UNICHECK

Ім'я користувача:
Олена Бондаренко

Дата перевірки:
31.05.2023 12:08:39 EEST

Дата звіту:
31.05.2023 12:10:13 EEST

ID перевірки:
1015342029

Тип перевірки:
Doc vs Internet + Library

ID користувача:
100005535

Назва документа: **Pohuliaka_Iehor_Vitaliiovych_PZ20130**

Кількість сторінок: **79** Кількість слів: **11333** Кількість символів: **98490** Розмір файлу: **9.30 MB** ID файлу: **1015010561**

18.6%
Схожість

Найбільша схожість: **6.57%** з джерелом з Бібліотеки (ID файлу: **1014971838**)

15.5% Джерела з Інтернету	425	Сторінка 81
7.82% Джерела з Бібліотеки	185	Сторінка 84

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет «Комп'ютерні технології і системи»

Кафедра «Комп'ютерні інформаційні технології»

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему: «Розробка ядра комп’ютерної гри жанру пригод на Unreal Engine»
за освітньою програмою: «Інженерія програмного забезпечення»
зі спеціальності: «121 Інженерія програмного забезпечення»
Виконав: студент групи «ПЗ20130»

	(підпис студента)	/Єгор ПОГУЛЯКА/ (Ім’я ПРІЗВИЩЕ)
Керівник:	(підпис)	/доц. Олександр ІВАНОВ/ (посада, Ім’я ПРІЗВИЩЕ)
Нормоконтролер:	(підпис)	/доц. Світлана ВОЛКОВА/ (посада, Ім’я ПРІЗВИЩЕ)

Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Дніпро – 2023 рік

Ministry of Education and Science of Ukraine
Ukrainian State University of Science and Technologies

Faculty «Computer technologies and systems»

Department «Computer information technology»

Explanatory Note

to Bachelor's Thesis

on the topic: «Development of the core of an adventure PC game on Unreal Engine»
according to educational curriculum «Software engineering»
in the Speciality: «121 Software engineering»

Done by the student of the group PZ20130: /Yehor POHULIAKA/

Scientific Supervisor: /Oleksandr IVANOV/

Normative controller: /Svitlana VOLKOVA/

Dnipro – 2023

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: Факультет «Комп'ютерні технології і системи»
Кафедра: «Комп'ютерні інформаційні технології»
Рівень вищої освіти: бакалавр
Освітня програма: «Інженерія програмного забезпечення»
Спеціальність: «121 Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри КІТ
_____ /Вадим ГОРЯЧКІН/
(підпис)

Дата _____

З А В Д А Н Н Я

на кваліфікаційну роботу бакалавра
студенту Погуляці Єгору Віталійовичу
1. Тема роботи: «Розробка ядра комп’ютерної гри жанру пригод на Unreal Engine»
Керівник роботи: Іванов Олександр Петрович, доцент
затверджені наказом № 1209 ст від 07.12.2022
2. Строк подання студентом роботи: __.__.202_ р.
3. Вихідні дані до роботи: _____

4. Зміст пояснювальної записки (перелік питань, які потрібно опрацювати):
Вступ
Аналіз предметної області
Організація команди та середовища розробки
Розробка ігрових механік та систем гри
Тестування відеогри
Загальні висновки

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень):
Презентація
Відео роботи програми

КАЛЕНДАРНИЙ ПЛАН

Стадія	Зміст	Строки виконання
Технічне завдання	Постановка задачі, збір інформації, виріб та обґрунтування критеріїв розробки. Попередній вибір методів рішення задач. Визначення вимог до технічних засобів. Узгодження і затвердження технічного завдання.	07.12.22 -26.12.22
Робочий проект	Програмування та відлагодження програми.	27.12.22 - 27.04.23
	Тестування програми	28.04.23 - 05.05.23
	Розробка, узгодження і затвердження програмної документації.	06.05.23 - 19.06.23
	Подання кваліфікаційної роботи до кафедри	14.06.23
	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	28.06.23

Студент _____ Єгор ПОГУЛЯКА
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник роботи _____ доц. Олександр ІВАНОВ
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка складається з 7 розділів:

– вступ – в даному розділі описується сутність розробки, її актуальність. Складається з 2 сторінки;

– аналіз предментої області – у цьому розділі ми розглядаємо та аналізуємо ігрові движки, існуючі жанри ігор, атмосферу та ігрові механіки в різних іграх та складаємо постановку подальшої роботи. Складається з 9 сторінок;

– організація команди та середовища – в цьому розділі розглядаються методи організації команди, створення дорожньої карти, види систем контролю версій, а також Blueprints та Behavior Trees в Unreal Engine. Складається з 10 сторінок;

– розробка ігрових механік та систем гри – у цьому розділі розробляються базові ігрові механіки та системи гри. Складається з 24 сторінок;

– тестування ігор – у цьому розділі розглядаються базові стратегії та особливості тестування ігор, а також приводяться тестові випадки. Складається з 5 сторінок.

– загальні висновки – підсумки всієї роботи. Складається з 1 сторінки;

– список використаних джерел – включає в себе бібліографічний список використаної літератури. Складається з 1 сторінки;

– додатки – містить технічне завдання та код робочого проекту. Кількість таблиць: 3 штук. Кількість рисунків: 34 штуки.

ЗМІСТ

ВСТУП

За останні роки ігрова індустрія стала однією з швидкозростаючих і найприбутковіших у світі розваг. Сьогодні ігри доступні на багатьох платформах, включаючи персональні комп'ютери, ігрові консолі та мобільні пристрої. Одним з головних факторів успіху у цій сфері є якість гри. Гарна гра має мати цікавий сюжет, захоплюючий сюжет, красиву графіку та звуковий супровід.

Unreal Engine – це потужний ігровий рушій, який розроблений компанією Epic Games. Він має широкий функціонал, який включає до себе можливості праці з 3D-моделями, текстурами, матеріалами, освітленням та багато чого іншого. На ринку користується великою популярністю за рахунок своєї гнучкості та потужності, а також завдяки тому, що він доступний для використання як для невеликих інди-команд, так і для великих компаній.

Цей дипломний проєкт призначений для демонстрації створення ядра ігрової логики в комп’ютерній грі “Nuclear Frost” на рушії Unreal Engine, розгляне ключові концепції та особливості праці з Unreal Engine і продемонструє практичні приклади того, як його можна використовувати для створення головних систем ігрової логики.

Nuclear Frost – це пригодницький шутер від першої особи, де герою доведеться зіткнутися віч-на-віч із погрозами, які породив новий світ. Боріться з мутантами, зберігайте своє тепло, уникайте радіації та боріться з грибком, щоб урятувати свій будинок. Світ гри давним-давно був зруйнований ядерною війною, що знищила населення Землі і перетворила її поверхню на крижану пустку. Лише жменька тих, хто вижив, сховалася в надрах бункера, а людська цивілізація виявилася на межі вимирання через грибок Крижаної Орусфаїри, що з'явився на світ.

Головний герой, за якого буде грати гравець, не застав самої війни, він народився вже в бункері і лише з розповідей знав який світ був "До". Весь час його житло зазнавало спалахів епідемії Льодяної Орусфаїри, які вдавалося лі-

квідувати, але тільки не зараз. Ситуація загострилася настільки, що майже більшість населення була заражена і герой у тому числі. Настав момент рішуче діяти, тому перед героєм поставлене завдання врятувати за будь-яку ціну свій рідний дім, навіть якщо за це доведеться віддати своє життя.

Головна мета гри полягає в тому, щоб занурити гравця в атмосферу замерзлого світу та боротьби людей за своє існування, а також показати, наскільки може бути небезпечним застосування ядерної зброї та зневага проблем клімату.

Загалом цей дипломний проєкт прагне забезпечити всебічне розуміння створення ядра ігрової логики у розробці гри “Nuclear Frost” та продемонструвати можливості Unreal Engine.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Аналіз ігрових двигунів

Сьогодні на ринку існує безліч ігрових рушіїв, які розробники ігор можуть використовуватия для створення комп'ютерних ігор, але деякі з них найпопулярніші і широко використовуються в індустрії ігор. Прикладами таких продуктів є Unreal Engine, Unity і CryEngine.

Однак Unreal Engine виділяється своєю чудовою графікою, сістемою освітлення та вбудованими інструментами, які облегчують процес розробки та дозволяють розробникам створювати ігри швидше да більш ефективніше.

1. Аналіз можливостей Unity

Unity – це популярний ігровий рушій із закритим ісходним кодом, який використовується для створення комп'ютерних ігор на різних платформах, включаючи PC, консолі та мобільні пристрої, визуальний вигляд інтерфейсу представленний на рис. 1.1. Однак Unreal Engine має відкритий ісходний код, що дозволяє при необхідності модіфікувати двигун, а також у ньому реалізовано більш якісна система освітлення, що робить графіку гри привабливіше.

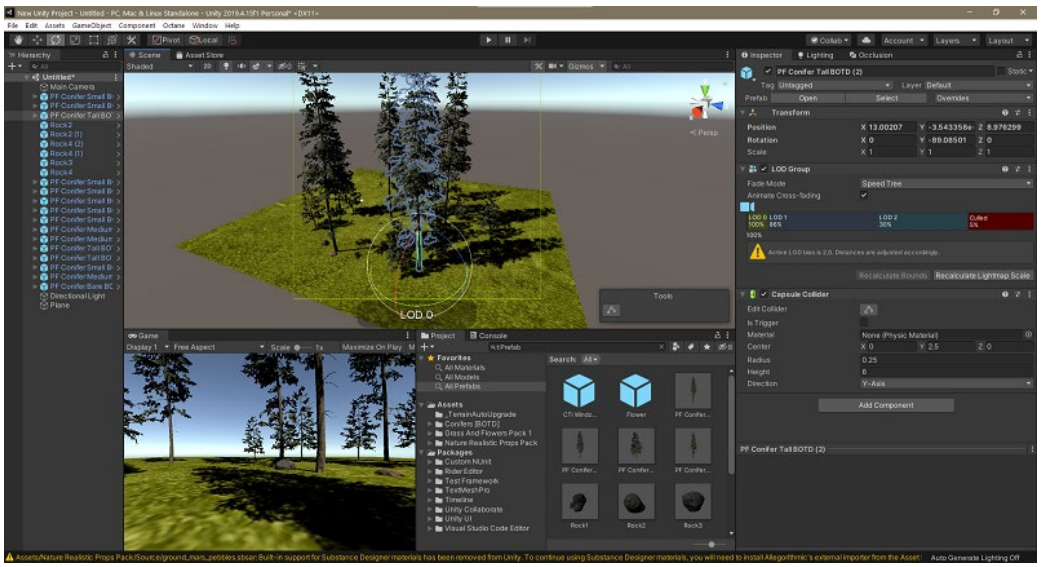

Рис. 1.1. Візуальний вигляд Unity

2. Аналіз можливостей CryEngine

CryEngine – це потужний ігровий рушій, який використовується для створення комп'ютерних ігор на різних платформах, включаючи PC та консолі, візуальний вигляд інтерфейсу представлений на рис. 1.2. Він має дуже гарну графіку завдяки чьому проєкти на ньому мають високу ступінь реалізма, а за допомогою системи плагинів можливо розширяти двигун, що дозволяє розробникам додавати нові функції та можливості. Однак, це рішення має деякі недоліки. Наприклад, CryEngine достатньо складний у використанні для початківців. Крім цього, він не так широко використовується у ігровій індустрії, як Unreal Engine чи Unity, це може означати, що його екосистема та спільнота розробників могуть бути не такими розвинутими.

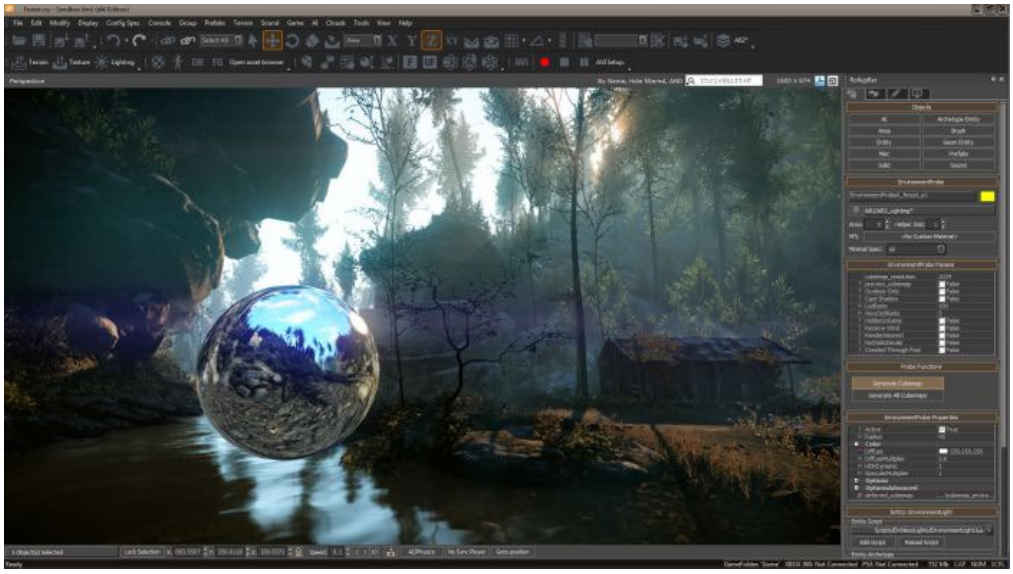

Рис. 1.2. Візуальний вигляд CryEngine

1.2. Аналіз жанрів ігор

Жанр відеогри використовується для класифікації відеоігор відповідно до інтерактивних ігрових дій гравця. Жанр відеогри не залежить від сценарію або змісту уявного ігрового світу, на відміну від творів літератури чи кіномистецтва. Єдиної класифікації жанрів відеоігор не існує, тому в різних джерелах одну й ту саму відеогру можуть відносити до різних. Також можливі змішання жанрів, коли гру неможливо віднести до одного жанру.

В основі сучасних розподілів відеоігор на жанри лежить вид активності, який найчастіше здійснює гравець в іграх даного жанру. Так відеоігри в загальному можуть поділятися на ігри руху, планування і сюжету або спілкування, дії та контролю. В багатьох класифікаціях визначення жанру відбувається за кількома вісями. Наприклад, за двома осями сюжет – свобода дії, або трьома абстракція – симуляція – свобода. Проте найчастіше використовуваною класифікацією, хоч і не прийнятою усіма, жанри з якої зустрічаються в більшості існуючих, є наведена нижче, яка виключає осі або багаторівневі поділи.

Екшен – в іграх такого жанру необхідно використовувати рефлекси та швидкість реакції для подолання ігрових обставин. Це один із базових жанрів і водночас найпоширеніший. Як правило екшн-ігри пов'язані із агресивними діями щодо противників і/або оточення. Персонаж гравця повинен битися, стріляти, переслідувати ціль чи самому уникати переслідування. Стосовно екшн-ігор, де наявні значні елементи пригодницьких ігор, застосовується термін Action-adventure. З-поміж них часто виділяється напрям аркадних ігор, ігровий процес яких вирізняється простотою та легкістю освоєння.

Цей жанр поділяється на велику кількість піджанрів, серед яких основними є:

– шутери – вимагають від гравця боротися з суперником шляхом стрілянини. Залежно від перспективи, поділяються на шутери від першої чи третьої особи. Існують різновиди як тактичні, в яких ігровий персонаж діє у складі команди, аркади, стелс-екшн, метою якого є приховані дії для виконання завдань, без прямого знищення противників. Щодо ігор, де основою ігрового процесу є знищення великих кількостей ворогів, а сама стрілянина в цьому переважає над тактикою і влучністю, застосовується термін Shoot 'em up;

– файтинги – імітують ближній бій, як правило один на один, на спеціальних аренах;

– beat 'em up – подібні на файтинги, з тією різницею, що персонажі вільно переміщуються ігровим світом (а не спеціальною ареною) і борються проти багатьох противників одночасно;

– платформери – персонаж мусить рухатися, стрибаючи по платформах, та долати перешкоди;

– лабіринти – персонаж рухається лабіринтом з метою знайти вихід, зібрати предмети і/або уникнути пасток і небезпек. Часто в іграх цього жанру є обмеження на час.

Стратегія – сенс стратегічних ігор полягає в плануванні дій та виробленні певної стратегії для досягнення якоїсь конкретної мети, наприклад, перемоги у військовій операції. Гравець керує не одним персонажем, а цілим підрозділом, підприємством чи навіть всесвітом. Відповідно до реалізації ігрового часу, стратегічні відеоігри поділяються на два основних різновиди:

– покрокові стратегічні ігри – гравець та його противник здійснюють дії один за одним, покроково, маючи змогу за один ігровий хід виконати певну кількість операцій;

– стратегічні ігри в реальному часі – і гравець і суперник виконують свої дії одночасно, проте часто масштаб часу відрізняється від реального. Наприклад, будівництво триває кілька секунд, а ігрова година складає кілька хвилин реального часу.

