

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**ДЕРЖАВНИЙ ЗАКЛАД  
«ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ  
ІМЕНІ ТАРАСА ШЕВЧЕНКА»**

**Навчально-науковий інститут математики та інформаційних технологій**

**Кафедри математики та інформатики**

**Чжао Дуншен**

**ДОСЛІДЖЕННЯ ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ РОЗРОБКИ  
ІГРОВИХ ПРОГРАМ ДЛЯ МОБІЛЬНИХ ПЛАТФОРМ**

**Магістерська робота  
за спеціальністю 122 «Комп'ютерні науки»**

Особистий підпис – \_\_\_\_\_

Науковий керівник – \_\_\_\_\_ д.т.н., професор Ю.Г. Козуб

В.о.зав. кафедри – \_\_\_\_\_ д.т.н., професор Ю.Г. Козуб

Полтава – 2025

## **Анотація**

*Чжао Дуншен.* Дослідження інструментальних засобів розробки ігрових програм для мобільних платформ. Кваліфікаційна робота магістра. Луганський національний університет імені Тараса Шевченка. 2025р.

У роботі проведено аналіз підходів до розробки мобільних додатків, досліджено методи побудови кривих Без'є третього порядку, розроблено алгоритм їх побудови як розширення редактору Unity, проаналізовано найбільш популярні підходи розробки штучного інтелекту для ігор та вибрано концепт Кінцевих автоматів як найзручніший варіант розробки простих ботів. Вивчено архітектурний шаблон ECS, на основі якого розроблено ігрову систему, інформаційне та програмне забезпечення казуальної гри засобами програмування C#.

Ключові слова: C#, ECS, ігровий движок, GAME ARTIFICIAL INTELLIGENCE, крива Без'є, ANDROID, UNITY, APK.

## **Abstract**

*Zhao Dongheng.* Research on the tools for developing game programs for mobile platforms. Master's degree thesis. Luhansk Taras Shevchenko National University. 2025.

The paper analyzes approaches to developing mobile applications, investigates methods for constructing third-order Bezier curves, develops an algorithm for their construction as an extension to the Unity editor, analyzes the most popular approaches to developing artificial intelligence for games, and selects the concept of Finite Automata as the most convenient option for developing simple bots. The ECS architectural template is studied, on the basis of which a game system, information and software for a casual game are developedg.

Keywords: C#, ECS, game engine, GAME ARTIFICIAL INTELLIGENCE, Bezier curve, ANDROID, UNITY, APK.

## ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ .....	6
ВСТУП .....	7
РОЗДІЛ 1 АНАЛІЗ ПРОГРАМНИХ І ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ РОЗРОБКИ ІГРОВИХ СИСТЕМ.....	10
1.1. Аналіз мов програмування для створення ігор .....	11
1.2. Аналіз існуючих програмних засобів для створення ігор .....	15
1.2.1. <i>Unity</i> .....	16
1.2.2. <i>Unreal Engine</i> .....	17
1.2.3. <i>Godot</i> .....	19
1.3. Реалізація штучного інтелекту в іграх .....	22
1.4. Висновки до розділу 1 .....	25
РОЗДІЛ 2 ПРОЕКТУВАННЯ ІГРОВОЇ СИСТЕМИ .....	26
2.1. Розробка ігрової концепції .....	27
2.2. Прототипування ігрового бота .....	28
2.3. Розробка ігрової механіки .....	29
2.4. Програмування та проектування ігрового середовища.....	31
2.4.1. Використання фреймворків .....	31
2.4.2. Побудова траєкторії руху за запланованим маршрутом ....	31
2.5. Тестування .....	42
2.6. Висновки до розділу 2 .....	43
РОЗДІЛ 3 РОЗРОБКА ІГРОВОГО ДОДАТКУ «ZIMA».....	44
3.1 Вибір та конфігурування програмного середовища .....	44
3.2 Опис використаного фреймворку.....	46
3.2.1 Концепції ECS .....	46
3.2.2 Архетипи (archetype).....	47
3.2.3 Memory chunks .....	48
3.2.4 Запити сутності .....	49
3.2.5 Організація системи .....	50
3.3 Сценарій ігрового додатку .....	50

3.4	Опис програмної реалізації .....	51
3.5	Тестування програмного забезпечення .....	54
3.6	Висновки до розділу 3 .....	57
ВИСНОВКИ.....		58
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ .....		60
ДОДАТОК.....		65

## ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

AI	-	artificial intelligence;
DOTS	-	Data-Oriented Tech Stack;
CMS	-	система управління вмістом;
ECS	-	Entity Component System;
EDGE	-	Enhanced Data rates for GSM Evolution;
UE	-	Unreal Engine;
FDA	-	Food and Drug Administration;
FSM	-	Finite-state machine;
GPRS	-	General Packet Radio Service;
XNA	-	XNA's Not Acronymed;
JS	-	JavaScript;
ВООЗ	-	Всесвітня організація охорони здоров'я;
ІКТ	-	інформаційно-комунікаційні технології;
ІС	-	інтелектуальна система;
ІТ	-	інформаційні технології;
ОС	-	операційна система;
ООП	-	об'єктно-орієнтоване програмування;
ПЗ	-	програмне забезпечення;
ШІ	-	штучний інтелект;
ПК	-	персональний комп'ютер;
СДУГ	-	синдромом дефіциту уваги і гіперактивності.

## ВСТУП

Сьогодні мобільні ігрові технології досягли того рівня, що при використанні смартфонів, планшетів, всіляких гаджетів, стали невід'ємною частиною сучасного суспільства, а у результаті кризи COVID-19 на світовому ігровому ринку в першій половині 2020 року спостерігалось безпрецедентний рівень залученості. Відеоігри, які завжди були формою розваги, все частіше стають способом впоратися з тим, що фізично не можуть впоратися навіть лікарі. Всесвітня організація охорони здоров'я (ВООЗ) запропонувала відеоігри як форму соціальної активності в цей ізольований час. Тепер існує навіть схвалений FDA рецепт на відеоігри під назвою EndeavorRX, який прописують дітям з синдромом дефіциту уваги і гіперактивності (СДУГ). Те, що колись було дозвіллям, тепер є життєва важливим для деяких людей, щоб керувати своїм життям, особливо під час ізоляції [18].

Все більша кількість людей переходить на мобільні телефони з сенсорними екранами, які в більшості своїй працюють під управлінням операційної системи Android або iOS. Швидке поширення Android пристроїв створює величезний попит на мобільні ігри, для їх створення використовується безліч різних технологій, мов, додатків і платформ. Кожен з інструментальних засобів несе полегшення та покращення використання передових технологій обробки графіки, фізики, забезпечення кросплатформеності розроблених проєктів розробки, але кожен з них має свої плюси і мінуси. Тому ця робота виконана на актуальну, на даний момент, тему «Дослідження інструментальних засобів розробки ігрових програм для мобільних платформ (C#)». Мова C# більш зручна і дружня програмна платформа. Код на C#, як правило, виглядає простіше і лаконічніше, вона підтримує такі інструменти, як Unity і Xamarin, які відмінно підходять для кросплатформеної розробки ігор і додатків [35, 53, 20].

**Метою** магістерської роботи є аналіз дослідження інструментальних засобів розробки ігрових програм для мобільних платформ, розробка методики та програмна реалізація казуальної ігрової системи з підтримкою штучного інтелекту за допомогою мови програмування C#.

Для досягнення поставленої мети в роботі виконано такі завдання:

1. Проведено аналіз найпопулярніших ігрових рушіїв для створення ігрового застосунку.
2. Розроблено ігрову логіку та визначено правила проектування мобільного додатку.
3. Створено ігрову концепцію, спроектовано ігрові механіки та підготовлено прототип дизайну гри.
4. Запропоновано використання кривих Без'є для генерування ігрового середовища та оцінювання ігрового процесу.
5. Реалізовано штучний інтелект для ігрового бота.
6. Розроблено нову мобільну гру «Zima» з використанням мови програмування C#.

Об'єкт дослідження – механізми управління проектуванням інформаційного та програмного забезпечення казуальної ігрової системи засобами C#.

Предмет дослідження – середовище розробки Unity, впровадження ігрової концепції та створених алгоритмів, а також реалізація штучного інтелекту ігрового бота.

**Методи дослідження.** Теоретичні методи: аналіз науково-технічної та навчально-методичної літератури, інтернет-ресурсів з проблеми дослідження;

**Наукова новизна** отриманих результатів полягає в створенні методики проектування мобільних додатків з впровадженням штучного інтелекту при розробці нової казуальної ігрової системи «Zima» за допомогою мови програмування C#.

**Практичне значення** отриманих результатів полягає в розробці ігрового мобільного додатку на C# з впровадженням штучного інтелекту; у використанні в якості методичного матеріалу при навчанні студентів методам програмування мультимедійних плагінів.

**Структура і обсяг роботи.** Робота складається з вступу, трьох розділів, висновків і додатків. Містить 69 сторінок друкованого тексту, 24 рисунків, 11 таблиць, список використаних джерел з 55 найменувань, додатку.



В першому розділі надано аналіз предметної області, програмних і інструментальних засобів розробки ігрових систем.

У другому розділі розглянуто основні принципи, процеси та етапи розробки проекту. Представлено ігрові концепції створення ігрової логіки та правил. Створено прототипування ігрового бота та імплементацію ігрової механіки. Обрано сучасні програмні засоби та інструменти для реалізації алгоритмів штучного інтелекту, ігрової системи та її тестування.

Третій розділ присвячено опису методики розробки головних концепцій казуальної ігрової системи «Zima» для ОС Android за допомогою мови програмування C# і платформи Unity.

У додатках представлено фрагменти лістингу коду та сертифікати апробації ігрової розробки.

## **РОЗДІЛ 1 АНАЛІЗ ПРОГРАМНИХ І ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ РОЗРОБКИ ІГРОВИХ СИСТЕМ**

На сьогоднішній день ігрова індустрія - це один з найзнаменитіших сегментів цифрового контенту в світі [34]. Вона постійно розвивається і росте, так, за підсумками 2019 року монетизація становить близько 170 мільярдів доларів.

Сьогодні багато провідних ІТ-компаній активно займаються розробкою мобільних ігор, оскільки попит на них залишається стабільно високим. Практично кожен другий розробник зосередився на створенні ігрових додатків. Це захоплення стало настільки популярним, що навіть соціальні платформи, такі як Facebook, Qzone, Twitter і Однокласники, почали розробляти інноваційні мобільні ігри, вбачаючи в них ефективний спосіб залучення користувачів.

Сучасні мобільні телефони за своїми можливостями прирівнюються до міні-комп'ютерів, надаючи користувачам широкі можливості для онлайн-ігор. Для цього достатньо мати екран із високою роздільною здатністю та якісною передачею кольору, високошвидкісний інтернет, вбудований браузер, сенсорний дисплей та доступ до магазинів ігрових додатків.

Конкуренція на глобальному ринку мобільних ігор надзвичайно жорстка, і компанії змушені впроваджувати інноваційні технології та інструменти, щоб забезпечити користувачам максимально комфортну платформу. Перед розробниками стоїть складне завдання – створити унікальні, сучасні рішення, які будуть цікавими гравцям різних вікових категорій.

Щороку ринок мобільних ігор поповнюється новими технологіями, функціями, інструментами та трендами. Головна мета кожної компанії, що спеціалізується на розробці ігрових додатків, – створити передову платформу, яка повністю задовольнить потреби користувачів.

### 1.1. Аналіз мов програмування для створення ігор

Перші програми полягали в установці ключових перемикачів на передній панелі обчислювального пристрою. Очевидно, у такий спосіб можна було скласти тільки невеликі програми. З розвитком комп'ютерної техніки з'явилася машинна мова, за допомогою якої програміст міг задавати команди, оперуючи з осередками пам'яті, повністю використовуючи можливості машини. Однак використання більшості комп'ютерів на рівні машинної мови важко, особливо це стосується введення-виведення. Тому від її використання довелося відмовитися.

Наступний крок був зроблений в 1954 році, коли було створено першу мову високого рівня - Фортран (англ. FORTRAN - FORMula TRANslator). Мови високого рівня імітують природні мови, використовуючи деякі слова розмовної мови і загальноприйняті математичні символи. Ці мови більш зручні для людини, за допомогою них, можна писати програми до декількох тисяч рядків довжиною. Однак легко зрозуміла в коротких програмах, ця мова ставала нечитабельною і важко керованою, коли справа стосувалася великих програм. Вирішення цієї проблеми прийшло після винаходу мов структурного програмування (англ. Structured programming language), таких як Алгол (1958), Паскаль (1970), Сі (1972).

У підсумку в кінці 1970-х і початку 1980-х були розроблені принципи об'єктно-орієнтованого програмування (ООП). ООП поєднує кращі принципи структурного програмування з новими потужними концепціями, базові з яких називаються інкапсуляцією, поліморфізмом і спадкуванням. Прикладом об'єктно-орієнтованих мов є: Object Pascal, C++, Java та ін. ООП дозволяє оптимально організовувати програми, розбиваючи проблему на складові частини, і працюючи з кожною окремо. Програма на об'єктно-орієнтованій мові, вирішуючи деяку задачу, по суті, описує частину світу, що відноситься до цього завдання.

На сьогоднішній день виділяють наступні мови програмування як основи для розробки [28]:

C# входить в першу п'ятірку найпопулярніших мов програмування на 2020 рік. Він використовується в багатьох великих компаніях, а також і в невеликих групах програмістів. Зараз компанія Microsoft робить великий акцент на розвиток універсальності і кросплатформеності для цієї мови. Вже зараз з її допомогою можна розробляти практично будь-який тип додатків.

Компанія Microsoft залишається однією з найбільших ІТ компаній світу, а C# її флагманською мовою програмування, яка постійно розвивається і вбирає в себе все нові можливості.

C# дозволяє стартувати розробку швидше, а це дозволяє швидше отримати прототип рішення. Швидкість розробки на початкових етапах проекту значно вище в порівнянні з іншими мовами.

C# спроектований кросплатформним, однак його розвиток не пішов в цьому напрямку, тому під Windows утворилася досить повна .net інфраструктура, а на інших платформах рівноцінної інфраструктури не з'явилося.

При розробці невеликих проектів продуктивність C # не поступається іншим мовам програмування, однак при збільшенні вихідного коду, алгоритмів і т.д. - швидкість роботи додатків значно падає.

C# володіє великою кількістю бібліотек зі старту, що істотно полегшує розробку.

