
{

}
```

У всіх конструкторах C#, крім System.Object, конструктори базового класу викликаються прямо перед виконанням першого рядка конструктора. Ці ініціалізатори конструкторів дозволяють задавати клас та конструктор, який підлягає виклику. Вони бувають двох видів.

Ініціалізатор у вигляді *base*(...) активізує конструктор базового класу поточного класу.

Ініціалізатор у вигляді *this*(...) дозволяє поточному базовому класу викликати інший конструктор, визначений у ньому самому. Це корисно, коли ви перевантажили кілька конструкторів і хочете бути впевненими, що завжди буде викликаний конструктор за замовчуванням.

Наприклад:

```
namespace ConsoleApplication1
{
    class Person
    {
        public string name; // имя
        public int age;     // возраст

        public Person() : this("Неизвестно")
        {
            Console.WriteLine(" Отработал конструкор Person()");
        }
        public Person(string name) : this(name, 0)
        {
            Console.WriteLine(" Отработал конструкор Person(string name)");
        }

        public Person(string name, int age)
        {
            this.name = name;
            this.age = age;
            Console.WriteLine(" Отработал конструкор Person(string name, int age)");
        }

        public void GetInfo()
        {
            Console.WriteLine($" Имя: {name}  Возраст: {age}");
        }
    }
```

Результат:

```
Отработал конструкор Person(string name, int age)
Отработал конструкор Person(string name)
Отработал конструкор Person()
Имя: Неизвестно  Возраст: 0
Отработал конструкор Person(string name, int age)
Имя: Richard  Возраст: 25
Отработал конструкор Person(string name, int age)
Отработал конструкор Person(string name)
Имя: Alex  Возраст: 0
```

При спадкуванні нерідко виникає необхідність при спадкуванні змінити в класі-спадкоємця функціонал методу, який був успадкований від базового класу. В цьому випадку клас-спадкоємець може перевизначати методи і властивості базового класу.

Методи і властивості, які ми хочемо зробити доступними для перевизначення, в базовому класі позначається модифікатором virtual. Такі методи і властивості називають *віртуальними*.

А щоб перевизначити метод в класі-спадкоємця, цей метод визначається з модифікатором override. Перевизначення методу в класі-спадкоємця повинен мати той же набір параметрів, що і віртуальний метод в базовому класі.

Розглянемо приклад:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public virtual void Display()
    {
        Console.WriteLine($" {FirstName} {LastName}");
    }
}


class Employee : Person
{
    public string Company { get; set; }
    public Employee(string firstName, string lastName, string company)
        : base(firstName, lastName)
    {
        Company = company;
    }

    public override void Display()
    {
        base.Display();
        Console.WriteLine($" работает в {Company}");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine();

        Person p1 = new Person("Bill", "Gates");
        p1.Display(); // вызов метода Display из класса Person

        Employee p2 = new Employee("Tom", "Smith", "Microsoft");
        p2.Display(); // вызов метода Display из класса Employee

        Console.ReadLine();
    }
}
```

```
Bill Gates
Tom Smith
работает в Microsoft
```

Також можна заборонити перевизначення методів і властивостей. В цьому випадку їх треба оголошувати з модифікатором sealed. При створенні методів з модифікатором sealed треба враховувати, що sealed застосовується в парі з override, тобто тільки в переобумовлених методах.

Наприклад:

```
public override sealed void Display()
{
    base.Display();
    Console.WriteLine($" работает в {Company}");
}
```

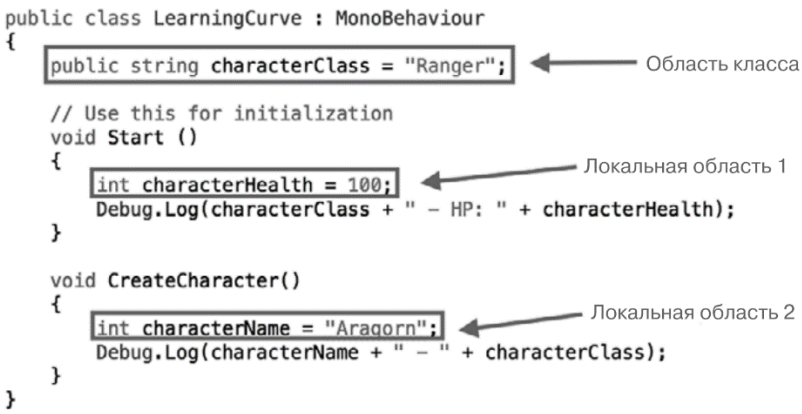
4. Скрипти в Unity

Область видимості змінної – це термін, що використовується для опису того, де існує дана змінна і звідки вона доступна в її класі.

У C# є три основні рівні області видимості змінних:

- Область видимості global вибирається для змінних, яких може отримати доступ вся програма (у разі гра). C# безпосередньо не підтримує глобальні змінні, але ця концепція корисна у певних випадках;
- Область видимості class або member вибирається для змінних, доступних у будь-якому місці класу, що містить.
- Область видимості local вибирається для змінної, доступної лише всередині певного блоку коду, в якому вона створена.

Наприклад:



У вікні Inspector в Unity можна переглядати лише змінні класу, що не підходить для локальних чи глобальних змінних.

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. 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 оканчується, 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*.

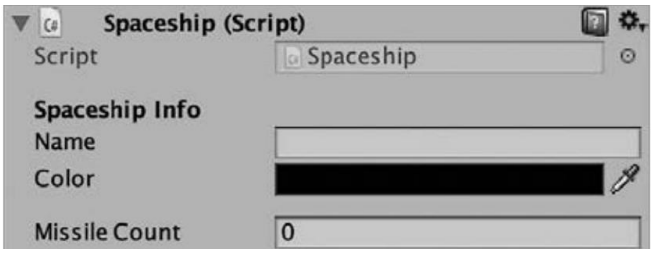
Unity створює мітку у вікні Inspector шляхом додавання пробілу скрізь, де в імені змінної зустрічається велика літера. Однак це виключно для відображення, і ви повинні завжди в коді використовувати ім'я змінної.

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:

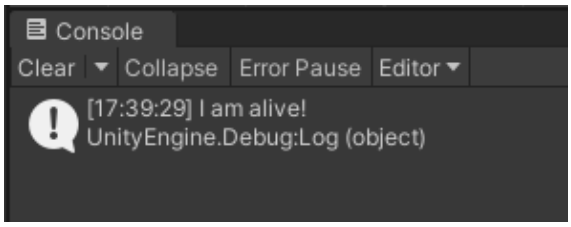
```
public class Spaceship : MonoBehaviour
{
[Header("Spaceship Info")]
public string name;
public Color color;

[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`:

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

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");
}
```

Контрольні питання

1. Що таке клас?
2. Що таке конструктор? Правила створення конструкторів?
3. Що таке перевантаження конструкторів та методів?
4. Для чого використовуються ключові слова private, public, protected? В чому різниця між нимим?
5. Що таке спадкування?
6. Скрипти в Unity спадкуються від якого класу?
7. Яка з функцій Start чи Update починає працювати першою?

Лекція 4. Введення в play-технології

1. Структура типової ігрової команди
2. Типи користувачів
3. Жанри комп'ютерних ігор
4. Монетизація в іграх
5. Життєвий цикл при розробці ігор

1. Структура типової ігрової команди

Перш ніж поринути у вивчення структури типового ігрового двигуна, коротко розглянемо структуру типової команди розробки гри.

Ігрові студії зазвичай формуються із фахівців п'яти основних напрямів:

- Розробників;
- Художників;
- Геймдизайнерів;
- Продюсерів;
- Іншого управлінського персоналу та персоналу підтримки.

Розробники проектують та створюють програмне забезпечення, що змушує гру та інструменти працювати.

Вони часто поділяються на дві групи:

- розробники середовища виконання (працюють над двигуном та грою як такою);
- розробники інструментів (створюють офлайнові інструменти, що дозволяють решті розробників діяти ефективніше).

Художники створюють весь візуальний та аудіоконтент гри. Для роботи потрібні художники та дизайнери усіх напрямків.

Творці концепт-артів створюють скетчі та замальовки, що дозволяють команді побачити результат розробки гри. Вони починають раніше за інших – на стадії розробки концепції, але зазвичай забезпечують візуальну спрямованість проекту протягом усього життєвого циклу.

Розробники 3D-моделей створюють тривимірну геометрію для всього у віртуальному світі гри. Сюди зазвичай входять:

- розробники моделей переднього плану;
- розробники заднього плану.

Ті, хто працює над переднім планом, створюють об'єкти, персонажів, транспорт, зброю – все, що наповнює ігровий світ.

Розробники моделей заднього плану створюють геометрію статичного тла – рослинність, будівлі, мости тощо.

Художники з текстур створюють двовимірні зображення, звані текстурами, які накладаються на поверхні 3D-моделей для забезпечення деталізації та реалізму.

Фахівці з освітлення створюють в ігровому світі джерела світла, як статичні, так і динамічні, працюють з кольором, інтенсивністю та напрямом освітлення, щоб забезпечити максимальну художність та емоційну дію кожної сцени.

Аніматори надають рухомим персонажам та об'єктам гри динамічність. Ігровий аніматор повинен мати унікальний набір навичок, щоб виконати анімацію, яка бездоганно узгоджуватиметься з технологічними особливостями ігрового двигуна.

Звукорежисери працюють разом з розробниками, щоб створити та змішати звукові ефекти та музику у грі. Актори озвучки наділяють персонажів голосами у більшості ігор.

Робота геймдизайнера полягає в розробці інтерактивної складової взаємодії користувача з грою, званої геймплеєм. Різні дизайнери працюють на різних рівнях деталізації.

Деякі (як правило, старші) геймдизайнери діють на макрорівні, визначаючи сюжет гри, загальну послідовність розділів або рівнів, а також основні цілі та завдання гравця.

Інші дизайнери працюють над певними рівнями або географічними областями всередині віртуального світу гри, створюючи шари геометрії статичного фону, визначаючи, коли і звідки з'являться вороги, розміщуючи такі запаси, як зброя та відновлюючи здоров'я аптечки, розробляючи елементи внутрішньо ігрових загадок тощо.

Деякі ігрові команди наймають одного чи кількох ігрових письменників. Робота цього фахівця може змінюватись від співпраці зі старшими геймдизайнерами з метою конструювання сюжету всієї гри до написання окремих ліній діалогів.

Деякі старші дизайнери відіграють роль менеджерів. У багатьох ігрових командах є геймдиректор, робота якого полягає у спостереженні за всіма аспектами геймдизайну, допомоги у керуванні термінами та забезпеченні того, щоб діяльність усіх дизайнерів була впорядкована протягом усієї роботи над продуктом. Старші дизайнери іноді виростають у продюсерів.

2. Типи користувачів

Для кого ми робимо ігри?

Багато розробників роблять ігри для себе, щоб отримати досвід або тому, що їм самим подобається конкретний жанр.

Цей підхід має право на існування, проте більшість гейм-дизайнерів все ж таки хочуть, щоб з їхнім дітищем познайомилося якнайбільше людей і, що важливо, щоб гравці гідно його оцінили.

Буває, що смаки гейм-дизайнера та аудиторії не збігаються.

Чим більше ви знатимете про переваги вашої аудиторії, тим краще зможете передбачити, які особливості вашої гри можуть принести їм задоволення. Більше того, у вас буде розуміння, де взагалі цю аудиторію шукати, через які канали просувати продукт і так далі.

Сьогодні конкуренція у промисловості більш ніж серйозна. На день виходить кілька тисяч ігор, і яку б чудову гру ви не створили, доведеться докласти зусиль, щоб про неї взагалі хоч хтось дізнався.

Найвідомішою класифікацією гравців за відданим їм типу задоволення є класифікація геймдизайнера і професора Університету Ессекса Річарда Бартла. Він ділить людей, які грають у розраховані на багато користувачів ігри, на чотири групи: ачивери, кілери, соціальники і дослідники. Багато законів взаємодії, помічені Бартлом, можна застосовувати і для однокористувальних ігор.

Головне задоволення Ачіверів (ще їх називають «кар'єристами») – це виклик. Вони виконують різні ігрові цілі в гонитві за радістю досягнення певних результатів, тобто для них важливі саме ігрові дії, що виконуються, і наступні за ними досягнення. Зазвичай такі гравці прагнуть збирання різних ресурсів, умінь, артефактів, нагород. Загалом їм важливий внутрішньоігровий прогрес як можливість досягти успіху. Практично всі ігрові платформи мають систему досягнень, і багато гравців з радістю витрачають час на гру, щоб отримати колекційну картку або значок.

Кілерам потрібна перемога. Цей тип гравців шукає змагань, насамперед з іншими людьми, щоб показати, що вони найкращі. Головне для кілерів – так чи інакше зіставляти себе з іншими гравцями, відчувати свою значущість та перевагу, причому домагатися цього швидко – рутини кілери не люблять.

Соціальники найбільше цінують взаємини з іншими гравцями, почуття товариства та власну популярність. Часто такі люди, знаходячи щось цікаве у вашій грі, наводять своїх друзів та знайомих. Саме вони найчастіше та найактивніше користуються внутрішньоігровим чатом, беруть участь у житті форумів, залишають відгуки, стають лідерами думок.

Дослідники воліють взаємодію з ігровим світом, прагнучи відкрити його у всій повноті. Бої та активні дії для них відходять на другий план. Дослідники цінують різноманіття ігрового контенту, сюжет, різні ігрові механіки, що дають змогу досконало вивчити можливості гри; рейтинги та змагання їм нецікаві.

За чотирма психотипами Бартла зібрано багато корисної інформації (метрики з повернення в гру, відсоток гравців, що платять тощо), тому для визначення своєї аудиторії гейм-дизайнери найчастіше користуються саме цією спрощеною системою.

3. Жанри комп'ютерних ігор

У книзі Рефа Костера «Розробка ігор і теорія розваг» *гра* визначена як інтерактивний досвід гравця, реалізований у вигляді послідовності шаблонних дій, які він або вона освоює і в кінцевому рахунку виконує. Костер стверджує, що дії освоєння та виконання у грі є головними в тому, що ми називаємо «розвагою».

Термін «*ігровий движок*» – це програмне забезпечення, яке розширюється і може застосовуватися як основа для безлічі різних ігор без значних модифікацій. Більшість ігрових двигунів розроблені та ретельно налаштовані для запуску конкретної гри на конкретній апаратній платформі.

Unity развивается с 2005 года. На Unity разработаны тысячи приложений и игр разных жанров на более чем 25 платформах. Изначально Unity создавался для компьютеров Mac, в следующих версиях добавлялись новые платформы: Windows, IPhone, Android, Xbox, Playstation и другие. Для написания скриптов движок использует язык программирования C#. В редакторе Unity простой интерфейс, что позволяет легко производить отладку игры. Главными преимуществами Unity считают кросс-платформенность и наличие визуальной среды разработки.

Unreal Engine випустила у 1998 році компанія Epic Games. Це один з перших універсальних двигунів, що поєднують фізику, рендеринг, штучний інтелект і готове середовище розробки. Спочатку він створювався для роботи над шутерами від першої особи, проте наступні версії застосовуються для ігор різних жанрів. Unreal Engine завжди приділяв багато уваги вражаючому малюнку, ним користуються і в кінематографі; сьогодні він надає просунутий інструментарій для створення фотореалістичної 3D-графіки у реальному часі.

