Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *Нечепоренка Івана Вячеславовича*
(ПІБ)

академічної групи *122-18-2*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

освітньої програми *Комп'ютерні науки*
(назва освітньої програми)

на тему: *Розробка мобільного додатка на двигуні Unity3D у стилі гри Geometry Dash*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	*доц. Кабак Л.В.*			
розділів:				
спеціальний	*доц. Кабак Л.В.*			
економічний	*доц. Касьяненко Л.В.*			
Рецензент				
Нормоконтролер	*доц. Гуліна І.Г.*			

Дніпро
2022

Міністерство освіти і науки України
НТУ «Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
завідувач кафедри
програмного забезпечення комп'ютерних систем
(повна назва)

_____ І.М. Удовик
(підпис) (прізвище, ініціали)

« » _____ 2022 року

ЗАВДАННЯ
на кваліфікаційну роботу ступеня

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *122-18-2* *Нечепоренка Івана Вячеславовича*
(група) (прізвище та ініціали)

тема кваліфікаційної роботи *Розробка мобільного додатка на двигуні Unity3D у стилі гри Geometry Rush*

затверджена наказом ректора НТУ «ДП» від №

Розділ	Зміст виконання	Термін виконання
Спеціальний	***На основі матеріалів виробничої практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі.*** ***Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	***13.05.2022 р.***
Економічний	***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки***	***27.05.2022 р.***

Завдання видав _____ *доц. Кабак Л.В.*
(підпис) (посада, прізвище, ініціали)

Завдання прийняв до виконання _____ *Нечепоренко І.В.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *14.01.2022 р.*

Термін подання кваліфікаційної роботи до ЕК: *13.06.2022 р.*

РЕФЕРАТ

Пояснювальна записка: 86 с., 45 рис., 3 дод., 21 джерел.

Темою кваліфікаційної роботи є «Розробка мобільного додатка на двигуні Unity3D у стилі гри Geometry Dash».

Мета кваліфікаційної роботи: створення ігрового додатку що допоможе розвинути просторове мислення та провести час із задоволенням.

У вступі розглядається аналіз та сучасний стан проблеми, конкретизується мета кваліфікаційної роботи та галузь її застосування, наведено обґрунтування актуальності теми та уточнюється постановка завдання.

У першому розділі проаналізовано предметну галузь, визначено актуальність завдання та призначення розробки, сформульовано постановку завдання, зазначено вимоги до програмної реалізації, технологій та програмних засобів.

У другому розділі проаналізовані наявні рішення, обрано платформи для розробки, виконано проектування і розробка програми, описана робота програми, алгоритм і структура її функціонування, а також виклик та завантаження програми, визначено вхідні і вихідні дані, охарактеризовано склад параметрів технічних засобів.

В економічному розділі визначено трудомісткість розробленої інформаційної системи, проведений підрахунок вартості роботи по створенню програми та розраховано час на його створення.

Практичне значення полягає у створенні програмного додатка, що надає можливість провести час із задоволенням.

Актуальність розробленого програмного продукту полягає в створенні такого додатку, який матиме попит серед цільової аудиторії – гравців у відеоігри.

Список ключових слів: ІГРОВИЙ ДВИГУН, КОМПОНЕНТ, КОМП’ЮТЕР, ІНФОРМАЦІЙНА СИСТЕМА, ОБЛІК, АЛГОРИТМ, ПРОЕКТУВАННЯ, МЕНЮ, ВКЛАДКА, ДОДАТОК.

ABSTRACT

Explanatory note: 86 p., 45 fig., 3 docs, 21 journals.

The theme of the qualification work is "Developing a mobile add-on on the Unity3D engine in the Geometry Dash style".

Meta kvartifikatsionnoj work: the creation of game add-on which will help to develop a spacious mind and spend time with pleasure.

The introduction examines the analysis and the current state of the problem, specifies the meta-qualification work and the area of application of the topic, provides a substantiation of relevance and clarifies the statement of objectives.

The first section analyzes the subject area, identifies the relevance of the task and the purpose of the development, formulates the statement of the task, and specifies the requirements to the software implementation, technologies and software tools.

In the second section, the existing solutions were analyzed, the platforms for development were reversed, the design and development of the program were carried out, the program operation, the algorithm and structure of its functioning, as well as the program activation and engagement were described, the input and output data were identified, and the composition of parameters of the technical facilities was characterized.

In the economic section, the labor intensity of the developed information system is determined, the cost of the work on the creation of the program is estimated and the time for its creation is calculated.

Practical importance lies in the creation of software add-on, which gives the opportunity to spend time with satisfaction.

The relevance of the developed software product lies in the creation of such an add-on, which will be popular among the target audience - video game players.

The list of keywords: game engine, component, COMPONENT, INFORMATION SYSTEM, CLOCK, ALGORITHM, DEVELOPMENT, MENU, ADDITIVE, ADDITIVE.

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

3D — three-dimensional;

MVC — Model-View-Controller;

IDE — Integrated Development Environment;

SSAO — Screen space ambient occlusion;

SDK — Software development kit;

GUI — graphical user interface;

VS — Visual Studio

PS — PlayStation;

PSP — PlayStation Portable;

OS — Operation System;

VR — Virtual Reality;

Mac OS — Macintosh Operating System;

UE — Unity Engine.

ЗМІСТ

ВСТУП

Цілі даної кваліфікаційної роботи та предмет її діяльності безпосередньо пов’язані з напрямом підготовки та відповідають загальній тематиці кваліфікації та переліку зазначених виробничих функцій, типових завдань, навичок та компетенцій бакалавра навчання в напрямі 122 «Комп’ютерні науки».

Предметом кваліфікаційної роботи є проектування та розробка ігрового додатку аркадної гри на движку Unity.

Мета кваліфікації – навчитися працювати з кількома підсистемами операційної системи. В першу чергу це вхідні події в системі та взаємодія з ними. Ці навички дуже важливі при розробці програмного забезпечення для контролю доступу та моніторингу процесів і діяльності користувачів. Вони також допоможуть краще зрозуміти механізми роботи з графічними об’єктами та їх взаємодію на досить низькому рівні.

Ця робота дає змогу ознайомитись із методами створення ігрових додатків та зрозуміти багатовимірні ігрові додатки.

Основна вимога до цього програмного забезпечення - стабільність. Оскільки Unity Engine є інтенсивним графічним двигуном через його якість графіки, швидкість додатків на малопотужних комп’ютерах може бути не дуже стабільною.

Таким чином, завданням цієї кваліфікації є розробка та розробка ігрового додатка, який надасть гравцеві захоплюючий аркадний досвід.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Загальні відомості з предметної галузі

Розробка відеоігор – це процес розробки відеоігор, в якому бере участь розробник відеоігор, яким може бути фізична особа або компанія з сотнями співробітників. Гра може бути створена кількома людьми з обмеженим бюджетом і за підтримки видавця. Тривалість і вартість розробки залежать від складності проекту. Оскільки відеогра — це комп'ютерна програма, її функціонування, технічні можливості, зміст та ігровий процес забезпечуються програмним кодом. Розробка ігор виконується за тими ж кроками, що й розробка програмного забезпечення, але передбачає більше роботи над змістом і механізмами побудови гри. Сучасні ігри здебільшого базуються на готових програмних модулях – ігрових ядрах, в яких уже реалізовані основні функції, що можуть поєднувати графіку, звук, об'єкти та їх рух. Щоб налаштувати ядро для реалізації певної ідеї, розробники доопрацьовують його, додаючи необхідні функції. Існують безкоштовні ігрові ядра і такі, для використання яких потрібна ліцензія. Одні ядра призначені для створення ігор певного жанру, інші - універсальних. Не кожне ядро може забезпечити однакові ігрові можливості та рівні графіки.

Деякі типи ядер дозволяють розробляти ігри для різних платформ, наприклад, Unity Engine підтримує розробку програмного забезпечення для PC, Xbox 360, PlayStation 4, PlayStation VR, Oculus, Android, iOS та багатьох інших.

Кожне ігрове ядро надає багато функцій, які використовуються в різних іграх. Гра, реалізована на такому ядрі, отримує всі ці

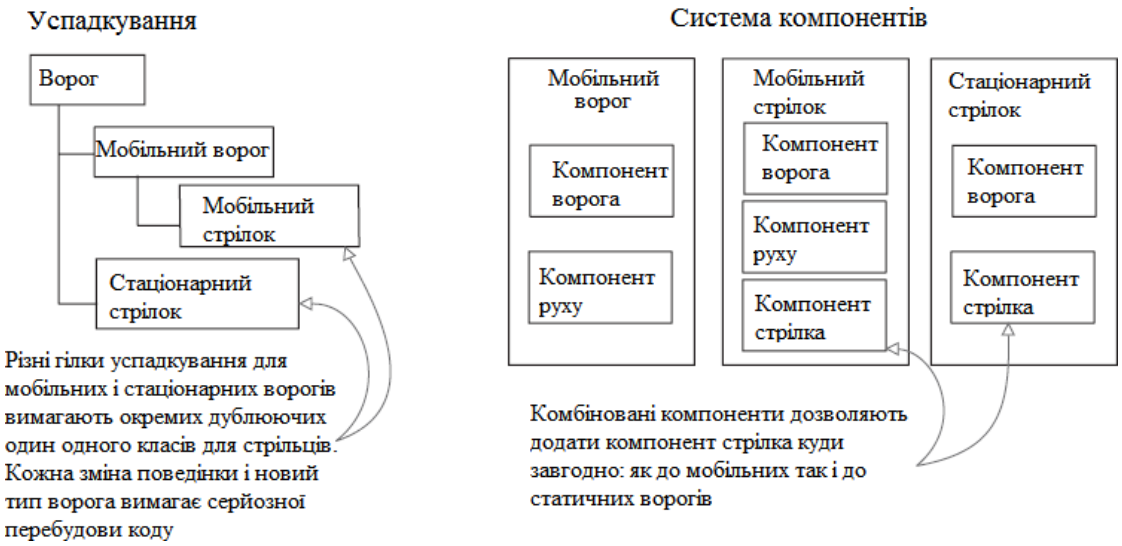
функціональні можливості, крім того, у функціонал додані власні ігрові ресурси та коди ігрового сценарію. Unity надає можливість моделювати фізичне середовище, зображення екранного простору (Screen Space Ambient Occlusion , SSAO), динамічні тіні та багато іншого. Багато ігрових ядер можуть похвалитися подібним набором функцій, але Unity має дві основні переваги перед іншими передовими інструментами розробки ігор: надзвичайно ефективний візуальний робочий процес і потужна кросплатформна підтримка.

Візуальний робочий процес має кілька переваг. Хоча інші інструменти розробки ігор часто являють собою набір різних частин, які потрібно відстежувати, або, можливо, працездатну бібліотеку, вам потрібно налаштувати власне інтегроване середовище розробки (Integrated Development Environment, IDE), порядок складання збірки ті інше в цьому роді, робочий процес Unity пов’язаний зі складно продуманим візуальним редактором. У цьому редакторі розробник складає сцени для майбутньої гри, об’єднуючи ігрові ресурси та код в інтерактивні об’єкти. Це дозволяє створювати професійні ігри швидко та ефективно, підтримуючи продуктивність розробників і надаючи їм список найновіших технологій розробки відеоігор.

Unity Toolkit пропонує потужну міжплатформну підтримку. У даному випадку мається на увазі не тільки місце розгортання (ви можете розгорнути гру на персональному комп’ютері, інтернеті, мобільному пристрої чи консолі), а й інструменти розробки. Існує всього декілька ігрових ядер, які підтримують однакову кількість цільових платформ розгортання, і жодна з них не полегшує операцію розгортання як Unity3D Engine.

На додаток до цих двох основних переваг, необхідно усвідомити ще одну річ, це надання модульної системи компонентів, що використовуються для побудови ігрових об’єктів. «Компоненти» в такій системі - це пов'язані пакети функціональних елементів, так що об'єкти створюються як набори компонентів, а не як жорстка ієрархія класів. Іншими словами, система компонентів є альтернативним (і зазвичай більш гнучким) підходом до об’єктно-

орієнтованого програмування, де ігрові об’єкти створюються шляхом злиття, а не успадкування. Порівняння підходів показано на схемі на малюнку 1.1.



Мал. 1.1. Порівняння системи спадкування та компонентної системи

У системі компонентів об’єкт існує в горизонтальній ієрархії, тому різні об’єкти складаються з різних наборів компонентів, а не з успадкованої структури, де різні об’єкти розташовані на різних гілках дерева. Такий макет спрощує створення прототипів, оскільки обробка необхідного набору компонентів набагато швидша і легша, ніж перебудова ланцюга успадкування при зміні кожного об’єкта. Крім того, у розробника є можливість писати код, який реалізує його власну систему компонентів, але Unity вже має дуже надійну версію такої системи, органічно інтегровану з візуальним редактором. Тому розробник має можливість не тільки програмно керувати компонентами, а й підключати та відключати між ними в редакторі. Можливості розробки не обмежуються складанням предметів із готових деталей; на його основі розроблені шаблони проектування, якими розробник може користуватися в повній мірі.

Щоб створити будь-яку гру в середовищі Unity, розробник повинен володіти принаймні однією з доступних (Unity) мов програмування: C# або

JavaScript. Мова програмування C# має багато переваг перед JavaScript і набагато менше недоліків. Однією з переваг є те, що C# строго типізований, чого не можна сказати про JavaScript. Наразі серед розробників існують різні думки щодо того, чи є найкращим підходом динамічна перевірка типів, наприклад, розробка веб-додатків, але при написанні програм для певних платформ (наприклад, Windows) це часто вигідно, а іноді навіть потрібна статична типізація. Unity навіть додала директиву #pragma, яка забезпечує статичну перевірку типів у JavaScript. Хоча з технічної сторони таке цілком допустимо, при цьому порушується один з основних принципів JavaScript.

Поведінка об'єктів у грі контролюється компонентами (Components), які з ними пов'язані. Ядро Unity дозволяє розробникам створювати власні компоненти за допомогою скриптів (сценаріїв). Вони дозволяють активувати події в грі, змінювати налаштування компонентів і будь-яким чином реагувати на введення користувача.

Програмування в середовищі Unity не відбувається, код існує у вигляді окремих файлів, розташування яких надає сам розробник на Unity. Файли сценаріїв можна створювати в Unity, але в будь-якому випадку розробнику знадобиться використовувати текстовий редактор або IDE, щоб написати код для цих спочатку порожніх файлів.

У 1970-х роках відеоігри стали одним з найпопулярніших розваг. У них грали вдома або в барах і кафе, де їх встановлювали у вигляді ігрових автоматів у коридорах, що нагадували аркади. Звідси і назва першого жанру електронних ігор – «аркади» (перші ігри типу «Зоряні війни», «Понг», «Покемон», «Арканоїд» — усі аркадні ігри). Незважаючи на деяку примітивність, цей жанр досить оригінальний і його важко порівняти з відомими казуальними іграми. Крім того, це чітко показує, як ігровий досвід залежить від технологічних пристроїв (екран, електронна підкладка, спеціальні елементи керування).

У світовій практиці аркади – це ігри на ігрових автоматах. Це не окремий ігровий жанр, а скоріше ігровий напрямок. Комп’ютерна гра чи відеогра

називається «аркадною», якщо вона безпосередньо перенесена з автомата або ж схожа по концепції з іграми для автоматів.

У сучасному розумінні Аркада (англ. Arcade) — це жанр відеоігор, ігровий процес (геймплей) яких заснований на швидкому реагуванні гравця, мінімальній свободі керованих персонажів і простому управлінні.

Класичні аркади характеризуються такими особливостями. Одноекранна гра. У класичних аркадах вся гра зосереджена на одному екрані. В першу чергу це пов’язано з історичними подіями, які відбулися через технічні обмеження, але в той же час істотно вплинули на дизайн гри. Так, гравці могли в будь-який момент побачити весь ігровий світ і приймати рішення на основі повної інформації про його статус. Багато ігор цього жанру мали більше ніж один екран і вони змінювали один одного як рівні. Типовими прикладами тут є Joust, Pac-Man, Mario Bros. Нескінченна гра. Потенційно гравці можуть грати в аркаду необмежений час і, відповідно, не можуть виграти. Це спонукало гравців стати перед іспитом – скільки вони можуть витримати у грі. В аркадах гравець жодного разу не вигравав, і кожна гра зазнавала невдачі. Ігри також були розроблені таким чином, щоб з часом рівні та ігровий процес стають все більш складним і нескінченна гра пропонує нескінченну складність. Ситуація змінилася з появою ринку домашніх комп’ютерів, оскільки видавці змінили своє бажання, щоб гравці повністю проходили гру і згодом захотіли купити нову. Ігровий рахунок. Майже всі класичні аркади містять ігровий обліковий запис, коли гравець отримує очки за виконання різних цілей або завдань. Ігрові бали дозволяють гравцеві зрозуміти, наскільки добре він грав, незважаючи на те, що виграти неможливо. При цьому типовий час гри для середнього гравця становить близько двох хвилин, а для досвідченого – до кількох десятків хвилин. Виходячи з цієї функції, аркади зазвичай мають таблицю записів, у яку гравець може ввести свої ініціали біля рахунку та порівняти з іншими гравцями. Швидке навчання, легкий ігровий процес. Класичні аркади характеризуються тим, що гравці можуть легко освоїти гру, але через її складність стати майстром

гри практично неможливо. Однак якщо гравець помирає під час проходження, це майже завжди його вина. В таких іграх немає «спеціальних комбінацій клавіш», які гравець повинен вивчити з документації для того, щоб зробити щось особливе. Дуже мало ігор розширюють концепт за допомогою очок здоров'я, щитів або таблеток сили. Це пояснюється тим, що з комерційної точки зору аркади повинні були охопити якомога більше гравців, а це означає, що практично кожен у барі чи магазині повинен мати можливість прийти і спробувати гру. При цьому простий ігровий процес не означає, що він «поганий» чи «обмежений» — він може бути «елегантним» і «відшліфованим».

