

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ МЕТОДІВ РОЗРОБКИ ТА ПРОГРАМНА
РЕАЛІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ

(назва теми)

RESEARCH ON DEVELOPMENT METHODS AND SOFTWARE
IMPLEMENTATION OF A GAME APPLICATION

Виконав(ла): студент(ка) 2 курсу, групи КНТ-213м
Спеціальності 122 Комп'ютерні науки

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Системи штучного інтелекту

КРИВЕНКО А.О.

(ПРИЗВИЩЕ та ініціали)

Керівник ГОФМАН Є.О.

(ПРИЗВИЩЕ та ініціали)

Рецензент КОЦУР М.І.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 122 Комп'ютерні науки
(код і найменування)
Освітня програма (спеціалізація) Системи штучного інтелекту
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2024 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

КРИВЕНКА Антона Олександровича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження методів розробки та програмна реалізація ігрового застосунку Research on Development Methods and Software Implementation of a Game Application

керівник проєкту (роботи) к.т.н., ГОФМАН Євген Олександрович,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 07 ” жовтня 2024 року № 398

2. Строк подання студентом проєкту (роботи) 10 грудня 2024 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Загальне уявлення про розробку ігор. 2. планування розробки проєкту. 3. Розробка проєкту

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	ГОФМАН Є.О., доцент		
Нормоконтролер	БЄЛОВА А.В. асистент		

7. Дата видачі завдання “ 03 ” жовтня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2-3 тижні	Розділ 1
3	Розробка та удосконалення методів, моделей й алгоритмів, вирішення задачі.	4-5 тижні	Розділ 2
4	Розробка архітектури програми.	6 тиждень	Розділ 3
5	Розробка програми.	7-8 тижні	Розділ 4
6	Тестування та експериментальне дослідження програмного забезпечення.	9 тиждень	Розділ 5
7	Оформлення пояснювальної записки та документів до неї.	10-11 тижні	Додатки
8	Нормоконтроль та рецензування.	12 тиждень	
9	Захист роботи.	12 тиждень	

Студент(ка)

_____ Антон КРИВЕНКО
(підпис) (Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

_____ Євген ГОФМАН
(підпис) (Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
67 с., 3 табл., 19 рис., 2 дод., 5 джерел.

ГРА, ІГРОВИЙ ПРОГРАМНИЙ ЗАСТОСУНОК, ANDROID, IOS, UNITY.

Розробка комп'ютерних та мобільних ігор — це складний і творчий процес, який набув широкої популярності завдяки розвитку сучасних технологій. Unity і WebGL є провідними інструментами для створення ігор, а доповнення у вигляді підтримки платформ Android та iOS значно розширює аудиторію користувачів. Unity — інтегроване середовище розробки, яке дозволяє створювати ігри для різних платформ, а WebGL забезпечує запуск графічно насичених додатків у веб-браузері.

Розробка ігор із використанням Unity WebGL, Android та iOS надає розробникам широкі можливості. До основних переваг належать підтримка сучасної графіки, кросплатформенність, оптимізована продуктивність та доступність як для мобільних пристроїв, так і для веб-середовища.

Процес створення гри складається з кількох етапів. Першим кроком є планування і дизайн. Тут визначаються концепція гри, її механіка, цільова аудиторія, а також обирається стиль оформлення. Для мобільних ігор важливо враховувати особливості взаємодії з сенсорним екраном.

На етапі розробки програмісти створюють функціонал гри, тоді як дизайнери працюють над графічним контентом. Unity пропонує інструменти для адаптації гри до різних платформ, зокрема Android, iOS і браузерів через WebGL. Це забезпечує доступність гри для різних категорій гравців.

Тестування є ключовим етапом для всіх платформ. Для мобільних додатків тестування охоплює перевірку продуктивності на різних пристроях, адаптивність інтерфейсу та стабільність роботи в умовах обмежених ресурсів.

Реліз гри на Android та iOS зазвичай супроводжується публікацією в магазинах додатків, таких як Google Play і App Store. Це передбачає дотримання специфічних вимог кожної платформи, таких як сертифікація, розміщення графічних матеріалів і підготовка описів.

Після виходу гри розробники продовжують підтримку продукту. Для мобільних ігор особливо важливими є регулярні оновлення, які включають новий контент, виправлення помилок і адаптацію до нових версій операційних систем.

Розробка ігор на Unity WebGL, Android та iOS — це захопливий і багатогранний процес, який потребує злагодженої роботи команди. Завдяки цим технологіям можна створювати сучасні, захоплюючі ігри, доступні як для мобільних пристроїв, так і для браузерів, що дозволяє охопити широку аудиторію.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 67 pages, 3 tables, 19 figures, 2 appendixes, 5 sources.

GAME, GAME SOFTWARE APPLICATION, ANDROID, IOS, UNITY.

The development of computer and mobile games is a complex and creative process that has gained wide popularity due to the development of modern technologies. Unity and WebGL are the leading tools for creating games, and the addition of support for Android and iOS platforms significantly expands the audience of users. Unity is an integrated development environment that allows you to create games for different platforms, and WebGL allows you to run graphically rich applications in a web browser.

Developing games using Unity WebGL, Android, and iOS provides developers with a wide range of opportunities. The main advantages include support for modern graphics, cross-platform compatibility, optimized performance, and accessibility for both mobile devices and the web.

The process of creating a game consists of several stages. The first step is planning and design. Here, the concept of the game, its mechanics, target audience, and design style are defined. For mobile games, it is important to take into account the peculiarities of interaction with the touch screen.

At the development stage, programmers create game functionality, while designers work on graphic content. Unity offers tools for adapting games to different platforms, including Android, iOS, and browsers via WebGL. This ensures that the game is accessible to different categories of players.

Testing is a key stage for all platforms. For mobile applications, testing includes checking performance on different devices, interface adaptability, and stability in conditions of limited resources. The release of a game on Android and iOS is usually accompanied by publication in app stores such as Google Play and

the App Store. This implies compliance with the specific requirements of each platform, such as certification, placement of graphic materials, and preparation of descriptions.

After the game is released, developers continue to support the product. Regular updates are especially important for mobile games, which include new content, bug fixes, and adaptation to new versions of operating systems.

The development of games on Unity WebGL, Android, and iOS is an exciting and multifaceted process that requires a well-coordinated teamwork. Thanks to these technologies, you can create modern, exciting games that are available for both mobile devices and browsers, which allows you to reach a wide audience.

ЗМІСТ

	С.
Вступ.....	10
1 Загальне уявлення про розробку ігор.....	12
1.1 Ролі у розробці ігор	12
1.1.1 Геймдизайнер.....	14
1.1.2 Дизайнер рівнів.....	14
1.1.3 Художник.....	14
1.1.4 Ігровий програміст	15
1.1.5 Звукорежисер.....	15
1.1.6 Тестувальники	15
1.2 Документування проєкту	16
1.3 Процес розробки.....	16
1.4 Передпродакшн	17
1.5 Виробництво	17
1.6 Підтримка.....	17
1.7 Аутсорсинг.....	18
1.8 Ігровий рушій (фреймворк)	18
1.9 Особливості ігрового рушія Unity	19
1.10 Мобільна розробка	20
2 Планування розробки проєкту.....	21
2.1 Мова програмування	21
2.2 Графіка для 3D гри	22
2.3 Жанр гри	23
2.4 Основні механіки гри	24
3 Розробка проєкту.....	26
3.1 Розробка рівня	26
3.2 Налаштування гравця	29
3.3 Детальна реалізація механіки телепорту.....	34

3.4 Камера та компонент CinemachineBrain.....	34
3.5 Менеджери гри	36
3.6 UI гри	37
3.6 Збереження даних гравця.....	40
3.7 Аудіо гри.....	41
3.8 Фінальний продукт	42
Висновки.....	44
Перелік джерел посилання	46
Додаток А Код програми	47
Додаток Б Слайди презентації.....	59

ВСТУП

Розробка комп'ютерних ігор — це складний і багатогранний процес, який охоплює різні етапи: від початкового планування до фінального випуску продукту. Сучасні технології, такі як Unity та WebGL, дають змогу створювати високоякісні ігри, які можуть запускатися безпосередньо в браузері, що робить їх доступними для широкої аудиторії. Ця робота охоплює основні аспекти розробки браузерної гри, зокрема технічні й творчі елементи процесу. Особлива увага приділена оптимізації продуктивності, кросплатформенності, а також розвитку мобільного геймінгу, що є важливою частиною сучасного ринку.

З поширенням інтернету та розвитком веб-технологій браузерні ігри здобувають дедалі більшу популярність, адже вони не вимагають завантаження чи встановлення додаткових програм. Щоб почати грати, достатньо відкрити веб-браузер і зайти на відповідний сайт. Що стосується мобільних ігор, вони також стають важливою частиною індустрії завдяки зручності використання смартфонів і можливості доступу до ігор будь-де й будь-коли.

Unity є надзвичайно потужним інструментом для розробки ігор, який забезпечує розробникам можливості для створення гнучких ігрових механік, реалістичної графіки та анімацій. Завдяки інтеграції з WebGL розробники можуть створювати ігри, які працюють у будь-якому сучасному браузері без необхідності встановлення додаткових плагінів. Крім того, Unity дозволяє створювати мобільні ігри, які оптимізовані для Android та iOS, розширюючи аудиторію гравців.

Розробка гри включає кілька важливих етапів: планування, дизайн, розробку, тестування та реліз. Кожен з цих етапів потребує специфічного підходу й уваги. На етапі планування визначаються загальна концепція, механіка гри та її стиль. Під час розробки програмісти та дизайнери створюють код гри й візуальні елементи, використовуючи потужні

інструменти Unity для реалізації своїх ідей. Тестування гри забезпечує її стабільність на різних пристроях і платформах, виявляючи помилки та недоліки. Реліз гри означає її вихід у світ, але гра потребує подальшої підтримки та оновлень для задоволення потреб користувачів.

Успішна розробка гри неможлива без тісної співпраці між членами команди: програмістами, дизайнерами, художниками та тестувальниками. Кожен член команди вносить свій вклад, об'єднуючи свої навички та знання для створення захоплюючої та якісної гри.

Ця робота докладно висвітлить основні етапи розробки гри, враховуючи важливі моменти планування, дизайну, технічні вимоги, методи тестування та оптимізації.

1 ЗАГАЛЬНЕ УЯВЛЕННЯ ПРО РОЗРОБКУ ІГОР

Розробка комп'ютерних ігор є складним і багатогранним процесом, що може здійснюватися як індивідуально, так і цілою командою розробників. Команди, що працюють над створенням ігор, можуть спеціалізуватися на конкретних платформах — комп'ютерних, консольних або мобільних. Часто вони отримують фінансування від видавців, великих компаній, які не лише надають кошти, але й контролюють процес створення, надають допоміжні ресурси і в кінцевому результаті розповсюджують гру на ринку. Зокрема, видавець допомагає у фінансуванні, управлінні проектом та забезпеченні маркетингової підтримки, що дозволяє розробникам зосередитися на технічних та творчих аспектах.

З розвитком цифрових платформ з'явилося багато нових можливостей для розповсюдження ігор, що дозволяють незалежним розробникам виходити на ринок без участі великих компаній. Такий самостійний випуск можливий через численні платформи-дистриб'ютори, як-от Steam, App Store, Google Play та інші. Однак великі проекти потребують значних ресурсів: бюджети на створення масштабних ігор, відомих як AAA-проекти, нерідко сягають десятків мільйонів доларів. Ігрова індустрія стрімко зростає, і кількість фахівців, потрібних для розробки таких ігор, також збільшується. Наприклад, в 1990-х роках команда з 10 осіб могла створити гру для PlayStation за один рік. Вже на початку 2000-х років для гри на PlayStation 2 була необхідна команда з 30-50 розробників та термін виконання в 2 роки, а в 2010-х роках для PlayStation 3 кількість учасників перевищувала 100 осіб, а тривалість проекту становила приблизно 3 роки.

1.1 Ролі у розробці ігор

З часів появи перших домашніх комп'ютерів та ігрових приставок у 1980-х роках процес розробки ігор суттєво змінився. Якщо раніше один

програміст міг самостійно впоратися з усіма завданнями, то сьогодні у розробці великих проектів беруть участь команди спеціалістів з різних галузей. Розглянемо основні ролі в сучасній розробці ігор [1].

