ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка 3D відеогри "Whispers of the Wildwood" в жанрі виживання з елементами RPG. Спец-частина: розробка backend-частини системи»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Дмитро БАРБАК
(підпис)

Виконав: здобувач вищої освіти групи ПД-43

Дмитро БАРБАК

Керівник: Ірина ЗАМРІЙ
д-р техн. наук, проф.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Барбаку Дмитру Олександровичу

1. Тема кваліфікаційної роботи: «Розробка 3D відеогри "Whispers of the Wildwood" в жанрі виживання з елементами RPG. Спец-частина: розробка backend-частини системи»
керівник кваліфікаційної роботи д-р техн. наук, професор Ірина ЗАМРІЙ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про розробку backend-части відеоігор жанру виживання з елементами RPG, опис методів розробку backend-части відеоігор жанру виживання з елементами RPG, технічна документація з описом бібліотек для розробки backend-части відеоігор.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд та аналіз існуючих методів та технологій розробки backend-частини відеоігор жанру виживання з елементами RPG.
2. Проектування backend-частини відеогри жанру виживання з елементами RPG.

3. Програмна реалізація та опис функціонування backend-частини відеогри жанру виживання з елементами RPG.
4. Тестування відеогри.

5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Алгоритм роботи застосунку.
5. Діаграма прецедентів.
6. Діаграма класів.
7. Діаграма послідовності.
8. Демонстрація роботи застосунку.
9. Апробація результатів дослідження.

6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Огляд та аналіз існуючих методів та технологій розробки backend-частини відеоігор жанру виживання з елементами RPG.	07.03-15.03.2026	
4	Проектування backend-частини відеогри жанру виживання з елементами RPG.	15.03-30.03.2026	
5	Програмна реалізація та опис функціонування backend-частини відеогри жанру виживання з елементами RPG.	31.03-20.04.2026	
6	Тестування відеогри	21.04.-25.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	26.04-30.04.2026	
8	Розробка демонстраційних матеріалів	01.05-10.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач вищої освіти _____ Дмитро БАРБАК
(підпис)

Керівник
кваліфікаційної роботи _____ Ірина ЗАМРІЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 52 стор., 9 табл., 21 рис., 21 джерел.

Мета роботи – покращення ігрового досвіду користувача за рахунок поєднання геймплею жанру виживання з ключовими елементами RPG.

Об'єкт дослідження – процес набуття досвіду під час дослідження механік відеогри у жанрі виживання з елементами RPG.

Предмет дослідження – програмні засоби та методи створення геймплею відеогри у жанрі виживання з елементами RPG.

Короткий зміст роботи: В роботі було проаналізовано особливості жанру виживанна в поеданні з RPGта визначено цільову аудиторію. Проаналізовано найпопулярніші ігри жанру виживання: Minecraft. The Forest, Rust, ARK: Survival Evolved, Don't Starve. Розроблено алгоритм роботи відеогри та програмно реалізовані ключові механіки: базові механіки гравця, збору ресурсів, крафту, будівництва, взаємодії з неігровими персонажами, ворогів, збереження та подальшого завантаження ігрового процесу. Проведено функціональне та модульне тестування відеогри. В роботі використано мову програмування C#, ігровий рушій Unityта середовище розробки Visual Studio.

Сферою використання застосунку є розвага, відпочинок та задоволення під час проходження гри.

КЛЮЧОВІ СЛОВА: ВИЖИВАННЯ, РОЗРОБКА, RPG, C# UNITY, VISUAL STUDIO, АНАЛІЗ, МЕХАНІКА, КЛАС, МЕТОД

ЗМІСТ

ВСТУП

Актуальність теми – сьогодні ігрова індустрія переживає етап постійної трансформації, що робить її одним із найбільш прогресивних сегментів усього цифрового світу. Варто зазначити, що у гравців особливий запит викликають саме 3D-проєкти. У них виживання (survival) складно переплітається з RPG-механіками: тут присутні і відкриті локації, і глибока прокачка, і повністю інтерактивне оточення. Втім, успіх таких ігор опирається не тільки на візуальний ряд. Набагато важливішим є стійкий «фундамент» – та сама внутрішня архітектура, здатна безперебійно обробляти величезні масиви даних та тримати всі системи в робочому стані. Саме через це критичним етапом у створенні дійсно якісного ігрового продукту виступає розробка backend-частини.

Диктується актуальність даного дослідження насамперед практичною потребою: необхідно створити надійне логічне ядро для 3D-ігор у жанрі survival-RPG. На це ядро (backend) покладаються ключові завдання. Від елементарного менеджменту інвентарю чи збереження прогресу – аж до надзвичайно складної взаємодії об’єктів безпосередньо в кадрі. Технічні вимоги зростають постійно. Тому стає цілком очевидним факт, що для реалізації подібного функціоналу необхідно застосовувати сучасні інструменти (зокрема рушій Unity та мову C#) і максимально гнучкі підходи до проектування структури програми.

*Мета робо*ти – покращення ігрового досвіду користувача за рахунок поєднання геймплею жанру виживання з ключовими елементами RPG.

Об’єкт дослідження – процес набуття досвіду під час дослідження механік відеогри у жанрі виживання з елементами RPG.

Предмет дослідження – програмні засоби та методи створення геймплею відеогри у жанрі виживання з елементами RPG.

Методи дослідження – для того, щоб реалізувати поставлені завдання, ми провели аналіз ринку відеоігор у жанрі survival-RPG. Також були детально вивчені вподобання цільової аудиторії. Шляхом порівняння найбільш успішних проєктів-

аналогів нам вдалося виділити кращі механіки. Це, своєю чергою, дозволило визначити оптимальний технологічний стек. Внаслідок проведеного аналізу основними засобами розробки було обрано мову програмування C#, середовище Visual Studio та платформу Unity.

Наукова новизна роботи – полягає вона насамперед в архітектурному поєднанні механік виживання та RPG-складової у межах єдиного, цілісного ігрового простору (який включає відкритий світ, квесту систему та інтерактивне середовище). У роботі продемонстровано суто комплексний підхід до розробки. Охоплено практично все: від базового керування головним персонажем до багатокомпонентних систем крафту, будівництва, штучного інтелекту ворогів і механізмів, що зберігають ігровий прогрес.

Практична значущість результатів – дає змогу користувачеві отримати унікальний ігровий досвід, а також справжнє задоволення від проходження гри.

1 АНАЛІЗ ЖАНРУ ВИЖИВАННЯ З ЕЛЕМЕНТАМИ RPG

1.1 Визначення жанру виживання з елементами RPG

На сьогоднішній день жанр survival-RPG впевнено утримує лідерські позиції на ринку відеоігор. І це цілком закономірно. Пояснюється така популярність досить просто: розробникам вдалося об'єднати відкритий світ і динамічний геймплей із глибокою системою розвитку персонажа. При цьому герою доводиться безперервно виживати у вкрай ворожому середовищі. Взагалі, базовим завданням у подібних проєктах завжди залишається збереження життя аватара. Доводиться постійно шукати матеріали, будувати якісь укриття. Адаптація до мінливих обставин стає ключовою вимогою.

Якщо ж порівнювати такі ігри зі звичайними симуляторами виживання, то саме RPG-елементи кардинально змінюють картину. Можливості гравця розширюються неймовірно. Банальне накопичення ресурсів відходить на другий план, поступаючись місцем повноцінній прогресії. На практиці це дає користувачу отримати можливість свідомо «прокачувати» свого героя, вивчати нові вміння та формувати абсолютно унікальний стиль гри.

Загалом, аудиторія цінує даний жанр одразу за кілька речей. Насамперед ідеться про свободу дій. Гравець сам вирішує, куди йому йти і які завдання виконувати першими. Але не менш важливу роль відіграє і постійна напруга. Самі механіки виживання змушують завжди бути напоготові: слідкувати за погодою, перевіряти рівень здоров’я та остерігатися ворогів. Це дає просто колосальне занурення у віртуальний світ. Більше того, саме наявність рольової складової мотивує проводити в грі десятки годин. Видимий результат розвитку утримує інтерес набагато краще за монотонний збір предметів.

Ризик під час мандрівок на нові території викликає в користувачів яскраві емоції. А через впровадження RPG-механік цей емоційний зв'язок стає ще

міцнішим так як гравець починає асоціювати себе з персонажем. Власні рішення тут мають цілком реальні наслідки.

Тому можна з упевненістю стверджувати, що survival-RPG виступає одним із найцікавіших напрямів у розробці відеоігор. Завдяки поєднанню дослідження та виживання з рольовою прокачкою, створюється по-справжньому унікальний досвід. І саме ця синергія дозволяє жанру залишатися актуальним, постійно еволюціонуючи.

1.2 Особливості поєднання RPG елементів з ігровим процесом жанру виживання

Аналізуючи сучасні тенденції геймдизайну, неважко помітити, що впровадження рольової складової в survival-ігри виявилося вкрай успішним кроком. Це відчутно поглиблює емоційний досвід від проходження. Якщо взяти до уваги класичні відеоігри цього напрямку (до яких можна віднести базові версії Minecraft або ж Don’t Starve), то там увага гравця була зосереджена переважно на підтриманні життєдіяльності та зборі припасів. Проте із додаванням RPG-механік ця парадигма кардинально змінюється. Вводиться повноцінна система прогресії. Як наслідок, замість постійного страху померти, ми отримуємо можливість планомірно розвивати свого персонажа.

Оця еволюція героя якраз і формує для користувача нову структуру цілей. У традиційних іграх жанру виживання ти зазвичай просто намагаєшся вижити будь-якою ціною. А от наявність рольових елементів змушує орієнтуватися на систематичне покращення показників. Навіть такі буденні речі, як видобуток матеріалів чи рядова битва, набувають зовсім іншої ваги. Вони приносять досвід, відкривають доступ до нових навичок та кращої зброї. Відповідно, бажання досліджувати віртуальний світ нікуди не зникає, а лише посилюється.

Окремої уваги заслуговує питання свободи вибору. Вона забезпечується саме завдяки рольовому відіграшу. Жорсткої лінійності тут немає. Користувач має змогу самостійно вирішувати, що йому ближче: зробити акцент на бойових вміннях, чи,

скажімо, зосередитися на будівництві, крафті та соціальній взаємодії. В якості прикладу можна навести такі масштабні проєкти як Rust та ARK: Survival Evolved. Завдяки високій варіативності тактик там створюється унікальний ігровий досвід для кожної людини. Враховуючи специфіку жанру, де треба регулярно підлаштовуватися під нові загрози, такий підхід повністю себе виправдовує.

Варто також згадати про емоційну прив'язаність до аватара, яка виникає через систему прокачування. Чим вищий рівень персонажа, тим більшою стає цінність зібраного ним інвентарю. Втрата всіх речей після невдалого бою (схожа механіка покарання часто зустрічається у тому ж Rust) викликає у гравців неабияку фрустрацію. Людина дуже гостро усвідомлює, скільки реальних годин було щойно втрачено разом із лутом.

Що ж до вивчення нових територій, то тут також відбуваються суттєві зміни. Експедиції перестають бути простою зміною локацій. Наприклад, у The Forest спуск у глибокі печери завжди супроводжується ризиком, але саме там сховані найважливіші для сюжету предмети та найсильніша зброя. Ризикувати стає цілком виправдано і вигідно. Таким чином, розвідка світу перетворюється на один із головних стимулів для проходження.

Підсумовуючи вищезазначене, популярність подібних гібридних проєктів багато в чому пояснюється тим, що аудиторія прагне отримати максимально наближений до реальності досвід. Гравцям завжди імпонувала можливість бачити наслідки своїх рішень. Коли ж суворі механіки виживання гармонійно поєднуються з можливістю розвивати героя і досліджувати невідомі землі, гра стає набагато цікавішою. Саме це і дозволяє утримувати інтерес геймерів на десятки годин.

1.3 Цільова аудиторія

Значною репрезентативністю відзначається спектр цільової аудиторії проєктів у даному жанрі. Зумовлено це насамперед тим, що в межах одного програмного продукту відбувається синтез кількох популярних ігрових механік. До пріоритетної групи споживачів входять користувачі, для яких критично важливими

є процес дослідження відкритих світів та високий ступінь ігрової свободи. Можливість безпосередньо впливати на вектор розвитку подій стає для них вирішальним фактором.

Найбільший інтерес у аудиторії викликає саме комплексна взаємодія систем життєзабезпечення, прогресії героя та механік накопичення ресурсів. Функціональні можливості гравця при цьому диверсифікуються поступово. У міру того, як освоюється ігрове середовище, відкриваються нові елементи геймплею. Високий рівень релевантності зазначений жанр демонструє переважно для молоді, яка є найбільш активною частиною ігрової спільноти. Саме ці користувачі висувають вимоги до якості ігрового процесу кінцевого продукту та ступеню імерсивності віртуальної атмосфери.

Для даної категорії геймерів розгалуженість систем розвитку та варіативність модернізації екіпірування є аспектами критичними. Повна інтерактивність довкілля – ще одна важлива вимога. Індивідуалізація досвіду забезпечується саме через інтеграцію RPG-компонентів. Процес проходження перестає бути лінійним. Він трансформується у персоналізований шлях технологічного та особистого зростання гравця.

1.4 Дослідження існуючих відеоігор жанру виживання

Сучасна індустрія відеоігор демонструє стійкий інтерес до жанру виживання, який трансформувався у складну екосистему механік. Survival-проєкти сьогодні – це не просто симулятори голоду, а багатошарові системи, що поєднують дослідження територій, менеджмент ресурсів та розгалужений крафт. Популярність напряму зумовлена високим рівнем реіграбельності та можливістю формування унікальних ігрових ситуацій, де гравець змушений постійно адаптуватися до мінливих умов.

Центральним елементом більшості аналогів виступає концепція «відкритого світу». Вона надає користувачеві суб'єктивну свободу, де стратегія виживання будується на основі особистого досвіду та взаємодії з агресивним середовищем.

1.4.1 Minecraft

Minecraft визначає канони жанру через призму повної персоналізації та творчої свободи. Геймплей побудований на циклічній активності: видобуток сировини – крафт – будівництво. Завдяки процедурній генерації ландшафтів (рис. 1.1) кожен сеанс гри пропонує унікальні умови, що фактично нівелює межі для дослідження.

Ключовим фактором успіху тут є воксельна архітектура світу, яка дозволяє гравцеві фізично змінювати ландшафт. Потреби персонажа (харчування, захист від мобів) інтегровані в процес творчості, що робить рутину виживання осмисленою. Наявність кооперативного режиму значно розширює соціальну взаємодію між гравцями , дозволяючи реалізовувати масштабні творчі проєкти.

Сильні сторони ігрового процесу:

– нескінченна варіативність світу завдяки процедурним алгоритмам генерації;

– глибока кастомізація оточення через будівництво;

– низький поріг входу при величезній глибині механік.

Слабкі сторони ігрового процесу:

– відсутність чітко вираженої сюжетної мети;

– спрощена бойова система, що не забезпечує складного виклику.

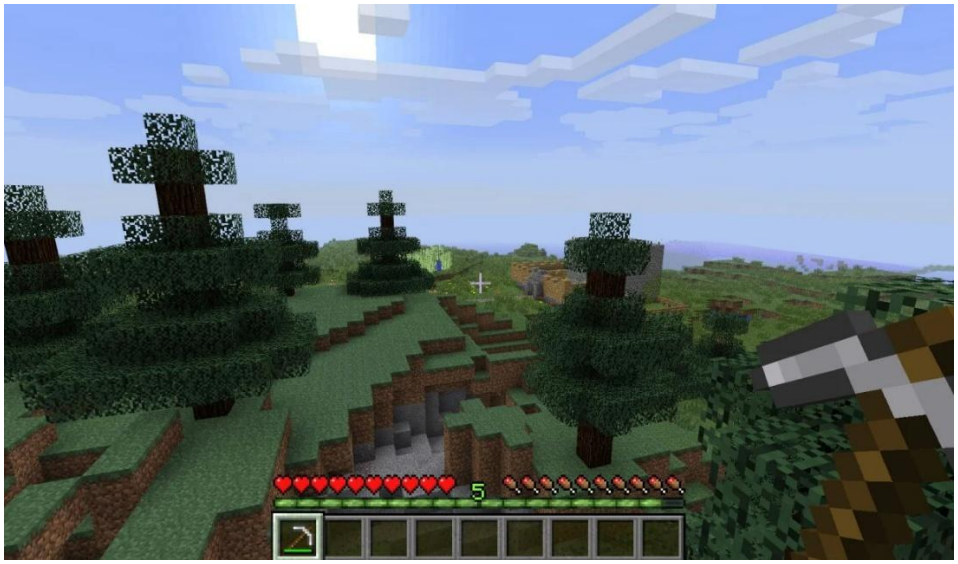


Рис. 1.1 Вигляд відеогри Minecraft

1.4.2 The Forest

У проєкті The Forest класичне виживання поєднується з елементами психологічного трилера. Основний акцент зміщено на взаємодію з ворожим оточенням – агресивними аборигенами, чий штучний інтелект змушує гравця діяти обережно. Екосистема острова відчувається живою та загрозливою, що створює постійний емоційний тиск.

Унікальною рисою гри є вертикальність геймплею через розгалужену систему печер. Це не лише локації для збору ресурсів, а й інструмент оповіді сюжету через оточення (рис. 1.2). Візуальна стилістика підкреслює ізоляцію персонажа, роблячи процес будівництва укріплень питанням виживання, а не лише естетики.

Сильні сторони ігрового процесу:

- густа атмосфера та якісна імплементація елементів горору;
- наявність логічно завершеної сюжетної лінії;
- складна поведінкова модель супротивників.

Слабкі сторони ігрового процесу:

- важка у розумінні система крафту що може заплутати гравця;
- висока складність для початківців через агресивність світу.

Рис. 1.2 Вигляд відеогри The Forest

1.4.3 Rust

Rust представляє сегмент жорсткого PvP-виживання, де основним джерелом загрози є не середовище, а інші учасники. Геймплей фокусується на територіальному домінуванні, рейд-механіках та постійній боротьбі за технологічну перевагу (рис. 1.3). Людський фактор робить кожну ігрову сесію непередбачуваною.

Тут критично важливим є перехід від примітивних знарядь до автоматичної вогнепальної зброї, що створює сильне відчуття прогресу. Виживання в Rust вимагає не лише навичок збору ресурсів, а й здатності до дипломатії або агресивного захисту власних територій.

Сильні сторони ігрового процесу:

- висока напруга та динаміка ігрового процесу;
- розгалужені системи будівництва фортифікацій;
- акцент на конкуренції та соціальній взаємодії.

Слабкі сторони ігрового процесу:

- високий рівень токсичності спільноти;
- критична залежність успіху від часу, проведеного онлайн.

Рис. 1.3 Вигляд відеогри Rust

1.4.4 ARK: Survival Evolved

Особливістю ARK є інтеграція доісторичної фауни в механіки виживання. Система приручення істот докорінно змінює ігровий темп: динозаври стають інструментами для видобутку, транспортування та ведення бойових дій (рис. 1.4). Рольова складова з прокачкою характеристик додає грі глибини, притаманної RPG.

Прогресія в ARK побудована на переході від кам'яного віку до футуристичних технологій, що забезпечує довгострокову мотивацію гравця. Величезний світ та різноманітність біомів вимагають детального вивчення потреб кожної прирученої істоти.

Сильні сторони ігрового процесу:

– унікальна механіка приручення тварин;
– масштабне дерево технологій та характеристик.

Слабкі сторони ігрового процесу:

– низький рівень оптимізації та високі вимоги до апаратної частини;
– надмірні потреби у часу на пізніх етапах гри.

Рис. 1.4 Вигляд відеогри ARK: Survival Evolved

1.4.5 Dont Starve

Don’t Starve – це хардкорний симулятор, де виживання залежить від здатності гравця планувати дії наперед. Окрім стандартних параметрів голоду, тут реалізовано показник «глузду», що додає грі психологічної глибини (рис. 1.5). Зміна пір року вимагає радикальної переорієнтації стратегії збору ресурсів, так як деякі з них доступні лише у відповідну пору.

Візуальний стиль у дусі Тіма Бертона створює унікальний ідентифікатор проєкту. Висока ціна помилки у вигляді перманентної смерті змушує гравця максимально відповідально ставитися до кожного знайденого ресурсу, що робить гру еталоном стратегічного виживання.

Сильні сторони ігрового процесу:

– оригінальний візуальний стиль та атмосфера;
– складна система взаємозалежних параметрів стану персонажа.

Слабкі сторони ігрового процесу:

– мінімальна допомога гравцеві у освоєнні ігрового процесу;
– високий поріг входження.

Рис. 1.5 Вигляд відеогри Don’t Starve

1.4.6 Порівняння існуючих відеоігор жанру виживання

Дослідження показало, що попри спільний жанровий фундамент, кожен аналог робить ставку на різні аспекти геймплею. Одні фокусуються на соціальній динаміці , інші – на атмосфері чи менеджменті складних систем. Це підтверджує гнучкість жанру та дозволяє виокремити найбільш ефективні рішення для впровадження у власну розробку (Таблиця 1.1).

Таблиця 1.1

Порівняння популярних ігор у жанрі виживання

Критерій / Гра	Minecraft	The Forest	Rust	ARK: Survival Evolved	Don’t Starve
Прості механіки збору ресурсів	+	-	+	+	-
Спрощена система крафту	-	-	+	+	-
Можливість створення бази	+	+	+	+	-
Інтуєтивні механіки виживання	+	+	+	-	-
Наявність квестової системи	-	-	+	-	+
Заздалегіть створена мапа світу	-	+	+	+	-
Наявність нейтральних неігрових персонажів	+	-	+	-	+
Наявність RPG елементу	-	+	+	+	+

1.5 Визначення функціональних та нефункціональних вимог до відеогри жанру виживання з елемениами RPG

На основі аналізу жанрових особливостей survival-RPG, вивчення вподобань аудиторії та дослідження існуючих аналогів було сформовано перелік ключових вимог до майбутньої гри. Ці критерії є фундаментом для проектування програмної структури та технічної реалізації ігрового процесу.