Жанр Arcade (аркада) є одним із найстаріших в ігровій індустрії і пройшов довгий шлях розвитку. Хоча його навряд чи можна назвати найпопулярнішим, він справив величезний вплив на інші жанри. Головна перевага аркад – короткий, але насичений ігровий процес. Поки що розробники створюють все більше нових проектів, що поєднують елементи аркади та інших жанрів, намагаючись залучити більше людей і створити щось цікаве та унікальне. Хоча пік популярності ігрових автоматів припадає на кінець 20 століття, у жанру все ще є величезна кількість прихильників, які не дають жанру припинити своє існування.

1.2. Мета дослідження та визначення проблеми

Мета розробки - спроектувати і розробити двовимірний додаток для аркадної гри на движку Unity.

Розроблений ігровий додаток буде використовуватися гравцями, які хочуть відпочити та розважитися. Допоможуть у цьому відеоігри – вони можуть відвернути увагу від оточуючих проблем і надати гравцеві позитив.

Основною метою програми є організація гри (ігор) на основі взаємодії користувача та ігрового пристрою через візуальний інтерфейс.

Гра призначена для користувачів телефонів з операційною системою Android.

1.3. Підстави для розробки

Відповідно до освітньої програми , згідно навчального плану та графіків навчального процесу, в кінці навчання студент виконує кваліфікаційну роботу.

Тема роботи узгоджується з керівником проекту, випускаючою кафедрою, та затверджується з наказом ректора.

Таким чином підставами для розробки (виконанням кваліфікаційної роботи) є:

- освітня програма спеціальності 122 “Комп’ютерні науки”;

- навчальний план та графік навчального процесу;

- наказ ректора Національного технічного університету “Дніпровська політехніка” № 317-с від 07.06.2022р;

завдання на кваліфікаційну роботу на тему “ Розробка мобільного додатка на двигуні Unity3D у стилі гри Geometry Rush”.

1.4. Постановка завдання

Основною метою кваліфікаційної роботи є проектування та розробка двовимірного додатка у жанрі аркадних ігор на движку Unity. Гра призначена для користувачів смартфонів з операційною системою Android. Основними особливостями розробки повинні бути:

- реалізація ігрової механіки, що дає гравцям можливість керувати грою в двовимірному ігровому просторі;
- розробка кількох ігрових сцен і можливість перемикання між ними;
- створення інтерфейсу ігрової програми з використанням можливостей ядра Unity Engine;
- створення головного меню ігрового додатка;
- тестування розробленого ігрового додатку..

Поставлена задача може бути досягнута при виконанні наступних вимог:

- вивчення предметної галузі завдання;
- проведення порівняльної характеристики можливостей аналогічних програм;
- вибір платформи розробки;
- імпортування двовимірних об’єктів з магазину Asset Store в Unity Engine;
- написання програмного коду.

Кінцевим результатом розробки має стати ігровий додаток для користувачів смартфонів під керуванням операційної системи Android, в якому реалізовано ігровий процес (геймплей), що базується на взаємодії користувача, апаратної складової операційної системи Android, розроблених механік ігрового додатку та візуального інтерфейсу.

1.5. Вимоги до програми або програмного виробу

1.5.1. Вимоги до функціональних характеристик

Для коректного запуску програми в ігровому середовищі Unity3D необхідно встановити Directx версії 11, старіші версії програми неможливо запустити через набір функцій, які використовуються в ігровому движку.

DirectX — це набір API, розроблений для легкої та ефективної обробки мультимедійних, ігрових завдань та завдань програмування відео для операційних систем Microsoft Windows.

Версію графічного драйвера, встановленого в системі, необхідно оновити, оскільки від цього залежить коректна робота ігрового додатку.

Оскільки проект уже запакований у виконуваний файл, користувачеві потрібно лише запустити його.

Введення даних необхідно здійснювати за допомогою периферійних пристроїв введення інформації та їх взаємодії з інтуїтивно зрозумілим інтерфейсом користувача розробленого ігрового додатка. Дані надсилаються з екрану телефона.

Ви також можете отримати сповіщення про потенційні проблеми під час запуску:

Warning - незвичайна ситуація з програмою. Це не критично, але користувач повинен звернути на це увагу.

Error - помилка в програмі. Деякі функції або файли можуть бути недоступними.

1.5.2 Вимоги до інформаційної безпеки

Оскільки програма вже запакована, вимоги до інформаційної безпеки не потрібні, просто дотримуйтесь ієрархії внутрішніх файлів, щоб уникнути можливих збоїв програми. Крім того, у користувача немає необхідність

реєстрації та створення облікового запису в програмі. З цієї причини немає вимог до інформаційної безпеки.

1.5.3 Вимоги до складу та параметрів технічних засобів

Для коректного функціювання ігрового додатку в редакторі Unity3D, мінімальна технічнаконфігурація ПК повинна бути наступною:

- смартфон під керуванням операційної системи Android 4.1 та вище;
- роздільна здатність екрану: 600*800.

1.5.4. Вимоги до інформаційної та програмної сумісності

Сучасні комп’ютерні ігри вимагають значних вимог до обладнання. Основою є швидкий процесор для правильної роботи механізмів у грі. Багатоядерні процесори дозволяли обробляти кілька завдань одночасно, що означало більш складну графіку, більш розвинений штучний інтелект і фізику в грі.

Для 3D-ігор потрібен потужний графічний процесор (GPU), який прискорює відтворення складних графічних сцен у режимі реального часу. Також можна використовувати кілька графічних процесорів на одному комп’ютері, використовуючи такі технології, як масштабований інтерфейс зв’язку (NVidia) або CrossFire (ATI).

Для 3D-ігри звук у вас повинен бути звукова карта (вбудовані аналоги часто не мають 3D-функцій і більш чіткого звуку).

Для управління персонажем у грі використовуються тачскрин, або перефирійні джойстики, кермо для гоночних ігор і геймпади для консольних ігор.

Версія встановленого в системі графічного драйвера також впливає на продуктивність і ігровий процес.

РОЗДІЛ2

ПРОЕКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1. Функціональне призначення інформаційної системи

Результатом цієї кваліфікаційної роботи має стати двовимірний додаток для аркадної гри, створений з використанням движка Unity. Розроблений ігровий додаток буде використовуватися гравцями, які хочуть відпочити та розважитися.

Основна мета ігрового додатка:

- організація ігрового процесу (ігрова);
- ігровий додаток призначений для використання на платформі смартфона (Android);
- розроблений додаток оснащений функціональним інтерфейсом, який складається з екрану головного меню, екрану налаштувань, магазину, інших додаткових екранів та екрану гри;

Головне меню дозволяє користувачеві вибрати певну опцію програми. Після вибору пункту меню - почати гру, запускається гра, в якій користувач має можливість перевірити функціональність гри, яка складається з п'яти основних елементів:

- аркадна система;
- механіка поведінки ігрових об'єктів у двовимірному просторі;
- платформна система;
- система частинок;
- акаунт гравця.

Елементом аркадної системи є те, що взаємодія розробленої механіки та ігрових елементів реалізує характерний для аркадних ігор ігровий процес (грабельність).

Механіка поведінки ігрових об'єктів у двовимірному просторі є результатом розвитку розроблених функцій, що визначають правила взаємодії ігрових об'єктів, створених за допомогою Unity Engine. Це досягається завдяки системі компонентів, що використовується в Unity Engine: для певного ігрового об’єкта створюється скрипт (скрипт), написаний мовою програмування C#.

Платформна система - у створюваному ігровому додатку реалізована платформна система, яка є однією з особливостей жанру Аркади. Розроблено особливий тип ігрового процесу - Endless Runner, підтип жанру аркади, в якому гравцеві доводиться переміщатися по нескінченних платформах.

Система частинок – система створює частинки, які служать візуальними графічними ефектами у випадкових точках певного простору. Ці частинки можуть мати форму, наприклад, кулі або конуса. Система визначає час життя частинки, ступінь утворення, а коли час життя частинки закінчується - система її руйнує.

Обліковий запис гравця - в ігровому додатку реалізовано елемент ігрового процесу, який дозволяє гравцеві контролювати свій ігровий обліковий запис.

2.2 Опис застосованих математичних методів

Для створення двовимірних ігор у жанрі Аркади використовувалися наступні математичні методи: векторні операції та арифметичні оператори додавання, віднімання, множення та ділення.

Операції над векторами, комутація: $\overline{a} + \overline{b} = \overline{b} + \overline{a}$.

Арифметичні оператори: «+», «-», «*», «/».

2.3 Опис використаної архітектури та шаблонів проектування

При розробці додатка для 2D-аркадної гри було обрано незалежно модифікований класичний шаблон програмування MVC (Model-View-Controller).

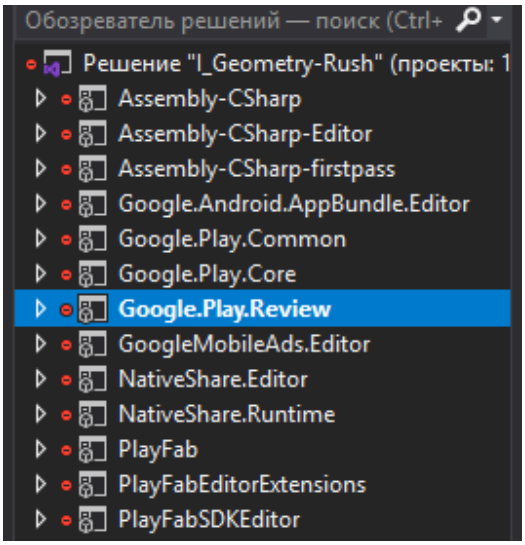
Модель відповідає за внутрішню логіку ігрового додатку. В проекті цей елемент шаблону проектування охоплює зберігання даних, а також правила і алгоритми обробки інформації.

View відповідає за відображення даних Моделі. Цей рівень відображає лише стан моделі у відповідний момент часу. Наступний рівень шаблону проекту MVC – Controller, який відповідає за обробку дій користувача.

Controller створює зв'язок між рівнем Model та діями користувача, що виникають у результаті взаємодії з рівнем View. Фактично, для будь-якої дії, яку користувач може виконати на рівні View, у Controller має бути визначений обробник події. Цей обробник виконає відповідні маніпуляції із Model і, якщо необхідно, повідомить View про будь-які зміни.

Для створення двовимірної ігрової програми був використаний вбудований IDE Unity Engine — Microsoft Visual Studio 2019. Завдяки вікну Оглядач рішень можна зручно переглядати структуру проекту та скрипти на мові програмування C#. Завдяки функціональності Visual Studio 2019 файли проекту зручно збираються в рішення (solution). Рішення — це фреймворк для організації проектів у Visual Studio. Він зберігає інформацію про стан проекту у двох файлах:

- файл SLN (текстовий, загальний);
- suo-файл (двійковий, специфічні для користувача параметри рішення).



Мал. 2.1. Структура організації файлів скриптів у Visual Studio 2019

2.4 Опис використаних технологій та мов програмування

Для розробки та створення двовимірного додатка для аркадних ігор були використані технології Unity 2020.3.16f1, інтегроване середовище розробки Microsoft Visual Studio 2019 та мова програмування C#.

Середовище Unity дозволяє розробляти ігрові програми для більшості популярних платформ. Ця базова технологія розробляє ігри, які можна запускати на персональних комп'ютерах (Windows, MacOS, Linux), смартфонах і планшетах (iOS, Android, Windows Phone) та ігрових консолях (PS, Xbox, Wii).

Кожне ігрове ядро надає багато функцій, які використовуються в різних іграх. Гра, реалізована на такому ядрі, отримує всі ці функції, додатково додаються власні ігрові ресурси та код сценарію гри (скрипти). Unity надає можливість моделювати фізичне середовище, карти нормалей, зображення ігрового світу екранного простору (SSAO), динамічні тіні тощо. Багато ігрових ядер мають подібний набір функцій, але Unity має дві основні переваги перед іншими високоякісними ядрами: надзвичайно ефективний візуальний робочий процес і потужна підтримка багатьох платформ.

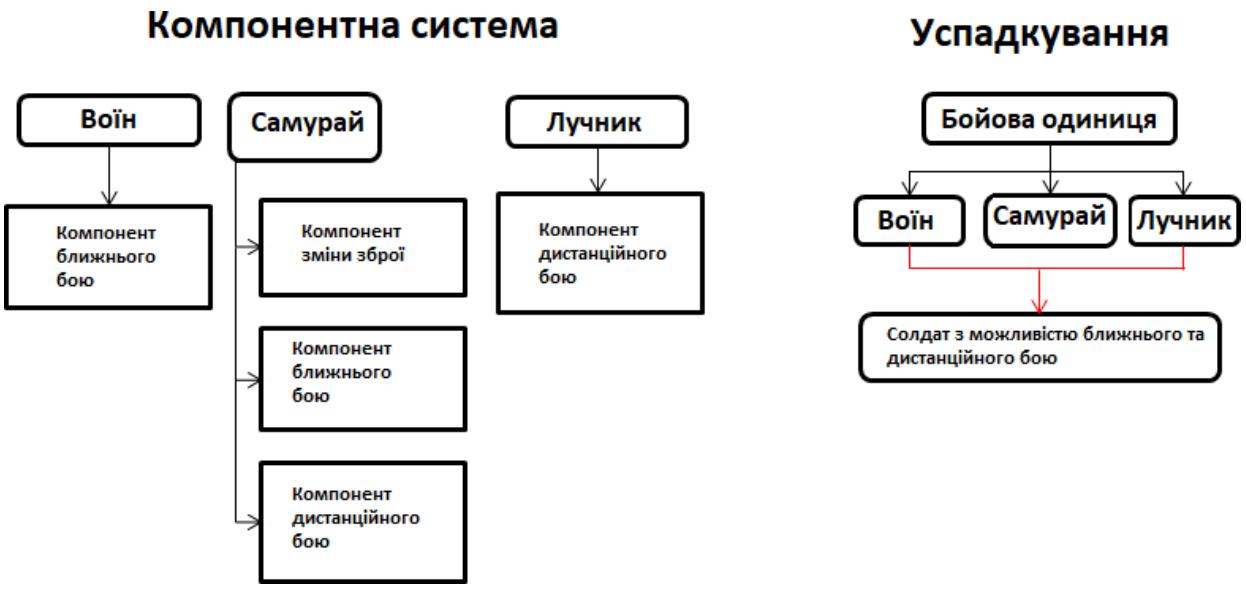
Візуальний робочий процес має певні переваги. Хоча інші інструменти розробки ігор часто являють собою набір різних частин, які потрібно відстежувати, або, можливо, робочу бібліотеку, вам потрібно налаштувати власне інтегроване середовище розробки (Integrated Development Environment, IDE), послідовність збірки тощо, робочий процес в Unity прив'язаний до складно продуманого візуального редактору. У цьому редакторі розробник складає сцени для наступної гри та об'єднує ігрові ресурси та код в інтерактивні об'єкти. Це дозволяє створювати професійні ігри швидко й ефективно, забезпечуючи продуктивність праці розробників та надаючи їм список найновіших технологій розробки відеоігор.

У наборі інструментів Unity існує потужна мультиплатформенна підтримка. У даному випадку мається на увазі не тільки місце реалізації (гра може бути реалізована на персональному комп'ютері, інтернеті, мобільному пристрої чи консолі), а й інструменти програмування. Така незалежність від платформи була пов'язана з тим, що Unity спочатку був розроблений виключно для комп'ютерів Mac, а пізніше був перенесений на комп'ютери з операційною системою Windows. Існує досить небагато ігрових ядер, що підтримують таку ж кількість цільових платформ розгортання, і жодне з них не робить операцію розгортання настільки легкою.

Крім цих двох основних переваг, слід відзначити ще одну річ – це забезпечення модульної системи компонентів, що використовуються при побудові ігрових об'єктів. «Компонентами» в такій системі називають зв'язками функціональних елементів, тому об'єкти створюються як набір компонентів, а не як фіксована ієрархія класів. Іншими словами, система компонентів є альтернативним (і зазвичай більш гнучким) підходом до об'єктно-орієнтованого програмування, де ігрові об'єкти створюються комбінацією, а не успадкуванням. Порівняння підходів показано на діаграмі на рисунку 2.2.

У системі компонентів об'єкт існує в горизонтальній ієрархії, тому різні

об’єкти складаються з різних наборів компонентів, а не з успадкованої структури, де різні об’єкти знаходяться на різних гілках дерева. Такий макет спрощує створення прототипу, оскільки обробка необхідного набору компонентів набагато швидша і легша, ніж переупорядкування рядка успадкування під час зміни кожного об’єкта.



