Міністерство освіти і науки України

Ніжинський державний університет імені Миколи Гоголя

Навчально-науковий інститут природничо-математичних, медико-біологічних наук та інформаційних технологій

Кафедра інформаційних технологій, фізико-математичних та економічних наук

Освітня програма: Комп’ютерні науки

Спеціальність: 122 Комп’ютерні науки

**КВАЛІФІКАЦІЙНА РОБОТА**

на здобуття освітнього ступеня *бакалавр*

**Розробка системи для створення сюжетно-орієнтованих ігор на С++ та рушія Unreal Engine**

студента **Циганкова Андрія Сергійовича**

**Науковий керівник:**

Фетісов Валерій Сергійович,

кандидат економічних наук, доцент

**Рецензент:**

Казачков Іван Васильович,

доктор технічних наук, професор

**Допущено до захисту:** \_\_\_\_ \_\_\_\_\_\_\_ 2023 р.

Завідувач кафедри

проф. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Казачков І.В.

Ніжин – 2023

Вступ

РОЗДІЛ 1. ЗАГАЛЬНА ХАРАКТЕРИСТИКА СФЕРИ РОЗРОБКИ КОМП’ЮТЕРНИХ ІГОР

1.1. Еволюція сюжетно-орієнтованих ігор

1.2 Аналіз ринкових тенденцій в розробці ігор

1.2.1 Дослідження платформ дистрибуцій комп’ютерних ігор

1.2.2 Загальний огляд процесу розробки ігри

1.3 Обґрунтування вибору рушія Unreal Engine 5

1.3.1 Переваги та недоліки Unreal Engine 5 у порівнянні з іншими ігровими рушіями

1.3.2 Аналіз популярності Unreal Engine 5 та його застосування

РОЗДІЛ 2. РОЗРОБКА СИСТЕМИ ДЛЯ СТВОРЕННЯ СЮЖЕТНО-ОРІЄНТОВАНИХ ІГОР НА МОВІ ПРОГРАМУВАННЯ С++ ТА РУШІЯ UNREAL ENGINE 5.

2.1 Огляд реалізації Garbage Collection рушія та його системи рефлексії

2.2 Аналіз способів взаємодії мови програмування С++ та внутрішньої мови програмування рушія – Blueprints

2.3 Проектування та розробка системи створення сюжетно-орієнтованих ігор

2.3.1 Розробка інтерактивних об'єктів та виконавців в ігровому світі

2.3.2 Розробка системи створення сюжету

2.3.3 Розробка системи серіалізації даних

2.3.4 Розробка UI

2.4 Рекомендації щодо подальшого розвитку написаного модуля

ВИСНОВКИ

СПИСОК ВИКОРИСТАНИХ ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

ДОДАТКИ

Вступ

**Актуальність теми.** Комп’ютерні ігри вже давно стали незмінною частиною сучасної культури, яка дає можливість людині відпочити від роботи, навчання, поспілкуватися з друзями тощо.

Кожного року виходить велика кількість ігор від відомих видавців: Ubisoft, Electronic Arts (EA), Activision-Blizzard-King (ABK), Sony на різні платформи, а саме: домашній комп’ютер, мобільний телефон, ігрова приставка від Sony, Microsoft або Nintendo. Ці студії випускають ігри різноманітного типу:

1. Сюжетно-орієнтовані ігри - серія Far Cry, Watch Dogs.
2. Battle Royale (Королівська битва) – Call of Duty Warzone, Apex Legends, Battlefield 2042.
3. Стратегії – серія Anno чи Sid Meier's Civilization.

Але треба враховувати, що ринок ігрової індустрії не можуть зайняти лише великі компанії. На ньому присутні і більш дрібні студії, які розробляють свій продукт: аркади, головоломки, різнопланові стратегії тощо. Сам ринок продажі комп’ютерних ігор є досить простим для входу в бізнес. Для того, щоб почати продавати гру, її необхідно розмістити на сервісах онлайн дистрибуції, таких як Steam чи Epic Games Store. Навпаки, розміщення гри в Xbox Microsoft Store чи Sony Playstation Store є більш складним процесом, оскільки вимоги дистриб’ютора не кожна студія в змозі виконати.

У зв’язку з постійним розширенням асортименту ігор, все більше зростає попит на якісні сюжетно орієнтовані ігри. Більшість студій створюють розробники, які мають досвід у розробці ігор для відомих видавців.

На відміну від популярних онлайн ігор, перевага сюжетно орієнтованих ігор в тому, що вони не вимагають конкретних технічних вимог від студії, а саме: наявність значних серверних потужностей, які підтримують одночасний доступ сотень або тисяч гравців одночасно, баз даних, складної мережевої архітектури тощо. Прикладом може бути гра Counter-Strike: Global Offensive. На момент написання роботи, пікове навантаження на сервери становили 1,255,606 гравців на день [1], що є досить високою технічною вимогою для підтримки видавцем гри. Сюжетна гра запускається лише на обладнанні гравця і не вимагає додаткових ресурсів від студії розробника.

Створення сюжету гри є складним процесом, який вимагає взаємодії великої кількості фахівців: дизайнерів геймплею, художників, програмістів тощо. При розробці будь-якої гри особливу увагу приділяють вибору рушія, оскільки від нього залежать не лише візуальні спроможності, а й особливості створення геймплею, швидкість роботи, оптимізація коду, доступність та ціна. Саме за цими критеріями Unreal Engine 5 є найзручнішим варіантом:

1. Рушій поставляється з відкритим кодом, що дає можливість розробникам модифікувати рушій, згідно вимог проєкту.
2. Рушій можна безкоштовно завантажити через платформу Epic Games Launcher, що дає можливість будь-кому, хто хоче працювати в сфері розробки ігор, почати вивчати рушій. І це збільшує кількість кандидатів та кількість вакансій на ринку праці.
3. Рушій – достатньо стабільний та оптимізований для роботи не лише для програмістів, а і для дизайнерів.
4. Рушій гнучко підтримує різноманітні плагіни.
5. Вигідна плата за використання рушія в комерційних цілях.

Тому, враховуючи вище викладеному, нами поставлена наступна **мета** дипломної роботи:

проаналізувати ринок дистрибуції комп’ютерних ігор, дослідити особливості рушія та механізми створення сюжетно-орієнтованої гри на ігровому рушії Unreal Engine 5.

Для вирішення даної мети нами сформульовані наступні **завдання:**

1. Опрацювати інформаційні джерела з теми дипломної роботи
2. Визначити специфіку розробки ігор в ігровій індустрії
3. Опрацювати особливості роботи з рушієм Unreal Engine 5
4. Спроектувати систему для створення сюжету з урахуванням особливостей рушія Unreal Engine 5.
5. Окреслити можливі способи покращення написаного модуля.

**Об'єктом нашого дослідження** є сюжетно орієнтовані комп’ютерні ігри.

**Предметом** – властивості та застосування рушія Unreal Engine 5 для написання системи, яка відповідає за створення сюжету в грі.

**Наукова новизна одержаних результатів**. Результати, які були одержані в дипломній роботі, дають основу розуміння того, як побудований сюжет в комп’ютерній грі та написання інструментів для створення сюжету.

**Практичне значення одержаних результатів.** Практичне значення одержаних результатів полягає в створенні модуля, який забезпечує створення сюжетно-орієнтованої гри. Цей модуль можна перенести до будь якого проєкта з метою створення сюжету.

**Особистий внесок дослідника**. Дипломна робота розглядає особливості рушія Unreal Engine 5: скриптову мову програмування рушія Blueprints, систему рефлексії та Garbage Collector рушія. У практичній частині дипломної роботи був наведений приклад розробки системи для створення сюжету гри.

**Структура роботи**. Дипломна робота складається зі вступу, теоретичного розділу, та експериментальної частини розділів, висновків і списку використаних джерел, до складу якого входить 45 найменувань. Кваліфікаційна робота налічує 81 сторінок друкованого тексту, проілюстрована 6 рисунками.

**РОЗДІЛ 1. ЗАГАЛЬНА ХАРАКТЕРИСТИКА СФЕРИ РОЗРОБКИ КОМП’ЮТЕРНИХ ІГОР**

**1.1. Еволюція сюжетно-орієнтованих ігор**

Перші комп’ютери з’явились під час Другої Світової Війни та, в першу чергу, створювались під військові проекти, а саме: криптоаналіз шифрувальної машини «Enigma»; для розрахунку траєкторії пуску ракет тощо. Але з часом, комп’ютери стали простими у виробництві та набули широкого поширення серед середнього та малого бізнесу з метою виконання різноманітних обчислень, які раніше виконували люди (наприклад бухгалтер), та серед наукової спільноти, які використовували комп’ютери у своїх дослідженнях [2].

З широким поширенням комп’ютерів, з’явилося велика кількість ентузіастів, які намагалися поширити застосування ПК у сфері розваг. Першою грою, яка була створена для ПК, є Spacewar! [3]. Вона являла собою просту симуляцію космічного шутеру та мала великий успіх, серед користувачів комп’ютерів.

Дана гра не мала сюжету, але довела, що комп’ютер можна використовувати не лише для математичних розрахунків або наукових симуляцій. Наступним кроком розвитку індустрії – створення текстових інтерактивних ігор, в яких гравець залежно від отриманого тексту писав відповідь на запитання. Залежно від того, як він відповідав, сюжет змінювався та пропонував різні варіанти розвитку події. Прикладом подібної гри можна вважати Colossal Cave Adventure [4] (рис. 1.1).

На початку зародження ери комп’ютерів, вони мали слабі процесори та дуже мало оперативної пам’яті, що ускладнювало створення будь-якої гри, та не існувало графічного прискорювача, оскільки не було потреб у використанні графічного інтерфейсу. Більшість комп’ютерів взаємодіяло з користувачем через команди середовища.

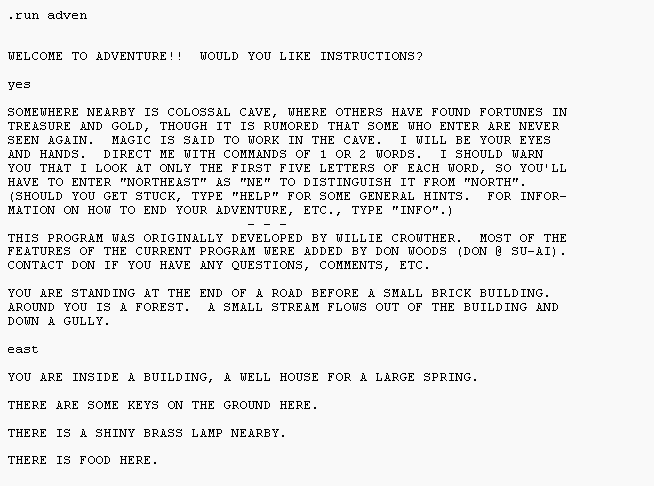


Рис. 1.1. Приклад взаємодії гравця з грою Colossal Cave Adventure

Наступним кроком розвитку ігор стала платформа Atari – портативний комп’ютер на базі мікропроцесора MOS6507 з тактовою частотою 1,86 МГц – спрощеної версії MOS6502 (використано в перших комп’ютерах Apple), який створено спеціально для ігор. Платформа дозволяла відобразити ігрове поле, спрайти (спрайт – це 2D зображення) гравців та інших об’єктів у грі використовуючи 210 доступних кольорів [5]. Як носій інформації, він використовував картриджі для завантаження ігор в пам’ять та запуск них. Для цієї платформи розроблено ігри за відомими фільмами, коміксами, книгами тощо.

Наступним кроком у розвитку ігор став вихід Wolfenstein 3D (1992) на рушії Wolfenstein 3D Engine. Цей рушій вперше дозволив «відмальовувати» псевдо 3D-графіку. Дана гра розширила можливість руху в ігровому світі (за координатною сіткою XY), але мала обмеження – відсутній рух по осі Z – тобто вгору та вниз.

Наступним технологічним успіхом став Doom (1993) на покращеному рушії на основі Wolfenstein 3D Еngine, який дозволяв рух по осях XYZ, хоча все ще були присутні обмеження щодо рендеру зображення на екран монітору. Гра мала сюжет, який був описаний на статичних кадрах або в описі до місії.

Наступною ітерацією рушія Wolfenstein 3D Еngine став Quake. На момент виходу гри, рушій підтримував повноцінне 3D-зображення, що дозволило «відмальовувати» низько-полігональні моделі, а не спрайти, які зображували лише перпендикулярно для персонажу. Ще одним нововведенням стала підтримка мережевої гри: як по локальній мережі, так і через Інтернет (різницю між Listen та Dedicated server буде описано у наступній главі).

Рушій гри Quake став основою для багатьох ігор, який заклав ідеї, що стали базисом у розробці ігор великої кількості компаній. Одне з розгалужень Quake Engine – рушій Source, на якому створено такі ігри як: CS:GO, Half-Life 2, Apex Legends тощо. На базі версії рушія Quake 3 створено idTech 6 – на якому випущено велику кількість ігор студії Bethesda – Wolfenstein, Doom 2016, Doom Eternal (наступна ітерація рушія - idTech7), Dishonored, та рушія, на якому створено серію ігор Call of Duty [6].

У 1998 році вийшла Unreal на новому рушії Unreal Engine 1, який розроблено Тімом Свіні. Unreal запровадив ідею мультитекстурування у процес створення ігор. Завдяки цій технології вдалось реалізувати різні накладення текстур на об'єкти, а саме: різні ефекти, типу лави чи води, які в основі мали рухому поверхню. Вперше в ігровій індустрії реалізовано Skybox [7] – об’єкт, що створював ілюзію неба, горизонту та відкритого простору. На момент виходу Unreal була найбільш технологічною серед інших ігор – мала мультиплеєр з підтримкою мережі, мала сюжет, який пояснював історію цього світу, гравця, його мотивацію тощо.

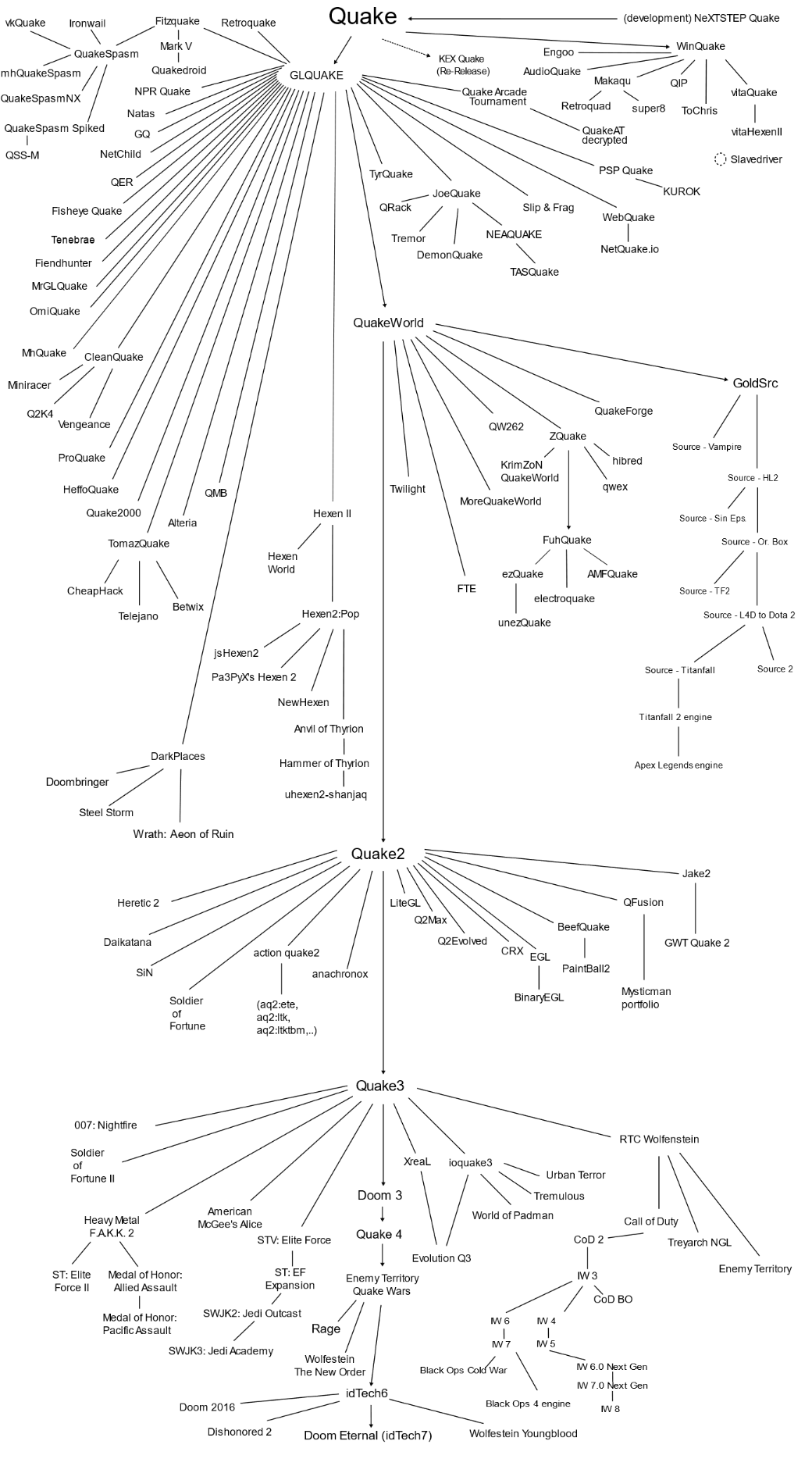


Рис 1.2. Дерево ігор, створених на основі рушія Quake [6]

Отже, враховуючи вище викладене, можна констатувати, що це стало поштовхом до розвитку сюжетно-орієнтованих ігор, та заклало головні ідеї в створенні та подачі сюжету для гравця. Ці ідеї дозволили виростили нове покоління, які створює свої ігри зі своїми підходами.

