ЗМІСТ

РЕФЕРАТ

Кваліфікаційна робота: 52 с., 10 рисунків, 14 джерел.

Мета роботи: розробка механіки гри для жанру стратегії в реальному часі (RTS) із використанням рушія Godot Engine. Основні завдання включають вивчення теоретичних аспектів жанру стратегій у реальному часі, дослідження функціональних можливостей рушія Godot Engine, створення архітектури ігрової логіки, а також реалізацію базових ігрових механік.

Об'єкт дослідження – процес розробки ігрової механіки для RTS-ігор у середовищі Godot Engine.

Предмет дослідження – алгоритми та програмні рішення, що забезпечують функціонування базових компонентів RTS-ігор, зокрема управління юнітами (unit – базова інтерактивна ігрова одиниця, яка виконує певні функції в межах ігрової системи), побудова бази, видобуток ресурсів і взаємодія з оточенням.

Методи вивчення – аналіз ігрових жанрів і принципів RTS-дизайну, використання інструментів програмування GDScript у Godot Engine, моделювання ігрових процесів.

Проведено дослідження можливостей рушія Godot Engine щодо реалізації механік реального часу, розглянуто приклади побудови логіки взаємодії юнітів, ресурсів і гравця. Розроблено прототип гри, що відображає основні принципи RTS-ігрового процесу. Проведено тестування функціональності ігрових механік, а також аналіз їх ефективності в контексті реалізованої моделі.

Ключові слова: RTS, GODOT ENGINE, ВІДЕОІГРИ.

ВСТУП

Індустрія розробки відеоігор на сьогодні є одним з найдинамічніших напрямків у сфері цифрових технологій. Кожного року з'являється велика кількість нових ігор, які вимагають не лише високоякісної графіки, а й продуманої механіки, що забезпечує гравцеві глибоке занурення в ігровий процес. Одним із жанрів, який зберігає свою популярність протягом десятиліть, є стратегія в реальному часі. Вона поєднує планування, реакцію на змінні умови гри та керування ресурсами, вимагаючи від розробника чіткої логіки та технічної досконалості в реалізації взаємодії між гравцем та ігровими об'єктами.

У зв'язку з цим питання побудови якісної системи механік для стратегічних ігор є надзвичайно актуальним. Багато сучасних інструментів дозволяють реалізувати складну логіку керування об'єктами, проте не всі з них забезпечують легкість масштабування та модульність архітектури. Серед відкритих рушіїв особливу увагу привертає Godot Engine, який завдяки своїй відкритості, доступу на всіх найпопулярніших платформах та активній спільноті стає привабливим вибором для розробників інді-рівня та для академічних досліджень у галузі ігрового програмування.

Робота присвячена дослідженню та розробці базової механіки для гри в жанрі стратегії в реальному часі, з урахуванням специфіки архітектури Godot Engine. Основна увага приділяється створенню системи керування бойовими одиницями, побудови споруд, взаємодії з ресурсами, а також впровадженню елементів користувацького інтерфейсу. У межах проєкту я зробив особливий акцент на можливість легкого розширення функціональності без потреби повного переписування коду, що є критично важливим аспектом у процесі еволюції будь-якого програмного продукту.

У процесі реалізації я проаналізував переваги та недоліки наявних підходів до створення RTS-механік, що дозволило виробити власну модель логіки взаємодії у грі, адаптовану до технічних можливостей рушія. Також

розглянув питання структурування коду в умовах роботи з вузлами сцени, використання сигналів для взаємодії між об'єктами та впровадження автоматизованих сценаріїв поведінки персонажів. Особливу увагу приділив стабільності системи при зростанні кількості об’єктів, що безпосередньо впливає на продуктивність гри в реальному часі.

Вибір теми зумовлений потребою у гнучких, відкритих рішеннях, які дозволяють моделювати складні ігрові механізми без обмежень, пов’язаних із закритістю програмного забезпечення чи надмірною складністю його структури. Godot Engine є вдалим прикладом інструменту, що дає змогу реалізувати повноцінний ігровий функціонал на базі відкритих стандартів. Таким чином, дана робота дозволяє не лише продемонструвати технічну реалізацію базової механіки стратегії в реальному часі, але й закладає основу для подальшого вдосконалення таких систем у майбутніх проєктах.

Метою роботи є створення технічно надійної, масштабованої та ефективної структури гри жанру RTS на основі Godot Engine, яка здатна забезпечити керування юнітами, будівництво, збір ресурсів та реалізацію базових елементів ШІ. У результаті розробки, я створив функціональну систему, придатну для подальшого розширення, адаптації під потреби різних ігрових сценаріїв та впровадження додаткових механік, таких як багатокористувацький режим чи просунуте дерево технологій.

РОЗДІЛ 1
ПРОЄКТУВАННЯ АРХІТЕКТУРИ ГРИ

1.1. Аналіз вимог до архітектури RTS-проєктів

Жанр стратегій у реальному часі (RTS, Real-Time Strategy) має характерні особливості, які висувають специфічні вимоги до структури ігрового проєкту [10]. Одним із ключових аспектів є забезпечення одночасного функціонування великої кількості активних об’єктів, що взаємодіють між собою в рамках загальної ігрової логіки. Це стосується як юнітів (ігровий персонаж, який в моєму проєкті має свою логіку переміщення, атаки, використання спеціальних здібностей тощо) під керуванням гравця, так і ворожих об’єктів, ресурсів, споруд та всього механізму управління. Така багатошарова взаємодія потребує від архітектури гри високого рівня модульності, точного контролю станів об'єктів і продуманого структурного ієрархічного підходу.

Ще однією визначальною вимогою для проєктів RTS є чітке розділення між графічною складовою та логікою гри. На відміну від багатьох інших жанрів, де увага фокусується на окремому персонажі чи обмеженій кількості об’єктів, стратегічний формат вимагає можливості ефективного керування десятками або навіть сотнями елементів без втрати продуктивності. Це створює необхідність впровадження централізованої системи управління подіями, глобального менеджера станів гри та оптимізованого використання пам’яті.

Особливе значення для RTS [14] має масштабованість проєкту. Через те, що ці ігри часто переживають кілька етапів розвитку — додаються нові механіки, з’являються системи дипломатії, торгівлі чи покращення штучного інтелекту супротивників — архітектура має спочатку ґрунтуватися на принципах гнучкості. Це передбачає використання абстракцій, шаблонів програмування та делегування обов’язків між класами, які опікуються відповідними модулями гри. Такий підхід полегшує інтеграцію нових функцій без істотних змін у вже наявній структурі.

Не менш важливою є реалізація складної та зручної системи користувацького інтерфейсу (UI). Для RTS характерна велика кількість елементів — інформаційні панелі, ресурси, кнопки команд — що вимагає вдалої інтеграції UI з основною логікою гри. Досягти цього можна через впровадження автономного UI-контролера, який отримує сигнали від об’єктів сцени і дозволяє уникати порушень у логіці програми. Завдяки цьому забезпечується двосторонній потік інформації між інтерфейсом і механіками гри.

Окремо слід підкреслити значення синхронізації процесів у реальному часі. Гравець взаємодіє із численними паралельними процесами: будівництвом споруд, переміщенням військ, боями, збиранням ресурсів тощо. Некоректна координація цих процесів може створити хаос і знизити загальну якість гри. Тому необхідно чітко розмежовувати логіку гри та її візуалізацію. Такий підхід дозволяє уникнути затримок або помилок оновлення позицій об’єктів і зберігати стабільну продуктивність.

Ще одним важливим аспектом при плануванні структури RTS є потенційне додавання мультиплеєру. Попри те, що онлайн-функціонал може бути відсутнім на початкових етапах розробки, добре підготовлена архітектура зі зрозумілим розділенням клієнтських і серверних функцій значно спростить інтеграцію цієї функції в майбутньому. Це потребує завчасного продумування механізмів синхронізації станів, обробки команд гравців і роботи з чергами подій. У наступних підрозділах буде детально розглянуто, як саме ці вимоги реалізуються в межах практичного проєкту.

1.2. Вибір ігрового рушія: переваги Godot Engine для реалізації RTS

Процес розробки гри завжди починається з вибору рушія, на якому буде реалізовано проєкт. Для гри жанру RTS вибір рушія має ключове значення, оскільки від нього залежить ефективність реалізації складної логіки, зручність розробки інтерфейсу користувача, продуктивність при обробці великої кількості об’єктів у реальному часі, а також можливість гнучко модифікувати структуру проєкту впродовж його розвитку. У цьому контексті Godot Engine демонструє

себе як потужний інструмент з низкою переваг, які роблять його доцільним вибором для реалізації RTS-механіки.

Однією з основних переваг Godot Engine [1] є його модульна архітектура на основі вузлів (nodes), яка дозволяє органічно структурувати всі об’єкти гри — від окремого юніта до цілого інтерфейсу або логічної системи. Кожен вузол виконує чітко визначену функцію, і завдяки цьому можна легко розділяти відповідальність між класами, створювати ієрархії об’єктів, реалізовувати багаторівневу поведінку і при цьому не втрачати контроль над внутрішніми процесами гри. Такий підхід ідеально підходить для RTS, де одночасно взаємодіють десятки й сотні активних елементів.

Середовище розробки Godot також забезпечує високий рівень зручності завдяки своїй інтуїтивно зрозумілій візуальній системі, що дозволяє ефективно працювати як з 3D, так і з 3D графікою. У випадку з RTS, де я звичайно реалізував ізометричну перспективу, рушій надає широкі можливості для налаштування камер, шарів відображення, а також логіки взаємодії між об’єктами у просторі. Це відкриває шлях для створення власних систем видимості, fog-of-war, обмежень огляду та зони дії, що є типовими для стратегічного жанру.

Ще однією важливою перевагою є вбудована мова сценаріїв GDScript, яка завдяки своїй легкості і близькості до Python дозволяє швидко створювати та змінювати логіку гри. Цей фактор надзвичайно важливий на етапі прототипування і тісної ітераційної розробки, коли функціональність постійно доповнюється або змінюється. Можливість швидко тестувати нові функції без складного компілювання, як це притаманно багатьом іншим рушіям, робить GDScript одним з найефективніших інструментів для створення гнучкої RTS-архітектури.

Godot також підтримує систему сигналів, яка дозволяє будувати подієво-орієнтовану архітектуру без необхідності в складних механізмах або безпосередніх залежностях між об’єктами. У грі жанру RTS, де важлива миттєва реакція на дії гравця, події, зміни станів об’єктів, завершення

будівництва тощо, система сигналів дозволяє суттєво спростити реалізацію таких механік. Кожен об’єкт може генерувати власні сигнали, на які інші компоненти реагують лише за потреби, зберігаючи при цьому низький рівень зв’язності та високу гнучкість структури.

Крім того, варто відзначити активну спільноту розробників навколо Godot, яка постійно оновлює документацію, створює нові інструменти, приклади, а також плагіни, які можуть бути корисними для реалізації специфічних RTS-елементів. Наприклад, є модулі для створення сіткових карт, AI навігації, процедурної генерації рівнів, які я використав як приклад оптимізації власних карт навігації. Завдяки відкритому коду рушія, за потреби розробник може внести зміни навіть на рівні ядра рушія, що відкриває ширші можливості для оптимізації.

Godot також пропонує високий рівень продуктивності при роботі з великою кількістю об'єктів у реальному часі, особливо в останніх версіях рушія. У поєднанні з грамотним дизайном архітектури гри це дозволяє досягти стабільної роботи навіть у проєктах зі складною логікою і великою кількістю елементів на екрані.

Варто згадати про переваги відкритої ліцензії Godot Engine — він є повністю безкоштовним і не вимагає виплат роялті або ліцензійних зборів при комерціалізації гри. Для незалежних розробників або команд з обмеженим бюджетом це може бути вирішальним фактором при виборі рушія, оскільки дозволяє вкладати більше ресурсів у саму розробку та маркетинг проєкту.

Підсумовуючи, можна сказати, що Godot Engine є конкурентоспроможним рішенням для реалізації гри в жанрі RTS завдяки своїй відкритій структурі, високій модульності, ефективному рушію подій, зручній мові програмування та активній спільноті. У наступних розділах буде розглянуто, як саме можливості Godot були використані на практиці під час реалізації окремих аспектів RTS-механіки.

1.3. Організація файлової структури та ієрархії сцен

Ефективна організація проєкту — один з ключових факторів успішної розробки гри, особливо у випадку з жанром RTS, який передбачає велику кількість взаємопов'язаних систем, багаторівневу логіку взаємодії між об'єктами, велику кількість сцен, скриптів і ресурсів. У межах використання Godot Engine структура проєкту має не лише відповідати принципам зручності для розробника, але й сприяти модульності, повторному використанню компонентів і швидкому орієнтуванню в системі [7]. Саме тому було приділено велику увагу створенню чіткої, ієрархічної файлової структури та логічного розбиття сцен на підкатегорії, що відображають їхнє функціональне призначення.