Серед них:

- продюсер — особа, що контролює процес створення гри. Продюсери можуть бути як внутрішніми (працюють у самій компанії-розробнику), так і зовнішніми (представники видавця). Зовнішні продюсери відстежують дотримання графіка та бюджетів, забезпечують своєчасне виконання завдань, надають ресурси командам, а також допомагають у вирішенні юридичних та адміністративних питань. Внутрішні продюсери зосереджуються на управлінні командою, плануванні роботи, підготовці звітів і організації робочого процесу. Крім того, продюсери займаються зв'язками з громадськістю, переговорами, контрактами, підбором персоналу, забезпеченням якості, організацією бета-тестування і локалізацією гри;

- видавець — компанія, яка займається публікацією та випуском гри. Видавці фінансують розробку, займаються маркетингом, аналізом ринку і рекламою. Вони володіють правами на гру і, як правило, самостійно поширюють її або передають права дистриб'юторам. Видавці також оплачують ліцензії на використання музики та торгових марок у грі, організують локалізацію і займаються розробкою графічного оформлення для гри (наприклад, дизайн обкладинки). Великі компанії часто забезпечують команди звуковим супроводом та спільними кодовими бібліотеками, що дозволяє оптимізувати роботу;

- команда розробки — група професіоналів, яка працює над створенням гри. Команда може налічувати від кількох осіб, зайнятих над невеликим мобільним проектом, до сотень і навіть тисяч спеціалістів у масштабних AAA-проектах. Основні фахівці у команді — це художники, програмісти, геймдизайнери та звукорежисери. Усі ці спеціалісти працюють над різними аспектами проекту, від візуальної складової до технічної реалізації ігрових механік.

1.1.1 Геймдизайнер

Геймдизайнер — це людина, яка створює основні концепції гри, правила та механіки. Він продумує ігровий процес, створює наративну частину гри, включаючи сюжети, діалоги, сцени, підказки тощо. Провідний геймдизайнер часто координує роботу інших геймдизайнерів і відповідає за узгодженість гри з початковою ідеєю. У великих проектах геймдизайнери можуть працювати над конкретними аспектами гри, наприклад, механікою бою, системою розвитку персонажів, користувацьким інтерфейсом і діалогами.

1.1.2 Дизайнер рівнів

Дизайнер рівнів (Level Designer) розробляє структуру ігрових рівнів, завдань та місій. Для цього використовуються інструменти розробки ігор, спеціальні редактори рівнів, що дозволяють створювати та тестувати ігрові простори, де відбуватимуться дії. Це можуть бути редактори для 2D та 3D-дизайну, вбудовані в движки або зовнішні додатки.

1.1.3 Художник

Художники працюють над візуальною частиною гри, створюють концепт-арт, спрайти, текстури, анімації, моделі персонажів, інтерфейси та інші елементи гри. Існують як 2D-художники, які створюють концептуальні зображення та малюнки для користувацького інтерфейсу, так і 3D-художники, відповідальні за моделювання, анімацію та створення 3D-середовищ. Директор художнього відділу стежить за узгодженістю стилю і координує роботу всієї команди.

1.1.4 Ігровий програміст

Програмісти займаються реалізацією усіх функціональних частин гри. Вони відповідають за розробку фізики, штучного інтелекту, графіки, звуку, та геймплейних механік. Програмісти пишуть код для руху об'єктів, зіткнень, анімацій, інтегрують 3D-моделі, розробляють та вдосконалюють графічні та фізичні рушії. Ігрові програмісти можуть спеціалізуватися на окремих аспектах, таких як фізика, ШІ або графіка, що дозволяє створювати високоякісні механіки та ефекти. Зазвичай провідні програмісти координують роботу команди, розробляють основні частини проекту і планують подальшу реалізацію.

1.1.5 Звукорежисер

Звукорежисери працюють над музичним супроводом, записують звуки для інтерфейсу, персонажів та різних дій. Вони забезпечують інтеграцію звукових ефектів у гру, створюють музичні композиції, синхронізують звукові ефекти з діями у грі, а також працюють над дубляжем. Звук — один з важливих аспектів, який дозволяє гравцю відчувати атмосферу гри та взаємодію з ігровим середовищем.

1.1.6 Тестувальники

Тестувальники або QA-інженери перевіряють якість гри, виявляють помилки, тестують різні ігрові сценарії і повідомляють про проблеми, що виникають. Вони грають у гру з різними комбінаціями налаштувань, шукаючи можливі баги, і стежать за тим, щоб усі елементи гри працювали коректно. Підтримка високої якості — це важливий етап, особливо перед випуском гри на ринок.

Розробка ігор є складним і багатостороннім процесом, що включає різні спеціалізації та напрями, що дозволяє втілювати найскладніші ідеї в реальність. Незалежно від того, чи є проект незалежним або великим AAA-проектом, всі учасники команди вносять свій унікальний вклад, щоб створити унікальний ігровий досвід для гравців.

1.2 Документування проєкту

Коли команда починає працювати над грою, стає необхідним вести документацію, що допомагає організувати процес розробки, знаходити колег і залучати інвестиції. Основні типи документів включають:

- концепт-документ. Короткий документ (2-6 сторінок), що описує основні ідеї та особливості гри, бажано з ілюстраціями;
- дизайн-документ. Детальніший документ, який розширює концепт-документ, коли гра переходить до стадії виробництва;
- документ-пропозиція. Короткий опис гри, який акцентує на її прибутковості, без внутрішніх деталей розробки;
- технічний документ. Включає технічні вимоги (полігони, fps, пам'ять тощо) і описує інструменти та мови програмування, які будуть використані.

1.3 Процес розробки

Процес розробки гри зазвичай складається з кількох етапів:

- підготовка (препродакшн);
- уточнення геймдизайну;
- виробництво;
- підтримка.

Ці етапи можуть варіюватися залежно від компанії та специфіки проєкту.

1.4 Передпродакшн

На цьому етапі оцінюються виробничі ризики, бюджет і гуманітарні питання. Розраховується необхідний контент, час і витрати, визначається кількість та кваліфікація художників, а також середовище розробки та графічні вимоги. Прототипуються основні елементи гри. Визначаються бюджет полігонів, продуктивність сервера для багатокористувацьких ігор тощо. Ідеї подаються на затвердження керівництву, а потім видавцю, якщо розробник і видавець — різні компанії. Геймдизайнер пише дизайн-документ, що містить концепцію, ігровий процес та ескізи, які можуть змінюватися під час розробки.

1.5 Виробництво

Цей етап передбачає значну роботу: програмісти пишуть код, художники створюють графіку, звукові фахівці працюють над ефектами, а композитори пишуть музику. Дизайнери створюють рівні, а сценаристи пишуть діалоги. Геймдизайнери постійно вдосконалюють концепцію гри, вносячи зміни, які потрібно документувати. Ранні рівні займають більше часу, оскільки дизайн і функціонал ще формуються. Як тільки гра готова до тестування, тестувальники долучаються до процесу, що стає все більш важливим, коли гра наближається до завершення.

1.6 Підтримка

Підтримка включає випуск патчів для виправлення помилок, виявлених після релізу гри. У випадку масових багатокористувацьких онлайн-ігор (ММО) підтримка може вимагати більше зусиль і часу, ніж сам процес виробництва, оскільки для підтримки інтересу гравців потрібні постійні оновлення та розвиток.

1.7 Аутсорсинг

Аутсорсинг є важливою практикою в розробці комп'ютерних ігор, особливо коли йдеться про такі аспекти, як створення музики, звукових ефектів, озвучення персонажів і захоплення руху. Ці процеси часто вимагають значних фінансових витрат, які можуть бути недоступні для деяких розробників, особливо у випадках, коли компанія не має спеціалізованого відділу для виконання цих завдань. З метою оптимізації витрат та ресурсів, більшість компаній вирішують передати ці функції стороннім виконавцям.

Процес аутсорсингу починається на етапі підготовки до виробництва, коли розробники оцінюють потреби проекту та складають бюджет на залучення зовнішніх послуг. Це може включати оцінку часу та витрат на виконання певних завдань. Деякі інструменти, такі як відеокompresори або редактори рівнів, також можуть бути передані стороннім студіям.

Вартість створення музики варіюється в залежності від її тривалості, методу виробництва (синтетичний чи живий) та досвіду композитора. Наприклад, у 2003 році вартість однієї хвилини якісної синтезаторної музики становила сотні до тисяч доларів. Щодо озвучення, це вимагає спеціальних навичок, тому зазвичай аутсорсується, оскільки лише великі телерадіокомпанії можуть дозволити собі мати власних дикторів. А студії захоплення руху, через високу вартість та складність обладнання, часто не можуть бути побудовані меншими компаніями, що робить аутсорсинг їхніх послуг доцільним рішенням.

1.8 Ігровий рушій (фреймворк)

Ігрові фреймворки – це програмні комплекси, що полегшують процес розробки ігор за рахунок наявності вбудованих інструментів для різних завдань, таких як графіка, обчислення фізики, алгоритми пошуку шляхів і

робота з 3D-редакторами. Існує безліч різних ігрових рушіїв, кожен з яких має свої особливості та переваги. Найпопулярніші з них включають:

- Unity. Універсальне середовище для розробки 2D та 3D ігор;
- Unreal Engine. Потужний рушій, широко використовується для розробки великих ігор;
- Construct 3. Платформа, що дозволяє створювати ігри без програмування;
- GameMaker, який спеціалізується на розробці 2D ігор.

У даній роботі обрано Unity, оскільки цей фреймворк є найбільш доступним і підходить для розробки на різних платформах, таких як Windows, Mac, Linux, Android, iOS, PlayStation 4, PlayStation 5, WebGL тощо.

1.9 Особливості ігрового рушія Unity

Unity пропонує безліч інструментів і функцій для розробки ігор різних типів і складності. Основними характеристиками є:

- редактор сцен. Дозволяє створювати та налаштовувати ігрові об'єкти, анімації, освітлення, тіні, а також додавати звукові ефекти та музику;
- підтримка мов програмування. Unity використовує C# як основну мову програмування, але також підтримує C++, JavaScript та Boo;
- фізичний рушій. забезпечує реалістичні фізичні ефекти, такі як гравітація та зіткнення;
- багатокористувацькі можливості. включає мережеві функції та інтеграцію з соціальними мережами.

Unity організовує всі ігрові об'єкти на основі трьох основних блоків: GameObjects, компоненти та змінні. Кожен об'єкт у грі є GameObject, і для його функціонування потрібні різні властивості, які надаються через компоненти.

Unity також підтримує розробку браузерних ігор з використанням технології WebGL, що дозволяє запускати ігри без додаткових плагінів. Для створення WebGL-збірки в Unity потрібно вибрати відповідну платформу у

налаштуваннях проекту. Готовий продукт може бути опублікований на веб-сайті або запусканий на локальному сервері. Unity підтримує API WebGL 1.0 та 2.0, але використання цієї технології може спричинити проблеми з продуктивністю та сумісністю на різних пристроях та браузерах.

1.10 Мобільна розробка

Unity також добре підходить для мобільної розробки, дозволяючи створювати ігри для Android та iOS. Розробники можуть використовувати можливості Unity для оптимізації своїх ігор під конкретні мобільні платформи, включаючи оптимізацію графіки, управління ресурсами та інтеграцію з функціями мобільних пристроїв, такими як сенсори, GPS та вібрація. Unity надає інструменти для тестування продуктивності на мобільних пристроях, що дозволяє розробникам забезпечити плавність роботи ігор на різних гаджетах.

Таким чином, Unity є потужним і універсальним інструментом для розробки ігор, який підтримує аутсорсинг, використовує різні фреймворки та має широкі можливості для мобільної розробки, що робить його привабливим вибором для розробників у всьому світі.

2 ПЛАНУВАННЯ РОЗРОБКИ ПРОЕКТУ

2.1 Мова програмування

Мовою програмування обрано C# для проекту на Unity, оскільки він є основною мовою цієї платформи. Unity пропонує інструменти та API, спеціально розроблені для C#, що сприяє ефективній розробці ігор. Дана мова програмування вирізняється простотою, зрозумілим синтаксисом та швидкістю розробки завдяки автоматичному управлінню пам'яттю. Інтегрований редактор Unity дозволяє легко використовувати C# для налаштування об'єктів та створення ігрової логіки. Крім того, код на C# у Unity можна легко компілювати для різних платформ.

