ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УНІВЕРСИТЕТ ЕКОНОМІКИ ТА ПРАВА «КРОК»

КВАЛІФІКАЦІЙНА РОБОТА

Тема: «Комп’ютерна гра-скролер «Space Switch», з використанням data-oriented design»

Ступінь вищої освіти – бакалавр
Спеціальність – 122 «Комп’ютерні науки»
Освітня програма «Комп’ютерні науки»

ПОЯСНЮВАЛЬНА ЗАПИСКА

Виконав: здобувач 4 курсу
групи КН-21
Ярослав КИЗИК
Керівник: к. військ. н. доцент кафедри
комп'ютерних наук
Володимир ТРОЦЬКО

Засвідчую, що кваліфікаційна робота оформлена відповідно до ДСТУ 3008:2015 та не містить запозичень з праць інших авторів без відповідних посилань.
Здобувач: _____
(підпис)

м. Київ – 2025 рік

ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД

«УНІВЕРСИТЕТ ЕКОНОМІКИ ТА ПРАВА «КРОК»

ЗАТВЕРДЖУЮ:
завідувач кафедри
комп'ютерних наук
Сергій МІЧКІВСЬКИЙ
«____»____20___р

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Кизик Ярослав Дмитрович

Тема роботи	Комп'ютерна гра-скролер «Space Switch», з використанням data-oriented design
Номер та дата наказу про затвердження теми	№121-7 від 24 грудня 2024 року
Коротка постановка завдання	Розробити комп'ютерну гру на Unity з використанням архітектурної парадигми ECS
Посилання на джерела інформації (не більше п'яти найменувань, які рекомендує науковий керівник)	1. Unity: ECS Concepts // Unity Docs – URL: https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/ecs_core.html (дата звернення: 16.02.2025) 2. Marios Koutroumpas: A simple guide to get started with Unity ECS // Medium – URL: https://medium.com/gitconnected/a-simple-guide-to-get-started-with-unity-ecs-b0e6a036e707 (дата звернення: 16.02.2025) 3. MY.GAMES: Explaining (and making the leap) to ECS in Unity // Medium – URL: https://medium.com/my-games-company/explaining-and-making-the-leap-to-ecs-in-unity-b6d786464d72 (дата звернення: 16.02.2025)
Вимоги до кваліфікаційної роботи	Кваліфікаційна робота має передбачити теоретичне, системотехнічне або експериментальне дослідження складного спеціалізованого завдання або практичної проблеми в галузі комп'ютерних наук, яке характеризується комплексністю та невизначеністю умов і потребує застосування теорій і методів інформаційних технологій.

Дата видачі завдання 27 грудня 2024 р.

Керівник Володимир ТРОЦЬКО

Здобувач освітнього ступеня бакалавра Ярослав КИЗИК

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання	Примітка
Підготовчий етап			
1	Вибір напряму дослідження	02.12.2024 р.	*виконано*
2	Формування теми та призначення керівника	16.12.2024 р.	*виконано*
3	Затвердження теми кваліфікаційної роботи	23.12.2024 р.	*виконано*
4	Затвердження завдання на кваліфікаційну роботу	27.12.2024 р.	*виконано*
Основний етап			
5	Розробка концепції кваліфікаційної роботи	13.01.2025 р.	*виконано*
6	Підбір та вивчення джерел інформації з напряму дослідження. Огляд існуючих аналогів	20.01.2025 р.	*виконано*
7	Затвердження розширеної постановки завдання. Підготовка та подання керівникові розділу 1 кваліфікаційної роботи	10.03.2025 р.	*виконано*
8	Проєктування. Підготовка та подання керівникові розділу 2 кваліфікаційної роботи	24.03.2025 р.	*виконано*
9	Підготовка доповіді для експертизи стану виконання кваліфікаційної роботи (проміжний контроль)	31.03-04.04.2025 р.	*виконано*
10	Реалізація. Підготовка та подання керівникові розділу 3 кваліфікаційної роботи	07.04.2025 р.	*виконано*
11	Підготовка та подання керівнику першого варіанту всієї кваліфікаційної роботи	14.04.2025 р.	*виконано*
12	Доопрацювання кваліфікаційної роботи з урахуванням зауважень керівника та представлення керівникові доопрацьованого варіанту кваліфікаційної роботи	21.04.2025 р.	*виконано*
Завершальний етап			
13	Представлення рукопису для перевірки на плагіат	28.04-04.05.2025 р.	*виконано*
14	Підготовка презентації та доповіді на передзахист	05.05-11.05.2025 р.	*виконано*
15	Передзахист кваліфікаційної роботи	12.05-16.05.2025 р.	*виконано*
16	Доопрацювання роботи за результатами передзахисту	19.05-06.06.2025 р.	*виконано*
17	Експертиза роботи керівником та зовнішнім експертом	09.06-15.06.2025 р.	*виконано*
18	Доопрацювання доповіді та презентації для захисту	09.06-15.06.2025 р.	*виконано*
19	Захист кваліфікаційної роботи	16.06-22.06.2025 р.	*виконано*

Керівник Володимир ТРОЦЬКО

Здобувач освітнього ступеня бакалавра Ярослав КИЗИК

Кизик Я.Д. Розробка комп’ютерної гри «Space Switch» з використанням data-oriented design..

Пояснювальна записка кваліфікаційної роботи за спеціальністю 122 – Комп’ютерні науки (освітня програма – Комп’ютерні науки) СО Бакалавр. – ВНЗ .Університет економіки та права .КРОК., Навчально-науковий інститут інформаційних та комунікаційних технологій, кафедра комп’ютерних наук, Київ, 2025.

У даній роботі описано розробку комп’ютерної гри на Unity з використанням дата орієнтовного підходу

Ключові слова: Розробка, комп'ютерна гра, аркада, Unity, gamedev.
Табл 3. Рисунок 39. Бібліограф 46.

Kyzyk Y.D. Development of the computer game "Space Switch" using data-oriented design.

Explanatory note of qualification work in specialty 122 - Computer Science (educational programme - Computer Science), Bachelor's degree - University of Economics and Law "KROK", Educational and Research Institute of Information and Communication Technologies, Department of Computer Science, Kyiv, 2023.

It is described the development of a computer game on Unity using a data-oriented approach

Keywords: Development, computer game, arcade, Unity, gamedev.
Tables 3. Figure 39. Bibliography 46.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ 6
ВСТУП 8
РОЗДІЛ 1 ПОСТАНОВКА ЗАВДАННЯ НА КОМП'ЮТЕРНУ ГРУ «SPACE SWITCH» З ВИКОРИСТАННЯМ DATA-ORIENTED DESIGN 11
1.1 Опис предметної області 11
1.2 Визначення потенційних конкурентних переваг 17
1.3 Постановка завдання на кваліфікаційну роботу 22
Висновки до розділу 1 23
РОЗДІЛ 2 ПРОЕКТУВАННЯ АРКАДНОЇ ГРИ НА ПАРАДИГМІ ECS 24
2.1 Проектування процесів 24
2.2 Моделювання даних 30
2.3 Проектування інтерфейсу 35
2.4 Опис архітектури 38
Висновки до розділу 2 41
РОЗДІЛ 3 РЕАЛІЗАЦІЯ ГРИ З ВИКОРИСТАННЯМ DATA-ORIENTED DESIGN 42
3.1 Опис інструментів 42
3.2 Опис архітектури 47
3.3 Реалізація геймплейної частини 52
3.4 Тестування програмного продукту 57
3.5 Використання програмного продукту 60
Висновки до розділу 3 63
ВИСНОВКИ 64
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 65

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ААА-ігри – високобюджетні відеоігри, які створюються великими студіями з використанням значних ресурсів і сучасних технологій.

AI (Artificial Intelligence) – штучний інтелект, що використовується у відеоіграх для створення поведінки NPC, процедурної генерації та інших механік.

Data-Oriented Design (DOD) — підхід до програмування, що фокусується на ефективній обробці даних процесором, оптимізуючи розташування та доступ до даних у пам’яті.

ECS (Entity Component System) – data-oriented архітектурна парадигма, що використовується у розробці відеоігор для оптимального управління ігровими об'єктами, їх компонентами та системами.

GameDev (Game Development) – процес розробки відеоігор, що включає програмування, графічний дизайн, звукорежисуру, сценарну розробку, тестування та маркетинг.

NPC (Non-Player Character) – персонаж у відеогрі, що керується штучним інтелектом, а не гравцем.

OOP (Object-Oriented Programming) – об'єктно-орієнтоване програмування, методологія розробки програмного забезпечення, що базується на використанні об'єктів і класів.

Pay-To-Win (P2W) - модель монетизації відеоігор, у якій гравці можуть отримати значну перевагу над іншими шляхом купівлі внутрішньоігрових предметів або покращень за реальні гроші.

Pipeline - це послідовність етапів обробки даних або виконання процесів у розробці відеоігор, що включає рендеринг, анімацію, обробку текстур та інші завдання.

Research - процес дослідження, аналізу та збору інформації для розробки та вдосконалення ігрових механік, штучного інтелекту та інших компонентів гри.

UI (User Interface) - це графічний інтерфейс користувача, який забезпечує взаємодію гравця з грою або програмою через візуальні елементи.

Аркадна гра — динамічна гра з простим керуванням і короткими ігровими сесіями, зазвичай орієнтована на досягнення високого рахунку. ААА-ігри – високобюджетні відеоігри, які створюються великими студіями з використанням значних ресурсів і сучасних технологій.

Геймплей – процес гри, тобто те, як саме користувач взаємодіє з механіками, світом і цілями гри.

Інпут – введення від гравця (натискання кнопок, рух мишки, дотики тощо).

Матчер – система або алгоритм, який підбирає об'єкти за певною логікою (наприклад, збіги у грі на відповідність).

Проджектайл – снаряд, який рухається у просторі після запуску (наприклад, куля чи ракета).

Скролер (жанр гри) — жанр відеоігор, де персонаж або середовище рухається у певному напрямку (горизонтально чи вертикально), створюючи ефект "прокручування" екрану.

Спавн – поява об'єкта у світі гри, зазвичай у заданій точці або за певних умов.

Фіча – окрема функція або можливість у грі чи програмі, що додає нову поведінку або контент.

ВСТУП

Актуальність теми. Розвиток комп'ютерних ігор є однією з найбільш динамічних сфер сучасних інформаційних технологій. Індустрія відеоігор постійно еволюціонує, використовуючи новітні підходи до розробки, що дозволяє створювати більш масштабні, продуктивні та гнучкі проекти. Різноманітність жанрів сьогодні вражає і захоплює користувачів, даючи змогу насолодитися різноманітними продуктами ігрового ринку. Особливої популярність завжди мали аркадні ігри, які є доволі прості у освоєнні, проте цікаві навіть для сотого проходження. Піджанр таких ігор – скролери за рахунок динаміки та різноманіття залишаються у топах навіть на поточний день. Унікальні механіки, особливо поведінки ворогів привертають увагу тисяч гравців, створюючи різноманітні геймплейні ситуації та можливості для вдосконалення майстерності.

З роками, не дивлячись на те, що core геймплею залишався незмінним, підходи до його розробки змінювались дуже швидко. Одним із ключових аспектів сучасного game dev є вибір підходу для сприйняття даних та взаємодій між ними. Легкість сприйняття даних людиною та обробки їх комп'ютером відіграють визначну роль у побудові ігрової архітектури. Для цього були винайдені цілі архітектури, які включають у себе правила та інструкції для представлення даних та взаємодій між ними.

Data-oriented design (DOD) є одним з найзручніших для описання динамічної структури, як гра, легкий для сприйняття людиною далекою від написання програм та надає програмі гнучкості у прийнятті майбутніх рішень.

Інструментарій для розробників має задовольняти низці умов, постійно удосконалюючи швидкість розробки, її якість та спрощуючи експлуатаційний період, дозволяючи легко впроваджувати нові механіки для задоволення потреб користувачів Однією з інноваційних архітектур у розробці ігор, яка описує собою DOD підхід є ECS. Він відокремлює дані від логіки обробки, що дозволяє підвищити продуктивність гри, особливо у проектах з великою кількістю об'єктів.

Гра «Space Switch» – є ідейним продовжувачем класичних скролерів, яка розширює їх механіки та можливості, особливо у сфері опису поведінки ворогів за допомогою використання data-oriented design у архітектурі за допомогою DOD та парадигми ECS. Впровадження такого підходу сприяє розширенню ігрових можливостей і створенню унікального користувацького досвіду.

Отже, дослідження та розробка комп'ютерної гри із використанням ECS є актуальними для галузі ігрової індустрії, оскільки цей підхід дозволяє створювати продуктивніші, масштабовані та гнучкіші ігрові системи. У межах кваліфікаційної роботи буде розглянуто процес розробки гри «Space Switch», що використовує DOD та його ключові механіки у розробці.

Мета дослідження. Метою проекту є розробка програмного забезпечення комп'ютерної гри «Space Switch», що використовує DOD для реалізації механік взаємодії між об'єктами.

Для досягнення мети виконано такі завдання:

- аналіз сучасних підходів до розробки відеоігор та DOD;
- визначення вимог до програмного забезпечення гри;
- проектування структури гри та її ігрових механік;
- реалізація ігрової логіки за допомогою DOD та ECS у Unity;
- тестування та оптимізація продуктивності гри.

Об'єкт дослідження: процес створення гри–скролера за парадигмою ECS, вплив парадигми на процес та результат розробки.

Предмет дослідження: комп'ютерна гра «Space Switch», побудована на базі data-oriented design.

Методологічна основа роботи. Дослідження базується на використанні аналітичних, порівняльних і експериментальних методів. Аналітичний метод застосовано для дослідження існуючих архітектур відеоігор та аналізу підходу до сприйняття та обробки даних. Порівняльний аналіз дозволив оцінити ефективність DOD у порівнянні з іншими підходами сприйняття даних.

Експериментальні методи використовувалися під час розробки гри, особливо у геймплейній частині.

Практичне значення. Результати роботи можуть бути використані для розробки високопродуктивних відеоігор із використанням DOD та впливу його дизайну на процес розробки. Використання цього дизайну програмного забезпечення дозволяє досягти високої продуктивності при обробці великої кількості ігрових об’єктів, що робить її придатною для розробки як інді–проектів, так і великих комерційних ігор.

Структура та обсяг пояснювальної записки. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та списку використаних джерел. Загальний обсяг пояснювальної записки становить 68 сторінок, з них основний зміст викладено на 66 сторінках. Робота містить 39 рисунків та 3 таблиці, що ілюструють принципи та ключові аспекти розробки гри.

РОЗДІЛ 1
ПОСТАНОВКА ЗАВДАННЯ НА КОМП'ЮТЕРНУ ГРУ «SPACE SWITCH» З ВИКОРИСТАННЯМ DATA-ORIENTED DESIGN

1.1 Опис предметної області

У сучасному світі ігри відіграють важливу роль у культурі та повсякденності більшості людей на Землі [1]. Індустрія розробки ігор виділяється як один із секторів, що розвиваються найшвидше, оскільки поєднує в собі багато аспектів роботи, таких як кодування, звуковий дизайн та сценарій, для створення цікавих і захопливих результатів.

За останнє десятиліття ігрова індустрія значно розширила своє охоплення аудиторії по всьому світу [2]. Мобільний геймінг став невід'ємною частиною буденності навіть людей похилого віку, що спонукає розробників та компанії створювати стратегії, які відповідають потребам різних груп користувачів. На тлі цього процесу ігрова індустрія постійно розвивається, намагаючись встигнути за напливом нових користувачів та утриманням старих юзерів.

Великий обсяг ринку спонукає розробників оптимізувати свої продукти, щоб залучити якомога ширшу базу користувачів. Аркадні ігри залишаються одним з найпоширеніших типів простих ігор на ринку (рис 1.1).

Аркадні ігри популярні завдяки своїй простоті, динаміці та миттєвому задоволенню від гри — не потрібно довго вчитуватись у правила, вчитися як у багатьох інших жанрах. А швидкий темп, азарт і притаманна епічність геймплею захоплює абсолютно всіх гравців, спонукаючи спробувати продукт. Вони ідеально підходять для коротких ігрових сесій, що добре поєднується з сучасним стрімким стилем життя багатьох людей.