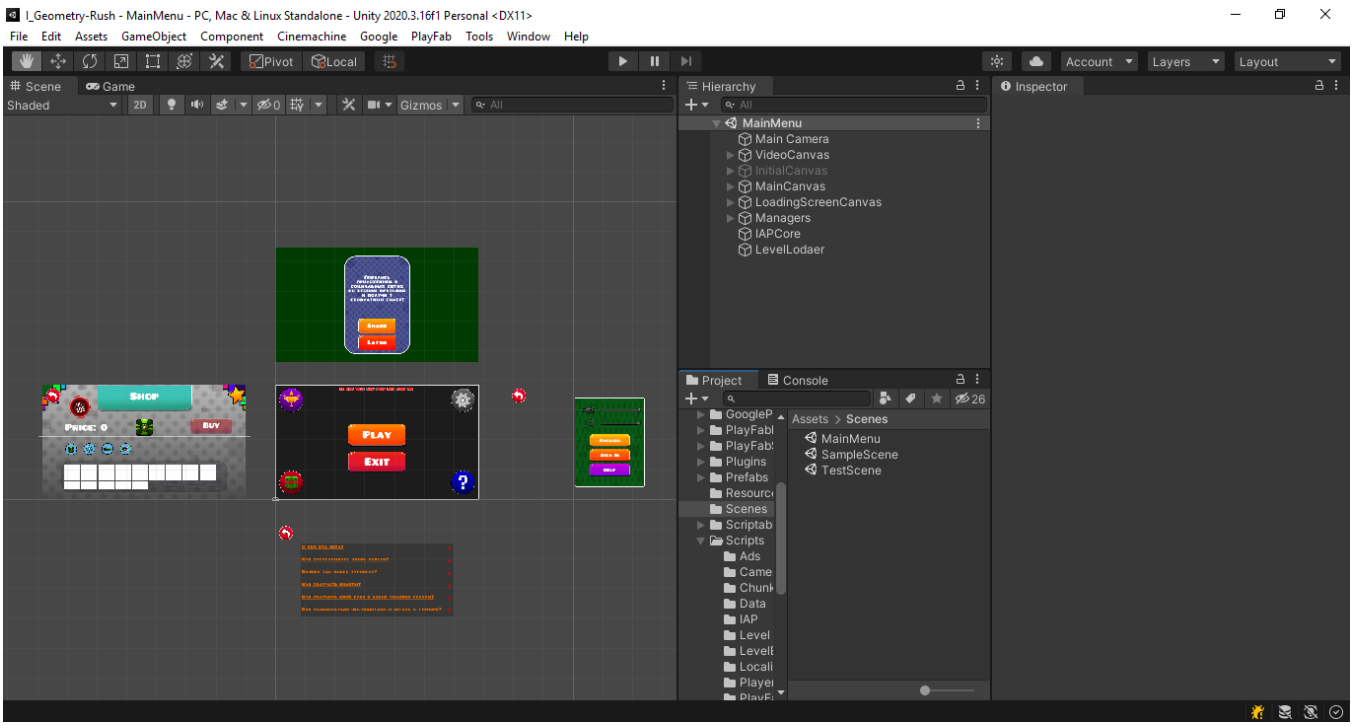
Мал. 2.2. Порівняння підходів у проектуванні ігрових об’єктів

Крім того, програміст має можливість писати код, реалізуючи власну систему компонентів, але Unity вже має дуже надійну версію такої системи, органічно інтегровану з візуальним редактором. Розробник має можливість не тільки програмувати управління компонентами, а й встановлювати і розривати зв'язки між ними в редакторі. Можливості розвитку не обмежуються складанням елементів з готових деталей; у коді розробник може використовувати успадкування та всі похідні від нього шаблони проектування.

Інтерфейс Unity розділений на кілька розділів: вкладка «Scene», вкладка «Game», панель інструментів, вкладка «Hierarchy», панель «Inspector», вкладки «Project» і «Console» (Малюнок 2.3).

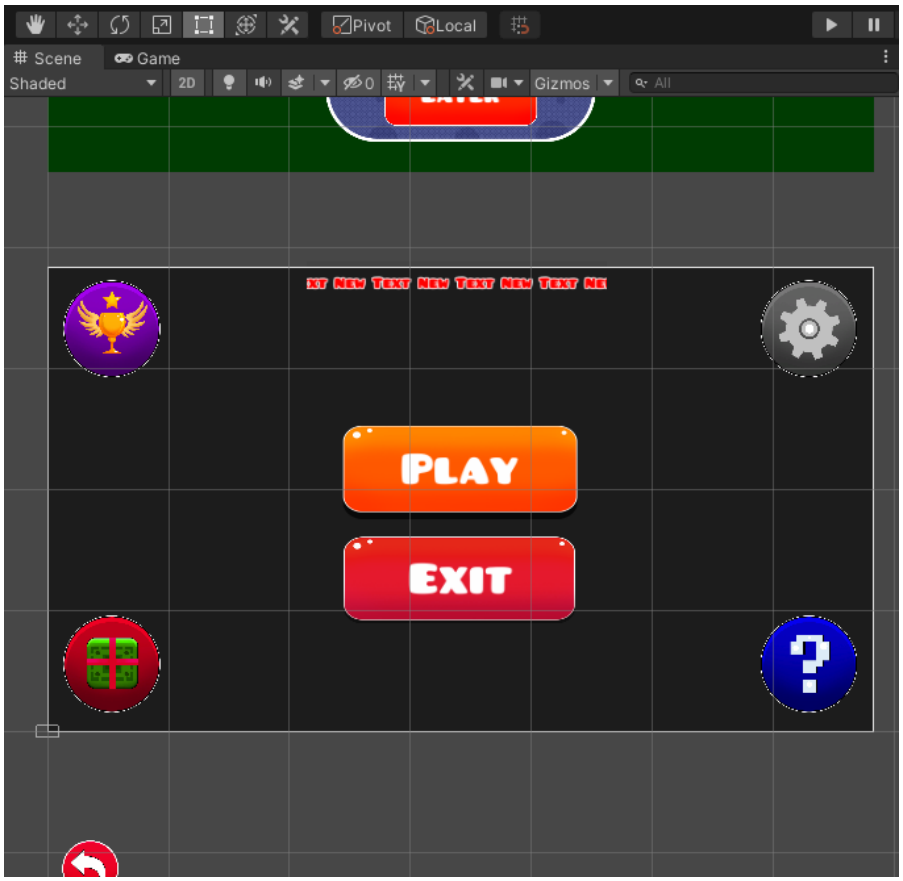
Кожна частина інтерфейсу має своє призначення, і всі вони відіграють важливу роль у процесі розробки гри:

– файли відображаються на вкладці Poject;
– об’єкти, розміщені на сцені, відображаються на вкладці «Scene»;
– панель інструментів містить елементи керування сценою;
– на вкладці Hierarchy можна змінювати відносини між об'єктами;
– панель Inspector надає інформацію про виділені об’єкти, а також про пов’язаний з ними код;
– тестування проекту виконується на вкладці Game, при цьому на вкладці Consol відображається повідомлення про помилки.



Мал. 2.3. Інтерфейс Unity

У центрі інтерфейсу знаходиться вкладка «Scene» (малюнок 2.4). Тут ви можете побачити, як виглядає ігровий світ, і розмістити об'єкти на сцені. Повинно бути зрозуміло, що все, що відображається на цій вкладці, буде відрізнятися від того, що відображається під час гри. Подання гри відображається на вкладці «Game», яка зазвичай знаходиться поруч із вкладкою «Scene». Після запуску гри на вкладці Game починає відображатися ігрове представлення.

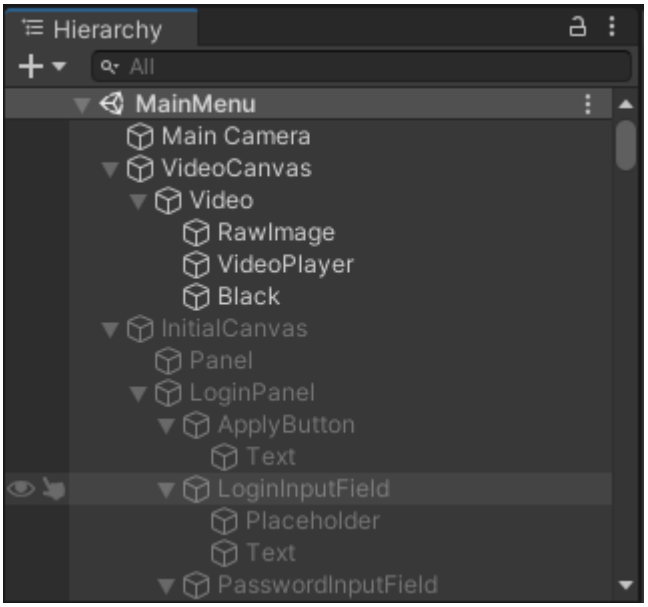


Мал. 2.4. Частина редактора, яка відображає лише панель інструментів і вкладка «Scene».

Вкладка Hierarchy (малюнок 2.5) містить список усіх присутніх на сцені об’єктів у вигляді дерева, гілки якого вкладені відповідно до ієрархічних

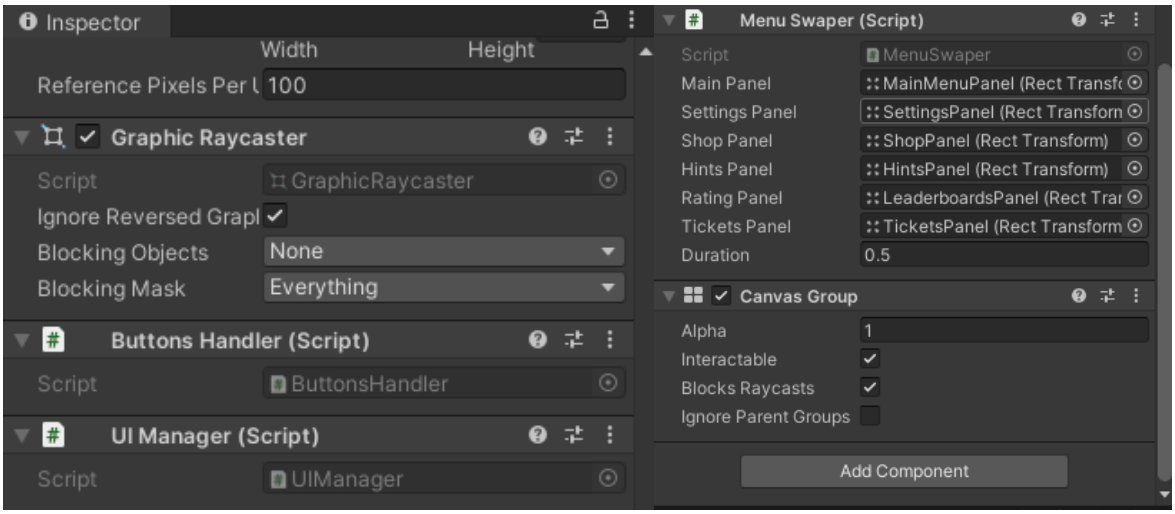
зв’язків між об’єктами. Вона надає можливість виділення об’єктів по іменах, позбавляючи від необхідності пошуку об’єкта на сцені. Ієрархічні зв’язки поєднують об’єкти один з одним.

На панелі Inspector (малюнок 2.6) відображаються дані поточного вибраного об’єкта. Після вибору різних об’єктів розробник може відразу побачити, як змінюється зовнішній вигляд панелі Inspector. Інформація, що відображається, зазвичай є списком компонентів, і розробник має можливість додавати нові компоненти до об’єктів або видаляти наявні.



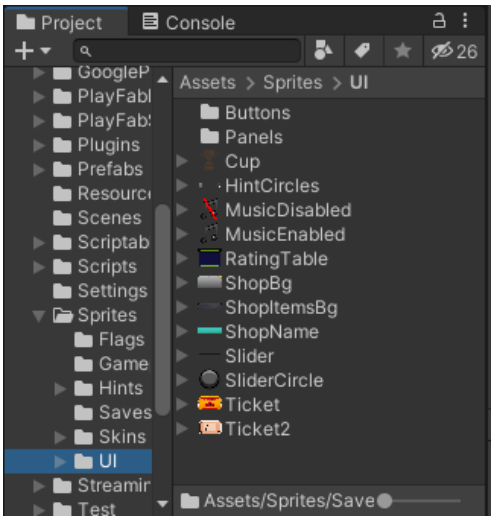
Мал. 2.5. Частина редактора, яка відображає лише вкладку Hierarchy

Усі об’єкти в грі містять принаймні один компонент – Transform, тому панель Inspector завжди відображає принаймні інформацію про положення та орієнтацію вибраного об’єкта. Багато об’єктів мають повний список компонентів, включаючи сценарії, пов’язані з цими об’єктами.



Мал. 2.6. Частина редактора, яка відображає лише вкладку Інспектор

У нижній частині екрана вкладки Project і Console показані на малюнках 2.7 і 2.8. На вкладці Project відображаються всі ресурси (графічні фрагменти, файли скриптів, префаби та ін.) проекту. У лівій частині цієї вкладки є список папок проекту; після вибору папки її файли з’являться праворуч. На вкладці Project також відображаються файли, які не входять до жодної сцени. Створення сцен також відображається на цій вкладці.



Мал. 2.7. Частина редактора, яка демонструє тільки вкладку Project та Console

Вкладка Console - панель, на якій відображаються повідомлення, пов'язані з кодом. Іноді вони навмисно розміщуються розробником у повідомленнях відладчика, іноді це повідомлення Unity, які з’являються, коли в сценарії, написаному розробником, виявляються помилки.

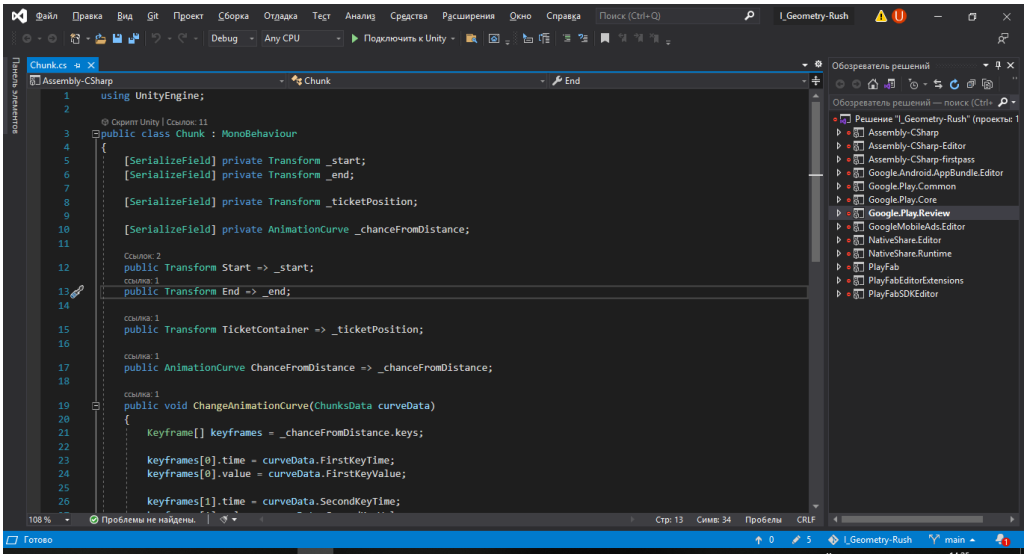
Щоб створити будь-яку гру в середовищі Unity, розробник повинен мати принаймні одну з доступних (на Unity) мов програмування: C# або JavaScript. Мова програмування C# має багато переваг перед JavaScript і набагато менше недоліків. Однією з переваг є те, що C # суворо типізований, чого не можна сказати про JavaScript. Наразі серед розробників існують різні думки щодо того, чи є динамічна перевірка типів найкращим підходом, як-от розробка веб-додатків, але під час написання програм для певних платформ (наприклад, Windows) це часто вигідно, а іноді навіть потрібна статична типізація. Unity навіть додала директиву #pragma, яка забезпечує статичну перевірку типів у JavaScript. Хоча технічно прийнятний, він порушує один з основних принципів JavaScript.

Поведінка об'єктів у грі контролюється компонентами (Components), які з ними пов'язані. Ядро Unity дозволяє розробнику створювати власні компоненти за допомогою скриптів (Scripts). Вони дозволяють активувати події в грі, змінювати налаштування компонентів і будь-яким чином реагувати на введення користувача.

Програмування не виконується в самому середовищі Unity, код існує у вигляді окремих файлів, розташування яких надасть сам розробник Unity. Файли сценаріїв можна створювати в додатку Unity, але в обох випадках розробнику знадобиться використовувати текстовий редактор або IDE, щоб написати код для цих спочатку порожніх файлів.

Unity постачається з Microsoft Visual Studio 2019, інтегрованим середовищем розробки (IDE) з підтримкою C# (Малюнок 2.8). Microsoft Visual Studio 2019 об’єднує файли в групи, які називаються рішеннями (solution). Інструмент Unity генерує зміст всіх файлів сценаріїв в рішення автоматично.

Створення сценаріїв здійснюється безпосередньо всередині Unity. Розробник може створити скрипт за допомогою меню Create у верхньому лівому куті панелі Project або вибравши в головному меню Assets > Create > C# Script (або JavaScript).



Мал. 2.8. Інтерфейс Microsoft Visual Studio 2019

У папці, вибраній на панелі Project, створюється новий сценарій. Назва нового сценарію буде виділено, і вам буде запропоновано ввести нову назву (малюнок 2.9).

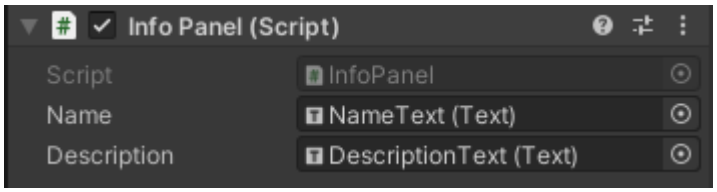
Мал. 2.9. Створення нового сценарію InfoPanel.cs

Коли ви двічі клацнете по скрипту в Unity, він відкриється в редакторі скриптів. За замовчуванням Unity використовуватиме Visual Studio 2019, але розробник має можливість вибрати будь-який редактор на панелі External Tools у налаштуваннях Unity.

