Міністерство освіти і науки України

Університет митної справи та фінансів

Факультет інноваційних технологій

Кафедра комп'ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота бакалавра

на тему :«Розробка 2D гри в жанрі «Симулятор офісного працівника» на основі рушія Unity»

Виконала: студентка групи ІПЗ21-2

Спеціальність 121 «Інженерія програмного забезпечення»

Лінкіна Ірина Едуардівна

(прізвище та ініціали)

Керівник к.т.н, доц. Фірсов О. Д.

(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент Університет митної справи та фінансів

(місце роботи)

Доцент кафедри кібербезпеки та

(посада)

інформаційних технологій

к.т.н., доцент Клим В.Ю.

(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2025

АНОТАЦІЯ

Лінкіна І.Е. Розробка 2D гри в жанрі симулятора офісного працівника на основі рушія Unity.

Кваліфікаційна робота на здобуття освітнього ступеня бакалавр за спеціальністю 121 «Інженерія програмного забезпечення». – Університет митної справи та фінансів, Дніпро, 2025.

Кваліфікаційна робота присвячена розробці 2D гри в жанрі симулятора офісного працівника за допомогою сучасного ігрового рушія Unity та мови програмування C#. Робота демонструє застосування сучасних інструментів та методологій розробки ігрових додатків, що є актуальним завданням у застосуванні знань і галузі розробки програмного забезпечення та геймдизайну.

В процесі дослідження було проведено аналіз сучасних методів підходів та методів до розробки ігрових додатків, а також обрано оптимальні програмні засоби для реалізації проєкту. Розробка виконувалась з використання ігрового рушія Unity та графічного редактора LibreSprite, що забезпечило створення оригінальних графічних елементів, таких як 2D спрайти, інтерфейсні кнопки ті інші візуальні компоненти. У проєкті реалізовано базову систему управління інтерфейсом, створення ігрових об’єктів та їх взаємодію.

Прикладне значення кваліфікаційної роботи полягає у розробці фундаментального ігрового застосунку, який може бути використаний як основа для подальшої модернізації функціоналу або як навчальний зразок у сфері розробки ігор.

Ключові слова: 2D-гра, Unity, LibreSprite, C#, геймдизайн, геймплей, симулятор.

ABSTRACT

Linkina I.E. Development of a 2D game in the genre of an office worker simulator based on the Unity engine.

Qualification work for a bachelor's degree in specialty 121 “Software Engineering”. - University of Customs and Finance, Dnipro, 2025.

The qualification work is devoted to the development of a 2D game in the genre of an office worker simulator using the modern Unity game engine and the C# programming language. The work demonstrates the use of modern tools and methodologies for developing game applications, which is an urgent task in the application of knowledge and the field of software development and game design.

The study analyzed modern methods of approaches and methods to the development of game applications, and selected the optimal software tools for the project. The development was carried out using the Unity game engine and the LibreSprite graphic editor, which ensured the creation of original graphic elements such as 2D sprites, interface buttons, and other visual components. The project implements a basic interface management system, the creation of game objects and their interaction.

The applied significance of the qualification work is the development of a fundamental gaming application that can be used as a basis for further modernization of functionality or as a training model in the field of game development.

Keywords: 2D game, Unity, LibreSprite, C#, game design, gameplay, simulator.

ЗМІСТ

ВСТУП

Ігрова індустрія вже багато років залишається однією з найдинамічніших та найприбутковіших галузей інформаційних технологій. Завдяки зростанню доступності інструментів розробки, збільшення попиту на мобільні та десктопні ігри, а також розвитку інді-спільноти, створення власної гри стало можливим навіть для початківців та студентів. Особливу популярність мають саме 2D-ігри, так як вони охоплюють різноманітні методи та підходи, що дозволяють спростити процес розробки створення ігор, підвищити їх якість та скоротити час розробки.

Важливим етапом у життєвому циклі будь-якого програмного продукту є його розробка, що відповідає сучасним вимогам та очікуванням гравців. При виборі програмного забезпечення необхідно звернути увагу на функціональні можливості, зручність використання та інтерфейс користувача, підтримка та оновлення. Також при розробці 2D-ігор є важливою частиною вибір відповідного ігрового рушія, переваги та недоліки якого можуть вплинути на успіх проєкту.

Актуальність полягає у тому, що розробка 2D-ігри дозволяє реалізувати комплекс практичних навичок, здобутих під час навчання. Зокрема: застосування об'єктно-орієнтованого програмування, робота з графічним і звуковим контентом, організація взаємодії з користувачем через інтерфейс створення цілісної структури додатку. У рамках кваліфікаційної роботи було створено ігровий продукт, який може бути використаний як демонстраційний приклад або як основа для подальшого вдосконалення.

Мета кваліфікаційної роботи полягає в моделюванні спрощеної діяльності офісного працівника шляхом розробки ігрового симуляційного програмного забезпечення, що поєднує інноваційні підходи до моделювання, геймплейної логіки та 2D-візуалізації на основі сучасного ігрового рушія Unity

Методи дослідження: обробка та аналіз інформації, методи проєктування та ітераційної розробки ігор, метод моделювання.

Об’єктом дослідження є розробка програмного забезпечення для створення ігрових застосунків.

У відповідності до поставленої мети в кваліфікаційній роботі поставлені такі завдання дослідження:

1) Проаналізувати технічні засоби, програмні платформи та інструменти, що застосовуються для розробки 2D-ігор.

2) Провести проєктування гри.

3) Розробка графічного контенту гри.

4) Провести тестування проєкту.

Предметом дослідження є процес розробки двовимірної комп’ютерної гри з використанням ігрового рушія Unity, що включає побудову геймплейної логіки, графічного інтерфейсу, інтеграцію аудіо та механізмів взаємодії з користувачем.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел з 15 найменувань. Загальний обсяг роботи — 50 сторінок кваліфікаційної роботи, 14 рисунків, 2 таблиці.

Практичне значення результатів полягає у створенні базової версії симулятора, яка може бути розширена до повноцінного комерційного продукту.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Розвиток індустрії комп'ютерних ігор та роль 2D-проєктів

Індустрія комп'ютерних ігор на сьогодні є однією з провідних сфер у глобальному IT-просторі. За останні два десятиліття ця сфера не просто зміцнила свої позиції на ринку розваг, перетворившись на повноцінний культурний феномен, але й почала активно проникати в інші галузі, на кшталт як культура, маркетинг, освіта та навіть медицина. Гейміфікація — явище, що стало рушієм нових форм взаємодії з користувачем, заснованим на механіках, запозичених саме з цифрових ігор. Це свідчить про універсальність та ефективність ігрового підходу в сучасних інформаційних системах. [1]

Індустрія комп'ютерних ігор пройшла шлях від початкових дослідів до всесвітнього культурного та економічного феномену, демонструючи вражаючу динаміку розвитку та безпрецедентні темпи збільшення. Заснована в середині XX століття з простих текстових ігор та аркадних автоматів, вона швидко еволюціонувала, поглинаючи нові технології та розширюючи свої творчі обрії. Ранні етапи розвитку характеризувалися обмеженими технічними можливостями, що зумовлювало домінування двовимірної графіки. Саме в цей період закладалися фундаментальні принципи ігрового дизайну, механік та наративів, які згодом стали класичними. Ігри, накшталт «Space Invaders», «Pac-Man» та «Super Mario Bros», не лише завоювали мільйони сердець, а й стали знаковими, сформувавши підґрунтя для подальших інновацій. Їхня простота у візуальному виконанні була компенсована глибоким геймплеєм, інтуїтивним управлінням та високою реграбельністю, що сприяло формуванню лояльної аудиторії.

З появою могутніших апаратних засобів наприкінці 1990-х та початку 2000-х років, індустрія зазнала чималого технологічного стрибка, зосередившись на опануванні тривимірної графіки. Це призвело до появи нових жанрів та помітного підвищення реалістичності віртуальних світів. Багато розробників та

гравців почали асоціювати прогрес лише з переходом до 3D, що на деякий час посунуло 2D-проєкти на другий план. Проте, попри домінування тривимірних ігор, двовимірні проєкти ніколи цілком не зникали з горизонту індустрії. Вони продовжували розвиватися у нішевих сегментах, часто знаходячи прихисток серед інді-розробників, які цінували їхню гнучкість, менші витрати на виробництво та здатність до експериментів з нетрадиційними ігровими механіками.

Останні роки ознаменувалися помітним відродженням зацікавлення до 2D-проєктів, що можна пояснити кількома чинниками. По-перше, зріла аудиторія, яка виросла на класичних 2D-іграх, відчуває ностальгію за цим стилем, що стимулює попит на ретро-проєкти та їх сучасні інтерпретації. По-друге, технологічний прогрес дозволив підняти візуальну якість 2D-графіки на новий рівень, використовуючи сучасні ефекти освітлення, тіней та анімації, що робить її не менш, а інколи й більш привабливою, ніж деякі 3D-моделі. По-третє, 2D-формат пропонує розробникам унікальні можливості для зосередження на глибині ігрового процесу, інноваційних механіках та художньому стилю, де відсутність третього виміру спонукає до більшої креативності в обмеженому просторі. Це особливо актуально для інді-студій, які можуть реалізувати свої унікальні ідеї з меншими бюджетами та часовими рамками, ніж при створенні ААА-проєктів. Отже, 2D-ігри сьогодні не є лише відлунням минулого, а є активним та динамічним сегментом індустрії, що постійно розвивається, пропонуючи гравцям свіжі та оригінальні враження. Вони демонструють, що інновації та успіх в ігровій індустрії не завжди залежать від технологічної складності, а великою мірою визначаються креативністю та якістю ігрового дизайну.

У межах ігрової індустрії на сьогодні особливе місце займають 2D-ігри. Ця ніша активно розвивається, пропонуючи гравцям унікальний естетичний досвід. Зокрема, завдяки відносній простоті реалізації, низьким вимогам до апаратних ресурсів кінцевого користувача, суттєво швидшому циклу розробки та унікальній естетичній привабливості, особливо піксельного стилю, 2D-ігри

стали основою для незалежних інді розробок, ефективних проєктів та домінуючою формою мобільних ігор.

Останніми роками в Україні спостерігається активне зростання зацікавленості до інді-розробки ігор. Це зумовлено як глобальними тенденціями в ігровій індустрії, так і локальними чинниками — доступністю інструментів, великою кількістю технічно підготовлених фахівців, розвитком IT-кластерів та спільнот. Зокрема, дедалі більше студентів і молодих розробників беруть участь у геймджемах (Global Game Jam, Ludum Dare, Indie Cup Ukraine), де часто втілюються саме 2D-проєкти через їхню технологічну простоту. [11]—[12]

Українські інді-команди створюють проєкти, що здобувають міжнародне визнання, наприклад «Stoneshard», «The Final Station» або «Sheriff Rage». Ці приклади підтверджують, що навіть невеликі команди, маючи достатні навички та ідею, спроможні реалізувати конкурентоздатний продукт. Важливу роль у цьому процесі відіграє саме розробка 2D-ігор, оскільки вона не вимагає значних фінансових чи технічних ресурсів, а отже — є ідеальною відправною точкою для молодих фахівців.

Симулятори як жанр не тільки моделюють конкретні процеси чи професії, а й мають важливе психологічне значення в контексті навчання. Вони сприяють розвитку таких когнітивних умінь, як логічне мислення, адаптація до змін, аналіз наслідків дій, стратегічне планування та концентрація. Ігрові симуляції дозволяють користувачам діяти у змодельованому середовищі з контрольованими параметрами, де можна без ризику вивчати нові принципи, тестувати гіпотези та відпрацьовувати стратегії. У контексті простої гри-симулятора, як у даному проєкті, ці елементи реалізовані через систему завдань і ухвалення рішень, що вимагає від гравця аналізу поточної ситуації та прогнозування наслідків. Отже, навіть базова ігрова механіка здатна сприяти формуванню корисних навичок, що згодом можуть бути застосовані в більш складних середовищах або професійних тренажерах.

Розробка гри в 2D-середовищі дозволяє концентруватися не лише на технічних аспектах реалізації програмного коду, а й на логіці взаємодії з

користувачем, дизайні рівнів, створенні сценаріїв і побудові повноцінного та інтуїтивно зрозумілого інтерфейсу користувача. Це забезпечує цілісне занурення в реальні умови проєктування програмного продукту, відтворюючи повний життєвий цикл розробки від концепції до готового рішення.

Але також розробка навчальних та демонстраційних 2D-ігор висуває низку потреб, що охоплюють як технічний, так і методичний аспекти. Зокрема, такі ігри мають бути технологічно простими, швидко розгортатися на різних платформах та забезпечувати ефективну взаємодію з юзером.

Вони повинні демонструвати базові принципи побудови архітектури програмного забезпечення, логіки, обробки подій та управління станами. Однією з важливих потреб є наявність чіткого інтерфейсу користувача, який дає змогу без труднощів взаємодіяти з ігровими об'єктами. Також рекомендовано наявність простого геймплейного циклу, який не перевантажує гравця, але дозволяє вибудувати логіку прийняття рішень. Завдяки цим якостям навчальна гра не лише виконує дидактичну функцію, а й виступає платформою для відпрацювання практичних навичок з розробки ПЗ, зокрема роботи з рушієм, UI, збереженням даних та обробкою подій.

Інтерактивність, гнучкість і кросплатформеність роблять ігри потужним інструментом для побудови не лише розважальних продуктів, а й навчальних платформ, симуляторів або експериментальних додатків. У контексті освітньої підготовки спеціалістів з інформаційних технологій, розробка ігор дозволяє закріпити на практиці такі навички як проєктування програмної архітектури, використання шаблонів програмування, робота з графікою, створення UI/UX, тестування, налагодження та багато іншого. Таким чином, створення гри як кваліфікаційного застосунку є обґрунтованим вибором як з точки зору актуальності та відповідності сучасним тенденціям ринку, так і з точки зору прикладного значення результатів.