**1.2 Аналіз ринкових тенденцій в розробці ігор**

Згідно GamingGorilla [8] рейтинг двадцятки найбільш популярних ігор у 2022–2023 роках є:

1. Call of Duty: Warzone (Modern Warfare)
2. Minecraft
3. Fortnite
4. Grand Theft Auto V
5. Tom Clancy’s Rainbow Six Siege
6. Super Smash Bros: Ultimate
7. Red Dead Redemption II
8. Overwatch
9. Rocket League
10. Roblox
11. League of Legends
12. PlayerUnknown’s Battlegrounds
13. Counter-Strike: Global Offensive
14. Call of Duty: Black Ops III
15. Super Mario Odyssey
16. The legend of Zelda: Breath Of The Wild
17. Spider-Man
18. Call of Duty: Black Ops II
19. The Elder Scrolls V: Skyrim
20. Super Smash Bros: For Wii

У цьому списку три позиції займає серія Call of Duty, яка має сюжетну складову, але основною частиною залишається мультиплеєр. У 2020 році з’явилась версія Call of Duty Warzone [9], яка є безкоштовною та підтримує мультиплеєр 150-200 гравців на одній карті.

Також, даний рейтинг містить кілька онлайн ігор, які не мають сюжетної компанії: Counter-Strike: Global Offensive, PlayerUnknown’s Battlegrounds, Fortnite, Tom Clancy’s Rainbow Six Siege, League of Legends та ігри, які мають сюжетну компанію та мультиплеєр: Overwatch, Red Dead Redemption II, GTA 5, Minecraft.

Щодо онлайн ігор, то слід відмітити їх складність у розробці, оскільки крім ігрового клієнту, вони додатково вимагають Listen або Dedicated сервер [10]. Listen server – тип сервера, який працює разом з клієнтом в одному процесі. Такі рішення характерні для ігор, що підтримують ігри по локальній мережі (прикладом може бути CS:GO) або для кооперативних ігор (The Dark Pictures Anthology від Supermassive Games). Dedicated server, навпаки, працює незалежно від клієнту і зазвичай розміщений на обладнанні компанії-розробника ігри.

Завдання сервера – контроль ігрового процесу різних гравців, забезпечити зручний процес проводження часу в грі, синхронізації даних між діями різних гравців та зменшення кількості людей, які намагаються обійти установленні умови розробника (чітери). Згідно статистичних даних [11] найбільш популярними іграми, де гравці використовують чіти (сheat) є: Fortnite, Overwatch, CS:GO тощо. Популярність гри створює попит на придбання ПЗ з метою її злому, а завдання розробника – знайти інструменти боротьби з гравцями, що використовують чіти. Відповідно, це призводить до додаткових витрат на створення програми-античіта (anti-cheat software), завдання якої ускладнити можливість створення чіта, але хакери, продовжують шукати нові вразливості в коді клієнта з метою подальшого написання чіта.

Гра, коли до новачків додають досвідчених гравців є нечесною та незбалансованою. Це ставить вимогу, щодо врахування можливостей і досвіду гравців на основі попередніх матчів та дозволяє підібрати команду, в якій гравці можуть грати на рівних. Підбір гравців за його навичками є частиною системи-матчмейкінгу (matchmaking) у грі. Це призводить до додаткових витрат на розробку системи матчмейкінгу (matchmaking server) та підтримки бази даних, яка містить дані для подальшого аналізу в процесі пошуку команди.

Сюжетні ігри не вимагають додаткової розробки коду античіту, серверу, системи матчмейкінгу та створення інфраструктури для підтримки серверів і інших компонентів онлайн гри. Серед таких сюжетних ігор, що не мають онлайн складової, можна відмітити Super Mario Odyssey, The Elder Scrolls V: Skyrim, Spider-Man, The Legend of Zelda: Breath Of The Wild. Ці ігри мають високий рейтинг популярності серед гравців.

Таким чином, враховуючи вище викладене, можна констатувати, що розробка онлайн гри складний процес, який вимагає не лише суттєвих фінансових витрат, а й підбору висококваліфікованого персоналу.

**1.2.1 Дослідження платформ дистрибуцій комп’ютерних ігор**

Одним із видів доставки легального цифрового контенту (дистрибуція, цифрова дистрибуція) є його поширення ресурсами Інтернет або на фізичних носіях (диски, флеш-карти тощо) [12]. Серед платформ дистрибуції ігор, які мають свою специфіку розробки та поширення та використовують онлайн магазини або диски можна відмітити [13]:

1. PC (Windows або Linux)
2. Sony PlayStation 4/5
3. Microsoft Xbox Series S/X, Xbox One
4. Nintendo Switch
5. Мобільні телефони

Платформа PC є найбільш різноманітною та складною для розробки, оскільки необхідно враховувати конфігурації ігрового комп’ютера, які варіюють у широких межах – від дешевих до дорогих. Згідно цього, розробник повинен таким чином оптимізувати гру, враховуючи характеристики ПК, щоб збільшити кількість потенційних покупців з бюджетними конфігураціями ПК та додати функціонал, який дозволить власникам потужних ПК насолоджуватися грою.

Прикладом такого функціоналу є технологія Nvidia Ray Tracing, яка дозволяє створювати реалістичні світи, додаючи відображення об’єктів на матеріали, які здатні віддзеркалювати світло. Наприклад [14], де показано застосування технології Ray tracing у грі Control 2019 (рис. 1.3).



Рис. 1.3. Застосування технології Real Time Ray Tracing у грі Control 2019

Більшість ігор створюють під операційну систему Windows 10/11. ОС Linux не є поширеною серед гравців, тому для неї ігри не розроблять, але портують за допомогою емуляторів:

* wine – для емуляції WinAPI на Linux
* Різноманітні бібліотеки, які емулюють графічні інтерфейси DirectX11/12, що є пропрієтарними технологіями Windows, використовуючи графічний інтерфейс Vulkan, який поширений в операційній системі Linux

Також до ринку ПК можна додати портативну консоль Steam Deck, яка працює на ОС Linux, на яку портують ігри з платформи Windows, але консоль дозволяє встановити на неї ОС Windows.

Наступною платформою, яку необхідно згадати – це Sony PlayStation, які побудовані на Unix-подібній системі Orbis OS [15], що є пропрієтарною технологією. Консоль має свою архітектуру процесора та графічного інтерфейсу. Відмінністю консолей від PC – кожна ревізія консолі має однакову конфігурацію. Це спрощує розробникам роботу, оскільки гра розробляється та оптимізується під конкретні технічні характеристики та обмеження платформи. Компанія Sony пропонує дві моделі консолей: PlayStation 4, PlayStation 5. Остання модель є більш потужною версією попередника. Також ця модель має деякі переваги, а саме підтримку різних режимів роботи (якість зображення, пріоритет FPS) та новий маніпулятор PlayStation DualSence, який має в собі покращений функціонал.

Основним конкурентом консолей від Sony – є платформа від Microsoft Xbox Series на базі ОС Windows [16]. Тобто платформа має підтримку WinAPI та використання графічного API DirectX11/12, що дозволяє спростити розробку ігор для PC та консолі.

Серед інших платформ можна відмітити Nintendo Switch та мобільні гаджети (смартфони, планшети), які ніяк не конкурують з основними платформами від Sony, Microsoft та PC. Вони технічно обмежені і не підтримують запуск ігор масштабу ААА, але мають свій ринок і свої способи поширення ігор та орієнтуються на різних користувачів:

* Nintendo Switch – кооперативні сімейні ігри або прості сюжетні (The Legend of Zelda: Breath of the Wild);
* мобільні гаджети – більше всього фокусуються на зручності та створюють можливості грати будь-де у різноманітні ігри, які орієнтовані на будь-яку аудиторію (дорослі, діти, жінки, чоловіки).

Розповсюдження цифрового контенту, для різних платформ, відбувається за допомогою онлайн магазинів та роздрібної торгівлі. Кожна платформа має один магазин або велику кількість онлайн магазинів. Платформою з найбільшою кількістю точок розповсюдження цифрового контенту є PC. Наприклад:

1. Steam – дозволяє опубліковувати різноманітні ігри для будь яких розробників, незалежно від їх фінансового статусу;
2. Origin – магазин розроблено Electronic Arts, який реалізує ігри від розробника ЕА або студій, які підпорядковані EA.
3. Ubisoft Connect – магазин розроблено Ubisoft, який, подібно Origin, публікує ігри власної розробки;
4. Epic Games Store – схожий на Steam, тобто, пропонує розробникам публікувати власні ігри;
5. Battlenet – реалізує розробки студії Activision-Blizzard.

Наявність великого списку магазинів, змушує потенційних покупців ігор встановлювати велику кількість магазинів на PC, оскільки деякі ігри можуть продаватися ексклюзивно в певному магазині.

Платформи Sony Playstation та Microsoft Xbox є закритими платформами для створення нових додатків, тому вони мають лише по одному офіційному магазину в якому можна купити ігри: PlayStation Store та Microsoft Store відповідно. Платформи підтримують можливість купівлі дискової версії гри, яка не вимагає реєстрації в онлайн магазинах та не вимагає завантаження файлів гри з мережі. Платформа Nintendo Switch є закритою, дуже схожа на консолі від Microsoft та Sony та має власний магазин Nintendo eShop.

Мобільні додатки та ігри можна завантажити в магазинах, залежно від платформи: Google PlayMarket (Android) і Apple App Store (IOS). В них найбільше поширені саме безкоштовні додатки. Згідно [17] у магазинах Google PlayMarket і Apple App Store відсоток безкоштовних додатків становить 96.96% та 94.58% відповідно. Мобільні додатки орієнтуються на простоту виробництва гри і технічними обмеженнями платформи та мають зовсім інші механізми отримання прибутку [18] від тих, що існують на платформі PC та консолей.

**1.2.2 Загальний огляд процесу розробки ігри**

Процес розробки гри є складним процесом, який залучає велику кількість фахівців, які мають свою роль та відповідальність у її створенні. У першу чергу, гра повинна розроблятися на рушії, який підтримує ігри конкретного жанру. Тобто розробляти гру про космос на рушії, який до цього використовувався для розробки симуляторів гонок, є поганою ідеєю, оскільки рушій доведеться модифікувати для того, щоб задовольнити вимоги проекту.

Керівником, який відповідає за проект, є креативний директор, який:

1. Визначає напрямки розвитку гри, контролює процес розробки та виконання поставлених задач згідно визначених термінів.
2. Визначає кількість фахівців, які залучені у розробку в певних сферах: Lead Gameplay Developer, Lead Sound Designer, Lead Render Developer тощо.

Директори підрозділів керують програмістами та планують подальшу розробку функціоналу поставленого креативним директором. Співробітників компанії, можна класифікувати таким чином:

1. Програмісти (Engine, Gameplay, Render, Tools developer). Їх завдання – розробляти рушій, логіку гри, інструментарій для створення гри тощо.
2. Дизайнери (Game, Level designer) – фахівці, які створюють ігрове середовище (world), документують ігрові механіки, ігрові системи тощо.
3. Програмісти ігрового коду та UI – фахівці, які працюють з документацією та створюють код, залежно від поставленого завдання, визначеною скриптовою мовою програмування.

Аналіз класифікації співробітників, дозволяє дійти висновку, що рушій повинен підтримувати data driven programming approach. Тобто ідею, де гру можна створити на основі даних та дати можливість будь-кому змінити параметри без перекомпіляції проекту. Цей підхід розширює можливість взаємодії дизайнерів та програмістів і прибирає необхідність модифікації або параметрів ігрової логіки (наприклад швидкість бігу персонажа у грі), або візуальної частини гри.

Необхідною вимогою для спрощення роботи з рушієм є присутність скриптової мови програмування, яка повинна відповідати таким умовам:

1. Простота у навчанні та роботі з нею.
2. Простота інтеграції скриптової мови з нативним кодом (С++).
3. Швидкість виконання коду скриптової мови.
4. Присутність GC (Garbage Collector).
5. Простота змін та завантаження скриптів під час виконання програми.

Прикладом скриптових мов, які задовольняють ці вимоги, є Lua, Pawn або Blueprints. Також студії можуть створювати нові мови програмування, які задовольняють вимогам проекту. Наприклад, мова програмування WitcherScript, яка використовується в серії ігор The Witcher, від студії CDProject RED.

Скриптова мова програмування створена для розподілу сфери відповідальності на фахівців з різною кваліфікацією. Відповідно, використовуючи скриптову мову програмування, програмісти можуть створювати критично важливі компоненти рушія та геймплею, а дизайнери чи програмісти ігрової логіки – взаємодії елементів. Наприклад, програміст створює Artificial Intelligence систему, а дизайнер – взаємодію бота з різними ігровими подіями (наприклад, систему патрулювання та реакцію на події, під час патрулювання; систему діалогів, залежно від стану гри тощо).

Також скриптова мова допомагає спростити модифікацію гри під час тестування. Якщо дизайнер знаходить некоректну поведінку його коду чи коду іншого програміста, він може відразу модифікувати скриптовий код та завантажити виправлену версії відразу в гру без необхідності перекомпіляції всього проекту.

Залежно від реалізації рушія скриптова мова програмування також може відігравати як спосіб визначення та оновлення даних в рефлексії рушія. Подібний метод редагування параметрів у системі рефлексії рушія, реалізовано в рушії компанії 4A Games [19].

**1.3 Обґрунтування вибору рушія Unreal Engine 5**

У сучасному світі ігрова індустрія розвивається досить швидко: постійно створюють нові проекти, для задоволення потреб гравців, а гравці, відповідно, очікують від розробників щось нове, яке підходить до їх смаків. Однією з ключових складових успішного проекту є вибір рушія, на якому буде розроблятися гра.

Unreal Engine 5 - популярний рушій для розробки ігор, який створено компанією Epic Games для своїх проектів. Першу гру на цьому рушії було створено у 1998 році і за ці роки він еволюціонував від пропрієтарного рушія до популярного рушія, який використовують розробники по всьому світі [20]. На даний момент рушій Unreal Engine 5 використовують для створення ігор під різні платформи: консолі, портативні консолі Nintendo Switch, ПК та мобільні пристрої.