Початкове структурування проєкту було зосереджене на розподілі всіх елементів гри на окремі каталоги. Для зручності та читабельності структури верхнього рівня було створено такі основні директорії: Scenes, Scripts, UI, Assets, Systems, Units, Buildings і World. Кожен із цих розділів відповідав за окремий функціональний блок гри. Наприклад, у теці Scenes зберігались основні сцени гри — головне меню, сцена гри, екран поразки та перемоги.

Особливу увагу я приділяв структурі сцен у Godot. Завдяки вузловій архітектурі рушія, кожен об'єкт гри проєктувався як окрема сцена, що дозволяло незалежно тестувати й редагувати компоненти без потреби перезапускати всю гру. Наприклад, базовий юніт складався з таких вузлів, як MeshInstance3D для візуального відображення, Area3D для виявлення зіткнень, NavigationAgent3D для реалізації пересування по навігаційній сітці, а також дочірніх вузлів для відображення здоров'я, виконання команд тощо. Усі ці вузли були згруповані в рамках окремої сцени Unit.tscn, яка потім використовувалася в інших сценах як екземпляр сцени.

Ієрархія сцен базувалася на принципі спадковості й повторного використання. Наприклад, базовий клас UnitBase містив універсальну логіку, спільну для всіх юнітів — рух, отримання пошкоджень, стан готовності. Від цього класу наслідувались інші, специфічніші — MeleeUnit, RangeUnit,

UtilityUnit тощо, які розширювали або перевизначали поведінку базового класу. Такий підхід дозволив не дублювати код і значно пришвидшив розробку нових типів персонажів [5].

Подібний підхід використовувався і для будівель. Базова сцена BuildingBase включала логіку встановлення будівлі на сітку, зону дії, а також візуальні індикатори будівництва. Від неї наслідувались окремі сцени таких будівель, як казарми, склади, ферми та оборонні башти. Кожна будівля містила свої унікальні властивості, але водночас зберігала загальний інтерфейс взаємодії з іншими елементами гри.

Також було реалізовано централізовану структуру для системних компонентів гри. У теці Systems розташовувались скрипти та сцени, відповідальні за загальну логіку гри: керування ресурсами, генерація рівня, штучний інтелект, навігація тощо. Наприклад, система ResourceManager відповідала за відстеження кількості ресурсів у гравця, додавання або витрати ресурсів, а також інформування інших систем про зміни. Ці компоненти були написані з урахуванням принципів незалежності та взаємодії через глобальні сигнали або реєстрацію в головному контролері гри.

Головна сцена Game.tscn виступала точкою входу до всієї логіки RTS. У ній додавались основні компоненти — сітка світу, система керування ігровими персонажами, користувацький інтерфейс, контролер ресурсів, а також динамічне завантаження початкових генералів і будівель. Завдяки такій структурі було легко підтримувати чистоту проєкту, швидко знаходити необхідні елементи та розширювати гру новими механіками, не порушуючи вже наявну архітектуру.

Для збереження порядку та читабельності кодової бази було прийнято рішення зберігати скрипти в окремій директорії Scripts, організованій за аналогією з теками сцен. Наприклад, скрипти юнітів зберігались у Scripts/Units, скрипти систем — у Scripts/Systems тощо. Кожен скрипт мав назву, яка чітко відображала його призначення, і був оформлений відповідно до єдиного стилю написання коду, що значно спростило підтримку проєкту в майбутньому.

Особливу увагу було приділено й ресурсам гри. Всі візуальні, аудіо та інші файли зберігались у теці Assets, де вони були розділені за категоріями: текстури, анімації, звукові ефекти, іконки інтерфейсу тощо. Це дало змогу не тільки зменшити час пошуку потрібного ресурсу, але й організувати механізми автозавантаження, кешування та заміни ресурсів у runtime (), що також є важливим для ігор стратегічного жанру.

Таким чином, завдяки ретельно продуманій організації структури проєкту вдалося досягти високої масштабованості, стабільності та гнучкості при подальшій розробці гри. У наступному розділі буде розглянуто, яким чином побудована логіка основних ігрових систем, зокрема обробки команд гравця, руху юнітів, збору ресурсів та взаємодії між об’єктами у грі.

1.4. Огляд схожих існуючих проєктів

Під час розробки концепції гри, я проаналізував низку існуючих стратегічних проєктів у жанрі RTS (Real-Time Strategy), які мали найбільший вплив на мої дизайнерські рішення. Такий аналіз дозволив мені глибше зрозуміти сильні сторони сучасних стратегій, типові помилки, а також визначити, яким чином моя гра може запропонувати гравцям новий досвід.

Першим проєктом, який справив суттєвий вплив, стала гра Warcraft III: Reign of Chaos (Blizzard Entertainment, 2002). Вона відзначилася не лише як культова RTS, але і як проєкт, що започаткував нові підходи до взаємодії між будівництвом, економікою і героями. Найбільший інтерес для мене становив її механізм генералів — потужних героїв, що можуть прокачуватися, взаємодіяти з об'єктами карти, збирати артефакти та лідирувати у бою. Саме цей принцип я використав як основу для концепції генерала, адаптувавши його під більш стратегічну роль, що поєднує командування та розвиток бази.

Ще одним важливим джерелом натхнення стала Dota 2 (Valve, 2013), хоча вона і не є класичною RTS, а радше MOBA. Особливий інтерес для мене становив підхід до героя як до центральної фігури ігрового процесу — з унікальними здібностями, впливом на командну стратегію та критичною

важливістю в бою. Я прагнув адаптувати ці принципи до свого генерала у своєму проєкті, наділяючи його унікальними активними й пасивними навичками, здатністю впливати на перебіг битви, а також ключовою умовою поразки у разі його знищення. Такий підхід дозволяє зробити гру більш динамічною та наповненою моментами тактичного напруження, подібно до кульмінаційних моментів у командних боях Dota 2.

Не можу оминути проєкт Northgard (Shiro Games, 2017), що має гібридну модель між традиційною RTS. Його сильна сторона — гнучка, але обмежена система територій, яка обмежує експансію і водночас стимулює прийняття важливих стратегічних рішень. Попри те, що моя гра не базується на розширенні територій через зони, я надихався цією ідеєю для обмеження доступу до ресурсів і створення точок напруги на карті, які мають стратегічну вагу.

Загалом, аналіз існуючих стратегічних ігор дозволив мені виявити ключові патерни, яких прагнуть сучасні гравці: асиметрія фракцій, значущі командири, стратегічна економіка та гнучке, але небезболісне будівництво. Мій кінцевий продукт поєднує знайомі гравцям механіки з власними інноваціями — генералом як центром не лише бою, а й економіки, унікальними расовими спорудами, що доступні лише конкретним фракціям, та суворою умовністю смерті генерала, як критичної точки гри. Це дозволяє створити ігровий процес, який водночас спирається на класичні принципи RTS, але має достатню глибину і новизну для залучення сучасної аудиторії.

РОЗДІЛ 2
РЕАЛІЗАЦІЯ БАЗОВОЇ МЕХАНІКИ СТРАТЕГІЇ В РЕАЛЬНОМУ ЧАСІ

2.1. Реалізація керування ігровими одиницями

У контексті розробки RTS-гри управління ігровими одиницями відіграє ключову роль, адже саме воно формує основний спосіб взаємодії гравця з ігровим світом. Механіка керування ігровими персонажами включає в себе декілька важливих аспектів: виділення одиниць, прийняття команд користувача, переміщення по карті, взаємодія з іншими об’єктами, а також візуальна й звукова індикація поточних дій. Реалізація цих функцій вимагала комплексного підходу, заснованого на чіткому розмежуванні обов’язків між підсистемами введення, навігації, логіки та візуалізації.

Першим етапом у реалізації керування стала побудова системи виділення юнітів. Було вирішено впровадити класичну для RTS-ігор систему «прямокутного виділення», яка дозволяє гравцеві виділяти одразу декілька одиниць шляхом створення прямокутника мишкою. Для цього в сцені інтерфейсу було створено окремий шар, який відповідав за відображення прямокутника виділення. При натисканні лівої кнопки миші зберігалася початкова позиція, а при русі курсора будувався прямокутник, який оновлювався в реальному часі. Після відпускання кнопки всі об’єкти, що мали колізійні області та потрапляли в межі прямокутника, додавалися до списку виділених.

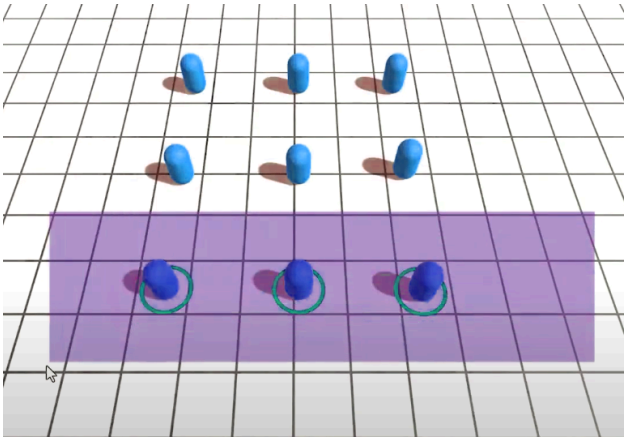

Рисунок 2.1. Демонстрація виділення об’єктів

Після виділення бойової одиниці, гравцеві надається можливість віддавати їм команди. Найбільш базовою командою стало переміщення — при натисканні правої кнопки миші на мапі, кожен виділений юніт отримував інструкцію перемістити до відповідної точки. Щоб уникнути того, що всі одиниці намагалися дістатися до однієї й тієї ж координати, було реалізовано алгоритм розподілу цілей, де кожному об’єкту надавалося унікальне зміщення навколо центральної точки. Це дозволяло створити більш природну поведінку руху персонажів групами та запобігало небажаним зіткненням.

Для реалізації логіки переміщення була задіяна навігаційна система Godot. Кожна бойова одиниця мала NavigationAgent3D, який відповідав за побудову шляху до цілі та уникнення перешкод під час руху. Крім того, кожна одиниця реагувала на переривання шляху або зміну цілі, що забезпечувало гнучкість у випадках динамічного середовища — наприклад, коли інші юніти або будівлі перекривали шлях.

Командна система, яка відповідала за прийом і обробку дій гравця, була централізована. У проекті був створений спеціальний компонент PlayerCommandProcessor, який перехоплював натискання кнопок миші та клавіші, визначав контекст взаємодії (чи натиснуто на землю, на ворога, на будівлю) та розподіляв команди відповідним одиницям. Ця система дозволила легко додати нові типи команд, наприклад, атаку, збирання ресурсів, будівництво або спеціальні дії. Кожна команда визначала тип взаємодії та передавалася через подієву модель або безпосередньо у StateManager відповідного об’єкта, який сам вирішував, як її обробити залежно від свого поточного стану.

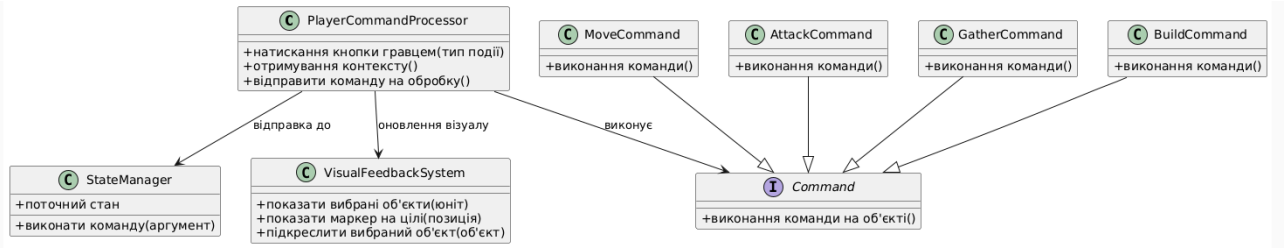

Рисунок 2.2. Процес відправлення команд від PlayerCommandProcessor

Ще одним важливим аспектом стало створення системи візуальної індикації. Для кожної виділеної одиниці активувалася відповідна рамка або іконка, яка сигналізувала гравцеві про поточне виділення. Крім того, на місці цілі юніта з’являвся маркер — стрілка, круг або анімація — яка вказувала на напрямок і точку переміщення. У разі атаки ціль також могла підсвічуватися. Це дозволяло гравцеві постійно тримати під контролем ситуацію, не гублячи контексту на полі бою.

Важливою частиною реалізації керування бойовими одиницями стала інтеграція поведінки штучного інтелекту. Після отримання команди, одиниця не просто переміщалася, а оцінювала обставини на місці прибуття: якщо в зоні знаходився ворог, могла автоматично перейти в режим атаки; якщо в межах дії була доступна будівля союзника — перейти в режим взаємодії. Таким чином, юніти мали базову автономність, яка забезпечувала живу поведінку в умовах мінімальної участі гравця. Це дозволяло сфокусувати увагу на стратегічних рішеннях, не перевантажуючи мікроконтролем.