Функціональні вимоги:

- реалізація елементів виживання для гравця;
- система будівництва споруд гравцем;
- можливість зберігати та завантажувати ігровий процес;
- квестова система з неігровими персонажами;
- переміщення гравця по заздалегіть створеній та стилізованій мапі ігрового світу;
- агресивно налаштовані до гравця неігрові персонажі.

Нефункціональні вимоги:

- сумістність з операційною системою Windows;
- забезпечення стабільного ігрового процесу з частотою кадрів 40-50к/с;
- мінімальні системні вимоги: Процесор: INTEL CORE I5-8400 або AMD RYZEN 3 3300Х RAM: 6 гб Відеокарта: NVIDIA GeForce GTX 1060 3GB або AMD Radeon RX 570 4GB;
- Рекомендовані системні вимоги: Процесор: INTEL CORE I5-8700 чи AMD RYZEN 5 3600Х RAM: 8 гб Відеокарта: NVIDIA GeForce GTX 1080Ti or AMD Radeon RX 5700 XT.

2 ЗАСОБИ РЕАЛІЗАЦІЇ ВІДЕОГРИ

2.1 Ігровий рушій Unity

Вибір Unity як базового середовища для розробки не є випадковим. Платформа фактично монополізувала сегмент інді-розробки, залишаючись при цьому затребуваною у великих студіях завдяки своїй універсальності. Головний аргумент на користь цього рушія – його еволюційний шлях: від простого інструмента для мобільних ігор до потужної системи з глибокою оптимізацією та підтримкою майже будь-яких цільових платформ.

Архітектурно Unity базується на концепції GameObject (рис. 2.1). Тут важливо розуміти: сам об’єкт є лише порожньою сутністю в ігровому просторі. Його властивості та логіка визначаються набором компонентів, що додаються розробником (рис. 2.2). Цей метод радикально спрощує проектування. Замість того, щоб щоразу переписувати базовий код, ми просто комбінуємо готові модулі, адаптуючи їх під потреби конкретного сценарію.

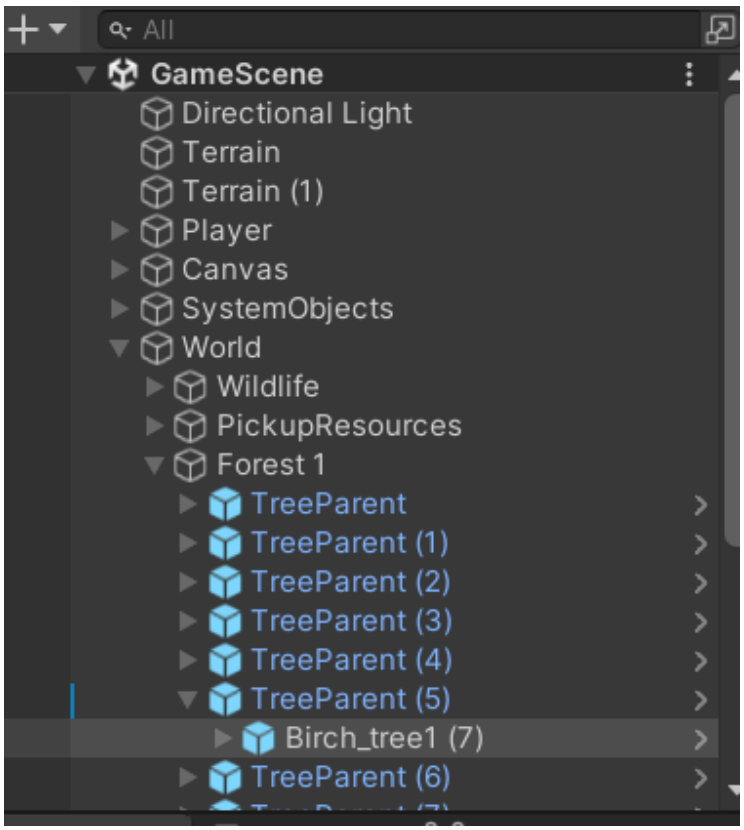


Рис. 2.1 Архітектура об’єктів у Unity

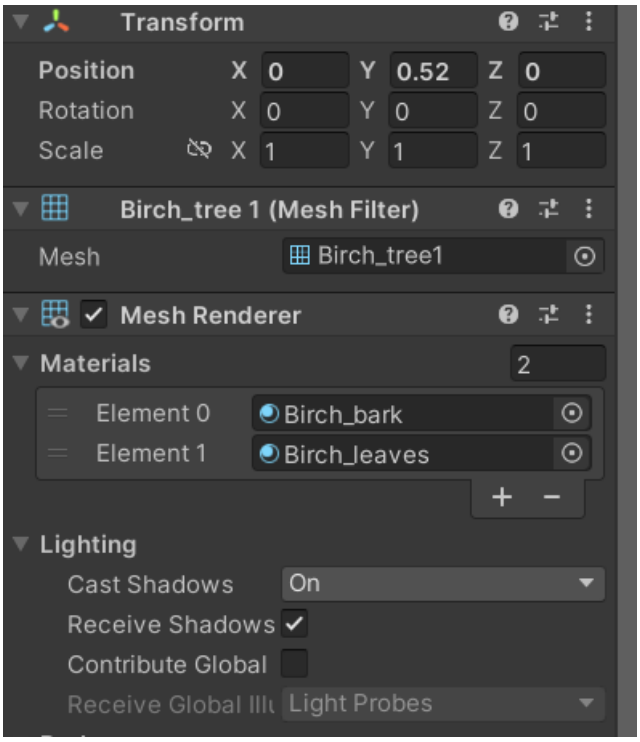


Рис. 2.2 Список компонентів доданих до об’єкта

Для роботи з фізикою в проекті було задіяно модулі Rigidbody та Collider (рис. 2.3). Саме вони відповідають за те, як об’єкт реагує на гравітацію чи зіткнення з оточенням. Плавність же візуальних переходів забезпечує система Animator (рис. 2.4). За допомогою складних графів станів вона дозволяє контролювати анімацію моделей залежно від ігрових подій.

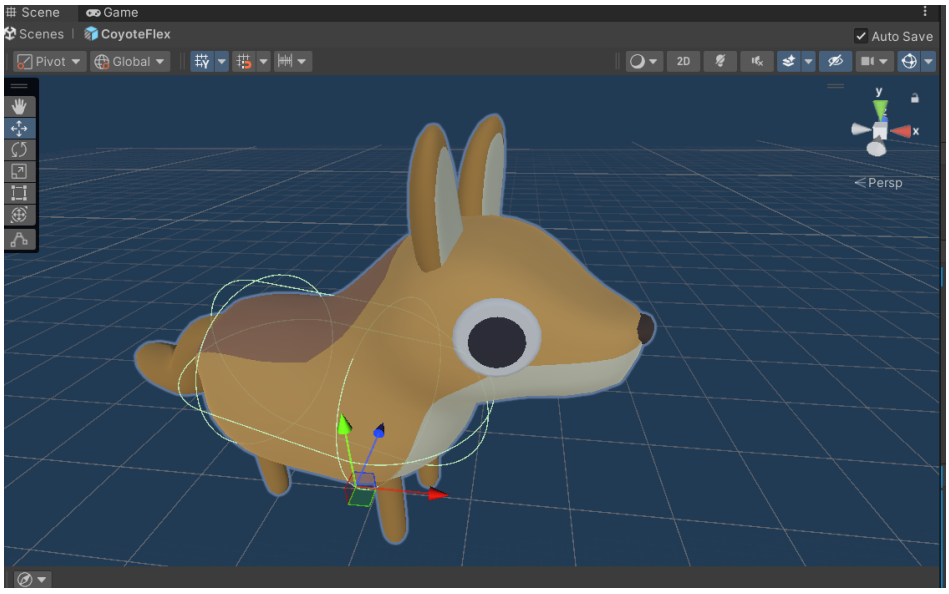


Рис. 2.3 Rigidbody та Collider у інспекторі

Рис. 2.4 Інтерфейс Animator у Unity

За візуальну частину відповідає цілий стек технологій: від шейдерів до систем постобробки. Важливим підсиленням для розробника є Asset Store. Це не просто магазин моделей, а повноцінна екосистема, що дозволяє інтегрувати в проект складні скрипти чи плагіни, значно економлячи час на рутинних завданнях. Окрім цього, Unity демонструє відмінну сумісність із системами контролю версій, що робить її придатною для командної розробки та масштабування складних проєктів.

2.1.1 Особливості розробки відеогри з використанням рушія Unity

В основі Unity лежать архітектурні рішення, які докорінно відрізняють його від класичних ігрових рушіїв. Головна відмінність – відмова від жорсткого успадкування класів на користь принципу композиції. В Unity будь-яка сутність – це «конструктор», що збирається з окремих функціональних блоків. Це дозволяє розробнику гнучко маніпулювати поведінкою об'єктів, не захаращуючи кодову базу зайвими зв'язками. Використання мови C# лише підсилює цю гнучкість, забезпечуючи високу швидкість написання скриптів при дотриманні сучасних стандартів ООП.

Одним із найсильніших інструментів Unity є режим Play Mode. Можливість миттєво запустити сцену прямо в редакторі та «на льоту» підправити параметри об'єктів – це те, що економить години робочого часу. На відміну від важких рушіїв,

де кожен тест вимагає тривалої перезбірки проєкту, Unity дозволяє проводити ітерації майже безперервно.

До того ж екосистема рушія доповнюється платформою Unity Asset Store (рис. 2.5). Наявність тисяч готових скриптів, моделей та візуальних ефектів робить розробку швидшою, а поріг входу – нижчим. Важливим фактором є і кросплатформеність: архітектура рушія побудована так, щоб перенесення гри з ПК на мобільні пристрої чи консолі вимагало лише косметичних змін у коді.

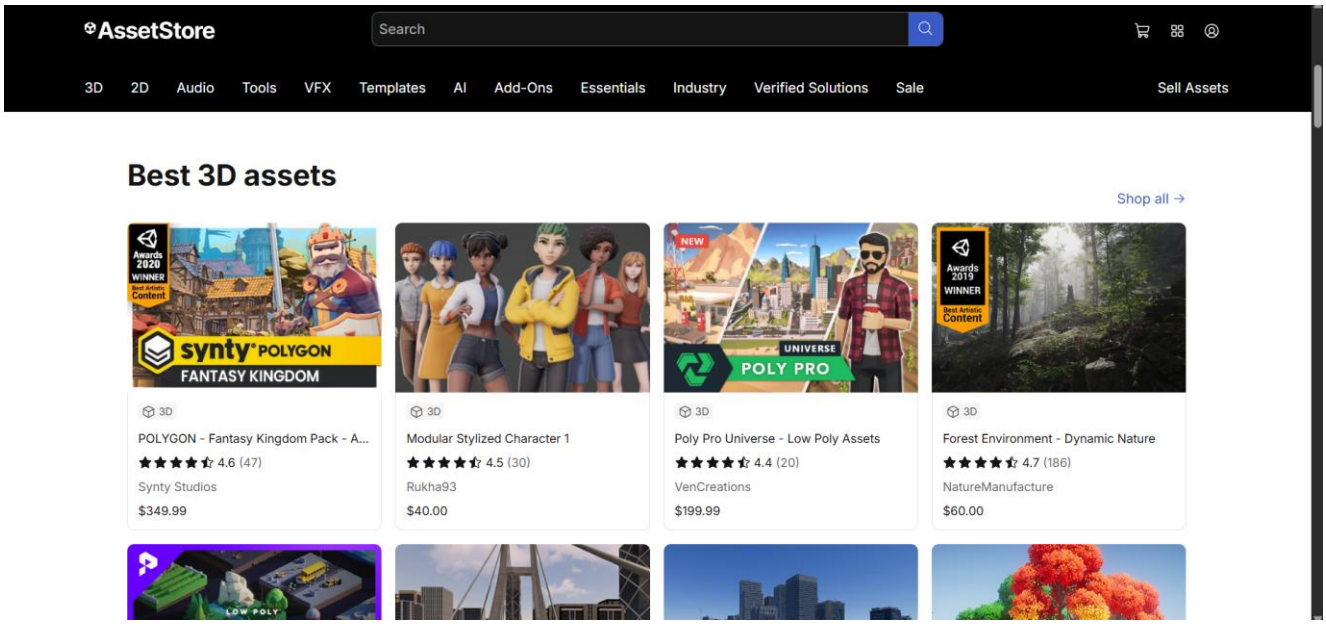


Рис. 2.5 Unity Asset Store

2.1.2 Недоліки рушія Unity

Попри свою популярність, Unity не є ідеальним інструментом. Основна проблема полягає в тому, що висока свобода дій розробника часто стає причиною проблем із продуктивністю. Якщо не приділяти належної уваги оптимізації, надмірна кількість активних об’єктів або «важкі» скрипти швидко «з’їдають» ресурси системи, що призводить до падіння кадрів.

Ще один нюанс – графіка. Без модифікації Unity може поступатися рушіям, заточеним під ААА-ринок. Щоб досягти фотореалістичного зображення, доводиться заглиблюватися в налаштування конвеєрів рендерингу (Scriptable Render Pipeline) та писати складні кастомні шейдери.

Окремо варто згадати ризики довгих проєктів. В Unity немає єдиного «правильного» стандарту організації коду, тому без самодисципліни архітектура гри швидко перетворюється на хаотичне накопичення скриптів, які важко масштабувати. Також розробники часто стикаються з проблемою оновлень: нова версія рушія може бути несумісною зі старими плагінами, що змушує витрачати час на переписування існуючого функціонала.

2.1.3 Порівняння ігрового рушія Unity з іншими популярними рушіями

Для фінального вибору технологій було порівняно можливості Unity з найближчими конкурентами – Unreal Engine та Godot (Таблиця 2.1).

Таблиця 2.1

Порівняння відеоігрових рушіїв

Критерій	Unity	Unreal Engine	Godot
Підтримка 2D-розробки	+	-	+
Підтримка 3D-графіки високого рівня	+	+	-
Фотореалістична графіка	-	+	-
Простота освоєння для новачків	+	-	+
Використання простої мови програмування	+	-	+
Візуальне програмування	+	+	-
Швидке прототипування	+	-	+
Підтримка мобільних платформ	+	+	+
Великий маркетплейс готових ресурсів	+	+	-
Велика спільнота розробників	+	+	-
Підходить для інді-розробки	+	-	+
Підходить для AAA-проєктів	-	+	-
Високі вимоги до комп’ютера	-	-	+

Аналіз показав, що Unity (рис. 2.6) пропонує найбільш збалансований інструментарій. Його головні «козирі» – це швидке прототипування, величезна база навчальних матеріалів та зручна робота з C#. Це робить його ідеальним для інді-команд та розробки ігор у жанрі survival-RPG, де потрібно часто змінювати та тестувати нові механіки виживання.

Рис. 2.6 Інтерфейс відеоігрового рушія Unity

Для порівняння, Unreal Engine (рис. 2.7) – це «важка артилерія» для створення фотореалістичних світів. Він домінує в AAA-сегменті завдяки потужним інструментам роботи зі світлом та фізикою. Проте для невеликої команди він часто виявляється занадто складним через високий поріг входження (C++) та серйозні вимоги до «заліза».

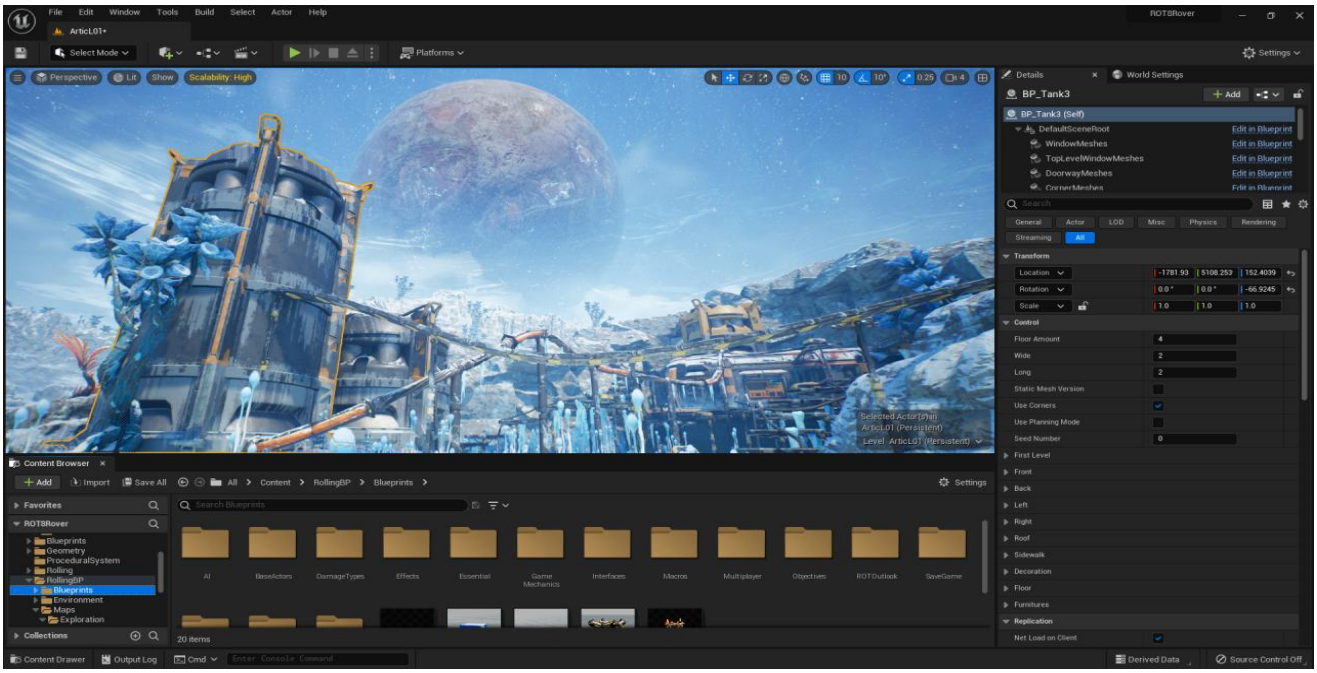


Рис. 2.7 Інтерфейс відеоігрового рушія Unreal Engine

З іншого боку, Godot (рис. 2.8) приваблює своєю легкістю та повністю безкоштовною моделлю розповсюдження. Це чудовий варіант для невеликих 2D-ігор, але його 3D-можливості та ринок готових асетів поки що значно поступаються Unity.

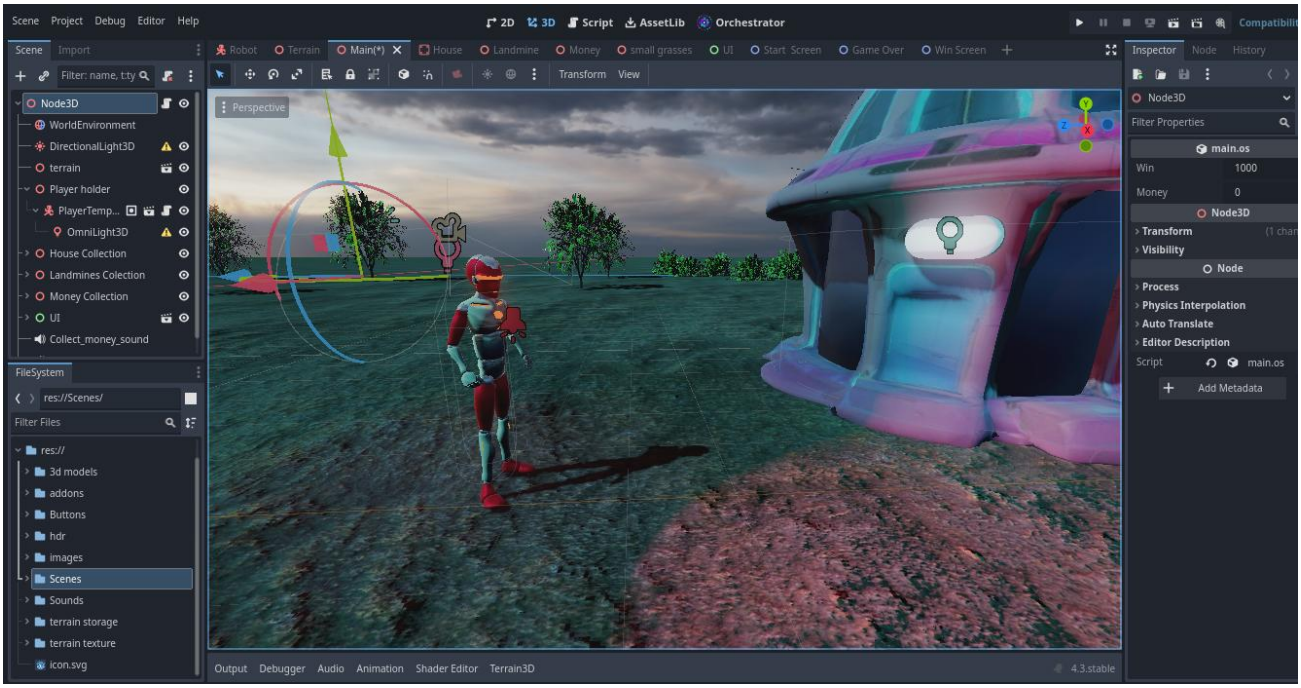


Рис. 2.8 Інтерфейс відеоігрового рушія Godot

Таким чином, враховуючи специфіку проєкту (3D, survival-RPG, необхідність швидкої розробки), Unity є найбільш раціональним та технічно виправданим вибором.