Основна бібліотека для створення ігор, яку надає Unity, це Unity.Engine, вона пропонує усі необхідні інструменти для редагування у коді через C#, такі як компоненти об'єктів: `sprite renderer`, `transform`, `rigidbody`, `colliders`, `triggers` та ін.. Також у функціонал входять інструменти анімування, налаштування візуальних партиклів, створення об'єктів, отримання доступу до компонентів об'єктів сцени, як через теги, так і через дочірні елементи. Додатково, можна вмикати та вимикати елементи сцени, що буде особливо корисно при реалізації логіки перезапуску рівня, а також можна зручно керувати усіма створеними сценами гри.

Також Unity.Engine пропонує інструменти для зберігання даних, наприклад `PlayerPrefs`. Він може зберігати рядкові, плаваючі та цілі числа в реєстрі платформи користувача. Unity зберігає `PlayerPrefs` у локальному реєстрі без шифрування, саме тому не рекомендується використання `PlayerPrefs` для зберігання конфіденційних даних. Шлях зберігання даних залежить від операційної системи на якій працює програма.

Серед можливих альтернатив можна виділити C++, який підтримує інший рушій – Unreal Engine, але зазвичай його обирають для більш складних проектів, які потребують складних ігрових механік чи інструментів по роботі із графікою, наприклад Ray Tracing.

Порівняння ключових функцій та відмінностей між рушіями Unity та Unreal Engine наведені у таблиці 2.1 [2].

Таблиця 2.1 – Порівняння Unity та Unreal Engine

Feature	Unreal Engine	Unity
Тип рушія	Кросплатформений	Кросплатформений
Творець	Epic Games	Unity Technologies
Feature	Unreal Engine	Unity
Мова програмування	C++	C#
Продвинуті функції	Продвинутий мультиплеєр, фотореалістичний VFX	Інструменти анімації, легке створення снапшотів
Вихідний код	Відкритий	Закритий
Графіка	Фотореалістична AAA графіка	Гідний рівень графіки, але поступається UE

2.2 Графіка для 3D гри

Unity надає можливості використовувати графіку будь-якої складності. У рамках даної роботи було обрано стиль low poly, що характеризується простими 3D формами та мінімальною деталізацією моделей. Моделі прості тому що у основі стилю лежать тривимірні об'єкти, що складаються із невеликої кількості полігонів. Такий підхід спрощує оптимізацію ігор, дозволяє створювати прості яскраві світи та сконцентруватись на core-механіках гри. Серед відомих ігор у даному стилі можна відмітити Firewatch та Astroneer у якості проектів-референсів (рис. 2.1).



Рисунок 2.1 – Приклад low poly 3D графіки

2.3 Жанр гри

Для того щоб поєднати невеликий обсяг гри та показати багато різних механік, було обрано жанр «Obstacle Course». Особливостями цього жанру вважають закінчену або нескінченну смугу перешкод, яку гравець має пройти на час або рейтинг. У таких іграх висока динаміка, гравець постійно має перебувати у русі, долаючи різні перешкоди — від стрибків до телепортації. У таких іграх важлива реакція гравця та увага.

У рамках даного проекту, метою є створити закінчену смугу із різними видами перешкод, гравець завжди стартує з початкової точки та має подолати усі перешкоди якнайшвидше. Серед основних механік які може використовувати гравець – біг, стрибки та телепортація.

Такий жанр користується популярністю, особливо у простій графіці, так як спочатку концепція гри виглядає простою, але постійні невдачі гравців забезпечують постійний інтерес. До того ж, гра розвиває пам'ять, моторику, та реакцію гравців, так як при невдачі рівень повторюється. Також гра може мати сотні рівнів, нескінченні рівні, та навіть мережеву гру. Створивши

основні механіки, можна продовжувати розвивати концепцію та залучати все більше гравців різної вікової категорії.

2.4 Основні механіки гри

Основною механікою повинні бути стрибки, біг та телепортація, яка має зробити процес подолання перешкод легшим та динамічнішим, але для того щоб зберегти баланс складності, кількість телепортацій за кожен забіг обмежена. Серед десяти можливих позицій, лише в трьох випадково з'являється предмет, який дозволить персонажу переміститись на зазначену відстань. Щоб така механіка вписувалась у загальний стиль гри, її супроводжує спеціальний ефект розроблений інструментом Unity Particle System. Телепорт можна виконати як стоячи на землі, так і у польоті, до того ж, механіка має працювати таким чином, щоб гравець не застряг у перешкоді, або інших текстурах.

Біг – найголовніша механіка, метою є зробити його рівномірним, повинна бути відсутня акселерація при діагональному переміщенні, а також паралельно із цим основна камера сцени повинна слідкувати за гравцем. Так як смуга перешкод не пряма, а має повороти, додатково реалізовано повороти камери на 90 градусів, якщо гравець пройшов зазначену частину смуги. Важливим етапом на цьому моменті є анімація персонажа, він має рухатись природньо та стосовно своєї швидкості.

Стрибки – основна механіка для подолання перешкод, тож метою є визначити довжину та висоту, які забезпечать змогу переміщатись рівно стільки, скільки це потрібно для проходження різних етапів рівня. Також потрібно зробити так, щоб доки гравець не завершить попередній стрибок, у нього немає змоги зробити його знову.

У результаті проект має містити усі основні описані механіки, закінчений рівень – платформи, перешкоди, стартова та фінішні позиції, UI для меню гри та перезапуску рівня при поразці, анімації гравця, різновидності

перешкод з унікальними анімаціями для кожної, та таблицю лідерів, яка відображає найкращий час за скільки пройдено рівень. Також проект має містити звукові ефекти та музику.

Необхідні інструменти Unity для реалізації описаних основних механік наведено в таблиці 2.2.

Таблиця 2.2 – Використання необхідних інструментів для реалізації механік

Механіка	Інструменти
Рух гравця	Colliders, Animation, Unity Engine (library), Animator
Стрибки гравця	Rigidbody, Tags, Unity Engine (library), Animator
Телепорт	Tags, Rigidbody, Unity Engine (library), Particle System
Камера	Cinemachine Core
Аудіо	Audio Source, Audio Clips, Audio Listener

3 РОЗРОБКА ПРОЄКТУ

3.1 Розробка рівня

Першим кроком було зроблено основний рівень гри (рис. 3.1) – закінчена смуга з перешкодами різного типу, кожного разу гравець стартує з початкової точки та має подолати незмінну смугу, при цьому види перешкод відрізняються на різних етапах гри, тобто чим далі заходить гравець, тим більше від нього вимагається реакції та уваги.

За задумкою, рівень має початись з простого повороту, на якому знаходяться ящики – перша перешкода для гравця, яку достатньо лише перестрибнути. Далі з'являються «ваги», які гравець має перестрибнути у правильний таймінг, а також підібрати відступ для стрибку, щоб не провалитись. Після цього етапу з'являються смертельні перешкоди, які якщо задіти – рівень має перезапуститись з самого початку. Серед таких на рівні є шипи з під землі, гостре лезо, яке незмінно крутиться із деякою швидкістю, кругові леза, які переміщуються по землі, молоти що працюють за принципом маятника, а також платформи які потрібно перестрибувати на швидкість, адже вони можуть скинути гравця та не є статичними. Ассет графіки узятий із офіційного магазину Unity [3].



Рисунок 3.1 – Спроектований рівень гри

Відстані між перешкодами було встановлено так, щоб стрибки дозволяли пройти весь рівень, якщо робити їх у правильні моменти. Також на рівні є батут, який підкидує гравця високо у гору, це додає динаміки та невеликий час коли гравець може оцінити наступний етап рівня, та прийняти рішення що робити далі, адже на цьому етапі є розвилка і гравець може сам обрати шлях куди йому йти.

Щоб рівень виглядав цілісним та закінченим, рівень поміщено на воду, з правильним кутом камери це буде виглядати ніби безкрайнє море, що додає масштабу грі та візуально загальне поліпшення.

Після етапу проектування рівня, спочатку було розставлено рівень без перешкод, налаштовані колайдери, щоб модель гравця могла рухатись, а не провалюватись через текстури. І лише після цього було додано перешкоди, які залишилось лише налаштувати. Для кожної перешкоди було додано Mesh Collider, а також додано тег “Obstacle”, який потім можна використати у коді для перевірок чи торкнулась модель гравця якоїсь із перешкод. Також на кожну модель перешкоди у інспекторі було додано скрипт Trap Controller, метою було створити один скрипт для усіх перешкод, але функціонал у кожній перешкоди можна було обирати. Такий підхід дозволяв робити цікаві комбінації для логіки поведінки перешкод. Таким чином, маємо декілька полів, які можна налаштувати прямо з інспектора Unity:

- Move Type – тип переміщення, наприклад, горизонтальне, кругове чи тип «маятник»;
- Loop Mode – логіка циклічності, наприклад, тип «йо-йо», повторення з нуля, або інкрементаційна логіка;
- Rotate Mode – тип обертання, наприклад, швидкий, відносно світових координат чи локальних;
- Target Pos – тривимірний вектор цільової позиції;
- Move Time – параметр, який відповідає за швидкість;
- Delay Time – параметр, який відповідає за затримку;

– `isMoving` – булева змінна для того, щоб вказати має перешкода рухатись чи тільки обертатись.

Таким чином маємо гнучкі налаштування (рис. 3.2) для кожного з типів перешкод, а їх поєднання дозволяє отримувати унікальні механіки, що додають різноманіття для гравця, чим далі він заходить.

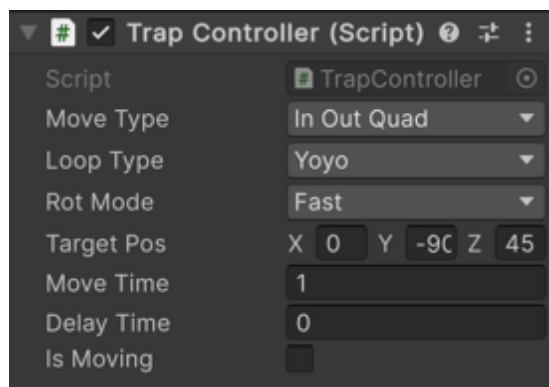


Рисунок 3.2 – Налаштування Trap Controller через інспектор

Також окремо була додана невидима текстура над рівнем моря, на неї встановлено колайдер із міткою “Is Trigger”, що вимикає усі фізичні границі колайдеру, але дає нам змогу написати код, який завершує гру якщо модель гравця пройде через таку текстуру (рис. 3.3).



Рисунок 3.3 – Спроектований рівень гри із доданими перешкодами

Вона була додана для того, щоб зробити механіку програшів у випадку, коли гравець програє не через перешкоди, а просто випадає з рівня. На таку текстуру додано тег Fall, який потім використано для реалізації такої перевірки через код.

3.2 Налаштування гравця

Під стилістику гри була обрана модель гравця (рис. 3.4) у тому ж стилі low poly, повертаючись до планування, потрібно реалізувати механіки бігу, стрибків та телепорту, для цього створений скрипт Player Controller, де прописана уся необхідна логіка для контролю ігрового персонажа. Використовуючи бібліотеку Unity Engine, для початку було створено механіку руху, відбувається вона через постійне оновлення вектору позиції гравця, який розраховується за формулою.



Рисунок 3.4 – Модель ігрового персонажа

У гравця є регульована швидкість, сила стрибка, а також константа яка відповідає за гравітацію. У коді ми визначаємо логіку, за якою стрибок

персонаж може виконати на клавішу Space лише у тому випадку, коли модель торкається моделей із тегом “Ground”. Також прописана логіка поразки, тобто коли персонаж торкнувся колайдеру із мітками Obstacle або Fall, що був вищеописаний у попередньому пункті. Для перемоги гравцеві потрібно пройти колайдер із міткою Finish, тільки тоді встановиться рекорд, який потім збережеться у пам’ять проекту та гравець зможе його побити (рис. 3.5).