Реалізація керування союзними персонажами також передбачала врахування продуктивності. У ситуації, коли гравець виділяв десятки або навіть сотні одиниць, кожна з яких мала власну логіку руху, стану та взаємодії, виникала загроза перевантаження процесора. Для запобігання цьому була реалізована система обмеження частоти оновлення логіки: об'єкти оновлювалися не кожен кадр, а через визначений інтервал, у залежності від своєї пріоритетності. Крім того, об’єкти, що перебували поза межами камери, мали спрощене оновлення логіки або тимчасово дезактивувалися.

На ранніх етапах розробки будь-якої гри, зокрема в жанрі RTS, важливо усвідомлювати майбутні потреби щодо розширення функціоналу, впровадження нових механік та повторного використання вже написаного коду. Розробка, орієнтована на масштабування, дозволяє заощадити значну кількість часу, зусиль і ресурсів у середньо- та довгостроковій перспективі. Саме тому одним з ключових етапів у побудові архітектури RTS-гри в Godot Engine стало ретельне

планування та впровадження принципів модульності, розширюваності й повторного використання.

Початковою точкою у цій роботі стало визначення фундаментальних компонентів гри, які, з високою ймовірністю, будуть використовуватись у багатьох місцях або вимагатимуть подальшої адаптації. Прикладами таких компонентів стали класи керування ресурсами, системи побудови та юнітів, логіка обробки команд гравця, навігація, обчислення шкоди та стану об’єктів. Всі ці елементи було реалізовано як максимально ізольовані модулі, що взаємодіють між собою через сигнали або чітко визначені інтерфейси.

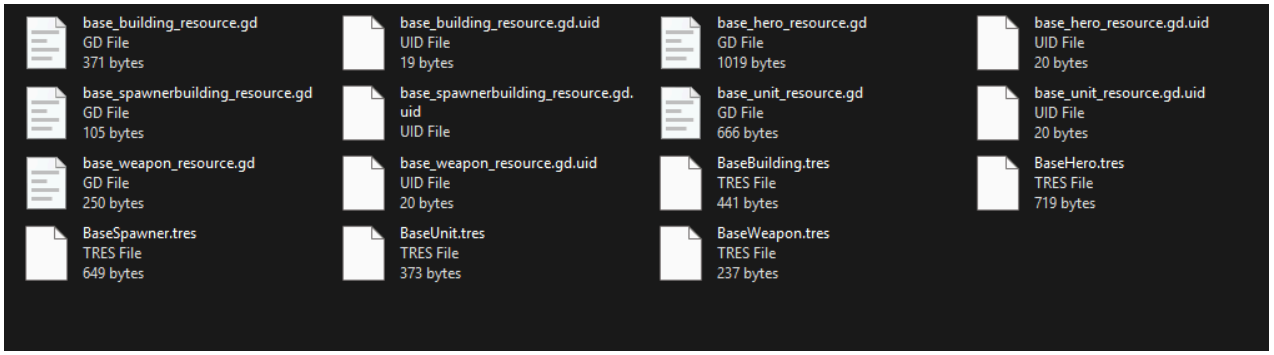

Рисунок 2.3. Створені базові структури для об’єктів

Завдяки сигнальній системі Godot вдалося уникнути жорсткої залежності між об’єктами, що стало визначальним у контексті масштабування. Наприклад, замість того, щоб кожен юніт безпосередньо звертався до менеджера ресурсів або головного контролера гри, він лише надсилав сигнал зі своїм запитом. Це дозволило змінювати поведінку системи без потреби вносити зміни до десятків інших скриптів, які з нею взаємодіють. Такий підхід є критично важливим, коли мова йде про розширення механік — наприклад, додавання нових типів ресурсів або поведінки юнітів.

В основі архітектури також лежав принцип єдиної відповідальності. Кожен клас або скрипт виконував лише одну конкретну функцію. Наприклад, UnitSelector відповідав тільки за виділення юнітів на мапі, тоді як UnitCommander керував відправкою команд на пересування або атаку. Це дало змогу тестувати та модифікувати окремі частини логіки незалежно, що значно

спростило налагодження й повторне використання компонентів в інших контекстах.

Ще одним важливим кроком стала абстрагована реалізація базових об’єктів гри. Було створено низку батьківських класів і шаблонних структур, від яких успадковувались специфічні юніти, будівлі або ресурси. Наприклад, клас UnitBase містив спільну логіку для всіх юнітів: обробку переміщення, застосування шкоди, анімацію та стани. Нові типи юнітів могли швидко реалізовуватись на основі цього класу, додаючи або змінюючи лише специфічну поведінку. Це дозволило заощадити час і підтримувати консистентність логіки в усіх ігрових об'єктах.

Подібним чином були структуровані дані об’єктів. Замість того щоб жорстко прописувати всі параметри юнітів або будівель у скриптах, їхні характеристики (вартість, здоров’я, час будівництва, швидкість руху тощо) зберігались у окремих ресурсах типу UnitData або BuildingData. Це дало змогу централізовано змінювати баланс гри та легко додавати нові об’єкти, не змінюючи код логіки. Крім того, такі ресурси могли використовуватись у редакторах рівнів або сценаріїв гри, що забезпечило додаткову гнучкість у розробці.

Особливої уваги заслуговує підхід до проєктування інтерфейсів. Користувацький інтерфейс також був реалізований у вигляді окремих сцен і скриптів, які могли легко підключатись або замінюватись. Кожен інтерфейсний компонент (панель ресурсів, меню будівництва, інформація про юніта) мав власну сцену, логіку оновлення даних та взаємодії з гравцем. Всі вони комунікували з основними системами через сигнали або абстрактні методи, що дозволяло змінювати або розширювати UI без втручання в основну логіку гри.

У контексті масштабування було враховано й можливість впровадження мультиплеєра або сценаріїв з декількома гравцями. Для цього було закладено фундамент для роботи з контролерами гравця як окремими сутностями. Замість того щоб мати жорстко закодовану логіку одного гравця, кожен гравець мав власний контролер, що відповідав за виділення юнітів, прийняття рішень та

взаємодію з системами. Це відкривало перспективи для додавання ШІ-суперників або мережевої гри без необхідності переписувати основну логіку взаємодії з об’єктами.

Не менш важливим аспектом став підхід до повторного використання коду між проєктами або при створенні нових ігор. Ключові компоненти гри були спроєктовані з урахуванням можливої зовнішньої інтеграції. Наприклад, системи роботи з сіткою світу, навігацією, побудовою інтерфейсу чи збором ресурсів реалізовувались у вигляді універсальних модулів, які не містили жорсткої прив’язки до конкретного проєкту. Таким чином, у майбутньому їх можна буде використати в інших RTS або навіть у суміжних жанрах без значних змін.

Зрештою, підготовка до масштабування — це не лише про архітектуру коду, а й про культуру розробки. У рамках проєкту впроваджувались практики ведення документації, регулярного коментування коду, дотримання стилю написання та структурування файлів. Це дозволяло будь-якому члену команди швидко орієнтуватися в кодовій базі та вносити зміни без ризику порушити логіку гри. Навіть у межах індивідуальної розробки це відіграє ключову роль, адже в масштабних проєктах легко втратити розуміння зв’язків між компонентами без належного супроводу.

У підсумку, реалізація керування ігровими одиницями у грі жанру RTS вимагала комплексного, модульного та масштабованого підходу, створення архітектури, орієнтованої на масштабування та повторне використання, стало одним з визначальних факторів у стабільності та ефективності розробки RTS-гри. У подальших розділах буде розглянуто, яким чином ця архітектура використовується у конкретних системах гри, таких як штучний інтелект ворогів, побудова маршрутів і бойова взаємодія.

2.2. Побудова системи менеджменту станів і об’єктів

У контексті розробки гри жанру RTS одним із критичних завдань стає ефективне управління великою кількістю об’єктів, які взаємодіють між собою на сцені в реальному часі. Це включає ігрові одиниці, будівлі, ресурси, ефекти, зони впливу, снаряди та інші сутності, які можуть створюватися, змінювати стан або знищуватися під час гри. Щоб уникнути хаосу та надмірної складності при реалізації логіки цих об'єктів, я створив систему менеджменту станів об’єктів і централізованого керування об'єктами.

Першим кроком стало запровадження чіткої концепції життєвого циклу об’єктів гри. Кожен об’єкт, який мав власну логіку, існував у межах певного набору станів, які визначали його поведінку в конкретний момент. Для прикладу, персонаж бути в одному з таких станів: очікування, виконання наказу, атака, переміщення, відступ або смерть. Таке розмежування дозволило впровадити патерн станів (State Pattern), який полягав у делегуванні логіки конкретному об’єкту-стану, замість того, щоб реалізовувати всі варіанти поведінки в одному великому скрипті.

Для кожного типу об'єкта було створено відповідний State Manager — компонент, що відповідав за зберігання поточного стану та зміну поведінки залежно від умов гри [3]. Наприклад, UnitStateManager відстежував поточну діяльність бойової одиниці, а також реагував на зовнішні події: отримання команди гравця, виявлення ворога поблизу або закінчення анімації атаки. Такий підхід не лише спростив читабельність коду, але й дозволив гнучко додавати нові стани або змінювати логіку поточних без ризику порушити інші гілки поведінки.

Ключовою перевагою цієї архітектури стала можливість створення станів як окремих класів або ресурсів. Наприклад, StateIdle, StateMove, StateAttack могли мати власні методи обробки входу в стан, оновлення під час виконання та виходу з нього. Це дозволяло кожному стану відповідати за себе, що своєю чергою, значно покращувало масштабованість та спрощувало тестування. Систему станів можна було під'єднати не лише до юнітів, але й до будівель, які

могли перебувати, наприклад, у станах будівництва, функціонування, ушкодження чи знищення.

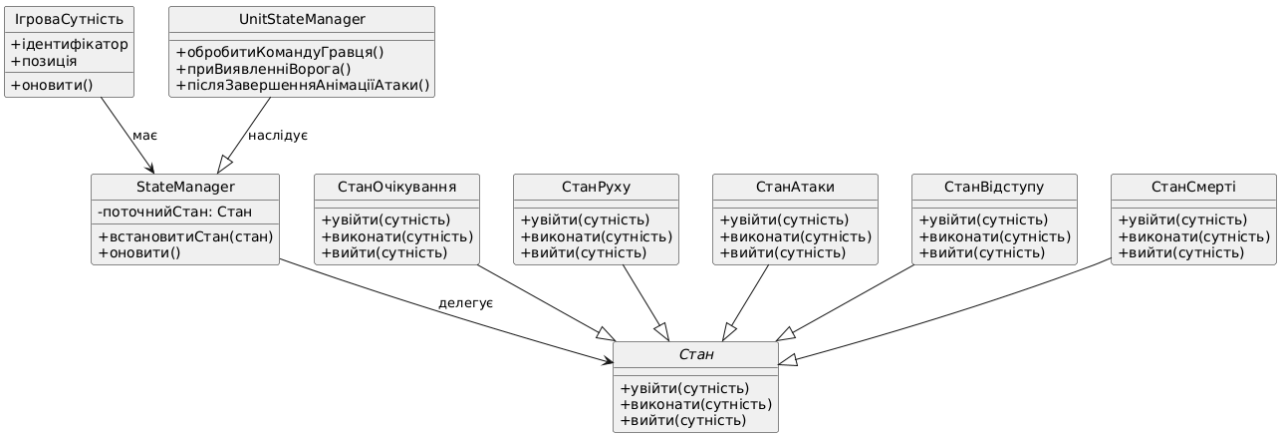

Рисунок 2.4. Структура системи управління станами ігрових об'єктів

Паралельно з менеджментом станів впроваджувалась система централізованого керування об'єктами на глобальному рівні. Основу цієї системи становив GameObjectManager — сцена, що відповідала за реєстрацію, зберігання та контроль над усіма активними ігровими об’єктами. Це дало змогу реалізувати зручний механізм ітерації по всіх активних об’єктах без потреби зчитувати інформацію безпосередньо зі сцени, що значно підвищило продуктивність та дозволило уникнути непотрібного доступу до дерева вузлів.

Окрему роль у цій системі відігравав поділ об’єктів за категоріями та фракціями. Усі об’єкти могли належати до певної групи (наприклад, "player_units", "enemy_units", "buildings") і фракції (наприклад, "human", "goblin"). Це спростило реалізацію логіки взаємодії — такі як перевірка колізій між ворогами, вибір цілей або визначення зони дії здібностей. Подібний поділ дозволив також реалізувати ефективні фільтри в системах відображення, навігації та бойової взаємодії.