Окрім ігор, рушій задіяно в кіновиробництві для розрахунку віртуальних просторів світу, які проектуються на великі 2D екрани. Прикладом є серіал, який створено з використанням такого підходу – Мандалорець (The Mandalorian) [21].

Студією-розробником Epic Games, на базі Unreal Engine 4, було створено проект рушія, а саме багатокористувацька гра Fortnite, яку з часом портовано на Unreal Engine 5. Ця гра дає поштовх для розвитку рушія в результаті якого, всі нові технології розвитку останнього, спочатку тестуються на проекті Fortnite, а потім переходять у відкритий доступ для всіх розробників.

**1.3.1. Переваги та недоліки Unreal Engine 5 у порівнянні з іншими ігровими рушіями**

Однією з основних переваг Unreal Engine у порівнянні 5 з іншими рушіями є його високоякісна графіка та фізика. В версії Unreal Engine 5 рушій оновив фізичний рушій на Chaos замінивши Nvidia Physics. Рушій має потужні інструменти для роботи з освітленням, водою, тіньовими ефектами та іншими графічними ефектами, що дозволяє створювати дуже реалістичні ігрові світи. Крім того, Unreal Engine 5 має дуже точну фізичну модель, що дозволяє створювати реалістичні фізичні ефекти, наприклад, відтворювати рух твердих тіл, розбиття предметів на частинки та інші ефекти. Прикладом подібного досягнення фізики та ефектів є Days Gone, де розробникам вдалося симулювали організовану поведінку ігрових персонажів на фізично застарілій платформі PlayStation 4.

Також до переваг рушія відносять простий інтерфейс та зручність для розробників, що дозволяє швидко створювати гру з мінімальним зусиллям. Рушій об’єднує більшість функціоналу в одній програмі, що спрощує роботу для розробників, але накладає ряд обмежень в розробці:

1. Складність у модифікації та написанні ігрової логіки, оскільки код редактора та самої гри будуть перетинатися між собою.
2. Неможливість додавати новий функціонал редактору іншими мовами програмування, що додає окремих вимог до кваліфікації програміста.

На відміну від Unreal Engine, пропрієтарні рушії побудовані на ідеї розподілу відповідальності: для кожної сфери розробки є свій інструмент. Наприклад, редактор ігрового простору, редактор UI, редактор рефлексії та сам рушій, який використовує дані отримані з цих редакторів з метою відображення ігрового простору.

Відкритість рушія дозволив створити базу знань: статті, практичні відеоматеріали тощо. І головне, рушій має підтримку спільноти, яка дозволяє розробникам знайти відповіді на свої запитання та вирішити будь-які проблеми.

Крім переваг, рушій має і недоліки, які можуть бути критичними для розробки ігор. Архітектуру рушія та його здатність до розширення побудовано на ідеї наслідування та перевантаженні віртуальних функцій. Подібна архітектура може впливати на завантаження оперативної пам’яті. Рушій Unity використовує архітектуру ECS (Entity-Component-System) [22], яка дозволяє розділити функціонал на невеличкі легкі компоненти, системи та сутності.

Наступною проблемою є обмежена багато-поточність рушія. Рушій виконує обчислення фізики, анімацій у багато-поточному режимі, але сама ігрова логіка виконується послідовно у головному потоці. Unreal Engine не має технічної можливості для розділення послідовного виконання головного ігрового потоку на одночасне виконання невеликих завдань на додаткових потоках. Разом з архітектурою рушія це додає складності в оптимізації ігри під такі ігрові платформи такі як мобільні гаджети, Nintendo Switch та PlayStation 4.

Критична проблема рушія пов’язана з повільним виконанням команд скриптової мови програмування Blueprints. Це вимагає перенесення частини коду в нативну частину гри (за умови використання С++) та залишити у скриптовій мові лише візуальні зміни для об’єктів або UI. Ця проблема збільшує кількість залучених фахівців та додає обмеження для дизайнерів, які могли створювати ігрову логіку відразу в Blueprints.

**1.3.2 Аналіз популярності Unreal Engine 5 та його застосування**

Нову версію рушія Unreal Engine 5 випущено у квітні 2021 року. Оскільки версію рушія випущено кілька років тому, то він ще не має великої кількості нових проєктів, які розроблено на цьому рушії. Тому основні проєкти було створено на попередній версії рушія Unreal Engine 4. На даний момент існують проєкти, які почали розробляти на UE4 та з часом перенесено на UE5. Прикладом може бути S.T.A.L.K.E.R. 2: Heart of Chornobyl.

Перехід до наступної версії рушія – складний процес, оскільки рушій має суттєві зміни у функціоналі та інструментах. Це вимагає переносу контенту на нову версію та виправлення проблем, які з’являються через зміни в коді.

Нижче наведено список ігор, які розроблено на версіях Unreal Engine 4 та 5 [23].

1. Fortnite
2. Hellblade: Senua’s Sacrifice
3. Street Fighter 5
4. Star Wars Jedi: Fallen Order
5. Sea of Thieves
6. Observer
7. Borderlands 3
8. Little Nightmares 1/2
9. Gears 5
10. Final Fantasy 7 Remake

Аналіз списку ігор свідчить, що цей рушій використовують різноманітні студії, які не мають та не створюють власного рушія. Також слід відмітити, що рушій Unreal Engine підтримує різноманітні жанри ігор: від симуляторів до ігор у відкритому світі, що робить його досить популярним рушієм, на якому просто та вигідно розробляти різноманітні ігри.

**РОЗДІЛ 2. РОЗРОБКА СИСТЕМИ ДЛЯ СТВОРЕННЯ СЮЖЕТНО-ОРІЄНТОВАНИХ ІГОР НА МОВІ ПРОГРАМУВАННЯ С++ ТА РУШІЯ UNREAL ENGINE 5.**

**2.1 Огляд реалізації Garbage Collection рушія та його системи рефлексії**

Ручне керування пам’яттю з’явилось зі створенням мови програмування С у 1970 році. На початку ери програмування програміст повинен був самостійно керувати пам’яттю: скільки створює, скільки видаляє. Такий підхід був проблемним, оскільки програміст міг робити помилки під час написання коду.

Починаючи з останнього десятиріччя ХХ століття, потужності процесора та об’єму оперативної пам’яті стало достатньо для розробки мов програмування, які не вимагали від роботи з пам’яттю – Python або Java – та дозволяли автоматизувати процес створення та видалення пам’яті. Б'ярн Страуструп, при створенні С++, вирішив не додавати в стандарт ідею автоматичного видалення пам’яті та залишити керування пам’яттю на розсуд програміста. Присутність ООП та ідіоми RAII [24] дозволило створювати нові способи напівавтоматичного керування пам’яттю – smart pointers (розумні вказівники) [25]. Згідно стандарту С++11 розумні вказівники додано до бібліотеки STL (Standart Template Library), реалізація якої є основною вимогою для компілятору.

Слід відмітити, що розумні вказівники не завжди є порятунком. Існує багато способів створити збій програми використовуючи вказівники. Нижче наведено приклад спільного використання сирого вказівника, яке може призвести до помилки виконання (див. додаток А).

Цей код може завершиться аварійно (деякі компілятори можуть мати модифікований оператор free, який проводить перевірку на подвійне видалення пам’яті, та не спробує видалити об’єкт другий раз, але писати код, який залежить від компілятора, є поганою практикою), так як два розумних вказівника будуть вказувати на одну ту саму пам’ять та вважати, що кожен з них володіє цим об’єктом винятково. Коли виконання коду вийде з функції, буде викликаний деструктор для обох розумних вказівників. Перший деструктор видалить об’єкт в купі, а другий розумний вказівник спробує видалити вже видалений об’єкт, що призводить до критичної помилки. Отже, розумні вказівники не є гарантовано безпечним способом керувати пам’яттю.

Наступним кроком покращення керування пам’яттю були спроби створення Garbage Collection (прибиральників сміття), які задовольняють вимоги проєкту. Один з таких реалізований в рушії Unreal Engine 5.

Garbage Collection є механізмом автоматичного керування об’єктів у пам'яті, який автоматично видаляє об'єкти, що не використовуються в програмі, або були позначені для видалення в коді. Це дозволяє зосередитися на розробці програм, не звертаючи уваги на ручне керування над об’єктами в пам’яті, що може призвести до збоїв програми або до витоку пам’яті.

Прибиральник сміття має свою специфіку роботи, яку враховують під час написання коду та створення архітектури гри. Основою будь-якого «живого» об’єкту в пам’яті є кореневий об’єкт. Будь який об’єкт, який походить від UObject може бути кореневим і мати свій життєвий цикл до моменту видалення програмістом. Але робити багато кореневих об’єктів не є гарною практикою в написанні коду, тому це зроблено розробниками рушія. Він має свої кореневі об’єкти, до яких прив’язуються інші об’єкти: UWorld, UGameInstance [26]. Також рушій дозволяє створювати об’єкти, час життя яких контролюється сторонніми бібліотеками. Для цього необхідно віднаслідуватися від структури FGCObject.

В основі GC покладено алгоритм «Mark and Sweep» [27]. Алгоритм розпочинає з проходження по всім об’єктам і якщо цей об’єкт, або об’єкт який може за ланцюжком залежності (батько або дитина), дістатися до кореневого об’єкту, буде вважатися «живим». Як кільки об’єкт, чи його батько, або нащадок втрачають посилання до кореневого об’єкту – такий об’єкт буде позначений міткою «Pending To Kill» та відповідно буде видалений з наступною ітерацією прибиральника сміття. Стандартне значення часового проміжку між ітераціям є приблизно п’ять секунд, але це може бути налаштовано для кожного проекту окремо. Тому будь який об’єкт, який наслідується від UObject, необхідно перевіряти на його валідність. Функція IsValid перевіряє чи об’єкт не є nullptr, та не має флагів RF\_InternalPendingKill або RF\_InternalGarbage.

Ще одна з проблем роботи з пам’яттю це боротьба циклічними посиланнями. Ця проблема виникає коли один об’єкт зберігає в собі посилання на інший об’єкт, та відповідно інший об’єкт тримає посилання на цей об’єкт. Такі об’єкти не можуть бути видалені автоматично так як вони мають по одному посиланню яке тримає їх «живими», що призводить до того, що об’єкти залишаються в пам’яті надовго, що може призвести до витоку пам’яті.

Прибиральник сміття вирішує цю проблему за допомогою рефлексії, яка є в рушії. За допомогою рефлексії Unreal Engine визначає «weak references», які не можуть використовувати посилання на кореневий об’єкт. Якщо об’єкт буде видаленим з пам’яті, всі weak references на цей об’єкт будуть занулені.

Рефлексія (англ. Reflection) - це механізм, який дозволяє програмі отримувати інформацію про типи об'єктів і їх властивості за час виконання [28]. Рефлексія дозволяє програмі аналізувати свій власний код і отримувати інформацію про об'єкти, що створюються під час виконання програми та змінювати властивості та методи під час виконання.

Рушій Unreal Engine дозволяє змінювати змінні, які були створені в С++ коді, новими значеннями в Blueprints, або виконувати функції в скриптовій мові, які були створені в С++ коді. Для цього Unreal Engine 5 використовує систему реєстрації та збереження метаданих, які зберігають інформацію про всі класи та об'єкти в грі.

Для збереження даних рефлексії класу в Unreal Engine 5 використовуються об'єкти типу UClass, які визначають клас з функції та змінні, які доступні в рефлексії рушія. Доступ до UClass об’єкта можна отримати викликавши статичну функцію UObject::StaticClass().

Обов’язковою частиною UClass є посилання на Class Default Object (CDO) . Це первинна версія будь якого класу. Він зберігає базові налаштування рефлексії для кожного об’єкту. CDO використовується для швидкого створення об’єкта в світі та можливості отримати базові значення для кожної змінної в системі рефлексії.

Для створення рефлексії рушій використовує окрему програму Unreal Header Tool, яку запускає Visual Studio перед компіляцією проєкту. Ця програма створює додатковий код, який реєструє змінні та функції в системі рефлексії. Цей згенерований код програма зберігає в файли подібного формату SomeClassName.generated.h та SomeClassName.gen.cpp.

Для того, щоб Unreal Header Tool розумів що саме треба додавати до системи рефлексії, необхідно визначити клас або структуру:

1. Для позначення класу необхідно використовувати макрос UCLASS() та додати в тіло класу макрос GENERATED\_BODY()
2. Для позначення структури необхідно використовувати макрос USTRUCT() та додати в тіло класу макрос GENERATED\_BODY().

Незважаючи на те, що підходи схожі, є суттєва різниця між класом та структурою: клас, на відміну від структури, повинен бути віднаслідуваний від базового класу, який підтримує рефлексію. Базовий клас в рушії це UObject. Програма не накладає ніякі обмеження на використання класів, та структур без макросів UCLASS(), USTRUCT(), GENERATED\_BODY() або які не наслідуються від базового класу рушія, але в такому випадку ці об’єкти не будуть мати можливсті працювати з рефлексією [29].

Рекомендованими параметрами макроса UCLASS() є Abstract – визначає клас, який не може бути створеним без нащадка в скриптовій мові Blueprints. Також для об’єктів які віднаслідуються від UObject, але не є нащадком класу AActor теж необхідно додавати параметр BlueprintType та Blueprintable.

А відповідно рекомендованими параметрами макроса USTRUCT() є BlueprintType – визначає структуру, яку необхідно додати до системи рефлексії щоб цю її можна було надалі створити або визначити в Blueprints.

UActorComponent не вимагає використання параметра Abstract, так як компонент можна модифікувати через Blueprint батьківського класу AActor, до якого він прив’язаний. Приклади створення класу та структури, які будуть доступні в системі рефлексії Unreal Engine 5 (див. додаток Б).

Unreal Header Tool згенерує визначення функцій та даних, конструктори, деструктори, та ряд специфічних функцій, які будуть використовуватися в рефлексії. Ці визначення будуть потім підставлені препроцесором замість макроса GENERATED\_BODY().

**2.2 Аналіз способів взаємодії мови програмування С++ та внутрішньої мови програмування рушія – Blueprints**

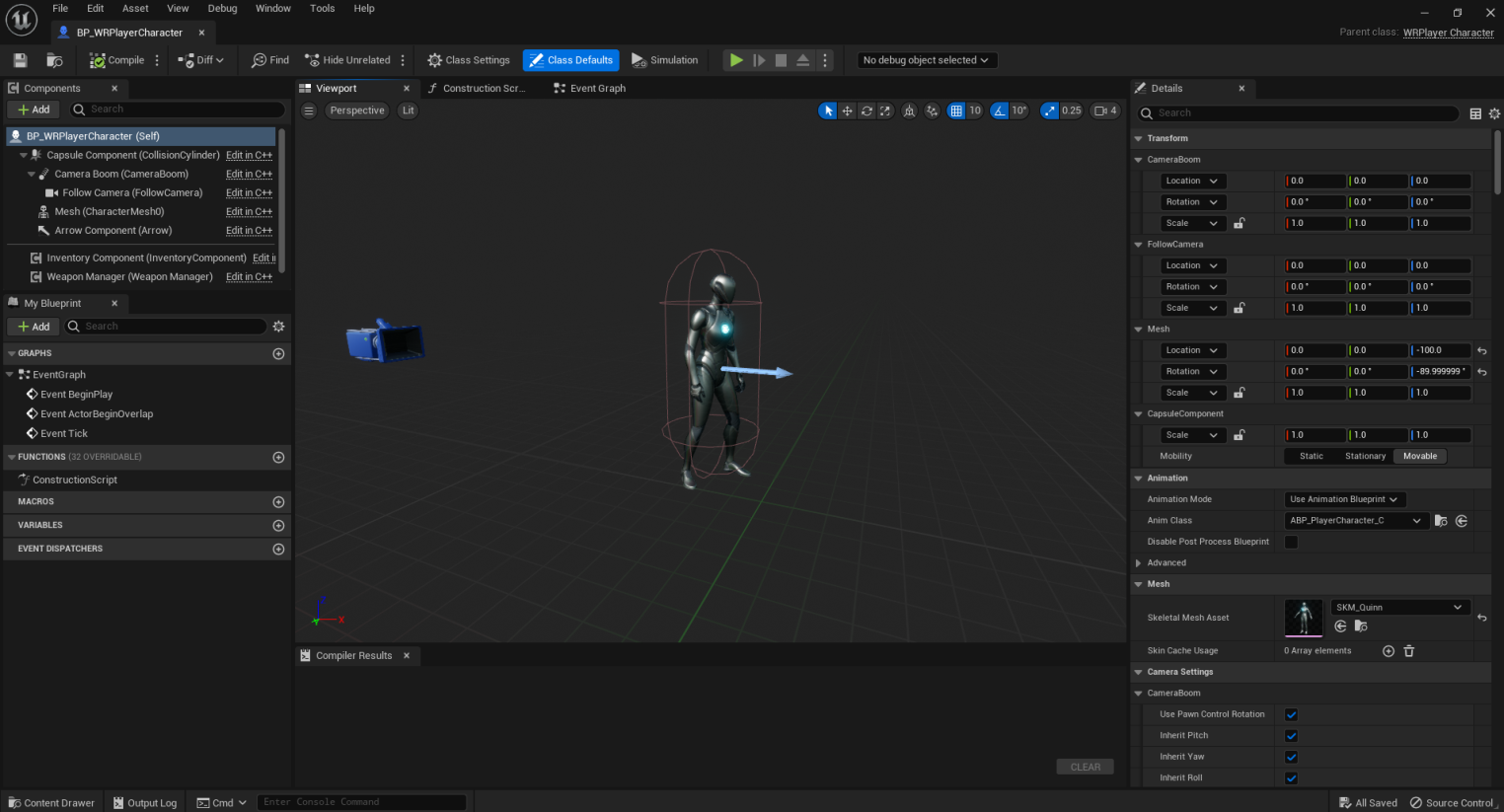


Рис. 2.1. Blueprints клас, який наслідується від класу рушія ACharacter.