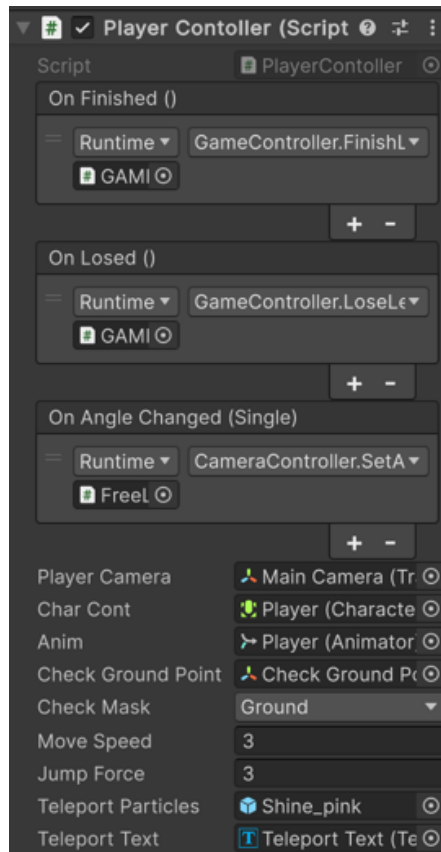


Рисунок 3.5 – Налаштування ігрового персонажа

Також було реалізовану унікальну механіку для цього жанру – телепорт. Він реалізований таким чином, що при кожному запуску рівня, з 10 слотів для предметів телепортації, обирається виключно 3 позиції. Такі предмети гравець може збирати такою ж логікою перетину колайдерів, тож якщо перетнути такий предмет, оновиться UI, який показує скільки разів у поточний час можна використати телепорт (рис. 3.6).



Рисунок 3.6 – Предмет для телепортації

На відміну від стрибка, телепорт можна використати навіть не торкаючись землі, та працює це так, що гравця перекидує на визначену відстань вперед та вгору, супроводжує це ефект створений у Particle System.

Ефект складається з чотирьох компонентів – світіння, зірки, тіні та відблиски, кожен з них налаштований за допомогою інструментів Unity, а саме швидкість ефекту, затримка, градієнт який змінюється відносно часу, розмір який також залежить від часу, форма їх розповсюдження, максимальна кількість одночасних ефектів для оптимізації, а також текстура для ефектів та їх налаштування рендерингу у світі гри.

У коді цей ефект викликається одразу після вдалої телепортації, він з'являється на місці де гравець був у момент телепорту, а також на місці куди він потрапив. При цьому, телепорт працює лише у випадку, коли він може бути вдалим, тобто перевіряється чи гравець не опиниться у текстурах після телепорту.

Ця механіка додає динаміки гри, наприклад моменти коли гравець невпевнений що зможе пройти перешкоду, він може використати телепорт, це економить час та гарантує що гравець пройде одну із перешкод. А механіка, що кожен забіг гравця місця де є предмети для телепортації випадкові, гарантує різноманіття при кожній спробі гравця пройти рівень, тож кожен раз

треба пристосовуватись до генерації телепортів. Обмежена кількість таких предметів зберігає складність рівня, адже лише три перешкоди можна пройти використовуючи телепорт (рис. 3.7).

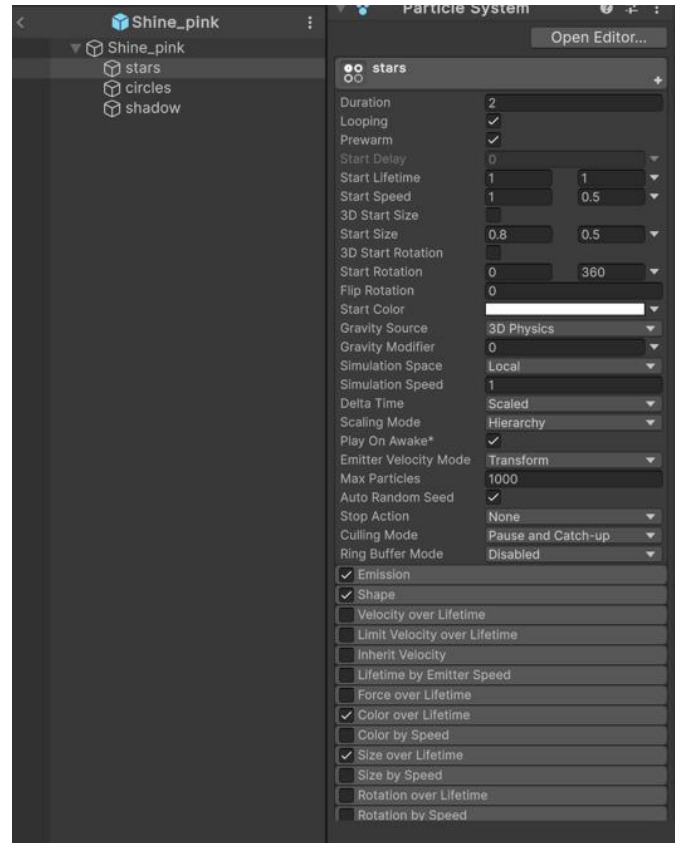


Рисунок 3.7 – Налаштування системи Particle System

Так як телепорт можна використати у польоті, це можна комбінувати із батутом в одному із етапів гри, що дозволить зекономити час який потребується на приземлення гравця (рис. 3.8).

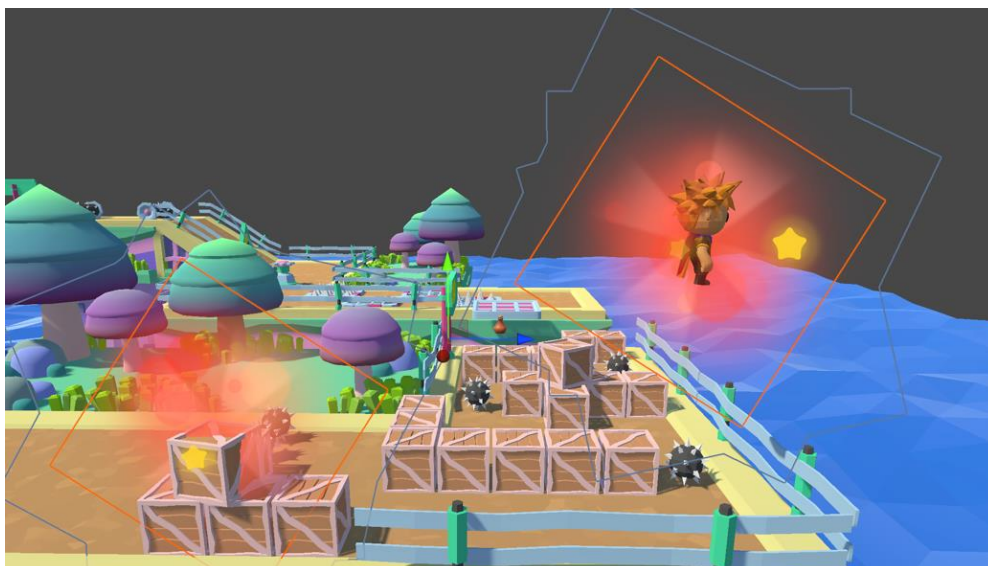


Рисунок 3.8 – Механіка телепорту разом із ефектами

Для бігу та стрибків було додано окремі анімації, для цього використані інструменти Unity, а саме Animator та Animation, для створення анімації були використані «кістки», тобто до кожної частини тіла гравця було додано об'єкт, за яким потім налаштована анімація (рис. 3.9).



Рисунок 3.9 – Кадр анімації гравця при стрибку

У купі маємо ігрового персонажа, який може переміщатись по смузі перешкод, оббігати або перестрибувати перешкоди, а також може телепортуватись щоб пройти будь-які етапи рівня швидше та без ризику. Також

налаштовані коллайдери гравця, які залежать від поточного стану персонажу та мають налаштування, при яких гравець може програти або виграти залежно від моделей з якими він стикається.

3.3 Детальна реалізація механіки телепорту

Щоб отримати ефект телепорту, у головному скрипті ігрового персонажа оголошено масив пустих об'єктів. Ці об'єкти вимкнені та слугують маркерами для зчитування вектору позиції. Як тільки рівень запускається або перезавантажується, проходиться масив з цих об'єктів та обирається 3 вектора позиції, які не мають повторюватись, маючи координати цих об'єктів, створюються об'єкти із тегом “Teleport”, вони мають візуальний вигляд пляшки, та налаштований ефект для покращеної видимості.

Як тільки гравець перетинає ці об'єкти, вони пропадають із сцени, а UI оновлюється через код, він показує гравцеві поточну кількість можливих телепортацій, аналогічно, якщо гравець витрачає заряди, код зменшує лічильник зарядів та оновлює UI.

Для того щоб гравець не телепортувався у текстури, спочатку розраховується вектор координат результатів телепортації, якщо на цьому місці відсутні будь-які текстури, тільки тоді модель персонажа переміщується на розрахований вектор.

3.4 Камера та компонент CinemachineBrain

CinemachineBrain — це ключовий компонент в Unity, який служить “мозком” для системи Cinemachine. Він виступає посередником між вашою Unity камерою та віртуальними камерами Cinemachine, забезпечуючи плавне та динамічне керування камерою у вашій грі. Він постійно відстежує всі активні віртуальні камери у сцені та вибирає ту, яка має найвищий пріоритет в даний момент. Це дозволяє легко перемикатися між різними ракурсами та

стилями зйомки в залежності від ігрової ситуації. Додатково, CinemachineBrain плавно зміщує переходи між віртуальними камерами, створюючи кінематографічний ефект. Ви можете налаштувати різні параметри змішування, такі як тривалість, криві інтерполяції та стиль переходу (рис. 3.10). Компонент також підтримує налаштування параметрів переходу [4]:

- тривалість змішування. Контролює, як довго триває зміна ракурсу;
- криві інтерполяції. Дає можливість налаштувати зміну швидкості переходу для досягнення потрібного емоційного ефекту;
- стилі переходів. Включають різні методи змішування, наприклад, лінійне або плавне.

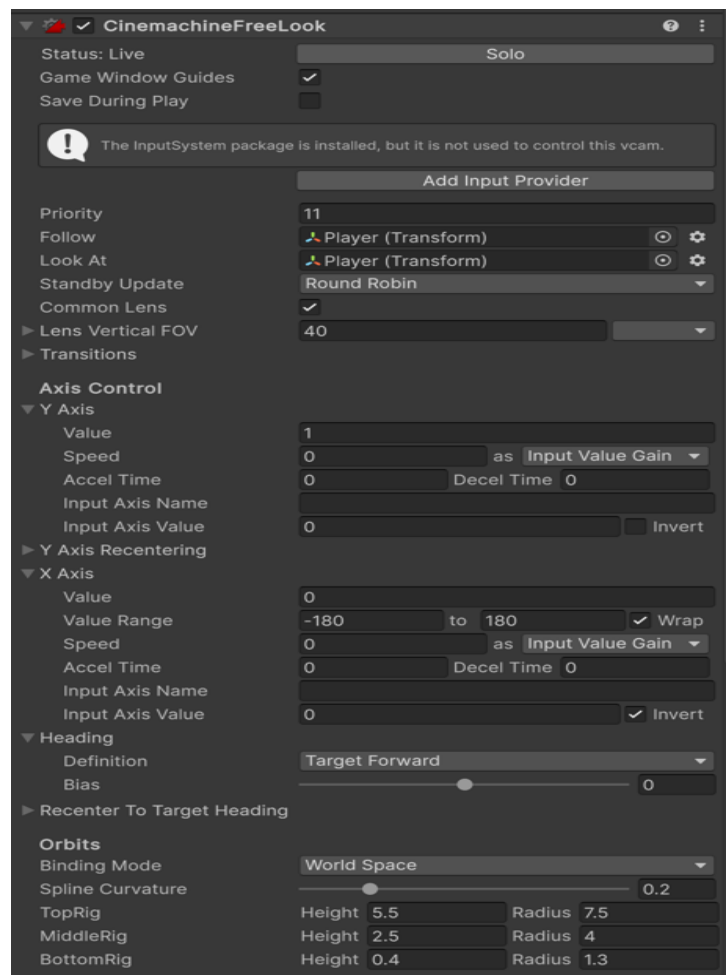


Рисунок 3.10 – Налаштування віртуальної камери

Для даного проєкту було створено об'єкт Camera Triggers, який містить дочерні пусті об'єкти, аналогічно до механіки слотів телепортації. На кожному з цих об'єктів є скрипт, який дозволяє розвертати камеру, коли гравець проходить визначений етап рівня. Це зроблено для того щоб камера завжди показувала частину рівня, яка знаходиться поперед гравця. Застосування віртуальної камери та компоненту Cinemachine гарантує плавні переходи та легко налаштовується через інспектор Unity, або через код.

Таким чином, гравець завжди бачить той етап рівня, який йому потрібно подолати, а зміна кадрів відбувається на зазначених місцях триггеру.