Не менш важливим аспектом менеджменту об'єктів стала синхронізація станів об’єктів з візуальним відображенням. Кожен стан об'єкта супроводжувався відповідними анімаціями, ефектами або звуками. Наприклад, при переході бойової одиниці в стан атаки викликалась відповідна анімація,

обчислювався час до завдання шкоди, і лише після завершення анімації здійснювався вплив на об’єкт-противника. Така система синхронізації забезпечувала візуальну узгодженість та давала гравцеві чіткий зворотний зв’язок щодо того, що відбувається на полі бою.

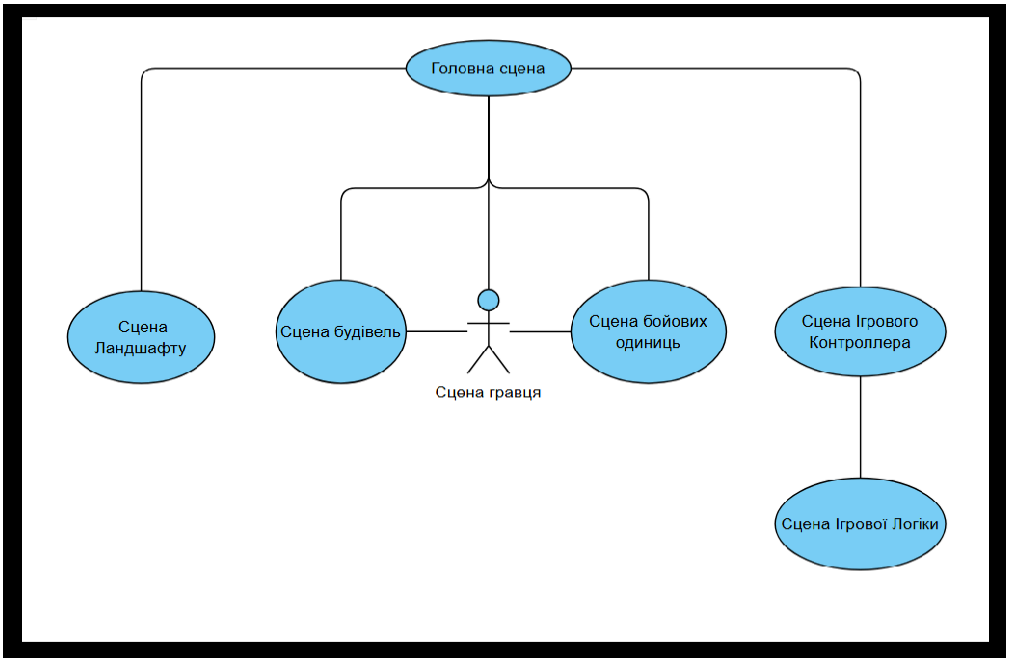

Рис. 2.5. Діаграма зв’язку сцен

Система менеджменту станів також була тісно пов’язана з подієвою моделлю гри. Кожна зміна стану або ключова подія (наприклад, смерть ігрового об’єкта, завершення будівництва, досягнення точки призначення) генерувала сигнал, яка розповсюджувалась серед зацікавлених об'єктів. Це дозволяло гнучко реагувати на зміни в ігровому середовищі — наприклад, система UI могла оновити панель інформації про будівлю, як тільки та змінила стан.

Загалом побудова системи менеджменту станів і об'єктів у межах розробки RTS-гри в Godot стала фундаментом для надійної та масштабованої ігрової логіки. Вона забезпечила чисту архітектуру, зручне розширення, тестування та адаптацію, що особливо важливо в умовах великої кількості взаємодіючих об’єктів. У подальших розділах буде показано, як ці системи взаємодіють із механіками навігації, побудови та бойових дій.

2.3. Створення сцени для тестування всіх механік та підтримки найпопулярніших ігрових маніпуляторів

На етапі початкової реалізації основних систем гри постала потреба у створенні єдиної, функціонально незалежної сцени, яка б дозволяла протестувати ключові ігрові механіки у середовищі, максимально наближеному до ігрового простору. Я вирішив реалізувати тестову сцену, яка слугувала б не лише полігоном для перевірки працездатності функцій, а й гнучким інструментом для налагодження взаємодії між підсистемами, включно з менеджментом об'єктів, станів, керуванням юнітами, побудовою споруд і обробкою ресурсів.

Основною вимогою до цієї сцени було забезпечення універсального середовища, в якому я зможу перевіряти як ізольовані блоки логіки, так і взаємодії між ними без потреби в інтеграції з повноцінною картою чи кампанією. Для цього я створив умовну "лабораторну" мапу з мінімальним ландшафтом, що включала кілька площин з різними зонами дій. У таких зонах я розміщував контрольованих юнітів, інтерактивні споруди, об'єкти для збирання ресурсів і джерела введення.

На рівні коду я реалізував можливість завантаження до сцени будь-якого з необхідних об’єктів без перезапуску гри, що значно пришвидшило процес тестування. Завдяки підключенню дебагових панелей із виводом поточних станів, подій і логічних переходів між ними, я міг оперативно виявляти помилки і проводити профілювання продуктивності кожної системи окремо.

Додатково мною було реалізовано логування подій у вигляді внутрішньої консолі, що дозволило фіксувати зміну станів об’єктів, результати взаємодії з ресурсами, виконання побудов і колізійні події. У такий спосіб я забезпечив собі стабільне і кероване середовище для поетапної перевірки та ітеративної розробки основних функціональностей гри.

Завдяки наявності сцени для тестування всіх механік, розробка значно прискорилась, оскільки я мав змогу здійснювати локальні зміни, не впливаючи на глобальні ігрові елементи. Крім того, такий підхід створив основу для

майбутніх автоматизованих тестів та перевірок коректності роботи нових механік у контрольованих умовах.

У межах згаданої сцени було виокремлено кілька функціональних зон, кожна з яких дозволяла зосередитись на конкретних аспектах логіки гри та взаємодій:

- Зона базового управління бойовими одиницями. В цій частині сцени розміщувалися кілька окремих юнітів із різними шаблонами поведінки, що дозволяло перевірити правильність обробки команди на переміщення, взаємодію з оточенням, вибір траєкторії та уникнення перешкод.
- Ділянка розміщення споруд. Тут я тестував побудову об’єктів за заданими параметрами — з урахуванням доступної площі, наявності ресурсів і перевірки коректності анулювання процесу будівництва. Також у цій зоні перевірялася прив’язка споруд до внутрішніх систем гри, таких як генерація ресурсів або формування гарнізонів.
- Сектор збирання ресурсів. Окремий простір було відведено під розміщення джерел ресурсів різного типу (дерево, золото, ядро), а також логіки їхнього видобування, споживання та виведення інформації на інтерфейс. Тут проводилася перевірка циклу взаємодії: юніт — ресурс — база.
- Область зіткнень та колізій. Ця зона призначалася для перевірки фізичних обмежень пересування, адекватної реакції на бар’єри, впливів з боку інших об’єктів, а також коректної взаємодії між різними типами юнітів, споруд та елементів середовища.
- Простір для моделювання сценаріїв зі зміною станів. У цьому фрагменті сцени тестувалися переходи між станами штучного інтелекту: спокій, бойова готовність, відступ, добування ресурсів. Окрему увагу приділено коректності збереження й відновлення контексту при зміні поточної задачі.
- Сегмент перевірки інтерфейсної взаємодії. Для тестування системи вибору об'єктів, відображення інформації про них, активації контекстних

дій та виведення повідомлень я виділив окрему ділянку з відповідною структурою інтерфейсних компонентів, синхронізованих із подіями в ігровому просторі.

Наявність цих локалізованих зон у межах однієї сцени забезпечила можливість незалежного та гнучкого тестування, що стало основою ефективного налагодження як окремих підсистем, так і їх інтеграційних взаємодій.

Зважаючи на сучасні тенденції ігрової індустрії, де користувачі очікують зручної підтримки широкого спектра пристроїв введення, я поставив перед собою завдання реалізувати базову, але гнучку систему підтримки найпопулярніших ігрових маніпуляторів, таких як геймпади Xbox, PlayStation та інші сумісні з XInput і SDL.

Godot Engine надає вбудовану підтримку геймпадів через систему InputEventJoypad, однак її використання потребує додаткового програмного забезпечення логіки контролю, оскільки структура подій відрізняється від клавіатурного чи машинного введення. Для початку я детально ознайомився з документацією Godot щодо Input Mapping та призначив дублюючі дії для усіх основних функцій гри — вибору одиниць, пересування камери, активації меню, підтвердження дій тощо.

Я реалізував внутрішній шар абстракції між логікою введення та поведінкою гравця. Це дало мені змогу не прив’язуватись до конкретного типу введення — чи то клавіатура, чи то геймпад, — а використовувати єдину структуру введення, яку я можу підключити до будь-якого пристрою. Усі дії виводились до окремого input manager, який дозволяв мені зручно призначати, змінювати або імітувати події введення для тестування.

Під час налаштування геймпадів я також враховував розбіжності в розкладках і номерах кнопок, характерних для контролерів різних платформ. Для зручності я реалізував механізм автоідентифікації контролера за його ім’ям, з подальшим налаштуванням відповідного профілю керування. У майбутньому

це дозволить автоматично підтримувати різноманітні пристрої без потреби ручної конфігурації з боку користувача.

Особливу увагу я приділив адаптації інтерфейсу гри до керування геймпадом. Наприклад, я додав підказки кнопок на екрані, можливість навігації між елементами за допомогою стиків і D-pad, а також динамічну зміну іконок кнопок у залежності від типу підключеного контролера.

Таким чином, завдяки впровадженню підтримки геймпадів на ранньому етапі розробки, я заклав фундамент для комфортного геймплею на різних платформах, що підвищує універсальність мого проєкту та розширює потенційну аудиторію гравців.

2.4. Механіка побудови споруд та збору ресурсів

Побудова споруд є однією з ключових механік в іграх жанру RTS, адже саме вона дозволяє гравцеві розширювати базу, здобувати ресурси, створювати нові одиниці та зміцнювати оборону [11]. У контексті розробки гри на рушії Godot ця система повинна бути реалізована з урахуванням інтуїтивності, гнучкості та технічної оптимізації. Розробка механіки будівництва включала в себе кілька етапів: створення інтерфейсу вибору споруд, візуалізацію процесу розміщення, перевірку коректності позиціонування, облік ресурсів, поетапне зведення будівель та інтеграцію з іншими системами гри.

Реалізація почалася зі створення інтерфейсу побудови. У нижній частині екрану було розміщено окрему панель, яка відображала доступні для гравця типи споруд: шахти, казарми, вежі, житлові будівлі тощо. Кожна іконка відповідала певному типу будівлі та містила інформацію про її вартість і призначення. При натисканні на іконку гравець переходив у режим побудови: курсор миші змінювався, а на полі з’являвся «привид» будівлі — напівпрозоре зображення, яке слідкувало за рухом миші та показувало попередній вигляд споруди в обраній точці.

Цей привид виконував декілька функцій. По-перше, він забезпечував візуальне попереднє уявлення, дозволяючи гравцеві зрозуміти, як споруда

виглядатиме в конкретному місці. По-друге, колір привиду змінювався залежно від можливості встановлення будівлі: зелений — дозволене місце, червоний — заборонене. Для цього було реалізовано систему перевірки колізій. Привид мав спеціальний шар колізії, який перевіряв наявність інших об'єктів, ресурсів, рельєфу або меж карти. Якщо в межах області розміщення виявлялися перешкоди, то будівництво було заборонене.

Після вибору коректної позиції та натискання кнопки миші, починався процес будівництва. На цьому етапі в гру вступала система ресурсів. Перед розміщенням перевірялася наявність у гравця достатньої кількості ресурсів — дерева, каменю, золота або інших одиниць, залежно від складності економіки. Якщо ресурси були в наявності, їхня кількість зменшувалась, і на мапі з'являвся об'єкт будівництва — вже не привид, а повноцінна сцена з унікальним життєвим циклом.

Будівля починала своє існування в стані зведення. Візуально це супроводжувалося поступовим збільшенням непрозорості, появою конструкційних деталей або робітників, які здійснювали будівництво. У коді будівлі був закладений таймер або логіка прогресу, що оновлювалась з кожним кадром. За бажанням, до цього етапу могли підключатися юніти-будівельники, які мусили взаємодіяти з будівлею, прискорюючи або дозволяючи її зведення. Таким чином, гра дозволяла реалізувати як автоматичне, так і вручну кероване будівництво, що відкривало широкі можливості для стратегічного дизайну.

Після завершення будівництва будівля переходила в активний стан, змінювала свій вигляд, вмикала колізійні властивості й починала виконувати свою функцію — наприклад, генерувати ресурси, тренувати війська або відкривати нові можливості для гравця. У коді це означало активацію відповідного скрипта або зміни стану об’єкта. Важливо, що споруда ставала повноцінним учасником ігрового середовища — її можна було атакувати, взаємодіяти з нею, знищувати чи ремонтувати.