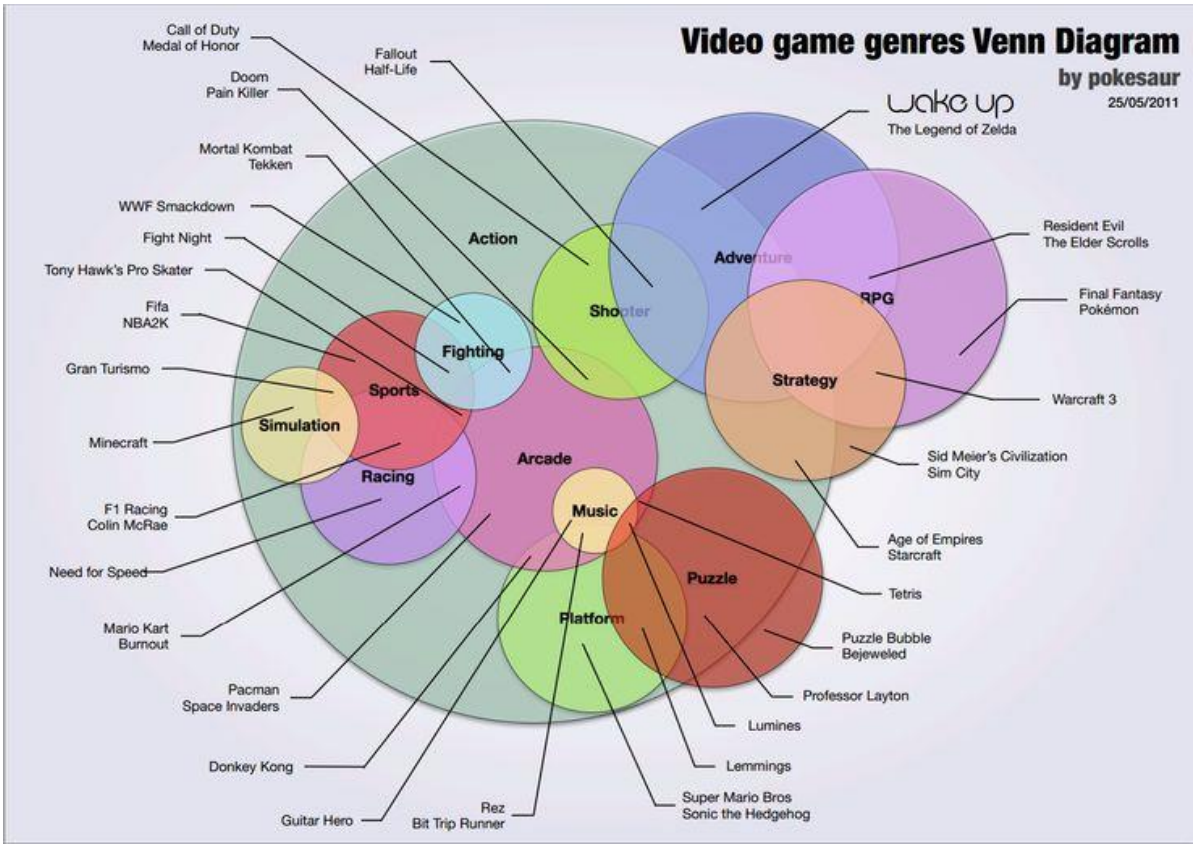


Рисунок 1.1 – Діаграма спектру жанрів ігрової індустрії

Джерело [3]

Проте, не дивлячись на простоту на перший погляд, аркадні ігри також потребують великих затрат у виробництві, проходячи великий цикл розробки гри:

1) прототипування – створення базової механіки гри для тестування ігрового процесу;
2) проектування гри – розробка концепції, написання документації, розробка геймдизайну;
3) програмування – реалізація механік гри на рушії;
4) створення графіки та звуку – розробка візуального стилю та музичного супроводу;
5) тестування та оптимізація – пошук та виправлення багів, оптимізація продуктивності;
6) реліз та підтримка – випуск гри, маркетинг та оновлення.

Під час розробки гри треба врахувати багато аспектів спільного pipeline, частини якого дозволяють процесу розробки гри мати комерційну ефективність (рис 1.2).

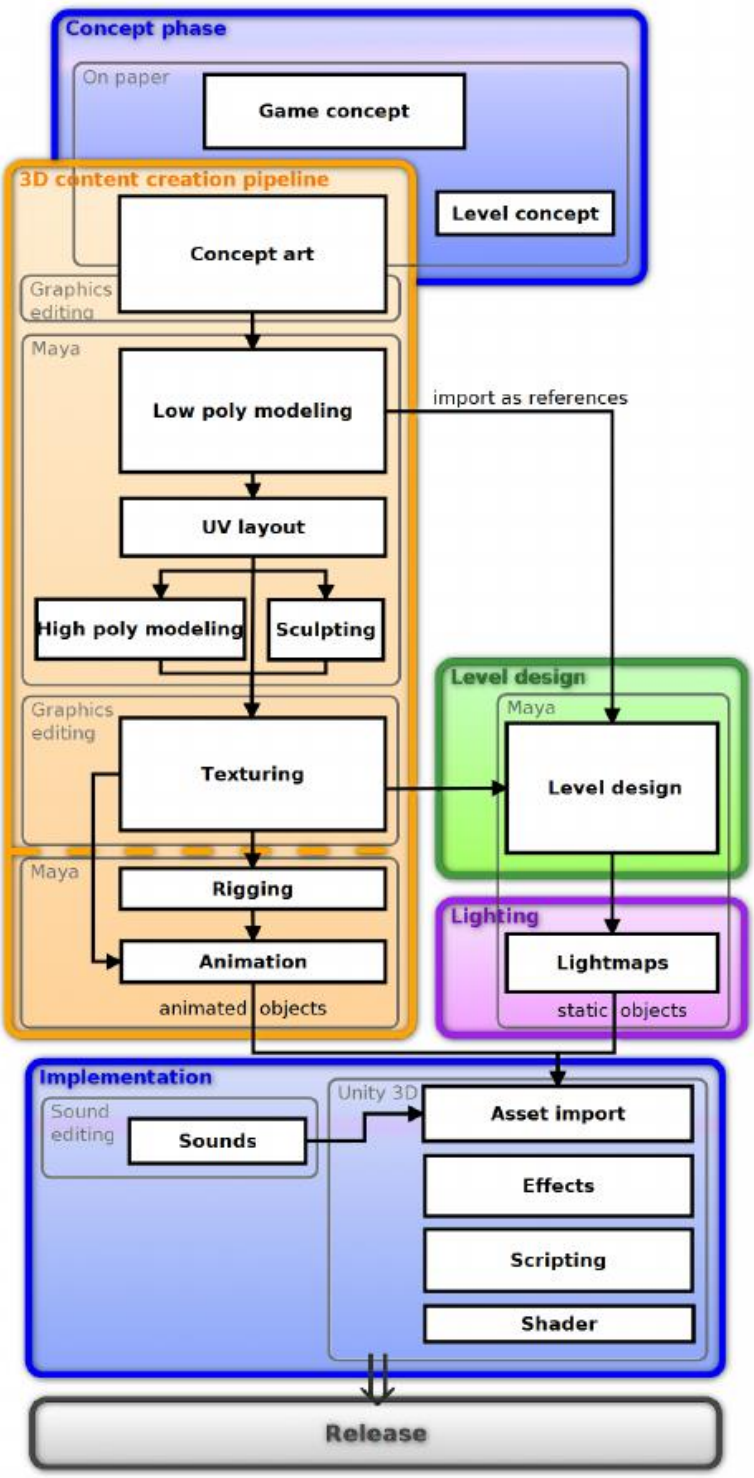


Рисунок 1.2 – Діаграма, яка ілюструє процеси розробки комерційної ігри

Джерело [4]

У умовах жорсткого pipeline розробки ігор ефективне використання ресурсів та сприйняття даних від одного етапу розробки до іншого стає критично важливим, особливо коли мова йде про масштабованість, простоту підтримки та швидкість розробки проекту. Ключову роль у менеджменту даних грає етап написання коду продукту, адже ті дані, які зручні для сприйняття людиною не завжди є такими для машин, що спонукає шукати підходи до реалізації програмного забезпечення з урахуванням особливостей передачі та обробки даних саме для технічних приладів.

Для цього впродовж десятиліть були створені різні принципи дизайну програмного забезпечення, які дозволяють перенести інформацію з людської мови вимог на програмну мову виконання. Найпопулярнішим рішенням досі є наприклад ООП [5]. Він є популярним напрямом сприйняття бізнес логіки та даних для переносу їх у світ програмного коду, адже пропонує інтуїтивний, на перший погляд принцип сприйняття даних через абстракції об’єктів та взаємодій між ними. Цей підхід користується неабиякою популярністю та дозволяє вирішувати проблематику даних у багатьох сферах IT на сьогодні.

Проте сфера game dev має низку своїх особливостей. Предметна область, яка у багатьох інших сферах є як правило сталою, у контексті гри може змінюватись кардинально за малий проміжок часу. Наприклад, навряд чи під час розробки банківської системи потрібно буде міняти принципи захищеності або оперування з грошима клієнтів, адже це ключові та сталі вимоги. Розробка гри , не дивлячись на те, що є бізнес процесом спирається більше на ідею, або представлення якогось процесу, який на думку розробників та замовника дозволить задовільнити поставленим вимогам. Під час же розробки ігрового продукту, принципи реалізації цієї ідеї можуть змінюватися після ігрових тестів, звичайних демонстрацій, коли стає зрозуміло, що вони не дають бажаного результату.

У цьому контексті традиційні об’єктно-орієнтовані підходи, попри свою зручність у моделюванні поведінки об’єктів, часто не забезпечують необхідної

гнучкості в масштабуванні, адже вони базуються на детермінованості предметної області з чіткими інтерфейсами взаємодії між даними.

У випадках розробки ігор усе більшої популярності набуває Data-Oriented Design [6] — архітектурний підхід, що фокусується не на структурі об’єктів та їх взаємодії, а на ефективному зберіганні та обробці даних, як окремих повноцінних сутностей. Кожен елемент гри у DOD відображається як сукупність простих, декларативних компонентів, що містять лише дані, без поведінки У традиційних об’єктно-орієнтованих підходах зі зростанням коду зростає і складність відстеження залежності між об’єктами, що може призводити до хаотичної архітектури, де зміни в одному місці спричиняють неочікувані наслідки в іншому. Натомість DOD заохочує поділ логіки на чіткі, незалежні системи та компоненти, що робить структуру проекту прозорою, передбачуваною та простою в навігації. Такий підхід значно знижує ризик надмірної залежності між елементами коду, що спрощує тестування, масштабування і повторне використання компонентів у межах інших ігрових систем.

У контексті аркадних ігор, де одночасно можуть існувати десятки і навіть сотні активних об’єктів (вороги, кулі, ефекти, input-сигнали тощо) які поєднують різноманітні механіки від взаємодії один з одним, Data-Oriented Design дає змогу уникнути типових проблем проектування, зменшивши зв’язки між різними модулями та системами гри, надаючи гнучкість та легкість до експериментування, шляхом надання фундаменту побудови стійкої до змін архітектури.

Правильний підхід до побудови архітектури дозволяє розробникам ефективно керувати ресурсами, запобігати проблемам продуктивності та спрощувати процес оновлення гри. Для реалізації підходу DOD у архітектурі застосунку одним з найпопулярніших рішень є використання Entity–Component–System (ECS) [7] (рис 1.3).

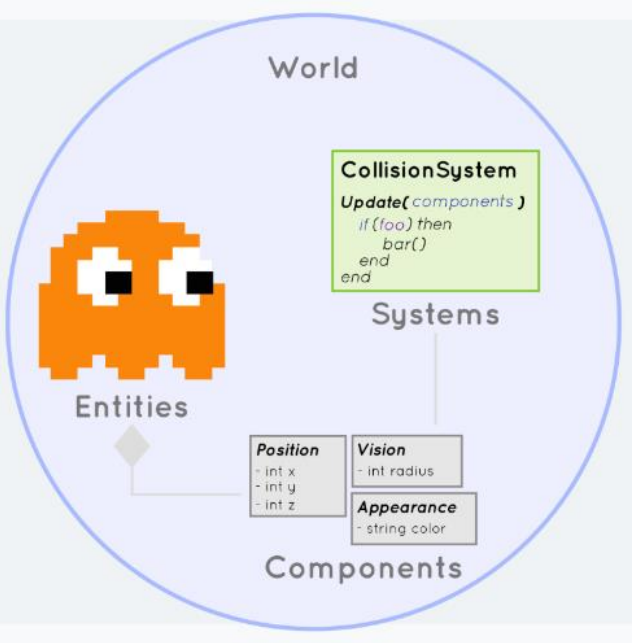


Рисунок 1.3 – Приклад сутності у світі ECS

Джерело [8]

ECS — це архітектурна парадигма, що забезпечує гнучке та продуктивне управління ігровими об'єктами. Її основна ідея полягає в розділенні даних і логіки: об'єкти (Entity) не містять поведінки, а лише служать контейнерами для компонентів (Component), які зберігають дані. Взаємодія між ними реалізується через системи (System), що виконують необхідні обчислення.

Парадигма ECS застосовується в розробці відеоігор, особливо у випадках, коли важлива продуктивність, оскільки він оптимізований для роботи з кеш–пам'яттю та дозволяє ефективно обробляти велику кількість об'єктів. На відміну від класичного підходу, де дані об'єкта можуть зберігатися в різних місцях пам'яті, ECS розташовує компоненти одного типу в масивах. Це дозволяє процесору швидко обробляти їх без додаткових звернень до пам'яті.

Оскільки системи працюють незалежно одна від одної, ECS дозволяє легко обробляти дані паралельно, що є критично важливим для продуктивних ігор. Завдяки чіткому розподілу ролей між Entity, Component і System,

зменшується кількість залежності між об'єктами, що спрощує їх тестування та налагодження.

Entity–Component–System є потужним архітектурною парадигмою, що дозволяє ефективно керувати великими ігровими проектами. Завдяки гнучкості, оптимізації кешу та можливості паралельного виконання ECS стає стандартом у розробці продуктивних ігор.

Попри складність реалізації, цей підхід дає значні переваги у масштабуванні та підтримці ігор, що робить його перспективним вибором для сучасних розробників. З огляду на тенденції розвитку індустрії, ECS продовжить впроваджуватися в ігрові рушії та стане ще більш поширеним у майбутніх проектах [9].

1.2 Визначення потенційних конкурентних переваг

Жанр скролл–шутерів бере свій початок ще з аркадних автоматів 80–х років, коли такі ігри, як "Galaga" [10] та "R–Type" [11], заклали основи динамічного геймплею та інтенсивних перестрілок. Технологічний прогрес трансформував жанр скролл-шутера, інтегрувавши покращену графіку та функції рольових ігор разом з більш складною ігровою механікою, отримавши більш деталізовану графіку, глибші механіки та елементи RPG [12]. Ігри в цьому жанрі приваблюють численних гравців завдяки активному впровадженню систем модернізації кораблів, а також різноманітним рівням складності та кооперативним режимам гри.

У сучасному ігровому середовищі жанр скролл–шутерів залишається популярним серед гравців завдяки динамічному геймплею, різноманітності ворогів та можливостям для вдосконалення ігрового процесу. Основними конкурентами проекту "Space Switch" є «Sky Force Reloaded» [13], «Razor2: Hidden Skies» [14] та «Hawk: Freedom Squadron» [15]. Ці ігри мають свої унікальні особливості та відмінності, які приваблюють гравців.

«Sky Force Reloaded»

Цей продукт відрізняється якісною графікою, плавною анімацією та добре збалансованою ігровою механікою. Основні переваги серії «Sky Force» включають:

- висока якість візуального оформлення;
- глибока система прокачування корабля RPG-like;
- відносно чесна монетизація – нема явних pay–to–win механік.
- режим кооперативної гри, який дозволяє проходити рівні разом з іншими.

Однак, серія «Sky Force» має певні недоліки, зокрема обмежену варіативність ворогів та відносно повільний темп додавання нового контенту (рис 1.4). Впродовж гри нові вороги з’являються нечасто – використовуються re-skin вже побачених раніше за механіками ворогів.

Рисунок 1.4 – «Sky Force Reloaded» скриншот геймплею. Кількість унікальних ворогів впродовж гри стрімко падає

Джерело [16]

«Razor2: Hidden Skies»

Ця гра орієнтована на більш хардкорну аудиторію і пропонує класичний підхід до жанру shoot ‘em up. Її ключові особливості:

- висока складність гри, що робить її привабливою для хардкорних гравців;
- технічна реалізація з акцентом на фізику польоту та поведінку ворогів;
- використання традиційного підходу до рівнів без надмірної казуалізації.

Головним недоліком гри є її обмежена привабливість для широкої аудиторії через високу складність, а також застарілий візуальний стиль (рис 1.5).

Рисунок 1.5 – «Razor2: Hidden Skies» скриншот геймплею. Добре видно елементи застарілого стилю гри. В центрі скупчення ускладнених для візуального сприйняття елементів

Джерело [14]

«Hawk: Freedom Squadron»

Ця гра має акцент на кооперативний режим та соціальну взаємодію між гравцями. Основні переваги:

- велика кількість рівнів та місій;
- мультиплеєрний режим;
- система щоденних завдань та подій.

Разом з тим, «Hawk: Freedom Squadron» активно використовує pay–to–win механіки, що може негативно впливати на баланс гри та спричиняти дискомфорт у безкоштовних гравців (рис 1.6).