(Рис 2.1) Зліва присутня панель, яка вказує на об’єкти, які зареєстровані в системі рефлексії та належать об’єкту, та функції й змінні які були створені саме в скриптовій мові програмування. Справа на панелі видно всі параметри рефлексії, які були зареєстровані в цьому об’єкті та доступні до модифікації програмістами або дизайнерами.

Для того, щоб можна було створювати параметри, які в результаті будуть зображені в редакторі, треба перед змінною додати макрос UPROPERTY().

UPROPERTY()

float SomeNumber = 12.f;

Макрос UPROPERTY може містити додаткові параметри для змінної. Нижче наведений список найбільш вживаних специфікаторів. З усім списком можна ознайомитися в документації рушія [30]:

1. BlueprintReadOnly – змінна доступна лише для читання в Blueprints.
2. BlueprintReadWrite – змінна може бути як і прочитаною, так і зміненою в скриптовій мові.
3. EditDefaultsOnly – зміна може бути змінена тільки в базовому Blueprints класі та тільки в редакторі.
4. EditAnywhere – змінна може бути створена як в базовому Blueprints класі в редакторі, так і в класі нащадку.
5. Category – опис категорії змінної. Змінні з однаковою категорією будуть фільтруватися в редакторі.

Окрім редагування змінних, Blueprints дозволяє викликати С++ функції використовуючи рефлексію. Додавання до рефлексії функцій є схожою операцією як з реєстрацією змінних в систему рефлексії. Для позначення функції яка буде видна в системі рефлексії, треба використовувати макрос UFUNCTION(). Макрос може мати такі специфікатори для С++ функції [31]:

1. BlueprintCallable – функція може бути викликана в блюпринтах
2. BlueprintImplementableEvent – функція яка визначена в C++ коді, та може викликатися, але її реалізація знаходиться в скриптовій мові.
3. BlueprintNativeEvent - функція яка визначена в С++ коді, та може викликатися, але реалізація визначена або в С++ коді, або в Blueprints.

Приклад використання рефлексії в коді (див. додаток В).

**2.3 Проектування та розробка системи створення сюжетно-орієнтованих ігор**

Для проектування системи необхідно враховувати в першу чергу архітектуру та особливість роботи Garbage Collector рушія. Механізм обов’язкової прив’язки до кореневого об’єкту змушує розділити всі об’єкти на два типи: об’єкти які прив’язані до світу, та існують поки світ завантажений в гру, та об’єкти які прив’язані до об’єкту, який буде існувати протягом роботи всієї гри. Відповідно, в першому випадку всі об’єкти будуть прив’язані до нащадку класа UWorld, а в другому – UGameInstance.

**2.3.1 Розробка інтерактивних об'єктів та виконавців в ігровому світі**

Об’єкти існують в світі з метою створення взаємодії між гравцем, та елементами сюжету в цьому світі. Будь яка гра може мати різну кількість ігрових світів, які відповідають за певну частину сюжету.

Гра може містити різні інтерактивні об’єкти такі як скрині, двері, NPC (Non playable character). Кожен з цих елементів може бути частиною сюжету. Що вимагає враховувати те, що завдання може бути виконаним різними способами:

1. Фізична подія. Наприклад колізія персонажа з дверима в світі. Результатом взаємодії є відкриття дверей та активація наступного завдання для гравця.
2. Подія з таймером. Наприклад гравець повинен виконати завдання за проміжок часу. Результатом є розблокування нового завдання чи отримання досягнення за виконане завдання.
3. Керування сюжету зі сторони. Будь який інший об’єкт може відмітити це завдання як виконане викликавши відповідну функцію. Наприклад, гравець підбирає нову зброю – клас інвентар реєструє виконане завдання.

Тому необхідно врахувати ці вимоги у реалізації виконавця завдань в ігровому світі, та його можливістю запрограмувати виконувати певні дії на початок виконання та на кінець виконання завдання.

Для того щоб забезпечити ці умови виконання завдання було створено клас UTaskExecutor, який наслідується від базового класу UObject. В цьому класі додані вище перераховані методи взаємодії, окрім того, до цього класу можна додавати нові типи взаємодії використовуючи зарезервовані значення в перечисленні (див. додаток Г).

Головною особливістю цього класу є те, що можна налаштовувати вимоги до виконання завдання: певний тип взаємодії, чи якийсь з кількох, чи всі обов’язкові до виконання.

У грі може бути велика кількість різноманітних завдань, які можуть належати одному актору. Для цього необхідно створити окремий ігровий клас-менеджер, який буде контролювати екземпляри класу UTaskExecutor.

Для класу-менеджера підходить клас-компонента, тобто нащадок від UActorComponent. Цей клас має назву UStorySystemComponent. Компонент може реєструвати та видаляти завдання залежно від прогресу сюжету гри [32].

Компонент – це об’єкт який не може існувати незалежно від об’єкта який може існувати в світі. В архітектурі рушія компонент може бути доданим до будь якого існуючого в світі актора: як до класу головного гравця, чи бота, чи до спеціального інтерактивного об’єкту.

Отже, виходячи з попереднього абзацу, окрім можливості додавати компонент до ігрового персонажу, також необхідно створити актора, який буде мати цей компонент, та буде надавати можливість взаємодіяти з виконавцем завдання. Для цього був створений клас AInteractableActor, який додає можливість обробляти різноманітні фізичні події (наприклад розрахунок колізії будь якого об’єкту з колізією в формі куба. Створення колізії в форі куба забезпечує клас UCollisionBox), та інші програмовані події.

Створення подій на основі колізій притаманно для завдань, які базуються на просторовому русі. Наприклад, дістатися до точки А, потім до точки Б і тому подібне. Цей актор має додатковий функціонал розрахунку складної колізії: тобто пошуку точки, до якої найближче знаходиться актор. Це необхідно для завдань, початком яких може бути певна анімація (наприклад анімація відкриття дверей), яка повинна програватися з конкретних координат та вимагає додаткового переміщення для коректного початку.

**2.3.2 Розробка системи створення сюжету**

Наступною частиною створення системи для сюжету гри є написання системи, яка дозволяє створити загальний сюжет: завдання, місії, глави, створення залежності між ними. Сюжетні завдання можуть знаходитися на різних ігрових картах, так і на різних World Partition [33]. Ці частини світу, або різні ігрові простори не будуть одночасно завантажені в гру з метою виконання вимог платформ щодо бюджетів оперативної пам’яті. Це накладає обмеження на виконавців завдань, які створені для конкретного актора та світу. Отримати інформацію з актора який існує лише ігровому в світі без завантаження його в оперативну пам’ять неможливо.

Загальний сюжет буде існувати незалежно від світу, та отримувати інформацію про стан завдань від виконавців завдань в ігровому просторі. Ці класи будуть мати кореневий об’єкт який віднаслідується від UGameInstance та мати назву UStorySystemGameInstance. Клас UGameInstance є об’єктом який не буде видалений протягом життєвого циклу всієї гри [34]. Цей клас також буде відповідати за загальну серіалізацію гри та створення файлів збереження прогресу гравця, яка буде описана в наступних главах.

Сюжет гри може бути досить об’ємним, тому його необхідно розділити на деякі логічні частини з якими простіше розуміти поділення сюжету. Також треба врахувати технічне обмеження, яке обґрунтовує необхідність подібного розділення на логічні системи.

Blueprints це скриптова мова програмування, яка зберігається в файлах-контейнерах типу uasset, які є бінарними файлами, а не текстовими. Бінарні файли не підтримують можливості вирішення конфліктів між різними ревізіями цього файлу. Якщо два програміста одночасно хочуть додати зміни в цей файл, то не існує ніякої можливості вирішити конфлікт між двома файлами. Одному з програмістів доведеться оновити цей файл до останньої ревізії та повторити вже виконану роботу в новій ревізії цього файлу.

Проблему можна вирішити використавши пропрієтарне програмне забезпечення Perforce для контролю версій. Perforce дозволяє блокувати файли, які не підтримують можливість вирішення конфліктів, для редагування, що не дозволить другому програмісту відредагувати файл, поки інший програміст не закінчить роботу з цим файлом та не звільнить його для редагування [35]. В такому випадку з’являється нове обмеження: якщо всі зміни будуть в одному файлі, то програмістам доведеться чекати поки цей файл буде розблокований, що створює проблему обмеженості доступу до файлу. Відповідно чим більше буде розділення на сюжетні елементи, тим простіше буде додати зміни в конкретний файл і не заважати іншим програмістам працювати над іншими завданнями та сюжетом.

Базовий клас, який буде відповідати за певну частину сюжету буде мати назву UStoryManagerBase. Цей клас доступний до розширення коду та додавання нових класифікаторів поділення сюжету на частини. Додатково був створений клас UActStoryManager, який дозволяє поділити сюжет гри на акти.

Завдання менеджера - це створити запрограмовані сюжетні місії, акти, глави або додати завдання під час проходження гри. Наприклад, після виконання завдання, гравець може отримати додаткове завдання.

Обов’язковою вимогою для менеджера є встановлення головного завдання з якого починається елемент сюжету, та можливість пошуку завдання за певними критеріями. Для виконання першої вимоги, структура, яка описує завдання, має поле типу bool яке має назву bMainTask. Менеджер зробить головним завдання, яке буде знаходитися першим в масиві завдання та має поле bMainTask зі значенням true. Друга вимога - створення функцій, які будуть за параметрами шукати завдання серед цього менеджера. Пошук можна здійснити викликавши одну з двох функцій (див. додаток Д.1).

Параметри пошуку є такими. Структура повністю підтримується для створення та редагування в Blueprints що дозволяє робити пошук завдань з скриптової мови програмування (див. додаток Д.2).

Завдання може мати залежності які встановлюють порядок виконання завдання. Відповідно, виконане завдання розблокує наступні завдання, та невиконане завдання не дає можливості почати наступні завдання. Для кожного класу завдання можна визначити її залежності. Ці дані описані в структурі FDependentTaskInfo (див. додаток Е).

Кожне завдання має свою унікальну назву. Ця назва є критерієм, який дозволяє пов’язувати відношення між завданням та виконавцем або залежність між завданнями. Кожен виконавець завдання підписується до завдання за його ім’ям та сповіщає коли завдання було виконане.

Унікальна назва буде мати тип FName. Цей клас зберігає хеш суму для рядка у вигляді простого числа типу uint64, а сам рядок знаходиться в окремому глобальному масиві рядків. Це було зроблено з метою оптимізації роботи з рядками. Операції порівняння між рядками є складною операцією O(n), де n – довжина рядка, а відповідно операція порівняння між числами є простою операцією - O(1).

Для опису завдання та необхідної інформації про завдання був створений базовий клас UStoryItemBase. Клас має такі поля для опису та базової логіки роботи з завданнями: збереження стану завдання, його прогрес, та всі активні виконавці завдань в конкретному світі (див. додаток Ж).

На основі цього класу було створено більш розширені типи завдань: UStoryTask, UStoryMission. UStoryMission зберігає в собі залежні завдання, які необхідно виконувати в порядку їх додавання до класу місії. Місія має функцію OnMissionInitialized, яка може бути реалізована в Blueprints або в нативному коді. Мета функції – виконати додатковий код після того, як була створена місія (див. додаток И.1).

Для додавання нового завдання під час виконання конкретної місії необхідно викликати функцію з параметрами нового завдання (див. додаток И.2).

Структура FStoryItemCreateInfo зберігає в собі інформацію про завдання яке треба створити та додати до місії: назва завдання, тип, клас завдання на яке треба додати (див. додаток И.3).

**2.3.3 Розробка системи серіалізації даних**

Наступною важливою підсистемою є збереження прогресу завдань в певний файл, який дозволить гравцю повернутися на тому місці, де він зупинився, або для того щоб дати можливість спробувати пройти завдання ще раз. Для цього необхідно зберегти інформацію про прогрес всіх завдань та місій, а потім прочитати та відновити прогрес. Цей процес називається серіалізацією даних.

Нижче наведений приклад серіалізації даних з використанням стандарту C++11. Нехай, цей клас має назву Person та має поля: рядок mName типу std::string, та число mAge типу int (див. додаток К.1). Ці поля треба серіалізувати [36].

В STL (Standart Template Library) визначено оператори для взаємодії з різними потоками даних (такими потоками є файл чи консоль) [37]:

1. << - Оператор запису в потік
2. >> - Оператор читання з потоку

Для того щоб створити серіалізацію класу Person необхідно визначити такі оператори запису та читання в потік (див. додаток К.2).

Також наведений остаточний приклад з серіалізацією та десеріалізацією даних цього класу в файл person.bin (див. додаток К.3).

Серіалізація в Unreal Engine є більш складною, але вона використовує схожий підхід. Базовий клас UObject має функцію Serialize [38], яка серіалізує клас:

virtual void Serialize(FArchive& Ar);

Базовим класом-контейнером для даних є FArchiver [39]. Цей клас підтримує читання та запис даних в бінарний файл, який буде містити інформацію про стан різних об’єктів в грі. Структура, яка наслідується від FArchiver та розширює систему серіалізації даних, є FSaveSubSystemArchive.

Ця структура визначає дві головні змінні які надалі будуть необхідні у створенні файла-збереження це:

1. ArIsSaveGame (цей нащадок класу FArchive є саме файлом збереження, так як FArchive може використовуватися в серіалізації даних для передачі по мережі).
2. ArNoDelta (ця змінна вказує що файли збереження є унікальними файлами, які не вимагають створення дельта файлів).

Дельта-файл - це файл який вказує на відмінність нової версії файлу від попередньої. Для зчитування останньої версії файлу необхідно взяти певну версію файлу та задіяти на ньому всі зміни, які були створені в кожному дельта-файлі. Такий підхід використовують для передачі даних по мережі (див. додаток Л.1).

FArchiver не поділяється на вхідний та вихідний потік даних та немає операторів << або >>. Поведінка архіву залежить від типу класів, які передаються в клас-контейнер. Для того, щоб створити файл-збереження та записати в нього дані, необхідно створити відповідний клас читач (FMemoryWriter), або клас записувач даних (FMemoryReader) в потік (див. додаток Л.2).