2.2 Мова програмування C#

Вибір C# як основного інструмента розробки в Unity – це передусім питання балансу. Дана мова програмування являє собою потужний інструмент для розробки високоякісних програмних рішень . Для ігрового проєкту, де архітектура може бути надзвичайно заплутаною через тисячі взаємодій, критично важливою є стабільність, яку надає платформа .NET. Це дозволяє розробнику не «винаходити колесо», а будувати масштабовані системи, використовуючи перевірені часом бібліотеки.

Якщо порівнювати C# із C++ чи Java, то його головна перевага – низький поріг входження при збереженні професійного функціонала. Програміст може менше думати про низькорівневе керування залізом і більше – про логіку самої гри. Велика заслуга в цьому належить механізму Garbage Collector. Автоматичне керування пам’яттю фактично страхує проєкт від витоків ресурсів, які в ручному режимі часто стають причиною критичних вильотів програми. У підсумку, C# стає тим містком, який дозволяє швидко перетворити ідею механіки на працюючий прототип.

2.2.1 Особливості використання мови програмування C# у Unity

В Unity програмування на C# невід’ємне від компонентного підходу. Кожен об’єкт на сцені (GameObject) – це, по суті, порожній тримач, чия поведінка залежить від «навішаних» на нього скриптів. Вся логіка тут тримається на життєвому циклі класу MonoBehaviour. Це набір подій, які рушій викликає автоматично: наприклад, Awake та Start ініціалізують дані, а Update чи FixedUpdate обробляють логіку та фізику кожного кадру. Така подієва модель дозволяє чітко

розмежувати, коли саме має спрацювати код, що значно полегшує налагодження гри.

Ще одна важлива фішка – синергія коду з редактором Unity. Ми можемо писати кастомні скрипти, які змінюють сам інтерфейс розробки (інспектори, вікна), підлаштовуючи рушій під потреби конкретного проєкту. Доступ до розгалужених API дозволяє «з коробки» працювати з такими складними речами, як навігація штучного інтелекту чи системи інвентарю.

Проте за зручність доводиться платити увагою до продуктивності. Той самий Garbage Collector може викликати мікрофризи, якщо створювати забагато об’єктів у циклі Update. Саме тому в проєкті було використано методи оптимізації: замість постійного пошуку компонентів через GetComponent, посилання на них кешуються заздалегідь, а важкі обчислення винесені в окремі методи або виконуються рідше, ніж раз на кадр.

2.2.2 Визначення об'єктно орієнтованого програмування у C#

Для гри в жанрі survival-RPG об’єктно-орієнтоване програмування (ООП) – це не просто теорія, а спосіб вижити в морі коду. Коли у вас є сотні предметів, ворогів та квестів, лише розбиття системи на окремі класи-об’єкти дозволяє тримати все під контролем. В архітектурі проєкту ми спиралися на чотири класичні «кити» ООП:

– інкапсуляція. Ми ховаємо внутрішню логіку скрипта (наприклад, формулу розрахунку здоров’я) за приватними змінними. Це гарантує, що інший розробник або випадковий скрипт не змінить дані «ззовні», зламавши логіку гри;

– успадкування. Замість того, щоб писати код для кожного типу зброї з нуля, ми створюємо базовий клас Item, а вже від нього наслідуємо Sword, Axe чи Bow, додаючи лише специфічні деталі. Це економить тисячі рядків коду;

– поліморфізм. Дозволяє нам звертатися до різних об’єктів однаково. Наприклад, ми можемо викликати метод .Interact() для дверей, скрині чи NPC Кожен об’єкт відреагує по-своєму, але для головного коду це буде одна проста команда;

– абстракція. Допомагає не потонути в деталях. Через інтерфейси (наприклад, IDamageable) ми визначаємо лише те, що об’єкт може отримати пошкодження, не вникаючи в те, як саме він це опрацює – чи це фізична броня, чи магічний щит.

Така структура робить структуру відеогри гнучкою. Ми можемо легко додати новий тип ворога чи механіку виживання, не переписуючи при цьому половину гри.

2.3. Середовище програмування Visual Studio у розробці відеоігор на рушії Unity

Для написання скриптів на C# було обрано середовище Visual Studio (рис. 2.9). У зв’язці з Unity воно працює не просто як редактор тексту, а як повноцінний діагностичний центр. Головна причина вибору саме цієї IDE – повна сумісність із рушієм. Це дозволяє контролювати стан коду на всіх етапах: від перших чернеток логіки до фінальних правок перед збіркою гри.

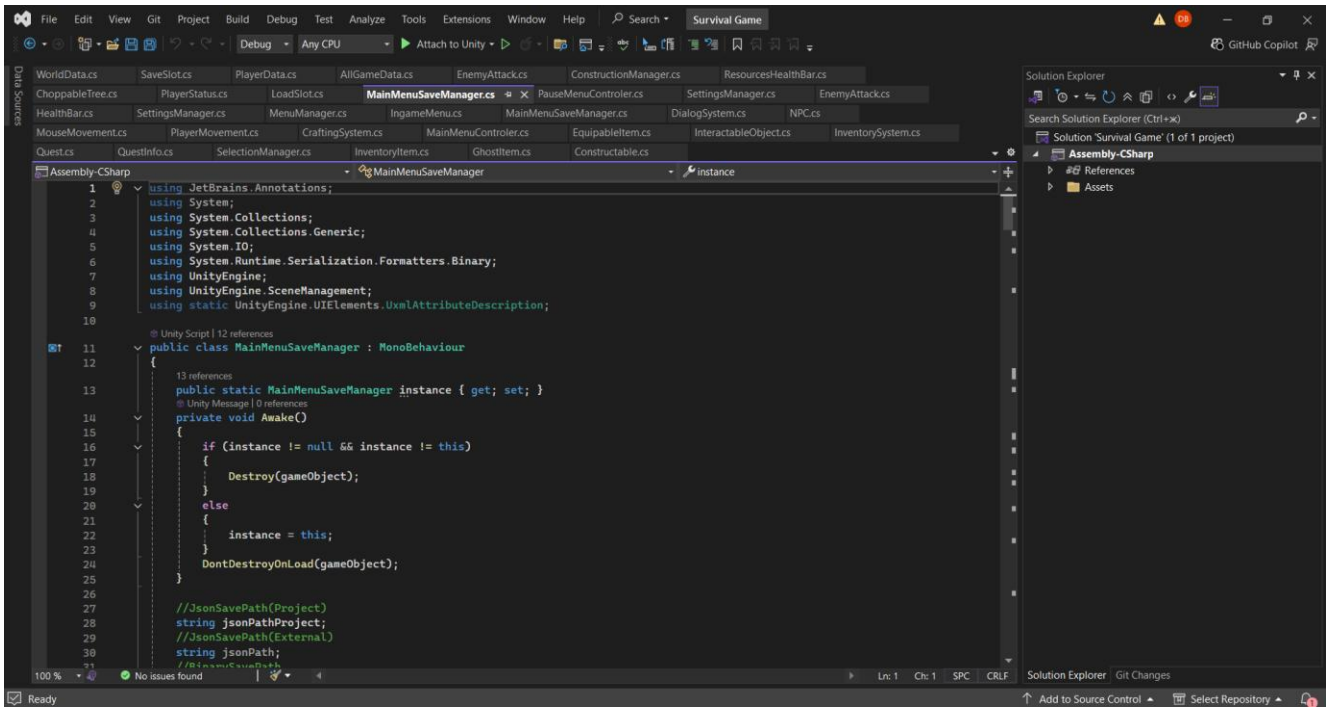


Рис. 2.9 Інтерфейс середовища програмування Visual Studio

Процес написання коду спрощує система IntelliSense. Вона працює як інтелектуальна підказка, що пропонує потрібні методи чи змінні прямо під час друку. Для нашого проекту, де кількість взаємопов’язаних скриптів постійно зростає, це стало критичним. Це не лише економить час, а й «відловлює» синтаксичні помилки ще до того, як код потрапить в Unity. Окрім того, вбудовані інструменти автоматично вирівнюють структуру тексту, що допомагає підтримувати проект у чистому стані.

Однак найважливіша функція Visual Studio – це налагодження. Ми підключаємо середовище до Unity, щоб відстежити виконання програми крок за кроком. Використовуючи точки зупинки, можна в будь-який момент «заморозити» гру і подивитися, що відбувається всередині змінних.

Також IDE допомагає з оптимізацією. Через вбудовані засоби аналізу ми моніторимо, як гра навантажує пам'ять та процесор. Це дозволяє вчасно знайти «вузькі місця» (bottlenecks), через які падає FPS. У поєднанні з функціями рефакторингу, Visual Studio дає можливість швидко змінювати структуру коду без ризику зламати вже працюючі частини гри.

3. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОЇ СКЛАДОВОЇ ГРИ

3.1 Формулювання концепції відеогри

Концепція розробленої відеогри базується на об’єднанні жанру виживання з елементами RPG, що не дає гравцю занудьгувати від монотонного ігрового процесу та додає нові ігромеханічні можливості. Головним геймплейним елементом є стандартна для відеоігор жанру виживання система життєвих показників: здоров’я, голод, спрага. Їх підтримання і є основним мотиватором для гравця досліджувати інші механіки такі як: добування ресурсів, будівництво, крафтинг. Роль же піджанру RPG у тому щоб збагатити ігровий процес гравця додаючи неігрових персонажів, квести та цікаві локації з ворогами.

Наявність елементів RPG також дозволяє розкрити наративний потенціал відеогри. Гравець бере на себе роль молодого бобра що втратив дім та сім’ю після неназваної катастрофи. Тепер його ціллю є зрозуміти де він і вижити у незнайомому середовищі.

При першому завантаженні персонаж з’являється на галявині поруч з невеликим селищем. Гравець має вибір що йому робити, добувати базові ресурси та створювати примітивні інструменти або одразу бігти розмовляти з мешканцями селища щоб отримати підказки або завдання. При бажанні закінчити ігрову сесію або перед важливим етапом ігрового процесу гравець може зберегти все зроблене ним у меню паузи, щоб у майбутньому мати можливість завантажити файл та продовжити там де зупинився.

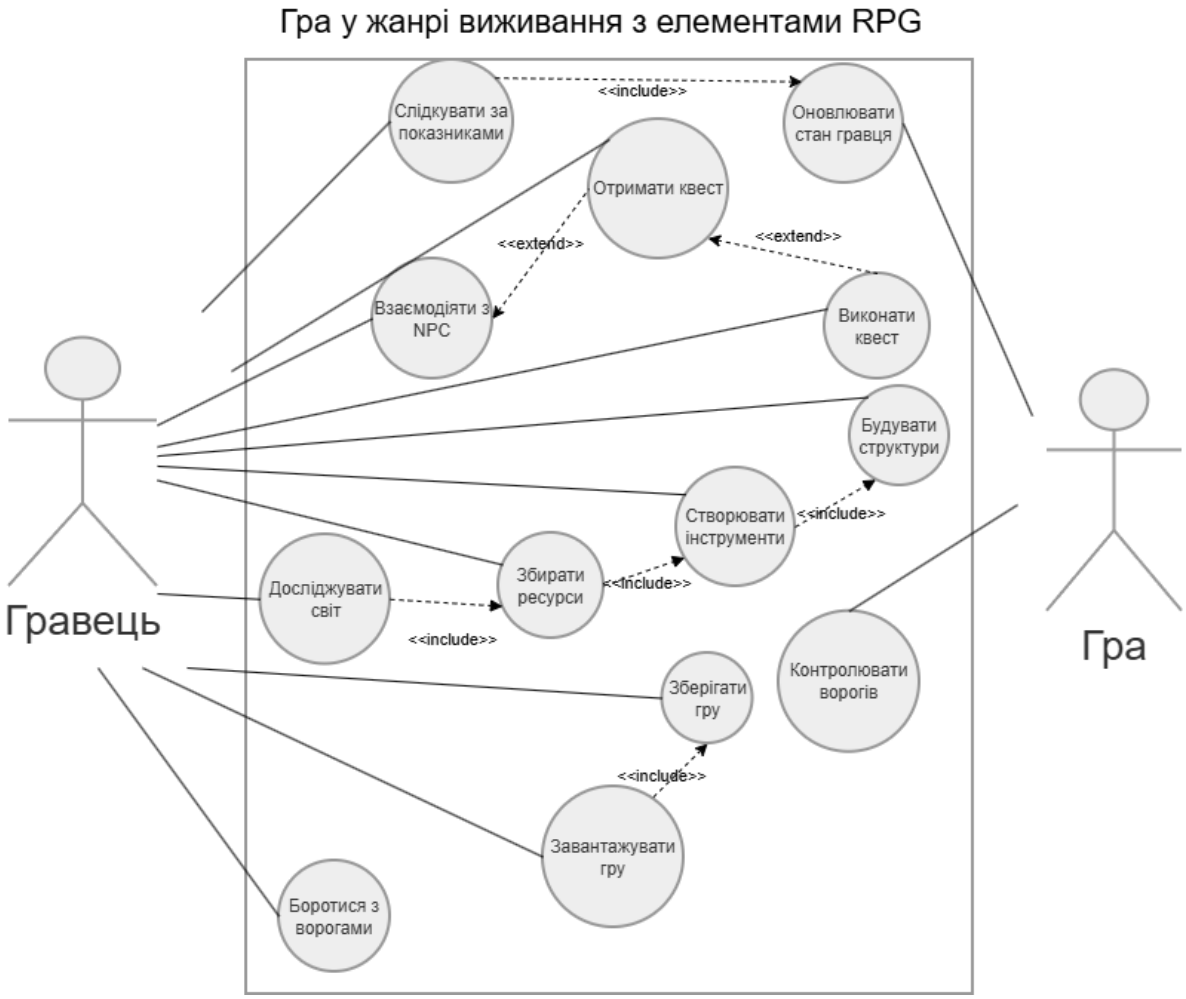


Рис. 3.1 Діаграмма прецедентів відеогри

3.2 Реалізація ігрових механік

3.2.1 Реалізація базових механік гравця

Механіки, пов’язані з гравцем, реалізовані через взаємодію кількох основних класів: PlayerMovement, MouseMovement, PlayerStatus, SelectionManager та EquipableItem. Разом вони формують повноцінну систему керування персонажем у середовищі, де гравець може пересуватись, взаємодіяти зі світом, використовувати інструменти та підтримувати власні життєві показники. Для більш наглядного показу взаємодії була створена діаграма класів (Рисунок 3.2).

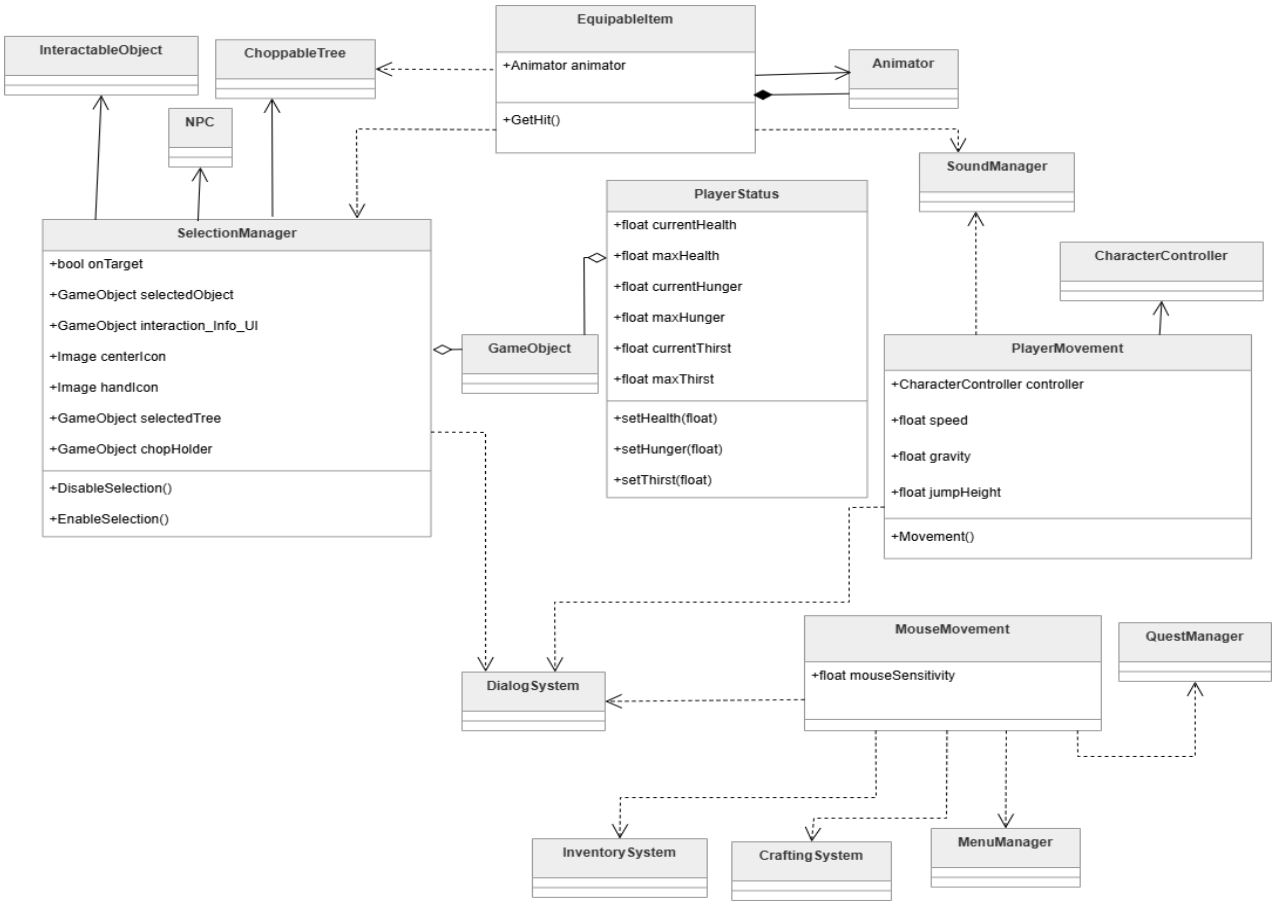


Рис. 3.2 Діаграма класів ігрового персонажа

Клас PlayerMovement відповідає за фізичний рух персонажа. Для переміщення використовується компонент CharacterController, який забезпечує плавний рух без необхідності прямого використання Rigidbody. Рух здійснюється за допомогою клавіш WASD через осі Horizontal та Vertical. Напрямок руху залежить від орієнтації персонажа, тому гравець завжди рухається відносно напрямку камери. У системі також реалізована механіка стрибка. Перед виконанням стрибка перевіряється, чи знаходиться персонаж на землі за допомогою Physics.CheckSphere. Якщо умова виконується, обчислюється вертикальна швидкість стрибка на основі значення гравітації та бажаної висоти. Гравітація постійно впливає на персонажа, що створює природну фізику падіння. Додатково реалізована система звуків руху: якщо позиція гравця змінюється, відтворюється звук кроків, а при зупинці звук вимикається. Рух персонажа може бути тимчасово заблокований, наприклад під час діалогів.

Клас MouseMovement реалізує систему руху камери від першої особи. Переміщення миші по горизонталі відповідає за поворот персонажа навколо осі Y, а вертикальний рух керує нахилом камери. Для уникнення неприродного перевертання камери використовується обмеження кута огляду через Mathf.Clamp. На початку гри курсор блокується в центрі екрана та приховується. Огляд мишкою автоматично вимикається під час відкриття інтерфейсів, таких як інвентар, меню крафту, діалоги або меню квестів. Це запобігає випадковому обертанню камери під час взаємодії з UI.

Система стану персонажа реалізована у класі PlayerStatus. Вона відповідає за показники здоров'я, голоду та спраги. Після першого запуску гри всі параметри встановлюються на максимальні значення. Механіка спраги реалізована через coroutine, яка кожні сім секунд автоматично зменшує рівень спраги гравця. Голод зменшується залежно від пройденої дистанції: система накопичує відстань, яку пройшов персонаж, і після досягнення певного порогу зменшує показник голоду. Таким чином створюється та сама основа механіки виживання, де активне пересування персонажа впливає на його потреби. Для отримання шкоди використовується метод TakeDamage, який зменшує здоров'я персонажа та перевіряє умову смерті. Значення здоров'я обмежується в допустимому діапазоні за допомогою Mathf.Clamp. Клас реалізований як Singleton, тому інші системи можуть глобально отримувати доступ до стану гравця.

Взаємодія зі світом реалізована через клас SelectionManager. Система постійно виконує Raycast із центру камери в напрямку погляду гравця та визначає, на який об'єкт наведений приціл. Якщо об'єкт містить компонент InteractableObject і гравець знаходиться в зоні взаємодії, на екрані відображається текстова підказка та змінюється іконка взаємодії. Якщо предмет має тег pickable, замість стандартного прицілу відображається іконка руки, що сигналізує про можливість підбору. Через цю систему також реалізована взаємодія з NPC. Якщо NPC знаходиться в радіусі взаємодії, на екрані з'являється напис «Talk», а натискання лівої кнопки миші запускає діалог. Під час активного діалогу UI взаємодії автоматично приховується. Окремо система підтримує взаємодію з деревами типу

ChoppableTree. Якщо дерево знаходиться в зоні досяжності, воно стає активним для рубки, зберігається як поточне вибране дерево, а на екрані з’являється індикатор рубки.