Сценарій взаємодіє з внутрішніми механізмами Unity для створення класу, успадкованого від вбудованого класу з назвою MonoBehavior. Слід вважати клас, як план компоненту нового типу, який може бути прикріплений до ігрового об’єкту. Кожен раз, коли розробник додає сценарій до ігрового об’єкта, створюється новий екземпляр об’єкта, визначеного цим планом. Ім’я класу береться з імені, зазначеного під час створення файлу. Ім’я класу та ім’я файлу мають бути однаковими, щоб сценарій був приєднаний до ігрового об’єкта.

Дві функції, визначені в класі, слід розглядати окремо. Функція Update — це місце, куди можна помістити код, який оброблятиме оновлення кадрів для ігрового об’єкта. Це може бути рух, тригер, дії та відповідна реакція на введення користувача, в основному все, що потрібно обробити під час гри. Щоб Update виконувало свою роботу правильно, часто буває корисно форматувати змінні, читати властивості та спілкуватися з іншими ігровими об’єктами перед виконанням будь-якої дії. Функція Start викликає Unity перед початком гри (тобто перед першим викликом функції Update) і є ідеальним місцем для ініціалізації змінних.

Сценарій визначає лише план компонента, і тому код не активується, якщо екземпляр сценарію не приєднано до ігрового об’єкта. Розробник може прикріпити сценарій, перетягнувши елемент сценарію до ігрового об’єкта на панелі Hierarchy або через вікно інспектора вибраного ігрового об’єкта. У меню Component також є підменю Scripts, яке містить усі сценарії, доступні у створеному проекті. Екземпляр скрипта виглядає так само, як і інші компоненти у вікні Inspector (малюнок 2.10).

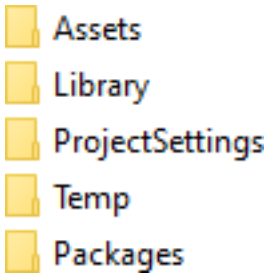


Мал. 2.10. Скрипт NewBehaviourScript.cs у представленні компонента

Після приєднання скрипт запуститься, коли ви натиснете кнопку Play та запустите гру.

2.5 Опис структури обчислювальної системи та алгоритмів її роботи

Розроблена конструкція має кореневу структуру, показану на малюнку 2.11. Список кодів наведено в Додатку А.



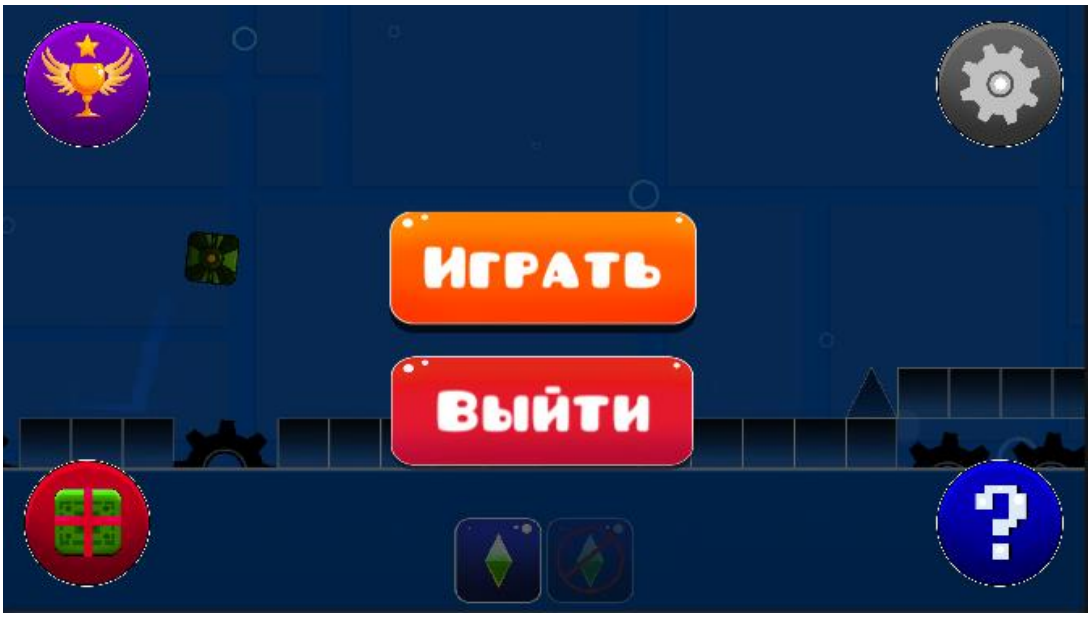
Мал. 2.11. Структура проекту розробленого ігрового додатку

Основні папки проекту:

- Assets – ця папка містить усі файли сценаріїв, матеріали та 3D-об’єкти;
- Assets – папка містить метадані, кеш та інші файли проекту;
- ProjectSettings - ця папка містить файли конфігурації;
- Temp - папка зберігає тимчасові файли;
- Packages — це спеціальна папка з файлами ядра Unity.

Для розробки ігрового додатка були розроблені сценарії керування компонентами (Components). Вони запускають ігрові події, змінюють і встановлюють певні значення параметрів компонентів і можуть будь-яким чином реагувати на дії користувача.

Ігровий додаток забезпечено головним меню, в якому гравець може вибрати режим гри, потрапити у внутрішньоігровий магазин, вибрати оптимальні налаштування для гри та дізнатися докладнішу інформацію про гру, зображені на малюнку 2.12.

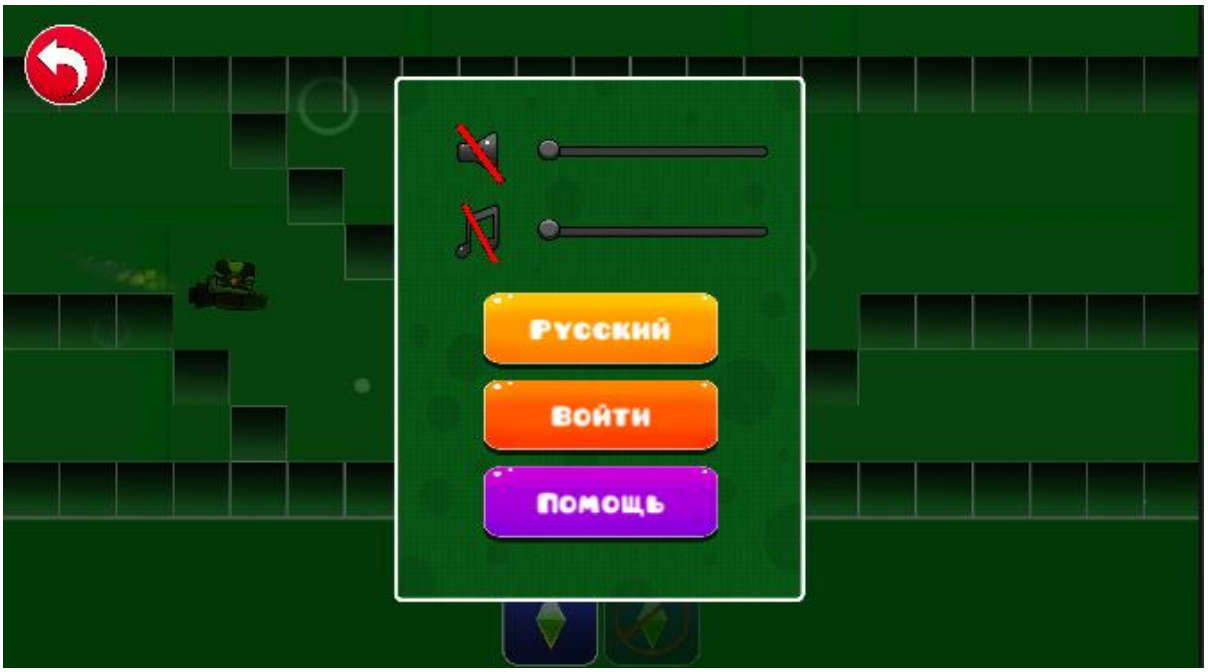


Мал. 2.12. Ігрова сцена із спроектованим прототипом головного меню

При натисканні на нижню праву кнопку з питанням активується скрипт з анімацією, який висуває вікно з популярними питаннями та відповідями цієї гри. При натисканні на + будь-якого пункту також відкривається шторка з відповіддю на будь-яке питання (малюнок 2.13).

Мал. 2.13. Екран гри з питаннями та відповідями

При натисканні на верхню праву кнопку активується анімація висування налаштувань (малюнок 2.14), у якому гравець може змінити настройки гучності (музики та загальних звуків), змінити мову, зареєструвати свій обліковий запис або отримати підтримку. Функція підтримки поки що недоступна, т.к. необхідно підключати свою пошту, а реєстрація фейкова гравця, т.к. вимагає хмарних синхронізацій із базами даних.



Мал. 2.14. Екран налаштувань

При натисканні на ліву нижню кнопку запускається анімація висунення магазину (малюнок 2.15), в якому гравець може вибрати різну категорію скінів для свого персонажа та придбати його для подальшого використання. Купівля поки що фейкова, т.к. передбачає впровадження SDK від Google та Appstore для працездатності покупок (Innap).

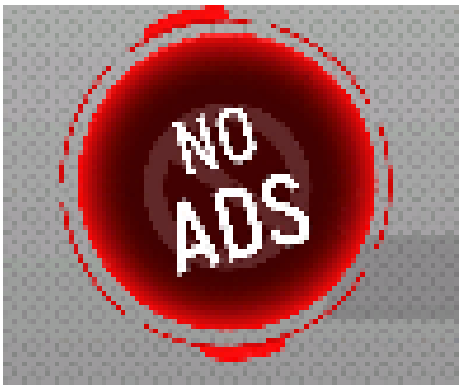
Мал. 2.15. Екран магазину

При натисканні на будь-яку з категорій перемикається панель з різними товарами, які гравець може купити (малюнок 2.16).. Вибравши певний товар та натиснувши відповідну кнопку у гравця з'являється фейкова панель покупки. Натиснувши Buy скін відкривається і стає доступним для використання (малюнок 2.17).

Мал. 2.16. Екран купівлі скіна

Мал. 2.17. Придбаний товар

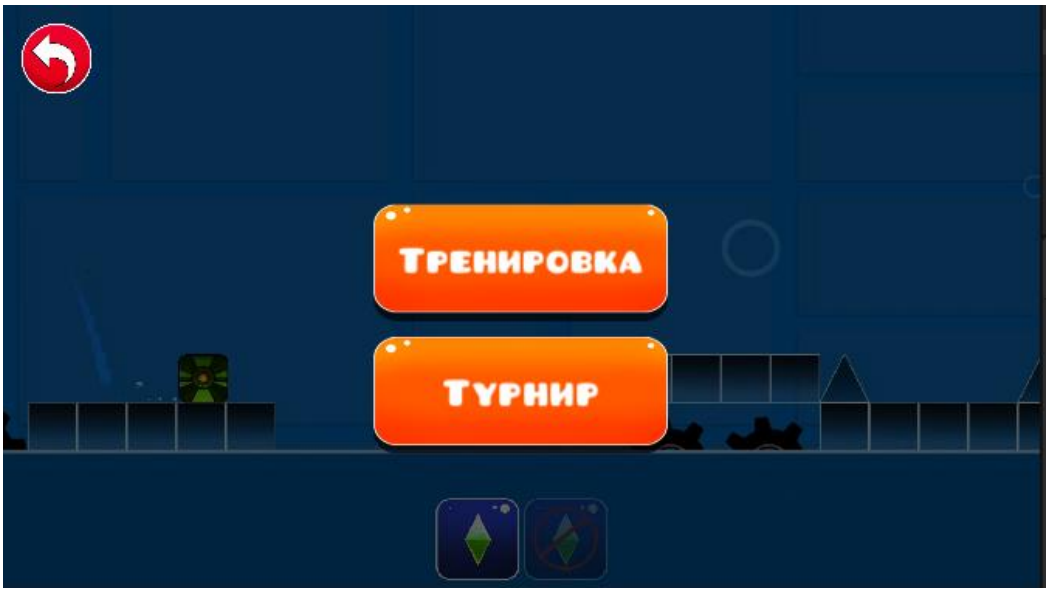
Також на екрані магазину є кнопка-заглушка (малюнок 2.18) для майбутньої функції відключення реклами. Поки кнопка нічого не призводить, т.к. SDK реклами не впроваджено у проект. Ліва верхня кнопка також є заглушкою (малюнок 2.19) і буде в майбутньому відображати статистику гравців, що авторизувалися в турнірах.



Мал. 2.18. Кнопка відключення реклами

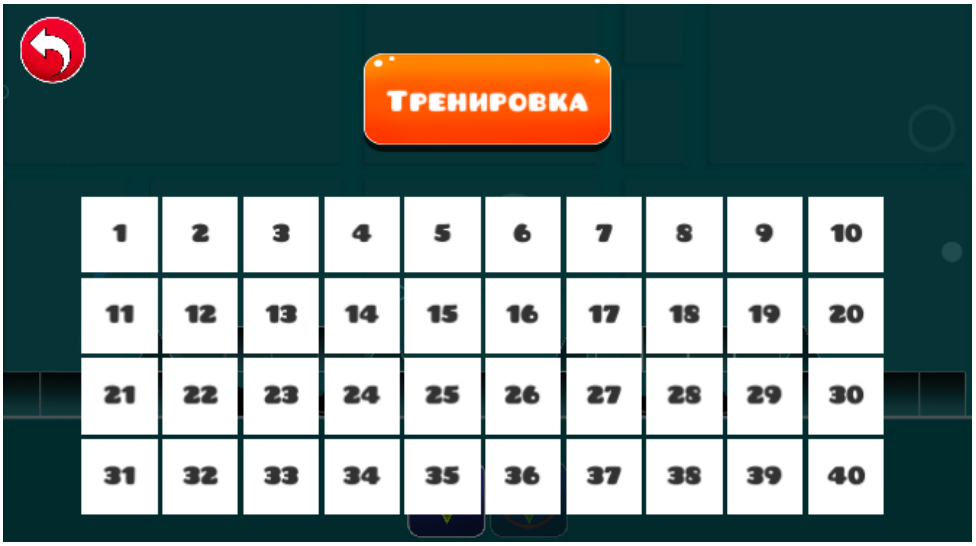
Мал. 2.19 Кнопка статистики турнірів

Після натискання на кнопку "Play" UI елементи головного меню змінюються на 2 інші кнопки: тренування та турніри (малюнок 2.20). Кнопка турніру є заглушкою, т.к. вимагає авторизації користувачів та прорахування особистих параметрів з облікових записів користувачів.



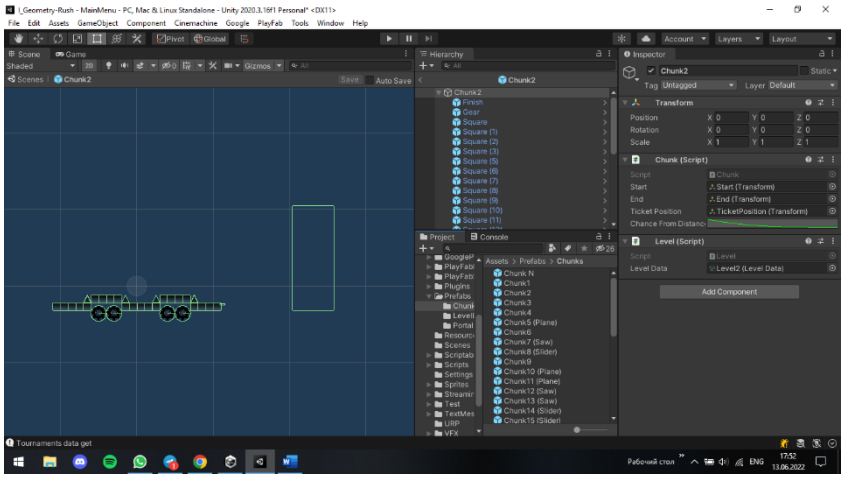
Мал. 2.20. Екран вибору типу гри

Після натискання кнопки "Тренування", гравець потрапляє в меню з осередками-рівнями та великою кнопкою "Тренування" (малюнок 2.21). Саме тренування складається з набору попередньо встановлених рівнів, які гравець може пройти самостійно поодинці в кожному з осередків. Ці рівні генеруються випадково в режимі тренування і підставляються один за одним, щоб створити ефект безкінечного рівня (Endless runner).

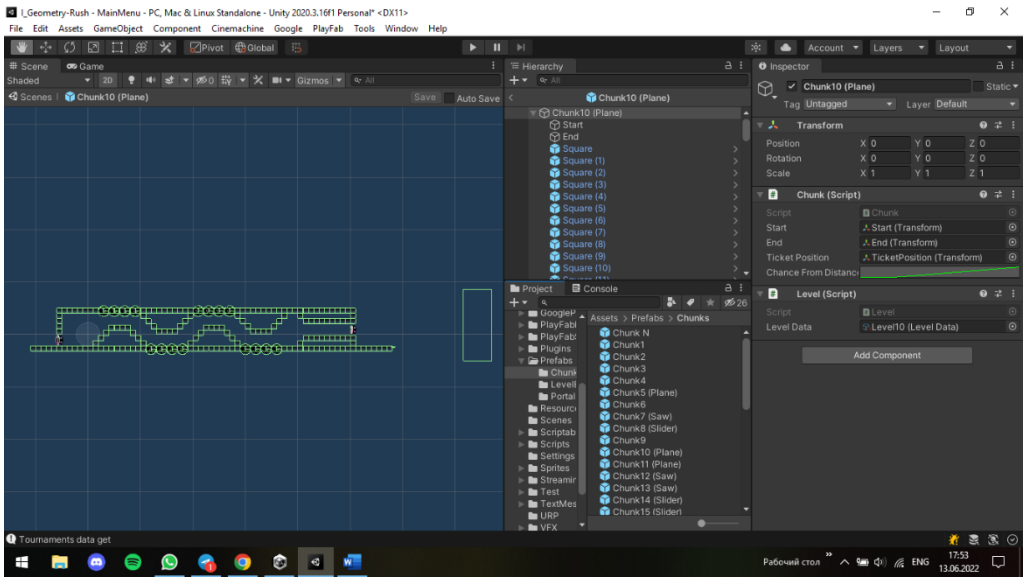


Мал. 2.21. Екран тренувань

Кожен рівень зібраний з різних спрайтів і наборів готових анімацій, наприклад, на малюнку 2.22 і 2.23 показано 2 з 40 так званих чанків - встановлених рівнів, які завантажуються на сцені тренування у випадковому порядку.