1.2 Особливості жанру симуляторів та концепція ігрового проєкту

У межах жанрової класифікації ігор симулятори є одним із найбільш гнучких і варіативних форматів. Основна їх особливість полягає в імітації реального або умовно реального процесу, системи чи діяльності, в якому користувач виконує повторювані дії та приймає рішення, що впливають на загальний результат та подальший розвиток подій. Симулятори можуть відтворювати як складні технічні операції, так і побутові, робочі або економічні сценарії, що робить їх привабливими для широкої аудиторії.

Жанр симулятора особливо цікавий з точки зору програмної реалізації, оскільки зосереджується на повторюваній логіці, сценаріях ухвалення рішень та моделюванні процесів, максимально наближених до справжнього чи умовно-реального середовища. Такі ігри потребують чіткої логічної структури, обліку змінних параметрів, реалізації внутрішньої економіки чи інших параметрів розвитку. У межах освітнього контексту симулятор дозволяє моделювати поведінку системи, одержувати зворотний зв'язок, впроваджувати стратегії управління ресурсами, а також відображати наслідки дій гравця. Це робить жанр максимально корисним, оскільки він дозволяє охопити широкий спектр навичок — від UI-розробки до збереження станів та анімацій.

Для реалізації ігрового застосунку було обрано концепцію спрощеного симулятора офісної праці. Така тематика є зрозумілою широкому колу користувачів, легко адаптується під ігрову логіку, не потребує складної фізики або тривимірної моделі. Основна мета користувача — утриматися на роботі, приймаючи або відхиляючи завдання, що надходять із певною періодичністю. У результаті гравець накопичує внутрішньоігрові ресурси, такі як гроші, та підвищує свій рівень репутації, на основі чого відбувається оцінка його успішності.

Обрана концепція дозволяє поєднати кілька ключових елементів сучасного геймдизайну в єдину систему. До них належать визначений ігровий цикл, поступове нарощування складності завдань та механік, інтегрована внутрішньоігрова економіка, система бонусів та покращень, звуковий супровід та візуальне зворотне інформування користувача через елементи інтерфейсу. Гра

має просту структуру, але логічно завершений сценарій, що дозволяє сприймати її як повноцінний продукт.

1.3 Аналіз інструментів для розробки 2D-ігри

Сучасний ринок пропонує розробникам широкий вибір програмного забезпечення та інструментів для створення 2D-ігор, що дозволяє підібрати оптимальний стек технологій під будь-який проєкт. У таблиці 1.1 нижче наведені найвідоміші ігрові рушії з їх особливостями — Unity, Godot, GameMaker Studio, Construct та Unreal Engine. Кожен із них має свої особливості, сильні сторони й обмеження, але обрано було саме рушій Unity.

Таблиця 1.1

Ігрові рушії з їх особливостями

Рушій	Мова програмування	Особливості
Unity	C#	Потужний, гнучкий, багатофункціональний, кросплатформений
Godot Engine	GDScript, C#, C++	Open-source, легкий, але менш стабільний
GameMaker Studio	GML	Призначений для 2D, платний
Construct	Блокова логіка	Візуальна розробка, без коду
Unreal Engine	C++ / Blueprints	Потужний, але переважно для 3D

Unity було обрано як базовий рушій для реалізації даного симулятора, оскільки він поєднує в собі зручний та інтуїтивно зрозумілий графічний редактор сцен, що значно спрощує візуальне проєктування та розміщення елементів. Компонентну архітектура Unity сприяє модульній та легко масштабованій розробці. Наявність підтримки C#, однієї з найбільш популярних та потужних

мов у галузі розробки ігор, дозволяє реалізувати складну логіку. Модульна структура, широкі можливості інтеграції графіки, анімації, звуку та UI-елементів роблять його універсальним інструментом. [2]

Крім того, значними перевагами Unity є його обширна та актуальна документація, надзвичайно активне та розгалужене ком'юніті розробників, що забезпечує швидкий доступ до допомоги та ресурсів, а також доступність безкоштовної ліцензії для некомерційного та навчального використання.

Візуальне оздоблення є одним з основних чинників, що формують перше враження від гри та впливають на загальний досвід користувача. У 2D-іграх найчастіше використовуються такі стилі: піксель-арт, flat-дизайн, векторна графіка, hand-drawn стиль, ізометричне зображення тощо. Кожен з них має свої технічні та естетичні властивості. Піксель-арт, який обрано для даного задуму, поєднує ностальгійність, стилізовану простоту та економічність. Він не потребує складних шейдерів, освітлення чи великих графічних ресурсів, що робить його оптимальним для навчальних та інді-проєктів. Крім того, піксельний стиль дозволяє створювати чітко читаємі інтерфейси та елементи геймплею, що позитивно впливає на юзабіліті. Простота та впізнаваність піксель-арту роблять його зручним інструментом для швидкого прототипування й адаптації до різних роздільностей екрану, для чого потрібно обрати зручний для користування редактор. [13]

При розробці 2D-графіки для гри важливо підібрати такий інструмент, котрий не тільки дозволяє продуктивно працювати зі спрайтами й анімацією, ай відповідає вимогам до інтеграції, зручності та економічності. У таблиці 1.2 нижче наведено порівняння найвідоміших піксель арт-редактори.

LibreSprite — це вільне програмне забезпечення з відкритим вихідним кодом, яке є форком Aseprite, проте доступне без ліцензійних обмежень. Його головною перевагою є наявність функцій професійного рівня (анімація, таймлайн, підтримка шарів), відсутність плати, а також сумісність із ігровими рушіями, зокрема Unity. У контексті навчального проєкту LibreSprite виявився

найкращим варіантом завдяки простому інтерфейсу, низьким системним вимогам і можливості створення ресурсів без грошових витрат. [5]

Таблиця 1.2

Порівняння піксель арт-редакторів

Застосунок	Тип ліцензії	Сумісність з Unity	Орієнтація	Вимоги до ресурсів
LibreSprite	Безкоштовний	+	Піксель-арт	Низькі
Aseprite	Платний	+	Піксель-арт	Низькі
Adobe Photoshop	Платний	+	Загальна графіка	Високі

На відміну від Adobe Photoshop, котрий є потужним багатофункціональним редактором, проте потребує платної підписки та складніший в опануванні, LibreSprite дозволяє зосередитися саме на потребах розробки піксельної графіки. Aseprite, хоч і має подібний інтерфейс, вимагає купівлі ліцензії, що унеможливлює його застосування у вільнодоступних навчальних проєктах.

Отже, LibreSprite є оптимальним вибором для розробки ігор у піксельному стилі в умовах навчального або інді-проєкту. Його використання дозволяє заощадити ресурси, не втрачаючи при цьому функціональності та зручності у роботі.

Комбінація Unity як основного ігрового рушія та LibreSprite як є оптимальним вибором для навчального 2D-проєкту. Таке поєднання дозволяє реалізувати повний цикл розробки, від концептуального та візуального оформлення до інтерактивної логіки, подальшого тестування та фінального результата для ПК.

1.4 Постановка задачі та технічна концепція проєкту

Даний дипломний проєкт має за мету створення повноцінного навчального ігрового продукту, який об’єднує в собі ключові технічні та функціональні елементи типової 2D-гри, демонструючи практичні навички розробки програмного забезпечення. У фокусі реалізації – симуляція офісної або праці, де гравцеві пропонується динамічно реагувати на вхідні завдання та приймати рішення, що безпосередньо впливають на внутрішньоігрові параметри, як-от фінансові ресурси та репутація. Основне завдання гравця полягає в ефективному "утриманні на роботі", що вимагає безперервного аналізу ситуації та ухвалення стратегічних рішень, оскільки накопичення внутрішньоігрових грошей та підвищення рівня репутації є ключовими показниками його успішності та прогресу в грі.

Технічна концепція розробленого симулятора базується на створенні інтерактивного середовища з чітко визначеним ігровим циклом та взаємопов'язаними системами. Ігровий процес циклічний, починається з генерації нового завдання, що є центральним елементом взаємодії гравця. Завдання розрізняються за рівнем складності, включаючи легкі, середні та складні варіанти, що безпосередньо впливає на обсяг потенційної винагороди та часові обмеження на його виконання. Для кожної категорії складності передбачені окремі набори текстових описів завдань, які зберігаються у відповідних масивах у класі GameManager.cs (easyTasks, mediumTasks, hardTasks). З них GameManager випадковим чином обирає рівень складності, а потім і саме завдання, забезпечуючи тим самим реграбельність та непередбачуваність ігрового процесу, оскільки гравець щоразу зіштовхується з новими викликами.

Для кожного завдання надається обмежений часовий ліміт, значення якого залежить від складності. Цей таймер (currentTaskTimer) реалізований як змінна типу float у GameManager.cs, яка зменшується щокадру, а його візуалізація здійснюється за допомогою UI-елемента Slider (timerSlider), який динамічно оновлюється, надаючи гравцеві візуальний зворотний зв'язок про залишок часу. Керування переходом між завданнями, включаючи короткі паузи після

виконання або пропуску, здійснюється за допомогою корутин (coroutines) у C#, що забезпечує плавний ігровий потік без блокування основного циклу виконання гри. Гравцеві надається вибір: виконати завдання, натиснувши "Complete Button", або пропустити його, скориставшись "Skip Button". Ці кнопки викликають відповідні методи CompleteTask() та SkipTask() у GameManager.cs. Успішне виконання завдання призводить до нарахування грошей та репутації, тоді як пропуск завдання (через вичерпання часу або свідоме рішення) веде до штрафів та збільшення лічильника пропущених завдань (missedTasksCount).

Внутрішньоігрова економіка є ключовим стимулом для гравця та основою системи прогресії. Вона базується на двох основних ресурсах: грошах та репутації, що зберігаються та управляються у класі UIManager.cs, який відповідає за їхнє відображення на екрані та оновлення. Центральним елементом прогресії є внутрішньоігровий магазин (shopPanel), який містить список доступних покращень (availableUpgrades), представлених об'єктами класу UpgradeData.cs. Кожен об'єкт UpgradeData інкапсулює інформацію про покращення, включаючи його унікальний ідентифікатор (id), назву (upgradeName), опис (description), вартість (cost), поточний (currentLevel) та максимальний рівень (maxLevel), а також числове значення впливу (effectValue) на ігровий процес. Візуальне представлення кожного покращення в магазині забезпечує клас ShopItemUI.cs, який динамічно відображає його характеристики та обробляє натискання кнопки «Buy».

Куплені покращення активно впливають на ігровий процес через метод ApplyUpgradeEffect() у GameManager.cs, який застосовує конкретні ефекти, як-от збільшення часу на завдання (Speed Boost), підвищення винагороди (Networking Skills), збільшення ліміту пропущених завдань (Stress Management), генерація пасивного доходу (Smart Investments), зниження штрафів (Backup System) або тимчасове збільшення репутації (Marketing Campaign).

Отже, технічна концепція проєкту передбачає створення модульної, розширюваної системи на основі Unity та C#, котра результативно моделює діяльність офісного працівника, дає гравцеві змогу впливати на перебіг подій

через прийняття рішень та систему вдосконалень, і забезпечує зрозумілий візуальний та звуковий зворотний зв'язок, що робить гру захоплюючою та інтерактивною.

1.5 Висновок до першого розділу

У ході аналізу предметної області визначено актуальність розробки 2D-ігор як ефективного інструменту для навчання і практичного засвоєння навичок програмування. Обґрунтовано доцільність використання саме формату симулятора як гнучкого, універсального та зрозумілого з точки зору реалізації геймплею. Проведено порівняння можливих інструментів, у результаті чого було обрано рушій Unity для розробки і LibreSprite — для створення графічного оформлення. Сформульовано загальну технічну концепцію проєкту та окреслено основні компоненти, які підлягають реалізації у процесі роботи над кваліфікаційною роботою.

2.2 АНАЛІЗ ТА ВИБІР МЕТОДІВ РЕАЛІЗАЦІЇ ГРИ

2.1 Вибір програмних засобів для реалізації проєкту

Дієва реалізація будь-якого програмного проєкту, особливо в сфері інтерактивних додатків, неможлива без ретельного аналізу наявних технологічних рішень та обґрунтованого вибору методів їх втілення. Цей розділ присвячений критичному огляду актуальних інструментів та підходів, що можуть бути застосовані для розробки сучасних ігрових симуляторів, а також обґрунтуванню конкретного стеку технологій, підібраного для створення симулятора управління задачами. Вибір програмного середовища та інструментарію є вирішальним чинником, що впливає на швидкість розробки, гнучкість архітектури, можливості масштабування та, зрештою, на якість кінцевого продукту. Від цього рішення залежить не тільки технічна успішність застосунку, але й економічна доцільність та зручність подальшої підтримки.

Процес розробки програмного забезпечення, зокрема ігрових додатків, є багатогранним і вимагає застосування спеціалізованих інструментів на кожному етапі – від концептуалізації та дизайну до кодування та тестування. Правильний вибір цих інструментів є ключовим для досягнення поставлених цілей проєкту. На сучасному ринку представлено безліч рішень, які розрізняються функціональністю, рівнем складності, вартістю, підтримуваними платформами та мовами програмування. Для кваліфікованого вибору потрібно здійснити порівняльний аналіз, враховуючи специфіку проєкту, наявні ресурси, а також необхідність забезпечення гнучкості та можливості подальшого розвитку.