*HTML і JavaScript.* HTML5-гру можна розробити з нуля, але зручніше і простіше це буде зробити за допомогою численних движків і фреймворків. Ось лише деякі з них:

- Construct 2 – фреймворк для розробки 2D-ігор, він призначений не тільки для професійних розробників, але і для людей, які не вміють програмувати: дизайнерів, художників або студентів. У комплекті є більше 20 плагінів і 70 візуальних ефектів для створення гри, яку потім можна відразу ж опублікувати на декількох платформах. Construct 2 можна спробувати безкоштовно або придбати повну версію інструменту;

- PhaserJS – безкоштовний движок з відкритим вихідним кодом

дозволяє створювати ігри із застосуванням Canvas і бібліотеки WebGL. На сайті Phaser є безліч прикладів і навчальних статей, які допоможуть у створенні власної гри;

- ImpactJS – почав свій шлях зі звання «Перший фреймворк для веб-ігор». Цей опенсорсний претендент поширюється безкоштовно і поставляється з хорошим редактором рівнів. Фреймворк не є самим зрозумілим або документованим, але його надійність вже доведена. Наприклад, розробникам з CrossCode взяли за основу ImpactJS для свого движка за його продуктивність і здатність масштабуватися під конкретну задачу;

- CreateJS – набір open source-бібліотек для розробки ігор: EaselJS призначена для роботи з HTML5 Canvas, SoundJS - для роботи з аудіо, TweenJS - для створення анімацій, а PreloadJS - для управління завантаженням всіх необхідних елементів;

- ThreeJS – движок для створення 3D-ігор з використанням WebGL. Вихідний код проекту відкритий і постійно оновлюється;

- PlayCanvas – движок, що дозволяє створювати 2D і 3D-ігри і розміщувати в цих іграх рекламу для отримання прибутку. PlayCanvas безкоштовний для публічних проектів, можна придбати і одну з двох платних версій з розширеними функціями;

- PixiJS – основною перевагою PixiJS є швидкість рендерингу. Движок повністю безкоштовний, він призначений для створення 2D-додатків: багато прикладів є на сайті проекту;

- ExcaliburJS – повноцінний ігровий фреймворк, написаний на Typescript. Повна система сцен і камер, спрайти і анімації, звуки, фізика і т.д. Багатьом дуже подобається API, що надається ExcaliburJS.

HTML і JavaScript незамінні при створення простих веб-додатків та ігор, проте на цьому гідності даних засобів закінчуються[51].

C++. Є основною мовою програмування при розробці ігор, так як дозволяє здійснити повний контроль над засобами і логікою [43].

C++ - надзвичайно потужна мова, що містить засоби створення ефективних програм практично будь-якого призначення, від низькорівневих утиліт і драйверів до складних програмних комплексів самого різного призначення.

Висока сумісність з мовою C дозволяє використовувати весь існуючий C-код (код C може бути з мінімальними переробками скомпільовано компілятором C++; бібліотеки, написані на C, зазвичай можуть бути викликані з C++ безпосередньо без будь-яких додаткових витрат, в тому числі і на рівні функцій зворотного виклику, дозволяючи бібліотекам, написаним на C, викликати код, написаний на C++).

Мовою C++ підтримуються різні стилі і технології програмування, включаючи традиційне директивне програмування, ООП, узагальнене програмування, метапрограмування (шаблони, макроси) та є можливість роботи на низькому рівні з пам'яттю, адресами, портами.

Дана мова програмування має можливість створення узагальнених контейнерів і алгоритмів для різних типів даних, їх спеціалізація і обчислення на етапі компіляції, використовуючи шаблони.

Кросплатформеність C++ дуже висока. Доступні компілятори для великої кількості платформ, на мові C++ розробляють програми для найрізноманітніших платформ і систем.

Мова спроектована так, щоб дати програмісту максимальний контроль над усіма аспектами структури і порядку виконання програми.

C++ є абсолютним рекордсменом по продуктивності, по підтримці платформ, по сумісності і за кількістю існуючих бібліотек серед інших мов програмування. Однак багато програмістів виділяють головний його недолік - складність освоєння даної мови.

Отримані дані зведено в таблицю 1.1. Критерії оцінюються за п'яти бальною шкалою, де п'ять – найкращий показник.

**Таблиця 1.1 – Порівняння мов програмування**

Критерії	C#	JavaScript/HTML	C++
----------	----	-----------------	-----

Продуктивність	3/5	2/5	5/5
Кількість бібліотек	3/5	2/5	5/5
Зручність програмування	4/5	3/5	3/5
Простота засвоєння	4/5	4/5	3/5
Кросплатформеність	3/5	5/5	4/5

Кожна мова програмування добре себе показує в певних завданнях. Наприклад: веб-розробка (JavaScript) не має прихильності до операційної системи; якщо в пріоритеті швидкість розробки (отримання прототипу додатку), то слід використовувати C#. В якості мови програмування для створення ігор найкраще підходить використання мови C++, так як вона підтримує величезну кількість бібліотек і має гарну продуктивність.

## 1.2. Аналіз існуючих програмних засобів для створення ігор

Термін «ігровий рушій» (або «движок», «двигун») виник у середині 1990-х років у контексті комп'ютерних ігор жанру шутер, таких як популярний Doom. Архітектура Doom була побудована так, що центральні компоненти гри (наприклад, тривимірна графіка, фізика, звук та інші підсистеми) були відокремлені від графічних ресурсів, ігрових правил та інших елементів. Такий підхід дозволяв з мінімальними зусиллями створювати нові ігри, оскільки більшість завдань, спільних для багатьох ігор, вирішувалися ігровим рушієм.

Перші ігрові рушії зазвичай розроблялися для запуску однієї гри на певній платформі. Навіть універсальні багатоплатформні рушії зазвичай орієнтовані на створення ігор певного жанру, наприклад шутерів або RPG. Це пов'язано з тим, що програмне забезпечення завжди є набором компромісів, заснованих на припущеннях про особливості майбутньої гри. Наприклад, рушій, оптимізований для роботи у закритих приміщеннях, використовує BSP-дерева для рендерингу об'єктів, розташованих поблизу камери. Натомість рушії, орієнтовані на відкриті простори, застосовують інші технології,

наприклад відтворення з різним рівнем деталізації залежно від відстані до об'єкта.

Окрім багаторазово використовуваних програмних компонентів, ігрові рушії зазвичай включають набір візуальних інструментів для розробки, які складають інтегроване середовище розробки. Це спрощує, прискорює та полегшує створення ігор. Такі рушії іноді називають «ігровим підпрограмним забезпеченням», оскільки вони надають гнучку та багаторазово використовувану платформу з усіма необхідними функціями для створення ігрового продукту, знижуючи витрати, складність і час розробки, що є критично важливими факторами в умовах високої конкуренції на ринку відеоігор.

Спочатку ігрові рушії створювалися компаніями для розробки власних ігор. Згодом з'явилися компанії, які спеціалізуються виключно на створенні рушіїв для інших розробників. За останні роки кількість таких рушіїв перевищила сотню. Багато з них стали безкоштовними або частково безкоштовними, що дозволяє будь-кому спробувати себе у створенні власної гри, ознайомившись із принципами роботи рушія.

Серед найпопулярніших ігрових рушіїв можна виділити Unity, Unreal Engine 4 та Godot, кожен із яких має свої переваги та недоліки.

### *1.2.1. Unity*

Unity — це інтегроване середовище для розробки комп'ютерних ігор, в якому використовуються різні засоби створення програмного забезпечення, такі як текстовий редактор, компілятор, відладчик тощо. Завдяки зручності використання, Unity спрощує процес розробки ігор, роблячи його комфортним, а мультиплатформенність рушія дозволяє охоплювати широкий спектр платформ.

Ігрові рушії зазвичай мають багато функціональних можливостей, таких як моделювання фізичних середовищ, динамічні тіні, карти нормалей тощо. Unity вирізняється серед інших двома ключовими перевагами: візуальним



середовищем розробки та кросплатформенністю. Візуальне середовище включає інструменти моделювання, інтегровану розробку та автоматичну збірку, що підвищує ефективність роботи розробників. Кросплатформенність охоплює не лише підтримку різних платформ (ПК, мобільні пристрої, консолі тощо), але й можливість використовувати середовище розробки під операційними системами Windows і MacOS.

Третьою перевагою Unity є модульна система компонентів, яка дозволяє створювати ігрові об'єкти шляхом комбінування функціональних елементів. На відміну від традиційного наслідування, у Unity об'єкти формуються шляхом додавання компонентів, що значно спрощує створення прототипів і прискорює розробку.

Проте Unity має й недоліки. Одним із них є обмеження візуального редактора при роботі з багатоконпонентними схемами. Другим недоліком є відсутність підтримки посилань на зовнішні бібліотеки, через що програмістам доводиться налаштовувати їх самостійно, що ускладнює командну роботу. Ще одна проблема стосується шаблонів екземплярів (prefabs). Хоча цей підхід забезпечує гнучкість у візуальному редагуванні об'єктів, редагування самих шаблонів має певні недоліки.

Також існує WebGL-версія Unity, яка через специфіку своєї архітектури (трансляція коду з C# у C++, а потім у JavaScript) має проблеми з продуктивністю, споживанням пам'яті та роботою на мобільних пристроях.

### *1.2.2. Unreal Engine*

Unreal Engine (UE), розроблений компанією Epic Games, є одним із найпопулярніших ігрових рушіїв, широко використовуваних для створення фільмів та AAA-проектів. Цей рушій вирізняється потужними графічними можливостями та, як і Unity, підтримує розробку ігор для більшості операційних систем, консолей і мобільних платформ. Завдяки сумісності з різними графічними драйверами, такими як Direct3D та OpenGL, підтримці

аудіосистем і функцій для мережевих ігор, Unreal Engine підходить для створення ігор будь-якого жанру. До складу рушія входять SDK та інтуїтивно зрозумілий редактор, які значно спрощують процес розробки. Робочий інтерфейс ігрового «движку» Unreal Engine представлено на рис. 1.1.

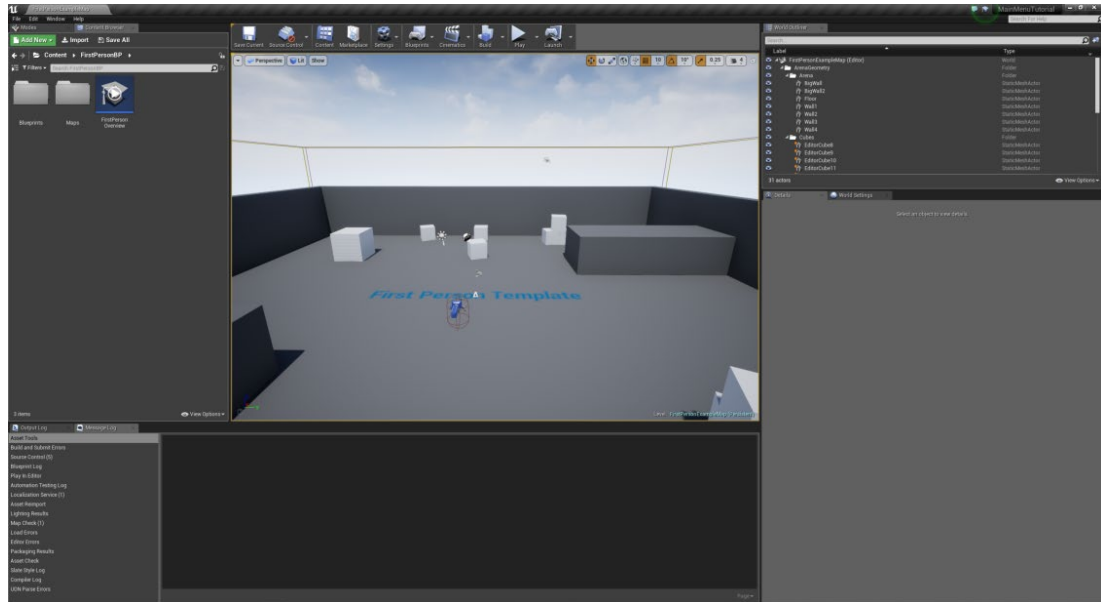


Рис. 1.1. Інтерфейс ігрового движка Unreal Engine

На відміну від Unity, Unreal Engine має відкритий вихідний код, написаний на мові програмування C++. Якщо порівнювати ці два рушії, варто зазначити, що Unity краще підходить для створення мобільних ігор і проєктів у 2D, тоді як Unreal Engine дозволяє створювати високотехнологічну графіку.

Однією з ключових особливостей Unreal Engine є система візуальних скриптів під назвою Blueprints. Вона дозволяє створювати скрипти без використання мов програмування, що зручно для реалізації різноманітних ігрових механік, наприклад, логіки персонажів чи процедурної генерації. Завдяки цьому UE також використовується для створення реалістичної графіки в архітектурній візуалізації. [22].

Недоліки Unreal Engine:

- Займає багато дискового простору як сама програма, так і створені на ній ігри.

- Високий поріг входу, хоч візуальне програмування Blueprints частково його знижує.
- Громіздкий інтерфейс із великою кількістю елементів.
- Проблеми з продуктивністю.

Unreal Engine здатен забезпечити складнішу та якіснішу графіку порівняно з Unity, проте вимагає більшого рівня підготовки від розробника. У простих проєктах різниця між рушіями майже не відчутна, і розробка таких ігор у Unity зазвичай займає менше часу.

### *1.2.3. Godot*

Годо (англ. Godot) — відкритий багатоплатформовий 2D та 3D гральний рушій під ліцензією MIT розробляється у співавторстві з Godot Engine Community з'явився в 2014 році, стабільна версія 2.0 вийшла в 2016. У 2018 розробники додали підтримку 3d і движок отримав назву версії 3.0. До публічного релізу у вигляді відкритого ПЗ рушій використовувався всередині деяких компаній Латинської Америки [19]. Оточення розробника працює на Windows, Linux, OS X, BSD і Haiku та може експортувати ігрові проєкти на ПК, консолі, мобільні та веб платформи [11]. Основна мета Godot — забезпечити максимально просте й самодостатнє середовище для розробки ігор. Цей рушій дозволяє створювати ігри з нуля, не потребуючи додаткових інструментів, окрім тих, які потрібні для створення ігрового контенту (наприклад, графічних елементів чи музики). Написання коду також здійснюється без використання стороннього програмного забезпечення[10].

Архітектура Godot базується на концепції дерева спадкових «сцен». Кожен елемент сцени (нода) може стати окремою самостійною сценою. Це дозволяє розробникам змінювати всю архітектуру проєкту та працювати з комплексними сценами, використовуючи прості абстракції.

Робочий інтерфейс ігрового «рушія» Godot представлено на рис. 1.2.

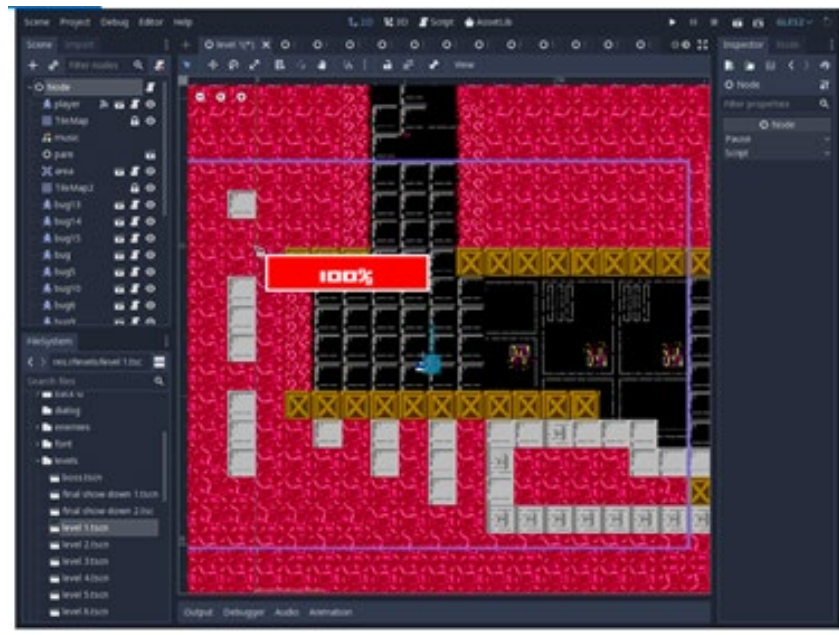


Рис. 1.2. Інтерфейс ігрового рушія Godot

Усі ігрові ресурси, включаючи скрипти, графічні асети та ігрові сцени, зберігаються в папці проєкту у вигляді бінарних файлів, а не складної бази даних. Ресурси, що не містять складних даних, зберігаються у простих текстових форматах. Такий підхід значно полегшує роботу команд, які використовують системи контролю версій. [10].

