СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра інформаційних технологій та програмування

ПОЯСНЮВАЛЬНА ЗАПИСКА

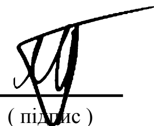
до кваліфікаційної випускної роботи

освітній ступінь бакалавр

спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)

спеціалізація „Інженерія програмного забезпечення”

на тему „Вдосконалення та оптимізація алгоритмів випадкової генерації для створення гри жанру rogue-lite”

Виконав: студент групи ІПЗ-20д  М.П, Тагієв
(підпис) (ініціали і прізвище)

Керівник Д.В. Ратов
(підпис) (ініціали і прізвище)

Завідувач кафедри О.І. Захожай
(підпис) (ініціали і прізвище)

Рецензент__________________

Київ– 2024

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра інформаційних технологій та програмування

Освітній ступінь бакалавр

спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)

спеціалізація „Інженерія програмного забезпечення”
(назва спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри
____________ Захожай О.І.
“____” ____________2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ ВИПУСКНУ РОБОТУ СТУДЕНТУ

Тагієв Мекан Первіз огли
(прізвище, ім’я, по батькові)

1. Тема роботи Вдосконалення та оптимізація алгоритмів випадкової генерації для створення гри жанру rogue-lite

Керівник роботи Ратов Деніс Валентинович Доц., к.т.н. ,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджений наказом університету від “___”__________20__року №___

2. Строк подання студентом роботи 20 травня 2024 р.

3. Вихідні дані до роботи Об'єктом даної роботи є процес огляду та розробки гри жанру rogue-lite

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вступ. Аналітичний огляд, з висвітленням наступних питань:що собою являють сучасні комп'ютерні ігри та їх класифікація, основні етапи розробки ігор. розгляд додатків-аналогів. Основна частина, в якої розыбрали: вібір додатків та інструментів для реалізації,проектування та реалізація проекту. Висновки. Перелік використаних джерел. код готового проекту

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслеників)

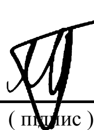
6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 30 березеня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання кваліфікаційної випускної роботи	Строк виконання етапів	Примітка
1	Одержання завдання на виконання роботи	30.03.20	
2	Укладання і погодження з керівником плану і етапів виконання роботи	06.04.20	
3	Узагальнення даних літературних джерел, укладання розділу «Аналіз предметної області»	13.04.20	
4	Аналіз шляхів виконання завдання. Вибір і погодження керівником оптимального шляху	20.04.20	
5	Укладання та тестування програмного продукту	27.04.20	
6	Укладання, оформлення та погодження пояснювальної записки з керівником	04.05.20	
7	Здача готової пояснювальної записки на кафедру	12.05.20	
8	Укладання доповіді і презентації	30.05.20	



Студент ________ Тагієв М.П.
(підпис) (ініціали і прізвище)

Керівник роботи ________ Ратов Д.В.
(підпис) (ініціали і прізвище)

ЛИСТ ПОГОДЖЕННЯ І ОЦІНЮВАННЯ

дипломної роботи студента гр. ІПЗ-20д Тагієва М.П.

Науковий керівник

Доцент, д.т.н.________________________________ Ратов Д.В.

Оцінка наукового керівника: ______________________

Рецензент ____

ПІБ, місто роботи, посада

Оцінка рецензента: __________________________________

Кінцева оцінка за результатами захисту:

____

Голова ЕК _________________ Меняйленко О.С.

підпис

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи на тему: «Вдосконалення та оптимізація алгоритмів випадкової генерації для створення гри жанру rogue-lite» містить: 65 основних сторінок, 63 додаткових,33 рисунки, 12 інформаційних джерел,2 таблиці,1 блок-схема.

Об’єкт дослідження – гра в жанрі 2D rogue-lite.

Предмет дослідження – технології розробки 2D ігор.

Мета кваліфікаційної роботи – розробити 2D гру у жанрі rogue-lite з використанням визначених у ході дослідження найефективніших інструментів та програмного забезпечення.

Методи дослідження – аналіз, порівняння, обробка літературних джерел та проектування.

Розробка прототипу гри є важливим етапом у створенні комп'ютерних ігор. Прототип надає розробникам можливість перевірити основні механіки, оцінити ігровий процес і внести необхідні зміни перед початком повномасштабної розробки. У даній роботі розглядається прототип 2D rogue-lite гри, його значення для подальшого розвитку проекту та використані інструменти й ресурс

ЗМІСТ

ВСТУП 2
1 КЛАСИФІКАЦІЙНЕ ДОСЛІДЖЕННЯ ТА АНАЛІЗ ІСНУЮЧИХ РОЗРОБОК КОМП'ЮТЕРНИХ ІГОР 3
1.1 Аналіз та загальна характеристика комп’ютерних ігор 3
1.2 Огляд жанру та аналіз існуючих розробок 7
1.2.1 Crypt of the NecroDancer 9
1.2.2 Enter the Gungeon 10
1.2.3 Slay the Spire 11
1.2.4 Barony 13
1.2.5 Порівняння різних ігор 14
1.3 Потенціал розвитку жанру 16
ВИСНОВОК ДО РОЗДІЛУ 1 21
2 РОЗРОБКА ПРОЕКТУ 22
2.1 Вибір середовища для розробки проекту 22
2.1.1 Godot Engine 24
2.1.2 LibGDX 29
2.1.3 Unity 33
2.2 Алгоритм реалізаці проекту 38
2.3 Мета проекту 41
2.4 Функціонал проекту 41
ВИСНОВОК ДО РОЗДІЛУ 2 43
3 РЕАЛІЗАЦІЯ ПРОЄКТУ 44
3.1 Графіка проекту 44
3.1.1 Спрайт меню 46
3.1.2 Спрайти головного героя 46
3.1.3 Спрайти ворогів 47
3.1.4 Спрайти інтерфейсу 47
3.1.5 Спрайти їжі 48
3.2 Анімації 49
3.3 Аудіофайли 50
3.4 Префаби 52
3.5 Меню гри 54
3.6 Рух об'єктів 56
3.7 Механіка смерті та життя 58
3.8 Вороги 60
3.9 Генерація рівня 60
3.10 Блок-схема алгоритму гри 62
ВИСНОВОК ДО РОЗДІЛУ 3 64
ВИСНОВОК 66
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 67
ДОДАТКИ 68

ВСТУП

В ігровій індустрії за останні роки спостерігається стрімкий розвиток, що в основному обумовлено швидкими темпами технологічного прогресу та поширенням Інтернету. Ігрові продукти стають все більш доступними та популярними, привертаючи увагу людей будь-якого віку. Важливо також відзначити, що сучасні комп'ютерні ігри відіграють важливу роль не лише у розважанні гравців, але й у навчанні та симуляціях, що збільшує попит на їх розробку та вдосконалення.

Жанр Roguelike відзначається особливим підходом до геймплею, який базується на випадковій генерації рівнів та персонажів. Це створює безліч варіантів проходження гри та надає їй непередбачуваність, що привертає багатьох гравців. Однак для успішної розробки ігор цього жанру потрібні не лише креативність та графічні навички, але й глибокі технічні знання в галузі програмування.

У даному контексті важливим аспектом є розуміння основних механік гри та її мети. Лише з чітким уявленням про геймплей та очікувані результати можна розробити ефективні алгоритми генерації рівнів, які забезпечать цікавий та викликаючий ігровий досвід для гравців.

1 КЛАСИФІКАЦІЙНЕ ДОСЛІДЖЕННЯ ТА АНАЛІЗ ІСНУЮЧИХ РОЗРОБОК КОМП'ЮТЕРНИХ ІГОР

1.1 Аналіз та загальна характеристика комп'ютерних ігор

Комп'ютерна гра - це програма, яка організовує хід гри через взаємодію з гравцем або функціонує самостійно як партнер по грі. В процесі гри за допомогою спеціальних алгоритмів моделюється взаємодія між ігровими персонажами та користувачами у віртуальному просторі. Гравець спостерігає за грою на екрані монітора та взаємодіє з нею за допомогою пристроїв управління (клавіатури, миші, джойстика та інших). Комп'ютерні ігри можуть бути джерелом натхнення, а не лише фільми або книги. Також, ігри можуть стати основою для створення книг, коміксів, фільмів та інших творів мистецтва. Наприклад, серія ігор 'Відьмак' стала основою для телесеріалу, а 'Assassin's Creed' надихнула кінематографічний фільм і комікси. Такі спільні всесвіти стають все популярнішими і дозволяють фанатам поглиблюватися в світ гри та відчувати нові емоції від улюблених персонажів.

Крім того, деякі ігри використовуються як навчальні матеріали або для наукових досліджень. Існують масштабні змагання з ігор, які називаються кіберспортом, вони відбуваються на різних рівнях, від регіональних до світових, і привертають увагу мільйонів гравців та глядачів.

Комп'ютерні ігри мають значний вплив на сучасне суспільство, і останнім часом спостерігається стійка тенденція до гейміфікації в неігровому прикладному програмному забезпеченні. Це означає використання ігрових елементів та механік для залучення та мотивації користувачів.

Наприклад, деякі європейські навчальні заклади використовують відомі ігри для навчання, а для армій створюються симулятори для тренування солдатів. Крім того, деякі країни визнали кіберспорт офіційним видом спорту, а уряд США визнав комп'ютерні ігри окремим видом мистецтва ще у 2011 році, поряд із театром та кіно.

Ці зміни вказують на те, що комп'ютерні ігри тісно вплетаються в наше сучасне життя. Їх сфера використання постійно розширюється: ігри використовуються не лише для розваг, але й для проведення наукових досліджень та навчання.

Внаслідок того, що критерії віднесення гри до того чи іншого жанру не є однозначними, класифікація комп'ютерних ігор недостатньо систематизована, і в різних джерелах можуть бути відмінності в описі жанру конкретного проекту. Проте, серед розробників ігор існує консенсус, що дозволяє визначати приналежність гри до одного з основних жанрів зазвичай однозначно.

Основним критерієм поділу жанрів є типові дії, які зазвичай виконуються у грі. Гри можна поділити на три великі групи: ігри контролю, ігри дії та ігри інформації. Кожна група має свої особливості, але одночасно має різні відхилення.

На основі цього розроблено 15 основних геймплейних елементів, які включають навчання, загадки, спілкування, роль, вивчення, збирання, ухилення, знищення, змагання, техніка, турбота, розвиток, контроль, тактика та планування. Ці елементи можуть бути використані для класифікації ігор на основі їх геймплею. Ці жанри були поділені на 3 різні класифікації(Таблиця 1.1).

Таблиця 1.1 – Жанрова класифікація комп’ютерних ігор

Категорія	Ігри інформації	Ігри дій	Ігри контролю
Гібридні жанри	Action-RPG Rogue-lite	MMOFPS Survival	RTS MOBA
5 елемантів	Open RPG	Open Action	Global Strategy
2 елемента	Puzzle Quest Browser RPG	Platformer Stealth-Action Fighting Racing	Economical Tower Defense Wargame Cardgame
1 елемент	Education Test Contact Hero Toure	Arcade Horror Shoote r Sport Simulator	Logic Tactic MicroControl Building Life Sim
Елемантарні жанри	Навчання Загадки Спілкування Роль Вивчення	Збирання Ухилення Нищення Змагання Водіння	Турбота Створення Контроль Тактика Планування

«RPG» — «рольова гра» — є золотою серединою групи. Ігри, в яких можна жити, змушують вас грати в ролі героя, і атмосфера, сюжет і ігровий світ є їх головними перевагами.

Ігри дії: У іграх цієї групи головним є рухи, які потрібно виконувати, керуючи якимось тілом (людським або гуманоїдним) або технічним засобом. «Action» — гра-бойовик — є золотою серединою групи. Найбільш рухливі ігри Ці ігри часто хвалять за те, що вони тренують швидкість реакції.

Ігри контролю: група ігор, основною метою яких є планування подій і управління, щоб отримати перевагу в майбутньому. Сюди входять різні

стратегії, економічні ігри, варгейми та тактики. Золотою серединою групи є «Strategy» (звичайна локальна стратегія).

Розподіл ігор на одиночні та мультиплеєрні є важливою класифікацією, що визначає режими гри. Одиночні ігри призначені для участі лише одного гравця, який зазвичай стикається з комп'ютером чи штучною інтелектом. Його завдання може включати проходження ігри, накопичення ресурсів або розвиток навичок.

Мультиплеєрні ігри, натомість, дозволяють більш ніж одній особі грати одночасно. Цей режим може бути реалізований різними способами:

-Мультиплеєр на одному комп'ютері: гравці грають на одному пристрої, ділячи клавіатуру або користуючись одним контролером.

-Офлайн-мультиплеєрні ігри: гравці знаходяться в одному фізичному просторі, проте грають без підключення до Інтернету. Це може бути режим спільної гри на консолях чи комп'ютерах через мережу LAN або бездротові з'єднання.

-Онлайн-мультиплеєрні ігри: гравці з'єднуються через Інтернет для гри разом. Цей тип ігор може включати як масові мультиплеєрні онлайн-ігри, де велика кількість гравців взаємодіють у великому віртуальному світі, так і менші групові чи кооперативні ігри, де декілька гравців спільно виконують завдання.

Ця класифікація допомагає визначати, який тип гри підходить кожному гравцеві та які можливості спільної гри доступні.

Розподіл комп'ютерних ігор за візуальною складовою допомагає краще розуміти їх естетичні та технічні аспекти. Ось короткий опис кожної з цих категорій:

-Текстові ігри: це ігри з мінімальною кількістю графічних елементів, де спілкування з гравцем відбувається за допомогою тексту. Вони часто спираються на уяву гравця та зосереджені на навчанні та вирішенні загадок.

-2D ігри: у цих іграх усі елементи розроблені за допомогою двовимірної графіки. Це можуть бути платформери, стратегії, аркади та інші.

-3D ігри: у цих іграх усі елементи розроблені за допомогою тривимірної графіки, що надає їм реалістичний вигляд. Це можуть бути шутери, екшн-ігри, симулятори та інші.

Після ретельної оцінки класифікацій комп'ютерних ігор було вирішено створити гру у жанрі rogue-lite. Хоча цей жанр має свої особливості, але він має багато прихильників.Rogue-liteвідрізняється головним атрибутом - процедурно генерований світ, що гарантує унікальність кожної гри кожного разу, коли гравець починає заново. Крім того, гра має високий рівень складності, і кожен крок може призвести до смерті гравця, що вимагає початку гри спочатку.

У rogue-liteгравець повинен боротися з різними монстрами, збирати предмети та досліджувати випадково створені місця. Зазвичай гра відбувається за принципом ходів, коли гравець робить хід, а потім монстри роблять свої ходи.

Крім того, у грі відсутня можливість зберегти свій прогрес, тому кожна спроба дає вам новий шанс дійти до кінця гри.

1.2 Огляд жанру та аналіз існуючих розробок

Жанр roguelike виник у 1980-х роках із появою ігри "Rogue", яка визначила основні характеристики цього жанру. Головними особливостями

roguelike є процедурно генеровані рівні, перманентна смерть персонажа та велика випадковість у геймплейних елементах. Ці аспекти роблять кожну гру у жанрі унікальною та викликають високий рівень динаміки та несподіваності.

Жанр roguelike пройшов довгий шлях від своєї першої появи. Він еволюціонував разом із технологічними змінами та змінами в смаках гравців. Від класичних представників жанру, таких як "Nethack" та "Angband", до сучасних інтерпретацій, таких як "The Binding of Isaac" та "Dead Cells", roguelike завжди залишався популярним жанром серед геймерів.

Основні елементи геймплею є процедурно генеровані рівні, велика кількість персонажів та предметів, а також випадкові події. Ці аспекти створюють непередбачуваність та викликають постійний інтерес гравців до гри.

Roguelike має безліч видатних представників, які відзначаються своєю унікальною механікою та глибиною геймплею. Ігри, такі як "Spelunky", "FTL: Faster Than Light" та "Enter the Gungeon", здобули велику популярність та визнання серед геймерської спільноти.

Сьогоднішні roguelike ігри продовжують залишатися актуальними та популярними серед гравців. Інді-розробники продовжують експериментувати з механікою та геймдизайном, щоб принести нові ідеї та інновації у жанр.

Майбутнє жанру rogue-lite обіцяє ще більше інновацій та розвитку. З використанням сучасних технологій штучного інтелекту та глибшої процедурної генерації, ми можемо очікувати народження нових захоплюючих ігор у цьому жанрі.

1.1.1 Crypt of the NecroDancer

Crypt of the NecroDancer - це унікальна інді-гра, яка поєднує в собі елементи рогалика і ритм-гри. Розроблена студією Brace Yourself Games, гра отримала велику популярність завдяки своїй оригінальності, захоплюючому геймплею та веселій атмосфері.

У Crypt of the NecroDancer гравці контролюють персонажа, який рухається та бореться з монстрами на ритмі музики. Кожен крок, атака або дія гравця повинні відбуватися в такт музики, що додає грі унікальну динаміку і складність(Рисунок 1.2.1).

Основні механіки гри Crypt of the NecroDancer включають:

-Ритмічний рух: Гравці повинні рухатися та боротися з ворогами в такт музики, дотримуючись ритму для збільшення точності та ефективності.

-Бої з монстрами: Гравці зустрічають різні монстри з унікальними атаками та властивостями, і повинні використовувати свої навички і стратегії, щоб їх перемогти.

-Експлорація підземелля: Гравці розгадують загадки, знаходять скарби та зброю, досліджуючи різні рівні підземелля.

-Персонажі та прогресія: Гравці можуть розблоковувати нових персонажів з унікальними властивостями та покращувати свої навички під час гри.

-Різноманітність музичних треків: Гра має велику кількість різноманітних музичних треків, які додають різноманіття і енергію до геймплею.

Рисунок 1.2.1-Знімок екрана ігрового процесу гри Crypt of the NecroDancer

1.1.2 Enter the Gungeon

Enter the Gungeon - це інді-гра, яка поєднує в собі жанри рогалика та вогняної стрільби. Розроблена студією Dodge Roll у квітні 2016 рогу, гра отримала велику популярність завдяки своїй високій складності, швидким і динамічним боям, а також великому арсеналу зброї і предметів.

Сюжет гри розгортається в міфічному світі, де головні герої шукають легендарний предмет, відомий як "Gun That Can Kill the Past" ("Пістолет, що може вбити минуле"). Гравці відправляються в Ганж (Gungeon) - великий склад небезпечних загадок і ворожих створінь, де кожен поворот може призвести до смертельної зустрічі або до відкриття нового артефакту(Рисунок 1.2.2).

Основні механіки гри Enter the Gungeon включають:

-Безперервні вогняні бої: Гравці беруть на себе роль героя, який бореться з хвилями ворогів, використовуючи різні види зброї, включаючи вогнепальні зброї, мечі, лазери та інші екзотичні артефакти.

-Генерація процедурних рівнів: Кожен прохід "Gungeon" створюється випадковим чином, що забезпечує невідому та викликаючу атмосферу кожного проходження.

-Збір предметів та зброї: Гравці можуть знаходити різноманітні предмети, зброю та артефакти, які покращують їхні можливості в бою.

-Множник гравців: Гра підтримує режим гри для одного гравця або спільну гру для до чотирьох гравців, що дозволяє друзям вирушати в Ганж разом.

Рисунок 1.2.2-Знімок екрана ігрового процесу гри Enter the Gungeon

1.1.3 Slay the Spire

Slay the Spire - це інді-гра, яка поєднує в собі елементи колодових ігор та рогалика. Розроблена студією MegaCrit у січні 2019 року, гра стала дуже популярною завдяки своїй стратегічній глибині, високій реіграбільності та захоплюючій геймплейній механіці.

У Slay the Spire гравці відправляються в подорож по вежі, яка заповнена небезпечними ворогами, загадковими подіями та скарбами. Основна мета полягає в тому, щоб пройти крізь різні рівні вежі, збираючи

картки для своєї колоди, покращуючи свої навички і зброю, а також пристосовуючись до змінюючихся умов(Рисунок 1.2.3).