Мал. 2.22. Відкритий префаб Chunk2



Мал. 2.23. Відкритий префаб Chunk10 (Plane)

Кожен із чанків складається своїм набором спрайтів (Sprite) та префабів (Prefab) для повноти рівня.

"Спрайт" - 2D графічні об'єкти, що використовуються для персонажів, бутафорії, снарядів та інших елементів 2D геймплею. Графіка виходить із растрових зображень - Texture2D. Клас Sprite насамперед визначає частину зображення, яка має бути використана для певного спрайту. Потім ця інформація використовується компонентом SpriteRenderer на ігровому об'єкті для відображення графіки.

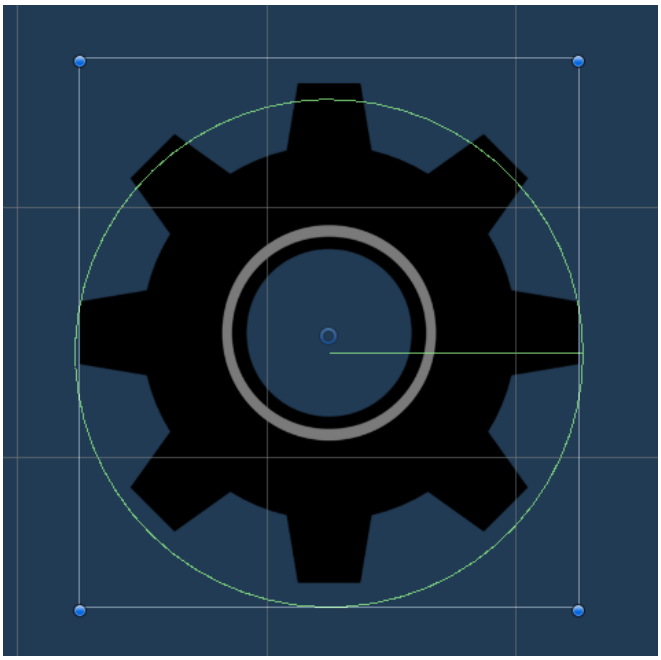
Prefab - це особливий тип ассетів, що дозволяє зберігати весь GameObject з усіма компонентами та значеннями властивостей. Префаб виступає в ролі шаблону для створення екземплярів об'єкта, що зберігається в сцені.

Всі рівні складаються з базових префабів-пасток, які включають певну картинку (Sprite) і її відповідний колайдер. Пересікаючись із колайдером (Colider), гравець програє у рівні.

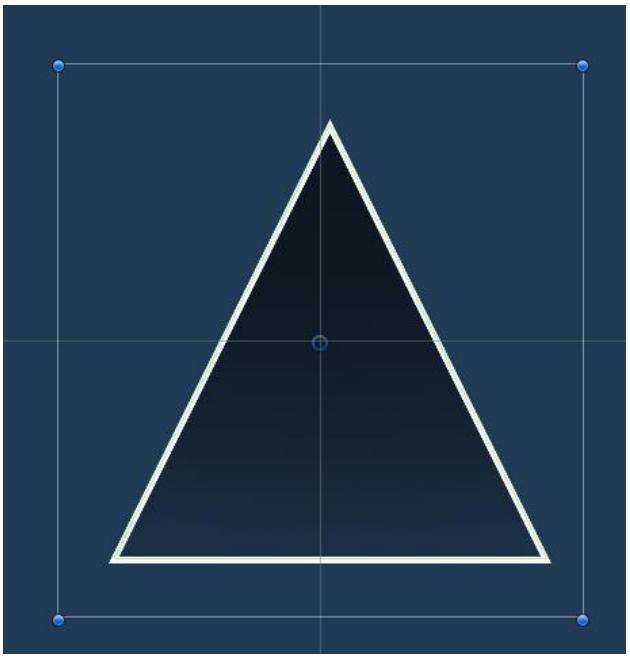
Colider - об'єкти, які визначають форму об'єкта для цілей фізичних зіткнень. Коллайдер, який є невидимим, не обов’язково повинен мати ту ж

форму, що й сітка об’єкта, і насправді грубе наближення часто є ефективнішим і нерозрізненим у ігровому процесі.

На малюнку 2.24 зображено префаб з однією з пасток. У ньому є спрайт шестерні, що відображає візуальну частину об'єкта та зелений контур колайдера, що відображає фізичну складову об'єкта.

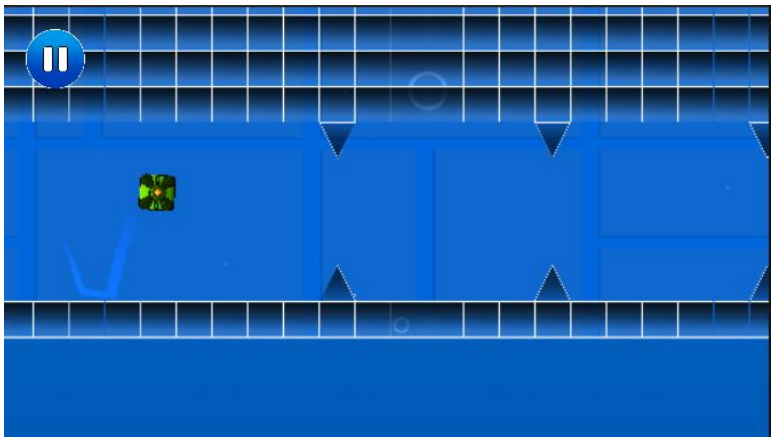


Мал. 2.24. Префаб з однією з пасток



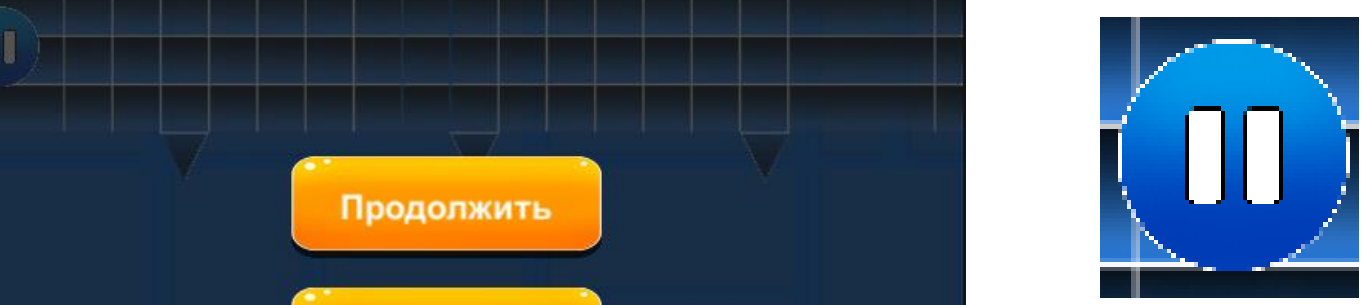
Мал. 2.25. Інший префаб з однією з пасток

При натисканні на будь-яку з кнопок, яка відповідає рівням, запускається сцена з певним чанком, де гравець може спробувати його пройти (малюнок 2.26).



Мал. 2.26 Екран з чанком-рівнем

На будь-якому екрані з грою (чи тренування або окремий рівень-чанк) присутня кнопка паузи (малюнок 2.27), натиснувши яку гравець може призупинити ігровий процес, повернутися в головне меню або продовжити поточну гру (малюнок 2.28).



Мал. 2.27. Кнопка паузи

Мал. 2.28. Екран паузи

У грі представлена найпростіша модель руху персонажа. Сам персонаж складається з певного спрайту (Sprite), системи частинок (Particle System) та підключених до нього скриптом.

Рухається персонаж щодо осі Ox із постійною швидкістю та нульовим прискоренням. При натисканні на ЛКМ (ліва кнопка миші) або на тачскрін пристрою, на якому запускатиме гра, персонаж підстрибує вгору, отримуючи імпульс поштовху. Також спрацьовує метод, який перевертає персонажа у певні моменти свого руху на 90 градусів за годинниковою стрілкою. Це додає різноманітність візуальної складової ігор жанру Arcade.

Під час розробки гри були створені такі скрипти:

- CameraMovement.cs - керує камерою;
- InfiniteStarfield.cs - керує системою частинок;
- Platforma.cs - в десять сценарії Він був реалізовано механіка взаємодіяплатформи з характером гравця;
- Player.cs - цей скрипт надає гравцеві механіку гри;
- Rotator.cs - цей скрипт реалізує механіку кришталевої анімації;
- GameManager.cs - скрипт контролює генерацію платформ, які забезпечують умовно нескінченну гру;
- MainMenu.cs - функціонал ігрового меню, реалізований в скрипті.

Скрипти Unity є компонентами (але лише ті, які успадковані від класу MonoBehavior - базового класу компонентів сценарію). Цей клас визначає, як компоненти приєднуються до ігрових об’єктів. Успадкування від цього класу надає кілька автоматично доданих методів. Це метод Start (), який викликається, коли об’єкт активується (і зазвичай з’являється після завантаження вмісту на рівні об’єкта), і метод Update (), який викликається в кожному кадрі. Таким чином, код запускається після його розміщення в цих попередньо встановлених методах.

Щоб керувати камерою в грі, потрібно керувати її налаштуваннями на вкладці «Інспектор» за допомогою сценарію CameraMovement.cs. У методі void Start() ми призначаємо змінну значення shift - різницю між значеннями компонентів перетворення, які містять координати розташування ігрових об'єктів. Потім у методі void Update ми перевіряємо здатність гравця рухатися. Цей тест виконується для того, щоб, коли гравець падає з платформи, камера перестала його стежити. Якщо цей сценарій не працював, ігрова камера завжди залишалася б на одному і тому ж місці, поки персонаж гравця рухався, і гра закінчувалась через кілька секунд після її початку.

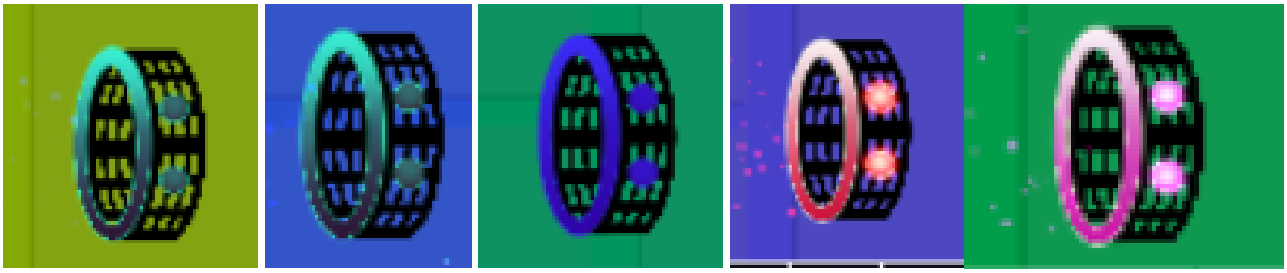
Particles реалізована в скрипті PlayerMover. Ця система відповідає за генерацію візуальних частинок, які з'являються в ігровому просторі. Завдяки створеній перспективі молекули забезпечують краще відчуття тривимірності навіть у двовимирній грі. Розроблений скрипт визначає правила генерації візуальних частинок, а саме: встановити максимальну кількість частинок, їх розмір, координати генерації, визначити відстань однієї частинки від іншої, а також щільність розподілу в ігровому просторі кімнати.

Після налаштування камери потрібно дати гравцеві можливість переміщатися в певне місце і взаємодіяти з об’єктами в грі. Це робиться за допомогою сценарію PlayerMover. Є в цьому скрипті

методи та змінні, які, наприклад, переміщують гравця по ігровому простору. Розроблені методи реалізують внутрішню ігрову механіку взаємодії гравця з ігровим простором та об’єктами всередині нього. Наприклад, чи перевіряється зіткнення колайдера персонажа з перешкодами, шипами тощо? Якщо так, гра закінчується, спрацьовує певна анімація системи частинок (Particle System), камера зупиняє свій рух і гравець повертається до головного меню. За зіткнення об'єктів у скриптах відповідає функція OnCollisionEnter().

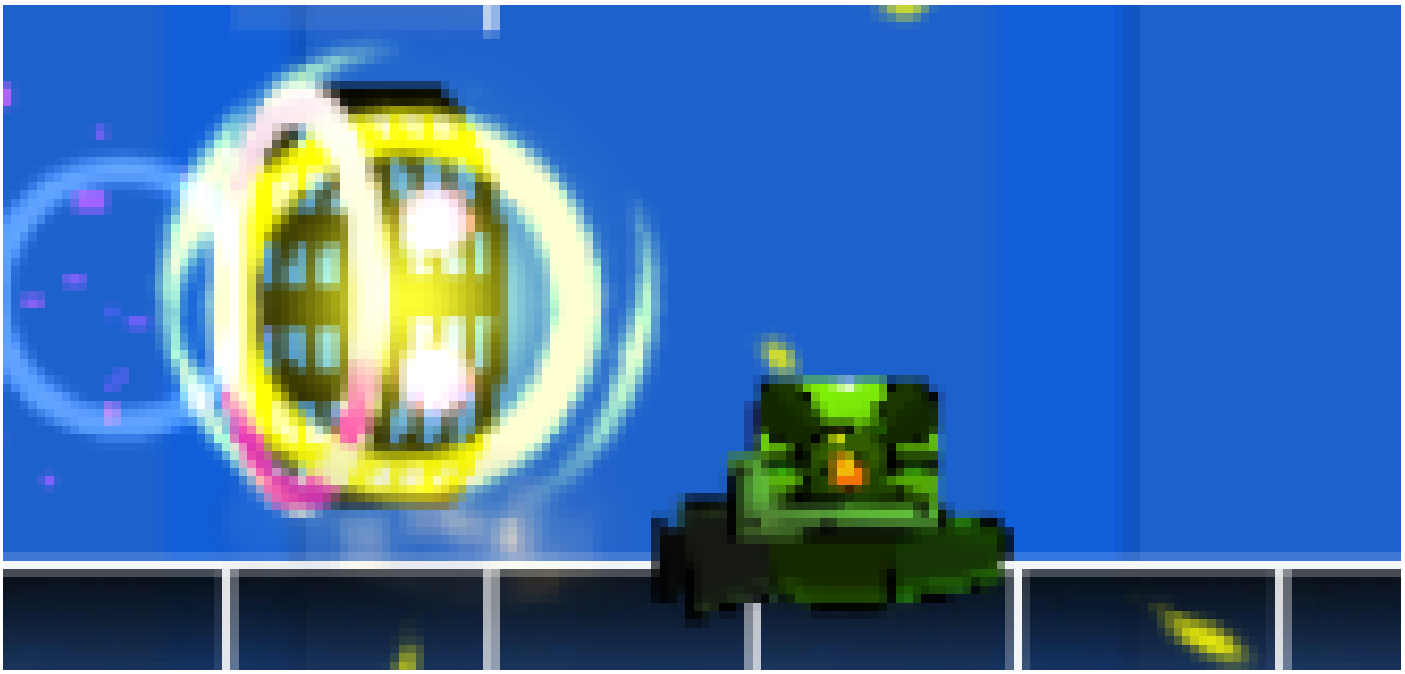
Мал. 2.29. Момент зіткнення гравця з перешкодою

За допомогою перевірки перетину колайдерів також можна налаштувати різноманітні події. Наприклад, при перетині колайдера гравця та колайдера спрайту кільця (певного ігрового об'єкта), спрацьовує вже функція OnTriggerEnter(), яка не зупиняє ігровий процес, а змінює механіку всього геймплею. Усього існує 5 видів таких кілець (малюнок 2.30).



Мал. 2.30. 5 видів кілець

У кожного кільця є своє призначення, всі вони мають унікальний спрайт різного кольору і набір VFX анімацій, створених за допомогою Particle System. При проходженні гравця через кільце активується метод, який змінює механіку пересування та активує анімацію Particle System (малюнок 2.31).

