МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет	Інформаційних технологій
Кафедра	Інформатики і прикладного програмного забезпечення
Спеціальність	Інженерія програмного забезпечення
Форма навчання	Денна

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Капустянського Максима Ігоровича
(прізвище, ім'я, по батькові здобувача)

на тему «Розробка гри жанру стратегія на мові С++»
(повна назва теми)

за матеріалами праць провідних спеціалістів з розробки ПЗ та проектування БД
(повна назва бази дослідження)

науковий керівник к.т.н. _____ Медведєв Д.Г.
(наук. ступінь, вчене звання) *(підпис)* *(прізвище, ініціали)*

Робота допущена до захисту в ЕК
Протокол засідання кафедри
від 11.06.2025 р. № 12
Завідувач кафедри _____
(підпис)
д.т.н., професор Зеленський О.С.
Наук. ступень, вчене звання *Ініціали, прізвище*

Кривий Ріг – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет	Інформаційних технологій
Кафедра	Інформатики і прикладного програмного забезпечення
Спеціальність	Інженерія програмного забезпечення
Форма навчання	Денна

«ЗАТВЕРДЖУЮ»
Завідувач кафедри _____ Зеленський О.С.
(підпис) (Прізвище, ініціали)
« 11 » червня 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ

1. Тема роботи «Розробка гри жанру стратегія на мові С++»

Керівник роботи к.т.н. Медведєв Д.Г.
затверджені наказом закладу вищої освіти від «04» березня 2025 р. № 222-ст

2. Строк подання здобувачем роботи до «09» червня 2025 р.

3. Зміст кваліфікаційної роботи, об’єкт, предмет та мета дослідження:

Розділ 1. Постановка задачі

Розділ 2. Розробка алгоритму розв’язання задачі

Розділ 3. Організація інформаційного забезпечення

Розділ 4. Розробка програмного забезпечення

Об’єкт дослідження: гра жанру стратегія

Предмет дослідження: інформаційна підтримка гри жанру стратегія

Мета кваліфікаційної роботи: *розробка гри жанру стратегія*

5. Дата видачі завдання «04» березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів МДР	Строк виконання етапів роботи	Відмітка керівника про виконання етапів (дата, підпис)
1	Підготовка розділу 1	04.04.2025-13.04.2025	
2	Підготовка розділу 2	14.04.2025-26.04.2025	
3	Підготовка розділу 3	27.04.2025-15.05.2025	
4	Підготовка розділу 4	16.05.2025-08.06.2025	
5	Реєстрація завершеної кваліфікаційної роботи	09.06.2025	Реєстраційний № «09»червня 2025 р.
6	Отримання відгуку від наукового керівника	10.06.2025	
7	Подання кваліфікаційної роботи на перегляд завідувачу кафедри	11.06.2025	
8	Отримання зовнішньої рецензії	12.06.2025	
9	Попередній захист кваліфікаційної роботи на кафедрі	13.06.2025	
10	Підготовка до захисту в ЕК	16.06.2025-21.06.2025	

Завдання підготував науковий керівник _____ Медведєв Д.Г.
(підпис) (прізвище та ініціали)

Завдання одержав _____ Капустянський М.І.
(підпис) (прізвище та ініціали)

ЗГОДА здобувача вищої освіти
Державного університету економіки і технологій про перевірку кваліфікаційної роботи на прояви академічного плагіату та розміщення в Репозитарії Університету

Я, Капустянський Максим Ігорович (ПІП),
підтримую політику Державного університету економіки і технологій з академічної доброчесності і відкритого доступу.

Засвідчую, що кваліфікаційна бакалаврська робота «Розробка гри жанру стратегія на мові С#» виконана самостійно та не містить академічного плагіату. Я не надавав і не одержував недозволену допомогу під час підготовки цієї роботи. Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Із чинним Положенням про запобігання та виявлення академічного плагіату в роботах здобувачів вищої освіти Державного університету економіки і технологій ознайомлений. Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі порушення норм академічної доброчесності робота не допускається до захисту або оцінюється незадовільно.

Також я поінформований, що відповідно до «Положення про Репозитарій Державного університету економіки і технологій» зазначена робота буде розміщена в Електронному архіві Університету. З умовами такого розміщення ознайомлений(на).

Дата підпис ініціали, прізвище (власноруч)

АНОТАЦІЯ

на кваліфікаційну бакалаврську роботу

«Розробка гри жанру стратегія на мові С++»

Капустянського Максима Ігоровича

Кваліфікаційна бакалаврська робота на здобуття освітньо-кваліфікаційного рівня бакалавра зі спеціальності 121 «Інженерія програмного забезпечення» – Державний університет економіки і технологій – Кривий Ріг, 2025.

У бакалаврській дипломній роботі представлено докладний алгоритм розробки програмного забезпечення відеоігор у жанрі стратегія.

Проведений збір матеріалу для створення багатофункціонального програмного забезпечення відеогри за допомогою ігрового двигуну Unreal Engine 4.

Unreal Engine 4 - це набір інструментів для розробки ігор, що має широкі можливості: від створення двомірних ігор на мобільні до AAA-проектів для консолей.

На основі проведеного аналізу існуючий методів та технологій був обраний набір методів для створення відеогри у жанрі стратегія, а також варіантів для поліпшення розробки подальших проектів у цьому жанрі.

Розроблений алгоритм дозволяє оцінити якісну демонстрацію сучасних алгоритмів розробки відеоігор, а також можливості toolkit для створення інших проектів даного напрямку.

Під час розробки слід уділити увагу: механіці, ергономіці. Виходячи з вищесказаного була сформульована мета цієї роботи, а актуальність цієї теми не підлягає сумніву.

Ключові слова: Unreal Engine, програмне забезпечення, технологія, розробка, RTS, Blueprint.

ЗМІСТ

ВСТУП 7

РОЗДІЛ 1 ПОСТАНОВКА ЗАДАЧІ..... 14

1.1. Характеристика задачі 14

1.2. Огляд існуючих програмних компонентів 19

1.3. Вхідна та вихідна інформація 27

РОЗДІЛ 2 РОЗРОБКА АЛГОРИТМУ РОЗВ'ЯЗАННІ ЗАДАЧІ 30

РОЗДІЛ 3 ОРГАНІЗАЦІЯ ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ..... 37

РОЗДІЛ 4 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЗАДАЧІ..... 44

4.1. Проектування розробки програмного забезпечення 44

4.2. Розробка програмного забезпечення на мові програмування C++ з використанням движку Unreal Engine 4..... 45

ВИСНОВКИ..... 55

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ..... 57

ДОДАТКИ..... **Ошибка! Закладка не определена.**

ВСТУП

Ігрова індустрія у сучасному світі займає одне з передових місць як відносно споживача, так відносно й до розробників та виробників різноманітної продукції. Комп'ютерні ігри отримують реалістичну графіку, а для їх роботи ринок збагачується все більш та більш потужними та дорогими компонентами.

Стратегічна гра – це гра (наприклад, відео- або настільна гра), в якій невизначеність і часто автономні навички прийняття гравців мають високе значення при визначенні результату. Майже всі стратегічні ігри вимагають внутрішнього мислення в стилі рішень і, як правило, дуже високої ситуативної обізнаності.

Стратегічна відеогра – це жанр відеоігор, який фокусується на вмілому мисленні та плануванні досягнення перемоги. Він підкреслює стратегічні, тактичні, а іноді й матеріально-технічні проблеми. Багато ігор також пропонують економічні виклики та розвідку. Вони, як правило, класифікуються на чотири підтипи, залежно від того, гра покрокова чи в реальному часі, та чи гра фокусується на стратегії чи тактиці.

Стратегічні відеоігри – це жанр відеоігор, який підкреслює вміле мислення та планування досягнення перемоги. Зокрема, гравець повинен спланувати ряд дій проти одного або декількох супротивників, а скорочення сил противника зазвичай є ціллю. Перемога досягається за рахунок вищого планування, а елемент випадковості займає меншу роль. У більшості стратегічних відеоігор гравцеві надається богоподібний погляд на ігровий світ і опосередковано контролює ігрові одиниці під їх командою. Таким чином, більшість стратегічних ігор в тій чи іншій мірі включають елементи ведення війни і містять поєднання тактичних та стратегічних міркувань. Окрім боротьби, ці ігри часто ставлять під сумнів здатність гравця досліджувати економіку чи керувати нею.

Незважаючи на те, що існує багато ігрових дій, які передбачають стратегічне мислення, їх рідкої класифікують як стратегічні ігри. Стратегічна гра, як правило, має більший обсяг, і їх основний акцент робиться на здатності гравця переважати свого опонента. Стратегічні ігри рідко пов'язані з фізичним завданням і, як правило, дратують стратегічно налаштованих гравців. Порівняно з іншими жанрами, такими як екшн або пригодницькі ігри, де один гравець перехоплює багатьох ворогів, стратегічні ігри зазвичай передбачають певний рівень симетрії між сторонами. Кожна сторона, як правило, має доступ до подібних ресурсів та дій, причому сильні та слабкі сторони кожної сторони в цілому є збалансованими.

Хоча стратегічні ігри передбачають стратегічні, тактичні, а іноді й логістичні завдання, вони відрізняються від головоломки. Стратегічна гра вимагає планувати навколо конфлікту між гравцями, тоді як головоломкові ігри вимагають планування ізольовано. Стратегічні ігри також відрізняються від побудови та управління імітацією, які включають економічні виклики без будь-яких боїв. У цих іграх може виникнути певна кількість конфліктів, але вони відрізняються від стратегічних ігор, оскільки вони не підкреслюють необхідності прямого дії на опонента.

Хоча стратегічні ігри схожі на рольові відеоігри, оскільки гравець повинен керувати підрозділами з різноманітними числовими атрибутами, RPG мають тенденцію до меншої кількості унікальних символів, тоді як стратегічні ігри фокусуються на більшій кількості досить схожих одиниць.

Стратегічні ігри дають гравцям непрямий контроль над багатьма підрозділами на полі бою. Багато ігор, наприклад Globulation 2, включають інші завдання, такі як будівництво будівель.

Гравець командує своїми силами, вибираючи одиницю, як правило, клацаючи мишкою та видаючи наказ з меню. Комбінації клавіш набувають важливого значення для просунутих гравців, оскільки швидкість часто є важливим фактором. Зазвичай підрозділи можуть рухатися, атакувати, зупинятись, утримувати позицію, хоча інші стратегічні ігри пропонують

більш складні замовлення. Підрозділи можуть навіть мати спеціалізовані здібності, такі як здатність стати невидимими для інших одиниць, як правило, врівноважених здібностями, які виявляють інакше невидимі речі. Деякі стратегічні ігри навіть пропонують спеціальні підрозділи лідера, які надають бонус іншим підрозділам. Підрозділи можуть також мати можливість плавати чи літати над інакше непрохідною місцевістю або забезпечувати транспорт для інших підрозділів. Не бойові здібності часто включають можливість ремонту або спорудження інших підрозділів або будівель.

Навіть у уявних чи фантастичних конфліктах стратегічні ігри намагаються відтворити важливі тактичні ситуації протягом історії. Такі прийоми, як флангування, проведення диверсій чи розрізання ліній подачі можуть стати невід’ємними частинами управління боєм. Місцевість стає важливою частиною стратегії, оскільки одиниці можуть отримати або втратити переваги на основі ландшафту. Деякі стратегічні ігри, такі як Civilization III та Medieval 2: Total War, передбачають інші форми конфлікту, такі як дипломатія та шпигунство. Однак війна є найпоширенішою формою конфлікту, оскільки дизайнерам ігор було складно зробити ненасильницькі форми конфлікту як привабливі.

Стратегічні ігри часто пов'язані з іншими економічними проблемами. Сюди можна віднести будівництво будинків, обслуговування населення та управління ресурсами. Ігри стратегії часто використовують віконний інтерфейс для управління цими складними проблемами.

Більшість стратегічних ігор дозволяють гравцям накопичувати ресурси, які можна перетворити на одиниці або перетворити на будинки, такі як фабрики, що виробляють більше одиниць. Кількість та види ресурсів змінюються від гри до гри. У деяких іграх буде наголошено на придбанні ресурсів, розкидаючи велику кількість по всій карті, в той час як інші ігри нададуть більше уваги тому, як керувати ресурсами та застосовувати їх,

балансуючи доступність ресурсів між гравцями. У меншій мірі деякі стратегічні ігри дають гравцям фіксовану кількість одиниць на початку гри.

Стратегічні ігри часто дозволяють гравцеві витрачати ресурси на оновлення або дослідження. Деякі з цих оновлень покращують всю економіку гравця. Інші оновлення застосовуються до підрозділу або класу підрозділів і розблокують або посилюють певні бойові здібності. Іноді вдосконалення вмикаються за рахунок побудови структури, яка забезпечує більш досконалі структури. Ігри з великою кількістю оновлень часто містять технологічне дерево , яке є рядом досягнень, які гравці можуть досліджувати, щоб розблокувати нові підрозділи, будівлі та інші можливості. Дерева технологій досить великі в деяких іграх, а 4X стратегічні ігри відомі тим, що мають найбільше.

Порядок побудови – це лінійна структура управління виробництвом, дослідженнями та ресурсами, спрямована на досягнення конкретної та спеціалізованої мети. Вони аналогічні шаховим отворам, оскільки гравець матиме на увазі певний порядок гри, однак сума порядку складання, стратегія, навколо якої будується порядок збирання або навіть який порядок побудови потім використовується, залежить від майстерності, здібності та інші фактори, такі як агресивний або оборонний кожен гравець.

Зазвичай застосовується лише до певних ігор з комп'ютерною стратегією, стратегія реального часу (RTS) вказує на те, що дії в грі є безперервними, і гравцям доведеться приймати свої рішення та дії на тлі постійно мінливого ігрового стану та комп’ютера стратегія гри в реальному часі характеризується отриманням ресурсів, створенням баз, дослідженням технологій та виробництвом підрозділів. Дуже мало ігор, що не є комп'ютерними стратегіями, в режимі реального часу; один із прикладів – Icehouse.

Деякі гравці заперечують важливість стратегії в стратегічних іграх у режимі реального часу, оскільки майстерність та спритність вручну часто розглядаються як вирішальний фактор у цьому жанрі гри. За словами Трої

Даннівей, "Гравець контролює сотні одиниць, десятки будівель та безліч різних подій, які відбуваються одночасно. Є лише один гравець, і він може звертати увагу лише на одне, за один раз. Експертні гравці можуть швидко перевернути між багатьма різними завданнями, у випадкових геймерів з цим більше проблем. "Ернест Адамс іде так далеко, що дозволяє припустити, що геймплей в реальному часі заважає стратегії. "Стратегічне мислення, принаймні на арені ігрового процесу, не піддається дій в режимі реального часу".

Багато гравців стратегії стверджують, що багато ігор RTS дійсно повинні бути позначені як ігри в режимі реального часу тактичні (RTT), оскільки гра грає повністю навколо тактики, мало стратегій або навіть ніякої стратегії. Зокрема, у багатокористувацьких онлайн-іграх (MMOG чи MMO) складно було впроваджувати стратегію, оскільки стратегія передбачає певний механізм "перемоги". Ігри MMO, за своєю природою, як правило, розроблені так, щоб не закінчуватися. Тим не менш, деякі ігри намагаються "зламати код", так би мовити, справжньої стратегії MMOG в реальному часі. Один із методів, яким вони це роблять, – це зробити захист сильнішим за зброю, тим самим значно уповільнивши боротьбу і даючи можливість гравцям більш ретельно розглянути свої дії під час протистояння. Настроюванні одиниці – це ще один спосіб додавання стратегічних елементів, якщо гравці справді здатні впливати на можливості своїх підрозділів. Галузь прагне представити нових кандидатів, гідних того, щоб їх знали "думкою стратегії", а не "стратегією спритності".

У той час як Герцог Цвей вважається першою справжньою грою на RTS, визначальним назвою для жанру став Дюна II Westwood Studios, за якою слідували їхні первісні ігри Command & Conquer. Серія «Загальне знищення» (1997), серія «Blizzard's Warcraft» (1994), серія «StarCraft» (1998) та «Ансамблі Студії» «Епоха імперій» (1997) - одні з найпопулярніших ігор RTS. оскільки NukeZone можна вважати також належним до цього жанру.

Unreal Engine 4 – це найновіша ітерація Unreal Engine, створена Epic Games. Він пропонує інтерфейс редактора для побудови ігрових світів та розміщення акторів, браузери вмісту для організації та імпорту активів та систему візуального сценарію під назвою "Креслення". Він доступний як повна програма і як некомпільований вихідний код для легкої модифікації основних функцій, а також додавання користувацького програмування.

Підтримувані платформи:

1. ПК з Windows.
2. Мак.
3. iOS (iPhone / iPad).
4. Android.
5. Xbox One (із власною ліцензійною угодою).
6. PlayStation 4 (із власною ліцензійною угодою).
7. Nintendo Switch.

Кожен аспект набору інструментів Unreal Engine розроблений з легкістю створення вмісту на увазі, надаючи можливість художникам та дизайнерам будувати все, від фотореалістичних сцен до високо стилізованого контенту, все у візуальному середовищі, надаючи програмістам надзвичайно модульну, масштабовану та розширювану рамку, яка включає повний доступ до вихідного коду. Unreal Engine включає в себе величезний набір API та інструментів, розроблених для того, щоб допомогти вам використовувати фізику, що працює на базі NVIDIA PhysX, а також вбудовану багатокористувацьку мережу, штучний інтелект, інструменти для анімації, редактор кінематографів Sequencer, систему відтворення, візуальні ефекти, відеокодеки, редактори листя та місцевості, аудіодопомога, потужний сценарій, список продовжується та продовжується.

Об’єкт роботи – гра жанру стратегія.

Предмет роботи – інформаційна підтримка гри жанру стратегія.

Виходячи із цього, **мета дипломної роботи** – розробити програмне забезпечення відеогри у жанрі стратегія для освоєння навичок необхідних для подальшого працевлаштування у обраній сфері.

Виходячи з мети, завданнями роботи стануть:

1. Охарактеризувати вхідну інформацію.
2. Охарактеризувати вихідну інформацію.
3. Розробити алгоритм розв’язання задач.;
4. Охарактеризувати методи, що використовуються для розв’язання задачі.
5. Охарактеризувати створення програмно-демонстраційного комплексу на мові програмування C++ у середовищі розробки Unreal Engine 4.