Порівняємо у таблиці 3.1 як даний підхід буде зменшувати затрати машинного часу, та наскільки доцільно його використовувати у порівнянні із звичайною камерою, налаштування якої відбувались би через код.

Таблиця 3.1 – Критерії та підходи

Критерій	Традиційний підхід	Cinemachine
Алгоритмічна складність	Залежить від кількості камер і логіки переходів ($O(n)$)	Постійна, обмежена лише вибором пріоритету ($O(1)$)
Витрати машинного часу	Зростають зі складністю сцени	Оптимізовані, зменшуються на ~20-30%
Гнучкість	Складно вносити зміни, потрібен код	Зручний інспектор і автоматизація
Ефекти переходів	Потребує ручної реалізації	Готові параметри змішування

3.5 Менеджери гри

Окрім скриптів, які відповідають за налаштування та взаємодію гравця з ігровим персонажем, камерою та логіку перешкод і перемоги, було додано три менеджери, які відповідають за основний процес гри (рис. 3.11).



Рисунок 3.11 – Менеджери гри

Серед цих компонентів, Input менеджер відповідає за зчитування клавіатури та гарантує механіку переміщення тільки тоді, коли гра активна, а не у стані перезапуску чи програшу. Game Controller слідує за усім що відбувається у грі, саме тут прописана логіка програшу, виграшу та показу UI у таких сценаріях.

Такий підхід дозволяє зберігати правильну архітектуру ООП за принципами SOLID, логіка результатів гри відбувається не у контролері персонажа, а у окремих модулях гри. UI Controller вмикає та вимикає UI гри, коли це потрібно. Якщо гравець програє, йому з'являється вікно де він може вийти з гри, або перезапустити рівень. Також цей модуль оновлює UI гри призначений для відслідковування часу який гравець витрачає щоб пройти рівень.

3.6 UI гри

Гра розпочинається з головного меню (рис. 3.12), яке відображає три кнопки, play – починає гру, тобто запускає сцену з грою, exit – дає можливість вийти з гри, а best – пропонує гравцеві подивитись свій кращий час, за який він пройшов рівень [5].

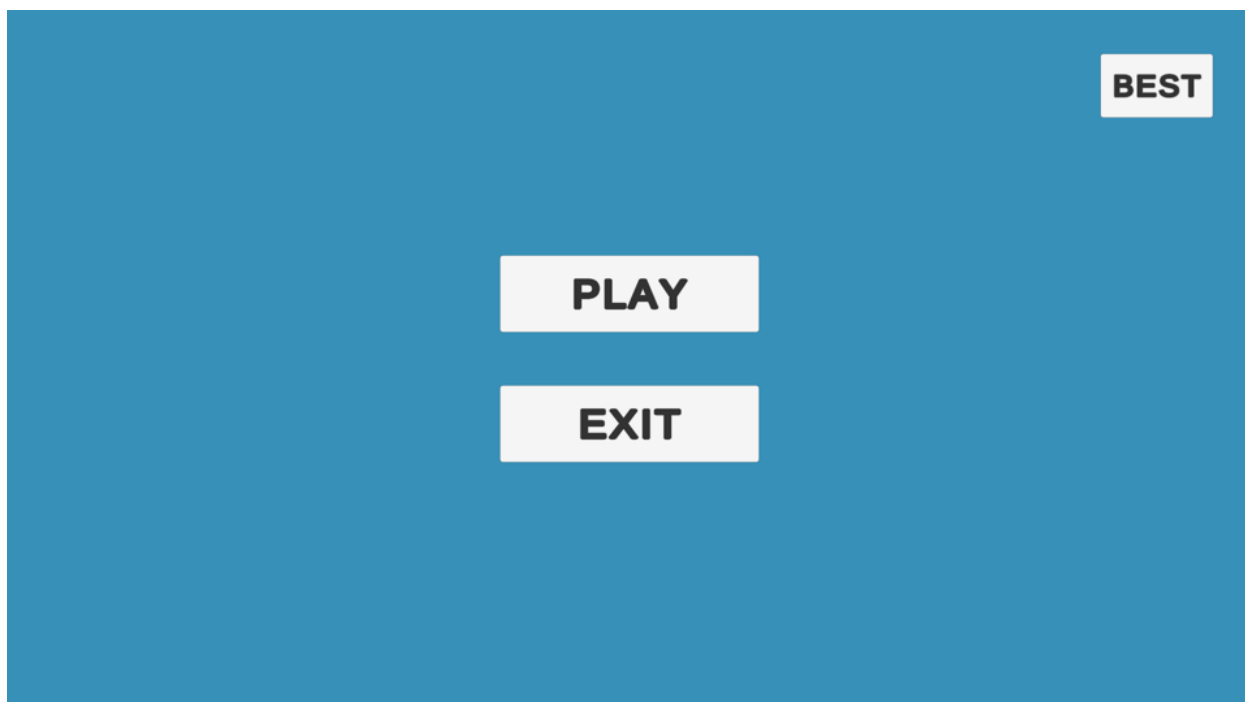


Рисунок 3.12 – Головне меню гри

Сцена головного меню та самої гри розділені для зручності гравців, і таке розділення допомагає оптимізації гри. Скрипти які працюють у головному меню не будуть працювати під час сцени з грою. Сцени налаштовані через інструменти Unity у вкладці Build Settings, де сцена з головним меню має індекс нуль, а сцена з грою – індекс 1 (рис. 3.13).

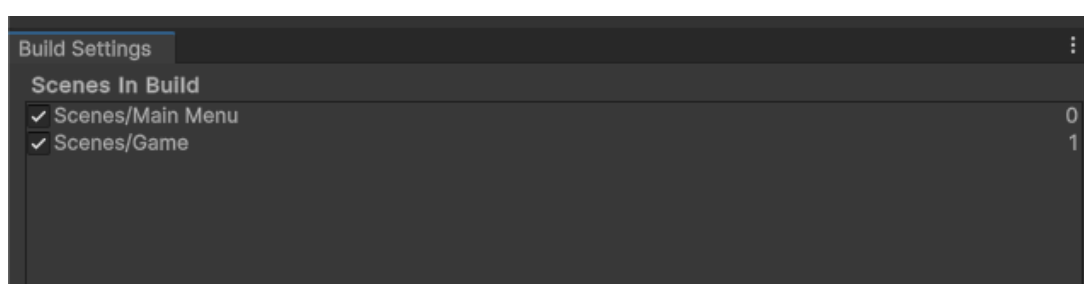


Рисунок 3.13 – Менеджмент сцен Unity

Серед ігрового UI, у сцені з грою є текст який відображає час, витрачений на поточну спробу, а також кількість можливих телепортацій. Для цього інтерфейсу налаштований шрифт та обраний білий колір, який у умовах яскравої різнокольорової гри видно найкраще (рис. 3.14).



Рисунок 3.14 – UI під час гри

Також є опціональний UI, який стає активним за умови програшу або виграшу гравця, у ньому можна перезапустити рівень, що скидує секундомер, телепорти та повертає гравця на початок рівня у стартову позицію (рис. 3.15).

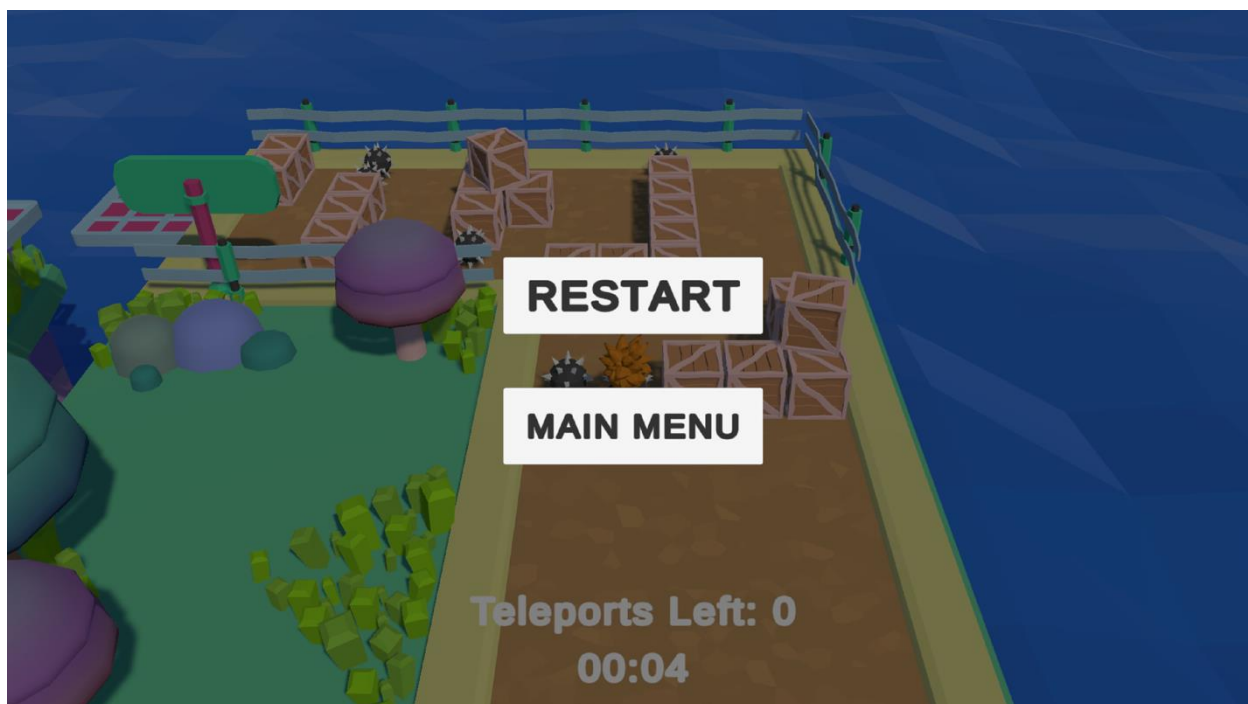


Рисунок 3.15 – UI програшу

3.6 Збереження даних гравця

Для збереження локальних даних, у даній роботі було використано інструмент Unity – PlayerPrefs. Це вбудований механізм Unity для зберігання і зчитування простих даних між ігровими сесіями. Він ідеально підходить для збереження налаштувань гри, наприклад рівня гучності, вибраної мови, стану графіки або досягнень гравця, таких як найкращий час у випадку даного проекту. PlayerPrefs дозволяє працювати з даними формату ключ-значення через C#, використовуючи текстові ключі для зберігання цілочислових, дробових та рядкових значень.

Дані, які зберігаються за допомогою PlayerPrefs, записуються у файл на пристрої користувача. Цей файл знаходиться в системній директорії, специфічній для операційної системи. PlayerPrefs зручний у використанні, адже не вимагає складного налаштування. Дані записуються за допомогою методів SetInt, SetFloat, SetString, а читаються методами GetInt, GetFloat, GetString. Для застосування змін потрібно викликати метод Save, хоча на більшості платформ Unity автоматично надає збереження даних після виклику одного з вищеперерахованих методів.

Однак PlayerPrefs має обмеження. Він підходить лише для зберігання невеликих обсягів даних. Його не варто використовувати для великих чи конфіденційних даних, оскільки вони не шифруються і можуть бути легко змінені. Для складніших випадків рекомендується використовувати інші методи збереження, такі як файли, серверні бази даних або серіалізацію.

У даному проекті PlayerPrefs використовуються у вищеописаному модулі Game Controller, та автоматично зберігає кращий результат за який була пройдена гра (рис. 3.16).

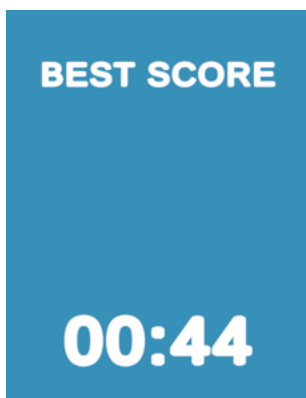


Рисунок 3.16 – Збережений рекорд секундоміра між сесіями гри

3.7 Аудіо гри

Для фонової музики використано інструмент Unity Audio Source. Це компонент, який використовується для відтворення звуку у сценах, компонент додається до об'єкта сцени і працює у поєднанні з аудіокліпом, який прив'язується до параметра Audio Clip. Після цього звук може бути відтворений за допомогою методів скрипту або автоматично, залежно від налаштувань.