Основні механіки гри "Slay the Spire" включають:

-Гра на картках: Гравці використовують картки, щоб виконувати різні дії в бою, такі як атака, захист, спеціальні ефекти тощо. Кожна колода може бути унікальною і підлаштованою під гравцям стиль гри.

-Процедурна генерація рівнів: Кожен проходження вежі створюється випадковим чином, що забезпечує непередбачуваність і виклик кожного нового проходження.

-Різноманітні персонажі: У грі присутні різні персонажі з унікальними наборами карток, здібностями та стратегіями.

-Прогресія і розвиток: Гравці можуть збирати нові картки, покращувати свої навички, отримувати бонуси та артефакти, що полегшують подорож.

-Вибір та рішення: Гравцям доводиться приймати стратегічні рішення на кожному кроці своєї подорожі.

Рисунок 1.2.3 -Знімок екрана ігрового процесу гри Slay the Spire

1.1.4 Barony

Barony - це інді-гра в жанрі ролевого екшену з елементами рогалика і песочниці. Розроблена студією Turning Wheel LLC у червні 2015 року, гра відрізняється своїм унікальним підходом до геймплею і атмосферою середньовічного фентезі.

У Barony гравці відправляються в подорож по великому підземелля, яке заповнене небезпечними монстрами, загадковими артефактами та скарбами. Головна мета полягає в тому, щоб дійти до найглибшого рівня підземелля і перемогти великого злочинця, який заснував його.

Основні механіки гри Barony включають:

-Рольові елементи: Гравці можуть вибирати різних персонажів з унікальними вміннями та характеристиками, а також розвивати їх навички і властивості протягом гри.

-Експлорація: Гравці мають вивчати різні рівні підземелля, знаходячи секрети, розгадуючи головоломки та борючись з монстрами.

-Бої: Гра включає в себе виборні бої з різними ворогами, які можуть включати в себе стрільбу, магічні закляття та ближній бій.

-Процедурна генерація рівнів: Кожне проходження генерує нові рівні підземелля, що забезпечує велику переігрованість.

-Мультиплеер: Гравці можуть грати як самотньо, так і з друзями у спільному режимі.

Рисунок 1.2.4 -Знімок екрана ігрового процесу гри Barony

1.1.5 Порівняння різних ігор

Таблиця 1.2 відображає, наскільки різноманітні можливості у геймдизайні, графіці та інших аспектах можуть бути в жанрі roguelike. Розробники можуть експериментувати з різними елементами, такими як графіка, геймплей та режим гри, щоб створювати унікальні та цікаві ігри, які привертають увагу гравців.

Таблиця 1.2-Таблиця порівняння ігор

Характеристика	Crypt of the NecroDancer	Enter the Gungeon	Slay the Spire	Barony
Жанр	roguelie,рит м-екшн	rogue-lite, екшн	rogue-lite, колодоскла-дальна	rogue-lite,ролева
Графіка	2D	2D	2D	3D
Геймплей	Ритмічні поєдинки, генерація рівнів	виживання, пошук зброї,битви з монстрами	Збирання колоди, битви з монстрами	дослідження підземелля, збирання скарбів, битви з ворогами
Режим гри	Одиночний гравець	Коопера-тив	Одиночний гравець	Кооператив
Генерація рівнів	процедурна	процедурна	процедурна	процедурна
Розвиток персонажа	Збір артефактів та відкриття персонажів	Збір та відкриття персонажів та зброї.	Збір та відкриття карток та персонажів	Збір артефактів, покращення здібностей персонажа

Жанр roguelike виявляється надзвичайно різноманітним та захоплюючим для як розробників, так і гравців. Це чітко відображається у порівняльній таблиці 1.2, де представлені різні ігри з цього жанру. Різноманіття виражається у кожному аспекті гри: від жанрових характеристик до графічних стилів, від механіки геймплею до способів

розвитку персонажів. Це вказує на те, що roguelike- це не просто один жанр ігор, а універсальна база для експериментів розробників.

Гравці, з свого боку, мають можливість насолоджуватися широким спектром геймплейних досвідів та вибирати ігри, які відповідають їхнім уподобанням та інтересам. Таким чином, різноманітність у жанрі rogue-lite створює дуже стимулююче та захоплююче середовище як для розробників, так і для гравців.

Потенціал розвитку жанру

Жанр rogulike став набувати популярність у 2014 році з виходом The Binding of Isaac: Rebirth а пік інтересу був у 2018 році з виходом Dead Cells.Хоча пік популярності пройшов але аудіторія у ігор залишалась і даже в наш час виходе велика кількість Roguelike.

За даними SteamDB ми бачимо що максимальний кількість гравців була 3 року назад зараз же 20 тисяч гравців є середнім онлайном гри(Рисунок 1.3.1).

Рисунок 1.3.1-Графік популярності The Binding of Isaac: Rebirth за весь час

На сьогодні найпопулярнішою грою за статистикою SteamDB є Hades 2, яка є rogue-lite(Рисинок 1.3.2).

Рисинок 1.3.2-Графік популярності Hades 2 за весь час

Статистика показує, як змінюються тренди в жанрі roguelike. Якщо раніше популярність базувалась на складності та простоті, то зараз гравців приваблює історія та візуал. Також більше уваги стали приділяти кооперативу. Можна сказати, що roguelike еволюціонував у rogue-lite.

Жанр rogue-lite, хоч і не лідер серед комп'ютерних ігор за популярністю, проте постійно привертає свою аудиторію та знаходить нових шанувальників. Щодо майбутнього розвитку цього жанру, очікується, що розробники продовжуватимуть експериментувати з механіками генерації випадкових рівнів та подій. Крім того, очікується збільшення кількості доступних персонажів з унікальними вміннями та здібностями.

Один із трендів в розвитку rogue-lite-ігор - це зростання ролі наративу та історії. Раніше такі ігри були спрямовані переважно на геймплей та випадковість подій, але тепер розробники все більше звертають увагу на наративну складову.

Також очікується подальше зростання популярності режиму гри в кооперативі. Цей тренд стає все більш популярним серед геймерів. Гра в кооперативному режимі дозволяє гравцям спільно пройти гру разом з друзями або іншими гравцями онлайн. Це створює можливість спілкування, взаємодії та спільної праці для досягнення спільних цілей. Крім того, цей режим дозволяє гравцям поділитися викликом та емоціями гри, що зближує їх та створює незабутні спогади.

Отже, хоча жанр rogue-lite може не бути найпопулярнішим серед геймерів, він все одно має свою власну аудиторію та потенціал для розвитку. Розробники продовжують експериментувати з новими ідеями та механіками, що обіцяє цікаві та захоплюючі ігрові досвіди для шанувальників жанру.

Усі ці тенденції можуть призвести до розробки ще більш складних та глибоких ігор у жанрі rogue-lite, що спрямовані на привертання ще більшої уваги гравців та стануть популярними на ринку комп'ютерних ігор.

Розвиток віртуальної реальності та інших інноваційних технологій також може сприяти популярності жанру rogue-lite. Використання технології віртуальної реальності дозволить гравцям ще глибше занурюватися в гру та відчувати себе частиною ігрового світу. Крім того, розробники можуть експериментувати з механікою гри, щоб зробити її більш захопливою та динамічною, використовуючи нові функції та можливості, які надають нові технології.

За останні роки кількість Roguelike-ігор значно зросла, багато з них стали піджанрами, що відрізняються унікальними механіками та характеристиками. Наприклад, "Roguelite", "Roguelike-like", "Roguelike RPG" та інші.

Гра "Dead Cells" є прикладом "Rogue-lite" ігри, яка має елементи жанру Roguelike, але також зосереджується на бойовій системі та прогресії персонажа.

Гра "Spelunky" є прикладом "Roguelike-like" ігри, яка має схожу механіку генерації випадкових рівнів та постійну смертність персонажа, але може бути менш важкою та має менше елементів RPG.

Гра "Darkest Dungeon" є прикладом "Roguelike RPG" ігри, яка зосереджується на розвитку персонажа та нарativі, але також має елементи випадкових рівнів та постійної смертності персонажів, що є характерним для жанру Roguelike.

Кожен з них має свої особливості та характеристики, які можуть привернути різні групи гравців.

Жанр Roguelike постійно знаходиться у стані розвитку та еволюції, пропонуючи гравцям нові інноваційні ідеї та механіки гри. Останнім часом цей жанр став більш доступним та зрозумілим для новачків завдяки спрощеним версіям, таким як "Roguelite" або "Roguelike-like". Завдяки постійному розвитку та експериментації з новими ідеями, жанр Roguelike має потенціал привернути ще більше гравців та зайняти своє місце на ринку комп'ютерних ігор.

Жанр rogue-lite є одним з тих жанрів, який може забезпечити безліч годин задоволення гравцям завдяки своїм випадково згенерованим рівням і постійною смертністю персонажів. Однак він може бути надзвичайно вимогливим та складним для тих, хто тільки починає грати у цей жанр. Тому важливо для розробників забезпечувати належну підготовку гравців до складності та випадковості ігрового процесу.

Одним із викликів для розробників є створення різноманітних та збалансованих геймплейних механік, які б дозволяли гравцям відчувати

себе унікальними та давали можливість змінювати ігровий досвід залежно від їх власних уподобань. Також важливо розвивати наративний аспект гри, щоб створити цікавий та захоплюючий світ, який би зацікавив гравців і надихнув їх на подальшу гру.

Жанр rogue-lite також може бути цікавим для використання у навчальних цілях, наприклад, для розвитку навичок прийняття рішень, стратегічного мислення та аналітичних здібностей. Деякі розробники вже створюють навчальні ігри на основі жанру rogue-lite, які можуть бути корисними для учнів та студентів.

У цілому, жанр rogue-lite є жанром з великим потенціалом та можливостями для розвитку та експериментацій, і ми можемо очікувати більше захоплюючих інноваційних ідей в майбутньому.

ВИСНОВОК ДО РОЗДІЛУ 1

Усі тенденції в розвитку жанру rogue-lite свідчать про його великий потенціал та привабливість для розробників і гравців. З роками цей жанр стає все більш доступним та цікавим завдяки спрощеним версіям, таким як "Roguelite" або "Roguelike-like", а також розвитку наративних аспектів гри та режиму гри в кооперативі.

Розробники постійно експериментують з новими ідеями та механіками, що обіцяє цікаві та захоплюючі ігрові досвіди для шанувальників жанру. Навіть у навчальних цілях цей жанр може мати великий потенціал для розвитку навичок прийняття рішень та стратегічного мислення.

Отже, жанр rogue-lite є динамічним і перспективним, з великим потенціалом для подальшого розвитку та експериментування. Чекаємо на ще більше захоплюючих інноваційних ідей в майбутньому.

2 РОЗРОБКА ПРОЕКТУ

2.1 Вибір середовища для розробки проекту

Розробка гри на власному рушії може мати як плюси, так і мінуси, і вибір між цими двома варіантами залежить від конкретних обставин та потреб проекту. Ось деякі аспекти, які слід враховувати при цьому виборі:

Переваги розробки на власному рушії:

-Контроль над функціональністю: Розробка на власному рушії дозволяє повністю контролювати функціональність гри та швидше внести зміни, якщо це необхідно.

-Унікальний стиль гри: Власний рушій дозволяє створити унікальний стиль гри, що може виділити вашу гру на ринку та привернути увагу гравців.

-Навчання та розвиток: Розробка на власному рушії може бути відмінною можливістю для навчання нових технологій та розвитку навичок програмування та геймдеву.

Недоліки розробки на власному рушії:

-Час та витрати: Розробка власного рушія може забрати багато часу та зусиль, особливо якщо ви не маєте достатнього досвіду у розробці ігрових движків.

-Необхідність підтримки: Підтримка власного рушія вимагає постійного зусилля для виправлення помилок, оновлення та розвитку, що може відволікати від основної розробки гри.

-Обмежені можливості: Порівняно з вже існуючими готовими рушіями, ваш власний рушій може мати обмежені можливості та функціональність, особливо на початковому етапі розробки.

Обрати готовий ігровий рушії для розробки гри може бути обґрунтовано з різних причин:

- Ефективність розробки: Готовий ігровий двигун зазвичай має вже побудовану базову функціональність та набір інструментів, які значно спрощують розробку гри. Це може допомогти вам значно зекономити час та зусилля, які витратилися б на розробку всього з нуля.

-Спільнота та підтримка: Популярні готові рушії зазвичай мають велику спільноту користувачів та активну підтримку з боку розробників. Це означає, що ви можете отримати доступ до великої кількості документації, підручників, форумів підтримки та навіть плагінів, які можуть полегшити вашу роботу та допомогти у вирішенні проблем.

-Якість та надійність: Використання вже перевіреного та випробуваного ігрового двигуна може забезпечити вам більшу якість та надійність гри. Багато готових рушіїв проходять інтенсивне тестування та мають велику кількість користувачів, які звітують про помилки та проблеми, що допомагає у виправленні їх швидко та ефективно.

-Масштабованість: Багато готових рушіїв розроблені з урахуванням масштабованості, що дозволяє легко розширювати функціональність вашої гри з розвитком проекту. Вам не потрібно буде вигадувати колесо, коли йдеться про додавання нових функцій чи модулів.

-Ресурсозбереження: Використання готового рушія може значно зменшити витрати на розробку та підтримку, оскільки ви можете скористатися вже наявними ресурсами та інструментами, замість того, щоб витрачати кошти та час на створення їх з нуля.

Отже, обираючи готовий ігровий рушії, можна отримати багато переваг, які допоможуть ефективно та якісно реалізувати свою ідею гри.

Є багато ігрових рушіїв з яких можно обрати найкращій для розробки гри в жанрі rogue-lite.

2.1.1 Godot Engine:

Godot - це відкритий ігровий рушій та середовище розробки (IDE), яке було розроблене компанією OKAM Studio у 2007 році із відкриттям вихідного коду у 2014 році. З тих пір він став набувати популярності серед розробників ігор, особливо серед початківців і тих, хто шукає безкоштовні інструменти для створення ігор.

Основні характеристики Godot:

-Безкоштовний та відкритий: Godot є повністю безкоштовним та відкритим програмним забезпеченням. Ви можете використовувати його для будь-яких цілей, включаючи комерційні проекти, безкоштовно, а також змінювати історичний код за бажанням.

-Підтримка мов програмування: Godot підтримує кілька мов програмування, зокрема GDScript (мова програмування, подібна до Python, спеціально розроблена для Godot), C#, C++ та VisualScript (візуальний скриптовий мова). Це дає вам можливість вибрати ту мову, яка найбільше вам підходить або з якою ви найкраще ознайомлені.

-Кросплатформенність: Гри, створені за допомогою Godot, можуть бути експортовані на різні платформи, включаючи Windows, macOS, Linux, Android, iOS, HTML5 та інші.

-Можливості 2D та 3D: Godot підтримує як розробку двовимірних, так і тривимірних ігор. Він має потужні засоби для роботи з графікою, анімацією, фізикою та багато іншого.

-Розширені можливості: Godot постійно розвивається та оновлюється, і його спільнота постійно вносить нові функції та покращення. Ви можете розширювати функціональність своєї гри, використовуючи різноманітні плагіни та розширення.

Загалом, Godot - це потужний та зручний інструмент для розробки ігор будь-якого рівня складності, від простих двовимірних ігор до складних тривимірних проектів. Він ідеально підходить як для початківців, так і для досвідчених розробників, завдяки своїм простим у використанні інструментам та гнучким можливостям.

Інтерфейс Godot є досить інтуїтивно зрозумілим та добре організованим, що робить роботу з цим двигуном дуже зручною(Рисунок 2.1.1).

Основні компоненти інтерфейсу Godot

-Редактор сцен (Scene Editor): Це основна область, де ви будете створювати та редагувати ваші ігрові сцени. Тут ви можете додавати об'єкти, налаштовувати їх властивості, створювати анімації та взаємодіяти з іншими елементами вашої гри.

-Вікно ієрархії (Scene Hierarchy): Це вікно показує ієрархію об'єктів на вашій сцені. Ви можете бачити всі об'єкти, які присутні на сцені, та керувати ними, робити з ними взаємодію та організовувати їх у групи.

-Вікно властивостей (Inspector): Це вікно відображає властивості об'єкта, який ви вибрали у вікні ієрархії або на сцені. Тут ви можете змінювати параметри об'єкта, такі як розмір, положення, матеріал, анімація та багато іншого.

-Вікно ресурсів (FileSystem/Assets): Це вікно показує ваші ресурси, такі як зображення, звуки, скрипти та інші файли, які використовуються в

вашій грі. Ви можете переглядати, додавати, видаляти та організовувати ваші ресурси за допомогою цього вікна.

-Панель інструментів (Toolbar): Ця панель містить корисні інструменти та кнопки швидкого доступу для роботи з вашим проектом. Тут ви можете знайти інструменти для вибору, переміщення, масштабування та інших операцій над об'єктами.

-Вікно скриптів (Script Editor): Це вікно призначене для написання, редагування та організації вашого коду. Ви можете створювати нові скрипти, відкривати вже існуючі та працювати з ними безпосередньо у вбудованому текстовому редакторі.

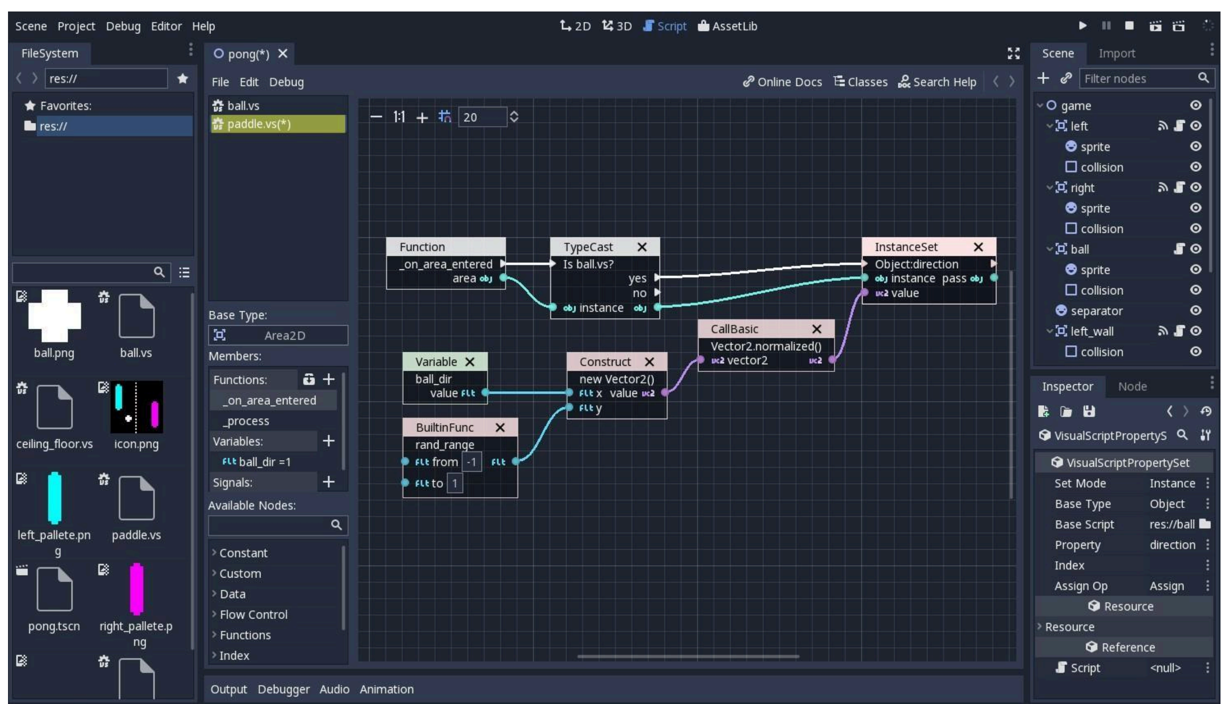


Рисунок 2.1.1-Інтерфейс Godot

Зараз Godot набуває популярності і на нему були розроблені такі ігри як(Рисунок 2.1.2):

-Life of Land гра є платформер з елементами головоломок, де гравець вирушає у подорож через різноманітні локації, вирішуючи загадки та збираючи ресурси для виживання.

