Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота

за освітнім ступенем «бакалавр»

на тему:

Комп’ютерна гра на основі рушія Unreal Engine

Виконав:
здобувач IV курсу
групи ІПЗ-41
спеціальності 121 «Інженерія
програмного забезпечення»
Ільюк Ілля Ігорович

Науковий керівник:
к.п.н., доцент Петренко С. В.

Рівне – 2024

АНОТАЦІЯ

Ільюк І. І. «Комп’ютерна гра на основі рушія Unreal Engine». – Кваліфікаційна робота на здобуття освітнього ступеня «бакалавр» за спеціальністю 121 Інженерія програмного забезпечення – Рівненський державний гуманітарний університет – Рівне, 2024. – 41 с.

У кваліфікаційній роботі розглянуто предметну область та проведено її аналіз, зокрема визначено та класифіковано основні види комп'ютерних ігор та визначено тенденції ігрової індустрії. На етапі теоретичного аналізу було проаналізовано існуючий інструментарій для розробки ігрових застосунків, що дало змогу зробити вибір технологічного стеку та обґрунтувати його.

Розроблено концепцію гри, створено загальну модель та описано функціональні можливості гри. Проведено моделювання основних ігрових компонентів та інтерфейсу. Здійснено ретельний аналіз ігрової механіки та визначено ключові елементи взаємодії користувача з ігровим середовищем.

У результаті роботи було успішно розроблено демонстраційну версію гри «Farming Game». Застосунок розроблено з використанням мови програмування C++ та ігрового рушія Unreal Engine, що забезпечує захоплюючий геймплей, реалістичну графіку та інтуїтивний інтерфейс для користувача. Крім того, гра включає різноманітні інтерактивні елементи та багатий контент.

Перспективи розвитку системи вбачаємо в імплементації нових модулів. Одним з них може бути режим мережевої гри. Також досить важливим завданням є оптимізація програмного продукту для зниження системних вимог.

Ключові слова: проєктування, C++, UML-діаграми, Unreal Engine.

ЗМІСТ

ВСТУП

У сучасному глобалізованому світі комп'ютерні ігри є не лише розважальними продуктами, але й важливими об'єктами дослідження та розвитку індустрії розваг. Нині, в епоху стрімкого розвитку технологій, рушій Unreal Engine став одним з найпопулярніших та потужних інструментів для розробки ігор.

Розвиток комп'ютерних ігор на сьогодні не тільки відображає швидкий технологічний прогрес, але й є ключовим елементом в культурному, розважальному та освітньому контекстах. Одним із найсуттєвіших аспектів створення ігор є використання потужних рушіїв, серед яких виділяється Unreal Engine. Цей рушій не лише надає можливості для творчого прояву, а й створює платформу для реалізації неймовірних ідей та концепцій у галузі ігрової індустрії.

Актуальність дослідження заснована на стрімкому розвитку сучасних технологій та зростаючій популярності комп'ютерних ігор серед різних прошарків суспільства. У глобальному контексті ігрова індустрія постійно розширюється, що створює величезні можливості для новаторських проєктів та відкриває двері для талановитих розробників. Unreal Engine завдяки своїм потужним можливостям та широкому спектру інструментів залишається однією з перших у виборі для створення ігор різних жанрів і складності.

Мета дипломної роботи полягає у вивченні ігрового рушія Unreal Engine та розробку ігрового застосунку на його основі.

Для реалізації поставленої мети визначено такі **завдання**:

1. Здійснити аналіз предметної області, а саме: дослідити історію розвитку ігор, їх класифікацію, процес розробки, обґрунтувати вибір технологій.
2. Спроєктувати ігровий застосунок в рамках Unreal Engine.
3. Розробити ігровий додаток.

Об'єктом дослідження є програмне забезпечення на основі рушія Unreal Engine.

Предметами дослідження є проєктування та розробка ігрового застосунку на основі рушія Unreal Engine. Дослідження спрямоване на вивчення цих аспектів для отримання глибокого розуміння процесу розробки ігрових застосунків.

Для досягнення поставленої мети в роботі використовувалися різноманітні **методи наукового дослідження**, що спрямовані на аналіз підходів до розробки ігрових застосунків та проєктування гри, а саме: аналітичний метод, метод моделювання та інженерно-технічний метод.

Практична цінність дослідження полягає в аналізі та емпіричній експертизі кращих практик створення ігрових застосунків на основі рушія Unreal Engine та розробці ігрового додатку.

Апробація і впровадження результатів дослідження. Основні результати дослідження обговорювалися і були схвалені на звітних наукових конференціях викладачів Рівненського державного гуманітарного університету, за матеріалами дослідження опубліковано тези на XVII Всеукраїнській науково-практичній конференції здобувачів вищої освіти та молодих учених «Наука, освіта, суспільство очима молодих» (м. Рівне, 17 травня 2024р.).

Структура та обсяг роботи. Дипломна робота складається зі вступу, трьох розділів, висновків до роботи, списку використаних джерел та додатків. Основний зміст викладено на 41 сторінках.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Комп’ютерна гра: визначення та класифікація

Комп’ютерна гра — це електронна гра, в ігровому процесі якої гравець використовує інтерфейс користувача, щоб отримати зворотну інформацію з відеопристрою. Електронні пристрої, які використовуються для того, щоб грати, називаються ігровими платформами. Наприклад, до таких платформ належать персональний комп'ютер та гральна консоль. Пристрій введення, який використовується для керування грою, називається ігровим контролером. Це може бути, наприклад, джойстик, клавіатура та мишка, геймпад або сенсорний екран[1].

Розвиток програмованих комп’ютерів і технологій формування зображення на екранах електронних пристроїв передував появі відеоігор. Багато електронних і механічних ігрових пристроїв існували в першій половині XX століття, але вони не були широко поширені. Початково ігрові програми, такі як шахи та хрестики-нулики, розроблялися в рамках військової програми США з метою створення комп’ютера, здатного прогнозувати, що робить противник.

Вільям Гіґінботам є автором першої успішної спроби створити розважальний пристрій, який використовує зворотний зв’язок із гравцем відео. Він створив Tennis For Two у 1958 році, але не вважав гру важливою. Зрештою він розібрав обладнання для інших наукових проектів.

У 1960-ті студенти Массачусетського технологічного інституту створили гру Spacewar!, яка у 1966 році підштовхнула Sanders Associates до думки про створення грального пристрою, який можна було б під’єднати до домашніх телевізорів. Студент Стенфорду Білл Піттс зі своїм другом Х’ю Таком, вражені Spacewar!, вирішили, створити ігровий автомат який дозволяв гравцям грати за

гроші. Цей автомат Computer Space і однойменна з ним гра 1971 року стали першими комерційними пристроями з відеогрою.

Згодом, через перенасичення ринку однотипними пристроями, у 1977 році стався перший обвал ринку, який закінчився 1978 року з виходом гри Space Invaders від Taito, яка отримала велику популярність і призвела до початку так званої «золотої епохи аркадних відеоігор» та повернула в індустрію багато компаній. Успіх цієї гри призвів до появи аркадних ігрових автоматів у таких місцях, як кінотеатри, ресторани та магазини. Це було видно у багатьох американських фільмах того часу. Відеоігри також стали популярним захопленням, яке стрімко поширювалося. Це призвело до значного висвітлення в ЗМІ. Незабаром гра була ліцензована для консолі Atari VCS, яка згодом стала Atari 2600. Продажі гри зросли у чотири рази. Компанія змогла відновитися після невдач у минулому та відновити довіру до ринку домашніх гральних консолей, розпочавши своє друге покоління, яке тривало до другого Північноамериканського обвалу ринку відеоігор.

Розробники були змушені створювати все нові і нові ігри, часто знижуючи якість. В результаті вони зіткнулися з обмеженістю того часу ігрових платформ і відсутністю інноваційних ідей. У 1986 році ринок консольних ігор стрімко впав з 3,2 млрд. доларів у 1983 році до 100 млн. доларів. Тим не менш, в той же час японська відеоігрова індустрія стрімко розвивалася, як і відеоігри для домашніх комп'ютерів. Поява Nintendo Entertainment System нормалізувала та навіть збільшила ринок. Японські розробники очолили третє покоління консолей, що призвело до успіху.

Домашні комп'ютери стали розглядатися як повноцінна ігрова платформа і конкурент приставок після поширення CD-дисків і впровадження CD-приводів як стандартної складової ПК. Оскільки якісний звук і відеозаписи можна було відтворювати на CD-дисках, на початку-середині 1990-х виникло багато інтерактивних фільмів. Розвиток персональних комп'ютерів дозволив створювати все більш видовищні та сюжетно наповнені ігри, які залучали професійних акторів і сценаристів. За останні кілька років відеоігри перейшли

від двовимірних і псевдотривимірних ігор (Doom, Duke Nukem) до повноцінних тривимірних ігор (Half-Life), які коштували стільки ж скільки фільми. Компанія Id Software розпочала новий підхід до створення інтерактивних творів, пропонуючи клієнтам інструменти для створення відеоігор і ліцензії на використання ігрових рушіїв. Це дозволило робити ігри простіше і вийти на ринок незалежним студіям.

У 2000-х роках графіка відеоігор ставала все реалістичнішою та з'явилися онлайн-режими, що дозволили багатьом гравцям взаємодіяти в реальному часі. Наприкінці десятиліття почалася інтеграція з соціальними мережами та ігровими сервісами. Цифрова дистрибуція перевершує фізичну. Це призвело до поширення інді-ігор, ігор від незалежних розробників чи навіть ігор, створених однією людиною. Завдяки моддингу та створенню інтерактивних творів у спеціальних редакторах гравці стали активними учасниками створення та модифікації відеоігор. Впровадження платних завантажуваних доповнень до ігор, спрямованих на покращення гри, стало нормою. У 2010 році стандарти якості графіки стрімко підвищилися, особливо після виходу восьмого покоління ігрових систем, що призвело до поширення дисплеїв 4K. Ігри та пристрої віртуальної реальності значно зросли. Індустрія ігор для мобільних пристроїв, зокрема смартфонів, помітно розвинулася. У культурі відеоігор стримінг став важливою частиною.

Кожна велика індустрія має свої класифікації, індустрія відео ігор не виключення. Класифікація ігор відбувається з метою систематизації ігрових продуктів, але через змішання жанрів класифікації можуть відрізнятися одна від одної.

В першу чергу ігри класифікують за масштабністю, зазвичай виділяють два види:

- Інді-ігри;
- ААА-ігри.

Загалом, інді-видання, як правило, дешевші, коротші та менші за обсягом, з більш стилізованим художнім оформленням.

Інді-ігри завжди розробляються окремими особами або невеликими командами, які рідко отримують фінансову підтримку від видавця. Насправді, багато інді-розробників покладаються на краудфандинг або особисті інвестиції, щоб покрити витрати на розробку. Оскільки розробники не мають величезного бюджету, ігри, як правило, менші за розміром і коротші за тривалістю. Саме тому багато з них мають стилізоване художнє оформлення.

Інді-стиль зазвичай вважається більш доступним для широкої аудиторії з кількох причин. По-перше, вони набагато дешевші за ААА-ігри - інді-тайтли рідко коштують більше двадцяти доларів. Крім того, вони мають відносно простий геймплей, їх легше взяти в руки і грати. Така простота дозволяє насолоджуватися грою найрізноманітнішим гравцям, незалежно від їхніх ігрових навичок чи досвіду. Нарешті, вони, як правило, значно менш вимогливі до апаратного забезпечення, а це означає, що гравцям на ПК не знадобляться найновіші комплектуючі, щоб добре їх запускати.

На відміну від інді-ігор, ААА-ігри розробляються великими студіями, в яких працюють сотні, а то й тисячі людей. Крім того, проекти підтримуються видавцем (наприклад, Activision), який забезпечує команду розробників величезним бюджетом. Через розмір бюджету та команди розробників, ААА-ігри зазвичай довгі, великі, з деталізованою та реалістичною графікою. Сам термін ААА означає очікування, що ці ігри є неймовірно якісними.

Внаслідок значно більших грошових інвестицій, які йдуть на створення ААА-ігор, вони завжди коштують дорожче, ніж інді-ігри. Зазвичай ААА-ігри коштують від $40 до $60. Ігровий процес в ААА-іграх, як правило, набагато складніший, а в багатокористувацьких іграх часто є крива навчання, яку гравцям доводиться проходити. Оскільки такі ігри зазвичай мають найсучаснішу візуальну складову, гравцям на ПК потрібне достатньо потужне апаратне забезпечення для безперебійного запуску гри[2].

Щодо інших класифікацій, здебільшого вона здійснюється за наступними критеріями:

- Кількістю гравців;

- Платформою;
- Жанром.