2.1.1 Порівняння ігрових рушіїв та середовища для розробки симуляторів

Ігрові рушії є основою будь-якої сучасної гри, надаючи розробникам інтегроване середовище для створення візуального контенту, реалізації ігрової логіки, управління ресурсами та інших критично важливих аспектів. При виборі

рушія для симулятора важливо зважати на його можливості щодо роботи з інтерфейсом користувача, моделюванням внутрішньої логіки системи, а також підтримкою різноманітних платформ для розгортання. У сучасній індустрії існує велика кількість двигунів, кожен з яких орієнтований на певні завдання, типи ігор та рівень складності розробки.

Серед провідних ігрових рушіїв на сьогодні можна виокремити Unity, Godot Engine та Unreal Engine. Unreal Engine, розроблений Epic Games, відомий своєю винятковою фотореалістичною графікою та потужним функціоналом для створення AAA-проєктів. Він переважно використовує C++ та систему візуального програмування Blueprints. Хоча Unreal Engine є надзвичайно потужним, його фокус більше зосереджений на 3D-іграх зі складною графікою, а поріг входження для вивчення може бути досить високим, особливо для розробки 2D-симуляторів з простим візуальним стилем. Це зробило його менш оптимальним вибором для нашого застосунку, де акцент був на логіці та інтерфейсі, а не на високореалістичній графіці.

Godot Engine є відкритою альтернативою, що стрімко набирає популярність. Він є легким, гнучким і підтримує різні мови програмування, включаючи свою власну мову GDScript, C# та C++. Godot ідеально підходить для 2D-ігор, пропонуючи зручні інструменти для роботи зі спрайтами та анімаціями. Його «відкритість» дає повний контроль над рушієм. Проте, порівняно з Unity, Godot має менше розгалужене ком'юніті, що може ускладнити пошук рішень для специфічних проблем, і його екосистема асетів не настільки розвинена. Для проєкту, який створюється в рамках дипломної роботи з обмеженим часом, більш широка підтримка та готові рішення Unity були перевагою. [8]

Unity від Unity Technologies, з іншого боку, є одним з найпопулярніших ігрових рушіїв у світі, що підтримує як 2D, так і 3D розробку. Його основною мовою програмування є C#, що є потужною об'єктно-орієнтованою мовою, яка інтегрується з Microsoft Visual Studio – провідним середовищем розробки для .NET. Ця комбінація надає розробнику надзвичайно потужні інструменти для налагодження, рефакторингу та управління кодом. C# в Unity реалізується через

компонентну архітектуру, що дозволяє створювати гнучкі та модульні системи, де кожен об'єкт може мати декілька скриптів-компонентів, які відповідають за певний функціонал. Цей підхід є ідеальним для розробки симуляторів, оскільки дозволяє легко створювати та модифікувати різні аспекти ігрової логіки, такі як управління ресурсами, поведінка завдань, взаємодія з UI.

Зокрема, для симуляторів, Unity пропонує виняткові можливості у царині UI/UX. Його вбудована система Canvas дає змогу створювати складні та динамічні інтерфейси, які легко адаптуються під різні розміри екранів. Це критично важливо для симулятора офісної праці, де значна частина взаємодії відбувається через кнопки, слайдери, текстові поля й інші елементи інтерфейсу. Можливість візуального дизайну UI безпосередньо в редакторі Unity, поєднуючи його з програмною логікою на C#, значно пришвидшує розробку.

Крім того, Unity відрізняється кросплатформенністю, що дозволяє легко експортувати проєкт для різних операційних систем (Windows, macOS, Linux), мобільних платформ (Android, iOS) та навіть веб-браузерів (WebGL). Це забезпечує широке охоплення аудиторії та гнучкість у розгортанні продукту. Велика та активна спільнота, величезна кількість навчальних матеріалів, уроків, форумів та готових асетів у Unity Asset Store суттєво прискорюють процес розробки та дають змогу швидко знаходити рішення для типових і нетипових завдань. Для проєкту, де важливо швидко освоїти інструментарій та отримати робочий результат, ці фактори є вирішальними. [9]

Під час реалізації програмної частини гри було проаналізовано декілька варіантів середовищ розробки, зокрема Visual Studio Community, JetBrains Rider, MonoDevelop, а також Visual Studio — текстовий редактор з розширеними можливостями для роботи з мовою C#. З огляду на специфіку проєкту, доступність, продуктивність та інтеграцію з Unity, було обрано саме Visual Studio Code.

Visual Studio є легким та швидким редактором коду з відкритим вихідним кодом, що підтримує велику кількість мов програмування, зокрема C#, за допомогою встановлення відповідних розширень, таких як C# for Visual Studio

Code. Однією з ключових переваг VS Code є його висока швидкість роботи та низькі вимоги до ресурсів системи, що особливо важливо на етапі навчальної розробки, коли проєкти не потребують складного інтегрованого середовища з великою кількістю сервісів.

У порівнянні з повноцінним Visual Studio Community, VS Code запускається значно швидше, не потребує попередньої конфігурації й не навантажує систему додатковими службами, які не застосовуються під час створення невеликих проєктів. В той час як Visual Studio забезпечує розширені можливості налагодження та профілювання, для втілення типової логіки 2D-гри ці можливості були надлишковими. Натомість VS Code забезпечив комфортну роботу з файлами, підтримку IntelliSense, автодоповнення, навігацію по класах і методах, а також швидке перемикання між файлами.

Ще одним варіантом було середовище JetBrains Rider — потужна IDE на основі IntelliJ, яка має глибоку інтеграцію з Unity. Проте воно є комерційним продуктом, а його безкоштовна версія доступна лише для людей із верифікованим акаунтом. Окрім того, Rider значно вимагає ресурсів системи, що робить його менш придатним для освітніх потреб або роботи на слабших пристроях.

MonoDevelop, котрий раніше активно застосовувався з Unity, зараз вже не підтримується, має обмежену функціональність, а в останніх версіях Unity офіційно не радиться до використання.

Отже, вибір Visual Studio є оптимальним для реалізації 2D-гри в навчальному середовищі. Він забезпечує швидку інтеграцію з Unity, має низьке навантаження на систему, підтримує основні функції налагодження та автодоповнення, а також дозволяє легко організовувати працю над проєктом. Використання цього середовища сприяло підвищенню продуктивності праці та зменшенню технічних бар'єрів під час реалізації ігрової логіки.

Саме ці переваги — потужна інтеграція з C# та Visual Studio, розвинена система UI, кросплатформенність та активна підтримка спільноти — стали ключовими аргументами на користь вибору Unity як основного ігрового рушія

для реалізації цього симулятора управління задачами. Це забезпечує не лише технічну ефективність, але й гнучкість у подальшому розвитку та вдосконаленні застосунку.

2.1.2 Інструменти для моделювання та дизайну контенту

Побудова інтерфейсу користувача є важливою складовою кожного ігрового проєкту, особливо у жанрі симулятора, де взаємодія з меню, панелями та кнопками є постійною. Unity пропонує потужну UI-систему на основі об'єктів Canvas, що дозволяє створювати як прості, так і важкі елементи інтерфейсу. Завдяки наявності Layout Group (Vertical, Horizontal, Grid) і компонентів типу Button, Image, TextMeshPro, Slider та інших, розробник може зручно структурувати інтерфейс, зробити його адаптивним до різних розширень екрана та легко змінювати без потреби редагування коду.

Окрему перевагу дає TextMeshPro — система рендерингу тексту, яка дозволяє використовувати шрифти високої чіткості, додавати тіні, контури та інші ефекти. Це особливо актуально для 2D-ігор, де текст є ключовим складником інтерфейсу. Залежно від складності гри, інтерфейс може будуватись як на одному Canvas з усіма елементами, так і на декількох шарах — наприклад, окремо для HUD, меню та вікон діалогів. У розробці цієї гри було застосовано розділення логіки UI (UIManager.cs) і логіки геймплею (GameManager.cs), що дозволило досягти високої гнучкості й незалежності між відображенням і даними.

Якість та стилістика візуального та звукового контенту є невід'ємною складовою будь-якої гри, впливаючи на занурення гравця та його загальне сприйняття продукту. Для 2D-ігор, особливо тих, що використовують піксельну графіку, вибір відповідних інструментів для створення арт-активів є надзвичайно важливим.

Для створення всієї графічної складової, що виконана у стилі піксель-арт, було обрано LibreSprite. Це безкоштовний, з відкритим вихідним кодом,

піксельний арт-редактор, який є форком, безкоштовною незалежною версією популярного Aseprite. LibreSprite ідеально підходить для роботи з піксельною графікою завдяки своєму спеціалізованому функціоналу: він дозволяє легко створювати та редагувати спрайти, працювати з шарами, використовувати анімаційні кадри та ефективно маніпулювати кольоровими палітрами. Можливість створення тайлів – невеликих графічних елементів, з яких можна будувати ігрові рівні – є однією з ключових переваг LibreSprite для 2D-ігор, що значно прискорює процес дизайну рівнів та фонів. Альтернативами могли б бути такі інструменти, як Adobe Photoshop, GIMP або сам Aseprite, але LibreSprite був обраний саме через його орієнтованість на піксель-арт, простоту застосування та безкоштовність, що відповідає вимогам роботи.

Отже, комбінація Unity для розробки логіки й інтеграції, LibreSprite для піксельної графіки утворює оптимальний комплект інструментів, що дозволяє реалізувати всі аспекти 2D-ігрового проєкту, починаючи від візуального дизайну, і закінчуючи інтерактивною логікою та звуковим супроводом, забезпечуючи високу якість кінцевого продукту в рамках ресурсів та строків дипломної роботи.

2.2 Методи реалізації ключових компонентів гри

Реалізація ігрового симулятора вимагає системного підходу до розробки його основних компонентів, кожен з котрих виконує свою унікальну функцію та взаємодіє з іншими частинами системи. Завдяки використанню Unity як ігрового рушія та C# як мови програмування, архітектура проєкту будується на принципах об'єктно-орієнтованого програмування та компонентного підходу.

В основі ігрової логіки лежить компонентна архітектура Unity. Кожен ігровий об'єкт може мати декілька компонентів, що представляють різні аспекти його поведінки – фізику, рендеринг, скрипти. Для реалізації основної логіки гри (управління станом гри, перемикання сцен, обробка подій) буде використано патерн Singleton для головного менеджера гри, що забезпечує єдину точку

доступу до централізованих функцій з будь-якої частини проєкту. Це полегшує управління глобальними параметрами та координацію взаємодії між різними системами (UI, завдання, магазин). Логіка оновлення стану гри та реакції на дії користувача реалізується через методи життєвого циклу MonoBehavior у скриптах C#.

Ядро ігрового процесу симулятора офісної праці базується на динамічній системі генерації та керування завданнями. Завдання розрізняються за рівнем складності, що впливає на винагороду та час на виконання. Реалізація передбачає використання списків завдань для кожного рівня складності, з яких випадковим чином обирається поточне завдання. Це забезпечує реграбельність та непередбачуваність ігрового процесу. Для відліку часу на виконання кожного завдання використовується таймер, що реалізований через слайдер UI, і його значення динамічно оновлюється. Керування часом та перехід між завданнями здійснюється за допомогою корутин (coroutines) у C#, що дозволяє асинхронно виконувати послідовності дій без блокування основного потоку виконання гри.

Для моделювання внутрішньоігрової економіки введено дві ключові метрики: гроші та репутація. Гроші заробляються за успішне виконання завдань, а репутація збільшується при завершенні завдань, що мають більший вплив. Невиконання або пропуск завдань призводить до штрафів, а саме зменшення грошей або репутації, що додає елемент ризику та стратегії. Магазин покращень є центральним елементом прогресії гравця. Кожне покращення представлено окремим об'єктом даних (наприклад, класом UpgradeData), що зберігає його вартість, опис та ефект. Реалізація магазину включає відображення доступних покращень в UI, перевірку наявності коштів для покупки та застосування ефектів покращень на ігрову логіку (наприклад, зміна базового часу на завдання або розміру винагороди). Система дає змогу розширювати асортимент магазину у майбутньому.

Розробка UI є ключовим аспектом симуляторів. В Unity UI створюється за допомогою системи Canvas та елементів Unity UI (кнопки, текстові поля TextMeshPro, слайдери, панелі). Для динамічного оновлення інформації, тобто

гроші, репутація, опис поточного завдання, таймер, використовуються відповідні компоненти TextMeshProUGUI та Slider, які програмно пов'язуються зі змінними в класах UIManager та GameManager. Обробка подій, таких як натискання кнопок, реалізується через систему подій Unity, що дозволяє легко прив'язувати методи скриптів до дій користувача. Динамічні контролюються через активацію або деактивацію GameObjects.

Процес розробки супроводжується безперервним тестуванням та налагодженням. Налагодження коду здійснюється безпосередньо в Visual Studio, інтегрованому з Unity, що дає змогу ставити точки зупину, breakpoints, переглядати значення змінних та відслідковувати хід виконання програми. Функціональне тестування ігрових механік проводиться вручну, перевіряючи коректність розрахунків, поведінку UI-елементів та послідовність ігрового циклу. Наявність логів у консолі Unity також значно полегшує виявлення і виправлення помилок.

2.2.1 Принципи проєктування ігрових механік для симуляторів

Проєктування ігрових механік для симуляторів потребує глибокого розуміння не тільки програмних аспектів, але й психологічних факторів взаємодії юзера з системою. Вдалий симулятор має не просто імітувати реальність, а й надавати гравцю чітку мотивацію, відчуття прогресу та занурення у віртуальний світ. У контексті розробки симулятора офісної праці було враховано декілька ключових принципів, що забезпечили його ігровий потенціал.