В реалізації функції Serialize в базовому класі UObject буде міститься цикл, який проітерується по всім полям рефлексії цього класу, та буде шукати змінну, який має флаг SaveGame. Ця змінна буде автоматично збережена та прочитана під час серіалізації та десеріалізації даних. Для того щоб створити поле, яке буде додане до серіалізації необхідно додати змінну до рефлексії з таким ключем:

UPROPERTY(SaveGame)

bool bTaskStarted = false;

Для того, щоб дати можливість серіалізувати гру за необхідністю (наприклад автоматичне збереження гри, чи ручне створення збереження), була створена функція SaveGame в класі USaveSubsystem, яка доступна для викликів як в С++, так в скриптовій мові Blueprints (див. додаток Л.3).

Функція створює FMemoryWriter та серіалізує клас UGameInstance, який зберігає в собі інформацію про всі завдання. Наступним кроком серіалізації – це ітерація по всьому світу та пошук акторів, які також необхідно серіалізувати. Для того, щоб додати об’єкт в світі, необхідно віднаслідувати клас від інтерфейсу ISaveSubsystemInterface та визначити функцію OnLevelDeserialized\_Implementation. Для кожного актора викликається функція Serialize. Потім серіалізовані дані записуються в файл-збереження.

Для того, щоб відновити прогрес з файлу-збереження, було створено дві функції. Перша з них завантажує останнє створене збереження гри, а друга завантажує файл, який має індекс, бо гра може мати більше одного збереження (див. додаток Л.4).

Функція LoadGame створює FMemoryReader та зчитує дані з архіву. Після цього відкриває світ, який був прописаний в файлі-збереженні. Після того, як світ був завантажений в пам’ять, викликається функція DeserializeGame, яка відновлює дані для кожного актора, який наслідується від інтерфейсу ISaveSubsystemInterface.

**2.3.4 Розробка UI**

Вище описаний код працює непомітно для гравця, тому необхідно створити спосіб показувати гравцю активні завдання.

Для того, щоб додати користувацький інтерфейс (User Interface), необхідно створити клас, який наслідується від AHUD. Це актор, який існує в світі, та відповідає за весь UI який може бачити гравець [40]. Для цього було створений клас AStorySystemHUD, який додає віджет, що показує список активних завдань. Нижче наведено код, який створює та додає до HUD гравця необхідний віджет. Цей код може бути перенесений в будь який інший HUD (див. додаток М.1).

UStorySystemTaskTargets є віджетом, який розміщує віджети що показують стан поточного завдання на екрані. Цей віджет розміщується за вказаними якорями та його розмір залежить повністю від контенту, який в ньому розміщений [41].

Віджет використовує делегат OnStoryItemChanged, який належать класу UStorySystemGameInstance. Цей делегат викликає функцію, яка оновлює UI, коли поточне завдання було змінено. Віджет може показувати будь яку кількість активних завдань в певний момент проходження гри.

Мінімальною одиницею для опису завдання чи його необхідних пунктів для виконання є клас віджет UTaskTargetDescription (рис. 2.2). Цей клас може бути:

1. Заголовок, в якому прописана назва завдання.
2. Частина завдання, в якому прописано опис під завдання.

Тип класу можна вказати за допомогою перечислення ETaskDescriptionType (див. додаток М.2) (рис. 2.3).

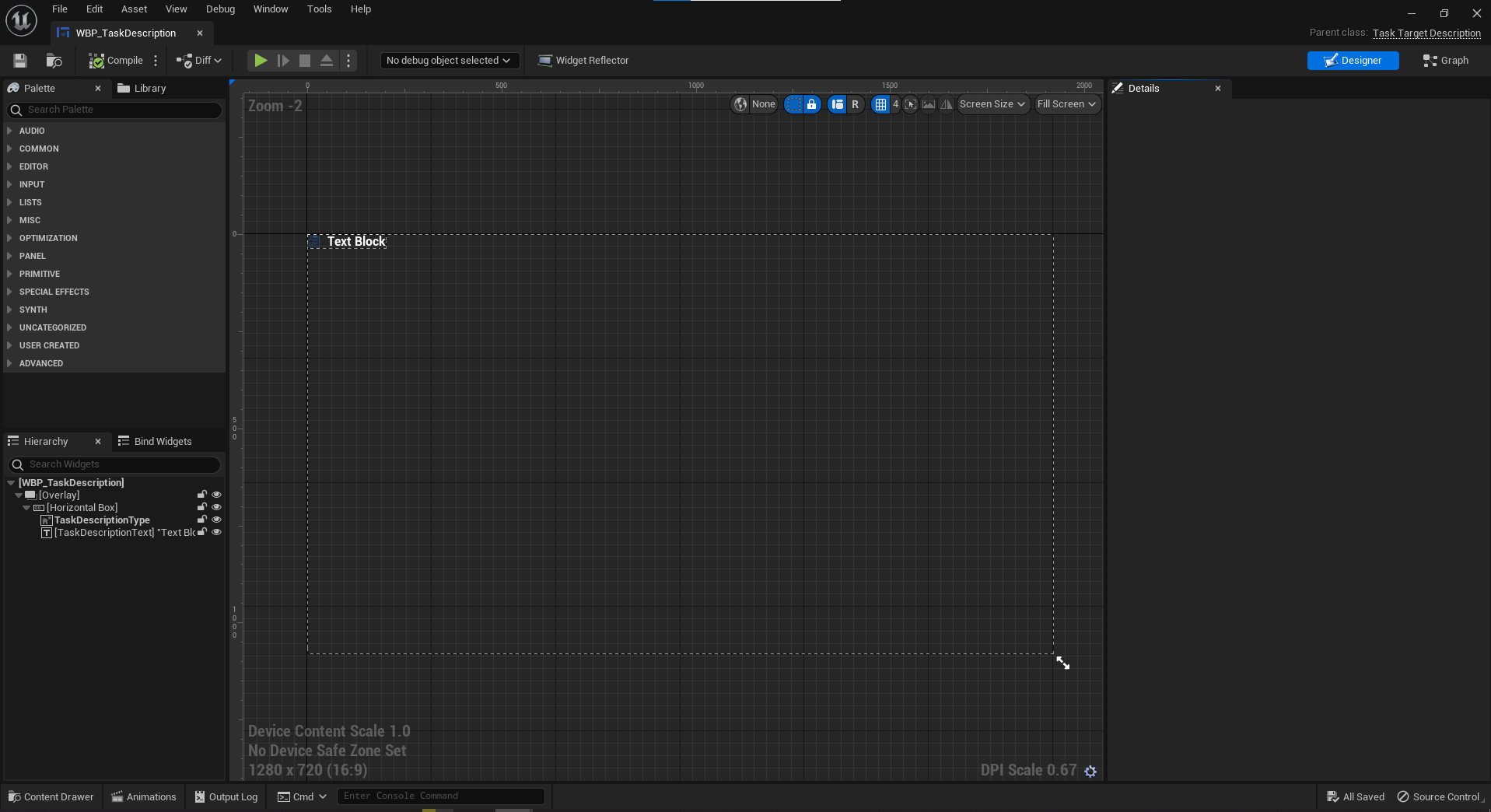


Рис.2.2. Редактор віджету UTaskTargetDescription.

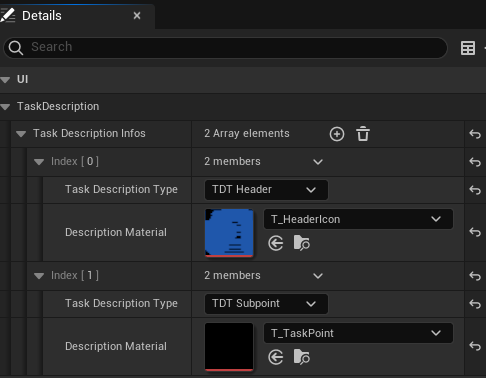


Рис. 2.3. Налаштування іконок для ETaskDescriptionType.

**2.4 Рекомендації щодо подальшого розвитку написаного модуля**

Цей код дає певну основу для того, щоб створювати ігри з сюжетом. Але будь який код можна додатково покращити. Нижче будуть перераховані аспекти, модифікація яких, може призвести до покращення модуля для створення сюжету в грі:

1. Перенести інформацію про завдання з тіла класу, до спеціальної структури, яка віднаслідується від UDataTable. Таким чином, інформацію про завдання можна буде створювати в окремому Blueprint який дозволить зручніше редагувати певну інформацію про завдання типу опису, тощо, та імпортувати відразу з csv, excel або json файлів.
2. Додати додаткову валідацію даних які вводить дизайнер завдань чи програміст. Необхідно перевіряти на коректність та присутність даних для певних полів (як наприклад ім’я завдання).
3. Розширити різні способи взаємодії з завданнями та способами виконання цих завдань.
4. Додати створення сюжетних елементів на вимогу для економії пам’яті. На даних момент UStorySystemGameInstance створює всі завдання, які існують в масиві завдань, але більша частина з них може не використовуватися, так як вони залежать від виконавців, які знаходяться в іншому ігровому світі
5. Оптимізувати вивантаження об’єктів в пам’яті, виконавці яких знаходяться на неактивних World Partition, та відповідно завантажувати необхідні об’єкти, якщо гравець перейшов на world partition де ці завдання є активними.

**ВИСНОВКИ**

1. Опрацьовано інформаційні джерела з теми дипломної роботи: проаналізовано ринок дистрибуції комп’ютерних ігор, досліджено особливості та механізми створення сюжетно-орієнтованої гри на ігровому рушії Unreal Engine 5.

2. Визначена специфіка розробки ігор в ігровій індустрії. Визначені кваліфікації, які відповідають за певні елементи під час розробки ігор. Визначено необхідність використання скриптової мови в рушії. Окреслений data driven programming підхід під час створення гри.

3. Опрацьовано особливості роботи з рушієм Unreal Engine 5. Встановлено, що рушій має ряд особливостей, які впливають на процес розробки. Цими особливостями є Garbage Collector, скриптова мова програмування Blueprints та система рефлексії, яка поєднує взаємодію всіх компонентів рушія.

4. Спроектовано систему для створення сюжету з урахуванням особливостей рушія Unreal Engine 5. В результаті роботи над проєктом створено модуль, який можна додавати до будь якого Unreal Engine 5 проєкту. Цей модуль надає можливість розробки сюжетно-орієнтованих ігор для студій-початківців в ігровій індустрії.

5. Окреслено можливі способи покращення написаного модуля, а саме: наслідування від базових класів та перевантаження функції з метою розширення способів взаємодії гравця з системою для створення сюжету; написання додаткових перевірок для даних, які задають ігрові дизайнери; перенесення даних про завдання до спеціальних файлів типу UDataTable, які спрощують подальше редагування даних про елементи сюжету та додавання них в рушій; оптимізація використання пам’яті для елементів загального сюжету з урахуванням різних світів та розподілення світу на елементи (World Partitions).

**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. Database of everything on Steam [Електронний ресурс] – Режим доступу до ресурсу: <https://steamdb.info/>
2. Enigma machine [Електронний ресурс] – Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/Enigma_machine>
3. Spacewar! [Електронний ресурс] – Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/Spacewar>!
4. Colossal Cave Adventure [Електронний ресурс] – Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/Colossal_Cave_Adventure>
5. Народження легенди: історія Атарі 2600 [Електронний ресурс] – Режим доступу до ресурсу: <https://playua.net/narodzhennya-legendy-istoriya-atari-2600/>
6. Рушії, які пішли від Quake [Електронний ресурс] – Режим доступу до ресурсу: <https://upload.wikimedia.org/wikipedia/commons/6/63/Quake_-_family_tree.svg>
7. Skybox (video games) [Електронний ресурс] – Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/Skybox_(video_games)>
8. The most popular video games now [Електронний ресурс] – Режим доступу до ресурсу: <https://gaminggorilla.com/most-popular-video-games-now/>
9. Call of Duty: Warzone [Електронний ресурс] – Режим доступу до ресурсу: <https://uk.wikipedia.org/wiki/Call_of_Duty:_Warzone>
10. Game server [Електронний ресурс] – Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/Game_server>
11. Hacking wins [Електронний ресурс] – Режим доступу до ресурсу: <https://surfshark.com/hacking-wins>
12. Digital distribution [Електронний ресурс] – Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/Digital_distribution>
13. The Year in Numbers 2020 [Електронний ресурс] – Режим доступу до ресурсу: <https://www.gamesindustry.biz/gamesindustry-biz-presents-the-year-in-numbers-2020>
14. Control - RTX ON vs RTX OFF Comparison - Real Time Ray Tracing - Max Settings - 4k [Електронний ресурс] – Режим доступу до ресурсу: <https://www.youtube.com/watch?v=45pfvYX-fxU>
15. PlayStation 4 system software [Електронний ресурс] – Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/PlayStation_4_system_software>
16. Xbox system software [Електронний ресурс] – Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/Xbox_system_software>
17. Google Play vs the Apple App Store: App Stats and Trends [Електронний ресурс] – Режим доступу до ресурсу: <https://42matters.com/stats>
18. Скільки заробляють на play market. Як люди заробляють на Google Play Market [Електронний ресурс] – Режим доступу до ресурсу: <https://virtuapc.ru/ipad/how-much-they-earn-on-the-play-market-how-people-make-money-on-the-google-play-store/>
19. 4A Games [Електронний ресурс] – Режим доступу до ресурсу: <https://www.4a-games.com.mt/>
20. Unreal Engine 5 [Електронний ресурс] – Режим доступу до ресурсу: <https://www.unrealengine.com/en-US/>
21. Unreal Engine: Film and television [Електронний ресурс] – Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/Unreal_Engine#Film_and_television>
22. Entity Component System [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/index.html>
23. 24 Great Games That Use The Unreal Engine 4 Game Engine [Електронний ресурс] – Режим доступу до ресурсу: <https://www.thegamer.com/great-games-use-unreal-4-game-engine/>
24. Resource Acquisition Is Initialization [Електронний ресурс] – Режим доступу до ресурсу: <https://uk.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization>
25. Smart pointers [Електронний ресурс] – Режим доступу до ресурсу: <https://en.cppreference.com/book/intro/smart_pointers>
26. Unreal Engine 5 Classes [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.1/en-US/API/Classes/>
27. Mark and Sweep garbage collection algorithm [Електронний ресурс] – Режим доступу до ресурсу: <https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/>
28. Рефлексія (програмування) [Електронний ресурс] – Режим доступу до ресурсу: <https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%BB%D0%B5%D0%BA%D1%81%D1%96%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)>
29. Objects [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.1/en-US/objects-in-unreal-engine/>
30. Property Specifiers [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.1/en-US/unreal-engine-uproperty-specifiers/>
31. Function Specifiers [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.1/en-US/function-specifiers/>
32. UActorComponent [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/Components/UActorComponent/>
33. World Partition [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.1/en-US/world-partition-in-unreal-engine/>
34. UGameInstance [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/Engine/UGameInstance/>
35. Version Control for Binary Files [Електронний ресурс] – Режим доступу до ресурсу: <https://www.perforce.com/blog/vcs/version-control-for-binary-files>
36. An Introduction to Object Serialization in C++ [Електронний ресурс] – Режим доступу до ресурсу: <https://www.codeguru.com/cplusplus/an-introduction-to-object-serialization-in-c/>
37. Operators overloading [Електронний ресурс] – Режим доступу до ресурсу: <https://en.cppreference.com/w/cpp/language/operators>
38. Рушій Unreal Engine [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.0/en-US/API/Runtime/CoreUObject/UObject/UObject/Serialize/2/>
39. UObject::Serialize [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.1/en-US/API/Runtime/Core/Serialization/FArchive/>
40. AHUD [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/GameFramework/AHUD/>
41. Creating Widgets [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unrealengine.com/5.1/en-US/creating-widgets-in-unreal-engine/>
42. More Than Stories With Buttons: Narrative, Mechanics, and Context as Determinants of Player Experience in Digital Games, June 2014, Journal of Communication 64(3):521-542, DOI:10.1111/jcom.12096
43. Paul Ralph and Kafui Monu. Toward a unified theory of digital games. The Computer Games Journal, 4: 81-100, 2015.
44. Raza, A. (2014). The Last of Us Is the Most Awarded Game in History by Critics. Gearnuke. Retrieved from <http://gearnuke.com/the-last-of-us-awarded-game-history-critics/>
45. Lankoski, P., & Björk, S. (2015). Formal Analysis of Gameplay. In Lankoski, P., & Björk, S. (Eds.), Game Research Methods (pp. 23-36). Retrieved from <http://press.etc.cmu.edu/content/game-research-methods-overview>