Клас EquipableItem відповідає за використання екіпірованого інструмента, наприклад сокири. Для цього використовується компонент Animator, який автоматично додається до об’єкта через атрибут RequireComponent. При натисканні лівої кнопки миші, якщо інвентар і меню крафту закриті, запускається анімація удару через тригер hit. Під час виконання анімації викликається метод GetHit(), який перевіряє, чи вибране дерево присутнє в SelectionManager. Якщо дерево знайдено, викликається метод GetHit у класі ChoppableTree, що завдає дереву шкоди або зменшує його міцність. Одночасно відтворюється звук рубки через SoundManager.

Усі ці компоненти працюють разом як єдина система. MouseMovement визначає напрямок погляду гравця, SelectionManager аналізує об’єкти перед камерою, PlayerMovement забезпечує переміщення персонажа, EquipableItem дозволяє використовувати інструменти для взаємодії зі світом, а PlayerStatus контролює життєві параметри персонажа. Така архітектура є модульною: кожна механіка винесена в окремий компонент, що спрощує підтримку, розширення та повторне використання коду в інших частинах проєкту.

3.2.2 Розробка механік збору ресурсів

Механіка збору ресурсів у грі побудована на системі взаємодії гравця з об’єктами навколишнього середовища через вибір цілі, перевірку дистанції та виконання відповідної дії (рисунок 3.3).

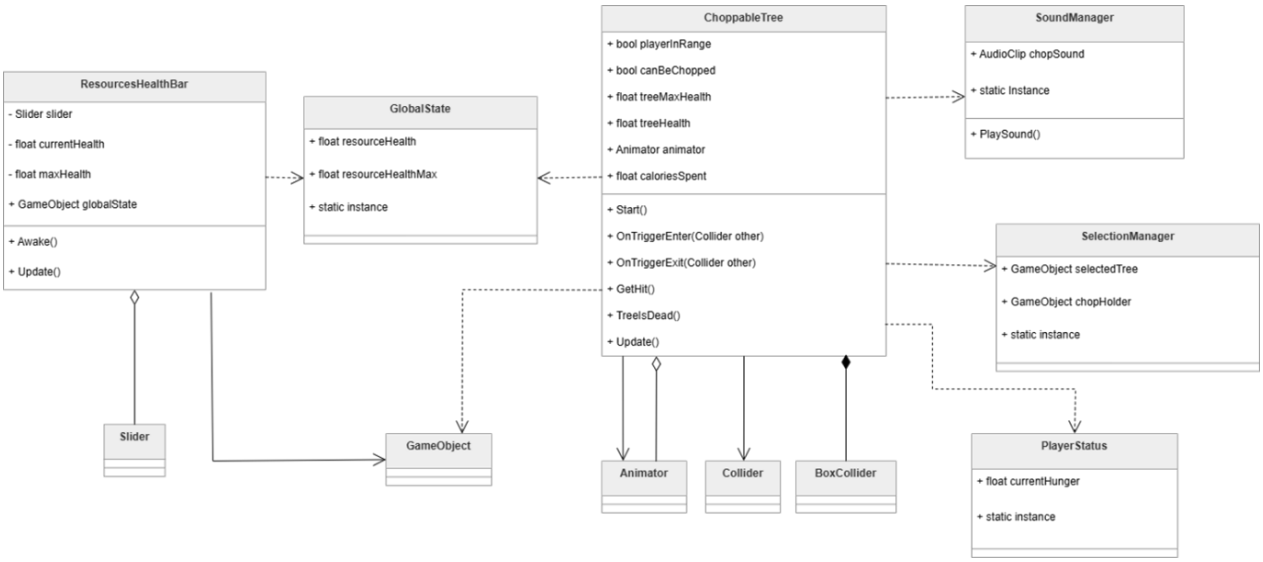


Рис. 3.3 Діаграма класів збору ресурсів

Усі предмети, які можна підібрати, використовують клас InteractableObject. Коли гравець входить у зону колайдера такого об’єкта, змінна playerInRange стає активною, що означає можливість взаємодії. Система вибору об’єктів SelectionManager постійно виконує Raycast із камери в напрямку курсора миші та визначає, на який об’єкт дивиться гравець. Якщо об’єкт є інтерактивним і гравець знаходиться поруч, на екрані відображається інтерфейс взаємодії з назвою предмета, а центральна іконка змінюється на іконку руки, сигналізуючи про можливість підбору. Підбір ресурсу виконується натисканням клавіші E. Перед додаванням предмета система інвентарю перевіряє, чи є вільне місце. Якщо інвентар не заповнений, предмет додається через метод AddToInventory, відтворюється звуковий ефект підбору через SoundManager, а сам об’єкт видаляється зі сцени за допомогою Destroy. Також назва підібраного предмета зберігається у списку itemsPicked, що дозволяє відстежувати вже зібрані ресурси. Якщо місця в інвентарі немає, виводиться повідомлення про переповнення інвентаря.

Для добування деревини реалізована окрема механіка через клас ChoppableTree. Дерева мають власне здоров’я (treeHealth) та максимальний запас міцності (treeMaxHealth). Коли гравець підходить до дерева, система визначає можливість рубки через playerInRange. Якщо дерево знаходиться в полі зору

гравця, SelectionManager активує режим рубки, зберігає посилання на вибране дерево та вмикає UI-елемент chopHolder. Під час удару по дереву викликається метод GetHit, який запускає анімацію хитання дерева через Animator, відтворює звук рубки та зменшує запас здоров’я дерева. Одночасно в персонажа зменшується рівень голоду (currentHunger).

Стан дерева відображається через систему GlobalState, у якій зберігаються поточне та максимальне значення здоров’я ресурсу. Клас ResourcesHealthBar отримує ці значення та оновлює UI-слайдер, який показує прогрес руйнування дерева у реальному часі. Це дозволяє гравцю бачити, скільки ударів залишилося до повного знищення ресурсу. Коли здоров’я дерева зменшується до нуля, викликається метод TreeIsDead. Оригінальне дерево видаляється зі сцени, вимикається режим рубки, очищується вибрана ціль у SelectionManager, а на місці дерева створюється об’єкт зрубаного дерева ChopedTree. Таким чином механіка забезпечує повний цикл взаємодії: виявлення ресурсу, відображення інтерфейсу, взаємодію, анімацію добування, використання витривалості персонажа та зміну стану світу після збору ресурсу.

3.2.3 Реалізація механік крафту

Клас CraftingSystem реалізує повну систему крафту предметів і будівель у грі, забезпечуючи відкриття інтерфейсу крафту, перемикання між категоріями, перевірку необхідних ресурсів та створення нових об’єктів. Основною задачею класу є керування взаємодією між інвентарем гравця, UI-інтерфейсом і рецептами створення предметів. Для цього використовується шаблон Singleton, завдяки якому доступ до системи крафту можна отримати через CraftingSystem.instance з будь-якої частини проєкту.

У класі зберігаються посилання на основні елементи інтерфейсу: головне меню крафту craftingScreenUI, меню інструментів toolScreenUI та меню будівництва buildsScreenUI. Також визначені кнопки категорій і кнопки створення конкретних предметів. Система підтримує декілька рецептів через об’єкти типу Blueprint. Наприклад, для створення сокири (Axe) необхідно 3 одиниці каменю

(Stone) та 2 палиці (Stick), для стіни (Wall) – 2 колоди (Log), а для фундаменту (Foundation) – 4 колоди. У кожному blueprint зберігається назва предмета, кількість необхідних ресурсів та їх типи. Під час запуску гри метод Start() ініціалізує всі UI-компоненти та прив'язує кнопки до відповідних функцій через AddListener. Кнопки категорій викликають методи OpenToolsCategory() та OpenBuildsCategory(), які перемикають між екранами інструментів і будівництва. Кнопки крафту запускають метод CraftItem(), передаючи відповідний blueprint.

Метод CraftItem() відповідає за безпосереднє створення предмета. Спочатку новий предмет додається до інвентарю через InventorySystem.Instance.AddToInventory(). Після цього система видаляє потрібні ресурси з інвентаря за допомогою RemoveItem(). Якщо рецепт містить один ресурс, видаляється лише один тип матеріалу, якщо два – видаляються обидва. Після успішного крафту відтворюється звуковий ефект через SoundManager, а далі запускається coroutine calculate(), яка оновлює список предметів в інвентарі та перевіряє доступність рецептів.

Відкриття та закриття системи крафту виконується у методі Update(). При натисканні клавіші U система перевіряє поточний стан меню. Якщо меню закрите, активується UI крафту, розблоковується курсор, робиться видимим покажчик миші та вимикається SelectionManager, щоб гравець не міг взаємодіяти зі світом під час крафту. Якщо меню вже відкрите, усі вікна крафту закриваються, а керування персонажем та система вибору об'єктів знову активуються.

Метод RefreshNeededItems() виконує аналіз інвентаря гравця та підраховує кількість ресурсів: каменю, палиць і колод. Для цього використовується список inventoryitemList, який отримується з InventorySystem. Після підрахунку система оновлює текстові поля вимог у UI, показуючи поточну кількість ресурсів. Далі перевіряється, чи вистачає ресурсів для створення предметів. Якщо ресурсів достатньо, відповідна кнопка крафту стає активною та відображається на екрані, інакше вона приховується. Таким чином гравець бачить лише ті предмети, які може створити в поточний момент.

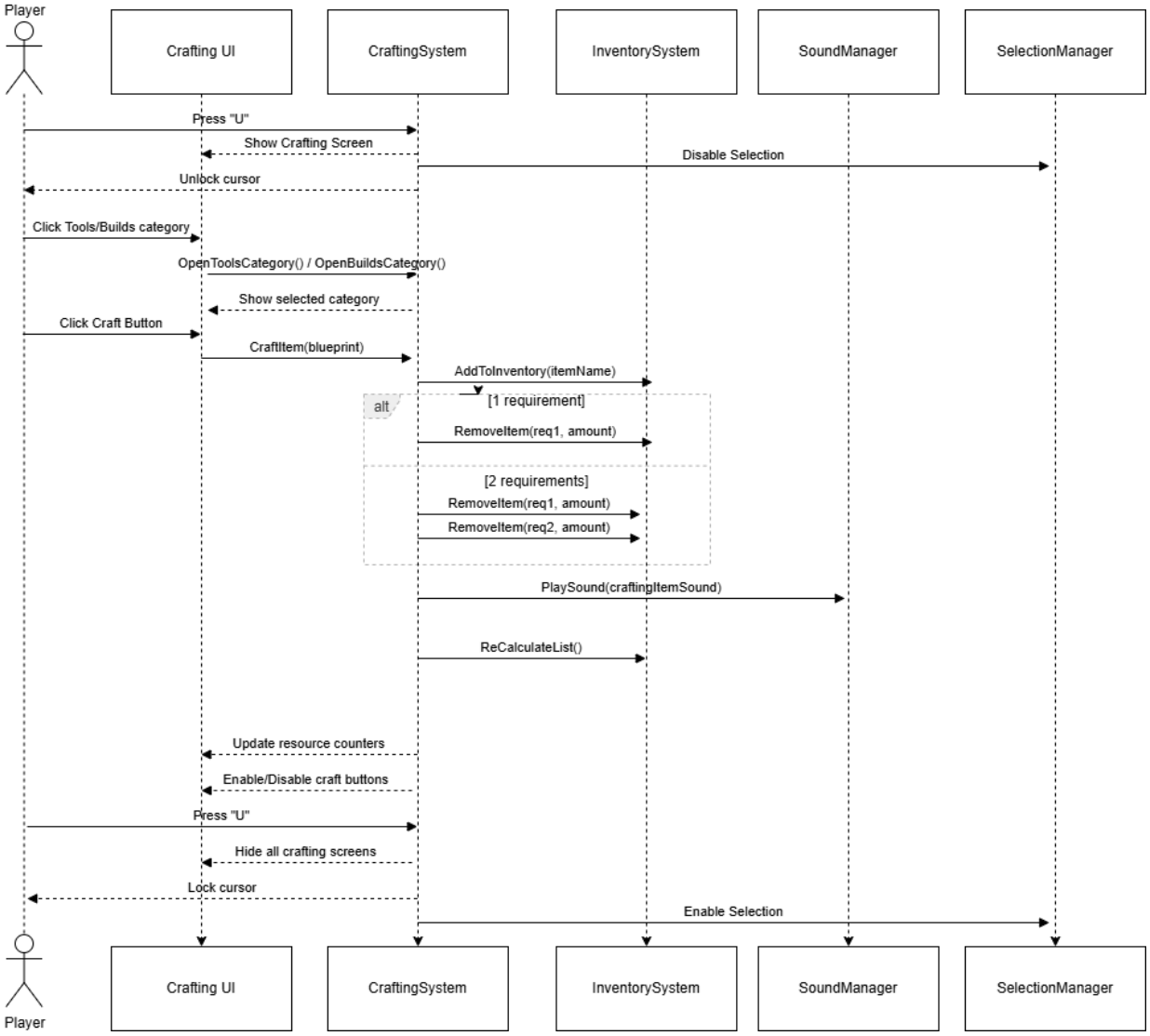


Рис 3.4 Діаграма послідовності системи крафту

3.2.4 Розробка системи будівництва

Механіка будівництва у грі реалізована через систему розміщення конструкцій, перевірки коректності побудови та керування “примарними” об’єктами, які дозволяють гравцю попередньо бачити майбутню споруду перед остаточним встановленням. Основою цієї системи є класи Constructable() та GhostItem(), які відповідають за логіку перевірки місця будівництва, зміну стану об’єктів та взаємодію з режимом конструювання. Детальніше взаємодію класів можна побачити на діаграмі (Рисунок 3.5).

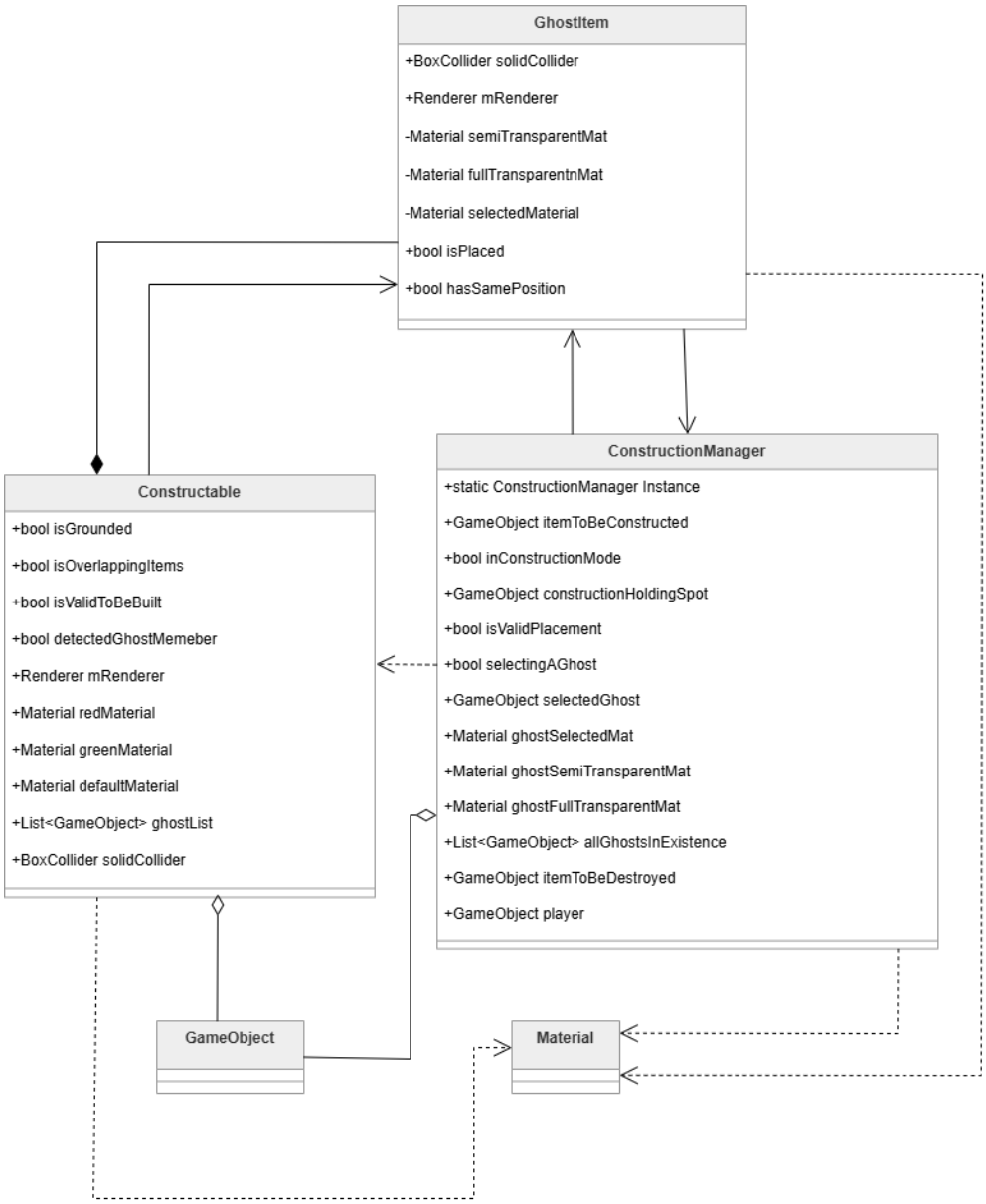


Рисунок 3.5 Діаграма класів системи будівництва

Клас Constructable() використовується для всіх об’єктів, які можуть бути побудовані гравцем. Він містить набір змінних, що визначають валідність розміщення конструкції. Змінна isGrounded показує, чи знаходиться об’єкт на поверхні землі, isOverlappingItems перевіряє перетин із деревами або предметами, а isValidToBeBuilt визначає, чи можна остаточно встановити споруду у вибраному місці. Додатково використовується прапорець detectedGhostMemeber, який дозволяє виявляти інші примарні елементи будівництва. Під час запуску об’єкта метод Start() отримує компонент Renderer, встановлює стандартний матеріал та збирає всі дочірні елементи конструкції до списку ghostList. Це потрібно для подальшого відокремлення “примарних” частин споруди після завершення

будівництва. У методі Update() система постійно перевіряє умови валідності розміщення. Якщо конструкція знаходиться на землі та не перетинається з іншими об’єктами, змінна isValidToBeBuilt стає істинною, і споруду можна встановити. Якщо хоча б одна умова порушена, побудова блокується. Для візуального зворотного зв’язку система використовує зміну матеріалів конструкції. Метод SetValidColor() змінює матеріал на зелений, показуючи, що місце підходить для будівництва. Метод SetInvalidColor() встановлює червоний матеріал, якщо побудова неможлива. Метод SetDefaultColor() повертає стандартний вигляд об’єкта. Це дозволяє гравцю миттєво бачити, чи можна розмістити споруду у вибраній позиції. Після завершення будівництва викликається метод ExtractGhostMembers(). Він проходить через список ghostList, від’єднує дочірні “ghost”-елементи від основного об’єкта та переводить їх у стан завершеної конструкції. Для кожного елемента вимикається твердий колайдер та встановлюється прапорець isPlaced = true, що означає остаточне розміщення споруди у світі.

Клас GhostItem відповідає за поведінку примарних елементів конструкції. Такі об’єкти використовуються як попередній перегляд майбутньої споруди. Під час ініціалізації система отримує посилання на матеріали прозорості з ConstructionManager: напівпрозорий, повністю прозорий та матеріал вибраного елемента. За замовчуванням ghost-об’єкт стає повністю прозорим, а його фізичний колайдер вимикається, щоб він не взаємодіяв із фізикою світу до моменту встановлення. У методі Update() клас GhostItem перевіряє, чи активований режим будівництва. Якщо режим активний, система ігнорує зіткнення між гравцем та ghost-об’єктом через Physics.IgnoreCollision(), щоб персонаж міг вільно пересуватись під час розміщення конструкції. Якщо ghost-об’єкт уже встановлений, вмикається твердий колайдер, що дозволяє Raycast-системі взаємодіяти з побудованою конструкцією. Коли режим будівництва вимикається, колайдери знову відключаються. Також система підтримує вибір окремих ghost-елементів. Якщо об’єкт збігається з selectedGhost у ConstructionManager, до нього застосовується спеціальний матеріал виділення (selectedMaterial). В іншому випадку використовується прозорий матеріал. Завдяки цьому гравець бачить, який саме елемент конструкції зараз вибраний або редагується.

Підсумовуючи механіка будівництва забезпечує повний цикл створення споруд а саме: вибір конструкції, перевірку допустимості встановлення, візуальне

підсвічування валідності, взаємодію з навколишнім середовищем, попередній перегляд через ghost-об’єкти та остаточне розміщення побудованих елементів.

3.2.5 Реалізація системи взаємодії з неігровими персонажами

Механіка взаємодії з NPC побудована як діалогова система, що поєднує розмову з видачею, прийняттям і завершенням квестів через єдине вікно діалогу. Взаємодію між класами які беруть участь у цій механіки можна побачити на діаграммі нижче (Рисунок 3.6).