РОЗДІЛ 1
ПОСТАНОВКА ЗАДАЧІ

1.1. Характеристика задачі

Для того, щоб правильно сформулювати характеристику задачі, необхідно зібрати матеріал з предметної області, і на його основі будувати програмне забезпечення, яке б відповідало меті дипломної роботи. Характеристика задачі полягає у описанні призначення задачі, підстав для їх рішення, предмету задачі, а також визначення перших вимог до розроблюваної програми, що стосуються її інтерфейсу, вхідної та вихідної інформації, звітування тощо.

Постановка задачі – це один з найбільш важливих етапів при створенні програмного забезпечення, адже від того, наскільки повно, точно і ясно визначені вимоги до розроблювального програмного забезпечення, його функцій та можливостей, багато в чому залежить якість розробки.

В даній дипломній роботі розроблятиметься програмне забезпечення для розробки гри у жанрі стратегія.

Для визначення найкращого методу створення ПЗ з описаним функціоналом, потрібно уважно вивчити сутність предмету випускної роботи, а саме стратегічної гри.

При створенні даного програмного забезпечення було поставлено основну задачу – спроектувати програмне забезпечення для розробки гри у жанрі стратегія.

Підкреслимо основні завдання:

1. Розробка загальної програмної інфраструктури.

2. Прорахунок ігрової механіки.

3. Проектування та розробка графічного інтерфейсу.

Ігровий розвиток – це мистецтво створення ігор та описує дизайн, розробку та випуск гри. Це може включати генерацію, проектування,

побудову, тестування та випуск концепції. Під час створення гри важливо подумати про механіку гри, нагороди, залучення гравців та дизайн рівня.

Розробником ігор може бути програміст, звуковий дизайнер, художник, дизайнер або багато інших ролей, доступних у цій галузі.

Розробкою ігор може займатися велика студія розвитку ігор або окрема особа. Він може бути таким маленьким або великим, як вам подобається. Поки він дозволяє гравцеві взаємодіяти з вмістом і вміє маніпулювати елементами гри, ви можете назвати це "грою". Щоб долучитися до процесу розробки ігор, не потрібно писати код. Художники можуть створювати та створювати об’єкти, тоді як розробник може зосередитись на програмуванні панелі здоров’я. Тестер може долучитися до того, що гра працює так, як очікувалося.

Основне правило для розвитку ігор – чим менша команда, тим більша роль.

У незалежних студіях для розробників є можливість взяти на себе відповідальність за звук, засоби символів та дизайн рівня.

У деяких випадках роль дизайнера та розробника буде змішуватися. Можуть витратити час на мозковий штурм та розробку дизайну персонажів, а потім прийняти остаточні концепції та втілити їх у життя в грі.

Розробники ігор мають перевагу перед дизайнерами ігор з точки зору пропозиції. Студії часто шукають великий талант програмування. «Дефіцит» хороших програмістів робить їх незамінними для студій, які прагнуть розвитку.

Концепт-художник – ця людина створює всі малюнки та ескізи необхідний для проекту, наприклад, символи, зброя, транспортні засоби та макети карт включено кілька фрагментів чого можна вважати концептуальним мистецтвом. Природно, художникам-концепціям це потрібно виявляти виняткову майстерність у створенні 2D мистецтво на основі ідей та описів.

Дизайнер рівнів – член команди повинен мати ретельний характер знання UnrealEd, а також те, що можна і неможливо зробити в ігровій карті. Рівень дизайнерам потрібно мати хороший, розуміння ігрового дизайну, наприклад який вигляд популярний в сьогоднішніх іграх і що буде найкраще виглядати у вашій конкретній ідеї. Вони також потребують гарного естетичного почуття дизайн інтер'єру для розміщення освітлення, декорування рівнів і переконання, що загальний вигляд кожної карти відповідає протягом всієї гри

Моделер – член команди перетворює 2D твори художника-концепції в 3D-об’єкти які можна імпортувати у гру. Зазвичай робота моделерів повинна бути відносно низькою в кількості полігонів, залежно від кількість фізичних деталей проекту. Моделеру повинні бути знайомі інструменти полігонального моделювання у вибраному 3D-пакеті і до цього слід бути готовим тісно співпрацьовувати з художниками-концептуалами та бути впевненими, що їх робота відповідає оригінальним конструкціям. Моделери повинні також залишатися в тісному контакті з дизайнерами рівнів, щоб вони моделювали всі необхідні статичні сітки для кожної карти.

Художник текстур – тактильний вигляд елементів гри залишаються художнику текстур, хто відповідає за команду, що здаються об'єктами проекту бути виготовленими з правильних матеріалів і мати відповідні кольори та затінення. Художники повинні мати широкі знання про 2D-програми для комп'ютерного мистецтва, такі як Photoshop, і знати, де його знайти джерела текстур реального світу.

Технік із налаштування створінь – відповідає за створення систем управління, які перетворюють персонажа з моделі у цифрову одиницю, яку потім аніматор може встановити для переміщення.

Техніки повинні мати глибоке розуміння символів та прийомів у 3D-пакеті який був вирішений використовуватися для проекту, наприклад Maya або 3ds max. Часто, особливо у менших командах, ця людина також виконує роль аніматора.

Аніматор не тільки розуміє методи анімації у вибраному програмному забезпеченні 3D-анімації, але і має ґрунтовні знання про спосіб переміщення об'єктів, особливо їх терміни. Технік із налаштування істот та аніматор тісно співпрацюють щоб не було суперечливих уявлень про те, як слід проектувати системи управління.

Програміст / кодер – член команди який знає, як працює UnrealScript і як ним користуватися, інтегрувати необхідну функціональність у гру чи проект. У більшості випадків програмісти також розуміють багато фундаментальних концепцій, які керують Unreal Engine,. Хороші програмісти також повинні зрозуміти можливості мови UnrealScript, щоб вони могли судити про труднощі або навіть можливість інтеграції нової ідеї в систему.

Керівник проекту – ця людина зазвичай не створює жодного з елементів гри, але все ж відіграє життєво важливу роль, гарантуючи, що кожна частина проекту буде завершена вчасно та будь-які внутрішні проблеми вирішуються.

Для вирішення проблем із ігровими рамками були розроблені такі інструменти, як libGDX та OpenGL. Вони допомогли розробити ігри набагато швидше і простіше, надаючи безліч заздалегідь зроблених функцій і функцій. Однак увійти в індустрію чи зрозуміти рамки для когось із непрограмістського походження було досить важко - звичайний випадок на сцені розвитку ігор.

Саме тоді були розроблені ігрові двигуни, такі як Construct, Game Maker, Unity та Unreal. Як правило, у двигуна є все, що було в рамках, але з більш привітним підходом, використовуючи графічний інтерфейс користувача (GUI) та допомагаючи в графічному розвитку гри.

У деяких випадках, таких як Game Maker та Construct, кількість попередньо виготовлених функцій настільки велика, що люди, які не мають навичок програмування, могли будувати гру з нуля, по-справжньому розширюючи сцену та роблячи розробку гри доступною майже для кожного.

Багато розробників вирішили розробити гру за допомогою двигуна розвитку ігор.

Ігрові двигуни можуть значно спростити процес створення гри та дати можливість розробникам повторно використовувати багато функціональних можливостей. Він також бере участь у візуалізації для 2D та 3D графіки, фізики та виявлення зіткнень, звуку, сценаріїв та багато іншого.

Деякі ігрові двигуни мають дуже круту криву навчання, наприклад, CryEngine або Unreal Engine. Тим не менш, інші інструменти дуже доступні для початківців, а деякі навіть не потребують вас, щоб мати можливість писати код для створення своєї гри, наприклад. Побудувати 2.

Unity Game Engine розміщується десь посередині, хоча він є для початківців дружнім, деякі популярні та комерційні ігри створені за допомогою Unity (наприклад, Overcooked, Superhot).

Ігровий движок BuildBox в основному призначений для розвитку гіперказуальних ігор.

Типові ігрові двигуни:

1. CryEngine.
2. Unreal Engine.
3. Unity Game Engine.
4. Game Maker.
5. Twine.
6. Source.
7. Frostbite.
8. Buildbox.

1.2. Огляд існуючих програмних компонентів

Ця бакалаврська дипломна робота, в основному, акцентована на теорії та практиці розробки гри жанру стратегія на C++ з використанням движку Unreal Engine 4 у реальних програмних проектах. Так сучасні ігрові двигуни можуть забезпечити приголомшливу графіку та допомогти полегшити виробництво.

Для більшості ігор ігрові двигуни можуть надати архітектуру з загальнодоступними можливостями візуалізації та доступністю швидкого графічного API, завдяки чому можливо забезпечити найбільшу візуальну вірність своїх ігор.

Ігровий движок – це основа для розробки ігор, яка підтримує та об'єднує декілька основних сфер. Можливість імпортувати мистецтво та активи, 2D та 3D, з іншого програмного забезпечення, наприклад, Maya або 3s Max або Photoshop; збирати ці активи на сцени та оточення; додавати освітлення, аудіо, спецефекти, фізику та анімацію, інтерактивність та логіку гри; і редагувати, налагоджувати та оптимізувати вміст для ваших цільових платформ.

BuildBox

Жанрова спрямованість: 2D-ігри будь-якого жанру і типу;

Платформа: Windows 7/8, OS X 10.8+, iOS, Android, OUYA;

Ліцензія: безкоштовна, pro версія коштує 2,675 $;

Мова інтерфейсу: англійська;

Мови програмування: без програмування, Drag & Drop;

Розробники: Secret Headquarters, Trey Smith, Nik Rudenko.

BuildBox – це простий конструктор ігор не вимагає знання мов програмування. Розробники конструктора стверджують що в BuildBox можна створити повноцінну 2D гру абсолютно без знань будь-яких мов програмування. Buildbox є кросплатформенних конструктором ігор, на ньому можна створювати ігри для Windows, MacOS, iOS, Android, OUYA, Amazon з

можливістю монетизації. У конструкторі BuildBox вбудований фізичний движок, що дозволяє робити гри з фізикою. На офіційному сайті є відео-уроки по роботі з конструктором.

З його допомогою абсолютно будь-який бажаючий, який не має серйозних технічних навичок, дійсно може робити дивовижні ігри без програмування і видавати їх на вибрані платформи: Mac, Windows, iPhone, iPad, Android, Amazon Fire Phone, Amazon TV, OUYA і т.п. Розробка ігор в Buildbox хіба що порівнянна зі створенням презентацій в PowerPoint. У програми відмінний інтерфейс, який інтуїтивно зрозумілий і відповідає за всі функції гри, не вимагає задіяти скриптинг і інші складні навички, що вимагають довгого вивчення

Buildbox вбудований ігровий движок Infinity Engine, який дозволяє робити гри в жанрі платформер за лічені хвилини, або гри будь-яких інших жанрів. Ви можете підключати до вашої грі різні ефекти, сліди, освітлення і багато інших Ви можете налаштовувати мета перемоги: вижити, пройти від точки до точки, вбити всіх, зібрати всі монети і багато інших У вас під рукою інтегрований редактор рівнів, система меню, редактор шрифтів, настройка параметрів і ін. В результаті ви можете монетизувати свої додатки, підключивши систему оголошень, вибравши з 9 великих гравців ринку, типу Chartboost.

При старті конструктора включається покрокова система навчання, яка допоможе зробити перші кроки на шляху до реалізації вашої власної гри. Створення гри стає легким процесом, схожим з додаванням картинок в фотоальбом. Щоб додати нового персонажа, ворога, об'єкт, платформу, декор, ефект або фон в вашу гру, ви просто закидаєте це в редактор рівнів. Далі ви переміщаєте об'єкт, дублюєте, розставляєте і налаштовуєте властивості.

На цьому конструкторі, наприклад, за 2 тижні була створена така гра як «Phases» на iOS, яка зайняла 28-е місце в ТОПі сервісу AppStore, отримавши більш 1.4 мільйонів завантажень.

Frostbite

Жанрова спрямованість: ігри будь-якого жанру і типу;

Платформа: Microsoft Windows, PlayStation 3, PlayStation 4, Xbox 360, Xbox One;

Ліцензія: пропріетарна;

Мова інтерфейсу: англійська;

Мови програмування: C++, C#;

Розробники: DICE.

Digital Illusions Creative Entertaiment – студія заснована чотирма шведами в 1992 році. Найвідоміша розробка студії - ігровий движок Frostbite Engine. Движок працює на всіх основних консолях, PC і мобільних платформах. Після успішного запуску в 2002 році серії Battlefield і тісної співпраці з Electronic Arts, поступово перейшла під її крило. DICE застосовують Frostbite Engine як у власних розробках, так і проектах інших філій Electronic Arts.

Першою грою, створеної з використанням цього движка стала Battlefield: Bad Company 2008 року. Движок був розроблений для заміни технічно застарілої технології Refractor Engine, яка використовувалася в попередніх іграх студії.

На даний момент існує п'ять версій движка - 1.0, 1.5, 2, 3, а також спеціальний варіант движка для мобільних систем - Frostbite GO.

Як уже згадувалося раніше, версій для першої модифікації Frostbite Engine існує дві - 1.0 і 1.5. На версії 1.0 вийшла всього одна гра і вона підтримувала тільки консолі xBox 360 і Playstation 3. Що гойдається API то перша його версія підтримувала DirectX 9.c і DirectX 10 / 10.1.

Тепер про найцікавіше - основною особливістю движка була обробка разрушаємості ландшафту і оточення: будівель, дерев, автомобілів. До слова про будівлі - в Battlefield: Bad Company будівлі руйнувалися в повному обсязі, а ламалися тільки його стіни тобто фундамент будівлі фактично залишався на місці.

Однією із заявлених особливостей движка була оптимізація для роботи на багатоядерних процесорах. Підтримувалося динамічне освітлення і затінення з функцією HBAO, процедурний шейдинг, різні пост-ефекти такі як HDR і depth of field. Максимальний розмір локації становить обмеження в 32 × 32 кілометри відображається площі і 4 × 4 кілометри ігрового простору. Крім цього, стверджується про те, що максимальна дистанція промальовування дозволяє побачити рівень аж до горизонту.

У движок був вбудований власний звуковий движок HDR Audio, що не вимагає використання спеціалізованих засобів, подібних EAX. Звуковий движок визначає який з звуків, будь то постріл або кроки, повинен звучати голосніше і в якому напрямку, що важливо для шутерів.

У березні 2013 року компанія DICE представила гру Battlefield 4, заявивши про те, що вона вийде на новій версії движка Frostbite - 3, а в травні 2014 року DICE, Visceral Games і EA анонсували ще одну гру на Frostbite 3 – Battlefield: Hardline. DICE не зупиняючись на досягнутому продовжують розвивати Frostbite Engine, роблячи його все більш реалістичним у всіх аспектах: картинка, фізика, звук.

Frostbite 3 може симулювати величезні і дуже деталізовані поля битв з функціональними руйнуваннями, які відбуваються як самі, так і з волі гравця. Безшовні локації і динамічне оточення, збудована за новим принципом. Раніше створювався вигляд навколишнього світу, так, наприклад, дерево розгойдується на вітрі. У разі Frostbite 3, розробнику надається інструмент для створення дерева, а фізичний движок вже сам визначає як оточення впливає на це саме дерево.

У всесвіті Frostbite все взаємодіє один з одним - вітер, хвилі і хмари все рухається у відповідності з силами природи. Наприклад, підірвавши машину, ви зможете побачити як похитнуться стоять поруч дерева. Величезні локації, в поєднанні з безліччю людей, що бігають і взаємодіючих з вами, створюють відчуття цілісної реальності.

Руйнація завжди була основною особливістю технології Frostbite, де все

спрямовано на прямий вплив ігрового процесу на оточення – тут вам і маленькі іскри з’являються при зіткненні, і величезні уламки, що розбиваються при падінні з висоти.

У Frostbite 3 безліч особливостей спрямованих на занурення в сюжет і ігровий процес. Одна з найважливіших речей – це лицьова анімація і то, як виглядають очі персонажів. Зображення персонажів в Battlefield ще ніколи не виглядало так реалістично. Застосування підповерхневого розсіювання, суть якого полягає в описі механізму поширення світла, при якому світло, проникаючи всередину напівпрозорого тіла через його поверхню, розсіюється всередині самого тіла, багато разів відбивається від частинок тіла у випадковому напрямку і на нерегулярні кути. Для зображення райдужної оболонки ока використовується Paralax анімація, що робить очі схожими на справжні.

Frostbite 3 є найпотужнішим інструментом в руках розробника, а мультиплатформеність движка включає в себе підтримку всього - від iPhone до консолей нового покоління. Движок надає розробнику все необхідне для створення ігор, дозволяючи приділити більше уваги творчій стороні, для більшого отримання задоволення від ігрового процесу (табл. 1.1).

Таблиця 1.1

Системні вимоги для FROSTBITE ENGINE 3

Назва компоненту	**Мінімальні системні вимоги**	**Ефективні системні вимоги**	**Оптимальні системні вимоги**
Процесор	AMD Athlon X4 или Core 2 Quad	AMD FX 6300 или Core i 5 2500	AMD FX 8350 или Core i7 2600
Оперативна пам'ять	4096 Мб RAM	6128 Мб RAM	8192 Мб RAM
Відео	1536 Мб	2048 Мб	3076 Мб
Відеокарта	Radeon HD 6970 или GeForce GTX 580	Radeon HD 7970 или GeForce GTX 680	Radeon HD 7990 или GeForce GTX 780 Ti
Операційні системи	Windows Vista/7/8	Windows Vista/7/8	Windows Vista/7/8

Движок CryEngine був розроблений німецькою студією Crytek для шутера Far Cry, який вийшов в 2004 році і мав величезний вплив на розвиток ігор з відкритим світом. Проект дозволяв переміщатися по величезній території без підзавантажень, заохочував вільний підхід до виконання місій, а також демонстрував приголомшливу графіку.

Незабаром після виходу Far Cry всі права на CryEngine були викуплені компанією Ubisoft, яка використовувала движок для декількох аддонів до шутера. Також він ліг в основу движка Dunia Engine, на якому були розроблені всі наступні частини серії Far Cry, і був ліцензований компанією NCSoft для MMORPG Aion: The Tower of Eternity.