-Usafi Shima це пазл-гра з елементами пригод, де гравцям доводиться досліджувати остров, вирішувати головоломки та допомагати його мешканцям.

-Tail Quest це комбінація tower defense і hack-and-slash гри, де гравці захищаються від ворогів, керуючи героями з анімалістичними хвостами.

-Cassette Beats: - це ритм-ігра, де гравцям доводиться розбивати об'єкти на різній ритмічний музиці, створюючи свої унікальні мелодії.

-Hal's Torment це хоррор-пригода, де гравцям доводиться досліджувати місце подій, розгадувати загадки та уникати небезпечних ситуацій.

-Lumencraft це містичний платформер, де гравцям доводиться використовувати світло для розв'язання головоломок та виживання у світі магії та загадок.

-Life of Land ця гра - це платформер з елементами головоломок, де гравець вирушає у подорож через різноманітні локації, вирішуючи загадки та збираючи ресурси для виживання.

-Usafi Shima - це пазл-гра з елементами пригод, де гравцям доводиться досліджувати остров, вирішувати головоломки та допомагати його мешканцям.

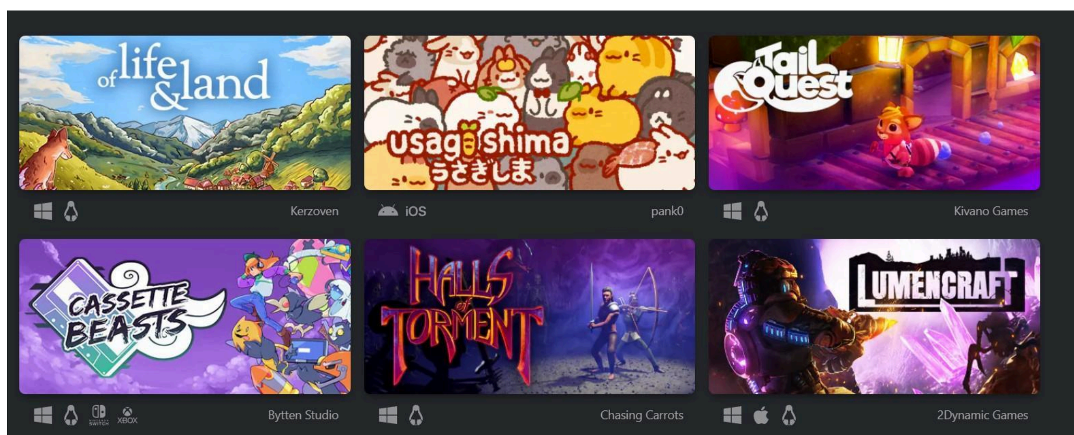


Рисунок 2.1.2-Ігри розроблені на Godot

Хоча Godot Engine може бути досить потужним інструментом для розробки ігор, включаючи рогалики, деякі розробники можуть виявити деякі обмеження чи недоліки, які призводять до вибору інших двигунів. Ось деякі можливі причини, чому деякі розробники можуть вирішити не використовувати Godot для розробки рогаликів:

-Відсутність спеціалізованих інструментів для рогаликів: На відміну від інших ігрових двигунів, таких як rogue-lite Toolkit (libtcod), Godot не має вбудованих інструментів або функціоналу, спеціально призначених для розробки рогаликів. Це може зробити процес розробки рогаликів менш зручним для деяких розробників.

-Складність процедурної генерації: В рогаликах процедурна генерація рівнів є ключовою особливістю. Хоча Godot підтримує процедурну генерацію, налаштування складних систем генерації може бути складнішим порівняно з іншими спеціалізованими інструментами.

-Швидкодія: Іноді рогалики можуть вимагати великої кількості обчислень та оптимізації для плавної гри на різних пристроях. Хоча Godot є потужним двигуном, існують інші двигуни, які можуть забезпечити кращу швидкодію або більшу ефективність для певних типів ігор.

-Спільнота та підтримка: На відміну від деяких інших ігрових двигунів, які мають довгу історію підтримки розробки рогаликів, Godot може мати меншу спільноту та менше ресурсів, призначених для розробки рогаликів.

Це не означає, що Godot погано підходить для розробки ігор у жанрі rogue-lite.

2.1.2 LibGDX

LibGDX - це відкритий фреймворк для розробки ігор, який базується на Java. Він є дуже популярним серед розробників, які хочуть створювати ігри для різних платформ, включаючи ПК, мобільні пристрої та веб.

LibGDX був розроблений Ніком Ковачем (Nick "badlogic" Kovacs), німецьким програмістом і геймдевелопером, який був зацікавлений у розробці відкритого фреймворку для створення ігор для платформи Java. Початково він розпочав роботу над фреймворком в 2009 році як проект для власних потреб у розробці гри. Після того, як він розповів про свій проект на форумах, він отримав великий інтерес від інших розробників, які шукали рішення для створення ігор на Java.

Основні переваги LibGDX:

-Кросплатформенність: LibGDX дозволяє розробникам створювати ігри для різних платформ, включаючи Android, iOS, Windows, macOS, Linux та HTML5.

-Швидкодія: Фреймворк LibGDX відомий своєю високою продуктивністю та ефективністю. Він надає прямий доступ до OpenGL та низькорівневої машинної графіки, що дозволяє розробникам отримати більше контролю над процесом розробки та оптимізацією швидкодії.

-Модульність: LibGDX має модульну архітектуру, яка дозволяє розробникам використовувати лише ті частини фреймворку, які їм потрібні для конкретного проекту.

-Спільнота та ресурси: LibGDX має велику активну спільноту розробників, яка надає безкоштовну підтримку та ресурси, такі як документація, форуми та сторонні бібліотеки.

-Простота використання: Хоча LibGDX має досить стандартну для розробки ігор на Java структуру, він все ж дозволяє створювати складні ігри зі складними геймплейними механіками.

Загалом, LibGDX є потужним та гнучким фреймворком для розробки ігор, і він може бути хорошим вибором для розробки рогаликів, особливо якщо ви зручні з Java та шукаєте кросплатформенну альтернативу.

Інтерфейс LibGDX досить простий та легкий у використанні, але водночас потужний для розробки ігор(Рисунок 2.1.3).

Ось деякі ключові компоненти інтерфейсу LibGDX:

-Клас Game або ApplicationListener: Це основний клас вашої гри, який реалізує інтерфейс ApplicationListener. Ви можете використовувати цей клас для керування життєвим циклом вашої гри та реалізації основного геймплею.

-Клас Screen: LibGDX використовує концепцію екранів для розділення різних частин вашої гри, таких як головне меню, гра, налаштування тощо. Кожен екран реалізує інтерфейс Screen.

-Клас Stage та Actor: Ці класи використовуються для створення та управління інтерфейсом користувача вашої гри. Вони дозволяють додавати текстові поля, кнопки, малюнки та інші елементи на екран.

-Малювання та графіка: LibGDX надає можливості для малювання графіки за допомогою класів, таких як SpriteBatch та ShapeRenderer. Ви можете створювати різні графічні об'єкти, такі як текстури, спрайти та форми, і відображати їх на екрані.

-Аудіо та звук: LibGDX дозволяє вам працювати з аудіофайлами, відтворювати звуки та музику, а також керувати їх параметрами, такими як гучність та панорама.

-Введення користувача: Фреймворк надає можливості для обробки введення користувача, такого як клавіші, миша та сенсорні взаємодії на мобільних пристроях.

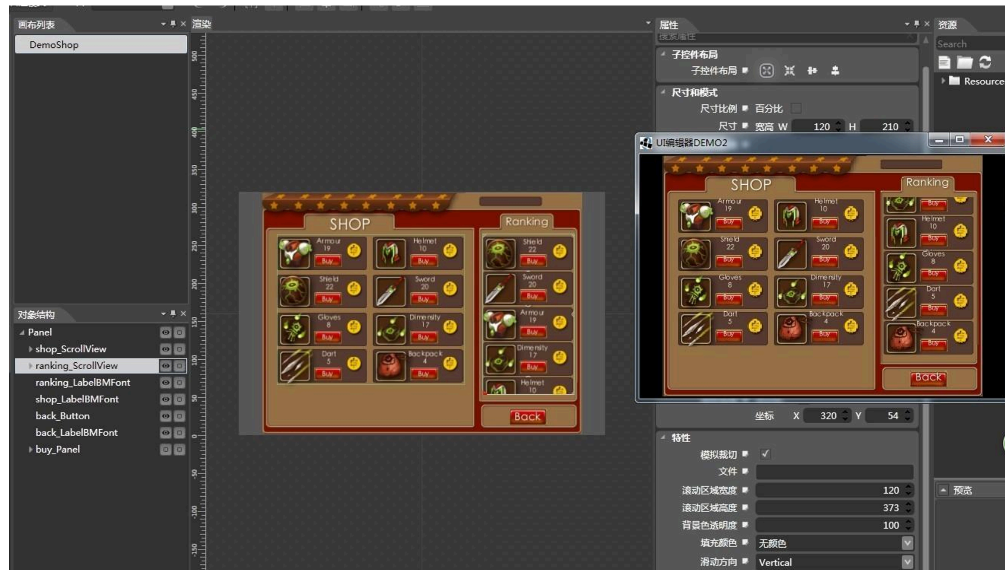


Рисунок 2.1.3-Інтерфейс LibGDX

Ігровий фреймворк LibGDX став основою для багатьох цікавих та успішних ігор на різних платформах. Ось деякі приклади ігор, створених з використанням LibGDX(Рисунок 2.1.4):

-Demise of Nations: Це стратегічна гра, в якій гравці керують своєю цивілізацією та змагаються за світове господарство. Гра надає можливість грати на різних історичних картах та виконувати різноманітні завдання.

-Age of Conquest IV: Це також стратегічна гра, де гравці керують своєю цивілізацією та стрімко розширюють свій вплив, завойовуючи та адміністуючи нові території. Гра має різноманітні режими та можливості налаштування.

-Heroes of Loot: Це ретро-рогалик, де гравці вирушають у підземелля, щоб вбити монстрів та зібрати скарби. Гра має швидкий темп гри та велику кількість ворогів та предметів.

-Slice Dice: Ця гра-головоломка, в якій гравці повинні різати фрукти та додавати їх до кошиків з відповідними числами, щоб набрати балів. Гра має простий, але дуже затягуючий геймплей.

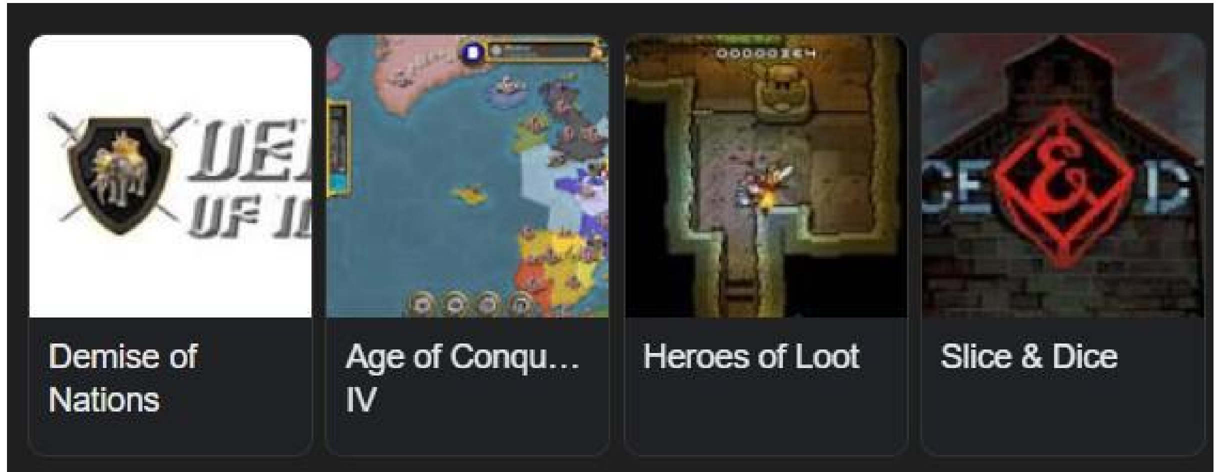


Рисунок 2.1.4-Ігри розроблені на Libgdx

LibGDX - це потужний інструмент для розробки ігор на платформі Java, проте для розробки конкретно рогаликів може виявитися менш практичним порівняно з іншими фреймворками або інструментами. Ось кілька причин, чому LibGDX може погано підходити для розробки рогаликів:

-Відсутність спеціалізованих інструментів: LibGDX не має вбудованих інструментів або функціоналу, спеціально призначених для розробки рогаликів, таких як підтримка процедурної генерації рівнів, що може ускладнити процес розробки.

-Складність процедурної генерації: Розробка складних систем процедурної генерації рівнів у LibGDX може виявитися складнішою порівняно з іншими фреймворками або бібліотеками, які спеціалізуються на цьому аспекті.

-Неоптимальність для деяких геймплейних механік: Хоча LibGDX дозволяє створювати різноманітні типи ігор, він може не бути

оптимальним для деяких унікальних геймплейних механік, що часто зустрічаються в рогаликах.

-Вимоги до відомостей Java: Розробка рогаликів у LibGDX може вимагати певного рівня знань Java, що може бути складним для новачків або тих, хто не має досвіду розробки на цій мові.

Хоча LibGDX може бути використаний для створення рогаликів, деякі розробники можуть вибрати інші інструменти або фреймворки, які мають більшу спеціалізацію на розробці рогаликів або надають більшу підтримку для цього жанру.

2.1.3 Unity

Unity - це інтегрований ігровий движок, розроблений компанією Unity Technologies. Він є одним з найпопулярніших інструментів для створення ігор і використовується розробниками з усього світу для різних проектів, від інді-ігор до великих триплей-А тайтлів.

Історія створення Unity почалась у 2002 році, коли девелопер Девід Геллерсон приступив до розробки програмного забезпечення для створення веб-прикладів та інтерактивних мультимедійних додатків.Через деякий час він разом з Йоакімом Антарою та Нікласом Хелмстремом заснував компанію Unity Technologies у 2004 році.

Перша версія Unity вийшла в 2005 році, і вона була призначена для розробки ігор для Mac OS X. Протягом наступних років компанія постійно розширювалася та вдосконалювала свій ігровий движок, додавши підтримку для різних платформ, таких як Windows, iOS, Android, Xbox, PlayStation та інші.

Unity має безліч переваг, які роблять його одним з найпопулярніших інструментів для розробки ігор. Ось кілька ключових переваг Unity:

-Кросплатформенність: Unity дозволяє розробляти ігри для різних платформ, включаючи ПК, консолі, мобільні пристрої (Android та iOS), веб-браузери та навіть віртуальну реальність. Це дозволяє розробникам легко переносити свої ігри на різні пристрої та ринки.

-Зручний інтерфейс розробки: Unity надає зручний інтерфейс розробки з великою кількістю інструментів та ресурсів, які полегшують процес розробки. Інтегрована система візуальної розробки, редактор сцен, компонентна архітектура та інші зручні функції дозволяють розробникам швидко та ефективно створювати ігри.

-Багатий функціонал: Unity має вражаючий набір функцій, які допомагають розробникам створювати вражаючі ігри. Це включає в себе різноманітні графічні ефекти, фізичний движок, системи анімації, штучний інтелект, аудіо-движок, мережеву підтримку та багато іншого.

-Активна спільнота та ресурси: Unity має велику та активну спільноту розробників, яка готова допомагати новачкам і досвідченим розробникам. Крім того, є багато ресурсів, таких як документація, онлайн-курси, форуми та інші, які допомагають розробникам вивчати і вдосконалювати свої навички.

-Unity Asset Store: Unity Asset Store надає доступ до великого вибору готових активів, таких як графічні моделі, звуки, скрипти та інші, що полегшує процес розробки. Це дозволяє розробникам ефективно використовувати сторонні ресурси для швидкої розробки своїх ігор.

Інтерфейс Unity - це дружелюбне середовище розробки, яке надає зручні інструменти для створення ігор(Рисунок 2.1.5).

Ось кілька ключових елементів інтерфейсу Unity:

-Редактор сцен: Це основний робочий простір, де ви створюєте та редагуєте вашу гру. Ви можете розміщувати об'єкти, встановлювати їхні властивості та налаштування, а також розміщувати компоненти для керування поведінкою об'єктів.

-Інспектор: Це вікно, що відображає властивості та компоненти обраного об'єкта. Тут ви можете налаштувати параметри об'єктів, додавати та видаляти компоненти, а також керувати іншими аспектами об'єкта.

-Проект: Це вікно, що містить всі ресурси вашого проекту, такі як зображення, звуки, скрипти та інші файли. Ви можете легко керувати цими ресурсами, перетягуючи їх у сцени або використовуючи їх у вашій грі.

-Ієрархія: Це вікно, що відображає ієрархію всіх об'єктів у поточній сцені. Ви можете легко переглядати та організовувати ваші об'єкти за їхнім розташуванням у сцені.

-Консоль: Це вікно, що відображає повідомлення, помилки та іншу відладкову інформацію під час роботи вашої гри. Воно допомагає відслідковувати проблеми та вирішувати їх у процесі розробки.

-Меню інструментів: Unity має різноманітні меню та панелі інструментів, які надають доступ до різних функцій та операцій, таких як створення нових об'єктів, імпорт ресурсів, запуск гри та інше.

Ці елементи створюють зручне та ефективне середовище для розробки ігор у Unity. Інтерфейс Unity дозволяє розробникам швидко та ефективно створювати якісні ігри для різних платформ.

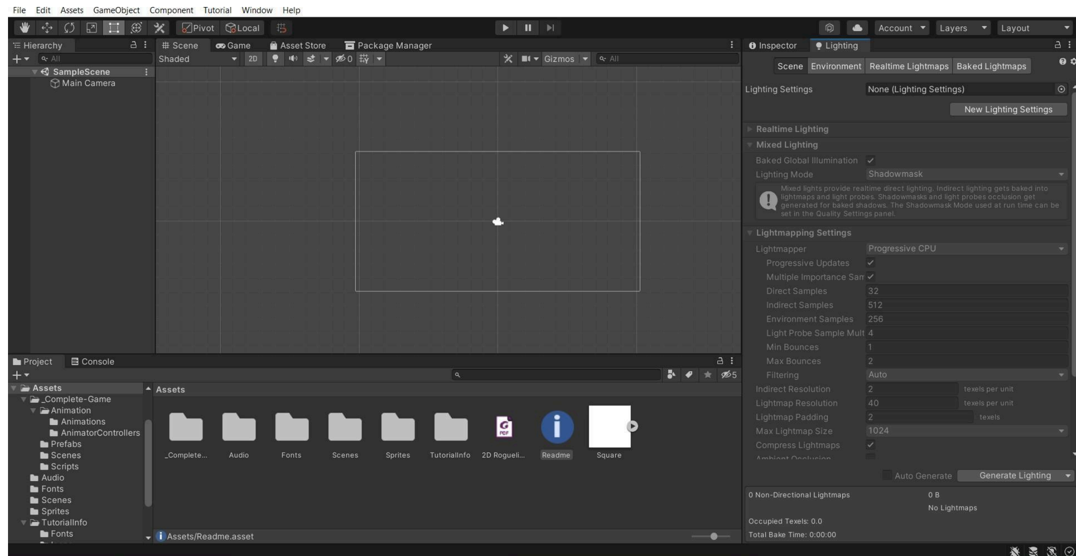


Рисунок 2.1.5-Інтерфейс Unity

Unity використовується для розробки ігор різних жанрів та рівнів складності, від простих інді-ігор до складних AAA-проектів(Рисунок 2.1.6).