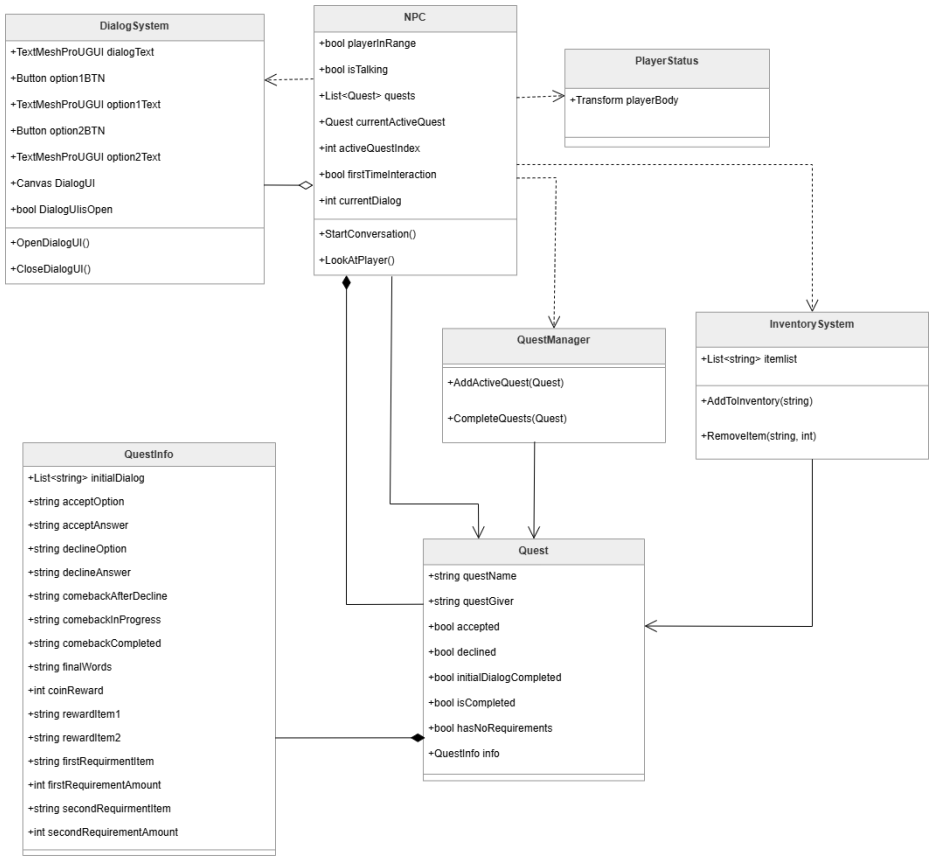


Рисунок 3.6 Діаграма класів системи взаємодії з неігровими персонажами

Клас DialogSystem відповідає за відображення та життєвий цикл інтерфейсу діалогів. Він реалізований як Singleton, що дозволяє будь-якому NPC отримувати доступ до одного спільного вікна діалогу. Система містить текст реплік, кнопки варіантів відповіді та Canvas інтерфейсу. Під час відкриття діалогу Canvas активується, курсор розблоковується та стає видимим, що дозволяє взаємодіяти з кнопками. Після завершення діалогу Canvas вимикається, курсор знову ховається,

а керування повертається гравцю. DialogSystem не містить логіки квестів – він виконує роль сервісу UI, який динамічно змінює тексти і поведінку кнопок залежно від стану розмови.

Клас NPC реалізує всю логіку взаємодії гравця з персонажем і виступає центральним вузлом квестової системи. Після входу гравця в тригер NPC встановлює прапорець playerInRange, що дозволяє системі взаємодії запустити розмову. Під час початку діалогу NPC повертається до гравця через PlayerStatus, щоб створити ефект живої розмови. Далі перевіряється стан поточного квесту: перша зустріч, прийнятий квест, відхилений квест або завершене завдання. Залежно від цього NPC формує відповідну гілку діалогу та динамічно призначає дії кнопок через AddListener. Таким чином одна і та ж кнопка може виконувати різні дії – переходити до наступної репліки, приймати квест, відмовлятися від нього або видавати нагороду.

Система діалогів базується на ScriptableObject QuestInfo, який зберігає всі тексти реплік, варіанти відповідей, фінальні слова та параметри винагороди. NPC не містить прописаних діалогів у коді – він лише читає дані з QuestInfo, що дозволяє легко створювати нові квести без зміни логіки. Початковий діалог складається зі списку рядків, які послідовно відображаються при натисканні кнопки Next. Коли гравець доходить до останньої репліки, система автоматично переходить до вибору прийняття або відхилення квесту. Після прийняття завдання NPC передає квест у QuestManager, який зберігає список активних і завершених квестів. Це створює централізовану систему прогресу. Якщо квест має вимоги, NPC під час наступної взаємодії перевіряє інвентар через InventorySystem. Перевірка виконується шляхом підрахунку кількості необхідних предметів у списку itemlist. Якщо вимоги виконані, NPC видаляє предмети з інвентаря та відкриває фінальний діалог завершення. Видача нагороди також інтегрована з іншими системами. NPC викликає метод завершення квесту в QuestManager, додає предмети у InventorySystem та повідомляє про отримання монет. Після завершення квесту індекс збільшується, і NPC автоматично переходить до наступного завдання зі

списку. Якщо квести закінчуються, персонаж переходить у фінальний стан і використовує лише завершальні репліки.

Уся взаємодія побудована як модульна система. DialogSystem відповідає за UI, NPC керує сценарієм розмови та станами квесту, QuestManager зберігає прогрес, InventorySystem забезпечує перевірку і зміну предметів, а PlayerStatus використовується для просторової взаємодії. Така архітектура дозволяє легко розширювати систему, додавати нові квести та NPC без зміни базової логіки.

3.2.6 Розробка ворогів.

Клас EnemyAttack реалізує базову бойову поведінку ворога, що поєднує навігацію за допомогою NavMesh та систему ближньої атаки гравця. Після запуску сцени у методі Start() ворог отримує компонент NavMeshAgent, який відповідає за пересування по навігаційній сітці, а також знаходить об’єкт гравця через тег Player і зберігає його Transform. Таким чином ворог одразу отримує ціль, але не починає переслідування, поки гравець не увійде в зону виявлення. Виявлення гравця реалізоване через тригер-колайдер. Коли гравець входить у цю зону, змінна playerDetected стає істинною, що активує поведінку ворога. Поки гравець знаходиться в межах тригера, метод Update постійно обчислює відстань між ворогом і гравцем. Додатково перевіряється, чи знаходиться ворог на NavMesh, що запобігає помилкам навігації, якщо агент тимчасово не підключений до навігаційної сітки. Логіка поведінки будується на простій системі станів переслідування та атаки, які перемикаються залежно від дистанції до гравця. Якщо гравець знаходиться поза межами радіуса атаки, ворог активує NavMeshAgent, знімає блокування руху та встановлює точку призначення на позицію гравця. Це створює ефект постійного переслідування, де ворог автоматично оновлює маршрут і рухається слідом за ціллю. Коли відстань стає меншою або дорівнює радіусу атаки, рух зупиняється і ворог переходить у фазу атаки. Сама атака побудована на системі затримки між ударами, що реалізована через перевірку часу. Метод Attack перевіряє, чи минув інтервал attackCooldown від попереднього удару. Якщо затримка завершилась, викликається метод TakeDamage у глобальному Singleton

класі PlayerStatus, який зменшує здоров’я гравця. Після нанесення шкоди зберігається новий час атаки, що не дозволяє ворогу завдавати шкоди кожен кадр. Коли гравець виходить із зони тригера, ворог миттєво втрачає ціль, встановлює playerDetected у значення false та зупиняє NavMeshAgent. Це означає, що ворог не патрулює карту самостійно, а активується лише при наближенні гравця. Саме так формується цілісна поведінка: ворог очікує у пасивному стані, виявляє гравця через тригер, переслідує його за допомогою навігації, зупиняється на дистанції удару та періодично завдає шкоди, поки гравець залишається в зоні виявлення.

3.3 Реалізація можливостей збереження та подальшого завантаження ігрового процесу.

Система збереження та завантаження ігрового прогресу реалізована через багаторівневу архітектуру, яка охоплює як дані гравця, так і стан ігрового світу, а також підтримує кілька слотів збереження та два формати збереження файлів JSON і binary. Взаємодію класів можна побачити на діаграмі (Рисунок 3.7).

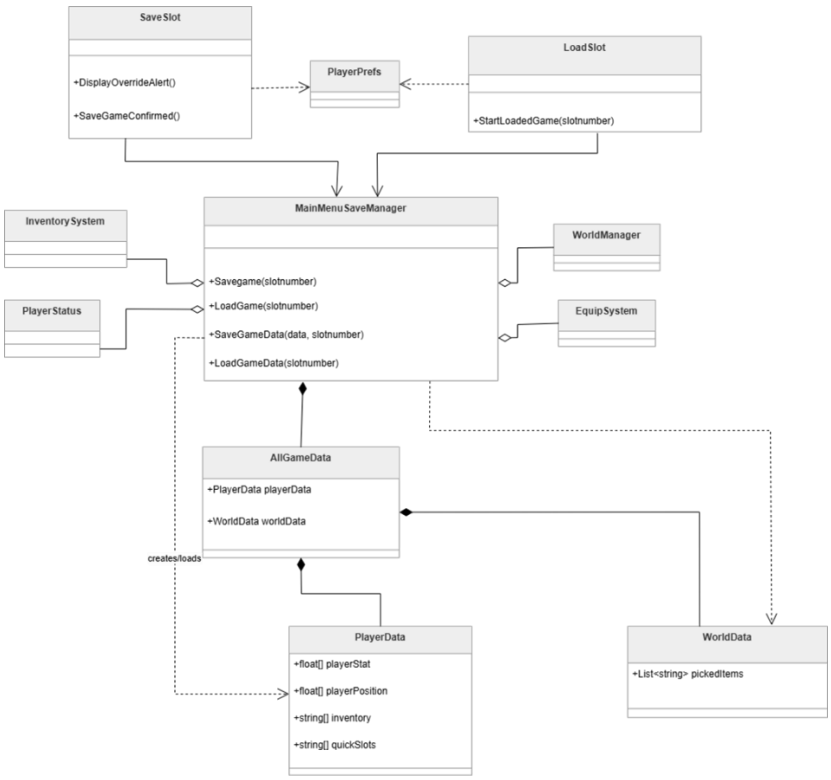


Рис. 3.7 Діаграма класів системи збереження та завантаження ігрового прогресу

Центральним елементом цієї системи є клас MainMenuSaveManager, який реалізований як Singleton, що гарантує існування лише одного екземпляра менеджера протягом усього проекту. Це досягається через перевірку в методі Awake, де при повторному створенні об’єкта він знищується, а перший екземпляр зберігається через DontDestroyOnLoad, що дозволяє зберігати дані між сценами.

Основною структурою даних виступає клас AllGameData, який є контейнером для всього стану гри. Він містить два ключові об’єкти: PlayerData і WorldData. Такий підхід дозволяє логічно розділити дані гравця та ігрового світу, що спрощує як збереження, так і подальше розширення системи. PlayerData відповідає за стан персонажа і включає масиви, які зберігають параметри здоров’я, голоду та спраги, позицію та ротацію гравця , інвентар і швидкі слоти. Уся ця інформація передається через конструктор, що забезпечує контрольоване створення об’єкта даних. WorldData, у свою чергу, зберігає список взаємодій із ігровим світом, зокрема список підібраних або знищених об’єктів , що дозволяє відновити стан світу після завантаження.

Процес збереження починається з методу Savegame, який створює новий екземпляр AllGameData і заповнює його поточним станом гри. Для цього використовуються два допоміжні методи: GetPlayerData() і GetWorldData(). GetPlayerData() збирає актуальні значення з різних систем гри. Зокрема, він отримує значення життєвих показників із PlayerStatus, координати гравця з playerBody, а також дані інвентарю з InventorySystem. Окремо формується масив позиції, який включає координати та ротацію, а також масив швидких слотів, який формується через метод GetQuickSlotcontent(). Цей метод проходить по списку слотів з EquipSystem, перевіряє наявність предметів у кожному слоті та формує масив назв об’єктів, що дозволяє коректно відновлювати префаби під час завантаження. Після формування об’єкта AllGameData він передається у метод SaveGameData , який визначає формат збереження. Якщо активовано JSON-збереження, викликається SaveDataToJson, інакше SaveDataTobinary. У випадку JSON система використовує JsonUtility.ToJson, який перетворює об’єкт у рядок, після чого він записується у файл через StreamWriter. Це дозволяє легко читати та

редактувати збереження, оскільки JSON є текстовим форматом. У випадку binary використовується BinaryFormatter, який серіалізує весь об’єкт у бінарний файл. Файли зберігаються у різних директоріях залежно від формату: JSON у Application.dataPath, a binary у Application.persistentDataPath.

Завантаження реалізується через метод LoadGameData, який симетрично визначає формат і викликає відповідний метод LoadDataFromJson або LoadDataFrombinary. В обох випадках відбувається десеріалізація файлу назад у об’єкт AllGameData. Далі метод LoadGame застосовує завантажені дані до ігрового світу. Він розділяє процес на два етапи: SetPlayerData і SetWorldData. Під час встановлення даних гравця система відновлює параметри здоров’я, голоду та спраги, після чого задає позицію і ротацію персонажа. CharacterController тимчасово вимикається, щоб уникнути конфліктів при телепортації, після чого позиція встановлюється, і контролер знову активується. Також очищається швидкість руху, щоб уникнути некоректної інерції після завантаження. Далі відновлюється інвентар шляхом проходження по масиву playerinventoryContent, де кожен елемент додається через InventorySystem. Швидкі слоти відновлюються шляхом інстанціювання префабів із Resources.Load і прив’язування їх до відповідних слотів у EquipSystem. Відновлення світу відбувається через SetWorldData. Система проходить по всіх об’єктах у WorldManager і видаляє ті, які були позначені як підібрані. Це дозволяє синхронізувати стан сцени з тим, що було збережено раніше. Після цього список pickedItems переприсвоюється в InventorySystem, що забезпечує узгодженість даних між інвентарем і світом. Процес запуску завантаженої гри відбувається через StartLoadedGame, який активує екран завантаження , блокує курсор і завантажує сцену. Після невеликої затримки через корутину DelayedLoading викликається LoadGame, що гарантує, що сцена повністю ініціалізована перед відновленням даних.

Окремо реалізовано систему слотів через класи LoadSlot і SaveSlot. LoadSlot відповідає за завантаження гри з конкретного слоту. У методі Update він постійно перевіряє, чи існує збереження через isSlotEmpty, і відповідно оновлює текст кнопки. При натисканні кнопки або запускається завантаження гри, або нічого не

відбувається, якщо слот порожній. SaveSlot, у свою чергу, реалізує логіку збереження. Якщо слот не порожній, одразу викликається SaveGameConfirmed, інакше показується попередження про перезапис. При підтвердженні користувачем викликається збереження, після чого генерується опис слота з датою і часом, який записується в PlayerPrefs, що дозволяє відображати інформацію про останнє збереження в меню. Метод DeselectButton, очищає вибір у EventSystem, щоб уникнути некоректного відображення UI після дій користувача.

У результаті вся система збереження побудована як модульна архітектура, де дані гравця і світу ізольовані, але синхронізуються через спільний контейнер AllGameData, а підтримка двох форматів збереження JSON і binary, забезпечуючи гнучкість і масштабованість системи.

4 ТЕСТУВАННЯ

Для перевірки реалізованих ігрових механік гри використовувалися декілька видів тестування. В основному задіяним було функціональне тестування, через те що головною метою даного процесу тестування є визначення коректності роботи механік гри відповідно до поставлених вимог. Оскільки більшість механік взаємодіють між собою, у проєкті також використовується інтеграційне тестування. Воно необхідне для перевірки правильності обміну даними між окремими механіками гри. Додатково у процесі використовувалися тестування ігрової логіки , яке дозволяє оцінити поведінку механік у реальному ігровому процесі. Крім цього, під час розробки використовувалося регресійне тестування. Після додавання нових механік або змін у коді повторно перевірялися вже реалізовані системи, щоб переконатися, що нові зміни не порушили існуючу функціональність гри. Для зручнішого перегляду тест кейси були розділені по механікам. Нижче можна побачити тест кейси до механік персонажа гравця (Таблиця 4.1).

Таблиця 4.1

Тест кейси механік персонажа гравця

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Рух вперед	Гравець знаходиться на землі	Натиснути W	Персонаж рухається вперед	Персонаж рухається вперед
Рух назад	Гравець знаходиться на землі	Натиснути S	Персонаж рухається назад	Персонаж рухається назад
Рух вліво, вправо	Гравець знаходиться на землі	Натиснути A Натиснути D	Персонаж рухається спочатку вліво а потім вправо	Персонаж рухається спочатку вліво а потім вправо
Стрибок на землі	Гравець стоїть на поверхні	Натиснути Space	Персонаж виконує стрибок	Персонаж виконує стрибок

Продовження таблиці 4.1

Тест кейси механік персонажа гравця

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Заборона подвійного стрибка	Персонаж у повітрі	Повторно натиснути Space	Другий стрибок не виконується	Другий стрибок не виконується
Дія гравітації	Персонаж у повітрі	Натиснути Space	Персонаж падає вниз	Персонаж падає вниз
Поворот камери по горизонталі	Камера активна	Рухати мишу по осі X	Камера повертається горизонтально	Камера повертається горизонтально
Поворот камери по вертикалі	Камера активна	Рухати мишу по осі Y	Камера нахиляється вертикально	Камера нахиляється вертикально
Блокування повороту камери при відкритих меню	Відкритий інвентар	Рухати мишу	Камера не рухається	Камера не рухається
Зменшення спраги	Гра активна	Очікувати 7 секунд	Спрага зменшується	Спрага зменшується
Зменшення голоду	Гравець рухається	Пройти Дистанцію 10 метрів	Голод зменшується	Голод зменшується
Отримання шкоди	HP > 0	Підійти до ворога. Отримати шкоду.	HP зменшується	HP зменшується
Смерть персонажа	HP > 0	Отримувати шкоду до настання смерті	Гравець вмирає	Гравець вмирає

Тестування механік збору ресурсів спрямоване на перевірку взаємодії гравця з предметами навколишнього середовища, системи підбору ресурсів, рубки дерев та оновлення стану ресурсів у реальному часі. Тест кейси до цих механік показані у таблиці (Таблиця 4.2).

Таблиця 4.2

Тест кейси механік збору ресурсів

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Виявлення предмета	Предмет у полі зору гравця	Навести приціл на предмет	Показується назва предмета поруч з точкою в центрі екрану	Показується назва предмета поруч з точкою в центрі екрану
Підбір предмета	Є місце в інвентарі та предмет у полі зору гравця	Навестись на предмет Натиснути E	Предмет додається до інвентарю	Предмет додається до інвентарю
Видалення предмета з мапи світу	Предмет підібрано	Почекати секунду	Підібраний об’єкт видаляється з мапи світу	Підібраний об’єкт видаляється з мапи світу
Переповнений інвентар	Інвентар повний	Підібрати предмет	Виводиться повідомлення про переповнений інвентар	Виводиться повідомлення про переповнений інвентар
Зменшення HP дерева	Дерево поруч з гравцем	Ударити сокирою по дереву	HP дерева зменшується	HP дерева зменшується
Зменшення голоду	Гравець рубає дерево	Виконати удари	Показник голоду зменшується	Показник голоду зменшується
Знищення дерева	Дерево поруч з гравцем	Рубати дерево до поки HP дерева не досягне нуля	Дерево видаляється З’являється пень на його місці	Дерево видаляється З’являється пень на його місці

При тестуванні системи крафту було створено тест-кейси, які охоплюють тільки програмну частину цієї механіки, їх ви можете побачити у таблиці нижче Таблиця 4.3).

Таблиця 4.3

Тест кейси системи крафту

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Крафт сокири	Є ресурси для крафту в інвентарі	Натиснути кнопку Craft у меню	Сокира додається до інвентарю	Сокира додається до інвентарю
Видалення ресурсів	Є ресурси для крафту в інвентарі	Створити сокиру	Ресурси видаляються з інвентарю	Ресурси видаляються з інвентарю
Недостатньо ресурсів для крафту	Ресурсів для крафту сокири не вистачає	Спробувати створити сокиру	Кнопка не реагує	Кнопка не реагує

Тестування системи будівництва проводилася для перевірки механік розміщення конструкцій та перевірки валідності побудови та роботи ghost-об’єктів. Тест кейси до механік будівництва можна побачити у таблиці (Таблиця 4.4).

Таблиця 4.4

Тест кейси механік будівництва

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Валідне місце	В системі будівництва гравець намагається збудувати підлогу на підходящому місці	Розмістити конструкцію	Конструкція має зелений колір і спокійно створюється	Конструкція має зелений колір і спокійно створюється
Невалідне місце	В системі будівництва гравець намагається збудувати підлогу на непідходящому місці	Розмістити конструкцію	Конструкція має червоний колір і не створюється	Конструкція має червоний колір і не створюється
Collider після встановлення	Будівництво завершене	Перевірити collision з колайдером	Collider активний і працює	Collider активний і працює

Продовження таблиці 4.4

Тест кейси механік будівництва

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Виділення selectedGhost	При будівництві гравець намагається збудувати підлогу поруч з іншою підлогою	Навести курсор на потенційне місце підлоги яке виглядає як полупрозорий квадрат	Цей полупрозорий елементи підсвічується	Цей полупрозорий елементи підсвічується

Для перевірки системи взаємодії з неігровими персонажами було розроблено тест-кейси, що охоплюють діалогову систему, прийняття та завершення квестів, перевірку предметів у інвентарі та видачу нагород. Тест кейси до цих механік можна побачити у таблиці нижче (Таблиця 4.5).

Таблиця 4.5

Тест кейси механік взаємодії з неігровими персонажами

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Тест показнику можливості заговорити	Гравець поруч з персонажем	Навестися прицілом на персонажа	Поруч з прицілом з’являється помітка можливості розмови	Поруч з прицілом з’являється помітка можливості розмови
Відкриття діалогу	Приціл на ведений на персонажа для розмови	Натиснути E	Відкривається меню діалогу	Відкривається меню діалогу
Розблокування курсора	Меню діалогу відкритий	Перемістити курсор	Курсор видимий та рухається	Курсор видимий та рухається
Перемикання реплік	Діалог активний	Натискати кнопку Next	Репліки змінюються	Репліки змінюються
Прийняття квесту	Є доступний для прийняття квест	Натиснути Accept	Квест видається гравцю	Квест видається гравцю

Продовження таблиці 4.5

Тест кейси механік взаємодії з неігровими персонажами

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Відхилення квесту	Є доступний для прийняття квест	Натиснути Decline	Квест не видається гравцю	Квест не видається гравцю
Перевірка предметів для квесту	Квест вимагає ресурси	При наявності потрібних ресурсів у інвентарі гравець починає розмову з персонажем у якого брав квест	Inventory перевіряється і квест завершується	Inventory перевіряється і квест завершується
Завершення квесту	Вимоги виконані	Поговорити з NPC	Квест завершується	Квест завершується
Видача нагороди	Квест завершено	Гравець завершає квест	Нагорода додається	Відповідає очікуваному результату

Тестування механіки ворогів спрямоване на перевірку системи виявлення гравця, навігації через NavMesh та переслідування з нанесення шкоди. Тест кейси до цієї механіки можна побачити в таблиці нижче(Таблиця 4.6).