Але це далеко не всі класифікації які можуть бути, до прикладу можна взяти різні рейтинги вмісту, характер видання, візуальне подання та інші.

За кількістю гравців ігри поділяють на два види:

- Однокористувацькі – гра для одного гравця;
- Багатокористувацькі – гра для більше ніж одного гравця.
- Багатокористувацькі ігри в свою чергу поділяються на:
- MMO (massively multiplayer online) – гра у якій на одному сервері знаходиться дуже велика кількість гравців. Здебільшого такі ігри мають відкритий світ;
- Кооператив – гра у якій гравці об’єднуються в групу та разом проходять гру.
- PvE(player versus environment) – гравці змагаються проти ШІ;
- PvP (player versus player) – гравці змагаються між собою.

За платформою ігри поділяються на:

- Мобільні;
- Комп’ютерні;
- Консольні;
- Браузерні.

Також гра може бути кросплатформною зі збереженням ігрового процесу у хмарі яку дає розробник або видавець, таким чином у гравця з’являється можливість грати в гру на своєму смартфоні та через деякий час продовжити проходження на консолі.

У відеоіграх на відміну від кіно або літератури жанр не впливає на сценарій та візуальний стиль, що робить розробку гри більш унікальною. Існує велика кількість жанрів та піджанрів комп’ютерних ігор, які постійно з’являються або зникають. Основними жанрами є:

- Екшен – жанр відеоігор, в якому дії в грі відбуваються дуже швидко. Граючи в ігри цього жанру велику роль відіграють рефлекси гравця;

- Аркада – жанр відеоігор, в якому примітивний ігровий процес;
- Пригоди – жанр відеоігор, в якому гравець бере на себе роль протагоніста в інтерактивній історії, рушійною силою якої є дослідження та/або розв'язання головоломок;
- Симулятор – жанр відеоігор, в якому максимально наближено до реальності відтворюються події, закони та тому подібне для як умога сильнішого занурення у ігровий процес;
- Стратегії – жанр відеоігор, в якому гравець повинен обдумувати кожен наступний крок заздалегіть;
- Рольова гра – жанр відеоігор, в якому гравець має дуже велику кількість можливостей у різних аспектах, від вибору зброї до варіантів фраз у діалогах які впливають на ігровий процес;
- Навчальна гра – жанр відеоігор, в якому основне завдання гри це навчити гравця чомусь;
- Головоломки – жанр відеоігор, в якому гравець повинен вирішувати логічні завдання, які вимагають використання різного роду здібностей гравця.

Варто відзначити, що гра може мати декілька жанрів. Наприклад Sherlock Holmes Chapter One від української студії Frogwares, має одразу два жанри – пригоди та бойовик.

Якщо говорити про ігрову індустрію сьогодення, то прослідковуються декілька головних тенденцій[3].

Виходить більше ігор, які призначені винятково для мобільних пристроїв. Їх випускають всі види студій, та найвищий показник у великих компаній. Там випуск винятково мобільних ігор зріс на 44% за рік.

Великі студії збільшують кількість багатоплатформних ігор. У 2022-му їх виходило на 16% більше, ніж у 2021-му. Тим часом інді-студії, де працює менше за 50 людей, перейшли до випуску ігор для однієї платформи. З них 77% створюють ігри для настільних пристроїв.

Інді-розробники випускають ігри досить швидко, а розробники працюють менше годин. У 2022 році 62% інді-ігор вийшли менш ніж за рік. При цьому маленькі й середні команди стали працювати на 1,2% менше годин (що в сумі дає помітний результат). А от у великих компаніях (300+ людей) час роботи навпаки зростає.

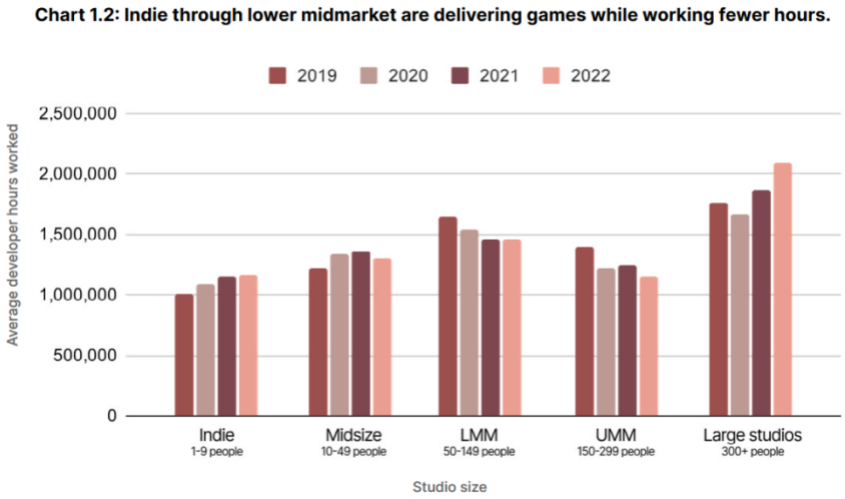


Рисунок 1.1. Середній час роботи розробників різних команд

Тривалість «життя» ігор зростає з року в рік. Студії працюють на довгострокову перспективу, подовжують актуальність мобільних ігор на 33%. Найбільші показники зростання у казино, карткових ігор та RPG. Також спостерігають вищі показники утримання гравців — і найбільша популярність тут у батл-пасів.

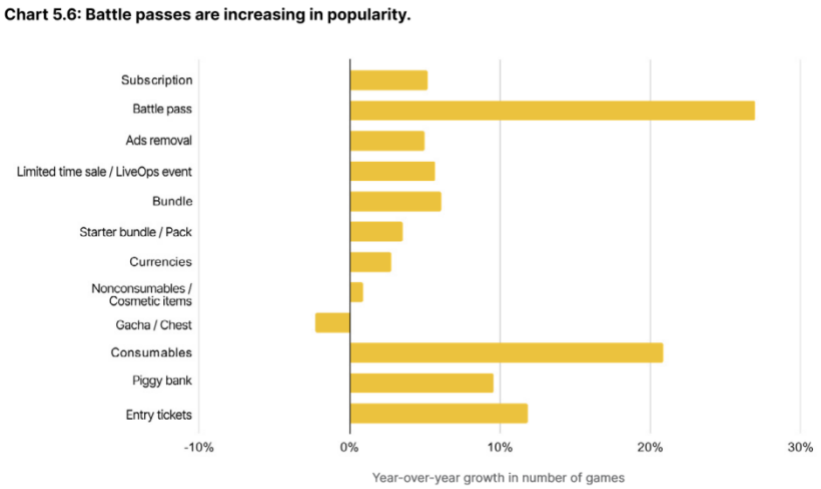


Рисунок. 1.2. Причини утримання гравців

У мобільні ігри грають більше людей, ніж у 2021 році. Щоденна кількість активних користувачів у світі зросла на 8% для середньої гри. Водночас кількість користувачів, які купують контент в грі, впала на 2%. Unity припускає, що монетизація через перегляд реклами в іграх може бути більш актуальною, ніж покупки в грі. При цьому і реклама, і покупки подобаються гравцям мобільних ігор більше, ніж одноразові платежі й підписки.

1.2. Етапи створення гри

Процес створення комп'ютерних ігор – це творчий та багатоступеневий процес, який об'єднує талант розробників, програмістів, дизайнерів та інших фахівців для створення унікального віртуального світу, призначеного не тільки для розваг, а навіть навчання.

Перед тим, як розглядати конкретні етапи створення, важливо зазначити, що кожен проект може мати свої унікальні особливості та вимоги. Однак загальний процес розробки гри можна умовно поділити на декілька ключових етапів, а саме:

1. Проєктування;
2. Розробка;
3. Випуск.

На початковому етапі проєктування комп'ютерної гри ключовою є ідея – основна концепція, яка визначає тематику та цільовий напрямок проекту. Розробники аналізують потенційну аудиторію та визначають основні цілі гри. Після цього визначити жанр стає не так і складно, оскільки ідея та жанр тісно пов’язані. Так, наприклад, якщо ідея майбутньої гри – саджати рослини, то її жанром буде симулятор, а якщо будувати місто, то стратегія.

Наступним кроком потрібно обрати сетинг. Сетинг – це концептуальна або візуальна обстановка, в якій відбуваються події гри. Сетинг включає в себе такі аспекти, як:

- Локації;

- Час і період;
- Культура;
- Технології;
- Атмосфера.

Його визначення сильно полегшує написання сюжету гри, історії персонажів, малювання та моделювання концептів локацій та персонажів. Для якнайбільшого охоплення аудиторії обирати сетинг, потрібно опираючись на дані з аналізу аудиторії.

На прикінці цього етапу у команди розробників мають бути чітко визначені основні принципи, які будуть лежати в основі гри, і створені базові концепти на основі напрацьованого матеріалу.

Наступним етапом реалізації проєтку є розробка, цей етап займає найбільше часу та ресурсів, оскільки результатом етапу повинен бути готовий продукт.

На цьому етапі більшість завдань виконується паралельно, де кожен відділ працює над своєю частиною. Загалом є такі відділи, які в свою чергу можуть ділитись на під відділи:

- Програмісти;
- Художники;
- Звукорежисери;
- Дизайнери;
- Менеджери;
- Аніматори;
- Тестувальники.

Відділ програмістів програмним кодом реалізує ігрові механіки та правила, створює файлову структуру, створює графіку, працює з рушієм на основі якого розробляється гра.

Відділ художників створює моделі персонажів, об’єктів оточення, мапи рівнів, концепт-арти(зображення на основі якого відбувається прогрес розробки).

Відділ звукорежисерів наповнює гру звуком, починаючи з запису голосів для персонажів, закінчуюючи звуком падіння та музикою у меню.

Відділ дизайнерів працює над історією гри, її сюжетом, історією персонажів, створенням ідей для механік гри.

Відділ менеджерів займається керуванням всіх процесів та виступає своєрідним містком між різними групами, які мають співпрацювати для виконання завдань.

Відділ аніматорів змушує персонажів рухатися за допомогою анімаційних технік.

Відділ тестувальників займається перевіркою роботи ігрових механік та коректність взаємодії з ігровим світом у робочій збірці гри.

Під час цього етапу, гра проходить величезну кількість ітерацій та постійно покращується, але існує 4 основні версії гри, а саме:

- Пре-альфа;
- Альфа;
- Бета;
- Реліз.

Пре-альфа – початкова версія гри, період версії від початку розробки до першої альфи.

Альфа – тестова версія гри, яка підлягає внутрішньому тестуванню. Має багато недоліків та тільки частину потрібних механік

Бета – тестова версія гри, яка пройшла внутрішнє тестування та готова до публічного. Зазвичай така версія має значно менше недоліків.

Для бета-тестування часто використовують потенційних гравців, для зменшення витрат на тестування та підігрівання їх інтересу. Таким чином, гравцям надається можливість зіграти в гру ще до її релізу.

Реліз – версія гри, яка готова до тиражування. Стабільна версія, яка або взагалі не має недоліків, або мінімальну кількість, яку виправлять з найближчим патчом(файли з виправленнями для готового продукту).

Наступним етапом після розробки є випуск продукту та його підтримка. Підтримка здійснюється випуском патчів(файли з виправленнями для готового продукту). Також для продовження життєвого циклу гри через деякий час, для неї випускають DLC (Downloadable content), які додають різного роду контент. Контен може бути просто косметичний або нові предмети, або ж нові локації/рівні з новими завданнями та механіками.

Вцілому, етапи розробки будь-якого програмного забезпечення мало чим відрізняються, окрім специфіки.

1.3. Аналіз ігрових рушіїв

Основою будь якої відеогри є її ігровий рушій, саме тому, аналіз і вибір рушія дуже важливе завдання. Ігровий рушій — програмний рушій, центральна програмна частина будь-якої відеогри, яка відповідає за всю її технічну сторону, дозволяє полегшити розробку гри шляхом уніфікації та систематизації її внутрішньої структури[4].

Є велика різниця між рушіями та кількістю роботи яку вони виконують. Одні з них мають можливості просто відображати графіку, а інші мають усю потрібну логіку для реалізації гри окремого жанру.

Зазвичай ігрові рушії створюються поверх низькорівневих фреймворків та включають в себе різні бібліотеки, наприклад для рендерингу, роботи із фізикою, звуком, математикою, системою скриптів, анімаціями, керуванням пам'яттю, штучним інтелектом тощо.