Ось декілька відомих ігор, створених з використанням Unity:

Cuphead-це видатна інді-гра, яка сполучає в собі ретро стиль мультфільмів 1930-х років з вимогливими боєвими сценами. Гравці беруть на себе роль головного героя, чашки Cuphead або його друга Mugman, які змушені виконувати завдання для Диявола, щоб врятувати свої душі. Гра вражає не лише своїм унікальним візуальним стилем, але й захоплюючим геймплеєм і великим різноманіттям бойових сцен.

Hearthstone: "Hearthstone" - це відома колекційна карточна гра, розроблена Blizzard Entertainment. Гравці будують колоди з карточок, які представляють різних персонажів та заклинання з унікальними властивостями, і змагаються один з одним у дуелі на арені. Гра отримала велику популярність завдяки своєму простому, але стратегічному геймплею, а також світовому визнанню у геймерській спільноті.

Kerbal Space Program: "Kerbal Space Program" - це симулятор космічного польоту, де гравці відправляють екіпажі космічних кораблів на різні місії у великий відкритий космос. Гра вражає своєю реалістичністю та складністю, дозволяючи гравцям досліджувати закони фізики та конструювати різноманітні космічні апарати.

Cities: Skylines: "Cities: Skylines" - це симулятор міського будівництва, де гравці беруть на себе роль мера та керують розвитком свого міста. Гравцям доводиться планувати і будувати інфраструктуру, забезпечувати житлові та комерційні потреби мешканців, а також вирішувати різноманітні проблеми, такі як забруднення та затори.

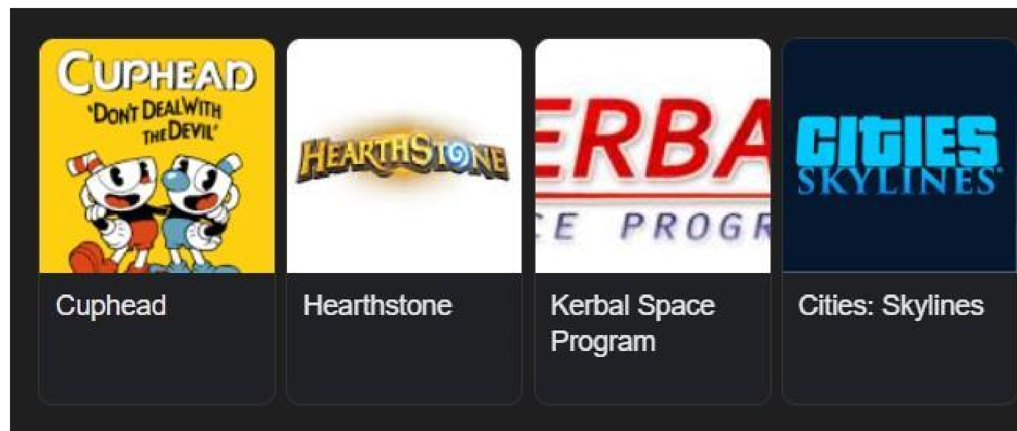


Рисунок 2.1.6-Ігри розроблені на Unity

Unity є ідеальним вибором для розробки гри жанру rogue-lite з кількох причин:

-Гнучкість і кросплатформенність: Unity дозволяє легко розробляти гри для різних платформ, таких як ПК, консолі, мобільні пристрої та веб. Це означає, що ваша гра rogue-lite може бути доступна на широкому спектрі пристроїв, що розширює вашу аудиторію.

-Швидкість розробки: Unity має широкий набір інструментів і ресурсів, які полегшують процес розробки. Ви можете швидко створювати і тестувати різні елементи гри, такі як рівні, персонажі та ігрові механіки.

-Активна спільнота та ресурси: Unity має велику та активну спільноту розробників, яка може допомогти вам з вирішенням проблем та навчанням нових навичок. Крім того, на Unity Asset Store є безліч готових ресурсів, які можуть бути корисними для розробки гри rogue-lite, таких як арт-асети, звукові ефекти та скрипти.

-Можливості графіки та анімації: Unity надає розробникам можливості створювати вражаючі візуальні ефекти та анімації. Це дозволяє вам створювати динамічні та захоплюючі візуальні ефекти, які додають глибину та настрій вашій грі rogue-lite.

-Підтримка штучного інтелекту: Unity має різноманітні ресурси та плагіни для створення складного і швидкого штучного інтелекту (ШІ) для вашої гри. Це дозволяє створювати інтелектуальних ворогів, розумний рівень генерації або систему штучного інтелекту для взаємодії з гравцем.

Враховуючи ці фактори, Unity добре підходить для розробки гри жанру rogue-lite, дозволяючи створювати вражаючі та захоплюючі ігри з різноманітними механіками та візуальними ефектами.

2.2 Алгоритм реалізаці проекту

Комп’ютерні ігри створюються в три основні етапи: проектування, розробка та видання, а також підтримка.

Починаючи з етапу проектування, визначається мета гри та ресурси, необхідні для її розробки. Визначення мети гри передбачає визначення того, що саме може привернути увагу гравців і спонукати їх грати. Наприклад, можливість грати в уявні ролі є основною концепцією рольових ігор (RPG), тоді як можливість брати участь у бойових діях є основною концепцією шутерів. Основна ідея гри та цілі тісно пов’язані з її

жанром. Таким чином, етап проектування є важливим для визначення концепції та основних параметрів гри.

Після того, як обирається жанр і ідея гри, наступним кроком є вибір сеттингу. Налаштування включає місце, час і умови події. Важливо пам'ятати, що вибір сеттингу може вплинути на процес створення сценарію гри, тому його краще вибирати заздалегідь, щоб враховувати інтереси цільової аудиторії. Наприклад, сценарій у фантастичному світі може вимагати використання особливих технологій і персонажів, тоді як сценарій у реальному світі може бути заснований на подіях або місцях у минулому.

Програмний код і ігровий рушій є компонентами, необхідним для створення гри жанру rogue-lite. Завдяки вибору конкретних засобів швидкість розробки та функціональність готового продукту визначаються. Наприклад, для браузерних ігор можна використовувати мови програмування, такі як JavaScript або HTML5 Canvas, тоді як для комп'ютерних ігор найкраще використовувати мови програмування C++ або C# разом із популярними ігровими движками, такими як Unity або Unreal Engine. Можливості гри та продуктивність розробки залежать від вибору платформи та мови програмування.

Відтворення фізики об'єктів, правила візуалізації графіки та інші технічні аспекти гри покладаються на ігровий рушій. Основними критеріями при виборі ігрового рушія є доступність і підтримка мов програмування. Наприклад, Unity — це ігровий рушій, який дозволяє використовувати мову програмування C# для створення ігор. Також Unitu має безкоштовну версію.

Розробка починається після визначення мети гри та вибору інструментів розробки. Це найбільш складний етап, який включає багато

етапів, щоб створити продукт, який працює. Спочатку визначається сюжет і механіка гри на основі цілей гри, а потім встановлюються всі об’єкти та правила взаємодії гравця. Паралельно з цим розробляється історія гри, яка визначає, чим гравець цікавиться. Сюжет може бути представлений як режисерський або літературний сценарій. На цій стадії також починається розробка графіки та дизайну гри. Концепт-арт, які є початковими ідеями для зовнішнього вигляду гри та персонажів, створюються на основі сюжету та попереднього дизайну.

Після цього наступним кроком є створення першої версії рівня. Це зазвичай проста локація з мінімальним набором необхідних речей. Тест проходимості проводиться за допомогою цієї версії; після успішного проходження тесту рівень поступово заповнюється іншими об’єктами.

Незабаром після створення перших рівнів починається розробка першої альфа-версії гри. Це дозволяє розробникам тестувати основні механіки гри та визначити, наскільки вони відповідають вимогам. Зазвичай об’єкти в альфа-версіях гри представлені як абстрактні об’єкти або навіть не мають текстури.

Якщо альфа-версія гри проходить успішне тестування, наступним кроком є розробка механіки та об’єктів гри. Наразі проводиться внесення перших сюжетних елементів, таких як відеоролики, сюжетні діалоги та кат-сцени, а також доопрацювання рівнів і механіки. Крім того, виправляються помилки та дефекти, виявлені під час тестування альфа-версії коду гри.

Після цього розробляється другий прототип гри, або бета-версія. Бета-версія була створена для тестування гри на проблеми та помилки. Насправді бета-версія повністю готова до гри, за винятком незначних деталей, які не впливають на геймплей. Все перевіряється під час

тестування бета-версії. Часто, особливо для мультиплеєрних ігор, до тестування запрошуються звичайні гравці. Це полегшує процес і зменшує навантаження на розробників.

Гра переходить до остаточного доопрацювання та виправлення критичних помилок, якщо вона успішно проходить бета-тестування. Після цього відбувається збірка завершеної версії гри та її реліз. Підтримка продукту продовжується після випуску гри, включаючи випуск патчів для виправлення помилок у грі. Для подовження життєвого циклу гри також можуть бути випущені додаткові контент-пакети (DLC), які додають нові предмети або можливості для гравців. Таким чином, етапи, необхідні для розробки комп’ютерних ігор, мало відрізняються від етапів, необхідних для розробки будь-якого іншого програмного продукту.

2.3 Мета проекту

Мета проекту полягає в тому, щоб провести аналіз інструментів розробки комп’ютерних ігор, а потім створити комп’ютерну гру у жанрі rogue-lite.

Розробка продукту, який використовує основні характеристики, що відрізняють жанр rogue-lite, є головним завданням. Метою проекту також є знайомство потенційної цільової аудиторії з цим захоплюючим жанром гри, хоча він не найпоширеніший. Розробка гри дозволить отримати цінний досвід, який буде корисним для майбутніх ігрових проектів і в інших сферах. Досвід розробки гри з використанням конкретного рушія також прискорить виконання наступних проектів.

2.4 Функціонал проекту

Проектом буде комп'ютерна гра у жанрі rogue-lite.Основне призначення до проекту полягає в тому, щоб гра була простою та веселою. Основні механіки проекту наведено нижче.

1. Процедурна генерація рівнів:Алгоритм генерації використовується для створення різних рівнів (локацій) у грі. Це дозволяє створювати нові рівні кожного разу, коли вони починають гру.

2. Перманентна смерть: ключова особливість жанру rogue-lite полягає в тому, що якщо персонаж помирає, гравець повинен починати гру з початку. Оскільки кожне рішення, яке приймає гравець, має певний вплив, це посилює виклики та підвищує напруженість у грі.

3. Покрокова система: принцип гри повинен базуватися на ходах, тобто гравець повинен виконувати дії по черзі, а потім отримувати відповідь від середовища гри. Це дозволяє гравцеві ретельно обмірковувати свої рухи та стратегію, а також адаптуватися до змін, які відбуваються в грі.

4. Випадковість: Випадковість є важливою частиною rogue-lite. Випадкові елементи, такі як розташування ворогів, предметів, пасток тощо, повинні бути присутні в кожній грі. Це робить гру непередбачуваною та змушує гравців адаптуватися до нових обставин.

Крім основних вимог, до функціональної частини проекту можуть бути додані додаткові елементи. Ці елементи можуть включати різні типи ворогів, босів, загадок, квестів, систему збереження гри, можливість гри в мультиплеєрному режимі та багато іншого. Але для створення проекту основні вимоги мають пріоритет.

ВИСНОВОК ДО РОЗДІЛУ 2

У цій роботі було проведено аналіз інструментів розробки комп'ютерних ігор та розглянуто основні етапи розробки гри жанру rogue-lite. Проект мав на меті створення комп'ютерної гри у цьому жанрі, використовуючи основні характеристики, які відрізняють його від інших.

Першим кроком було визначено середовище, яке краще підходить для розробки гри з урахуванням специфіки жанру.Другим кроком було визначено мету проекту, яка включала розробку гри з урахуванням специфіки жанру, а також пізнавальну цінність для розробників та цільової аудиторії. Далі було розглянуто функціонал проекту, який описав основні механіки гри, такі як процедурна генерація рівнів, перманентна смерть, покрокова система та випадковість. Крім того, було відзначено, що до функціональної частини проекту можуть бути додані додаткові елементи для збагачення геймплею.

Загальною метою проекту було розробити гру, яка буде простою та веселою для гравців, водночас використовуючи ключові особливості жанру rogue-lite. Процес розробки було розкрито через етапи проектування, розробки, тестування та випуску гри, враховуючи потреби цільової аудиторії та розвиток продукту до його остаточної версії.

Ця робота не лише дозволила вирішити завдання зі створення комп'ютерної гри, але й надала цінний досвід у сфері розробки ігор та впровадження ключових аспектів жанру у практичному застосуванні. Результатом є гра, яка має потенціал привернути увагу гравців та стати популярною серед шанувальників комп'ютерних ігор.

3 РЕАЛІЗАЦІЯ ПРОЄКТУ

3.1 Графіка проекту

Однією з основних складових будь-якої комп'ютерної гри є графіка. Перед початком опису частин графічного оформлення, важливо розібратися в основних поняттях, що використовуються у геймдизайні.

Основні поннятя:

1. Спрайти та текстури:

- Використання спрайтів для представлення об'єктів, персонажів та інших елементів гри.

- Використання текстур для задання вигляду та стилю графічних елементів.

2. Анімація:

- Створення анімованих спрайтів для рухомих об'єктів, персонажів та інших анімованих елементів гри.

- Використання анімаційних засобів Unity для створення та керування анімаціями.

3. Ефекти та частинки:

- Додавання візуальних ефектів, таких як вибухи, вогні, дим та інші частинкові ефекти.

- Використання системи частинок Unity для створення та управління візуальними ефектами.

4. Інтерфейс користувача (UI):

- Розроблення графічного інтерфейсу для взаємодії з гравцем, включаючи кнопки, меню, панелі та інші елементи.

- Використання системи UI Unity для створення та керування інтерфейсом користувача.

5. Засоби редактора Unity:

- Використання різноманітних засобів редактора Unity для розміщення та налаштування графічних елементів.

- Використання попереднього перегляду та налаштування для візуалізації графіки прямо в середовищі розробки Unity.

6. Оптимізація графіки:

- Забезпеченняоптимальної продуктивності гри шляхом використання оптимізованих спрайтів, текстур та ефектів.

- Використання пакування текстур та інших засобів для зменшення розміру файлів та збільшення продуктивності.

7. Арт-дизайн та стиль:

- Розроблення стилю гри, включаючи кольорову палітру, образи персонажів, фони та інші візуальні елементи.

- Використання арт-дизайну для створення привабливого та консистентного вигляду гри.

Більша частина спрайтів для проєкту була взята з Asset Store. Це значно полегшило процес розробки, оскільки готові ресурси дозволили швидше створити візуальну частину гри. Використання вже готових асетів також допомогло зекономити час і зусилля, які могли бути витрачені на створення оригінальних спрайтів з нуля.

3.1.1 Спрайт меню.

Спрайт для меню гри (Рисунок 3.1.1) були створені використовуючи нейромережу Dream від WOMBO. Цей процес полягав у завантаженні опису гри в нейромережу, що змогла перетворити його на візуальні елементи.

Рисунок 3.1.1-Спрайт головного меню.

3.1.2 Спрайти головного героя

Спрайт головного героя (Рисунок 3.2.1) буде безіменний сталкер. До цього асету одразу додавалася базова анімація, яка включає рухи персонажа під час отримання ушкоджень та його атаки. Це дозволяє створити більш зрозумілий геймплей, де головний герой реагує на події у грі, роблячи бойові сцени більш зрозумілими для гравців.

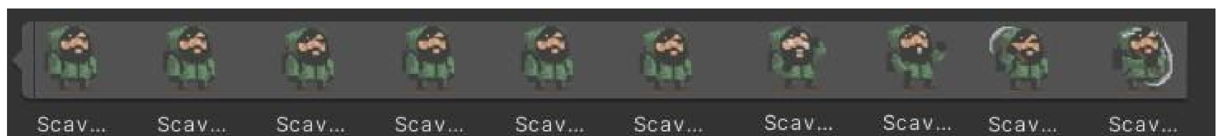


Рисунок 3.1.2-Спрайт головного героя

3.1.3 Спрайти ворогів

Спрайти ворогів (Рисунок 3.1.3), які будуть нападати на ігрового персонажа для ускладнення гри, включають два види: гулі та альгулі. Гулі – це вороги, які мають базові атаки та анімації. Альгулі, у свою чергу, є більш потужними і небезпечними ворогами. Вони мають вдосконалені атаки та анімації, що робить їх більш серйознішою загрозою для гравця.

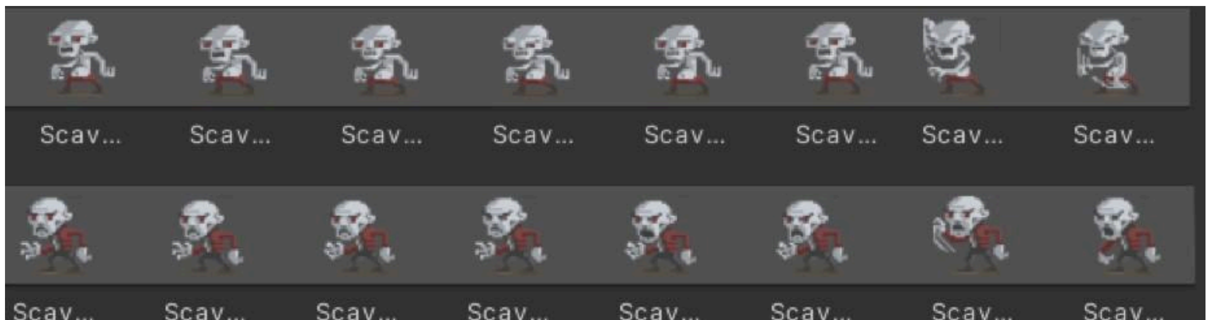


Рисунок 3.1.3-Спрайти ворогів

3.1.4 Спрайти інтерфейсу.

У грі було обрано графічні елементи для землі(Рисунок 3.1.4.1), перешкод і стін, які використовуються для візуального відображення об'єктів в грі. Спрайти землі відтворюють поверхню, по якій головний герой може вільно переміщатися.

Рисунок 3.1.4.1-Спрайти землі.

Спрайти перешкод(Рисунок 3.1.4.2) в грі використовуються для створення об'єктів, які персонаж не може пройти, але може їх знищити або перескочити.

Рисунок 3.1.4.2-Спрайти перешкоджень.

Спрайти стін (Рисунок 3.1.4.3) виступають у ролі меж гри, обмежуючи область переміщення персонажа. Ці графічні об'єкти є непрохідними для персонажа, тобто він не може подолати їх або пройти через них.

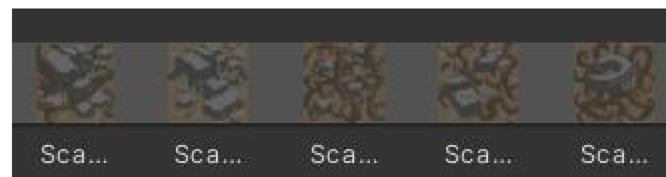


Рисунок 3.1.4.3-Спрайт стін

3.1.5 Спрайти їжі

Спрайт їжі (Рисунок 3.1.6) в грі функціонує як засіб набору очок що дозволяє гравцеві продовжувати гру та збільшувати свій рахунок.

Рисунок 3.1.5-Спрайт їжі

3.2 Анімації

Анімації в іграх - це ключовий аспект, який додає реалізм та життєвість до віртуального світу. Вони відображають рухи, дії та взаємодію персонажів і об'єктів у грі.

У Unity, для створення анімацій використовується система Mecanim, яка дозволяє легко створювати та управляти анімаціями. Основними компонентами системи Mecanim є Animator та Animation Controller.

Animator - це компонент, який призначений для управління анімаціями об'єктів. Він включає в себе різні параметри, які визначають поточний стан анімації, такі як рух, обертання, зміна спрайтів тощо(Рисунок 3.2.1).