Пригоди – в пригодницьких відеоіграх гравець керує ігровим персонажем, який рухається по сюжету та виконує зумовлені сценарієм завдання, покладаючись на свою уважність та логіку, здійснює пошуки підказок і вирішує загадки. Всередині жанру виділяються основні піджанри: інтерактивна література, інтерактивні фільми та візуальні романи. Часто за аналогією до пригодницьких фільмів пригодницькими називаються ті відеоігри, сюжет яких динамічно розгортається, насичений яскравими подіями, швидкою зміною обстановки, а персонажі проявляють кмітливість та сміливість, а не грубу силу.

Симулятор – в широкому розумінні всі ігри є симуляторами. У вужчому значенні це відеоігри, призначені для складання уявлення про дійсність за допомогою відображення певних реальних явищ та властивостей у віртуальному середовищі. Існує чимало піджанрів, як технічні (управління складними технічними пристроями, авіаційною технікою та інші аркадні, спортивні, економічні та інші.

1.3. Аналіз атмосфери та ігрових механік в різних іграх

1. Cryostasis: Sleep of Reason

Cryostasis: Sleep of Reason – це комп’ютерна гра у жанрі жах виживання, вона розроблена українською компанією Action Forms. Сюжет гри розповідає про моряка на ім’я Александр Нестеров, який опиняється на загубленій на льоду арктичної станції "Полярний", де гравець повинен виживати в екстремальних умовах холоду та снігових бур.

Головною особливістю гри є ігрова механіка холоду. Протягом усієї гри, гравець повинен слідкувати за температурою свого героя, щоб уникнути переохолодження. Коли персонаж знаходиться у холодному місці, то його термометр починає опускатись, а коли він достигне критичної відмітки, то головний герой почне замерзати та його здоров’я буде зменьшуватись.

Щоб уникнути переохолодження, гравець повинен знаходити джерела тепла (двигун, лампочка, багаття та інше) та грітися біля них, щоб підтримувати температуру тіла на необхідному рівні. На рис. 1.3 зображено як виглядає механіка холода у грі.

Другою особливістю гри є її атмосфера. Оточення холоду було дуже деталізовано відтворено розробниками. Ігровий світ заповнений кучугурами снігу та крижаними утвореннями, які можна розглянути навіть у найдрібніших деталях. У грі також присутні широкі та просторі приміщення на станції, де гравець може насолодитися чудовими видами льодовиків та снігових пусток.

Крім цього, гра використовує ефекти замерзання, щоб створити відчуття холоду та самотності в ігровому світі. Наприклад, коли персонаж знаходиться в холодному місці, гравець може побачити, як його дихання утворює хмари пари, а звуки навколишнього середовища стають приглушеними та моторошними. На рис. 1.4 зображена атмосфера гри Cryostasis: Sleep of Reason.

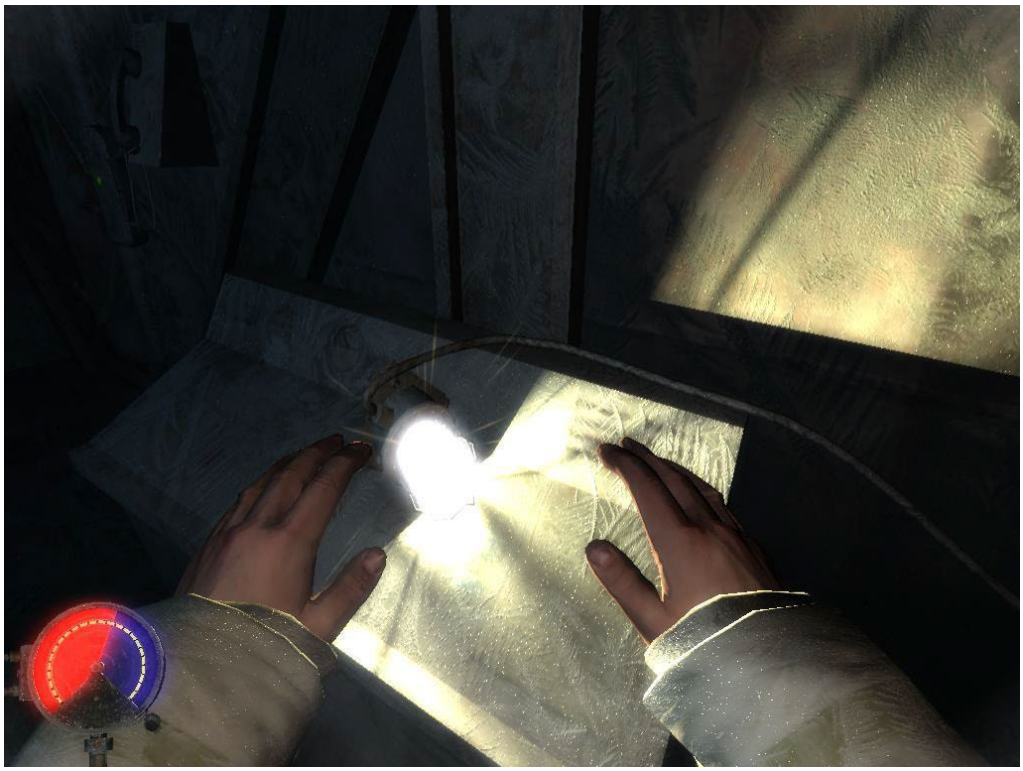

Рис. 1.3. Механіка холода у Cryostasis: Sleep of Reason

Рис. 1.4. Атмосфера у Cryostasis: Sleep of Reason

2. Pathologic 2

Pathologic 2 – це пригодницька гра з елементами виживання та рольовими елементами. У цій грі гравець бере на себе роль лікаря, яки має вивчити дивний та незрозумілий спалах захворювання у місті, де всі перебувають під загрозою.

Атмосфера чуми є одним з головних елементів гри та створює відчуття страху, тривоги та розпачу у гравця. У грі присутня похмура і напружена атмосфера, яка передає гравцеві почуття загрози, що насувається, і розпачу.

Навколишній світ виконаний у темних та похмурих тонах, що підкреслює невідворотність лиха. Водночас він є чудовим тлом для гри та створює у гравця почуття живого та зараженого середовища.

Вцілому, атмосфера чуми створює глибоке і несамове почуття, яке є одним з головних елементів гри.

На рис. 1.5 та 1.6 зображена атмосфера чуми в грі.

Рис. 1.5. Атмосфера чуми на вулицях Pathologic 2

Рис. 1.6. Атмосфера чуми у приміщеннях Pathologic 2

3. The Long Dark

The Long Dark – це дослідницька гра-симулятор виживання, у якій гравці-одинаки повинні подбати про себе під час вивчення великих морозних безлюдних територій, які пережили геомагнітну катастрофу. Тут немає зомбі, лише тільки ви, холод і всі небезпеки.

Головною особливістю гри є її високий рівень реалізму та реалістичність умов виживання. Гравцеві доводиться стежити за своїми запасами їжі, води та тепла, а також враховувати погодні умови та добу. У грі також є різні небезпеки, такі як хижі тварини та несподівані небезпеки, які можуть загрожувати життю персонажа.

Вцілому, The Long Dark є унікальною грою, яка доставляє гравцю високий рівень реалізма холода та створює реальне відчуття боротьби за виживання.

На рис. 1.7 зображена атмосфера The Long Dark.

Рис. 1.7. Атмосфера The Long Dark

1.4. Постановка задачі

Опираючись на аналіз схожих відеоігор, можна сказати, що для реалізації цікавої та атмосферної комп’ютерної гри “Nuclear Frost” у сеттингу ядерної зими потрібно розробити наступні ігрові механики:

– Механіка холоду.

Необхідно розробити систему холода, завдяки якій гравцю необхідно буде в умовах суворих морозів доглядати за тим, щоб його герой не замерз, а для цього треба грітися у різних джерел тепла, які перед цим треба розпалювати. Даний показник також має впливати на швидкість пересування, чим сильніше замерз герой, тим повільніше він рухається. Також, при досягненні критичної позначки повинно зменшуватись здоров'я гравця.

– Механіка радіації.

Також треба розробити логіку радіації яка буде зменьшувати гравцю його здоров’є, чим вище рівень показника, тим швидше повинно зменьшуватись його здоров’є та тим швишде він може померти, а для того щоб позбавитись радіації, гравець повинен вийти із забрудненого місця, після чого вона почне пропадати. Для сповіщення ігрока о зонах радіації має бути лічильник Гейгера який своїм пищанням буде повідомляти о присутності радіації.

– Бойова механіка.

Необхідно розробити у грі бойову систему, яка дозволить гравцю брати до рук зброю, стріляти з неї, вбивати монстрів та отримувати урон від ворогів.

Також потрібно взяти до уваги, що має бути відповідний звуковий супровід та ефект, коли куля влучає у якийсь матеріал (дерево, сніг, метал та інші).

– Монстри.

Необхідно розробити декілька монстрів зі штучним інтелектом, які повинні нападати на героя, шукати ворога якщо втратять його з поля зору, втікати та лікуватись коли їх рівень здоров’я буде занато низький.

Окрім ігрових механік ще необхідно зробити наступні пункти:

– реализувати систему діалогів, яка буде відвопідати за спілкування гравця з різними персонажами;

– створити інтерактивну система, вона має реалізовувати логіку взаємодії ігрока з різними об’єктами;

– реалізувати система збереження, яка буде відповідати за збереження та завантаження гри;

– створити систему завдань, вона має реалізовувати головну логіку праці місії на ігровій мапі;

– реалізувати систему завантажувального екрану, яка буде запускати відео під час завантаження гри.

Висновки до розділу:

Під час опису різних ігор було проаналізовано їх атмосферу, геймплейні механіки, різновид жанрів, та завдяки цьому виділено сильні сторони які необхідно реалізувати у проєкті.

РОЗДІЛ 2. ОРГАНІЗАЦІЯ КОМАНДИ ТА СЕРЕДОВИЩА

2.1. Організація команди

Керування розробкою комп'ютерної гри у наш час вимагає командної праці, дисципліни, організованості та чіткого розуміння ролі кожного учасника команди.

Команду можна описати як групу, що об'єднується для виконання одного або кількох завдань та досягнення передбачених цілей. Практика більшості розробників підтверджує, що командна робота зазвичай приносить кращі результати, оскільки навіть висококваліфікований спеціаліст стає сильнішим, працюючи з іншими людьми.

Управління командою є одним з найважливіших комплексних навичок, і включає такі процеси:

- встановлення цілей і передача їх команді;
- мотивація та надихання;
- регулярне спілкування з командою та надання зворотного зв’язку;
- вирішення конфліктів та створення здорової атмосфери в колективі, сприятливої корпоративній культурі;
- розширення можливостей команди у професійному плані.

Ефективне керівництво командою проекту сприяє розкриттю потенціалу кожної окремої особи і колективу в цілому.

Для досягнення злагодженості в команді необхідно побудувати ієрархію, де кожен член розуміє свою роль та місце. Кількість рівнів в ієрархії залежатиме від складності проекту та розміру команди. Крім того, важливо:

- сформулювати спільну зрозумілу мету проекту;
- встановити залежності між групами людей, які сприятимуть досягненню цілей;;
- з'ясувати, хто відповідає за які аспекти проекту;;

– забезпечити, щоб кожен член команди розумів очікування, що стосуються їхньої ролі.

Додатково, для ефективної організації робочого процесу рекомендується використовувати інструмент таск-трекера.

На рис. 2.1 наведено приклад використання JIRA, відомого інструменту для керування завданнями в команді.

Так, на прикладі додатку можна побачити дошку, на якій організовано роботу з завданнями на різних етапах проекту. Для кожного завдання вказано відповідальну особу, команду виконавців та категорію.

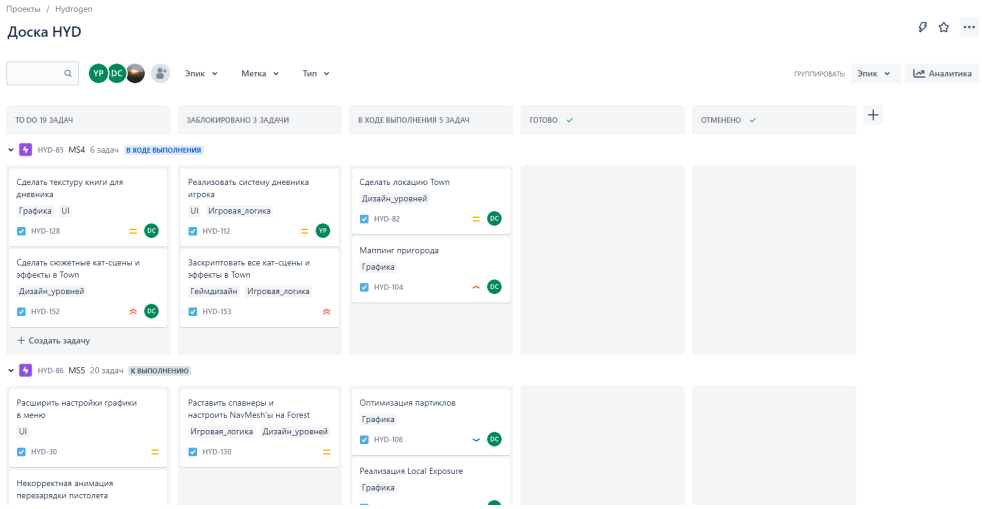

Рис. 2.1. Дошка Nuclear Frost в JIRA

JIRA надає можливість взаємодіяти з завданнями за допомогою таких функцій:

– створення унікальних етапів робочого процесу, наприклад, "Виконати", "В процесі", "Виконано" та інші;

– встановлення термінів виконання для кожного окремого завдання або процесу;

– створення основних та проміжних завдань;

– доручення завдань конкретним учасникам команди;

– визначення типів проблем;

– розподіл пріоритетів для завдань та проблем.

2.2. Дорожня карта

Для розпочатку розробки будь-якого IT-продукту або запуску нового проекту необхідно скласти його дорожню карту. Це є першим кроком на цьому шляху, без якого неможливо розпочати рух, оскільки невідомо, куди саме потрібно рухатися.

Дорожня карта проєкту (Road Map) є документом, який вкоротку описує стратегію розвитку певної ініціативи. Цей документ може застосовуватись у різних контекстах, таких як розробка продукту, конкретний проєкт або сегмент ринку. Головна мета дорожньої карти - чітко візуалізувати глобальний план або стратегію розвитку проєкту на початковому етапі. Це допомагає спрямувати всі зусилля в потрібному напрямку і може бути корисним навіть для існуючих підприємств, де використовується для передбачення стратегії впровадження змін, таких як випуск нового продукту або модернізація виробництва.

Road map вирішує кілька завдань, що сприяють ефективній роботі організації або команди в напрямку спільної мети:

– визначення пріоритетів: допомагає визначити найважливіші та пріоритетні цілі та завдання для досягнення спільної мети;

– планування: допомагає розробити детальний план дій з урахуванням обмежень часу, бюджету та інших факторів, необхідний для досягнення цілей та завдань;

– управління ресурсами: дозволяє ефективно управляти ресурсами, такими як час, бюджет та людські ресурси, для оптимального використання під час виконання завдань;

– комунікація: дорожня карта є інструментом комунікації, що дозволяє передавати інформацію про плани та цілі між членами команди або зацікавленими сторонами;

– вимірювання прогресу: допомагає відстежувати прогрес у досягненні цілей та завдань, а також вчасно коригувати плани, якщо необхідно;

– оцінка ризиків: допомагає визначити можливі ризики та проблеми, які можуть виникнути під час досягнення цілей, та розробити стратегії та плани дій для їх зниження або запобігання.

Для побудови правильної дорожньої карти необхідно виконати кілька кроків:

– визначення цілей проекту: потрібно визначити, які конкретні цілі має досягти проект і як вони співвідносяться з загальними цілями команди;

– визначення ключових завдань: потрібно визначити основні завдання, які потрібно виконати для досягнення цілей. Ці завдання можуть бути поділені на більш дрібні підзавдання.;

– оцінка часу та ресурсів: потрібно оцінити, скільки часу та ресурсів потрібно для виконання кожного завдання. Це допоможе зрозуміти, які завдання можуть виконуватись паралельно, а які мають бути виконані послідовно.;

– визначення пріоритетів: необхідно визначити, які завдання є пріоритетними і мають бути виконані в першу чергу. Це допоможе забезпечити більш ефективне використання ресурсів та досягнення цілей проекту вчасно;

– візуалізація дорожньої карти: дорожня карта повинна бути візуалізована у вигляді графіка Ганта або тимчасової лінії. Це допоможе краще організувати завдання та показати, як вони пов'язані між собою;

– регулярне оновлення та коригування дорожньої карти: дорожню карту слід регулярно оновлювати та коригувати, щоб відображати зміни в умовах проекту та забезпечити досягнення цілей у встановлений термін..

На рис. 2.2 зображено приклад побудованої road map.