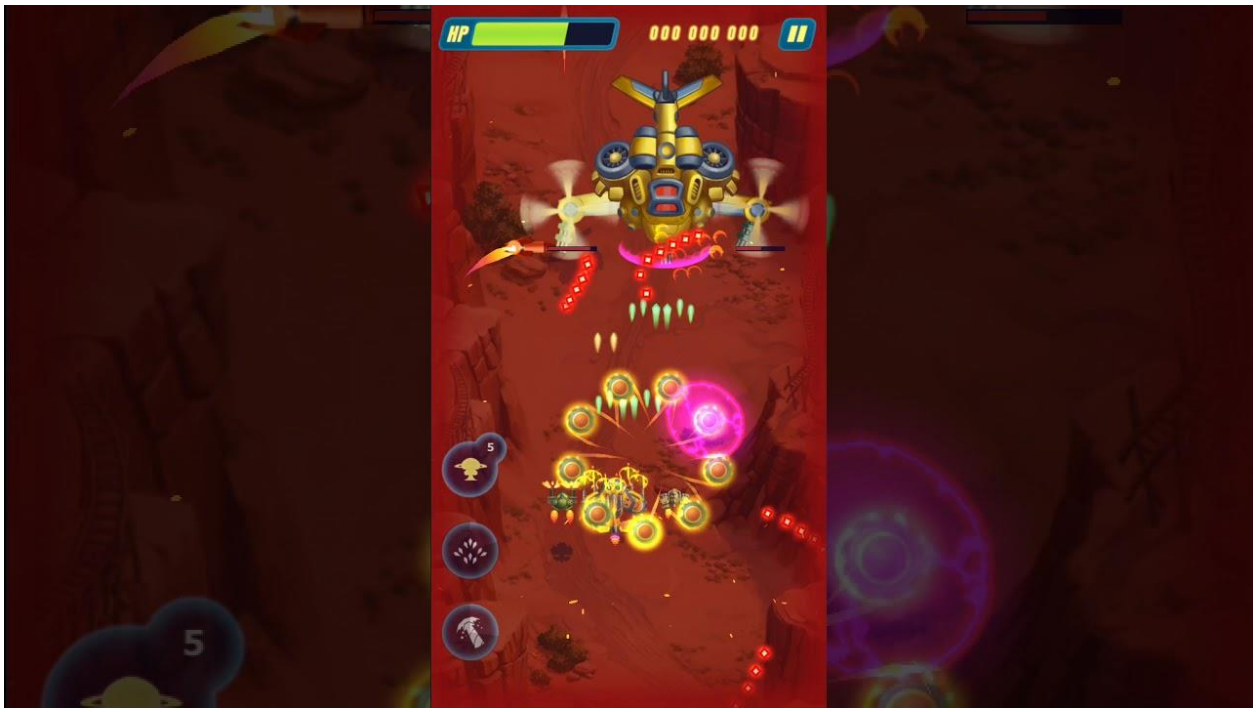


Рисунок 1.6 – «Hawk: Freedom Squadron» скриншот геймплею. Видно елементи UI, що відповідають за донатні здібності

Джерело [15]

Після огляду списку конкурентів було зазначено спільні та унікальні для кожного з них негативні риси, які будуть враховані під час розробки «Space Switch». З іншого боку, були враховані сильні сторони конкурентів, що в

сукупності з врахуванням їх недоліків надало «Space Switch» низку суттєвих переваг, які дозволять йому конкурувати з вищезгаданими іграми (табл 1.1):

Таблиця 1.1 – Переваги «Space Switch» над конкурентами

Перевага	*Реалізація*
Різноманіття ворогів та екшену	На відміну від більшості конкурентів, де зустрічається обмежений набір ворогів, “Space Switch” пропонує широке різноманіття супротивників із унікальними тактиками та механіками. Динамічні зміни бойових сценаріїв та комбінації ворогів додають непередбачуваності та збільшують реіграбельність
Швидкість впровадження нових геймплейних елементів	Гнучка архітектура гри дозволяє легко та швидко додавати нові механіки, зброю, рівні та модифікатори. Гравці матимуть можливість регулярно отримувати новий контент без тривалих очікувань, що сприятиме постійному інтересу до гри
Швидкість розробки	Завдяки використанню сучасних інструментів розробки, проект має високу швидкість впровадження нових оновлень та поліпшень, що забезпечує оперативну реакцію на відгуки гравців. Гнучкість у зміні механік дозволяє адаптуватися до побажань аудиторії швидше за конкурентів
Відсутність pay-to-win системи	На відміну від «Hawk: Freedom Squadron», де платний контент може створювати дисбаланс, «Space Switch» не матиме елементів, що дають несправедливу перевагу. Основний акцент буде зроблено на чесний прогрес, заснований на навичках гравця та його стратегії

З огляду на проведений аналіз, «Space Switch» має низку ключових переваг, які роблять його привабливим для гравців, особливо тих, хто цінує динамічний ігровий процес без залежності від покупок. Завдяки різноманіттю ворогів, швидкому впровадженню нового контенту та справедливій монетизації, він може стати гідним конкурентом на ринку скролл–шутерів.

1.3 Постановка завдання на кваліфікаційну роботу

Завданням кваліфікаційної роботи є розробка програмного забезпечення для гри з використанням Data-Oriented Design на парадигмі ECS. Це програмне забезпечення пропонує користувачам унікальний досвід гри подібного жанру, а розробникам – pipeline, який дозволить швидко і ефективно впроваджувати бізнес логіку. Для реалізації бізнес ідей, необхідно виконати наступні задачі:

- провести проектування структури гри;
- провести research ассетів графіки, звуків тощо;
- провести моделювання геймплейних аспектів;
- реалізувати програмне забезпечення;
- провести тестування.

Для вирішення цієї задачі будуть реалізовані системи, які дозволять впровадити зазначені аспекти:

- система керування гравцем;
- система спавну ворогів;
- система пересування ворогів;
- зручний та інтуїтивний UI;
- система поведінки снарядів.

Усі зазначені системи будуть реалізовані або суто за допомогою інструментів проектування ECS та data-oriented design, або за допомогою комбінації цих інструментів з ООР.

Висновки до розділу 1

Проведено дослідження предметної області, представленої грою у жанрі аркадного скролера та процесом її розробки. Охарактеризовано підхід Data-Oriented Design його переваги для розробки ігор, та предмету роботи зокрема.

Аналіз існуючих конкурентів та їхніх особливостей виявив низку недоліків основних ігор–скролерів, які може компенсувати «Space Switch». Також було досліджено pipeline конкурентів та виявлено їхні слабкі сторони, зокрема щодо швидкості прототипування. Для розробки проекту обрано альтернативний підхід – використання парадигми ECS.

На основі аналізу визначено ключові завдання проекту: реалізацію незадоволених потреб користувачів, які не врахували конкуренти, а також оптимізацію швидкості розробки. Окреслено основні етапи створення «Space Switch», складено задачі та визначено ключові аспекти розробки в контексті обраної парадигми.

РОЗДІЛ 2
ПРОЕКТУВАННЯ АРКАДНОЇ ГРИ НА ПАРАДИГМІ ECS

2.1 Проектування процесів

Проектування продукту має надати результат, який буде відповідати застосунку, що зможе задовільнити усі умови – функціональні та нефункціональні. Функціональні вимоги – це вимоги до продукту, основні та конкретні функції, що описують внутрішню структуру і роботу програми, які система повинна виконувати (обчислення даних, маніпулювання даними тощо). Для програмного забезпечення «Space Switch» до таких вимог належать:

- бізнес логіка описана за допомогою мови програмування правила поведінки об’єктів у грі, їх взаємодія та обробка первинних даних та даних, отриманих під час вза’ємодії цих об’єктів між собою;
- системна логіка описана високорівневими шлюзами взаємодії бізнес логіки із зовнішниіми даними, їх корекція\доповнення а також актуалізація за потреби;
- Beatuful Core - це основна частина візуальної презентації гри (particles, моделі, спрайти, шейдера тощо) які і задають художню стилістику грі;
- UI інтерфейс взаємодії користувача.

Нефункціональні вимоги – параметри, що не є безпосереднім функціоналом програми, але які впливають на якість, сприйняття користувачем та характеристики її роботи (безпека, швидкість тощо). Для програмного забезпечення «Space Switch» до таких вимог належать:

- Open Source – це код програмного забеспечення має бути розміщений на платформі для вільного ознайомлення користувачами та контрибуції за бажанням [17];

- дані статичного характеру (які не змінюються під час сеансу гри – конфіги балансу, ворогів, гравця тощо) мають мати можливість знаходитись у хмарному сховищі для майбутнього можливого мастабування;
- гра має працювати стабільно, проходячи увесь цикл користування не минаючи основних етапів. Кількість багів має прагнути до 0;
- модулі програмного забеспечення повинні мати обширні можливості для відного легкого,дешевого та стрімкого масштабування (кількість ворогів, рівнів тощо);
- гра має стабільно працювати на більшості конфігурації ПК на 2025 рік. Мають бути відсутні стрімки просадки у ФПС, неоптимізований батчинг, несжаті текстури.

На діаграмі use-cases (рис 2.1) зображено основні дії, які може виконувати користувач у межах гри Space Switch, а також взаємозв’язки між цими діями. Гравець, що ініціює взаємодію з системою, доступні варіанти: запуск гри, ініціалізація, зміна параметрів, керування ігровим процесом, зупинка гри, її перезапуск, а також перегляд рекорду очок. Центральним елементом взаємодії є сценарій «Грати гру», який включає можливість зупинки гри (через паузу) та пов’язаний із завершенням гри, підкреслюючи, що ці дії можуть бути частиною загального процесу запуску або підготовки гри.

Центральними елементом гри є сутність гравця, яка представлена космічним кораблям, керування яким і визначає геймплей та виконання ігрових інструкцій. Гравець, як ключова фігура та актор будує навколо себе ядро взаємодії частин програми (система очок, рух камери, game loop [18] тощо).

Опис потоку процесів, які відбуватимуться під час гри здійснений за допомогою діаграми діяльності (рис. 2.2). Вона демонструє моделювання процесів та взаємозв’язків у програмі, які можуть відбутися під час роботи.

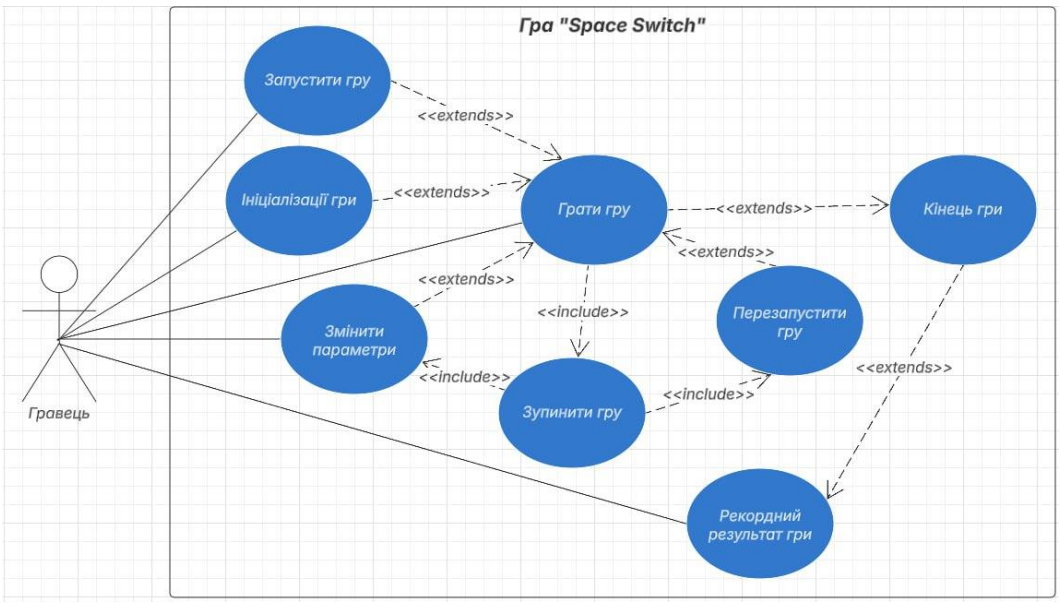


Рисунок 2.1 – Діаграма use-case для гри «Space Switch»

Джерело розроблено автором

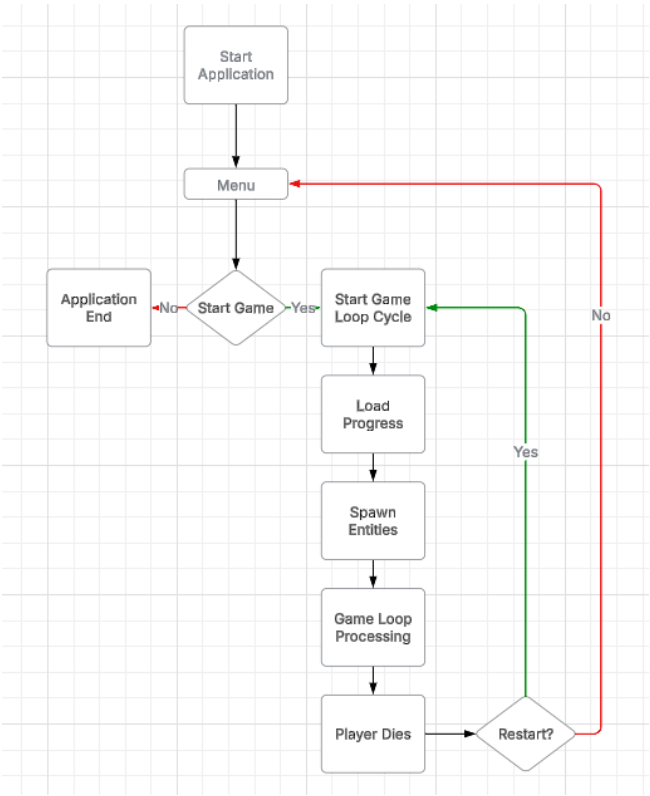


Рисунок 2.2 – Діаграма діяльності застосунку

Джерело розроблено автором

На початковому етапі користувач потрапляє в головне меню, де він може обрати, чи розпочати гру, чи завершити роботу програми. Якщо користувач вирішує не грати, застосунок завершує роботу. Якщо ж він обирає початок гри, запускається ігровий процес – створюються ігрові сутності, такі як персонаж, вороги, предмети та інші елементи рівня. Після цього починається основний ігровий цикл, у якому здійснюється обробка ігрової логіки, взаємодія персонажа з оточенням, рух ворогів тощо. Якщо гравець гине під час проходження рівня, гра пропонує йому варіант перезапустити гру, у разі відмови – повертає у головне меню, де цикл повторюється.

Ілюстрація послідовних процесів у застосунку було здійснено з використанням діаграми послідовності (рис. 2.3). Діаграма послідовності демонструє основний цикл гри гравця (game loop) та його залежність від програмних сервісів та дій ворога. Вона охоплює ключові етапи життєвого циклу ігрової сесії: від ініціалізації гри до завершення через перемогу або поразку, демонструючи взаємодію між основними об’єктами гри — гравцем, програмними сервісами та ворогами — у вигляді подій, що послідовно викликають одна одну.

Алгоритм починається з відкриття гравцем застосунку, після чого завантажується головне меню. Гравець запускає гру, внаслідок чого програмні сервіси створюють корабель гравця та інтерфейс користувача (UI). Далі система генерує кораблі двох ворогів. Після цього настає активна фаза гри: гравець починає керування своїм кораблем та виконує постріл. Якщо постріл потрапляє у ворожий корабель, програмна логіка фіксує отримання шкоди, а в разі знищення ворога — подію смерті та нарахування очок гравцю. Після цього другий ворог, виконуючи постріл що шкодить гравцеві та за умови критичної кількості здоров’я гравця, останній може загинути, що спонукає сервіси викликати вікно програшу, де гравець може завершити гру, вийшовши з застосунку.

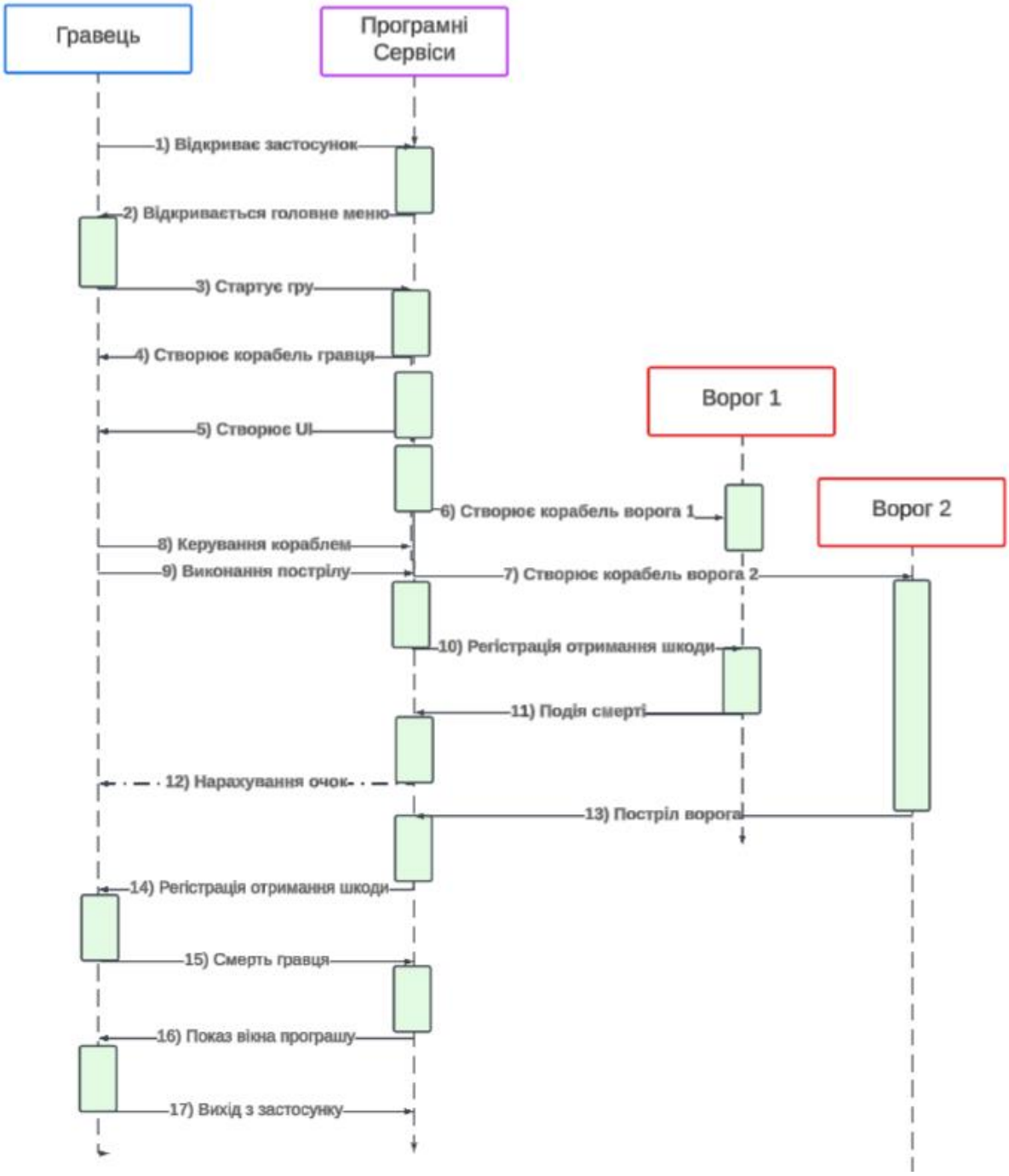


Рисунок 2.3 – Діаграма послідовності з описаними життєвими циклами гравця і ворога зверху униз

Джерело розроблено автором

Комунікація компонентів у ECS відбувається через велику кількість етапів, які проходять дані для обробки системами. На прикладі системи

пострілів можна простежити основні комунікації між компонентами всередині модулів архітектури (рис 2.4).