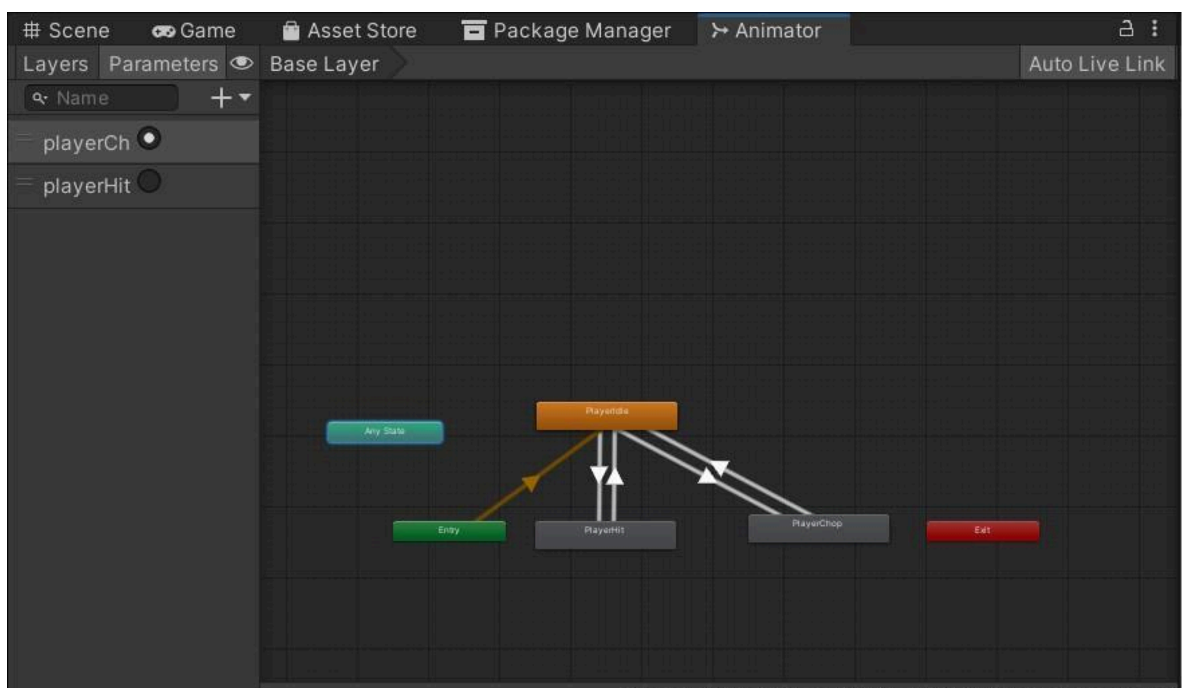


Рисунок 3.2.1-Вікно Animator

Animation Controller - це візуальне представлення станів та переходів між ними. Він дозволяє програмувати поведінку анімацій і переходів між ними за допомогою різних умов та подій(Рисунок 3.2.2).

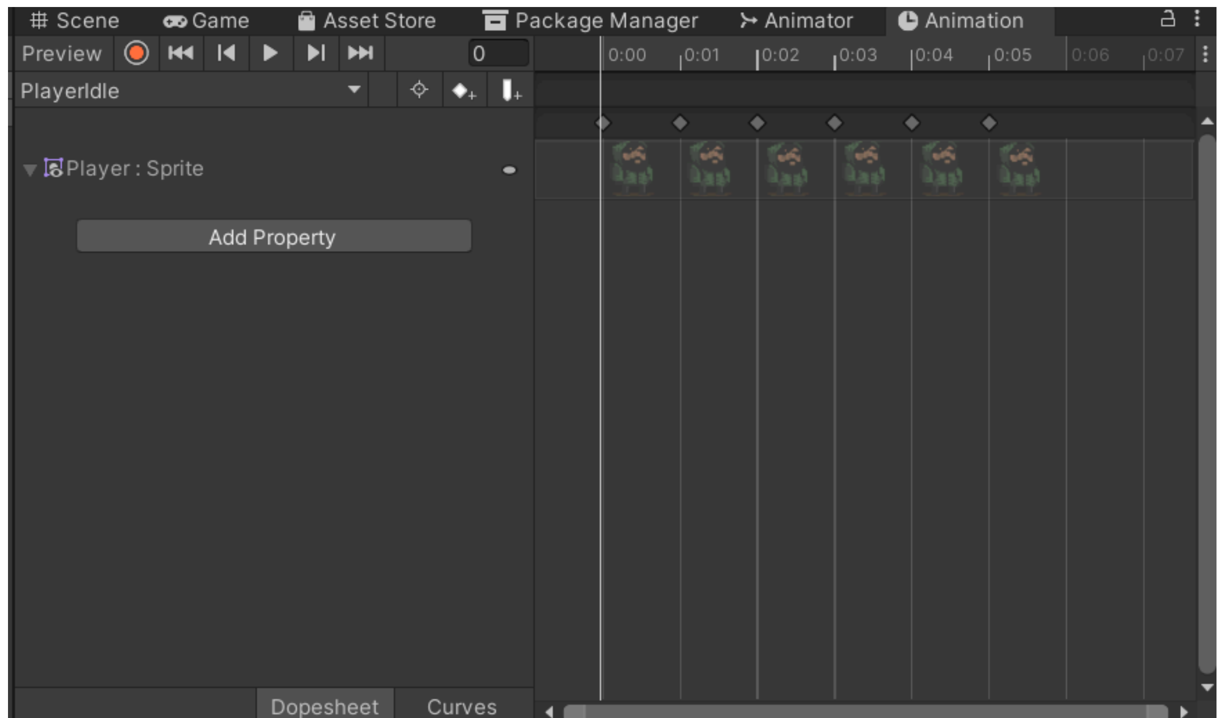


Рисунок 3.2.2-Вікно Animation Controller

Для створення анімацій у Unity можна використовувати спеціальний редактор анімацій, який дозволяє створювати та редагувати анімації безпосередньо у середовищі розробки. Також можна імпортувати готові анімації з зовнішніх програм, таких як Blender або Maya.

3.3 Аудіофайли

Саундтрек у Unity є важливим аспектом, який значно впливає на атмосферу гри та загальний ігровий досвід. В Unity саундтрек може включати в себе музику, звукові ефекти та голосові записи, які допомагають створити емоційний зв'язок з гравцем і підсилюють враження від гри.

Основні компоненти роботи з аудіо в Unity:

1. AudioSource: Це компонент, який додається до об'єкта в грі і відповідає за відтворення звуку. Він містить налаштування для вибору аудіокліпу, регулювання гучності, панорамування та просторового звуку. Кожен об'єкт, який повинен відтворювати звук, потребує наявності компонента AudioSource.

2. AudioClip: Це файл звукового запису, який відтворюється за допомогою AudioSource. Unity підтримує різні формати аудіофайлів, такі як WAV, MP3, OGG. AudioClip додається до проекту через імпорт файлу у Unity.

Рисунок 3.3.1-Вікно AudioSource та AudioClip

3. AudioMixer: Це потужний інструмент для обробки звуку, який дозволяє змішувати кілька аудіоджерел, застосовувати ефекти, регулювати рівні гучності та створювати аудіогрупи. AudioMixer допомагає

створювати більш складні звукові сцени та керувати загальним звучанням гри(Рисунок 3.3.2).

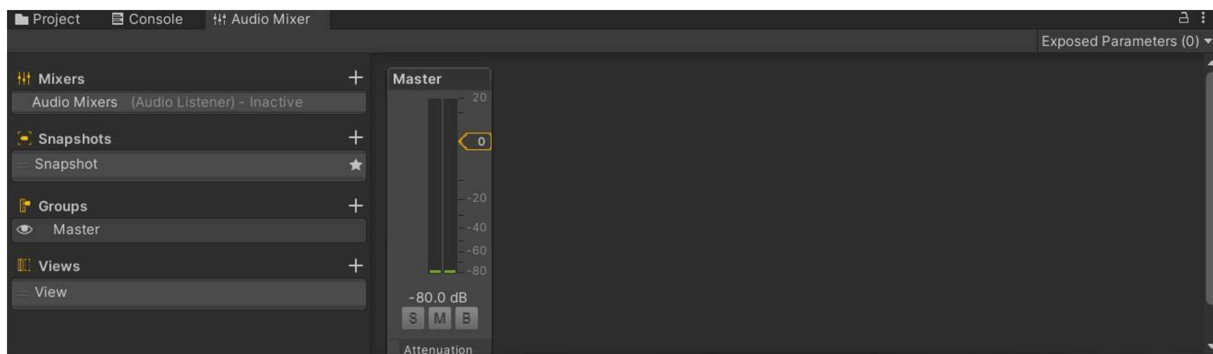


Рисунок 3.3.2 AudioMixer

4. Імпорт аудіофайлів: Спочатку необхідно імпортувати потрібні аудіофайли в проект. Це можна зробити шляхом перетягування файлів у папку Assets в Unity(Рисунок 3.3.3).

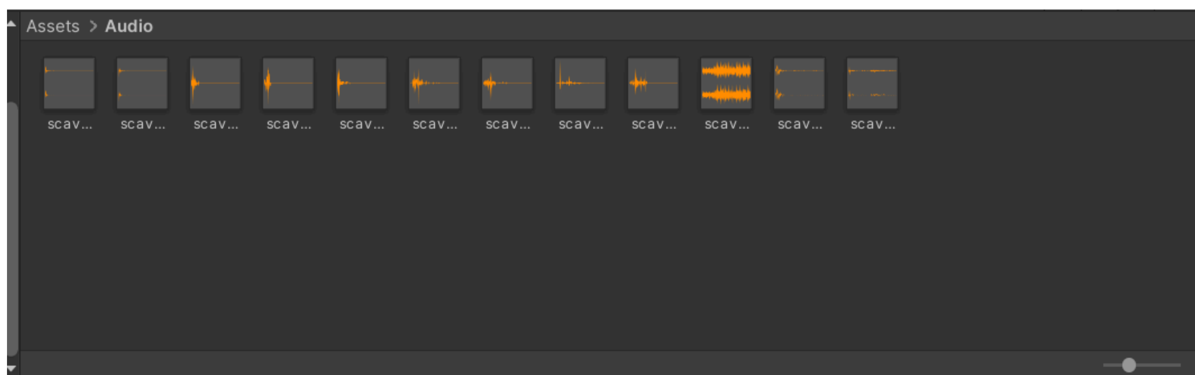


Рисунок 3.3.3-Імпортовані аудіофайли

3.4 Префаби

Префаби(Prefabs) в Unity представляють собою зручний механізм для створення та використання готових об'єктів або груп об'єктів у проекті. Їх використання спрощує процес розробки, оскільки дозволяє створювати складні об'єкти шляхом комбінування простіших компонентів без необхідності постійного повторювання однакових дій.

При розробці власної гри або додатку, студії Unity часто використовують префаби для швидкого створення та редагування ігрових об'єктів. Наприклад, префаб дерева може містити всі необхідні компоненти, такі як модель, текстури, колізії тощо. Після створення такого префабу, його можна легко додавати у сцену гри безпосередньо з бібліотеки префабів Unity.

Однією з головних переваг використання префабів є можливість автоматичної синхронізації змін. Якщо ви змінюєте один префаб, ці зміни автоматично відображаються у всіх екземплярах цього префабу, що використовуються у вашій грі. Це забезпечує консистентність та ефективність управління вмістом вашого проекту.

Загальна ідея полягає в тому, що префаби роблять процес розробки більш ефективним та простим, дозволяючи швидко створювати, редагувати та використовувати готові компоненти.

У рамках проекту було створено кілька префабів, таких як земля, стіни, вихід, їжа та перешкоди. Окрім цього, для полегшення роботи та забезпечення більшого зручності в грі, до префабів були додані компоненти гравця, ворогів, аудіо-супроводу та інтерфейсу користувача (UI). Це дозволяє легко використовувати ці компоненти та інтегрувати їх у гру, забезпечуючи більше можливостей для розширення та покращення розробки(Рисунок 3.2.1).

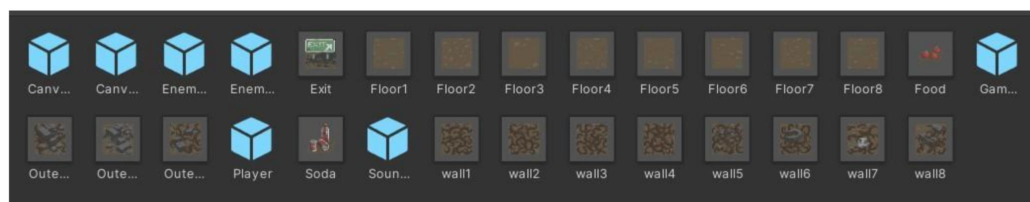


Рисунок 3.4-Прафаби проекту

Коли мова йде про ігри з видом зверху, це передбачає розробку фізичних характеристик платформ, по яких буде рухатися головний

персонаж. Цей процес передбачає додавання до кожної платформи спеціального компонента «Box Collider 2D», який визначає форму кожної платформи для взаємодії з іншими об’єктами.

Фізичні характеристики гравця та ворогів повинні бути доступними, щоб він міг взаємодіяти з навколишнім середовищем. Для досягнення цього ми додаємо до персонажа компоненти «Capsule Collider 2D», який визначає його форму у вигляді капсули, і компонент «Rigidbody 2D», який дозволяє об’єкту рухатися відповідно до фізичних законів.

Таким чином, ми включаємо вищезазначені компоненти в процес створення префаб-платформ для гри, щоб надати їм фізичні характеристики. Таким чином, відповідні фізичні характеристики будь-якої нової платформи, яку ми створимо на основі цього префабу, будуть автоматично забезпечені.

3.5 Меню гри

Початкове меню гри визначає перше враження гравця про гру.Необхідно створити привабливий, простий і зручний інтерфейс, який привертає увагу гравця.

Розробка меню до гри в Unity включає створення інтерфейсу користувача (UI) за допомогою Unity UI System.

По-перше, потрібно створити нову сцену. Для кращої орієнтації в проєкті, краще дати їй назву "MainMenu". Далі необхідно створити компонент Canvas, який буде контейнером для всіх елементів UI. До нього додаємо спрайт меню та кнопки PlayButton і ExitButton.

По-друге, щоб полегшити орієнтування в Canvas, створимо об'єкт MainMenu, до якого додаємо кнопки та їхні спрайти(Рисунок 3.5.1).

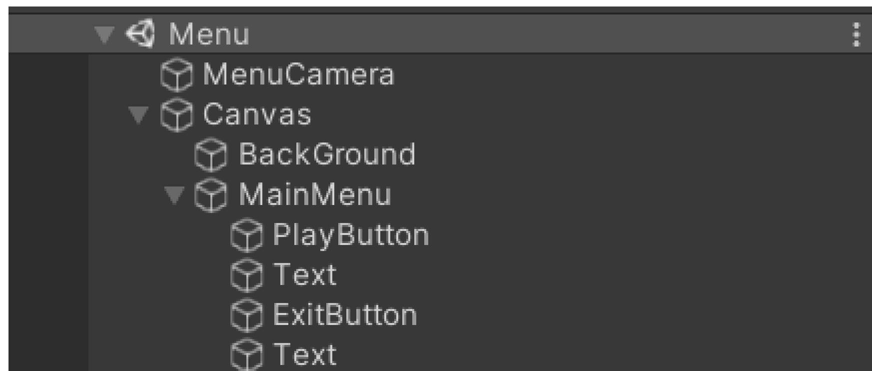


Рисунок 3.5.1-Ієрархія меню

По-третє, починаємо розробляти скрипт для головного меню, щоб воно почало функціонувати. Створюємо скрипт з назвою MenuController, в якому реалізуємо функціонал кнопок(Рисунок 3.5.2).

```
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class MenuController : MonoBehaviour
{
    // Кнопки
    public Button PlayButton;    // Кнопка "Грати"
    public Button ExitButton;    // Кнопка "Вийти"

    void Start()
    {
        // Призначення функцій для кнопок
        PlayButton.onClick.AddListener(PlayGame);
        ExitButton.onClick.AddListener(ExitGame);
    }

    // Функція для початку гри
    public void PlayGame()
    {
        // Логуємо повідомлення про завантаження сцени геймплею
        Debug.Log("Loading Gameplay scene...");
        // Завантажуємо сцену геймплею
        SceneManager.LoadScene("Gameplay");
    }

    // Функція для виходу з гри
    public void ExitGame()
    {
        // Логуємо повідомлення про вихід з гри
        Debug.Log("Exiting game...");
        // Завершуємо додаток
        Application.Quit();

        // Цей рядок потрібний тільки для редактора, щоб переконатися, що скрипт працює
#if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
#endif
    }
}
```

Рисунок 3.5.2-Скрипт реалізації кнопок.

3.6 Рух об'єктів

Написання скрипту для руху є наступним кроком у розробці ігрового персонажа.Зараз встановлюємо змінні(Рисунок 3.6.1):

```
public abstract class MovingObject : MonoBehaviour
{
    public float moveTime = 0.1f;           // Час, який потрібно для переміщення об'єкта, в секундах.
    public LayerMask blockingLayer;         // Шар, на якому буде перевірятися колізія.

    private BoxCollider2D boxCollider;      // Компонент BoxCollider2D, прикріплений до цього об'єкта.
    private Rigidbody2D rb2D;               // Компонент Rigidbody2D, прикріплений до цього об'єкта.
    private float inverseMoveTime;          // Використовується для більш ефективного переміщення.
    private bool isMoving;                  // Чи переміщується об'єкт в даний момент.
```

Рисунок 3.6.1-Зміні для скрипта MovingObject

Змінна moveTime визначає час, який потрібно об'єкту для переміщення з однієї позиції на іншу. Вона вимірюється в секундах і використовується для контролю швидкості руху об'єкта. Наприклад, якщо moveTime дорівнює 0.1, то об'єкт буде переміщатися з однієї позиції на іншу протягом 0.1 секунди.

Змінна blockingLayer використовується для визначення шару (layer), на якому буде перевірятися наявність колізій або перешкод. У Unity можна налаштувати різні шари, які взаємодіють з різними типами об'єктів. blockingLayer вказує, з якими саме шарами об'єкт буде взаємодіяти при спробі руху чи колізії. Наприклад, якщо об'єкт знаходиться на шарі "Стіна", то встановлення blockingLayer на шар "Стіна" дозволить об'єкту взаємодіяти зі стінами під час руху або колізій.

Змінна boxCollide представляє посилання на компонент BoxCollider2D, який прикріплений до об'єкта. BoxCollider2D визначає область простору, яка взаємодіє з іншими об'єктами у грі. Вона використовується для визначення між об'єкта, з якими може взаємодіяти фізика.

Змінна rb2D представляє посилання на компонент Rigidbody2D, який прикріплений до об'єкта.Rigidbody2D відповідає за фізичну поведінку об'єкта, таку як рух, сили, зіткнення і т.д. Вона використовується для керування фізикою об'єкта під час його переміщення.

Змінна inverseMoveTime використовується для підвищення ефективності обчислення часу переміщення. Вона представляє обернений час переміщення (1/moveTime) і використовується для оптимізації обчислень при переміщенні об'єкта.

Змінна isMoving вказує на те, чи знаходиться об'єкт у процесі руху в даний момент. Вона використовується для контролю над тим, чи можна почати новий рух об'єкта, поки він вже знаходиться у русі.

Оскільки гравець може переміщатися на будь-яку відстань в просторі, розробник повинен встановити певні обмеження. Швидкість руху гравця визначається за допомогою змінної moveTime, що вказує на час, який потрібний для пересування гравця на одну одиницю в просторі. Ця змінна використовується для обчислення швидкості руху об'єкта у методі SmoothMovement. Використовуючи цю швидкість у методі SmoothMovement, код забезпечує плавне переміщення об'єкта з поточної позиції до цільової позиції. Це досягається за допомогою корутину SmoothMovement, який виконується у циклі, зміщуючи об'єкт на певну відстань у кожній ітерації. Цей процес триває, поки залишкова відстань до цільової позиції не стає дуже малою (float.Epsilon), і тоді корутин завершується. Такий підхід забезпечує гармонійне і плавне переміщення об'єкта в грі.