Окрема увага приділялася логіці скасування побудови. Якщо гравець передумав до моменту розміщення будівлі, він міг натиснути клавішу Escape

або клікнути правою кнопкою миші, щоб вийти з режиму побудови. Якщо ж споруда вже будувалася, але гравець вирішив її скасувати, частина ресурсів поверталася назад. Це забезпечувало додаткову гнучкість у стратегічних рішеннях і дозволяло уникнути фрустрації від випадкових дій.

У проєкті було також закладено основу для подальшого масштабування системи побудови. Всі типи будівель створювалися як окремі сцени, що успадковували базовий клас BuildingBase. Це дозволяло легко додавати нові типи споруд з унікальною поведінкою, не змінюючи загальну архітектуру. Крім того, уся логіка побудови (інтерфейс, перевірки, ресурсна система) була винесена в окремий модуль, що забезпечувало повторне використання коду в майбутніх проектах або модулях гри.

Таким чином, механіка побудови споруд стала важливою частиною загального дизайну гри. Вона забезпечила гравцеві стратегічну глибину, дозволивши впливати на економіку, оборону, розвиток і динаміку боїв. Завдяки ретельному плануванню, технічному розмежуванню відповідальностей і продуманому UX-орієнтованому підходу, ця система стала надійною, зрозумілою та масштабованою основою для подальшого розвитку RTS-механіки у грі.

Також доволі важливим аспектом є добре спроєктована та збалансована система ресурсів. Вона формує основу для економічного розвитку, впливає на темп гри, визначає стратегічні рішення гравця і безпосередньо пов’язана з іншими механіками — побудовою, виробництвом одиниць, апгрейдами тощо. У межах реалізації RTS-механіки в Godot Engine було створено систему збору ресурсів, яка охоплювала джерела ресурсів у світі, взаємодію з робочими одиницями, зберігання та подальшу обробку.

Початковим кроком було визначення типів ресурсів, які будуть використовуватись у грі. У базовій конфігурації було передбачено декілька основних категорій: дерево, камінь, залізо та золото. Кожен із них мав своє призначення і добувався з конкретного джерела у світі. Наприклад, дерево

здобувалося з лісових масивів, камінь — із покладів скель, а золото — з родовищ руди. Всі ці об'єкти були реалізовані як окремі сцени з власними властивостями: обсягом ресурсу, швидкістю добування, доступністю для декількох робітників одночасно.

У рамках розробки ігрової системи я визначив три основні типи ресурсів, кожен з яких виконує унікальну функцію в ігровій економіці та напряму впливає на стратегічні рішення гравця. Це — Дерево, Золото та Ядро. Вибір саме цих ресурсів був зумовлений бажанням створити зрозумілу, але водночас гнучку та стратегічно насичену систему прогресу.

1. Дерево виступає базовим ресурсом, який є фундаментом для первинної розбудови. Його основне призначення — слугувати сировиною для спорудження початкових будівель, оборонних веж, а також деяких видів юнітів. Я реалізував систему збору дерева через автоматизацію його збору на головній споруді гравця. Збирання здійснюється автоматично, та не використовує нічого специфічного, приріст до швидкості отримання цього ресурсу залежить від кількості будівель на території гравця. Кожен об’єкт має параметр міцності, що зменшується з кожним циклом збору.
2. На рівні ігрової реалізації Золото добувається через знищення ворожих споруд та ворожих бойових одиниць. Його функціональне призначення полягає у створенні просунутих бойових юнітів, розвитку технологій, розблокуванні нових можливостей або розбудові потужніших споруд.
3. Ядро є рідкісним і винятковим ресурсом, який можна отримати тільки після перемоги над спеціальним босом. У рамках мережевої архітектури впроваджено механізм персонального контролю боса для кожного гравця. Інформація про оновлення балансу синхронізується з усіма гравцями для відображення загального лічильника, однак використовувати цей ресурс може тільки його власник.

Щоб уникнути перевантаження сцени великою кількістю індивідуальних об’єктів, було реалізовано оптимізацію збору: кожен ресурс мав обмежену

кількість доступних місць для одночасної роботи, і нові юніти не могли добувати ресурс, якщо всі доступні місця були зайняті. Це змушувало гравця раціонально розподіляти ресурси та уникати накопичення неефективних дій. Крім того, джерела ресурсів поступово вичерпувалися — з кожним збиранням зменшувалася внутрішня змінна, і коли вона досягала нуля, об’єкт зникав або змінювався на «вичерпаний», втрачаючи функціональність.

Після доставки ресурсу до складу, загальна кількість доступного ресурсу змінювалась у глобальному лічильнику, що відображався в UI. Цей лічильник був пов’язаний із системами побудови та прокачування, забезпечуючи пряму залежність між економікою та розвитком гравця. Було також реалізовано механізм витрат: під час будівництва або тренування бойових одиниць ресурси списувалися з лічильника, а в разі їх нестачі відображалося попередження, і дія блокувалася.

Обробка ресурсів як окремий етап стала наступним логічним кроком. Щоб надати гнучкості гравцеві, було реалізовано можливість автоматизації процесу: ресурси передавалися між складами, робітники самостійно обирали доступні будівлі для обробки, а пріоритети виробництва можна було змінювати через інтерфейс. Це дозволяло створювати логістичні ланцюжки — наприклад, один склад зберігав сировину, інший займався обробкою, а третій — зберігав готову продукцію для використання у будівництві чи торгівлі.

Інтеграція системи збору ресурсів з іншими частинами гри відбувалася через сигнальну систему Godot. Події типу «ресурс зібрано», «ресурс вичерпано» або «виробництво завершено» викликали відповідні реакції в UI, у логіці будівель та у штучному інтелекті. Це забезпечувало узгоджену роботу всієї системи, дозволяючи створити живий світ, де економіка реагує на дії гравця і вимагає постійного планування.

2.5. Інтерфейс користувача для управління гравцем

Інтерфейс користувача у стратегіях реального часу є одним із ключових інструментів, що визначають комфорт гравця, швидкість прийняття рішень, зручність керування ігровими одиницями та ефективність взаємодії з усіма ігровими механіками. Основна задача цього компонента полягає у тому, щоби подати всю важливу інформацію в доступному вигляді та дозволити гравцеві швидко та інтуїтивно виконувати потрібні дії. Під час реалізації інтерфейсу для RTS-механіки в Godot Engine головна увага приділялася поєднанню функціональності, естетики, адаптивності та взаємодії з внутрішніми системами гри [13].

Інтерфейс було побудовано на базі системи Control та CanvasLayer, що дозволяють створювати незалежні від 2D/3D-сцен UI-елементи, які завжди залишаються на одному шарі перед камерою. Це дозволило уникнути зміщень чи зсувів при зміні положення гравця у просторі. Основні елементи інтерфейсу включали: панель ресурсів, панель вибраних одиниць, нижню інформаційну панель з діями, мінікарту, а також повідомлення про події.

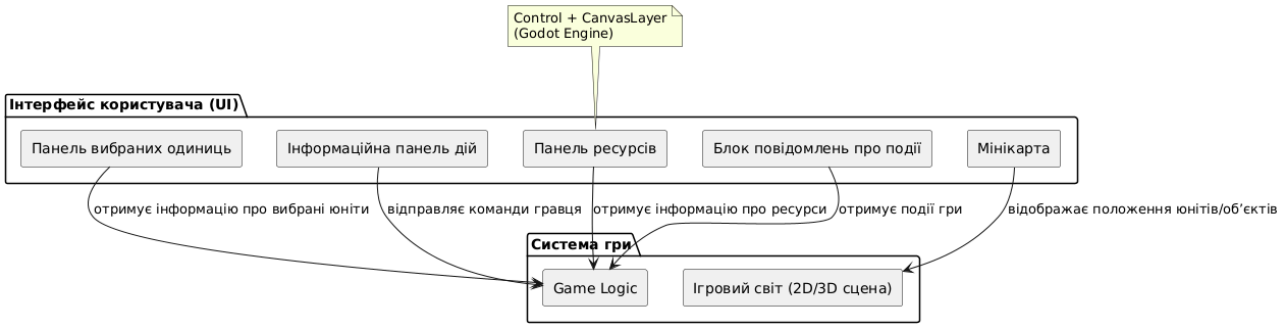

Рис. 2.6. Структура інтерфейсу користувача

Панель ресурсів знаходилась у верхній частині екрана і відображала загальну кількість усіх наявних ресурсів: дерева, каменю, заліза, золота та їжі. Для кожного ресурсу була створена окрема секція з відповідною іконкою та числовим значенням. Ці значення динамічно оновлювались через сигнальну систему: при кожному додаванні або витраті ресурсу викликався сигнал, що

оновлював відображення даних без потреби в щосекундному опитуванні стану. Крім того, у випадку дефіциту ресурсу або негативного балансу (наприклад, більше споживання їжі, ніж її виробництво) відображалась спеціальна анімація попередження.

Панель вибраних одиниць активувалась у нижньому лівому куті при натисканні на кнопку миші або виділенні об’єктів. Вона динамічно відображала усі вибрані об’єкти із мініатюрними портретами, іконками стану (переміщується, атакує, добуває, бездіє), а також окремою панеллю зі спільними командами. Якщо вибрані одиниці мали однаковий тип, відкривалася панель специфічних дій — наприклад, для робочих це могли бути кнопки будівництва, збору ресурсів або ремонту. Всі команди, що виконувались через UI, були напряму прив’язані до системи станів підконтрольних персонажів, що забезпечувало миттєву реакцію на дії гравця.

Центральна частина нижнього інтерфейсу була призначена для взаємодії з будівлями, спорудженням та іншими спеціальними діями. Ця панель змінювалася залежно від об’єкта, вибраного гравцем. Наприклад, при натисканні на головну будівлю вона відображала список доступних одиниць на ігровому полі, їхні атрибути, час створення. При натисканні на будівлю відображалися рецепти обробки ресурсів. Всі ці дії реалізовувались через уніфіковану систему кнопок з прив’язкою до відповідних функцій у логіці ігрових об’єктів.

Важливою складовою став візуальний зворотний зв’язок. Усі кнопки мали три стани — звичайний, наведення та натиснутий, що задавались через окремі теми і стилі. Після виконання дії відображалася коротка анімація підтвердження або повідомлення про помилку (наприклад, якщо бракує ресурсів). Також були передбачені гарячі клавіші для швидкого запуску певних дій, наприклад, Q, W, E, R — для базових команд, що значно прискорювало темп гри для досвідчених гравців.

Окремо реалізовувалася мінікарта, яка не лише відображала загальний стан карти, але й дозволяла взаємодію: клік по області мапи переміщував

камеру у відповідне місце, а кольорові маркери на мапі дозволяли орієнтуватися в розташуванні своїх і ворожих одиниць, ресурсів, споруд тощо. Для реалізації цього було створено окрему сцену з відображенням спрощеного масштабу ігрового світу у вигляді 3D-контейнера, синхронізованого з основною картою.

Інтерфейс був адаптивним, тобто масштабувався під різні роздільні здатності екрану. Було реалізовано прив’язку UI-елементів до відповідних куточків або областей екрана, а також можливість змінювати розмір елементів залежно від роздільності. Це дозволило забезпечити сумісність як для віконного режиму, так і для повноекранного, з урахуванням різних пропорцій екрану.

Особливу увагу було приділено тестуванню UX — інтерфейс перевірявся на різні сценарії використання, включно з виділенням великої кількості одиниць, одночасною побудовою кількох будівель, швидким натисканням кнопок і комбінованими діями (наприклад, виділення, атака, скасування, переміщення). Усі виявлені конфлікти або збої були усунені, а логіка інтерфейсу приведена до уніфікованої структури.

Таким чином, інтерфейс користувача став важливою частиною ігрового досвіду, об’єднуючи всі ключові механіки гри, дозволяючи гравцеві керувати подіями швидко, інтуїтивно та без зайвих зусиль. Його побудова в Godot Engine показала гнучкість інструментів для UI та їхню придатність до реалізації складних стратегічних систем з численними взаємодіями.

2.6. Використання модульного програмування та його поєднання з об'єктноорієнтованим підходом

У процесі розробки складних ігор у жанрі стратегії в реальному часі особливу роль відіграє правильна організація коду. Для досягнення гнучкості, зручності розширення та підтримки проєкту необхідно застосовувати сучасні принципи розробки програмного забезпечення. Одними з найважливіших підходів, що забезпечують ці цілі, є модульне програмування та об'єктноорієнтований дизайн (ООП). Їхнє поєднання у середовищі Godot Engine

дає змогу створити надійну архітектуру гри, яка буде стійкою до змін та легкою в обслуговуванні.