**ДОДАТКИ**

ДОДАТОК А

Розумні вказівники, що приводять до помилки виконання

#include <memory>

class A {};

int main()

{

A\* pA = new A;

*std*::*shared\_ptr*<A> sA(pA);

*std*::*shared\_ptr*<A> sB(pA);

}

ДОДАТОК Б

Створення класу та структури, які доступні в системі рефлексії

UCLASS(Abstract, BlueprintType, Blueprintable)

class BDEPROJECT\_API UMyClass : public UObject

{

GENERATED\_BODY()

}

USTRUCT(BlueprintType)

struct FMyStruct

{

GENERATED\_BODY()

}

ДОДАТОК В

Приклад використання рефлексії в коді

UProperty\* Health = ActorClass->FindPropertyByName(TEXT("Health"));

// Отримання значення властивості

float HealthValue = HealthProperty->GetPropertyValue\_InContainer(MyActor);

// Виклик методу за іменем

UFunction\* TakeDamageFunction = ActorClass->FindFunctionByName(TEXT("TakeDamage"));

FVector DamageVector(1.f, 0.f, 0.f);

MyActor->ProcessEvent(TakeDamageFunction, &DamageVector);

ДОДАТОК Г

Перечислення методів взаємодії гравця з виконавцем в світі

UENUM(BlueprintType)

enum class EInteractionType : uint8

{

IT\_None = 0x0 UMETA(DisplayName = "None"),

IT\_ByTimer = 0x1 UMETA(DisplayName = "End by timer"),

IT\_ByCondition = 0x2 UMETA(DisplayName = "End by condition"),

IT\_ByOverlap = 0x4 UMETA(DisplayName = "End by collision overlap"),

IT\_Custom1 = 0x8 UMETA(DisplayName = "Custom mode"),

IT\_Custom2 = 0x10 UMETA(DisplayName = "Custom mode"),

IT\_Custom3 = 0x20 UMETA(DisplayName = "Custom mode"),

IT\_Custom4 = 0x40 UMETA(DisplayName = "Custom mode"),

IT\_Custom5 = 0x80 UMETA(DisplayName = "Custom mode"),

};

ДОДАТОК Д.1

Функції для пошуку завдань в менеджері

/\* Пошук за іменем завдання \*/

UFUNCTION(BlueprintCallable)

virtual UStoryItemBase\* FindItemByName(const FName& TaskName) const;

/\* Пошук за параметрами \*/

UFUNCTION(BlueprintCallable)

virtual void FindTasksByParams(const FStoryItemSearchParams& StoryItemSearchParams, TArray<UStoryItemBase\*>& OutArray);

ДОДАТОК Д.2

USTRUCT(BlueprintType)

struct FStoryItemSearchParams

{

GENERATED\_BODY()

/\* Пошук за іменем завдання \*/

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|SearchInfo")

FName ItemName;

/\* Пошук за критерієм завершеності завдання \*/

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|SearchInfo")

bool bCompleted = false;

/\* Пошук за типом завдання: завдання чи місія \*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|SearchInfo")

EStoryItemType StoryItemType = EStoryItemType::SIP\_NoType;

/\* Пошук всіх завдання конкретного типу \*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|SearchInfo")

TSubclassOf<class UStoryItemBase> ItemClass;

/\* Критерій пошуку: всі завдання чи одне які задовольняють умову \*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|SearchInfo")

ESearchType SearchType;

};

ДОДАТОК Е

Структури для опису залежності завдань

UENUM(BlueprintType)

enum class ETaskDependency : uint8

{

TD\_Unlock UMETA(DisplayName = "Tasks which unlock finishing the current one"),

TD\_Requied UMETA(DisplayName = "Tasks which required to finish to unlock the current one"),

};

USTRUCT(BlueprintType)

struct FDependentTaskInfo

{

GENERATED\_BODY()

/\* Тип залежності: необхідно виконати або розблокування нових \*/

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|Config")

ETaskDependency TaskDependency;

/\* Ім’я іншого завдання \*/

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|Config")

FName TaskName;

};

ДОДАТОК Ж

Поля класу UStoryItem

/\* Унікальне ім’я завдання \*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="StorySystem|Item")

FName ItemName = DefaultItemName;

/\* Тип завдання: місія або завдання \*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Item")

EStoryItemType StoryItemType = EStoryItemType::SIP\_NoType;

/\* Назва завдання яке буде відображено гравцю на дисплеї монітору \*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Item")

FText TaskHeader;

/\* Опис завдання яке буде відображено гравцю на дисплеї монітору \*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Item")

FText TaskDescription;

/\* Поточний стан завдання: почато, в прогресі, закінчено, не валідне \*/

UPROPERTY(BlueprintReadOnly, SaveGame, Category = "StorySystem|Item")

EStoryItemState StoryItemState = EStoryItemState::SIS\_None;

/\* Виконавці які підписані на це завдання \*/

UPROPERTY(BlueprintReadOnly, Category = "StorySystem|Item")

TArray<UTaskExecutor\*> Executors;

ДОДАТОК И.1

Особливості класу UStoryMission

UFUNCTION(BlueprintNativeEvent)

void OnMissionInitialized();

virtual void OnMissionInitialized\_Implementation();

ДОДАТОК И.2

UFUNCTION(BlueprintCallable)

virtual void AddNewStoryItem(const FStoryItemCreateInfo& CreateInfo);

ДОДАТОК И.3

USTRUCT(BlueprintType)

struct FStoryItemCreateInfo

{

GENERATED\_BODY()

/\* Назва завдання \*/

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|ItemInfo")

FName ItemName;

/\* Тип: місія, або завдання \*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Item")

EStoryItemType StoryItemType;

/\* UClass об’єкта, що наслідується від UStoryTask \*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Item")

TSubclassOf<class UStoryTask> ItemClass;

};

ДОДАТОК К.1

Серіалізація в С++

class Person

{

public:

*std*::*string* mName;

int mAge;

};

ДОДАТОК К.2

friend *std*::*ostream*& operator<<(*std*::*ostream*& os, const Person& person)

{

os << person.mName << '\n';

os << person.mAge << '\n';

return os;

}

friend *std*::*istream*& operator>>(*std*::*istream*& is, Person& person)

{

*std*::*getline*(is, person.mName);

is >> person.mAge;

return is;

}

ДОДАТОК К.3

#include <fstream>

#include <iostream>

#include <string>

class Person

{

public:

*std*::*string* mName;

int mAge;

friend *std*::*ostream*& operator<<(*std*::*ostream*& os, const Person& person)

{

Продовження додатку К3

os << person.mName << '\n';

os << person.mAge << '\n';

return os;

}

friend *std*::*istream*& operator>>(*std*::*istream*& is, Person& person)

{

*std*::*getline*(is, person.mName);

is >> person.mAge;

return is;

}

};

int main()

{

Person p;

p.mName = "John";

p.mAge = 30;

*std*::*ofstream* file("person.bin", *std*::*ios*::*binary*);

file << p;

file.*close*();

*std*::*ifstream* infile("person.bin", *std*::*ios*::*binary*);

Person p2;

infile >> p2;

infile.*close*();

*std*::*cout* << "Name: " << p2.mName << '\n';

*std*::*cout* << "Age: " << p2.mAge << '\n';

}

ДОДАТОК Л.1

Система серіалізації в Unreal Engine 5

/\*

\* Used for loading and saving.

\*/

struct *FSaveSubsystemArchive* : public FObjectAndNameAsStringProxyArchive

{

*FSaveSubsystemArchive*(FArchive& InInnerArchive, bool bInLoadIfFindFails);

};

*FSaveSubsystemArchive*::*FSaveSubsystemArchive*(FArchive& InInnerArchive, bool bInLoadIfFindFails)

: FObjectAndNameAsStringProxyArchive(InInnerArchive, bInLoadIfFindFails)

{

// Архів є файлом збереження

ArIsSaveGame = true;

// До архіву не треба застосовувати дельта-файл

ArNoDelta = true;

}

ДОДАТОК Л.2

// Запис даних до файлу

UGameInstance\* GameInstance = GetGameInstance();

GameSaveData.GameInstance = *FObjectSaveData*(GetGameInstance());

FMemoryWriter MemoryWriter(GameSaveData.GameInstance.RawData, true);

*FSaveSubsystemArchive* Archive(MemoryWriter, false);

GameInstance->Serialize(Archive);

// Читання даних у файл

UGameInstance\* GameInstance = GetGameInstance();

FMemoryReader MemoryReader(GameSaveData.GameInstance.RawData, true);

*FSaveSubsystemArchive* Archive(MemoryReader, false);

GameInstance->Serialize(Archive);

ДОДАТОК Л.3

UFUNCTION(BlueprintCallable, Category = "SaveSubsystem")

virtual void SaveGame();

ДОДАТОК Л.4

UFUNCTION(BlueprintCallable, Category = "SaveSubsystem")

virtual void LoadLastGame();

UFUNCTION(BlueprintCallable, Category = "SaveSubsystem")

virtual void LoadGame(int32 SaveId);

ДОДАТОК М.1

Створення UI

APlayerController\* PC = GetWorld()->GetFirstPlayerController();

*UStorySystemTaskTargets*\* StorySystemTaskTargets = CreateWidget<*UStorySystemTaskTargets*>(PC, *UStorySystemTaskTargets*::StaticClass());

StorySystemTaskTargets->AddToViewport();

ДОДАТОК М.2

UENUM(BlueprintType)

enum class ETaskDescriptionType : uint8

{

TDT\_Header,

TDT\_Subpoint,

};

ДОДАТОК Н

Заголовки модуля для створення сюжетно орієнтованої гри

StorySystemType.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "UObject/NoExportTypes.h"

#include "StorySystemType.generated.h"

UENUM(BlueprintType)

enum class *EStoryItemType* : uint8

{

*SIP\_NoType* UMETA(DisplayName = "Used only in base class"),

*SIP\_Mission* UMETA(DisplayName = "Item describes game mission"),

*SIP\_Task* UMETA(DisplayName = "Item describes game task"),

};

UENUM(BlueprintType)

enum class *ESearchType* : uint8

{

*ST\_Combine* UMETA(DisplayName = "Combine tasks"),

*ST\_FindOne* UMETA(DisplayName = "Search one task by parameter"),

ST\_None UMETA(DisplayName = "Default mode combine"),

};

/\*

\* Search params for task

\*/

USTRUCT(BlueprintType)

struct *FStoryItemSearchParams*

{

GENERATED\_BODY()

/\*

\* Search task by name

\*/

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|SearchInfo")

FName ItemName;

/\*

\* Search by is completed param

\*/

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|SearchInfo")

bool bCompleted = false;

/\*

\* Search by item type

\*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|SearchInfo")

*EStoryItemType* *StoryItemType* = *EStoryItemType*::*SIP\_NoType*;

/\*

\* Search by class

Продовження додатку Н

\*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|SearchInfo")

TSubclassOf<class *UStoryItemBase*> *ItemClass*;

/\*

\* Search type

\*/

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|SearchInfo")

*ESearchType* *SearchType*;

};

UENUM(BlueprintType)

enum class *EStoryItemState* : uint8

{

*SIS\_None* UMETA(DisplayName = "State none"),

*SIS\_Started* UMETA(DisplayName = "State item started"),

*SIS\_InProgress* UMETA(DisplayName = "State in progress"),

*SIS\_Finished* UMETA(DisplayName = "State finishied"),

*SIS\_Invalid* UMETA(DisplayName = "State invalid"),

};

UENUM(BlueprintType)

enum class *EStoryEndReason* : uint8

{

*SER\_Interupted* UMETA(DisplayName = "Interupted story"),

*SER\_Finished* UMETA(DisplayName = "Finished story"),

};

StorySystemGameInstance.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "Engine/GameInstance.h"

#include "StorySystemGameInstance.generated.h"

class *UStoryManagerBase*;

class *UTaskExecutor*;

class *UStoryItemBase*;

/\*

\*

\*/

USTRUCT(BlueprintType)

struct FStoryManagerSetupConfig

{

GENERATED\_BODY()

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Setup")

TSubclassOf<*UStoryManagerBase*> ManagerClass;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Setup")

bool bMainAct = false;

};

Продовження додатку Н

DECLARE\_MULTICAST\_DELEGATE\_TwoParams(FOnStoryItemChangedSignature, *UStoryItemBase*\* /\*NewItem\*/, *UStoryItemBase*\* /\*OldItem\*/)

DECLARE\_MULTICAST\_DELEGATE(FOnForceUpdateUISignature)

UCLASS(BlueprintType)

class STORYSYSTEM\_API UStorySystemGameInstance : public UGameInstance

{

GENERATED\_BODY()

public:

virtual void Init() override;

virtual void Shutdown() override;

virtual void Serialize(FArchive& Ar) override;

UFUNCTION(BlueprintCallable)

virtual TArray<*UStoryItemBase*\*> GetFirstInitedTasks() const;

UFUNCTION(BlueprintCallable)

virtual *UStoryManagerBase*\* GetActiveStoryManager() const;

UFUNCTION(BlueprintCallable)

virtual *UStoryManagerBase*\* GetActiveStoryManagerByIndex(int32 Index) const;

UFUNCTION(BlueprintCallable)

virtual bool RegisterExecutor(*UTaskExecutor*\* TaskParams);

UFUNCTION(BlueprintCallable)

virtual bool EndStory(*UTaskExecutor*\* TaskParams);

UFUNCTION(BlueprintCallable)

virtual bool ActivateNextManager(*UStoryManagerBase*\* ThisManager);

UFUNCTION(BlueprintCallable)

virtual void ResetManagers();

UFUNCTION(BlueprintCallable)

virtual void InitializeMainManager();

public:

FOnStoryItemChangedSignature OnStoryItemChanged;

FOnForceUpdateUISignature OnForceUpdateUI;

private:

*UStoryItemBase*\* Private\_FindTask(*UTaskExecutor*\* TaskParams);

void CreateManagers();

void DeleteManagers();

protected:

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem")

TArray<FStoryManagerSetupConfig> StoryManagerSetupConfig;

UPROPERTY(BlueprintReadWrite, Category = "StorySystem")

TArray<*UStoryManagerBase*\*> StoryManagers;

Продовження додатку Н

protected:

UPROPERTY(SaveGame)

int32 MainActIndex = -1;

UPROPERTY(SaveGame)

int32 CurrentActiveActIndex = -1;

private:

bool bMainActCreated = false;

};

StorySystem.h

// Copyright Epic Games, Inc. All Rights Reserved.