Жанри комп'ютерних ігор, з одного боку, мають досить чіткі межі, але з іншого – нерідко досить складно класифікувати ту чи іншу гру в межах одного жанру. У деяких іграх жанри переплітаються, деякі ігри створюють власні жанри.

Platformers (платформні ігри або платформери) це ігри, де герой подорожує ігровим світом, переходячи з платформи на платформу – стрибаючи над прірвами, перепливаючи річки, дертися на стіни і попутно борючись з лиходіями. Спочатку платформери – це двовимірні ігри з видом збоку.

Action – це ігровий жанр, який цікавий тим, що гравець під час ігрового процесу має постійно щось робити. Найбільш популярне заняття у таких іграх – це знищення ворогів. А найпопулярнішим видом екшенів стали так звані шутери.

Типовий сценарій *шутерів* – ви керуєте будь-яким механізмом (або героєм), блукаєте ігровим світом (або переміщаєтеся в обмежених межах) і знищуєте полчища ворогів.

FPS або *First Person Shooters* – тобто "стрілялки від першої особи" - це ігри, які вже багато років перебувають у стані постійного вдосконалення та розвитку. Як правило, у такій грі популярний вигляд з очей героя – ви бачите ігровий світ так, як бачить його герой. Тривимірний світ у таких іграх дуже важливий – тому перші помітні FPS з'явилися на початку 1990-х з розвитком тривимірної ігрової графіки.

Ігри стилю *Survival Horror* ("жахи", "ігри на виживання") – це ігри, як правило, що мають властивості FPS-ігор, проте крім звичайних для битв елементів, містять елементи

жанру жахів. Гравця оточує похмура атмосфера, монстри, він потрапляє у важкі колотнечі, які повинні тримати його у постійній напрузі

Ігри в стилі *Fighting*, бійки – це ігри, де ви вступаєте в бій з віртуальним противником, який або управляється комп'ютером, або - людиною. Цей жанр виник у середині 1980-х років і популярний досі.

Ще один стиль ігор – це суміш файтингу та платформерів – в англійському варіанті вони називаються *Beat'em up*. Як правило, ціль такої гри – пройти гру, знищивши всіх ворогів. Герой гри зазвичай користується якоюсь зброєю, проте система ударів і рухів у таких іграх зазвичай проста – кілька базових ударів, плюс як мінімум один "суперудар", який дозволить "розкидати" цілу юрбу ворогів. Ці ігри були особливо популярні у 1990-х роках.

RPG – Role Playing Game (рольові ігри) – ігри, де ви повинні відігравати роль будь-якого персонажа. Перші рольові ігри виникли завдяки автоматизації класичних ігор, у які грали без участі комп'ютерів – популярність таких ігор зумовила популярність комп'ютерних RPG. Зазвичай події у таких іграх розвиваються хід за ходом, у сучасних RPG є епізоди (бої, наприклад), які передбачають гру в реальному часі.

Puzzle (головоломки) – це ігри, які зазвичай не вимагають серйозних графічних можливостей системи – їхня головна цінність в ідеї гри.

RTS – Real Time Strategy (стратегії реального часу) – це ігри, де ви, як правило, керуєте будь-якою армією, причому події розвиваються в реальному часі.

Massively Multiplayer Online Games (MMO) – або онлайнові ігрові світи – це ігри, в яких люди буквально живуть день і ніч, підключившись до ігрового серверу через Інтернет. Існує кілька різновидів цих ігор. Перша скорочено називається MMORPG – тобто онлайнова рольова гра, а друга – MMOFPS – онлайнова стрілялка. Перші ігри цього стилю з'явилися торік у 1970-х роках – тоді це були текстові ігри. Пізніше, на початку 1990-х, вони набули графічного інтерфейсу і з того часу розвиваються, займаючи уми все більшої кількості гравців.

Simulator (симулятор) – це гра, покликана як найточніше (чи хоча б правдоподібно) щось імітувати. Популярність симуляторів полягає в тому, що вони дозволяють «не виходячи з дому» відчути себе за штурвалом літака або вертольота, покерувати гоночною машиною або посмикати за важелі танка, стати директором транспортної компанії, політати в космосі, повоювати і таке інше. Розвиток цих ігор полягає у підвищенні якості графічної складової гри, у покращенні імітації реальних подій.

Клас *Adventure* включає ігри пригодницької тематики. Пік популярності цих ігор припав на початок 1990-х років. Під час гри гравець розгадує загадки, вирішує головоломки, спілкується з іншими персонажами. Гра чимось нагадує RPG, проте тут персонаж зазвичай не піддається розвитку – всю гру можна представити у вигляді якогось наперед визначеного плану дій, який ви проходите за допомогою вашого персонажа.

Casual games – це ігри, призначені для широкої аудиторії користувачів комп'ютерів та мобільних пристроїв. Вони не мають чітких жанрових обмежень, однак, це зазвичай ігри досить прості в освоєнні, на проходження рівнів таких ігор не потрібно багато часу. У такі ігри грають користувачі соціальних мереж, власники планшетів, люди, які мають мобільні телефони чи ігрові консолі.

Віртуальна реальність (virtual reality, VR) може бути визначена як багатоаспектна мультимедійна або комп'ютерна реальність, що імітує присутність користувача в середовищі, яке є місцем у реальному чи уявному світі.

Створена комп'ютером віртуальна реальність (computer-generated (CG) VR) є підмножиною цієї технології, в ній віртуальний світ генерується виключно за допомогою комп'ютерної графіки. Користувач може побачити це віртуальне середовище, надягаючи гарнітуру.

Гарнітура відображає контент прямо перед очима користувача, при цьому система відстежує її переміщення в реальному світі, так що рухи віртуальної камери ідеально

відповідають рухам людини, що наділа гарнітуру. Користувач зазвичай тримає пристрої, які дозволяють системі відстежувати рухи обох рук. Це дає можливість гравцеві взаємодіяти із віртуальним світом.

Терміни «*доповнена реальність*» (augmented reality, AR) та «*змішана реальність*» (mixed reality, MR) часто плутають або використовують взаємозамінно. Обидві технології представляють користувачеві реальний світ із комп'ютерною графікою.

В обох пристрій перегляду, такий як смартфон, планшет або технічно допрацьована пара окулярів, виводить статичні зображення реального світу або що змінюється в реальному часі, а поверх нього накладається комп'ютерна графіка.

Деякі розрізняють ці дві технології, використовуючи термін «доповнена реальність» для опису технологій, у яких комп'ютерна графіка накладається зображення реального світу, але з прив'язується до нього.

Термін «змішана реальність» частіше застосовується до використання комп'ютерної графіки для візуалізації уявних об'єктів, які прив'язані до реального світу і здаються існуючими в ньому. Однак ця відмінність у жодному разі не є загальноприйнятою.

4. Монетизація в іграх

Монетизація в іграх – це конвертація гравців у гроші, незалежно від того, заробляємо ми на продажі самої гри або контенту. Вибір монетизаційної стратегії залежить від особливостей проекту, регіону та переваг цільової аудиторії.

Розглянемо деякі моделі монетизації:

- BUY-TO-PLAY – «купи, щоб грати». Гра виставляється в магазинах, і щоб отримати доступ до неї, гравець повинен оплатити покупку.
- PAY-TO-PLAY (підписковa модель) передбачає, що для постійного доступу до ігрового контенту потрібно оформити платну передплату.
- FREE-TO-PLAY – монетизаційна модель, що дозволяє грати без обов'язкового внесення коштів. Такі ігри заробляють, пропонуючи гравцям за гроші отримати ігрову перевагу, унікальні предмети, відкрити новий ігровий контент тощо. Вбудовані покупки допомагають гравцям просунутися в грі: за додаткову плату можна прискорити процес прокачування, отримати потужні ігрові предмети та інші бонуси. Часто розробники пропонують купити чисто декоративні елементи, що дають змогу зовні виділитися серед інших гравців.
- ВБУДОВАНА РЕКЛАМА – ще одна важлива модель монетизації. Вона дозволяє розробникам заробляти завдяки тому, що гравцеві показується реклама інших ігор, додатків, послуг тощо.
- FREEMIUM – монетизаційна модель, що пропонує гравцю безкоштовну обмежену версію гри з можливістю придбати розширену або premium-версію. Відмінність від класичного Free-to-play у тому, що гравцеві до внесення оплати надають доступ до спочатку урізаного геймплей або обмеженого набору контенту.
- PRODUCT PLACEMENT, тобто наявність неявної (прихованої) реклами – також можливість отримати прибуток. Втома гравців від нав'язливої прямої реклами підштовхнула розробників ігор до партнерства з різними брендами.

5. Життєвий цикл при розробці ігор

На рисунку 4.1 продемонстровано робочий процес розробника ігор.

Рисунок 4.1 – Робочий процес розробника ігор

Розглянемо специфічні дії при створенні комп'ютерних ігор в рамках життєвого циклу.

Концептування (Concept) – на цьому першому етапі команда розробляє концепцію гри, і проводить початкове опрацювання ігрового дизайну. Головна мета даного етапу – це геймдизайнерська документація, що включає розгорнутий документ, що описує гру, як кінцевий бізнес-продукт і Concept Document (початкове опрацювання всіх аспектів гри).

Розробка гри починається із створення концепції гри. Власне, *концепція* – це центральна ідея, навколо якої будується все інше. Концепція гри виражається як *концепт-документ*.

Концепт-документ є першим ігровим документом, що дозволяє отримати загальне уявлення про гру, фіксує її основні особливості. Концепт-документ може будуватися за такою схемою:

– Вступ;
– Жанр та аудиторія;
– Основні особливості гри;
– Опис гри;
– Порівняння та передумови створення гри;
– Платформа;
– Контакти.

Дизайн документ містить детальний опис гри, ігрових ресурсів, ігрової логіки. Дизайн-документ можна вважати розширеною версією концепт-документу.

Дизайн-документ може мати таку структуру:

1. Введення
2. Концепція
    2.1. Вступ
    2.2. Жанр та аудиторія
    2.3. Основні особливості гри
    2.4. Опис гри
    2.5. Передумови створення
    2.6. Платформа
3. Функціональна специфікація
    3.1. Принципи гри
        3.1.1. Суть ігрового процесу
        3.1.2. Хід гри та сюжет

3.2. Фізична модель
3.3. Персонаж гравця
3.4. Елементи гри
3.5. "Штучний інтелект"
3.6. Багатокористувацький режим
3.7. Інтерфейс користувача
3.7.1. Блок-схема
3.7.2. Функціональний опис та управління
3.7.3. Об'єкти інтерфейсу користувача
3.8. Графіка та відео
3.8.1. Загальний опис
3.8.2. Двовимірна графіка та анімація
3.8.3. Тривимірна графіка та анімація
3.8.4. Анімаційні вставки
3.9. Звуки та музика
3.9.1. Загальний опис
3.9.2. Звук та звукові ефекти
3.9.3. Музика
3.10. Опис рівнів
3.10.1. Загальний опис дизайну рівнів
3.10.2. Діаграма взаємного розташування рівнів
3.10.3. Графік запровадження нових об'єктів
4. Контакти

У *вступі* міститься коротке формулювання ідеї гри. Введення має містити інформацію, яка дозволить читачеві зрозуміти сутність гри, її жанр, аудиторію. Фактично, введення – це той мінімум інформації, який дозволяє читачеві – видавцеві, майбутньому члену команди розробників, майбутньому гравцю – зрозуміти – чи цікава йому дана гра, чи хоче він мати до неї відношення.

Відомості про *жанр гри* та *цільової аудиторії* дуже важливі, оскільки дозволяють зрозуміти особливості позиціонування гри, особливості майбутнього просування готового продукту. Як правило, цей розділ має сенс включати інформацію про цільові групи користувачів, яким може бути цікава гра. Тут же корисно навести результати попередньо проведеного дослідження.

Таке дослідження можна провести наступним чином: скласти питання, що складається з кількох питань, які призначені для з'ясування ставлення гравців до тих чи інших концепцій, покладених в основу гри. Питання має сенс складати як анкети, що містить варіанти відповіді питання.

Основні особливості гри – цей розділ повинен включати опис ключових особливостей гри, що відрізняють її від ігор того ж жанру, розрахованих на ту ж цільову групу. У цьому розділі слід вказати приблизний обсяг гри. Обсяг гри може бути виміряний у годиннику або в інших одиницях.

Порівняння і передумови створення – цей розділ повинен містити порівняльний аналіз пропонованої гри з існуючими іграми, аналіз ринкових тенденцій. Тут же слід торкнутися питань ліцензування.

Прототипування (Prototyping) – важливий етап проектування будь-якої гри – створення прототипу. Те, що добре виглядає «на папері», зовсім не обов'язково буде цікавим у реальності. Прототип реалізується з метою оцінки основного ігрового процесу, перевірки різних гіпотез, проведення тестів ігрових механік, перевірки ключових технічних моментів.

Дуже важливо на етапі створення прототипу реалізовувати лише те, що потрібно перевірити й у стислий термін. Прототип може бути простим у реалізації, т.к. після досягнення поставлених перед ним цілей, він має бути «викинутий».

Вертикальний зріз (Vertical Slice) – мета вертикального зрізу – отримати мінімально можливу повноцінну версію гри, що включає повністю реалізований основний ігровий процес. При цьому високу якість опрацювання обов'язково потрібно втілити тільки для тих ігрових елементів, які суттєво впливають на сприйняття продукту. При цьому всі базові фічі гри присутні як мінімум у чорновій якості. Реалізовано мінімальний, але достатній реалізації повноцінного ігрового процесу набір контенту (один рівень чи одна локація).

Виробництво контенту (Content production) – на цьому етапі виготовляється достатня кількість контенту для першого запуску на зовнішню аудиторію. Реалізуються всі фічі, які заплановані до закритого бета-тестування. Це найбільш тривалий етап, який може займати для великих клієнтських проектів рік і більше. На цьому етапі задіяна найбільша кількість фахівців, які займаються виробництвом всього основного наповнення гри. Художники створюють усі графічні ресурси, геймдизайнери налаштовують баланс та заповнюють конфіги, програмісти реалізують та полірують усі фічі.

Friends & Family / CBT (закрите бета-тестування) – на етапі CBT продукт вперше демонструється досить широкому загалу, хоча й лояльному продукту чи компанії. Серед найважливіших завдань на цьому етапі виступають: пошук та виправлення гейм-дизайнерських помилок, проблем ігрової логіки та усунення критичних багів. На цьому етапі в грі присутні вже всі ключові фічі, створено достатньо контенту для повноцінної гри тривалий час, налаштовано збір та аналіз статистики. Тестування відбувається за тест-планом, проводяться стрес-тести вже із залученням реальних гравців.

Soft Launch / OBT (відкритий бета-тест) – на цьому етапі продовжується тестування гри, але вже на широкій аудиторії. Йде оптимізація під великі навантаження. Гра має бути готовою для прийому великого трафіку. На цьому етапі повністю завершується розробка нових фічів. Відбувається feature freeze, програмісти перестають реалізовувати щось нове, а повністю переключаються на налагодження та тюнінг наявних фічів. Геймдизайнери, продюсер та аналітики роблять висновки із зібраної на CBT статистики та перевіряють ефективність монетизації. При цьому до початку етапу повинна повністю функціонувати інфраструктура проекту: сайт, групи соц. мереж, канали залучення (User Acquisition), підтримка користувачів.