У даному випадку компонент Audio Listener прив'язаний до основної камери, яка завжди активна, це дозволяє приймати усі звуки з Audio Source та відтворювати їх одночасно у динаміки пристроїв, на яких запущена гра. Для Audio Source створений окремий об'єкт Music, та налаштований під даний проект (рис. 3.17).

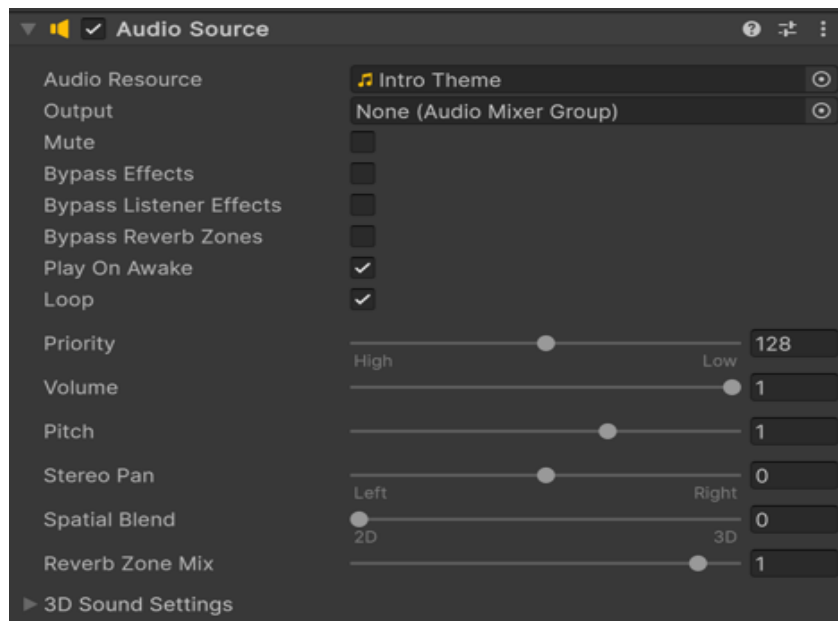


Рисунок 3.17 – Налаштований компонент для музики

Серед налаштувань виділимо Audio Resource, тут вказується файл музики який буде відіграватись, також Play On Awake забезпечить що музика увімкнеться одразу як тільки буде запущена сцена, а Loop – зациклить вказаний файл. Volume дозволяє налаштувати гучність, Pitch – ефект глибини, а Spatial Blend може забезпечити просторове звучання, наприклад для навушників, які підтримують технологію Dolby Atmos.

3.8 Фінальний продукт

У результаті ми отримуємо проект із вищеописаними механіками пункту 2. Гра має закінчений рівень, який можна проходити декілька разів, щоб покращити кращий час (рис. 3.18). Базовий UI як для меню, так і для ігрового процесу. Персонаж, який може переміщуватись, проходити перешкоди шляхом стрибків та телепортів, у грі присутня логіка програшу та виграшу, фонове аудіо та збереження даних між сеансами. Гра виконана з використанням мови програмування C# та програми Unity3D. Рівень побудовану із 3D low poly моделей у одному стилі. Архітектура проекту зберігає принципи SOLID ООП, має захищені поля, більшість з яких

серіалізується з інспектора Unity, а логіка гри відокремлена у різних модулях та менеджерах. У грі присутні базові анімації з використанням bone-animation технології та налаштовані різні види перешкод які заважають гравцеві пройти рівень.

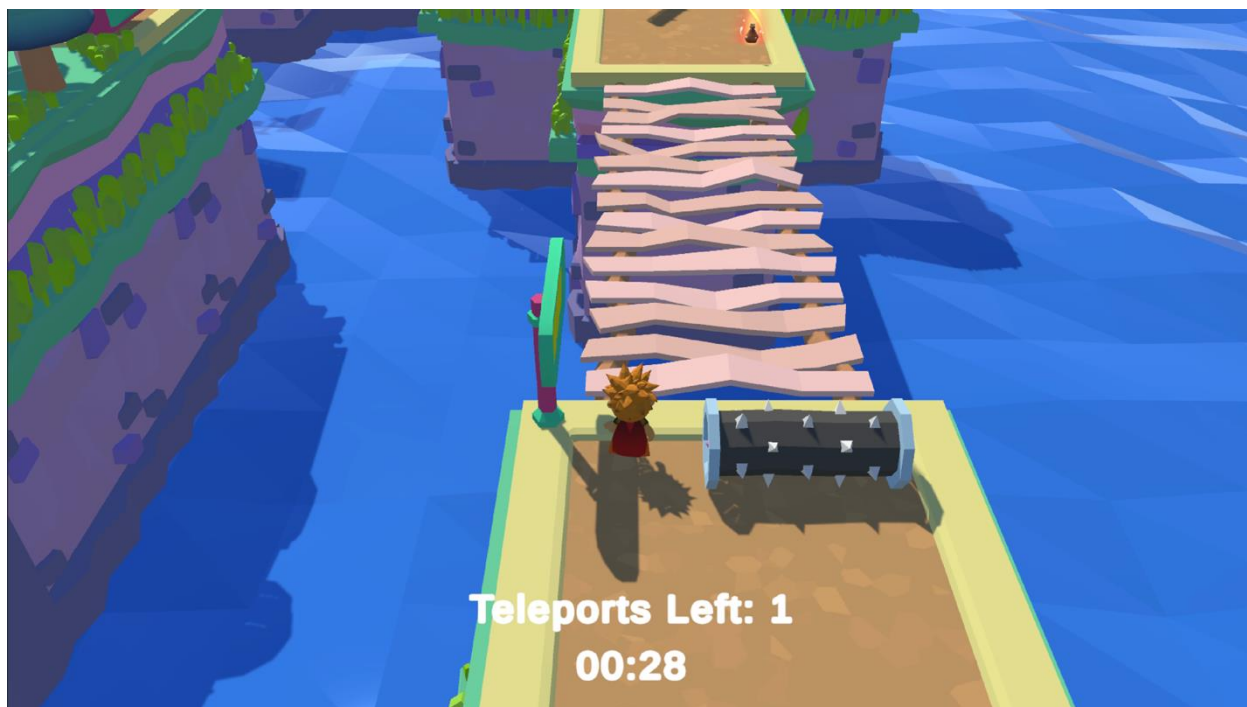


Рисунок 3.18 – Фінальний вигляд гри

ВИСНОВКИ

У результаті виконання даної роботи, були покращені базові навички по роботі з рушієм Unity3D та опановані нові професійні. У грі використані основні принципи ООП С#, зберігаючи SOLID архітектуру, продемонстровано можливості бібліотеки Unity Engine, так як через код створено багато логіки взаємодії із тегами, об'єктами, слоями, анімацією, дочірніми компонентами у рушії Unity.

Додатково було пропрацьовано інструменти для створення кісткової анімації, систему створення ефектів, проведена робота із віртуальними камерами та компонентом Cinematic Brain, налаштування звукового супроводження гри, інструменти по роботі із UI та взаємодія з їх компонентами через код. Також гра має систему зберігання даних у директорії пристрою, на якому вона запущена.

Спочатку було проаналізовано та сплановано основні етапи розробки, стиль графіки та жанр гри, продумані основні механіки, архітектура проекту та цілісний вигляд гри.

Створена гра у жанрі Obstacle Course містить стиль 3D графіки low poly, соге-механіки та систему ігрових менеджерів. Серед механік доступно переміщення, стрибок та унікальна механіка серед цього жанру – телепорт, який додає грі різноманітності завдяки продуманій логіці його використання, а також динамічність для ігрового процесу. Біг та стрибки супроводжуються кістковими анімаціями, а телепорт – створеними ефектами завдяки інструменту Particle System.

Гра має базовий UI, а саме головне меню гри, панелі для перезапуску рівня у випадку програшу або виграшу, а також відображення секундоміра та зарядів телепортів. Рівень закінчений та має велику різноманітність перешкод, які створені завдяки комбінації різних методів коду, та легко налаштовуються через інспектор Unity. Передбачені різні сценарії програшу, наприклад, коли гравець випадає за карту, або стикається з перешкодами.

Також було створено віртуальні камери, які змінюють свій ракурс залежно від етапу рівня, для цього використані складні компоненти Cinematic камери, що забезпечує зв'язок віртуальних камер із основною. Завдяки бібліотеці PlayerPrefs, гра зберігає найкращий результат між сесіями на пристрої.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Ролі у розробці ігор [Електронний ресурс]. – Режим доступу : <https://www.cgspectrum.com/career-pathways/game-development>.
2. Порівняння Unity та Unreal Engine [Електронний ресурс]. – Режим доступу : <https://hackr.io/blog/unity-vs-unreal-engine>.
3. Графіка для гри [Електронний ресурс]. – Режим доступу : <https://assetstore.unity.com>.
4. Налаштування віртуальних камер [Електронний ресурс]. – Режим доступу:
<https://docs.unity3d.com/Packages/com.unity.cinemachine@3.0/manual/CinemachineBrain.html> – налаштування віртуальних камер.
5. Налаштування UI гри [Електронний ресурс]. – Режим доступу : <https://medium.com/@michael.desena.dev/creating-ui-in-unity-6a51992db40f>.

ДОДАТОК А**Код програми**

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

namespace Assets.Scripts
{
    public class GameController : MonoBehaviour
    {
        [SerializeField] private UnityEvent<float> OnTimeChanged; // to update timer text
        [SerializeField] private UnityEvent OnFinished; // to call some code from others scripts if level passed
        [SerializeField] private UnityEvent OnLosed; // to call some code from others scripts if level losed

        private float timer = 0f;
        private bool gameIsStarted = false;

        private void Start()
        {
            StartGame();
        }

        public void StartGame()
        {
            gameIsStarted = true;
        }

        public void SetScore()
        {
            if (PlayerPrefs.HasKey("Score"))
            {
                float lastScore = PlayerPrefs.GetFloat("Score");
                if (lastScore > timer) { PlayerPrefs.SetFloat("Score", timer); }
            }

            else { PlayerPrefs.SetFloat("Score", timer); }
        }

        public void FinishLevel()

```

```

    {
        SetScore();
        OnFinished.Invoke();
    }

    public void LoseLevel()
    {
        OnLosed.Invoke();
    }

    private void Update()
    {
        if (gameIsStarted)
        {
            timer += Time.deltaTime;
            OnTimeChanged.Invoke(timer);
        }
    }
}

```

```

using Assets.Scripts;
using UnityEngine;
using UnityEngine.Events;
using TMPro;
using UnityEngine.Serialization;

```

```

public class PlayerContoller : MonoBehaviour
{
    [SerializeField] private UnityEvent OnFinished;
    [SerializeField] private UnityEvent OnLosed;
    [SerializeField] private UnityEvent<float> OnAngleChanged;

    [SerializeField] private Transform playerCamera;
    [SerializeField] private CharacterController charCont;
    [SerializeField] private Animator anim;
    [SerializeField] private Transform checkGroundPoint;
    [SerializeField] private LayerMask checkMask;
    [SerializeField] private float moveSpeed = 3f;
    [SerializeField] private float jumpForce = 3f;
    [SerializeField] private GameObject TeleportParticles;

```

```

private const float gravityForce = -9.81f;
private Vector3 velocity;
private bool onGround = true;

private int teleportCount;
[SerializeField] private TMP_Text teleportText;

public void UpdateMovePosition(Vector2 inputDir)
{
    Vector3 moveDir = playerCamera.forward * inputDir.y + playerCamera.right
* inputDir.x;

    if (moveDir.magnitude > 0f)
    {
        charCont.Move(moveDir.normalized * moveSpeed * Time.deltaTime);
    }
}

public void UpdateViewRotation(Vector2 inputDir)
{
    Vector3 dir = playerCamera.forward * inputDir.y + playerCamera.right *
inputDir.x;

    if (inputDir.magnitude > 0.1f)
    {
        var targetRotation = Quaternion.LookRotation(dir.normalized, Vector3.up);
        transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation,
Time.deltaTime * 10f);
        transform.localEulerAngles = new Vector3(0f,
transform.localEulerAngles.y, transform.localEulerAngles.z);
    }
}

private void Teleport()
{
    if (teleportCount <= 0)
    {
        Debug.Log("No teleports available!");
        return;
    }
}

```

```

    teleportCount--;
    float teleportDistance = 5f;
    float teleportHeight = 2f;