Одним з фундаментальних аспектів є створення ефективних систем прогресії та зворотного зв'язку. Гравцю необхідно постійно відчувати, що його дії мають наслідки і ведуть до розвитку. У нашому проєкті це реалізується через дві основні метрики: гроші та репутацію. Кожне успішно виконане завдання безпосередньо впливає на ці показники, відображаючись на екрані та створюючи миттєвий позитивний зворотний зв'язок. Система поліпшень у магазині є ще

одним рівнем прогресії: купівля нового поліпшення не лише змінює ігровий баланс (наприклад, збільшуючи винагороду або скорочуючи час на завдання), але й надає гравцю відчуття досягнення та контролю над своєю віртуальною кар'єрою. Цей елемент забезпечує довгострокову мотивацію, оскільки гравець прагне зібрати достатньо ресурсів для розблокування потужніших апгрейдів.

Досягнення найкращого балансу ігрового процесу є вкрай важливим для утримання зацікавленості гравця. Надто легка гра швидко набридне, а надто важка – викличе розчарування. В симуляторі баланс досягається через варіативність задач за складністю та часом на виконання, а також через систему винагород і штрафів. Ефекти від поліпшень теж ретельно калібруються, щоб вони надавали відчутну користь, але не робили гру надто простою. Наприклад, збільшення ліміту пропущених завдань або пасивний дохід додають гнучкості та стратегічної глибини, але не цілком усувають виклик. Такий підхід стимулює гравця до поліпшення своєї «ефективності» та управління ризиками.

Реграбельність забезпечується механізмом випадкової генерації завдань з різних категорій складності. Оскільки задачі вибираються динамічно з попередньо визначених масивів, кожний ігровий сеанс може мати унікальну послідовність викликів, що запобігає одноманітності та заохочує гравця до повторних проходжень. Додатково, система поліпшень також сприяє реграбельності, оскільки гравець може експериментувати з різними стратегіями розвитку, обираючи різні покращення та спостерігаючи за їхнім впливом на ігровий процес. Це додає елемент стратегії та дозволяє гравцю знаходити свій найкращий шлях до перемоги.

Зрештою, створення імерсії (занурення) у віртуальний світ симулятора досягається через сукупність візуальних, звукових та інтерактивних складників. Візуальна частина, виконана у стилі піксель-арт, разом з відповідним фоновим зображенням офісу, створює певну атмосферу робочого простору. Інтерактивні елементи UI, такі як динамічний таймер, оновлення показників грошей та репутації, а також візуальні зміни кнопок у крамниці, забезпечують інтуїтивно зрозумілу взаємодію та підтримують постійний зв'язок гравця з ігровою

системою. Всі ці складники працюють разом, щоб створити цілісний та переконливий ігровий досвід, який спонукає гравця занурюватися у віртуальну симуляцію.

2.3 Висновок до другого розділу

Підсумовуючи розгляд наявних рішень, що здійснено в цьому розділі, можна констатувати, що вибрані інструменти, середовища та методи реалізації проєкту є повністю обґрунтованими з точки зору поставлених цілей кваліфікаційної роботи. Усі рішення були прийняті з урахуванням сучасних вимог до розробки 2D-ігор, а також особливостей навчального й демонстраційного програмного забезпечення.

Здійснений порівняльний аналіз рушіїв показав, що серед доступних платформ для створення 2D-проєктів (Unity, Godot, GameMaker, Construct, Unreal Engine) саме Unity забезпечує найкраще поєднання простоти, гнучкості та функціонального наповнення. Важливо підкреслити, що рушій має розвинену систему роботи з 2D-графікою, підтримує всі сучасні стандарти організації інтерфейсу та об'єктно-орієнтовану архітектуру, що дозволяє реалізовувати масштабовані та модульні проєкти.

Важливою перевагою Unity стало також те, що він активно підтримує мову програмування C#, яка має високий рівень читабельності, підтримує сучасні конструкції об'єктно-орієнтованого програмування, та широко застосовується в індустрії. Використання C# сприяє ефективному структуруванню логіки, формалізації геймплейних взаємодій та використанню відомих патернів, зокрема Singleton для глобального керування ігровим станом. Саме завдяки компонентній моделі Unity було реалізовано незалежні блоки логіки, пов'язані з UI, обробкою завдань, звуковими ефектами та взаємодією з ігровими змінними.

Суттєву роль у формуванні зручного робочого процесу відіграло обране середовище розробки — Visual Studio Code. Завдяки своїй швидкості, підтримці C# через розширення OmniSharp, можливостям автодоповнення, навігації по

класах і методах, воно дозволило ефективно реалізувати скрипти, що відповідають за основну логіку гри. Інтеграція з Unity відбувалась безперешкодно, що підтверджує доцільність використання VS Code у подібних проєктах.

Графічна складова гри була реалізована за допомогою редактора LibreSprite. Його вибір зумовлений необхідністю створення піксельної 2D-графіки в стилістиці ретро-ігрових додатків. LibreSprite повністю відповідає цим вимогам: він має всі базові функції для створення спрайтів, підтримує шари, палітри, анімацію кадрів і дозволяє експортувати ресурси в форматах, придатних для використання в Unity. Його інтерфейс зручний навіть для новачків, а відкритий вихідний код забезпечує гнучкість і відсутність ліцензійних обмежень.

Таким чином, аналіз наявних інструментів, технологій і підходів до розробки ігор дозволив обґрунтовано сформувати технічний стек для реалізації симулятора, що відповідає критеріям ефективності, простоти, доступності та масштабованості. Вибір Unity, C#, LibreSprite та Visual Studio Code забезпечив оптимальні умови для створення функціонального навчального ігрового продукту, а реалізовані методи і принципи дозволили закласти надійну архітектурну основу, що відповідає сучасним підходам до програмної інженерії.

РОЗДІЛ 3. РОЗРОБКА 2D-ГРИ В ЖАНРІ СИМУЛЯТОРА ОФІСНОГО ПРАЦІВНИКА

У цьому розділі розглянуто етапи практичної реалізації 2D-гри у жанрі симулятора офісного працівника з використанням рушія Unity. Реалізація включає створення меню, ігрової сцени, розробку інтерфейсу користувача, спрайтів, внутрішньоігрового магазину, а також налаштування логіки генерації та обробки завдань.

3.1 Архітектура та структура програмного забезпечення

Ефективність і подальша масштабованість будь-якого програмного продукту значною мірою залежать від його архітектури та внутрішньої структури. На етапі проектування було обрано підхід, який дозволяє забезпечити гнучкість, модульність та зручність підтримки коду. Це особливо актуально для ігрових застосунків, де логіка може бути досить складною та потребувати постійних доповнень.

Архітектура гри реалізована за модульним принципом, де кожен функціональний елемент винесено в окремий скрипт. Основу побудови становить скрипт «GameManager.cs», котрий виконує роль центрального керуючого елемента. Він відповідає за глобальне управління, а саме ініціалізацію гри, генерацію та управління завданнями з різними рівнями складності, обробку їх виконання або пропуску, контроль над внутрішньоігровою економікою, а також за логіку завершення гри — перемоги чи поразки, та координує взаємодію з іншими менеджерами. У ньому реалізовано основний ігровий цикл, що включає генерацію завдань, реакцію на дії гравця та оновлення внутрішньоігрових параметрів, а також застосування ефектів від покращень. На кшталт як бонусний час, бонусні гроші або репутація, зменшення штрафів, пасивний дохід та маркетингові кампанії.

У сцені Unity створено об’єкти Canvas для відображення текстових і графічних елементів, що є основою користувацького інтерфейсу. Клас UIManager.cs відповідає виключно за відображення інформації користувачу та обробку базових UI-взаємодій. Його основна функція – оновлення текстових полів (гроші, репутація, опис завдання), стану слайдерів та кнопок. UIManager отримує дані від GameManager або інших компонентів і відображає їх, не містячи складної ігрової логіки. Він зберігає поточні значення грошей та репутації, а також текст поточного завдання, забезпечуючи їх актуальне відображення на екрані. Таке розділення відповідальності між GameManager (логіка) та UIManager (відображення) забезпечує чистоту архітектури та спрощує налагодження.

Взаємодія між цими класами відбувається наступним чином: GameManager ініціалізує список UpgradeData та передає кожен об'єкт UpgradeData до відповідного ShopItemUI через метод ShopItemUI.Setup(). Коли користувач натискає кнопку "Купити" на ShopItemUI, останній делегує запит на покупку до GameManager, викликаючи GameManager.PurchaseUpgrade(). GameManager перевіряє наявність коштів через UIManager.currentMoney, обробляє транзакцію, застосовує ефект покращення через ApplyUpgradeEffect(), та оновлює стан ShopItemUI та UIManager. AudioController відтворює звуки за запитом GameManager або UIManager, наприклад, PlayButtonClickSFX().

Також у сцені Unity створено об’єкти Canvas для відображення текстових і графічних елементів, як видно на скріншотах та. Крім того присутні окремі об'єкти, такі як AudioController для керування звуком, UIManager для керування виводом даних, а також ShopPanel, який є частиною ієрархії Canvas і керується GameManager, та інші об’єкти, пов’язані з інтерфейсом. Кожен об’єкт має відповідний компонент або скрипт, що дозволяє досягнути високої гнучкості в реалізації логіки гри.

3.2 Етапи створення спрайтів та кнопок до гри

Процес розробки візуального контенту для 2D-гри,зокремавстилі піксель-арт, вимагає послідовного підходу та використання спеціалізованих знарядь. Це забезпечує єдність стилю та ефективність інтеграції графічних активів у ігровий двигун.

На першому етапі здійснювалося створення всіх необхідних спрайтів. Для цього використовувався редактор LibreSprite, що ідеально підходить для піксельної графіки. Розробка починалася з базових елементів фону, таких як офісне приміщення, меблі, вікна, що формують основне ігрове середовище, як це видно з рис. 3.1. Далі створювалися спрайти для ігрових об'єктів, таких як персонаж гравця, елементи, що відображають завдання, та інші дрібні деталі, що збагачують візуальне відчуття. Особлива увага приділялася деталізації, щоб зберегти характерний для піксель-арту стиль, водночас забезпечуючи чіткість та впізнаваність елементів. Кожен спрайт створювався з урахуванням його подальшого використання в анімаціях, тому розроблялися окремі кадри для кожного руху чи стану.

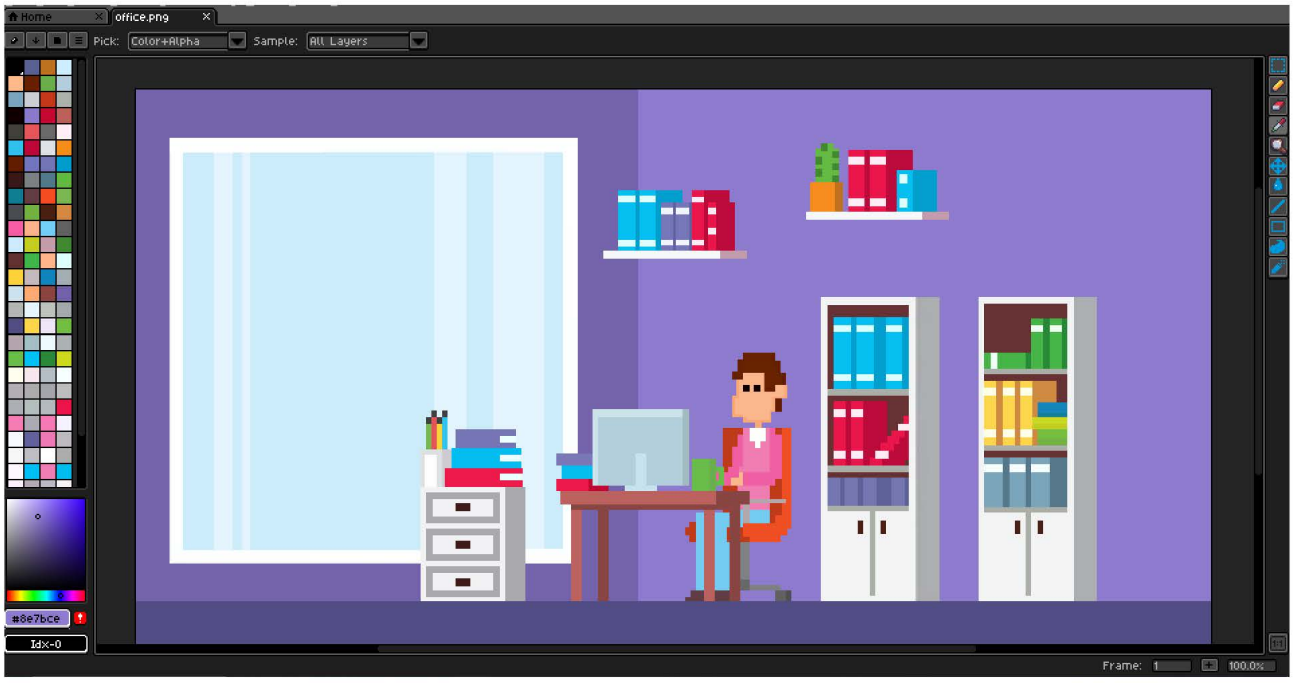

Рисунок 3.1 – Базові елементи фону

Паралельно зі створенням ігрових спрайтів, розроблялися візуальні складові користувацького інтерфейсу, зокрема кнопки. Кнопки є ключовими елементами взаємодії у симуляторі, тому їхній дизайн мав бути не тільки естетичним, але й функціональним. Кожна кнопка, наприклад, «START», розроблялася як набір спрайтів для різних станів: нормальний та при наведенні курсору. Це забезпечує візуальний зворотний зв'язок для гравця, підтверджуючи його дії. Дизайн кнопок також виконувався у піксельному стилі, щоб гармонійно вписуватися в загальну естетику гри (рис. 3.2).