Отже, загальний алгоритм руху об'єкта можна описати таким чином:

1. Метод Move перевіряє можливість переміщення об'єкта у вказаному напрямку (xDir, yDir). Він використовує Physics2D.Linecast для

перевірки наявності перешкод на шляху і повертає true, якщо рух успішний.

2. Якщо рух успішний, викликається корутин SmoothMovement, який починає плавне переміщення об'єкта до цільової позиції.

3. У корутині SmoothMovement об'єкт зміщується крок за кроком в напрямку цільової позиції. Для цього використовуються методи Vector3.MoveTowards та Rigidbody2D.MovePosition.

4. Рух завершується, коли об'єкт досягає цільової позиції.

3.7 Механіка смерті та життя

У проекті за життя відповідає "food", яке ініціалізується у змінній "playerFoodPoints" в класі "GameManager" у методі "Start" за допомогою рядка "food = GameManager.instance.playerFoodPoints;". Це значення оновлюється при кожному зміщенні персонажа(Рисунок 3.7.1).

```
//Start перевизначає функцію Start з MovingObject
protected override void Start()
{
    //Отримуємо посилання на компонент аніматора гравця
    animator = GetComponent<Animator>();

    //Отримуємо поточну кількість очок їжі, збережених в GameManager.instance між рівнями.
    food = GameManager.instance.playerFoodPoints;

    //Встановлюємо текст foodText для відображення поточної кількості очок їжі гравця.
    foodText.text = "Food: " + food;

    //Викликаємо функцію Start базового класу MovingObject.
    base.Start();
}
```

Рисунок 3.7.1-ініціалізація playerFoodPoints

Для поповнення життя у проекті герою потрібно підбирати "Soda" та "Food", які збільшують кількість очок персонажа(Рисунок 3.7.2).

```
else if (other.tag == "Food")
{
    //Додаємо pointsPerFood до поточного загального рахунку їжі гравця.
    food += pointsPerFood;

    //Оновлюємо текст foodText, щоб відображати поточний загальний рахунок і повідомити гравця, що вони заробили очки.
    foodText.text = "+" + pointsPerFood + " Food: " + food;

    //Викликаємо функцію RandomizeSfx з SoundManager і передаємо два звуки їжі для вибору між ними для відтворення звукового ефекту їжі.
    SoundManager.instance.RandomizeSfx(eatSound1, eatSound2);

    //Вимикаємо об'єкт їжі, з яким зіткнувся гравець.
    other.gameObject.SetActive(false);
}
```

Рисунок 3.7.2-Рахунок їжі

Смерть героя настає у момент, коли його показники їжі падають до нуля або нижче. Постійно проводиться перевірка на цей рахунок. Якщо це стається, викликається метод "CheckIfGameOver", який відповідає за відтворення звуку смерті героя, припиняє фонову музику та викликає метод "GameOver"(Рисунок 3.7.3).

```
private void CheckIfGameOver()
{
    //Перевіряємо, чи загальна кількість очків їжі менше або дорівнює нулю.
    if (food <= 0)
    {
        //Викликаємо функцію PlaySingle з SoundManager та передаємо їй gameOverSound як аудіо кліп для відтворення.
        SoundManager.instance.PlaySingle(gameOverSound);

        //Зупиняємо фонову музику.
        SoundManager.instance.musicSource.Stop();

        //Викликаємо функцію GameOver з GameManager.
        GameManager.instance.GameOver();
    }
}
```

Рисунок 3.7.3-Метод CheckIfGameOver

3.8 Вороги

Метод SetupScene класу BoardManager відповідає за генерацію ворогів. Він визначає їх можливу кількість та позицію(Рисунок 3.8.1).

```
// SetupScene ініціалізує наш рівень та викликає попередні функції для розміщення ігрової дошки
public void SetupScene(int level)
{
    // Створює зовнішні стіни та підлогу.
    BoardSetup();

    // Скидаємо наш список gridpositions.
    InitialiseList();

    // Інстанціюємо випадкову кількість стінних плиток на основі мінімального та максимального значень, на випадкових позиціях.
    LayoutObjectAtRandom(wallTiles, wallCount.minimum, wallCount.maximum);

    // Інстанціюємо випадкову кількість плиток їжі на основі мінімального та максимального значень, на випадкових позиціях.
    LayoutObjectAtRandom(foodTiles, foodCount.minimum, foodCount.maximum);

    // Визначаємо кількість ворогів на основі поточного номеру рівня, на основі логарифмічної прогресії
    int enemyCount = (int)Mathf.Log(level, 2f);

    // Інстанціюємо випадкову кількість ворогів на основі мінімального та максимального значень, на випадкових позиціях.
    LayoutObjectAtRandom(enemyTiles, enemyCount, enemyCount);

    // Інстанціюємо плитку виходу в верхньому правому куті нашої ігрової дошки
    Instantiate(exit, new Vector3(columns - 1, rows - 1, 0f), Quaternion.identity);
}
```

Рисунок 3.8.1-реалізація методу SetupScence

Визначаємо кількість ворогів на основі поточного номера рівня, використовуючи логарифмічну прогресію: int enemyCount= (int)Mathf.Log(level, 2f). Функція Mathf.Log обчислює логарифм числа level за основою 2, а int приводить результат до цілого числа.

Потім метод LayoutObjectAtRandom обирає випадкові позиції зі списку доступних gridPositions і розміщує на цих позиціях ворогів зі списку enemyTiles. Параметр enemyCount обмежує їх кількість.

3.9 Генерація рівня

В цьому коді випадково генерується рівень. Метод SetupScene викликає кілька функцій, щоб розмістити об’єкти (наприклад, стіни, їжу, ворогів тощо) на ігровому полі.

Генерація рівня у даному коді виконується у кілька кроків(Рисунок 3.9.1).

```
public void SetupScene(int level)
{
    // Створює зовнішні стіни та підлогу.
    BoardSetup();

    // Скидаємо наш список gridpositions.
    InitialiseList();

    // Інстанціюємо випадкову кількість стінних плиток на основі мінімального та максимального значень, на випадкових позиціях.
    LayoutObjectAtRandom(wallTiles, wallCount.minimum, wallCount.maximum);

    // Інстанціюємо випадкову кількість плиток їжі на основі мінімального та максимального значень, на випадкових позиціях.
    LayoutObjectAtRandom(foodTiles, foodCount.minimum, foodCount.maximum);

    // Визначаємо кількість ворогів на основі поточного номеру рівня, на основі логарифмічної прогресії
    int enemyCount = (int)Mathf.Log(level, 2f);

    // Інстанціюємо випадкову кількість ворогів на основі мінімального та максимального значень, на випадкових позиціях.
    LayoutObjectAtRandom(enemyTiles, enemyCount, enemyCount);

    // Інстанціюємо плитку виходу в верхньому правому куті нашої ігрової дошки
    Instantiate(exit, new Vector3(columns - 1, rows - 1, 0f), Quaternion.identity);
}
```

Рисунок 3.9.1-Генерація рівня

1. Метод BoardSetup() відповідає за створення зовнішніх стін і підлоги для ігрового рівня. Це основа, на якій будуть розміщуватися інші об'єкти.

2. Метод InitialiseList() скидає та ініціалізує список gridPositions, який містить усі доступні позиції на ігровій дошці, де можуть бути розміщені об'єкти.

3. Метод LayoutObjectAtRandom розміщує випадкову кількість стінних плиток (wallTiles) на основі мінімального (wallCount.minimum) та максимального (wallCount.maximum) значень. Позиції для стін вибираються випадковим чином зі списку gridPositions.

4. LayoutObjectAtRandom(foodTiles,foodCount.minimum,foodCount .ma ximum);

Аналогічно до стін, цей метод розміщує випадкову кількість плиток їжі (foodTiles) на випадкових позиціях, визначених на основі мінімального (foodCount.minimum) та максимального (foodCount.maximum) значень.

5. Instantiate(exit, new Vector3(columns - 1, rows - 1, 0f), Quaternion.identity); Нарешті, плитка виходу (exit) розміщується в

верхньому правому куті ігрової дошки (координати (columns - 1, rows - 1)). Quaternion.identity означає, що об'єкт не має обертання.

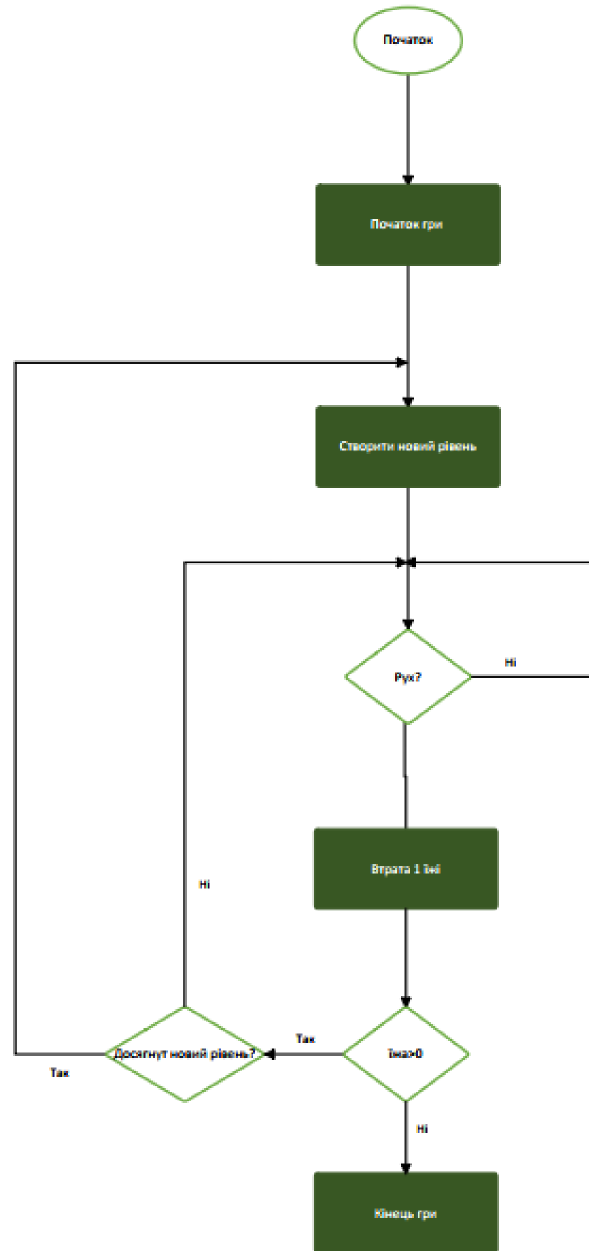
У результаті, методи та кроки забезпечують генерацію рівня зі стінами, їжею, ворогами та виходом у відповідних позиціях, роблячи кожен рівень унікальним і викликовим для гравців.

3.10 Блок-схема алгоритму гри

Блок-схема пояснює процес гри, включаючи створення рівнів і управління їжею.

Створення початкових умов є першим кроком до початку гри. Після цього програма визначає, чи є рух у грі. Гравець втрачає одиницю їжі, якщо він рухається. Крім того перевіряється на наявність їжі. Якщо це не так, гра завершується. Якщо це так, це означає, що досягнуто нового рівня. Це означає, що створюється новий рівень, і гра продовжується. Якщо це не так, гравець продовжує перебувати на поточному рівні. До того, як ви досягнете нового рівня або досягнете кінцевого стану гри, цей процес повторюється(Блок-схема 3.10).

Блок-схема 3.10- алгоритму гри



ВИСНОВОК ДО РОЗДІЛУ 3

Цей розділ повністю присвячений виконанню теми проекту, яка є двовимірною rogue-lite грою на Unity. Спочатку було проаналізовано основні поняття графічної складової, такі як спрайти і префаби, які використовуються в грі. Прототип розпочався з використання графічних елементів, таких як префаби та спрайти, для створення грального світу. У ньому описано основні функції, які використовувалися для роботи з графікою.

Після аналізу функцій компонентів ігрового рушія, які надають фізичні властивості об'єктам, було розпочато написання коду. Спочатку було створено меню гри. Потім було проведено перевірку функціональності компонентів ігрового рушія, які надають об'єктам фізичні характеристики, і розпочато написання коду. На початку було створено модель героя гри та написані скрипти для його руху. Були представлені основні класи, функції та методи, які були використані для оживлення героя. Крім того, була представлена схема реалізації анімації головного героя під час руху.

Також було детально описано створення ворогів героя та реалізація штучного інтелекту. Були визначені обставини смерті героя. Розроблено простий графічний інтерфейс, який відображає кількість днів і життя. Були розроблені відповідні скрипти для запуску цього інтерфейсу.

У наступному кроці реалізації розглядається генерація ворогів і рівнів у стилі roguelike. Були розглянуті різні методи створення ворожих персонажів і різноманітних типів ворогів, кожен з яких мав свою унікальну особливість. Написані скрипти для динамічної генерації рівнів, включаючи створення різних кімнат, коридорів і розташування ворогів у цих зонах.

Крім того, була введена система смерті гравця. Гра завершується, коли гравець помирає. Створені скрипти для обробки ситуації смерті гравця включають анімацію, звукові ефекти та перехід до екрану завершення гри.

ВИСНОВОК

Робота складається з трьох розділів. Перший містить загальну інформацію про комп’ютерні ігри та класифікацію їх жанрів. Крім того, було проаналізовано основні переваги та недоліки поточних прототипів.

Другий розділ містить опис жанру, який було обрано для проекту. Він також аналізує існуючі методи та функції розробки ігор, а також дає обґрунтування для вибору конкретного ігрового рушія для розробки. Крім того, в розділі представлено повний алгоритм створення гри, аналіз можливої аудиторії гравців і загальну актуальність проекту. Також сформульовано мету розробки та функціонал гри, до якого має відповідати гра.

У третьому розділі детально описано весь процес завершення проекту. Починаючи з визначення основних понять щодо візуального складового проекту, далі розглядаються основні графічні компоненти та компоненти, які використовувалися під час розробки гри. Опис кожного етапу розробки також міститься, а також важливі методи та класи, які були використані під час написання коду. Крім того, розглядається створення меню початку та завершення гри, а також як його реалізувати.

Завершальним результатом роботи над проектом є робочий прототип гри в жанрі Rogue-lite, яка не вимагає надзвичайно складних технічних характеристик. Прототип має просте меню, простий інтерфейс, мету гри та умови для її завершення. Надалі цей прототип гри може бути доповнений новими елементами та деталями, а також офіційно випущений як повноцінна гра на основі певного веб-ресурсу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. "Unity in Action: Multiplatform Game Development in C#" - Джо Хокінс.
2. "Unity 2021 Cookbook: Over 140 recipes to take your Unity game development skills to the next level" - Мэттью Джонсон, Джеймс А. Генрі
3. "Game Programming Patterns" - Роберт Найстром
4. Жанр відеоігор-https://uk.wikipedia.org/wiki/%D0%96%D0%B0%D0%BD%D1%80%D0%B8_%D0%B2%D1%96%D0%B4%D0%B5%D0%BE%D1%96%D0%B3%D0%BE%D1%80
5. Wombo-https://www.wombo.ai/
6. Gamedev-http://gamedev.net/
7. Asset store-https://assetstore.unity.com/
8. Steam store-https://store.steampowered.com/
9. SteamDB-https://steamdb.info/
10. Unity-https://learn.unity.com/
11. Godot-https://godotengine.org/
12. Libgdx-https://libgdx.com/

ДОДАТКИ

ДОДАТОК ІНСТРУКЦІЯ КОРИСТУВАЧА

Ласкаво просимо до гри "Wanderer's End". У цій грі ви керуватимете безіменним сталкером, який досліджує небезпечні території, бореться з ворогами та збирає їжу. Ваша мета – вижити в цьому ворожому світі та досягти виходу на кожному рівні.

Ласкаво просимо до гри "Wanderer's End". У цій грі ви керуватимете безіменним сталкером, який досліджує небезпечні території, бореться з ворогами та збирає корисні предмети. Ваша мета – вижити в цьому ворожому світі та досягти виходу на кожному рівні.

4.1 Системні вимоги

Мінімальні:

- ОС: Windows 7/8/10

-Процесор: Intel Core i3

- Оперативна пам'ять: 4 GB RAM

- Відеокарта: NVIDIA GeForce GTX 650

- Місце на диску:

2 GB Рекомендовані:

- ОС: Windows 10

- Процесор: Intel Core i5

- Оперативна пам'ять: 8 GB RAM

- Відеокарта NVIDIA GeForce GTX 960

- Місце на диску: 2 GB

4.2 Інтерфейс

На початку гри вас зустрічає головне меню. У цьому меню є дві основні кнопки: "Play" та "Exit". Натискання кнопки "Play" запускає гру, дозволяючи вам розпочати вашу пригоду. Кнопка "Exit" закриває гру, якщо ви вирішили вийти з неї.

Рисунок 4.2 Меню

4.3 Ігровий процес

1. Персонаж з'являється у лівому нижньому куті екрану. Для його переміщення використовуйте клавіші WASD або стрілки ←, →, ↑, ↓.

2. Щоб перейти на наступний рівень, вам потрібно дістатись до виходу "exit".

3. Слідкуйте за показником food якщо він спаде до нуля, гра завершиться.Коли персонаж рухається їда падає на 1 одиницю Використовуйте food та soda, щоб підвищити свій показник food.

4. Деякі перешкоди можна ламати за допомогою кірки. Для зламання використовуйте клавіші WASD або стрілки ←, →, ↑, ↓.

5. Будьте обережні з ворогами; якщо ви з ними зіткнетесь, вони можуть відібрати у вас очки їжі.

Рисунок 4.3-Інструкція до ігрового процессу

ДОДАТКИ ПРОГРАМНОГО КОДУ

1. MenuController

```
using UnityEngine;

using UnityEngine.SceneManagement;

using UnityEngine.UI;


public class MenuController : MonoBehaviour

{

    // Кнопки

    public Button PlayButton;    // Кнопка "Грати"

    public Button ExitButton;    // Кнопка

    "Вийти"


    void Start()

    {

        // Призначення функцій для кнопок

        PlayButton.onClick.AddListener(PlayGame);

        ExitButton.onClick.AddListener(ExitGame);

    }


    // Функція для початку
```

гри public void PlayGame()

```
    {
        // Логуємо повідомлення про завантаження сцени геймплею
        Debug.Log("Loading Gameplay scene...");
        // Завантажуємо сцену геймплею
        SceneManager.LoadScene("Gameplay");
    }

    // Функція для виходу з гри
    public void ExitGame()
    {
        // Логуємо повідомлення про вихід з гри
        Debug.Log("Exiting game...");
        // Завершуємо
        додаток
        Application.Quit();

            // Цей рядок потрібний тільки для редактора, щоб переконатися, що скрипт
працює
  #if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
  #endif
    }
```

```
}
```

2. GameManager

---

using UnityEngine;

using UnityEngine.SceneManagement;

using System.Collections;

namespace Completed

{

using System.Collections.Generic; //Дозволяє нам використовувати

списки. using UnityEngine.UI; //Дозволяє нам використовувати UI.

public class GameManager : MonoBehaviour

{

public float levelStartDelay = 2f; //Час очікування перед початком рівня, в секундах.

public float turnDelay = 0.1f; //Затримка між кожним ходом гравця.

public int playerFoodPoints = 100; //Початкове значення очок їжі гравця.

public static GameManager instance = null; //Статичний екземпляр GameManager, який дозволяє отримати до нього доступ з будь-якого іншого скрипта.