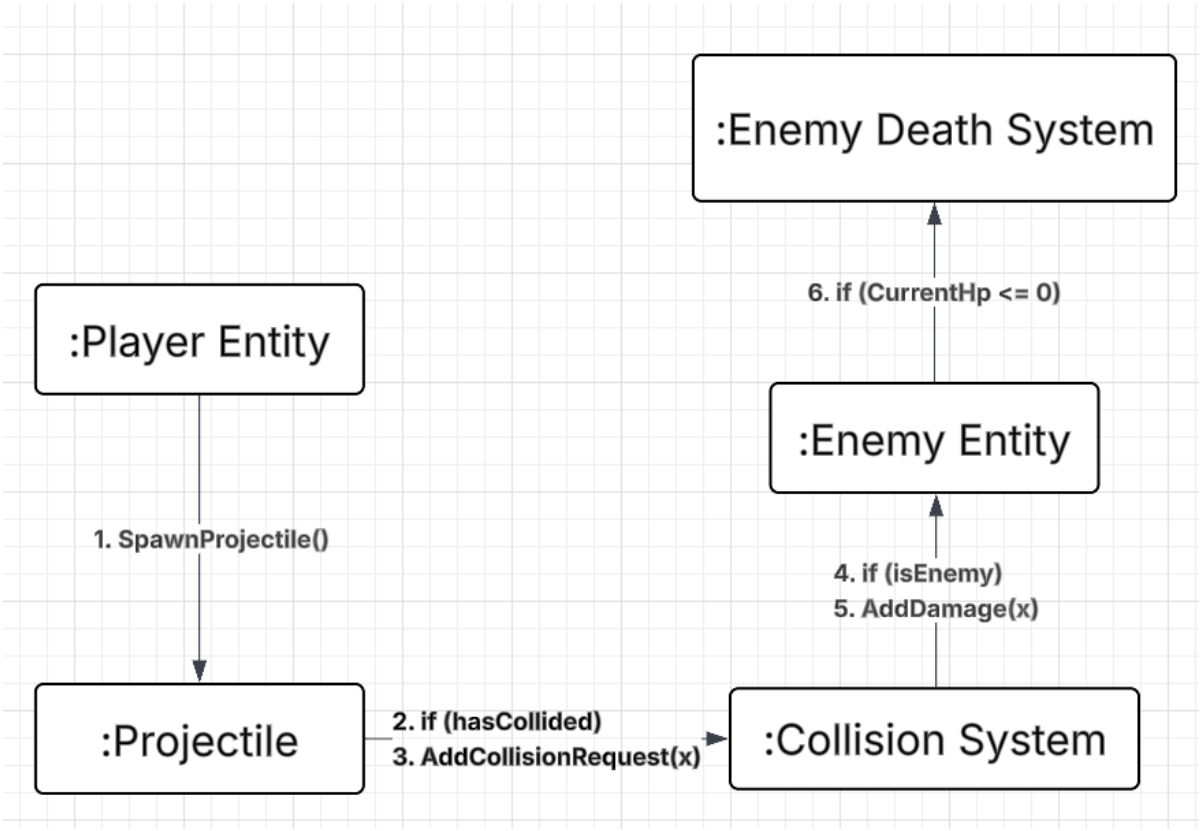


Рисунок 2.4 – Діаграма комунікацій процесу пострілу гравця із ураженням ворога

Джерело розроблено автором

Гравець створює снаряд через метод SpawnProjectile, після чого projectile рухається у просторі, допоки не зіштовхнеться з колайдером іншого об’єкту, передавши дані про колізію до системи обробки. Вона, обробивши сутність колізії, у разі якщо це ворог, запросить через метод API ворога отримання пошкодження від снаряду. Ворог, обробивши пошкодження, у разі закінчення очок здоров’я потрапить до системи обробки смерті, яка виконає усі необхідні маніпуляції з мертвим ворогом, наприклад нарахує очок за його смерть гравцеві та вимкне ще працюючі системи ворога.

2.2 Моделювання даних

Не дивлячись на те, що ООП є найбільш розповсюдженим підходом для моделювання даних у розробці скролерів йому притаманні ряд недоліків у контексті розробки ігрового продукту. В ООП основною структурною одиницею є клас, який об’єднує дані (поля) та методи (функції) в єдину логічну сутність. Об’єкти взаємодіють між собою через виклики методів, змінюючи свій внутрішній стан , описаний у полях (рис 2.5).

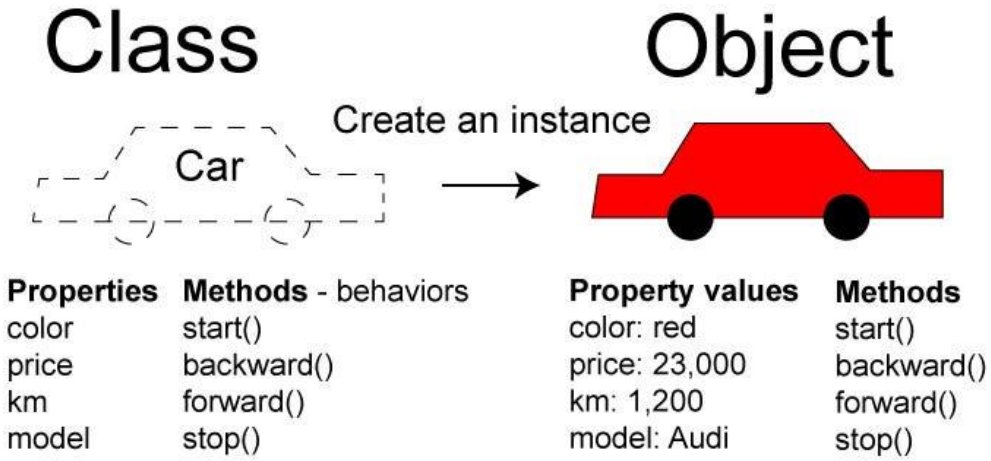


Рисунок 2.5 – Представлення об’єкту з точки зору ООП

Джерело [19]

Це робить цей підхід зручним для моделювання складних ієрархічних структур, але призводить до тісного зв’язку між класами та проблеми масштабуванням та розвитком програмного забезпечення. Будь який ігровий продукт, як правило не є сталою предметною областю – вимоги можуть змінюватися дуже швидко, особливо це стосується скролерів, де різноманіття снарядів та ворогів грає визначну роль у цікавості геймплею. Гра «Space Switch» робить акцент на різноманітті та швидкості впровадження нових механік. Це робить ООП малопридатним для реалізації цього продукту.

Як альтернатива, запропоновано використати підхід, що базується на ECS. Цей підхід, навпаки, пропонує більш гнучку й модульну архітектуру, розділяючи дані (компоненти) та логіку (системи) (рис. 2.6).

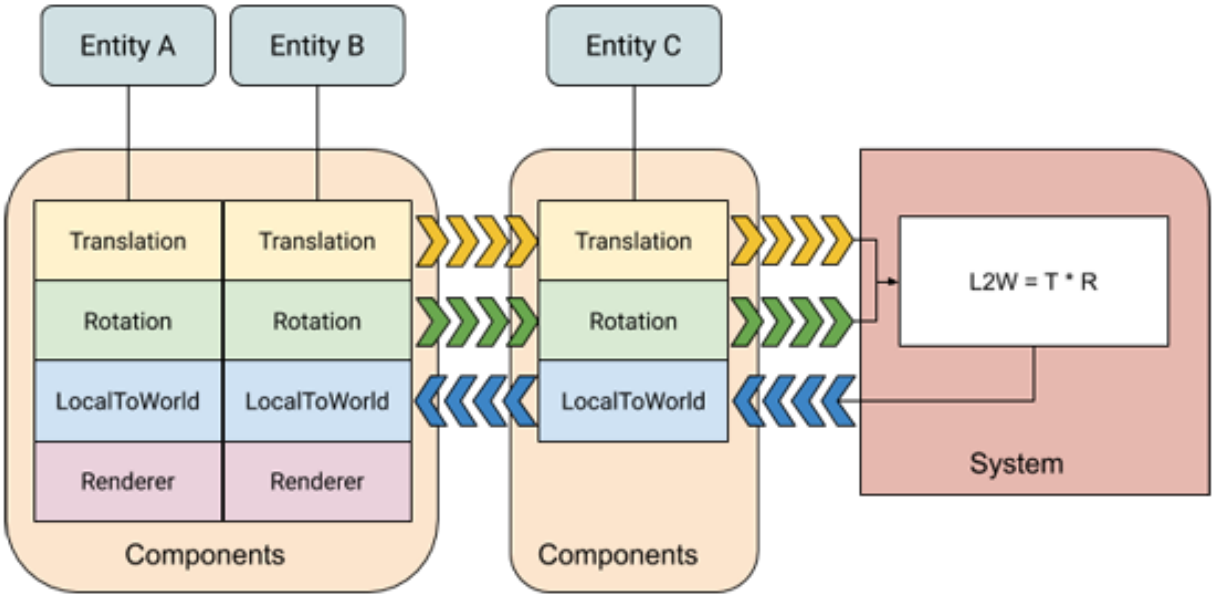


Рисунок 2.6 – Представлення об’єкту з точки зору ECS

Джерело [20]

У цій моделі кожен об’єкт гри представлений сутністю (Entity), яка містить лише унікальний ідентифікатор, а його поведінка визначається набором незалежних компонентів. Системи керують сутностями, фільтруючи наявні на них компоненти, застосовуючи до них відповідну логіку. Це забезпечує кращу продуктивність у великих ігрових проектах, оскільки дозволяє ефективніше представляти логіку застосунку та розширювати наявні системи, не втручаючись у роботу вже наявних систем, ізолюючи їх з можливістю повторного використання.

Особливість ECS дає більш гнучку та легку для розробки та підтримки архітектуру, яка зручно компонує аспекти гри у невеликі відокремлені модулі – фічі, які дозволяють ізолювати концепції одна від одної, уникаючи прямих

зв’язків між класами реалізації цих фіч, даючи змогу думати більше саме про логіку, а не про пов’язання різних модулів між собою.

Враховуючи це, програмний код було розбито на окремі фічі, які не мають прямі зв’язки і можуть бути представлені як окремі модулі із своєю внутрішньою структурою. Як приклад представлена діаграма класів (рис. 2.7) модуля колекціонування цілей, яка використовується різними модулями з метою отримання цілей для атаки.

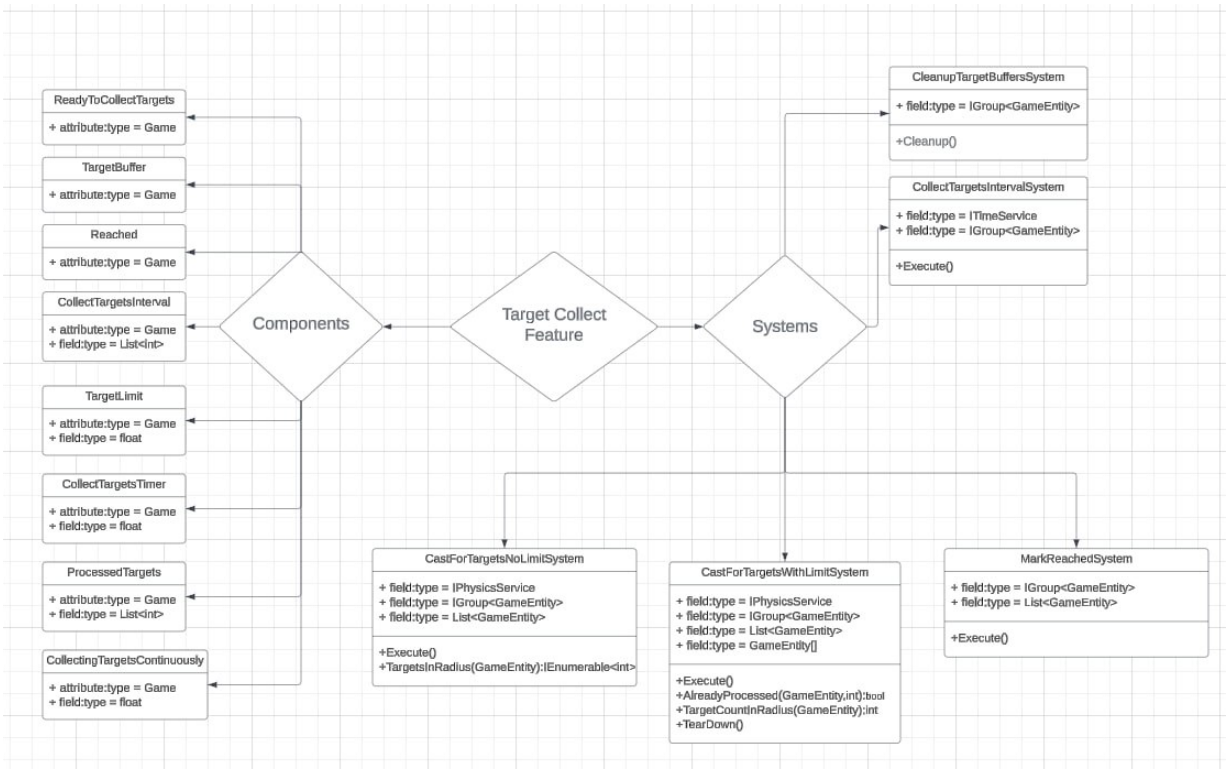


Рисунок 2.7 – Діаграма класів модуля колекціонування цілей для атаки

Джерело розроблено автором

Оскільки вона ізольована та базується на своїх даних, то результати її роботи може використовувати будь який юзер – гравець, ворог, окремий снаряд тощо, без необхідності проектування прямих, або абстрактних зв’язків.

Основа підходу ECS – оперування над компонентами, які належать певним entity (аналог об’єкту у ООП), наприклад основні сутності у грі – гравець, ворог та снаряди (рис 2.8).

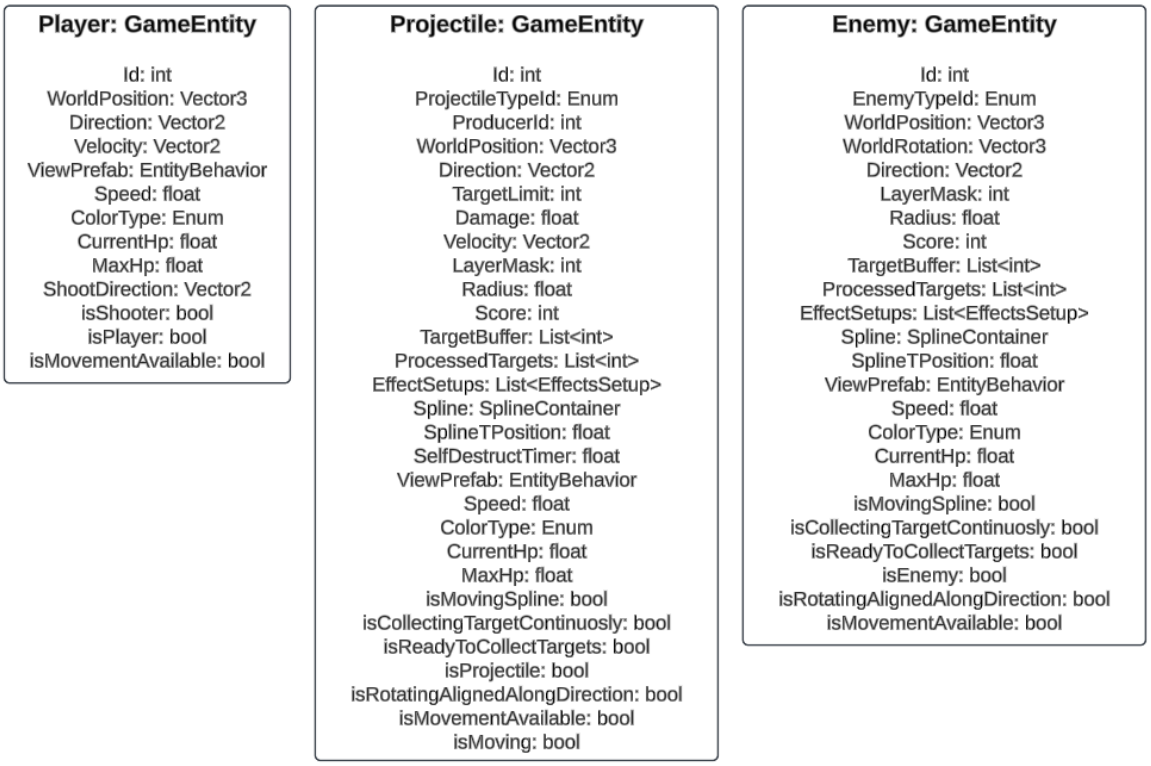


Рисунок 2.8 – Ілюстрація представлення об'єктів гравця, снаряду та ворога з наборами компонентів та їх типами

Джерело розроблено автором

Сутності містять велику кількість компонентів, які і дають можливість охарактеризувати поточну сутність як ворог чи гравець. Не дивлячись на їх велику кількість, компоненти є примітивами, як правило, із єдиним значенням елементарного типу мови C# [21] (int, float, enum тощо).

Структурно проект поділяється на окремі модулі, які скомпоновані по папках у файловій системі проекту (рис 2.9).

Керування game loop відбувається через низку високорівневих компоненти-акторів, які описані на діаграмі компонентів (рис 2.10). Вони визначають ключові аспекти, як менеджмент ресурсів та перемикання етапів цього циклу.

Взаємодія юзера відбувається через root стейт машину, котра перемикаючи стейти звертається до компонентів Static Data Service та Progress

Provider, які відіграють ключову роль у керуванні ресурсами гри. Static Data Service займається менеджментом статичних, незмінних даних (конфіги з балансом, із статами тощо), які знаходяться на ремоут сервері Firebase [22], а Progress Provider динамічними та локальними на пристрої даними – наприклад, кількість максимальних очок, які набив гравець на поточному девайсі, збережених у форматі JSON [23].