Для того, щоб досягти потрібного рівня оптимізації під рушій та інтеграції в середовище розробки, Godot використовує власну скриптову мову. Гра створюється або повністю на C++, або з використанням власної високорівневої динамічно типізованої скриптової мови програмування під назвою GDScript, синтаксис якої нагадує мову Python. Відмінністю від Python є в першу чергу чітка типізація змінних при оголошенні та загальна оптимізація скриптової системи під базовану на сценах архітектуру рушія [9]. Також існує версія Godot з підтримкою мови C # (Godot Engine Mono Version), проте в даний момент вона все ще допрацьовується і нестабільна.

Графічна система для всіх підтримуваних платформ побудована на OpenGL ES 2.0. Рендеринг включає в себе технології order-independent transparency, normal mapping, specular, повноекранні постефекти типу FXAA, bloom, DOF, HDR, гама-корекції, distance fog, динамічні тіні на основі

shadow maps та інші.

Для створення шейдерів використовується спрощена шейдерна мова, що є близької підмножиною мови GLSL. Потім шейдер можна використовувати в матеріалі, екранному ефекті для 2д-візуалізації. Шейдер поділяється на секції vertex та fragment. Також можливе повноцінне створення шейдерів візуальному редакторі.

Godot має окрему графічну підсистему для 2D, яку можна використовувати незалежно від 3D.

Завдяки своїй відкритості Godot багатомовний, він підтримує майже всі мови світу, має приємний інтерфейс і зручний інструмент для написання коду. Опенсорсний движок також має і недоліки, місцями сируватий і має проблеми з продуктивністю якщо на екрані знаходиться велика кількість об'єктів.

Недоліки:

- деякі речі треба шукати, доустановлювати або навіть самим програмувати;
- просідання за продуктивністю з великою кількістю об'єктів;
- у порівнянні з великими движками мало туторіалів, баз знань і можливості вирішити проблему. А через зміну версій, ті що є часто застарівають.

**Таблиця 1.2 – Порівняння програмних засобів**

Критерії	Unity	Unreal Engine	Godot
Мова написання гри	C# /js/ Python	Unreal Script	C++, GDScript C#
Доступність	Умовно безкоштовна	Умовно безкоштовна	Безкоштовна
2D-графіка	+	+	+
3D-графіка	+	+	+
Обробка колізії	+	+	+
Обробка фізики	+	+	+
Вбудовані механізми відладки	+	+	+

Отриманий аналіз порівняння програмних засобів зведено в

таблицю 1.2.

Серед конструкторів, досить привабливо виглядає Godot, що номінально обіцяє широкий спектр функціоналу та кросплатформенність. До того ж знову таки, має чималу популярність та підтримку товариства користувачів, незважаючи на відносну молодість проекту. Єдиним мінусом є нестабільність роботи Godot з C#. Тому для реалізації магістерського дослідження обираємо Unity.

### 1.3. Реалізація штучного інтелекту в іграх

Ігровий штучний інтелект (AI, від англ. *artificial intelligence*) — це набір програмних методів і шаблонів, що використовуються в комп'ютерних іграх для створення ілюзії інтелекту, яким керує комп'ютер. Ігровий AI базується на алгоритмах із теорії управління, робототехніки, комп'ютерної графіки та інформатики загалом.

Реалізація AI суттєво впливає на ігровий процес, системні вимоги та бюджет проекту, тому розробники прагнуть створювати оптимізований AI, який не перевантажує ресурси. На відміну від AI в інших галузях, ігровий AI часто використовує спрощення, імітації та хитрощі. Наприклад, у шутерах надто точний рух і миттєве прицілювання ботів можуть зробити гру несправедливою, тому їхні можливості спеціально обмежують. Водночас боти повинні виконувати складні дії, наприклад, організовувати засідки чи діяти в команді, для чого застосовуються контрольні точки, розташовані на рівні.

Концепція AI з'явилася ще в 1956 році, хоча спершу не мала прямого зв'язку з поняттям інтелекту. Перший бот у відеогрі з'явився ще раніше — у 1951 році. Його створили для гри в шахи, і він складався зі списку шахових алгоритмів. У відеоіграх 60-х і 70-х років AI зазвичай не використовувався, адже більшість із них були розраховані на двох гравців [33].

Сьогодні евристичні алгоритми ігрового AI знаходять застосування в різних аспектах ігрового процесу. Найпоширеніше використання —

управління неігровими персонажами (NPC). Ще однією важливою сферою є пошук шляху, що широко застосовується в стратегіях реального часу. Цей алгоритм визначає, як NPC може перейти з однієї точки до іншої, враховуючи ландшафт, перешкоди та інші фактори.

Концепція непередбачуваного (англ. *emergent*) AI була реалізована в деяких іграх нещодавно. У таких іграх ігрові боти здатні "вчитися" на основі дій гравця, змінюючи свою поведінку відповідно до цього. Хоча ці рішення зазвичай обмежуються певною кількістю варіантів, це часто створює враження інтелекту в грі. Розробники часто намагаються створити AI, який моделював би роботу людського мозку, але такі приклади є рідкісними, оскільки ця задача набагато складніша, ніж створення непередбачуваних для гравця дій.

У 1992 році британський вчений Стів Гранд вирішив спробувати розробити комерційне програмне забезпечення і запропонував ідею віртуального домашнього улюбленця, який жив би на робочому столі Windows. Гра мала навчатися новим трюкам через внутрішню нейромережу. Згодом ця ідея перетворилася на повноцінну гру про вигаданих істот, норнів. Пухнасте створіння вилуплюється з яйця, і гравець повинен допомогти йому вивчити світ: навчати словами, повторюючи їх і показуючи значення, а також змушувати виконувати різні дії, заохочуючи гарну поведінку. Проте норни часто забувають свій досвід і роблять помилки — це було значним досягненням розробника.

Штучний інтелект кожної істоти побудований на нейромережі з тисячі нейронів, поділених на кластери. Кожен кластер відповідає за одну з функцій: почуття, фокусування уваги, пам'ять і прийняття рішень. Норни асоціюють свої дії з позитивними або негативними наслідками та роблять узагальнення на основі попереднього досвіду, що дозволяє їм адаптуватися до нових ситуацій.

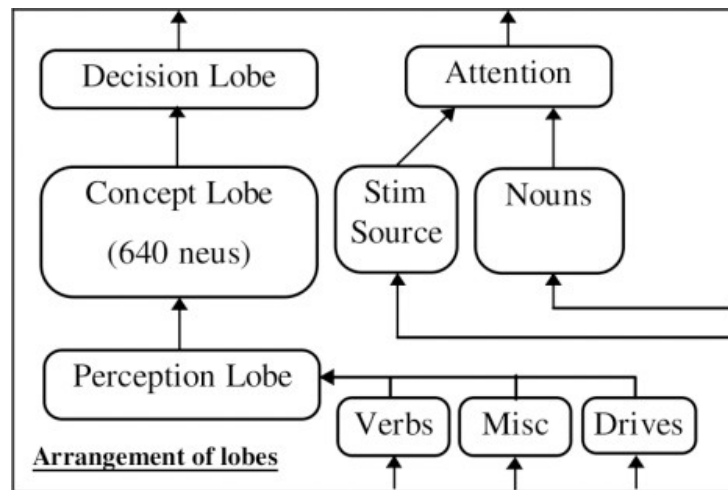


Рис. 1.3. Загальна схема мозку норни

Поведінка норнів регулюється гормонами, подібно до живих істот, тому вони можуть відчувати потребу в спілкуванні та розвагах, а також переживати біль, голод і втому. Ці фактори впливають на баланс хімічних елементів у їхній крові, що можна відстежувати на спеціальній консолі. [36].

Ще одне важливе застосування AI в іграх — це моделювання поведінки гравця, щоб зрозуміти, як гравець взаємодіє з грою. Складний AI має враховувати, що робить гравець і які відчуття він переживає під час гри. Для оцінки ігрового досвіду гравця розробники використовують методи машинного навчання, такі як контрольоване навчання (наприклад, машини опорних векторів або нейронні мережі), щоб створювати моделі досвіду гравця. Тренувальні дані для цих моделей складаються з аспектів гри або взаємодії гравця з грою, а цілі — це оцінки досвіду гравця, зібрані, наприклад, через фізіологічні вимірювання або анкети.

Залучення гравців є складнішою задачею. Розробники зазвичай виділяють чотири основні моделі поведінки гравців, важливі для ігрового AI:

- Створення розумних і людиноподібних NPC для покращення взаємодії з гравцями;
- Прогнозування поведінки гравців для покращення процесу тестування ігри та її дизайну;



- Класифікація поведінки гравців для персоналізації ігрового процесу;
- Виявлення повторюваних шаблонів чи послідовностей дій для визначення поведінки гравця у грі.

Штучний інтелект стає все більш популярним у ігровій індустрії. Особливості ігор роблять їх ідеальним середовищем для застосування методів AI, зокрема глибокого навчання та навчання з підкріпленням [2].

#### **1.4. Висновки до розділу 1**

Провівши аналіз дослідження існуючих програмних і інструментальних засобів розробки ігрових систем, дійшли до висновку, що за останні роки мобільні відеоігри зайняли чималу нішу на ринку розваг та дозвілля. Як наслідок все більше людей починають захоплюватись розробкою відеоігор. В свою чергу з'явився чималий попит на інструментальні засоби що полегшують розробку, покращують графіку, оптимізують фізику тощо.

Для розробки та програмної реалізації ігрової системи, що придатна для гри нон-геймерам та дозволяє змагатися зі штучним інтелектом, було обрано середовище Unity, яке підходить для створення та генерації простих ігор завдяки підтримці мови програмування C#, витрати достатньо меншого часу на реалізацію ігрової логіки, завдяки гнучкій мультиплатформеності рушія.

## РОЗДІЛ 2 ПРОЕКТУВАННЯ ІГРОВОЇ СИСТЕМИ

Розробка відеогри має низку послідовних етапів (рис.2.1), загалом їх п'ять: розробка ігрової концепції, прототипування ігрового боту, розробка ігрових механік, їм передують проектування (пре-продакшену) – генерування геймдизайнером ідеї щодо майбутньої гри, вибір жанру, тематики, особливостей ігрового процесу, розробка сценарію та образів персонажів з оточенням. Менеджер координує дії різних людей, залучених до розробки, складає план їхньої роботи, встановлює терміни її виконання, планує витрати.

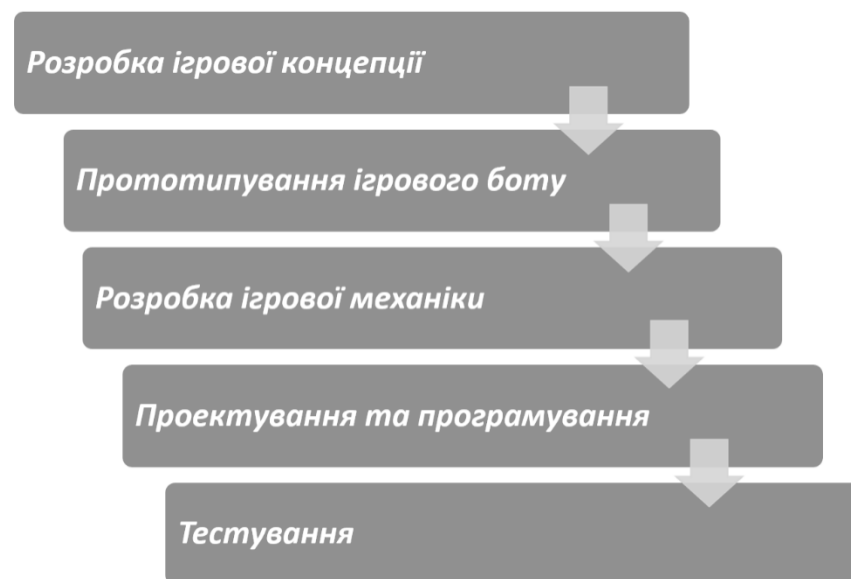


Рис. 2.1. Етапи розробки комп'ютерних ігор

Готова гра в свою чергу має пройти низку етапів, в ході яких потрапляє до гравців і підтримує інтерес до себе. Індустрія відеоігор включає у себе багато людей з різними професіями та ролями: програмістів, які відповідають за технічні можливості гри, художників, моделювальників та аніматорів, які створюють графічний контент, композиторів та звукорежисерів, які створюють звукове оформлення та музичний супровід, який нерідко видається окремим накладом. За успішне завершення роботи над проектом відповідають продюсери. Відеоігри, які розробляються незалежними розробниками чи

аматорами називаються інді-іграми. Такі ігри нерідко створюються за допомогою спеціальних програм, які істотно полегшують (або навіть ліквідують) процес розробки коду або графіки, наприклад, як RPG Maker.

## 2.1. Розробка ігрової концепції

Гра являє собою змагання проти штучного інтелекту. Основний ігровий процес полягає у стрільбі різнокольоровими кулями по таких самих кульках, які рухаються вздовж визначеного маршруту. Змагання відбувається в режимі реального часу: одночасно з вами грає штучний інтелект, маючи такі ж цілі. Переможцем стає той, хто набере найбільшу кількість балів за обмежений час.

Для опису роботи ігрової системи створимо діаграму ігрової логіки, яка відображає ключові взаємодії в межах гри. Уся логіка гри може бути умовно поділена на три основні частини:

- Логіка ігрового меню;
- Логіка геймплею;
- Логіка інтерфейсу гри.