Release – на цьому етапі має бути повністю налагоджено оперування продукту (технічна підтримка, робота з ком'юніті), дотримуються маркетингові та фінансові плани, ведуться роботи з покращення фінансових показників, активно відпрацьовуються канали залучення трафіку. Команда розробки цьому етапі займається виправленням технічних багів, виявлених у процесі експлуатації та оптимізацією продукту.

Контрольні питання

1. З кого складається типова ігрова команда?
2. Для чого потрібно розуміти які типи користувачів існують?
3. Які жанри комп’ютерних ігор ви знаєте?
4. Що таке монетизація?
5. З яких етапів складається життєвий цикл при розробці ігор?

Лекція 5. Інтерфейс Unity

1. Unity Hub
2. Редактор Unity

1. Unity Hub

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.

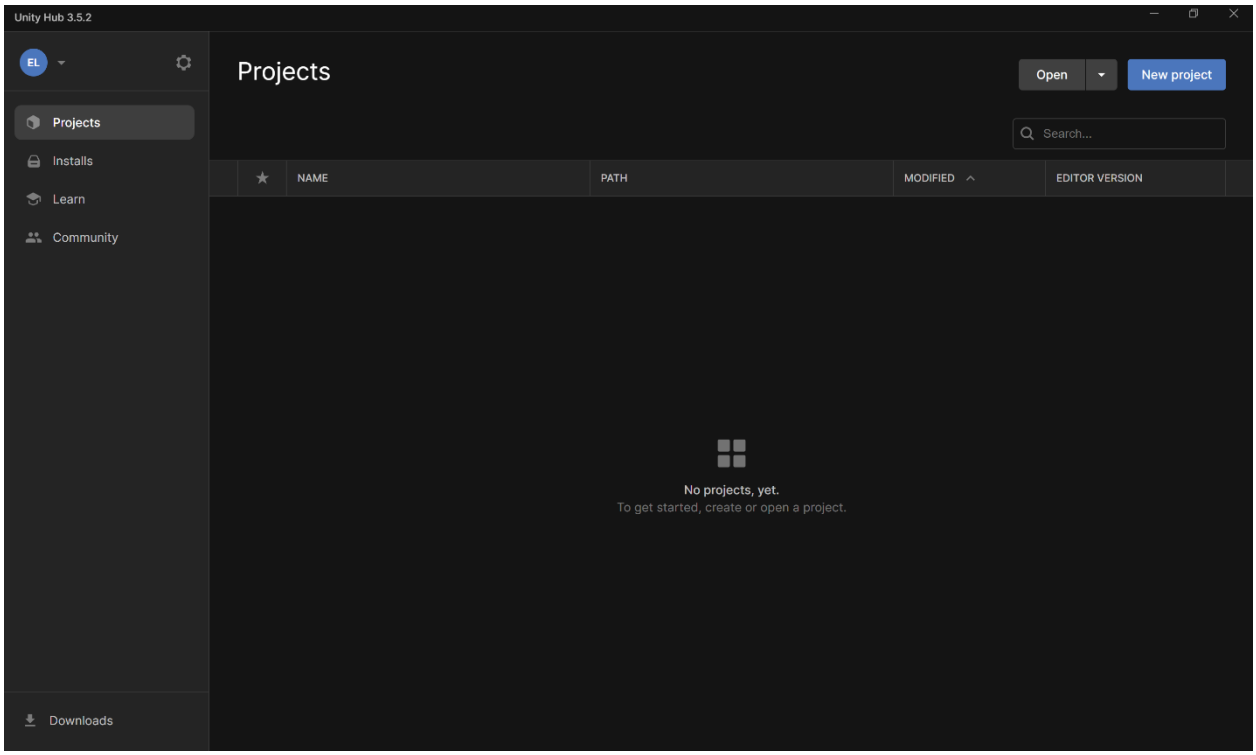


Рисунок 5.1 – Вікно Unity Hub

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.

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 2D проекту зайдіть у Unity Hub та натисніть кнопку New project.

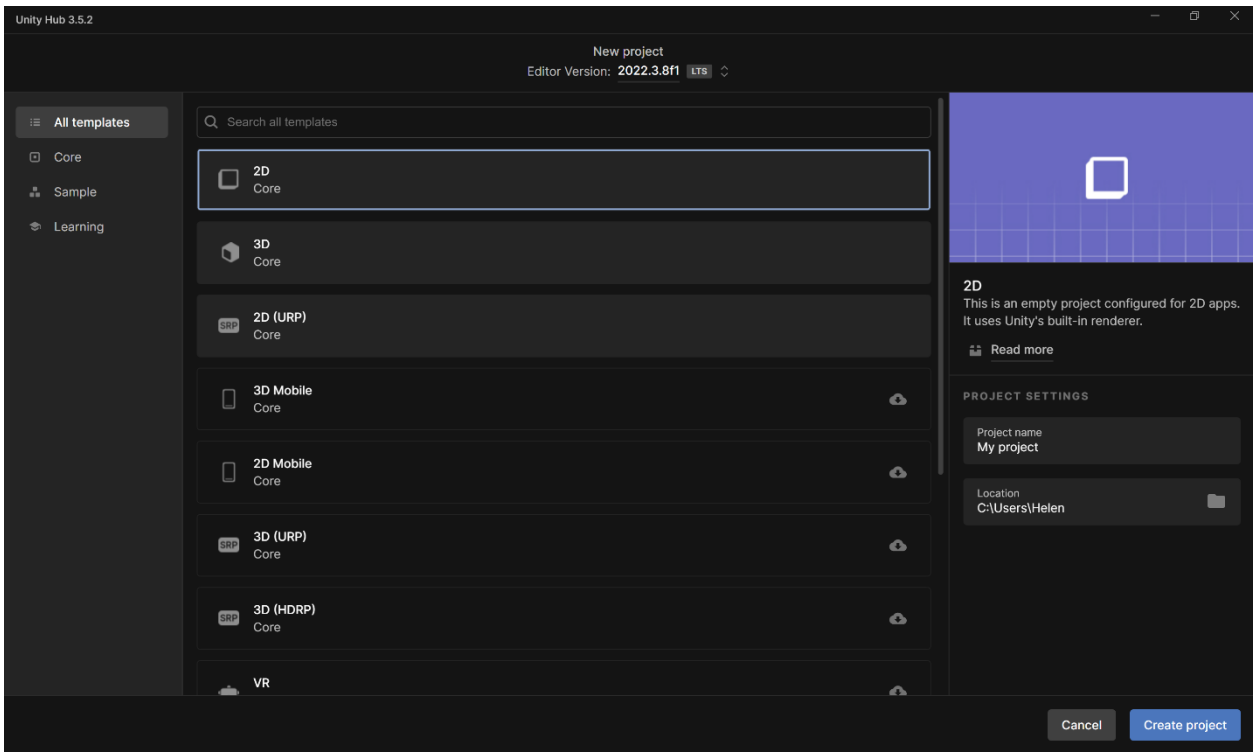


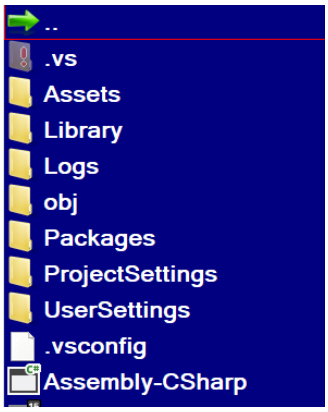
Рисунок 5.2 – Вікно для вибору шаблону для гри, що створюється

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

2. Редактор Unity

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

– Assets,
– ProjectSettings
– Library.



Папка Assets містить усі файли, які використовуються грою: рівні, текстури, звукові ефекти та сценарії. Папка Library містить внутрішні дані для Unity, а папка ProjectSettings – файли з налаштуваннями 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.

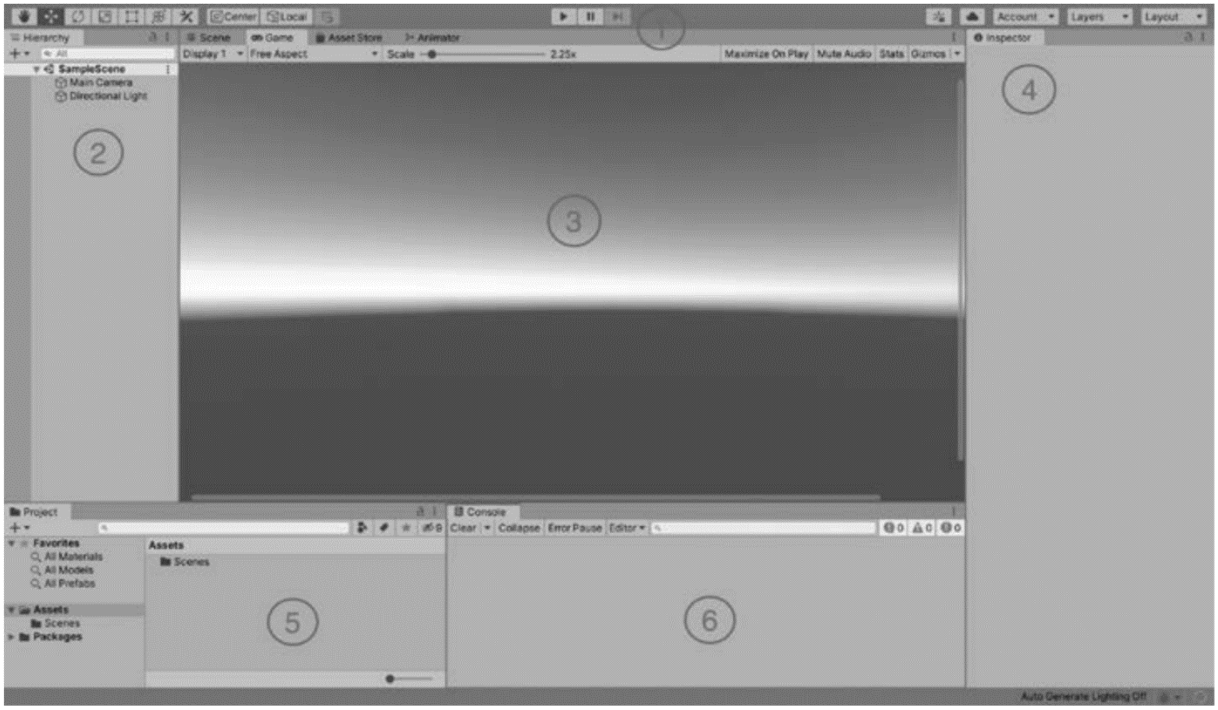


Рисунок 5.3 – Вікно редактору Unity

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

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

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

Редактор Unity завжди знаходиться в одному з двох режимів: Edit Mode (Режим редагування) або Play Mode (Режим відтворення). У режимі редагування, що діє за 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.

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.

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.

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.

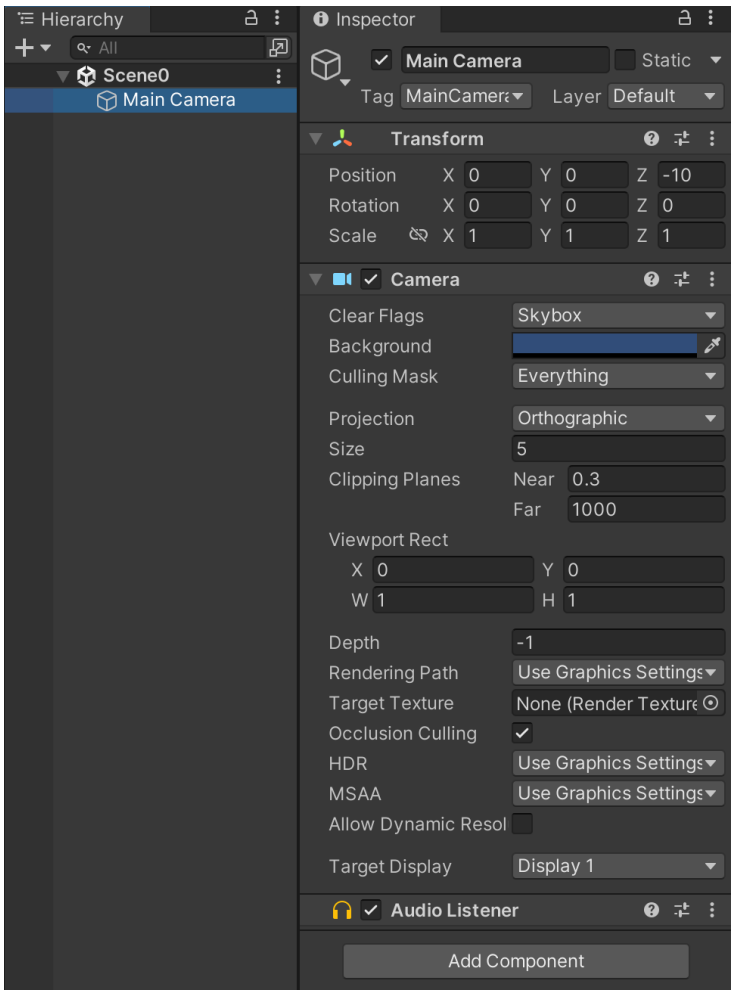


Рисунок 5.4 – Панель Hierarchy та панель Inspector для об’єкту MainCamera

Контрольні питання

1. Що таке Unity Hub та для чого він використовується?
2. Для чого використовується папка Assets?
3. Для чого використовується вкладка Hierarchy?
4. Для чого використовується панель Inspector?
5. В яких режимах може бути подання сцени?

Лекція 6. 2D гра. Основні об'єкти

1. Сцена
2. Ассети
3. GameObject

1. Сцена

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.