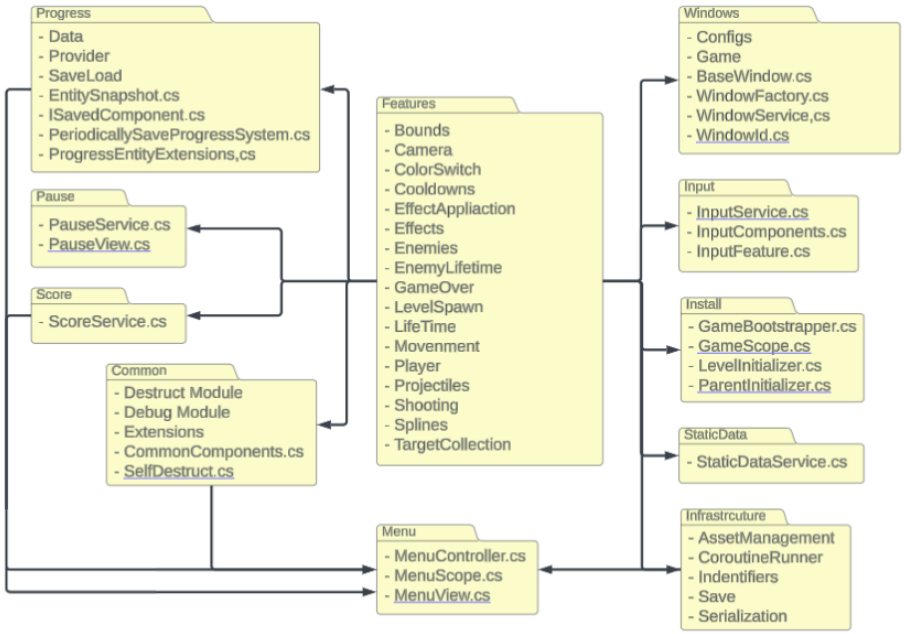


Рисунок 2.9 – Діаграма пакетів. Кожен пакет представлений окремим модулем, який компонує у собі під-модулі, або окремі .cs файли

Джерело розроблено автором

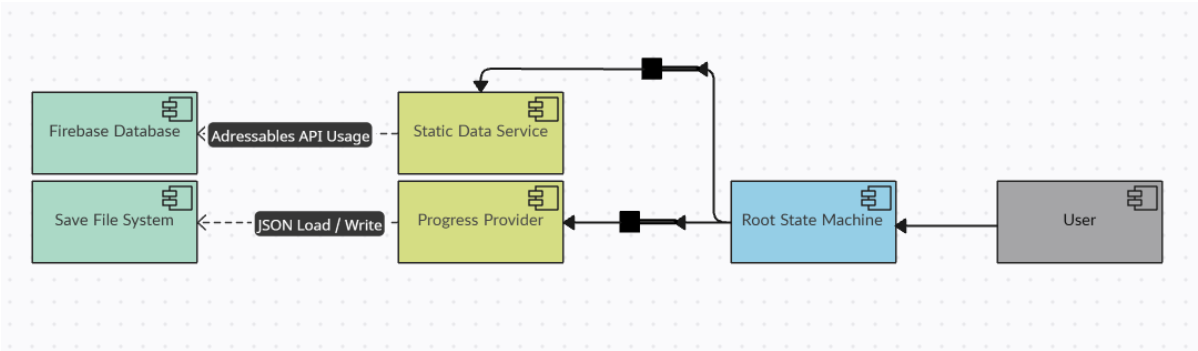


Рисунок 2.10 – Діаграма компонентів застосунку

Джерело розроблено автором

У фізичному представлені застосунок представляє собою зв’язок remote сервера Firebase із персональним комп’ютером, на якому здійснюється запуск застосунку, взаємодія між якими відбувається через власний API всередині модуля локальних даних та наданим середовищем API для керуванням remote запитами (рис 2.11).

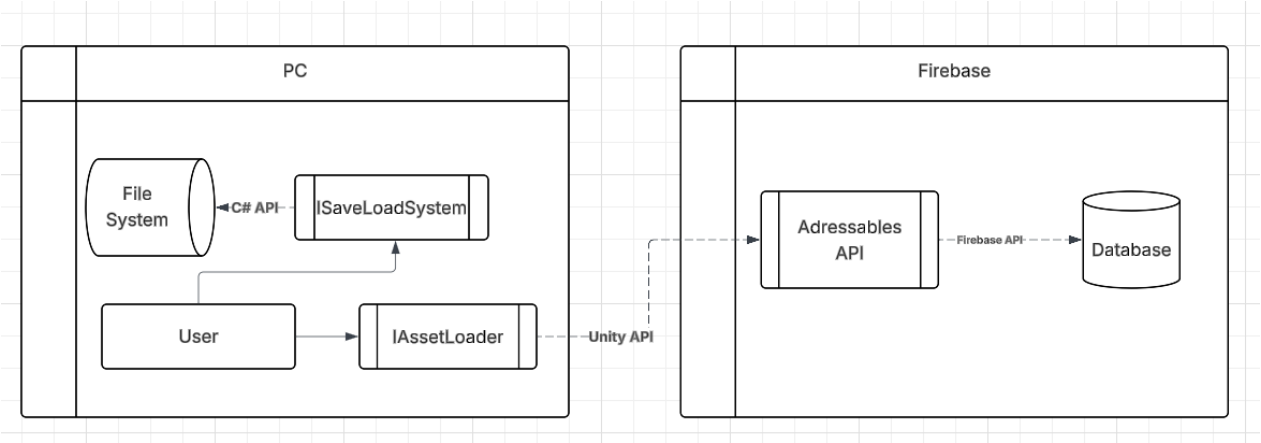


Рисунок 2.11 – Діаграма розгортання фізичного середовища

Джерело розроблено автором

2.3 Проектування інтерфейсу

У динамічних аркадних іграх типу скрол–шутерів інтерфейс користувача виконує не лише інформаційну, але й функціональну роль. Через високу швидкість подій, велику кількість рухомих об’єктів і візуальну насиченість сцени, надмірна кількість елементів інтерфейсу або їх неузгоджене оформлення здатні значно ускладнити сприйняття та знизити якість ігрового досвіду.

Досвід інших проектів у цьому жанрі дозволяє простежити низку поширених проблем. У грі «Razor2: Hidden Skies» використовується класичний інтерфейс із великою кількістю статичних елементів, що постійно присутні на екрані (рис. 2.12). Таке рішення не враховує контекст ігрових подій і створює ситуацію, коли важлива інформація може губитися серед

другорядних візуальних об’єктів. Крім того, графічний стиль UI у Razor2 містить застарілі декоративні рішення — рамки, текстури з низькою прозорістю, дрібні шрифти — що ускладнює сприйняття.

Рисунок 2.12 – Гра «Razor2: Hidden Skies» в якій використовується застарілий інтерфейс із великою кількістю статичних елементів

Джерело [14]

У випадку з «Hawk: Freedom Squadron» інша проблема, хоча інтерфейс візуально яскравіший, він перевантажений елементами, що стосуються ігрових покупок, щоденних завдань і подій (рис. 2.13). У результаті увага гравця розсіюється між геймплейними та сервісними блоками, що є критичним недоліком у грі, де темп змін подій дуже високий.

На фоні попередніх конкурентів лише «Sky Force Reloaded» демонструє краще рішення – помірну мінімізацію елементів інтерфейсу (рис. 2.14). Більшість інформації з’являється лише в моменти, коли вона справді необхідна. У таких проектах візуальна мова інтерфейсу побудована навколо простих іконок, мінімальної кількості кольорів та ефектів прозорості, що дозволяє інтегрувати UI в загальну стилістику гри, не конфліктуючи з нею.

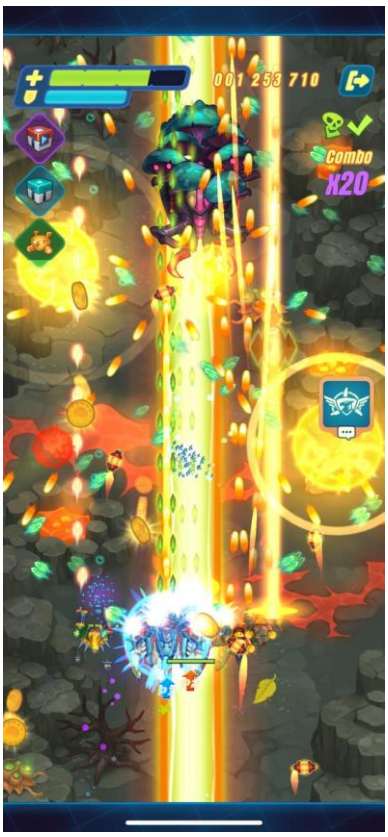


Рисунок 2.13 – Інтерфейс гри «Hawk: Freedom Squadron» з перевантаженими елементами

Джерело [15]

Рисунок 2.14 – Гра «Sky Force Reloaded» з помірною мінімізацію елементів інтерфейсу

Джерело [16]

З урахуванням вищезгаданого можна стверджувати, що для гри «Space Switch», орієнтованої на активний бойовий процес з високою щільністю подій, найбільш доцільним є підхід до інтерфейсу, в основі якого лежить мінімалізм, сприйняття і візуальна чистота.

Щодо стилістики, за рахунок аналізу стало зрозуміло про ефективність стриманого футуристичного оформлення інтерфейсу – з використанням прозорих панелей, неонових підсвіток, простих форм та обмеженої палітри. Такий підхід не лише узгоджується з космічною тематикою, але й забезпечує візуальний контраст із насиченим фоном ігрових сцен.

Інтерфейс «Space Switch» формується з урахуванням цих принципів: візуальна простота, контекстне відображення інформації, узгодженість зі стилістикою, ефективне використання простору та обмеженої кількості ефектів. Такий підхід дозволятиме не лише зберігати чистоту екрану, а й підтримувати загальну динаміку геймплею.

2.4 Опис архітектури

Опис архітектури виконано з акцентом на окремі модулі, враховуючи специфіку побудови зв’язків між модулями у проекті, написаному на парадигмі ECS.

Структурно проект поділено на фічі – окремі самодостатні модулі, які виконують свою роботу ізольовано, а за необхідності даних від інших систем передають дані через компоненти. На прикладі процесу керування гравцем свого корабля, на діаграмі Крухтена 4+1 [24] (рис 2.15) проілюстровано етапи та складові процесу від введення даних на переміщення, до зміни позиції гравця у грі.

Основний процес зміни позиції описується алгоритмом: «дані про інпут збираються системою → система фільтрує та надає зручний формат цих даних для пайплайну гри → для гравця задається новий напрям руху, базуючись на цих даних → гравець змінює свою позицію».

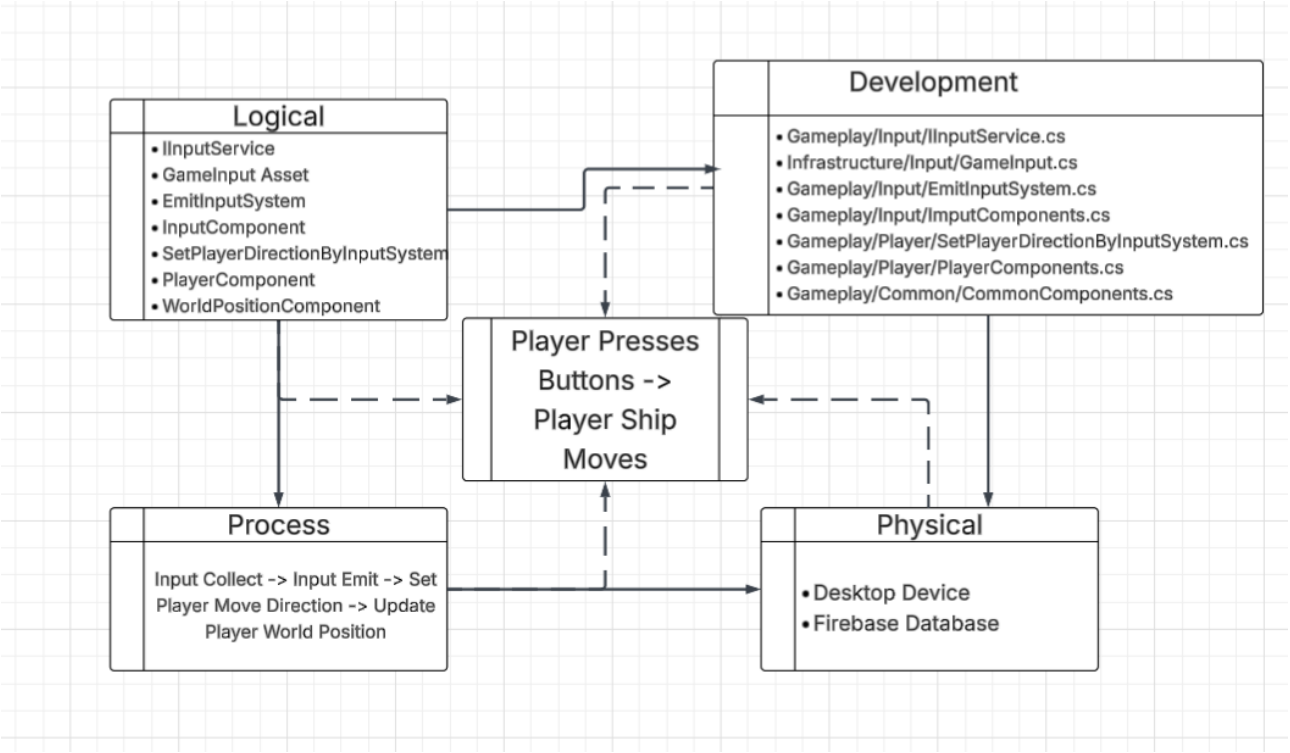


Рисунок 2.15 – Діаграма Крухтена 4+1. Опис процесу керування гравцем корабля

Джерело розроблено автором

Логічне керування процесом відбувається за роботи самостійно розробленого сервісу IInputService та згенерованому за допомогою середовища GameInput клас, який і реєструє натискання клавіш, передаючи їх до IInputService. Отримані дані компонуються у InputComponent, де перехоплюються системою SetPlayerDirectionByInputSystem, яка оновлює позицію гравця по компонентам PlayerComponent та WorldPositionComponent. Для розробників на діаграмі вказані шляхи до вищезгаданих компонентів у файловій системі проекту, а також фізичні елементи застосунку, на яких базується менеджмент даних під час роботи застосунку.

Опис повної структури від загальних аспектів до конкретних було здійснено за допомогою методу C4 [25] (рис 2.16).

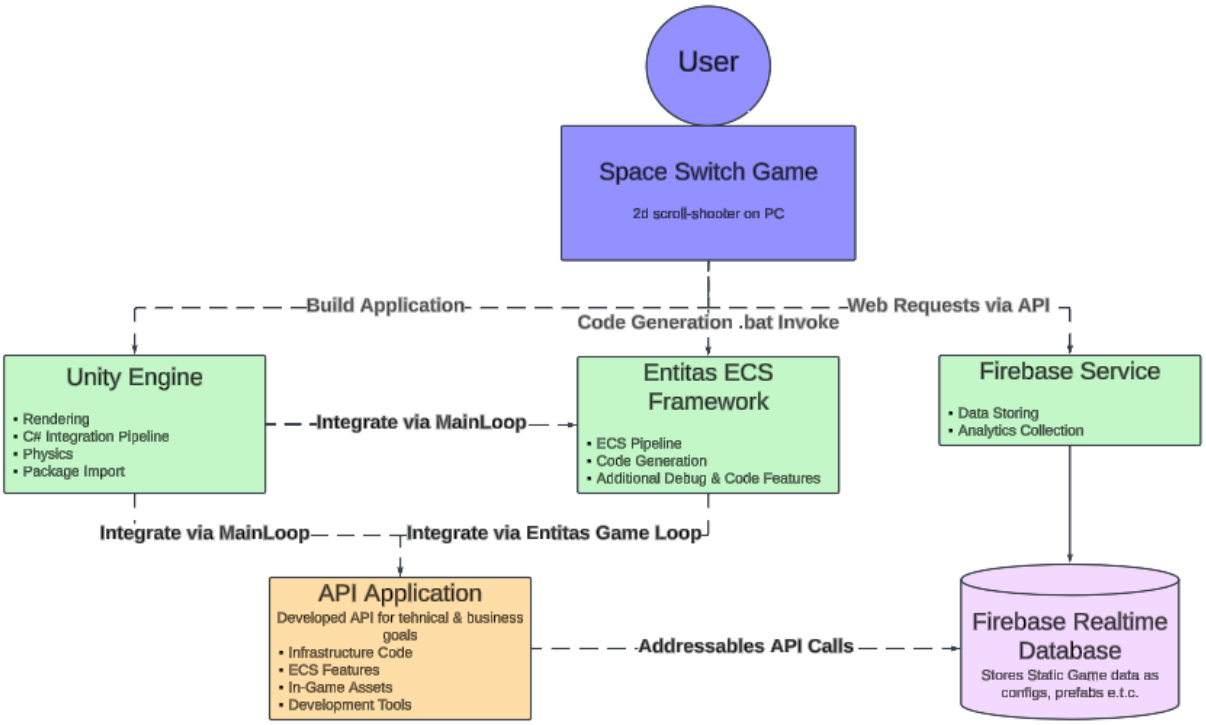


Рисунок 2.16 – Діаграма C4. Загальний опис структури компонентів у архітектурі та зв'язків між ними

Джерело розроблено автором

Основні складові проекту – Unity Engine, фреймворк Entitas, який реалізує ESC, сервіс Firebase із базою даних на сервері, та безпосередньо Application API, написаний у процесі розробки.

Середовище UnityEngine надає доступ до реалізації систем рендерінгу, фізики, імпорту пакетів а також будує шлюзи для інтеграції кастомного коду, який може бути написаний розробниками. Фреймворк Entitas реалізовує парадигму ECS та дає змогу інтегрувати у свій головний цикл обробки сутностей написані розробниками системи, компоненти та фічі.

Сервіс Firebase виконує функції бази даних, надаючи доступ до статичної інформації на сервері, а також може бути використаний для збору аналітики застосунку.