Модульне програмування полягає у розбитті програми на окремі частини — модулі, які виконують певні функції незалежно один від одного. У контексті RTS гри це означає, що такі компоненти, як логіка керування одиницями, механіка побудови споруд, система збору ресурсів, інтерфейс користувача та інші, реалізуються окремо. Кожен модуль має власний набір відповідальностей і чітко визначені межі, що значно спрощує розуміння і контроль над системою. Завдяки такому поділу розробникам легше підтримувати й удосконалювати кожен блок без ризику порушити інші частини проєкту.

Використання модульного підходу підвищує зручність командної роботи. Різні члени команди можуть одночасно працювати над окремими модулями, не заважаючи один одному. При цьому в результаті відсутність жорсткої залежності між компонентами сприяє уникненню конфліктів і спрощує інтеграцію. Для розробника стає простішим відстежувати помилки, проводити локальне тестування, а також здійснювати перевірку логіки гри без значних витрат часу.

Об'єктноорієнтований підхід базується на використанні класів та об’єктів, які інкапсулюють дані й поведінку. Основними принципами ООП є наслідування, поліморфізм, інкапсуляція та абстракція. У Godot Engine це дозволяє створювати ієрархії класів, де базові класи визначають загальний функціонал, а похідні класи — спеціалізовані варіанти об’єктів з унікальними характеристиками. Наприклад, базовий клас "BaseUnit" може містити спільні атрибути, такі як швидкість пересування, здоров’я, методи атаки і захисту, а класи "Піхотинець", "Броненосець" або "Артилерія" — розширювати цей функціонал з урахуванням специфіки кожного виду.

Поєднання модульного програмування з ООП забезпечує створення масштабованої структури коду. Модулі стають логічними контейнерами для класів і допоміжних скриптів, які взаємодіють між собою через визначені інтерфейси. Це дозволяє уніфікувати підхід до розробки, уникаючи дублювання

коду та забезпечуючи повторне використання компонентів у різних частинах гри. Наприклад, один і той самий клас ресурсу може бути використаний як у системі збору, так і в механіці торгівлі.

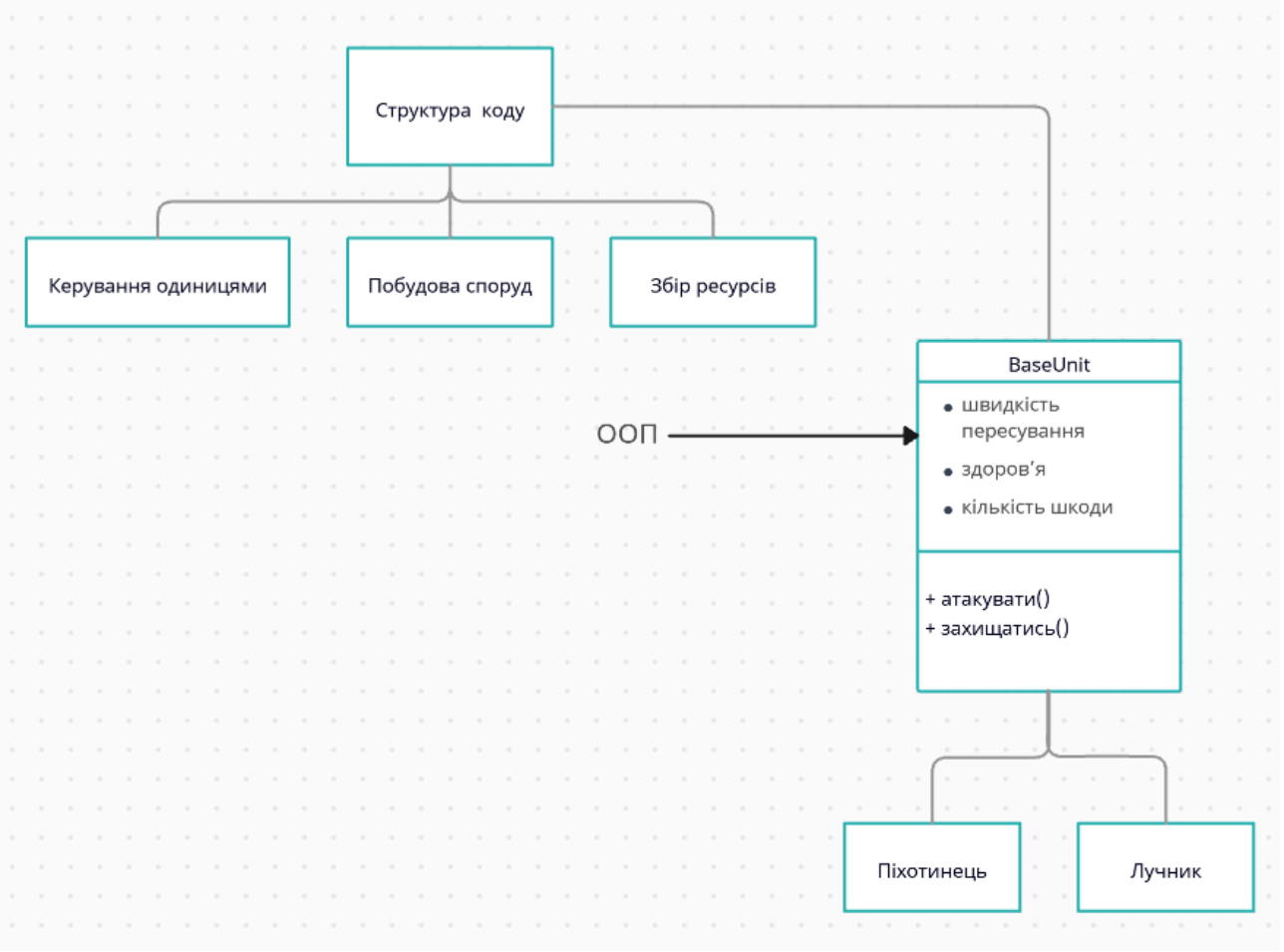

Рисунок 2.7. Поєднання модульного програмування та ООП

Особливо важливим є те, що такий підхід значно полегшує внесення змін і додавання нового функціоналу. При розробці RTS гри часто виникає потреба додати нові типи одиниць, споруд чи ресурсів, а також змінити існуючі механіки відповідно до ігрового балансу або відгуків користувачів. Модульна структура та ООП дозволяють робити це без необхідності переписувати великий шматок коду, адже розробник може створити новий клас або модуль, наслідуючи існуючі, і інтегрувати їх у загальну систему.

Крім того, у Godot Engine для реалізації модульної архітектури й ООП часто використовується сцено-орієнтований підхід, де сцени слугують

своєрідними модулями з набором скриптів та ресурсів. Такий підхід відповідає принципам модульності, оскільки сцени можуть бути автономними, а їхнє поєднання формує складніші ігрові системи. Це надає додаткову гнучкість, дозволяючи динамічно завантажувати і замінювати компоненти під час виконання гри.

Застосування подібної архітектури також впливає на тестування. Модульний код легше піддається тестуванню, оскільки кожен модуль або клас може бути протестований окремо від інших, що забезпечує вищу якість і стабільність проєкту. Крім того, чітко структуровані модулі спрощують виявлення та усунення помилок.

Варто відзначити, що поряд із перевагами, модульний і ООП підхід вимагає ретельного планування і дисципліни у написанні коду. Необхідно продумувати межі модулів, правильно визначати інтерфейси та залежності, аби уникнути надмірної зв’язності, яка знижує переваги модульності. Важливо також дотримуватися єдиних стандартів кодування, що допоможе зберегти читабельність і зручність роботи над проєктом у команді.

Таким чином, використання модульного програмування у поєднанні з об'єктноорієнтованим дизайном у Godot Engine є фундаментальним аспектом при розробці механіки RTS гри. Цей підхід забезпечує ефективну організацію коду, сприяє масштабованості проєкту, полегшує підтримку і розширення функціоналу, а також покращує якість кінцевого продукту.

РОЗДІЛ 3
ОПТИМІЗАЦІЯ, ТЕСТУВАННЯ, ПІДГОТОВКА ДО МАСШТАБУВАННЯ

3.1. Оптимізація продуктивності в Godot Engine

При створенні стратегічної гри в реальному часі з використанням Godot Engine важливо не лише реалізувати механіки, а й забезпечити високу продуктивність, особливо при великій кількості об’єктів, активних юнітів, частих оновленнях станів та взаємодії з численними системами. Оптимізація продуктивності не є етапом, який виконується лише наприкінці розробки, а постійним процесом, що супроводжує створення всіх систем гри. Вона вимагає комплексного підходу, що охоплює скриптову логіку, оновлення сцен, використання ресурсів, мережеву взаємодію та графічну відтворюваність.

Однією з ключових компонентів оптимізації є мінімізація частоти оновлення непотрібних процесів. У багатьох випадках спостерігається помилка, коли логіка великої кількості об’єктів оновлюється кожен кадр, хоча це не має необхідності. У моєму проєкті було запроваджено системи з використанням Timer, process_mode, а також подій, які спрацьовують лише при зміні стану. Наприклад, оновлення показників здоров’я або зміна стану об’єкта активувалося лише при настанні відповідної події, а не щосекундно чи покадрово. Це значно зменшило навантаження на CPU при масштабних ігрових ситуаціях, коли на сцені могли бути сотні активних об’єктів.

Також було проведено аудит використання сигналів. Замість постійного опитування стану об'єктів, як це часто роблять початківці, було запроваджено механізм сигналів, які генеруються лише при реальній зміні стану. Наприклад, юніт не перевіряє кожен кадр, чи отримав нову команду, а реагує на сигнал від менеджера, що сповіщає його про зміну поведінки. Такі практики дозволили уникнути надмірної кількості непотрібних викликів функцій і циклів, що особливо важливо при великій кількості об’єктів.

Графічна оптимізація також відіграла суттєву роль. Було прийнято рішення використовувати комбіновані спрайти та анімації замість складної 3D-графіки, оскільки основною ціллю було забезпечити стабільну продуктивність на широкому спектрі пристроїв. Усі спрайти зберігалися у вигляді атласів, що дозволило мінімізувати кількість текстурних перемикань при рендері. До того ж Godot дозволяє групувати об’єкти в CanvasItemGroup, що зменшує кількість обчислень для рендерингу та дозволяє використовувати батчинг. Також використовувались VisibilityNotifier та OcclusionCulling, щоб уникнути оновлення об'єктів, які не знаходяться в полі зору гравця.

Особлива увага була приділена обробці колізій та фізики. У грі використовувалися Area3D замість RigidBody3D для більшості логічних перевірок, оскільки Area3D менш ресурсомісткий та не вимагає складних фізичних обчислень. При перевірці наявності ворогів або ресурсів юніти використовували overlapping_areas, які активувались лише в моменти потреби, а не весь час. Це дозволило суттєво знизити навантаження при масштабних бойових діях, де одночасно взаємодіє багато об’єктів.

Скрипти були оптимізовані шляхом зменшення кількості вкладених функцій, уникнення глибокої рекурсії, а також попередньої ініціалізації об’єктів [2]. Наприклад, всі повторювані функції були винесені у менеджери або утилітарні класи, і використовувались через делеговані виклики. Також активно використовувались пули об’єктів (object pooling) — замість того, щоб постійно створювати та знищувати об’єкти, наприклад, кулі або ресурси, система повторно використовувала вже створені екземпляри, що значно зменшувало витрати на пам’ять і зменшувало кількість GarbageCollection-пауз (зупинок для збирання сміття).

Окремий аспект оптимізації — правильне використання ієрархії сцен. Уникаючи надмірної глибини вкладення вузлів та використовуючи компонування через скриптові зв’язки, було досягнуто більш швидкого завантаження сцен і кращої роботи в редакторі. Також сцени, які мали бути постійно доступними, але не видимими на сцені, завантажувалися асинхронно з

використанням ResourceLoader.load_interactive, що дозволяло уникати зменшення продуктивності під час гри.

Проведення профілювання за допомогою вбудованого інструмента Debugger та вкладки Profiler у Godot Engige дозволило наочно визначити вузькі місця продуктивності. За результатами профілювання були виявлені та оптимізовані кілька функцій, які займали найбільше часу, зокрема пов’язані з частим оновленням UI та запитами до багатьох об’єктів одночасно. На основі цих спостережень була реалізована система кешування часто використовуваних значень, що дозволило уникнути дублювання запитів.

Завдяки всім цим заходам вдалося забезпечити стабільну частоту кадрів навіть на слабших пристроях. При тестуванні на типовій машині з інтегрованою графікою гра демонструвала стабільні 60 FPS навіть при значному навантаженні: одночасній побудові кількох споруд, великій кількості активних юнітів та виконанні бойових дій.

Таким чином, оптимізація продуктивності у Godot Engine стала не окремим етапом, а невід’ємною частиною всієї розробки, дозволяючи досягти стабільності, масштабованості та приємного досвіду для гравця незалежно від апаратної конфігурації його пристрою.