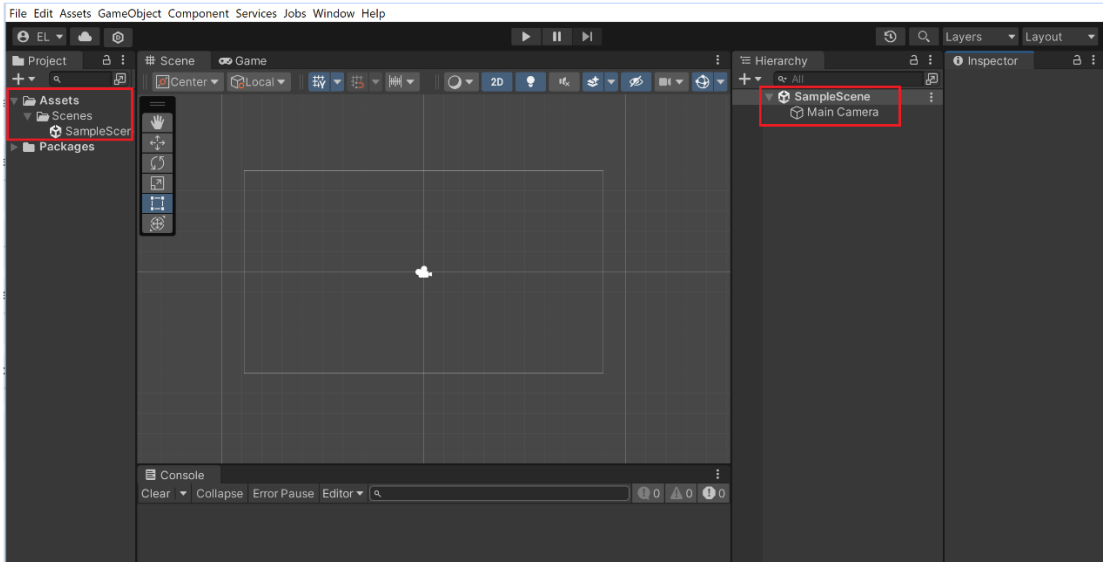


Рисунок 6.1 – Сцена в Unity

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)

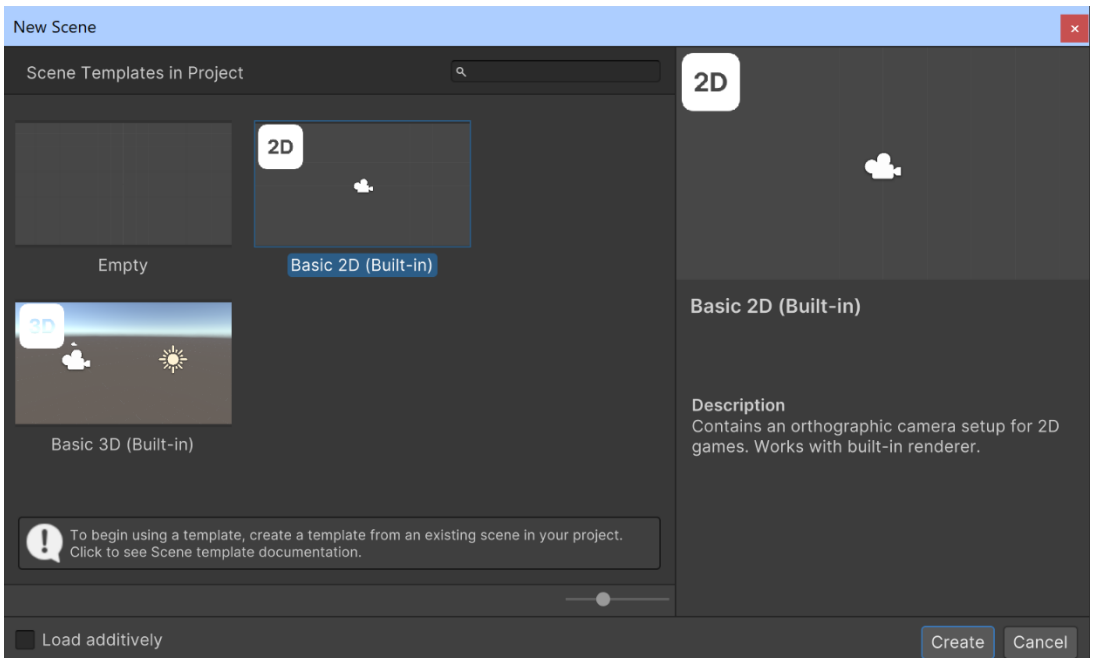
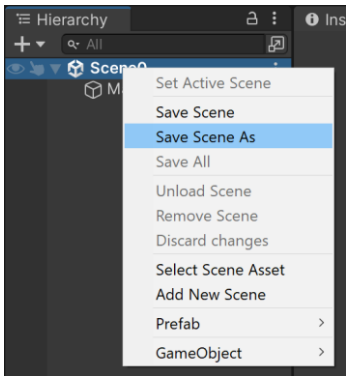


Рисунок 6.2 – Діалогове вікно «Нова сцена»

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.

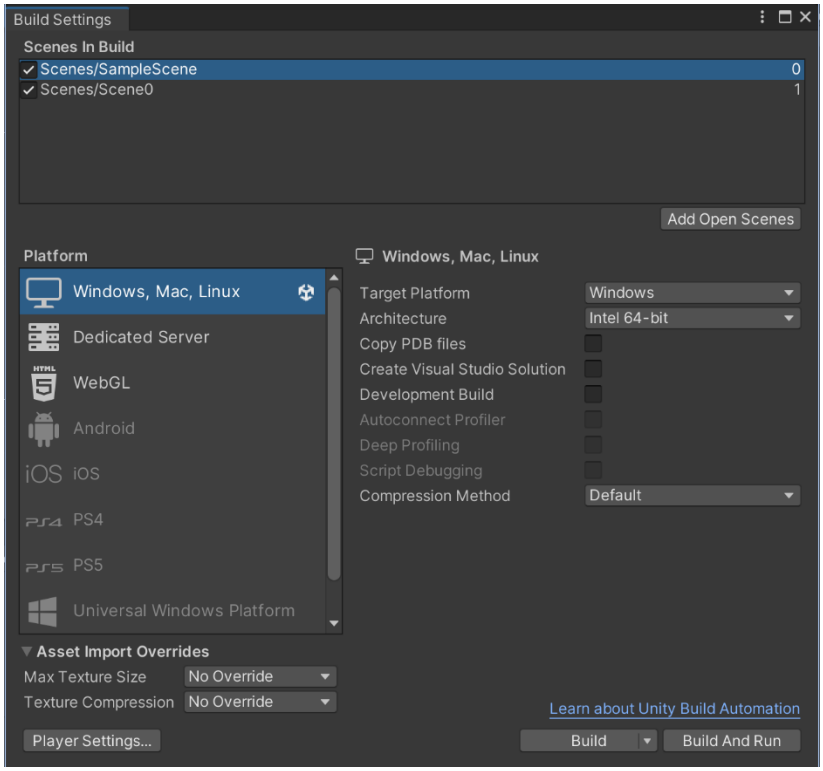


Рисунок 6.3 – Вікно «Build Settings»