Висновки до розділу 2

На етапі проектування було позначено основні вимоги та ключові діючи актори гри. У ході моделювання даних був обраний оптимальний архітектурний підхід для опису акторів та взаємодії динамічних даних. Був обраний мінімалістичний підхід до опису інтерфейсу гри та використані перевірені рішення для структуризації елементів UI.

Запропоновані рішення дозволять забезпечити ефективну роботу застосунку, його масштабованість та зручність використання. Проектування інтерфейсу враховувало потреби користувачів, що сприятиме покращенню взаємодії з програмою. Таким чином, на базі здійсненого проектування можна побудувати програмний продукт, який задовільнить усім поставленим завданням розділу I.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ГРИ З ВИКОРИСТАННЯМ DATA-ORIENTED DESIGN

3.1 Опис інструментів

Основним середовищем для розробки гри «Space Switch» є ігровий рушій Unity [26] – сучасна платформа, орієнтована на створення інтерактивного 2D та 3D контенту. Unity забезпечує гнучке середовище для інтеграції графіки, фізики, анімації, звуку та script-логіки гри. Завдяки своїй модульності та активній екосистемі, Unity дозволяє розробникам легко підключати сторонні пакети та використовувати готові рішення типових проблем для економії сил та часу (рис 3.1).

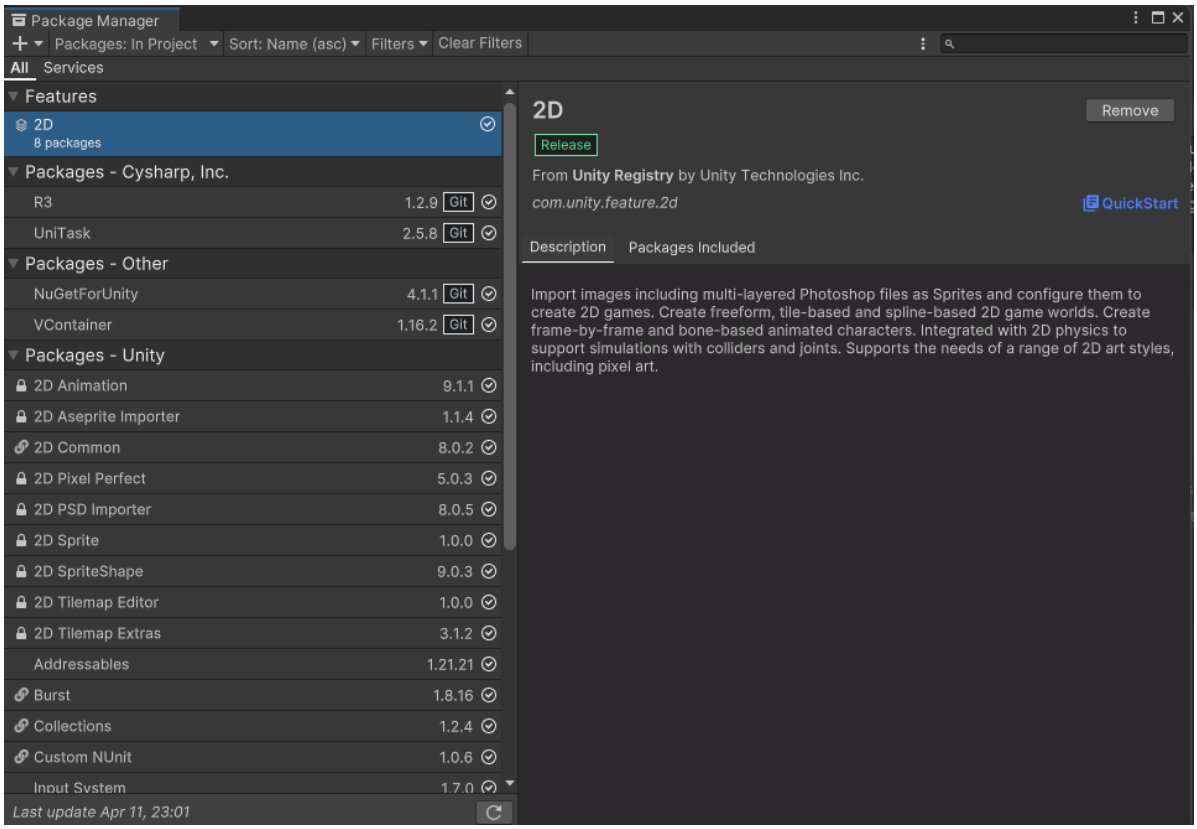


Рисунок 3.1 – Вікно менеджеру пакетів у Unity. Завдяки його інтерфейсу можливий імпорт необхідних пакетів з вже реалізованими аспектами, як графіка, скрипти тощо.

Джерело – [27]

Використовуючи Package Manager, у процесі створення гри були використані наступні ключові пакети:

- TextMeshPro (TMP) – розширення для роботи з текстом, що дозволяє відображати якісний текст з підтримкою шрифтів, стилів, рендерінгу та локалізації [28];
- Addressables – система управління асетами (ресурсами), яка дозволяє завантажувати ресурси в гру під час виконання застосунку. Addressables значно спрощує процес керування асетами у реальному часі [29];
- VContainer – швидка та відома реалізація DI контейнера [30], яка дозволить забути про необхідність розробників керувати життєвим циклом об’єктів під час роботи застосунку [31];
- Splines – реалізація та представлення ліній Без`є. У грі SpaceSwitch використовується для управління рухом ворогів та снарядів [32];
- DOTween – tween–движок для ease анімації значень (позиція, обертання, розмір тощо). У грі використовується для створення плавних ефектів, наприклад, появи об'єктів та у UI анімаціях [33].

Для реалізації DOD за допомогою парадигми ECS був використаний ринковий стандарт – фреймворк Entitas [34], що реалізує патерн ECS у класичному вигляді, з невеликими доповненнями для зручності розробки і сприйняття коду.

Entitas використовує data–oriented підхід – всі дані гри організовані у вигляді простих структур (компонентів) (рис 3.2 приклад), а вся поведінка реалізується в системах (systems). Компоненти використовуються не тільки як представлення даних, але і як маркери для фільтрації сутностей для роботи систем. Так системи отримують тільки сутності, які підпадають під повні вимоги фільтрації, оминаючи інші, без необхідності подальшої перевірки наявності чи відсутності певного набору даних на кожній сутності (рис 3.3 приклад).

```
7 usages  Yaroslav Kyzyk  1 exposing API
[Game] public class Speed : IComponent { public float Value; }
7 usages  Yaroslav Kyzyk  1 exposing API
[Game] public class Direction : IComponent { public Vector2 Value; }
7 usages  Yaroslav Kyzyk  1 exposing API
[Game] public class Velocity : IComponent { public Vector2 Value; }
7 usages  Yaroslav Kyzyk  1 exposing API
[Game] public class ClampedZ : IComponent { public float Value; }
3 usages  Yaroslav Kyzyk
[Game] public class Moving : IComponent { }
```

Рисунок 3.2 – Компоненти, які описують дані руху об'єктів

Джерело розроблено автором

```
1 usage  Yaroslav Kyzyk
public sealed class VelocityUpdateSystem : IExecuteSystem
{
    private readonly IGroup<GameEntity> _entities;

    Yaroslav Kyzyk
    public VelocityUpdateSystem(GameContext context)
    {
        _entities = context.GetGroup(GameMatcher
            .AllOf(
                GameMatcher.Speed,
                GameMatcher.Direction,
                GameMatcher.Velocity
            ));
    }

    Yaroslav Kyzyk
    public void Execute()
    {
        foreach (GameEntity entity in _entities)
        {
            entity.ReplaceVelocity(entity.Speed * entity.Direction);
        }
    }
}
```

Рисунок 3.3 – Система калькуляції швидкості об'єкту у просторі. У системі наявний фільтр IGroup на певний набор компонентів, який дасть цій системі тільки ті сутності, які підпадати під логіку обробки

Джерело розроблено автором

Подібна структура з розподіленням та оперуванням даними та логікою дозволяє досягати таких переваг:

- мінімальний зв'язок між об'єктами;
- висока модульність коду – системи можна легко тестувати та використовувати повторно у різних проектах;
- можливість легко додавати/відключати механіки без втручання у глибинні зв’язки у коді проекту;
- простота профілювання та оптимізації (всі дані розділені по контекстах).

Для інтеграції нових компонентів, описаних розробниками Entitas використовує власний code generator, що створює допоміжні API для роботи з цими компонентами: доступ до полів, фільтрацію сутностей, пул компонентів відбувається за допомогою згенерованих частин коду (рис 3.4 та рис 3.5). Це значно підвищує читабельність та розуміння коду розробниками, уникаючи написання шаблонного коду для зв’язку між компонентами та фреймворком.

Після налаштування середовища для генерації, фреймворк Entitas генерує великий обсяг допоміжного коду під час виконання налаштованої для цього команди, а саме:

- генеруються ID компонентів для їх швидкого пошуку;
- автоматично створюються Matcher–и – об'єкти, що дозволяють фільтрувати сутності за наявністю певних компонентів (рис 3.4 приклад);
- створюються розширення типу entity.AddComponentX() або entity.ReplaceComponentX() для роботи з компонентами без необхідності писати вручну логіку додавання або видалення (рис 3.5 приклад).

```
public sealed partial class GameMatcher {

    static Entitas.IMatcher<GameEntity> _matcherSpeed;

    5 usages  Yaroslav Kyzyk
    public static Entitas.IMatcher<GameEntity> Speed {
        get {
            if (_matcherSpeed == null) {
                var matcher = (Entitas.Matcher<GameEntity>)Entitas.Matcher<GameEntity>.AllOf(GameComponentsL
                matcher.componentNames = GameComponentsLookup.componentNames;
                _matcherSpeed = matcher;
            }

            return _matcherSpeed;
        }
    }
}
```

Рисунок 3.4 – Згенерований фільтр для GameEntity на пошук та використання компонента швидкості, який може бути використаний у системі для отримання сутностей з цим компонентом.

Джерело розроблено автором

```
public partial class GameEntity {

    1 usage  Yaroslav Kyzyk
    public Code.Gameplay.Features.Movement.Speed speed { get { return (Code.Gameplay.Features.Movement.Speed)GetComponent(GameComponentsLookup.Speed); } }
    5 usages  Yaroslav Kyzyk
    public float Speed { get { return speed.Value; } }
    Yaroslav Kyzyk
    public bool hasSpeed { get { return HasComponent(GameComponentsLookup.Speed); } }

    5 usages  Yaroslav Kyzyk
    public GameEntity AddSpeed(float newValue) {
        var index = GameComponentsLookup.Speed;
        var component = (Code.Gameplay.Features.Movement.Speed)CreateComponent(index, typeof(Code.Gameplay.Features.Movement.Speed));
        component.Value = newValue;
        AddComponent(index, component);
        return this;
    }

    Yaroslav Kyzyk
    public GameEntity ReplaceSpeed(float newValue) {
        var index = GameComponentsLookup.Speed;
        var component = (Code.Gameplay.Features.Movement.Speed)CreateComponent(index, typeof(Code.Gameplay.Features.Movement.Speed));
        component.Value = newValue;
        ReplaceComponent(index, component);
        return this;
    }

    Yaroslav Kyzyk
    public GameEntity RemoveSpeed() {
        RemoveComponent(GameComponentsLookup.Speed);
        return this;
    }
}
```

Рисунок 3.5 – API GameEntity для керування компонентом швидкості. Через методи AddSpeed та RemoveSpeed користувач може керувати наявністю компонента Speed для сутності.

Джерело розроблено автором

Генерація дає доступ до зручного API, який спрощує орієнтування у кодовій базі та написання ігрової логіки, бо описаний інтерфейс зрозумілою до сприйняття мовою.

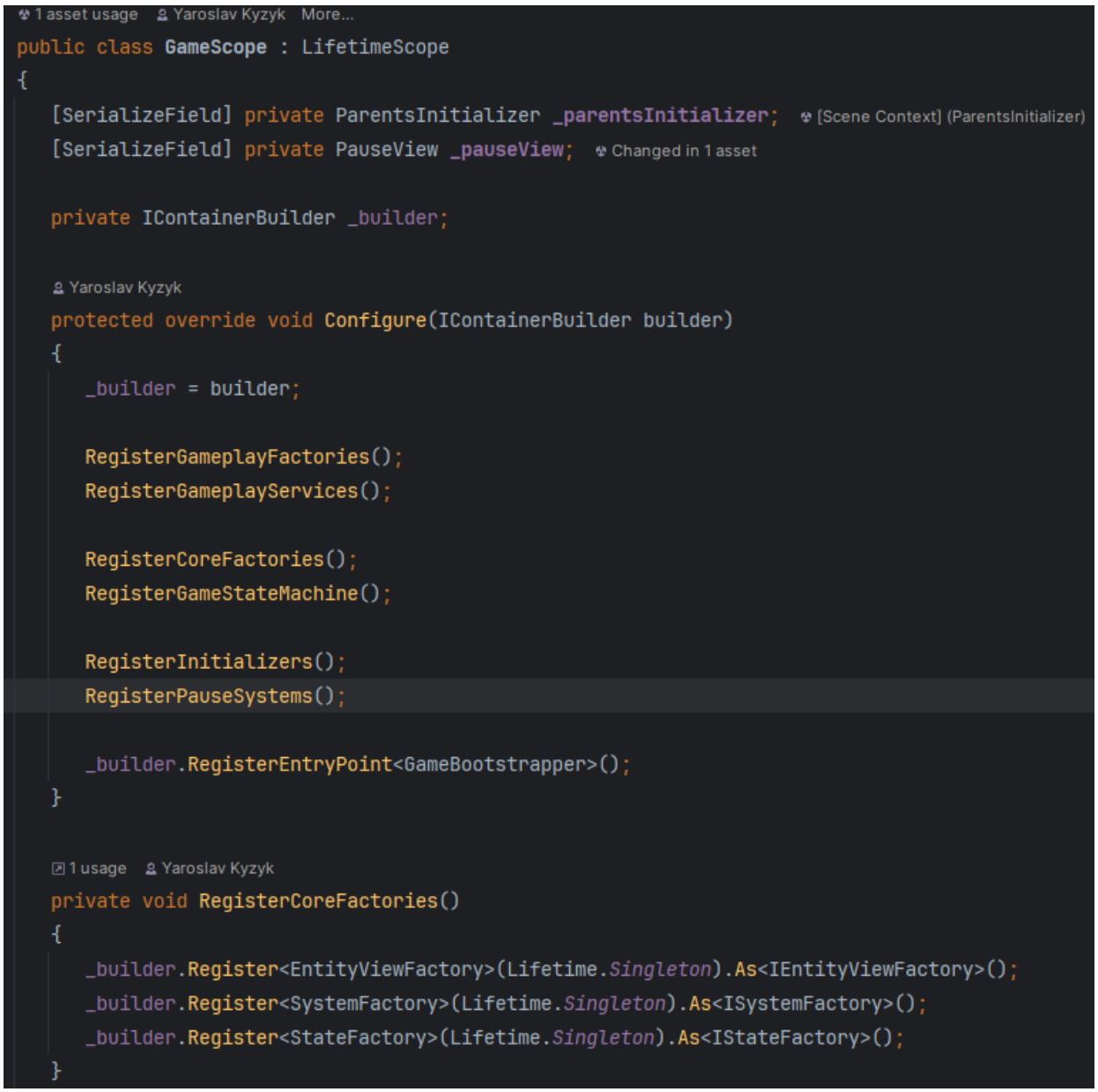
Завдяки Entitas, розробка гри «Space Switch» ведеться в структурованій та контрольованій середі, що дозволяє масштабувати гру, розділяти логіку між членами команди, а також спрощує налагодження та тестування, а згенерований API спрощує для розробників сприйняття кодової бази.

3.2 Опис архітектури

Архітектура проекту «Space Switch» побудована з дотриманням принципів чистого коду [35], інверсії залежностей (Dependency Inversion) [36], ізоляції відповідальності, а також компонентного й станового підходів. В основі гри – добре структурована ECS архітектура, підтримана DI – контейнером, розширеною відомими патернами, як FSM [37] та поділом коду на інфраструктурну й геймплейну частину.

Для керування життєвим циклом об’єктів, а також для впровадження залежностей було використано DI – контейнер. Його функція полягає у реєстрації класів (рис 3.6), які наявні у проекті, їх створенням на початку роботи програми та знищенням при необхідності, і, що найважливіше – надання необхідних екземплярів у інші класи. Це дозволяє не перейматися про необхідність описувати посилання на кожний екземпляр класу, передавати його у інші та знищувати – тепер цим займається контейнер.

«Space Switch» використовує популярне для Unity рішення – VContainer [31], що є швидкою та простою для сприйняття реалізацією DI – контейнера. Його реалізація дозволяє відокремити залежності від логіки об'єктів, спростити підтримку коду, його тестування а також забезпечити чіткий життєвий цикл цих об’єктів.



```
1 asset usage  Yaroslav Kyzyk  More...
public class GameScope : LifetimeScope
{
    [SerializeField] private ParentsInitializer _parentsInitializer;  [Scene Context] (ParentsInitializer)
    [SerializeField] private PauseView _pauseView;  Changed in 1 asset

    private IContainerBuilder _builder;

    Yaroslav Kyzyk
    protected override void Configure(IContainerBuilder builder)
    {
        _builder = builder;

        RegisterGameplayFactories();
        RegisterGameplayServices();

        RegisterCoreFactories();
        RegisterGameStateMachine();

        RegisterInitializers();
        RegisterPauseSystems();

        _builder.RegisterEntryPoint<GameBootstrapper>();
    }

    1 usage  Yaroslav Kyzyk
    private void RegisterCoreFactories()
    {
        _builder.Register<EntityViewFactory>(Lifetime.Singleton).As<IEntityViewFactory>();
        _builder.Register<SystemFactory>(Lifetime.Singleton).As<ISystemFactory>();
        _builder.Register<StateFactory>(Lifetime.Singleton).As<IStateFactory>();
    }
```

Рисунок 3.6 – Основний scope сервісів проекту, де відбувається реєстрація сервісів у контейнер

Джерело розроблено автором

У проекті, він виконує наступні функції:

- створює простори залежностей (scopes), які відокремлюють різні масиви об’єктів, ізолюючи їх взаємодію із іншими;
- реєструє ігрові та системні об’єкти для подальшого створення;

- впорядковує порядок ініціалізації та знищення об’єктів через прописаний список реєстрованих об’єктів;
- надає доступу до об’єктів із будь–якої частини коду, не порушуючи принципів слабкого зв’язування.