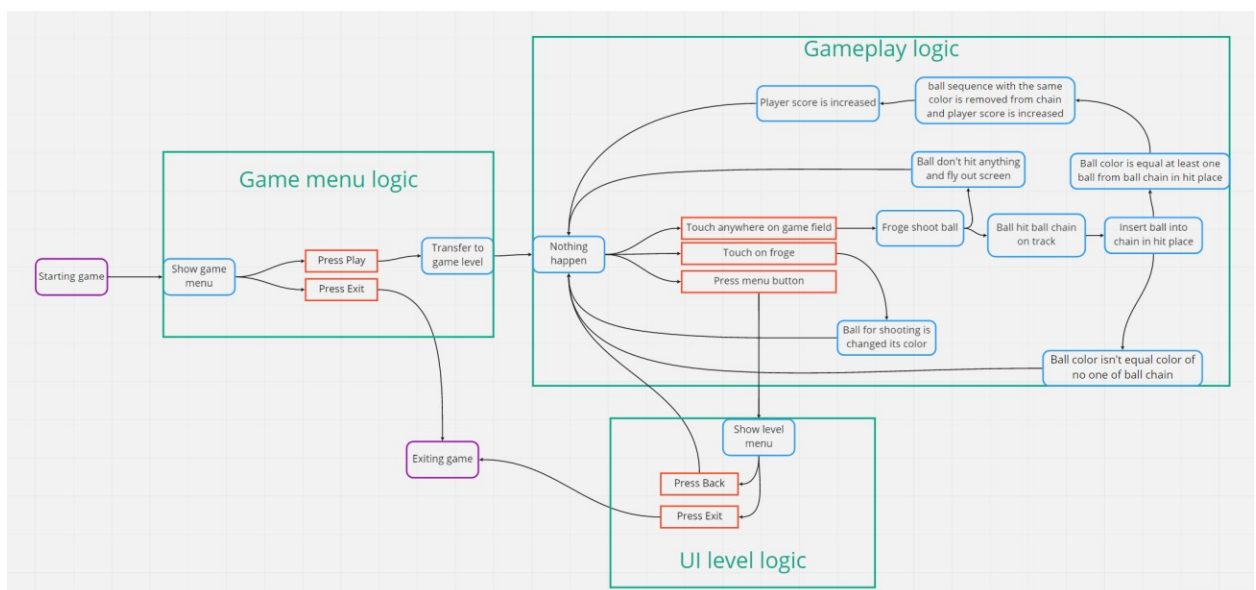


Рис. 2.2. Діаграма логіки гри

На рис. 2.2 зображено діаграму, що описує логіку роботи ігрової системи. Після входу в ігрове меню гравець може розпочати рівень або вийти

з гри. У межах рівня гравець змагається зі штучним інтелектом відповідно до процесу, описаного на діаграмі. У будь-який момент гри можна викликати ігровий інтерфейс, щоб призупинити гру, продовжити її або завершити.

## 2.2. Прототипування ігрового бота

Штучний інтелект (ШІ) є ключовим елементом майже будь-якої одиночної гри. Персонажі, якими керує комп'ютер, створюють атмосферу і допомагають гравцеві відчувати себе частиною ігрового світу. На базовому рівні ШІ емулює поведінку інших гравців або сутностей (усіх елементів гри, які можуть діяти або на які можна впливати, від персонажів до об'єктів, таких як ракети чи датчики здоров'я). Основна концепція ШІ полягає у моделюванні поведінки. По суті, ігровий ШІ більше «штучний», ніж «інтелектуальний».

Ця система може варіюватися від простої, побудованої на правилах, до складної, що імітує поведінку людини або приймає нестандартні рішення [2].

Ось кілька поширених підходів до реалізації ШІ в ігрових системах:

- *Дерево рішень*: система, де рішення організовані у вигляді дерева. Алгоритм проходить дерево, щоб дійти до «листа» — вузла, що містить остаточне рішення. У дереві рішень вузли можуть бути двох типів:
  - *Вузли рішень* — перевіряють умови та обирають одну з альтернатив, кожна з яких веде до свого вузла.
  - *Кінцеві вузли* — дії, що реалізують остаточне рішення дерева.
- *Кінцеві автомати (FSM)*: метод моделювання об'єкта, який протягом життя переходить між різними станами. Кожен стан може описувати фізичний стан об'єкта або набір поведінкових моделей, що відповідають контексту гри.
- *Системи на основі корисності (Utility)*: агент має набір можливих дій і обирає ту, яка має найвищу корисність. Корисність — це міра важливості або доцільності виконання певної дії для агента.

Для реалізації проекту за обраним сценарієм застосуємо складну

систему кінцеві автомати, яка призначена для імітації поведінки людини, для прийняття нестандартних рішень.

### **2.3. Розробка ігрової механіки**

Ігрова механіка визначає насиченість ігрового процесу, правила, за якими грається ігрова система. Основою механіки є ігрові об'єкти, такі як персонажі, об'єкти, з якими вони можуть маніпулювати, декорації. Частиною ігрової механіки є управління, якими чином гравець керує персонажем та ігровим світом.

За взаємодію об'єктів, яка відбувається без контролю гравця, відповідає фізичний рушій. До прикладу, він реалізує закони інерції, гравітацію, поведінку рідин, властивості предметів. Штучний інтелект відповідає за поведінку персонажів, як вони реагуватимуть на дії гравця. Багато подій в грі відбуваються за скриптами. Самі події придумуються сценаристами, а скрипти реалізуються програмістами.

Якщо проаналізувати сучасні гіперказуальні ігрові проєкти, можна виокремити найбільш поширені механіки та шаблони. Варто зазначити, що немає універсального підходу до структурування таких механік, і в різних джерелах зустрічаються різні класифікації. Основні з них виглядають так:

#### **2.3.1. Механіка часу**

Ігри з механікою синхронізації передбачають обмеження часу, що спонукає гравців грати швидше. Складність гри визначається швидкістю руху її компонентів. Використання механіки часу робить ігрові сесії короткими, що відповідає концепції гіперказуальних ігор. Щоб уникнути розчарування через швидкий програш, розробники часто дають гравцям кілька спроб і роблять перезапуск гри максимально простим.

#### **2.3.2. Механіка швидкості**

Ця механіка змушує гравців діяти швидко, реагуючи на ігрові загрози, щоб уникнути програшу. Класичним прикладом є *Rac-Man*, де гравець

ухиляється від ворогів. Інший приклад — гра *Змійка*, у якій складність зростає зі збільшенням довжини змії після збору яблук.

### 2.3.3. *Механіка головоломки*

Жанр головоломок охоплює близько 60% мобільного ігрового ринку. Ці ігри розраховані на прості повсякденні розваги та пропонують легкі розумові завдання. Наприклад, *Temple* є класичним прикладом головоломки, яка створює ілюзію вирішення задачі чи складання пазла. Гіперказуальні головоломки зазвичай прості, але дають гравцям відчуття досягнення завдяки швидкому прогресу та виконанню великої кількості завдань за короткий час.

### 2.3.4. *Механіка злиття*

У таких іграх гравці комбінують або замінюють елементи, щоб знищити якомога більше об'єктів. Наприклад, ігри, де потрібно поєднувати три або більше елементів одного типу, щоб видалити їх. Це проста, але захоплива механіка. Додатковий інтерес викликають ланцюгові реакції, коли усунення однієї лінії спричиняє ще більше видалень, створюючи ефект задоволення.

### 2.3.5. *Механіка повороту*

Ігри з цією механікою пропонують гравцям рухатися по заданому шляху, уникаючи перешкод. Для цього використовується механіка повороту. Розробники створюють ілюзію безкінечного руху, хоча насправді гравець обирає лише з кількох варіантів руху [16].

Усі зазначені механіки спрямовані на розвагу користувачів і урізноманітнення ігрового процесу. Такі ігри орієнтовані насамперед на простоту та доступність геймплея, уникаючи складних механік і надмірної уваги до графічної досконалості, характерних для більш складних ігор. У сучасних умовах основна мета будь-якої гри — забезпечити задоволення і приємний досвід для гравця.

## 2.4 Програмування та проектування ігрового середовища

### 2.4.1. Використання фреймворків

Для розробки ігрового середовища будемо використовувати фреймворк Entitas на базі Unity. Його імпортуємо в Unity як плагін. Цей фреймворк дозволяє реалізовувати кодову логіку на базі архітектурного шаблону ECS.

ECS (Entity Component System) – архітектурний шаблон, суть якого полягає в розділенні логіки та даних. В процесі розробки, ми оперуємо трьома складовими:

**Entity** – загальний об’єкт, що відповідає за зберігання даних. По суті, являється аналогом класу з ООП, але якби клас лише зберігав дані всередині себе.

**Component** – представляє собою атомарну частину даних, які також мають унікальне призначення. Наприклад, рівень здоров’я гравця (данні можуть бути представлені `int` змінною, але те що це здоров’я гравця гарантує, що цей компонент буде прикріплений до сутності гравця)

**System** – це невелика частина логіки додатку, що обробляє деякі сутності гри через зміну значень їх компонентів. Системи можуть бути різних видів, наприклад, система, що працює кожен кадр гри, або лише один раз на початку гри, або система спрацьовує на зміну деякого компонента.

**Entitas** – швидкий та легкий у використанні C# ECS фреймворк. Розроблений спеціально під ігровий двигун Unity. Він має гарну документація та велике ком’юніті. В останній час фреймворк припинили підтримувати та розробляти, тому варто чекати, що через декілька років фреймворк застаріє. Також к цьому ведуть новини розробки власного ECS фреймворка всередині ігрового двигуна від компанії Unity.

### 2.4.2. Побудова траєкторії руху за запланованим маршрутом

Одним із найважливіших і найскладніших аспектів ігрового середовища є створення шляху для руху об’єктів (наприклад, кульок). Для цього в Unity

реалізується спеціальний модуль, який дозволяє проектувати траєкторію безпосередньо в редакторі та імпортувати її в гру.

Шлях будується за допомогою **кривих Без'є третього порядку** (рис. 2.3). Ці криві визначають траєкторію на основі опорних точок. Їх можна уявити як графік переміщення точки з початкової до кінцевої позиції залежно від часу.

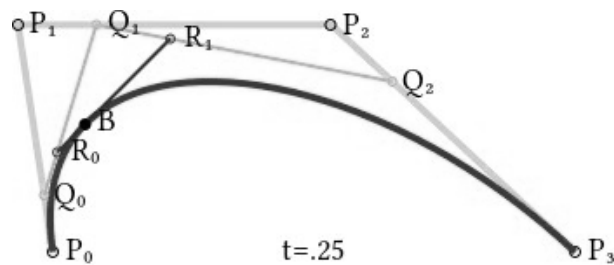


Рис. 2.3. Побудова кривої Без'є третього порядку

Криві Без'є третього порядку формуються за допомогою спеціальних важелів, які виходять із кожної опорної точки кривої. На кінцях цих важелів розташовані точки управління, які, немов магніт, впливають на вигин і плавність переходів кривої. (рис. 2.4).

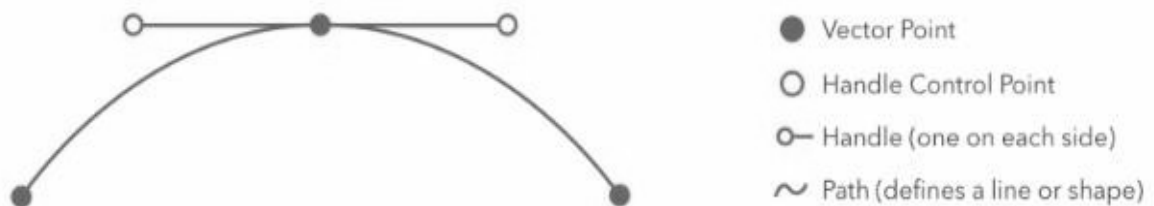


Рис. 2.4. Крива Без'є з важелем Існує

Розглянемо декілька видів керування важелями в кривих Без'є, які будемо використовувати при розробці ігрової системи (рис. 2.5):



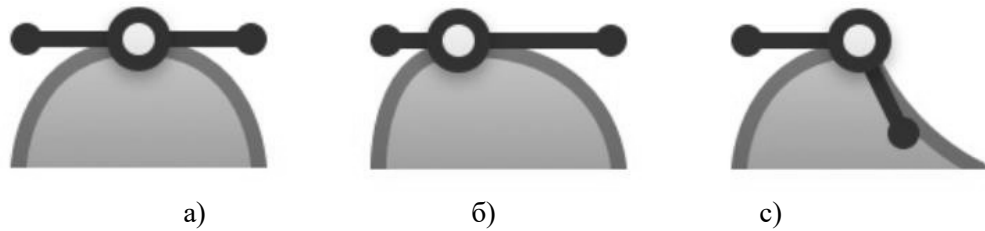


Рис. 2.5. Візуальне представлення способів керування в кривих Без'є

«Mirrored» (дзеркальний метод). У цьому підході використовуються два важелі, які рівновіддалені від опорної точки та розташовані під однаковим кутом (рис. 2.5, а).

«Asymmetric» (асиметричний метод) нагадує дзеркальний, оскільки ручки управління також знаходяться під одним кутом, але довжина важелів, що з'єднують їх з опорною точкою, є різною (рис. 2.5, б).

«Disconnected» (розімкнутий метод) забезпечує незалежне керування кожною ручкою, що дозволяє гнучкіше змінювати форму кривої (рис. 2.5, с).

Криві Без'є широко застосовуються у багатьох сферах ІТ, зокрема у створенні шрифтів, графічних форматів, 3D-графіки, векторної 2D-графіки, а також у CSS для опису плавності анімації. Ми ж знайшли для них нове й досить специфічне застосування у нашому проекті.

На рис. 2.6 продемонстровано побудову шляху для гри «Zima» за допомогою кривих Без'є.

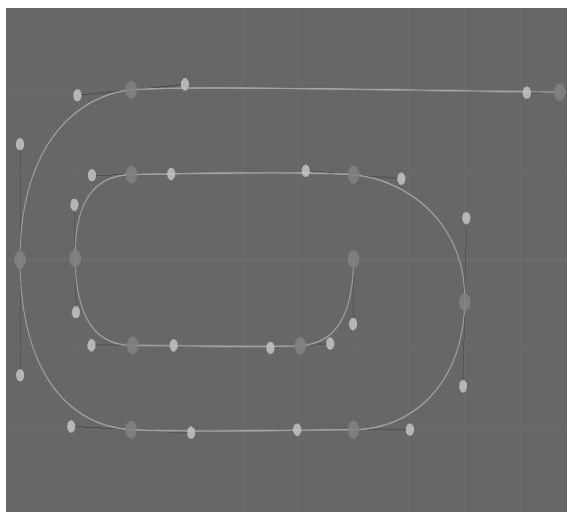


Рис. 2.6. Шлях для гри «Zima», побудований за допомогою кривих Без'є

### 2.4.3. Принципи роботи кінцевих автоматів

FSM (Finite-state machine) - використання такого прийому непогано, коли необхідно орієнтуватися на конкретні події, а модель поведінки повинна залишатися незмінною, поки не зміняться умови.

Механізм FSM головним чином базується на моделях поведінки автоматів і закладає фундамент для наступного методу, який стане вдосконаленою версією поточного.

Цей метод включає послідовну реалізацію трьох класів, сенс використання яких стане ясний на заключному етапі:

#### 1. Клас **Condition**:

```
public class Condition
{
    public virtual bool Test ()
    {
        return false;
    }
}
```

#### 2. Клас **Transition**:

```
public class Transition
{
    public Condition condition;
    public State target;
}
```

#### 3. Та клас **State**:

```
using UnityEngine ;
using System . Collections . Generic;
public class State : MonoBehaviour
{
    public List<Transition> transitions;
}
```

#### 4. Реалізуємо функцію **Awake**:

```
public virtual void Awake ()
{
    transitions = new List<Transition> ( ) ;
    // тут настройка переходів
}
```

#### 5. Визначимо функцію ініціалізації:

```
public virtual void OnEnable ()
{
    // тут виконується ініціалізація стану
}
```

#### 6. Визначимо функцію завершення:

```
public virtual void OnDisable ()
{
    // тут виконується фіналізація стану
}
```

#### 7. Визначимо функцію реалізації моделі поведінки, що відповідає стану:

```
public virtual void Update ()
{
    // Буде реалізована нижче
    // тут визначається модель поведінки
}
```

#### 8. Реалізуємо функцію, що визначає стан для наступного кроку:

```
public void LateUpdate ()
{
    foreach (Transition t in transitions)
    {
        if (t.condition.Test ( ) )
        {
            t.target.enabled = true;
            this.enabled = false;
            return;
        }
    }
}
```

Кожний стан - це сценарій **MonoBehaviour**, який виконує або не виконує перехід до наступного стану (рис 2.7). Використаємо функцію **LateUpdate**, щоб не міняти звичного підходу до розробки моделей поведінки, і реалізуємо в ній перевірку необхідності переходу в інший стан. Важливо відключати кожний стан в об'єкті гри, який відрізняється від початкового.