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);
    }

}
```

2. Ассети

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

Папка Assets – головна папка, в якій містяться всі ассети, які можуть бути використані проектом на Unity. Вміст Project view відповідає вмісту папки Assets. Більшість функцій API припускають, що все міститься в папці Assets і не вимагають явної вказівки розташування.

Рекомендується під кожен вид ассетів створювати свою окрему папку в Assets.

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

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

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

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

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

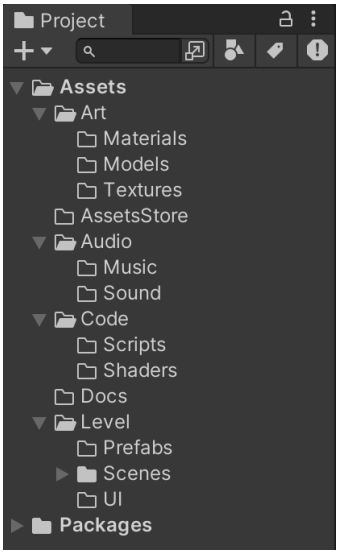
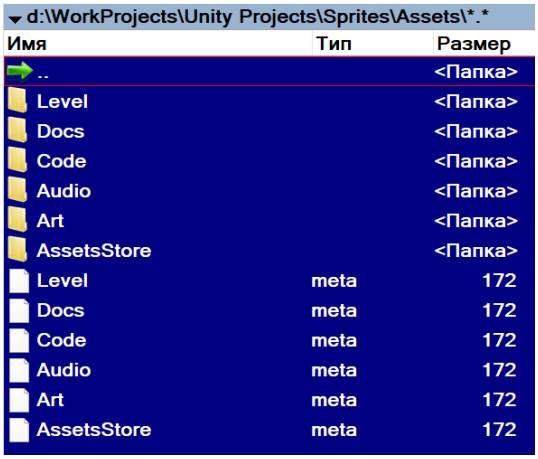


Рисунок 6.4 – Приклад структури папок в папці Assets

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

Приклад:



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

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

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

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

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

– Якщо ресурс текстури втрачає свій файл .meta, будь-які матеріали, що використовують цю текстуру, втрачають посилання на цю текстуру. Щоб це

виправити, вам потрібно буде вручну перепризначити цю текстуру всім матеріалам, для яких вона потрібна.

– Якщо ресурс сценарію втрачає свій файл. Щоб виправити це, вам потрібно буде вручну перепризначити цей скрипт для будь-яких ігрових об'єктів, яким він потрібний.

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

3. GameObject

Клас Unity GameObject використовується для представлення всього, що може існувати у Сцені.

Кожен об'єкт у вашій грі – це GameObject, від персонажів та колекційних предметів до джерел світла, камер та спеціальних ефектів.

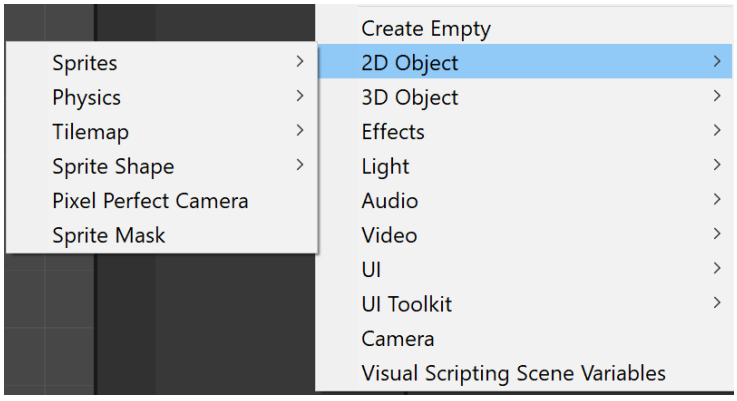
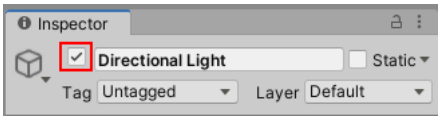


Рисунок 6.4 – Приклад GameObject

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

Статус ігрового об'єкта активний представлений прапорцем ліворуч від імені об'єкта. Ви можете керувати цим за допомогою GameObject.SetActive.



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

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

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

Наприклад:

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

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

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

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

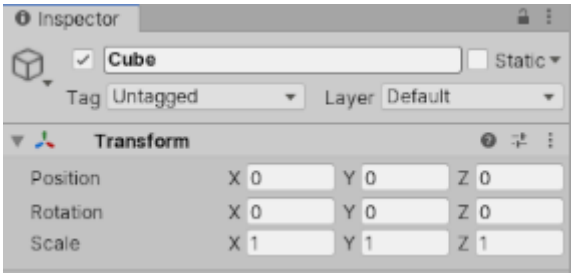


Рисунок 6.5 – Компонент Transform

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

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

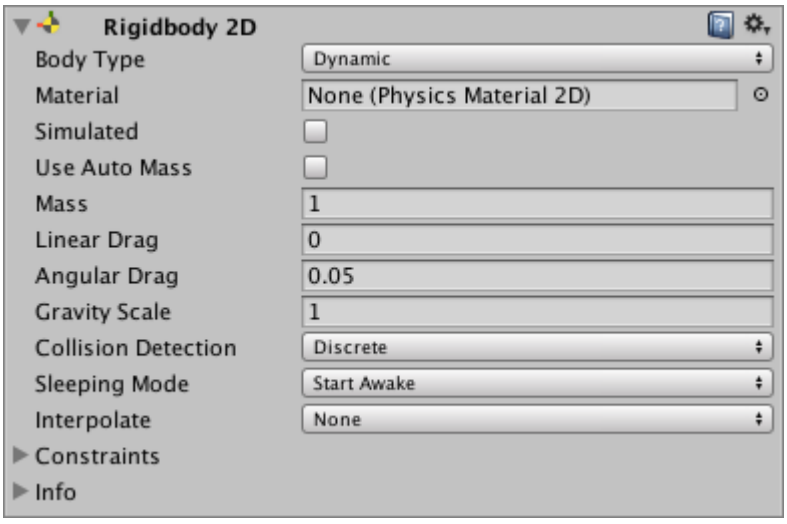


Рисунок 6.6 – Приклад компоненти Rigidbody 2D

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

- Dynamic
- Kinematic
- Static

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

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

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

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

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

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

Коли тверде тіло переміщається зі швидкістю, меншою за певний мінімальний поріг, фізичний двигун припускає, що воно зупинилося і перебуває в спокої. При цьому, об'єкт не буде знову рухатися до тих пір, поки з ним не відбудеться зіткнення або до нього не застосують силу, так що він йде в "сплячий" режим.

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

Розглянемо деякі властивості компонента Rigidbody 2D:

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

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

Найпростіші коллайдери є примітивними типами коллайдерів. У 3D це Box Collider, Sphere Collider і Capsule Collider. У 2D ви можете використовувати Box Collider 2D і Circle Collider 2D. Ви можете додати будь-яку їх кількість до одного GameObject, щоб створити складені коллайдери.

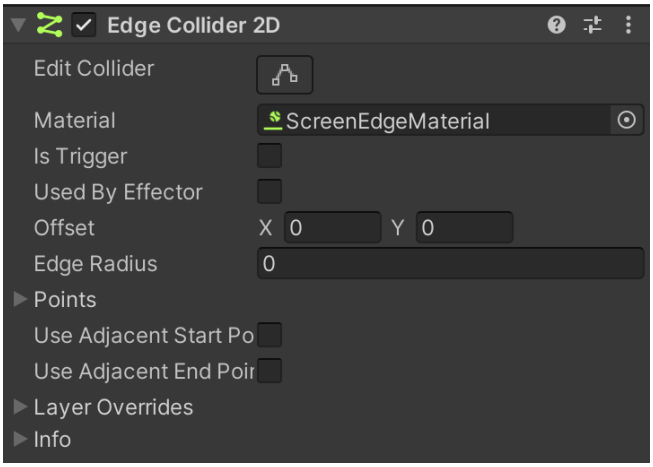


Рисунок 6.7 – Приклад компоненти Edge Collider 2D

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

Коли коллайдери взаємодіють, її поверхні потрібно імітувати властивості матеріалу, з якого вони теоретично повинні складатися. Наприклад, шар льоду буде більш скользким, в той час як резиновий м'яч буде пропонувати більше тренувань і буде дуже пружним. Хоча форма коллайдерів і не деформується під час зіткнення, їх тренування і пружність можна налаштувати за допомогою фізичних матеріалів (Physics Materials).

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

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

Контрольні питання

1. Що таке сцена та для чого вона призначена?
2. З яких ассетів складається папка Assets?
3. Що таке GameObject?
4. Для чого використовується компоненти?
5. Що робить компонент Rigidbody?
6. Що робить компонент Collider?

Лекція 7. 2D гра. Анімація персонажу

1. Спрайти
2. Анімаційні спрайти
3. Анімація

1. Спрайти

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

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

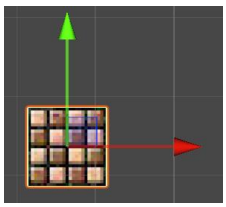
Спрайти (sprites) являють собою двовимірні зображення, що відображаються безпосередньо на екрані, в той час як такі ж зображення, що відображаються на поверхні тривимірних моделей, називаються текстурами.

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

Є кілька способів додати спрайт до свого проекту. Один із способів – просто скопіювати PNG у папку зі спрайтами за допомогою операційної системи, і тоді він стане частиною проекту. Інший спосіб зробити це перетягнути спрайт файл з File Explorer в проект.

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

Якщо вибрати ігровий об'єкт у вигляді спрайту на сцені, можна побачити компоненти, які за умовчанням прикріплені до нових ігрових об'єктів, доданих до сцени. Ми маємо засіб візуалізації спрайтів Sprite Renderer, що дозволяє візуалізувати або малювати спрайти, пов'язані з цим ігровим об'єктом. Крім того, є інструмент Transform. Transform вказує положення ігрового об'єкта в сцені, його поворот навколо осей x, y і z, а також масштаб ігрового об'єкта.



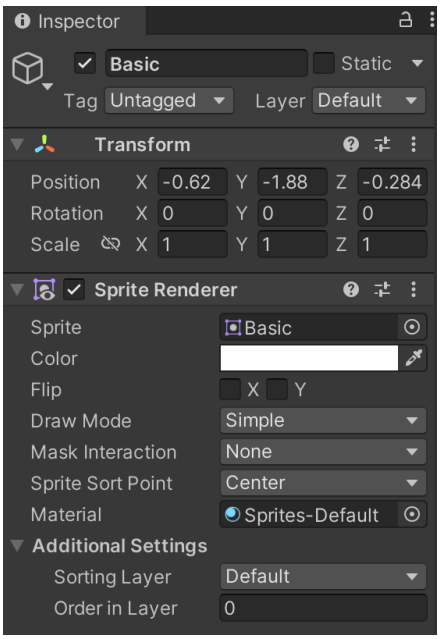


Рисунок 7.1 – Приклад спрайту на сцені та його компонентів Transform та Sprite Renderer

Розглянемо властивості Sprite Renderer.

Властивість	Функція
Sprite	Спрайт, який треба рендерувати. Спрайти можна створити з текстур, використовуючи налаштування Sprite в імпортері текстур.
Color	Колір меша, що рендериться.
Flip	Перевертає спрайт у площині X або Y.
Material	Матеріал, що використовується для рендеру спрайту.
Sorting Layer	Шар, що використовується для завдання пріоритету накладання під час рендерингу.
Order In Layer	Пріоритет накладання спрайту у межах його шару. Чим нижче число, тим раніше рендеруватиметься спрайт, а спрайти з числами вище будуть малюватись поверх тих, що нижче.

7. Анімаційні спрайти

У двомірних іграх повсюдно поширені анімовані спрайти. Вони створюються малюванням кожного кадру анімації та подальшого відтворення отриманої послідовності кадрів у Unity. Кадри можна імпортувати у вигляді окремих зображень, але їх зазвичай викладають у вигляді однієї картинки, яка називається *листом спрайтів* (sprite sheet). Листи спрайтів можна автоматично генерувати в Unity. При імпорті аркуша у списку Sprite Mode налаштувань спрайту слід вибрати варіант Multiple

Аркуш спрайтів з'являється на вкладці Project як один ресурс, але клацання на розташованій збоку стрілці розкриває його і дає можливість побачити окремі спрайти. Замість того, щоб перетягувати спрайти в сцену по одному за раз, можна перетягнути їх групою.

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

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

7. Анімація

Анімація персонажу може зберігатися в наступних видах:

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

Рисунок 7.2 – Приклад листу спрайтів

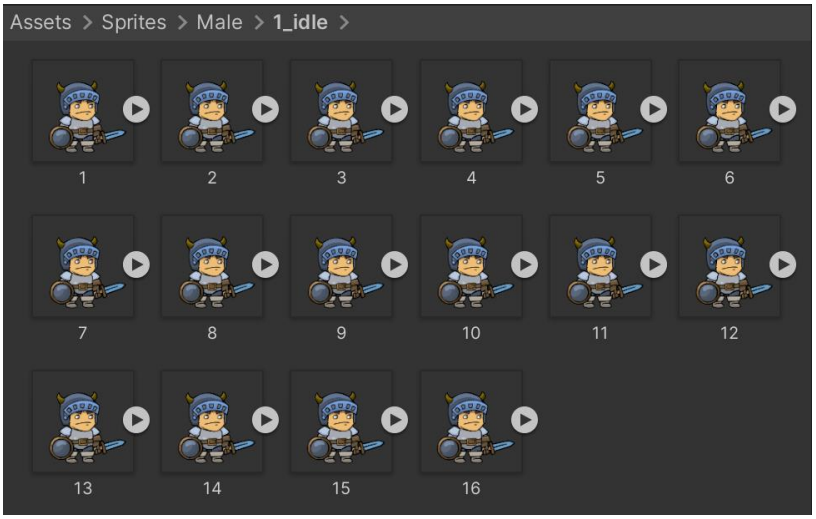


Рисунок 7.3 – Приклад анімації в окремих файлах

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

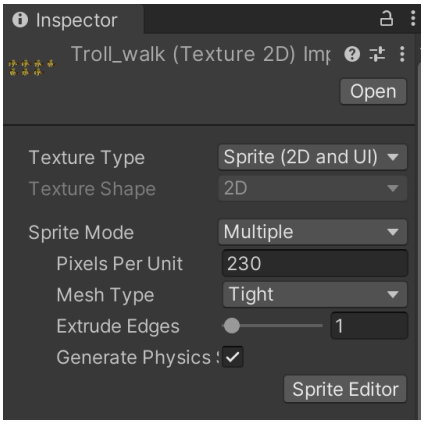


Рисунок 7.4 – Налаштування листу спрайтів в Inspector

Редактор Sprite Editor дозволяє витягувати елементи складеного зображення. Щоб визвати редактор Sprite Editor виділяємо в Project спрайт який складається з кадрів руху персонажу та Inspector натискаємо на кнопку Sprite Editor. В результаті відкриється закладка Animator.

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

Контрольні питання

1. Що таке спрайти та для чого вона призначена?
2. Які бувають спрайти?
3. Що робить компонент Sprite Renderer?
4. Що дозволяє робити редактор Sprite Editor?
5. Що робить компонент Animator?

Лекція 8. 2D гра. Управління персонажем

1. Налаштування проекту та персонажу
2. Рух персонажу

1. Налаштування проекту та персонажу

Додаймо управління до нашого персонажу. Для цього створимо скрипт, наприклад, Hero та прикріпимо його до головного персонажу.

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

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

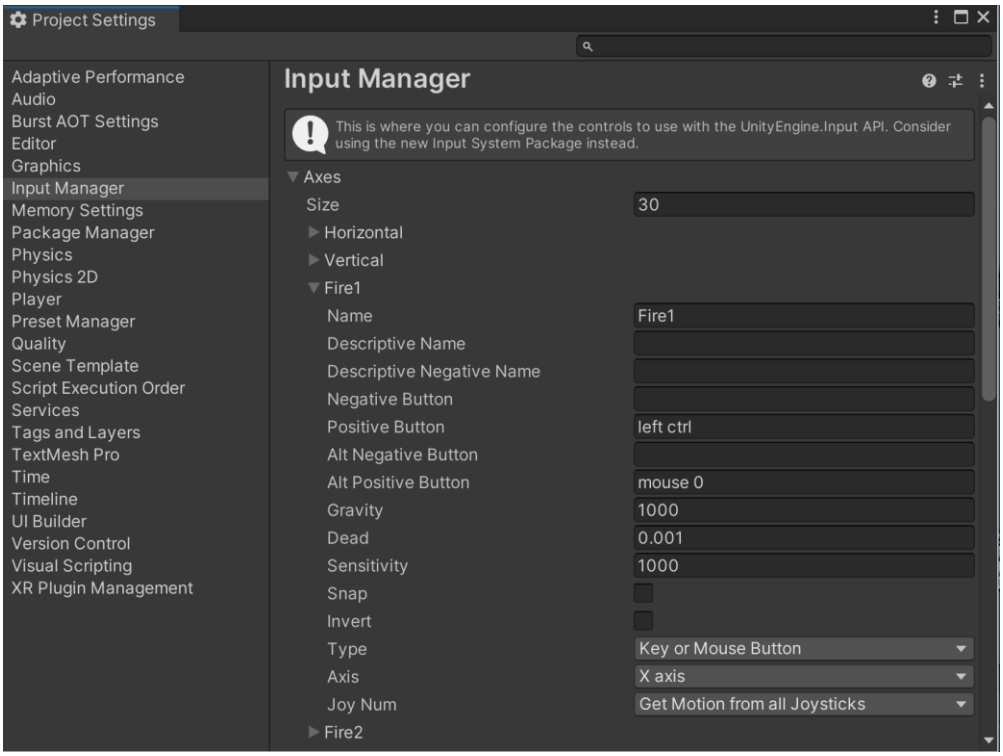


Рисунок 6.1 – Менеджер введення (Input Manager)

Менеджер введення використовує такі типи елементів керування:

- Клавіша – це будь-яка клавіша на фізичній клавіатурі, наприклад W, Shift або пробіл.
- Під кнопкою розуміється будь-яка кнопка на фізичному контролері (наприклад, геймпадах), наприклад, кнопка X на контролері Xbox One.
- Віртуальна вісь зіставляється з елементом керування, наприклад, кнопкою або клавішею. Коли користувач активує елемент керування, вісь набуває значення в діапазоні [–1..1]. Ви можете використовувати це значення у своїх скриптах.

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

Кожна вісь введення має такі властивості:

Властивості.	Функція
Name	Ім'я осі. Ви можете використовувати це для доступу до осі зі скриптів.
Descriptive Name, Descriptive Negative Name	Ці значення застаріли та не працюють. Раніше вони відображалися для користувача на екрані Rebind Controls під час запуску, але цей екран також застарів.
Negative Button, Positive Button	Елементи управління для переміщення осі в негативному та позитивному напрямку відповідно. Це можуть бути клавіші на клавіатурі, кнопки на джойстику або миші.
Alt Negative Button, Alt Positive Button	Альтернативні елементи управління для переміщення осі в негативному та позитивному напрямку відповідно.
Gravity	Швидкість в одиницях за секунду, з якою вісь падає в нейтральне положення за відсутності вхідних даних.
Dead	Як далеко користувач повинен перемістити аналоговий джойстик, перш ніж ваш додаток зареєструє рух. Під час виконання введення з усіх аналогових пристроїв, що потрапляють у цей діапазон, вважатиметься нульовим.
Sensitivity	Швидкість в одиницях за секунду, з якою вісь рухатиметься до цільового значення. Це лише для цифрових пристроїв.
Snap	Якщо увімкнене, значення осі буде скинуто до нуля при натисканні кнопки, що відповідає протилежному напрямку.
Type	Тип введення, який керує віссю. Виберіть із цих значень: - Клавіша чи кнопка миші - Рух миші - Вісь джойстика
Axis	Вісь підключеного пристрою, який керує цією віссю.
JoyNum	Підключений джойстик, який керує цією віссю. Ви можете вибрати конкретний джойстик або запросити введення з усіх джойстиків.

Значення осі можуть бути:

- Від –1 до 1 для джойстика та введення з клавіатури. Нейтральне положення для цих осей – 0. Деякі типи елементів керування, наприклад, кнопки на клавіатурі, не чутливі до інтенсивності введення, тому вони не можуть видавати значення, відмінні від –1, 0 або 1.
- Дельта миші (наскільки миша перемістилася протягом останнього кадру) для введення мишею. Значення для осей введення миші може бути більше 1 або менше –1, коли користувач швидко переміщає мишу.

Щоб додати віртуальну вісь, збільште число у полі Size. Це створює нову вісь унизу списку. Нова вісь копіює властивості попередньої осі у списку.

Щоб скопіювати віртуальну вісь, клацніть її правою кнопкою миші та виберіть Duplicate Array Element.

Щоб видалити віртуальну вісь, ви можете:

- Зменшіть число у полі Size. Це видалить останню вісь у списку.

– Натисніть правою кнопкою миші будь-яку вісь та виберіть Delete Array Element.
Примітка. Цю дію не можна скасувати.

Назви ключів відповідають наступним угодам про імена:

Key family	Угода про іменування
Letter keys	a, b, c…
Number keys	1, 2, 3…
Arrow keys	up, down, left, right
Numpad keys	[1], [2], [3], [+], [equals]…
Modifier keys	right shift, left shift, right ctrl, left ctrl, right alt, left alt, right cmd, left cmd
Special keys	backspace, tab, return, escape, space, delete, enter, insert, home, end, page up, page down
Function keys	f1, f2, f3…

Кнопки миші називаються mouse 0, mouse 1, mouse 2 тощо.

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

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

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

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

– Rigidbody2D
– Коллайдер (CircleCollider2D, BoxCollider2D)

Компонент Rigidbody 2D поміщає об'єкт під управління фізичного двигуна.

Фізичний двигун – система, яка імітує аспекти фізичних систем, щоб об'єкти могли правильно прискорюватися та піддаватися впливу зіткнень, гравітації та інших сил.

2D об'єкти можуть рухатися тільки в площині XY і можуть обертатися тільки навколо осі, перпендикулярної цій площині.

Зверніть увагу, що хоча 2D-об'єкти Rigidbody часто описуються як такі, що стикаються один з одним, стикаються саме 2D-об'єкти Collider, прикріплені до кожного з цих тіл. 2D-об'єкти Rigidbody не можуть стикатися без колайдерів.

У компоненті Rigidbody 2D вгорі є налаштування Тип тіла (Body type). Вибраний варіант впливає на інші параметри, доступні в компоненті.

Існує три варіанти типу тіла:

– Dynamic
– Kinematic
– Static

Кожен визначає загальну та фіксовану поведінку. Будь-який 2D-колайдер, прикріплений до 2D-об'єкта Rigidbody, успадковує Тип тіла 2D-об'єкта Rigidbody.

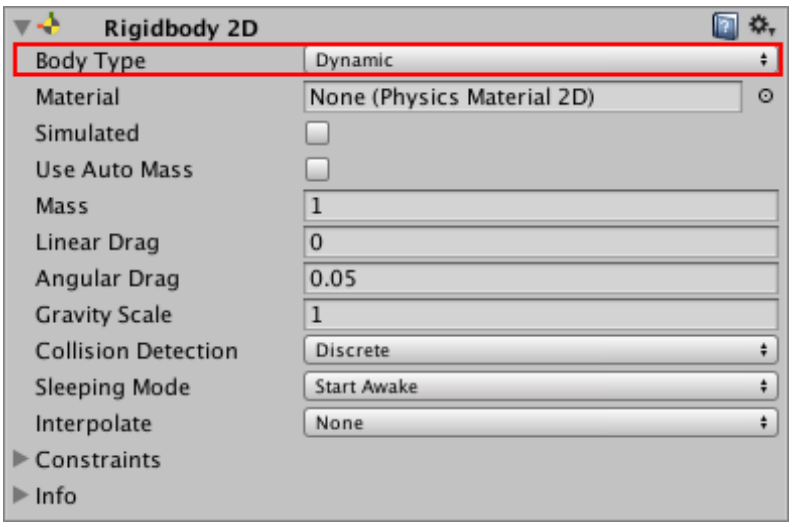


Рисунок 6.2 – Компонент Rigidbody 2D

Dynamic Rigidbody 2D призначений для переміщення під час моделювання. Він має повний набір доступних йому властивостей, таких як кінцева маса і лобовий опір, і на нього впливають гравітація та сили. Динамічне тіло стикається з будь-яким іншим типом тіла і є найбільш інтерактивним з типів тіла. Це тип тіла за замовчуванням для Rigidbody 2D, тому що це найпоширеніший тип тіла для речей, які мають рухатися.

Кінематичне тверде тіло (Kinematic Rigidbody 2D) рухається за рахунок своєї швидкості, але швидкість не впливають сили або гравітація. Kinematic Rigidbody 2D не стикається з іншими Kinematic Rigidbody 2D або Static Rigidbody 2D, він стикається тільки з Dynamic Rigidbody 2D.

Static Rigidbody 2D спроектований таким чином, щоби взагалі не рухатися при моделюванні. Якщо з ним щось стикається, статичне тверде тіло 2D поводиться як нерухомий об'єкт (якби він мав нескінченну масу). Статичне тіло стикається тільки з динамічним твердим тілом 2D. Зіткнення двох Static Rigidbody 2D не підтримується, оскільки вони не призначені для переміщення.

2D-колайдери визначають форму двомірного GameObject для цілей фізичного collisions. Колайдер, не обов'язково має таку саму форму, як і об'єкт. Грубе наближення часто буває більш ефективним та непомітним в ігровому процесі.

Будь-який компонент Collider 2D, доданий до того ж GameObject або дочірньому GameObject, неявно приєднується до цього Rigidbody 2D. Коли 2D-колайдер прикріплений до 2D-твердого тіла, він рухається разом із ним.

2D-коллайдер ніколи не слід переміщувати безпосередньо за допомогою Transform або будь-якого іншого зміщення коллайдера. Натомість слід переміщати Rigidbody 2D. Це забезпечує найкращу продуктивність та гарантує правильне зіткнення.

2. Рух персонажу

GetComponent<НазваКомпонента>(), дозволяє знайти на об'єкті відповідний компонент.

SetFloat() – метод класу Animator, який задає значення для параметра з ім'ям, що дорівнює першому аргументу виклику, і значення дорівнюватиме другому аргументу.

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

Код, який рухатиме персонажа по горизонталі:

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

Щоб персонаж стрибав, до нього треба прикладати силу за допомогою AddForce().

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

Щоб персонаж повертався обличчям у потрібну сторону, додайте до Update наступний код:

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

Цей код змінює масштаб по осі X так, щоб коли користувач натискає "вправо", масштаб дорівнює 1, а коли "вліво" - масштаб дорівнює -1. Таким чином, весь об'єкт дзеркально відображається вздовж осі X.

Створимо можливість персонажу підніматися сходами.

Додаймо до об’єкту, який складається зі спрайтів елементу сходів, наприклад, Box Collider 2D. Встановимо межі цього колайдера, щоб вони покривали всі сходи, а також щоб персонаж міг проходити повз них, встановимо колайдер сходів в режим IsTrigger = true.

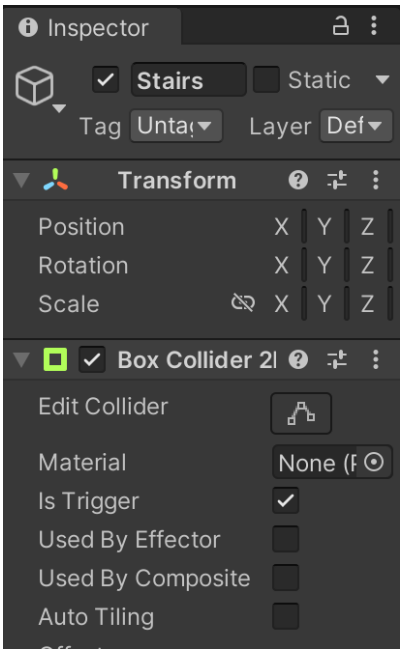


Рисунок 6.3 – Компонент Box Collider 2D

Нам потрібно, щоб коли лицар входить у зону тригера, у нього включався режим “на сходах”, а коли виходить із зони – вимикався.

Для цього є два стандартні методи MonoBehaviour: OnTriggerEnter2D та OnTriggerExit2D. Як аргумент у них автоматично передається посилання на колайдер, який увійшов/вийшов у/з зони тригера.