Основний ігровий цикл (game loop) реалізовний за допомогою патерну Finite State Machine (FSM). Кожен стейт представляє собою окремий етап роботи застосунку (рис 3.7):

```
public class GameLoopState : EndOfFrameExitState
    0+1 usages  Yaroslav Kyzyk
    public override void Enter()
    {
        _gamePlaygroundFeature = _systems.Create<GamePlaygroundFeature>();
        _gamePlaygroundFeature.Initialize();

        _spawnService.StartEnemySpawning();
        _curtain.Hide().Forget();
    }

    0+1 usages  Yaroslav Kyzyk
    protected override void OnUpdate()
    {
        _gamePlaygroundFeature.Execute();
        _gamePlaygroundFeature.Cleanup();
    }

    0+1 usages  Yaroslav Kyzyk
    protected override void ExitOnEndOfFrame()
    {
        _spawnService.StopEnemySpawning();

        _gamePlaygroundFeature.DeactivateReactiveSystems();
        _gamePlaygroundFeature.ClearReactiveSystems();

        DestructEntities();

        _gamePlaygroundFeature.Cleanup();
        _gamePlaygroundFeature.TearDown();
        _gamePlaygroundFeature = null;
    }
```

Рисунок 3.7 – Стейт головного циклу гри. Він керує ECS системами, після їх ініціалізації у попередніх стейтах.

Джерело розроблено автором

Кожен стан інкапсулює власну логіку, чітко розподіляючи ігровий цикл на частини:

- GameBootstrapState – первинна ініціалізація, завантаження конфігурацій, Addressables, створення об’єктів.
- GameLoopState – запуск, інтегрування та обробка ECS логіки.
- GameOverState – відкриття вікна програшу та перераховує рахунок.

Архітектурно проект поділено на декілька ключових модулів, що відповідають різним аспектам гри (табл 3.1):

Таблиця 3.1 – Опис архітектурних модулів у грі

Модуль	*Зміст*
Infrastructure	Базові сервіси, scopes DI–контейнера, патерн FSM, Addressables loader, Scene loader, save–load system, input тощо
Common	Спільні для використання ресурси та кодові пресети (системи, додаткові типи даних, розроблені автором, extension тощо)
UI	Реалізація вікон, менеджер UI (UIService), шаблони для навігації між вікнами та логіка цих вікон
Menu	Логіка у головного меню – показ рекорду очок, UI для старту та кінця гри, переходи до геймплею
Gameplay	Основна логіка геймплею, ECS–системи, сутності, компоненти об'єднані у окремі фічі
StaticData / Configs	Скриптовані об’єкти з конфігураціями, які завантажуються через Addressables та зберігають статичну інформацію, яка не оновлюватиметься у результаті дії гравця

Таке структурування дозволяє легко масштабувати проект, уникаючи циклічних залежностей, і підтримувати розділення обов'язків у модулях, при цьому не лишаючи можливості розробника повторно використовувати спільні дані.

Самі сцени Unity також поділено за логічними етапами у грі:

- Boot – мінімальна сцена завантаження того, що гравець не має бачити, але що є важливою частиною для систем. Гола сцена, на якій буде ініціалізовано ресурси за екраном завантаження;
- Menu – сцена з UI–елементами головного меню, де гравець може обрати між початком гри та виходом з нею, а також подивитися останній найвищий рейтинг;
- Game – сцена, в якій розгортається геймплей, який базується на ECS. Запуск усіх систем, спавн об’єктів і обробка усіх фіч за їх порядком.

Така архітектура реалізовує поетапне завантаження (lazy loading) [38], з чіткою відповідальністю кожного модуля за свій набір даних та логіку, зменшенням часу першого запуску ігрового процесу.

Геймплей у грі «Space Switch» реалізовано за допомогою так званих фіч (feature), кожна з яких відповідає за окремий аспект гри: рух, стрільба, зіткнення, спавн, ефекти тощо. Це дозволяє не лише ізолювати функціональність, а й підтримувати масштабованість, модульність і зрозумілий життєвий цикл об’єктів (рис 3.8).

Це дозволяє легко керувати порядком виконання систем, активувати / деактивувати фічі за необхідності та у перспективі масштабувати логіку уникаючи втручання у вже наявну логіку будуванням шлюзів зв’язку між системами – все виконується послідовно без прив’язки один до одного через код.

Таким чином, на базі описаних вище модулів та пакетів було розроблено стійку основу для архітектури, яка дозволить зосередитись на геймплейній частині, відокремивши її у модуль із своїм життєвим циклом та своєю внутрішньою ієрархією.

```
1 usage  Yaroslav Kyzyk
public class MovementFeature : Feature
{
    Yaroslav Kyzyk
    public MovementFeature(ISystemFactory systems)
    {
        Add(systems.Create<DirectionalDeltaMoveSystem>());
        Add(systems.Create<OrbitalDeltaMoveSystem>());
        Add(systems.Create<OrbitCenterFollowSystem>());

        Add(systems.Create<TurnAlongDirectionSystem>());

        Add(systems.Create<UpdateTransformPositionSystem>());
        Add(systems.Create<UpdateTransformRotationSystem>());
        Add(systems.Create<VelocityUpdateSystem>());

        Add(systems.Create<ClampZPositionSystem>());

        Add(systems.Create<RotateAlongDirectionSystem>());
    }
}
```

Рисунок 3.8 – Фіча руху об'єктів, яка об'єднує у собі різні системи – від калькуляції напрямку, до окремих реалізації руху.

Джерело розроблено автором

3.3 Реалізація геймплейної частини

Основна механіка скролл–шутера – видимість нескінченного лінійного руху реалізована класичним шляхом: замість того, щоб рухати гравця, рухається самі частини бекграунду, підставляючись перед початком обзору камери та зникаючи як тільки вони вийшли з нього (рис 3.9).

Керування кораблем гравця здійснюється через систему обробки вводу, що містить обробку натискань клавіш та кнопок миші. Відповідна система інтерпретує введення користувача у вигляді запитів на зміну позиції, які потім перехоплюються іншими системами, такими як система руху сутностей, або система пострілів.

Для гравця та ворогів логіка для пересування у просторі описана у одній і тій же системі, що значно спрощує сприйняття і коду, і логіки застосунку – коли подібні за поведінкою сутності описані одними і тими самими інструментами. Також, система пострілів реалізована через спавн об'єктів-снарядів з однієї фабрики, за тією ж логікою, але за різними умовами – для гравця від input, для ворогів за таймером.

```
public sealed class HandleLevelPartsSwitchSystem : IExecuteSystem
    Yaroslav Kyzyk
    public HandleLevelPartsSwitchSystem(GameContext context, ICameraProvider cameraProvider, Lev
    {
        _cameraProvider = cameraProvider;
        _levelPartsHandleService = levelPartsHandleService;
        _levelParts = context.GetGroup(GameMatcher
            .AllOf(
                GameMatcher.LevelPart,
                GameMatcher.WorldPosition
            ));
    }

    Yaroslav Kyzyk
    public void Execute()
    {
        foreach (GameEntity levelPart in _levelParts.GetEntities(_buffer))
        {
            if (levelPart.LevelPart.TopRight.y < _cameraProvider.WorldLeftBottomBoundPosition.y)
            {
                levelPart.isDestructed = true;
                _levelPartsHandleService.SetLevelPartToPool(levelPart.LevelPart);
            }

            if(_levelPartsHandleService.LastCreatedPart != levelPart.LevelPart)
                continue;

            if (levelPart.LevelPart.TopRight.y < _cameraProvider.WorldRightTopBoundPosition.y)
                _levelPartsHandleService.SetNextPart(levelPart.LevelPart);
        }
    }
}
```

Рисунок 3.9 – Система, яка керує позиціонуванням частин бекграунду. Перевіряючи у кожної частини, система або прибирає частину з поля зору або додає нову.

Джерело розроблено автором

Система спавну ворогів відповідає за періодичне створення ворожих сутностей у грі. Вона ініціалізується при старті рівня й підтримує таймер, що викликає створення нових ворогів за заданою чергою у конфігах. Кожен створений ворог отримує власну логіку руху за сплайном [39] (рис 3.10), яка реалізована у вигляді execute–системи. Вона кожен тік змінює позицію ворога, змушуючи його рухатись та обертатись по траєкторії сплайну.

Особливістю «Space Switch» є механіка надання кольору об’єкту. Колір об’єкта являє собою властивість, від якої залежить, чи може снаряд нанести шкоду цьому об’єкту. У разі якщо кольори об’єкту та снаряду не співпадають – то снаряд ігнорує об’єкт.

Оскільки механіки працюють незалежно одна від одної, та їх можна легко комбінувати – надавати сутностям різний колір в залежності від скрипту (гравцю за input, ворогу від його скрипт-поведінки) зміни кольору. При зміні кольору виконується активація відповідного рендеру за зміною матеріалу та візуальних ефектів, а також може змінюватись колір снарядів, який об’єкт створює (рис 3.10).

Ключа механіка шутера – стрільба виконує спавн снарядів за записати певної сутності, надаючи снаряду необхідні компоненти та властивості (рис 3.11). Сутність (у даному випадку, гравець) випускає снаряд, який може нанести шкоду ворожій сутності (для ворога – гравець, для гравця – ворог відповідно) та може бути наділений різними особливостями, наприклад різною траєкторією руху, або самознищенням через певний проміжок часу, при недосягненні якоїсь цілі.

Особливістю «Space Switch» є механіка надання кольору об’єкту. Колір об’єкта являє собою властивість, від якої залежить, чи може снаряд нанести шкоду цьому об’єкту. У разі якщо кольори об’єкту та снаряду не співпадають – то снаряд ігнорує об’єкт.

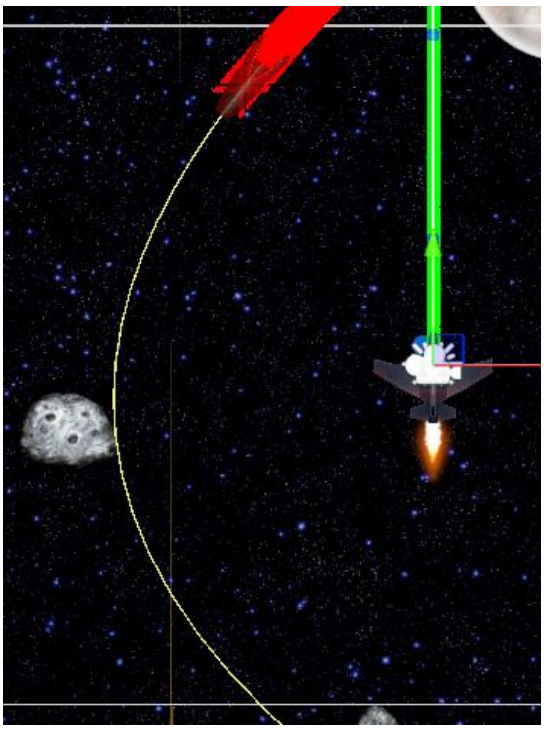


Рисунок 3.10 – Візуалізація шляху руху ворога за сплайном

Джерело розроблено автором

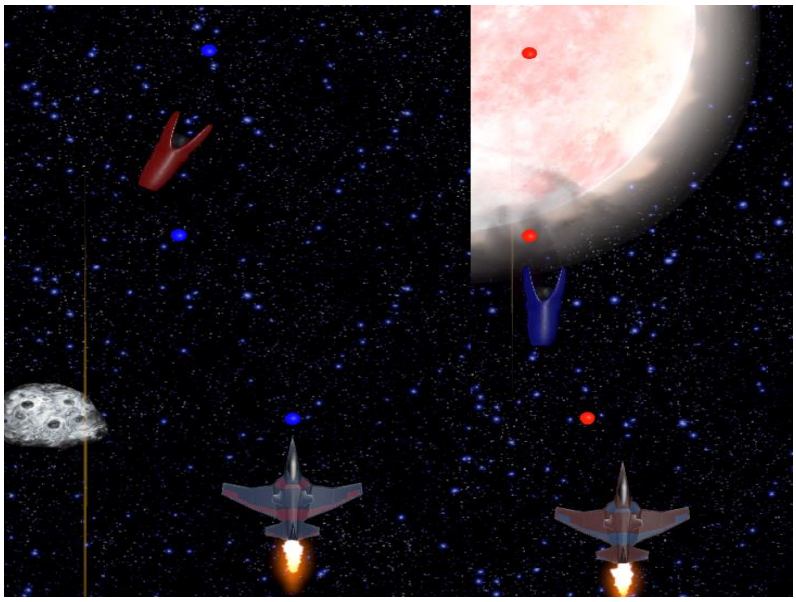


Рисунок 3.11 – Візуалізація роботи систем стрільби гравця, та зміні кольорів об’єктів

Джерело розроблено автором

Останньою важливою частиною є система керування буфером зіткнень. Вона дає можливість розділити момент детермінації зіткнення та його обробку, що дозволяє уникнути непередбачуваних ефектів у межах одного кадру. Компоненти колайдера [40] на об'єкті в Unity додаються до буфера сутності, та за проходженням по цій сутності системи колізій, вона буде оброблена належним чином у контексті execute–систем (рис 3.12).

```
public class EntityBehaviour : MonoBehaviour, IEntityView
    Yaroslav Kyzyk
    private void Construct(ICollisionRegistry collisionRegistry) =>
        _collisionRegistry = collisionRegistry;

    6 usages   Yaroslav Kyzyk
    public void SetEntity(GameEntity entity)
    {
        _entity = entity;
        _entity.AddView(this);
        _entity.Retain(this);

        foreach (IEntityComponentRegistrar registrar in GetComponentsInChildren<IEntityComponentRegistrar>
            registrar.RegisterComponents();

        foreach (Collider col in GetComponentsInChildren<Collider>(includeInactive: true))
            _collisionRegistry.Register(col.GetInstanceID(), _entity);
    }
    0+1 usages   Yaroslav Kyzyk   More...
    public IEnumerable<GameEntity> SphereCast(Vector3 position, float radius, int layerMask)
    {
        int hitCount = Physics.OverlapSphereNonAlloc(position, radius, OverlapHits, layerMask);

        DrawDebug(position, radius, 1f, Color. red);

        for (int i = 0; i < hitCount; i++)
        {
            GameEntity entity = _collisionRegistry.Get<GameEntity>(OverlapHits[i].GetInstanceID());
            if (entity == null)
                continue;

            yield return entity;
        }
    }
```

Рисунок 3.12 – Система реєстрації колайдерів. Під час виконання методу SetEntity колайдери реєструються, для подальшого використання, наприклад у методі SphereCast.

Джерело розроблено автором

3.4 Тестування програмного продукту

Тестування програмного продукту було здійснено з урахуванням специфіки предметної області та технологій, які лежать у основі розробки, зокрема підходу ECS. Процес тестування складався з трьох основних етапів: тестування функціоналу командою QA, unit-тестування [41] основних модулів, а також інтеграційне тестування фіч.

Для тестування функціональних вимог застосунку було сформовано команду з трьох QA-спеціалістів, яка впродовж тижня проводила тестування фінального білду застосунку (табл 3.2):

Таблиця 3.2 – Опис команди тестування, конфігурацій та розподілення обов'язків між членами команди QA.

Посада	*Конфігурація*	*Задачі*
Lead	Windows 11, PC, Ryzen 5 3600x	Тестування інфраструктурних аспектів, менеджмент пам'яттю, ключових вимог, важких кейсів
Middle	Windows 10, PC, Intel Core i9-14900k	Тестування важливих функціональних вимог застосунку, кейсів середньої складності геймплею та інфраструктури
Junior	Windows 11, Asus Vivobook 15, Ryzen 5 7430U	Тестування дрібних аспектів роботи геймпленої частини, взаємодії об'єктів та документування

Структура команди включала lead-тестувальника, відповідального за координацію процесу, та двох функціональних тестувальників різного рівня кваліфікації для ефективного розподілу роботи.

Команда використовувала стандартну методику ручного тестування black-box [42] на основі use-cases та чек-листів від розробника для більш точкового тестування окремих аспектів застосунку. Це дозволило охопити весь масив можливих кейсів роботи застосунку під час експлуатації. Усі виявлені баги документувалися в системі таск-трекінгу Notion [43], де були оброблені розробником (рис 3.13).