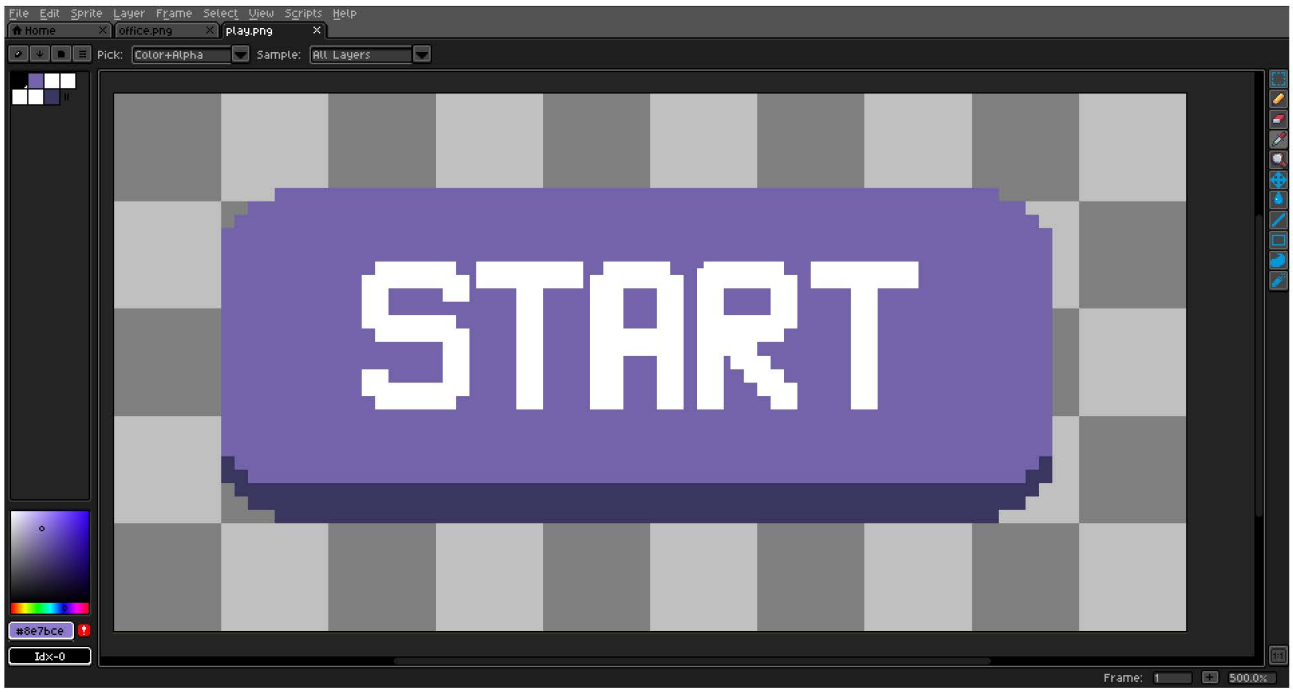

Рисунок 3.2 – Приклад однієї з кнопок

Після створення всіх графічних ресурсів у LibreSprite, вони експортувалися у форматі PNG та імпортувалися до Unity. В Unity кожен імпортований файл налаштовувався як Sprite (2D and UI) з відповідними параметрами Pixels Per Unit (що визначає розмір спрайта у світових одиницях Unity) та Filter Mode, який встановлювався на Point (no filter) для збереження ясності пікселів та запобігання їх розмиванню. Таким чином, етап створення спрайтів та кнопок є фундаментальним для формування візуального вигляду гри та її інтерактивної складової.

3.3 Інструменти розробки

У процесі реалізації програмного продукту, крім основних інструментів — ігрового рушія Unity, графічного редактора LibreSprite та середовища програмування Visual Studio Code — використовувались також допоміжні інструменти, що сприяли організації робочого процесу, спрощенню створення контенту, тестуванню та налагодженню.

Для втілення цього проєкту було використано низку програмних засобів, кожний з яких виконував свою конкретну роль у процесі розробки, забезпечуючи результативність та високу якість кінцевого виробу. Вибір цих інструментів був обґрунтований у попередньому розділі, а тут ми деталізуємо їх практичне використання.

Основним середовищем для розробки та інтеграції всіх складових гри виступив Unity Editor. Це потужна інтегрована платформа, котра надала всі потрібні засоби для побудови ігрових сцен, управління ігровими об'єктами, налаштування компонентів та візуальної розробки інтерфейсу. В Unity Editor здійснювалося побудова сцен, управління ієрархією, налаштування компонентів, управління асетами та тестування.

Побудова сцен створювалась та організовувалась у вигляді двох ігрових сцен, таких як Menu та основна ігрова сцена — StartFirm. Це включало в собі розташування фонових зображень, UI-елементів, ігрових об’єктів та їхніх компонентів. Кожна сцена є окремим файлом у проєкті в Unity, що дозволяє працювати над ними незалежно.

Керування ієрархіє є критично важливим процесом для організації застосунку та швидкого доступу до його елементів. У вікні Hierarchy всі ігрові об’єкти організовані у деревовидну структуру, як це показано на рис. 3.3.

Рисунок 3.3 – Деревовидна структура Menu

Ключові принципи керування ієрархією включали у себе батьківсько-дочірні зв'язки, використання порожніх GameObject-контейнерів та логічне сортування. UI-елементи, такі як кнопки та текстові поля, були зроблені дочірніми до відповідних панелей. Наприклад, RestartButton і QuitButton є дочірніми до GameOverPanel та WinPanel. Це дозволяє легко активувати або деактивувати цілі групи елементів одночасно. (рис. 3.4)

Рисунок 3.4 – Приклад дочірніх елементів

Для групування логічно пов'язаних елементів, які не мають власного візуального представлення, створювалися порожні GameObject, такі як TaskPanel і ShopPanel. Це значно покращує читабельність ієрархії та спрощує управління великою кількістю об'єктів. Об'єкти у вікні Hierarchy сортувалися таким чином, щоб найбільш важливі менеджери (GameManager, UIManager, AudioController) знаходилися на верхньому рівні, а UI-панелі та їхні дочірні елементи були організовані за функціональним призначенням.

Після створення GameObject та додавання до них скриптів, відбувалося їхнє налаштування через вікно Inspector. Це включало призначення посилань для публічних змінних у скриптах шляхом перетягування відповідних GameObject або їхніх компонентів, встановлення параметрів для вбудованих компонентів Unity, таких як Rect Transform, Slider, Button та конфігурацію публічних змінних у скриптах задля налаштування ігрового балансу.

Управління асетами, що включало імпорт графічних та звукових файлів, їхнє налаштування та організацію у вікні Project у логічну структуру папок, наприклад, Assets/Scripts, Assets/Sprites, Assets/Audio, Assets/Prefabs, Assets/Scenes, також виконувалося в Unity Editor. Важливим аспектом було створення префабів для багаторазово застосовуваних ігрових об'єктів, таких як ShopItemUI, що дало змогу створити шаблони об'єктів з усіма їхніми компонентами та налаштуваннями, а потім багаторазово використовувати їх у різних сценах, значно прискорюючи розробку та спрощуючи підтримку. Тестування в режимі Play Mode безпосередньо в Unity Editor давало змогу оперативно перевіряти зміни та виявляти візуальні та логічні помилки в реальному часі.

Для написання програмного коду мовою C# застосовувалася Microsoft Visual Studio. Це інтегроване середовище розробки є стандартним для Unity і надає розширені функції для кодування, рефакторингу та, що найважливіше, налагодження. Інтеграція Visual Studio з Unity дозволяє писати та редагувати скрипти зі зручним редактором коду, а також відлагоджувати код, встановлюючи breakpoints у C# скриптах для покрокового виконання, перегляду та зміни

значень змінних в реальному часі під час роботи гри в Unity Editor. Це є невід'ємним інструментом для виявлення логічних помилок та розуміння поведінки програми.

Для створення всієї піксельної графіки застосовувався LibreSprite. Цей спеціалізований редактор дозволив ефективно працювати з піксель-артом, створюючи анімаційні кадри, шари та палітри, що ідеально відповідало обраній візуальній стилістиці проєкту.

Отже, комбінація Unity Editor як інтегрованої платформи, Visual Studio як середовища для кодування та налагодження, LibreSprite для графіки забезпечила повний цикл розробки та ефективне вирішення поставлених завдань.

3.4 Розробка проєкту

Розробка симулятора офісного працівника відбувалася послідовно, крок за кроком, з безперервним тестуванням та інтеграцією нових функціональних блоків. На початковому етапі було створено базову структуру проєкту в Unity. Це включало створення двох головних сцен: «Menu» для головного меню та «StartFirm» для безпосередньо ігрового процесу. Для кожної сцени було налаштовано основний Canvas, який слугує контейнером для всіх елементів інтерфейсу.

Головне меню забезпечує інтуїтивно зрозумілу навігацію до основного ігрового режиму, налаштувань та інформаційних блоків, створюючи перше враження від продукту та визначаючи зручність подальшого ігрового досвіду.

Перед початком безпосередньої розробки були сформульовані ключові функціональні вимоги до сцени «Menu». Це включало потребу надання можливості переходу до початку нової ігрової сесії, доступу до вікна налаштувань (включаючи регулювання гучності музики та звукових ефектів, а також перемикання повноекранного режиму), а також функції виходу з програми. Особливу увагу було приділено візуальній привабливості та інтерактивності елементів меню, що передбачало зміну їх стану при наведенні

курсору та натисканні для кращого зворотного зв'язку з гравцем. Важливою умовою також стала масштабованість архітектури сцени, яка дозволила б легко інтегрувати нові опції або додаткові розділи в майбутньому.

Проектування архітектури сцени «Menu» було здійснено з урахуванням модульного підходу, що забезпечило гнучкість у розробці та подальшому вдосконаленні. Основними компонентами сцени стали графічні елементи, що включали фонове зображення та стилізований логотип гри, що відповідають її загальній тематиці. Інтерактивна складова була представлена набором кнопок, кожна з яких асоціювалася з певною функцією. Наприклад, кнопка «Грати» здійснює завантаження основної ігрової сцени "Game", тоді як кнопка «Налаштування» відкриває відповідне діалогове вікно. Для кожної кнопки було детально продумано її візуальні стани: стандартний, активний (при наведенні курсору). Взаємодія з цими елементами реалізована через систему обробки подій, яка викликає відповідні методи класу GameManager або AudioController при кліку, такі як RestartGame(), QuitGame() або SetMusicVolume(). Крім того, в сцені реалізовано відтворення фонової музики меню, яка автоматично запускається при завантаженні сцени «MainMenu» та зупиняється при переході на інші сцени.

Безпосередня реалізація сцени «Menu» була виконана в середовищі Unity з використанням мови програмування C#. Процес включав створення ієрархії об'єктів для UI елементів, таких як Canvas, панелі, кнопки та текстові поля. Кожен інтерактивний елемент інтерфейсу (кнопка, повзунок гучності, перемикач повноекранного режиму) був пов'язаний з відповідними методами в сценаріях GameManager та AudioController. Наприклад, для регулювання гучності музики та звукових ефектів були використані повзунки musicVolumeSlider та sfxVolumeSlider, які безпосередньо взаємодіють з функціями SetMusicVolume та SetSFXVolume класу AudioController, маніпулюючи параметрами AudioMixer. Перемикання повноекранного режиму реалізовано за допомогою fullscreenToggle, який викликає метод SetFullscreen. Запуск ігрового процесу відбувається через метод LoadScene класу SceneManager при натисканні

відповідної кнопки. Окрім того, було інтегровано звуковий ефект натискання кнопок, що збільшує тактильний відгук інтерфейсу. Такий підхід до реалізації забезпечив не тільки функціональність, але й високу ступінь інтерактивності та зручності для кінцевого користувача, що є важливим для ігрових додатків.

Отже, підсумуємо, у сцені «Menu» було розміщено фонове зображення, що відповідає візуальному стилю гри, а також три ключові кнопки «START», «SETTINGS» та «EXIT». Ці кнопки були візуально оформлені у піксельному стилі, а їхня логіка прив'язана до методів у скрипті GameManager для переходу між сценами або виходу з гри (рис. 3.5).

Рисунок 3.5 – Сцена «Menu»

Окрема панель «OptionsMenu» була створена для налаштувань, що містить повзунки для регулювання гучності музики та звукових ефектів, а також перемикач повноекранного режиму (рис. 3.6). Усі ці елементи були програмно пов'язані з класом AudioController, який реалізовано як Singleton для зручного доступу та збереження налаштувань між сесіями за допомогою PlayerPrefs.

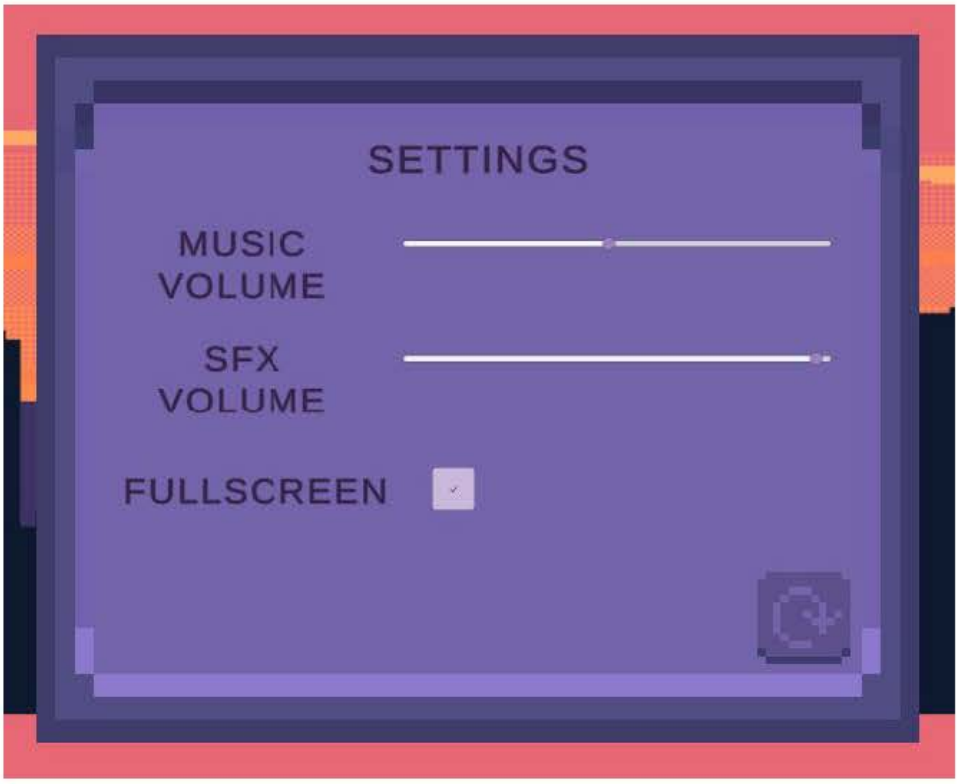

Рисунок 3.6 – Налаштування

Після налаштування головного меню, головна увага була зосереджена на ігровій сцені «StartFirm». Тут було розташовано фонове зображення офісного приміщення, що створює атмосферу симулятора. На Canvas ігрової сцени було створено основний HUD (Head-Up Display), що містить текстові поля для відображення грошей (Money Text) та репутації (Reputation Text), котрі керуються класом UIManager (рис. 3.7). Також на HUD була розміщена кнопка «SHOP», котра активує панель магазину.