```
public class Stair : MonoBehaviour
{
    void OnTriggerEnter2D (Collider2D collider)
    {
    }

    void OnTriggerExit2D(Collider2D collider)
    {
    }
}
```

Щоб персонаж міг переміщатися вгору-вниз, коли знаходиться біля сходів, додаймо в метод Update в скрипті персонажу відповідний код:

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

Контрольні питання

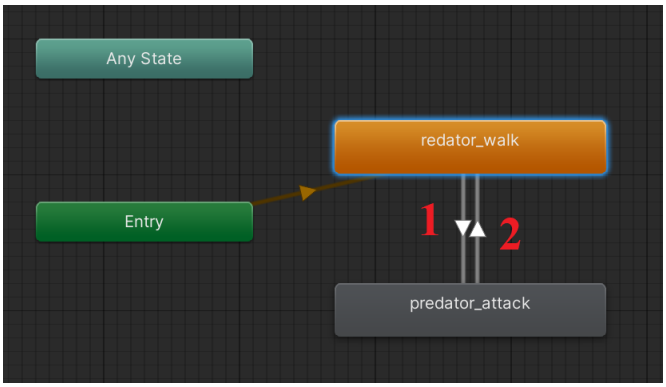
1. Для чого використовується Input Manager?
2. Що таке вісь в Input Manager?
3. Які властивості має кожна вісь?
4. Які компоненти потрібно додати до персонажу, щоб він міг керуватися клавішами?
5. Для чого використовуються методи OnTriggerEnter2D та OnTriggerExit2D?

Лекція 9. 2D гра. Додавання ворогів

1. Налаштування проекту та персонажу
2. Рух персонажу

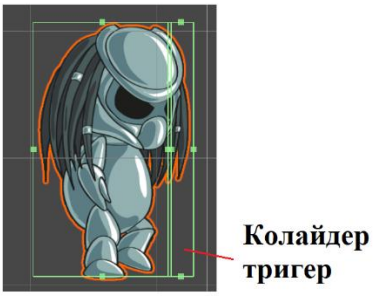
1. Управління ворогами

Додаймо до анімації ворогів анімацію атаки. Налаштуємо контролер анімації ворога, додавши новий стан – атака. Перехід на атаку будемо виконувати зі стану walk. Створимо в Animator змінну типу тригер – Attack.



Для переходу 1 – Has Exit Time галочка знята, для 2 – поставлена, перехід 1 відбувається при активації тригера Attack, перехід 2 – просто після завершення анімації атаки.

Додаймо до ворога ще один колайдер у режимі тригера. Цей колайдер буде фіксувати зіткнення з головним персонажем та показ анімації атаки.



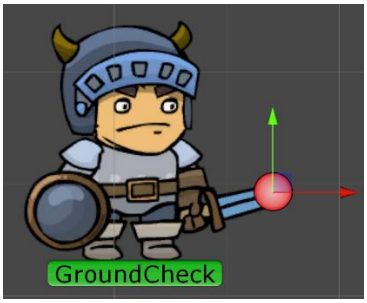
Додаймо до скрипту Enemy до методу OnTriggerEnter2D код, якщо в колайдер - тригер будуть потрапляти об'єкти і якщо це головний персонаж, то розпочати анімацію атаки.

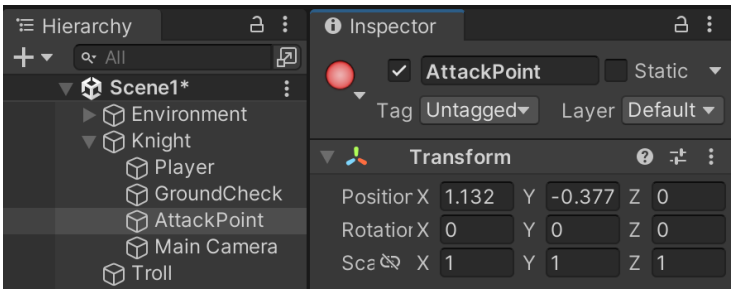
```
Knight knight =
    collider.gameObject.GetComponent();
if (knight != null) {
        animator.SetTrigger("Attack");
}
```

Щоб ворог атакував головного героя неодноразово, а доти, доки головний герой перед ним, змініть назву методу OnTriggerEnter2D на OnTriggerStay2D.

Тепер треба зробити так, щоб головний герой завдав шкоди, а не просто махав зброєю.

Додамо до об'єкта Knight порожній дочірній GameObject – це буде точка, в якій наноситься шкода. Встановимо для цього об'єкта іконку, перейменуємо AttackPoint і помістимо її на кінець зброї.





У скрипті Hero оголосимо змінну attackPoint типу Transform для цього об'єкта, виведемо її в інспектор і перетягнемо об'єкт у поле, щоб зберегти посилання на цей об'єкт.

Також оголосимо змінну attackRange типу float, і задаємо її значення інспекторі, наприклад, рівним 0.5. Це буде радіус навколо об'єкту AttackPoint, в якому будуть вражати ворожі колайдери.

Тепер треба навчити ворога та головного персонажу завдавати один одному ушкоджень. Спробуймо реалізувати це за допомогою інтерфейсів.

Інтерфейс – це список методів та властивостей, які мають бути реалізовані у класі, який успадковує цей інтерфейс (або реалізовує). Якщо ми знаємо, що якийсь клас реалізує певний інтерфейс, то ми можемо бути впевнені, що в цьому класі є набір методів та властивостей, які є обов'язковими для даного інтерфейсу.

Кожен клас може бути успадкований тільки від одного іншого класу, але може при цьому мати скільки завгодно успадкованих інтерфейсів

Створимо новий скрипт IDestructable, і видалимо з нього весь код і додаймо наступний код:

```
interface IDestructable
{
float Health { get; set; }
void Hit(float damage);
void Die();
}
```

Тепер будь-який клас, який реалізує цей інтерфейс, повинен мати публічну властивість Health типу float, публічний метод Hit, який повертає void, приймає параметр типу float, і публічний метод Die, який так само повертає void, і не приймає параметрів.

Лекція 10. 2D гра. Додавання оточення

1. Додавання оточення
2. Створення підвісного мосту
3. Система частинок

1. Додавання оточення

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

Додаємо на сцену елементи сцени. Наприклад:

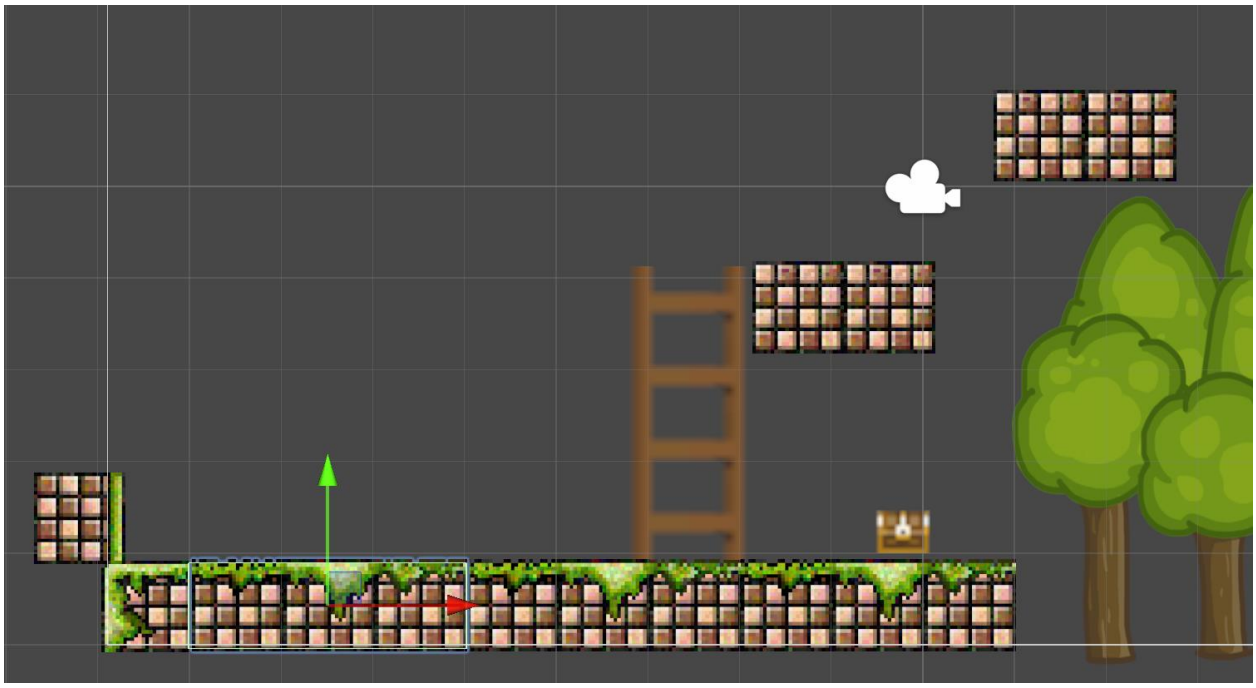
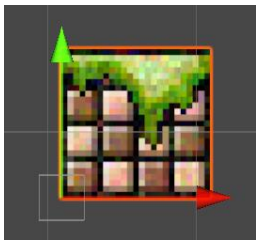


Рисунок 10.1 – Приклад сцени

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



Бувають такі ситуації у грі, коли персонажу треба забратися кудись, наприклад сходами. Розглянемо яку функціональність потрібно додати персонажу, щоб він міг це робити.

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

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

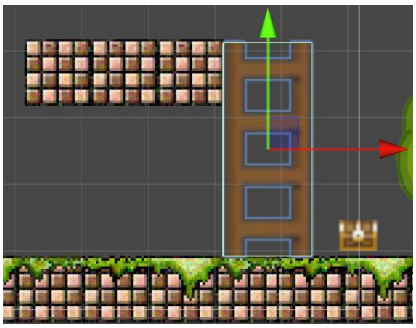


Рисунок 10.2 – Приклад використання сходів

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

```
private bool onStair = false;