Crytek тим часом зайнялася створенням движка CryEngine 2, на якому і був розроблений знаменитий Crysis (а також аддони Crysis Warhead і Crysis Wars). Подальші ітерації - CryEngine 3 (зараз належить Amazon), CryEngine (4), CryEngine V – є закономірним розвитком CryEngine 2. Втім, починаючи з 2013 року, привласнення версіями движка порядкових номерів вважається умовною, так як сама Crytek вважає за краще називати його CryEngine, без будь-яких цифр.

Ігри на движку CryEngine розробляються не тільки студією, що створила його. Спочатку його могли ліцензувати сторонні компанії за фіксовану плату, а освітні установи могли використовувати його безкоштовно, але на некомерційній основі - тільки для навчання студентів. Але починаючи з 2016 року движок і SDK (набір засобів розробки) поширюються безкоштовно для всіх бажаючих, але з умовою виплати Crytek 5% прибутку при доходах, перевищує 5000 доларів / євро (починаючи з версії 5.5, на більш ранніх версіях роялті не виплачується).

Движок відрізняється просунутими можливостями по розробці відеоігор і підтримкою самих передових технологій, включаючи DirectX 12, Vulkan API, VR, написання скриптів на C #, попіксельне освітлення в реальному часі, карти віддзеркалень, деталізовані текстури, туман, поверхні з відблисками, реалістичну фізику, просунуту анімацію і багато іншого.

CryEngine дозволяє створювати гри з майже фотореалістичною графікою. При належному умінні проекти, розроблені за його допомогою, перевершують за якістю картинки будь-які ігри на Unreal Engine 4 або Unity. До того ж, движок містить функціональний realtime renderer, що дозволяє швидко випробувати щойно створений рівень або сцену. Нарешті, обов'язково варто згадати GameSDK - інструмент, на основі якого можна швидко створювати власні ігри, використовуючи в тому числі Ассет з офіційного сайту Crytek.

Є у движка і недоліки. Так, багато розробники відзначають труднощі при роботі з ним, що виникають із-за складності збірки билда, наявності багів в редакторі, скромного вибору Ассет, обмежень для розробки мережевої гри, а також відсутності хорошою техпідтримки і активного ком'юніті, в результаті чого часто доводиться вирішувати проблеми методом проб і помилок, не маючи можливості порадитися з досвідченими колегами.

При всій своїй потужності, CryEngine досить складний в освоєнні, так що необхідно добре знати області розробки, щоб створювати з його допомогою гри.

Звичайно ж, головною і найкращою грою на движку CryEngine є Crysis - шутер, який в 2007 році показав геймерам, наскільки гарними можуть бути віртуальні світи. Правда, для максимальних налаштувань графіки були потрібні топові конфігурації ПК, але результат того вартий. Сиквели Crysis орієнтувалися вже на консолі, тому були настільки технологічними, але, тим не менш, демонстрували цілком пристойну картинку, по частині якої впевнено обходили своїх сучасників.

Продовжує список ігор ще один проект від Crytek, виконаний в жанрі розрахованого на багато користувачів хоррор-шутера з елементами «Королівської битви». Тут движок дозволив авторам реалізувати вражаюче красиві болота і ліси Луїзіани, по яких бродять жахливі чудовиська і найманці, що полюють на них заради цінних трофеїв.

Наступною грою стала action RPG від чеської студії Warhorse, яка

переносить геймерів в середньовічну Богемію, демонструючи замальовки з життя суспільства XIV століття. Гра може похвалитися відмінним сюжетом, нетиповою бойовою системою і живим світом, але також здатна неприємно здивувати слабкою оптимізацією і багами.

Шутер, що відноситься до вмираючого напрямку immersive sim, який пропонує гравцям досліджувати орбітальну станцію, атакували інопланетної формою життя. Prey дарує повну свободу у виборі варіантів проходження, майстерно розповідає історію через оточення, а також вражає захоплюючими космічними пейзажами.

Unreal Engine 4 та Unity3D

Unity – це міжплатформенний ігровий движок, розроблений Unity Technologies, вперше оголошений та випущений у червні 2005 року на Всесвітній конференції розробників Apple Inc. як ігровий двигун Mac OS X. Станом на 2018 рік, двигун був розширений для підтримки понад 25 платформ. Двигун може бути використаний для створення тривимірних, двовимірних, віртуальної реальності та ігор з доповненою реальністю, а також для моделювання та іншого досвіду. Двигун був прийнятий у галузях, що не входять до відеоігор, таких як кіно, автомобілебудування, архітектура, інженерія та будівництво.

Unreal Engine – це ігровий движок, розроблений компанією Epic Games, вперше продемонстрований у 1998 році шутер від першої особи Unreal. Хоча спочатку розроблений для шутерів від першої особи, він з успіхом застосовується в багатьох інших жанрах, включаючи платформи, бойові ігри, MMORPG та інші RPG. Завдяки своєму коду, написаному на C ++, Unreal Engine має високу ступінь портативності і є інструментом, який сьогодні застосовують багато розробників ігор, і він доступний для джерела. Найновіша версія - Unreal Engine 4, яка вийшла у 2014 році.

Unity Ігрова логіка описується з використанням середовища Mono. Ігрові скрипти маніпулюють ігровими об'єктами [GameObject]. Ігрові об'єкти можуть мати кілька ігрових скриптів або взагалі жодного.

Unreal Engine 4 Ігрова логіка описується з використанням C ++ і / або редактора креслень [Blueprint Editor]. Класи C ++ і креслення керують акторами [Actors] сцени. Креслення схожі на префаб [Prefabs] Unity. Креслення є батьківський клас, інтерфейси і будь-які компоненти, які ви додаєте через редактор креслень, а так само єдину логіку поведінки креслення. Зазвичай гру представляють набором функціональних систем написаних на C++. Наведемо основні типи даних Unity3D та Unreal Engine4 у табл. 1.2.

Таблиця 1.2

Основні типи даних

Unity3D	**Unreal Engine4**
int	int32, int24, int8
string	FString
Transform	FTransform
Quaternion	FQuat
Rotation	FRotator
Gameobject	Actor
Array	TArray

1.3. Вхідна та вихідна інформація

До вхідної інформації відносять дані, що необхідні для розв’язання аналітичних задач. Вхідна інформація є найбільш детальною і становить основу для наступної логічної та арифметичної обробки даних. До вхідної інформації може належати не лише змінна, а й умовно-постійна та постійна інформація за особливо великої ролі умовно-постійної.

Вхідною інформацією для цієї розробки є:

1. Положення камери за координатами
2. Відгук на натискання кнопок мишки
3. Відгук на натискання кнопок клавіатури
4. Положення актору та юнітів за координатами

Приклади вхідної інформації представленні у таблиці 1.3

Таблиця 1.3

Опис складових елементів об’єкта «RtsCamera»

Ім'я	**Поле**	**Тип даних поля**	**Довжина в байтах**
Швидкість камери	MoveSpeed	Float	4
Швидкість повороту	RotateSpeed	Float	4
Положення курсору	CustoY	Float	4
Приближення	CameraZoom	Float	4
Положення курсору	CustoX	Float	4
Координата краю екрана	ScreenHit	Float	4
Можливість повороту	ShifRotate	Boolean	1
Перевірка натискання	Touch	Boolean	1
Перевірка множиного вибору	Multip	Boolean	1
Додавання до відділу	AddSquad	Boolean	1
Місце будівництва	Placebuild	Boolean	1
Непідходяще місце	BadTerrain	Boolean	1

Атрибут «MoveSpeed» містить у собі данні про швидкість камери.

Атрибут «RotateSpeed» мітить у собі інформацію про швидкість повороту камери.

Атрибут «CustoY» містить у собі дані про координату положення курсору миши відносно вікна програми.

Атрибут «CameraZoom» містить у собі координати камери змінюючи котору можно приблизити або віддалит картинку.

Атрибут «CustoX» містить у собі дані про координату положення курсору миши відносно вікна програми.

Атрибут «ScreenHit» містить у собі координати вікна програми при наведенні на котрі буде змінюватися положення камери.

Атрибут «ShifRotate» містить у собі дані про можливість повороту

перспективи камери відносно ігоровго вікна.

Атрибут «Touch» містить у собі дані про натискання на ігрового актору.

Атрибут «Multip» містить у собі дані про множинний вібір акторів.

Атрибут «AddSquad» містить у собі дані про додавання обраних акторів до відділення.

Атрибут «Placebuild» містить у собі дані про можливість побудувити будівлю на данній місцевості.

Атрибут «BadTerrain» містить у собі дані про неможливість побудувити будівлю на данній місцевості.

Атрибут «CanSell» містить у собі дані про можливість продажу.

Атрибут «Selecteactor» містить у собі дані про обранного актору.

Атрибут «SelectsActors» містить у собі дані про масив обраних акторів.

Атрибут «Squad1» містить у собі інформацію про перше відділення.

Вихідною інформацією є результат виконання створених алгоритмів та візуалізація їх на моніторі, а саме:

1. Згенерована карта.
2. Створений Hud за допомогою которого виконується більша частина функціоналу програми.
3. Створенний Widjet у якості головного меню програми.

Висновки до розділу 1

Отже під час написання першого розділу бакалаврської дипломної роботи було сформульовано характеристику задачі, мету її вирішення та цілі, які воно переслідує. Була визначена структура та зміст вхідної та вихідної інформації, основні вимоги до їх оформлення, джерела, які забезпечують задачу вхідною інформацією, та особи, які мають отримати вихідну інформацію.

РОЗДІЛ 2
РОЗРОБКА АЛГОРИТМУ РОЗВ’ЯЗАННІ ЗАДАЧІ

Кожен з нас постійно зустрічається з безліччю завдань від найпростіших і добре відомих до дуже складних. Для багатьох задач існують певні правила (інструкції, розпорядження), що пояснюють виконавцю, як вирішувати дану задачу. Ці правила людина може вивчити заздалегідь або сформулювати сам у процесі виконання завдання. Такі правила прийнято називати алгоритмами. [1]

Написанню програми завжди передує розробка деякого плану розв'язання задачі. Поняття алгоритму — одне з основних у програмуванні та інформатики. Алгоритм — це певна послідовність дій, написана на зрозумілій виконавцю алгоритмічній мові і визначаюча процес переходу від вихідних даних до результату.

При створенні алгоритму ми будемо дотримуватись таких особливостей:

- алгоритм завершиться після виконання скінченної кількості кроків;
- процес, що визначається алгоритмом, розділимо на окремі елементарні етапи, кожен з яких називається кроком алгоритмічного процесу;
- кожен крок алгоритму буде точно визначений. Дії, які необхідно здійснити, будуть чітко та недвозначно визначені для кожного можливого випадку;
- алгоритм має деяку кількість вхідних даних, тобто, величин, заданих до початку його роботи або значення яких визначають під час роботи алгоритму;

У цьому визначенні вже зазначені основні властивості алгоритму. По-перше, алгоритм складається з кінцевого набору інструкцій або кроків, по-друге, кожен крок трактується виконавцем єдиним чином, що дозволяє гарантовано отримати рішення для деякого набору вхідних даних, по-третє,

алгоритм завжди зводиться до деякого перетворення вихідних даних у результат чи результати. Для машини, зрозуміло, потрібна більш чітка формалізація завдання, ніж для людини, розуміти природну мову комп'ютери поки нездатні, звідси необхідність врахування при складанні алгоритму обмеженого набору інструкцій ЕОМ.

Перед тим, як почати створювати програму, необхідно точно визначити мету її написання. Так, коли мета точно визначена, слід розглянути більш докладно методи досягнення поставленої перед програмою мети, ці методи досягнення бувають представлені в наборі певних дій, при цьому кожна дія запускається по черзі, тобто у визначеному заздалегідь заданому порядку.

Блок схеми функіоналу представлені на рис. 2.1, рис. 2.2, рис. 2.3, рис. 2.4.

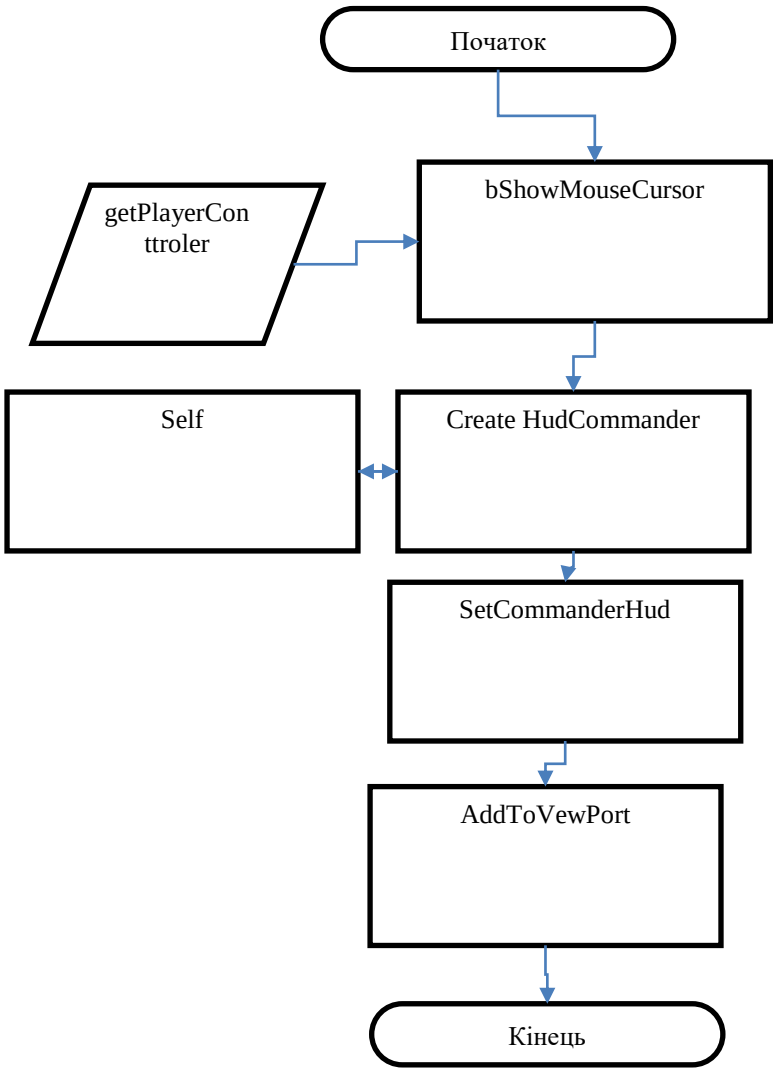

Рис. 2.1. Блок-схема створення Hud

Опис блок-схеми створення Hud.

1. «Початок».

2. «bShowMouseCurso». Встановлює чи повинен відображатися курсор миші.

3. «getPlayerConttroler». Повертає контролер гравця за вказаним індексом гравця.

4. «CreateHudCommander». Створює HUD гравця. HUD є базовим об'єктом для відображення елементів, накладених на екран. У кожного гравця, керованого людиною в грі, є свій екземпляр класу AHUD, який звертається до свого індивідуального огляду. У випадку з мультиплеерними іграми на багатопланових екранах декілька Viewports мають один і той же екран, але кожен HUD все ще звертається до власного Viewport. Тип або клас HUD для використання визначається типом, який використовується.

5. «SetCommanderHud». Записує HUD, який зараз використовується цим контролером гравця

6. «AddToVewPort». Додає до вікна перегляду гри та заповнює весь екран, якщо тільки не буде викликано SetDesiredSizeInViewport, щоб явно встановити розмір.

7. «Кінець»

Опис блок-схеми створення кулі.

1. «Початок».

2. «GetWorldLocation». Повернення місця розташування компонента у світовому просторі.

3. «RifleBarrel». Отримання скелетної сітки. Скелетні сітки часто використовуються в Unreal Engine 4 для представлення символів або інших анімаційних об'єктів. 3D-моделі, такелаж та анімація створюються у зовнішньому додатку для моделювання та анімації (3DSMax, Maya, Softimage тощо), а потім імпортуються в Unreal Engine 4 та зберігаються в пакети за допомогою браузера вмісту Unreal Editor.

4. «Make Transform». Перетворення від місця розташування, обертання та масштабу.

5. «GetActorRotation». Повертає обертання компонента кореня цього актора.

6. «SpawnActor Bullet Firev». Створюється акотор кулі.

7. «FireWeapon». Івент при виконні котрого здійснюється сигнал щодо пострілу.