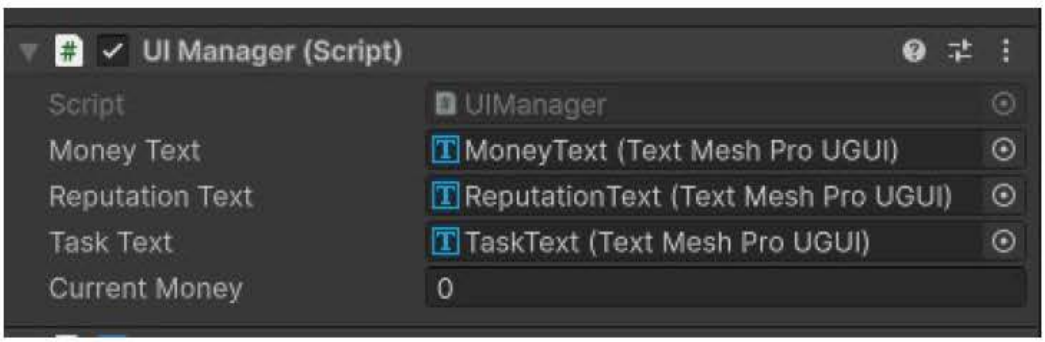

Рисунок 3.7 – Властивості скрипта UIManager

Центральним складником ігрової сцени є панель завдань (TaskPanel). Ця панель містить текстове поле для відображення опису поточного завдання (CurrentTask), слайдер (TimerSlider), що візуалізує час на виконання, та дві кнопки «Complete Button» та «Skip Button». Логіка генерації завдань, відліку часу та обробки дій гравця була реалізована у скрипті GameManager. Завдання різної складності (легкі, середні, складні) зберігаються у масивах рядків, з яких GameManager випадковим чином вибирає нове завдання. Таймер динамічно оновлюється у методі Update() GameManager і відображається на TimerSlider. При натисканні кнопок «Complete» або «Skip», GameManager обробляє відповідні дії, нараховуючи або списуючи гроші та репутацію через UIManager, а також оновлюючи лічильник пропущених завдань (рис. 3.8).

Рисунок 3.8 – Основний геймплей проєкту

Далі було розроблено внутрішньоігровий магазин (ShopPanel). Ця панель активується з ігрової сцени та дає змогу гравцеві купувати покращення. Магазин містить заголовок, кнопку закриття та динамічний список елементів покращень. Кожен елемент магазину є окремим префабом ShopItemUI, який містить назву покращення, опис, вартість та кнопку «Купити». Дані про покращення зберігаються у класі UpgradeData, який є простим класом даних. GameManager

ініціалізує список UpgradeData у методі InitializeUpgrades() та динамічно заповнює елементи ShopItemUI у магазині, передаючи їм відповідні дані. Логіка купівлі (PurchaseUpgrade()) та застосування ефектів покращень (ApplyUpgradeEffect()) також реалізована у GameManager. Наприклад, покращення "Speed Boost" збільшує час на виконання завдань, «Networking Skills» збільшує винагороду, а «Smart Investments» додає пасивний дохід (рис. 3.9).

Рисунок 3.9 – Внутрішньоігровий магазин

Паралельно з основними механіками, впроваджувалися системи завершення гри. Було розроблено окрему UI-панель GameOverPanel, яка активується у разі поразки гравця. Логіка активації цієї панелі реалізована у GameManager: GameOverPanel з'являється, коли кількість пропущених завдань (missedTasksCount) досягає або перевищує maxMissedTasks. При активації GameOverPanel ігровий процес зупиняється (Time.timeScale = 0f;), а всі активні корутини (блоки коду, які виконуються асинхронно) завершують свою роботу (StopAllCoroutines()). Панель містить кнопки "RestartButton" та "QuitButton", які програмно прив'язані до методів RestartGame() (перезавантаження поточної

сцени) та QuitGame() (вихід з програми) у GameManager, надаючи гравцеві можливість повторити спробу або вийти (рис 3.10).

Рисунок 3.10 – Екран програшу

Аналогічно, для сценарію перемоги було створено окрему UI-панель WinPanel. Вона активується, коли репутація гравця (GetCurrentReputation()) сягає заданого значення maxReputationForWin у GameManager. При активації WinPanel ігровий процес також зупиняється (Time.timeScale = 0f;) та всі корутини завершуються. Як і GameOverPanel, панель перемоги містить ідентичні кнопки "RestartButton" та «QuitButton» з відповідними функціями RestartGame() та QuitGame(), що забезпечує єдиний механізм управління завершенням гри (рис. 3.11).

Протягом всього процесу розробки відбувалася інтеграція візуального контенту. Спрайти, зроблені у LibreSprite, імпортувалися в Unity, налаштовувалися та застосовувалися для створення ігрових об'єктів та UI-елементів. Фонова музика для меню та ігрової сцени, а також звукові ефекти для кнопок та ігрових подій, були інтегровані з відкритих джерел. AudioController керує їхнім відтворенням, перемиканням та регулюванням гучності через UI-налаштування. Усі ці елементи були інтегровані в єдину, цілісну програмну

структуру, що забезпечує функціональність та, що важливо, можливість подальшого масштабування та розширення в майбутньому.

Рисунок 3.11 – Екран перемоги

Розроблена аудіосистема була сконструйована з урахуванням забезпечення динамічного контролю звуковим супроводом гри, включаючи фонову музику для різних сцен та звукові ефекти для інтерактивних частин. Головним компонентом цієї системи є один екземпляр класу AudioController, реалізований як Singleton, що гарантує його унікальність та збереження стану між сценами завдяки використанню DontDestroyOnLoad(gameObject), як це показано на рис. 3.12. Це рішення запобігає дублюванню аудіоконтролера при перезавантаженні сцен та забезпечує безперервне відтворення музики та збереження налаштувань гучності.

```
Unity Message | 0 references
void Awake()
{
    DontDestroyOnLoad(gameObject);

    AudioController[] controllers = FindObjectsByType<AudioController>(FindObjectsSortMode.None);
    if (controllers.Length > 1)
    {
        Destroy(gameObject);
    }
}
```

Рисунок 3.12 – Метод DontDestroyOnLoad

Для реалізації контролю гучності використовується компонент AudioMixer (mainMixer), що дозволяє групувати аудіоканали та керувати ними централізовано. Це забезпечує гнучкість у налаштуванні звукового балансу та можливість застосування складних ефектів до різних груп звуків.

При старті додатку (метод Awake) AudioController ініціалізується як єдиний екземпляр, що запобігає створенню дублікатів. У методі Start відбувається завантаження збережених налаштувань гучності та повноекранного режиму з PlayerPrefs.

Також на цьому етапі підписуються обробники подій onValueChanged для повзунків гучності та перемикача повноекранного режиму, що дозволяє динамічно оновлювати параметри звуку та відображення при зміні користувачем налаштувань (3.13).

```
Unity Message | 0 references
void Start()
{
    if (mainMixer == null)
    {
        Debug.LogError("AudioMixer is not assigned to AudioController");
    }

    LoadSettings();

    if (musicVolumeSlider != null)
        musicVolumeSlider.onValueChanged.AddListener(SetMusicVolume);
    if (sfxVolumeSlider != null)
        sfxVolumeSlider.onValueChanged.AddListener(SetSFXVolume);

    if (fullscreenToggle != null)
        fullscreenToggle.onValueChanged.AddListener(SetFullscreen);

    SceneManager.sceneLoaded += OnSceneLoaded;
    OnSceneLoaded(SceneManager.GetActiveScene(), LoadSceneMode.Single);
}
```

Рисунок 3.13 – Обробник подій onValueChanged

Керування гучністю музики та звукових ефектів здійснюється методами SetMusicVolume та SetSFXVolume відповідно. Ці методи встановлюють значення параметрів MusicVolume та SFXVolume у AudioMixer відповідно до логарифмічної шкали, що забезпечує більш природне сприйняття гучності для людського вуха. Збереження цих налаштувань відбувається за допомогою PlayerPrefs.SetFloat() з використанням констант MUSIC_VOLUME_KEY та SFX_VOLUME_KEY. Аналогічно, повноекранний режим контролюється методом SetFullscreen(bool isFullscreen), який оновлює властивість Screen.fullScreen та зберігає стан у PlayerPrefs (3.14).

```
2 references
public void SetMusicVolume(float volume)
{
    mainMixer.SetFloat("MusicVolume", volume == 0 ? -80f : Mathf.Log10(volume) * 20);
    PlayerPrefs.SetFloat(MUSIC_VOLUME_KEY, volume);
}

2 references
public void SetSFXVolume(float volume)
{
    mainMixer.SetFloat("SFXVolume", volume == 0 ? -80f : Mathf.Log10(volume) * 20);
    PlayerPrefs.SetFloat(SFX_VOLUME_KEY, volume);
}
```

Рисунок 3.14 – Методи SetMusicVolume та SetSFXVolume

3.5 Висновок до третього розділу

У цьому розділі було детально розглянуто всі аспекти практичної реалізації ігрового застосунку, що є симулятором офісного працівника. Проаналізована архітектура програмного забезпечення, що базується на компонентному підході Unity, дозволила створити гнучку та модульну систему. Описано ключові класи – GameManager, UIManager, UpgradeData, ShopItemUI та AudioController – та їх взаємодію, що демонструє чітке розділення відповідальності та ефективну координацію між компонентами. Застосування патерну Singleton для AudioController забезпечило централізований доступ до управління звуком, що є важливим для ігрових додатків.

Описано послідовний процес розробки проекту, починаючи від налаштування сцен та базового UI, і закінчуючи імплементацією ключових ігрових механік: динамічної системи завдань з різними рівнями складності та таймером, комплексної економічної моделі з грошима та репутацією, а також інтегрованого магазину покращень, що впливають на ігровий процес. Деталізовано логіку завершення гри, що включає сценарії перемоги та поразки, з відповідною активацією UI-панелей. Описано впровадження допоміжних систем, таких як збереження прогресу за допомогою PlayerPrefs, інтеграція візуального контенту (спрайти, 2D-анімації) та звукового супроводу, отриманого з відкритих джерел. Отримані результати є основою для подальшого розвитку гри: додавання нових механік і цілей, ускладнення геймплею та логіки гри, системи збереження прогресу тощо.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було розроблено функціональний прототип 2D-гри в жанрі симулятора офісного працівника, що дозволило повністю реалізувати поставлену мету дослідження. У межах роботи розглянуто актуальні підходи до створення навчальних ігрових додатків, проаналізовано технічні можливості сучасних інструментів, здійснено проєктування основних компонентів програмного продукту та реалізовано його ключові функції. Розробка проводилась із дотриманням методологічних принципів програмної інженерії, що передбачають послідовне планування, модульну побудову системи, тестування логіки та адаптацію інтерфейсу до потреб користувача.

У межах роботи також було проведено оцінку графічних редакторів, які придатні для створення ресурсів 2D-ігри. LibreSprite було обрано як оптимальний варіант через його безкоштовність, відкриту ліцензію, підтримку шарів, анімацій і зручну інтеграцію зі структурою Unity. Попри наявність більш складних або комерційних інструментів, таких як Aseprite або Adobe Photoshop, LibreSprite виявився найдоцільнішим у контексті навчального проєкту, оскільки дозволяє швидко створювати та експортувати графічні елементи у придатному форматі. Його використання забезпечило зменшення навантаження на технічну частину процесу розробки й дозволило сконцентруватися на логіці гри.

Розроблений програмний продукт є цілісним, функціональним, адаптованим для користувача і відповідає навчальним та технічним критеріям. Його структура дозволяє не лише використовувати його як завершений застосунок, але й застосовувати як шаблон або основу для подальших проєктів. Практична цінність роботи полягає в тому, що гра створена без використання комерційних інструментів, із мінімальними ресурсними витратами, за короткий проміжок часу, а її функціонал охоплює ключові аспекти розробки програмного забезпечення.

У ході реалізації були виявлені як сильні сторони проєкту, так і певні обмеження. До позитивних результатів належать стабільність функціонування гри, простота її модифікації, адаптивність інтерфейсу та зручність користування. Обмеженням є обмежена складність геймплею, відсутність довготривалої стратегії, обмежена кількість завдань і шаблонна логіка гри. Отримані результати можуть бути використані для подальшого розвитку гри: додавання нових механік і цілей, ускладнення геймплею та логіки гри, системи збереження прогресу тощо. Крім того, проєкт може бути корисним як наприклад у навчальному процесі або як основа для комерційного застосунку у сфері ігрової розробки.