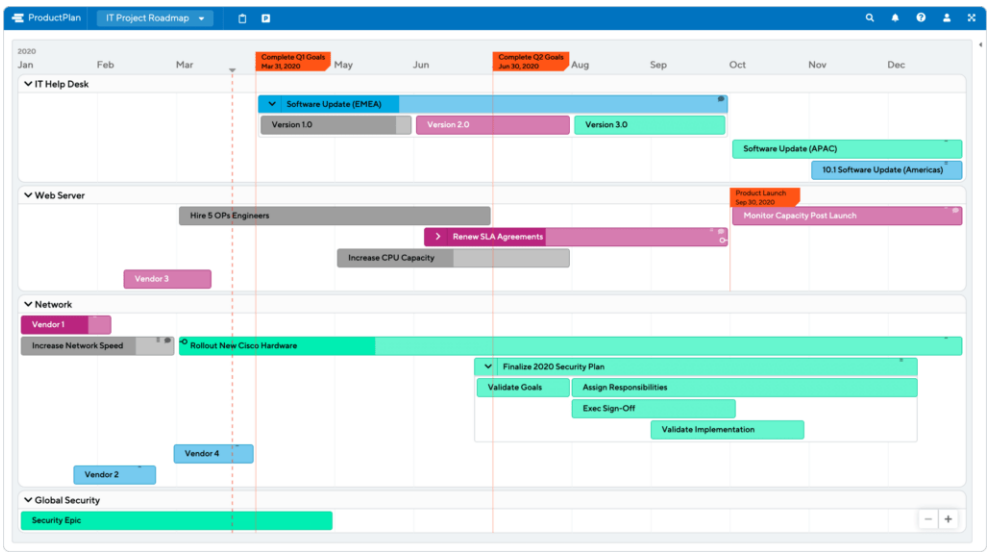

Рис. 2.2. Приклад дорожньої карти

2.3. Система керування версіями

Система керування версіями (Version Control System) - це програмний засіб, призначений для управління версіями різних типів інформації, такої як вихідний код програми, скрипт, веб-сторінка, веб-сайт та інше. Цей інструмент дозволяє одночасно та без перешкод виконувати колективну роботу над груповими проектами.

Система контролю версій дозволяє зберігати попередні версії файлів та використовувати їх при необхідності. Вона зберігає повну інформацію про кожну версію файлу, а також повну структуру проекту на всіх етапах розробки. Місце зберігання файлів називається репозиторієм. Усередині репозиторію можуть бути створені паралельні робочі гілки, що дозволяють розробникам незалежно працювати та експериментувати з кодом.

Використання системи контролю версій є обов'язковим при роботі над великими проектами, в яких бере участь велика кількість розробників. Системи контролю версій надають ряд додаткових можливостей, зокрема:

– можливість створення різних варіантів одного документу;

– документування всіх змін, включаючи інформацію про автора, час та змінені рядки;

– реалізація функції контролю доступу користувачів до файлів, включаючи можливість обмеження доступу;

– можливість створення документації проєкту з поетапним описом змін для кожної версії;

– можливість надання пояснень до змін та їх документування.

У сучасному світі багато організацій впроваджують системи контролю версій в свої робочі процеси. Практично кожна компанія, що займається розробкою програмного забезпечення, користується цими системами. Однак, окрім комерційних організацій, системи контролю версій широко використовуються в університетах по всьому світу. Наприклад, варто відзначити роботу двох професорів з Торонто - Грегорі В. Вілсона та Карен Рейд. У своєму новому проекті вони прагнуть створити навчальне середовище для студентів, в якому планується використання систем контролю версій, систем відстеження задач та веб-журналів.

Університети дедалі більше зацікавлені у створенні та використанні середовищ, де всі студенти технічних спеціальностей матимуть доступ до систем контролю версій, веб-журналів та інших інструментів. Навіть у США Міністерство освіти займається цим питанням, сприяючи впровадженню подібних систем на рівні держави.

На сьогоднішній день існує два популярні інструменти керування версіями, які широко використовуються в галузі розробки комп'ютерних ігор: Subversion (SVN) і Perforce (P4).

Subversion (SVN) є централізованою системою керування версіями, де дані зберігаються в одному центральному сховищі. При збереженні нових версій використовується дельта-компресія, що означає, що система виявляє відмінності між новою та попередньою версіями і зберігає лише ці відмінності, уникнувши зайвого дублювання даних. Сховище може знаходитися на локальному диску або на мережевому сервері. Клієнт Subversion безпосередньо

звертається до локального сховища. Для доступу до віддаленого сервера можуть використовуватися власний мережевий протокол або стандартний протокол WebDAV, що підтримується за допомогою спеціального модуля для вебсервера Apache.

Клієнти створюють локальні робочі копії файлів з сховища, а потім вносять зміни до цих копій і зберігають їх знову у сховище. Багато клієнтів можуть одночасно звертатися до сховища. Якщо використовується доступ через WebDAV, система також підтримує прозоре керування версіями. Це означає, що якщо будь-який клієнт WebDAV відкриває файл для редагування та зберігає його на мережевому ресурсі, автоматично створюється нова версія цього файлу.

Subversion замінив CVS і поступово став найбільш поширеним інструментом для керування версіями, витісняючи свого попередника. Багато спільнот розробників відкритого програмного забезпечення перейшли на використання Subversion. Серед них варто відзначити такі відомі проекти, як Apache Software Foundation, KDE, GNOME, GCC, MediaWiki, Python, Samba, Mono та багато інших.

Попри поширення децентралізованих систем, Subversion залишається дуже популярним серед комерційних компаній і проектів, які використовують централізований підхід до керування версіями і конфігурацією програмних систем.

На рис. 2.3 зображено приклад SVN репозиторію Nuclear Frost.

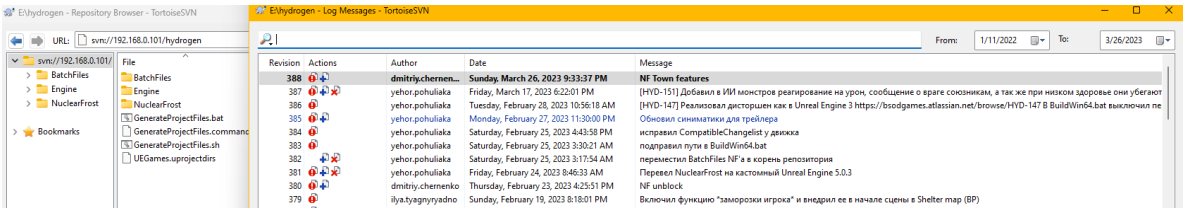

Рис. 2.3. SVN репозіторій Nuclear Frost

Perforce – це платна система керування версіями, яка знаходить широке застосування у розробці програмного забезпечення та інших проектах з великим обсягом коду. Вона була створена у 1995 році і отримала популярність серед великих компаній, таких як Google, Electronic Arts, NVIDIA, Salesforce, Cisco та інших.

Perforce має численні переваги, такі як висока швидкість та масштабованість, здатність працювати з різними типами файлів (текстові, бінарні, зображення і т.д.), підтримка гілок та злиття, а також багатий набір інструментів і можливостей для налаштування системи під потреби розробників.

Крім того, Perforce відрізняється високим рівнем безпеки, що включає механізми аутентифікації та авторизації користувачів, захист від несанкціонованого доступу та втрати даних. Також система надає можливість резервного копіювання та відновлення інформації, що дозволяє забезпечити надійність та цілісність даних.

На рис. 2.4 зображено приклад Perforce репозиторію.

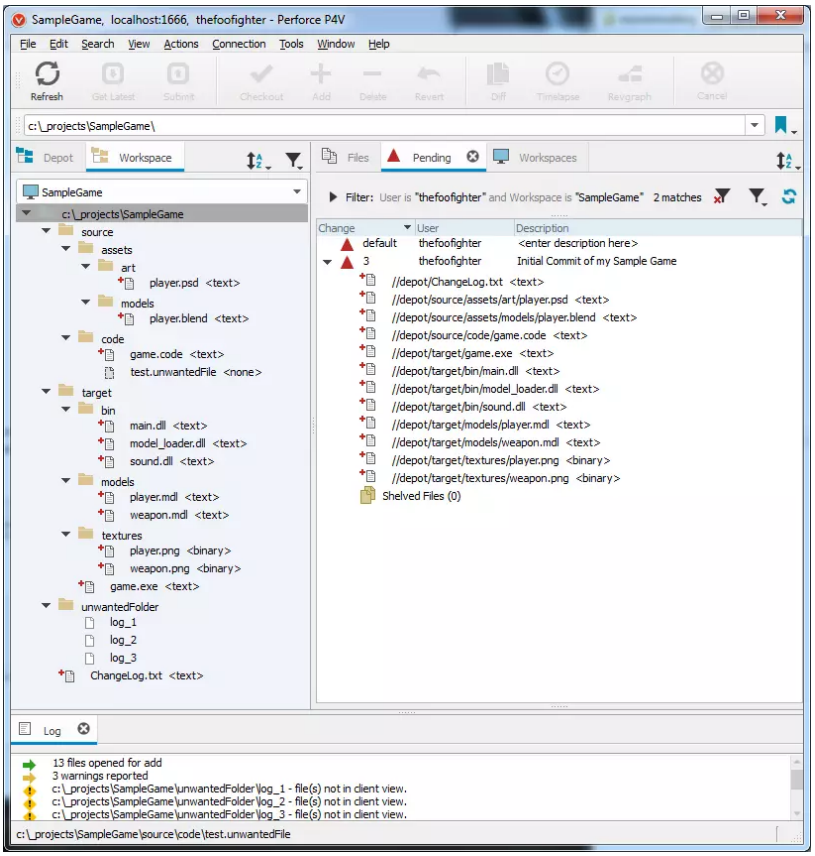

Рис. 2.4. Perforce репозіторій

2.4. Blueprints в Unreal Engine

Blueprints є компонентом в Unreal Engine, що надає можливість візуального програмування. За його допомогою розробники, дизайнери та художники можуть створювати різноманітні ігрові події, логіку та навіть матеріали без необхідності писати код.

У порівнянні з програмістами, дизайнери зазвичай не володіють програмувальними мовами настільки добре, щоб вносити зміни в код. Тому для них використання блупринтів стає корисним рішенням. Блупринти надають можливість дизайнерам створювати функціональність без потреби в програмуванні. Вони мають свої обмеження, але можуть бути розширені за допомогою коду для додаткової гнучкості.

Хоча теоретично можна створити гру виключно за допомогою блупринтів, такий підхід не є оптимальним, оскільки це може призвести до втрати оптимізації та гнучкості. Користуючись блупринтами, ми можемо насолоджуватися комфортом і простотою розробки, але це може вплинути на продуктивність та можливість вносити глибокі зміни в гру.

На рис. 2.5 зображено приклад Blueprints.

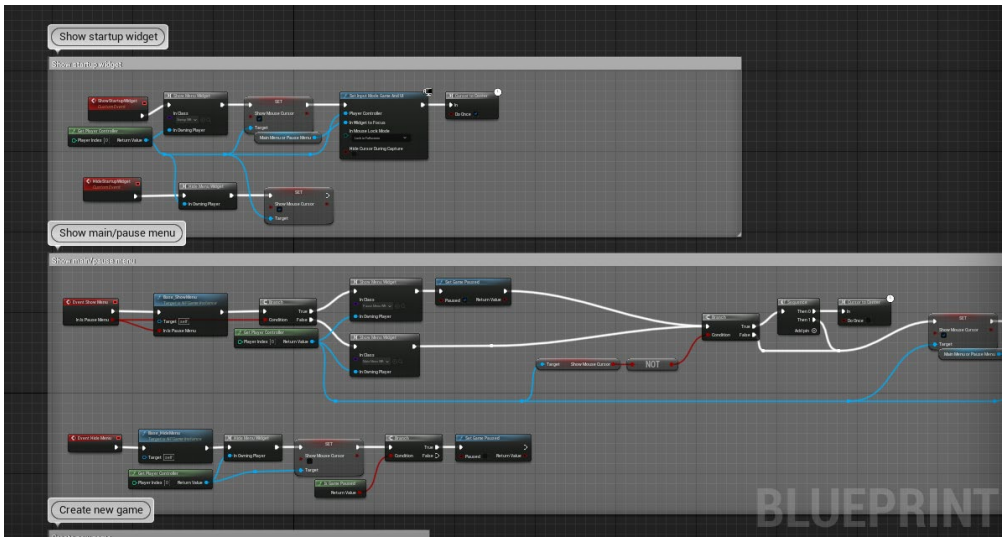

Рис. 2.5. Приклад Blueprints в Unreal Engine

Blueprints використовує події, що визначають початок конкретної послідовності дій і спрацьовують під час виклику з коду гри, щоб активувати мережу графів подій (Event Graph). Вони дозволяють Blueprint виконувати різноманітні дії у відповідь на різні події, такі як початок гри, скидання рівня, отримання збитків і так далі. В одному Event Graph можна використовувати будь-яку кількість подій, проте можна використовувати лише одну подію кожного типу.

У редакторі Blueprints вже доступно багато вузлів для обробки різних подій, проте є можливість створити власні вузли для обробки подій, наприклад, натискання певної клавіші. Для цього вам потрібно налаштувати новий вхідний вісь (Input Axis) у налаштуваннях вашого проекту і призначити йому клавішу на клавіатурі, миші або контролері. Після цього ви можете створити відповідний вузол в редакторі Event Graph. Ви можете побудувати логіку обробки цієї події, і коли задана клавіша буде натиснута, будуть виконуватися описані дії.

2.5. Behavior Trees в Unreal Engine

Behavior Trees у Unreal Engine можуть бути використані для створення штучного інтелекту для неігрових персонажів. У порівнянні з Behavior Tree, який визначає послідовність розгалужень і логіку виконання, Дерево поведінки використовує інший компонент, відомий як Blackboard, який виступає в ролі "мозку" дерева поведінки. На рис. 2.6 зображено приклад Behavior Tree у Nuclear Frost.

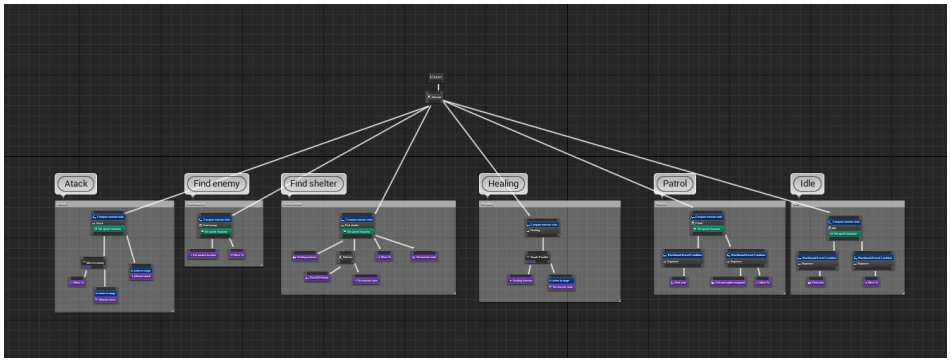

Рис. 2.6. Behavior Tree для монстра у Nuclear Frost

Blackboard вміщує низку користувацьких ключів, які містять інформацію, використовувану Behavior Tree для прийняття рішень. Наприклад, може бути ключ з логічним значенням під назвою "IsLightOn", на який Behavior Tree посилається, щоб перевірити, чи змінилось значення. Якщо значення є істинним, дерево поведінки може виконати розгалуження, що змусить персонажа втекти. Якщо значення є хибним, воно може перейти до іншої гілки, де персонаж, можливо, буде рухатись випадковим чином по оточуючому середовищу. Behavior Trees можуть бути настільки простими, як у прикладі з персонажем, або настільки складними, як симуляція поведінки іншого гравця у багатокористувацькій грі, де вони шукають приховку, відкривають вогонь по іншим гравцям і шукають предмети. На рис. 2.7 зображено приклад Blackboard для монстра у Nuclear Frost.

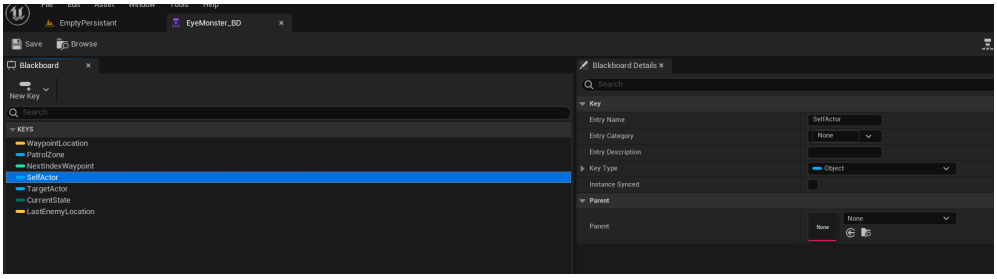

Рис. 2.7. Blackboard для монстра у Nuclear Frost

Висновки до розділу

Під час опису методів організації команди було з’ясовано як працювати у команді та контролювати завдання. Також при описі систем контролю версій з’ясовано, що при розробці проєкту необхідно використовувати чи Subversion, чи Perforce, а також проаналізовано що таке Blueprints та Behavior Trees в Unreal Engine.

РОЗДІЛ 3. РОЗРОБКА ІГРОВИХ МЕХАНІК ТА СИСТЕМ ГРИ

3.1. Плагін завантажувального екрану

Плагін завантажувального екрану – це плагін, який забезпечує відображення якогось зображення чи відео під час завантаження гри. На рис. 3.1 зображено приклад такого екрану у Nuclear Frost.