#pragma once

#include "CoreMinimal.h"

DECLARE\_LOG\_CATEGORY\_EXTERN(LogStorySystem, Log, All);

StorySystem.Build.cs

using UnrealBuildTool;

public class StorySystem : ModuleRules

{

public StorySystem(ReadOnlyTargetRules TargetRules) : base(TargetRules)

{

PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "DeveloperSettings" });

}

}

TaskTargetDescription.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "Blueprint/UserWidget.h"

#include "Materials/Material.h"

#include "Components/Image.h"

#include "Components/TextBlock.h"

#include "Engine/Texture.h"

#include "TaskTargetDescription.generated.h"

UENUM(BlueprintType)

enum class ETaskDescriptionType : uint8

{

TDT\_Header,

TDT\_Subpoint,

};

Продовження додатку Н

USTRUCT(BlueprintType)

struct FTaskDescriptionInfo

{

GENERATED\_BODY()

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "UI|TaskDescription")

ETaskDescriptionType TaskDescriptionType;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "UI|TaskDescription")

UTexture2D\* DescriptionMaterial;

};

UCLASS()

class STORYSYSTEM\_API UTaskTargetDescription : public UUserWidget

{

GENERATED\_BODY()

public:

virtual void UpdateWidgetByTaskDescription(ETaskDescriptionType NewTaskDescriptionType, const FText& Text);

protected:

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "UI|TaskDescription")

TArray<FTaskDescriptionInfo> TaskDescriptionInfos;

UPROPERTY(meta=(BindWidget))

UImage\* TaskDescriptionType;

UPROPERTY(meta = (BindWidget))

UTextBlock\* TaskDescriptionText;

};

StorySystemTaskTargets.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "Blueprint/UserWidget.h"

#include "Components/Overlay.h"

#include "StorySystemTaskTargets.generated.h"

class *UStoryItemBase*;

class UTaskTargetDescription;

UCLASS()

class STORYSYSTEM\_API *UStorySystemTaskTargets* : public UUserWidget

{

GENERATED\_BODY()

public:

virtual void NativeConstruct() override;

virtual void NativeDestruct() override;

Продовження додатку Н

protected:

virtual void *OnItemChanged*(*UStoryItemBase*\* Item, *UStoryItemBase*\* *OldItem*);

virtual void *BuildUI*(const TArray<*UStoryItemBase*\*>& Items);

virtual void OnForceUpdateUI();

private:

FDelegateHandle *OnItemChangedDelegateHandle*;

FDelegateHandle *OnForceUpdateUIDelegateHandle*;

protected:

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="UI|Items")

TSubclassOf<UTaskTargetDescription> *TaskTargetDescriptionClass*;

UPROPERTY(meta = (BindWidget))

UOverlay\* *ItemsOverlay*;

UPROPERTY(BlueprintReadOnly)

TArray<*UStoryItemBase*\*> *ActiveItems*;

};

StorySystemHUD.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "GameFramework/HUD.h"

#include "StorySystem/UI/StorySystemTaskTargets.h"

#include "StorySystemHUD.generated.h"

UCLASS()

class STORYSYSTEM\_API AStorySystemHUD : public AHUD

{

GENERATED\_BODY()

public:

AStorySystemHUD();

virtual void BeginPlay() override;

virtual void SetNewVisibilityForStorySystemTargets(bool bVisible);

protected:

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "HUD|StorySystemContainer")

TSubclassOf<*UStorySystemTaskTargets*> StorySystemTaskTargetsClass;

UPROPERTY(BlueprintReadWrite, Category = "HUD|StorySystemContainer")

*UStorySystemTaskTargets*\* StorySystemTaskTargets;

};

Продовження додатку Н

StoryTask.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "StorySystem/StoryItems/StoryItemBase.h"

#include "StoryTask.generated.h"

UCLASS(Abstract, BlueprintType, Blueprintable)

class STORYSYSTEM\_API *UStoryTask* : public *UStoryItemBase*

{

GENERATED\_BODY()

public:

*UStoryTask*(const FObjectInitializer& ObjectInitializer);

virtual void EndStory(*EStoryEndReason* *StoryEndReason*) override;

};

StoryMission.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "StorySystem/StoryItems/StoryItemBase.h"

#include "StoryMission.generated.h"

USTRUCT(BlueprintType)

struct *FStoryItemCreateInfo*

{

GENERATED\_BODY()

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|ItemInfo")

FName *ItemName*;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Item")

*EStoryItemType* *StoryItemType*;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Item")

TSubclassOf<class UStoryTask> *ItemClass*;

};

UCLASS(Abstract, BlueprintType, Blueprintable)

class STORYSYSTEM\_API *UStoryMission* : public *UStoryItemBase*

{

GENERATED\_BODY()

public:

virtual void *StartStory*() override;

UFUNCTION(BlueprintCallable)

virtual void *AddNewStoryItem*(const *FStoryItemCreateInfo*& CreateInfo);

Продовження додатку Н

UFUNCTION(BlueprintNativeEvent)

void *OnMissionInitialized*();

virtual void *OnMissionInitialized\_Implementation*();

UFUNCTION(BlueprintCallable)

virtual *UStoryItemBase*\* *FindItemByName*(const FName& TaskName);

virtual void *FindTasksByParams*(const *FStoryItemSearchParams*& StoryItemSearchParams, TArray<*UStoryItemBase*\*>& OutArray) override;

virtual TArray<FText> *GetTaskDescription*() const override;

virtual void *Serialize*(FArchive& Ar) override;

virtual void *EndStory*(*EStoryEndReason* StoryEndReason) override;

protected:

*UStoryItemBase*\* *GetNextTask*() const;

virtual void *InitChildItemsInMission*();

private:

bool *bItemsInitiedInMission* = false;

protected:

UPROPERTY(BlueprintReadOnly, Category = "StorySystem|Items")

TMap<FName, *UStoryItemBase*\*> *ChildStoryItems*;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Items")

TArray<*FStoryItemCreateInfo*> *MissionCreateInfo*;

};

StoryItemBase.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "UObject/NoExportTypes.h"

#include "StorySystem/StorySystemType.h"

#include "StoryItemBase.generated.h"

/\*

\*

\*/

DECLARE\_DYNAMIC\_MULTICAST\_DELEGATE\_TwoParams(FOnItemStateChangedSignature, class *UStoryItemBase*\*, Item, *EStoryItemState*, *ItemState*);

class *UTaskExecutor*;

class *UStoryManagerBase*;

/\*\*

\*

\*/

UCLASS(Abstract)

Продовження додатку Н

class STORYSYSTEM\_API *UStoryItemBase* : public UObject

{

GENERATED\_BODY()

friend class *UStoryItemBase*;

public:

*UStoryItemBase*(const FObjectInitializer& ObjectInitializer);

UFUNCTION(BlueprintCallable)

virtual FName GetItemName() const;

UFUNCTION(BlueprintCallable)

virtual void *SetStoryItemType*(*EStoryItemType* InStoryItemType);

UFUNCTION(BlueprintCallable)

virtual void *SetItemName*(FName InItemName);

UFUNCTION(BlueprintCallable)

virtual *EStoryItemState* *GetItemState*() const;

UFUNCTION(BlueprintCallable)

virtual bool *IsItemCompleted*() const;

UFUNCTION(BlueprintCallable)

virtual void *StartStory*();

virtual bool *ValidateStoryStart*();

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "ValidateStoryStart"))

bool *ReceiveValidateStoryStart*();

virtual bool *AreRequirementsSatisfiedToStartTask*();

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "AreRequirementsSatisfiedToStartTask"))

bool *ReceiveAreRequirementsSatisfiedToStartTask*();

UFUNCTION(BlueprintCallable)

virtual void EndStory(*EStoryEndReason* *StoryEndReason*);

UFUNCTION(BlueprintCallable)

virtual *EStoryItemType* *GetStoryItemType*() const;

UFUNCTION(BlueprintCallable)

virtual void *FindTasksByParams*(const *FStoryItemSearchParams*& StoryItemSearchParams, TArray<*UStoryItemBase*\*>& OutArray);

UFUNCTION(BlueprintCallable)

virtual *UStoryManagerBase*\* GetManager() const;

UFUNCTION(BlueprintCallable)

virtual FText *GetTaskHeader*() const;

UFUNCTION(BlueprintCallable)

virtual TArray<FText> *GetTaskDescription*() const;

Продовження додатку Н

protected:

UFUNCTION(BlueprintCallable)

virtual void *SetItemState*(*EStoryItemState* NewStoryItemState);

public:

virtual void RegisterExecutor(*UTaskExecutor*\* *Executor*);

/\*

\* Called when world is being destructed

\*/

virtual void *CleanupExecutors*();

protected:

virtual void *OnItemStateChanged*(*EStoryItemState* NewStoryItemState);

public:

UPROPERTY(BlueprintReadOnly, Category = "StorySystem|Item")

FName *DefaultItemName* = NAME\_None;

UPROPERTY(BlueprintReadWrite, Category="StorySystem|Item")

FOnItemStateChangedSignature *OnItemStateChangedDelegate*;

protected:

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="StorySystem|Item")

FName ItemName = *DefaultItemName*;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Item")

*EStoryItemType* *StoryItemType* = *EStoryItemType*::*SIP\_NoType*;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Item")

FText *TaskHeader*;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Item")

FText *TaskDescription*;

UPROPERTY(BlueprintReadOnly, SaveGame, Category = "StorySystem|Item")

*EStoryItemState* *StoryItemState* = *EStoryItemState*::*SIS\_None*;

UPROPERTY(BlueprintReadOnly, Category = "StorySystem|Item")

TArray<*UTaskExecutor*\*> *Executors*;

};

TaskExecutor.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "UObject/NoExportTypes.h"

#include "StorySystem/StoryItems/StoryItemBase.h"

#include "StorySystem/StorySystemType.h"

#include "TaskExecutor.generated.h"

UENUM(BlueprintType)

Продовження додатку Н

enum class *EInteractionType* : uint8

{

*IT\_None* = 0x0 UMETA(DisplayName = "None"),

*IT\_ByTimer* = 0x1 UMETA(DisplayName = "End by timer"),

*IT\_ByCondition* = 0x2 UMETA(DisplayName = "End by condition"),

*IT\_ByOverlap* = 0x4 UMETA(DisplayName = "End by colision overlap"),

*IT\_Custom1* = 0x8 UMETA(DisplayName = "Custom mode"),

*IT\_Custom2* = 0x10 UMETA(DisplayName = "Custom mode"),

*IT\_Custom3* = 0x20 UMETA(DisplayName = "Custom mode"),

*IT\_Custom4* = 0x40 UMETA(DisplayName = "Custom mode"),

*IT\_Custom5* = 0x80 UMETA(DisplayName = "Custom mode"),

};

UENUM(BlueprintType)

enum class *ECombineType* : uint8

{

*CT\_AND* UMETA(DisplayName = "Task must be completed by triggering all interaction types"),

*CT\_OR* UMETA(DisplayName = "Task must be completed by triggering one of interaction types"),

*CT\_NONE* UMETA(DisplayName = "None"),

};

USTRUCT(BlueprintType)

struct *FOnIteractableActionEnded*

{

GENERATED\_BODY()

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|ActionEndParams")

*EInteractionType* *InteractionType* = *EInteractionType*::*IT\_None*;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|ActionEndParams")

*EStoryEndReason* *StoryEndReason*;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|ActionEndParams")

bool *bForceEnd* = false;

};

UENUM(BlueprintType)

enum class *EPhysicsEventType* : uint8

{

*PET\_BeginOverlap* UMETA(DisplayName = "BeginOverlap phys event"),

*PET\_EndOverlap* UMETA(DisplayName = "EndOverlap phys event"),

*PET\_None* UMETA(DisplayName = "None"),

};

USTRUCT(BlueprintType)

struct *FPhysicsTaskEvent*

{

GENERATED\_BODY()

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|Physics")

*EPhysicsEventType* *PhysicsEventType* = *EPhysicsEventType*::*PET\_None*;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|Physics")

Продовження додатку Н

UPrimitiveComponent\* *OverlappedComponent* = nullptr;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|Physics")

AActor\* OtherActor = nullptr;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|Physics")

UPrimitiveComponent\* *OtherComp* = nullptr;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|Physics")

int32 *OtherBodyIndex* = -1;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|Physics")

bool bFromSweep = false;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|Physics")

FHitResult *SweepResult*;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Interactable|Physics")

FVector *ComplexCollisionPoint* = FVector::ZeroVector;

};

UENUM(BlueprintType)

enum class ETaskState : uint8

{

*TS\_NotStarted* = 0 UMETA(DisplayName = "Not started"),

*TS\_Execution* = 1 UMETA(DisplayName = "Execution"),

*TS\_End* = 2 UMETA(DisplayName = "End"),

};

DECLARE\_DYNAMIC\_MULTICAST\_DELEGATE(FOnTaskActionEventSignature);

UCLASS(Abstract, BlueprintType, Blueprintable)

class STORYSYSTEM\_API *UTaskExecutor* : public UObject

{

GENERATED\_BODY()

public:

virtual void Initialize();

virtual void Uninitialize();

virtual void StartTask();

public:

virtual void *OnTaskInitialized*();

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "OnTaskInitialized"))

void *ReceiveOnTaskInitialized*();

virtual void *OnTaskInteractEventEnded*();

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "OnTaskInteractEventEnded"))

void *ReceiveOnTaskInteractEventEnded*();

virtual bool *IsTaskCompleted*();

Продовження додатку Н

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "IsTaskCompleted"))

bool *ReceiveIsTaskCompleted*();

virtual bool *CanInteractWithTask*();

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "CanInteractWithTask"))

bool *ReceiveCanInteractWithTask*();

virtual bool *ApplyExecutionCost*();

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "ApplyExecutionCost"))

bool *ReceiveApplyExecutionCost*();

UFUNCTION(BlueprintCallable)

virtual void *ResetTask*();

UFUNCTION(BlueprintCallable)

virtual void *TaskForceEnd*();

UFUNCTION(BlueprintCallable)

virtual AActor\* GetActor() const;

UFUNCTION(BlueprintCallable)

virtual ETaskState GetTaskState() const;

UFUNCTION(BlueprintCallable)

virtual void *TryToFinishCurrentState*();

public:

UFUNCTION(BlueprintCallable)

virtual void RegisterExecutor();

UFUNCTION(BlueprintCallable)

virtual uint8 *GetActiveFlags*() const;

UFUNCTION(BlueprintCallable)

virtual bool *IsTaskInitialized*() const;

UFUNCTION(BlueprintCallable)

virtual FName *GetTaskName*() const;

UFUNCTION(BlueprintCallable)

virtual bool *HasAnyInteractionFlags*(const TArray<*EInteractionType*>& TestTypes) const;

UFUNCTION(BlueprintCallable)

virtual *EStoryEndReason* *GetStoryEndReason*() const;

UFUNCTION(BlueprintCallable)

virtual void *SetNewStoryEndReason*(*EStoryEndReason* InStoryEndReason);

protected:

virtual bool *StartInteraction*();

virtual bool *EndInteraction*(const *FOnIteractableActionEnded*& ActionEnded);

virtual void *UpdateTaskState*();

virtual bool *UpdateTaskState\_Implementation*(ETaskState NewTaskState);

Продовження додатку Н

virtual bool *IsTaskValid*() const;

virtual void *SetupTaskFlags*();

virtual void *SetupTaskStates*();

public:

virtual void *TimeredTaskCompleteFunction*();

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "TimeredTaskCompleteFunction"))

void *ReceiveTimeredTaskCompleteFunction*();

private:

FTimerHandle *TimeredTask\_TimerHandle*;

public:

UPROPERTY(BlueprintReadWrite, Category = "StoryTasks|Delegates")

FOnTaskActionEventSignature *OnTaskStartedEvent*;

UPROPERTY(BlueprintReadWrite, Category = "StoryTasks|Delegates")

FOnTaskActionEventSignature *OnTaskEndedEvent*;

public:

virtual void *OnBeginOverlap*(const *FPhysicsTaskEvent*& PhysicsTaskEvent);

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "OnBeginOverlap"))

void *ReceiveOnBeginOverlap*(const *FPhysicsTaskEvent*& PhysicsTaskEvent);

virtual void *OnEndOverlap*(const *FPhysicsTaskEvent*& PhysicsTaskEvent);

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "OnEndOverlap"))

void *ReceiveOnEndOverlap*(const *FPhysicsTaskEvent*& PhysicsTaskEvent);

public:

UFUNCTION(BlueprintCallable)

virtual void *RequestEndConditionEvent*();

virtual void *OnConditionEvent*();

UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "OnConditionEvent"))

void *ReceiveOnConditionEvent*();

protected:

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Interactable|TaskName")

FName *InGameTaskName* = NAME\_None;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Interactable|TaskRules")

TArray<*EInteractionType*> *InteractionTypes*;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Interactable|TaskRules")

bool *bTaskCanBeReset* = false;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Interactable|TaskRules")

float *ResetCooldown* = 3.f;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Interactable|TaskRules")

bool *bUseOnlyBeginOverlapEvent* = false;