Отримані результати можуть слугувати основою для подальшої роботи. У перспективі можливим є розширення функціоналу за рахунок введення сюжетних ліній, складніших алгоритмів генерації завдань, побудови сценаріїв із часовими обмеженнями, додавання ігрової статистики, рейтингової системи, а також адаптація гри для інших платформ, зокрема мобільних пристроїв. Усі ці напрямки відкривають можливості для подальшої дослідницької або комерційної реалізації і підтверджують потенціал виконаної роботи як повноцінного прикладного програмного проєкту.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Newzoo. Global Games Market Update Q1 2025 [Електронний ресурс] Режим доступу: https://newzoo.com/resources/blog/global-games-market-update-q1-2025

2. Unity Technologies. Unity Manual [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/Manual/index.html

3. Lobel A., Engels R., Stone L. Video games as tools for learning. – Computers in Human Behavior, 2020.

4. Fullerton T. Game Design Workshop: A Playcentric Approach to Creating Innovative Games. – CRC Press, 2018.

5. LibreSprite. Офіційний сайт [Електронний ресурс]. – Режим доступу: https://libresprite.github.io

6. Aseprite. Офіційний сайт [Електронний ресурс]. – Режим доступу: https://www.aseprite.org

7. Adobe Photoshop. Офіційний сайт [Електронний ресурс]. – Режим доступу: https://www.adobe.com/products/photoshop.html

8. Godot Engine. Офіційний сайт [Електронний ресурс]. – Режим доступу: https://godotengine.org

9. Unity Learn. Освітня платформа [Електронний ресурс]. – Режим доступу: https://learn.unity.com

10. GameDev.net. Інформаційний ресурс для розробників ігор [Електронний ресурс]. – Режим доступу: https://www.gamedev.net

11. Global Game Jam. Офіційний сайт [Електронний ресурс]. – Режим доступу: https://globalgamejam.org

12. Indie Game Developer Network (IGDN). Офіційний сайт [Електронний ресурс]. – Режим доступу: https://www.igdnonline.com

13. Schell J. The Art of Game Design: A Book of Lenses. – CRC Press, 2019.

14. Алан Торн. Опанування Unity Scripting, 2015. 380 с.

15. Джеремі Гібсон Бонд. Unity і C#. Геймдев від ідеї до реалізації, третє видання, 2022. 944 с.

ДОДАТОК А

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

public class GameManager : MonoBehaviour
{
    public UIManager uiManager;

    [Header("UI Elements")]
    public GameObject taskPanel;
    public TextMeshProUGUI currentTaskDescriptionText;
    public Slider timerSlider;
    public Button completeButton;
    public Button skipButton;
    public GameObject gameOverPanel;
    public GameObject winPanel;
    public int maxReputationForWin = 100;

    [Header("Shop UI Elements")]
    public GameObject shopPanel;
    public Button openShopButton;
    public Button closeShopButton;
    public ShopItemUI[] shopItemUIs;

    [Header("Game Settings")]
    public string[] easyTasks;
    public string[] mediumTasks;
    public string[] hardTasks;
    public float easyTaskTime = 10f;
    public float mediumTaskTime = 7f;
    public float hardTaskTime = 5f;
    public int rewardMoney = 10;
    public int rewardReputation = 5;
    public int penaltyMoney = 5;
    public int maxMissedTasks = 3;

    [Header("Timing Settings")]
    public float initialTaskDelay = 3f;
    public float taskCooldownDuration = 2f;

    public List<UpgradeData> availableUpgrades = new List<UpgradeData>();

    private int missedTasksCount = 0;
    private float currentTaskTimer;
    private bool taskActive = false;
    private string currentTaskString;

    private float bonusTaskTime = 0f;
    private int bonusMoneyPerTask = 0;
    private int bonusReputationPerTask = 0;
```

```
private int reducedPenaltyMoney = 0;

private bool smartInvestmentsPurchased = false;
private int passiveIncomeAmount = 0;
private int tasksSinceLastPassiveIncome = 0;
public int tasksPerPassiveIncome = 3;

private bool marketingCampaignActive = false;
private float marketingReputationMultiplier = 1f;
private float marketingDuration = 0f;
private float marketingTimer = 0f;


void Start()
{
   if (taskPanel != null) taskPanel.SetActive(false);
   if (gameOverPanel != null) gameOverPanel.SetActive(false);
   if (winPanel != null) winPanel.SetActive(false);
   if (shopPanel != null) shopPanel.SetActive(false);

   if (completeButton != null) completeButton.onClick.AddListener(CompleteTask);
   if (skipButton != null) skipButton.onClick.AddListener(SkipTask);

   if (openShopButton != null) openShopButton.onClick.AddListener(OpenShop);
   if (closeShopButton != null) closeShopButton.onClick.AddListener(CloseShop);

   if (uiManager == null)
   {
      uiManager = FindAnyObjectByType<UIManager>();
      if (uiManager == null)
      {
         Debug.LogError("UIManager not found.");
      }
   }

   InitializeUpgrades();
   UpdateShopUI();

   StartCoroutine(StartGameDelay());
}

void Update()
{
   if (taskActive)
   {
      currentTaskTimer -= Time.deltaTime;
      if (timerSlider != null)
      {
         timerSlider.value = currentTaskTimer / (GetCurrentTaskMaxTime() + bonusTaskTime);
      }

      if (currentTaskTimer <= 0)
      {
         SkipTask();
      }
   }
```

```
        if (marketingCampaignActive)
        {
            marketingTimer -= Time.deltaTime;
            if (marketingTimer <= 0)
            {
                marketingCampaignActive = false;
                marketingReputationMultiplier = 1f;
                Debug.Log("Marketing Campaign ended.");
            }
        }
    }

    float GetCurrentTaskMaxTime()
    {
        float baseTime = easyTaskTime;
        if (currentTaskString != null)
        {
            if (currentTaskString.Contains("Easy")) baseTime = easyTaskTime;
            else if (currentTaskString.Contains("Medium")) baseTime = mediumTaskTime;
            else if (currentTaskString.Contains("Hard")) baseTime = hardTaskTime;
        }
        return baseTime;
    }

    void InitializeUpgrades()
    {
        availableUpgrades.Clear();
        availableUpgrades.Add(new UpgradeData("speed_boost", "Speed Boost", "Increases task time
by 2 seconds", 20, 2f));
        availableUpgrades.Add(new UpgradeData("networking_skills", "Networking Skills", "Increases
money and reputation earned from completed tasks", 30, 0f));
        availableUpgrades.Add(new UpgradeData("stress_management", "Stress Management",
"Allows for one more missed task before Game Over", 100, 1f));
        availableUpgrades.Add(new UpgradeData("smart_investments", "Smart Investments", "Gain a
small amount of passive income every few tasks", 200, 30f));
        availableUpgrades.Add(new UpgradeData("backup_system", "Backup System", "Reduces the
money penalty for skipping tasks", 120, 5f));
        availableUpgrades.Add(new UpgradeData("marketing_campaign", "Marketing Campaign",
"Boosts reputation gain significantly for a short period", 80, 2f));
    }

    void UpdateShopUI()
    {
        if (shopItemUIs == null || shopItemUIs.Length == 0)
        {
            Debug.LogWarning("ShopItemUIs empty.");
            return;
        }

        for (int i = 0; i < shopItemUIs.Length; i++)
        {
            if (shopItemUIs[i] == null)
            {
                Debug.LogWarning($"ShopItemUI at index is null.");
                continue;
            }
```

```
        if (i < availableUpgrades.Count)
        {
            shopItemUIs[i].gameObject.SetActive(true);
            shopItemUIs[i].Setup(availableUpgrades[i], this);
        }
        else
        {
            shopItemUIs[i].gameObject.SetActive(false);
        }
    }
}

public void OpenShop()
{
    if (shopPanel != null)
    {
        shopPanel.SetActive(true);
        UpdateShopUI();
        taskActive = false;
        if (taskPanel != null) taskPanel.SetActive(false);
        StopAllCoroutines();
    }
}

public void CloseShop()
{
    if (shopPanel != null)
    {
        shopPanel.SetActive(false);
        if (missedTasksCount < maxMissedTasks)
        {
            StartCoroutine(TaskCooldown());
        }
    }
}

public void PurchaseUpgrade(UpgradeData upgrade)
{
    if (uiManager == null)
    {
        Debug.LogError("UIManager is null.");
        return;
    }

    if (upgrade.CanBuy())
    {
        if (uiManager.currentMoney >= upgrade.cost)
        {
            uiManager.AddMoney(-upgrade.cost);
            upgrade.LevelUp();
            ApplyUpgradeEffect(upgrade);
            UpdateShopUI();

            Debug.Log("Purchased: " + upgrade.upgradeName + " for $" + upgrade.cost);
        }
        else
        {
```

```
            Debug.Log("Not enough money to buy " + upgrade.upgradeName + ". Need: $" +
(upgrade.cost - uiManager.currentMoney));
        }
    }
    else
    {
        Debug.Log(upgrade.upgradeName + " is already purchased or at max level.");
    }
}

void ApplyUpgradeEffect(UpgradeData upgrade)
{
    switch (upgrade.id)
    {
        case "speed_boost":
            bonusTaskTime += upgrade.effectValue;
            Debug.Log("Bonus task time increased to: " + bonusTaskTime);
            break;
        case "networking_skills":
            bonusMoneyPerTask += 5;
            bonusReputationPerTask += 3;
            Debug.Log("Money and Reputation bonuses applied.");
            break;
        case "stress_management":
            maxMissedTasks += (int)upgrade.effectValue;
            Debug.Log("Max missed tasks increased to: " + maxMissedTasks);
            break;
        case "smart_investments":
            smartInvestmentsPurchased = true;
            passiveIncomeAmount = (int)upgrade.effectValue;
            Debug.Log("Smart Investments purchased! Passive income of $" + passiveIncomeAmount
+ " every " + tasksPerPassiveIncome + " tasks.");
            break;
        case "backup_system":
            reducedPenaltyMoney += (int)upgrade.effectValue;
            if (penaltyMoney - reducedPenaltyMoney < 0)
            {
                penaltyMoney = 0;
            }
            else
            {
                penaltyMoney -= reducedPenaltyMoney;
            }
            Debug.Log("Penalty money reduced to: " + penaltyMoney);
            break;
        case "marketing_campaign":
            marketingCampaignActive = true;
            marketingReputationMultiplier = upgrade.effectValue;
            marketingDuration = 10f;
            marketingTimer = marketingDuration;
            Debug.Log("Marketing Campaign activated! Reputation x" +
marketingReputationMultiplier + " for " + marketingDuration + " seconds.");
            break;
    }
}

IEnumerator StartGameDelay()
```

```
{
    if (uiManager != null)
    {
        uiManager.SetTask("Waiting for first task...");
    }
    yield return new WaitForSeconds(initialTaskDelay);
    StartNewTaskProcess();
}

IEnumerator TaskCooldown()
{
    taskActive = false;
    if (taskPanel != null) taskPanel.SetActive(false);

    if (uiManager != null)
    {
        uiManager.SetTask("Awaiting next task...");
    }
    yield return new WaitForSeconds(taskCooldownDuration);
    StartNewTaskProcess();
}

void StartNewTaskProcess()
{
    if (missedTasksCount >= maxMissedTasks)
    {
        GameOver();
    }
    else
    {
        GenerateNewTask();
    }
}

void GenerateNewTask()
{
    if (taskPanel != null) taskPanel.SetActive(true);
    taskActive = true;

    string[] tasksToChooseFrom = null;
    float chosenTaskTime = 0f;

    List<int> availableDifficulties = new List<int>();
    if (easyTasks != null && easyTasks.Length > 0) availableDifficulties.Add(0);
    if (mediumTasks != null && mediumTasks.Length > 0) availableDifficulties.Add(1);
    if (hardTasks != null && hardTasks.Length > 0) availableDifficulties.Add(2);

    if (availableDifficulties.Count == 0)
    {
        currentTaskString = "No tasks defined.";
        chosenTaskTime = 10f;
        Debug.LogError(currentTaskString);
    }
    else
    {
        int randomDifficultyIndex = Random.Range(0, availableDifficulties.Count);
        int chosenDifficulty = availableDifficulties[randomDifficultyIndex];
```

```
            switch (chosenDifficulty)
            {
                case 0:
                    tasksToChooseFrom = easyTasks;
                    chosenTaskTime = easyTaskTime;
                    break;
                case 1:
                    tasksToChooseFrom = mediumTasks;
                    chosenTaskTime = mediumTaskTime;
                    break;
                case 2:
                    tasksToChooseFrom = hardTasks;
                    chosenTaskTime = hardTaskTime;
                    break;
            }

            if (tasksToChooseFrom != null && tasksToChooseFrom.Length > 0)
            {
                currentTaskString = tasksToChooseFrom[Random.Range(0,
tasksToChooseFrom.Length)];
            }
            else
            {
                currentTaskString = "Error: Task list empty.";
                chosenTaskTime = 10f;
                Debug.LogError(currentTaskString);
            }
        }

        if (currentTaskDescriptionText != null)
        {
            currentTaskDescriptionText.text = currentTaskString;
        }
        if (uiManager != null)
        {
            uiManager.SetTask(currentTaskString);
        }

        currentTaskTimer = chosenTaskTime + bonusTaskTime;
        if (timerSlider != null) timerSlider.value = 1f;
    }

    public void CompleteTask()
    {
        if (!taskActive) return;