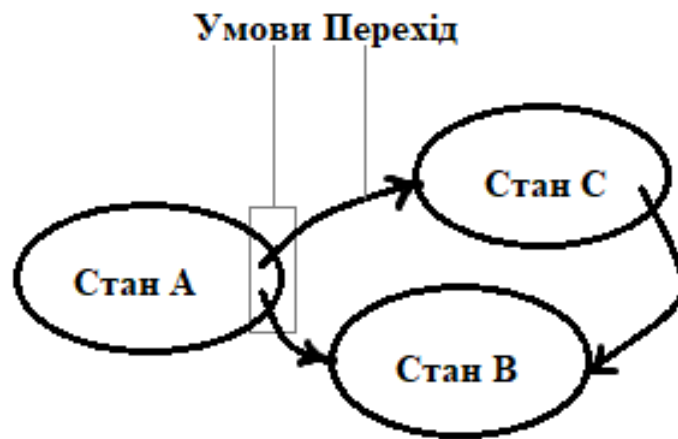


Рис.2.7. Переходи між станами

Для ілюстрації розробки дочірніх класів, які успадковують клас **Condition**, розглянемо кілька прикладів: один перевіряє входження значення в діапазон, а другий перевіряє дотримання двох умов:

Нижче наводиться визначення першого класу - **ConditionFloat**:

```

using UnityEngine ;
using System. Collections;
public class ConditionFloat : Condition
{
    public float valueMin;
    public float valueMax ;
    public float valueTest;
    public override Bool Test ()
    {
        if (valueMax >= valueTest && valueTest >=
valueMin)
            return true ;
        return false;
    }
}

```

та другого класу **ConditionAnd**:

```

using UnityEngine ;
using System . Collections;
public class ConditionAnd : Condition
{
    public Condit ion conditionA;
    public Condition conditionB;
    public override Bool Test ()
    {
        if {conditionA. Test () && conditionB . Test ( )

```

```

    )
        return true ;
    return false ;
    }
}

```

Кінцеві автомати можна вдосконалити, ввівши підтримку шарів або ієрархій. Сам принцип залишається тим же, але стани отримують можливість включати власні кінцеві автомати, що робить їх більш гнучкими і масштабованими.

Створимо стан, спроможний утримувати внутрішні стани, для розробки багаторівневих ієрархічних автоматів:

1. Визначимо клас **StateHighLevel**, що успадковує клас **State**:

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;
public class StateHighLevel : State
{
}

```

2. Додамо властивості для управління внутрішніми станами:

```

public List<State> states;
public State stateinitial;
protected State stateCurrent;

```

3. Перевизначимо функцію ініціалізації:

```

public override void OnEnable ()
{
    if (stateCurrent == null)
        stateCurrent = stateinitial;
    stateCurrent.enabled = true ;
}

```

4. Перевизначимо функцію фіналізації:

```

public override void OnDisable ()
{
    base.OnDisable ( ) ;
    stateCurrent.enabled = false;
    foreach (State s in states)
    {
        s.enabled = false;
    }
}

```

Клас стану верхнього рівня дозволяє активувати внутрішні кінцеві автомати, рекурсивно змінюючи його внутрішній стан. Принцип роботи залишається незмінним завдяки списку станів і підходу батьківського класу до виконання переходів.

#### *2.4.4. Розробка UML-діаграм*

Для проектування програми використовувалася мова UML (Unified Modeling Language), оскільки вона є універсальним інструментом для графічного моделювання. UML — це відкритий стандарт, який використовує графічні позначення для створення абстрактної моделі системи. Це дозволяє спланувати та описати архітектуру проєкту без написання коду, а також робить моделі зрозумілими без спеціальних знань [8].

UML була створена для визначення, візуалізації, проєктування та документування програмних систем. Однак її застосування виходить за межі розробки програмного забезпечення: вона використовується для моделювання бізнес-процесів, проєктування систем і представлення організаційних структур. UML допомагає розробникам узгодити графічні позначення для представлення загальних понять, таких як класи, компоненти, узагальнення, агрегації та поведінка, що дозволяє зосередитися на проєктуванні й архітектурі.

Семантика UML визначає дві основні категорії моделей: структурні (статичні) та поведінкові (динамічні).

- Структурні моделі описують структуру компонентів системи, включаючи класи, інтерфейси, атрибути та зв'язки.
- Поведінкові моделі демонструють функціонування системи, включаючи методи, взаємодії між об'єктами та зміни станів компонентів і системи загалом [4].

Формальна структура UML базується на чотирьох рівнях абстракції:

1. Позначка-метамодель: найвищий рівень абстракції, що визначає UML на базовому рівні.
2. Метамодель: конкретизація позначка-метамоделі, яка визначає мову для опису моделей.
3. Модель: приклад метамоделі, що використовується для опису конкретної системи або предметної області.
4. Об'єкти користувача: конкретні дані чи екземпляри моделі, які представляють обрану предметну область.

На метамодельному рівні UML представлений трьома логічними пакетами:

- Основні елементи
- Елементи поведінки
- Загальні механізми [21].

Концептуальна модель UML включає будівельні блоки, правила їхнього поєднання та загальні механізми [3, 17, 6]. Основні елементи UML — це сутності (абстракції) та відносини між ними, які комбінуються за певними правилами, утворюючи діаграми.

UML є універсальним інструментом для моделювання та аналізу, що дозволяє створювати систематизовані та зрозумілі архітектурні моделі проєктів.

В UML визначено чотири типи сутностей [3]:

– структурні сутності, що поділяються на основні: клас (Class), інтерфейс (Interface); кооперація (Collaboration); прецедент (Use case); активний клас (Active class); компонент (Component); вузол (Node); різновиди основних: актор (Actor), сигнал (Signal), утиліта (Utility, вид класів), процес (Process), нитка (Thread, вид активних класів)) інші; додатка (Application); документ (Document); файл (File), бібліотека (Library); сторінка (Page); таблиця (Table, вид компонентів)).

– сутності поведінки (Behavioral things): взаємодія (Interaction); автомат (State machine).

- сутності, що групують, - пакет (Packages).
- анотаційні сутності – примітка (Note).

Основними типами відносин в UML є: «залежності» (Dependency), «асоціації» (Association), включно з їхнім різновидом — «агрегацією» (Aggregation), «узагальнення» (Generalization) та «реалізація» (Realization). Також існують варіації цих відносин, наприклад, уточнення, трасування, включення та розширення (для залежностей).

Для створення коректно оформленої моделі UML передбачає дотримання визначених правил, які забезпечують точність і однозначність:

- надання імен сутностям, відносинам і діаграмам;
- визначення області дії імен (контексту, у якому ім'я має значення);
- забезпечення видимості імен (для їх використання іншими елементами);
- підтримання цілісності (узгодженості та правильності зв'язків між елементами);
- забезпечення виконання моделі [3].

Ефективність та зручність використання UML досягаються завдяки застосуванню певних угод, так званих загальних механізмів, таких як:

- специфікації (Specifications),
- доповнення (Adornments),
- прийняті розподіли (Common divisions),
- механізми розширення (Extensibility mechanisms) [4, 17, 6].

Кожен елемент нотації UML має унікальне графічне позначення та специфікацію, яка являє собою текстове представлення синтаксису й семантики відповідного будівельного блоку.

Більшість будівельних блоків UML характеризуються дихотоміями:

- «клас / об'єкт»,
- «інтерфейс / реалізація».

Ці дихотомії лежать в основі об'єктно-орієнтованого моделювання систем і забезпечують структурований підхід до відображення реальності.



В якості основного засобу опису проекту було обрано і побудовано діаграми використання, які показують функціональність майбутньої системи. Варіанти використання показують можливі взаємини між системою і користувачем. Вони відображають зовнішній інтерфейс системи і вказують форму того, що система повинна зробити. Дійова особа (актор) - це елемент, який не є системою, а взаємодіє з нею, через особливий інструмент – варіант.

В ході проектування було виділено наступні актори:

- гравець (Дід Мороз червоного кольору) - користувач додатка, що володіє повним доступом до функцій відеогри. Може брати участь в грі в ролі снайпера, управляти налаштуваннями програми.
- другий гравець (Дід Мороз синього кольору) – бот. Може брати участь в грі, володіючи доступом до керування обирати колір кульок.

Загальна діаграма використання представлена на рис. 2.8.

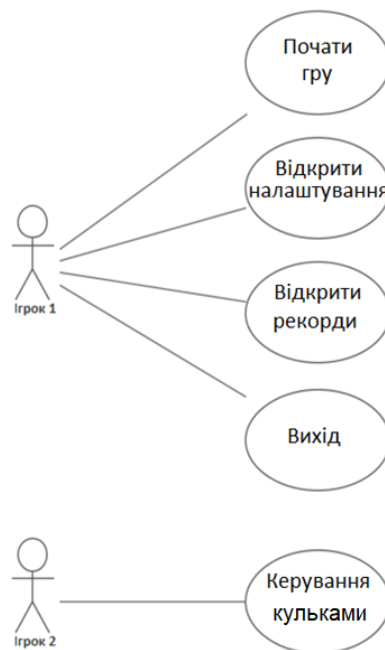


Рис. 2.8. Загальна UML діаграма використання

Розібрано кожен варіант використання детальніше. Для цього представлено діаграми відносин варіантів використання. При запуску рівня користувачем, він може виконати певні дії, як варіант користувач може

використовувати дію «Відкрити параметри». В даному варіанті використання у дійової особи (гравця) є можливість налаштувати візуальні компоненти додатку. При налаштуванні деяких компонентів необов'язково налаштовувати деякі функції, які позначені ставленням «розширити». Діаграма відносин варіанти використання «Відкрити параметри» представлена на рис. 2.9.

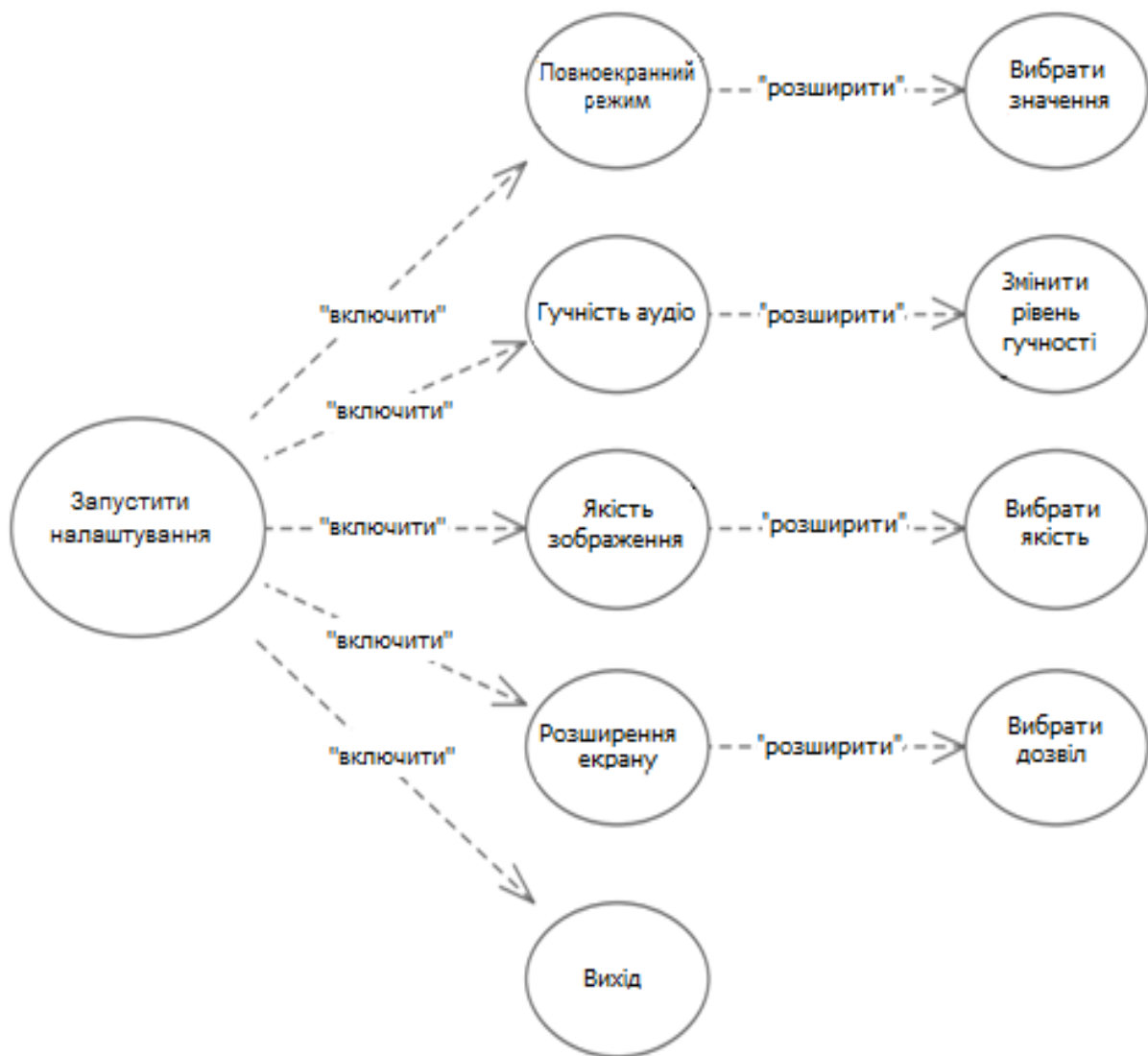


Рис. 2.9. Діаграма відносини для дії «Відкрити параметри»

## 2.5. Тестування

Після завершення роботи над ігровою концепцією, програмною реалізацією алгоритмів штучного інтелекту, механікою, за яких гра може функціонувати, відбувається її доопрацювання. Гра, не зібрана до кінця, але в яку можливо грати, називається альфа-версією. Вона може містити значні

помилки і не доопрацювання, як відсутність певних можливостей, музики або об'єктів. Виявленням проблем займаються тестери, котрі грають в цю гру, намагаючись сповна скористатися всіма доступними можливостями в ній. На пізнішому етапі виходить бета-версія, до тестування якої можуть залучатися і потенційні покупці гри. В бета-версії відбувається подальший пошук помилок, перевірка коректності взаємодії об'єктів ігрового світу, управління. Можливі внесення змін в оформлення, зміна ігрового балансу, тощо.

## **2.6. Висновки до розділу 2**

Проведено аналіз процесу проектування ігрової системи, що орієнтована на нон-геймерів і дає можливість змагатися з штучним інтелектом. Основні етапи: розробка ігрової концепції (створення ігрової логіки та правил), прототипування ігрового бота, імплементація ігрової механіки, програмна реалізація алгоритмів штучного інтелекту, програмна реалізація ігрової системи та її тестування.

Для опису ігрової системи була створена діаграма ігрової логіки, яка зображає основні ігрові взаємодії в системі. Для реалізації проекту за обраним сценарієм використовувалась складна система кінцевих автоматів, що імітує людську поведінку, даючи змогу приймати нестандартні рішення. Правила, на яких базується розроблена ігрова система, а також насиченість ігрового процесу визначають такі ігрові механіки, як механіка часу, спритності, головоломки, злиття та повороту. Для реалізації кодової логіки на основі архітектурного шаблону ECS було обрано фреймворк Entitas, який інтегрується в Unity як плагін. У середовищі Unity реалізовано модуль для створення і імпорту шляху прямо в редакторі. Принцип побудови шляху заснований на використанні кривих Без'є третього порядку.

## РОЗДІЛ 3 РОЗРОБКА ІГРОВОГО ДОДАТКУ «ZIMA»

### 3.1 Вибір та конфігурування програмного середовища

Для розробки даного ігрового середовища був обраний ігровий двигун Unity 3D, через його простоту освоєння, легке налаштування на різноманітні платформи, а саме мобільні, а також за наявну експертизу в роботі з цим програмним продуктом. Для написання програмного коду будемо використовувати Visual Studio 2019, він має гарну інтеграцію з Unity3D.

В нашому випадку, ми маємо головну сцену всередині Unity (для різних проєктів, їх може бути різна кількість, для нашого проєкту достатньо однієї). На нашій сцені розміщенні головні об'єкти ігрового середовища. В Unity усі об'єкти на сцені вважаються як `GameObject` – це основний об'єкт, з яким працює кожен розробник в редакторі, а також в коді продукту; він дозволяє отримати доступ до усього, що зв'язано з даним `GameObject` та навіть більше. `GameObject` може містити в собі компоненти, вони бувають вбудовані в сам ігровий двигун, та власні. Власні компоненти являють собою скрипти, логіку яких задає розробник. В програмному коді, кожен скрипт являє собою клас, що наслідує від класу `MonoBehaviour`. Цей клас надає легкий інтерфейс для впровадження власної логіки, а також багату кількість зворотних викликів на різні випадки.

Для ігрового проєкту будемо використовувати один скрипт, як головний ігровий контролер. Основна ж логіка буде реалізована за допомогою архітектурного шаблону ECS. Але для деяких випадків, коли ECS не може вирішити поставлену проблему (наприклад, фізика) ми будемо використовувати власні скрипти, отримувати їх в нашому основному потоці систем та обробляти.

На рисунку 3.1 представлено зовнішній вигляд редактора Unity3D.

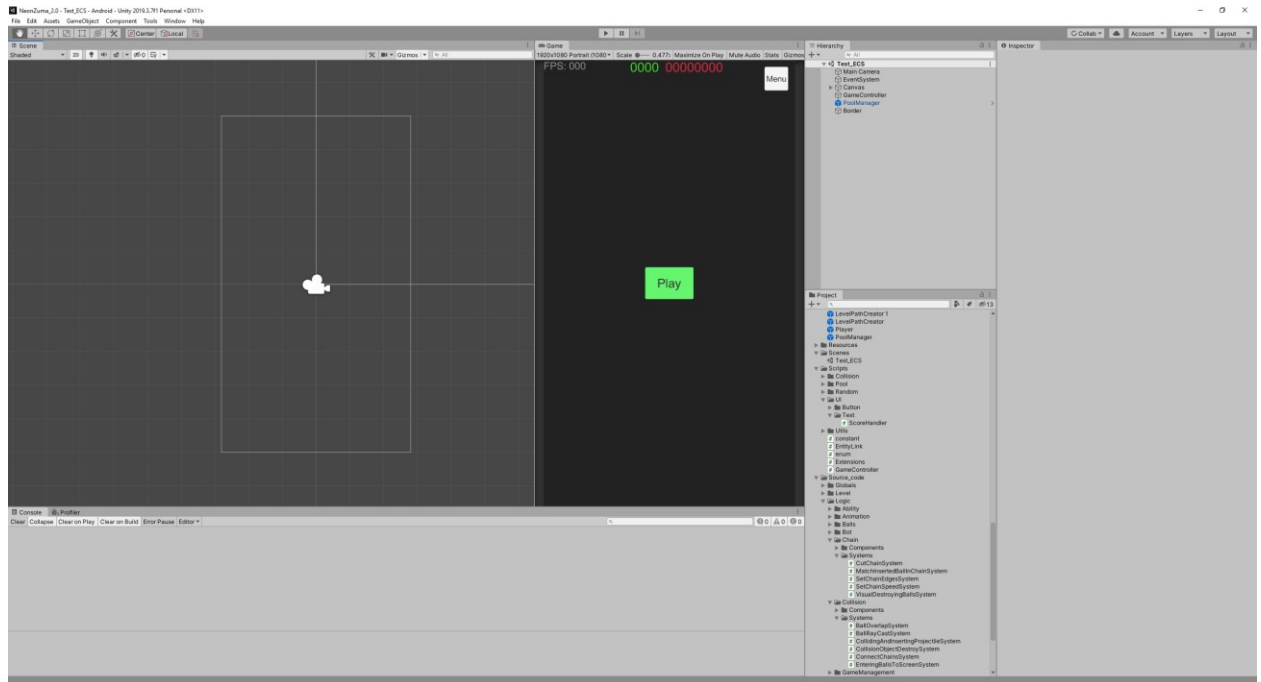


Рис. 3.1. Редактор Unity3D

На рисунку 3.2 представлений зовнішній вигляд середовища розробки Visual Studio 2019, яка була використана для написання коду.

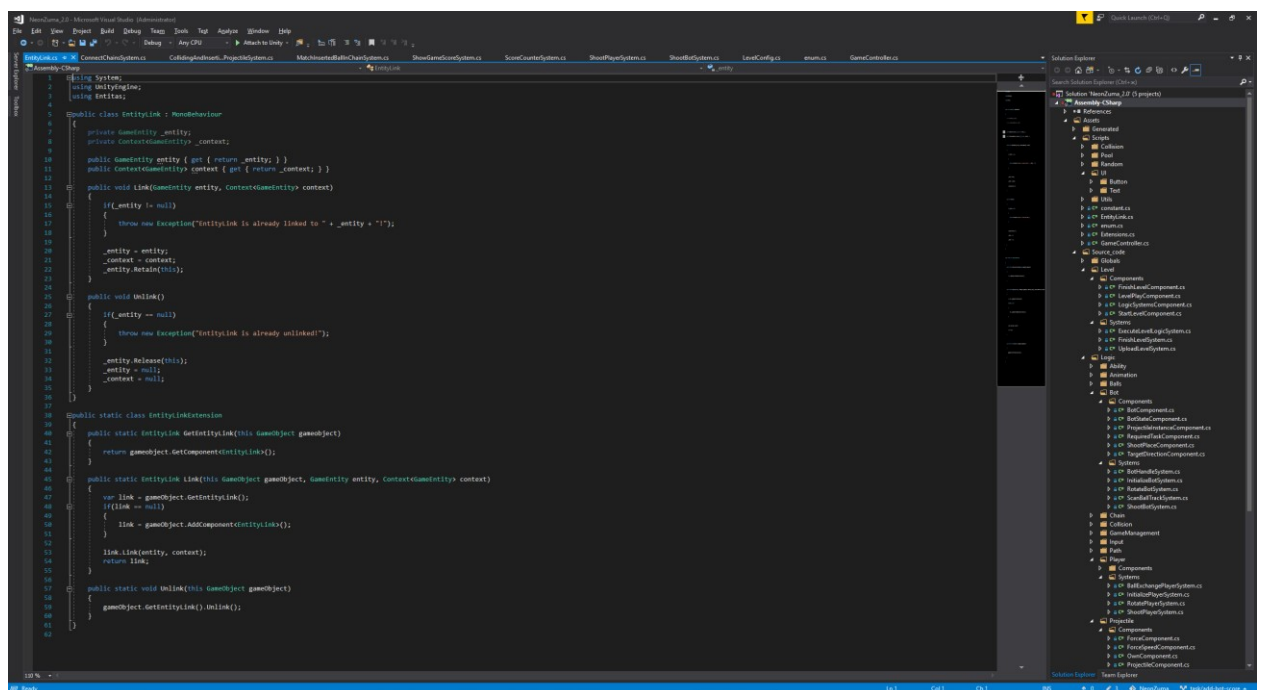


Рис. 3.2. Середовище розробки Visual Studio 2019

## 3.2 Опис використаного фреймворку

### 3.2.1 Концепції ECS

Система компонентів сутності (ECS) є ядром Unity Data-Oriented Tech Stack (DOTS) та орієнтований на дані технологічного стеку Unity3D. Як випливає з назви, ECS складається з трьох основних частин:

Сутності (Entities) - суб'єкти або речі, які заповнюють гру або програму.

Компоненти (Components) - дані, пов'язані з об'єктами гри, але організовані самими даними, а не сутностями. Ця різниця в організації є однією з ключових відмінностей між об'єктно-орієнтованим дизайном та орієнтованим на дані.

Системи (Systems) - логіка, яка перетворює дані компонента з поточного стану в наступний - наприклад, система може оновлювати положення всіх рухомих об'єктів за їх швидкістю, кратним часовому інтервалу з попереднього кадру.