Як діятиме цей плагін: весь час гри він буде слухати подію, яка відповідає за завантаження нової карти і коли спрацьовуватиме, то цей плагін запустить програвання відео або зображення.

Рис. 3.1. Приклад завантажувального екрану

Для того щоб це зробити необхідно виконати наступні кроки:

1. Відкрити проєкт гри у Unreal Engine та у вікні створення нового плагіну обрати Blank і заповнити відповідні поля.

2. Створити класс FloadingScreenModule де буде знаходитись основна логіка завантажувального екрану. Приклад реалізації цього класу:

```
// ILoadingScreenModule.h
// Copyright 1998-2019 Epic Games, Inc. All Rights Reserved.
#pragma once
#include <Core/Public/Modules/ModuleManager.h>
class ILoadingScreenModule : public IModuleInterface
{
public:
	virtual void SetLoadingScreen( FName InScreenName ) = 0;
	virtual void ResetLoadingScreen() = 0;
	static inline ILoadingScreenModule& Get() { return FModuleManager::LoadModuleChecked< ILoading-
ScreenModule >("LoadingScreen"); }
	static inline bool IsAvailable() { return FModuleManager::Get().IsModuleLoaded( "LoadingScreen" );
}
};
```

```
// LoadingScreenModule.cpp
// Copyright 1998-2019 Epic Games, Inc. All Rights Reserved.
#include "ILoadingScreenModule.h"
#include "LoadingScreenSettings.h"
#include "SSimpleLoadingScreen.h"
#include "Framework/Application/SlateApplication.h"
#define LOCTEXT_NAMESPACE "LoadingScreen"
class FLoadingScreenModule : public ILoadingScreenModule
{
public:
	FLoadingScreenModule();
	virtual void StartupModule() override;
	virtual void ShutdownModule() override;
	virtual bool IsGameModule() const override
	{
		return true;
	}
	virtual void SetLoadingScreen( FName InScreenName ) override;
	virtual void ResetLoadingScreen() override;
private:
	void HandlePrepareLoadingScreen();
	void BeginLoadingScreen(const FLoadingScreenDescription& ScreenDescription);
	FName  LoadingScreenName;
};
IMPLEMENT_MODULE(FLoadingScreenModule, LoadingScreen)
FLoadingScreenModule::FLoadingScreenModule(){}
void FLoadingScreenModule::StartupModule()
{
	if ( !IsRunningDedicatedServer() && FSlateApplication::IsInitialized() )
	{
		// Load for cooker reference
		const ULoadingScreenSettings* Settings = GetDefault<ULoadingScreenSettings>();
		for ( const FStringAssetReference& Ref : Settings->StartupScreen.Images ) { Ref.TryLoad(); }
		for ( const FStringAssetReference& Ref : Settings->DefaultScreen.Images ) { Ref.TryLoad(); }
		if ( IsMoviePlayerEnabled() )
		{
			GetMoviePlayer()->OnPrepareLoadingScreen().AddRaw(this, &FLoadingScreenMod-
ule::HandlePrepareLoadingScreen);
		}
		BeginLoadingScreen(Settings->StartupScreen);
	}
}
void FLoadingScreenModule::HandlePrepareLoadingScreen()
{
	const ULoadingScreenSettings*		Settings = GetDefault<ULoadingScreenSettings>();
	const FLoadingScreenDescription*	LoadingScreenDescription = Settings->LoadingScreens.Find(
LoadingScreenName );
	if ( !LoadingScreenDescription ) { BeginLoadingScreen( Settings->DefaultScreen ); }
	else { BeginLoadingScreen( *LoadingScreenDescription ); }
}
void FLoadingScreenModule::BeginLoadingScreen(const FLoadingScreenDescription& ScreenDescription)
{
	FLoadingScreenAttributes LoadingScreen;
	LoadingScreen.MinimumLoadingScreenDisplayTime = ScreenDescription.MinimumLoadingScreenDisplayTime;
	LoadingScreen.bAutoCompleteWhenLoadingCompletes = ScreenDescription.bAutoCompleteWhenLoadingCom-
pletes;
	LoadingScreen.bMoviesAreSkippable = ScreenDescription.bMoviesAreSkippable;
	LoadingScreen.bWaitForManualStop = ScreenDescription.bWaitForManualStop;
	LoadingScreen.MoviePaths = ScreenDescription.MoviePaths;
	LoadingScreen.PlaybackType = ScreenDescription.PlaybackType;
	LoadingScreen.bAllowEngineTick = ScreenDescription.bAllowEngineTick;
	if ( ScreenDescription.bShowUIOverlay ) { LoadingScreen.WidgetLoadingScreen = SNew(SSimpleLoading-
Screen, ScreenDescription); }
	GetMoviePlayer()->SetupLoadingScreen(LoadingScreen);
}
#undef LOCTEXT_NAMESPACE
```

3. Створити клас SsimpleLoadingScreen який буде описувати віджет UI через який буде малюватись відео чи зображення. Код цього класу:

```
// SSimpleLoadingScreen.h
// Copyright 1998-2017 Epic Games, Inc. All Rights Reserved.
#pragma once
```

```
#include <SlateCore/Public/Widgets/SCompoundWidget.h>
#include "LoadingScreenSettings.h"
class FDeferredCleanupSlateBrush;

class SSimpleLoadingScreen : public SCompoundWidget
{
public:
	SLATE_BEGIN_ARGS(SSimpleLoadingScreen) {}
	SLATE_END_ARGS()
	void Construct(const FArguments& InArgs, const FLoadingScreenDescription& ScreenDescription);
private:
	float GetDPIScale() const;

private:
	TSharedPtr<FDeferredCleanupSlateBrush> LoadingScreenBrush;
	TSharedPtr<FDeferredCleanupSlateBrush> PieceImageBrush;
};

// SSimpleLoadingScreen.cpp
// Copyright 1998-2019 Epic Games, Inc. All Rights Reserved.
#include "SSimpleLoadingScreen.h"
#include "Widgets/SOverlay.h"
#include "Widgets/Images/SThrobber.h"
#include "Engine/Texture2D.h"
#define LOCTEXT_NAMESPACE "LoadingScreen"
static float PointSizeToSlateUnits(float PointSize)
{
	//FreeTypeConstants::HorizontalDPI = 96;
	const float SlateFreeTypeHorizontalResolutionDPI = 96.0f;
	const float FreeTypeNativeDPI = 72.0f;
	const float PixelSize = PointSize * (SlateFreeTypeHorizontalResolutionDPI / FreeTypeNativeDPI);
	return PixelSize;
}
void SSimpleLoadingScreen::Construct(const FArguments& InArgs, const FLoadingScreenDescription& InScreenDe-
scription)
{
	const ULoadingScreenSettings* Settings = GetDefault<ULoadingScreenSettings>();
	const FSlateFontInfo& TipFont = Settings->TipFont;
	const FSlateFontInfo& LoadingFont = Settings->LoadingFont;
	TSharedRef<SOverlay> Root = SNew(SOverlay);
	if ( InScreenDescription.Images.Num() > 0 )
	{
		const int32 ImageIndex = FMath::RandRange(0, InScreenDescription.Images.Num() - 1);
		const FStringAssetReference& ImageAsset = InScreenDescription.Images[ImageIndex];
		UObject* ImageObject = ImageAsset.TryLoad();
		if ( UTexture2D* LoadingImage = Cast<UTexture2D>(ImageObject) )
		{
			LoadingScreenBrush = FDeferredCleanupSlateBrush::CreateBrush(LoadingImage);
	}
	TSharedRef<SWidget> TipWidget = SNullWidget::NullWidget;
	if ( Settings->Tips.Num() > 0 )
	{
		const int32 TipIndex = FMath::RandRange(0, Settings->Tips.Num() - 1);
		TipWidget = SNew(STextBlock)
			.WrapTextAt(Settings->TipWrapAt)
			.Font(TipFont)
			.Text(Settings->Tips[TipIndex]);
	}
	else
	{ TipWidget = SNew(SSpacer); }
	TSharedRef<SWidget>  ThrobberWidget = SNullWidget::NullWidget;
	{
		{
			UObject*           ImageObject = InScreenDescription.PieceImage.TryLoad();
			if ( UTexture2D* PieceImage = Cast<UTexture2D>( ImageObject ) )
			{
				float        Size = PointSizeToSlateUnits( LoadingFont.Size ) * 1.5f;
				PieceImageBrush = FDeferredCleanupSlateBrush::CreateBrush( PieceImage, FVector2D(
Size, Size ) );
			}
		}
		switch ( InScreenDescription.ThrobberType )
		{
		case EThrobberType::TT_Circular:
		{
			TSharedRef<SCircularThrobber> CircularThrobberWidget = SNew( SCircularThrobber )
				.Radius( PointSizeToSlateUnits( LoadingFont.Size ) / 2.0f )
```

```
                        .NumPieces( InScreenDescription.NumPieces )
                        .PieceImage(   PieceImageBrush.IsValid()   ?   PieceImageBrush->GetSlateBrush()   :
nullptr );

                    ThrobberWidget = CircularThrobberWidget;
                    break;
            }
                    ThrobberWidget = OrdinaryThrobberWidget;
                    break;
            }
...
```

4. Створити клас ULoadingScreenSettings за допомогою якого зможемо у двигуні редагувати екран завантаження. Приклад цього класу:

```
// LoadingScreenSettings.h
// Copyright 1998-2019 Epic Games, Inc. All Rights Reserved.

#pragma once

#include "CoreMinimal.h"
#include "Engine/DeveloperSettings.h"
#include "LoadingScreenSettings.generated.h"

UENUM( BlueprintType )
enum class EThrobberType : uint8
{
      TT_Ordinary        UMETA( DisplayName = "Throbber" ),
      TT_Circular        UMETA( DisplayName = "Circular Throbber" ),
};

USTRUCT(BlueprintType)
struct LOADINGSCREEN_API FLoadingScreenDescription
{
      GENERATED_USTRUCT_BODY()
      FLoadingScreenDescription();
      UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Loading)
      float MinimumLoadingScreenDisplayTime = -1;
      UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Loading)
      bool bAutoCompleteWhenLoadingCompletes = true;
      UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Images)
      TEnumAsByte<EStretch::Type> ImageStretch = EStretch::ScaleToFit;
      UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Images)
      FLinearColor BackgroundColor = FLinearColor::Black;
      UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Images)
      FLinearColor TipBackgroundColor = FLinearColor(0, 0, 0, 0.75f);
};
UCLASS(config=Game, defaultconfig, meta=(DisplayName="Loading Screen"))
class LOADINGSCREEN_API ULoadingScreenSettings : public UDeveloperSettings
{
      GENERATED_UCLASS_BODY()
public:
      UPROPERTY(config, EditAnywhere, Category=Screens)
      FLoadingScreenDescription StartupScreen;
      UPROPERTY(config, EditAnywhere, Category=Screens)
      FLoadingScreenDescription DefaultScreen;
      UPROPERTY( config, EditAnywhere, Category=Screens )
      TMap< FName, FLoadingScreenDescription > LoadingScreens;
      UPROPERTY(config, EditAnywhere, BlueprintReadWrite, Category=Advice)
      FSlateFontInfo TipFont;
      UPROPERTY(config, EditAnywhere, BlueprintReadWrite, Category = Display)
      FSlateFontInfo LoadingFont;
      UPROPERTY(config, EditAnywhere, BlueprintReadWrite, Category=Advice)
      float TipWrapAt;
      UPROPERTY(config, EditAnywhere, BlueprintReadWrite, Category=Advice, meta = (MultiLine = "true"))
      TArray<FText> Tips;
};
```

5. Скомпіліювати проєкт у Visual Studio та запустити Unreal Engine.

6. У двигуні відкрити Project Settings та знайти налаштування нового плагіну. На рис. 3.2 зображено налаштування нового модуля, де можливо обрати для якої мапи що буде відображатись.

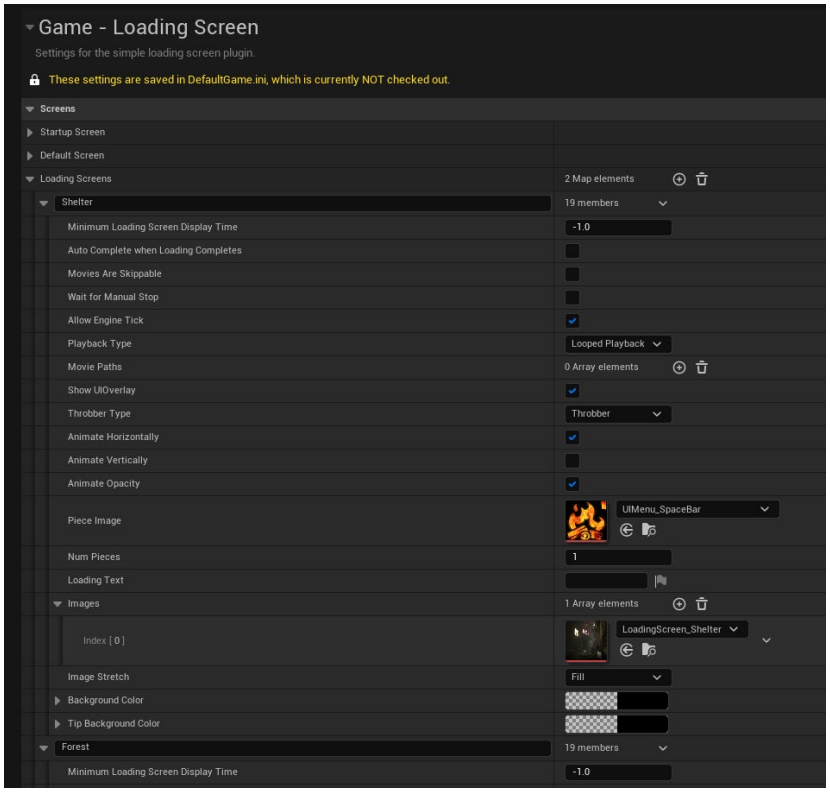

Рис. 3.2. Налаштування нового модуля

3.2. Інтерактивна система

Інтерактивна система – це програмна система, яка забезпечує взаємодію між гравцем та ігровим середовищем. Інтерактивна система дає можливість гравцеві взаємодіяти з ігровим світом шляхом вводу команд, дій та рухів за допомогою контролера або іншого пристрою введення.

Інтерактивні системи дозволяють гравцеві відчути себе частиною ігрового світу та вплинути на події, що відбуваються в грі. Вони є важливим елементом будь-якої гри та дозволяють розширити можливості гравців.

Як буде діяти ця система: вона буде використовуватись лише для взаємодії з придметами та персонажами, а для того щоб дізнатись про те, що гравець навів курсор на такого актора, будемо пускати проміть завдяки якому ми

дізнаємось про це. У випадку якщо промінь зіткнувся з об’єктом, з яким герой може взіємодіяти, то на еркані з’явиться відповідне повідомлення.

На рис. 3.3 зображено як ця система працює у Nuclear Frost.