[HideInInspector] public bool playersTurn = true; //Булевий прапор для перевірки, чи зараз хід гравця, прихований в інспекторі, але доступний.

```
        private Text levelText;                        //Текст для відображення
поточного номера рівня.
        private GameObject levelImage;                 //Зображення для
блокування рівня під час налаштування, фон для levelText.
        private BoardManager boardScript;              //Зберігає посилання на
наш BoardManager, який налаштовує рівень.
        private int level = -2;                        //Поточний номер рівня, виражений
в грі як "День 1".
        private List<Enemy> enemies;                   //Список всіх ворожих
одиниць, використовується для видачі їм команд на рух.
        private bool enemiesMoving;                    //Булевий прапор для
перевірки, чи вороги рухаються.
        private bool doingSetup = true;                //Булевий прапор для
перевірки, чи ми налаштовуємо дошку, запобігає руху гравця під час налаштування.



        // Awake завжди викликається перед будь-якими функціями Start
        void Awake()
        {
            //Перевіряємо, чи екземпляр вже існує
            if (instance == null)
```

```
        //якщо ні, встановлюємо екземпляр на цей об'єкт
        instance = this;

    //Якщо екземпляр вже існує і це не цей об'єкт:
    else if (instance != this)
        //Тоді знищуємо цей об'єкт. Це забезпечує наш шаблон одиничності, тобто може існувати тільки один екземпляр GameManager.
        Destroy(gameObject);

    //Встановлюємо цей об'єкт так, щоб він не знищувався при перезавантаженні сцени
    DontDestroyOnLoad(gameObject);

    //Призначаємо enemies новому списку об'єктів Enemy.
    enemies = new List<Enemy>();

    //Отримуємо посилання на компонент прикріпленого скрипта BoardManager
    boardScript = GetComponent<BoardManager>();

    //Викликаємо функцію InitGame для ініціалізації першого рівня
    InitGame();
```

```
}


//це викликається тільки один раз, і параметр вказує, що це викликається тільки після завантаження сцени
//(інакше, наш зворотний виклик завантаження сцени викликався б при першому завантаженні, а ми цього не хочемо)
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterSceneLoad)]
static public void CallbackInitialization()
{
    //реєструємо зворотний виклик, щоб він викликався кожного разу при завантаженні сцени
    SceneManager.sceneLoaded += OnSceneLoaded;
}

//Це викликається кожного разу, коли завантажується сцена.
static private void OnSceneLoaded(Scene arg0, LoadSceneMode arg1)
{
    instance.level++;
    instance.InitGame();
}
```

```
//Ініціалізує гру для кожного рівня.
void InitGame()
{
    //Поки doingSetup є true, гравець не може рухатися, запобігаємо руху гравця під час відображення титульної картки.
    doingSetup = true;

    //Отримуємо посилання на наш image LevelImage, знайшовши його за іменем.
    levelImage = GameObject.Find("LevelImage");

    //Отримуємо посилання на текстовий компонент LevelText, знайшовши його за іменем і викликавши GetComponent.
    levelText = GameObject.Find("LevelText").GetComponent<Text>();

    //Встановлюємо текст levelText на рядок "Day" і додаємо поточний номер рівня.
    levelText.text = "Day " + level;

    //Активуємо levelImage, блокуючи вид гравця на ігрову дошку під час налаштування.
    levelImage.SetActive(true);
```

```
            //Викликаємо функцію HideLevelImage з затримкою в секундах рівній levelStartDelay.
            Invoke("HideLevelImage", levelStartDelay);

            //Очищаємо будь-які об'єкти Enemy в нашому списку, щоб підготуватися до наступного рівня.
            enemies.Clear();

            //Викликаємо функцію SetupScene скрипта BoardManager, передаємо їй поточний номер рівня.
            boardScript.SetupScene(level);

            playerFoodPoints = 100;
        }

        //Ховає чорне зображення, яке використовується між рівнями
        void HideLevelImage()
        {
            //Вимикає об'єкт levelImage.
            levelImage.SetActive(false);
```

```
            //Встановлює doingSetup в false, дозволяючи гравцю знову рухатися.
            doingSetup = false;
        }

        //Update викликається кожен кадр.
        void Update()
        {
            //Перевіряємо, що playersTurn, enemiesMoving або doingSetup наразі не є true.
            if (playersTurn || enemiesMoving || doingSetup)
                //Якщо будь-яке з цих значень true, виходимо і не починаємо MoveEnemies.
                return;

            //Починаємо рухати ворогів.
            StartCoroutine(MoveEnemies());
        }

        //Викликайте це, щоб додати переданого ворога до списку об'єктів Enemy.
        public void AddEnemyToList(Enemy script)
        {
```

```
            //Додаємо ворога до списку enemies.
            enemies.Add(script);
        }

        //GameOver викликається, коли гравець досягає 0 очок їжі
        public void GameOver()
        {
            //Встановлюємо levelText для відображення кількості пройдених рівнів та повідомлення про завершення гри
            levelText.text = "After " + level + " days, you starved.";

            //Включаємо об'єкт чорного фону levelImage.
            levelImage.SetActive(true);

            //Вимикаємо цей GameManager.
            enabled = false;
        }


        //Корутина для послідовного руху ворогів.
        IEnumerator MoveEnemies()
        {
```

```
//Поки enemiesMoving є true, гравець не може рухатися.
enemiesMoving = true;

//Очікуємо turnDelay секунд, за замовчуванням .1 (100 мс).
yield return new WaitForSeconds(turnDelay);

//Якщо вороги не з'явилися (наприклад, на першому рівні):
if (enemies.Count == 0)
{
    //Очікуємо turnDelay секунд між ходами, замінюючи затримку, викликану рухом ворогів, коли їх немає.
    yield return new WaitForSeconds(turnDelay);
}

//Перебираємо список об'єктів Enemy.
for (int i = 0; i < enemies.Count; i++)
{
    //Викликаємо функцію MoveEnemy ворога за індексом i в списку enemies.
    enemies[i].MoveEnemy();
```

```
                    //Очікуємо moveTime ворога перед переміщенням наступного ворога,
                    yield return new WaitForSeconds(enemies[i].moveTime);
                }

                //Після того, як вороги закінчили рухатися, встановлюємо playersTurn на true, щоб гравець міг рухатися.
                playersTurn = true;

                //Вороги закінчили рухатися, встановлюємо enemiesMoving на false.
                enemiesMoving = false;
            }
        }
    }
```

3. BoardManager

```
using
UnityEngine;
using System;
using System.Collections.Generic;      //Дозволяє використовувати списки.
using Random = UnityEngine.Random;      //Вказує Random використовувати генератор випадкових чисел у Unity.
```

```
namespace Completed
{

    public class BoardManager : MonoBehaviour
    {
        // Використання Serializable дозволяє вбудувати клас з підпропертісами у
інспектор.
        [Serializable]
        public class
        Count
        {
            public int minimum;            // Мінімальне значення для нашого класу
Count.
            public int maximum;           // Максимальне значення для нашого класу
Count.


            // Конструктор присвоєння.
            public Count(int min, int max)
            {
                minimum = min;
                maximum =
```

max;

```
        }
    }


    public int columns = 8;                                  // Кількість стовпців нашої ігрової дошки.
    public int rows = 8;                                     // Кількість рядків нашої ігрової дошки.
    public Count wallCount = new Count(5, 9);                // Нижня та верхня межа для випадкової кількості стін на рівень.
    public Count foodCount = new Count(1, 5);                // Нижня та верхня межа для випадкової кількості об'єктів їжі на рівень.
    public GameObject exit;                                  // Префаб для створення виходу.
    public GameObject[] floorTiles;                          // Масив префабів підлоги.
    public GameObject[] wallTiles;                           // Масив префабів стін.
    public GameObject[] foodTiles;                           // Масив префабів їжі.
    public GameObject[] enemyTiles;                          // Масив префабів ворогів.
    public GameObject[] outerWallTiles;                      // Масив префабів зовнішніх стін.

    private Transform boardHolder;                           // Змінна для зберігання посилання на трансформацію нашого об'єкта дошки.
```

```
}

private List<Vector3> gridPositions = new List<Vector3>();                //
Список можливих місць для розміщення плиток.



// Очищає наш список gridPositions та готує його для створення нової дошки.
void InitialiseList()
{
    // Очистити наш список gridPositions.
    gridPositions.Clear();

    // Проходить по вісі x (стовпці).
    for (int x = 1; x < columns - 1; x++)
    {
        // В межах кожного стовпця проходить по вісі y (рядки).
        for (int y = 1; y < rows - 1; y++)
        {
            // На кожному індексі додає новий Vector3 у наш список з
координатами x та y цієї позиції.
            gridPositions.Add(new Vector3(x, y, 0f));
        }
    }
}
```

```
// Налаштовує зовнішні стіни та підлогу (фон) гри.
void BoardSetup()
{
    // Створити дошку та призначити її трансформацію boardHolder.
    boardHolder = new GameObject("Board").transform;

    // Проходить по вісі x, починаючи з -1 (щоб заповнити кут) з плитками підлоги або зовнішніми плитками стіни.
    for (int x = -1; x < columns + 1; x++)
    {
        // Проходить по вісі y, починаючи з -1, щоб розмістити плитки підлоги або зовнішні плитки стіни.
        for (int y = -1; y < rows + 1; y++)
        {
            // Вибирає випадкову плитку з нашого масиву префабів підлоги та готується до її створення.
            GameObject toInstantiate = floorTiles[Random.Range(0, floorTiles.Length)];
```

// Перевіряє, чи поточна позиція знаходиться на краю дошки, якщо так, то вибирає випадковий зовнішній префаб стіни з нашого масиву зовнішніх плиток стіни.

if (x == -1 || x == columns || y == -1 || y == rows)

toInstantiate = outerWallTiles[Random.Range(0, outerWallTiles.Length)];

// Створює екземпляр GameObject за допомогою префаба, вибраного для toInstantiate, в позиції, що відповідає поточній позиції сітки у циклі, інстанціюємо його як GameObject.

GameObject instance =

Instantiate(toInstantiate, new Vector3(x, y, 0f), Quaternion.identity) as GameObject;

// Призначає батьківський об'єкт нашому новосформованому екземпляру

// Присвоюємо батьківський об'єкт нашому новоствореному екземпляру, це просто організаційний момент для уникнення забруднення ієрархії.

instance.transform.SetParent(boardHolder);

}

}

}

```
// RandomPosition повертає випадкову позицію зі списку gridPositions.
Vector3 RandomPosition()
{
    // Оголошуємо ціле випадкове число randomIndex, встановлюємо його значення в випадкове число між 0 та кількістю елементів у нашому списку gridPositions.
    int randomIndex = Random.Range(0, gridPositions.Count);

    // Оголошуємо змінну типу Vector3 з назвою randomPosition, встановлюємо її значення в запис з randomIndex з нашого списку gridPositions.
    Vector3 randomPosition = gridPositions[randomIndex];

    // Видаляємо запис з randomIndex зі списку, щоб його не можна було використовувати знову.
    gridPositions.RemoveAt(randomIndex);

    // Повертаємо випадково обрану позицію Vector3.
    return randomPosition;
}
```

```
        // LayoutObjectAtRandom приймає масив гральних об'єктів для вибору разом із мінімальним і максимальним діапазоном для кількості об'єктів для створення.
        void LayoutObjectAtRandom(GameObject[] tileArray, int minimum, int maximum)
        {
            // Вибираємо випадкову кількість об'єктів для інстанціювання в межах мінімальних і максимальних лімітів
            int objectCount = Random.Range(minimum, maximum + 1);

            // Інстанціюємо об'єкти до випадкового обраного ліміту objectCount
            for (int i = 0; i < objectCount; i++)
            {
                // Вибираємо позицію для randomPosition, отримуючи випадкову позицію зі списку доступних Vector3, збережених в gridPosition
                Vector3 randomPosition = RandomPosition();

                // Вибираємо випадкову плитку з tileArray та присвоюємо її tileChoice
                GameObject tileChoice = tileArray[Random.Range(0, tileArray.Length)];

                // Інстанціюємо tileChoice на позиції, поверненій від RandomPosition, без зміни обертання
```

```
                Instantiate(tileChoice, randomPosition, Quaternion.identity);
            }
        }


        // SetupScene ініціалізує наш рівень та викликає попередні функції для розміщення ігрової дошки
        public void SetupScene(int level)
        {
            // Створює зовнішні стіни та підлогу.
            BoardSetup();

            // Скидаємо наш список gridpositions.
            InitialiseList();

            // Інстанціюємо випадкову кількість стінних плиток на основі мінімального та максимального значень, на випадкових позиціях.
            LayoutObjectAtRandom(wallTiles, wallCount.minimum, wallCount.maximum);

            // Інстанціюємо випадкову кількість плиток їжі на основі мінімального та максимального значень, на випадкових позиціях.
```

```
            LayoutObjectAtRandom(foodTiles, foodCount.minimum, foodCount.maximum);

            // Визначаємо кількість ворогів на основі поточного номеру рівня, на основі логарифмічної прогресії
            int enemyCount = (int)Mathf.Log(level, 2f);

            // Інстанціюємо випадкову кількість ворогів на основі мінімального та максимального значень, на випадкових позиціях.
            LayoutObjectAtRandom(enemyTiles, enemyCount, enemyCount);

            // Інстанціюємо плитку виходу в верхньому правому куті нашої ігрової дошки
            Instantiate(exit, new Vector3(columns - 1, rows - 1, 0f), Quaternion.identity);
        }
    }
}
```

4.Enemy

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

namespace Completed

{

// Ворог успадковується від MovingObject, нашого базового класу для об'єктів, які можуть рухатися, Player також успадковується від цього.

public class Enemy : MovingObject

{

public int playerDamage; // Кількість очок їжі, яку віднімається у гравця при атакі.

public AudioClip attackSound1; // Перший з двох аудіо кліпів, що відтворюються при атакі гравця.

public AudioClip attackSound2; // Другий з двох аудіо кліпів, що відтворюються при атакі гравця.

private Animator animator; // Змінна типу Animator для зберігання посилання на компонент Animator ворога.

private Transform target; // Трансформ для спроби рухатися в напрямку кожного ходу.

private bool skipMove; // Логічне значення для визначення, чи має ворог пропустити хід чи зробити його цього ходу.

//Start перевизначає віртуальну функцію Start базового класу.

```
protected override void Start()
{
    // Реєструє цього ворога у нашому екземплярі GameManager, додаючи його до списку об'єктів Enemy.
    // Це дозволяє GameManager видавати команди на рух.
    GameManager.instance.AddEnemyToList(this);

    // Отримуємо та зберігаємо посилання на приєднаний компонент Animator.
    animator = GetComponent<Animator>();

    // Знаходимо об'єкт Player за його тегом та зберігаємо посилання на його компонент трансформа.
    target = GameObject.FindGameObjectWithTag("Player").transform;

    // Викликаємо функцію Start нашого базового класу MovingObject.
    base.Start();
}

//Перевизначає функцію AttemptMove класу MovingObject для включення функціональності, необхідної для пропуску ходів ворогом.
```

```
//Див. коментарі в MovingObject для більш детальної інформації про те, як працює базова функція AttemptMove.
protected override void AttemptMove<T>(int xDir, int yDir)
{
    //Перевіряємо, чи skipMove дорівнює true, якщо так, встановлюємо його false та пропускаємо цей хід.
    if (skipMove)
    {
        skipMove =
        false; return;

    }

    //Викликаємо функцію AttemptMove з MovingObject.
    base.AttemptMove<T>(xDir, yDir);

    //Тепер, коли ворог зробив хід, встановлюємо skipMove в true для пропуску наступного ходу.
    skipMove = true;
}
```

//MoveEnemy викликається GameManager на кожному ходу для того, щоб кожен ворог намагався рухатися в напрямку гравця.

public void MoveEnemy()

{

//Оголошуємо змінні для напрямків руху по вісі X та Y, які можуть бути від -1 до 1.

//Ці значення дозволяють нам вибирати між головними напрямками: вгору, вниз, вліво і вправо.

int xDir =

0; int yDir

= 0;

//Якщо різниця між позиціями практично нуль (Epsilon), виконуємо наступне:

if (Mathf.Abs(target.position.x - transform.position.x) < float.Epsilon)

//Якщо координата Y позиції цілі (гравця) більше, ніж координата Y позиції цього ворога, встановлюємо напрямок Y 1 (рухаємося вгору). Якщо ні, встановлюємо його -1 (рухаємося вниз).

yDir = target.position.y > transform.position.y ? 1 : -1;

//Якщо різниця між позиціями не практично нуль (Epsilon), виконуємо наступне:

else

//Перевіряємо, чи позиція X цілі більше, ніж позиція X цього ворога, якщо так, встановлюємо напрямок X 1 (рухаємося вправо), якщо ні, встановлюємо його -1 (рухаємося вліво).

xDir = target.position.x > transform.position.x ? 1 : -1;

//Викликаємо функцію AttemptMove і передаємо параметр Player, оскільки ворог рухається і може потенційно зіткнутися з гравцем.

AttemptMove<Player>(xDir, yDir);

}

//OnCantMove викликається, якщо ворог намагається рухатися на місце, зайняте гравцем, він перевизначає функцію OnCantMove класу MovingObject

//і приймає параметр T, який ми використовуємо для передачі компонента, з яким ми очікуємо зіткнення, у цьому випадку - Player

protected override void OnCantMove<T>(T component)

{

//Оголошуємо hitPlayer та встановлюємо його рівним зустрінутому компоненту.

Player hitPlayer = component as Player;

//Викликаємо функцію LoseFood у hitPlayer, передаючи їй playerDamage, кількість очків їжі, яку слід відняти.

```
                hitPlayer.LoseFood(playerDamage);

                //Встановлюємо тригер атаки аніматора для виклику анімації атаки ворога.
                animator.SetTrigger("enemyAttack");

                //Викликаємо функцію RandomizeSfx з SoundManager, передаючи в неї два аудіо кліпи для вибору випадковим чином між ними.
                SoundManager.instance.RandomizeSfx(attackSound1, attackSound2);
            }
        }
    }
```

5. Loader

```
using UnityEngine;
using System.Collections;

namespace Completed
{
    public class Loader : MonoBehaviour
    {
```

```
}

        public GameObject gameManager;                  // Префаб GameManager для
інстанціювання.
        public GameObject soundManager;                // Префаб SoundManager для
інстанціювання.



    void Awake()
    {
            // Перевіряємо, чи вже призначено GameManager статичній змінній
GameManager.instance або чи вона ще null
        if (GameManager.instance == null)

            // Інстанціюємо префаб GameManager
            Instantiate(gameManager);

            // Перевіряємо, чи вже призначено SoundManager статичній змінній
SoundManager.instance або чи вона ще null
        if (SoundManager.instance == null)

            // Інстанціюємо префаб SoundManager
            Instantiate(soundManager);
    }
  }
```

}

6. MovingObject

---

using UnityEngine;

using System.Collections;

namespace Completed

{

// Ключове слово abstract дозволяє створювати класи та члени класів, які є неповними і повинні бути реалізовані у похідному класі.

public abstract class MovingObject : MonoBehaviour

{

public float moveTime = 0.1f; // Час, який потрібно для переміщення об'єкта, в секундах.

public LayerMask blockingLayer; // Шар, на якому буде перевірятися колізія.

private BoxCollider2D boxCollider; // Компонент BoxCollider2D, прикріплений до цього об'єкта.

private Rigidbody2D rb2D; // Компонент Rigidbody2D, прикріплений до цього об'єкта.

```
        private float inverseMoveTime;            // Використовується для більш ефективного переміщення.
        private bool isMoving;          // Чи переміщується об'єкт в даний момент.


        // Захищені віртуальні функції можуть бути перевизначені в похідних класах.
        protected virtual void Start()
        {
            // Отримання посилання на компонент BoxCollider2D цього об'єкта
            boxCollider = GetComponent<BoxCollider2D>();

            // Отримання посилання на компонент Rigidbody2D цього об'єкта
            rb2D = GetComponent<Rigidbody2D>();

            // Зберігання оберненого значення часу переміщення для ефективності.
            inverseMoveTime = 1f / moveTime;
        }


        // Метод Move повертає true, якщо переміщення вдале, і false, якщо ні.
```

// Move приймає параметри для напрямку x, напрямку y та RaycastHit2D для перевірки колізії.

protected bool Move(int xDir, int yDir, out RaycastHit2D hit)

{

// Збереження початкової позиції для переміщення, на основі поточної позиції об'єкта.

Vector2 start = transform.position;

// Розрахунок кінцевої позиції на основі переданих параметрів напрямку під час виклику Move.