Створення власного ігрового рушія це не легкий процес, який значно збільшить час розробки відеогри та витрати пов’язані з розробкою, але є декілька причин, для його створення, а саме:

- Рушії, які існують не підтримують або дуже складно реалізувати підтримку нової технології, яку ви плануєте використовувати;

- Бажання оптимізувати розробку власних ігор. Існуючі рушії мають забагато функціоналу який ви не використовуєте, але модифікуєте рушій під себе, тому вам краще було б зробити свій спеціалізований рушій;
- Бажання бути незалежними від власника рушія;
- На основі цього рушія планується створення не одного продукту.

Якщо, жодна з причин для вас не релевантна, тоді слід вибрати один з існуючих. На даний час існує величезна кількість ігрових рушіїв, але не всі з них є у вільному доступі. Більшість з них, розробленні для внутрішнього використання у компанії та не призначені для ліцензування. Тому для аналізу було обрано найвідоміші з доступних, а саме:

- Unreal Engine;
- Unity;
- GameMaker;
- Godot.

Unreal Engine – ігровий рушій розроблений компанією Epic Games. Спочатку розроблявся для шутерів від першої особи, але з часом виріс у величезний та потужний продукт для розробки не тільки ігор різних жанрів а навіть і використання у кіноіндустрії.

Написаний переважно на C++ з частинками C#. Для написання ігрової логіки використовується C++ та Blueprint. Рушій дозволяє створювати відеоігри для всіх операційних систем та платформ включаючи VR та AR. Для забезпечення такої мультиплатформності рушій використовує модульну систему залежностей та підтримує різні системи рендерингу, відтворення звуку, модулі для роботи з мережею та підтримуваними пристроями введення.

Даний рушій добре підходить для AAA-проєктів, оскільки підтримує багато сучасних технологій та високу якість графіки.

З кожним виходом нової версії рушія, умови ліцензування постійно змінювались, та мали не завжди привабливі умови для покупців, однак зараз все набагато простіше. Зараз існує 3 види ліцензій:

- Стандартна ліцензія;

- Програма для підприємств;
- Користувацька ліцензія.

Стандартна ліцензія створена для тих кому не потрібна преміум підтримка або спеціальні умови. За таку ліцензію не потрібно платити до моменту коли продукт заробить 1 мільйон доларів США валового доходу, після цього потрібно буде сплачувати 5% від валового доходу.

Програма для підприємств створена для не ігрових продуктів, яким потрібна преміум підтримка, приватне навчання та/або індивідуальні умови ліцензування. Ціна для одного користувача 1500 доларів США на рік.

Користувацька ліцензія створена для професійної ігрової розробки з преміум підтримкою, приватним навчанням та спеціальними умовами ліцензування. Ціна такої ліцензії не визначена на перед та обговорюється при укладанні договору.

Unity – ігровий рушій розроблений компанією Unity Technologies.

Написаний переважно на C++. Для написання ігрової логіки використовується C# та JavaScript. Рушій дозволяє створювати відеоігри для більшості платформ, але не у безкоштовній версії[5].

Рушій можна використовувати для створення тривимірних і двовимірних ігор, а також інтерактивних симуляцій та інших додатків. Рушій був прийнятий в інших галузях, окрім відеоігор, таких як кіно, автомобільна промисловість, архітектура, інженерія та будівництво.

Він особливо популярний для розробки мобільних ігор для iOS та Android, вважається простим у використанні для розробників-початківців, а також популярний для розробки інді-ігор.

Unity поширюється безкоштовно, але крім безкоштовної, існують чотири плани Unity Personal, Unity Pro, Unity Enterprise і Unity Industry. Вони відрізняються вартістю та функціональністю.

Unity Personal має багато обмежень, проте має можливість поширювати ігри, якщо щорічний дохід з гри не перевищує 100000 доларів США за останні 12 місяців.

Unity Pro має різні інструменти, підтримку та ресурси, які допоможуть створювати кращі проекти швидше. Також підтримує багатоплатформну публікацію та включає в себе екосистему продуктів і сервісів Unity Cloud. Коштує 2040 доларів США на рік.

Unity Enterprise це план ля серйозних команд будь-якого розміру зі складними проектами. Включає в себе прискорену технічну підтримку, доступ до вихідного коду Unity, надається озширена довгострокова підтримка (LTS) та доступні хмарні інструменти для співпраці Unity Cloud.

Unity Industry надає набір 3D-продуктів і сервісів для створення користувацьких додатків для AR/VR, мобільних, десктопних і веб-програм у реальному часі. Ціна 4950 доларів США на рік.

GameMaker – серія кросплатформних ігрових рушіїв, створених Марком Овермарсом у 1999 році і розроблених YoYo Games з 2007 року. Остання ітерація GameMaker була випущена у 2022 році.

GameMaker дозволяє створювати кросплатформні та мультижанрові відеоігри за допомогою спеціальної мови візуального програмування drag-and-drop або скриптової мови, відомої як Game Maker Language, яка може бути використана для розробки більш просунутих ігор, які не можуть бути створені лише за допомогою функцій візуального програмування[6].

Він в першу чергу призначений для створення ігор з 2D графікою, дозволяючи використовувати растрову графіку, векторну графіку (через SWF), та 2D скелетну анімацію (через Spine від Esoteric Software) разом з великою стандартною бібліотекою для малювання графіки та 2D примітивів.

Рушій використовує Direct3D на Windows, UWP та Xbox One; OpenGL на macOS та Linux; OpenGL ES на Android та iOS, WebGL або 2d canvas на HTML5 та власні API на консолях.

Основним елементом рушія є IDE з вбудованими редакторами растрової графіки, дизайну рівнів, скриптів, контурів і шейдерів (GLSL або HLSL). Додаткова функціональність може бути реалізована мовою скриптів

програмного забезпечення або власними розширеннями для конкретної платформи.

Рушій має 3 ліцензії:

Безкоштовна – безкоштовна ліцензія, яка має усі функції рушія окрім експорту для консолей. Надає комерційну ліцензію лише на GX.games.

Професійна – ліцензія, яка має усі функції рушія окрім експорту для консолей. Її вартість 100 доларів США. Надає комерційну ліцензію для усіх доступних платформ.

Підприємницька – ліцензія, яка має усі функції рушія разом з експортм для консолей. Її вартість 80 доларів США на рік. Надає комерційну ліцензію для усіх доступних платформ.

Godot – кросплатформенний, безкоштовний ігровий рушій з відкритим вихідним кодом, випущений під дозвільною ліцензією MIT. Спочатку він був розроблений аргентинськими розробниками програмного забезпечення Хуаном Лінецьким та Аріелем Манзуром для кількох компаній у Латинській Америці перед його публічним випуском у 2014 р. Середовище розробки працює на багатьох платформах і може експортуватися ще на кілька. Воно призначене для створення 2D і 3D ігор для ПК, мобільних і веб-платформ, а також може використовуватися для розробки неігрового програмного забезпечення, зокрема редакторів[7].

Godot дозволяє розробникам відеоігор створювати 3D і 2D ігри, використовуючи різні мови програмування, такі як C++, C# і GDScript. Він використовує ієрархію вузлів для полегшення процесу розробки. Класи можуть бути похідними від типу вузла для створення більш спеціалізованих типів вузлів, які успадковують поведінку. Вузли організовані всередині "сцен", які є багаторазовими, екземплярами, успадкованими та вкладеними групами вузлів.

Unity і Unreal Engine 4 на сьогоднішній день найпопулярніші ігрові рушії, але GameMaker та Godot теж набувають популярності через свою ціну.

Якщо потрібно розробити мобільну гру з внутрішніми покупками та рекламами, можна взяти любий з перелічених рушіїв, але саме на Unity це буде

реалізувати найлегше. Він має безліч плагінів для розробки ігор саме для таких видів розробок.

Якщо це 2D гра, теж можна обрати Unity, але для збільшення свого прибутку краще було б обрати Godot або GameMaker, через ціну їх ліцензій.

Для 3D гри з хорошою графікою краще обрати Unreal Engine.

Важливим аспектом вибору рушія є мова програмування якою програмується логіка. Якщо ви віддаєте перевагу якійсь окремій мові, тоді вибір рушія стає очевиднішим.

Теж хорошим аспектом для вибору є важливість відкритості коду рушія, оскільки виправити якусь частинку рушія швидше, ніж надіятись, що це зробить видавець.

Підставами для використання у цій роботі саме Unreal Engine є наступні фактори:

По-перше, після створення революційного Unreal Engine 5 багато відомих і великих компаній (CD Project Red, Flying Wild Hog, GSC Game World, Microsoft тощо) повністю переходять на Unreal Engine[8];

По-друге, революційні технології та вагомі оновлення. Наприклад, Nanite це нова система віртуалізованої геометрії Unreal Engine 5, Lumen це нова повністю динамічна система глобального освітлення та віддзеркалень Unreal Engine 5, розроблена для консолей нового покоління та високопродуктивних ПК, Virtual Shadow Map це новий метод мапування тіней, розроблений для створення послідовних тіней високої роздільної здатності[9];

По-третє, вигідна модель монетизації. Unreal Engine постачається повністю завантаженим і готовим до роботи з коробки, з усіма функціями та повним доступом до вихідного коду. Початок розробки гри безкоштовний, далі - 5% роялті починають стягуватися лише тоді, коли ваша гра заробить понад 1 мільйон доларів США[10].

РОЗДІЛ 2
ПРОЄКТУВАННЯ, МОДЕЛЮВАННЯ ТА ДИЗАЙН

2.1. Концепція та загальна модель гри

Початковим етапом розробки комп'ютерної гри є її проектування та визначення концепції. На цьому етапі ми визначимо ідею гри, жанр, сетинг, основну механіку. Отже, ось що у нас вийде:

- ідеєю гри – розвиток свого господарства;
- жанр – симулятор;
- сетинг – середньовіччя;
- основна механіка – вирощування господарських рослин.

Вибір ідеї гри базувався на наступних аспектах:

Популярність: Ігри, де гравці можуть будувати та розвивати свої власні господарства, завжди були популярними. Гравці насолоджуються можливістю контролювати різні аспекти світу, вирощувати ресурси, розширювати територію та взаємодіяти з іншими персонажами.

Геймплейна глибина: Розвиток господарства може бути складним та вимагати стратегічного мислення. Гравці повинні планувати ресурси, ефективно використовувати простір та вирішувати економічні завдання.

Залучення гравців: Ігри з розвитком господарства можуть залучити гравців на довгий час. Вони можуть відчувати себе власниками світу, де кожне їхнє рішення має вагу.

Соціальний аспект: Гри з розвитком господарства можуть включати взаємодію з іншими гравцями, обмін ресурсами та спільний розвиток. Це може створити активну спільноту гравців.

Актуальність: З урахуванням сучасних тенденцій, ігри, де гравці можуть будувати свої господарства, залишаються актуальними та привабливими для аудиторії.

Жанр та основна механіка були обрані виходячи з ідеї, оскільки основною механікою розвитку господарства не може бути бій, а жанром перегони.

Вибір сетингу базувався на легкому втіленні та швидкій ідентифікації.

При проєктуванні слід враховувати особливості рушія, зокрема загальну модель гри та зв’язки основних класів.

Запуск гри здійснюється одним з двох шляхів: шлях редактора або шлях автономного редактора. Загальний порядок подій полягає в ініціалізації рушія, створенні та ініціалізації GameInstance, завантаженні рівня і, нарешті, початку гри. Однак між автономним режимом і режимом редактора існують відмінності, як з точки зору точного порядку виклику деяких функцій, так і з точки зору того, які саме функції викликаються. На блок-схемі нижче показано два шляхи, які йдуть паралельно, поки не зійдуться на початку гри.