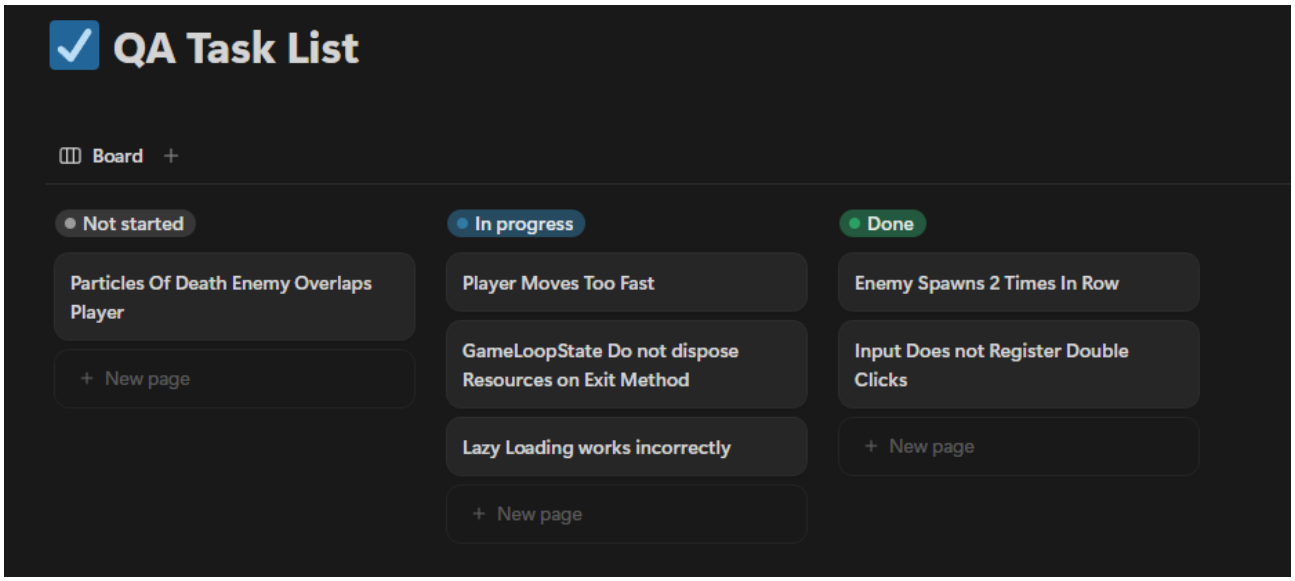


Рисунок 3.13 – Ілюстрація дошки із документуванням багів у застосунку Notion

Джерело розроблено автором

Використовуючи принципи LiveOps [44] та Agile [45], тестування проводилось кожен день для безперервної інтеграції правок та виправлення помилок, багів у нових версіях застосунку допоки не був отриманий кінцевий результат застосунку.

Для більш детального тестування дрібних аспектів застосунку та впевненості у вірності роботи цих аспектів, було використано unit-тестування за допомогою вбудованих інструментів фреймворку Unity (рис 3.14).

```
6   public class CooldownSystemTests
10      private TestTimeService _time;
11
12      [SetUp]
        IL code
13      public void SetUp()
14      {
15          _context = new GameContext();
16          _time = new TestTimeService { DeltaTime = 0.02f };
17          _system = new CooldownSystem(_context, _time);
18      }
19
20      [Test]
        IL code
21      public void Cooldown_Decreases_Correctly()
22      {
23          var entity = _context.CreateEntity();
24          entity.AddCooldown(2f);
25          entity.AddCooldownLeft(1.5f);
26
27          _system.Execute();
28
29          Assert.AreEqual(1.0f, entity.CooldownLeft, 0.0001f);
30          Assert.IsFalse(entity.isCooldownUp);
31          Assert.IsTrue(entity.hasCooldownLeft);
32      }
33
34      [Test]
        IL code
35      public void Cooldown_Expires_And_Flag_Set()
36      {
37          var entity = _context.CreateEntity();
38          entity.AddCooldown(1f);
39          entity.AddCooldownLeft(0.4f);
40
41          _system.Execute();
42
43          Assert.IsTrue(entity.isCooldownUp);
44          Assert.IsFalse(entity.hasCooldownLeft);
45      }
```

Рисунок 3.14 – Приклад unit-тестів для модуля Cooldown. Імітуючи роботу системи через ручне вписування часу, перевіряється чи закінчився таймер за відведений час

Джерело розроблено автором

Особливістю юніт-тестування архітектурі на ECS є те, що система ділиться на чітко ізольовані компоненти, які не мають внутрішньої логіки, а лише зберігають дані, що спрощує написання логіки тестування та зменшує погрішності через відсутність зв’язків між різними модулями. Основні метрики, які використовувалися для оцінки якості тестів:

- покриття відсотку коду тестами;
- час виконання тестів;
- наявність негативних кейсів під час виконання пайплайну тестування;

Було написано низку тестів, які гарантували коректність роботи функціональної логіки без необхідності залучати QA команду для процесу тестування результату невеликих змін у коді.

Враховуючи специфіку підходу ECS, інтеграційне тестування окремих аспектів було поділено на тестування окремих фіч та їх взаємодії одна з одною. На базі юніт-тестів та із залученням команди тестування була протестована інтеграція фіч зміни кольору із фічею пострілів. За допомогою юніт-тестів було визначено, що окремі логічні аспекти у системах працюють коректно ізольовано у своїх системах, у той час, коли команда тестувальників провела перевірку та документування сумісної роботи систем.

У результаті, тестування показало бажану сумісну поведінку фіч – снаряди від пострілів набували кольору об’єкта, що їх випускав. За таким принципом було проведено інтеграційні тестування інших фіч у проекті.

Загалом, використання різних підходів до тестування (QA, юніт-тестування та ізольоване) у комплексі забезпечило ефективну перевірку функціональних вимог продукту.

3.5 Використання програмного продукту

За результатами проведеної роботи було розроблено повноцінний MVP продукт 2D-скролера [46], який відповідає поставленим вимогам, реалізовує ігровий цикл, що дозволяє користувачеві експлуатувати основні use-cases продукту.

Стартує гра у головному меню (рис 3.15), де користувач може побачити останній рейтинг очок на девайсі та або почати гру, або закінчити роботу застосунку.

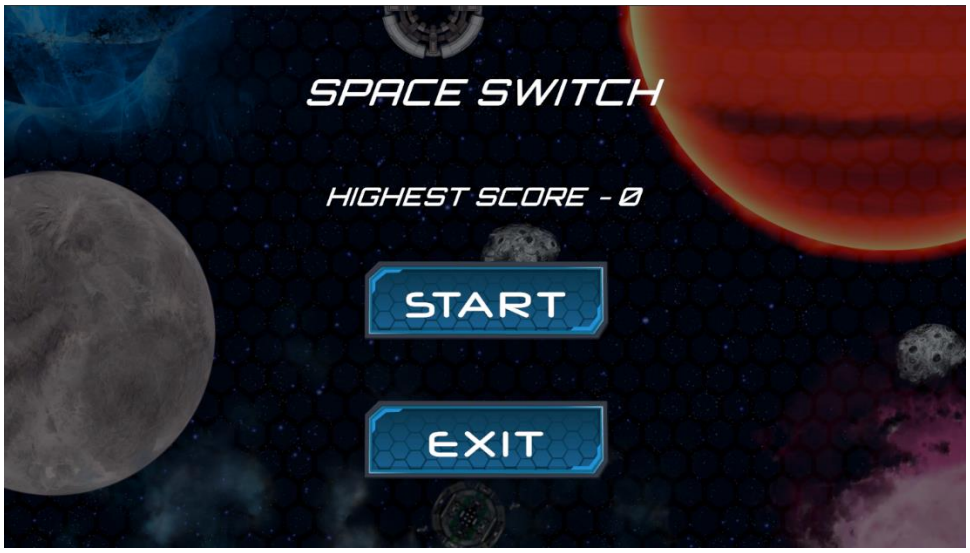


Рисунок 3.15 – Ілюстрація головного меню з рейтингом очок та кнопками для подальшого вибору дій
Джерело розроблено автором

Після натискання на кнопку «Start» гравець починає гру, отримуючи можливість керувати своїм кораблем (рис 3.16). Гравець має наступний можливий набір дій щодо свого аватару:

- керувати напрямом руху корабля (WASD / стрілочки Up/Left/Down/Right);
- змінити колір корабля (Space/Right Mouse Button)

Корабель здійснює постріли автоматично з певною періодичністю, гравець лише може керувати положенням корабля та його кольором, який впливає на колір снарядів, що з корабля вилітають.

У разі необхідності, гравець може поставити гру на паузу, у меню якої (рис 3.16) буде відображатись кнопки для дій – продовження ігрового процесу, виход до меню та рестарт нинішньої гри.

Рисунок 3.16 – Ілюстрація меню паузи. На панелі зображені кнопки продовження гри, виходу до меню та рестарту
Джерело розроблено автором

Ігра продовжується до моменту виходу за бажанням гравця, або ж через смреть його аватара-корабля. У другому випадку, основні процеси у грі, такі як спавн нових ворогів та нарахування очок завершується, а гравець побачить меню «Game Over» (рис 3.17), на якому буде зображено кількість набитих очок. У цьому ж меню гравець може обрати між перезапуском гри, або виходом до меню.

Рисунок 3.17 – Ілюстрація меню закінчення гри. На панелі відображається кількість набитих за гру очок та кнопки рестарту і виходу до меню
Джерело розроблено автором

Висновки до розділу 3

На етапі реалізації «Space Switch» було продемонстровано процес створення гри з акцентом на програмному забезпеченні та реалізації технічних вимог продукту.

Особливу увагу було приділено до опису інструментів та середовища, яке відіграє ключову роль у розробці. Unity як ігрового рушія, низки його популярних пакетів та фреймворку ECS – Entitas дало змогу запровадити зручний процес розробки програмного забезпечення.

Архітектура застосунку була реалізована з використанням Unity пакетів та архітектурних патернів, таких як FSM. Це дало змогу побудувати фундамент кодової бази застосунку, який може легко бути розширений під будь які технічні вимоги, не ризикуючи стабільністю вже існуючих систем.

Описано реалізацію основних геймплейних складових застосунку, їх особливості та їх взаємодію одна з одною. Продемонстровано їх вигляд у грі та принцип роботи у програмному коді.

Проведене тестування з використанням як живих спеціалістів команди QA, так і програних unit-тестів забезпечило чітке виконання вимог застосунку та стабільність під час експлуатації.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було розроблено гру-скролер «Space Switch» з використанням data-oriented design. У ході розробки було проведено аналіз конкурентів та пошук можливостей для задоволення потреб користувачів та всіх вимог, висунутих у ході аналізу.

Під час етапу проектування було враховано досвід конкурентів, запроваджено використання парадигми ECS та розбити основну частину програми на окремі модулі, що дозволило спланувати стійку до змін архітектуру застосунку.

Розробка здійснювалася на рушію Unity з використанням його пакетів а також фреймворку Entitas. Ці інструменти дали змогу реалізувати низку важливих фіч, такі як спавн ворогів, зчитування вводу гравця, рух заднього фону та зміну кольорів сутностей, маючи можливість легко впровадити нові елементи, не порушуючи логіки існуючих. Проведене тестування забезпечило перевірку стабільності роботи застосунку.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Herman Narula – A billion new players are set to transform the gaming industry. URL: https://www.wired.com/story/worldwide-gamers-billionplayers (дата звернення: 14.04.2025).
2. James Batchelor – Video games to pass $300bn revenue, 3.8 billion players by 2030 // Games Industry – Midia Research URL: https://www.gamesindustry.biz/midia-research-video-games-to-pass-300bn-revenue-38-billion-players-by-2030 (дата звернення: 19.04.2025).
3. Pokesaur – Game Theory: Video Game Genres Venn Diagram // WordPress URL: https://pokesaur.wordpress.com/2011/05/25/game-theory-video-game-genres-venn-diagram/ (дата звернення: 19.04.2025).
4. ResearchGate – Game production pipeline overview. URL: https://www.researchgate.net/figure/Game-production-pipeline-overview-Concept-content-creation-pipeline-level-design_fig2_267417785 (дата звернення: 19.04.2025).
5. Nataliya Revutska – What is OOP // SoftServe URL: https://career.softserveinc.com/en-us/stories/what-is-object-oriented-programming-oop-explaining-four-major-principles (дата звернення: 19.04.2025).
6. Arpanext – A Comprehensive Guide To Data-Oriented Design For Improved Software Efficiency // Medium URL: https://arpanext.medium.com/a-comprehensive-guide-to-data-oriented-design-for-improved-software-efficiency-6434d520d0e4 (дата звернення: 19.04.2025).
7. Leo – Why is Entity Component System (ECS) so awesome for game development? // Medium URL: https://medium.com/source-true/why-is-entity-component-system-ecs-so-awesome-for-game-development-f554e1367c17 (дата звернення: 19.04.2025).
8. Nidorx – Tiny and easy to use ECS (Entity Component System) library for game programming // GitHub URL: https://github.com/nidorx/ecs-lib (дата звернення: 19.04.2025).
9. Denis Kondratev – Exploring Unity DOTS and ECS: Is it a Game Changer? // Hackernoon URL: https://hackernoon.com/exploring-unity-dots-and-ecs-is-it-a-game-changer (дата звернення: 19.04.2025).
10. Wikipedia – Galaga. URL: https://uk.wikipedia.org/wiki/Galaga (дата звернення: 19.04.2025).
11. Wikipedia – R-Type. URL: https://uk.wikipedia.org/wiki/R-Type (дата звернення: 19.04.2025).
12. Techtarget – role-playing game (RPG). URL: https://www.techtarget.com/whatis/definition/role-playing-game-RPG#:~:text=A%20role%2Dplaying%20game%20(RPG)%20is%20a%20game%20in,BattleTech%2C%20and%20Star%20Wars%20Galaxies. (дата звернення: 19.04.2025).

13. Wikipedia – Sky Force. URL: https://en.wikipedia.org/wiki/Sky_Force (дата звернення: 19.04.2025).
14. Steam – Razor2: Hidden Skies. URL: https://store.steampowered.com/app/34920/Razor2_Hidden_Skies/?l=ukrainian (дата звернення: 19.04.2025).
15. Google Play – Hawk: Freedom Squadron. URL: https://play.google.com/store/apps/details?id=com.my.hawk.air.shooter (дата звернення: 19.04.2025).
16. Steam – Sky Force Reloaded. URL: https://store.steampowered.com/app/667600/Sky_Force_Reloaded/ (дата звернення: 19.04.2025).
17. Oleksiy Pletnov – Open source: що це, для чого і як розпочати // DOU URL: https://dou.ua/lenta/articles/open-source-reasons-to-join/ (дата звернення: 19.04.2025).
18. M.R.M Abdullah – Intro to Game Dev - Understanding the GAME LOOP // Medium URL: https://m-abdullah-ramees0916.medium.com/the-game-loop-f6f5cb68c00 (дата звернення: 19.04.2025).
19. Moises Gamio – Understanding OOP concepts // Codersite URL: https://codersite.dev/understanding-oop-concepts/ (дата звернення: 19.04.2025).
20. Unity – ECS Concepts. URL: https://docs.unity3d.com/Packages/com.unity.entities@0.2/manual/ecs_core.html (дата звернення: 19.04.2025).
21. Microsoft – C#. URL: https://dotnet.microsoft.com/ru-ru/languages/csharp (дата звернення: 19.04.2025).
22. Google – Firebase | Google's Mobile and Web App Development Platform. URL: https://firebase.google.com/ (дата звернення: 19.04.2025).
23. W3Schools – JSON – Introduction. URL: https://www.w3schools.com/js/js_json_intro.asp (дата звернення: 19.04.2025).
24. Wikipedia – 4+1 architectural view model. URL: https://en.wikipedia.org/wiki/4%2B1_architectural_view_model (дата звернення: 19.04.2025).
25. Wikipedia – Модель C4. URL: https://uk.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_C4 (дата звернення: 19.04.2025).
26. Unity – What is Unity? URL: https://unity.com/ (дата звернення: 19.04.2025).
27. Unity – Package Manager Window. URL: https://docs.unity3d.com/Manual/upm-ui.html (дата звернення: 19.04.2025).

28. Unity – TextMeshPro Documentation. URL: https://docs.unity3d.com/Packages/com.unity.ugui@2.0/manual/TextMeshPro/index.html (дата звернення: 19.04.2025).
29. Unity – Addressables. URL: https://docs.unity3d.com/Manual/com.unity.addressables.html (дата звернення: 19.04.2025).
30. Dependency Injection Principles, Practices, and Patterns / Mark Seemann, Steven van Deursen, 2019 – 552 ст.
31. Hadashikick – VContainer Documentation. URL: https://vcontainer.hadashikick.jp/ (дата звернення: 19.04.2025).
32. Unity – Splines Documentation. URL: https://docs.unity3d.com/Packages/com.unity.splines@2.4/manual/index.html (дата звернення: 19.04.2025).
33. Demigiant – DOTween Documentation. URL: https://dotween.demigiant.com/ (дата звернення: 19.04.2025).
34. Sschmid – Entitas – The Entity Component System Framework for C# and Unity. URL: https://github.com/sschmid/Entitas (дата звернення: 19.04.2025).
35. Clean Code: A Handbook of Agile Software Craftsmanship / Robert C. Martin, 2008 – 416 ст.
36. Matthias Schenk – SOLID - Dependency Inversion Principle (Part 5) // Medium URL: https://medium.com/@inzuael/solid-dependency-inversion-principle-part-5-f5bec43ab22e (дата звернення: 19.04.2025).
37. Refactoring Guru – State. URL: https://refactoring.guru/design-patterns/state (дата звернення: 19.04.2025).
38. MND Web Docs – Lazy Loading. URL: https://developer.mozilla.org/en-US/docs/Web/Performance/Guides/Lazy_loading (дата звернення: 19.04.2025).
39. Unity Docs – Splines Package. URL: https://docs.unity3d.com/Packages/com.unity.splines@1.0/manual/getting-started-with-splines.html (дата звернення: 19.04.2025).
40. Chris Hilton – Collider's and Triggers in Unity - Understanding the Basics. Medium // URL: https://christopherhilton88.medium.com/colliders-and-triggers-in-unity-understanding-the-basics-7192714f3440 (дата звернення: 19.04.2025).
41. Abhinav – Intro to Unit Tests. Medium // URL: https://medium.com/interleap/intro-to-unit-tests-f2b7750c2d3c (дата звернення: 19.04.2025).
42. QA TestLab – Яка різниця між Black Box & White Box Testing? URL: https://training.qatestlab.com/blog/technical-articles/whats-the-difference-between-black-box-white-box-testing/ (дата звернення: 19.04.2025).
43. Notion Website. URL: https://www.notion.com/ (дата звернення: 19.04.2025).

44. Apptica – LiveOps: Best Practices & Mechanics. Medium // URL: https://medium.com/@Apptica/liveops-best-practices-mechanics-f31615dcda85 (дата звернення: 19.04.2025).
45. Clean Agile: Back to Basics / Robert C. Martin, 2019 – 240 ст.
46. DarkHaunt – «Space Switch» // GitHub URL: https://github.com/DarkHaunt/SpaceSwitch (дата звернення: 19.04.2025).