Продовження додатку Н

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Interactable|TaskRules")

bool *bUseOnlyEndOverlapEvent* = false;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Interactable|TaskRules")

float *UpdateTimerFunctionTime* = -1;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Interactable|TaskRules")

bool *bTimerDoOnceOnly* = false;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Interactable|TaskRules")

*ECombineType* *CombineType* = *ECombineType*::*CT\_NONE*;

UPROPERTY(BlueprintReadWrite, Category = "StorySystem|Item")

*EStoryEndReason* *StoryEndReason* = *EStoryEndReason*::*SER\_Finished*;

private:

UPROPERTY(SaveGame)

uint8 *ActiveFlags* = 0;

UPROPERTY(SaveGame)

uint8 *CurrentFlagsSet* = 0;

UPROPERTY(SaveGame)

bool *bTaskEnded* = false;

UPROPERTY(SaveGame)

bool *bTaskStarted* = false;

UPROPERTY(SaveGame)

bool *bTaskInitialized* = false;

UPROPERTY(SaveGame)

*EInteractionType* *LastInteractedType* = *EInteractionType*::*IT\_None*;

UPROPERTY(SaveGame)

ETaskState *CurrentTaskState* = ETaskState::*TS\_NotStarted*;

TMap<ETaskState, ETaskState> TransitionMap;

bool *bTaskWasStartedLocally* = false;

};

StorySystemComponent.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "Components/ActorComponent.h"

#include "StorySystem/SaveSubsystem/SaveSubsystemInterface.h"

#include "StorySystemComponent.generated.h"

class *UTaskExecutor*;

Продовження додатку Н

UCLASS()

class STORYSYSTEM\_API *UStorySystemComponent* : public UActorComponent, public *ISaveSubsystemInterface*

{

GENERATED\_BODY()

public:

*UStorySystemComponent*();

UFUNCTION(BlueprintCallable)

virtual bool *AddTaskToExecution*(*UTaskExecutor*\* Task, TSubclassOf<*UTaskExecutor*> TaskClass = nullptr);

UFUNCTION(BlueprintCallable)

virtual bool *DeleteTaskFromExecution*(*UTaskExecutor*\* Task, FName TaskName = NAME\_None);

UFUNCTION(BlueprintCallable)

virtual *UTaskExecutor*\* *UnregisterTaskFromExecution*(FName TaskName);

UFUNCTION(BlueprintCallable)

virtual AActor\* GetActor() const;

UFUNCTION(BlueprintCallable)

virtual bool *ResetTasks*();

UFUNCTION(BlueprintCallable)

const TArray<*UTaskExecutor*\*>& *GetActiveTasks*() const;

protected:

virtual void BeginPlay() override;

virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;

virtual void Serialize(FArchive& Ar) override;

virtual void *InitializeTask*(*UTaskExecutor*\* TaskToInit);

virtual void DeleteTask(*UTaskExecutor*\* TaskToDelete);

virtual void OnLevelDeserialized\_Implementation() override;

virtual void *CreateNewTasks*();

virtual void *CleanupAllTasks*();

protected:

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StoryComponent|Data")

TArray<TSubclassOf<*UTaskExecutor*>> *TasksToExecute*;

UPROPERTY(BlueprintReadOnly, Category = "StoryComponent|Data")

TArray<*UTaskExecutor*\*> ActiveTasks;

};

StoryInteractablewActor.h

Продовження додатку Н

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "GameFramework/Actor.h"

#include "StorySystem/SaveSubsystem/SaveSubsystemInterface.h"

#include "StoryInteractableActor.generated.h"

class *UStoryItemBase*;

class UBoxComponent;

class *UStorySystemComponent*;

struct *FPhysicsTaskEvent*;

UCLASS(Abstract)

class STORYSYSTEM\_API *AStoryInteractableActor* : public AActor, public *ISaveSubsystemInterface*

{

GENERATED\_BODY()

public:

*AStoryInteractableActor*();

protected:

virtual void BeginPlay() override;

virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;

virtual void OnLevelDeserialized\_Implementation() override;

protected:

UFUNCTION()

void *OnBeginOverlap\_Internal*(UPrimitiveComponent\* *OverlappedComponent*, AActor\* OtherActor, UPrimitiveComponent\* *OtherComp*,

int32 *OtherBodyIndex*, bool bFromSweep, const FHitResult& *SweepResult*);

UFUNCTION()

void *OnEndOverlap\_Internal*(UPrimitiveComponent\* *OverlappedComponent*, AActor\* OtherActor, UPrimitiveComponent\* *OtherComp*, int32 *OtherBodyIndex*);

virtual void *ProcessComplexCollision*(*FPhysicsTaskEvent*& Event);

protected:

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InteractableActor|Collision")

UBoxComponent\* BoxComponent;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InteractableActor|Component")

*UStorySystemComponent*\* *StorySystemComponent*;

#if WITH\_EDITORONLY\_DATA

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InteractableActor|Component")

class UStaticMeshComponent\* MeshComponent;

#endif // if WITH\_EDITORONLY\_DATA

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "InteractableActor|Collision")

bool *bUseComplexCollision* = false;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "InteractableActor|Collision")

Продовження додатку Н

float *AcceptableCollisionRadius* = 100.f;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "InteractableActor|Collision")

TArray<FVector> *ComplexCollisionPoints*;

};

StoryManagerBase.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "UObject/NoExportTypes.h"

#include "StorySystem/StorySystemType.h"

#include "StoryManagerBase.generated.h"

UENUM(BlueprintType)

enum class *ETaskDependency* : uint8

{

*TD\_Unlock* UMETA(DisplayName = "Tasks which unlock finishing the current one"),

*TD\_Requied* UMETA(DisplayName = "Tasks which required to finish to unlock the current one"),

};

USTRUCT(BlueprintType)

struct *FDependentTaskInfo*

{

GENERATED\_BODY()

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|Config")

*ETaskDependency* *TaskDependency*;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|Config")

FName TaskName;

};

class *UStoryItemBase*;

USTRUCT(BlueprintType)

struct *FStoryItemSetupConfig*

{

GENERATED\_BODY()

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|Config")

TSubclassOf<*UStoryItemBase*> *StoryItemClass*;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|Config")

bool *bMainTask* = false;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|Config")

TArray<*FDependentTaskInfo*> *DependentTasks*;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|Config")

TArray<*FDependentTaskInfo*> *UnlockTasks*;

Продовження додатку Н

};

USTRUCT(BlueprintType)

struct *FStoryItemSettings*

{

GENERATED\_BODY()

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "StorySystem|Item")

*UStoryItemBase*\* *StoryItem*;

};

UCLASS(Abstract)

class STORYSYSTEM\_API *UStoryManagerBase* : public UObject

{

GENERATED\_BODY()

public:

virtual void Initialize();

virtual void Uninitialize();

virtual void Serialize(FArchive& Ar) override;

virtual void *InitializeStarterItems*();

virtual void *OnTaskWasCompleted*(*UStoryItemBase*\* Task);

virtual TArray<*UStoryItemBase*\*> GetFirstInitedTasks() const;

UFUNCTION(BlueprintCallable)

virtual *UStoryItemBase*\* *FindItemByName*(const FName& TaskName);

UFUNCTION(BlueprintCallable)

virtual void *FindTasksByParams*(const *FStoryItemSearchParams*& StoryItemSearchParams, TArray<*UStoryItemBase*\*>& OutArray);

UFUNCTION(BlueprintCallable)

virtual bool *ActivateTask*(const FName& InTaskName);

UFUNCTION(BlueprintCallable)

virtual TArray<FName> *GetNextTasksForCurrentOne*(const FName& CurrentTask) const;

UFUNCTION(BlueprintCallable)

virtual TArray<FName> *GetRequiredTasksToUnlockTask*(const FName& TaskToUnlock) const;

public:

UPROPERTY(BlueprintReadOnly, Category = "StorySystem|Items")

FName *LastValidTask*;

protected:

virtual void *OnWorldDestructed*();

private:

void *Private\_GetTaskRelatedNames*(const TArray<*FDependentTaskInfo*>& Tasks, *ETaskDependency* *TaskDependency*, TArray<FName>& OutResult) const;

void *Private\_SubscribeToWorldDestructionEventHandler*(UWorld\* World, bool bSessionEnded, bool bCleanupResources);

Продовження додатку Н

private:

FDelegateHandle *OnWorldDestructedHandle*;

FName *LastActiveTask* = NAME\_None;

protected:

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "StorySystem|Items")

TArray<*FStoryItemSetupConfig*> *ItemsSetupArray*;

UPROPERTY(BlueprintReadOnly, Category = "StorySystem|Items")

TMap<FName, *FStoryItemSettings*> Items;

UPROPERTY(BlueprintReadOnly, SaveGame, Category = "Items")

TArray<FName> *CurrentActiveTasksName*;

UPROPERTY(BlueprintReadOnly, SaveGame, Category = "Items")

TArray<*UStoryItemBase*\*> *FirstlyInitedTasks*;

};

ActStoryManager.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "StorySystem/Managers/StoryManagerBase.h"

#include "ActStoryManager.generated.h"

UCLASS(Abstract, BlueprintType, Blueprintable)

class STORYSYSTEM\_API UActStoryManager : public *UStoryManagerBase*

{

GENERATED\_BODY()

};

SaveData.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "UObject/StrongObjectPtr.h"

#include "SaveData.generated.h"

/\*\*

\* Base data that will be saved/loaded.

\*/

USTRUCT()

struct FBaseSaveData

{

GENERATED\_BODY()

Продовження додатку Н

public:

FBaseSaveData();

virtual ~FBaseSaveData() {}

virtual bool Serialize(FArchive& Archive);

friend FArchive& operator << (FArchive& Archive, FBaseSaveData& SaveData)

{

SaveData.Serialize(Archive);

return Archive;

}

FName Name;

};

FORCEINLINE bool operator == (const FBaseSaveData& First, const FBaseSaveData& Second) { return First.Name == Second.Name; }

/\*\*

\* Object data that will be saved/loaded.

\*/

USTRUCT()

struct FObjectSaveData : public FBaseSaveData

{

GENERATED\_BODY()

public:

FObjectSaveData();

FObjectSaveData(const UObject\* Object);

virtual bool Serialize(FArchive& Archive) override;

public:

TStrongObjectPtr<UClass> Class;

TArray<uint8> RawData;

};

/\*\*

\* Actor data that will be saved/loaded.

\*/

USTRUCT()

struct FActorSaveData : public FObjectSaveData

{

GENERATED\_BODY()

public:

FActorSaveData();

FActorSaveData(const AActor\* Actor);

virtual bool Serialize(FArchive& Archive) override;

public:

TArray<FObjectSaveData> ComponentsSaveData;

Продовження додатку Н

FTransform Transform;

};

/\*\*

\* Level data that will be saved/loaded.

\*/

USTRUCT()

struct FLevelSaveData : public FBaseSaveData

{

GENERATED\_BODY()

public:

FLevelSaveData();

FLevelSaveData(const FName& LevelName);

virtual bool Serialize(FArchive& Archive) override;

public:

TArray<FActorSaveData> ActorsSaveData;

};

/\*\*

\* Global game data that will be saved/loaded.

\*/

USTRUCT()

struct FGameSaveData : public FBaseSaveData

{

GENERATED\_BODY()

public:

FGameSaveData();

virtual bool Serialize(FArchive& Archive) override;

public:

FName LevelName;

FLevelSaveData Level;

FObjectSaveData GameInstance;

FTransform StartTransform;

bool bAlreadySerialized;

};

SaveSubSystem.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "Subsystems/GameInstanceSubsystem.h"

#include "SaveData.h"

Продовження додатку Н

#include "SaveSubsystem.generated.h"

UCLASS()

class STORYSYSTEM\_API *USaveSubsystem* : public UGameInstanceSubsystem

{

GENERATED\_BODY()

public:

virtual void Initialize(FSubsystemCollectionBase& Collection) override;

virtual void Deinitialize() override;

public:

const FGameSaveData& *GetGameSaveData*() const;

UFUNCTION(BlueprintCallable, Category = "SaveSubsystem")

virtual void SaveGame();

UFUNCTION(BlueprintCallable, Category = "SaveSubsystem")

virtual void *LoadLastGame*();

UFUNCTION(BlueprintCallable, Category = "SaveSubsystem")

virtual void LoadGame(int32 SaveId);

virtual void *SerializeLevel*(const ULevel\* Level, const ULevelStreaming\* StreamingLevel = nullptr);

virtual void *DeserializeLevel*(ULevel\* Level, const ULevelStreaming\* StreamingLevel = nullptr);

protected:

virtual void *SerializeGame*();

virtual void *DeserializeGame*();

virtual void *WriteSaveToFile*();

virtual void *LoadSaveFromFile*(int32 SaveId);

virtual void OnPostLoadMapWithWorld(UWorld\* LoadedWorld);

virtual void *DeserializeActor*(AActor\* Actor, const FActorSaveData\* ActorSaveData);

virtual FString *GetSaveFilePath*(int32 SaveId) const;

virtual int32 *GetNextSaveId*() const;

virtual void OnActorSpawned(AActor\* SpawnedActor);

virtual void *NotifyActorsAndComponents*(AActor\* Actor);

protected:

FGameSaveData *GameSaveData*;

FString *SaveDirectoryName*;

TArray<int32> SaveIds;

Продовження додатку Н

FDelegateHandle *OnActorSpawnedDelegateHandle*;

bool *bUseCompressedSaves* = false;

/\*\* Used to avoid double @OnLevelDeserialized invocation \*/

bool *bIgnoreOnActorSpawnedCallback* = false;

};

SaveSubsystemInterface.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "UObject/Interface.h"

#include "SaveSubsystemInterface.generated.h"

UINTERFACE(MinimalAPI)

class *USaveSubsystemInterface* : public UInterface

{

GENERATED\_BODY()

};

class STORYSYSTEM\_API *ISaveSubsystemInterface*

{

GENERATED\_BODY()

public:

/\*\*

\* Use this method instead @AActor::BeginPlay() for logic of savable actors.

\* It will be called always after @AActor::@BeginPlay().

\*/

UFUNCTION(BlueprintCallable, BlueprintNativeEvent, Category = "SaveSubsystem")

void *OnLevelDeserialized*();

virtual void OnLevelDeserialized\_Implementation() PURE\_VIRTUAL(*ISaveSubsystemInterface*::*OnLevelDeserialized\_Implementation*, );

};

SaveSubsystemTypes.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "Serialization/ObjectAndNameAsStringProxyArchive.h"

const FName FileExtensionSave = TEXT("SAV");

struct FSaveSubsystemArchive : public FObjectAndNameAsStringProxyArchive

{

FSaveSubsystemArchive(FArchive& InInnerArchive, bool bInLoadIfFindFails);

};

Продовження додатку Н

class FSaveDirectoryVisitor : public IPlatformFile::FDirectoryVisitor

{

public:

FSaveDirectoryVisitor(TArray<int32>& InSaveIds);

virtual bool Visit(const TCHAR\* FilenameOrDirectory, bool bIsDirectory) override;

protected:

TArray<int32>& SaveIds;

};

struct BoolScopeWrapper

{

public:

BoolScopeWrapper(bool& bInValue, bool bNewValue);

virtual ~BoolScopeWrapper();

private:

bool& bValue;

bool bInitialValue;

};

SaveSubsystemUtils.h

// Bachelor's degree examination. Story system for games

#pragma once

#include "CoreMinimal.h"

#include "Kismet/BlueprintFunctionLibrary.h"

#include "SaveSubsystemUtils.generated.h"

class ULevel;

UCLASS()

class STORYSYSTEM\_API USaveSubsystemUtils : public UBlueprintFunctionLibrary

{

GENERATED\_BODY()

public:

UFUNCTION(BlueprintCallable, Category = "SaveSubsystem|Utils")

static void BroadcastOnLevelDeserialized(ULevel\* Level);

};