8. «Кінець»

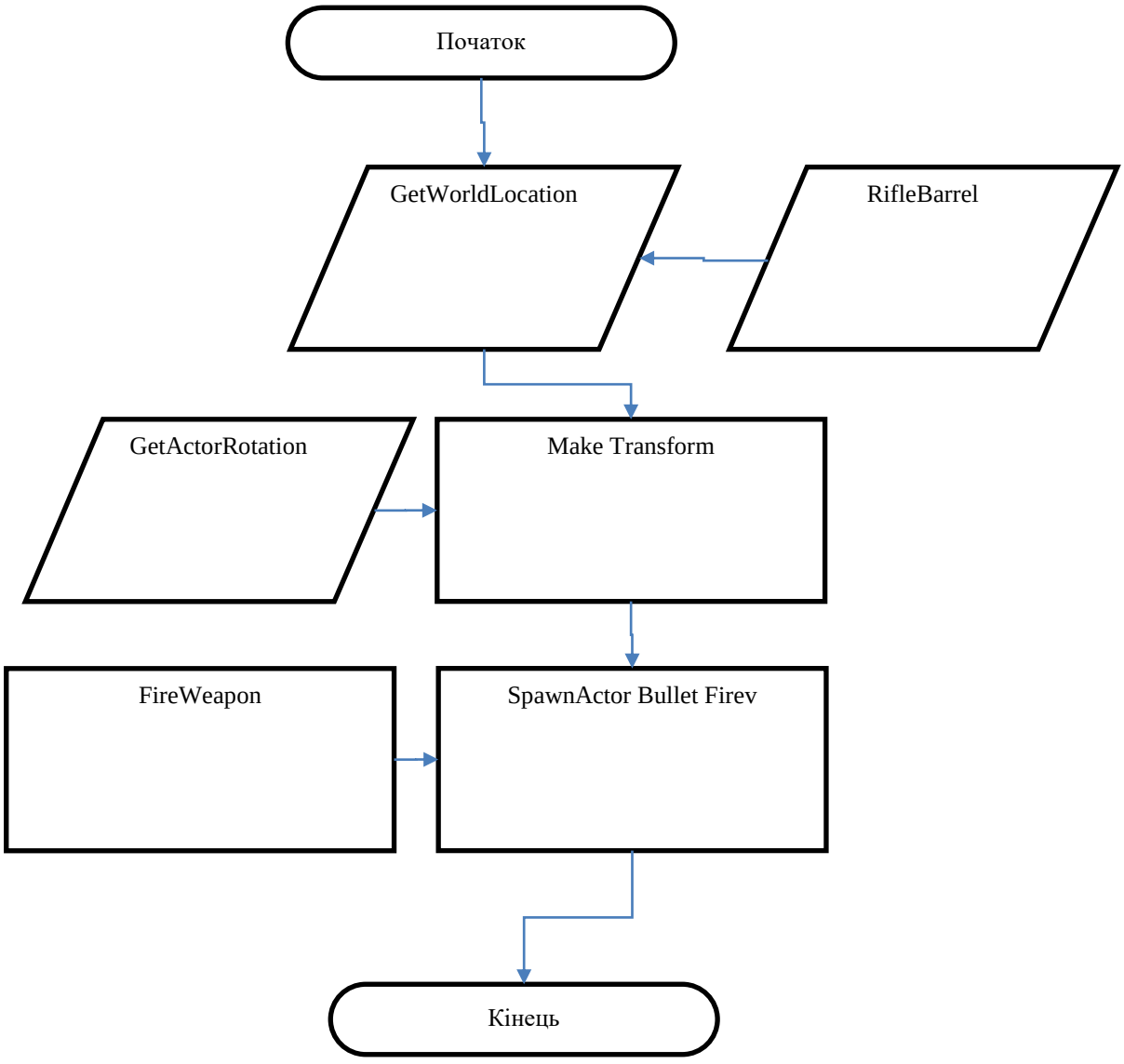

Рис. 2.2. Блок-схема створення кулі

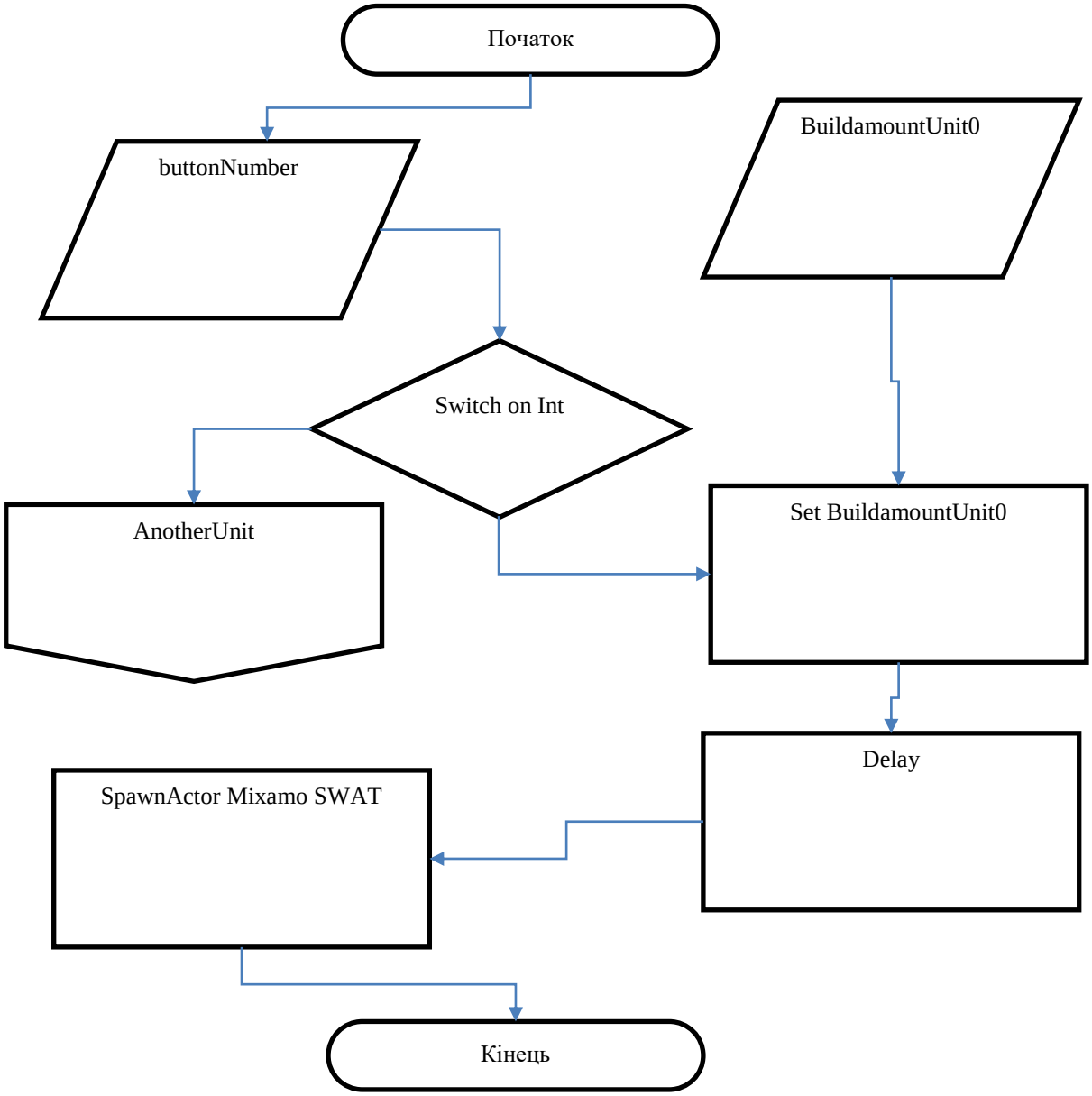

Рис. 2.3. Блок-схема створення юніту

Опис блок-схеми створення юніту.

1. «Початок». Початок розрахунку формули.

2. «buttonNumber». Отримання змінної яка з’являється під час виклику івенту натискання кнопки будівництва.

3. «Switch on Int». За отриманою змінною вирішується котрий актор буде збудований.

4. «AnotherUnit». Процес створення інших юнітів.

5. «BuildamountUnit0». Отримання данних для очереду будівництва.

6. «Set BuildamountUnit0». Додавання та збережння акторів до

очереду будівництва.

7. «Delay». Виконання дії із запізненням (вказано в секундах). Виклик ще раз під час відліку буде проігноровано.

8. «SpawnActor Mixamo SWAT». Процес створення нового екземпляра Актора відомий як spawn. Spawnакторів виконується за допомогою функції UWorld :: SpawnActor (). Ця функція створює новий екземпляр вказаного класу та повертає вказівник на щойно створений Актор. UWorld :: SpawnActor () може використовуватися лише для створення примірників класів, які успадковують клас Actor у своїй ієрархії.

9. «Кінець».

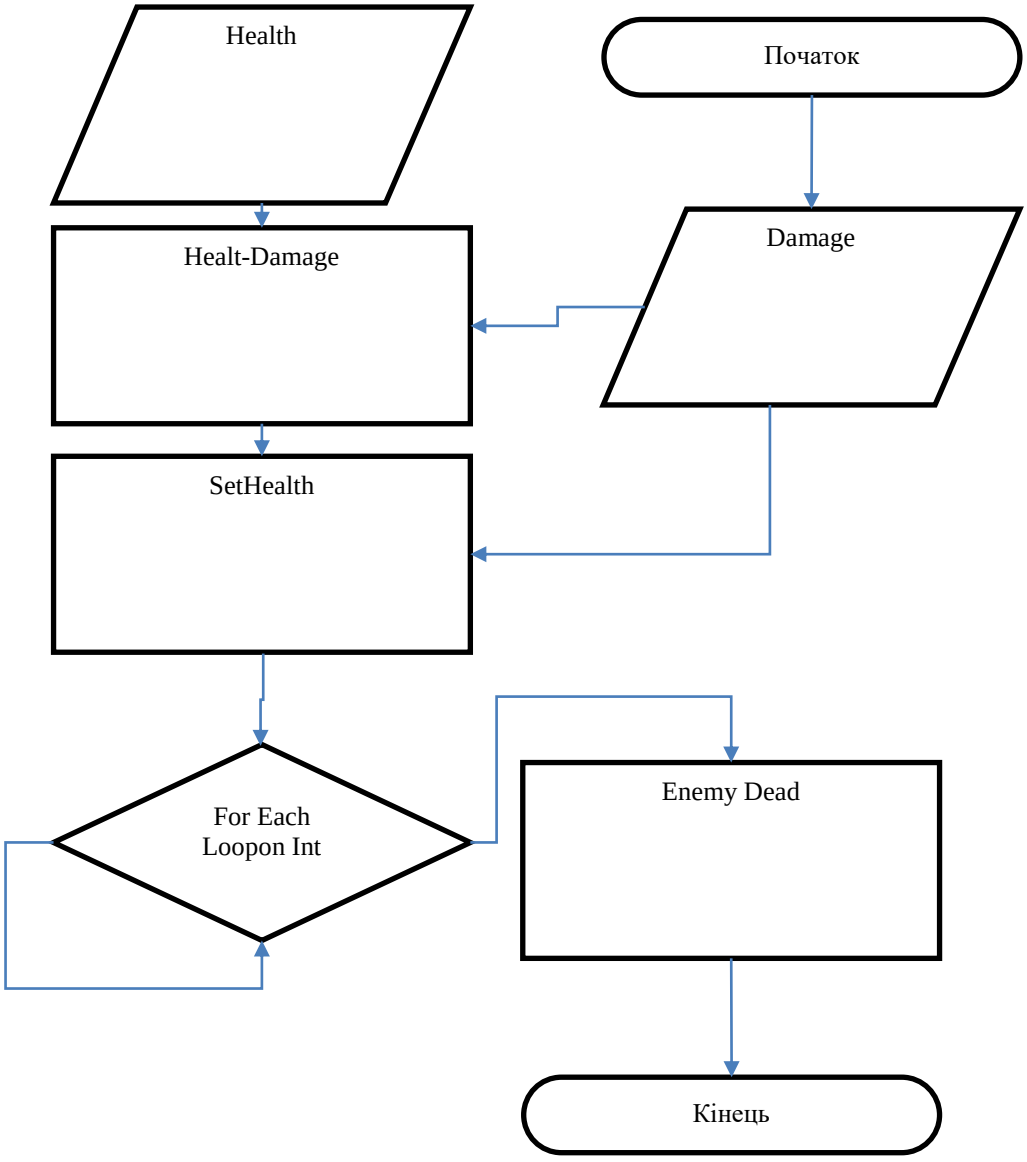

Рис. 2.4. Блок-схема розрахунку смерті актору

Опис блок-схеми розрахунку смерті актору.

1. «Початок».
2. «Damage». Отримання змінної кількості урону по актору.
3. «Health». Отримання змінної здоров’я актору по котрому проходить урон.
4. «Healt-Damage». Розрахунок залишків здоров’я актору.
5. «SetHealth». Запис змінної здоров’я актору.
6. «For Each». Цикл при котрому якщо здоров’я актору дорівнює нулю то він закінчується.
7. «Enemy Dead». Івент при здійсненні котрого актор помирає.
8. «Кінець».

Висновки до розділу 2

Отже під час написання другого розділу кваліфікаційної роботи згідно з першим розділом було сформульовано алгоритми вирішення поставлених задач та побудовано блок-схеми.

РОЗДІЛ 3
ОРГАНІЗАЦІЯ ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

Опис складових елементів об’єкту «Charecter_master» представалений в таблиці 3.1.

Таблиця 3.1

Опис об'єкту Character_ master

Ім'я	**Поле**	**Тип даних поля**	**Довжина в байтах**
Базова швидкість пострілів	BaseTurnRate	Float	4
Здоров’я	Health	Float	4
Змінна швидкості пострілів	firerate	Float	4
Перевірка пострілів	attacking	Boolean	1
Перевірка в бою юніт або ні	IsFight	Boolean	1
Перевірка чи обраний юніт	Is selected	Boolean	1
Перевірка на наявність юніту у радіусі	EnemyInsigh	Boolean	1
Вид пострілу	FireType	int	4
Актор для пострілу	EnemyTofight	actor	-
Список ворогів	EnemyList	array	-

Атрибут «BaseTurnRate» містить у собі значення за замовчуванням щодо швидкості пострілів.

Атрибут «Health» мітить у собі інформацію про залишок очок здоров’я обраного актору або будинку.

Атрибут «firerate» містить у собі дані про швидкість пострілів акторів залежно від типу актору

Атрибут «attacking» містить у собі дані про те знаходиться актор у події пострілу у його сторону.

Атрибут «IsFight» містить у собі дані про участь обраного актору гравця та противника у бойових діях.

Атрибут «Is selected» містить у собі дані те чи обраний даний актор.

Атрибут «EnemyInsigh» містить у собі дані про положення акторів та чи знаходяться вони у радіусі пострілу.

Атрибут «FireType» містить у собі дані про тип пострілу залежно від обраного типу актору.

Атрибут «EnemyTofight» містить у собі дані про актора до якого повинні початися дії атаки.

Атрибут «EnemyList» містить у собі дані про перелік усіх ворожих цілей для атаки.

Опис складових елементів об’єкта «Hud» представлений на таблиці 3.2.

Таблиця 3.2

Опис об'єкту Hud

Ім'я	**Поле**	**Тип даних поля**	**Довжина в байтах**
Кнопка будівництва першого типу юніту	Button_122	Button	-
Кнопка будівництва другого типу юніту	Button_215	Button	-
Кнопка будівництва третього типу юніту	Button_318	Button	-
Кнопка будівництва четвертого типу юніту	Button_456	Button	-
Будинок за замовчуванням	defalt	Button	-
Виклик вкладки пехоти	Infantry	Button	-
Будівництво	PowerBuildButton	Button	-
Будівництво	ResBuildButton	Button	-
Продаж	selling	Button	-
Виклик вкладки техніки	Vehiclebtn	Button	-

Атрибут «Button_122» містить у собі інформацію про створення юніту першого типу.

Атрибут «Button_215» містить у собі інформацію про створення юніту другого типу.

Атрибут «Button_318» містить у собі інформацію про створення юніту третього типу.

Атрибут «Button_456» містить у собі інформацію про створення юніту четвертого типу.

Атрибут «defalt» містить у собі дані про будівлю для будування юнитів за замовчуванням.

Атрибут «Infantry» містить у собі дані про перелік усіх юнітів піхотного типу.

Атрибут «PowerBuildButton» містить у собі дані про будівництво будинку.

Атрибут «ResBuildButton» містить у собі дані про будівництво будинку.

Атрибут «selling» містить у собі дані про продаж будинку або актору.

Атрибут «Vehiclebtn» містить у собі дані про перелік усіх акторів типу техніки.

Опис складових елементів об’єкту «RtsCamera» представлені у таблиці 3.3.

Таблиця 3.3

Опис об’єкту RtsCamera

Ім'я	**Поле**	**Тип даних поля**	**Довжина в байтах**
Швидкість камери	MoveSpeed	Float	4
Швидкість повороту	RotateSpeed	Float	4
Положення курсору	CustoY	Float	4
Приближення	CameraZoom	Float	4
Положення курсору	CustoX	Float	4
Координата краю екрана	ScreenHit	Float	4

Продовження таблиці 3.3

Можливість повороту	ShifRotate	Boolean	1
Перевірка натискання	Touch	Boolean	1
Перевірка множинного вибору	Multip	Boolean	1
Додавання до відділу	AddSquad	Boolean	1
Місце будівництва	Placebuild	Boolean	1
Непідходяще місце	BadTerrain	Boolean	1
Продаж	CanSell	Boolean	1
Обраний персонаж	Selecteactor	Actor	-
Масив персонажів	SelectsActors	Array	-
Масив першого відділення	Squad1	Array	-

Атрибут «MoveSpeed» містить у собі данні про швидкість камери.

Атрибут «RotateSpeed» мітить у собі інформацію про швидкість повороту камери.

Атрибут «CustoY» містить у собі дані про координату положення курсору мишки відносно вікна програми.

Атрибут «CameraZoom» містить у собі координати камери, змінюючи яку можна наблизити або віддалити зображення.

Атрибут «CustoX» містить у собі дані про координату положення курсору мишки відносно вікна програми.

Атрибут «ScreenHit» містить у собі координати вікна програми при наведенні на котрі буде змінюватися положення камери.

Атрибут «ShifRotate» містить у собі дані про можливість повороту перспективи камери відносно ігрового вікна.

Атрибут «Touch» містить у собі дані про натискання на ігрового актору.

Атрибут «Multip» містить у собі дані про множинний вибір акторів.

Атрибут «AddSquad» містить у собі дані про додавання обраних акторів до відділення.

Атрибут «Placebuild» містить у собі дані про можливість побудувати будівлю на даній місцевості.

Атрибут «BadTerrain» містить у собі дані про неможливість побудувати

будівлю на даній місцевості.

Атрибут «CanSell» містить у собі дані про можливість продажу.

Атрибут «Selecteactor» містить у собі дані про обраного актору.

Атрибут «SelectsActors» містить у собі дані про масив обраних акторів.

Атрибут «Squad1» містить у собі інформацію про перше відділення.

Опис складових елементів об’єкту «HudCommnader» представлений у таблиці 3.4.

Таблиця 3.4

Опис об'єкту HudCommnader

Ім'я	**Поле**	**Тип даних поля**	**Довжина в байтах**
Гравець	Commander	actor	-
Точка створення	SetSpawn	Boolean	1
Чи обраний будинок	Buildselected	Boolean	1
Обраний будинок	selectedactorbuild	actor	-
Барак за замовчуванням	Defaltinfantry	actor	-
Завод за замовчуванням	DefaltVichele	actor	-
Будинок за замовчуванням	DefaltBuilder	string	-
Змінна кнопки для створення юнітів	buttonnumber	int	4
Перевірка чи використовується будівля	BarackAvailbele	Boolean	1
Перевірка чи використовується будівля	FactoryAvailable	Boolean	1
Продаж	isSeling	Boolean	1
Текст першої позиції будівництва юнітів	B_text_0	string	-
Текст другої позиції будівництва юнітів	B_text_1	string	-
Текст третьої позиції будівництва юнітів	B_text_2	string	-
Текст четвертої позиції будівництва юнітів	B_text_3	string	-
Кількість ресурсів у текстовому варіанті	Resorse	string	-
Кількість ресурсів у числовому варіанті	ResorseInt	int	4
Місце точки створення за замовчуванням	SetDefaultSpawnPoint	string	-

Атрибут «Commander» містить у собі дані про гравця.