    Vector3 teleportDirection = transform.forward * teleportDistance +
    Vector3.up * teleportHeight;
    Vector3 targetPosition = transform.position + teleportDirection;

    var particle1 = Instantiate(TeleportParticles, gameObject.transform.position,
    Quaternion.identity);
    var emissionModule1 =
    particle1.gameObject.GetComponent<ParticleSystem>().emission;
    emissionModule1.rateOverTime = 0;
    SetEmissionForChildren(particle1, 0);
    Destroy(particle1, 2f);

    float checkRadius = 0.5f;
    if (!Physics.CheckSphere(targetPosition, checkRadius, checkMask))
    {
        charCont.enabled = false;
        transform.position = targetPosition;
        charCont.enabled = true;

        var particle2 = Instantiate(TeleportParticles,
    gameObject.transform.position, Quaternion.identity);
        var emissionModule2 =
    particle2.gameObject.GetComponent<ParticleSystem>().emission;
        emissionModule2.rateOverTime = 0;
        SetEmissionForChildren(particle2, 0);
        Destroy(particle2, 2f);
    }
    else
    {
        Debug.Log("Can't TP");
    }

    UpdateTeleportUI();
}

void SetEmissionForChildren(GameObject parent, float rateOverTime)
{

```

```

foreach (Transform child in parent.transform)
{
    ParticleSystem particleSystem = child.GetComponent<ParticleSystem>();
    if (particleSystem != null)
    {
        var emission = particleSystem.emission;
        emission.rateOverTime = rateOverTime;
    }
}

public void Move()
{
    anim.SetInteger("State", 1);
}

public void StopMove()
{
    anim.SetInteger("State", 0);
}

public void Jump()
{
    if (onGround)
    {
        anim.SetTrigger("Jump");
        velocity.y = Mathf.Sqrt(jumpForce * -2f * gravityForce);
    }
}

public void JumpForward(float upForce, float forwardForce)
{
    if (onGround)
    {
        anim.SetTrigger("Jump");
        velocity.y = Mathf.Sqrt(upForce * -2f * gravityForce);
        velocity.z = -Mathf.Sqrt(forwardForce * -2f * gravityForce);
    }
}

private void ApplyGravity()
{

```

```

        if (onGround && velocity.y < 0f) { velocity.y = 0f; velocity.z = 0f; }
        else { velocity.y += gravityForce * 1.5f * Time.deltaTime; }

        charCont.Move(velocity * Time.deltaTime);
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Fall"))
            || other.gameObject.CompareTag("Obstacle")) { OnLosed.Invoke(); }
        else if (other.gameObject.TryGetComponent(out CameraTrigger trigger)) {
OnAngleChanged.Invoke(trigger.GetValue()); }
        else if (other.gameObject.CompareTag("Finish")) { OnFinished.Invoke(); }
        else if (other.gameObject.CompareTag("TeleportItem"))
        {
            teleportCount++;
            Destroy(other.gameObject);
            UpdateTeleportUI();
        }
    }

    private void UpdateTeleportUI()
    {
        teleportText.text = "Teleports Left: " + teleportCount;
    }

    private void Update()
    {
        onGround = Physics.CheckSphere(checkGroundPoint.position, 0.1f,
checkMask);
        ApplyGravity();

        if (Input.GetKeyDown(KeyCode.LeftShift))
        {
            Teleport();
        }
    }
}

using System.Collections;
using System.Collections.Generic;

```

```

using UnityEngine;
using Cinemachine;

namespace Assets.Scripts
{
    public class CameraController : MonoBehaviour
    {
        [SerializeField] private CinemachineFreeLook cmCamera;

        public void SetAngle(float value) // rotate camera when angle of map is changed
        {
            Vector3 currentRotation = cmCamera.transform.rotation.eulerAngles;
            currentRotation.y += value;
            cmCamera.transform.rotation = Quaternion.Euler(currentRotation);
            cmCamera.m_XAxis.Value += value;
        }
    }
}

```

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;
using Assets.Input;
using static UnityEngine.InputSystem.InputAction;

namespace Assets.Scripts
{
    public class InputManager : MonoBehaviour
    {
        [SerializeField] private UnityEvent<Vector2> OnMoving;
        [SerializeField] private UnityEvent OnStartMove;
        [SerializeField] private UnityEvent OnStopMove;
        [SerializeField] private UnityEvent OnJumping;
        private InputController inputCont;

        private void Awake() { inputCont = new InputController(); }
        private void OnEnable() => inputCont.Enable();
    }
}

```

```

private void OnDisable() => inputCont.Disable();

public void OnStartMoving(CallbackContext context)
{
    if (context.performed) { OnStartMove.Invoke(); }
}

public void OnStopedMoving(CallbackContext context)
{
    if (context.canceled) { OnStopMove.Invoke(); }
}

public void OnJump(CallbackContext context)
{
    if (context.performed) { OnJumping.Invoke(); }
}

private void Update()
{
    OnMoving.Invoke(inputCont.Player.Move.ReadValue<Vector2>());
}
}

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using TMPro;
using System;

namespace Assets.Scripts
{
    public class MenuController : MonoBehaviour
    {
        [SerializeField] private GameObject mainMenu;
        [SerializeField] private TextMeshProUGUI scoreText;

        private void Start()
        {
            var score = PlayerPrefs.GetFloat("Score");

```

```

        var ts = TimeSpan.FromSeconds(score);

        scoreText.text = string.Format("{0:00}:{1:00}", ts.Minutes, ts.Seconds);
    }

    public void StartGame()
    {
        Time.timeScale = 1f;
        SceneManager.LoadScene(1);
    }

    public void ShowBestScore(GameObject menuToOpen)
    {
        mainMenu.SetActive(false);
        menuToOpen.SetActive(true);
    }

    public void CloseBestScore(GameObject menuToClose)
    {
        mainMenu.SetActive(true);
        menuToClose.SetActive(false);
    }

    public void QuitGame()
    {
        Application.Quit();
    }
}

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

public class RandomActivate : MonoBehaviour
{
    [SerializeField] private int objectsToActivate = 3; // Скільки об'єктів потрібно увімкнути

    void Start()
    {
        ActivateRandomChildren();
    }
}

```

```

    }

    private void ActivateRandomChildren()
    {
        Transform[] children = new Transform[transform.childCount];
        for (int i = 0; i < transform.childCount; i++)
        {
            children[i] = transform.GetChild(i);
            children[i].gameObject.SetActive(false);
        }

        int[] randomIndices = GetRandomIndices(transform.childCount,
        objectsToActivate);

        foreach (int index in randomIndices)
        {
            children[index].gameObject.SetActive(true);
        }
    }

    private int[] GetRandomIndices(int totalCount, int countToPick)
    {
        List<int> indices = new List<int>();
        for (int i = 0; i < totalCount; i++)
        {
            indices.Add(i);
        }

        for (int i = 0; i < indices.Count; i++)
        {
            int randomIndex = Random.Range(i, indices.Count);
            (indices[i], indices[randomIndex]) = (indices[randomIndex], indices[i]);
        }

        return indices.GetRange(0, countToPick).ToArray();
    }
}

```

```

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

namespace Assets.Scripts
{
    public class RotateObject : MonoBehaviour
    {
        [SerializeField] private Vector3 rotSpeed;

        private void Update() // just for good visual
        {
            transform.Rotate(rotSpeed * Time.deltaTime);
        }
    }
}

```

```

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

namespace Assets.Scripts
{
    public class Trampolin : MonoBehaviour
    {
        private void OnTriggerEnter(Collider collision)
        {
            if (collision.gameObject.TryGetComponent(out PlayerContoller
playerCont))
            {
                playerCont.JumpForward(10f, 3.5f);
            }
        }
    }
}

```

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;
using UnityEngine.SceneManagement;

namespace Assets.Scripts
{
    public class UIController : MonoBehaviour
    {
        [SerializeField] private GameObject finishMenu;
        [SerializeField] private GameObject gameOverMenu;
        [SerializeField] private TextMeshProUGUI timerText;
        [SerializeField] private TextMeshProUGUI timerTextFinishMenu;

        public void UpdateTimerText(float value)
        {
            var ts = TimeSpan.FromSeconds(value);

            timerText.text = string.Format("{0:00}:{1:00}", ts.Minutes, ts.Seconds);
            timerTextFinishMenu.text = string.Format("{0:00}:{1:00}", ts.Minutes,
ts.Seconds);
        }

        public void ShowFinishMenu()
        {
            Time.timeScale = 0f;
            finishMenu.SetActive(true);
        }

        public void ShowGameOverMenu()
        {
            Time.timeScale = 0f;
            gameOverMenu.SetActive(true);
        }

        public void RestartGame()
        {
            Time.timeScale = 1f;
            SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
        }
    }
}

```