Рис. 3.3. Інтерактивна система у Nuclear Frost

Для того щоб це реалізувати необхідно виконати наступні кроки:

1. Для початку небхідно створити інтерфейс інтерактивного об’єкту через який буде відбуватись візємодія користувача. Приклад коду такого інтерфейсу:

```
// InteractiveObjectInterface.h
// Copyright Broken Singularity, All Rights Reserved.
// Authors: Yehor Pohuliaka (zombiHello)

#pragma once
#include "CoreMinimal.h"
#include "UObject/ObjectMacros.h"
#include "UObject/Interface.h"
#include "../../NuclearFrost.h"
#include "InteractiveObjectInterface.generated.h"

UINTERFACE()
class NUCLEARFROST_API UInteractiveObjectInterface : public UInterface
{
	GENERATED_UINTERFACE_BODY()
};

class NUCLEARFROST_API IInteractiveObjectInterface
{
public:
	GENERATED_IINTERFACE_BODY()
	virtual void Use( class ANFPlayerController* InPlayerController ) = 0;
	virtual void HoverStart() = 0;
	virtual void HoverEnd() = 0;
	virtual class UNFInteractiveObjectComponent* GetInteractiveObjectComponent() const = 0;
};

bool IsInteractiveObject( class AActor* InActor );
```

```
// InteractiveObjectInterface.cpp
// Copyright Broken Singularity, All Rights Reserved.
// Authors: Yehor Pohuliaka (zombiHello)

#include "InteractiveObjectInterface.h"
#include "BaseInteractiveActor.h"
#include "BaseInteractiveCharacter.h"

UInteractiveObjectInterface::UInteractiveObjectInterface(const FObjectInitializer& InObjectInitializer) :
Super( InObjectInitializer ) {}

bool IsInteractiveObject( class AActor* InActor )
{
	NF_CHECK_RETURN( InActor, false );
	return InActor->IsA< ABaseInteractiveActor >() || InActor->IsA< ABaseInteractiveCharacter >();
}
```

2. Коли ми маємо інтерфейс інтерактивного об’єкту ми можемо у ANFPlayerController створити метод який буде слідкувати за наведення курсора на такий об’єкт, викликатись буде кожний тік фрейму ігрової логіки. Код для цього кроку:

```
// NFPlayerController.cpp
...
void ANFPlayerController::UpdateUsableActor()
{
	// Ray tracing to find the actor to interact with. if successful, we remember he for use
	UWorld*			uworld = GetWorld();
	NF_CHECK( uworld );

	FHitResult		hitResult;
	FCollisionQueryParams	collisionParams;
	FVector start = PlayerCameraManager->GetCameraLocation();
	FVector end = start + PlayerCameraManager->GetCameraRotation().Vector() * maxDistanceUsingActor;

	collisionParams.AddIgnoredActor( nfCharacter );
	bool		successful	=	uworld->LineTraceSingleByChannel(	hitResult,	start,	end,
ECC_Camera, collisionParams );
	AActor*			actor		=	hitResult.GetActor();

	// If we view interactive actor and widget UseActor not showed - show this widget on viewport
	if ( successful && actor && IsInteractiveObject( actor ) )
	{
		// If this is the same actor, then we ignore him
		if ( usableObejctInView != actor )
		{
			IInteractiveObjectInterface*	oldInteractiveObject = Cast< IInteractiveObjectInter-
face >( usableObejctInView );
			IInteractiveObjectInterface*	newInteractiveObject = Cast< IInteractiveObjectInter-
face >( actor );
			if ( oldInteractiveObject )
			{
				oldInteractiveObject->HoverEnd();
			}

			usableObejctInView = actor;
			newInteractiveObject->HoverStart();
		}
	}

	// Else we not view interactive actor and in this case, if widget UseActor showing - hide this widget
from viewport
	else if ( usableObejctInView )
	{
		Cast< IInteractiveObjectInterface >( usableObejctInView )->HoverEnd();
		usableObejctInView = nullptr;
	}
}
...
```

3. Тепер необхідно на кнопку взаємодії викликати метод IinteractiveObjectInterface::Use, для цього у ANFPlayerController треба підписатись на відповідну подію та реалізувати метод ANFPlayerController::Use. Код цього кроку:

```
// NFPlayerController.cpp
...
void ANFPlayerController::SetupInputComponent()
{
	...
	// Binding use events
	InputComponent->BindAction( "Use", IE_Pressed, this, &ANFPlayerController::Use );
	InputComponent->BindAction( "ItemInspection", IE_Pressed, this, &ANFPlayerController::ItemInspection
);
	...
}
...
void ANFPlayerController::Use()
{
	if ( !usableObejctInView || isFreeze ) { return; }
	Cast<IInteractiveObjectInterface>( usableObejctInView )->Use( this );
}
```

4. Для того щоб повідомлення з’являлось на екрані необхідно у віджеті HUD (користувацький інтерфейс) додати цей надпис та у Blueprint GameHUD_BP викликати показ цього елемента. На рис. 3.4 зображен віджет напису у HUD, а на рис. 3.5 виклик показа цього елемента.

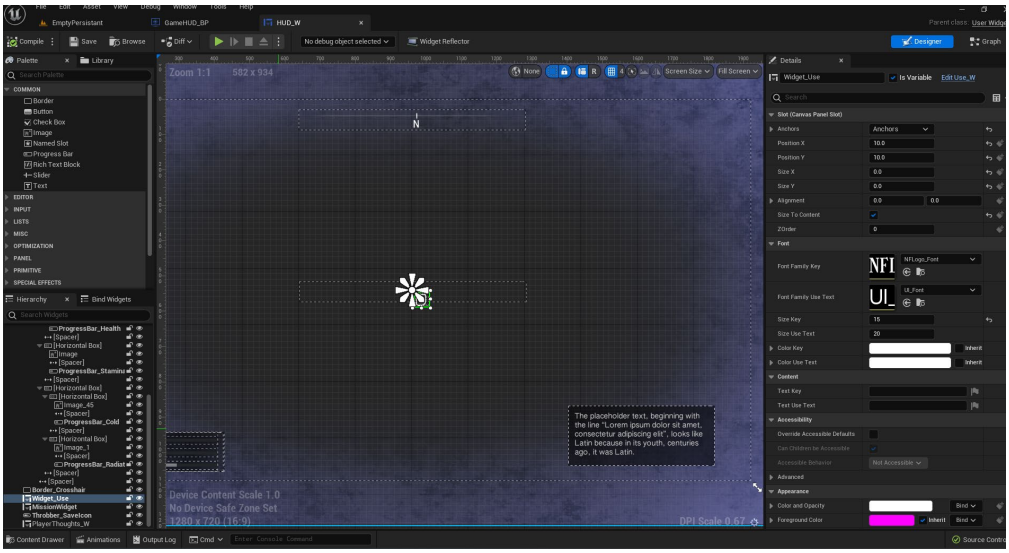

Рис. 3.4. Віджет з надписом в HUD

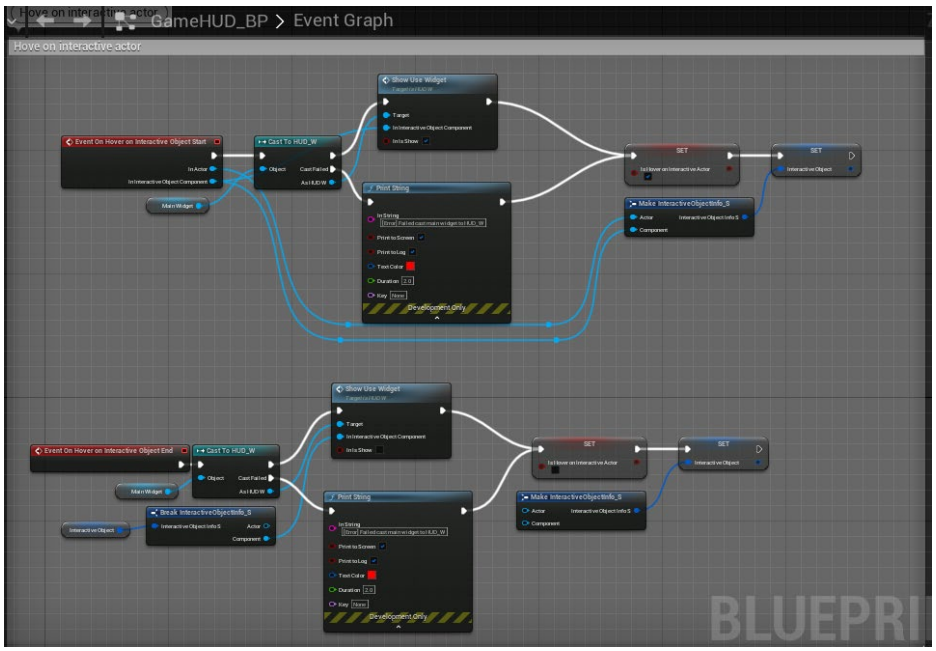

Рис. 3.5. Подія показа надпису у HUD

3.3. Система збереження гри

Система збереження гри (Save system) в іграх – це механізм, який дозволяє гравцям зберігати прогрес у грі та потім відновлювати його з того ж самого місця після перезавантаження гри або виходу з неї.

Це важлива функція, оскільки багато сучасних ігор мають складні сюжети та геймплей, які можуть займати години, навіть дні, щоб їх пройти. Без можливості збереження прогресу гравці могли б дуже швидко втратити інтерес до гри, якщо їм потрібно було б починати все з початку кожного разу, коли вони хочуть продовжити грати.

Система збереження гри може працювати по-різному в різних іграх, але зазвичай гравці мають можливість зберегти гру в певних точках або в будь-який момент, коли вони хочуть зупинитися. Деякі ігри можуть навіть автоматично зберігати гру після важливих подій або досягнень, щоб гравці не забували про це.

Для того щоб створити таку систему необхідно виконати наступні кроки:

1. Для початку необхідно створити клас UNFSaveGame за допомогою якого будуть зберігатися необхідні дані. Код цього кроку:

```
// NFSaveGame.h
// Copyright Broken Singularity, All Rights Reserved.
// Authors: Yehor Pohuliaka (zombiHello)

#pragma once
#include "CoreMinimal.h"
#include "GameFramework/SaveGame.h"
#include "UObject/ObjectMacros.h"
#include "NFPlayerController.h"
#include "Misc/DateTime.h"
#include "NFSaveGame.generated.h"
#define ACTOR_TAG_SAVEGAME            FName( TEXT( "SaveGame" ) )

USTRUCT()
struct FActorSaveData
{
	GENERATED_BODY()
	UPROPERTY()
	FName              ActorName;
	UPROPERTY()
	FTransform         Transform;
	UPROPERTY()
	TArray<uint8>      ByteData;
};

USTRUCT( BlueprintType )
struct FSaveGameInfo
{
	GENERATED_BODY()
	FSaveGameInfo();
	UPROPERTY( BlueprintReadOnly )
	bool                                bTransitSave;
	UPROPERTY( BlueprintReadOnly )
	FDateTime                           DateSave;
	UPROPERTY( BlueprintReadOnly )
	FText                               Title;
	UPROPERTY( BlueprintReadOnly )
	FText                               Description;
};

USTRUCT( BlueprintType )
struct FLevelSaveData
{
	GENERATED_BODY()
	UPROPERTY( BlueprintReadOnly )
	FString                                   LevelName;
	UPROPERTY( BlueprintReadOnly )
	TArray<FString>                     LevelsLoaded;
	UPROPERTY( BlueprintReadOnly )
	TArray<FString>                     LevelsLoadedNotVisible;
	UPROPERTY()
	TArray<uint8>                       ByteData;
};

UCLASS()
class NUCLEARFROST_API UNFSaveGame : public USaveGame
{
	GENERATED_BODY()

public:
	UNFSaveGame();
	UPROPERTY( BlueprintReadOnly, Category = "UI" )
	FSaveGameInfo                       Info;
	UPROPERTY()
	TArray<FActorSaveData>          SavedActors;
	UPROPERTY( BlueprintReadOnly, Category = "Player" )
	FNFPlayerState                      PlayerState;
	UPROPERTY( BlueprintReadOnly, Category = "Player" )
	FName                               CurrentMissionName;
	UPROPERTY( BlueprintReadOnly, Category = "Player" )
	int32                               CurrentPhaseMission;
```

```
	UPROPERTY( BlueprintReadOnly, Category = "Level" )
	FLevelSaveData                          LevelData;
};
```

2. Для того щоб працювало збереження необхідно у UNFGameInstance реалізувати метод SaveGameToMemory та SaveGameToSlot, а також метод LoadGameFromSlot для завантаження. Код цього кроку:

```
// NFGameInstance.cpp
...
void UNFGameInstance::SaveGameToMemory( bool bTransitSave /* = false */ )
{
	UWorld*            World = GetWorld();
	NF_CHECK( World );
	ANFPlayerController* PC = Cast<ANFPlayerController>( UGameplayStatics::GetPlayerController( World, 0
) );
	UNFMissionManager* MissionManager      = nullptr;
	if ( PC ) { MissionManager = PC->GetMissionManager(); }
	ANuclearFrostHUD* NFHUD = PC ? PC->GetNFHUD() : nullptr;
	if ( NFHUD ) { ->OnShowSaveIcon( true ); }
	CachedSavedGame= NewObject<UNFSaveGame>( this, TEXT( "NFSaveGame" ) );
	CachedSavedGame->Info.DateSave = FDateTime::UtcNow();
	CachedSavedGame->Info.bTransitSave = bTransitSave;
	CachedSavedGame->LevelData.LevelName = !bTransitSave ? UGameplayStatics::GetCurrentLevelName( World
) : TEXT( "" );
	const TArray<ULevelStreaming*>& StreamingLevels = World->GetStreamingLevels();
	for ( uint32 Index = 0, NumStreamingLevels = StreamingLevels.Num(); Index < NumStreamingLevels;
++Index )
	{
		ULevelStreaming* LevelStreaming = StreamingLevels[Index];
		if ( LevelStreaming->IsLevelLoaded() )
		{
			FString PackageName;
			if ( GIsEditor ) { PackageName = LevelStreaming->PackageNameToLoad.ToString(); }
			else { PackageName = LevelStreaming->GetWorldAssetPackageName(); }
			if ( LevelStreaming->IsLevelVisible() ) { CachedSavedGame->LevelData.LevelsLoaded.Add(
PackageName ); }
			Else { CachedSavedGame->LevelData.LevelsLoadedNotVisible.Add( PackageName ); }
		}
	}
	if ( PC )
	{
		CachedSavedGame->PlayerState PC->GetPlayerState();
		{
			UNFMission* CurrentMission = MissionManager->GetCurrentMission();
			if ( !bTransitSave && CurrentMission )
			{
				const FNFMissionInfo& MissionInfo = CurrentMission->GetInfo();
				CachedSavedGame->CurrentMissionName = MissionInfo.name;
				CachedSavedGame->CurrentPhaseMission = CurrentMission->GetPhase();
				CachedSavedGame->Info.Title = MissionInfo.nameUI;
				CachedSavedGame->Info.Description = MissionInfo.descriptionsUI[ CachedSavedGame-
>CurrentPhaseMission ];
			}
		}
	}
	Else { CachedSavedGame->PlayerState                        =     ANFPlayerController::GetDefaultPlay-
erState(); }
	CachedSavedGame->PlayerState.bIgnoreTransform= bTransitSave;
	CachedSavedGame->SavedActors.Empty();
	if ( !bTransitSave )
	{
		for ( FActorIterator It( World ); It; ++It )
		{
			AActor* Actor = *It;
			if ( Actor->IsPendingKillPending() || !Actor->ActorHasTag( ACTOR_TAG_SAVEGAME ) ) { con-
tinue; }
			FActorSaveData ActorData;
			ActorData.ActorName = Actor->GetFName();
			ActorData.Transform = Actor->GetActorTransform();
			FMemoryWriter MemWriter( ActorData.ByteData );
			FObjectAndNameAsStringProxyArchive Ar( MemWriter, true );
```

```
                  Ar.ArIsSaveGame = true;
                  Actor->Serialize( Ar );
                  CachedSavedGame->SavedActors.Add( ActorData );
            }
            {
                  FMemoryWriter MemWriter( CachedSavedGame->LevelData.ByteData );
                  FObjectAndNameAsStringProxyArchive Ar( MemWriter, true );
                  Ar.ArIsSaveGame = true;
                  World->GetLevelScriptActor()->Serialize( Ar );
            }
      }
      if ( NFHUD ) { NFHUD->OnShowSaveIcon( false, 3.f ); }
}
void UNFGameInstance::SaveGameToSlot( int32 InSlot )
{
      FString SaveFileName = UNuclearFrostUtils::FormatSaveGameSlot( InSlot );
      NF_CHECK( InSlot >= 0 );
      SaveGameToMemory();
      UGameplayStatics::SaveGameToSlot( CachedSavedGame, SaveFileName, 0 );
#if !UE_BUILD_SHIPPING
      if ( CVarEnableMakeDebugSaves->GetBool() ) { UGameplayStatics::SaveGameToSlot( CachedSavedGame, UNu-
clearFrostUtils::FormatDebugSaveGameSlot( CachedSavedGame->CurrentMissionName.ToString() ), 0 ); }
#endif // !UE_BUILD_SHIPPING
}
bool UNFGameInstance::LoadGameFromSlot( const FString& InSaveFileName, bool InLoadLevel /*= false*/ )
{
      CachedSavedGame = Cast<UNFSaveGame>( UGameplayStatics::LoadGameFromSlot( InSaveFileName, 0 ) );
      if ( !CachedSavedGame ) { return false; }
      if ( InLoadLevel )
      {
            FName LevelName = FName( CachedSavedGame->LevelData.LevelName );
            UNuclearFrostUtils::OpenLevelWithCustomLodingScreen( GetWorld(), LevelName, LevelName );
      }
      return true;
}
```