Атрибут «SetSpawn» мітить у собі інформацію про точку в котрій будуть створенні юніти.

Атрибут «Buildselected» містить у собі дані про обрану будівлю.

Атрибут «selectedactorbuild» містить у собі дані про актора котрого обрано для будування.

Атрибут «Defaltinfantry» містить у собі дані про будівлю у котрій будуть створюватися юніти піхотного типу.

Атрибут «DefaltVichele» містить у собі дані про будівлю у котрій будуть створюватися юніти типу техніки.

Атрибут «DefaltBuilder» містить у собі дані про будівлю за замовчуванням.

Атрибут «buttonnumber» містить у собі дані про юніт, який буде побудовано у обраній будівлі.

Атрибут «BarackAvailbele» містить у собі дані про те чи є у данний момент будівля занята створенням юніта.

Атрибут «FactoryAvailable» містить у собі дані про те чи є у данний момент будівля занята створенням юніту.

Атрибут «B_text_0» містить у собі дані про назву та тип юніту.

Атрибут «B_text_1» містить у собі дані про назву та тип юніту.

Атрибут «B_text_2» містить у собі дані про назву та тип юніту.

Атрибут «B_text_3» містить у собі дані про назву та тип юніту.

Атрибут «Resorse» містить у собі інформацію про кількість ресурсів гравця.

Атрибут «ResorseInt» містить у собі інформацію про кількість ресурсів гравця.

Атрибут «SetDefaultSpawnPoint» містить у собі інформацію про точку створення юнітів за замовчуванням.

Опис складових елементів об’єкта «Barrack» представлений у таблиці 3.5.

Таблиця 3.5

Опис об'єкту Barrack

Ім'я	Поле	Тип даних поля	Довжина в байтах
Змінна кнопки для створення юнітів	ButtonNumber	int	4
Кількість юнітів в черзі	BuildamountUnit0	int	4
Ресурси	NeVMoney	int	4

Атрибут «ButtonNumber» містить у собі дані про юніт, який буде побудовано у обраній будівлі.

Атрибут «BuildamountUnit0» мітить у собі інформацію про порядок створення юнітів.

Атрибут «NeVMoney» містить у собі інформацію про кількість ресурсів гравця які будуть витраченні на створення юнітів.

Висновки до розділу 3

Отже під час написання третього розділу дипломної роботи згідно поставлених цілей були розробленні необхідні змінні та структури інформаційних масивів.

РОЗДІЛ 4
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЗАДАЧІ

4.1. Проектування розробки програмного забезпечення

Програмне забезпечення, що являється демонстраційним комплексом для студентів спеціальності «інженерія програмного забезпечення», та представляє собою демонстрацію результатів розробки гри жанру стратегія на C++ з використанням движку Unreal Engine 4.

Дане програмне забезпечення виконано за допомогою останніх версій Unreal Engine 4 та технологіями які було додано за останні оновлення. Згідно цього, розробка та використання самої розробки як гри, потребує досить великих потужностей від техніки. А саме Windows 7 64-bit або Mac OS X 10.9.2 або старше, Чотирьох-ядерний процесор Intel або AMD, 2.5 GHz або краще, NVIDIA GeForce 470 GTX або AMD Radeon 6870 HD або краще, 8 Гб ОЗП.

Для вирішення цієї проблеми було прийняте рішення створити новий проект, на платформі персонального комп’ютера. При подальшому освоєнню даного інструментарію та поліпшення навичок оптимізації можливо створювати проекти й на інших платформах а саме:

1. ПК.
2. Xbox One.
3. PlayStation 4.
4. PlayStation Portable.
5. PlayStation Vita.
6. iOS.
7. Android.
8. Nintendo Switch.

Слід зауважити, що Unreal Engine 4 не створений для розробок відеоігор у жанрі стратегія. Тому для створення цього проекту було

переосмислений певний функціонал обраного ігрового рушію й адаптований під поставленні завдання. Також підібрані розширення для поліпшення розробки та уникнення труднощів не пов’язаних з програмуванням.

4.2. Розробка програмного забезпечення на мові програмування C++ з використанням движку Unreal Engine 4

Програмний продукт буде розроблено за допомогою інструментальних засобів мови програмування C++ та середовища розробки Unreal Engine.

C++ — універсальна мова програмування високого рівня з підтримкою декількох парадигм програмування. Зокрема: об’єктно-орієнтованої та процедурної. Розроблена Б’ярном Страуструпом (англ. Bjarne Stroustrup) в AT&T Bell Laboratories (Мюррей-Хілл, Нью-Джерсі) у 1979 році та названа «C з класами». Страуструп перейменував мову у C++ у 1983 р. Базується на мові Сі. Визначена стандартом ISO/IEC 14882:2003.

У 1990-х роках C++ стала однією з найуживаніших мов програмування загального призначення.

При створенні C++ прагнули зберегти сумісність з мовою C. Більшість програм на C справно працюватимуть і з компілятором C++. C++ має синтаксис, заснований на синтаксисі C.

Нововведеннями C++ порівняно з C є:

1. підтримка об'єктно-орієнтованого програмування через класи;
2. підтримка узагальненого програмування через шаблони;
3. доповнення до стандартної бібліотеки;
4. додаткові типи даних;
5. обробка винятків;
6. простори імен;
7. вбудовані функції;
8. перевантаження операторів;
9. перевантаження імен функцій;

10.посилання і оператори управління вільно розподіленою пам'яттю.

Для розробки гри були задіяні усі останні технлогії Unreal Engine 4, а саме Редактор Креслень Blueprints (рис. 4.1).

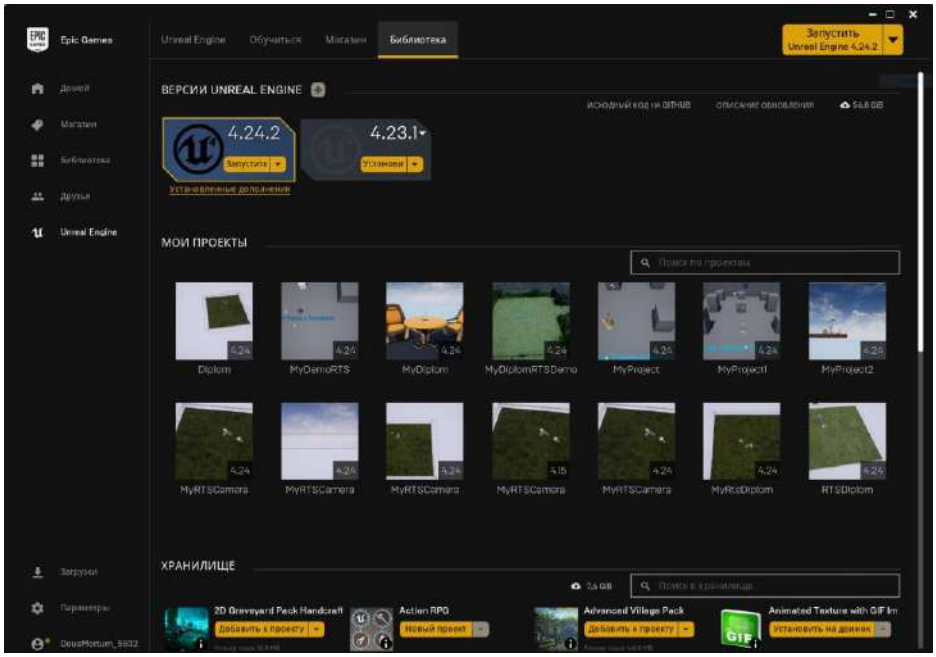

Рис. 4.1. Epic Games Launcher

Для початку роботи з Unreal Engine 4 треба встановити Epic Games Launcher. Після встановлення та запуску програми ми потрапляємо до вікна в котрому ми можемо обрати яку версію Unreal Engine встановити, перелік проектів які були завантажені або створені, а також до сховища додатків у котрому знаходяться придбані модифікації створені користувачами (рис. 4.2).

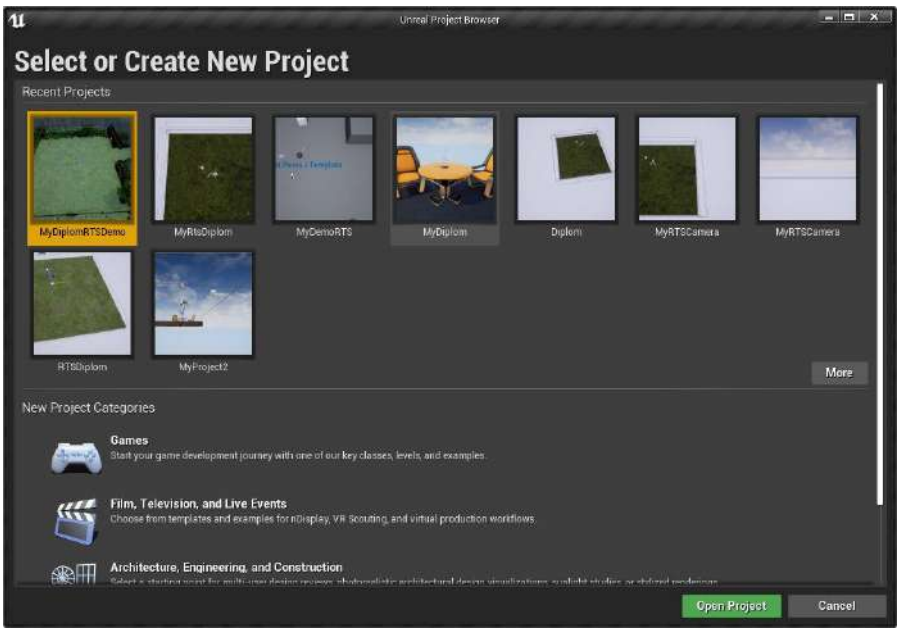

Рис. 4.2. Головне вікно Unreal Engine

Після відкриття проекту ми потрапляємо до головного вікна у котрому ми можемо відкрити існуючі проекти або створити новий (рис. 4.3).

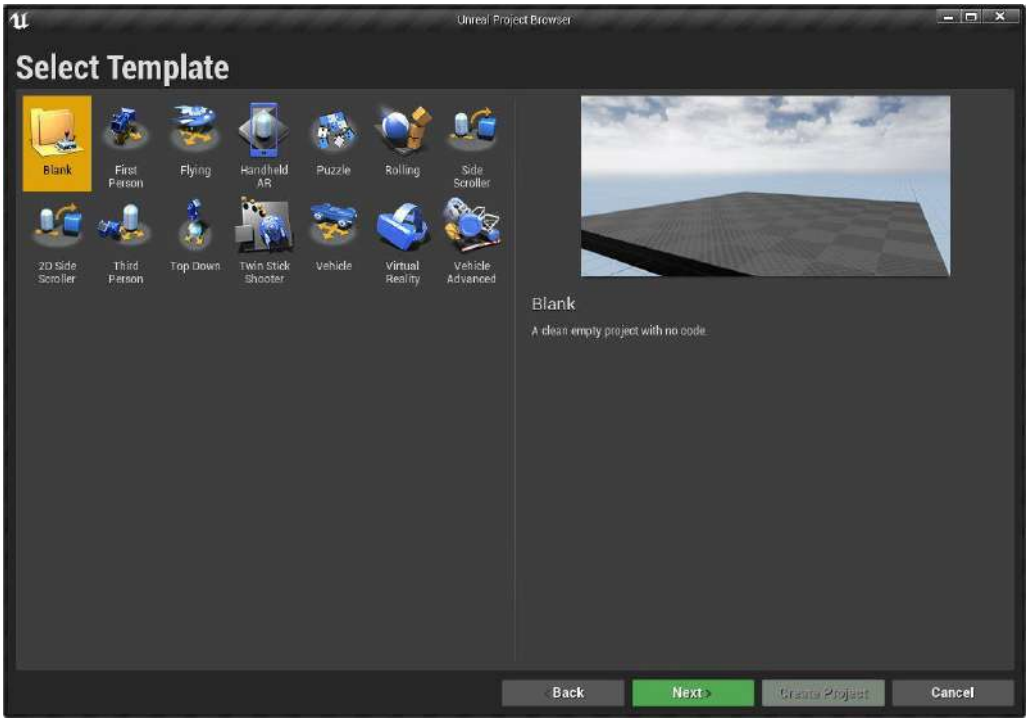

Рис. 4.3. Створення нового проекту

Почавши створювати новий проекти ми потрапляємо до темплейтів. Тут ми можемо за допомогою шаблонів створити проект з шаблонним функціоналом необхідним для обраної цілі (рис. 4.4).

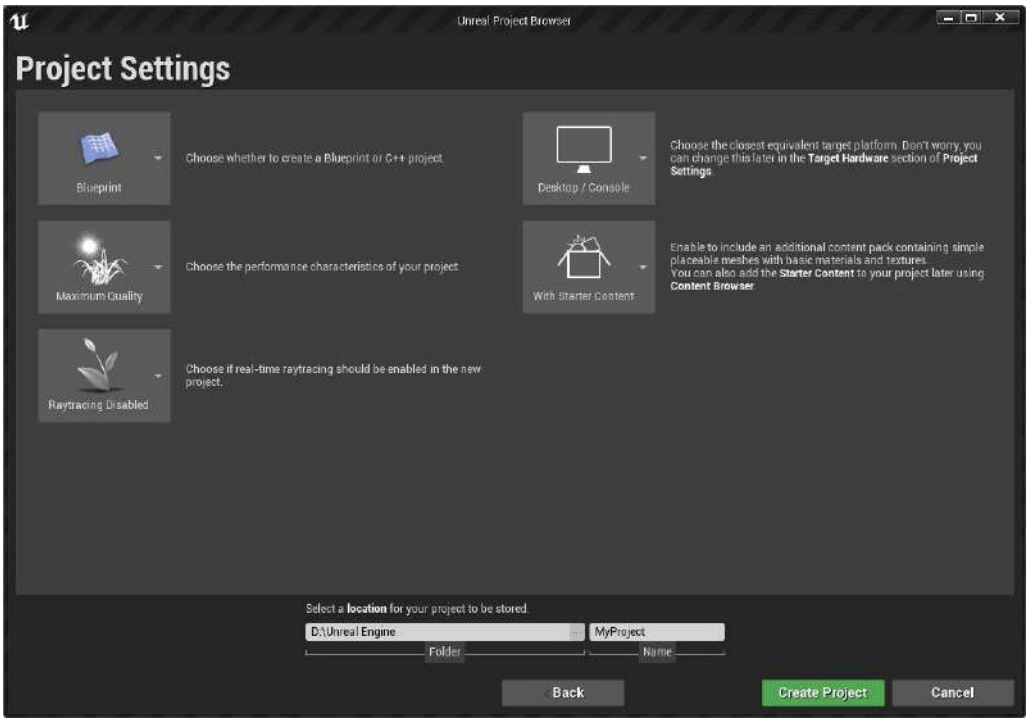

Рис. 4.4. Налаштування проекту

Після обирання шаблони ми потрапляємо до налаштування у котрих можливо обрати тип проекту те деякі графічні налаштування та базовий контент.

Target Hardware: при виборі Mobile / Tablet будуть відключені деякі ефекти постобробки. Також можна буде використовувати мишу для сенсорного введення.

Graphical Target: при виборі Scalable 3D or 2D будуть відключені деякі ефекти постобробки.

Starter Content: можна включити цю опцію, щоб додати базовий контент (Starter Content). Після створення проекту ми потропляємо до згенерованого вікна. Де ми маємио певні інструменти навігації (рис. 4.5).

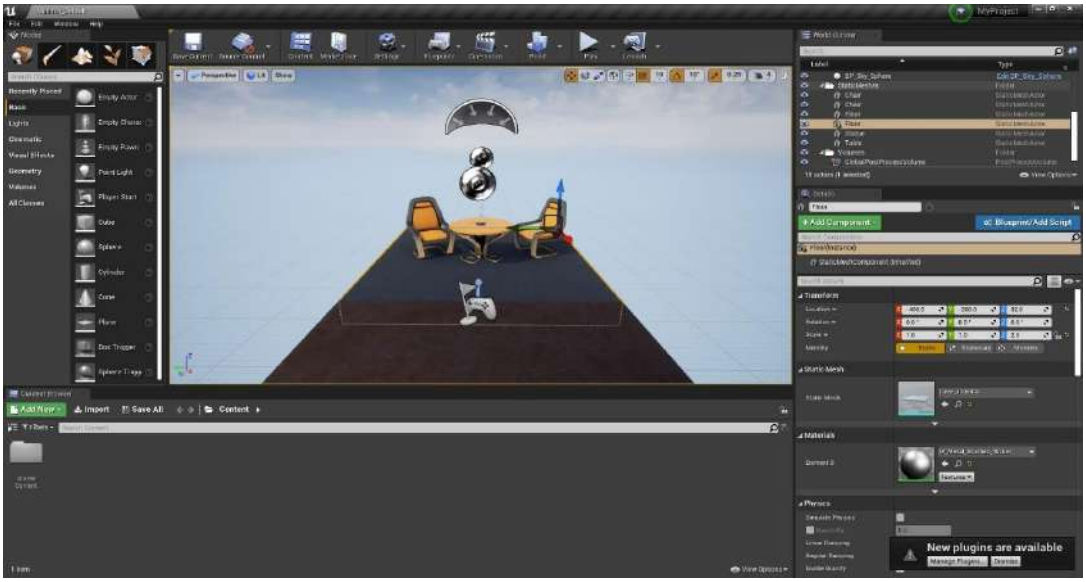

Рис. 4.5. Вікно створеного проекту

Content Browser: в цій панелі відображаються всі файли проекту. Її можна використовувати для створення папок та організовувати файли. Тут також можна виконувати пошук по файлах за допомогою пошукового рядка або фільтрів.

World Outliner: відображає всі об'єкти на поточному рівні. Ви можете впорядкувати список, розподіливши пов'язані об'єкти по папках, а також шукати і фільтрувати їх за типами.

Details: тут відображаються всі властивості вибраного об'єкту. Ця панель використовується для зміни параметрів об'єкта. Внесені зміни

вплинуть тільки на обраний екземпляр об'єкта. Наприклад, якщо в сцені є дві сфери, то при зміні розміру однієї зміни торкнуться тільки неї (рис. 4.6).