Таблиця 4.6

Тест кейси механік ворогів

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Виявлення гравця	Гравець входить у зону бачення ворога	Зайти в зону зору ворога	Ворог бачить гравця	Ворог бачить гравця
Переслідування	Гравець виявлений	Відійти від ворога	Ворог рухається за гравцем	Ворог рухається за гравцем
Перехід в атаку	Гравець у зоні атаки	Підійти впритул до ворога	Ворог атакує	Ворог атакує

Продовження таблиці 4.6

Тест кейси механік ворогів

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Час між атаками	Ворог атакував	Залишатись поруч	Є затримка в 3 секунди між атаками	Є затримка в 3 секунди між атаками
Втрата цілі	Гравець виходить із зони зору ворога	Відійти на велику відстань від ворога	Ворог втрачає гравця із зору і зупиняється	Ворог втрачає гравця із зору і зупиняється

Для перевірки системи збереження та завантаження було створено тест-кейси, які охоплюють створення save-файлів, завантаження даних з файлу та роботу слотів збереження. Тест Кейси до цієї механіки можна побачити у таблиці (Таблиця 4.7).

Таблиця 4.7

Тест кейси механік збереження ігрового процесу

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Створення save файлу	Гра активна	Натиснути кнопку Save у меню паузи та обрати слот для збереження	Створюється save-файл	Створюється save-файл
Завантаження гри	Є save-файл	Натиснути Load у меню та обрати слот з файлом	Гра завантажується з збереженим прогресом	Гра завантажується з збереженим прогресом
Відновлення позиції	Завантажується гра	Завантажити save-файл	Позиція відновлюється з збереженого файлу	Позиція відновлюється з збереженого файлу

Продовження таблиці 4.7

Тест кейси механік збереження ігрового процесу

Назва тесту	Передумови	Кроки	Очікуваний результат	Отриманий результат
Видалення picked items	При завантаженні були підібрані предмети з мапи	Завантажити save-файл	Після завантаження предмет відсутній	Після завантаження предмет відсутній
Перезапис save-файлу у слоті	Слот зайнятий	Зберегти файл повторно	Дані перезаписуються	Дані перезаписуються

У результаті проведеного тестування встановлено, що всі основні ігрові механіки функціонують коректно відповідно до поставлених вимог. Функціональне та інтеграційне тестування підтвердили правильну роботу окремих систем і їх взаємодію між собою, а тестування ігрової логіки – стабільну поведінку механік у реальному ігровому процесі. Проведене регресійне тестування показало, що внесені зміни не порушують вже реалізовану функціональність. В кінці тестування всі не критичні помилки були усуненні.

ВИСНОВКИ

Підсумком виконання даної кваліфікаційної роботи стала розробка повноцінної відеогри. Вона належить до жанру виживання з інтегрованими RPG-елементами. Проєкт вдало поєднує в собі дослідження відкритого світу, безпосередньо механіки виживання, а також крафтинг, систему будівництва й комунікацію з неігровими персонажами. Як основний інструментарій для створення продукту використовувалися ігровий рушій Unity та мова C#.

Загалом під час роботи вдалося досягти таких результатів:

1. За результатами аналізу предметної області визначено ключові особливості поєднання жанру виживання та RPG, вивчено запити аудиторії та виділено сильні й слабкі сторони популярних аналогів. Це дозволило сформувати підсумковий перелік функціональних і нефункціональних вимог до проєкту.

2. Дослідження технологічного ринку дозволило обґрунтувати вибір технологій розробки, що довело доцільність використання рушія Unity разом із мовою C# та середовищем Visual Studio, а також підтверджено ефективність застосування принципів ООП для створення гнучкого коду.

3. На основі визначених функціональних і нефункціональних умов, стеку технологій для Backend-частини системи було створено комплексну ігрову логіку, включаючи контролер переміщення гравця, систему моніторингу показників життєдіяльності (здоров’я, голод, спрага) та розгалужену підсистему взаємодії з навколишнім середовищем для збору ресурсів, крафту і зведення будівель. Окрему увагу приділено механікам діалогів з NPC та розробці надійної системи збереження, яка гарантує стабільну фіксацію ігрового прогресу та динамічне оновлення стану світу залежно від дій користувача.

4. Завдяки проведеному комплексному тестуванню розробленого програмного продукту, яке охоплювало перевірку базових механік, стабільності системи збережень та коректності взаємодії з оточенням, було підтверджено працездатність усіх ігрових підсистем. Під час фінальних ітерацій тестування

критичних помилок чи дефектів у логіці виявлено не було, а знайдені маленькі помилки було усунено на фінальному етапі.

Робота пройшла апробацію на наукових конференціях, за результатами апробації опубліковано тези доповідей:

Барбак Д.О., Замрій І.В. Визначення вимог до 3D гри в жанрі survival з елементами RPG. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2025, С.468 -471.

Барбак Д.О., Замрій І.В. Визначення вимог до backend-частини 3D гри в жанрі виживання з елементами RPG. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2026, С.132 -135.

ПЕРЕЛІК ПОСИЛАНЬ

1. Adams, E., Dormans, J. Game Mechanics: Advanced Game Design. Berkeley CA: New Riders, 2012, 352 p.

2. Néd R. The art of survival design : a blueprint to survival game design beyond the genre. *Theseus*. URL: https://www.theseus.fi/handle/10024/872026.

3. Understanding the Impact of Perceived Challenge on Narrative Immersion in Video Games: The Role-Playing Game Genre as a Case Study / J. Domingues та ін. *MDPI*. URL: https://www.mdpi.com/2078-2489/15/6/294.

4. Bowman S. L. Finding the self in role-playing games: Weaving myth, narrative, and identity. *Taylor & Francis Online*. URL: https://www.tandfonline.com/doi/full/10.1080/25741136.2024.2324085#abstract.

5. Design and Construction of the Action Role-Playing Game: Supply Odyssey Using the Game Development Life Cycle Method. *Journal of Advances in Information and Industrial Technology*. URL: https://journal.ittelkom-sby.ac.id/jaiit/article/view/720.

6. Gallegos A. Minecraft Review. *IGN*. URL: https://www.ign.com/articles/2011/11/24/minecraft-review.

7. Hafer L. The Forest Review. *IGN*. URL: https://www.ign.com/articles/2018/05/10/the-forest-review-2.

8. Barbosa A. Rust Review: Life Is Fleeting. *Game Spot*. URL: https://www.gamespot.com/reviews/rust-review-life-is-fleeting/1900-6416858/.

9. Celebrating a Decade of ARK: Survival Evolved — 10 Things Happening Now in the ARK Universe. *IGN*. URL: https://www.ign.com/articles/celebrating-a-decade-of-ark-survival-evolved-10-things-happening-now-in-the-ark-universe.

10. Sliva M. Don’t Starve Review. *IGN*. URL: https://www.ign.com/articles/2013/05/02/dont-starve-review.

11. Singh S., Kaur A. Game Development using Unity Game Engine. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/abstract/document/10007155.

12. Doran J. Unity 2021 Shaders and Effects Cookbook, 4-th ed., Birmingham:

Packt, 2021, 738 p.

13. Unity, a 2D and 3D Game Engine. *Springer Nature Link*. URL: https://link.springer.com/rwe/10.1007/978-3-031-23161-2_536.

14. Programming in Unity. *Unity Documentation*. URL: https://docs.unity3d.com/Manual/scripting.html.

15. Thorn A. Pro Unity Game Development with C#, Berkeley: Apress, 2014, 397 p.

16. Abramowicz K., Borczuk P. Comparative analysis of the performance of Unity and Unreal Engine game engines in 3D games. *Journal of Computer Sciences Institute*. URL: https://ph.pollub.pl/index.php/jcsi/article/view/5473.

17. C# language documentation. *Microsoft Documentation*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/.

18. Visual Studio Documentation. *Microsoft Documentation*. URL: https://learn.microsoft.com/en-us/visualstudio/windows/?view=vs-2022

19. Nystrom, R. Game Programming Patterns, London: Genever Benning, 2014, 354 p.

20. Hocking J. Unity in Action: Multiplatform Game Development in C#, 3rd, Shelter Island: Manning, 2022, 416 p.

21. Block-Based Object-Oriented Programming / O. Allen та ін. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/abstract/document/9829246.

ДОДАТОК А ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка 3D відеогри "Whispers of the Wildwood" в жанрі виживання з елементами RPG. Спец-частина: розробка backend-частини системи.

Виконав студент 4 курсу
групи ПД-43
Барбак Дмитро Олександрович
Керівник роботи
завідувач кафедри ІПЗ, док.тех.наук, проф. Замрій Ірина Вікторівна

Київ – 2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: покращення ігрового досвіду користувача за рахунок поєднання геймплею жанру виживання з ключовими елементами RPG.

Об’єкт дослідження: отриманий гравцем досвід від процесу дослідження механік відеогри у жанрі виживання з елементами RPG.

Предмет дослідження: програмні засоби та методи створення геймплею відеогри у жанрі виживання з елементами RPG.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати особливості у відеоіграх жанру виживання з елементами RPG.
2. Провести аналіз існуючих популярних відеоігор жанру виживання.
3. Визначити функціональні та не функціональні вимоги до відеогри.
4. Оглянути засоби реалізації відеоігор жанру виживання з елементами RPG.
5. Сформулювати концепцію відеогри.
6. Реалізувати ігрові механіки: базові механіки гравця, збір ресурів, крафту, будівництва, взаємодії з неігровими персонажами, збереження та подальшого завантаження ігрового процесу та ворогів.
7. Провести тестування розроблених механік.

КОНЦЕПТ ГРИ

Сетинг - середньовічне фентезі з розумними тваринами які перейшли до феодального рівня розвитку соціума.

Сюжет - гравець бере на себе роль молодого бобра, який залишився без дому та сім'ї після загадкового катаклізму. Опинившись в незнайомій частині великого лісу, користувач має вижити та знайти відповідь на питання: Що саме сталося з великою плотиною?

Ключові механіки:
1) показники виживання (голод, спрага, здоров'я);
2) добування ресурсів;
3) будівництво бази;
4) взаємодія з неігровими персонажами;
5) квести.

Ключові особливості:
1. Поєднання жанру виживання з елементами жанру RPG.
2. Заздалегіть створена мапа світу з тематичними локаціями.

АНАЛІЗ АНАЛОГІВ

Критерій / Гра	Minecraft	The Forest	Rust	ARK: Survival Evolved	Don't Starve	Whispers of the Wildwood
Прості механіки збору ресурсів	+	-	+	+	-	+
Спрощена система крафту	-	-	+	+	-	+
Можливість створення бази	+	+	+	+	-	+
Інтуїтивні механіки виживання	+	+	+	-	-	+
Наявність квестової системи	-	-	+	-	+	+
Заздалегіть створена мапа світу	-	+	+	+	-	+
Наявність нейтральних неігрових персонажів	+	-	+	-	+	+
Наявність RPG елементу	-	+	+	+	+	+

5

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Переміщення гравця по заздалегіть створеній та стилізованій мапі ігрового світу.
2. Реалізація елементів виживання для гравця.
3. Система будівництва споруд гравцем.
4. Можливість зберігати та завантажувати ігровий процес.
5. Квестова система з неігровими персонажами.
6. Агресивно налаштовані до гравця неігрові персонажі.

Нефункціональні вимоги:

1. Переміщення гравця по заздалегіть створеній та стилізованій мапі ігрового світу. Сумістність з операційною системою Windows.
2. Забезпечення стабільного ігрового процесу з частотою кадрів 40-50к/с.
3. Мінімальні системні вимоги: Процесор: INTEL CORE I5-8400 або AMD RYZEN 3 3300 X RAM: 6 гб Відеокарта: NVIDIA GeForce GTX 1060 3GB або AMD Radeon RX 570 4GB.
4. Рекомендовані системні вимоги: Процесор: INTEL CORE I5-8700 чи AMD RYZEN 5 3600 X RAM: 8 гб Відеокарта: NVIDIA GeForce GTX 1080Ti or AMD Radeon RX 5700 XT.