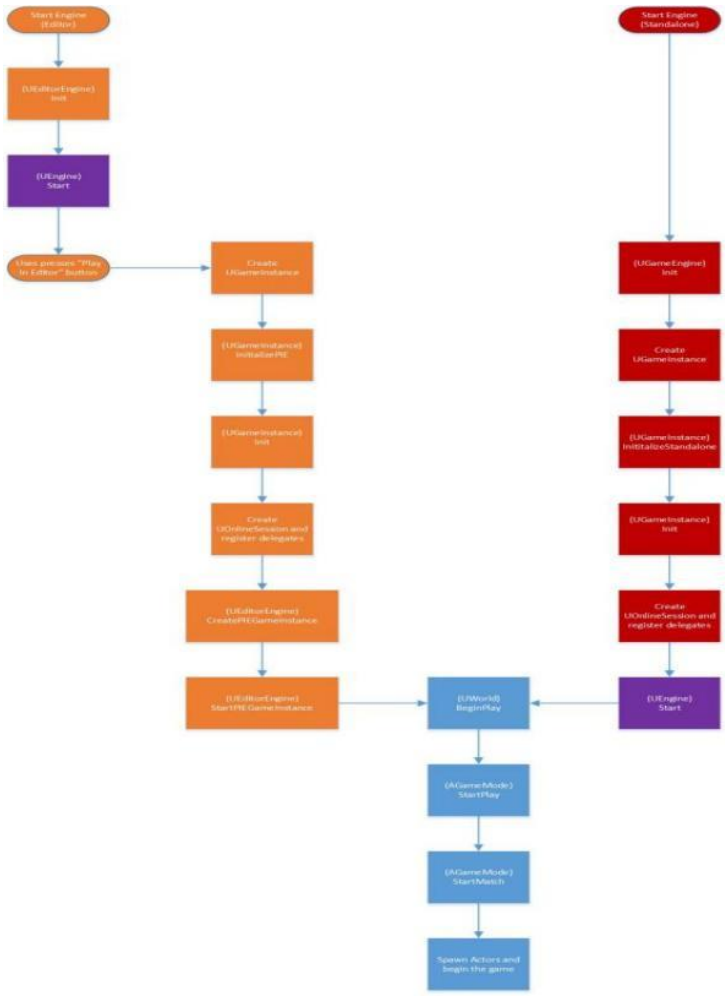


Рисунок. 2.1. Шляхи ініціалізації рушія

У режимі Standalone, який використовується у іграх, що граються поза редактором, об'єкти, необхідні для гри, створюються та ініціалізуються одразу після ініціалізації рушія під час запуску. Такі об'єкти, як GameInstance, створюються та ініціалізуються перед запуском рушія (на відміну від створення та ініціалізації рушія), а стартова мапа завантажується одразу після виклику функції запуску рушія. Ігровий процес офіційно починається з цього моменту, коли на рівні створюється відповідний ігровий режим і стан гри, а потім і інші актори[11].

У режимі редактора, який використовується режимами Play In Editor та Simulate In Editor, використовується інший потік. Рушій ініціалізується і запускається негайно, оскільки це необхідно для запуску редактора, але створення та ініціалізація об'єктів, таких як GameInstance, відкладається доти, доки користувач не натисне кнопку для запуску сеансу PIE або SIE. Крім того, актори на рівні дублюються, щоб зміни в грі не впливали на рівень у редакторі, і кожен об'єкт, включаючи об'єкт GameInstance, має окрему копію для кожного екземпляра PIE. Шлях редактора знову приєднується до автономного шляху з початком ігрового процесу у класі UWorld.

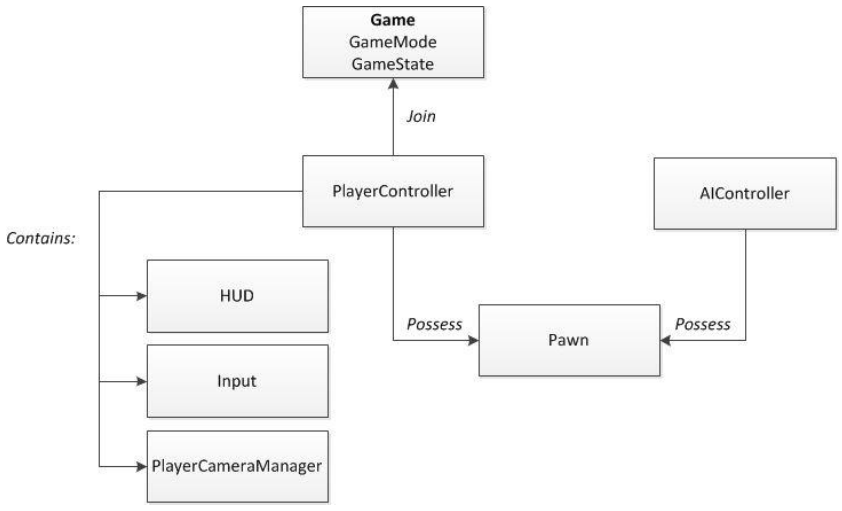


Рисунок. 2.2. Схема зв’язків основних класів

Взаємозв'язок класів фреймворку після початку гри. Блок-схема нижче ілюструє, як основні класи геймплею пов’язані один з одним. Гра складається з

GameMode та GameState. Гравці-люди, які приєднуються до гри, асоціюються з PlayerController'ами. Ці PlayerController дозволяють гравцям володіти пішаками у грі, щоб вони могли мати фізичне представлення на рівні. PlayerController також надає гравцям елементи керування введенням, головний дисплей, абоHUD, та PlayerCameraManager для керування видом з камери[12].

2.2. Проєктування функціональних можливостей гри

Гра повинна мати наступні функціональні можливості:

- вільне пересування гравця по мапі;
- саджання та збирання рослин;
- швидкодія програмного застосунку;
- збереження прогресу гри.

Функціонал головного меню:

- початку нової гри;
- продовження гри ;
- виходу з гри.

Функціонал меню паузи:

- повна зупинка гри;
- можливість продовження;
- вихід у головне меню;
- вихід з гри.

2.3. Моделювання ігрових компонентів

Екземпляри класу гравця використовуються для представлення нас у грі. Це буде об’єкт яким ми будемо керувати на протязі гри, тому він має в собі містити функції для переміщення та огляду. Також він буде містити у собі компоненту інвентарю, яка буде виконувати більшість роботи пов’язаної з геймплейом.

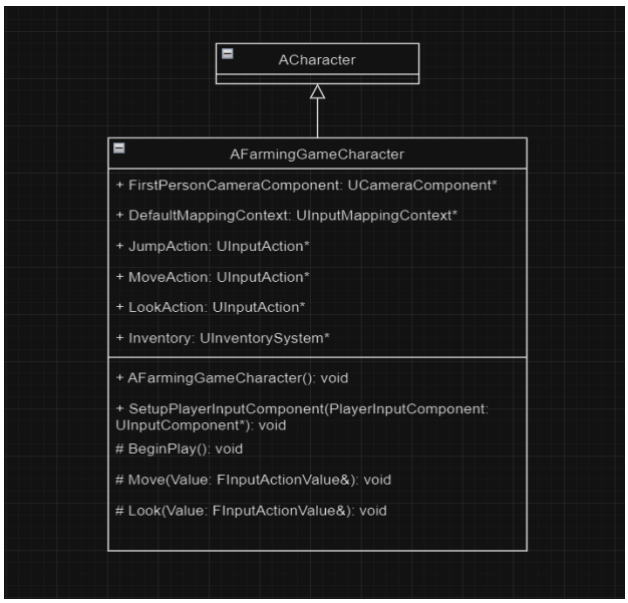


Рисунок. 2.3. UML-діаграма класу кравця

Компонента інвентару буде представляти інвентар гравця або ж скрині. Тому, вона обов’язково повинна мати поле Content у якому будуть зберігатись всі речі. Також компонента мусить мати функції додавання та видалення предмету з інвентару. Оскільки персонаж не маючи інвентару не зміг би взяти предмет до нього, логічно було б розмістити функціонал для взаємодії з предметами теж тут. Таким функціоналом буде функція, яка безпосередньо взаємодіє з предметами та функція, яка трасує простір перед гравцем.

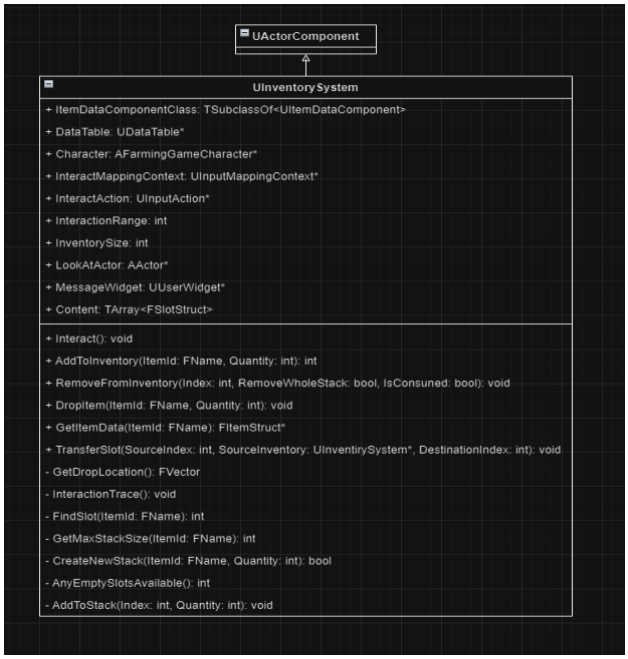


Рисунок. 2.4. UML-діаграма компоненти інвентару

Також для полегшення розробки будуть створені дві структури. Одна з яких буде представляти собою слот у інвентарі тому для цього їй потрібно поле для збереження імені предмету та поле для збереження кількості предметів у слоті, Друга ж буде використовуватись як основа у базі даних, тому вона повинна містити усю інформацію про предмет.

Рисунок. 2.5. UML-діаграма структур

Ієрархія класів рослин буде створена для симуляції процесу вирощування рослин у грі, де кожна рослина може мати різні характеристики та можливості взаємодії. Ієрархія включає кілька інтерфейсів та класів, що дозволяють організувати і розширювати функціональність системи. Далі буде детально описана структура та функціональність кожного компонента ієрархії.

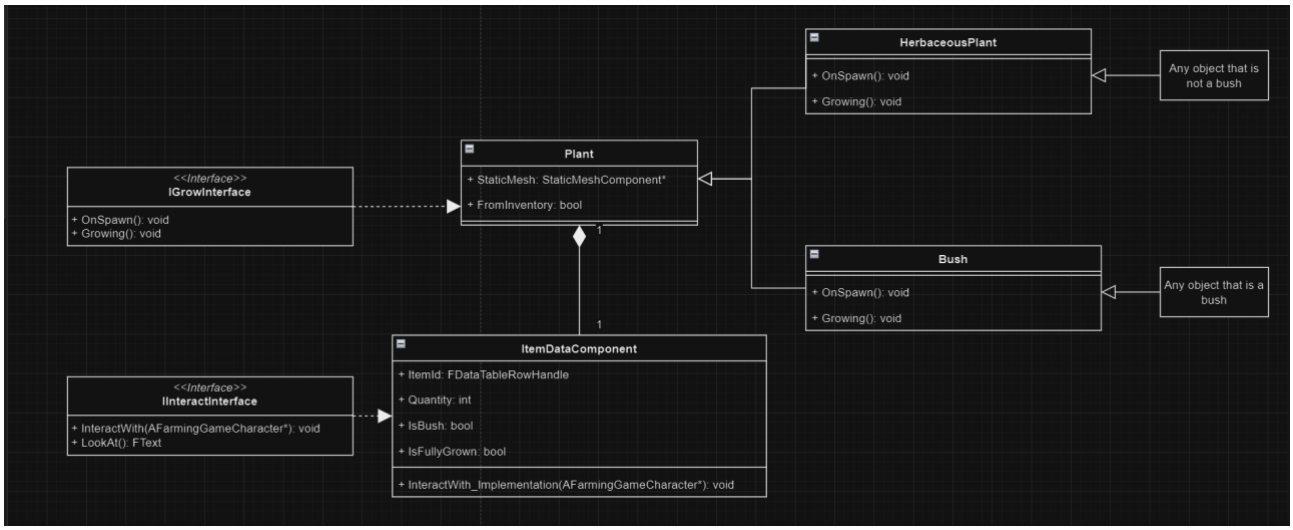


Рисунок. 2.6. UML-діаграма ієрархії класів для представлення рослин у грі

Інтерфейс IGrowInterface – визначає базові методи, які повинні реалізовувати всі об'єкти, що можуть рости:

- OnSpawn(): void – метод, що при потребі викликається при створенні об'єкта.
- Growing(): void – метод, що викликається для початку процесу росту об'єкта.

Інтерфейс IInteractInterface – визначає методи для взаємодії з об'єктами:

- InteractWith(AFarmingGameCharacter*): void – метод для взаємодії з об'єктом.
- LookAt(): FText – метод для отримання текстової інформації про об'єкт при погляді на нього.

Клас Plant – реалізує інтерфейс IGrowInterface і представляє базовий клас для всіх рослин. Він має такі властивості:

- StaticMesh: StaticMeshComponent* – компонент статичної моделі рослини.
- FromInventory: bool – вказує, чи був об'єкт доданий з інвентаря.

HerbaceousPlant – наслідується від Plant і представляє трав'янисті рослини, які зникають після збору врожаю. Він реалізує методи OnSpawn() і Growing() з інтерфейсу IGrowInterface.

Bush – також наслідує від Plant, але представляє кущі, які не зникають після збору врожаю. Він також реалізує методи OnSpawn() і Growing() з інтерфейсу IGrowInterface.