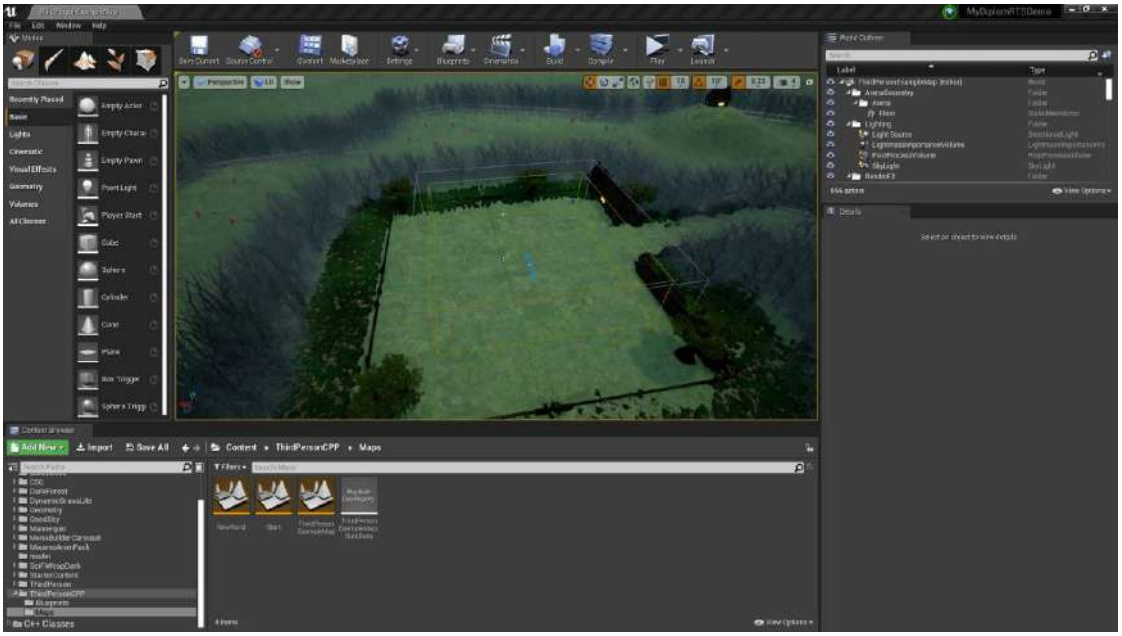

Рис. 4.6. Головне вікно проекту

Так виглядає головне вікно проекту. На дисплеї ми бачимо згенерований ландшафт та декорації. Кожен з цих об’єктів має колізії та не дозволяє будувати поряд з ними (рис. 4.7).

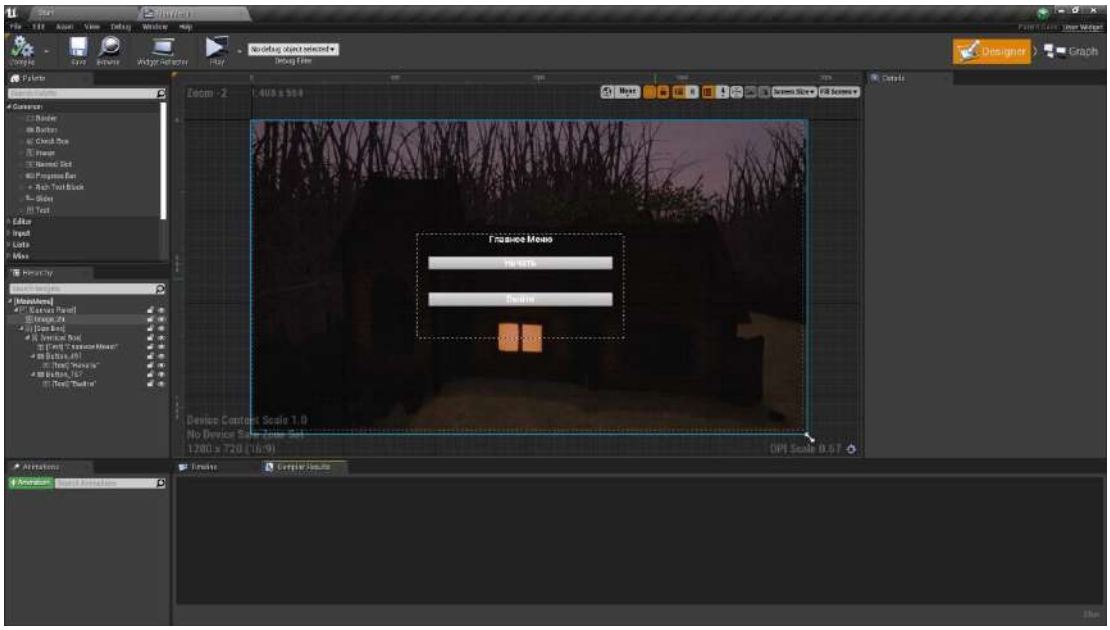

Рис. 4.7. Головне меню гри

Після початку гри з’являється данне меню. У котрому ми можимо обрати початок гри або вийти з неї (рис. 4.8).

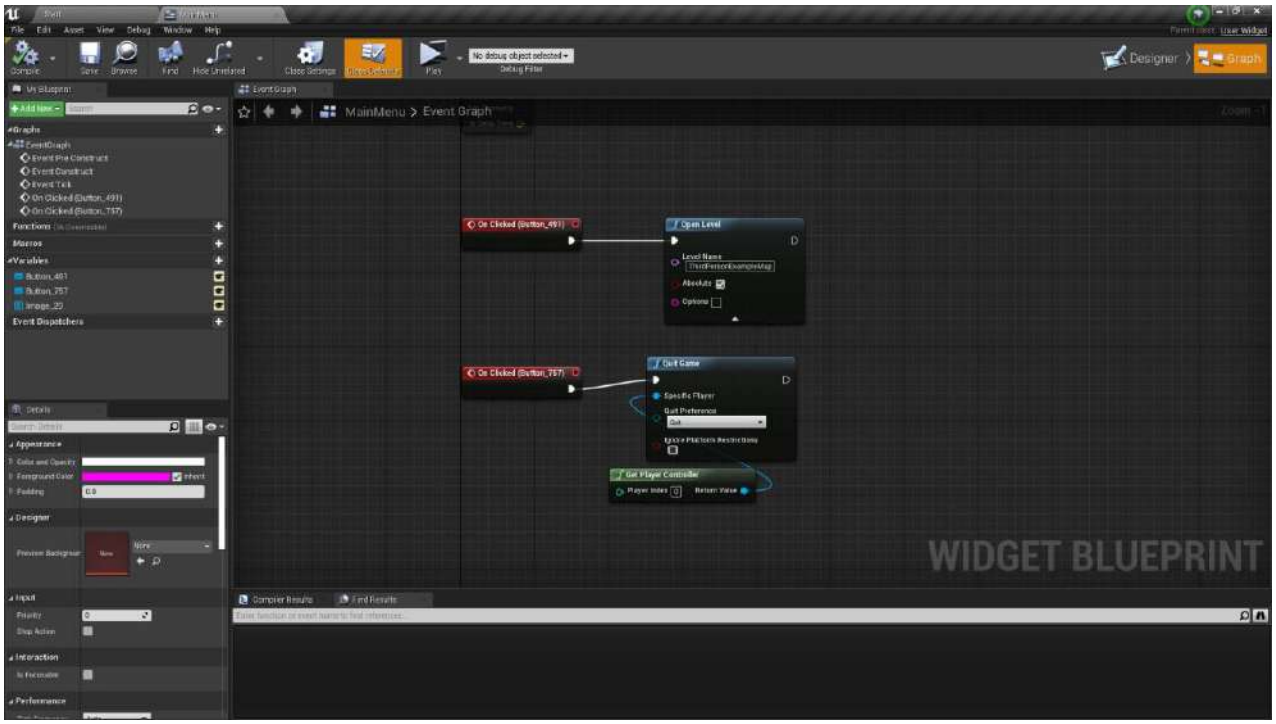

Рис. 4.8. Blueprint головного вікна

У данній схемі представлений функціонал головного меню а саме вихід та перехід до рівня початку (рис. 4.9).

Рис. 4.9. Вигляд екрану бою

Актор або юніт має певну поведінку. Наприкладі, цих акторів які ворожі до усіх, після початку гри згідно логіки розпочнуть бій між собою (рис. 4.10).

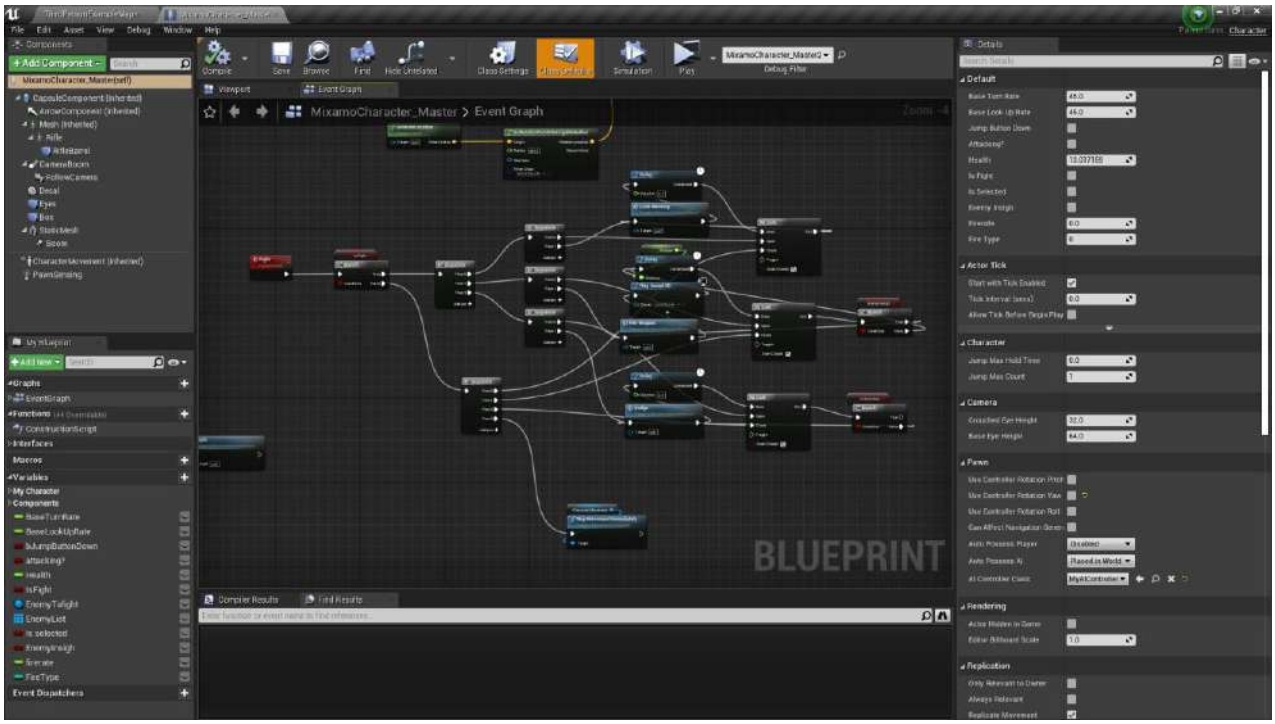

Рис. 4.10. Blueprint логіки акторів

Так як технологія Behavior tree є досить важкою для розуміння, була створена аналогова система з трьох позицій: Атаки, Знаходження ворога та привороту (рис. 4.11).

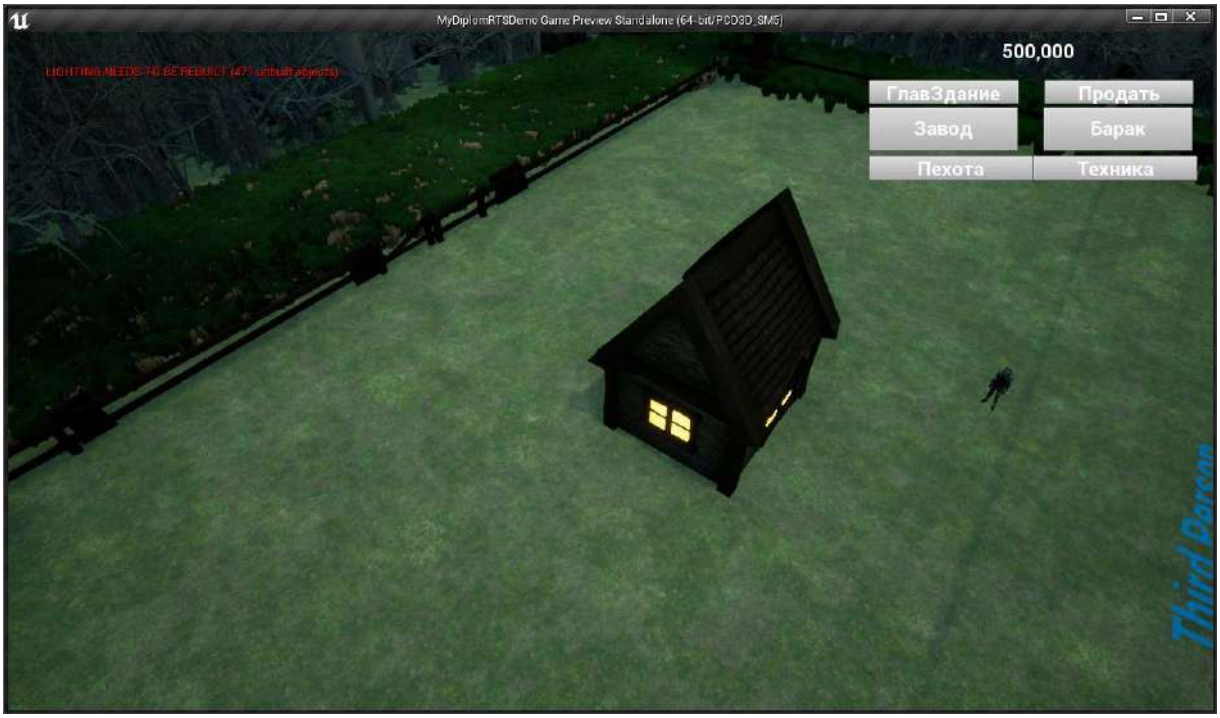

Рис. 4.11. Створення будівлі

Для створення юнітів треба спочатку побудувати будівлю. Після натискання кнопки барак можна розмістити будівлю але лише на вільній ділянці (рис. 4.12).

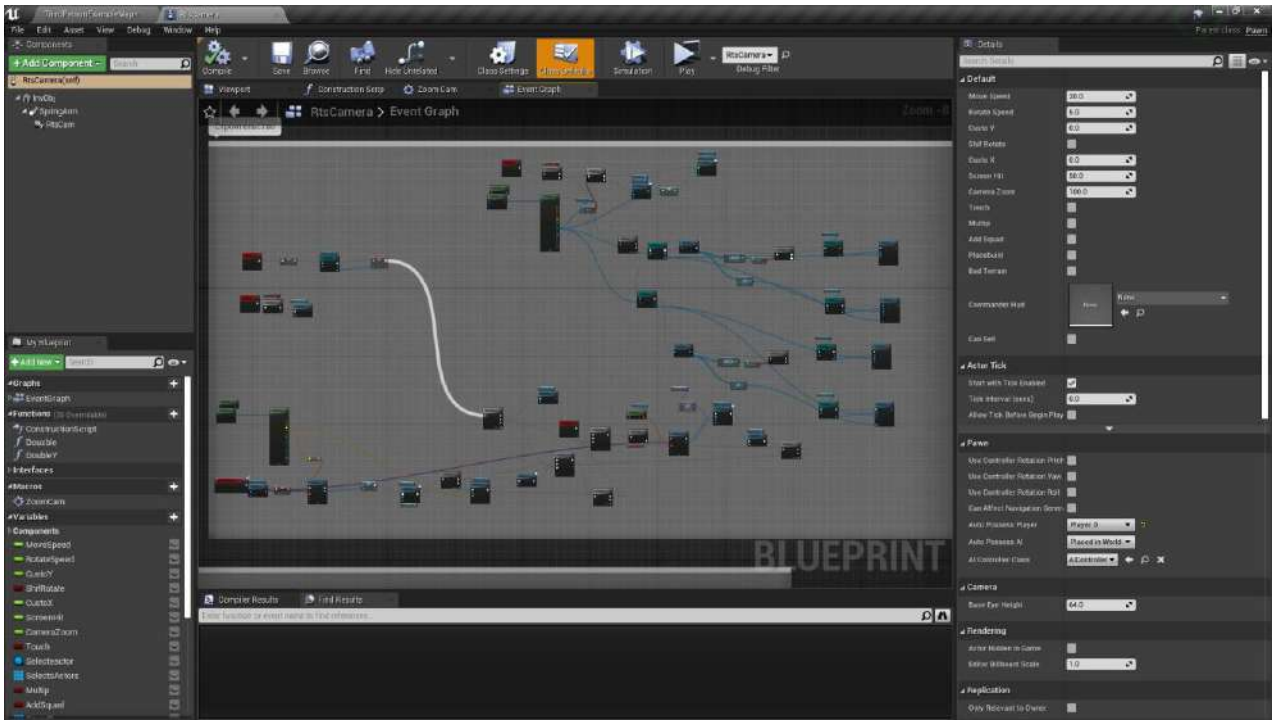

Рис. 4.12. Blueprint створення будівель

У данні схемі представлений варіант виконання будівництва, перевірка можливості будування, а тако знешкодження актору будівлі (рис. 4.13).

Рис. 4.13. Створення юнітів

Після побудування барак ми отримаємо можливість створення юнітів. На вибір є 4 типи юнітів з різними показниками здоров’я, ціни, швидкості стрільби (рис. 4.14).