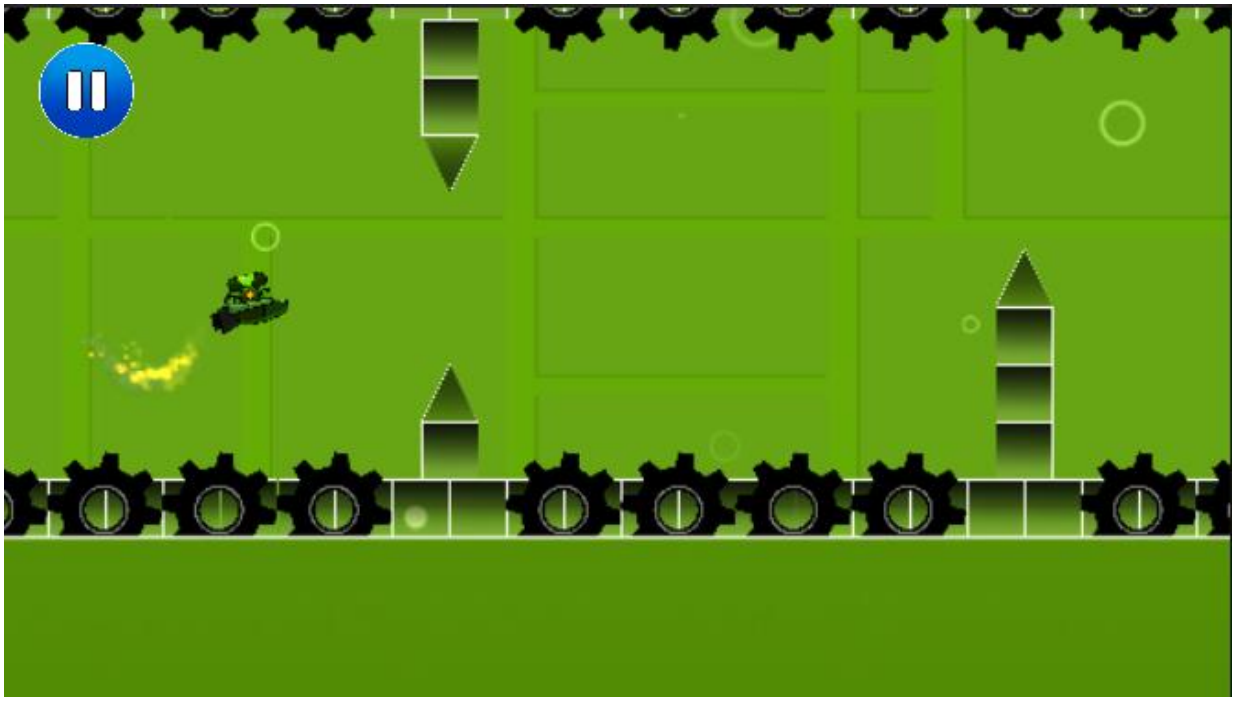


Мал. 2.31. Активація порталу

Усього існує 5 видів різного геймплею: Saw, Slider, Plane, Mini та Basic. Проходячи через відповідні кільця у певних чанках-рівнях, можна активувати певні механіки управління.

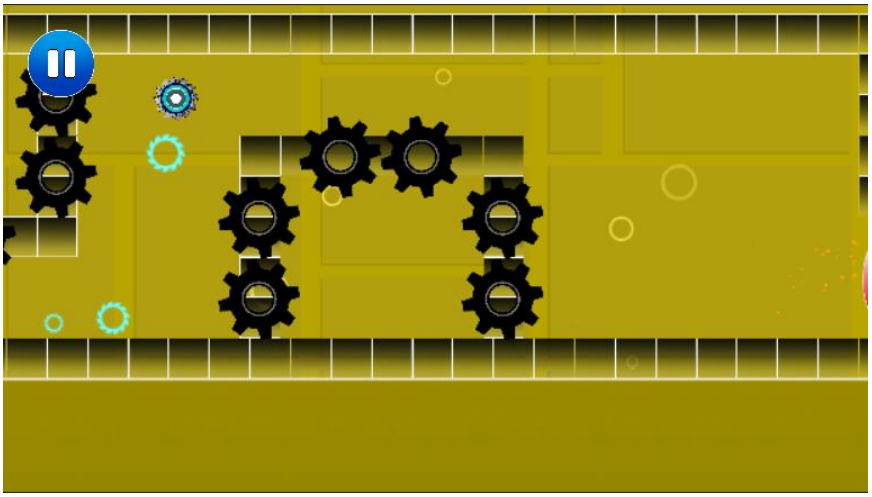
У механіці Plane (малюнок 2.32) спрайт головного персонажа змінюється на ракету, також змінюється управління зі звичайного стрибка та імпульсу,

натисканням ЛКМ, на керований політ з утримання ЛКМ. Утримуючи ЛКМ, персонаж отримує параметр прискорення з вектором нагору (0.1), завдяки чому персонаж може підлітати нагору, поки гравець не відпустить ЛКМ. Після відтискання лівої кнопки миші персонаж падає під дією гравітації. Завдяки цій механіці гравець повинен відхилятися від об'єктів, щоб набрати максимальний рахунок. Також при утриманні ЛКМ із певного місця спрайту персонажа вилітають частки, згенеровані заздалегідь за допомогою компонента Particle System.



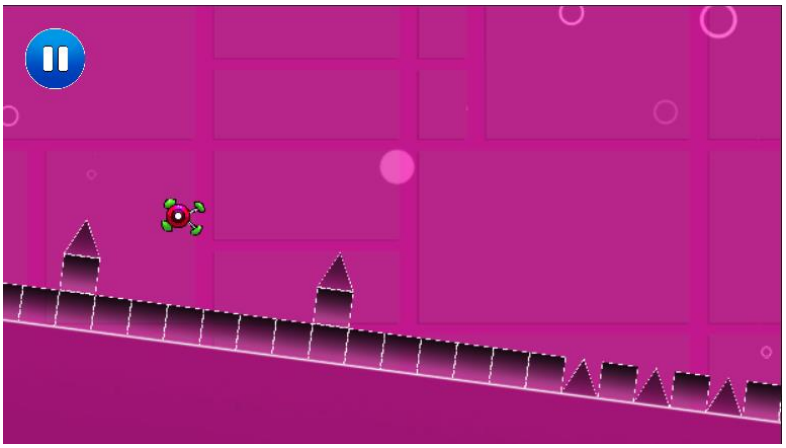
Мал. 2.32 Відображення механіки Plane

У механіці Saw (малюнок 2.33) спрайт персонажа змінюється на пилку. У головному скрипті управління персонажем (PlayerMover) активується функція, що постійно обертає спрайт за годинниковою стрілкою, щоб показати реалістичний рух пили. Сама механіка пилки обумовлюється зміною гравітації. Як тільки гравець натисне ЛКМ, ігровий об'єкт персонажа поміняє свої атрибути гравітації на негативні (-1), тим самим пила полетить нагору і основний геймплей відбуватиметься на верхній частині рівня.



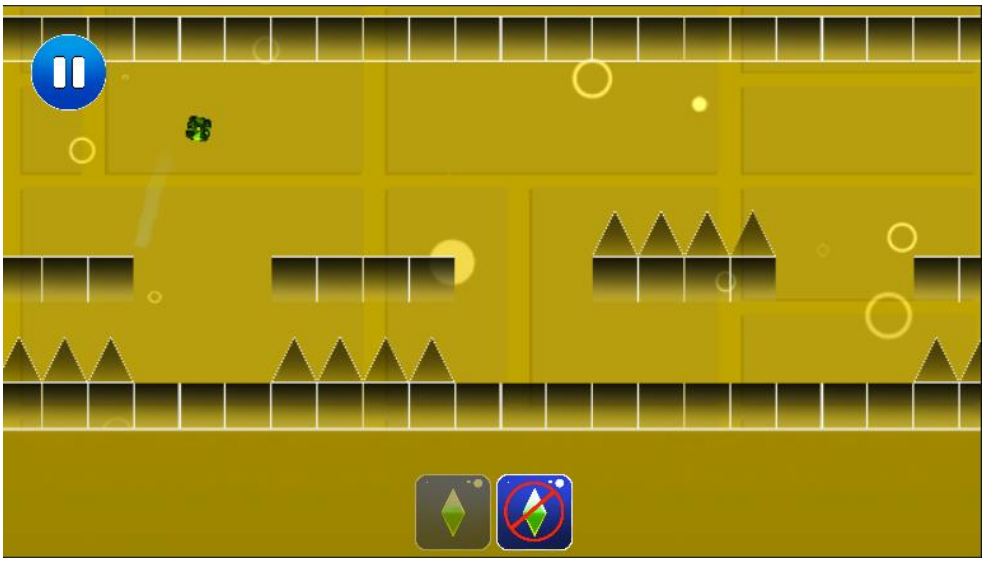
Мал. 2.33. Відображення механіки Saw

У механіці Slider спрайт персонажа змінюється на Slider (малюнок 2.34), і в залежності від чанка-рівня перснаж може отримати як позитивне, так і негативне миттєве прискорення до своєї швидкості руху по осі Ox. Також для додаткового реалізму змінюється нахил повороту основної камери у певний бік у відповідності з типом прискорення.



Мал. 2.34. Відображення механіки Slider

При проходженні через передостанній вид кільця спрайт персонажа просто зменшується в розмірах, завдяки параметру Scale в його компоненті Transform (рисунок 2.35). Інші атрибути персонажа, такі як: імпульс і висота стрибка, кут повороту, швидкість та гравітація залишаються незмінними.



Мал. 2.35. Відображення механіки Mini

Останнє кільце повертає вихідні розмір, спрайт, параметри та механіки руху персонажа.

Усі режими перемикаються через глобальний сценарій управління персонажем PlayerMover.cs. У ньому описана функція, що викликається при перетині колайдерів головного персонажа та будь-якого з кілець (малюнок 2.36). Після цього з об'єкта кільця передається параметр, який викликає відповідну функцію зміни керування (MovementType.Default, MovementType.Scaled, MovementType.GravityChanging, MovementType.Plane, MovementType.Slider).

```
switch (_moveType)
{
    case MovementType.Default:
        DefaultMovement();
        break;
    case MovementType.Scaled:
        DefaultMovement();
        break;
    case MovementType.GravityChanging:
        GravityMovement();
        break;
    case MovementType.Plane:
        PlaneMovement();
        break;
    case MovementType.Slider:
        SliderMovement();
        break;
}
```

Мал. 2.36. Відображення функцій вибору механік у коді

2.6 Обґрунтування та організація вхідних та вихідних даних програми

Ігровий движок Unity Engine працює з моделлю Event.

Події – це події, які запускаються з коду гри, якщо певна умова виконується. Вони дозволяють виконувати ряд дій у відповідь на певні події, що відбуваються в грі.

Наприклад, для обробки події клацання лівою кнопкою миші розроблений ігровий додаток реалізує керування персонажем гравця. У цій програмі вхідним є ліва кнопка миші.

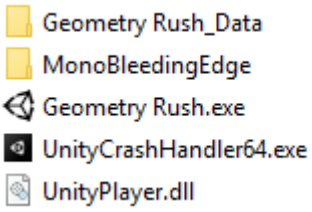
Користувач клацає лівою кнопкою миші на події з кодом ключа 0, потім обробник події ловить подію і викликає метод ChangeDirection (), який відповідає за зміну напрямку персонажа гравця.

Вихідні дані розробленого ігрового додатка можна розглядати як результат роботи камери. Камера – це пристрій, який знімає гравця та відображає світ. Налаштувавши і керуючи камерою, ви можете зробити презентацію гри дійсно особливою. На сцені можна мати необмежену кількість камер, але в проекті розробки гри є лише одна. Завдяки камері плеєр може переглядати візуальні дані з монітора.

2.7. Опис розробленого програмного продукту

Щоб перетворити розроблений ігровий додаток у продукт, готовий до клієнта, необхідно створити набір ігрових програм. Build — це набір ігрових програм, які гравці можуть використовувати. Упакування здійснюється завдяки функціональності Unity Engine. Процес пакування є автоматичним, вам просто потрібно ввести цільову платформу розгортання та натиснути Створити та запустити.

Результат ущільнення гра Програми є папку З готовий длявикористовується гравцями (малюнок 2.37).



Мал. 2.37. Результат набору ігрових програм для використання в іграх

Для роботи з програмою просто запустіть файл «GeometryRush.exe». Після цього запуститься ігровий додаток. У разі невідповідності мінімальним специфікаціям пристрою, на якому була запущена ця програма, користувач буде повідомлений.

2.7.1 Використані технічні заходи

При розробці та тестуванні системи була використана клієнтська персональна ЕОМ з наступними мінімальними характеристиками:

- процесор Intel i3;
- монітор 60 Гц;
- не менше 2000 Мб оперативної пам'яті;
- 3 ГБ безкоштовного місяця для ігрового движка;
- 229 МБ вільного місця для гри;
- клавіатура;
- миша.

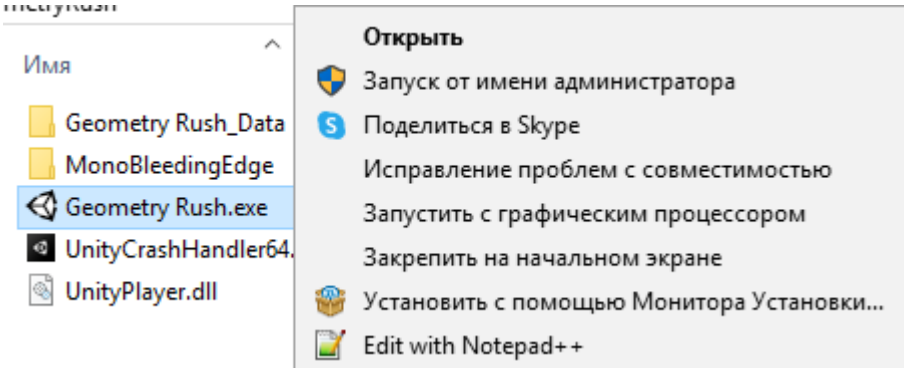
2.7.2 Використані програмні засоби

Для роботи програмного забезпечення необхідні наступні програмні інструменти:

- Версія Unity Engine не нижче 2020.3.16f1;
- Visual Studio версії 19 або вище.

2.7.3 Викли та завантаження програми

Для роботи з розробленим ігровим додатком потрібно запустити файл «GeometryRush» з папки ігрового додатка (малюнок 2.38).

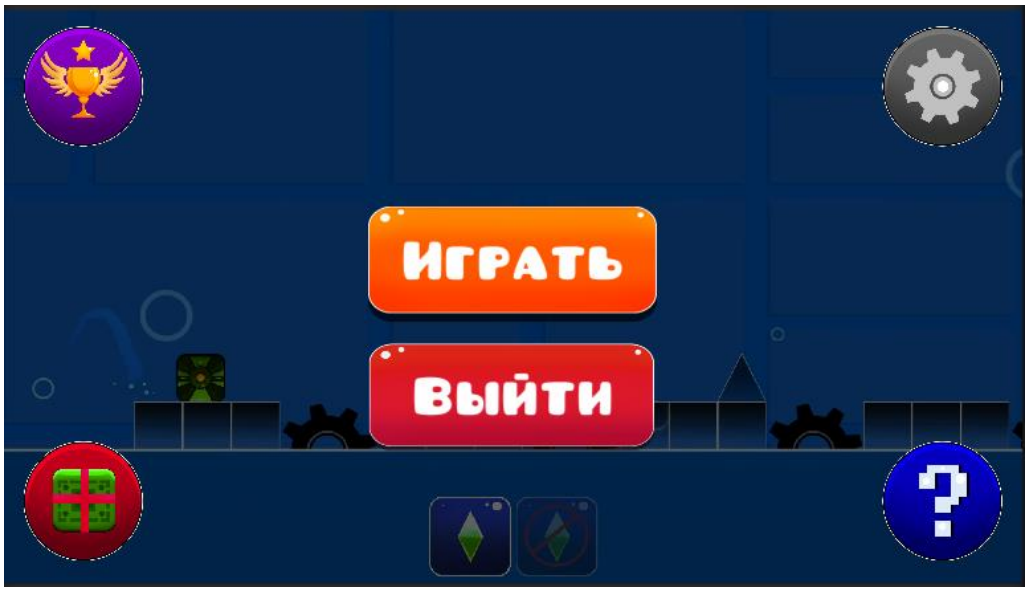


Мал. 2.38. Запуск ігрового додатку із папки

Після запуску файлу GeometryRush, користувач побачить екран завантаження гри (рисунок 2.39)

Мал. 2.39. Екран завантаження гри

Після успішного завантаження ігрової програми, якщо комп’ютер користувача відповідає вимогам, зазначеним у розділі 1.5.3., відкриється перша ігрова сцена - Головне меню (малюнок 2.40). Після цього користувач може відкривати різні вікна, прописані в попередніх розділах і грати в саму гру.



Мал. 2.40. Початок гри

РОЗДІЛ 3

ЕКОНОМІЧНИЙРОЗДІЛ

3.1. Розрахунокскладність і вартість розробки програмного забезпечення,

1. Вихідні дані:
2. орієнтовна кількість операторів програми - 1048;
3. коефіцієнт складності програми - 1,3;
4. поправочний коефіцієнт для програми під час її розробки - 0,07;
5. погодинна оплата програміста - 140 грн/год;
6. швидкість зростання вартості робочої сили через недостатній опис проблеми

   - 1,2;
7. кваліфікаційний коефіцієнт програміста, визначений стажем роботи за цією спеціалізацією - 1,2;
8. вартість години роботи за комп'ютером -50 грн/год.

Раціоналізація роботи в процесі розробки програмного забезпечення значно ускладнюється через творчий характер роботи програміста. Отже, складність розробки програмного забезпечення може бути розрахована на основі системи моделей з різними оцінками точності.

Складність розробки програмного забезпечення можна розрахувати за формулою:

$$m = \tau_O + \tau_a + \tau_a + \tau_n + \tau_{ідеальний} + \tau_{мене}$$, людино-години, (3.1)

де витрати на підготовку та опис посади (припускалися 50 людино-годин);

ty- витрати праці, пов'язані з вивченням алгоритму розв'язання задачі;

a- витрати праці, пов'язані з розробкою блок-схеми алгоритму;

місто-витрати плати за програмування за готовою блок-схемою;

Насправді- витрати праці, пов'язані з конфігурацією програми на комп'ютері;

td- витрати на оплату праці, пов'язані з виготовленням документації.

Вартість роботи компонента залежить від умовної кількості операторів у програмному забезпеченні, яке розробляється.