3.2. Тестування основних компонентів гри

Тестування є критично важливим етапом у розробці будь-якої гри, а особливо для стратегій реального часу, де багато систем взаємодіють одночасно і навіть незначні помилки можуть призводити до критичних збоїв або втрати балансу. В процесі розробки гри на базі Godot Engine тестування не обмежувалося лише виявленням несправностей. Воно містило в себе аналіз коректності роботи ігрових механік, логіки станів, взаємодії користувача з інтерфейсом, відповідності внутрішніх подій очікуваній поведінці, а також перевірку стабільності продуктивності в різних сценаріях.

Першочергово було зосереджено увагу на модульному тестуванні найважливіших частин гри. Основними об’єктами тестування стали системи

керування ігровими одиницями, менеджмент станів, обробка колізій, побудова споруд та інтерфейс користувача. Наприклад, для системи керування одиницями створювалися серії сценаріїв, де ігрові одиниці отримували команди руху, атаки, збору ресурсів, після чого перевірялася правильність обраної анімації, реакція на перешкоди, логіка обробки команд при одночасному надходженні кількох дій. Усі тести реалізовувалися в окремих сценах з максимальною ізоляцією, щоб уникнути впливу інших систем.

Крім перевірки окремих елементів, проводилося інтеграційне тестування, яке дозволяло виявити помилки, що виникають під час взаємодії між різними частинами гри. Наприклад, при взаємодії системи побудови споруд із системою ресурсів могло виникнути збої, якщо ресурс списувався до підтвердження побудови або будівля не з'являлася при повному завершенні процесу. Такі помилки виявлялися в інтегрованих тестових середовищах, де користувач проходив сценарії, що імітували реальний ігровий процес.

Особливе значення приділялося перевірці роботи інтерфейсу. Для цього були створені макети типових ігрових ситуацій: будівництво на переповненій карті, виділення групи юнітів з різними командами, натискання кнопок інтерфейсу під час бою або руху. Перевірялася точність реакції на натискання, відсутність затримок або дублювання команд. Також тестувалися крайні ситуації, такі як спроба побудувати споруду без ресурсів, натискання кнопки при відсутності активних об'єктів, або відкриття кількох вкладок одночасно. Усі ці ситуації аналізувалися з погляду дизайну та внутрішньої логіки обробки подій.

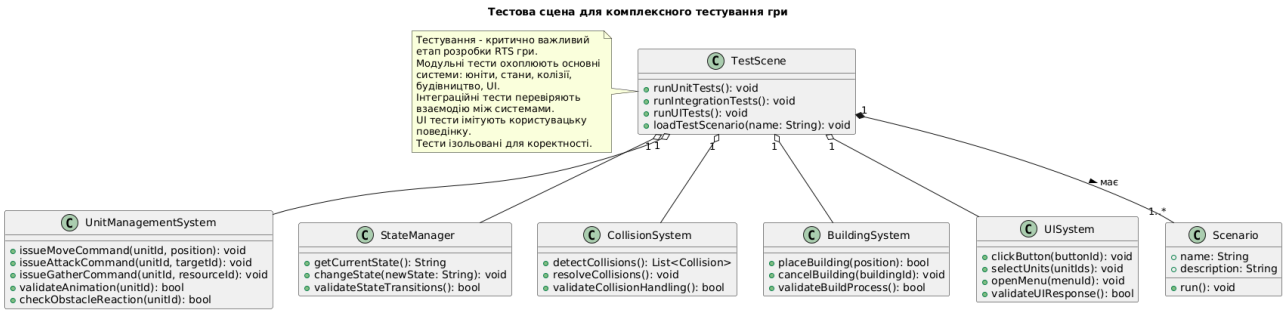

Рисунок 3.1. Структура сцени для тестування систем

Завдяки відкритій структурі Godot та використанню GDScript, вдалося реалізувати і просту систему логування всередині гри. Кожна ключова подія, така як зміна стану об’єкта, виклик функцій об'єкта, старт або завершення будівництва, зберігалася у внутрішній системі, що дозволяло мені відстежувати проблемні ситуації у реальному часі. Цей підхід виявився особливо корисним у випадках, коли поведінка об’єкта була непередбачуваною — логування дозволяло точно виявити, на якому етапі відбулася помилка, і за яких умов.

Окрім ручного тестування, я розглядав можливість використання автоматичних юніт-тестів через Godot Unit Test Framework або написання кастомних скриптів. Хоча GDScript не має таких розвинених засобів юніт-тестування, як деякі інші мови програмування, була створена базова структура, де об'єкти запускалися в симульованому середовищі та перевіряли відповідність результатів очікуваним. Наприклад, створювався тест на побудову споруди за обмежений проміжок часу з конкретною кількістю ресурсів, і система автоматично перевіряла, чи зменшилася кількість ресурсів, чи з’явилася споруда, і чи не виникло помилок у консолі. Такі тести дозволяли зменшити кількість ручної перевірки та спростити ре-факторинг коду, адже при внесенні змін можна було швидко впевнитися, що базові функції залишаються працездатними.

У фінальних тестах велику увагу я приділив стабільності та витривалості систем. Наприклад, запускалася велика кількість юнітів на мапі одночасно, здійснювалось масове будівництво, або виконувалося кілька сценаріїв гравця поспіль без перезавантаження сцени. Такі стрес-тести виявляли проблеми з очищенням пам’яті, витоками ресурсів, зависаннями анімацій, а також інші дрібні, але критичні несправності. У разі виявлення нестабільної поведінки, система створювала знімок сцени та стану об’єктів, що дозволяло відтворити ситуацію пізніше без потреби в точному повторенні дій вручну.

Таким чином, тестування стало не лише етапом верифікації роботи гри, а й повноцінною практикою контролю якості на всіх рівнях — від простих скриптів до складних взаємодій у грі. Завдяки системному підходу вдалося

створити стабільну основу для подальшого розвитку гри, не боячись, що нові функції зламають старі, а помилки стануть помітними лише гравцям.

3.3. Побудова інструментів для розробника

Розробка складної гри, зокрема стратегії реального часу, вимагає ефективної внутрішньої інфраструктури не лише для фінального продукту, але і для зручності самого процесу створення контенту. Побудова спеціалізованих інструментів для розробника у Godot Engine дозволила значно пришвидшити швидкість роботи, мінімізувати людський фактор у технічних помилках та забезпечити високу гнучкість під час прототипування нових функціональностей. У цьому підрозділі я розглянув ті інструменти, які були створені або адаптовані спеціально під потреби розробки RTS-механік, а також принципи, які лежали в основі їх побудови.

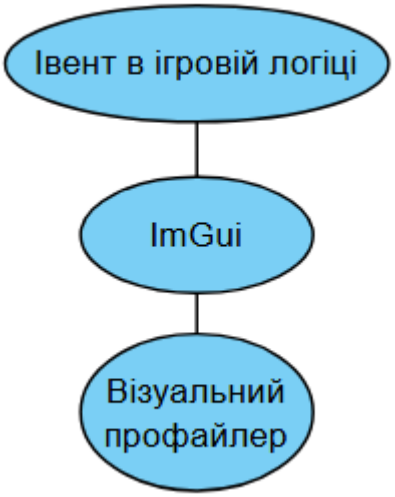

Рис. 3.2. Діаграма порядку виводу інформації для налагодження процесів гри

Першим і, безумовно, найбільш вагомим інструментом стала система додавання об'єктів на мапу. Для багатьох ігрових механік потрібна наявність початкових юнітів, будівель, ресурсів чи зон. Щоб не розміщувати їх вручну на

сцені кожного разу, було створено редактор шаблонів, де розробник міг вказати конфігурацію початкової локації у вигляді простої JSON-структури або списку у спеціальному редакторі у вікні інспектора. Після цього на сцену автоматично завантажувалися відповідні елементи із коректними властивостями. Це дозволило легко тестувати різні комбінації стартових умов або перевіряти геймплей у симульованих ситуаціях.

Наступним кроком стала розробка внутрішньої панелі налагодження, що включала реальний час запису даних стану ігрових об'єктів, активацію певних подій, ручне викликання функцій (наприклад, побудова споруди, зміна ресурсу, знищення об’єкта). Це було реалізовано через окрему вкладку в редакторі та активне GUI-поле поверх сцени. Головною метою цього інструменту було надати змогу швидко перевіряти, як працюють ті чи інші компоненти, без потреби зупиняти гру, переходити у код, змінювати властивості вручну і знову запускати сцену. Така інтерактивна взаємодія особливо допомогла на етапі тестування механіки менеджменту станів, де було необхідно швидко змінювати поведінку кожного ігрового елемента для перевірки переходів між режимами.

Окремо була реалізована система для візуального налагоджування зон дії юнітів і будівель. Наприклад, при наведенні на об'єкт відображалася його зона видимості, радіус атаки, або зона збору ресурсів. Це відображалося у вигляді кольорових кіл або полігонів, які автоматично оновлювалися під час зміни параметрів. Такий підхід дозволив точно налагодити логіку детекції та виявити багато ситуацій, де координати працювали некоректно або зсувалися внаслідок переміщень або масштабування. Крім того, це дозволило зробити процес створення нових об’єктів більш контрольованим — кожен новий юніт одразу перевірявся на відповідність очікуваним зонам взаємодії.

Ще одним важливим компонентом внутрішніх інструментів стала система шаблонів для об’єктів. У грі існує багато повторюваних структур — наприклад, споруди з однаковими базовими функціями, але різними параметрами або зовнішнім виглядом. Замість створення кожної з нуля, було реалізовано гнучку систему спадкування сцен, де батьківська сцена визначала основну логіку

(життєвий цикл, інтеракції, обробка подій), а дочірні сцени лише перевизначали конкретні властивості — текстури, характеристики, вартість. Це дозволило значно спростити процес додавання нових елементів у гру, зменшити кількість помилок, пов’язаних із дублюванням коду, та пришвидшити ітерації.

Також у процесі роботи було додано невелику систему генерації тестових ресурсів. Це була окрема утиліта, що автоматично створювала ресурсні вузли на мапі з випадковими параметрами (кількість, тип, координати), що дозволяло перевіряти логіку роботи AI та механік збору. Інструмент мав просте меню налаштувань — користувач міг обрати тип ресурсу, кількість вузлів, зону розміщення. Такий підхід дозволив відмовитися від статичної мапи ресурсів, постійно змінюючи умови тестування без редагування сцени вручну.

Для зручності взаємодії з великим обсягом юнітів на сцені, був створений внутрішній “Entity Manager” — інструмент, який виводив у режимі реального часу всі активні об'єкти, групував їх за типами, дозволяв швидко видаляти, телепортувати, дублювати чи змінювати параметри. Особливо корисним це стало при виникненні багів, пов’язаних із логікою чи колізіями, де потрібно було віднайти конкретний об’єкт серед десятків подібних. Entity Manager також став основою для реалізації майбутніх редакторських функцій на рівні гравця-користувача (наприклад, у редакторі мап).

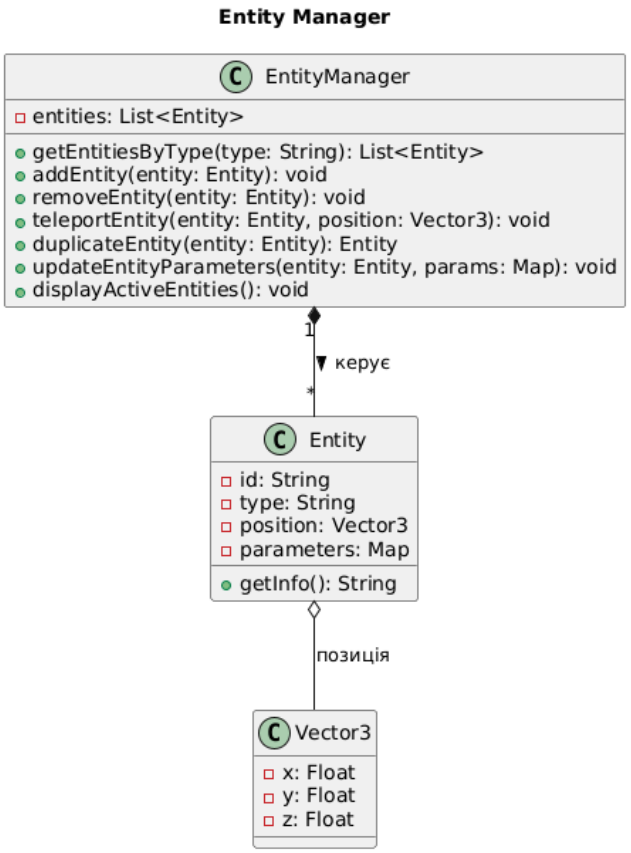

Рисунок 3.3. Система керування ігровими одиницями

Побудова цих інструментів стала результатом потреб, які виникали в ході розробки. Вони не лише пришвидшили сам процес програмування, а й значно зменшили складність тестування та покращили загальну якість реалізації ігрових механік. Завдяки Godot Engine, який дозволяє легко створювати редакторські компоненти і взаємодіяти з інспектором, побудова внутрішніх інструментів стала органічною частиною виробничого циклу. У фінальному результаті це дало можливість сфокусуватися на креативних аспектах гри, не витрачаючи час на рутинні дії.