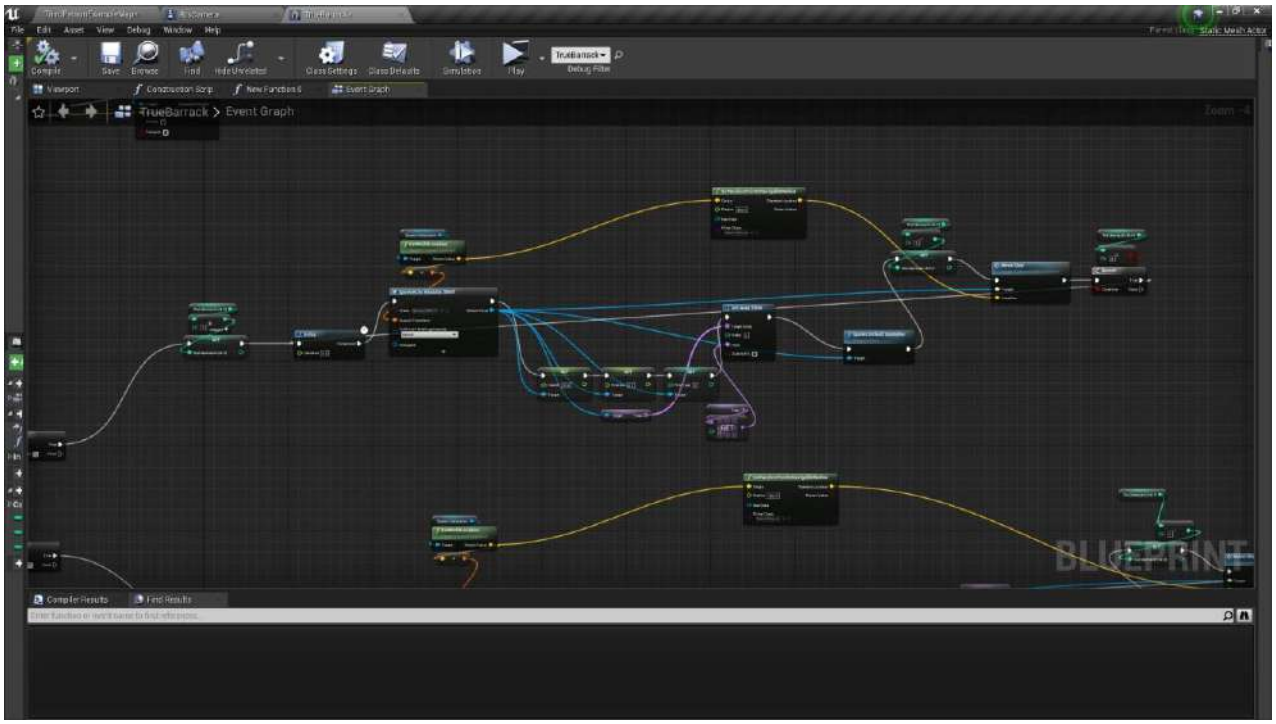

Рис. 4.14. Blueprint створеня юнітів

Так виглядаж створення юнітів за допомогою Blueprint. Від вибору юніту та його текстури до показників здоров’я та Tag номеру для системи свій-чужий (рис. 4.15).

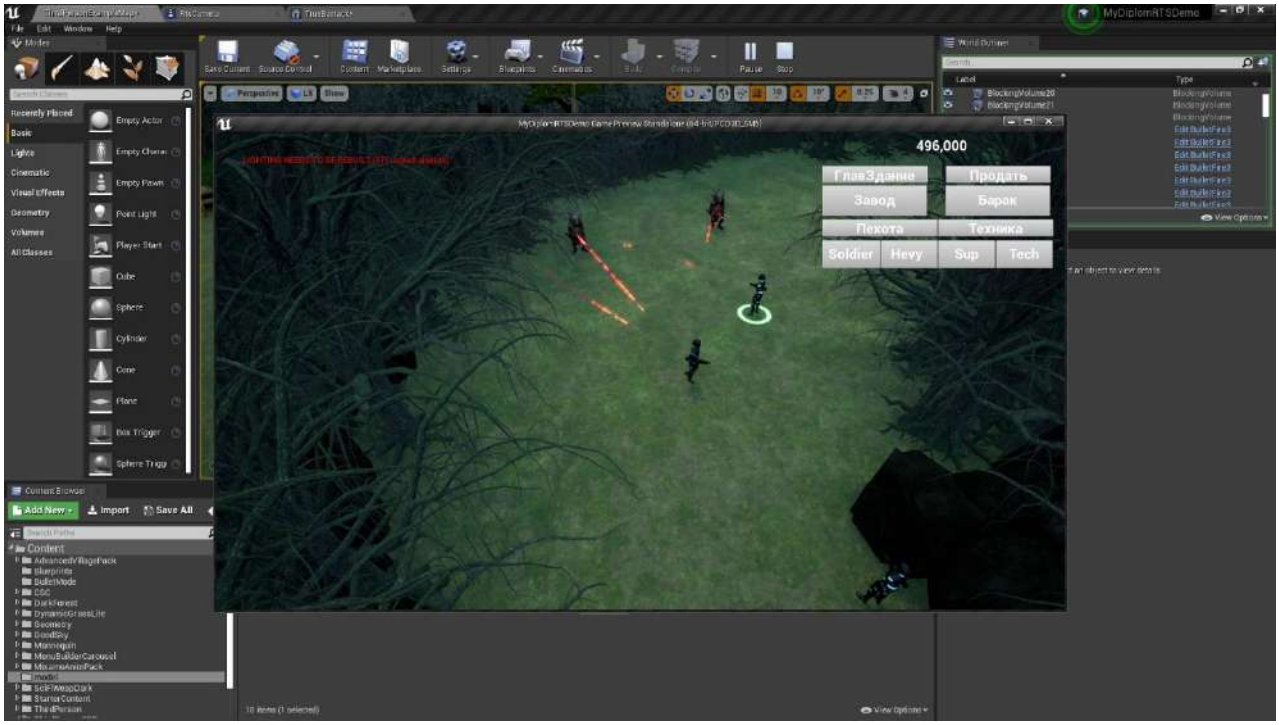

Рис. 4.15. Вигляд екрану бою

Після виконання усіх необхідних деталей, побудувавши юнітів ми відправляємо їх у бій. У котрому за виконанням логіки вони будуть робити 3

дії: атакувати противника, намагатися його побачити та спробувати зробити ухилитися від пострілів.

Висновки до розділу 4

У даному розділі наводиться розробка програмного забезпечення гри жанру стратегія на мові C++. використовуючи рушій Unreal Engine 4, розроблені скрипти для власної стратегії, які працюють при певних умовах та реагують на події.

ВИСНОВКИ

У даній бакалаврській дипломній роботі був розроблено програмний комплекс для демонстрації можливостей Unreal Engine 4 та процес освоєння данної технології.

Під час написання бакалаврської дипломної роботи були сформовані цілі та мета згідно з обраної теми. Збудовані характеристики задачі та була визначена структура та зміст вхідної й вихідної інформації, основні вимоги до їх відображення в зручному для користувача вигляді, проведені дослідження щодо даної теми у мережі інтернет, були ретельно переглянуті матеріали та програмні засоби що використовувалися під час роботи. Були отримані консультації зі провідними спеціалістами даних технології та були проведені бесіди з членами фірм котрі спеціалізуються на даному програмному забезпеченні.

Для реалізації задачі був використаний Unreal Engine 4 та мова програмування C++, а також система візуального скриптинга Blueprint. Вона є швидким способом створення прототипів ігор. Замість порядкового написання коду все можна робити візуально: перетягувати Ноди (вузли), задавати їх властивості в інтерфейсі і з'єднувати їх «проводи».

Розроблена програма виконує усі необхідні функції котрі повинна виконувати гра у жанрі стратегія, а саме:

1. Зручне керування та широкі можливості камери.
2. Будівництво будівель.
3. Створення юнітів.
4. Працюючий штучний інтелект.
5. Можливість оперувати юнітами.
6. Використання юнітів для бою.

На основі проведеної роботи, можна з успіхом розвивати та вдосконалювати розроблений програмно-демонстраційний комплекс, та додавати до нього новий функціонал та реалізацію нових алгоритмів

штучного інтелекту, мультиплеєр, більшу кількість юнітів. Або використовувати як тулкіт для нових проектів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Документація до середовища розробки Unreal Engine [Електронне джерело]. – Режим доступу: https://docs.unrealengine.com/en-US/index.html
2. Дунаев С. Доступ к базам данных и техника работы в сети. − М.: Диалог-МИФИ, 2000. − 416 с.
3. Зеленский А.С. Методические указания для самостоятельного изучения работы с базами данных на Visual C++ с использованием объектов ActiveX Data Object (ADO) по дисциплинам «Мониторинг информационных технологий», «Информационные системы в экономике», «Автоматизация проэктирования информационных систем» / Зеленский А.С., Лысенко В.С., Баран С.В. – КЭИ ГВУЗ “КНЭУ”, 2008. – 54 с.
4. Зеленский А.С. Методические указания использования объектов ADO при работе с базами данных на Visual C++ в примерах по дисциплинам: «Мониторинг информационных технологий», «Информационные системы в экономике», «Автоматизация проэктирования информационных систем» (практическая основа для выполнения лабораторних, индивидуальных, курсових и дипломних работ) / Зеленский А.С., Лысенко В.С. –КЭИ ГВУЗ “КНЭУ” – 2008. – 65 с.
5. Зеленский А.С. Методические указания к выполнению лабораторных и индивидуальных работ на основе типовых примеров разработки программного обеспечения в Visual C++ 6.0 по дисциплине «Новые информационные технологии» для магистров специальности «Экономическая кибернетика» (программа «Информационный менеджмент») / Зеленский А.С., Лысенко В.С., Баран С.В. – КЭИ ГВУЗ “КНЭУ”, 2007. – 63 с.
6. Зеленський О.С. Інструментальні засоби прикладного програмування з використанням мови Visual C++. Частина 1. [навч. посіб.] /О.С. Зеленський, В.С. Лисенко, В.Б. Хоцкіна, І.Є. Афанасьєв – КЕІ ДВНЗ "КНУ", 2013. – 295 с.

7. Зеленський О.С. Інструментальні засоби прикладного програмування з використанням мови Visual C++. Частина 2. [навч. посіб.] /О.С. Зеленський, В.С. Лисенко, В.Б. Хоцкіна, І.Є. Афанасьєв – КЕІ ДВНЗ "КНУ", 2013. – 268 с.

8. Зеленський О.С. Методичні вказівки для самостійної роботи студентів спеціальності «Економічна кібернетика» та студентів напряму підготовки «Програмна інженерія». Програмування діалогів у Visual C++ / Зеленський О.С., Лисенко В.С., Афанасьєв І.Є. – КЕІ ДВНЗ “КНЕУ”, 2010. – 49 с.

9. Зеленський О.С. Методичні вказівки до самостійного вивчення стандартних команд графічної бібліотеки OPENGL з використанням мови C++ для студентів спеціальності "Економічна кібернетика"/ Зеленський О.С., Лисенко В.С., Баран С.В. – КЕІ ДВНЗ “КНЕУ”, 2006. –41 с.

10. Зеленський О.С. Методичні вказівки до самостійного вивчення структур створення додатків в Visual C++ з дисциплін: «Об’єктно-орієнтоване програмування», «Інформаційні системи в економіці», «Інструментальні засоби прикладного програмування» / Зеленський О.С., Лисенко В.С., Афанасьєв І.Є. – КЕІ ДВНЗ “КНЕУ”, 2009. – 62 с.

11. Зеленський О.С. Об'єктно-орієнтоване програмування [навч. посіб.] /О.С. Зеленський, В.С. Лисенко – КЕІ ДВНЗ "КНЕУ імені Вадима Гетьмана", 2011. – 215 с.

12. Зеленський О.С. Основи програмування [навч. посіб.] /О.С. Зеленський, В.С. Лисенко – КЕІ ДВНЗ "КНЕУ імені Вадима Гетьмана", 2010. – 269 с.

13. Зеленський О.С. Об'єктно-орієнтоване програмування [навч. посіб.] /О.С. Зеленський, В.С. Лисенко // КЕІ ДВНЗ "КНЕУ імені Вадима Гетьмана". – 2011. – 215 с.

14. Зеленський О.С. Основи програмування [навч. посіб.] /О.С. Зеленський, В.С. Лисенко // КЕІ ДВНЗ "КНЕУ імені Вадима Гетьмана". – 2010. – 269 с.

15. Краснов М.В. OpenGL. Графика в проектах Delphi./ Краснов М.В. – СПб. : БХВ-Петербург, 2004. – 352 с.

16. Маслов В.В. Основы программирования на языке Java.- М.: Горячая линия -Телеком, 2000. − 132 с.

17. Морган М. Java 2. Руководство разработчика. − М.: Издатель- ский дом "Вильямс", 2000. − 720 с.

18. Опис використання C++ у Unreal Engine від автора Reuben Ward [Електронне джерело]. – Режим доступу: https://www.youtube.com/watch?v=3JpNil0_gm4

19. Опис використання системи Blueprint у Unreal Engine [Електронне джерело]. – Режим доступу: https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html

20. Опис створення штучного інтелекту в Unreal Engine [Електронне джерело]. – Режим доступу: https://docs.unrealengine.com/en-US/Gameplay/AI/index.html

21. Опис як робити переміщення персонажу в Unreal Engine 4 від автора Awesome Tuts [Електронне джерело]. – Режим доступу: https://www.youtube.com/watch ?v=a5gA7q5GD7Q

22. Роджерс Д. Математические основы машинной графики:/ Роджерс Д., Адамс Дж. – Пер. с англ.– М.: Мир, 2001. – 604 с.

ДОДАТКИ

Додаток А

Файл Game Mode.h

```
#pragma once

#include "GameFramework/GameMode.h"
#include "RTSGameMode.generated.h"

/**
 *
 */
UCLASS()
class RTS_API ARTSGameMode : public AGameMode
{
   GENERATED_BODY()

      ARTSGameMode();
}
```

Файл Game Mode.cpp

```
#include "RTS.h"

#include "RTSGameMode.h"

#include "RTSPlayer/RTSPlayerController.h"
#include "RTSPlayer/RTSPlayerCameraSpectatorPawn.h"

ARTSGameMode::ARTSGameMode()
{
   // C++ classes
   PlayerControllerClass = ARTSPlayerController::StaticClass();
   DefaultPawnClass = ARTSPlayerCameraSpectatorPawn::StaticClass();
}
```

Продовження додатку А

Файл RTSPlayerController .h

```
#pragma once

#include "GameFramework/PlayerController.h"
#include "RTSPlayerController.generated.h"

/**
 *
 */
UCLASS()
class RTS_API ARTSPlayerController : public APlayerController
{
   GENERATED_BODY()

      ARTSPlayerController();

};
```

Файл RTSPlayerController .cpp

```
#include "RTS.h"
#include "RTSPlayerController.h"


ARTSPlayerController::ARTSPlayerController()
{
   bShowMouseCursor = true;
   bEnableClickEvents = true;
   bEnableTouchEvents = true;
}
```

Додаток Б

Файл RTSPlayerCameraSpectatorPawn .h

```
#pragma once

#include "GameFramework/SpectatorPawn.h"
#include "RTSPlayerCameraSpectatorPawn.generated.h"

/**
 * this is the default RTS camera handling movememnt, rotation, and zoom
 */
UCLASS()
class RTS_API ARTSPlayerCameraSpectatorPawn : public ASpectatorPawn
{
   GENERATED_BODY()

public:

   //------------------------------------

   /** Constructor */
   ARTSPlayerCameraSpectatorPawn(const FObjectInitializer& ObjectInitializer);

   //------------------------------------

   /** Camera Component */
   UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera)
   class UCameraComponent* CameraComponent;

   //------------------------------------

   //** Camera XY limit */
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
   float CameraXYLimit;
```

```
//** Camera height over terrain */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
float CameraHeight;

//** Camera min height over terrain */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
float CameraHeightMin;

//** Camera max height over terrain */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
float CameraHeightMax;

/** Camera Rotation around Axis Z */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
float CameraZAnlge;

/** Camera Height Angle */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
float CameraHeightAngle;

/** Camera Pitch Angle Max */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
float CameraHeightAngleMax;

/** Camera Pitch Angle Min */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
float CameraHeightAngleMin;

/** Camera Radius (Distance) From Pawn Position */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
float CameraRadius;

/** Camera Radius Max */
```

```
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
    float CameraRadiusMax;

    /** Camera Radius Min */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
    float CameraRadiusMin;

    /** Camera Zoom Speed */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
    float CameraZoomSpeed;

    /** Camera Rotation Speed */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
    float CameraRotationSpeed;

    /** Camera Movement Speed */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
    float CameraMovementSpeed;

    /** Camera Scroll Boundary */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
    float CameraScrollBoundary;

    /** Should the camera move? */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera)
    bool bCanMoveCamera;

    //------------------------------------

private:

    /** Sets up player inputs
    *    @param InputComponent - Input Component
```

```
    */
    void SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent);

    //-----------------------------------

public:

    /** Zooms In The Camera */
    void ZoomInByWheel();

    /** Zooms Out The Camera */
    void ZoomOutByWheel();

    /** Rotate The Camera Left */
    void RotateLeftByWheel();

    /** Rotate The Camera Right */
    void RotateRightByWheel();

    /** Rotate The Camera Up */
    void RotateUpByWheel();

    /** Rotate The Camera Down */
    void RotateDownByWheel();

    //---

    /** Calculates the new Location and Rotation of The Camera */
    void RepositionCamera();

    //-----------------------------------

private:
```

Продовження додатку Б

```
        // set them to +/-1 to get player input from keyboard
        float FastMoveValue;                    // movement speed multiplier : 1 if
shift unpressed, 2 is pressed
        float RotateValue;                      // turn instead of move camera

        float MoveForwardValue;
        float MoveRightValue;
        float MoveUpValue;
        float ZoomInValue;

        //---

    public:

        /** Left or Right Shift is pressed
        * @param direcation - (1.0 for Right, -1.0 for Left)
        */
        void FastMoveInput(float Direction);

        /** Left or Right Ctrl is pressed
        * @param direcation - (1.0 for Right, -1.0 for Left)
        */
        void RotateInput(float Direction);

        /** Input recieved to move the camera forward
        * @param direcation - (1.0 for forward, -1.0 for backward)
        */
        void MoveCameraForwardInput(float Direction);

        /** Input recieved to move the camera right
        * @param direcation - (1.0 for right, -1.0 for left)
        */
        void MoveCameraRightInput(float Direction);
```

```
    /** Input recieved to move the camera right
    * @param direcation - (1.0 for right, -1.0 for left)
    */
    void MoveCameraUpInput(float Direction);

    /** Input recieved to move the camera right
    * @param direcation - (1.0 for right, -1.0 for left)
    */
    void ZoomCameraInInput(float Direction);

    //---

private:

    /** Moves the camera forward
    * @param direcation - (+ forward, - backward)
    */
    FVector MoveCameraForward(float Direction);

    /** Moves the camera right
    * @param direcation - (+ right, - left)
    */
    FVector MoveCameraRight(float Direction);

    /** Gets the roatation of the camera with only the yaw value
    * @return - returns a rotator that is (0, yaw, 0) of the Camera
    */
    FRotator GetIsolatedCameraYaw();

    //---

    /** Moves the camera up/down
    * @param direcation - (+ up, - down)
```