3. Для того щоб коректно завантажувались збереженні данні необхідко у класі ANuclearFrostGameMode реалізувати метод InitGame та HandleStartingNewPlayer_Implementation, це забезпечить відновлення станів акторів на момент збереження гри. Код цього кроку:

```
// NuclearFrostGameMode.cpp
...
void ANuclearFrostGameMode::InitGame( const FString& InMapName, const FString& InOptions, FString& InEr-
rorMessage )
{
      Super::InitGame( InMapName, InOptions, InErrorMessage );
      UNFSaveGame*       SaveGame = nullptr;
      {
            UNFGameInstance* GameInstance = Cast<UNFGameInstance>( GetGameInstance() );
            if ( GameInstance ) { SaveGame = GameInstance->GetCachedSavedGame(); }
      }
      if ( SaveGame && !SaveGame->Info.bTransitSave )
      {
            for ( uint32 Index = 0, NumLoadedLevels = SaveGame->LevelData.LevelsLoaded.Num(); Index <
NumLoadedLevels; ++Index )
            {
                  FLatentActionInfo LatentActionInfo;
                  LatentActionInfo.UUID = Index;
                  UGameplayStatics::LoadStreamLevel(    GetWorld(),    FName(    SaveGame->LevelData.Lev-
elsLoaded[Index] ), true, true, LatentActionInfo );
            }
            for ( uint32 Index = 0, NumLevelsLoadedNotVisible = SaveGame->LevelData.LevelsLoadedNotVisi-
ble.Num(); Index < NumLevelsLoadedNotVisible; ++Index )
            {
                  FLatentActionInfo LatentActionInfo;
                  LatentActionInfo.UUID = Index;
                  UGameplayStatics::LoadStreamLevel( GetWorld(), FName( SaveGame->LevelData.LevelsLoaded-
NotVisible[Index] ), false, true, LatentActionInfo );
            }
            GetWorld()->FlushLevelStreaming();
            for ( uint32 Index = 0, Count = SaveGame->SavedActors.Num(); Index < Count; ++Index )
```

```
		{
			const FActorSaveData& ActorData = SaveGame->SavedActors[ Index ];
			AActor* Actor = nullptr;
			for ( FActorIterator It( GetWorld() ); It; ++It )
			{
				AActor* TempActor = *It;
				if ( !TempActor->ActorHasTag( ACTOR_TAG_SAVEGAME ) || TempActor->GetFName() != 
ActorData.ActorName ) { continue; }

				Actor = TempActor;
				break;
			}
			if ( !Actor ) { continue; }
			Actor->SetActorTransform( ActorData.Transform );
			FMemoryReader MemReader( ActorData.ByteData );
			FObjectAndNameAsStringProxyArchive Ar( MemReader, true );
			Ar.ArIsSaveGame = true;
			Actor->Serialize( Ar );
		}
		if ( SaveGame->LevelData.ByteData.Num() > 0 )
		{
			FMemoryReader MemReader( SaveGame->LevelData.ByteData );
			FObjectAndNameAsStringProxyArchive Ar( MemReader, true );
			Ar.ArIsSaveGame = true;
			GetWorld()->GetLevelScriptActor()->Serialize( Ar );
		}
		if ( SaveGame->SavedActors.Num() > 0 )
		{
			for ( FActorIterator It( GetWorld() ); It; ++It )
			{
				AActor* Actor	= *It;
				Bool bFinded = false;
				if ( !Actor->ActorHasTag( ACTOR_TAG_SAVEGAME ) ) { continue; }
				for ( uint32 Index = 0, Count = SaveGame->SavedActors.Num(); Index < Count; ++Index 
)
				{
					if ( Actor->GetFName() == SaveGame->SavedActors[ Index ].ActorName )
					{
						bFinded = true;
						break;
					}
				}
				if ( !bFinded ) { Actor->Destroy(); }
			}
		}
	}
}
void ANuclearFrostGameMode::HandleStartingNewPlayer_Implementation( APlayerController* InNewPlayer )
{
	Super::HandleStartingNewPlayer_Implementation( InNewPlayer );
	UNFSaveGame* SaveGame = nullptr;
	{
		UNFGameInstance*	GameInstance = Cast<UNFGameInstance>( GetGameInstance() );
		if ( GameInstance ) { SaveGame = GameInstance->GetCachedSavedGame(); }
	}
	if ( SaveGame )
	{
		ANFPlayerController* PC = Cast<ANFPlayerController>( InNewPlayer );
		if ( PC ) { PC->RestoreFromPlayerState( SaveGame->PlayerState ); }
	}
}
```

3.4. Система завдань

Система завдань – це система, що допомагає гравцеві керувати та виконувати різноманітні завдання в грі. Зазвичай це включає в себе створення списку завдань або місій, які гравець повинен виконати для продовження гри або отримання додаткових бонусів.

Система завдань може містити інформацію про цілі завдань, їх стан, терміни виконання, нагороди та бонуси за виконання. Гравець може відстежувати прогрес виконання завдань та отримувати сповіщення про найбільш важливі та викликаючі завдання.

Система завдань у грі допомагає гравцеві керувати своїм часом та зосереджувати свої зусилля на досягненні мети гри.

Як система має працювати: буде два класи UNFMissions (клас завдання) та клас UNFMissionManager (клас який відповідає за прогрес та оновлення завдань). Усі завдання у менеджері будуть зберігатись як лінійний список, на рис. 3.6 зображено схематичний приклад такого зберігання.

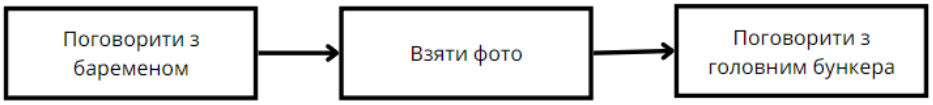

Рис. 3.6. Приклад слідування завдань

Для того щоб це реалізувати необхідно виконати наступні кроки:

1. Для початку необхідно створити структуру FNFMissionInfo, яка буде зберігати назву та опис місії, та клас UNFMission. Код цього кроку:

```
// Copyright Broken Singularity, All Rights Reserved.
// Authors: Yehor Pohuliaka (zombiHello)
#pragma once
#include "CoreMinimal.h"
#include "UObject/NoExportTypes.h"
#include "Engine/DataTable.h"
#include "NFMission.generated.h"
USTRUCT( BlueprintType )
struct NUCLEARFROST_API FNFMissionInfo : public FTableRowBase
{
    GENERATED_USTRUCT_BODY()
    FName name;
    int32 numPhases;
    FText nameUI;
    TArray< FText > descriptionsUI;
    TArray< bool > FlagsNeedSaveGame;
    TArray<FName> NotepadContentNames;
};
UCLASS( BlueprintType )
```

```
class NUCLEARFROST_API UNFMission : public UObject
{
    GENERATED_BODY()
public:
    UNFMission( const FObjectInitializer& InObjectInitializer = FObjectInitializer::Get() );
    virtual ~UNFMission();
    void Initialize( const FNFMissionInfo& InMissionInfo, TArray< class AMissionWaypointActor* > InWaypointActors, class ANFPlayerController* InOwnerPlayerController );
    void UpdateState( ENFMissionState InState );
    void UpdatePhase( int32 InNewPhase );
    FORCEINLINE const FNFMissionInfo& GetInfo() const {return info;}
    FORCEINLINE ENFMissionState GetState() const{return state;}
    FORCEINLINE uint32 GetPhase() const{return phase;}
    FORCEINLINE TArray< class AMissionWaypointActor* > GetWaypointActors() const{return waypointActors;}
    FORCEINLINE class ANFPlayerController* GetOwnerPlayerController() const{return ownerPlayerController;}
    FORCEINLINE void SetNextMission( UNFMission* InNextMission ){nextMission = InNextMission;}
    FORCEINLINE UNFMission* GetNextMission() const{return nextMission;}
    FNFChangedStateMissionDelegate onChangedStateMissionDelegate;
          FNFChangedPhaseMissionDelegate onChangedPhaseMissionDelegate;
private:
    void UpdateCompassMarker();
    ENFMissionState state;
    FNFMissionInfo info;
    int32 phase;
    TArray< class AMissionWaypointActor* > waypointActors;
    class ANFPlayerController* ownerPlayerController;
    class UUserWidget* compassMarkerWidget;
    class AMissionWaypointActor* currentWaypointActors;
    UNFMission* nextMission;
};
```

2. Далі необхідно створити UNFMissionManager який буде керувати всіма процесами (оновлення станів, виклик подій та інше). Код цього кроку:

```
// Copyright Broken Singularity, All Rights Reserved.
// Authors: Yehor Pohuliaka (zombiHello)
#pragma once
#include "CoreMinimal.h"
#include "UObject/NoExportTypes.h"
#include "NFMission.h"
#include "NFMissionManager.generated.h"
UCLASS()
class NUCLEARFROST_API UNFMissionManager : public UObject
{
    GENERATED_BODY()
public:
    /** Constructor */
```

```
    UNFMissionManager( const FObjectInitializer& InObjectInitializer = FObjectInitializer::Get()
);
    virtual ~UNFMissionManager();
    void AddNextMission( UNFMission* InMission );
    void RemoveAllMission();
    UNFMission* FindMission( const FName& InName ) const;
    void SetCurrentMission( UNFMission* InMission );
    void Init();
    FORCEINLINE UNFMission* GetCurrentMission() const{return currentMission;}
    FORCEINLINE bool IsInit() const{return bInit;}
    FNFChangedCurrentMissionDelegate             onChangedCurrentMissionDelegate;
private:
    void RemoveMissionByPointer( UNFMission* InMission );
    void OnChangedStateMission( UNFMission* InMission, ENFMissionState InState );
    void OnChangedPhaseMission( UNFMission* InMission, int32 InPhase );
    UNFMission* currentMission;
    TArray< UNFMission* > missions;
    Bool bInit;
};
```

3. Тепер ми можемо на Blurprint створити завдання, на рис. 3.7 зображено приклад такого скрипта.

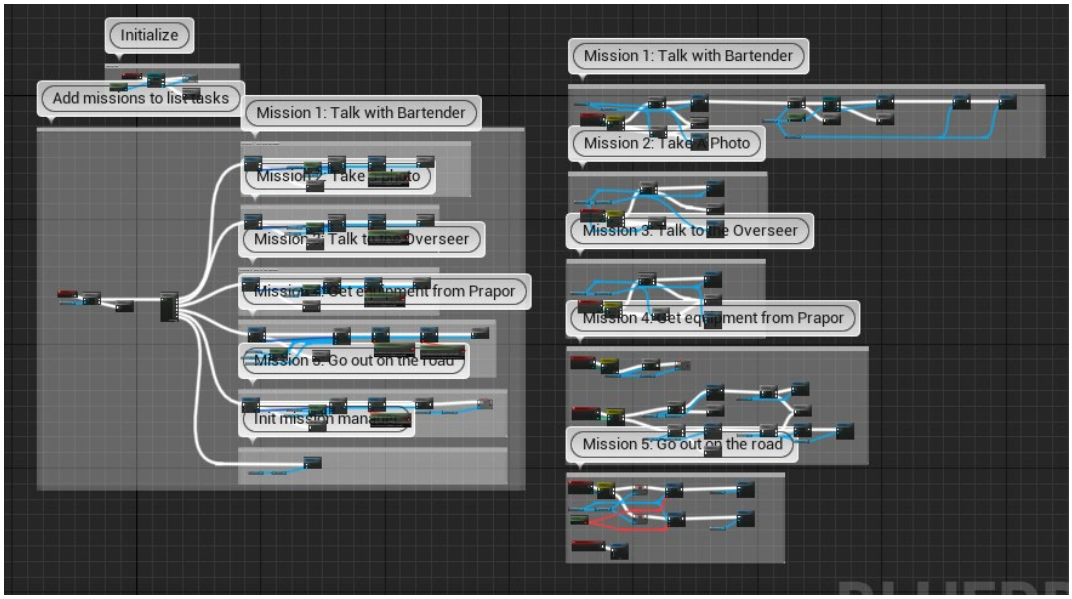

Рис. 3.7. Приклад скриптів завдань для локації Shelter

3.5. Система діалогів

Система діалогів – це механізм, що дозволяє гравцеві взаємодіяти з іншими персонажами гри через текстові або голосові діалоги. Система діалогів

може бути вбудована в гру, щоб допомогти гравцеві отримати більш глибоке розуміння сюжету гри та мотивації персонажів.

Зазвичай система діалогів у грі дозволяє гравцеві вибирати різні варіанти відповіді на запитання чи дії персонажів гри. Відповідно до вибору гравця, діалог може рухатися в різні напрямки, впливаючи на подальший розвиток сюжету.

Система діалогів може також використовуватися для навчання гравця, зокрема, для навчання гравця різних навичок або інформування про правила гри. В цілому, система діалогів є важливою складовою гри, що допомагає забезпечити глибину та взаємодію гравця з грою та її персонажами.

Як система має працювати: текст діалогів буде зберігатись у таблицях, сам діалог буде працювати за допомогою Behavior Tree та має бути UDialogSystemComponent, який забезпечить працю цієї системи.

Для того щоб це реалізувати необхідно виконати наступні кроки:

1. Створюємо компонент UDialogSystemComponent, який забезпечить роботу діалогів та буде запускати Behavior Tree.

2. Створити віджет для відображення діалогів та варіантів відповідей на екрані. На рис. 3.8 зображено створенний віджет для цього кроку.

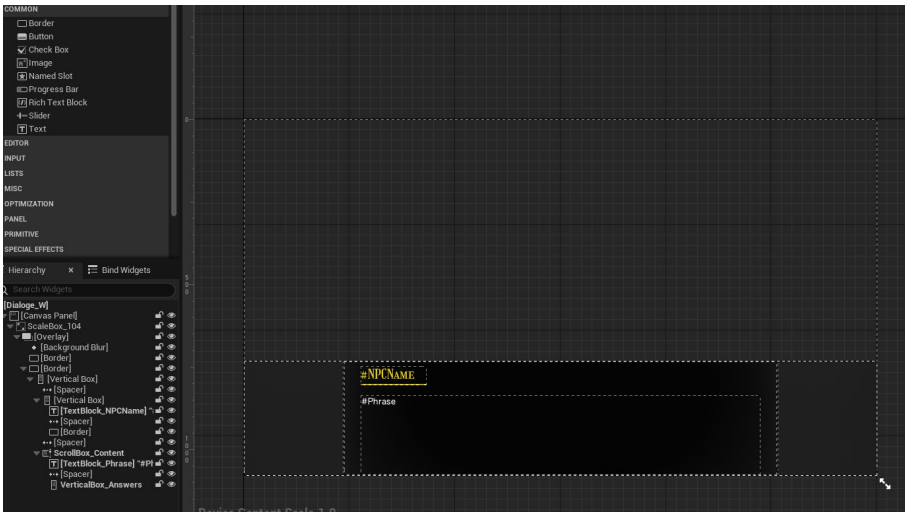

Рис. 3.8. Створенний віджет для діалогів

3. Для того щоб у Behavior Tree можливо було створювати логіку діалогів, потрібно створити декілька классів успадкованих від UBTTaskNode. Які класи необхідно створити: UDialogSayBTTaskNode (розпочати діалог), UDialogStopBTTaskNode (зупинити діалог), UDialogSayBTTaskNode (сказати фразу та чекати відповіді від гравця) та UUpdateMissionStateBTTaskNode (оновити стан поточного завдання). На рис. 3.9 зображено створенні ноди які можливо використовувати у Behavior Tree.

Рис. 3.9. Створенні завдання Behavior Tree для роботи діалогів

4. Створити таблицю де буде зберігатись фраза та варіанти відповедей. На рис. 3.10 зображено створену таблицю.

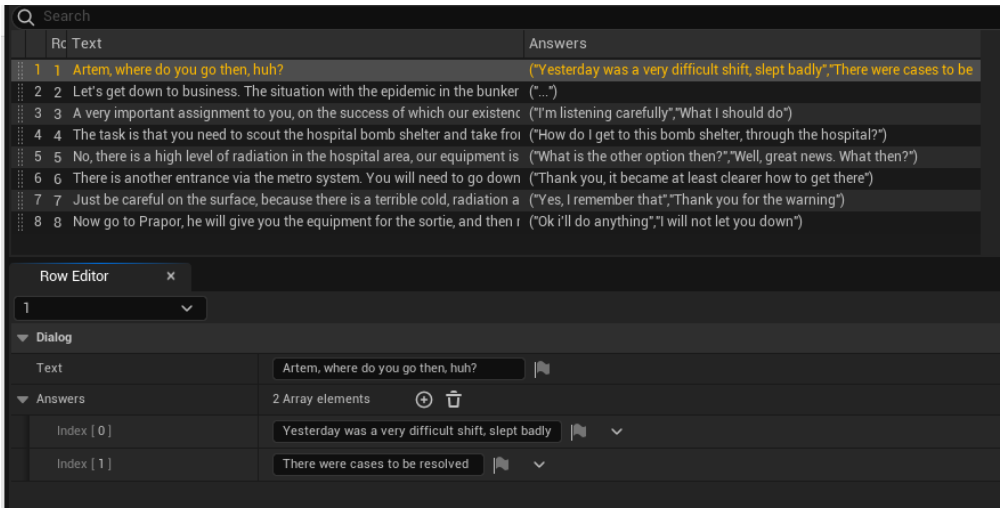

Рис. 3.10. Створена таблиця з фразами та відповідями

5. Створити дерево діалогу з використанням нодів, які були створенні на попередніх кроках. На рис. 3.11 зображено приклад створенного дерева діалогу.

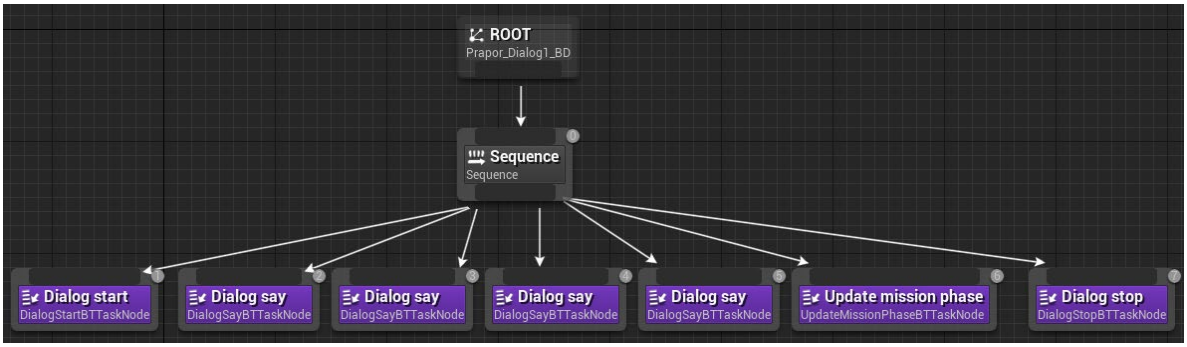

Рис. 3.11. Створено дерево діалогу

6. Створити персонажа, з яким гравець має спілкуватись, додати до нього компонент UDialogSystemComponent та вказати необхудну таблицю фраз та дерево діалогу. На рис. 3.12 зображено створенного персонажа з яким гравець може спілкуватись.