```
    }  
  
    public void ToMainMenu()  
    {  
        SceneManager.LoadScene(0);  
    }  
}  
}
```

ДОДАТОК Б
Слайди презентації

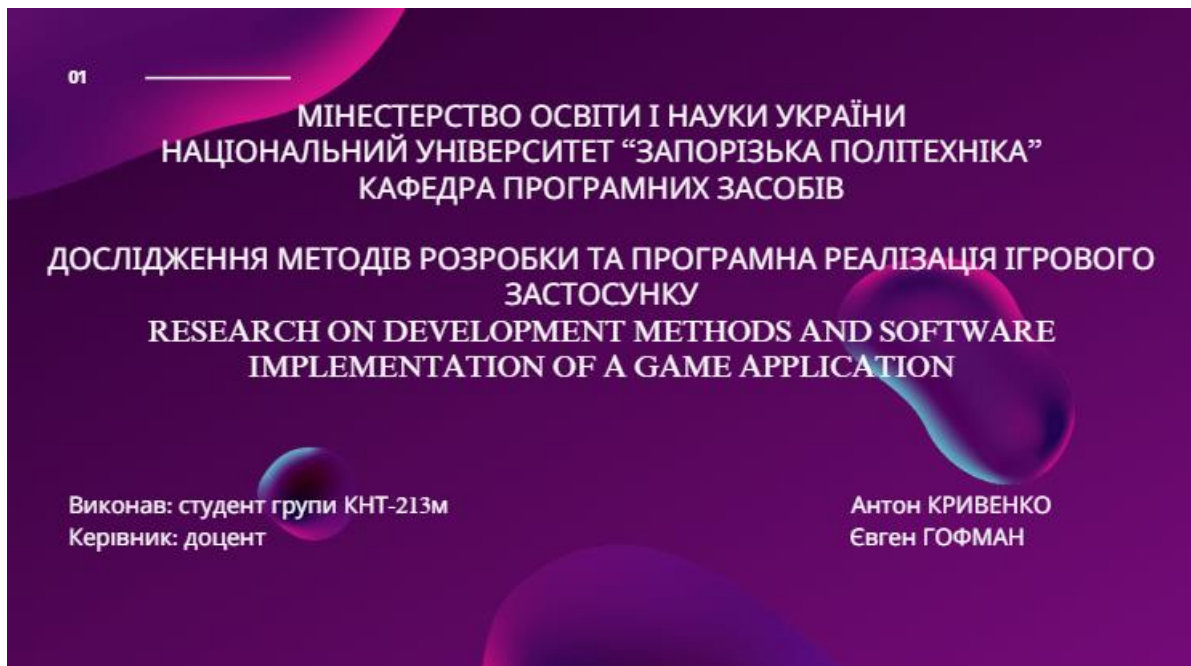


Рисунок Б.1 – Слайд 1

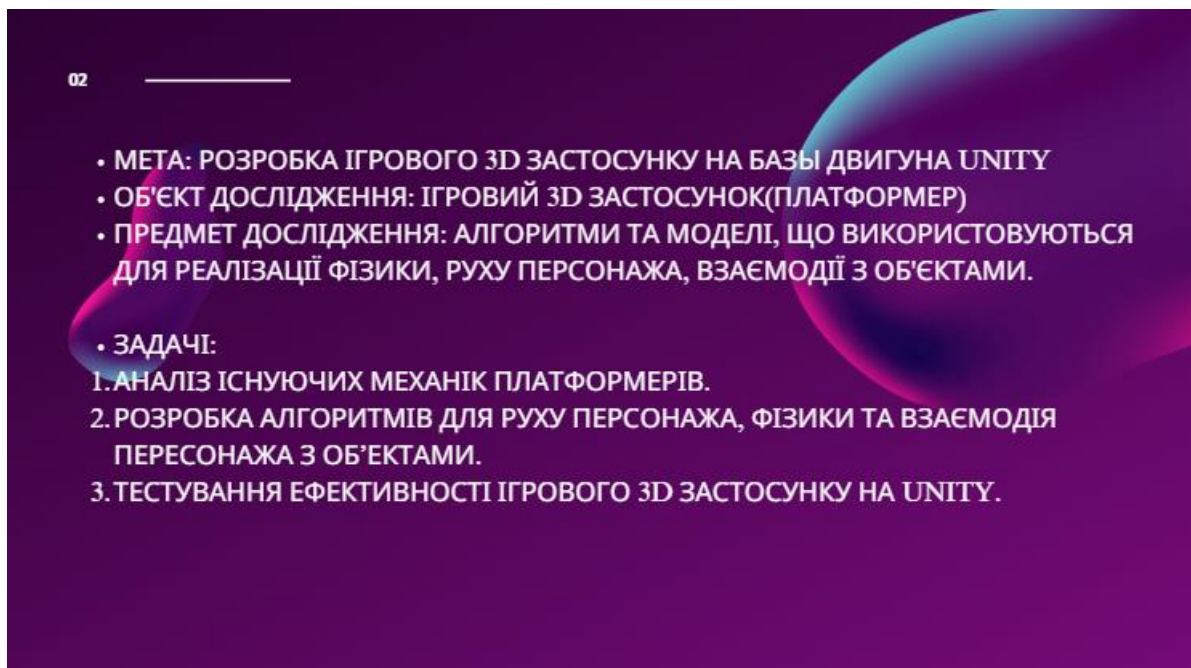


Рисунок Б.2 – Слайд 2

03

Аналіз предметної області

№	Назва ігрової механіки	Що розвиває	Алгоритмічна складність	Витрати пам'яті
1	Рух персонажа (Move)	Реакція, орієнтація	Виконується за сталий час	Використовує мінімум пам'яті
2	Стрибки (Jump)	Реакція, точність	Виконується за сталий час	Використовує мінімум пам'яті
3	Взаємодія з об'єктами (Interact)	Пам'ять, логіка	Виконується за сталий час	Використовує мінімум пам'яті
4	Вороги та перешкоди (Enemies)	Реакція, пам'ять	Залежить від кількості об'єктів	Вимагає більше пам'яті залежно від кількості об'єктів

Рисунок Б.3 – Слайд 3

04

Власні методи, моделі, алгоритми

№	Назва ігрової механіки	Що розвиває	Алгоритмічна складність	Витрати пам'яті	Примітка
1	Рух персонажа з анімацією	Реакція, візуальний ефект	Виконується за сталий час	Використовує мінімум пам'яті	Власна реалізація
2	Стрибок персонажа	Реакція, точність	Виконується за сталий час	Використовує мінімум пам'яті	Власна реалізація
3	Взаємодія з предметами	Пам'ять, взаємодія	Виконується за сталий час	Використовує мінімум пам'яті	Власна реалізація

Рисунок Б.4 – Слайд 4



Рисунок Б.5 – Слайд 5

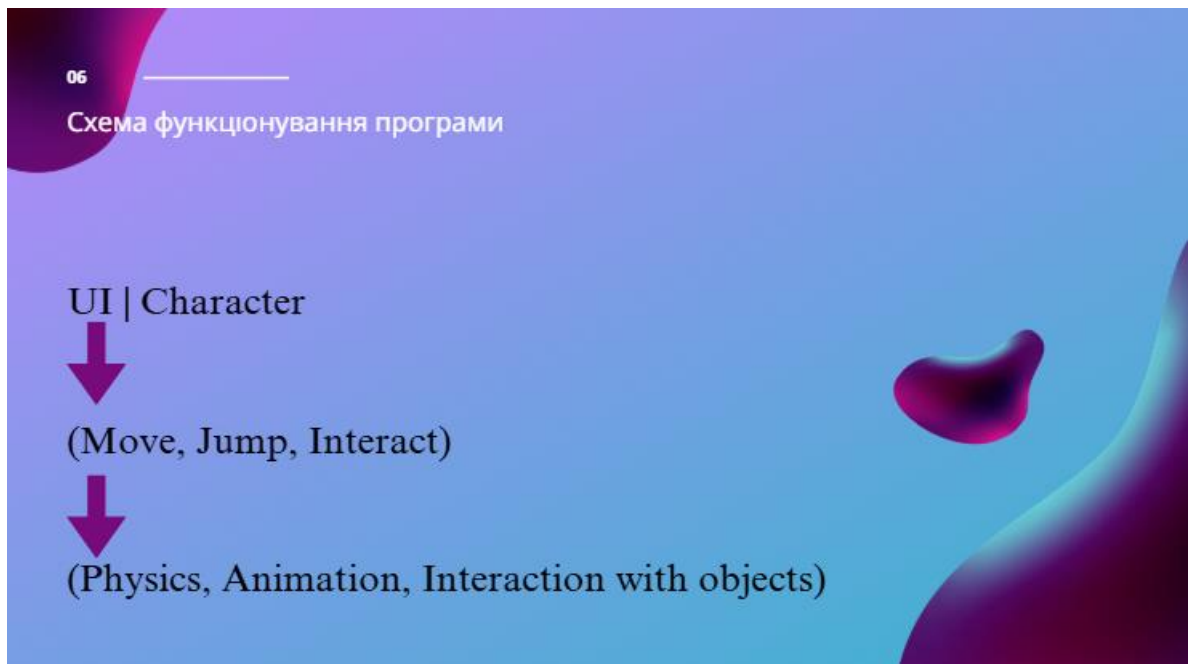


Рисунок Б.6 – Слайд 6

07

Схема даних або структури даних

Компонент	Структура даних	Роль
Система рахунку	PlayerPrefs, timer	Зберігання та оновлення найкращого рахунку гравця, відстеження часу гри.
Інвентар гравця	teleportCount (цілочисленна змінна)	Відстеження кількості доступних телепортів.
Події та взаємодії	UnityEvent (наприклад, OnFinished, OnLost, OnAngleChanged)	Обробка взаємодій, таких як завершення рівня, програш, зміна кута камери.

Рисунок Б.7 – Слайд 7

08

Вибір мови програмування та середовища розробки

Мова / Середовище	Переваги	Недоліки
C# (Unity)	Легка інтеграція з Unity, висока швидкість розробки	Обмеження на деякі низькорівневі операції
C++ (Unreal Engine)	Якісна графіка, висока швидкість роботи, можливість працювати без коду(Blueprint)	Більш актуальний під ПК та великих проектів

Рисунок Б.8 – Слайд 8

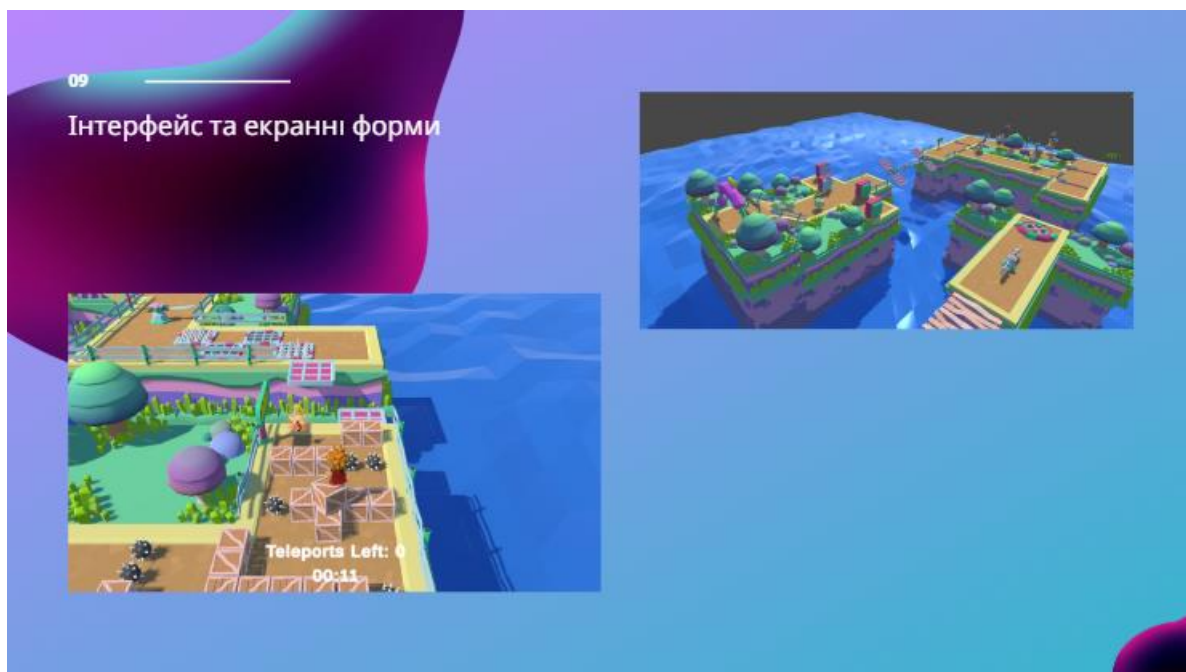


Рисунок Б.9 – Слайд 9

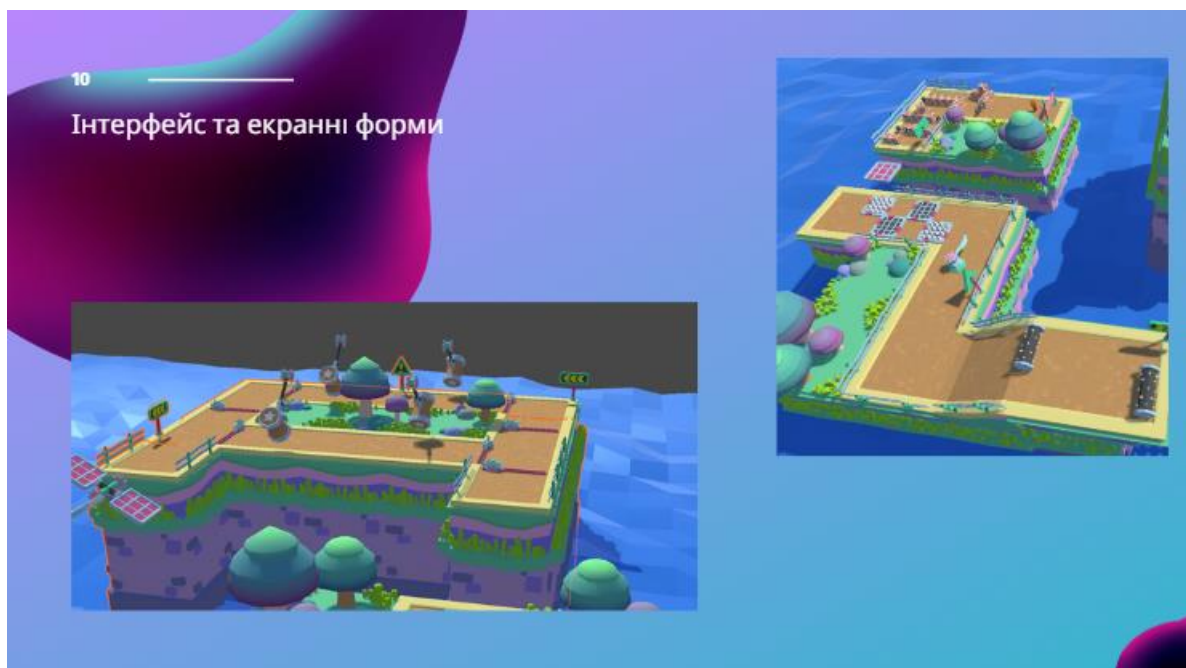


Рисунок Б.10 – Слайд 10

11

Техніко-економічні показники

Параметр	Значення
ОЗУ	32 ГБ
Процесор	Intel Core i5 (10-го покоління)
Час тестування	Середній час гри — 5 хвилин
Операційна система	Windows 10

Рисунок Б.11 – Слайд 11

12

Висновки

- **Досягнута мета:** Розроблений ігровий 3D застосунок на базі двигуна Unity.

Рішення задач:

- Розроблені механіки руху персонажа, стрибків, взаємодії з об'єктами.
- Використання фізики та анімації для покращення ігрового процесу.
- Практична цінність: Створена основа для подальшого розвитку данного проекту на Unity.

Рисунок Б.12 – Слайд 12