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

        onStair = value;
    }

}
```

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

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

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

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

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

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

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

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

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

2. Створення підвісного міста

Додаймо до сцени підвісний міст.

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

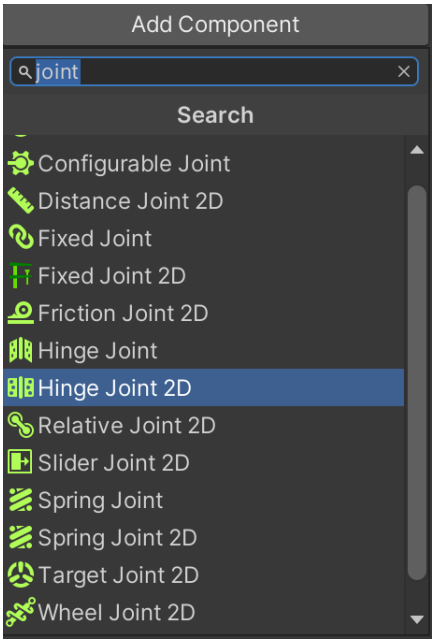
До спрайту ділянки мосту додати Rigidbody2D та BoxCollider2D.

2D джоінти являють собою звичайні фізичні об’єкти, яким можна придавати силу, переміщати, кидати, зтиснути тощо, в загальному – робити все те, що ми зазвичай робимо з фізичними об’єктами в Unity.

Особливість же джоінтів в тому, що з їх допомогою можна створити зв'язок з іншими фізичними об'єктами на сцені.

В цьому випадку всі дії, які ми здійснюємо над одним об’єктом, будуть також впливати на інші – пов’язані з ним об’єкти.

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



Hinge Joint 2D (або шарнірна петля) – це компонент, що контролюється фізикою Rigidbody, який дозволяє приєднати Sprite до точки в просторі, навколо якої цей об'єкт може обертатися. Обертання може бути пасивним (наприклад, у відповідь на зіткнення) або активним, коли двигун приводить його в рух.

Цей компонент корисний, наприклад, для створення дверних петель, коліс, маятників та інших об'єктів, які мають обертатися навколо певної точки.

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

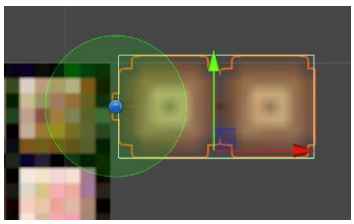
Налаштування компонента:

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

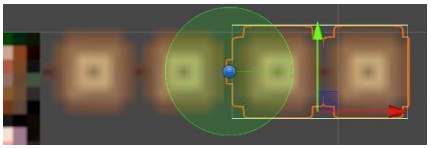
Hinge Joint 2D дозволяє застосувати силу для вдосконалення повороту об'єкта, а також ввести обмеження на кут повороту. Для цього в компоненті є поле Motor, де вказані дві змінні – Motor Speed та Maximum Motor Force, для швидкості обертання та максимальної сили, що застосовуються для вдосконалення обертання.

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

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



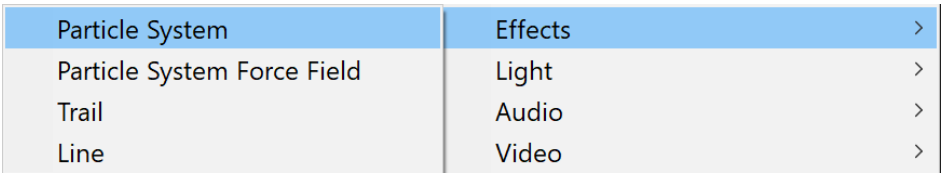
Створимо копію спрайту ділянки мосту зі всіма компонентами. В копії в полі Connected Rigid Body додаймо посилання на Rigidbody2D попередньої ділянки мосту. Так продовжуємо поки не будемо мати необхідну довжину мосту.



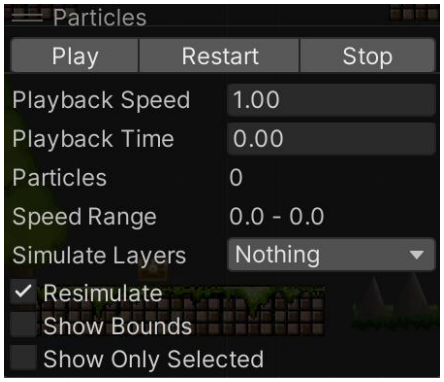
В останній спрайт ділянки мосту додаймо ще один компонент Hinge Joint 2D та в полі Connected Rigid Body нічого не додаємо.

3. Система частинок

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



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



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

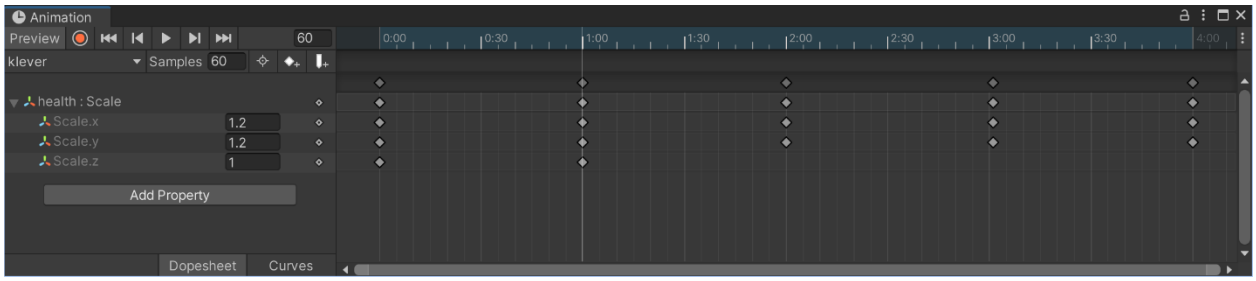
4. Пульсуюча анімація предмету

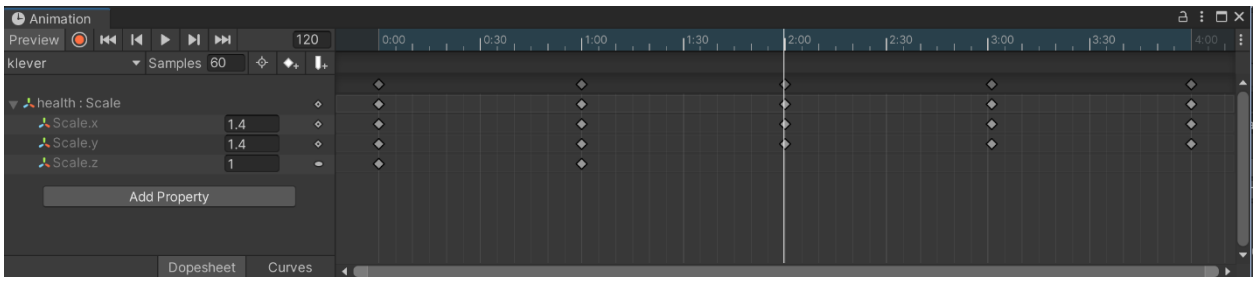
Обираємо об’єкт який бажаємо зробити пульсуючим. Додаємо до нього компонент Collider2D. Викликаємо вікно Animation для цього об’єкту.

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

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

Встановлюємо моменти часу на timeline. В каждому моменту часу змінюємо обрані властивості.





Контрольні питання

1. Що таке оточення в грі?
2. Що таке Tile?
3. Яку функціональність потрібно додати персонажу, щоб він міг ходити сходами?
4. Який компоненти потрібно додати об’єкті, щоб він виконував функції підвісного мосту?
5. Що таке система частинок? Для чого вони використовуються?
6. Як створити пульсуючу анімацію для об’єкта?

Лекція 11. Формати для зберігання даних. Криптографічний захист

1. Структура та правила створення XML
2. Структура та правила створення JSON
3. Стандарти кібербезпеки.
4. Методи шифрування даних

1. Структура та правила створення XML

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 2.1 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.

2. Структура та правила створення JSON

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.

3. Стандарти кібербезпеки

Якщо розглядати гру як комп'ютерну систему, то існують стандарти інформаційної безпеки.

Стандарт – документ, у якому з метою добровільного багаторазового використання встановлюються характеристики продукції, правила здійснення та характеристики процесів

виробництва, експлуатації, зберігання, перевезення, реалізації та утилізації, виконання робіт чи надання послуг.

Стандартизація – діяльність із встановлення правил і показників з метою їх добровільного багаторазового використання, спрямовану досягнення впорядкованості у сферах виробництва та обігу продукції і на підвищення конкурентоспроможності продукції, робіт чи послуг.

Основна роль у розвитку інформаційного суспільства належить міжнародним стандартам, створюваним на основі шести принципів, визначених Світовою організацією торгівлі (СОТ): відкритість, прозорість, неупередженість і дотримання консенсусу, ефективність і доцільність, узгодженість і націленість на розвиток.

В даний час переважна більшість ІС всіх класів та призначень будуються на основі технології відкритих систем. Її суть полягає у використанні стандартних інтерфейсів між різнорідними апаратними та програмними компонентами систем.

Технологія відкритих систем є основою створення інфраструктури всіх рівнів – від підприємства та галузі до національної інформаційної інфраструктури. Крім того, вона забезпечує інтеграцію зі світовим інформаційним простором і тим самим зі світовою економікою.

Впровадження принципів відкритих систем на всіх етапах життєвого циклу проектування ІС базується на стандартизації інформаційних технологій, що є інтеграційним механізмом та потужним засобом управління процесами розвитку інформатизації.

Стандарти інформаційної безпеки (СІБ). Головне завдання СІБ – створення основи для взаємодії між виробниками, споживачами та експертами з кваліфікації продуктів інформаційних технологій.

Починаючи з початку 80-х років, були створені десятки міжнародних і національних стандартів у галузі інформаційної безпеки, які в певній мірі доповнюють один одного.

Найбільш відомі стандарти за хронологією їх створення:

- Критерій оцінки надійності комп'ютерних систем «Помаранчева книга» (США);
- Гармонізовані критерії європейських країн;
- Рекомендації Х.800;
- Німецький стандарт BSI;
- Британський стандарт BS 7799;
- Стандарт ISO 17799;
- Стандарт «Загальні критерії» ISO 15408;
- Стандарт COBIT

Ці стандарти можна розділити на два види:

- Оціночні стандарти, спрямовані на класифікацію інформаційних систем і засобів захисту за вимогами безпеки;
- Технічні специфікації, які регламентують різні аспекти реалізації засобів захисту.

Важливо відзначити, що між цими видами нормативних документів існує логічний взаємозв'язок. Оціночні стандарти виділяють найважливіші, з точки зору ІБ, аспекти ІС, граючи роль архітектурних специфікацій. Інші технічні специфікації визначають, як будувати ІС запропонованої архітектури.

У 1989 році Міністерство торгівлі та промисловості Великобританії (Department of Trade and Industry, DTI) опублікувало перший стандарт у галузі інформаційної безпеки, який називався PD 0003 «Практичні правила управління ІБ».

Він являв собою перелік засобів управління безпекою, які в той час вважалися адекватними, нормальними і хорошими, що застосовуються як до технологій, так і серед того часу. Документ DTI опубліковано як керівний документ британської системи стандартів (British Standard, BS).

1995 року Британський інститут стандартів (British Standards Institution, BSI) прийняв національний стандарт BS 7799-1 «Практичні правила управління ІБ».

Він описував 10 областей та 127 механізмів контролю, необхідних для побудови системи управління інформаційною безпекою СУІБ (Information Security Management System, ISMS), визначених на основі найкращих прикладів зі світової практики. Цей стандарт став основою всіх міжнародних стандартів СУІБ.

У 7799-1 «Частині 1: Практичні рекомендації» (1995 р.) визначаються і розглядаються наступні аспекти ІБ:

- Політика безпеки.
- Організація захисту.
- Класифікація та управління інформаційними ресурсами.
- Управління персоналом.
- Фізична безпека.
- Адміністрування комп'ютерних систем та мереж.
- Управління доступом до систем.
- Розробка та супровід систем.
- Планування безперебійної роботи організації.
- Перевірка системи на відповідність вимогам ІБ.

У 1998 році з'явилася друга частина цього стандарту – BS 7799-2 «СУІБ – Специфікація та посібник із застосування», яка визначила загальну модель побудови СУІБ та набір обов'язкових вимог, на відповідність яким має проводитися сертифікація.

У «Частина 2: Специфікації системи» (1998 р.) розглядає ці ж аспекти з точки зору сертифікації інформаційної системи на відповідність вимогам стандарту. Вона визначає можливі функціональні специфікації корпоративних систем управління інформаційною безпекою з точки зору їх перевірки на відповідність вимогам першої частини даного стандарту. Відповідно до положень цього стандарту також регламентується процедура аудиту інформаційних корпоративних систем.

З появою другої частини BS 7799, яка визначила, що повинна представляти собою СУІБ, почався активний розвиток системи сертифікації в галузі управління безпекою.

Наприкінці 1999 року експерти Міжнародної організації зі стандартизації (International Organization for Standardization, ISO) та Міжнародної електротехнічної комісії (International Electrotechnical Commission, IEC) дійшли висновку, що в рамках існуючих стандартів відсутній спеціалізований стандарт управління ІБ.

Відповідно, було ухвалено рішення не займатися розробкою нового стандарту, а за погодженням із британським інститутом стандартів, взявши за базу BS 7799-1, прийняти відповідний міжнародний стандарт ISO/IEC.

Наприкінці 1999 року обидві частини BS 7799 були переглянуті та гармонізовані з міжнародними стандартами систем управління якістю ISO/IEC 9001 та екологією ISO/IEC 14001, а через рік без змін BS 7799-1 був прийнятий як міжнародний стандарт ISO/IEC 17709: «Інформаційні технології – Практичні правила управління ІБ».

У складі ISO/IEC за розробку сімейства міжнародних стандартів з управління ІБ відповідає підкомітет №27, тому було прийнято схему нумерації даного сімейства стандартів з використанням серії послідовних номерів, починаючи з 27000.