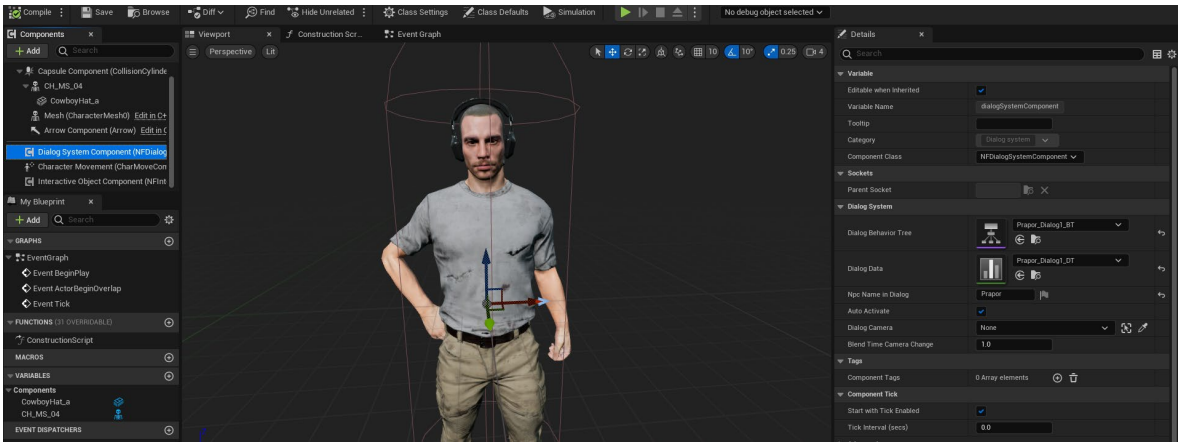

Рис. 3.12. Створений персонаж з яким можливо говорити

7. Розмістити персонажа на мапі. На рис. 3.13 зображено те, як система діалогів працює у грі.

Рис. 3.13. Створенні діалоги у Nuclear Frost

3.6. Механіка холоду та радіації

Ігрова механіка холоду та радіації – це елементи геймплею в комп'ютерних іграх, які спрощено моделюють вплив низьких температур і високої радіації на гравця або його персонажа.

Механіка холоду відтворює вплив низьких температур на персонажа гравця. Зазвичай це означає, що гравець повинен збирати різні ресурси, щоб уникнути замерзання або отримання пошкоджень від холоду. Гравці можуть використовувати різноманітні засоби, такі як теплові одяг, багатопредметні керосинові пальники, щоб залишатися в теплі.

Механіка радіації відображає вплив високої радіації на персонажа гравця. В комп'ютерних іграх це може включати в себе втрату здоров'я, рак, або інші наслідки від довготривалого впливу радіації. Гравці можуть використовувати захисні препарати, такі як протирадіаційні ліки, щоб запобігти впливу радіації на свого персонажа.

Обидва ці елементи додають глибину до геймплею та можуть бути додані до різних жанрів, від відкритого світу до науково-фантастичних ігор.

Як це буде працювати: весь час гравець на вулиці буде замерзати та щоб погрітися необхідно знайти та розпалити багаття, а радіація буде знаходитись в певних місцях, та якщо туди зайти то буде зараження.

Для того щоб це реалізувати необхідно виконати наступні кроки:

1. Створити компонент UNFHeatSourceComponent який буде відповідати на теплову зону на мапі.

2. Написати логику радіації та холоду в ANFPlayerController.

3. Створити у Unreal Engine актора, який буде виконувати роль джерела тепла. На рис. 3.14 зображено приклад створенного актора.

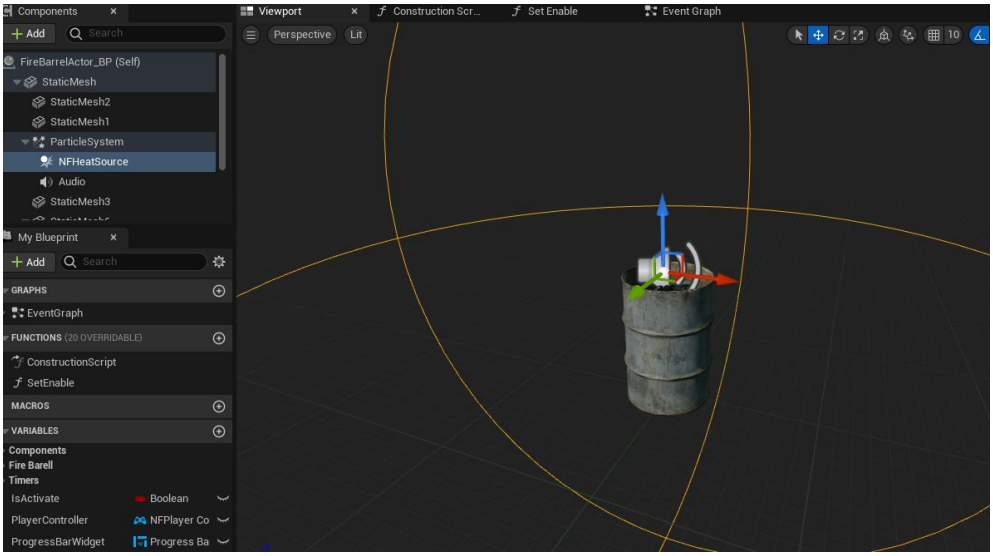

Рис. 3.14. Створене джерело тепла

4. У актора необхідно на Blueprint написати логику використання предмету гравцем. На рис. 3.15 зображено реалізований скрипт на Blueprint

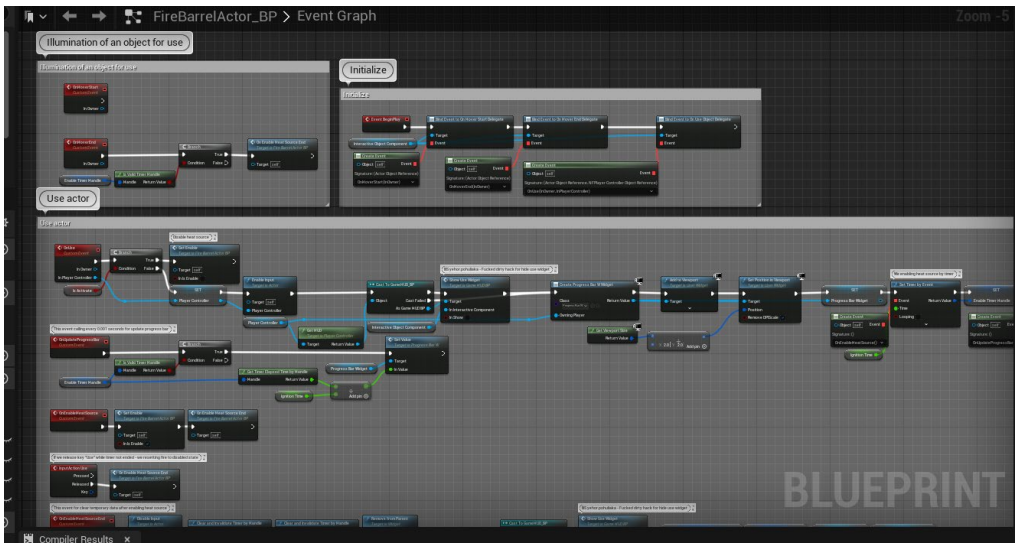

Рис. 3.15. Реалізований скрипт на Blueprint

5. Необхідно розмістити актора на мапі. На рис. 3.16 зображено актора який був розміщений на мапі.

Рис. 3.16. Розміщений актор на мапі Forest

6. Необхідно створити матеріал постпроцесу який буде відповідати ефекту радіації. На рис. 3.17 зображен такий матеріал.

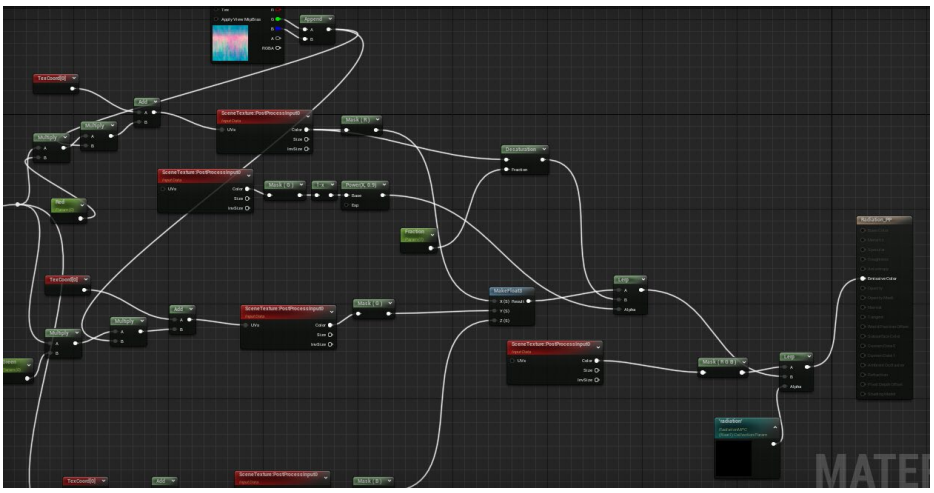

Рис. 3.17. Матеріал постпроцесу радіації

3.7. Монстри

Монстри – це персонажі в комп'ютерних іграх, які призначені для битви з гравцем або його персонажем. Вони можуть мати різні форми, розміри, властивості та можуть походити з різних світів та культур. У багатьох іграх монстри є ключовими геймплейними елементами, які гравцеві потрібно подолати для досягнення мети.

Монстри можуть бути створені з різних матеріалів, таких як метал, камінь, м'ясо або навіть з магічних елементів, таких як етер або мана. Вони можуть мати свої унікальні властивості, такі як супершвидкість, вогненні кулі або інші небезпечні атаки, що роблять бій з ними більш складним і захоплюючим.

У деяких іграх гравці можуть також взаємодіяти з монстрами, виконувати завдання або навіть приручати їх, щоб отримати користь у грі. У багатьох іграх монстри можуть бути показані як частина більшої історії або фантазії, доповнюючи світ та додаючи глибину до геймплею.

Як це буде працювати: створимо клас персонажа, контролера та за допомогою Behavior Tree запрограмуємо поведінку ворога.

Для того щоб це реалізувати необхідно виконати наступні кроки:

1. Створити клас AMonsterCharacter який буде описувати монстра як персонажа.

2. Створити клас AMonsterAIController який буде керувати AMonsterCharacter та реагувати на різні події у світі гри.

3. Створити у Unreal Engine дерево поведінки Behavior Tree, яке буде описувати те, як монстр буде себе поводити при різноманітних подіях (атакувати, лікуватись та інше). На рис. 3.18 зображено створенне дерево поведінки.

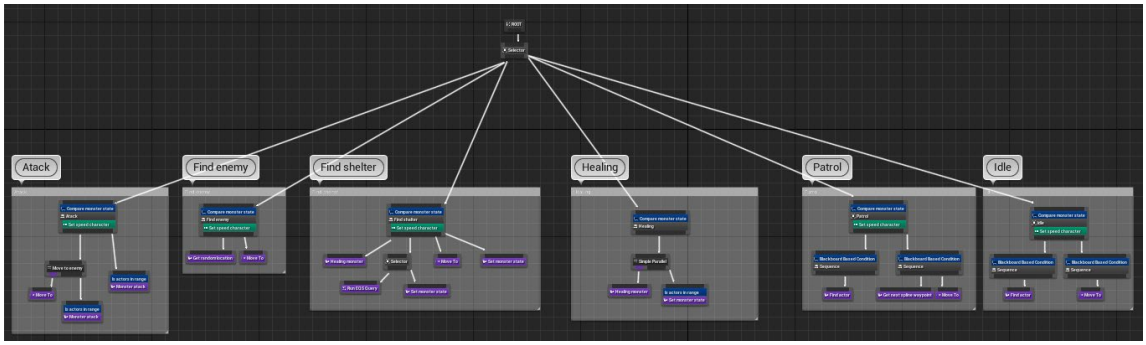

Рис. 3.18. Дерево поведінки монстра

4. Створити Blueprint актора та розташувати на мапі. На рис. 3.19 зображено монстра розташованого на мапі.

Рис. 3.19. Монстр на мапі Forest

Висновки до розділу

Під час реалізації головних маханік та систем гри було розглянуто та створенно наступні елементи: завантажуючий екран, система завдань, система діалогів, монстри, інтерактивна система та механіки холоду і радіації.

РОЗДІЛ 4. ТЕСТУВАННЯ ІГОР

4.1. Базові стратегії тестування ігор

Тестування гри - це захоплюючий етап процесу розробки відеоігор. Незважаючи на багатство дивовижних ігор на ринку, всі вони проходять суворе тестування перед випуском. У цьому процесі тестери ігор стараються виявити помилки і баги, такі як невидимі стіни, деформовані персонажі, проблеми з кодом мережі, відсутність текстур, незбалансована зброя, діри на карті, довгий час завантаження та нестабільні елементи керування. Гарантія якості є надзвичайно важливою частиною процесу розробки гри, оскільки вона забезпечує бездоганний, захоплюючий і приємний ігровий досвід для гравців, без неприємних перерв.

Для успішного виявлення помилок у грі, тестери виконують кілька основних кроків:

– планування та проектування тесту. Цей перший крок включає в себе задання запитань, таких як: які зміни були внесені в дизайн від останньої збірки, які нові конфігурації підтримує гра, які додаткові тестові сценарії були додані, які функції були вилучені;

– підготовка до тестування. Власники гри або видавці готують документи та тестові середовища, необхідні для проведення тестування гри;

– виконання тестів. Набори тестів запускаються з новою збіркою гри. Якщо виявляється помилка, вона перевіряється детально, щоб отримати всі необхідні деталі, які будуть включені до звіту;

– повідомлення про результати. На цьому етапі складається звіт з усіма виявленими помилками та інформацією про них;

– виправлення помилок. Виявлені помилки обговорюються командою тестувальників разом з розробниками гри, і встановлюється правильне рішення для виправлення проблеми.

Тестування гри, як і будь-який інший етап у процесі її розробки, вимагає планування для досягнення найкращих результатів. Стратегічне планування відіграє важливу роль у виявленні можливих проблем, що можуть виникнути під час тестування гри, та визначенні ефективних сценаріїв їх вирішення. Це сприяє полегшенню отримання позитивних результатів. Крім того, на етапі планування стратегії тестування гри визначаються ролі команди, інструменти та документація, які будуть використовуватися.

Процес тестування гри може бути реалізований за допомогою різних стратегій, які використовуються тестувальниками ігор. До найпопулярніших стратегій належать:

– спеціальне тестування: Ця стратегія передбачає виконання тестування вільною формою, де тестувальник ігор має бути розслабленим, але зосередженим на пошуку помилок. Спеціальне тестування дозволяє виявити помилки, які інакше було б складніше виявити.

– тестування функціональності: Ця стратегія спрямована на перевірку того, чи працює остаточна гра відповідно до початкових специфікацій. Зазвичай перевіряються загальні проблеми, пов'язані з графікою, інтерфейсом користувача, звуком або механікою гри. На цьому етапі не оцінюється розважальний аспект гри, але акцент робиться на основних елементах, які мають бути функціональними. На рис. 4.1 зображено приклад помилки у грі.

Рис. 4.1. Приклад помилки у The Witcher 3: Wild Hunt

– тестування на сумісність: Ця стратегія тестування гри визначає, чи є гра оптимізованою для різних розмірів екранів та чи відповідає вона основним вимогам програмного забезпечення, апаратного забезпечення та графіки;

– тестування прогресу: Цей метод спрямований на те, щоб протестувати, чи можна пройти гру повністю без жодних перерв чи збоїв. Тестувальники ігор шукають проблеми та затримки під час прогресування у лінійній грі. Більшість зупинок у прогресі зазвичай пов'язані з проблемами в сценаріях гри, але іноді можуть бути викликані непередбачуваними ситуаціями;

– регресійне тестування: Після виправлення помилки розробниками, її вважають виправленою назавжди, але існує ризик того, що вона може викликати нові помилки. Для вирішення цієї проблеми використовується регресійне тестування. Цей метод дозволяє тестувальникам ігор знайти старі помилки у поточному коді. Хоча регресійне тестування може здатися не таким важливим, воно допомагає виявити неприємні проблеми.

4.2. Особливості тестування ігор

Незважаючи на те, що тестування ігор і тестування програмного забезпечення можуть мати схожі принципи, оскільки обидва вимагають перевірки коду для забезпечення високої якості робочого програмного забезпечення, між ними існують значні відмінності.