ItemDataComponent – компонент призначений для зберігання базових даних про об'єкти. Він включає такі властивості:

- ItemId: FDataTableRowHandle – унікальний ідентифікатор предмета, через який здійснюється пошук по базі даних.
- Quantity: int – кількість врожаю у об’єткі.
- IsBush: bool – вказує, чи є об'єкт кущем.
- IsFullyGrown: bool – вказує, чи повністю виріс об'єкт.

Цей компонент також реалізує методи з інтерфейсу IInteractInterface, а саме InteractWith_Implementation(AFarmingGameCharacter*): void – реалізація взаємодії з об'єктом.

Ця ієрархія класів забезпечить достатньо розширювану структуру для створення різних типів рослин у грі. Використання інтерфейсів дозволить легко додавати нові типи рослин та взаємодії без необхідності значних змін у існуючій кодовій базі. Компонент ItemDataComponent забезпечить зберігання та управління даними про об'єкти, що робить систему більш модульною та зручною для роботи.

Після завершення проєктування класів рослин, стало зрозуміло, що полів структури FItemStruct не достатньо для повного опису рослини, тому набір полів буде наступним:

- FText Name;
- FText Description;
- UTexture2D* Thumbnail;
- TSubclassOf<AActor> ItemClass;
- int StackSize;
- int TimeToHarvest;
- int TimeToC_Stage;
- int TimeToB_Stage;
- int TimeToA_Stage;
- UStaticMesh* SM;
- UStaticMesh* SM_Starter;
- UStaticMesh* SM_C;
- UStaticMesh* SM_C_Harvest;
- UStaticMesh* SM_B;
- UStaticMesh* SM_B_Harvest;
- UStaticMesh* SM_A;
- UStaticMesh* SM_A_Harvest.

РОЗДІЛ 3
РОЗРОБКА ІГРОВОГО ДОДАТКУ

3.1. Створення початкового проєкту

Для створення проєкту буде використовуватися Epic Games Launcher. У вкладці Unreal Engine вибираємо пункт Library та обираємо версію рушія:

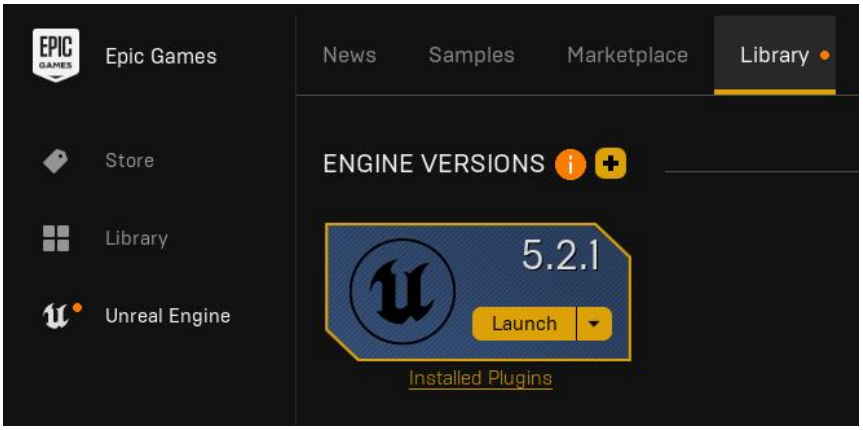


Рисунок. 3.1. Вікно запуску рушія у Epic Games Launcher

Після запуску рушія відкривається діалогове вікно для створення проєктів:

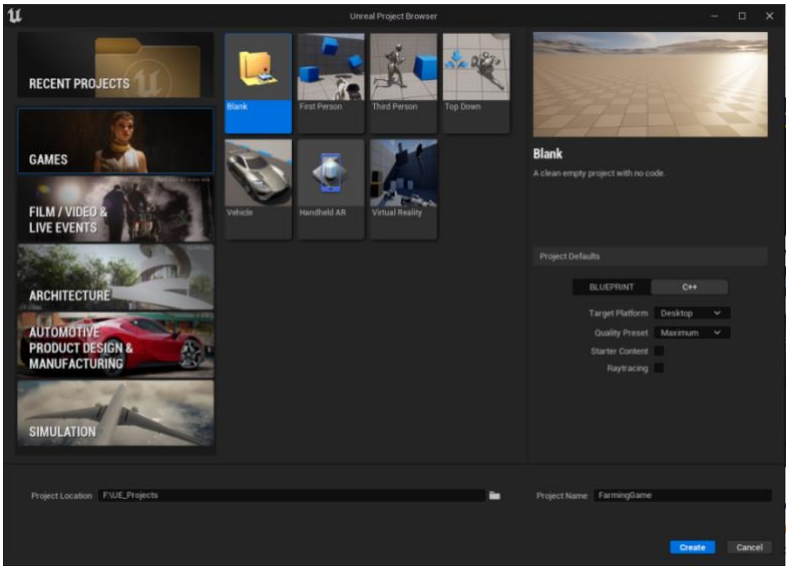


Рисунок. 3.2. Вікно запуску рушія у Epic Games Launcher

У ньому вибираємо Games -> Blank. Обираємо шлях до проєкту та називаємо його FarmingGame. У базових налаштуваннях створення обираємо C++ для можливості його використання та цільову платформу Desktop.

Також додаємо до нашого проєкту набір асетів з рослинами Ultimate Farming[13] та набір з будівлями Modular Medieval Town with Interior[14]:

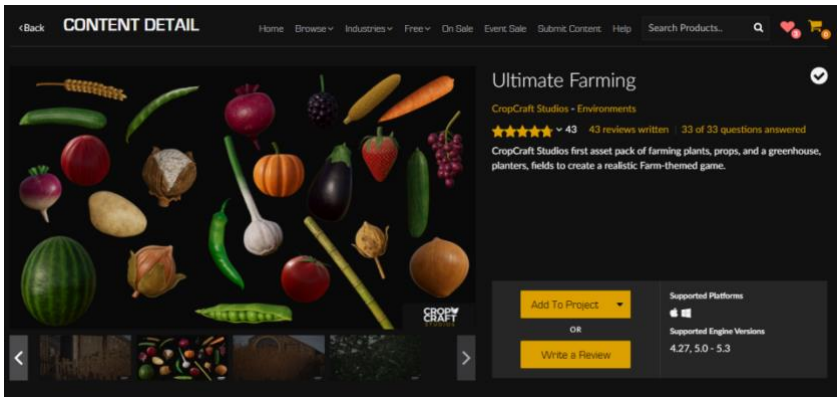


Рисунок. 3.3. Набір Ultimate Farming

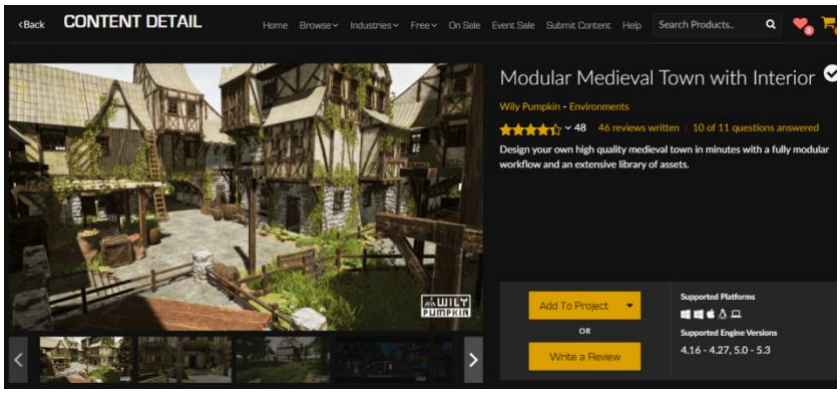


Рисунок. 3.4. Набір Modular Medieval Town with Interior

3.2. Створення ігрових локацій

Вперше відкриваючи проєкт цього шаблону ви потрапите на стандартну не збережену карту, але для нас вона завелика та потребує забагато часу, щоб привести її до ладу, тому користуватися нею ми не будемо. Для створення ігрової області будемо використовувати карту з доданого набору,

модифікувавши її під нас. Для модифікування карти ми змінюємо режим редактора на Landscape, комбінацією клавіш Shift+2 або обираємо відповідний режим у підменю[15].

Обравши інструмент Add додаємо нові компоненти збільшуючи нашу потенційну ігрову область. Після завершення розширення потрібно додати рельєфності, щоб карта не була пласкою. Це можна здійснити за допомогою інструментів з розділу Sculpt[16].

Далі створимо зону господарювання для нашого гравця використовуючи різні об’єкти з доданих наборів. Додамо будинок для гравця та декілька місць для саджання рослин. Перемістимо об’єкт Player Start до будинку гравця, щоб при запуску гри він з’являвся саме там. Для додавання рослинності будемо використовувати Foliage - це спеціальний режим, як Landscape, для швидкого збільшення кількості об’єктів, витрачаючи мінімальну кількість ресурсів для них[17].

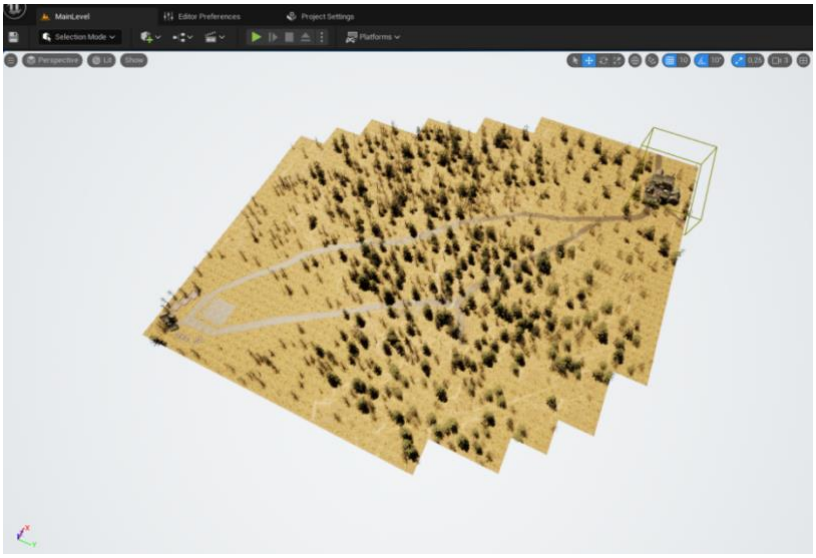


Рисунок. 3.5. Результат роботи над початковою картою

Встановимо створену карту, як карту за замовчуванням, щоб при відкриванні проєкту у редакторі відкривалась наша карта. За наступним шляхом Edit-> Project Settings-> Maps & Mods знаходимо налаштування Editor Startup Map та Game Default Map, змінюємо їх на нашу MainLevel.

3.3. Розробка класу гравця та ігрових механік

Для створення класу гравця, використовуючи Unreal Editor, створимо новий клас на основі класу Character та реалізуємо вище зазначене проєктування.

Наступним кроком буде реалізація методів для пересування, огляду та підписки на події руху.

Щоб персонаж мав змогу рухатись потрібно не тільки відповідні функції, а також валідні змінні, які будуть приймати вхідні дані з пристроїв введення. Для зручності створимо Blueprint нашого гравця, щоб задавати змінні саме там.

Нам потрібні 3 об’єкти InputAction для огляду, стрибку та руху з відповідними типами даних, а саме: Bool, Vector2D та Vector2D відповідно.

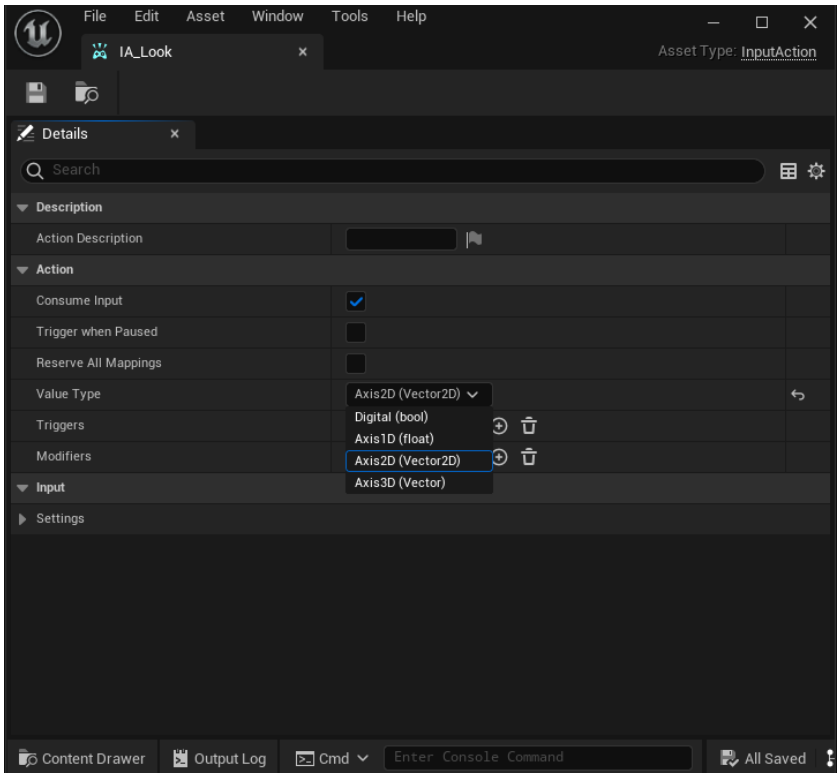


Рисунок. 3.6. Створення об’єкту InputAction

Також нам потрібний об’єкт InputMappingContext у якому співставимо створені нами InputAction та відповідальні клавіші пристроїв введення[18].