6

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

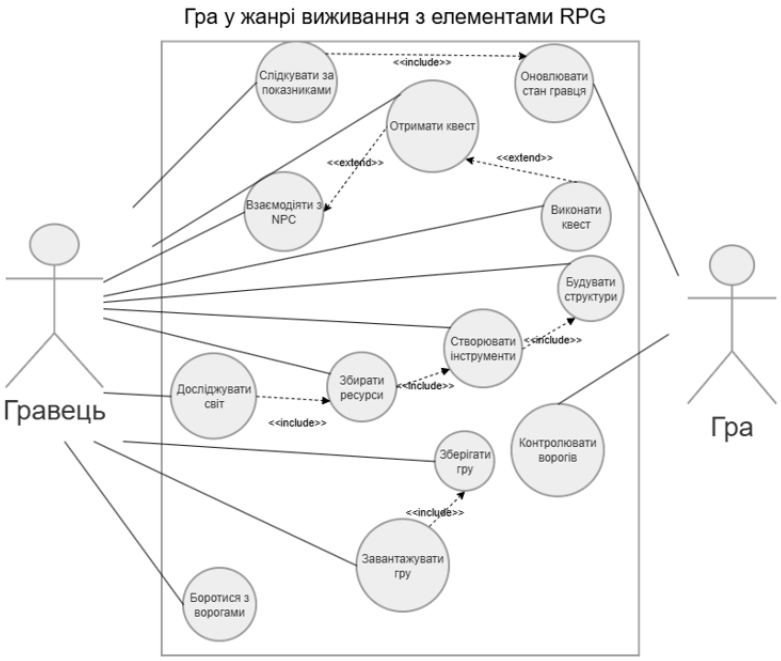
Ігровий рушій Unity

Мова програмування C#

Середовище розробки Visual Studio

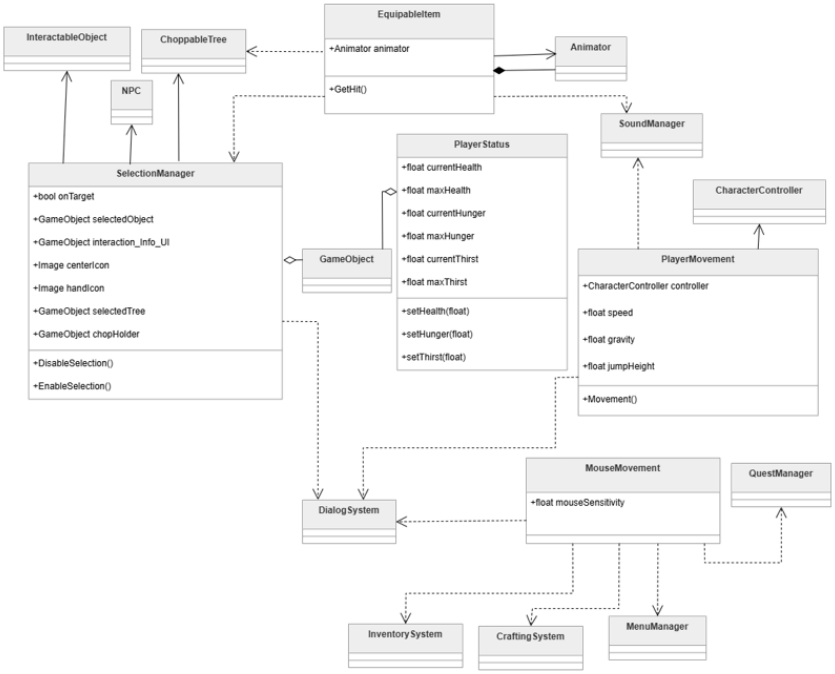
7

ДІАГРАМА ПРЕЦЕДЕНТІВ ВІДЕОГРИ



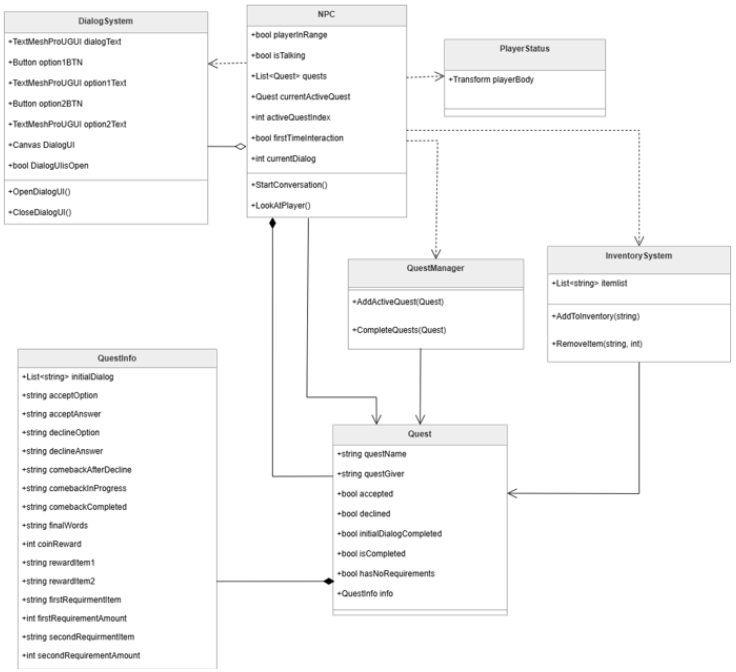
8

ДІАГРАМА КЛАСІВ ГРАВЦЯ



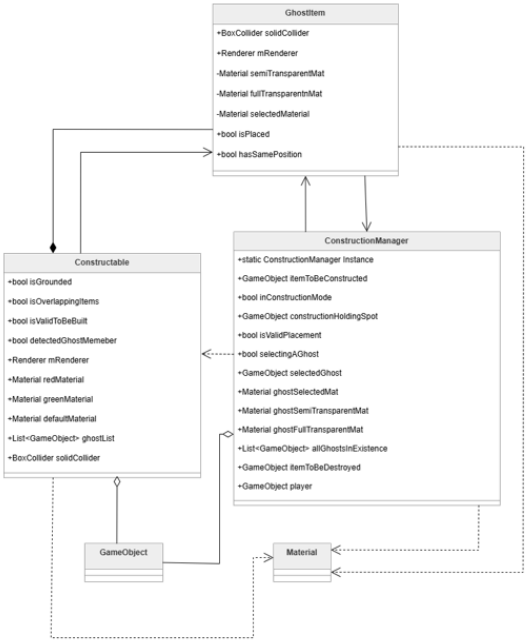
9

ДІАГРАМА КЛАСІВ СИСТЕМИ NPC



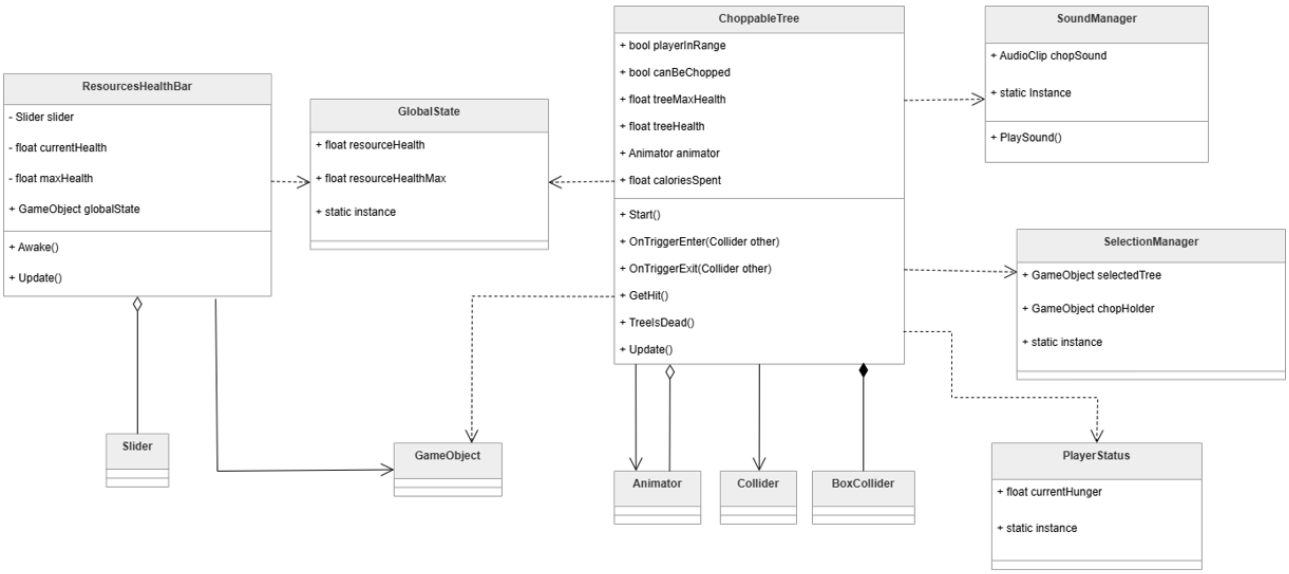
10

ДІАГРАМА КЛАСІВ СИСТЕМИ БУДІВНИЦТВА



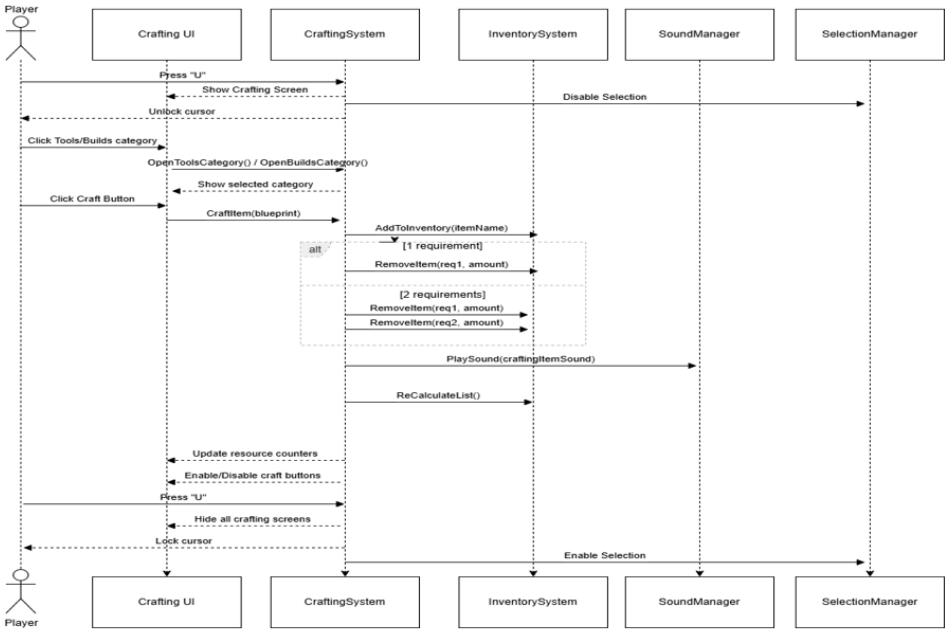
11

ДІАГРАМА КЛАСІВ СИСТЕМИ ЗБОРУ РЕСУРСІВ



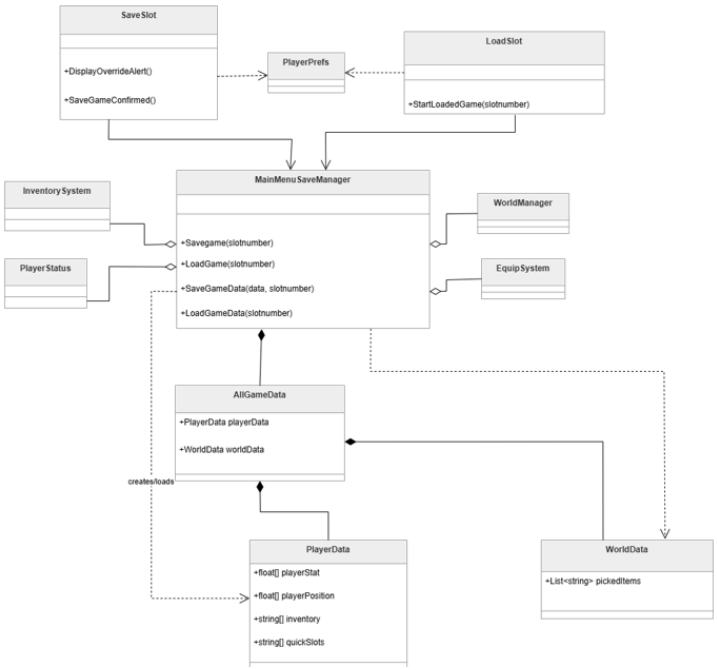
12

ДІАГРАМА ПОСЛІДОВНОСТІ СИСТЕМИ КРАФТУ



13

ДІАГРАМА КЛАСІВ СИСТЕМИ ЗБЕРЕЖЕННЯ ПРОЦЕСУ ГРИ



14

ДЕМОНТРАЦІЯ РОБОТИ ВІДЕОГРИ

15

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Барбак Д.О., Замрій І.В. Визначення вимог до 3D гри в жанрі survival з елементами RPG. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2025, С.468 -471.

2. Барбак Д.О., Замрій І.В. Визначення вимог до backend-частини 3D гри в жанрі виживання з елементами RPG. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2026, С.132 -135.

16

ВИСНОВКИ

1. На основі проведеного аналізу особливостей відеоігор жанру виживання з елементами RPG та дослідження існуючих популярних проєктів визначено ключові характеристики та підходи до реалізації ігрового процесу, та функціональні і нефункціональні вимоги до відеогри.
2. У результаті огляду сучасних засобів розробки відеоігор жанру виживання з елементами RPG обрано інструменти та технології реалізації відеогри: Відеоігровий рушій Unity, мова програмування C#, середовище розробки Visual Studio.
3. На основі визначених функціональних і нефункціональних умов, стеку технологій для Backend-частини системи було створено комплексну ігрову логіку, зокрема механіки гравця, збору ресурсів, крафтингу, будівництва, взаємодії з неігровими персонажами, а також систему збереження та завантаження ігрового прогресу та поведінки ворогів.
4. Після проведеного тестування у вигляді 46 юніт тестів, було підтверджено коректну роботу механік відеогри.

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

Вихідний код файлу ConstructionManager

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


public class ConstructionManager : MonoBehaviour
{
    public static ConstructionManager Instance { get; set; }
    public GameObject itemToBeConstructed;
    public bool inConstructionMode = false;
    public GameObject constructionHoldingSpot;
    public bool isValidPlacement;
    public bool selectingAGhost;
    public GameObject selectedGhost;
    // Materials we store as refereces for the ghosts
    public Material ghostSelectedMat;
    public Material ghostSemiTransparentMat;
    public Material ghostFullTransparentMat;
    // We keep a reference to all ghosts currently in our world,
    // so the manager can monitor them for various operations
    public List<GameObject> allGhostsInExistence = new
List<GameObject>();
    public GameObject itemToBeDestroyed;
    public GameObject player;
    private void Awake()
    {
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
        }
        else
        {
            Instance = this;
        }
    }
    public void ActivateConstructionPlacement(string
itemToConstruct)
    {
        GameObject item =
Instantiate(Resources.Load<GameObject>(itemToConstruct));
        //change the name of the gameobject so it will not be
(clone)
        item.name = itemToConstruct;
      item.transform.SetParent(constructionHoldingSpot.transform,
false);
        itemToBeConstructed = item;
        itemToBeConstructed.gameObject.tag =
"activeConstructable";
        // Disabling the non-trigger collider so our mouse can cast
a ray

itemToBeConstructed.GetComponent<Constructable>().solidC
ollider.enabled = false;


        // Actiavting Construction mode
        inConstructionMode = true;
    }
    private void GetAllGhosts(GameObject
itemToBeConstructed)
    {
        List<GameObject> ghostlist =
itemToBeConstructed.gameObject.GetComponent<Constructa
ble>().ghostList;
        foreach (GameObject ghost in ghostlist)
        {
            Debug.Log(ghost);
            allGhostsInExistence.Add(ghost);
        }
    }
    private void PerformGhostDeletionScan()
    {
        foreach (GameObject ghost in allGhostsInExistence)
        {
            if (ghost != null)
            {
                if
(ghost.GetComponent<GhostItem>().hasSamePosition ==
false) // if we did not already add a flag
```

```
            {
                foreach (GameObject ghostX in allGhostsInExistence)
                {
                    // First we check that it is not the same object
                    if (ghost.gameObject != ghostX.gameObject)
                    {
                        // If its not the same object but they have the same position
                        if (XPositionToAccurateFloat(ghost) == XPositionToAccurateFloat(ghostX) && ZPositionToAccurateFloat(ghost) == ZPositionToAccurateFloat(ghostX))
                        {
                            if (ghost != null && ghostX != null)
                            {
                                // setting the flag

ghostX.GetComponent<GhostItem>().hasSamePosition = true;
                                break;
                            }}}}}}}
        foreach (GameObject ghost in allGhostsInExistence)
        {
            if (ghost != null)
            {
                if (ghost.GetComponent<GhostItem>().hasSamePosition)
                {
                    DestroyImmediate(ghost);
                }
            }

        }
    }

    private float XPositionToAccurateFloat(GameObject ghost)
    {
        if (ghost != null)
        {
            // Turning the position to a 2 decimal rounded float
            Vector3 targetPosition = ghost.gameObject.transform.position;
            float pos = targetPosition.x;
            float xFloat = Mathf.Round(pos * 100f) / 100f;
            return xFloat;
        }
        return 0;
    }

    private float ZPositionToAccurateFloat(GameObject ghost)
    {

        if (ghost != null)
        {
            // Turning the position to a 2 decimal rounded float
            Vector3 targetPosition = ghost.gameObject.transform.position;
            float pos = targetPosition.z;
            float zFloat = Mathf.Round(pos * 100f) / 100f;
            return zFloat;

        }
        return 0;
    }

    private void Update()
    {

        if (itemToBeConstructed != null && inConstructionMode)
        {
            if (itemToBeConstructed.name == "FoundationModel")
            {
                if (CheckValidConstructionPosition())
                {
                    isValidPlacement = true;

itemToBeConstructed.GetComponent<Constructable>().SetValidColor();
                }
                else
                {
                    isValidPlacement = false;
```

```
itemToBeConstructed.GetComponent<Constructable>().SetInvalidColor();
            }
        }


        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;
        if (Physics.Raycast(ray, out hit))
        {
            var selectionTransform = hit.transform;
            if (selectionTransform.gameObject.CompareTag("ghost") && itemToBeConstructed.name == "FoundationModel")
            {
                itemToBeConstructed.SetActive(false);
                selectingAGhost = true;
                selectedGhost = selectionTransform.gameObject;
            }
            else if (selectionTransform.gameObject.CompareTag("wallGhost") && itemToBeConstructed.name == "WallModel")
            {
                itemToBeConstructed.SetActive(false);
                selectingAGhost = true;
                selectedGhost = selectionTransform.gameObject;
            }
            else
            {
                itemToBeConstructed.SetActive(true);
                selectedGhost = null;
                selectingAGhost = false;
            }

        }
    }

    // Left Mouse Click to Place item
    if (Input.GetMouseButtonDown(0) && inConstructionMode)
    {
        if (isValidPlacement && selectedGhost == false && itemToBeConstructed.name == "FoundationModel") // We don't want the freestyle to be triggered when we select a ghost.
        {
            PlaceItemFreeStyle();
            DestroyItem(itemToBeDestroyed);
        }

        if (selectingAGhost)
        {
            PlaceItemInGhostPosition(selectedGhost);
            DestroyItem(itemToBeDestroyed);
        }
    }

    if (Input.GetKeyDown(KeyCode.X) )
    {    // Left Mouse Button
        itemToBeDestroyed.SetActive(true);
        itemToBeDestroyed = null;
        DestroyItem(itemToBeConstructed);
        itemToBeConstructed = null;
        inConstructionMode = false;


    }
  }
  void DestroyItem(GameObject item)
  {
    DestroyImmediate(item);
    InventorySystem.Instance.ReCalculateList();
    CraftingSystem.instance.RefreshNeededItems();
  }
  private void PlaceItemInGhostPosition(GameObject copyOfGhost)
  {

    Vector3 ghostPosition = copyOfGhost.transform.position;
    Quaternion ghostRotation = copyOfGhost.transform.rotation;

    selectedGhost.gameObject.SetActive(false);
```

```
        // Setting the item to be active again (after we disabled it in the ray cast)
        itemToBeConstructed.gameObject.SetActive(true);
        // Setting the parent to be the root of our scene
itemToBeConstructed.transform.SetParent(transform.parent.transform.parent, true);
        var randomOffset = UnityEngine.Random.Range(0.01f,0.03f);
        itemToBeConstructed.transform.position = new Vector3(ghostPosition.x,ghostPosition.y,ghostPosition.z+randomOffset);
        itemToBeConstructed.transform.rotation = ghostRotation;
        // Enabling back the solider collider that we disabled earlier
itemToBeConstructed.GetComponent<Constructable>().solidCollider.enabled = true;
        // Setting the default color/material
itemToBeConstructed.GetComponent<Constructable>().SetDefaultColor();
        if (itemToBeConstructed.name == "FoundationModel")
        {
            // Making the Ghost Children to no longer be children of this item
itemToBeConstructed.GetComponent<Constructable>().ExtractGhostMembers();
            itemToBeConstructed.tag = "placedFoundation";
            //Adding all the ghosts of this item into the manager's ghost bank
            GetAllGhosts(itemToBeConstructed);
            PerformGhostDeletionScan();
        }
        else
        {
            itemToBeConstructed.tag = "placedWall";
            DestroyItem(selectedGhost);
        }
        itemToBeConstructed = null;
        inConstructionMode = false;
    }
    private void PlaceItemFreeStyle()
    {
        // Setting the parent to be the root of our scene
itemToBeConstructed.transform.SetParent(transform.parent.transform.parent, true);
        // Making the Ghost Children to no longer be children of this item
itemToBeConstructed.GetComponent<Constructable>().ExtractGhostMembers();
        // Setting the default color/material
itemToBeConstructed.GetComponent<Constructable>().SetDefaultColor();
        itemToBeConstructed.tag = "placedFoundation";
itemToBeConstructed.GetComponent<Constructable>().enabled = false;
        // Enabling back the solider collider that we disabled earlier
itemToBeConstructed.GetComponent<Constructable>().solidCollider.enabled = true;
        //Adding all the ghosts of this item into the manager's ghost bank
        GetAllGhosts(itemToBeConstructed);
        PerformGhostDeletionScan();
        itemToBeConstructed = null;
        inConstructionMode = false;
    }
```

```
    private bool CheckValidConstructionPosition()
    {
        if (itemToBeConstructed != null)
        {
            return
itemToBeConstructed.GetComponent<Constructable>().isVali
dToBeBuilt;
        }
        return false;
    }
}
```

Вихідний код файлу CraftingSystem

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

public class CraftingSystem : MonoBehaviour
{
    public GameObject craftingScreenUI;
    public GameObject toolScreenUI;
    public GameObject buildsScreenUI;
    public List<string> inventoryitemList = new List<string>();
    //Катенорії
    Button toolsBTN;
    Button buildsBTN;
    //Крафт
    Button craftAxeBTN;
    Button craftWallBTN;
    Button craftFoundationBTN;
    //Требования
    Text AxeReq1;
    Text AxeReq2;
    Text WallReq;
    Text FoundationReq;
    public bool isOpen;
    //Blueprints
    public Blueprint AxeBL = new Blueprint("Axe", 2, "Stone",
3, "Stick", 2);
    public Blueprint WallBL = new Blueprint("Wall", 1, "Log",
2, "", 0);
    public Blueprint FoundationBL = new
Blueprint("Foundation", 1, "Log", 4, "", 0);
    public static CraftingSystem instance { get; set; }

    private void Awake()
    {
        if (instance != null && instance != this)
        {
            Destroy(gameObject);
        }
        else
        {
            instance = this;
        }
    }
    // Start is called before the first frame update
    void Start()
    {
        isOpen = false;
        toolsBTN=
craftingScreenUI.transform.Find("ToolsButton").GetCompone
nt<Button>();
        toolsBTN.onClick.AddListener(delegate {
OpenToolsCategory(); });
        buildsBTN =
craftingScreenUI.transform.Find("BuildsButton").GetCompone
nt<Button>();
        buildsBTN.onClick.AddListener(delegate {
OpenBuildsCategory(); });
        //Axe
        AxeReq1 =
toolScreenUI.transform.Find("Axe").transform.Find("Rec1").G
etComponent<Text>();
        AxeReq2 =
toolScreenUI.transform.Find("Axe").transform.Find("Rec2").G
etComponent<Text>();
```

```
        craftAxeBTN =
toolScreenUI.transform.Find("Axe").transform.Find("Button").
GetComponent<Button> ();

        craftAxeBTN.onClick.AddListener(delegate {
CraftItem(AxeBL); });

        //Wall

        WallReq =
buildsScreenUI.transform.Find("Wall").transform.Find("Rec1"
).GetComponent <Text>();

        craftWallBTN =
buildsScreenUI.transform.Find("Wall").transform.Find("Button
").GetComponent<Button>() ;

        craftWallBTN.onClick.AddListener(delegate {
CraftItem(WallBL); });

        //Foundation

        FoundationReq =
buildsScreenUI.transform.Find("Foundation").transform.Find("
Rec1").GetComponent<Text>();

        craftFoundationBTN=
buildsScreenUI.transform.Find("Foundation").transform.Find("
Button").GetComponent<Button>();

        craftFoundationBTN.onClick.AddListener(delegate {
CraftItem(FoundationBL); });

    }

    private void OpenBuildsCategory()

    {

        craftingScreenUI.SetActive(false);

        buildsScreenUI.SetActive(true);

    }

    void OpenToolsCategory()

    {

        craftingScreenUI.SetActive(false);

        toolScreenUI.SetActive(true);

    }

    void CraftItem(Blueprint blueprintToCraft)

    {


InventorySystem.Instance.AddToInventory(blueprintToCraft.It
emname);

        if (blueprintToCraft.numOfRequirements == 1)

        {


InventorySystem.Instance.RemoveItem(blueprintToCraft.Req1,
blueprintToCraft.Req1amount);

        }

        else if(blueprintToCraft.numOfRequirements == 2)

        {


InventorySystem.Instance.RemoveItem(blueprintToCraft.Req1,
blueprintToCraft.Req1amount);


InventorySystem.Instance.RemoveItem(blueprintToCraft.Req2,
blueprintToCraft.Req2amount);

        }


SoundManager.Instance.PlaySound(SoundManager.Instance.cr
aftingItemSound);

        StartCoroutine(calculate());


    }

    public IEnumerator calculate()

    {

        yield return 0;

        InventorySystem.Instance.ReCalculateList();

        RefreshNeededItems();

    }

    // Update is called once per frame

    void Update()

    {


        if (Input.GetKeyDown(KeyCode.U) && !isOpen )

        {


            craftingScreenUI.SetActive(true);

            Cursor.lockState = CursorLockMode.None;

            Cursor.visible = true;

            SelectionManager.instance.DisableSelection();


SelectionManager.instance.GetComponent<SelectionManager
>().enabled = false;

            isOpen = true;


        }

        else if (Input.GetKeyDown(KeyCode.U) && isOpen)

        {

            craftingScreenUI.SetActive(false);

            toolScreenUI.SetActive(false) ;

            buildsScreenUI.SetActive(false);

            if (!InventorySystem.Instance.isOpen)
```

```
        {
            Cursor.lockState = CursorLockMode.Locked;
            Cursor.visible = false;
            SelectionManager.instance.EnableSelection();

SelectionManager.instance.GetComponent<SelectionManager
>().enabled = true;
        }
        isOpen = false;
    }
  }
  public void RefreshNeededItems()
  {
    int stone_count = 0;
    int stick_count = 0;
    int log_count = 0;

    inventoryitemList = InventorySystem.Instance.itemlist;
    foreach(string itemName in inventoryitemList)
    {
      switch(itemName)
      {
        case "Stone":
          stone_count+=1;
          break;

        case "Stick":
          stick_count+=1;
          break;

        case "Log":
          log_count+=1;
          break;
      }
    }
    //AXE//
    AxeReq1.text = "3 Stone[" + stone_count + "]";
    AxeReq2.text = "2 Stick[" + stick_count + "]";

    if(stone_count>=3 && stick_count>=2)
    {
      craftAxeBTN.gameObject.SetActive(true);
    }
    else
    {
      craftAxeBTN.gameObject.SetActive(false);
    }
    //Wall//
    WallReq.text = "2 Log[" + log_count + "]";

    if (log_count >= 2)
    {
      craftWallBTN.gameObject.SetActive(true);
    }
    else
    {
      craftWallBTN.gameObject.SetActive(false);
    }
    //Foundation//
    FoundationReq.text = "4 Log[" + log_count + "]";

    if (log_count >= 4)
    {
      craftFoundationBTN.gameObject.SetActive(true);
    }
    else
    {
      craftFoundationBTN.gameObject.SetActive(false);
    }
  }
}
```

Вихідний код файлу InteractableObject

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

```
public class InteractableObject : MonoBehaviour
{
   public bool playerInRange;
   public string ItemName;

   public string GetItemName()
   {
      return ItemName;
   }

   void Update()
   {
     if(Input.GetKeyDown(KeyCode.E) && playerInRange &&
SelectionManager.instance.onTarget&&
SelectionManager.instance.selectedObject == gameObject)
      {
         if (!InventorySystem.Instance.CheckifFull())
         {

InventorySystem.Instance.AddToInventory(ItemName);
            Destroy(gameObject);

SoundManager.Instance.PlaySound(SoundManager.Instance.pi
ckupSound);

InventorySystem.Instance.itemsPicked.Add(gameObject.name)
;
         }
         else
         {
            Debug.Log("Inventory Full");
         }
      }
   }
   private void OnTriggerEnter(Collider other)
   {
      if (other.CompareTag("Player"))
      {
         playerInRange = true;
      }
   }
   private void OnTriggerExit(Collider other)
   {
      if (other.CompareTag("Player"))
      {
         playerInRange = false;
      }

   }
}
```

Вихідний код файлу NPC

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

public class NPC : MonoBehaviour

{
   public bool playerInRange;
   public bool isTalking;
   TextMeshProUGUI npcDialogText;
   Button optionButton1;
   TextMeshProUGUI optionButton1Text;
   Button optionButton2;
   TextMeshProUGUI optionButton2Text;

   public List<Quest> quests;
   public Quest currentActiveQuest = null;
   public int activeQuestIndex = 0;
   public bool firstTimeInteraction = true;
   public int currentDialog;

   private void Start()
   {
      npcDialogText = DialogSystem.instance.dialogText;
```