Як було зазначено, архітектура фокусується на даних. Системи зчитують потоки компонентних даних, а потім перетворюють дані з вхідного стану у вихідний стан, які потім індексують.

Наступна схема ілюструє взаємодію цих трьох основних частин:

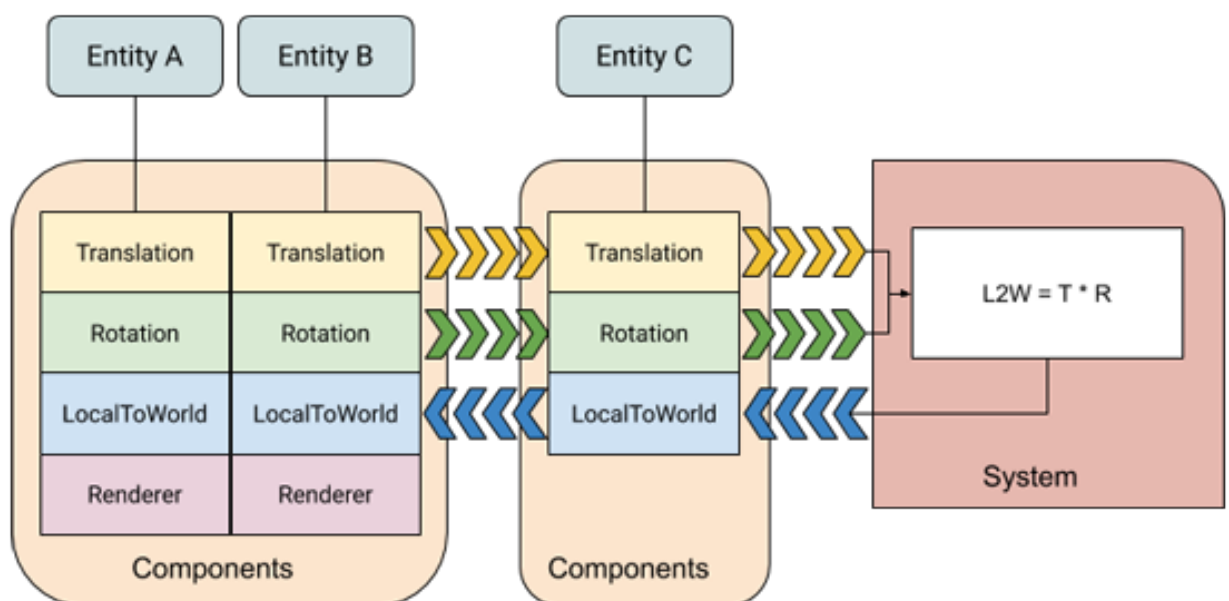


Рис. 3.3 Схема архітектури ECS

На цій схемі система читає компоненти перекладу та обертання, множить їх, а потім оновлює відповідні компоненти LocalToWorld ( $L2W = T * R$ ).

Той факт, що сутності A і B мають компонент *Renderer*, а сутність C ні, не впливає на систему, оскільки система не дбає про компоненти *Renderer*.

Можна налаштувати систему так, щоб вона вимагала компонент *Renderer*, і в цьому випадку система ігнорує компоненти сутності C; або, навпаки, коли створюється система для виключення сутностей із компонентами *Renderer*, вона потім ігнорує компоненти сутностей A та B.

### 3.2.2 *Архетипи (archetype)*

Унікальна комбінація типів компонентів називається *EntityArchetype*. Наприклад, тривимірний об'єкт може мати компонент для свого світового перетворення, один - для лінійного руху, один - для обертання і один - для візуального зображення. Кожен екземпляр одного з цих тривимірних об'єктів відповідає одному об'єкту, але оскільки вони мають однаковий набір компонентів, ECS класифікує їх як єдиний архетип:

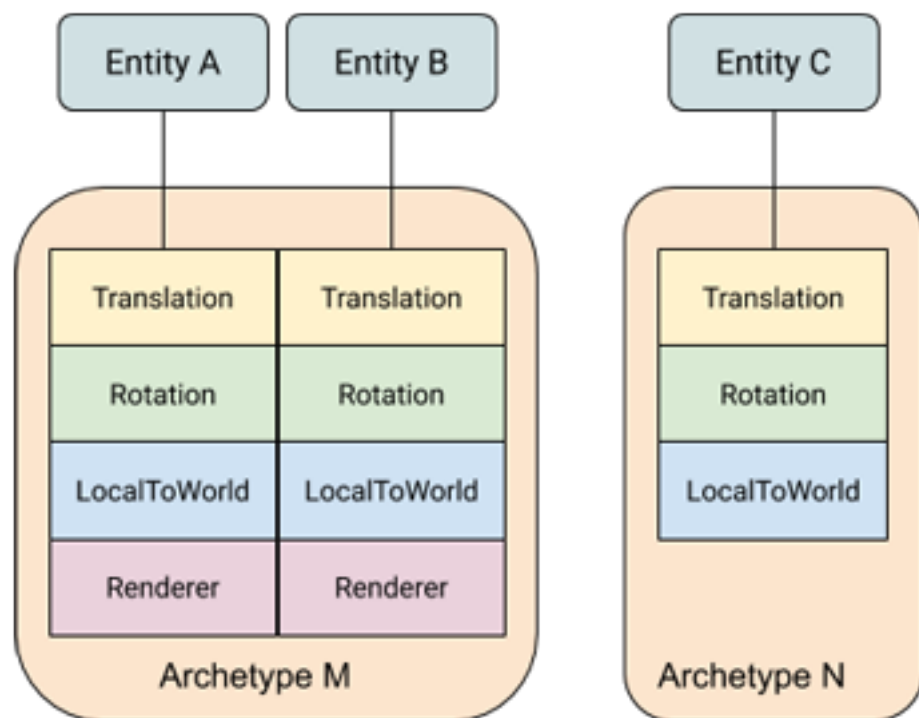


Рис. 3.4. Схема архетипів сутностей

На цій діаграмі сутності (рис. 3.4) А і В поділяють архетип М, тоді як сутність С має архетип N.

Щоб плавно змінити архетип сутності, можна додавати або видаляти компоненти під час виконання. Наприклад, якщо видалити компонент *Renderer* із сутності В, він переходить до архетипу N.

### 3.2.3 Memory chunks

Архетип сутності визначає де ECS зберігає компоненти цієї сутності. ECS розподіляє пам'ять по «елементам», кожен представлений об'єктом *ArchetypeChunk*. Кожен елемент завжди містить сутності одного архетипу. Коли елемент пам'яті стає повним, ECS виділяє новий елемент пам'яті для будь-яких нових сутностей, створених з тим самим архетипом. Якщо додаємо або видаляємо компоненти, які потім змінюють архетип сутності, ECS переміщує компоненти для цієї сутності в інший елемент пам'яті.

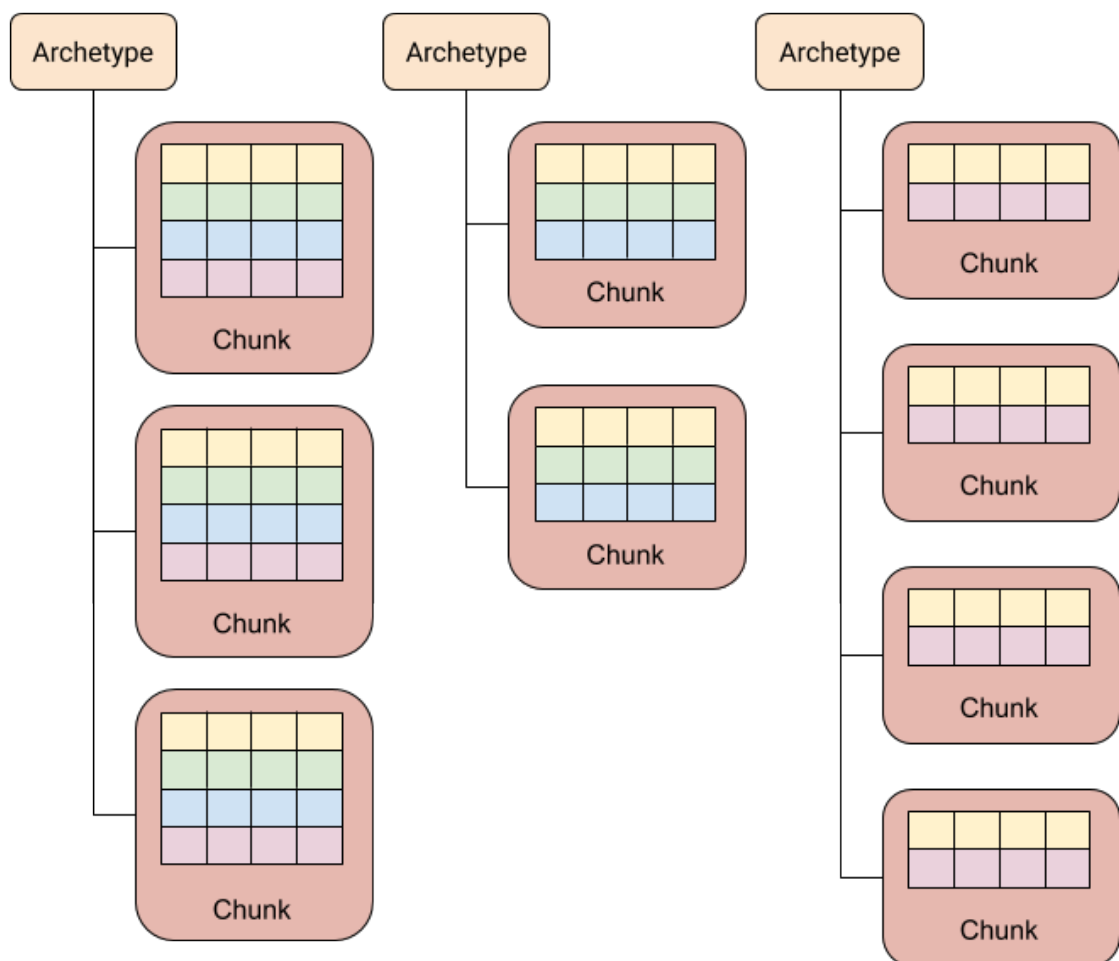


Рис. 3.5. Структурна схема елементів пам'яті



Ця організаційна схема (рис. 3.5) забезпечує взаємозв'язок «один до багатьох» між архетипами та елементами. Це також означає, що для пошуку всіх сутностей із заданим набором компонентів потрібен пошук лише серед існуючих архетипів, які, як правило, невеликі за кількістю, а не серед усіх сутностей, які, як правило, значно більші за кількістю.

ECS не зберігає сутності, що знаходяться в фрагменті, у певному порядку. Коли сутність створюється або змінюється на новий архетип, ECS поміщає його в перший елемент, що зберігає архетип, і який має простір. Однак елементи пам'яті залишаються щільно упакованими; коли сутність видаляється з архетипу, ECS переміщує компоненти останньої сутності в фрагменті де нещодавно були звільнені слоти в масивах компонентів.

Варто зауважити, що значення спільних компонентів у архетипі також визначають, які сутності зберігаються в якому фрагменті. Усі сутності в даному фрагменті мають однакові значення для будь-яких спільних компонентів. Якщо змінити значення будь-якого поля в спільному компоненті, модифікована сутність переміщується в інший елемент, як це було б, якби був змінений архетип цієї сутності. За необхідності виділяється новий елемент пам'яті.

#### 3.2.4 Запити сутності

Щоб визначити, які сутності система повинна обробити, використовується EntityQuery. Запит сутності здійснює пошук у існуючих архетипах для тих, що мають компоненти, що відповідають вашим вимогам. За допомогою запиту можна вказати такі вимоги до компонентів:

All - архетип повинен містити всі типи компонентів у категорії **Усі**.

Any - архетип повинен містити принаймні один із типів компонентів у будь-якій категорії.

None - архетип не повинен містити жодного з типів компонентів у категорії **Жодного**.

Запит сутності надає список фрагментів, що містять типи компонентів,

необхідних запиту. Потім ви можете переглядати компоненти цих фрагментів безпосередньо за допомогою IJobChunk (ітератор по екземплярам ArchetypeChunk).

### *3.2.5 Організація системи*

ECS організовує системи за світом (клас World), а потім за групами (клас ComponentSystemGroup). За замовчуванням ECS створює світ за замовчуванням із заздалегідь визначеним набором груп. Він знаходить усі доступні системи, створює їх екземпляри та додає до попередньо визначеної групи моделювання у типовому світі.

Можна вказати порядок оновлення систем в одній групі. Група - це своєрідна система, тому ви можете додати групу до іншої групи та вказати її порядок, як і будь-яку іншу систему. Усі системи в групі оновлюються перед наступною системою чи групою. Якщо порядок не вказано, ECS вставляє системи в порядок оновлення детермінованим способом, який не залежить від порядку створення. Іншими словами, один і той самий набір систем завжди оновлюється в однаковому порядку в межах своєї групи, навіть коли ви явно не вказується порядок.

Оновлення системи відбувається в основному потоці. Однак системи можуть використовувати завдання для вивантаження роботи в інші потоки. SystemBase забезпечує простий спосіб створення та планування робочих місць.

## **3.3 Сценарій ігрового додатку**

Гра складається з серії рівнів, виконаних у новорічному стилі, де гравець повинен завадити ланцюжку різнокольорових кульок досягнути лузи. Гравець керує фігуркою Діда Мороза, що може обертатися навколо своєї осі, та вистрілює кульки випадкового кольору. Індикатор на мішку Діда Мороза сигналізує про колір наступної кульки. Низки кульок одного кольору (три й більше штук) вибухають, скорочуючи ланцюжок. Тому гравцеві слід вистрілювати кульки так, щоб вони формували подібні низки. Основне завдання кожного рівня – протриматися до кінця снігопаду, що затихає при

знищенні кульок та мати рахунок більше чим у супротивника (бота). Після того, як він зникне, нові кульки перестають з'являтися. Якщо ланцюжок потрапляє в лузу гравець починає рівень заново. За знищення кульок нараховуються бали, спонукаючи гравця повертатися до гри й побити колишній рекорд.

### 3.4 Опис програмної реалізації

Наведемо основні елементи програмної реалізації та їх опис. Так як програмний продукт був виконаний за допомогою ECS фреймворка то в описі будуть представлені системи (як елементи логіки) та компоненти (як елементи даних). Усі системи та компоненти поділені по різним контекстам в залежності від їх призначення.

Список контекстів в проекті:

- Level;
- Animation;
- Balls;
- Chain;
- Collision;
- GameManagement;
- Input;
- Path;
- Player;
- Projectile;
- Score;
- Utils.

Наведемо частину опису елементів програмного продукту, по причині об'ємності та кількості останніх.

**Таблиця 3.1** – Опис загальних систем контексту Level

System	Types	Description
<i>ExecuteLevelLogic</i>	Execute	Логіка виконання систем, що являє собою ігрову логіку рівня.

**Таблиця 3.2** – Опис активних систем контексту Level

Reactive System	Types	Entity	Triggers	Description
<i>FinishLevel</i>	✗	Manage	FinishLevel	Логіка закінчення ігрового рівня та його розвантаження, а також виклик усіх подій, підписаних на закінчення рівня
<i>UploadLevel</i>	✗	Manage	StartLevel	Логіка початку ігрового рівня та його загрузка, а також виклик усіх подій, підписаних на початок рівня

**Таблиця 3.3** – Опис компонентів контексту Level

Component	Contexts	Unique	Event	Fields	Description
<i>FinishLevel</i>	Manage	✓	✓	▶	Флаг про те, що рівень завершено
<i>LevelPlay</i>	Manage	✓	✓	▶	Флаг того, що логіка ігрового рівня виконується, якщо вимкнути – то виконання ігрових систем призупиниться
<i>LogicSystems</i>	Manage	✓		Systems	Компонент зберігає системи, що відносяться до ігрового рівня
<i>StartLevel</i>	Manage	✓	✓	▶	Флаг про те, що рівень почався

**Таблиця 3.4** – Опис загальних систем контексту Animation

Reactive System	Types	Entity	Triggers	Description
<i>FinishMoveAnimation</i>	TearDown	Game	AnimationDone	Логіка обробки закінчення анімації та виклик пост подій
<i>MoveAnimationControl</i>	TearDown	Game	MoveAnimation	Логіка запуску та переривання анімації руху об'єктів
<i>ScaleAnimationControl</i>	TearDown	Game	ScaleAnimation	Логіка запуску та переривання анімації масштабування об'єктів

**Таблиця 3.5**– Опис компонентів контексту Animation

Component	Contexts	Unique	Event	Fields	Description
<i>MoveAnimation</i>	Game			float Vector3 Action	Дані для запуску анімації руху об'єкта
<i>AnimationDone</i>	Game			▶	Флаг про те, що анімація закінчена
<i>AnimationInfo</i>	Game			List	Лист подій, що повинні бути оброблені по завершенню анімації
<i>ScaleAnimation</i>	Game			float float Action	Дані для запуску анімації масштабування об'єкта

**Таблиця 3.6** – Опис загальних систем контексту Balls

System	Types	Description
<i>CheckAndSpawnBall</i>	<ul style="list-style-type: none"> <li>– Execute</li> <li>– Initialize</li> <li>– TearDown</li> </ul>	Логіка перевірки можливості появи нового шару, а також сама логіка створення
<i>UpdateBallDistanceBySpeed</i>	<ul style="list-style-type: none"> <li>– Execute</li> <li>– Initialize</li> <li>– TearDown</li> </ul>	Логіка оновлення позиції шарів відносно швидкості, тобто рух по треку

**Таблиця 3.7** – Опис реактивних систем контексту Balls

Reactive System	Types	Entity	Triggers	Description
<i>ChangeBallPositionOnPath</i>	TearDown	Game	DistanceBall	Логіка зміни позиції шара відносно відстані від початку
<i>CountBallColors</i>	×	Game	AnyOf: AddedBall RemovedBall	Система підрахунку кількості шарів кожного кольору
<i>UpdateColorBall</i>	×	Game	AllOf: Color Sprite	Логіка оновлення кольору шара

**Таблиця 3.8 – Опис компонентів контексту Balls**

Component	Contexts	Unique	Event	Fields	Description
<i>AddedBall</i>	Game			▶	Флаг про те, що шар з'явився в ігровій зоні
<i>BackEdge</i>	Game			▶	Флаг про те, що шар є останнім в ланцюгу
<i>BallColors</i>	Global	☑		Dictionary <ColorBall, int>	Кількість шарів кожного кольору на ігровому рівні
<i>BallId</i>	Game			int	Ідентифікатор шара для його пошуку та ідентифікації
<i>CheckTargetBall</i>	Game			▶	Флаг про те, що цей шар було додано в ланцюг
<i>Color</i>	Game			ColorBall	Колір шара
<i>DistanceBall</i>	Game			float	Дистанція, що була пройдена шаром від початку трека
<i>FrontEdge</i>	Game			▶	Флаг про те, що шар є найпершим у ланцюгу
<i>GroupDestroy</i>	Game			int	Номер для групування шарів при їх знищенні
<i>GroupSpawn</i>	Game			int	Кількість шарів, які треба створити одночасно на треку
<i>RemovedBall</i>	Game			▶	Флаг про те, що шар зник з ігрової зони гравця

### 3.5 Тестування програмного забезпечення

Для тестування ігрової системи було запущено декілька ігрових сесій, мета яких виявити якомога більше багів та помилок розробки. На рисунку 3.6 подано вид ігрового меню після запуску додатку.

Нажимаємо кнопку **Гра** та починаємо ігрову сесію (рис. 3.7).

Бот під управлінням штучного інтелекту одразу починає діяти і першим пострілом досягає поставленої задачі (рис. 3.8).



Рис. 3.6. Екран на початку гри

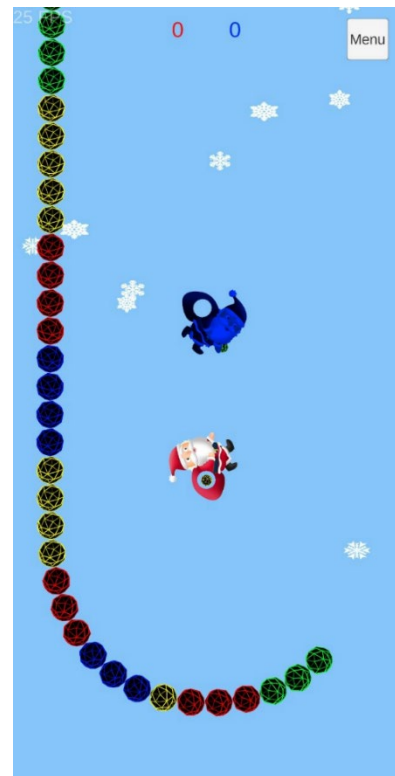