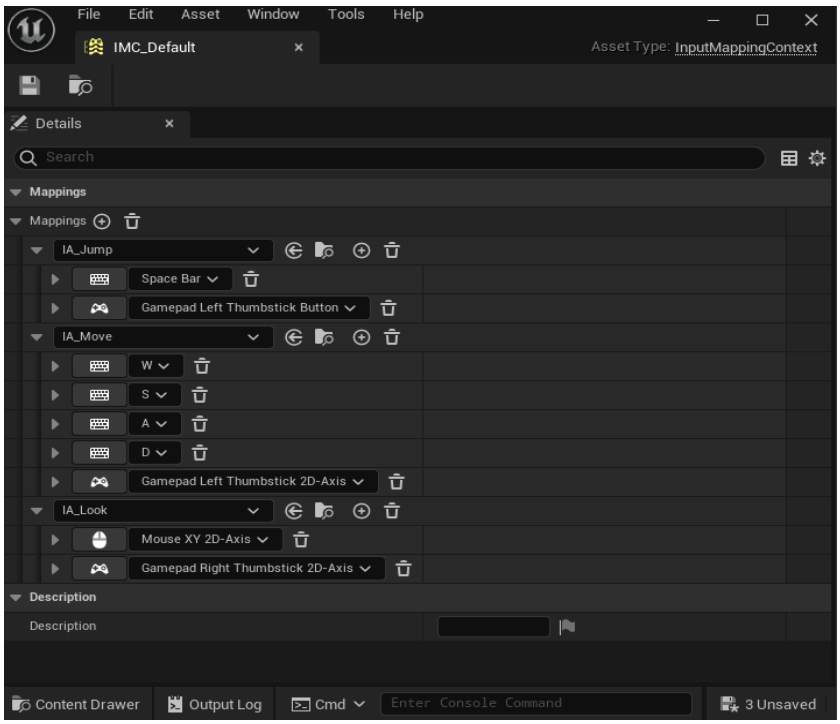


Рисунок. 3.7. Задання значень у InputMappingContext

Тепер у деталях нашого BP_FarmingGameCharacter встановимо ці об’єкти у відповідні властивості.

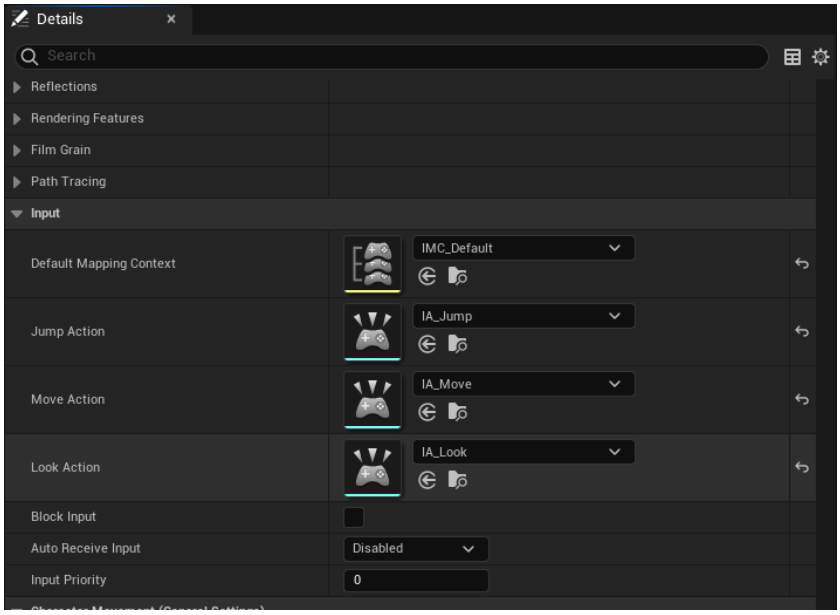


Рисунок. 3.8. Задання значень за замовчуванням у класі гравця

Якщо запустити гру, виявиться, що персонаж якого ми створили все ще спавниться. Щоб це виправити ми створимо Blueprint Class на основі класу

який створився сам при генерації проєкту та назвемо його BP_FarmingGame_GM. В деталях змінимо клас який генерується за замовчуванням на клас нашого героя.

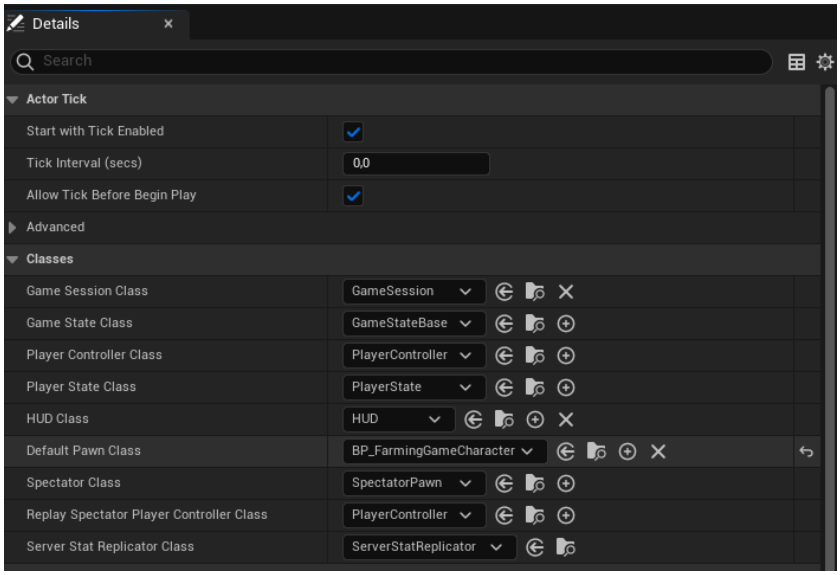


Рисунок. 3.9. Задання класу гравця за замовчуванням

У налаштуваннях проєкту змінимо GameMode на щойно створений та налаштований. Після цих дій, буде спавнитись створений нами клас.

Програмний код класу гравця, а саме заголовкового файлу та реалізацію методів у .cpp файлі можна переглянути у ДОДАТОК А.

Додамо механізм взаємодії з предметами. Для цього будемо використовувати Unreal Interface. Створимо новий інтерфейс наслідуючись від Unreal Interface та назвемо його IInteractInterface. Додамо у нього функцію LookAt та InteractWith, відмітивши їх специфікатором BlueprintNativeEvent для зручності подальшого використання[19].

Функція LookAt буде відповідати за відображення інформації про об’єкт, інформація може бути різна, оскільки кожен клас об’єкту буде перевизначати цю функцію, а отже і інформацію, що віддається.

Функція InteractWith буде відповідати безпосередньо за взаємодію між гравцем та предметом.

Наприклад, якщо об’єктом являється яблуко, то інформацією яку буде віддавати функція LookAt стане назва, а InteractWith буде переносити яблуко вам у інвентар.

```
FText ATESTActor::LookAt_Implementation()
{
    FString str = "Pick up {ItemName}";

    FFormatNamedArguments Args;

    if (!ItemDataComponent->ItemId.IsNull())
    {
        FName LocalRowName = ItemDataComponent->ItemId.RowName;
        FItemStruct* OutRow = ItemDataComponent->ItemId.DataTable->FindRow<FItemStruct>(LocalRowName, ContextString: "");
        if (OutRow != nullptr)
        {
            Args.Add( InKey: TEXT("ItemName"), OutRow->Name);
        }
    }

    return FText::Format( Fmt: FTextFormat::FromString(str), Args );
}
```

Рисунок. 3.10. Приклад реалізації функції LookAt

```
void UItemDataComponent::InteractWith_Implementation(AFarmingGameCharacter* Character)
{
    if (Character->Inventory->AddToInventory(ItemId.RowName, Quantity) >=0)
    {
        if (GetOwner() != nullptr)
        {
            GetOwner()->Destroy();
        }
    }
}
```

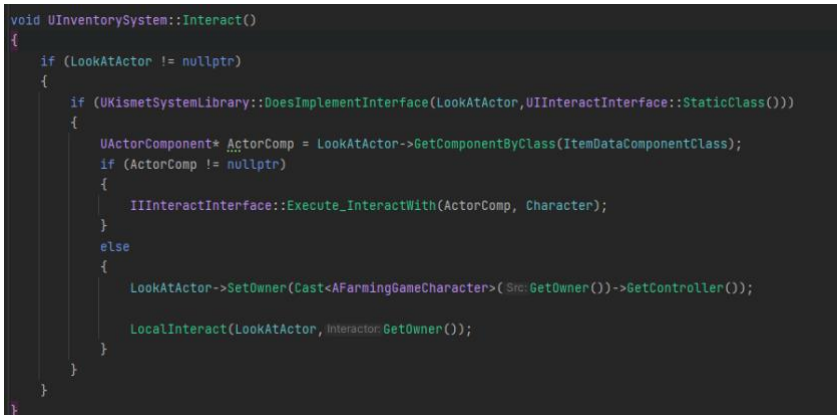
Рисунок. 3.11. Приклад реалізації функції InteractWith

Створення системи інвентару є ключовим елементом багатьох ігор, особливо тих, що включають елементи виживання, пригод або рольові ігри. Система інвентаря дозволяє гравцям збирати, зберігати та використовувати різні предмети, що вони знаходять під час гри. Інвентар, може бути, як у героя, так і до прикладу скрині. Тому, ми створимо компонент інвентару, якбий був спроєктований вище, що буде містити у собі всі його механіки, а не створювати дублікати у кожній сутності, яка його потребувала б.

Першим кроком є створення класу UInventorySystem, що успадковує UActorComponent. В цьому класі ми визначимо всі необхідні змінні та методи.

Для взаємодії з інвентарем, необхідно реалізувати метод Interact, що буде викликатися при натисканні відповідної клавіші або кнопки. Цей метод може включати логіку, що визначає, який об’єкт буде взаємодіяти з інвентарем, і яку

дію виконувати наприклад, підняти предмет, використати його або перемістити. У нашому ж випадку дією буде додавання предмету в інвентар або відкриття інвентару іншого об’єкту.