У 2005 році підкомітет SC 27 «Методи захисту інформаційних технологій» Об'єднаного технічного комітету JTC 1 «Інформаційні технології» ISO/IEC розробив сертифікаційний стандарт ISO/IEC 27001 «Інформаційні технології – Методи та засоби забезпечення безпеки – СУІБ – Вимоги», прийшов на заміну BS 7799-2.

В Україні ISO/IEC 27001:2005 як національний стандарт введено в дію лише з 1 липня 2012 року.

У 2005 році на основі ISO/IEC 17799:2000 був розроблений ISO/IEC 27002:2005 «Інформаційні технології - Методи та засоби забезпечення безпеки – Зведення норм та правил управління ІБ».

На початку 2006 року було прийнято новий британський національний стандарт у галузі управління ризиками ІБ BS 7799-3, який у 2008 році отримав статус міжнародного

стандарту ISO/IEC 27005 «Інформаційні технології – Методи та засоби забезпечення безпеки – Управління ризиками ІБ».

У 2004 році Британським інститутом стандартів було опубліковано стандарт ISO/IEC TR 18044 «Інформаційна технологія – Методи забезпечення безпеки – Управління інцидентами ІБ».

У 2011 році на його базі було розроблено стандарт ISO/IEC 27035 «Інформаційна технологія – Методи забезпечення безпеки – Управління інцидентами ІБ».

У 2009 році було прийнято стандарт ISO/IEC 27000 «IT – СУІБ – Загальний огляд та термінологія», останнє оновлення якого відбулося 14 січня 2014 року. Він надає огляд систем управління ІБ та визначає відповідні терміни.

25 вересня 2013 року були опубліковані нові версії стандартів ISO/IEC 27001 та 27002. З цього моменту стандарти серії ISO/IEC 27k (управління ІБ) повністю інтегровані зі стандартами серії ISO/IEC 20k (управління IT-сервісами).

Вся термінологія ISO/IEC 27001 перенесена до ISO/IEC 27000, який визначає загальний термінологічний апарат для всього сімейства стандартів ISO/IEC 27k.

4. Методи шифрування даних

Криптографія – це наука про те, як забезпечити таємність повідомлення.

Криптоаналіз – це наука про те, як розкрити шифроване повідомлення, тобто як отримати відкритий текст не знаючи ключа.

Основною характеристикою шифру є *криптостійкість*, що визначає його стійкість до розкриття методами криптоаналізу. Зазвичай ця характеристика визначається інтервалом часу, необхідним розкриття шифру.

Стійкість конкретного шифру оцінюється шляхом можливих спроб розкриття і залежить від кваліфікації криптоаналітиків, що атакують шифр. Процедуру атаки на шифр для з'ясування його стійкості називають *перевіркою стійкості*.

Існують такі шифри:

- шифри перестановок;
- шифри заміни;
- шифри гамування;
- шифри, засновані на аналітичних перетвореннях даних, що шифруються.

Основною схемою класифікації всіх криптоалгоритмів є така:

- Тайнопис
- Криптографія з ключем:
  - Симетричні криптоалгоритми
  - Асиметричні криптоалгоритми

Криптографія з ключем – алгоритм впливу на дані, що передаються, відомий усім стороннім особам, але він залежить від деякого параметра - "ключа", яким володіють тільки відправник і одержувач

Симетричні криптоалгоритми – для шифрування та дешифрування повідомлення використовується один і той самий блок інформації (ключ).

Асиметричні криптоалгоритми – для шифрування повідомлення використовується один ("відкритий") ключ, відомий усім охочим, а для дешифрування - інший ("закритий"), що існує тільки в отримувача.

Залежно від розміру блоку інформації криптоалгоритми поділяються на:

- Потокові шифри
- Блокові шифри

Потокові шифри. Одиницею кодування є один біт. Результат кодування не залежить від попереднього вхідного потоку. Схема застосовується у системах передачі потоків інформації, тобто у випадках, коли передача інформації починається і закінчується у довільні моменти часу і може випадково перериватися.

Блокові шифри. Одиницею кодування є блок із кількох байтів. Результат кодування залежить від усіх вихідних байт цього блоку. Схема застосовується при пакетній передачі інформації та кодування файлів.

Розглянемо детальніше симетричні блокові шифри.

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

Таким чином, схему роботу блокового шифру можна описати функціями

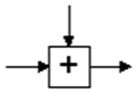
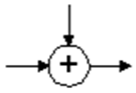
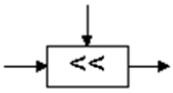
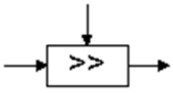
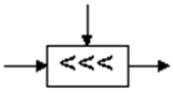
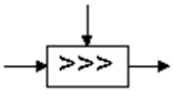
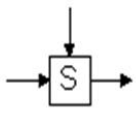
Z = EnCrypt(X, Key)

та

X = DeCrypt(Z, Key)

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

Над числами блоковим криптоалгоритмом виробляються за певною схемою наступні дії:

	Додавання	X'=X+V
	Виключне АБО	X'=X XOR V
	Арифметичний зсув вліво	X'=X SHL V
	Арифметичний зсув праворуч	X'=X SHR V
	Циклічний зсув ліворуч	X'=X ROL V
	Циклічний зсув праворуч	X'=X ROR V
	S-box (англ. substitute)	X'=Table[X,V]

Як параметр V для будь-якого з цих перетворень може використовуватися:

– фіксоване число (наприклад, X'=X+125)
– число, що отримується з ключа (наприклад, X'=X+F(Key))
– число, що отримується з незалежної частини блоку (наприклад, X2'=X2+F(X1))

Останній варіант використовують у схемі, яка називається *мережею Фейштеля* (нім. Feistel).

У 80-х роках у США було прийнято стандарт симетричного криптоалгоритму для внутрішнього застосування DES (Data Encryption Standard), який набув досить широкого поширення свого часу. Однак, на даний момент цей стандарт повністю неприйнятний для використання з двох причин.

Все це спонукало Американський інститут стандартизації NIST – National Institute of Standards & Technology на оголошення у 1997 році конкурсу на новий стандарт симетричного криптоалгоритму. Цього разу вже було враховано основні промахи шифру-попередника, а до розробки були підключені найбільші центри криптології з усього світу.

Підсумком конкурсу мав стати «незасекречений, відкритий усім, хто цікавиться, алгоритм шифрування, здатний захистити секретну державну інформацію в наступному столітті».

Переможець цього змагання, названого AES – Advanced Encryption Standard, стане де-факто світовим криптостандартом на найближчі 10-20 років.

Вимоги до кандидатів на AES в 1998 році:

- алгоритм має бути симетричним;
- алгоритм має бути блоковим шифром;
- алгоритм повинен мати довжину блоку 128 біт, і підтримувати три довжини ключа: 128, 192 та 256 біт.

На першому етапі до оргкомітету змагання надійшло 15 заявок із різних куточків світу. Протягом 2 років фахівці комітету, досліджуючи самостійно, та вивчаючи публікації інших дослідників, обрали 5 найкращих представників, які пройшли у «фінал» змагання. Переможцем конкурсу став шифр Rijndael.

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

Оперує блоками даних розміром 128 і довжиною ключа 128, 192 або 256 біт – назва стандарту відповідно «AES-128», «AES-192» та «AES-256». Алгоритм представляє кожен блок даних, що кодуються у вигляді двовимірного масиву байт розміром 4x4, 4x6 або 4x8 в залежності від встановленої довжини блоку.

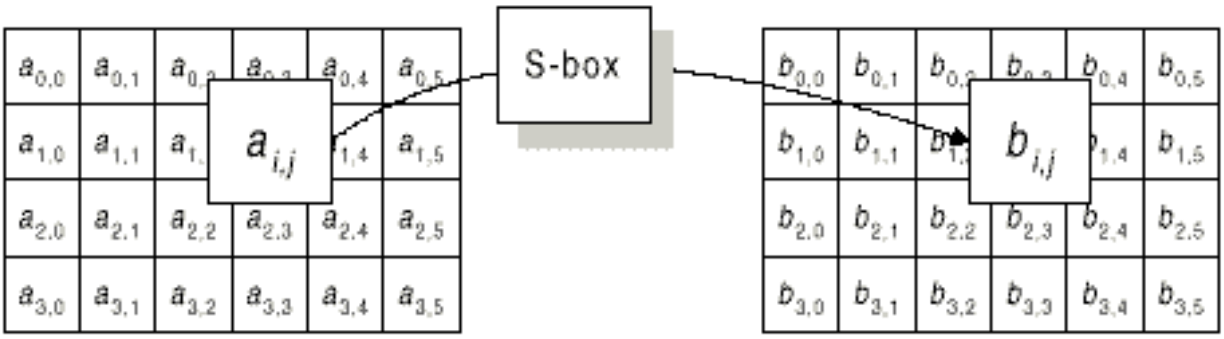
Далі на відповідних етапах перетворення виробляються або над незалежними стовпцями, або над незалежними рядками, або взагалі окремими байтами в таблиці. Структура та послідовність операцій дозволяють виконувати цей алгоритм ефективно як на 8-бітних так і на 32-бітових процесорах.

У структурі алгоритму закладено можливість паралельного виконання деяких операцій, що у багатопроцесорних робочих станціях може підняти швидкість шифрування вчетверо.

Алгоритм складається з деякої кількості раундів (від 10 до 14 – це залежить від розміру блоку та довжини ключа), у яких послідовно виконуються такі операції:

В алгоритмі виконуються такі операції:

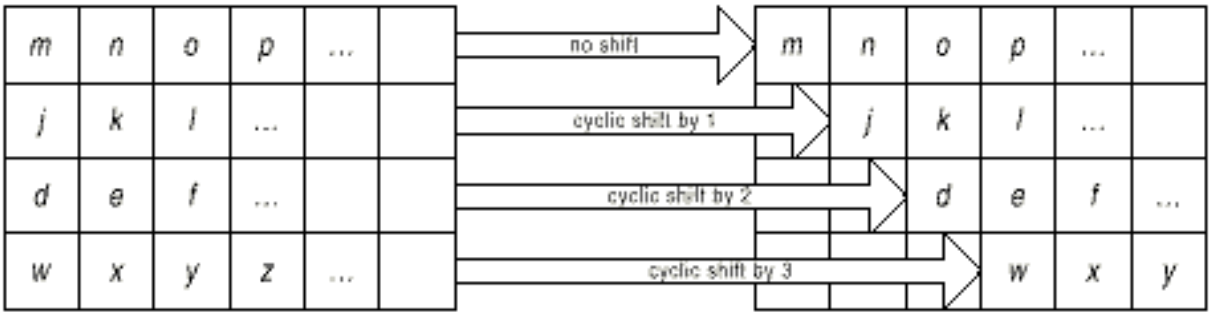
- ByteSub – таблична підстановка 8x8 біт



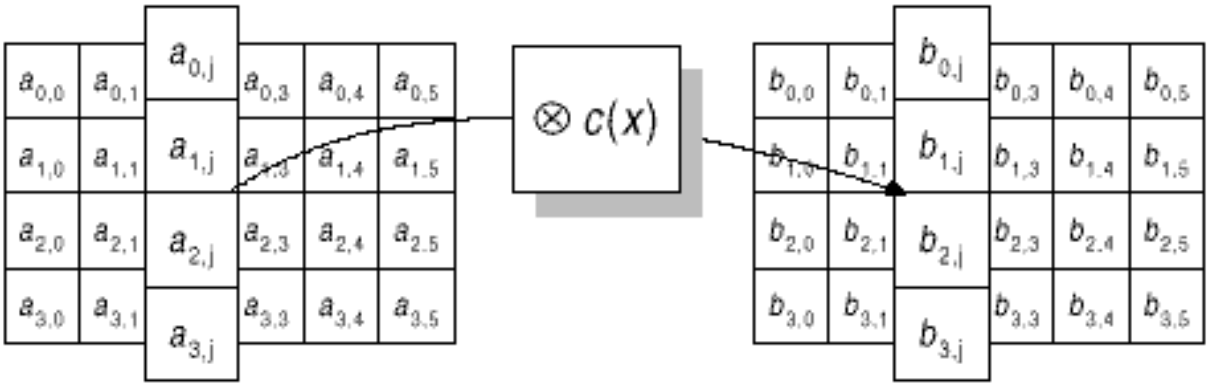
Таблиця S-Box заміни байт:

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

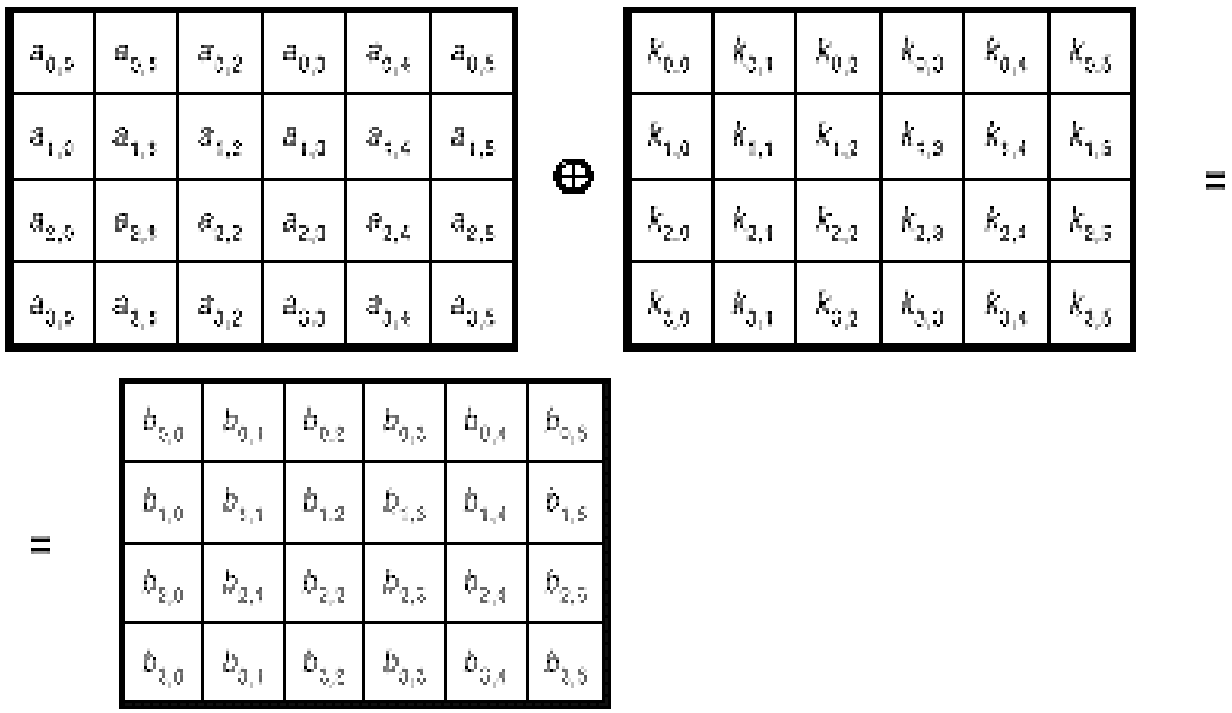
– ShiftRow – зсув рядків у двовимірному масиві на різні зсуви



– MixColumn – математичне перетворення, що перемішує дані всередині стовпця.



– AddRoundKey – додавання матеріалу ключа операцією XOR



Раундові ключі виходять із ключа шифрування за допомогою алгоритму вироблення ключів. Він містить два компоненти: розширення ключа (Key Expansion) та вибір раундового ключа (Round Key Selection).

Функції розшифрування.

– InvSubBytes – зворотна SubBytes заміна байтів – побайтова нелінійна підстановка в S-блоках з використанням фіксованої таблиці замін. Таблиця S-Box інверсної заміни байт:

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

– InvShiftRows – зворотний зсув рядків ShiftRows – циклічний зсув рядків масиву State на різну кількість байт;

– InvMixColumns – відновлення значень стовпців – множення стовпців стану.

Контрольні питання

1. Яка структура XML та JSON?
2. Чим відрізняється структура XML та JSON?
3. Що таке стандарт?
4. Для чого використовуються стандарти?
5. Що таке криптографія?
6. Що таке шифрування? Що потрібно для виконання шифрування?

Рекомендована література

Основна

1. Lanzinger F. 2D Game Development with Unity: CRC Press, 2020. – 444 p. URL: https://www.google.com.ua/books/edition/2D_Game_Development_with_Unity/hjwDEAAAQBAJ?hl=ru&gbpv=0

2. Felicia P. The Ultimate Guide to 2D Games with Unity: Amazon Digital Services LLC – Kdp, 2020. – 488 p. URL: https://www.google.com.ua/books/edition/The_Ultimate_Guide_to_2D_games_with_Unit/9AYBEAAAQBAJ?hl=ru&gbpv=0

3. Halpern J. Developing 2D Games with Unity. Independent Game Programming with C#: Apress, 2018. – 383 p. URL: https://www.google.com.ua/books/edition/Developing_2D_Games_with_Unity/loF8DwAAQBAJ?hl=ru&gbpv=0

4. Murray Jeff W. 2D Unity. Your First Game from Start to Finish: No Starch Press, 2016. – 312 p. URL: https://www.google.com.ua/books/edition/2D_Unity/0-8cswEACAAJ?hl=ru

Допоміжна

5. Sapio F. Unity 2D Game Development Blueprints / F. Sapio, A. Saher : Packt Publishing, Limited, 2016. – 252 p. URL: https://www.google.com.ua/books/edition/Unity_2D_Game_Development_Blueprints/KMzpjwEACAAJ?hl=ru

6. 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

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

Інформаційні ресурси

9. Офіційний сайт Unity [Електронний ресурс] URL: https://unity.com/ (Дата звернення: 01.09.2024)

10. Про прийняття національних стандартів, зміни до національного стандарту та скасування національних стандартів [Електронний ресурс] URL: https://zakon.rada.gov.ua/rada/show/v0210774-23#Text (Дата звернення: 01.09.2024)