3.4. Підготовка гри до розширення функціоналу

Ще одним із ключових аспектів успішної розробки складних проєктів є здатність системи легко адаптуватися та масштабуватися зі зростанням обсягу контенту і функціоналу. У контексті створення RTS-механік у Godot Engine це

особливо важливо, оскільки ігри такого жанру часто потребують додавання нових юнітів, споруд, ресурсів, а також розширення логіки штучного інтелекту та взаємодії гравця зі світом. Підготовка гри до розширення — це не лише технічне завдання, але і продумана архітектурна стратегія, яка враховує потенційні зміни та дозволяє мінімізувати витрати часу і ресурсів у майбутньому.

Першим кроком у цій підготовці стала чітка модульність коду. Всі основні компоненти — управління юнітами, будівництвом, ресурсами, інтерфейсом — були розділені на окремі класи та системи з добре визначеними інтерфейсами взаємодії. Це означає, що зміни у функціоналі одного модуля не впливають безпосередньо на інші, що зменшує ризик появи непередбачуваних багів. Такий підхід значно спрощує додавання нових механік, оскільки можна фокусуватися лише на конкретному компоненті без необхідності переписувати увесь код.

Додатково було впроваджено шаблони проєктування, такі як фабрики для створення об’єктів ігрового світу та менеджери подій, що полегшують організацію взаємодії між різними частинами системи. Завдяки фабрикам з’явилася можливість легко додавати нові типи юнітів або споруд, не змінюючи існуючий код, а лише створюючи нові класи, які наслідують базову логіку. Менеджери подій, у свою чергу, забезпечують слабке зв’язування між компонентами, дозволяючи підписуватися на події та реагувати на них без прямої залежності між об’єктами.

Ще одним важливим аспектом є використання ресурсних файлів (Resource) у Godot для зберігання параметрів юнітів, споруд, навичок та інших налаштувань. Такий підхід дозволяє легко змінювати характеристики без потреби змінювати код, що особливо зручно для дизайнерів і тестувальників. Крім того, це створює основу для майбутнього імпорту та експорту налаштувань, що може бути використано для створення редакторів або модифікацій гри.

Для збереження та завантаження стану гри була реалізована система серіалізації, яка дозволяє коректно зберігати стан усіх ключових об’єктів із

можливістю подальшого відновлення. Важливою вимогою до цієї системи було те, щоб вона була гнучкою і легко адаптувалася до додавання нових типів об’єктів або полів. Для цього було вибрано формат збереження на базі JSON із чітко визначеними схемами даних, що забезпечує сумісність та розширюваність.

Важливою частиною підготовки стало також документування архітектури і структури проекту. Чітка і доступна документація дозволяє не лише новим членам команди швидко орієнтуватися в коді, але і самим розробникам уникнути плутанини під час масштабування проекту. Документація включає опис ключових класів, інтерфейсів, взаємодій між модулями, а також правила і рекомендації щодо додавання нових елементів.

Не менш значущою є автоматизація тестування. Створення базового набору юніт-тестів і інтеграційних тестів для основних систем допомагає впевнено вносити зміни у код без страху зламати вже працюючі функції. Завдяки автоматизованому тестуванню розробник отримує швидкий зворотний зв’язок про коректність роботи системи при додаванні нових функціональних можливостей.

Таким чином, підготовка гри до розширення функціоналу — це комплекс заходів, які забезпечують стабільність, гнучкість і масштабованість проекту. Вона включає не лише технічні рішення, а й організаційні аспекти, які разом створюють надійну базу для подальшого розвитку гри, спрощують підтримку коду та інтеграцію нових ідей. Завдяки цьому розробка RTS-механік у Godot стала більш ефективною і прогнозованою, що є важливою складовою успіху будь-якого ігрового проекту.

3.5. Підготовка до випуску гри на платформі Steam

Підготовка до випуску гри на такій популярній платформі, як Steam [4], є складним і багатоступеневим процесом, який вимагає не лише завершення розробки, а й ретельного планування маркетингової, технічної та

адміністративної складових. Цей підпункт присвячено основним крокам, необхідним для успішного запуску проєкт RTS на цій платформі.

Перш за все, важливо підготувати ігровий продукт відповідно до вимог Steam, що включає створення облікового запису розробника, реєстрацію гри в Steamworks, завантаження бета-версії для тестування і налаштування сторінки продукту. Окрему увагу приділяють оформленню сторінки гри: розробка привабливого опису, вибір знімків з екрана, трейлерів і ключових особливостей, які допоможуть привернути увагу потенційних покупців.

Технічна підготовка включає інтеграцію Steamworks SDK, що дозволяє реалізувати підтримку досягнень, хмарних збережень, мультиплеєра, а також механізмів захисту від піратства. Особливо важливо протестувати гру на стабільність і сумісність із різними конфігураціями користувачів, щоб уникнути негативних відгуків після релізу.

Серед численних цифрових платформ саме Steam залишається найбільш привабливим середовищем для розміщення незалежних ігор. Варто зосередити увагу на ключових перевагах, які визначають доцільність вибору цієї платформи:

- Широка користувацька база. Steam має понад сто мільйонів активних користувачів щомісяця, що забезпечує надзвичайно високу видимість продукту навіть без значних маркетингових витрат на старті.
- Зручна система публікації. Платформа Steamworks надає розробникам повний інструментарій для управління грою: оновленнями, бета-тестуванням, аналітикою, а також контролем над розподілом версій гри залежно від регіону або мови.
- Інтегровані маркетингові можливості. Розробники можуть використовувати внутрішні рекламні інструменти, наприклад, акційні розпродажі, функцію «Wishlist», тематичні події та

персоналізовані рекомендації, що автоматично просувають гру відповідно до інтересів користувача.

- Інтеграція спільноти. Кожна гра на платформі має власний форум, сторінку обговорень, можливість залишати рецензії, ділитися знімками з екрана, модифікаціями або геймплейними відео. Це сприяє формуванню лояльної спільноти навколо проєкту.
- Підтримка автоматизованого оновлення. Користувачі завжди отримують актуальні версії гри без додаткових дій, що суттєво знижує ризики помилок, пов’язаних із несумісністю клієнта.
- Можливість реалізації додаткового контенту. Завдяки системі DLC, мікротранзакцій та майстерні Steam Workshop розробники можуть реалізовувати як платні доповнення, так і створення інструментів для спільноти, стимулюючи подальшу активність користувачів.
- Стабільна фінансова модель. Не зважаючи на комісію в 30%, Steam пропонує прозору і стабільну систему розрахунків, підтримку з боку Valve у випадку спірних ситуацій і багаторічний досвід взаємодії з незалежними розробниками.

Ці чинники визначають не лише високий потенціал початкових продажів, а й довгострокову життєздатність проєкту в екосистемі Steam. Інтеграція із цією платформою дозволяє зосередитися не лише на технічній реалізації гри, але й на її сталому комерційному розвитку.

Крім того, планується організація маркетингової кампанії, що включає створення спільноти навколо гри, взаємодію з блогерами та стрімерами, а також використання рекламних інструментів Steam. Цей комплекс заходів спрямований на максимальне охоплення аудиторії та забезпечення високого стартового продажу.

Таким чином, підготовка до випуску на Steam охоплює не тільки технічні аспекти, а й комплекс організаційних і маркетингових дій, що є ключовими для успішного запуску гри і подальшого розвитку проєкту.

ВИСНОВКИ

У межах виконання кваліфікаційної роботи я здійснив повноцінне проєктування, реалізацію та часткову апробацію базових і розширених механік для гри у жанрі стратегії в реальному часі (RTS) з використанням сучасного, відкритого ігрового рушія Godot Engine версії 4.4.1. Розроблений мною прототип, демонструє функціональну завершеність обраної архітектури, відповідає вимогам модульності, масштабованості та повторного використання коду, що є критично важливими аспектами при створенні сучасних ігор із високою складністю логіки взаємодії.

У процесі розробки я розглянув теоретичні засади організації RTS-механік, що включає управління ігровими одиницями, будівництво споруд, збір і переробку ресурсів, а також реалізацію інтерфейсів користувача. Особливу увагу приділено створенню гнучкої системи менеджменту станів об’єктів та підтримці ізольованого тестування кожного функціонального модуля. Ієрархія проєкту сформована відповідно до кращих практик: із суворим розділенням логіки, представлення та ресурсної обробки, що дозволяє в подальшому безперешкодно оновлювати гру, впроваджувати нові сценарії та компоненти.

Також було реалізовано систему модульного програмування із застосуванням принципів об’єктно-орієнтованого підходу. Це дало змогу зменшити дублювання коду, підвищити його читабельність і спростити розширення функціональності. Паралельно з цим створено сцену для тестування всіх основних і допоміжних механік, що стало ефективним інструментом для перевірки працездатності окремих систем у контрольованих умовах.

Окрему увагу, я приділив адаптації управління під найпоширеніші ігрові маніпулятори, зокрема геймпади, що суттєво підвищує доступність гри для широкої аудиторії. Здійснено перевірку коректної роботи керування як через мишу і клавіатуру, так і через альтернативні пристрої введення, з урахуванням стандартів UX для консолей та ПК.

Продуктивність гри було оптимізовано шляхом аналізу роботи основних вузлів рушія, використання інструментів профілювання та зменшення кількості оновлень у неактивних частинах сцени. Також реалізовано базовий інструментарій для внутрішнього налагодження, що дозволяє виявляти логічні помилки на ранніх етапах.

Розглянув можливості масштабування гри та розширення її функціоналу: заплановано введення нових типів юнітів, розширення системи ресурсів, введення багатокористувацького режиму та вихід гри в Steam. Проаналізовано процедуру публікації гри у Steam через Steamworks SDK, а також вимоги до оптимізації проєкту для широкого кола гравців.

У результаті цієї роботи я сформував повноцінну технічну базу для подальшої розробки повноформатної стратегії. Сформована архітектура дає змогу гнучко модифікувати будь-який з елементів без ризику порушення внутрішньої логіки гри, а також забезпечує сумісність із зовнішніми інструментами й можливість публікації на ігрових платформах. Усі поставлені цілі було досягнуто у повному обсязі, що підтверджує практичну цінність виконаної кваліфікаційної роботи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Документація Godot – гілка 4.4. Godot Engine. URL: https://docs.godotengine.org/en/stable/.
2. Machalewski, Tomasz & Marek, Mariusz & Ochmann, Adrian. (2023). Optimization of Parameterized Behavior Trees in RTS Games. 10.1007/978-3-031-23492-7_33.
3. Advanced state machine techniques in Godot 4. The Shaggy Dev. URL: https://shaggydev.com/2023/11/28/godot-4-advanced-state-machines/.
4. Steamworks SDK Documentation. Steamworks SDK. URL: https://partner.steamgames.com/doc/sdk.
5. Gamma E. et al. *Design patterns: Elements of reusable object-oriented software*. Boston: Addison-Wesley, 1995. 395 p.
6. Rollings A., Adams E. *Andrew Rollings and Ernest Adams on game design*. Berkeley, CA: New Riders, 2003. 719 p.
7. Chen, S. *Game Engine Architecture*. 3rd ed. Boca Raton: CRC Press, 2020. 1024 p.
8. Gleicher M. et al. *Foundations of game engine development: Volume 1, Mathematics and Physics*. Boca Raton: CRC Press, 2018. 760 p.
9. Nystrom R. *Game programming patterns*. Geneseo: Nystrom, 2014. 359 p. URL: https://gameprogrammingpatterns.com/ (дата звернення: 11.02.2025).
10. Адамс Е. *Основи дизайну комп'ютерних ігор*. Київ: Видавничий дім «Комп'ютерні технології», 2007. 320 с.
11. Розробка RTS ігор: основні механіки та виклики. *GameDev.UA*. URL: https://gamedev.ua/articles/rts-game-development/ (дата звернення: 04.10.2024).
12. Оптимізація продуктивності ігор на Unity та Godot. Dev.to. URL: https://dev.to/articles/game-performance-optimization-unity-godot/ (дата звернення: 24.03.2025).
13. Godot Engine: Real-Time Strategy game development tutorial series. YouTube. URL:

https://www.youtube.com/playlist?list=PLNCY6D2C91e6005E20F02A00E10515152 (дата звернення: 18.01.2025).

14. Designing a scalable RTS game architecture. Gamasutra. URL: https://www.gamasutra.com/view/feature/132475/designing_a_scalable_rts_game_.php (дата звернення: 16.02.2025).