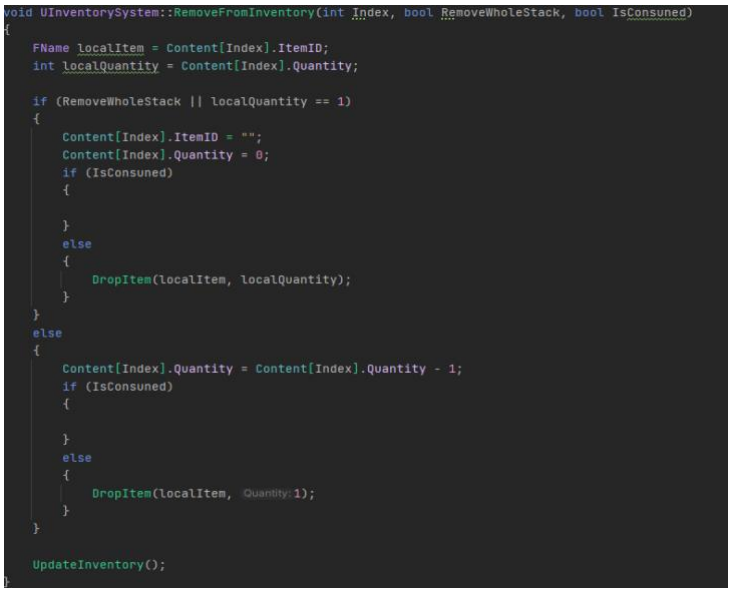


```
void UInventorySystem::Interact()
{
    if (LookAtActor != nullptr)
    {
        if (UKismetSystemLibrary::DoesImplementInterface(LookAtActor,UIInteractInterface::StaticClass()))
        {
            UActorComponent* ActorComp = LookAtActor->GetComponentByClass(ItemDataComponentClass);
            if (ActorComp != nullptr)
            {
                IIInteractInterface::Execute_InteractWith(ActorComp, Character);
            }
            else
            {
                LookAtActor->SetOwner(Cast<AFarmingGameCharacter>( Src: GetOwner())->GetController());

                LocalInteract(LookAtActor, Interactor: GetOwner());
            }
        }
    }
}
```

Рисунок. 3.12. Реалізації функції Interact

Також важливими методами є AddToInventory та RemoveFromInventory, які відповідають за додавання та видалення предметів з інвентаря. У їх реалізації беруть участь багато інших менших функцій для підтримкі коду в стані прийнятному для читання. Наприклад AddToInventory перевіряє наявність пустих слотів або існуючих стеків, що можуть бути заповнені, а RemoveFromInventory використовує функцію DropItem для викидання предмету у світ.



```
void UInventorySystem::RemoveFromInventory(int Index, bool RemoveWholeStack, bool IsConsuned)
{
    FName localItem = Content[Index].ItemID;
    int localQuantity = Content[Index].Quantity;

    if (RemoveWholeStack || localQuantity == 1)
    {
        Content[Index].ItemID = "";
        Content[Index].Quantity = 0;
        if (IsConsuned)
        {

        }
        else
        {
            DropItem(localItem, localQuantity);
        }
    }
    else
    {
        Content[Index].Quantity = Content[Index].Quantity - 1;
        if (IsConsuned)
        {

        }
        else
        {
            DropItem(localItem, Quantity: 1);
        }
    }

    UpdateInventory();
}
```

Рисунок. 3.12. Реалізації функції RemoveFromInventory

Наступним кроком буде реалізація структури FSlotStruct, яка була спроєктована раніше. У нашому класі створимо змінну масив Content з типом даних створеної структури, таким чином отримаємо місця збреження у інвентарі. Також реалізуємо структуру FItemStruct, за попереднім проєктуванням, яка буде містити інформацію про річ, яка зберігається.

Далі перейдемо до реалізації спроєктованої ієрархії класів рослин. В першу чергу створимо інтерфейс IGrowInterface. За цим створимо ItemDataComponent, який буде реалізовувати інтерфейс створений до цього IInteractInterface. Додаємо у нього змінні та перевизначимо функцію InteractWith наступним чином:

```
void UItemDataComponent::InteractWith_Implementation(AFarmingGameCharacter* Character)
{
    if (IsFullyGrown)
    {
        if (Character->Inventory->AddToInventory(ItemId.RowName, Quantity) >=0)
        {
            if (GetOwner() != nullptr)
            {
                if (!IsBush)
                {
                    GetOwner()->Destroy();
                }
                else
                {
                    Cast<UFarmingGameInstance>(Src: GetWorld()->GetGameInstance())->OnBushInteract.Broadcast(TargetActor: GetOwner());
                }
            }
        }
    }
}
```

Рисунок. 3.13. Реалізації функції InteractWith у ItemDataComponent

Важливо зазначити, що дана функція буде викликатися при спробі взаємодії з об’єктами класів наших рослиин, тобто при спробі збору врожаю. Тому дана реалізації функції забезпечить нам такі можливості, як:

- Перевірка чи рослина виросла;
- Додавання врожаю в інвентар;
- Зниження рослини, якщо це не кущ;
- Та оновлення таймеру дозрівання плодів, якщо рослина являється кущем.

Після цього, створимо клас Plant та дочірні йому класи. У Plant додамо потрібні поля, а у дочірніх класах встановимо значення за замовчуваннями та перевизначити функції з інтерфейсу IGrowInterface. До прикладу у ItemDataComponent класу Bush потрібно встановити IsBush за замовчуванням True.

Завершальним етапом створення функціоналу, ми вважаю є функція саджання. Дана функція реалізована у класі нашого гравця та викликається при натисканні лівої кнопки миші. Вона отримує усі об’єкти, які увійшли в колізію з нашим гравцем та намагається їх перетворити в потрібний нам компонен для саджання. При успішному перетворенні ми саджаємо рослину з насінини або саджанця та запускаємо процес росту видаляючи одну насінину з інвентару.

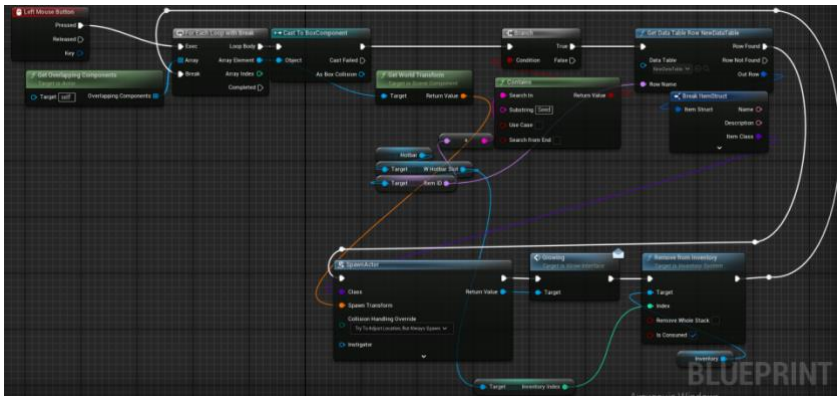


Рисунок. 3.14. Реалізації функції саджання у класі гравця

3.4. Проєктування інтерфейсу

UMG(Unreal Motion Graphics) є основним інструментом для створення інтерфейсів у Unreal Engine. Це повнофункціональний UI-редактор, що дозволяє розробникам створювати складні інтерфейси за допомогою простого перетягування елементів та додаванням логіки через Blueprints або C++[20].

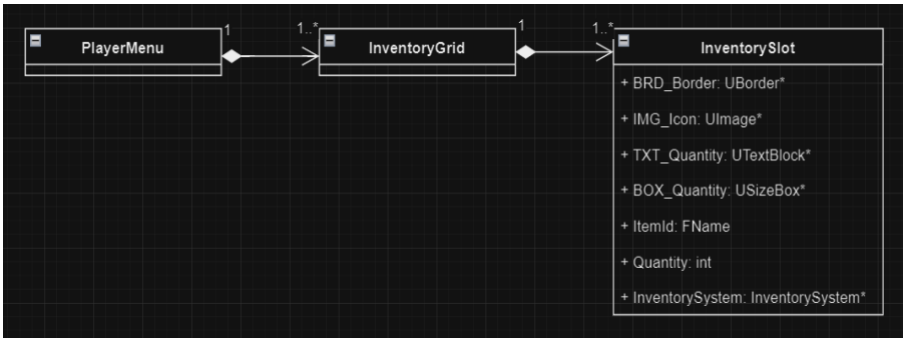


Рисунок. 3.15. Спроєктована ієрархія для інтерфейсу інвентару

Для прикладу розглянемо створення інтерфейсу інвентарю. Спроєктувавши наш інтерфейс приступаємо до його реалізації. Основними його

компонентами будуть: InventorySlot відповідає за окремий слот інвентарю, InventoryGrid містить сукупність усіх слотів та PlayerMenu відображає усе це для гравця одним об’єктом.

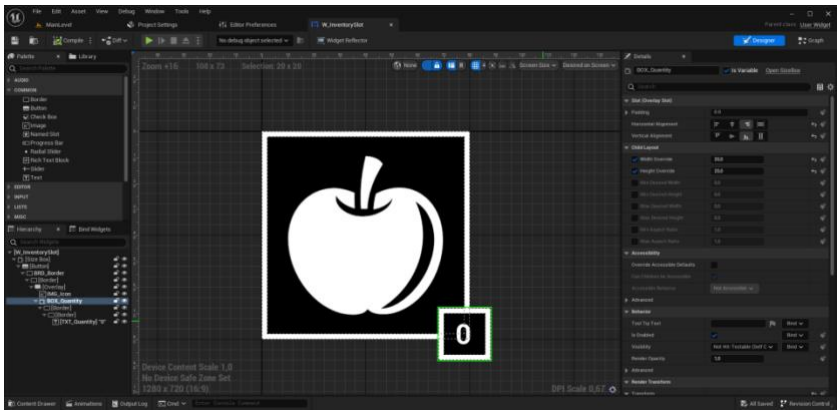


Рисунок. 3.14. Візуалізація InventorySlot

Усі вони повинні наслідуватись від класу UUsetWidget, для того щоб ми мали можливість змінювати його елементи вмісту використовуючи C++ клас, який теж повинен наслідуватись від UUsetWidget.

Після створення інтерфейсу візуально, потрібно реалізувати їх логіку. Для цього можна використати Blueprint, який пов’язаний з цим інтерфейсом, або змінити батьківський клас на потрібний йому C++ клас з реалізованою для нього логікою.

```
void UUW_InventorySlot::NativePreConstruct()
{
    Super::NativePreConstruct();

    if (DataTable != nullptr)
    {
        FItemStruct* OutRow1 = DataTable->FindRow<FItemStruct>(ItemID, ContextString: "");

        if (!ItemID.IsNone())
        {
            IMG_Icon->SetBrushFromTexture(OutRow1->Thumbnail);
            TXT_Quantity->SetText(FText::FromString(FString::FromInt(Quantity)));

            IMG_Icon->SetVisibility(ESlateVisibility::Visible);
            BOX_Quantity->SetVisibility(ESlateVisibility::Visible);
        }
        else
        {
            IMG_Icon->SetVisibility(ESlateVisibility::Hidden);
            BOX_Quantity->SetVisibility(ESlateVisibility::Hidden);
        }

    }
    else
    {
        IMG_Icon->SetVisibility(ESlateVisibility::Hidden);
        BOX_Quantity->SetVisibility(ESlateVisibility::Hidden);
    }
}
```

Рисунок. 3.15. Реалізована логіка роботи InventorySlot

3.5. Реалізація механізмів збереження та завантаження гри

Процес збереження гри. Значення поняття "збереження гри" може значно відрізнятися від гри до гри, але загальна ідея надання гравцям можливості вийти з гри, а потім продовжити гру з того місця, на якому вони зупинилися, є частиною більшості сучасних ігор. Залежно від типу гри, яку ви створюєте, вам може знадобитися лише кілька базових даних, таких як остання контрольна точка, до якої дійшов гравець, і, можливо, які предмети він знайшов. Або ж вам може знадобитися набагато детальніша інформація, наприклад, довгий список соціальних взаємодій гравця з іншими ігровими персонажами чи поточний статус різноманітних квестів, цілей місій або підсюжетів. Unreal Engine має систему збереження та завантаження, яка обертається навколо одного або декількох користувацьких класів SaveGame, які ви створюєте для задоволення конкретних потреб вашої гри, включаючи всю інформацію, яку вам потрібно зберегти протягом декількох ігрових сесій. Система підтримує можливість мати кілька збережених ігрових файлів і зберігати у них різні класи SaveGame. Це корисно для відокремлення глобально розблокованих функцій від даних гри, що стосуються конкретного проходження[21].

Також для збереження і завантаження використовують GameInstance, оскільки об’єкт цього класу створюється перед початком гри та знищується одним з останніх, що забезпечує доступ до нього на протязі всієї гри тільки в одному екземплярі.

Отже, для збереження і завантаження нашої гри, створимо клас на основі SaveGame, у який будемо зберігати дані з гравця. Також створимо клас на основі GameInstance та назвемо його FarmingGameInstance. Перевизначимо у ньому функцію Init, яка викликається при ініціалізації гри, щоб при запуску гра перевіряла чи існує слот для збереження гри, якщо “так” то зберігаємо дані у нього, а якщо ні, створюємо його.