Продовження додатку Б

```
	*/
	float MoveCameraUp(float Direction);

	//---

	/** Zooms the camera in/out
	* @param direcation - (+ in, - out)
	*/
	void ZoomCameraIn(float Direction);

	/** Turns the camera up/down
	* @param direcation - (+ up, - down)
	*/
	void TurnCameraUp(float Direction);

	/** Turns the camera right/left
	* @param direcation - (+ right, - left)
	*/
	void TurnCameraRight(float Direction);

	//------------------------------------

public:

	/** Tick Function, handles keyboard inputs */
	virtual void Tick(float DeltaSeconds) override;

	//------------------------------------

	// detect landscape and terrain static-mesh
	// usage: RTS Obstacle and RTS Building placement onto landscape, terrain static-mesh
	float	GetLandTerrainSurfaceAtCoord(float XCoord, float YCoord) const;
```

Продовження додатку Б

```
    //-----------------------------------
};
```

Файл RTSPlayerCameraSpectatorPawn .cpp

```
#include "RTS.h"
#include "RTSPlayerCameraSpectatorPawn.h"

//////////////////////////////////////////////////////////////

ARTSPlayerCameraSpectatorPawn::ARTSPlayerCameraSpectatorPawn(const
FObjectInitializer& ObjectInitializer)
{
    // enable Tick function
    PrimaryActorTick.bCanEverTick = true;

    // disable standard WASD movement
    bAddDefaultMovementBindings = false;

    // not needed Pitch Yaw Roll
    bUseControllerRotationPitch = false;
    bUseControllerRotationYaw = false;
    bUseControllerRotationRoll = false;

    // collision
    GetCollisionComponent()->bGenerateOverlapEvents = false;
    GetCollisionComponent()->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    GetCollisionComponent()->SetCollisionProfileName(TEXT("NoCollision"));
    GetCollisionComponent()->SetSimulatePhysics(false);

    // set defaults

    CameraXYLimit = 25000.f;
    CameraHeight = 1000.f;
```

Продовження додатку Б

```
        CameraHeightMin = 300.f;            // 100 for debugging
        CameraHeightMax = 5000.f;

        CameraRadius = 2000.f;
        CameraRadiusMin = 1000.f;           // 100 for debugging
        CameraRadiusMax = 8000.f;

        CameraZAnlge = 0.f;                 // yaw

        CameraHeightAngle = 30.f;           // pitch
        CameraHeightAngleMin = 15.f;
        CameraHeightAngleMax = 60.f;

        CameraZoomSpeed = 200.f;            // wheel
        CameraRotationSpeed = 10.f;         // wheel + ctrl
        CameraMovementSpeed = 3000.f;       // in all directions

        CameraScrollBoundary = 25.f;        // screen edge width

        bCanMoveCamera = true;

        // intialize the camera
        CameraComponent =
ObjectInitializer.CreateDefaultSubobject<UCameraComponent>(this, TEXT("RTS Camera"));
        CameraComponent->AttachToComponent(RootComponent,
FAttachmentTransformRules::KeepRelativeTransform);
        CameraComponent->bUsePawnControlRotation = false;

        RepositionCamera();
    }


    ////////////////////////////////////////////////////////////////
```

Продовження додатку Б

```
void ARTSPlayerCameraSpectatorPawn::SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent)
{
    if (!PlayerInputComponent) return;

    Super::SetupPlayerInputComponent(PlayerInputComponent);

    // action mappings

    // mouse zoom
    PlayerInputComponent->BindAction("ZoomOutByWheel", IE_Pressed, this, &ARTSPlayerCameraSpectatorPawn::ZoomOutByWheel);
    PlayerInputComponent->BindAction("ZoomInByWheel", IE_Pressed, this, &ARTSPlayerCameraSpectatorPawn::ZoomInByWheel);

    // mouse rotate (+Ctrl or +Alt)
    // unnecessary...
    //PlayerInputComponent->BindAction("RotateLeftByWheel", IE_Pressed, this, &ARTSPlayerCameraSpectatorPawn::RotateLeftByWheel);
    //PlayerInputComponent->BindAction("RotateRightByWheel", IE_Pressed, this, &ARTSPlayerCameraSpectatorPawn::RotateRightByWheel);
    // needed...
    PlayerInputComponent->BindAction("RotateUpByWheel", IE_Pressed, this, &ARTSPlayerCameraSpectatorPawn::RotateUpByWheel);
    PlayerInputComponent->BindAction("RotateDownByWheel", IE_Pressed, this, &ARTSPlayerCameraSpectatorPawn::RotateDownByWheel);

    // axis mappings

    // keyboard move (WASD, Home/End)
    PlayerInputComponent->BindAxis("MoveForward", this, &ARTSPlayerCameraSpectatorPawn::MoveCameraForwardInput);
```

Продовження додатку Б

```
        PlayerInputComponent->BindAxis("MoveRight", this,
&ARTSPlayerCameraSpectatorPawn::MoveCameraRightInput);
        PlayerInputComponent->BindAxis("MoveUp", this,
&ARTSPlayerCameraSpectatorPawn::MoveCameraUpInput);
        PlayerInputComponent->BindAxis("ZoomIn", this,
&ARTSPlayerCameraSpectatorPawn::ZoomCameraInInput);

        // double speed (WASD +Shift)
        PlayerInputComponent->BindAxis("FastMove", this,
&ARTSPlayerCameraSpectatorPawn::FastMoveInput);
        // yaw and pitch (WASD +Ctrl)
        PlayerInputComponent->BindAxis("Rotate", this,
&ARTSPlayerCameraSpectatorPawn::RotateInput);
    }

    ////////////////////////////////////////////////////////////////

    void ARTSPlayerCameraSpectatorPawn::ZoomInByWheel()
    {
      if (!bCanMoveCamera)  return;

      CameraRadius -= CameraZoomSpeed * FastMoveValue;
      CameraRadius = FMath::Clamp(CameraRadius, CameraRadiusMin,
CameraRadiusMax);

      //RepositionCamera();
    }

    void ARTSPlayerCameraSpectatorPawn::ZoomOutByWheel()
    {
      if (!bCanMoveCamera)  return;
```

```
    CameraRadius += CameraZoomSpeed * FastMoveValue;
    CameraRadius = FMath::Clamp(CameraRadius, CameraRadiusMin, CameraRadiusMax);

    //RepositionCamera();
}


void ARTSPlayerCameraSpectatorPawn::RotateLeftByWheel()
{
    if (!bCanMoveCamera)  return;

    CameraZAnlge -= CameraRotationSpeed * FastMoveValue;

    //RepositionCamera();
}


void ARTSPlayerCameraSpectatorPawn::RotateRightByWheel()
{
    if (!bCanMoveCamera)  return;

    CameraZAnlge += CameraRotationSpeed * FastMoveValue;

    //RepositionCamera();
}


void ARTSPlayerCameraSpectatorPawn::RotateUpByWheel()
{
    if (!bCanMoveCamera)  return;

    CameraHeightAngle += CameraRotationSpeed * FastMoveValue;
```

Продовження додатку Б

```
        CameraHeightAngle = FMath::Clamp(CameraHeightAngle, CameraHeightAngleMin, CameraHeightAngleMax);

        //RepositionCamera();
    }


    void ARTSPlayerCameraSpectatorPawn::RotateDownByWheel()
    {
        if (!bCanMoveCamera)    return;

        CameraHeightAngle -= CameraRotationSpeed * FastMoveValue;
        CameraHeightAngle = FMath::Clamp(CameraHeightAngle, CameraHeightAngleMin, CameraHeightAngleMax);

        //RepositionCamera();
    }

    //---------------

    void ARTSPlayerCameraSpectatorPawn::RepositionCamera()
    {
        FVector NewLocation(0.f, 0.f, 0.f);
        FRotator NewRotation(0.f, 0.f, 0.f);

        float sinCameraZAngle = FMath::Sin(FMath::DegreesToRadians(CameraZAnlge));
        float cosCameraZAngle = FMath::Cos(FMath::DegreesToRadians(CameraZAnlge));

        float sinCameraHeightAngle = FMath::Sin(FMath::DegreesToRadians(CameraHeightAngle));
        float cosCameraHeightAngle = FMath::Cos(FMath::DegreesToRadians(CameraHeightAngle));
```

Продовження додатку Б

```
    NewLocation.X = cosCameraZAngle * cosCameraHeightAngle * CameraRadius;
    NewLocation.Y = sinCameraZAngle * cosCameraHeightAngle * CameraRadius;
    NewLocation.Z = sinCameraHeightAngle * CameraRadius;

    // do not allow camera component to go under ground - not enough alone, actor also needed to be limited
    float TerrainSurfaceZ = GetLandTerrainSurfaceAtCoord(GetActorLocation().X + NewLocation.X, GetActorLocation().Y + NewLocation.Y);
    if (GetActorLocation().Z + NewLocation.Z < TerrainSurfaceZ + CameraHeight)
    {
      //FVector NewLocation = CameraComponent->GetComponentLocation();
      NewLocation.Z = TerrainSurfaceZ - GetActorLocation().Z + CameraHeight;
    }

    // new camera location
    CameraComponent->SetRelativeLocation(NewLocation);

    // new camera rotation
    NewRotation = (FVector(0.0f, 0.0f, 0.0f) - NewLocation).Rotation();
    CameraComponent->SetRelativeRotation(NewRotation);
  }


  ////////////////////////////////////////////////////////////////


  void ARTSPlayerCameraSpectatorPawn::FastMoveInput(float Direction)
  {
    if (!bCanMoveCamera)    return;

    // left or right does not matter, to set double speed in any direction
    FastMoveValue = FMath::Abs(Direction) * 2.0f;
```

```
    // used as multiplier so must be 1 if not pressed
    if (FastMoveValue == 0.0f)
    {
        FastMoveValue = 1.0f;
    }
}


void ARTSPlayerCameraSpectatorPawn::RotateInput(float Direction)
{
    if (!bCanMoveCamera)    return;

    // left or right does not matter
    RotateValue = FMath::Abs(Direction);
}


void ARTSPlayerCameraSpectatorPawn::MoveCameraForwardInput(float Direction)
{
    if (!bCanMoveCamera)    return;

    MoveForwardValue = Direction;
}


void ARTSPlayerCameraSpectatorPawn::MoveCameraRightInput(float Direction)
{
    if (!bCanMoveCamera)    return;

    MoveRightValue = Direction;
}
```

Продовження додатку Б

```
void ARTSPlayerCameraSpectatorPawn::MoveCameraUpInput(float Direction)
{
   if (!bCanMoveCamera)    return;

   MoveUpValue = Direction;
}


void ARTSPlayerCameraSpectatorPawn::ZoomCameraInInput(float Direction)
{
   if (!bCanMoveCamera)    return;

   ZoomInValue = Direction;
}


//-------------------------------------------------------------


FVector ARTSPlayerCameraSpectatorPawn::MoveCameraForward(float Direction)
{
   float MovementValue = Direction * CameraMovementSpeed;
   FVector DeltaMovement = MovementValue * GetIsolatedCameraYaw().Vector();
   //FVector NewLocation = GetActorLocation() + DeltaMovement;
   //SetActorLocation(NewLocation);
   return DeltaMovement;
}


FVector ARTSPlayerCameraSpectatorPawn::MoveCameraRight(float Direction)
{
   float MovementValue = Direction * CameraMovementSpeed;
```

Продовження додатку Б

```
    FVector DeltaMovement = MovementValue * (FRotator(0.0f, 90.0f, 0.0f) + GetIsolatedCameraYaw()).Vector();
    //FVector NewLocation = GetActorLocation() + DeltaMovement;
    //SetActorLocation(NewLocation);
    return DeltaMovement;
}


FRotator ARTSPlayerCameraSpectatorPawn::GetIsolatedCameraYaw()
{
    // FRotator containing Yaw only
    return FRotator(0.0f, CameraComponent->ComponentToWorld.Rotator().Yaw, 0.0f);
}

//---------------

float ARTSPlayerCameraSpectatorPawn::MoveCameraUp(float Direction)
{
    float MovementValue = Direction * CameraMovementSpeed;
    //FVector DeltaMovement = FVector(0.0f, 0.0f, MovementValue);
    //FVector NewLocation = GetActorLocation() + DeltaMovement;
    //NewLocation.Z = FMath::Clamp(NewLocation.Z, CameraRadiusMin, CameraRadiusMax);
    //SetActorLocation(NewLocation);
    return MovementValue;
}

//---------------

void ARTSPlayerCameraSpectatorPawn::ZoomCameraIn(float Direction)
{
    float MovementValue = Direction * CameraMovementSpeed;
    CameraRadius += MovementValue;
```

Продовження додатку Б

```
    CameraRadius = FMath::Clamp(CameraRadius, CameraRadiusMin, CameraRadiusMax);

    //RepositionCamera();
}


void ARTSPlayerCameraSpectatorPawn::TurnCameraUp(float Direction)
{
    CameraHeightAngle -= Direction * CameraRotationSpeed * 10.0f;
    CameraHeightAngle = FMath::Clamp(CameraHeightAngle, CameraHeightAngleMin, CameraHeightAngleMax);

    //RepositionCamera();
}


void ARTSPlayerCameraSpectatorPawn::TurnCameraRight(float Direction)
{
    CameraZAnlge += Direction * CameraRotationSpeed * 10.0f;

    //RepositionCamera();
}


//////////////////////////////////////////////////////////////////


void ARTSPlayerCameraSpectatorPawn::Tick(float DeltaSeconds)
{
    Super::Tick(DeltaSeconds);

    // mouse position and screen size
```

Продовження додатку Б

```
FVector2D MousePosition;
FVector2D ViewportSize;

UGameViewportClient* GameViewport = GEngine->GameViewport;

// it is always nullptr on dedicated server
if (!GameViewport) return;
GameViewport->GetViewportSize(ViewportSize);

// if viewport is focused, contains the mouse, and camera movement is allowed
if (GameViewport->IsFocused(GameViewport->Viewport)
   && GameViewport->GetMousePosition(MousePosition) && bCanMoveCamera)
{
   //-------------------
   // movement direction by mouse at screen edge

   if (MousePosition.X < CameraScrollBoundary)
   {
      MoveRightValue = -1.0f;
   }
   else if (ViewportSize.X - MousePosition.X < CameraScrollBoundary)
   {
      MoveRightValue = 1.0f;
   }

   if (MousePosition.Y < CameraScrollBoundary)
   {
      MoveForwardValue = 1.0f;
   }
   else if (ViewportSize.Y - MousePosition.Y < CameraScrollBoundary)
   {
      MoveForwardValue = -1.0f;
   }
```

Продовження додатку Б

```
//------------------
// tweak camera actor position

FVector ActualLocation = GetActorLocation();
FVector ActualMovement = FVector::ZeroVector;

// horizontal movement
if (RotateValue == 0.f)
{
   ActualMovement += MoveCameraForward(MoveForwardValue * FastMoveValue * DeltaSeconds);
   ActualMovement += MoveCameraRight(MoveRightValue * FastMoveValue * DeltaSeconds);
}
ActualLocation += ActualMovement;

// vertical movement
CameraHeight += MoveCameraUp(MoveUpValue * FastMoveValue * DeltaSeconds);
CameraHeight = FMath::Clamp(CameraHeight, CameraHeightMin, CameraHeightMax);

// adjust actor height to surface
float TerrainSurfaceZ = GetLandTerrainSurfaceAtCoord(ActualLocation.X, ActualLocation.Y);
ActualLocation.Z = TerrainSurfaceZ + CameraHeight;

// limit movement area
ActualLocation.X = FMath::Clamp(ActualLocation.X, -CameraXYLimit, CameraXYLimit);
ActualLocation.Y = FMath::Clamp(ActualLocation.Y, -CameraXYLimit, CameraXYLimit);
```

Продовження додатку Б

```
        // move actor
        SetActorLocation(ActualLocation);

        //------------------
        // tweak camera component relative transform

        // set rotation parameters
        if (RotateValue != 0.f)
        {
            TurnCameraUp(MoveForwardValue * FastMoveValue * DeltaSeconds);
            TurnCameraRight(MoveRightValue * FastMoveValue * DeltaSeconds);
        }

        // set zoom distance
        ZoomCameraIn(ZoomInValue * FastMoveValue * DeltaSeconds);

        // adjust camera component relative location and rotation
        RepositionCamera();

        //------------------
        // debug
        //DrawDebugSphere(
        //                GetWorld(),
        //                GetCollisionComponent()->GetComponentLocation(),
        //                GetCollisionComponent()->GetScaledSphereRadius(),
        //                8,
        //                FColor::White,
        //                false,
        //                -1.f
        //            );
        //------------------
    }
}
```

Продовження додатку Б

```
//////////////////////////////////////////////////////////////

float ARTSPlayerCameraSpectatorPawn::GetLandTerrainSurfaceAtCoord(float XCoord, float YCoord) const
{
    FCollisionQueryParams TraceParams(FName(TEXT("LandTerrain")), false, this); // TraceTag (info for debugging), bTraceComplex, AddIgnoredActor
    TraceParams.bFindInitialOverlaps = false;                    // needed

    FHitResult Hit;

    FVector Start = FVector(XCoord, YCoord, GetActorLocation().Z + CameraRadius);
    FVector End = FVector(XCoord, YCoord, -500.f);

    // ECC_ channels should be set properly !!!
    bool bHit = GetWorld()->LineTraceSingleByChannel(Hit, Start, End, ECollisionChannel::ECC_WorldStatic, TraceParams);

    if (bHit)
    {
       return Hit.ImpactPoint.Z;    // for shape trace it differs from Location
    }

    return 0.f;        // water level
}

//////////////////////////////////////////////////////////////
```