Умовна кількість операторів (брендів):

$$Q = q \cdot X\ 1 + (1 + \pi),$$

де q - розрахункова кількість операторів (1300);

C- коефіцієнт складності програми (1,6);

c- поправочний коефіцієнт програми під час її створення

(0,05). Отже, умовна кількість операторів у програмі:

$$Q = 1{,}3 \cdot 1048 \cdot (1 + 0{,}05) = 1430{,}52$$

Вартість робіт з розробки опису проблеми визначається з урахуванням специфікації опису та кваліфікації програміста:

$$m_{ty} = \frac{Q\Im}{(75..85)\ \square}\ \text{к}\ , \text{людино-години},$$

де В – швидкість збільшення витрат на оплату праці через недостатній опис проблеми;

к- кваліфікаційний коефіцієнт програміста, визначений стажем роботи за цією спеціальністю, для стажування від 3 до 5 років становить 1,2.

Припустимо збільшення витрат на оплату праці через недостатню посадову інструкцію на 50% або менше (В = 1,2). Враховуючи кваліфікаційний коефіцієнт k = 1,2, отримуємо витрати на оплату праці на складання опису завдання:

$ty = \frac{(1430 \cdot 1{,}2)}{(75 \cdot 1{,}2)}$ = 19 годин роботи

Вартість роботи з розробки алгоритму розв'язування задач визначається за формулою:

$$m_a = \frac{Q}{(20 \square 25) \square_{к}}, \text{ людино-години, (3.2)}$$

де Q — умовне число програмних операторів;

к- кваліфікаційний коефіцієнт розробника.

Підставивши відповідні значення у формулу (3.2), людино-години:

$$TAk = \frac{1165}{(20 \cdot 1.2)} = 49 \text{ годин роботи.}$$

Витрати на розробку програми за готовою схемою:

$$m_n = \frac{Q}{(20 \square 25) \square_{к}}, \text{ людино-годин.}$$

$$mн = \frac{(1165 \cdot 1{,}2)}{(20 \cdot 1.2)} = 58 \text{ годин роботи.}$$

Витрати на винагороду за встановлення програми на комп'ютер:

- за умови автономного налаштування одного завдання:

$$m_{ідеальний} = \frac{Q}{(4..5) \square_{к}}, \text{ людино-годин.}$$

$$m_{ідеальний} = \frac{1165}{(5 \cdot 1.2)} = 194 \text{ години роботи.}$$

ы

– підлягає комплексній модифікації завдання:

$$t_{відл}^{k} = 1{,}5 \; \square \; т_{ідеальний}, \text{ людино-годин.}$$

$$m_{тощо}к = 1{,}5 \cdot 194 = 291 \text{ людино-год.}$$

Витрати на оплату праці на підготовку документації визначаються за формулою:

$$m_{мене} = т_{тощо} + т_{робити}, \text{ людино-години,}$$

де*tdr*- складність підготовки матеріалів і рукописів:

$$m_{тощо} = \frac{Q}{(15..20) \; \square \; к}, \text{ людино-години,}$$

tdo- складність складання, друку та реєстрації документації:

$$m_{робити} = 0{,}75 \square \; т_{тощо}, \text{ людино-годин.}$$

Підставляючи відповідні значення, отримуємо:

$$tdr = \frac{1165}{(18 \cdot 1.2)} = 54 \text{ години роботи.}$$

$$tdo = 0{,}75 \cdot 316 = 41 \text{ людино-год.}$$

$$td = 54 + 41 = 95 \text{ годин роботи.}$$

Повертаючись до формули (3.1), отримуємо повну оцінку трудомісткості розробки програмного забезпечення:

m= 50 + 22 + 49 + 58 + 194 + 95 = 468 людино-годин.

3.2. Розрахувати вартість створення програми

Вартість розробки програмного забезпечення КПО включає вартість виконавця програми ЗЗП та вартість машинного часу, необхідного для встановлення програми на комп'ютері:

$К_{NA}=_{ЗП}+_{М.В}$, грн

Оплата праці підрядників визначається за формулою:

$ВІД_{ЗП}= t \cdot С_{PR}$, гривня,

де: t - загальна витрата праці, людино-год;

СПР- середня погодинна заробітна плата програміста, грн/год

Якщо врахувати, що середня погодинна заробітна плата програміста становить 58 грн/год, то отримаємо:

ЗЗП= 468 · 140 = 65 520 грн.

Вартість машинного часу, необхідного для налагодження комп'ютерної програми, визначається за формулою:

$ВІД_{тv}= т_{ідеальний} С_{мученик}$, грн., (3,3)

де *Насправді*- складність конфігурації програми на комп'ютері, год;

Побачити- комп. годин, грн./год. (13 грн./год.).

Підставивши відповідні значення у формулу (3.3), визначте витрати, необхідні для коригування машинного часу:

Змв= 224 · 50 = 11 200 грн.

Отже, вартість розробки програмного забезпечення:

КПО= 65 520 + 11 200 = 76 720грн.

Орієнтовний період розробки програмного забезпечення:

$$T= \frac{m}{Б_к{}^{™}{}_c}$$ пропускати.

де Bk - кількість акторів (дорівнює 1);

Fp- місячний фонд робочого часу (при 40-годинному робочому тижні). *Fp*= 176 годин).

Отже, вартість розробки програмного забезпечення:

$$T=\frac{468}{(1 \cdot 176)}\approx 3 \text{ місяці}$$

додаток: Ігровий додаток призначений для того, щоб надати гравцеві можливість заощадити час і насолоджуватися вільним часом. Ціна цієї програми 76 720 грн. і не потребує додаткових витрат на розробку проекту. Орієнтовний термін розробки - 3 місяці. Термін пов’язаний з великою кількістю операторів і включає час на дослідження та розробку алгоритму вирішення проблем, розробку проекту, розробку ігрового додатка та підготовку документації.

ВИСНОВКИ

Метою кваліфікаційної роботи є розробка 2D-ігрового додатка з жанру Arcade, який забезпечить користувачеві ігровий процес та дозволить скоротити час перебування на карантині. При розробці були враховані сучасні методи та інструменти для створення різноманітних ігор.

Розроблений дизайн призначений для розважальних цілей, але допомагає користувачеві розвивати навички мозку – покращувати реакцію та чутливість рухомих об’єктів у грі.

Програма працює на Android, що широко використовується цільовою аудиторією продукту. Розробка велася на високорівневій мові програмування C#, що дозволяє відносно точно маніпулювати ресурсами і дозволяє досягати достатніх результатів при швидкості роботи програми. Допоміжні методи були використані ігровим движком Unity Engine. Основним завданням користувача було приємно провести час.

«Економічний розділ» визначає складність розробки програмного забезпечення (468 людино-годин), орієнтовну вартість розробки програмного забезпечення (76 720грн.) та розрахунковий термін розробки (3 місяці).

СПИСОК ВИКОРИСТАНІ ДЖЕРЕЛА

1. Алан Торн – Опанування сценаріїв Unity, 2015 (останній перегляд 10 січня 2022)
2. АланThorn - Unity Animation Essentials, 2015 (останній перегляд 10 січня 2022)
3. Алан Торн – Як обдурити в Unity 5: Поради та підказки щодо розробки ігор, 2015 р. (останній огляд 10 січня 2022 р.)
4. Русе, Річард. Ігровий дизайн: теорія і практика:. - 2-й - бульвар Лос-Піос, Плано, Техас, США: Wordware Publishing, 2004.- 698 с. - ISBN 1-55622-912-7. (Останній перегляд 19 травня 2022 р.)
5. Мур, Майкл Е. Основи ігрового дизайну:- 2-й - Нью-Йорк, США: CRC Press, 2011.- 376 с. - ISBN 13: 978-1-4398-6776-1. (Останній перегляд 16 квітня 2022 р.)
6. Роджерс, Скотт. Підніміть рівень! Посібник із чудового дизайну відеоігор: [англійською]. - 2.
   - США: Wiley, 2010. -492 абзац. - ISBN 978-0-470-68867-0. (Останній перегляд перевірено 24
   квітень 2022 року)
7. Шваб, Брайан. Програмування ігрового движка AI: [польська]. – 2-й – Канада: Техніка курсу, 2009. – 710 с – ISBN 978-1-5845-0572-3. (Останній перегляд 24 квітня 2022 р.)
8. Кент Л., Стівен. Остаточна історія відеоігор: [англійською]. - 1-й - Нью-Йорк: Three Rivers Press, 2001.- 608 с. - ISBN 0-7615-3643-4. (Остання перевірка 14 травня 2022 р.)
9. Джозеф Хокінг - Єдність у дії. Мультиплатформеннийрозробка ігор на C # з Unity 5, 2015 р. (останнє перевірено 14 травня 2022 р.)
10. Кріс Дікінсон - Unity 5 Game Optimization, 2015 (перегляд 14 травня 2022)
11. Кенні Ламмерс – Кулінарна книга шейдерів та ефектів Unity, 2013 р. (останнє перевірено 14 травня 2022 р.)

12. Linowes J .: Unity Projects / Linowes J.-2015.-286 c. (Остання перевірка 14 травня 2022 р.)

13. Баррат, Джеймс. Новітній винахід людства / Джеймс Баррат. - М., 2015. 299 п. (Остання перевірка 14 травня 2022 р.)
14. Платов, В. Я. Розробка та організація ігрових додатків. Підручник / В.Я. Літак. - М.:, 2015.- 192 с (Останнє перегляд 2 червня 2022)
15. Анбанова, Т.П. Бізнес-план: досвід, проблеми. Зміст бізнес-плану, приклад дослідження / Т. П. Анбанова, Л.В. Мясоєдова, Т.А. Грамотенко та ін. - М .: Раніше, 2012.- 204 с. 59 (останнє перегляд 2 червня 2022)
16. Хорхе, Jednota Palacios 5.x. Програмування в іграх. Путівник / Паласіос Хорхе. - М .: ДМК Прес, 2017.- 427 с (Останнє перегляд 2 червня 2022)
17. Алгазинов, Э. К. Комп'ютерний аналіз і моделювання інформаційних систем і процесів / Є.К. Алгазінов, А.А. Сирота. - М .: Діалог-Міфі, 2009. - 416 дюймів (останній перегляд 2 червня 2022 р.)
18. Кім Дж. Моделювання та оптимізація дерев на основі віртуальної реальності з метою створення захоплюючого віртуального ландшафту. Симетрія 2016, 8, 93. (востаннє перевірено 2 червня 2022 року)
19. Слейтер, М.; Усох М. Моделювання периферійного зору в імерсивних віртуальних середовищах. Розрахунок. Діаграма. 1993, 17, 643-653. (Останнє відвідування: 2 відгуки 2022)
20. Чон, К.; Лі, Дж.; Кім Дж. Дослідження нових віртуальних машинСистема реальності в Maze Terrain. всередині Дж. Хум. Розрахунок. Вплинути. 2018, 34, 129-145. (Остання перевірка 2 червня 2022 р.)
21. Голдстоун В. Юніті Основи розробки ігор. / В Голдстоуні. Бірмінгем: Packt Publishing Ltd., - 2009. - 316 стор. (Остання перевірка 2 червня 2022 р.)