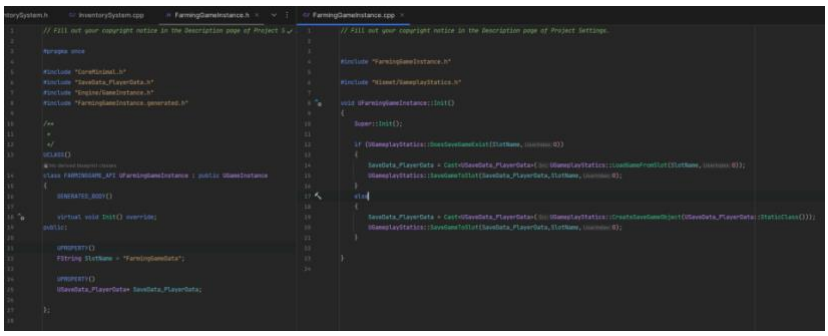


Рисунок. 3.16. Програмний код FarmingGameInstance

Після цього змінимо налаштування проєкту, щоб проєкт використовував щойно створений клас:

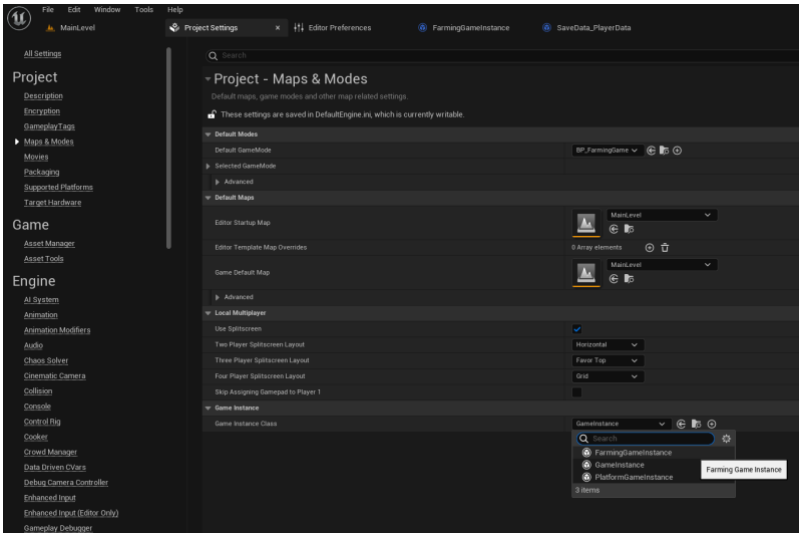


Рисунок. 3.17. Зміна налаштувань проєкту

Збереження інвентарю гравця буде реалізоване у InventorySystem, як дві функції, одна з яких буде завантажувати, а друга зберігати інвентар в момент його зміни шляхом підписки на делегат.

```
void UInventorySystem::SaveInventory()
{
    UFarmingGameInstance* GameInstance = Cast<UFarmingGameInstance>( Src: GetWorld()->GetGameInstance());
    GameInstance->SaveData_PlayerData->InventoryContent = Content;

    UGameplayStatics::SaveGameToSlot(GameInstance->SaveData_PlayerData, GameInstance->SlotName,  UserIndex: 0);
}

void UInventorySystem::LoadInventory()
{
    UFarmingGameInstance* GameInstance = Cast<UFarmingGameInstance>( Src: GetWorld()->GetGameInstance());

    Content = GameInstance->SaveData_PlayerData->InventoryContent;

}
```

Рисунок. 3.18. Реалізація механізму збереження та завантаження

```cpp
void UInventorySystem::BeginPlay()
{
    Super::BeginPlay();

    Character = Cast<AFarmingGameCharacter>( Src: GetOwner());
    if (Character != nullptr){...}

    FSlotStruct init;
    init.Quantity = 0;
    init.ItemID = FName();
    Content.Init(init,InventorySize);

    LoadInventory();

    OnInventoryUpdated.AddDynamic(this, &UInventorySystem::SaveInventory);
}
```

Рисунок. 3.19. Підписка на делегат на початку гри

За таким самим принципом, реалізуємо ці механізми у інших частинах нашого ігрового додатку

ВИСНОВКИ

Дипломна робота на тему «Комп’ютерна гра на основі рушія Unreal Engine» була успішно виконана.

У процесі виконання дипломної роботи було здійснено всебічний аналіз предметної області, пов’язаної з розробкою комп’ютерних ігор, що включав визначення, класифікацію та розгляд основних етапів створення ігор. Проведено дослідження існуючих ігрових рушіїв, які відіграють ключову роль у розробці сучасних ігор.

Особлива увага була приділена створенню концепцію, дослідженню загальної моделі гри, проєктуванню функціональних можливостей гри та моделюванню ігрових компонентів. Це дозволило створити чітке уявлення про структуру майбутнього ігрового додатку.

На етапі розробки було створено початковий проєкт гри, розроблені ігрові локації, клас гравця та основні ігрові механіки. Важливим аспектом розробки було проєктування інтерфейсу, що забезпечує зручність користування додатком, а також реалізація механізмів збереження та завантаження гри, що є критично важливими для сучасних ігрових додатків.

Кінцевим результатом роботи є демонстраційна версія комп’ютерної гри жанру симулятор із реалістичною графікою. Результати роботи можуть бути використані як основа для ігрових додатків. Дипломна робота демонструє практичну значущість отриманих знань та навичок у проєктуванні.

На наш погляд, основною перспективою подальших наукових розвідок є теми:

- оптимізація – оптимізація програмного продукту завжди була важливою темою як для розробників, так і для клієнтів. Ігри не є виключенням, оскільки чим краща отимізація тим більше охочих пограти;
- клієнт-серверної архітектури в рамках Unreal Engine – перспективність теми заснована на високій популярності онлайн ігор, які реалізовані саме таким чином.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Відеогра [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Відеогра.
2. The major differences between «indie» and «AAA» video games [Електронний ресурс] – Режим доступу до ресурсу: https://www.windowscentral.com/gaming/warhammer-40000-rogue-traders-first-dlc-expansion-is-revealed.
3. В Україні на 66% більше нових інді-ігор. Головне зі звіту Unity про ігрову індустрію [Електронний ресурс] – Режим доступу до ресурсу: https://gamedev.dou.ua/news/unity-gaming-report-2023-trend-and-changes/.
4. Ігровий рушій [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Ігровий_рушій.
5. Unity [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Unity_(ігровий_рушій).
6. Game Maker:Studio [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/GameMaker:_Studio.
7. Godot [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Godot.
8. Як стати Unreal Engine Developer та розробляти ігри. Кар’єра в геймдеві [Електронний ресурс] – Режим доступу до ресурсу: https:/gamedev.dou.ua/articles/unreal-engine-developer-in-gamedev.
9. Unreal Engine 5.0 - Unreal Engine Public Roadmap [Електронний ресурс] – Режим доступу до ресурсу: https:/portal.productboard.com/epicgames/1-unreal-engine-public-roadmap/tabs/46-unreal-engine-5-0.
10. Game Engine | Build Multi-Platform Video Games - Game Engine [Електронний ресурс] – Режим доступу до ресурсу: https://www.unrealengine.com/en-US/uses/games.
11. Game Flow Overview [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/GameFlow.

12. Unreal Architecture [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture.
13. Ultimate Farming [Електронний ресурс] – Режим доступу до ресурсу: https://marketplace-website-node-launcher-prod.ol.epicgames.com/ue/marketplace/en-US/product/ultimate-farming.
14. Modular Medieval Town with Interior [Електронний ресурс] – Режим доступу до ресурсу: https://marketplace-website-node-launcher-prod.ol.epicgames.com/ue/marketplace/en-US/product/modular-medieval-town-with-interior.
15. Creating Landscapes [Електронний ресурс] – Режим доступу до ресурсу: https://dev.epicgames.com/documentation/en-us/unreal-engine/creating-landscapes-in-unreal-engine?application_version=5.2.
16. Sculpt Mode [Електронний ресурс] – Режим доступу до ресурсу: https://dev.epicgames.com/documentation/en-us/unreal-engine/landscape-sculpt-mode-in-unreal-engine?application_version=5.2.
17. Foliage Mode [Електронний ресурс] – Режим доступу до ресурсу: https://dev.epicgames.com/documentation/en-us/unreal-engine/foliage-mode-in-unreal-engine?application_version=5.0.
18. Enhanced Input [Електронний ресурс] – Режим доступу до ресурсу: https://dev.epicgames.com/documentation/en-us/unreal-engine/enhanced-input-in-unreal-engine?application_version=5.2.
19. Implement Interface in Blueprint [Електронний ресурс] – Режим доступу до ресурсу: https://dev.epicgames.com/documentation/en-us/unreal-engine/implementing-blueprint-interfaces-in-unreal-engine.
20. Widget Blueprints [Електронний ресурс] – Режим доступу до ресурсу: https://dev.epicgames.com/documentation/en-us/unreal-engine/widget-blueprints-in-umg-for-unreal-engine?application_version=5.0.

21. Saving and Loading Your Game [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unrealengine.com/5.0/en-US/saving-and-loading-your-game-in-unreal-engine.

ДОДАТКИ

ДОДАТОК А

Програмний код класу гравця

Програмний код з файлу FarmingGameCharacter.h:

```
#pragma once
#include "CoreMinimal.h"
#include "Camera/CameraComponent.h"
#include "GameFramework/Character.h"
#include "InputActionValue.h"
#include "InventorySystem.h"
#include "FarmingGameCharacter.generated.h"

UCLASS()
class FARMINGGAME_API AFarmingGameCharacter : public ACharacter
{
   GENERATED_BODY()

   UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta =
(AllowPrivateAccess = "true"))
   UCameraComponent* FirstPersonCameraComponent;

   UPROPERTY(EditAnywhere, BlueprintReadOnly, Category=Input,
meta=(AllowPrivateAccess = "true"))
   class UInputMappingContext* DefaultMappingContext;

   UPROPERTY(EditAnywhere, BlueprintReadOnly, Category=Input,
meta=(AllowPrivateAccess = "true"))
   class UInputAction* JumpAction;

   UPROPERTY(EditAnywhere, BlueprintReadOnly, Category=Input,
meta=(AllowPrivateAccess = "true"))
   class UInputAction* MoveAction;

   UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta =
(AllowPrivateAccess = "true"))
   class UInputAction* LookAction;

public:
   AFarmingGameCharacter();

   UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = Inventory)
   UInventorySystem* Inventory;
```

```
protected:
    virtual void BeginPlay() override;

    void Move(const FInputActionValue& Value);

    void Look(const FInputActionValue& Value);

public:
    virtual void Tick(float DeltaTime) override;

    virtual void SetupPlayerInputComponent(class UInputComponent*
PlayerInputComponent) override;

};
```

Програмний код з файлу FarmingGameCharacter.cpp:

```
#include "FarmingGameCharacter.h"
#include "EnhancedInputComponent.h"
#include "EnhancedInputSubsystems.h"
#include "Components/CapsuleComponent.h"

AFarmingGameCharacter::AFarmingGameCharacter()
{
    PrimaryActorTick.bCanEverTick = true;

    GetCapsuleComponent()->InitCapsuleSize(40.f, 80.0f);

    FirstPersonCameraComponent =
CreateDefaultSubobject<UCameraComponent>(TEXT("FirstPersonCamera"));
    FirstPersonCameraComponent->SetupAttachment(GetCapsuleComponent());
    FirstPersonCameraComponent->SetRelativeLocation(FVector(-10.f, 0.f, 60.f));
    FirstPersonCameraComponent->bUsePawnControlRotation = true;

    Inventory = CreateDefaultSubobject<UInventorySystem>(TEXT("Inventory"));
}

void AFarmingGameCharacter::BeginPlay()
{
    Super::BeginPlay();

    if (APlayerController* PlayerController = Cast<APlayerController>(Controller))
    {
```

```
    if (UEnhancedInputLocalPlayerSubsystem* Subsystem =
ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerContr
oller->GetLocalPlayer()))
    {
      Subsystem->AddMappingContext(DefaultMappingContext, 0);
    }
  }
}

void AFarmingGameCharacter::Tick(float DeltaTime)
{
  Super::Tick(DeltaTime);

}

void AFarmingGameCharacter::SetupPlayerInputComponent(UInputComponent*
PlayerInputComponent)
{
  Super::SetupPlayerInputComponent(PlayerInputComponent);

  if (UEnhancedInputComponent* EnhancedInputComponent =
CastChecked<UEnhancedInputComponent>(PlayerInputComponent))
  {
    EnhancedInputComponent->BindAction(JumpAction,  ETriggerEvent::Triggered,
this, &ACharacter::Jump);
    EnhancedInputComponent->BindAction(JumpAction,
ETriggerEvent::Completed, this, &ACharacter::StopJumping);

    EnhancedInputComponent->BindAction(MoveAction, ETriggerEvent::Triggered,
this, &AFarmingGameCharacter::Move);
    EnhancedInputComponent->BindAction(LookAction, ETriggerEvent::Triggered,
this, &AFarmingGameCharacter::Look);

  }
}

void AFarmingGameCharacter::Move(const FInputActionValue& Value)
{
  FVector2D MovementVector = Value.Get<FVector2D>();

  if (Controller != nullptr)
  {
    AddMovementInput(GetActorForwardVector(), MovementVector.Y);
    AddMovementInput(GetActorRightVector(), MovementVector.X);
  }
```

```
}

void AFarmingGameCharacter::Look(const FInputActionValue& Value)
{
    FVector2D LookAxisVector = Value.Get<FVector2D>();

    if (Controller != nullptr)
    {
        AddControllerYawInput(LookAxisVector.X*(-1));
        AddControllerPitchInput(LookAxisVector.Y);
    }
}
```