```
        optionButton1 = DialogSystem.instance.option1BTN;
        optionButton1Text = DialogSystem.instance.option1Text;
        optionButton2 = DialogSystem.instance.option2BTN;
        optionButton2Text = DialogSystem.instance.option2Text;
    }
    public void StartConversation()
    {
        LookAtPlayer();
        isTalking = true;
        if (firstTimeInteraction)
        {
            firstTimeInteraction = false;
            currentActiveQuest = quests[activeQuestIndex];
            StartQuestInitialDialog();
            currentDialog = 0;
        }
        else
        {



            if (currentActiveQuest.declined)
            {

                DialogSystem.instance.OpenDialogUI();


                npcDialogText.text =
currentActiveQuest.info.comebackAfterDecline;


                SetAcceptAndDeclineOptions();
            }


            if (currentActiveQuest.accepted &&
currentActiveQuest.isCompleted == false)
            {
                if (AreQuestRequirmentsCompleted())
                {


                    SubmitRequiredItems();


                    DialogSystem.instance.OpenDialogUI();


                    npcDialogText.text =
currentActiveQuest.info.comebackCompleted;


                    optionButton1Text.text = "[Take Reward]";
                    optionButton1.onClick.RemoveAllListeners();
                    optionButton1.onClick.AddListener(() => {
                        ReceiveRewardAndCompleteQuest();
                    });
                }
                else
                {
                    DialogSystem.instance.OpenDialogUI();


                    npcDialogText.text =
currentActiveQuest.info.comebackInProgress;


                    optionButton1Text.text = "[Close]";
                    optionButton1.onClick.RemoveAllListeners();
                    optionButton1.onClick.AddListener(() => {
                        DialogSystem.instance.CloseDialogUI();
                        isTalking = false;
                    });
                }
            }


            if (currentActiveQuest.isCompleted == true)
            {
                DialogSystem.instance.OpenDialogUI();


                npcDialogText.text =
currentActiveQuest.info.finalWords;


                optionButton1Text.text = "[Close]";
                optionButton1.onClick.RemoveAllListeners();
                optionButton1.onClick.AddListener(() => {
                    DialogSystem.instance.CloseDialogUI();
                    isTalking = false;
                });
            }
```

```
        if (currentActiveQuest.initialDialogCompleted == false)
        {
            StartQuestInitialDialog();
        }

    }

}

private void SubmitRequiredItems()
{
    string firstRequiredItem = currentActiveQuest.info.firstRequirmentItem;
    int firstRequiredAmount = currentActiveQuest.info.firstRequirementAmount;
    if (firstRequiredItem != "")
    {
        InventorySystem.Instance.RemoveItem(firstRequiredItem, firstRequiredAmount);
    }
    string secondtRequiredItem = currentActiveQuest.info.secondRequirmentItem;
    int secondRequiredAmount = currentActiveQuest.info.secondRequirementAmount;
    if (firstRequiredItem != "")
    {
        InventorySystem.Instance.RemoveItem(secondtRequiredItem, secondRequiredAmount);
    }

}

private bool AreQuestRequirmentsCompleted()
{
    print("Checking Requirments");

    string firstRequiredItem = currentActiveQuest.info.firstRequirmentItem;
    int firstRequiredAmount = currentActiveQuest.info.firstRequirementAmount;
    var firstItemCounter = 0;

    foreach (string item in InventorySystem.Instance.itemlist)
    {
        if (item == firstRequiredItem)
        {
            firstItemCounter++;
        }
    }

    string secondRequiredItem = currentActiveQuest.info.secondRequirmentItem;
    int secondRequiredAmount = currentActiveQuest.info.secondRequirementAmount;

    var secondItemCounter = 0;

    foreach (string item in InventorySystem.Instance.itemlist)
    {
        if (item == secondRequiredItem)
        {
            secondItemCounter++;
        }
    }

    if (firstItemCounter >= firstRequiredAmount && secondItemCounter >= secondRequiredAmount)
    {
        return true;
    }
    else
    {
        return false;
    }
}

private void StartQuestInitialDialog()
{
    DialogSystem.instance.OpenDialogUI();
```

```
        npcDialogText.text =
currentActiveQuest.info.initialDialog[currentDialog];
        optionButton1Text.text = "Next";
        optionButton1.onClick.RemoveAllListeners();
        optionButton1.onClick.AddListener(() => {
            currentDialog++;
            CheckIfDialogDone();
        });

        optionButton2.gameObject.SetActive(false);
    }

    private void CheckIfDialogDone()
    {
        if (currentDialog ==
currentActiveQuest.info.initialDialog.Count - 1)
        {
            npcDialogText.text =
currentActiveQuest.info.initialDialog[currentDialog];

            currentActiveQuest.initialDialogCompleted = true;

            SetAcceptAndDeclineOptions();
        }
        else
        {
            npcDialogText.text =
currentActiveQuest.info.initialDialog[currentDialog];

            optionButton1Text.text = "Next";
            optionButton1.onClick.RemoveAllListeners();
            optionButton1.onClick.AddListener(() => {
                currentDialog++;
                CheckIfDialogDone();
            });
        }
    }

    private void SetAcceptAndDeclineOptions()
    {
        optionButton1Text.text =
currentActiveQuest.info.acceptOption;
        optionButton1.onClick.RemoveAllListeners();
        optionButton1.onClick.AddListener(() => {
            AcceptedQuest();
        });

        optionButton2.gameObject.SetActive(true);
        optionButton2Text.text =
currentActiveQuest.info.declineOption;
        optionButton2.onClick.RemoveAllListeners();
        optionButton2.onClick.AddListener(() => {
            DeclinedQuest();
        });
    }

    private void DeclinedQuest()
    {
        currentActiveQuest.declined = true;

        npcDialogText.text =
currentActiveQuest.info.declineAnswer;
        CloseDialogUI();
    }

    private void AcceptedQuest()
    {

QuestManager.Instance.AddActiveQuest(currentActiveQuest);
        currentActiveQuest.accepted = true;
        currentActiveQuest.declined = false;

        if (currentActiveQuest.hasNoRequirements)
        {
            npcDialogText.text =
currentActiveQuest.info.comebackCompleted;
            optionButton1Text.text = "[Take Reward]";
            optionButton1.onClick.RemoveAllListeners();
            optionButton1.onClick.AddListener(() => {
                ReceiveRewardAndCompleteQuest();
            });
            optionButton2.gameObject.SetActive(false);
        }
```

```
        else
        {
            npcDialogText.text =
currentActiveQuest.info.acceptAnswer;
            CloseDialogUI();
        }



    }

    private void ReceiveRewardAndCompleteQuest()
    {

QuestManager.Instance.CompleteQuests(currentActiveQuest);
        currentActiveQuest.isCompleted = true;

        var coinsRecieved = currentActiveQuest.info.coinReward;
        print("You recieved " + coinsRecieved + " gold coins");

        if (currentActiveQuest.info.rewardItem1 != "")
        {

InventorySystem.Instance.AddToInventory(currentActiveQuest
.info.rewardItem1);
        }

        if (currentActiveQuest.info.rewardItem2 != "")
        {

InventorySystem.Instance.AddToInventory(currentActiveQuest
.info.rewardItem2);
        }

        activeQuestIndex++;

        // Start Next Quest
        if (activeQuestIndex < quests.Count)
        {
            currentActiveQuest = quests[activeQuestIndex];
            currentDialog = 0;
            DialogSystem.instance.CloseDialogUI();
            isTalking = false;
        }
        else
        {
            DialogSystem.instance.CloseDialogUI();
            isTalking = false;
            print("No more quests");
        }

    }

    private void CloseDialogUI()
    {
        optionButton1Text.text = "[Close]";
        optionButton1.onClick.RemoveAllListeners();
        optionButton1.onClick.AddListener(() => {
            DialogSystem.instance.CloseDialogUI();
            isTalking = false;
        });
        optionButton2.gameObject.SetActive(false);
    }

    public void LookAtPlayer()
    {
        var player = PlayerStatus.instance.playerBody.transform;
        Vector3 direction = player.position - transform.position;
        transform.rotation = Quaternion.LookRotation(direction);

        var yRotation = transform.eulerAngles.y;
        transform.rotation = Quaternion.Euler(0, yRotation, 0);

    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            playerInRange = true;
        }
```

```
    }
    private void OnTriggerExit(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            playerInRange = false;
        }

    }
}
```

Вихідний код файлу PlayerMovement

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

public class PlayerMovement : MonoBehaviour
{
    public CharacterController controller;

    public float speed = 12f;
    public float gravity = -9.81f * 2;
    public float jumpHeight = 3f;

    public Transform groundCheck;
    public float groundDistance = 0.4f;
    public LayerMask groundMask;

    public Vector3 velocity;

    bool isGrounded;
    bool isMoving;
    public Vector3 lastPotition = new Vector3(0f, 0f, 0f);

    // Update is called once per frame
    void Update()
    {
        if (DialogSystem.instance.DialogUIisOpen == false)
        {
            Movement();
        }
    }
    public void Movement()
    {
        //checking if we hit the ground to reset our falling velocity, otherwise we will fall faster the next time
        isGrounded = Physics.CheckSphere(groundCheck.position, groundDistance, groundMask);

        if (isGrounded && velocity.y < 0)
        {
            velocity.y = -2f;
        }

        float x = Input.GetAxis("Horizontal");
        float z = Input.GetAxis("Vertical");

        //right is the red Axis, foward is the blue axis
        Vector3 move = transform.right * x + transform.forward * z;

        controller.Move(move * speed * Time.deltaTime);

        //check if the player is on the ground so he can jump
        if (Input.GetButtonDown("Jump") && isGrounded)
        {
            //the equation for jumping
            velocity.y = Mathf.Sqrt(jumpHeight * -2f * gravity);
        }

        velocity.y += gravity * Time.deltaTime;

        controller.Move(velocity * Time.deltaTime);

        if (lastPotition != gameObject.transform.position)
        {
            isMoving = true;

SoundManager.Instance.PlaySound(SoundManager.Instance.grassWalkSound);
        }
```

```
        else
        {
            isMoving = false;
            SoundManager.Instance.grassWalkSound.Stop();
        }
        lastPotition = gameObject.transform.position;
    }
}
```

Вихідний код файлу PlayerStatus

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

public class PlayerStatus : MonoBehaviour
{
    public static PlayerStatus instance { get; set; }

    public float currentHealth;
    public float maxHealth;

    public float currentHunger;
    public float maxHunger;

    float distancrTraveled = 0;
    Vector3 lastPosition;
    public GameObject playerBody;

    public float currentThirst;
    public float maxThirst;
    public bool ThirstActive;
    private void Awake()
    {
        if (instance != null && instance != this)
        {
            Destroy(gameObject);
        }
        else
        {
            instance = this;
        }
    }

    private void Start()
    {
        currentHealth = maxHealth;
        currentHunger = maxHunger;
        currentThirst = maxThirst;
        StartCoroutine(Thirst());
    }
    IEnumerator Thirst()
    {
        while (true)
        {
            currentThirst -= 1;
            yield return new WaitForSeconds(7);
        }
    }
    // Update is called once per frame
    //Tests
    void Update()
    {
        distancrTraveled +=
Vector3.Distance(playerBody.transform.position, lastPosition);
        lastPosition = playerBody.transform.position;
        if (distancrTraveled >=5)
        {
            distancrTraveled = 0;
            currentHunger -= 1;
        }



        if(Input.GetKeyDown(KeyCode.N))
        {
            currentHealth -= 10;
        }
    }
    public void setHealth(float newHealth)
    {
```

```
        currentHealth = newHealth;
    }
    public void setHunger(float newHunger)
    {
        currentHunger = newHunger;
    }
    public void setThirst(float newThirst)
    {
        currentThirst = newThirst;
    }
    public void TakeDamage(float amount)
    {
        currentHealth -= amount;
        currentHealth = Mathf.Clamp(currentHealth, 0, maxHealth);

        Debug.Log("Player Health: " + currentHealth);

        if (currentHealth <= 0)
        {
            Debug.Log("Player died");
        }
    }
}
```

Вихідний код файлу MainMenuSaveManager

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using UnityEngine;
using UnityEngine.SceneManagement;
using static UnityEngine.UIElements.UxmlAttributeDescription;


public class MainMenuSaveManager : MonoBehaviour
{
    public static MainMenuSaveManager instance { get; set; }
    private void Awake()
    {
        if (instance != null && instance != this)
        {
            Destroy(gameObject);
        }
        else
        {
            instance = this;
        }
        DontDestroyOnLoad(gameObject);
    }


    //JsonSavePath(Project)
    string jsonPathProject;
    //JsonSavePath(External)
    string jsonPath;
    //BinarySavePath
    string binaryPath;


    string fileName = "SaveGame";


    public bool isSavingtoJson;
    public bool isLoading;
    public Canvas loadingScreen;
    private void Start()
    {
        jsonPathProject = Application.dataPath + Path.AltDirectorySeparatorChar;
        jsonPath = Application.persistentDataPath + Path.AltDirectorySeparatorChar;
        binaryPath = Application.persistentDataPath+ Path.DirectorySeparatorChar;
    }
    #region Saving


    public void Savegame(int slotnumber)
    {
        AllGameData data = new AllGameData();
        data.playerData = GetPlayerData();
        data.worldData = GetWorldData();
        SaveGameData(data, slotnumber);
```

```
    }

    private WorldData GetWorldData()
    {
        List<string> itemsPicked = InventorySystem.Instance.itemsPicked;
        return new WorldData(itemsPicked);
    }

    private PlayerData GetPlayerData()
    {
        float[] playerStat = new float[3];
        playerStat[0] = PlayerStatus.instance.currentHealth;
        playerStat[1] = PlayerStatus.instance.currentHunger;
        playerStat[2] = PlayerStatus.instance.currentThirst;

        float[] playerPosition = new float[6];
        playerPosition[0] = PlayerStatus.instance.playerBody.transform.position.x;
        playerPosition[1] = PlayerStatus.instance.playerBody.transform.position.y;
        playerPosition[2] = PlayerStatus.instance.playerBody.transform.position.z;
        playerPosition[3] = PlayerStatus.instance.playerBody.transform.position.x;
        playerPosition[4] = PlayerStatus.instance.playerBody.transform.position.y;
        playerPosition[5] = PlayerStatus.instance.playerBody.transform.position.z;

        string[] inventory = InventorySystem.Instance.itemlist.ToArray();
        string[] quickSlots = GetQuickSlotcontent();

        return new PlayerData(playerStat, playerPosition, inventory, quickSlots);
    }

    private string[] GetQuickSlotcontent()
    {
        List<string> temp = new List<string>();
        foreach (GameObject slot in EquipSystem.Instance.quickSlotsList)
        {
            if (slot.transform.childCount !=0)
            {
                string name = slot.transform.GetChild(0).name;
                string str2 = "(Clone)";
                string cleanName = name.Replace(str2, "");
                temp.Add(cleanName);
            }
        }
        return temp.ToArray();
    }

    public void SaveGameData(AllGameData gameData, int slotnumber)
    {
        if (isSavingtoJson)
        {
            SaveDataToJson(gameData, slotnumber);
        }
        else
        {
            SaveDataTobinary(gameData, slotnumber);
        }
    }
     public AllGameData LoadGameData(int slotnumber)
    {
        if (isSavingtoJson)
        {
            AllGameData gameData = LoadDataFromJson(slotnumber);
            return gameData;
        }
        else
        {
            AllGameData gameData = LoadDataFrombinary(slotnumber);
            return gameData;
        }

    }
    public void LoadGame(int slotnumber)
    {
```

```
        SetPlayerData(LoadGameData(slotnumber).playerData);
        SetWorldData(LoadGameData(slotnumber).worldData);
        isLoading = false;
        LoadingScreenOff();
    }

    private void SetWorldData(WorldData worldData)
    {
        foreach (Transform itemType in
WorldManager.instance.allItems.transform)
        {
            foreach (Transform item in itemType.transform)
            {
                if (worldData.pickedItems.Contains(item.name))
                {
                    Destroy(item.gameObject);
                }
            }
        }
        InventorySystem.Instance.itemsPicked =
worldData.pickedItems;
    }

    private void SetPlayerData(PlayerData playerData)
    {
        PlayerStatus.instance.currentHealth =
playerData.playerStat[0];
        PlayerStatus.instance.currentHunger =
playerData.playerStat[1];
        PlayerStatus.instance.currentThirst =
playerData.playerStat[2];

        GameObject player = PlayerStatus.instance.playerBody;

        Vector3 loadedPosition = new Vector3(
            playerData.playerPosition[0],
            playerData.playerPosition[1],
            playerData.playerPosition[2]
        );

        Vector3 loadedRotation = new Vector3(
            playerData.playerPosition[3],
            playerData.playerPosition[4],
            playerData.playerPosition[5]
        );

        CharacterController cc =
player.GetComponent<CharacterController>();
        PlayerMovement movement =
player.GetComponent<PlayerMovement>();

        if (cc != null) cc.enabled = false;

        player.transform.position = loadedPosition;
        player.transform.rotation =
Quaternion.Euler(loadedRotation);

        if (movement != null)
            movement.velocity = Vector3.zero;

        if (cc != null) cc.enabled = true;
        foreach (string item in
playerData.playerinventoryContent)
        {
            InventorySystem.Instance.AddToInventory(item);
        }
        foreach (string item in playerData.playerQuckslotContent)
        {
            GameObject availableSlot =
EquipSystem.Instance.FindNextEmptySlot();
            var itemToAdd =
Instantiate(Resources.Load<GameObject>(item));
            itemToAdd.transform.SetParent(
availableSlot.transform, false );

        }

    }
    public void StartLoadedGame(int slotnumber)
    {
        LoadingScreenON();
        isLoading = true;
        SceneManager.LoadScene("GameScene");
        StartCoroutine(DelayedLoading(slotnumber));
```

```
    }

    private IEnumerator DelayedLoading(int slotnumber)
    {
        yield return new WaitForSeconds(1f);
        LoadGame(slotnumber);
        print("Game loaded correct");
    }

    #endregion

    #region To Binary
    public void SaveDataTobinary(AllGameData gameData, int slotnumber)
    {
        BinaryFormatter formatter = new BinaryFormatter();

        FileStream stream = new FileStream(binaryPath + fileName + slotnumber + ".bin", FileMode.Create);
        formatter.Serialize(stream, gameData);
        stream.Close();

        print(binaryPath + fileName + slotnumber + ".bin");
    }

    public AllGameData LoadDataFrombinary(int slotnumber)
    {
        if (File.Exists(binaryPath + fileName + slotnumber + ".bin"))
        {
            BinaryFormatter formatter= new BinaryFormatter();
            FileStream stream = new FileStream(binaryPath + fileName + slotnumber + ".bin",FileMode.Open);

            AllGameData data = formatter.Deserialize(stream) as AllGameData;
            stream.Close();
            print("Data Loaded from binary");
            return data;
        }
        else
        {
            return null;
        }
    }
    #endregion

    #region To Json
    public void SaveDataToJson(AllGameData gameData, int slotnumber)
    {
        string json = JsonUtility.ToJson(gameData);
       // string encrypted = Encrypter(json);
        using (StreamWriter writer = new StreamWriter(jsonPathProject + fileName + slotnumber + ".json"))
        {
            writer.Write(json);
            print("Saved" + jsonPathProject + fileName + slotnumber + ".json");
        };
    }

    public AllGameData LoadDataFromJson(int slotnumber)
    {
        using (StreamReader reader = new StreamReader(jsonPathProject + fileName + slotnumber + ".json"))
        {
            string json = reader.ReadToEnd();
            //string decrypted = Encrypter(json);
            AllGameData data = JsonUtility.FromJson<AllGameData>(json);
            return data;
        }

    }
    #endregion

    #region Settings

    #region Volume
    [System.Serializable]
    public class SavedVolume
```

```
    {
        public float sound;
        public float effects;
        public float all;
    }
    public void SaveVolume(float _sound, float _effects, float
_all)
    {
        SavedVolume volumeSettings = new SavedVolume()
        {
            sound = _sound,
            effects = _effects,
            all = _all
        };
        PlayerPrefs.SetString("Volume",
JsonUtility.ToJson(volumeSettings));
        PlayerPrefs.Save();
    }
    public SavedVolume LoadVolume()
    {
        return
JsonUtility.FromJson<SavedVolume>(PlayerPrefs.GetString("
Volume"));

    }
    #endregion

    #endregion

    #region Loading
    public void LoadingScreenON()
    {
        loadingScreen.gameObject.SetActive(true);
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;

    }
    public void LoadingScreenOff()
    {
        loadingScreen?.gameObject.SetActive(false);
    }
    #endregion

    #region Encryption
    public string Encrypter(string datastring)
    {
        string keyword = "277353";
        string result = "";
        for (int i = 0; i < datastring.Length; i++)
        {
            result += (char)(datastring[i] ^ keyword[i %
keyword.Length]);
        }
        return result;

    }
    #endregion

    #region Misc
    public bool DoExist(int slotnumber)
    {
        if (isSavingtoJson)
        {
            if (System.IO.File.Exists(jsonPathProject + fileName +
slotnumber +".json"))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            if (System.IO.File.Exists(binaryPath + fileName +
slotnumber + ".bin"))
            {
                return true;
            }
            else
```

```
        {
            return false;
        }
    }
}

public bool isSlotEmpty(int slotnumber)
{
    if (DoExist(slotnumber))
    {
        return false;
    }
    else
    {
            return true;
        }
    }

    public void DeselectButton()
    {
        GameObject myEventSystem =
GameObject.Find("EventSystem");

myEventSystem.GetComponent<UnityEngine.EventSystems.E
ventSystem>().SetSelectedGameObject(null);
}
    #endregion
}
```