ДОДАТОК А

КОД ПРОГРАМИ

PlayerMover.cs – головний файл керування ігроком

```
using DG.Tweening;
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;

[RequireComponent(typeof(Rigidbody2D), typeof(CircleCollider2D), typeof(PlayerSkin))]
[RequireComponent(typeof(BoxCollider2D))]
public class PlayerMover : MonoBehaviour
{
    [SerializeField] private float _jumpForce = 5f;
    [SerializeField] private float _planeRaiseForce = 10f;

    [SerializeField] private float _rotationTime = 0.5f;
    private float _rotationAngle = 0f;
    private float _planeRotationAngle = 40f;

    private float _cameraRotationAngle = 7.5f;
    private float _rotationAngelOnDeath = 7.5f;

    public float RotationAngle { get => _rotationAngle; set => _rotationAngle = value; }

    [SerializeField] private LayerMask _safeObstaclesLayerMask;

    [SerializeField] private ParticleSystem _deathParticle;
    [SerializeField] private ParticleSystem _jumpParticle;

    private Rigidbody2D _rigidbody;
    private PlayerSkin _playerSkin;
    private BoxCollider2D _boxCollider;
```

```
private CircleCollider2D _circleCollider;
private Collider2D _hitCollider;

private Vector2 _jumpDirection;
private Vector3 _spawnPosition;

public float SetSpawnPosition { get => _spawnPosition.y; set => _spawnPosition = new Vector3(_spawnPosition.x, value,
    _spawnPosition.z); }

private readonly float _respawnTime = 2f;
private readonly float _deathParticlePlaybackTime = 0.33f;

private readonly float _initialGravity = 11.5f;
private readonly float _planeGravity = 2f;

private readonly float _positionInfelicity = 0.1f;
private readonly float _overlapBoxInfelicity = 0.05f;

private readonly float _upPlaneRotationSmooth = 3f;
private readonly float _downPlaneRotationSmooth = 1.5f;

private MovementType _moveType = MovementType.Default;

private List<MovementType> _savedMoveTypes = new List<MovementType>();
public MovementType SavedMoveType => _savedMoveTypes[_savedMoveTypes.Count - 1];

public bool IsDefaultMoveType => _moveType == MovementType.Default || _moveType == MovementType.Slider ||
    _moveType == MovementType.Scaled;

private bool _isGrounded;
private bool _isJumping;
private bool _isRepeatingJump;
private bool _isDead;
public bool IsDead => _isDead;
public bool IsGrounded => _isGrounded;
```

```
public event Action Died;
public event Action Revieved;
public event Action<float> SliderPortalTriggered;
public event Action<bool> PlaneInputChanged;

private Coroutine _deathCoroutine;
private Coroutine _fallingCoroutine;

private float _sawRotationAngle = -180f;
private Sequence _tweener;

private void Awake()
{
    _rigidbody = GetComponent<Rigidbody2D>();
    _playerSkin = GetComponent<PlayerSkin>();
    _boxCollider = GetComponent<BoxCollider2D>();
    _circleCollider = GetComponent<CircleCollider2D>();

    _spawnPosition = transform.position;
    _jumpDirection = transform.up;

    _savedMoveTypes.Add(MovementType.Default);
}

private void Update()
{
    if (_isDead) { return; }
    if (IsPointerOverUIObject()) { return; }

    if (Input.GetMouseButtonUp(0) && (_moveType == MovementType.Default || _moveType == MovementType.Slider ||
    _moveType == MovementType.Scaled))
    {
        _isRepeatingJump = false;
    }

    if (_moveType == MovementType.GravityChanging)
```

```
    {
        _hitCollider = Physics2D.OverlapCircle(transform.position, _circleCollider.radius, _safeObstaclesLayerMask);
    }
    else if (_moveType != MovementType.Plane)
    {
        _hitCollider = Physics2D.OverlapBox(transform.position, new Vector2(transform.localScale.x - _overlapBoxInfelicity,
    transform.localScale.y + _overlapBoxInfelicity), transform.rotation.z, _safeObstaclesLayerMask);
    }
    else
    {
        _hitCollider = Physics2D.OverlapBox(transform.position, new Vector2(transform.localScale.x, transform.localScale.y +
    0.3f), transform.rotation.z, _safeObstaclesLayerMask);
    }

    _isGrounded = _hitCollider != null;

    Move();
}

private void Move()
{
    switch (_moveType)
    {
        case MovementType.Default:
            DefaultMovement();
            break;
        case MovementType.Scaled:
            DefaultMovement();
            break;
        case MovementType.GravityChanging:
            GravityMovement();
            break;
        case MovementType.Plane:
            PlaneMovement();
            break;
        case MovementType.Slider:
```

```
            SliderMovement();
            break;
    }

    if (_spawnPosition.x > transform.position.x + _positionInfelicity || _spawnPosition.x < transform.position.x -
    _positionInfelicity)
    {
        TryDie();
    }
    else if (_spawnPosition.x != transform.position.x)
    {
        transform.position = new Vector3(_spawnPosition.x, transform.position.y, transform.position.z);
    }
}

private void DefaultMovement()
{
    if (Input.GetMouseButtonDown(0))
    {
        _isRepeatingJump = true;
    }

    if (_isRepeatingJump && _isGrounded)
    {
        _isGrounded = false;
        _isJumping = true;
        StartCoroutine(Jump());
    }
}

private void SliderMovement()
{
    if (Input.GetMouseButtonDown(0))
    {
        _isRepeatingJump = true;
    }
```

```
    if (_isRepeatingJump && _isGrounded)
    {
        _isGrounded = false;
        _isJumping = true;
        StartCoroutine(Jump(true));
    }
}

private void GravityMovement()
{
    if (Input.GetMouseButtonDown(0) && _isGrounded)
    {
        _rigidbody.gravityScale = -_rigidbody.gravityScale;

        _tweener.Kill(false);
        transform.rotation = Quaternion.Euler(Vector3.zero);
        _tweener = DOTween.Sequence();

        _sawRotationAngle = _sawRotationAngle > 0 ? -180f : 180f;

        var rotation = transform.DORotate(new Vector3(0f, 0f, _sawRotationAngle), 0.6f).SetEase(Ease.Linear);

        _tweener.Append(rotation).SetLoops(-1, LoopType.Incremental);
    }
}

private void PlaneMovement()
{
    if (Input.GetMouseButtonDown(0))
    {
        _rigidbody.velocity = new Vector2(0, 0);
        PlaneInputChanged.Invoke(true);
    }
```

```
    if (Input.GetMouseButtonUp(0))
    {
        PlaneInputChanged.Invoke(false);
    }

    if (Input.GetMouseButton(0))
    {
        _planeRotationAngle = Mathf.Abs(_planeRotationAngle);
        _rigidbody.AddForce(_planeRaiseForce * Time.deltaTime * Vector2.up, ForceMode2D.Impulse);
    }
    else
    {
        _planeRotationAngle = -Mathf.Abs(_planeRotationAngle);
    }

    var smooth = _planeRotationAngle > 0 ? _upPlaneRotationSmooth : _downPlaneRotationSmooth;

    if (_isGrounded)
    {
        transform.rotation = Quaternion.Lerp(transform.rotation, Quaternion.Euler(0f, 0f, 0f), smooth * 2f * Time.deltaTime);
        return;
    }

    transform.rotation = Quaternion.Lerp(transform.rotation, Quaternion.Euler(0f, 0f, _planeRotationAngle), smooth *
    Time.deltaTime);
}

private IEnumerator Jump(bool _isSilder = false)
{
    _isRepeatingJump = false;
    _rigidbody.velocity = new Vector2(0, 0);
    _rigidbody.AddForce(_jumpDirection * _jumpForce, ForceMode2D.Impulse);

    if (_fallingCoroutine == null && !_isSilder)
    {
        _rotationAngle -= 179f;
```

```
        _fallingCoroutine = StartCoroutine(Rotate(Quaternion.Euler(new Vector3(0f, 0f, _rotationAngle)), _rotationTime));
    }
    else if (_isSilder && _fallingCoroutine == null)
    {
        _fallingCoroutine = StartCoroutine(WaitForGround(_rotationTime));
    }

    yield return new WaitForFixedUpdate();
    _isRepeatingJump = true;
}

private IEnumerator Rotate(Quaternion endValue, float lerpDuration)
{
    float time = 0;
    var startValue = transform.rotation;

    while (time < lerpDuration)
    {
        transform.rotation = Quaternion.Lerp(startValue, endValue, time / lerpDuration);
        time += Time.deltaTime;

        yield return null;
    }

    _rotationAngle -= 1f;

    transform.rotation = Quaternion.Euler(new Vector3(0f, 0f, _rotationAngle));

    _fallingCoroutine = null;
    yield return new WaitUntil(() => _isGrounded);
    _isJumping = false;
}

private IEnumerator WaitForGround(float duration)
```

```
{
    yield return new WaitForSeconds(duration);
    _fallingCoroutine = null;

    yield return new WaitUntil(() => _isGrounded);
    _isJumping = false;
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.TryGetComponent(out Obstacle obstacle))
    {
        if (obstacle.ObstacleType == ObstacleType.Spike && _deathCoroutine == null)
        {
            TryDie();
        }
    }
}

private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.TryGetComponent(out Obstacle obstacle))
    {
        switch (obstacle.ObstacleType)
        {
            case ObstacleType.GravityPortal:
                {
                    _moveType = _moveType == MovementType.Default ? MovementType.GravityChanging :
    MovementType.Default;
                    transform.rotation = Quaternion.Euler(Vector3.zero);
                    _rotationAngle = 0f;

                    if (_moveType == MovementType.Default)
                    {
                        _rigidbody.gravityScale = Mathf.Abs(_rigidbody.gravityScale);
```

```
            _circleCollider.enabled = false;
            _boxCollider.enabled = true;
            _jumpParticle.gameObject.SetActive(true);
            StopRotation();
        }
        else
        {
            _circleCollider.enabled = true;
            _boxCollider.enabled = false;
            _jumpParticle.gameObject.SetActive(false);
            StartRotation();
        }
    }
    break;
case ObstacleType.PlanePortal:
    {
        _moveType = _moveType == MovementType.Default ? MovementType.Plane : MovementType.Default;
        _rigidbody.gravityScale = _moveType == MovementType.Plane ? _planeGravity : _initialGravity;
        transform.rotation = Quaternion.Euler(Vector3.zero);
        _rotationAngle = 0f;
    }
    break;
case ObstacleType.SliderPortal:
    {
        _moveType = _moveType == MovementType.Default ? MovementType.Slider : MovementType.Default;
        transform.rotation = Quaternion.Euler(Vector3.zero);
        _rotationAngle = 0f;

        if (_moveType == MovementType.Slider)
        {
            SliderPortalTriggered?.Invoke(_cameraRotationAngle);
            _cameraRotationAngle = -_cameraRotationAngle;
        }
        else
        {
```

```
                SliderPortalTriggered?.Invoke(0f);
                _cameraRotationAngle = Mathf.Abs(_cameraRotationAngle);
            }
        }
        break;
    case ObstacleType.SliderRotationChanger:
        {
            SliderPortalTriggered?.Invoke(_cameraRotationAngle);
            _cameraRotationAngle = -_cameraRotationAngle;
        }
        break;
    case ObstacleType.ScalePortal:
        {
            _moveType = _moveType == MovementType.Default ? MovementType.Scaled : MovementType.Default;
        }
        break;
}

if (_moveType == MovementType.Default || _moveType == MovementType.Scaled)
{
    _jumpParticle.Play();
}
else
{
    _jumpParticle.Stop();
}
}
}

public void TryFallDown(float angle, float rotationTime)
{
    if (!_isJumping && _hitCollider == null)
    {
        angle -= 1;
        _rotationAngle -= angle;
```

```
        StartCoroutine(Rotate(Quaternion.Euler(new Vector3(0f, 0f, _rotationAngle)), rotationTime));
    }
}

private void TryDie()
{
    if (_deathCoroutine == null)
    {
        _deathCoroutine = StartCoroutine(Die());
    }
}

private IEnumerator Die()
{
    _isDead = true;
    _rigidbody.velocity = Vector2.zero;
    _rigidbody.isKinematic = true;
    _playerSkin.SpriteRenderer.enabled = false;

    StopRotation();

    Died?.Invoke();

    _deathParticle.gameObject.SetActive(true);
    yield return new WaitForSeconds(_deathParticlePlaybackTime);
    _deathParticle.gameObject.SetActive(false);
    yield return new WaitForSeconds(_respawnTime);

    DeathType();
    _rigidbody.isKinematic = false;
    _playerSkin.SpriteRenderer.enabled = true;
    _isDead = false;
    Revieved?.Invoke();
```

```
    _deathCoroutine = null;
}

private void StartRotation()
{
    transform.rotation = Quaternion.Euler(Vector3.zero);
    _tweener = DOTween.Sequence();
    var rotation = transform.DORotate(new Vector3(0f, 0f, _sawRotationAngle), 0.6f).SetEase(Ease.Linear);

    _tweener.Append(rotation).SetLoops(-1, LoopType.Incremental);
}

private void StopRotation()
{
    _tweener.Kill(true);

    if (_sawRotationAngle > 0)
        _sawRotationAngle = -Mathf.Abs(_sawRotationAngle);
}

private bool IsPointerOverUIObject()
{
    PointerEventData eventDataCurrentPosition = new PointerEventData(EventSystem.current);
    eventDataCurrentPosition.position = new Vector2(Input.mousePosition.x, Input.mousePosition.y);
    List<RaycastResult> results = new List<RaycastResult>();
    EventSystem.current.RaycastAll(eventDataCurrentPosition, results);
    return results.Count > 0;
}

private void DeathType()
{
    if(_savedMoveTypes.Count == 0)
    {
        Debug.LogError("SAVED MOVE TYPES COUNT IS 0");
        return;
```

```
}

switch(_savedMoveTypes[_savedMoveTypes.Count - 1])
{
    case MovementType.Default:
        {
            _circleCollider.enabled = false;
            _boxCollider.enabled = true;
            _isRepeatingJump = false;
            _rigidbody.gravityScale = Mathf.Abs(_initialGravity);
            transform.SetPositionAndRotation(_spawnPosition, Quaternion.Euler(Vector3.zero));
            _moveType = MovementType.Default;
            _rotationAngle = 0f;
            transform.localScale = Vector3.one;
            _cameraRotationAngle = Mathf.Abs(_cameraRotationAngle);
            SliderPortalTriggered?.Invoke(0f);
            break;
        }
    case MovementType.Scaled:
        {
            _circleCollider.enabled = false;
            _boxCollider.enabled = true;
            _isRepeatingJump = false;
            _rigidbody.gravityScale = Mathf.Abs(_initialGravity);
            transform.SetPositionAndRotation(_spawnPosition, Quaternion.Euler(Vector3.zero));
            _moveType = MovementType.Default;
            _rotationAngle = 0f;
            transform.localScale = Vector3.one;
            _cameraRotationAngle = Mathf.Abs(_cameraRotationAngle);
            SliderPortalTriggered?.Invoke(0f);

            break;
        }
    case MovementType.GravityChanging:
        {
```

```
            _circleCollider.enabled = true;
            _boxCollider.enabled = false;
            _isRepeatingJump = false;
            _rigidbody.gravityScale = Mathf.Abs(_initialGravity);
            transform.SetPositionAndRotation(_spawnPosition, Quaternion.Euler(Vector3.zero));
            _moveType = MovementType.GravityChanging;
            _rotationAngle = 0f;
            transform.localScale = Vector3.one;
            _cameraRotationAngle = Mathf.Abs(_cameraRotationAngle);
            SliderPortalTriggered?.Invoke(0f);
            StartRotation();

            break;
        }
    case MovementType.Plane:
        {
            _circleCollider.enabled = false;
            _boxCollider.enabled = true;
            _isRepeatingJump = false;
            _rigidbody.gravityScale = Mathf.Abs(_planeGravity);
            transform.SetPositionAndRotation(_spawnPosition, Quaternion.Euler(Vector3.zero));
            _moveType = MovementType.Plane;
            _rotationAngle = 0f;
            transform.localScale = new Vector3(0.7f, 0.7f, 0.7f);
            _cameraRotationAngle = Mathf.Abs(_cameraRotationAngle);
            SliderPortalTriggered?.Invoke(0f);

            break;
        }
    case MovementType.Slider:
        {
            _circleCollider.enabled = false;
            _boxCollider.enabled = true;
            _isRepeatingJump = false;
            _rigidbody.gravityScale = Mathf.Abs(_initialGravity);
```

```
            transform.SetPositionAndRotation(_spawnPosition, Quaternion.Euler(Vector3.zero));
            _moveType = MovementType.Slider;
            _rotationAngle = 0f;
            transform.localScale = Vector3.one;

            _cameraRotationAngle = _rotationAngelOnDeath;
            SliderPortalTriggered?.Invoke(_cameraRotationAngle);
            _cameraRotationAngle = -_cameraRotationAngle;

            break;
        }
    }
}

public void SaveMoveType()
{
    _savedMoveTypes.Add(_moveType);

    if(_moveType == MovementType.Slider)
    {
        _rotationAngelOnDeath = -_cameraRotationAngle;
    }
}

public void RemoveMoveType()
{
    if (_savedMoveTypes.Count == 0)
    {
        Debug.LogError("SAVED MOVE TYPES COUNT IS 0");
        return;
    }

    _savedMoveTypes.RemoveAt(_savedMoveTypes.Count - 1);
}
```

```
        public enum MovementType
        {
            Default,
            GravityChanging,
            Plane,
            Slider,
            Scaled
        }

        void OnDrawGizmos()
        {
            Gizmos.color = Color.red;
            Gizmos.DrawWireCube(transform.position, new Vector2(transform.localScale.x, transform.localScale.y - 2f));
            Gizmos.DrawWireSphere(transform.position, 0.5f);
        }
}
```

Chunks.cs – файл завантаження чанків-рівнів

```
using UnityEngine;

public class Chunk : MonoBehaviour
{
    [SerializeField] private Transform _start;
    [SerializeField] private Transform _end;

    [SerializeField] private Transform _ticketPosition;

    [SerializeField] private AnimationCurve _chanceFromDistance;

    public Transform Start => _start;
    public Transform End => _end;

    public Transform TicketContainer => _ticketPosition;

    public AnimationCurve ChanceFromDistance => _chanceFromDistance;

    public void ChangeAnimationCurve(ChunksData curveData)
    {
        Keyframe[] keyframes = _chanceFromDistance.keys;

        keyframes[0].time = curveData.FirstKeyTime;
        keyframes[0].value = curveData.FirstKeyValue;

        keyframes[1].time = curveData.SecondKeyTime;
        keyframes[1].value = curveData.SecondKeyValue;

        keyframes[2].time = curveData.ThirdKeyTime;
        keyframes[2].value = curveData.ThirdKeyValue;

        _chanceFromDistance.keys = keyframes;
```

```
    }
}
```

AudioSettings.cs – скрипт керуванням звуком

```
using UnityEngine;
using UnityEngine.Audio;
using UnityEngine.UI;

public class AudioSettings : MonoBehaviour
{
    [SerializeField] private AudioMixerGroup _audioMixerGroup;
    [SerializeField] private Slider _musicSlider;
    [SerializeField] private Slider _generalSlider;

    [SerializeField] private Button _musicButton;
    [SerializeField] private Button _generalButton;

    [SerializeField] private Sprite _enabledMusic;
    [SerializeField] private Sprite _disabledMusic;

    [SerializeField] private Sprite _enabledGeneral;
    [SerializeField] private Sprite _disabledGeneral;

    private Image _musicButtonImage;
    private Image _generalButtonImage;

    public void MuteMusic()
    {
        _audioMixerGroup.audioMixer.GetFloat("MusicVolume", out float volume);

        if (volume > -80f)
        {
            _audioMixerGroup.audioMixer.SetFloat("MusicVolume", -80f);
            _musicSlider.value = 0f;
            _musicButtonImage.sprite = _disabledMusic;
        }
        else
        {
            _audioMixerGroup.audioMixer.SetFloat("MusicVolume", 0f);
            _musicSlider.value = 1f;
            _musicButtonImage.sprite = _enabledMusic;
        }
    }

    public void MuteGeneral()
    {
        _audioMixerGroup.audioMixer.GetFloat("GeneralVolume", out float volume);

        if (volume > -80f)
        {
            _audioMixerGroup.audioMixer.SetFloat("GeneralVolume", -80f);
            _generalSlider.value = 0f;
            _generalButtonImage.sprite = _disabledGeneral;
        }
        else
        {
            _audioMixerGroup.audioMixer.SetFloat("GeneralVolume", 0f);
            _generalSlider.value = 1f;
            _generalButtonImage.sprite = _enabledGeneral;
        }
    }
```

```
    public void ChangeMusicVolume(float volume)
    {
        if (volume < 0.01f)
        {
            _audioMixerGroup.audioMixer.SetFloat("MusicVolume", -80f);
            _musicButtonImage.sprite = _disabledMusic;
            return;
        }

        _audioMixerGroup.audioMixer.SetFloat("MusicVolume", Mathf.Lerp(-50f, 0f, volume));

        if(_musicButtonImage.sprite != _enabledMusic)
        {
            _musicButtonImage.sprite = _enabledMusic;
        }
    }

    public void ChangeGeneralVolume(float volume)
    {
        if (volume < 0.01f)
        {
            _audioMixerGroup.audioMixer.SetFloat("GeneralVolume", -80f);
            _generalButtonImage.sprite = _disabledGeneral;
            return;
        }

        _audioMixerGroup.audioMixer.SetFloat("GeneralVolume", Mathf.Lerp(-50f, 0f, volume));

        if (_generalButtonImage.sprite != _enabledGeneral)
        {
            _generalButtonImage.sprite = _enabledGeneral;
        }

    }

    public void LoadAudioSettings()
    {
        _musicButtonImage = _musicButton.GetComponent<Image>();
        _generalButtonImage = _generalButton.GetComponent<Image>();

        _musicSlider.value = PlayerPrefs.GetFloat("MusicSlider", 1f);
        _generalSlider.value = PlayerPrefs.GetFloat("GeneralSlider", 1f);

        ChangeMusicVolume(_musicSlider.value);
        ChangeGeneralVolume(_generalSlider.value);
    }

    private void OnDisable()
    {
        PlayerPrefs.SetFloat("MusicSlider", _musicSlider.value);
        PlayerPrefs.SetFloat("GeneralSlider", _generalSlider.value);
    }
}
```

CameraRotator.cs – скрипт керування камерою для одного з режимів

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

public class CameraRotator : MonoBehaviour
{
```

```
    [SerializeField] private PlayerMover _playerMover;

    private void OnEnable()
    {
        _playerMover.SliderPortalTriggered += SetAngle;
    }

    private void OnDisable()
    {
        _playerMover.SliderPortalTriggered -= SetAngle;
    }

    public void SetAngle(float value)
    {
        StartCoroutine(LerpRotation(value, 0.5f));
        //transform.rotation = Quaternion.Euler(new Vector3(0f, 0f, value));
    }

    private IEnumerator LerpRotation(float value, float duration)
    {
        float time = 0;
        Quaternion startValue = transform.rotation;
        Quaternion endValue = Quaternion.Euler(new Vector3(0f, 0f, value));

        while (time < duration)
        {
            transform.rotation = Quaternion.Lerp(startValue, endValue, time / duration);
            time += Time.deltaTime;
            yield return null;
        }

        transform.rotation = endValue;
    }
}
```

ДОДАТОК Б
ВІДГУК КЕРІВНИКА ЕКОНОМІЧНОГО РОЗДІЛУ

ДОДАТОК С
ПРЕЛІК ФАЙЛІВ НА ДИСКУ

Ім’я файлу	Опис
Пояснювальні документи	
Диплом 2022 Нечепоренко 122-18-2.doc	Пояснювальна записка до кваліфікаційної роботи. Документ Word
Диплом 2022 Нечепоренко 122-18-2.pdf	Пояснювальна записка до кваліфікаційної роботи в форматі PDF
Програма	
GeometryRush.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Нечепоренко.ppt	Презентація кваліфікаційної роботи