У відміну від тестування ігор, тестування програмного забезпечення включає в себе використання автоматизованих сценаріїв, які розробляються до і після створення тестів. Для налаштування таких автоматизованих сценаріїв, тестувальники користуються різноманітними інструментами та фреймворками під час процесу тестування програмного забезпечення.

У відміну від тестування програмного забезпечення, тестування ігор включає велику кількість аспектів, особливо в разі мобільних ігор, які потребують особливої уваги. Деякі з цих аспектів включають:

– графіка є ключовим фактором у привертанні користувачів до гри. Геймерам часто подобається реалістична графіка, яка підтримується пристроєм і працює на ігровому двигуні. Фактично, успіх гри на ринку сьогодні неможливий без вражаючої графіки. У цьому випадку продуктивність гри - це не окремий параметр, а сукупність багатьох факторів, які кожен гравець бере до уваги. Наприклад, споживання заряду батареї - якщо мобільна гра споживає надмірно багато енергії, гравці можуть відмовитися від неї.

– крім того, тестери ігор повинні запускати гру на різних пристроях і вимірювати важливі параметри пристрою, такі як використання процесора, температура пристрою, використання графічного процесора, використання даних та багато іншого.

Для функціонального тестування ігор часто потрібні спеціалізовані фреймворки. Деякі елементи геймплею та поведінки не можуть бути повністю автоматизовані і вимагають ручної перевірки. Навіть у випадках, коли автоматизація застосовується, вона вимагає значних зусиль та використання інструментів автоматизації для створення тестових сценаріїв, оскільки випадки використання в іграх зазвичай виходять за межі простого пошуку статичних елементів на екрані.

Тестування програмного забезпечення переважно фокусується на перевірці інтерфейсу користувача, функціональності, безпеці та системних аспектах, тоді як тестування ігор акцентується на досягненні реалістичності, штучного інтелекту та багатокористувацькому геймплеї.

4.3. Тестові випадки для Nuclear Frost

Таблиця 4.1 – Тестові випадки

Опис	Предумова	Кроки	Очікуваний результат
Перевірка додавання місії	Немає	1. Створити новий об'єкт UNFMissionManager. 2. Створити об'єкт місії з необхідними параметрами. 3. Викликати метод AddMission() на об'єкті UNFMissionManager, передавши об'єкт місії як аргумент.	Додання місії до списку місій у UNFMissionManager.
Перевірка видалення місій	Місії додани до UNFMission Manager	1. Створити новий об'єкт UNFMissionManager. 2. Створити два об'єкта місії з необхідними параметрами. 3. Викликати метод AddMission() на об'єкті UNFMissionManager, передавши об'єкти місії як аргумент. 4. Викликати метод RemoveAllMission() на об'єкті UNFMissionManager.	Видаленні усі місії зі списку у UNFMissionManager.
Перевірка збереження гри	Немає	1. Створити новий об'єкт UNFSaveGame. 2. Встановити значення рівня, кількості життів та інші параметри гри. 3. Викликати метод SaveGameToSlot () на об'єкті UNFGameInstance.	Збереження гри з встановленими параметрами.
Перевірка завантаження гри	Гра була збережена раніше	1. Створити новий об'єкт UNFSaveGame. 2. Викликати метод LoadGameFromSlot () на об'єкті UNFGameInstance.	Завантаження гри з раніше збереженими параметрами.
Перевірка відкривання	Двері в початковому	1. Створити новий об'єкт ADoorActor.	Двері переходять у закритий

двері	закритому стані	2. Викликати метод Use() на об'єкті ADoorActor.	стан.
Перевірка закривання двері	Двері в початковому відкритому стані	1. Створити новий об'єкт ADoorActor. 2. Викликати метод Use() на об'єкті ADoorActor.	Двері переходять у закритий стан.
Перевірка блокування двері	Двері в початковому відкритому або закритому стані	1. Створити новий об'єкт ADoorActor. 2. Викликати метод UpdateState() з параметром DS_Locked на об'єкті ADoorActor.	Двері блокуються і переходять у стан "заблоковано".
Перевірка розблокування двері	Двері в початковому заблокованому стані	1. Створити новий об'єкт ADoorActor. 2. Викликати метод UpdateState() з параметром DS_Closed на об'єкті ADoorActor.	Двері розблоковуються і переходять у стан "розблоковано".
Перевірка початку діалогу	Немає активного діалогу	1. Створити новий об'єкт UNFDialogSystemComponent. 2. Викликати метод DialogStart() на об'єкті UNFDialogSystemComponent.	Початок діалогу з вказаного початкового вузла.
Вибір варіанту в діалозі	Наявність діалогу з варіантами відповідей	1. Створення UNFDialogSystemComponen 2. Створення діалогу з варіантами відповідей 3. Вибір варіанту відповіді гравцем	Обрання відповідного варіанту та продовження діалогу відповідно до вибраного варіанту

Висновки до розділу

Оскільки популярність ігор продовжує зростати, увага до тестування також зростатиме. Тому кожен ігровий бізнес сьогодні зосереджується на впровадженні ефективних методів і стратегій тестування, щоб забезпечити належний ігровий досвід для своїх клієнтів. На початку необхідно уважно стежити за життєвим циклом розробки гри та враховувати різні типи тестування, щоб створити гру саме так, як хотіли б ваші клієнти. Крім того, тестування гри – це повторюваний процес, який необхідно виконувати для виявлення та усунення нових дефектів і помилок з кожним новим випуском гри.

Загальні висновки

Результатом роботи є відеогра Nuclear Frost розроблемна на мові C++ з використанням рушія Unreal Engine 5 для платформи Windows, яка занурює гравця в холодний світ постапокаліптичного світу, де герой має боротися за виживання.

На осовнові розглянутих та описаних аналогів, з оцінкою їх плюсів та мінусів, були поставлені задачі проектування та розробки проекту.

На етапі організації команди та середовища було розглянуто, що це є важливими аспектами у створенні ігор. Команда розробників має бути організована та координована для ефективної роботи над проектом. Дорожня карта визначає план розробки гри та допомагає контролювати хід роботи. Системи контролю версій, такі як SVN та Perforce, забезпечують відстеження змін та спільну роботу над кодом та ресурсами гри. Blueprints та Behavior Trees у Unreal Engine надають інструменти для створення ігрових механік та поведінки об'єктів у грі.

На етапі розробки було розглянуто різні аспекти створення гри. Плагін завантажувального екрану допомагає створити привабливий та інформативний початок гри. Інтерактивна система надає можливість взаємодії гравця з ігровим світом та об'єктами. Система збереження гри дозволяє зберігати прогрес гравця та відновлювати гру зі збереженої точки. Система місій пропонує гравцю цілі та завдання для виконання у грі. Система діалогів створює можливість комунікації з NPC та розвитку сюжету гри. Механіка холоду та радіації вносить елементи виживання та стратегії в ігровий процес. Монстри створюють загрозу та викликають напругу, додаючи динаміку до ігрового світу.

На етапі тестування було розглянуто, що тестування відеоігор має свої особливості та стратегії. Базові стратегії тестування ігор включають функціональне тестування та тестування продуктивності.

Список використаних джерел

1. Навчальний посібник для підготовки кваліфікаційної роботи на здобут-тя ОС Бакалавр 2.

2. Герберт Шілдт. Повний довідник за C++ 4 видання. – М.: Osborne, 2016. – 800 с.

3. Технічна документація з Unreal Engine [Електронний ресурс] https://docs.unrealengine.com/

4. [Електронний ресурс] Організація команди - https://relevant.software/blog/how-to-effectively-manage-your-remote-development-team/

5. [Електронний ресурс] Вілсон, Грегорі; Рейд, Карен - https://dl.acm.org/doi/10.1145/1047124.1047441

6. [Електронний ресурс] Unreal Engine Blueprint - https://gamedev.dou.ua/blogs/visual-programming-blueprints/

7. [Електронний ресурс] Unreal Engine Behavior Tree - https://docs.unrealengine.com/5.0/en-US/behavior-trees-in-unreal-engine/

8. [Електронний ресурс] Створення роадмапи - https://wezom.com.ua/ua/blog/chto-takoe-roadmap-vidy-primery

9. [Електронний ресурс] Жанри відеоігор - https://www.hp.com/us-en/shop/tech-takes/video-game-genres

10. Джейсон Грегори. Архітектура ігрового двигуна 3 видання. – М.: New York, 2021. – 1130 с.

11. Meyer B (1997) Об'єктно-орієнтована конструкція програмного забезпечення, 2-ге видання. Прентіс Холл, Енглвуд Кліпп, Нью-Джерсі

12. Роберт Нистром. Патерни програмування ігор. – М. Boston, 2014. – 321 с.

Додатки

Додаток А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ

Проректор Українського державного університету науки і технологій

Анатолій РАДКЕВИЧ

26.12.22

РОЗРОБКА ЯДРА КОМП'ЮТЕРНОЇ ГРИ ЖАНРУ ПРИГОД НА UNREAL ENGINE

Технічне завдання

ЛИСТ ЗАТВЕРДЖЕННЯ

1116130.01322-01-ЛЗ

Представники

підприємства-розробника

Завідувач кафедри КІТ

Вадим ГОРЯЧКІН

26.12.22

Керівник розробки

Олександр ІВАНОВ

26.12.22

Виконавець

Єгор ПОГУЛЯКА

26.12.22

Норм-контролер

Світлана ВОЛКОВА

26.12.22

2023

ЗАТВЕРДЖЕНО

1116130.01322-01-ЛЗ

РОЗРОБКА ЯДРА КОМП’ЮТЕРНОЇ ГРИ ЖАНРУ ПРИГОД НА UNREAL ENGINE

Технічне завдання

1116130.01322-01

Листів 14

2023

ЗМІСТ

1 ВВЕДЕННЯ

Відеогра “Nuclear Frost” – це пригодницький шутер від першої особи, де герою доведеться зіткнутися віч-на-віч із погрозами, які породив новий світ. Боріться з мутантами, зберігайте своє тепло, уникайте радіації та боріться з грибком, щоб урятувати свій будинок. Світ гри давним-давно був зруйнований ядерною війною, що знищила населення Землі і перетворила її поверхню на крижану пустку. Лише жменька тих, хто вижив, сховалася в надрах бункера, а людська цивілізація виявилася на межі вимирання через грибок Крижаної Орусфаїри, що з'явився на світ.

Головний герой, за якого буде грати гравець, не застав самої війни, він народився вже в бункері і лише з розповідей знав який світ був "До". Весь час його житло зазнавало спалахів епідемії Льодяної Орусфаїри, які вдавалося ліквідувати, але тільки не зараз. Ситуація загострилася настільки, що майже більшість населення була заражена і герой у тому числі. Настав момент рішуче діяти, тому перед героєм поставлене завдання врятувати за будь-яку ціну свій рідний дім, навіть якщо за це доведеться віддати своє життя.

Головна мета гри полягає в тому, щоб занурити гравця в атмосферу замерзлого світу та боротьби людей за своє існування, а також показати, наскільки може бути небезпечним застосування ядерної зброї та зневага проблем клімату.

Програмний продукт має наступні ключові особливості гри:

– Nuclear Frost – це захоплюючий дух атмосферний шутер від першої особи. Відчуйте себе в мертвому замерзлому світі;

– станьте свідком жаху, який породжує грибок Крижана Орусфаїра і боріться з ним;

– досліджуйте світ спустошений льодом та радіацією;

– боріться за своє тепло;

– допоможіть героєві врятувати свій будинок від вимирання.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є наказ від 07.12.22 №1209ст ректора Українського державного університету науки і технологій “Про призначення керівників та затвердження тем бакалаврських робіт” за спеціальністю 121 “Інженерія програмного забезпечення» факультету “Комп’ютерних технологій і систем” по кафедрі “Комп’ютерні інформаційні технології”.

Тема дипломної роботи - “РОЗРОБКА ЯДРА КОМП’ЮТЕРНОЇ ГРИ ЖАНРУ ПРИГОД НА UNREAL ENGINE”. Керівник - доцент Іванов О. П.

3 ПРИЗНАЧЕННЯ РОЗРОБКИ

Функціональне призначення полягає у керуванні головним героєм, дослідження світу, взаємодією з предметами, діалогами з персонажами, виконання завдань, бойову систему, арт-дизайн та атмосферу.

Експлуатаційне призначення полягає у зануренні в захоплюючий ігровий світ та розважанні гравця.

4 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

4.1 Вимоги до функціональних характеристик

Програмний продукт має включати наступні пункти для керування головним героєм:

– плавне та чуйне керування персонажем;

– можливість переміщення в різних напрямках, бігу, стрибків та інших дій.

До дослідження ігрового світу має включати наступні пункти:

– великий та різноманітний віртуальний світ для дослідження;

– реалістичні та деталізовані локації з різними інтерактивними елементами;

До взаємодії з предметами має включати наступні пункти:

– можливість взаємодії з різними предметами в ігровому світі, такими як двері, важелі, вимикачі та інші;

– можливість збирати предмети та використовувати їх для вирішення головоломок та виконання завдань.

До діалогів з персонажами має включати наступні пункти:

– система діалогів, що дозволяє гравцеві взаємодіяти з іншими персонажами гри;

– різні варіанти відповідей та діалогів;

– текстові діалоги з можливістю вибору відповідей.

До виконання завдань має включати наступні пункти:

– різноманітні завдання та квести, які гравець може виконувати під час гри;

– чіткі цілі та інструкції для виконання завдань;

– відстеження прогресу виконання завдань.

До бойової системи має включати наступні пункти:

– система бою, що дозволяє гравцеві боротися з ворогами чи монстрами у грі.

4.2 Вимоги до надійності

Вимоги до надійності наступні:

– гра повинна бути стабільною та надійною, не викликаючи системні збої чи вильоти;

– гра повинна забезпечувати надійне збереження прогресу гравця;

– гра повинна бути оптимізована для досягнення високої продуктивності.

4.3 Вимоги експлуатації

Програмний продукт повинен використовуватись у приміщеннях які відповідають умовам роботи ЕОМ, а саме мають такі кліматичні, санітарні та гігієнічні умови, які відповідають ДНАОП 0.00-1.13-99 (див. табл. 1).

Таблиця 1. Кліматичні умови

Пора року	Категорія робіт згідно з ГОСТ 12.01-005-88	Температура повітря, град.С	Відносна вологість повітря, %	Швидкість руху повітря, м/с
		Оптимальна	Оптимальна	Оптимальна
Холодна	легка-1-а	22-24	40-60	0,1
	легка-1-б	21-23	40-60	0,1
Тепла	легка-1-а	23-25	40-60	0,1
	легка-1-б	22-24	40-60	0,2

Працювати з програмою може людина, що має базові навички роботи з комп’ютером.

4.4 Вимоги до складу та параметрів технічних засобів

Продукт, що розробляється має наступні мінімальні технічні вимоги до характеристик комп’ютера:

- 64-розрядні процесори та операційна система;
- оперативна пам’ять – 4 ГБ;
- відеокарта – NVIDIA GTX 1650 або вище;
- операційна система – Windows 10 або вище;
- DirectX – версія 12;
- процесор – Intel Core i3 або вище.

4.5 Вимоги до інформаційної та програмної сумісності

Відеогра розробляється для всіх видів операційних систем сімейства “Windows” починаючи від версії 10 та наступні версії.

4.6 Вимоги до маркування і упаковки

Відеогра продаватиметься на майданчиках цифрової дистрибуції, тому упаковка не потрібна.

Замість упаковки має бути лише зображення з логотипом гри, яка виконуватиме маркетингову роль для залучення покупців.

4.7 Вимоги до транспортування та зберігання

Транспортування повинне забезпечувати збереження програмного продукту його цілісність і запобігання несанкціонованого доступу до нього. Програмний виріб міститься на серверах цифрової дистрибуції та передаватиметься по Ethernet.

5 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

До складу документації мають входити:

- специфікація;
- текст програми;
- опис програми;
- керівництво користувача.

Вся документація програмного додатку повинна задовольняти вимоги до програмної документації.

.

6 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Таблиці 1. – Стадії та етапи розробки

Стадія	Зміст	Строки виконання
Технічне завдання	Постановка задачі, збір інформації, виріб та обґрунтування критеріїв розробки. Попередній вибір методів рішення задач. Визначення вимог до технічних засобів. Узгодження і затвердження технічного завдання.	07.12.22 -26.12.22
Робочий проект	Програмування та відлагодження програми.	27.12.22 - 27.04.23
	Тестування програми	28.04.23 - 05.05.23
	Розробка, узгодження і затвердження програмної документації.	06.05.23 - 19.06.23

7 ПОРЯДОК І КОНТРОЛЬ ПРИЙМАННЯ

Контроль за виконанням роботи здійснює керівник розробки доц. Іванов О. П.

Прийом здійснюється комісією у складі:

- Горячкін В. М. (керівник підрозділу);
- Іванов О. П. (керівник розробки).

8 БІБЛІОГРАФІЧНИЙ СПИСОК

1. Івченко, Ю.М. Основи стандартизації програмних систем: методичні вказівки до дипломного проектування та лабораторних робіт/уклад.: Ю.М. Івченко, В. І. Шинкаренко, В. Г. Івченко; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Д.: Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В. Лазаряна, 2009. - 38