        Debug.Log("Task completed: " + currentTaskString);
        if (uiManager != null)
        {
            uiManager.AddMoney(rewardMoney + bonusMoneyPerTask);
            uiManager.AddReputation(Mathf.RoundToInt(rewardReputation + bonusReputationPerTask
* marketingReputationMultiplier));
        }
        missedTasksCount = 0;

        if (uiManager != null && uiManager.GetCurrentReputation() >= maxReputationForWin)
```

```
        {
            WinGame();
            return;
        }

        if (smartInvestmentsPurchased)
        {
            tasksSinceLastPassiveIncome++;
            if (tasksSinceLastPassiveIncome >= tasksPerPassiveIncome)
            {
                uiManager.AddMoney(passiveIncomeAmount);
                Debug.Log("Passive income received: $" + passiveIncomeAmount);
                tasksSinceLastPassiveIncome = 0;
            }
        }

        StartCoroutine(TaskCooldown());
    }

    public void SkipTask()
    {
        if (!taskActive) return;

        Debug.Log("Task skipped: " + currentTaskString);
        if (uiManager != null)
        {
            uiManager.AddMoney(-penaltyMoney);
        }
        missedTasksCount++;

        if (smartInvestmentsPurchased)
        {
            tasksSinceLastPassiveIncome++;
            if (tasksSinceLastPassiveIncome >= tasksPerPassiveIncome)
            {
                uiManager.AddMoney(passiveIncomeAmount);
                Debug.Log("Passive income received: $" + passiveIncomeAmount);
                tasksSinceLastPassiveIncome = 0;
            }
        }

        StartCoroutine(TaskCooldown());
    }
    public void WinGame()
    {
        Debug.Log("WIN!");
        taskActive = false;
        if (taskPanel != null) taskPanel.SetActive(false);
        if (shopPanel != null) shopPanel.SetActive(false);
        if (gameOverPanel != null) gameOverPanel.SetActive(false);

        if (winPanel != null) winPanel.SetActive(true);
        else Debug.LogWarning("WinPanel is not assigned in GameManager.");

        Time.timeScale = 0f;

        AddListenersToWinButtons();
```

```
        StopAllCoroutines();
    }

    void AddListenersToWinButtons()
    {
        if (winPanel == null)
        {
            Debug.LogWarning("WinPanel is null, cannot add listeners to win buttons.");
            return;
        }

        Button restartBtn = null;
        Button quitBtn = null;

        Transform restartBtnTransform = winPanel.transform.Find("RestartButton");
        Transform quitBtnTransform = winPanel.transform.Find("QuitButton");

        if (restartBtnTransform != null) restartBtn = restartBtnTransform.GetComponent<Button>();
        else Debug.LogWarning("RestartButton not found on WinPanel. Ensure it's named
'RestartButton' and is a child of WinPanel.");

        if (quitBtnTransform != null) quitBtn = quitBtnTransform.GetComponent<Button>();
        else Debug.LogWarning("QuitButton not found on WinPanel. Ensure it's named 'QuitButton'
and is a child of WinPanel.");

        if (restartBtn != null) restartBtn.onClick.RemoveAllListeners();
        if (quitBtn != null) quitBtn.onClick.RemoveAllListeners();

        if (restartBtn != null)
        {
            restartBtn.onClick.AddListener(RestartGame);
        }
        if (quitBtn != null)
        {
            quitBtn.onClick.AddListener(QuitGame);
        }
    }
    void GameOver()
    {
        Debug.Log("GAME OVER!");
        taskActive = false;
        if (taskPanel != null) taskPanel.SetActive(false);
        if (gameOverPanel != null) gameOverPanel.SetActive(true);
        StopAllCoroutines();

        Button restartBtn = null;
        Button quitBtn = null;

        if (gameOverPanel != null)
        {
            Transform restartBtnTransform = gameOverPanel.transform.Find("RestartButton");
            Transform quitBtnTransform = gameOverPanel.transform.Find("QuitButton");

            if (restartBtnTransform != null)
            {
                restartBtn = restartBtnTransform.GetComponent<Button>();
            }
```

```
            else
            {
                Debug.LogWarning("RestartButton not found.");
            }
            if (quitBtnTransform != null)
            {
                quitBtn = quitBtnTransform.GetComponent<Button>();
            }
            else
            {
                Debug.LogWarning("QuitButton not found");
            }
        }
        else
        {
            Debug.LogWarning("GameOverPanel empty.");
        }

        if (restartBtn != null) restartBtn.onClick.RemoveAllListeners();
        if (quitBtn != null) quitBtn.onClick.RemoveAllListeners();

        if (restartBtn != null)
        {
            restartBtn.onClick.AddListener(RestartGame);
        }
        if (quitBtn != null)
        {
            quitBtn.onClick.AddListener(QuitGame);
        }
    }

    public void RestartGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    public void QuitGame()
    {
        Application.Quit();
        Debug.Log("Exiting game.");
    }
}
```

ДОДАТОК Б

```
using UnityEngine;
using TMPro;

public class UIManager : MonoBehaviour
{
    public TextMeshProUGUI moneyText;
    public TextMeshProUGUI reputationText;
    public TextMeshProUGUI taskText;

    public int currentMoney = 0;
    private int currentReputation = 0;
    private int maxReputation = 100;
    private string currentTask = "Starting the workday";

    void Start()
    {
        UpdateUI();
    }

    public void AddMoney(int amount)
    {
        currentMoney += amount;
        UpdateUI();
    }

    public void AddReputation(int amount)
    {
        currentReputation += amount;
        if (currentReputation > maxReputation)
        {
            currentReputation = maxReputation;
        }
        UpdateUI();
    }

    public void SetTask(string newTask)
    {
        currentTask = newTask;
        UpdateUI();
    }
    public int GetCurrentReputation()
    {
```

```
        return currentReputation;
    }

    void UpdateUI()
    {
        if (moneyText != null)
        {
            moneyText.text = "Money: $" + currentMoney.ToString();
        }
        else
        {
            Debug.LogWarning("Empty");
        }

        if (reputationText != null)
        {
            reputationText.text = "Reputation: " + currentReputation.ToString() +
"/" + maxReputation.ToString();
        }
        else
        {
            Debug.LogWarning("Empty");
        }

        if (taskText != null)
        {
            taskText.text = "Task: " + currentTask;
        }
        else
        {
            Debug.LogWarning("Empty");
        }
    }
}
```

ДОДАТОК В

```
using UnityEngine;
using TMPro;
using UnityEngine.UI;

public class ShopItemUI : MonoBehaviour
{
    public TextMeshProUGUI upgradeNameText;
    public TextMeshProUGUI descriptionText;
    public TextMeshProUGUI costText;
    public Button buyButton;
    public TextMeshProUGUI buyButtonText;

    private UpgradeData currentUpgrade;
    private GameManager gameManager;

    void Start()
    {
        if (buyButton != null)
        {
            buyButton.onClick.AddListener(OnBuyButtonClicked);
        }
    }

    public void Setup(UpgradeData upgrade, GameManager gm)
    {
        currentUpgrade = upgrade;
        gameManager = gm;

        if (upgradeNameText != null) upgradeNameText.text =
upgrade.upgradeName;
        if (descriptionText != null) descriptionText.text = upgrade.description;
        if (costText != null) costText.text = "$" + upgrade.cost.ToString();

        UpdateBuyButtonState();
    }

    void UpdateBuyButtonState()
    {
        if (buyButton != null)
        {
            if (currentUpgrade.isPurchased)
            {
```

```
            buyButton.interactable = false;
            if (buyButtonText != null) buyButtonText.text = "Purchased";
            if (costText != null) costText.text = "Owned";
        }
        else if (currentUpgrade.currentLevel >= currentUpgrade.maxLevel)
        {
            buyButton.interactable = false;
            if (buyButtonText != null) buyButtonText.text = "Max Level";
            if (costText != null) costText.text = "Max";
        }
        else
        {
            buyButton.interactable = true;
            if (buyButtonText != null) buyButtonText.text = "Buy";
        }
    }
}

void OnBuyButtonClicked()
{
    if (gameManager != null && currentUpgrade != null)
    {
        gameManager.PurchaseUpgrade(currentUpgrade);
    }
}
}
```

ДОДАТОК Д

```
using UnityEngine;

[System.Serializable]
public class UpgradeData
{
    public string id;
    public string upgradeName;
    public string description;
    public int cost;
    public bool isPurchased = false;
    public int currentLevel = 0;
    public int maxLevel = 1;
    public float effectValue;

    public UpgradeData(string id, string name, string desc, int cost, float effect,
int maxLvl = 1)
    {
        this.id = id;
        this.upgradeName = name;
        this.description = desc;
        this.cost = cost;
        this.effectValue = effect;
        this.maxLevel = maxLvl;
    }

    public bool CanBuy()
    {
        return !isPurchased || (currentLevel < maxLevel);
    }

    public void LevelUp()
    {
        if (currentLevel < maxLevel)
        {
            currentLevel++;
            if (currentLevel == maxLevel)
            {
                isPurchased = true;
            }
        }
    }
}
```

ДОДАТОК Е

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

public class AudioController : MonoBehaviour
{
    public AudioMixer mainMixer;

    public Slider musicVolumeSlider;
    public Slider sfxVolumeSlider;

    public Toggle fullscreenToggle;

    public AudioSource menuMusicSource;
    public AudioClip menuMusicClip;

    public AudioSource gameMusicSource;
    public AudioClip gameMusicClip;

    public AudioSource sfxSource;
    public AudioClip buttonClickSFX;

    private const string MUSIC_VOLUME_KEY = "MusicVolume";
    private const string SFX_VOLUME_KEY = "SFXVolume";
    private const string FULLSCREEN_KEY = "Fullscreen";

    void Awake()
    {
        DontDestroyOnLoad(gameObject);

        AudioController[] controllers = FindObjectsByType<AudioController>(FindObjectsSortMode.None);
```

```
    if (controllers.Length > 1)
    {
        Destroy(gameObject);
    }
}

void Start()
{
    if (mainMixer == null)
    {
        Debug.LogError("AudioMixer is not assigned to AudioController");
    }

    LoadSettings();

    if (musicVolumeSlider != null)
        musicVolumeSlider.onValueChanged.AddListener(SetMusicVolume);
    if (sfxVolumeSlider != null)
        sfxVolumeSlider.onValueChanged.AddListener(SetSFXVolume);

    if (fullscreenToggle != null)
        fullscreenToggle.onValueChanged.AddListener(SetFullscreen);

    SceneManager.sceneLoaded += OnSceneLoaded;
    OnSceneLoaded(SceneManager.GetActiveScene(), LoadSceneMode.Single);
}

void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    Debug.Log("Scene Loaded: " + scene.name);
    if (scene.name == "MainMenu")
    {
        if (menuMusicSource != null && menuMusicClip != null)
        {
            if (gameMusicSource != null && gameMusicSource.isPlaying) gameMusicSource.Stop();
```

```
            if (!menuMusicSource.isPlaying)
            {
                menuMusicSource.clip = menuMusicClip;
                menuMusicSource.loop = true;
                menuMusicSource.Play();
            }
        }
        else
        {
            Debug.LogWarning("Menu music is not assigned.");
        }
    }
    else if (scene.name == "Game")
    {
        if (gameMusicSource != null && gameMusicClip != null)
        {
            if (menuMusicSource != null && menuMusicSource.isPlaying) menuMusicSource.Stop();
            if (!gameMusicSource.isPlaying)
            {
                gameMusicSource.clip = gameMusicClip;
                gameMusicSource.loop = true;
                gameMusicSource.Play();
            }
        }
        else
        {
            Debug.LogWarning("Game music is not assigned.");
        }
    }
    else
    {
        if (menuMusicSource != null && menuMusicSource.isPlaying) menuMusicSource.Stop();
        if (gameMusicSource != null && gameMusicSource.isPlaying) gameMusicSource.Stop();
    }
}
```

```
    public void SetMusicVolume(float volume)
    {
        mainMixer.SetFloat("MusicVolume", volume == 0 ? -80f : Mathf.Log10(volume) * 20);
        PlayerPrefs.SetFloat(MUSIC_VOLUME_KEY, volume);
    }

    public void SetSFXVolume(float volume)
    {
        mainMixer.SetFloat("SFXVolume", volume == 0 ? -80f : Mathf.Log10(volume) * 20);
        PlayerPrefs.SetFloat(SFX_VOLUME_KEY, volume);
    }

    public void SetFullscreen(bool isFullscreen)
    {
        Screen.fullScreen = isFullscreen;
        PlayerPrefs.SetInt(FULLSCREEN_KEY, isFullscreen ? 1 : 0);
    }

    void LoadSettings()
    {
        float musicVol = PlayerPrefs.GetFloat(MUSIC_VOLUME_KEY, 1f);
        float sfxVol = PlayerPrefs.GetFloat(SFX_VOLUME_KEY, 1f);

        bool isFullscreen = PlayerPrefs.GetInt(FULLSCREEN_KEY, 1) == 1;

        if (musicVolumeSlider != null) { musicVolumeSlider.value = musicVol; SetMusicVolume(musicVol);
}
        if (sfxVolumeSlider != null) { sfxVolumeSlider.value = sfxVol; SetSFXVolume(sfxVol); }

        if (fullscreenToggle != null) { fullscreenToggle.isOn = isFullscreen; SetFullscreen(isFullscreen); }
        else { Screen.fullScreen = isFullscreen; }
    }

    public void PlaySFX(AudioClip clip)
```

```
    {
        if (sfxSource != null && clip != null)
        {
            sfxSource.PlayOneShot(clip);
        }
        else
        {
            if (sfxSource == null) Debug.LogWarning("SFX AudioSource is not assigned");
            if (clip == null) Debug.LogWarning("SFX AudioClip is null");
        }
    }

    public void PlayButtonClickSFX()
    {
        PlaySFX(buttonClickSFX);
    }

    void OnDestroy()
    {
        SceneManager.sceneLoaded -= OnSceneLoaded;
    }
}
```