Vector2 end = start + new Vector2(xDir, yDir);

// Вимкнення boxCollider, щоб linecast не виявляв колізію з власним колайдером цього об'єкта.

boxCollider.enabled = false;

// Виклик linecast від початкової точки до кінцевої, перевірка колізії на blockingLayer.

hit = Physics2D.Linecast(start, end, blockingLayer);

// Повторне включення boxCollider після linecast

boxCollider.enabled = true;

```
            // Перевірка, чи нічого не було виявлено, і чи об'єкт не рухається
            вже. if (hit.transform == null && !isMoving)
            {
                // Запуск ко-рутини SmoothMovement, передавши Vector2 end як
пункт призначення.
                StartCoroutine(SmoothMovement(end));

                // Повернення true для підтвердження успішності переміщення.
                return true;
            }

            // Якщо щось було виявлено, повернення false, переміщення не
вдалося.
            return false;
        }


        // Ко-рутина для плавного переміщення об'єктів з одного простору до
наступного, приймає параметр end для визначення, куди переміститися.
        protected IEnumerator SmoothMovement(Vector3 end)
        {
            // Об'єкт зараз переміщується.
```

```
            isMoving = true;

            // Розрахунок залишкової відстані до переміщення на
основі квадратного модуля різниці між поточною позицією та параметром end.
            // Використовується квадратний модуль замість модуля, оскільки це
обчислювально дешевше.
            float sqrRemainingDistance = (transform.position - end).sqrMagnitude;

            // Поки ця відстань більша за дуже мале значення (Епсилон, майже
нуль):
            while (sqrRemainingDistance > float.Epsilon)
            {
                // Знаходження нової позиції пропорційно ближче до end, на
основі moveTime
                Vector3 newPostion = Vector3.MoveTowards(rb2D.position, end,
inverseMoveTime * Time.deltaTime);

                // Виклик MovePosition на прикріпленому Rigidbody2D
та переміщення його до розрахованої позиції.
                rb2D.MovePosition(newPostion);

                // Перерахування залишкової відстані після переміщення.
                sqrRemainingDistance = (transform.position - end).sqrMagnitude;
```

```
// Повернення та повторення до тих пір, поки sqrRemainingDistance не буде досить близьким до нуля, щоб завершити функцію.
            yield return null;
        }

        // Переконання, що об'єкт знаходиться точно у кінці свого руху.
        rb2D.MovePosition(end);

        // Об'єкт більше не рухається.
        isMoving = false;
    }

    // Ключове слово virtual означає, що AttemptMove може бути перевизначений в похідних класах за допомогою ключового слова override.
    // AttemptMove приймає загальний параметр T для визначення типу компонента, який ми очікуємо, що наша одиниця буде взаємодіяти з ним, якщо заблокована (Player для Enemies, Wall для Player).
    protected virtual void AttemptMove<T>(int xDir, int
        yDir) where T : Component
    {
```

```
// Hit зберігатиме все, що наш linecast попаде, коли Move буде викликано.
RaycastHit2D hit;

// Встановлення canMove в true, якщо Move вдалося, false, якщо ні.
bool canMove = Move(xDir, yDir, out hit);

// Перевірка, чи нічого не було виявлено linecast
if (hit.transform == null)
    // Якщо нічого не було виявлено, повернення та виконання подальшого коду не відбувається.
    return;

// Отримання посилання на компонент типу T, прикріплений до об'єкта, який був виявлений
T hitComponent = hit.transform.GetComponent<T>();

// Якщо canMove дорівнює false, і hitComponent не дорівнює null, означає, що MovingObject заблоковано та відбулася взаємодія з чимось.
if (!canMove && hitComponent != null)

    // Виклик функції OnCantMove та передача їй hitComponent як параметр.
```

```
                    OnCantMove(hitComponent);
            }


            // Ключове слово abstract вказує на те, що річ, яку модифікують, має
відсутню або неповну реалізацію.
            // OnCantMove буде перевизначено функціями у похідних класах.
            protected abstract void OnCantMove<T>(T component)
                where T : Component;
        }
    }
```

7.Player

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;   //Дозволяє нам використовувати
UI. using UnityEngine.SceneManagement;

namespace Completed
{
```

//Player успадковує від MovingObject, наш базовий клас для об'єктів, які можуть рухатися, Enemy також успадковує від нього.

public class Player : MovingObject

{

public float restartLevelDelay = 1f; //Час затримки в секундах для перезапуску рівня.

public int pointsPerFood = 10; //Кількість очок, яку додають до очок їжі гравця при підборі об'єкта їжі.

public int pointsPerSoda = 20; //Кількість очок, яку додають до очок їжі гравця при підборі об'єкта соди.

public int wallDamage = 1; //Скільки шкоди гравець наносить стіні, коли рубає її.

public Text foodText; //Текст UI для відображення поточної кількості очок їжі гравця.

public AudioClip moveSound1; //1 з 2 аудіо кліпів, які відтворюються, коли гравець рухається.

public AudioClip moveSound2; //2 з 2 аудіо кліпів, які відтворюються, коли гравець рухається.

public AudioClip eatSound1; //1 з 2 аудіо кліпів, які відтворюються, коли гравець збирає об'єкт їжі.

public AudioClip eatSound2; //2 з 2 аудіо кліпів, які відтворюються, коли гравець збирає об'єкт їжі.

public AudioClip drinkSound1; //1 з 2 аудіо кліпів, які відтворюються, коли гравець збирає об'єкт соди.

```
        public AudioClip drinkSound2;           //2 з 2 аудіо кліпів, які
відтворюються, коли гравець збирає об'єкт соди.
        public AudioClip gameOverSound;             //Аудіо кліп, який відтворюється,
коли гравець помирає.


        private Animator animator;                  //Використовується для зберігання
посилання на компонент аніматора гравця.
        private int food;                           //Використовується для зберігання
загальної кількості очок їжі гравця під час рівня.
  #if UNITY_IOS || UNITY_ANDROID || UNITY_WP8 || UNITY_IPHONE
        private Vector2 touchOrigin = -Vector2.one; //Використовується для зберігання
місця початку торкання екрану для мобільних керувань.
  #endif



        //Start перевизначає функцію Start з MovingObject
        protected override void Start()
        {
            //Отримуємо посилання на компонент аніматора гравця
            animator = GetComponent<Animator>();
```

```
        //Отримуємо поточну кількість очок їжі, збережених в GameManager.instance між рівнями.
        food = GameManager.instance.playerFoodPoints;

        //Встановлюємо текст foodText для відображення поточної кількості очок їжі гравця.
        foodText.text = "Food: " + food;

        //Викликаємо функцію Start базового класу MovingObject.
        base.Start();
    }

    //Ця функція викликається, коли поведінка стає вимкненою або неактивною.
    private void OnDisable()
    {
        //Коли об'єкт Player вимкнено, зберігаємо поточну локальну кількість очок їжі в GameManager, щоб їх можна було завантажити на наступному рівні.
        GameManager.instance.playerFoodPoints = food;
    }

    private void Update()
    {
```

//Якщо це не хід гравця, виходимо з функції.

if (!GameManager.instance.playersTurn) return;

int horizontal = 0; //Використовується для зберігання горизонтального напрямку руху.

int vertical = 0; //Використовується для зберігання вертикального напрямку руху.

//Перевіряємо, чи ми працюємо в Unity editor або в автономній збірці.

#if UNITY_STANDALONE || UNITY_WEBPLAYER

//Отримуємо введення від менеджера введення, округлюємо його до цілого і зберігаємо в horizontal для встановлення напрямку руху по осі x

horizontal = (int)(Input.GetAxisRaw("Horizontal"));

//Отримуємо введення від менеджера введення, округлюємо його до цілого і зберігаємо в vertical для встановлення напрямку руху по осі y

vertical = (int)(Input.GetAxisRaw("Vertical"));

//Перевіряємо, чи рухаємося горизонтально, якщо так, встановлюємо vertical на нуль.

if (horizontal != 0)

{

```
                vertical = 0;
            }
            //Перевіряємо, чи ми працюємо на iOS, Android, Windows Phone 8 або Unity iPhone
#elif UNITY_IOS || UNITY_ANDROID || UNITY_WP8 || UNITY_IPHONE

//Перевіряємо, чи введення зареєструвало більше нуля торкань
if (Input.touchCount > 0)
{
    //Зберігаємо перше виявлене торкання.
    Touch myTouch = Input.touches[0];

    //Перевіряємо, чи фаза цього торкання дорівнює Began
    if (myTouch.phase == TouchPhase.Began)
    {
        //Якщо так, встановлюємо touchOrigin на позицію цього торкання
        touchOrigin = myTouch.position;
    }

    //Якщо фаза торкання не Began, і замість цього дорівнює Ended, і x координата touchOrigin більша або дорівнює нулю:
    else if (myTouch.phase == TouchPhase.Ended && touchOrigin.x >= 0)
```

```
{
    //Встановлюємо touchEnd на позицію цього торкання
    Vector2 touchEnd = myTouch.position;

    //Обчислюємо різницю між початком і кінцем торкання по осі x.
    float x = touchEnd.x - touchOrigin.x;

    //Обчислюємо різницю між початком і кінцем торкання по осі y.
    float y = touchEnd.y - touchOrigin.y;

    //Встановлюємо touchOrigin.x на -1, щоб наша умова else if оцінювалася як false і не повторювалася негайно.
    touchOrigin.x = -1;

    //Перевіряємо, чи різниця по осі x більша за різницю по осі y.
    if (Mathf.Abs(x) > Mathf.Abs(y))
        //Якщо x більший за нуль, встановлюємо horizontal на 1, інакше встановлюємо його на -1
        horizontal = x > 0 ? 1 : -1;
    else
        //Якщо y більший за нуль, встановлюємо vertical на 1, інакше встановлюємо його на -1
```

```
                    vertical = y > 0 ? 1 : -1;
        }
    }

    #endif //Кінець секції компіляції залежної від мобільної платформи, яка почалася вище з #elif
                    //Перевіряємо, чи маємо ненульове значення для horizontal або vertical
                    if (horizontal != 0 || vertical != 0)
                    {
                         //Викликаємо AttemptMove, передаючи в загальному параметрі Wall, оскільки саме з цим гравець може взаємодіяти, якщо зустріне його (атакуючи його)
                         //Передаємо horizontal і vertical як параметри, щоб вказати напрямок руху гравця.
                         AttemptMove<Wall>(horizontal, vertical);
                    }
               }

               //AttemptMove перевизначає функцію AttemptMove в базовому класі MovingObject
               //AttemptMove приймає загальний параметр T, який для Player буде типу Wall, а також приймає цілі числа для напрямку руху по x і y.
               protected override void AttemptMove<T>(int xDir, int yDir)
```

{

//Кожен раз, коли гравець рухається, віднімаємо очки

їжі. food--;

//Оновлюємо відображення тексту їжі для відображення поточного рахунку.

foodText.text = "Food: " + food;

//Викликаємо метод AttemptMove базового класу, передаючи компонент T (в цьому випадку Wall) і напрямок руху по x і y.

base.AttemptMove<T>(xDir, yDir);

//Hit дозволяє нам звертатися до результату Linecast, виконаного в Move.

RaycastHit2D hit;

//Якщо Move повертає true, тобто гравець зміг рухатися в порожній простір.

if (Move(xDir, yDir, out hit))

{

//Викликаємо RandomizeSfx з SoundManager для відтворення звуку руху, передаючи два аудіо кліпи на вибір.

```
SoundManager.instance.RandomizeSfx(moveSound1,
moveSound2);
            }

            //Оскільки гравець перемістився і втратив очки їжі, перевіряємо, чи не закінчилася гра.
            CheckIfGameOver();

            //Встановлюємо булеву змінну playersTurn в GameManager на false, оскільки хід гравця закінчився.
            GameManager.instance.playersTurn = false;
        }

        //OnCantMove перевизначає абстрактну функцію OnCantMove в класі MovingObject.
        //Вона приймає загальний параметр T, який у випадку Player є Wall, який гравець може атакувати і знищувати..
        protected override void OnCantMove<T>(T component)
        {
```

//Встановлюємо hitWall, щоб дорівнював компоненту, переданому як параметр.

Wall hitWall = component as Wall;

//Викликаємо функцію DamageWall стіни, яку ми вдаряємо.

hitWall.DamageWall(wallDamage);

//Встановлюємо тригер атаки контролера анімації гравця, щоб відтворити анімацію атаки гравця.

animator.SetTrigger("playerChop");

}

//OnTriggerEnter2D відсилається, коли інший об'єкт потрапляє в тригер колайдер, приєднаний до цього об'єкта (тільки 2D фізика).

private void OnTriggerEnter2D(Collider2D other)

{

//Перевіряємо, чи тег тригера, з яким зіткнувся, є Exit.

if (other.tag == "Exit")

{

//Викликаємо функцію Restart для початку наступного рівня з затримкою restartLevelDelay(за замовчуванням 1 секунда).

```
            Invoke("Restart", restartLevelDelay);

            //Вимикаємо об'єкт гравця, оскільки рівень завершено.
            enabled = false;
        }

        //Перевіряємо, чи тег тригера, з яким зіткнувся, є Food.
        else if (other.tag == "Food")
        {

            //Додаємо pointsPerFood до поточного загального рахунку їжі гравця.
            food += pointsPerFood;

            //Оновлюємо текст foodText, щоб відображати поточний загальний рахунок і повідомити гравця, що вони заробили очки.
            foodText.text = "+" + pointsPerFood + " Food: " + food;

            //Викликаємо функцію RandomizeSfx з SoundManager і передаємо два звуки їжі для вибору між ними для відтворення звукового ефекту їжі.
            SoundManager.instance.RandomizeSfx(eatSound1, eatSound2);
```

//Вимикаємо об'єкт їжі, з яким зіткнувся гравець.

other.gameObject.SetActive(false);

}

//Перевіряємо, чи тег тригера, з яким зіткнувся, є Soda.

else if (other.tag == "Soda")

{

//Додаємо pointsPerSoda до загального рахунку їжі гравця.

food += pointsPerSoda;

//Оновлюємо текст foodText, щоб відображати поточний загальний рахунок і повідомити гравця, що вони заробили очки.

foodText.text = "+" + pointsPerSoda + " Food: " + food;

//Викликаємо функцію RandomizeSfx з SoundManager і передаємо два звуки пиття для вибору між ними для відтворення звукового ефекту пиття.

SoundManager.instance.RandomizeSfx(drinkSound1, drinkSound2);

//Вимикаємо об'єкт соди, з яким зіткнувся гравець.

other.gameObject.SetActive(false);

}

}

//Restart перезавантажує сцену при виклику.

private void Restart()

{

//Завантажуємо останню завантажену сцену, в даному випадку Main, єдина сцена в грі. І завантажуємо її в режимі "Single", щоб вона замінила існуючу

//і не завантажувала всі об'єкти сцени в поточній сцені.

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex, LoadSceneMode.Single);

}

//LoseFood викликається, коли ворог атакує гравця.

//Вона приймає параметр втрати, який вказує, скільки очок втратити.

public void LoseFood(int loss)

{

//Встановлюємо тригер для аніматора гравця для переходу до анімації playerHit.

animator.SetTrigger("playerHit");

//Віднімаємо втрачені очки їжі від загальної кількості гравця.

```
            food -= loss;

            //Оновлюємо відображення їжі з новим загальним.
            foodText.text = "-" + loss + " Food: " + food;

            //Перевіряємо, чи закінчилася гра.
            CheckIfGameOver();
        }

        //CheckIfGameOver перевіряє, чи гравець вичерпав усі очки їжі, і якщо так, завершує гру.
        private void CheckIfGameOver()
        {
            //Перевіряємо, чи загальна кількість очків їжі менше або дорівнює нулю.
            if (food <= 0)
            {
                //Викликаємо функцію PlaySingle з SoundManager та передаємо їй gameOverSound як аудіо кліп для відтворення.
                SoundManager.instance.PlaySingle(gameOverSound);

                //Зупиняємо фонову музику.
```

```
                    SoundManager.instance.musicSource.Stop();


                    //Викликаємо функцію GameOver з
                    GameManager.
                    GameManager.instance.GameOver();
                }
            }
        }
    }
```

8.SoundManager

```
using UnityEngine;
using System.Collections;


namespace Completed
{
    public class SoundManager : MonoBehaviour
    {
        public AudioSource efxSource;                   // Перетягніть посилання на аудіо джерело, яке буде відтворювати звукові ефекти.
        public AudioSource musicSource;                 // Перетягніть посилання на аудіо джерело, яке буде відтворювати музику.
```

public static SoundManager instance = null; // Дозволяє іншим скриптам викликати функції з SoundManager.

public float lowPitchRange = .95f; // Найнижче, на яке може бути випадково настроєний звуковий ефект.

public float highPitchRange = 1.05f; // Найвище, на яке може бути випадково настроєний звуковий ефект.

void Awake()

{

// Перевіряємо, чи вже існує екземпляр SoundManager.

if (instance == null)

// Якщо ні, встановлюємо його на цей.

instance = this;

// Якщо екземпляр вже існує:

else if (instance != this)

// Знищуємо це, це забезпечує нашу паттерн одиночності, тому що може бути тільки один екземпляр SoundManager.

Destroy(gameObject);

// Встановлюємо SoundManager як DontDestroyOnLoad, щоб він не знищувався при перезавантаженні сцени.

DontDestroyOnLoad(gameObject);

```
}

// Використовується для відтворення одиночних звукових кліпів.
public void PlaySingle(AudioClip clip)
{
    // Встановлюємо кліп нашого аудіо джерела efxSource на кліп, переданий як параметр.
    efxSource.clip = clip;

    // Відтворюємо
    кліп.
    efxSource.Play();
}

// RandomizeSfx вибирає випадково між різними аудіо кліпами і трохи змінює їх частоту.
public void RandomizeSfx(params AudioClip[] clips)
{
    // Генеруємо випадкове число між 0 і довжиною нашого масиву кліпів, переданих.
    int randomIndex = Random.Range(0, clips.Length);
```

```
        // Вибираємо випадкову частоту для відтворення нашого кліпа між нашими
високими та низькими частотними діапазонами.
            float randomPitch = Random.Range(lowPitchRange, highPitchRange);

// Встановлюємо частоту аудіо джерела на випадково вибрану частоту.
            efxSource.pitch = randomPitch;
// Встановлюємо кліп на кліп нашого випадково вибраного індексу.
            efxSource.clip = clips[randomIndex];

            // Відтворюємо
            кліп.
            efxSource.Play();
        }
    }
}
```

9. Wall

```
using UnityEngine;
using System.Collections;

namespace Completed
```

{

```
public class Wall : MonoBehaviour
{
    public AudioClip chopSound1;          // Перший із двох аудіо кліпів, що відтворюються, коли гравець атакує стіну.
    public AudioClip chopSound2;          // Другий із двох аудіо кліпів, що відтворюються, коли гравець атакує стіну.
    public Sprite dmgSprite;              // Альтернативний спрайт для відображення після атаки гравця на стіну.
    public int hp = 3;          // Очки здоров'я стіни.


    private SpriteRenderer spriteRenderer;
// Зберігає посилання на компонент SpriteRenderer, приєднаний до цього об'єкта.
    void Awake()
    {
// Отримуємо посилання на компонент SpriteRenderer.
        spriteRenderer = GetComponent<SpriteRenderer>();
    }


// DamageWall викликається, коли гравець атакує стіну.
    public void DamageWall(int loss)
```

```
            {
    // Викликаємо функцію RandomizeSfx з SoundManager для відтворення одного з двох звуків атаки.
                SoundManager.instance.RandomizeSfx(chopSound1, chopSound2);



    // Встановлюємо spriteRenderer на спрайт пошкодженої стіни.
                spriteRenderer.sprite = dmgSprite;


    // Віднімаємо втрати від загальної кількості очок здоров'я.
                hp -= loss;
    // Якщо кількість очок здоров'я менше або дорівнює нулю:
                if (hp <= 0)
                    // Вимикаємо об'єкт.
                    gameObject.SetActive(false);
            }
        }
    }
```