Рис. 3.7. Початок ігрового рівня

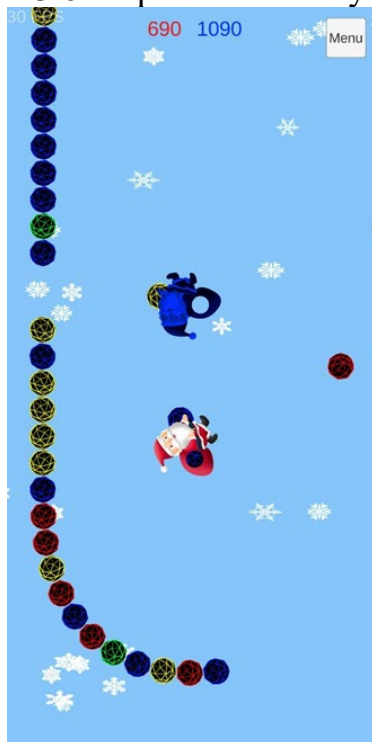


Рис. 3.8. Перші дії боту

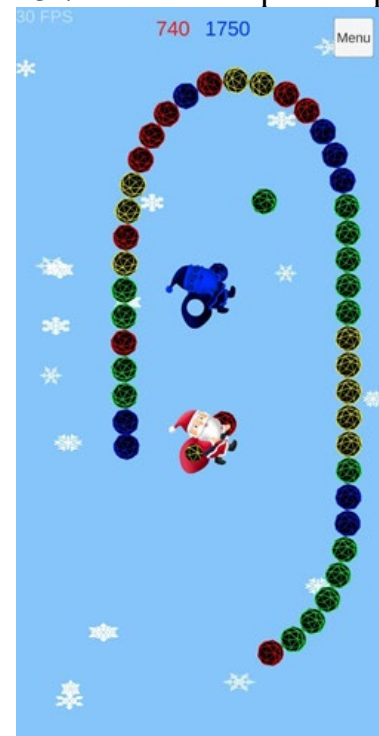


Рис. 3.9. Ігровий процес через деякий час гри

Після деякого часу гри ми можемо бачити рахунок як гравця, так і бота в верху ігрового екрану (червоним – гравець, синім – бот) (рис. 3.9).

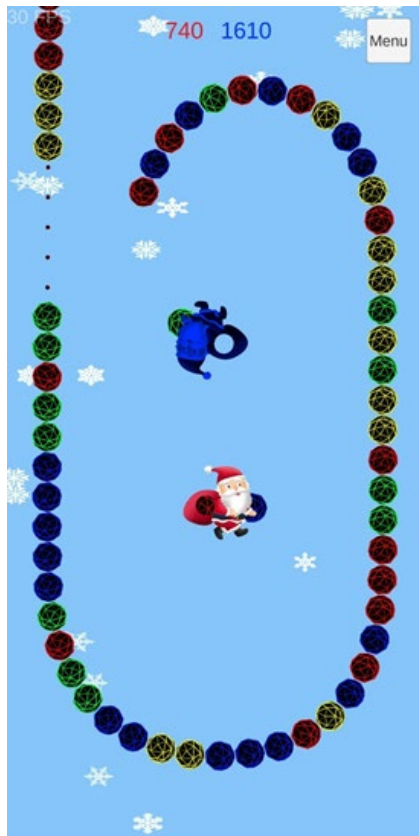


Рис. 3.10. Початок кінця гри та знищення усіх шарів



Рис. 3.11. Закінчення знищення ігрових шарів

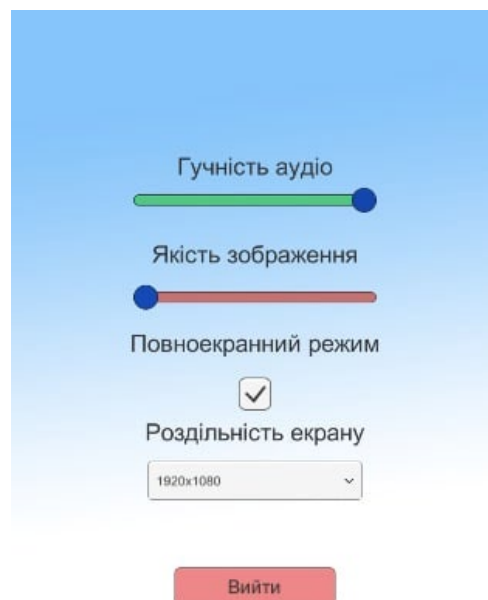


Рис. 3.12. Меню налаштувань



Ігровий процес продовжується до тих пір, поки шари не досягнуть кінцевої точки шляху та гра не буде завершена. Завершення гри можна бачити на рисунках 3.10 та 3.11

Після програшу гравець може порівняти свої результати та почати нову ігрову сесію. За допомогою Меню налаштувань (рис.3.12) користувач може змінити гучність, яскравість зображення, обрати режим та роздільну здатність екрану.

### **3.6 Висновки до розділу 3**

У третьому розділі описано процес створення ігрового додатку «Zima» за допомогою мови програмування C#, платформи Unity 3D, інтегрованого середовища розробки Visual Studio 2019. Надано ретельний опис процесу створення ігрового додатку, основні аспекти використаної архітектури ECS, наведені основні елементи програмної реалізації та їх опис, представлено концепцію гри «Zima», налаштування гри, зміну платформи, а також елементи опису графічного інтерфейсу головного меню та панелі виходу. Код розробленої гри досить простий, це дозволяє в майбутньому модернізувати її на свій розсуд під необхідні параметри, можлива модернізація гри до мережевої версії, зі статистикою, міні соціальною мережею, рейтингом гравців і призами, тому що.

## ВИСНОВКИ

В ході дослідження, було надано огляд та аналіз популярних мов програмування і сучасних засобів розробки ігор, отримано визначення оптимальних сфер використання для інструментальних засобів спираючись на аналіз процесу розробки прототипів. Розроблено методику проектування мобільних ігрових застосувань на основі використання випробуваних і перевірених досвідом досягнень в областях розробки і функціонування мобільних додатків і програмного забезпечення. На основі проведеного аналізу розроблено ігрову концепцію, створено ігрову логіку та правила, спроектовано ігрові механіки та прототип дизайну гри. Для генерування ігрового середовища і оцінки ігрового процесу запропоновано використання кривих Без'є третього порядку, розроблено алгоритм їх побудови як розширення редактору Unity. Проаналізовано найбільш популярні підходи до розробки штучного інтелекту для ігор, обрано та реалізовано концепт Кінцевих автоматів як найзручніший варіант розробки ігрових ботів.

Для програмної реалізації ігрової системи використовувалося середовище розробки Unity3D - це ігровий движок, що зібрав в собі фізичний, графічний рушій, великий набір інструментів та широкий функціонал. Гарний баланс малої кількості написаного коду та використання вбудованого інструментарію для налаштування елементів гри. Зручна візуалізація продукту та чудові можливості відладки як коду так і поведінки гри в цілому дають великі можливості для повторного використання коду та оптимізації графіки, фізики. Unity є найпотужнішим засобом для розробки і надто легким для використання представленого функціоналу. Він повністю задовольняє за своїми базовими характеристиками до того має потужну підтримку товариства, що значно знижує поріг входження та номінально обіцяє широкий спектр функціоналу та кросплатформеність, що дозволило виконати якісну імплементацію ігрової концепції та розроблених алгоритмів, спростити процес виробництва контенту та дизайну гри, реалізувати штучний інтелект ігрового бота. Для перевірки

працездатності системи було проведено її закриті альфа-тестування.

Серед програмних платформ обрано C#. За допомогою мови програмування C# і платформи Unity вирішено всі завдання, розроблено головні концепції та програмно реалізовано на основі архітектурного шаблону ECS нову казуальну ігрову систему з підтримкою штучного інтелекту, як супротивника гравця.

Результатом магістерської роботи є ігровий додаток «Zima» - це файл з розширенням .apk. Код гри побудовано таким чином, щоб в подальшому без труднощів можна вносити необхідні зміни. Наприклад такі, як підтримка на iOS пристроях, на комп'ютерах та гральних консолях, мережевий режим гри, рівні складності, рейтинг серед гравців.

Роботу може бути рекомендовано в якості методичного матеріалу при навчанні студентів методам програмування мультимедійних плагінів.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. AI - Agents & Environments. Назва з екрану. URL: [https://www.tutorialspoint.com/artificial\\_intelligence/artificial\\_intelligence\\_agents\\_and\\_environments.htm](https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_agents_and_environments.htm). ( Дата звернення: 28.11.2020).
2. Artificial Intelligence in Games. Назва з екрану. URL: <https://medium.com/aifrontiers/an-overview-of-artificial-intelligence-for-video-games-f491229c0e7d>. ( Дата звернення: 30.11.2020).
3. B.P. Douglass Real-Time UML. Developing Efficient Objects for Embedded Systems: Addison-Wesley Publishing Co., 1998, 365 p.
4. Booch G. Object-oriented analysis and design with applications. Second edition. The Benjamin/Cummings Publishing Company, Inc. 1994. 589 p.
5. C Sharp. URL: [https://uk.wikipedia.org/wiki/C\\_Sharp](https://uk.wikipedia.org/wiki/C_Sharp). (Дата звернення: 27.11.2020).
6. Chonoles M. J., Schardt J.A. UML 2 for Dummies. - Hungry Minds, 2003. - 412 p.
7. Designing Artificial Intelligence for Games. Назва з екрану. URL: [https://software.intel.com/content/www/us/en/develop/articles/designing\\_artificial-intelligence-for-games-part-1.html](https://software.intel.com/content/www/us/en/develop/articles/designing_artificial-intelligence-for-games-part-1.html). ( Дата звернення: 30.11.2020).
8. Enterprise Architect. Назва з екрану. URL: <https://sparxsystems.com/> (дата звернення 30.11.2020).
9. GDScript History. Godot documentation. Godot. URL: <https://web.archive.org/web/20160206055951/http://godotengine.org/projects/godot-engine/wiki/GDScript>. ( Дата звернення: 30.11.2020).
10. Godot documentation. Назва з екрану. URL: <https://docs.godotengine.org/en/stable/about/introduction.html>
11. Godot. URL: <https://uk.wikipedia.org/wiki/Godot>. ( Дата звернення: 30.11.2020).
12. Habrahabr – «Dagger 2. Часть вторая» URL: <https://habrahabr.ru/post/279641/> (Дата звернення: 27.11.2020).

13. Habrahabr – «Dagger 2. Часть первая». URL: <https://habrahabr.ru/post/279125/>. (Дата звернення: 27.11.2020).
14. Habrahabr – «Структура приложения для Android. URL: [habrahabr.ru/company/ncloudtech/blog/274025/](https://habrahabr.ru/company/ncloudtech/blog/274025/) (Дата звернення: 27.11.2020).
15. HTML5 Game Engines (Портал розробників Html5 ігор). URL: <https://html5gameengine.com/> (Дата звернення: 27.11.2020).
16. Hyper-Casual Games: Mobile Gamings Greatest Genre URL: <https://clevertap.com/blog/hyper-casual-games/>.
17. IBM DevOps. Назва з екрану. URL: <http://www.rational.com/uml>. (дата звернення 30.11.2020).
18. Kyte Kathryn. BREAKING DOWN THE 2020 GLOBAL GAMES MARKET REPORT. july 23, 2020. URL: <https://sidewalkhustle.com/breaking-down-the-2020-global-games-market-report/>.
19. Linietsky, Juan. Godot 2.0: Talking with the Creator. URL: <https://80.lv/articles/godot2-interview/>. (Дата звернення: 30.11.2020).
20. Nat Friedman. Xamarin Blog. C # is the best language for mobile development . URL: <https://devblogs.microsoft.com/xamarin/> (Дата звернення: 27.11.2020).
21. Object Management Group, 2003. OMG Unified Modeling Language Specification. Назва з екрану. URL: [www. omg. org](http://www.omg.org). (Дата звернення 30.11.2020).
22. Santee A., Fernandes B. About Ethanon Engine. URL: <http://doc.ethanonengine.com/> (Дата звернення 12.12.2020).
23. Unity Technologies Made with Unity. Назва з екрану. URL: <http://madewith.unity.com/> (дата звернення 24.11.2020).
24. Unity3d или Unreal Engine 4. URL: <https://stfalcon.com/ru/blog/post/unity3d-vs-unreal-engine-4>. (Дата звернення: 30.11.2020).
25. Азарова С. Какие языки программирования нужно знать, чтобы разрабатывать приложения под Android? URL: <https://medium.com/nuances-of-programming>. (Дата звернення: 27.11.2020)

26. Аналіз популярності веб сайтів. URL: <http://compare.easycounter.com/> (Дата звернення: 27.11.2020).
27. Використання веб-ресурсів для покращення візуального сприйняття інформації. URL: <http://inmad.vntu.edu.ua/portal/index.php>. Назва з екрану (Дата звернення: 27.11.2020).
28. Выбор игрового движка и языка программирования для разработки своих игр. Назва з екрану. URL: [https://pikabu.ru/story/vyibor\\_igrovogo\\_dvizhka\\_i\\_yazyika\\_programmirovaniya\\_dlya\\_razrabotki\\_svoikh\\_igr\\_5142472](https://pikabu.ru/story/vyibor_igrovogo_dvizhka_i_yazyika_programmirovaniya_dlya_razrabotki_svoikh_igr_5142472) (дата звернення 15.04.2020).
29. Гилл А. Введение в теорию конечных автоматов. М.:Наука, 1966. - 272с.
30. Движок Unity – особливості, переваги і недоліки. Назва з екрану. URL: <https://cubiq.ru/dvizhok-unity/>
31. Дмитрієва Н.Ю., Козуб Г.О. Розробка казуального ігрового додатку з підтримкою штучного інтелекту // Science and education: problems, prospects and innovations. Abstracts of the 3rd International scientific and practical conference. CPN Publishing Group. Kyoto, Japan. 2020. Pp.311-314. URL: <https://sci-conf.com.ua/iii-mezhdunarodnaya-nauchno-prakticheskaya-konferentsiya-science-and-education-problems-prospects-and-innovations-2-4-dekabrya-2020-goda-kioto-yaponiya-arhiv/>.
32. Дэвид Гриффитс. Head First. Программирование для Android. М.:Питер, 2016. 704 с.
33. История развития ИИ в играх: эволюция, алгоритмы, хардкор. Назва з екрану. URL: <https://stopgame.ru/blogs/topic/93248>
34. Ігри, в які грають люди. Назва з екрану. URL: <https://business.ua/economy/3996-ihry-v-iaki-hraiut-liudy> (дата звернення 15.04.2020).
35. Краткий обзор языка C#. URL: <https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/> . Назва з екрану (Дата звернення: 27.11.2020).
36. Лучший искусственный интеллект в играх, или Почему ИИ – это подделка. Назва з екрану. URL: <https://www.igromania.ru/article/29712/>

Luchshiy\_iskusstvennyy\_intelle kt\_v\_igrah\_ili\_Pochemu\_II-yeto\_poddelka.html.  
(Дата звернення: 29.11.2020).

37. Меднікс З., Дорнін Л. Програмування під Android. URL: [http://mikrotik.kpi.ua/index.php/courses-list/android/39-create-your-first-app-for-and](http://mikrotik.kpi.ua/index.php/courses-list/android/39-create-your-first-app-for-android)  
roid. Загол. з екрана. (Дата звернення: 27.11.2020).

38. Офіційний github репозиторій Pixijs. URL: <https://github.com/pixijs/pixi.js> – (Дата звернення: 27.11.2020).

39. Офіційний портал розробників XNA. URL: [https://msdn.microsoft.com/ru- RU/games-development-msdn](https://msdn.microsoft.com/ru-ru/games-development-msdn). ( Дата звернення: 27.11.2020).

40. Офіційний сайт GameMaker. URL: <http://www.yooyogames.com/gamemaker> ( Дата звернення: 27.11.2020).

41. Офіційний сайт Unity. URL: <http://unity3d.com/ru/> (Дата звернення: 27.11.2020).

42. Паласиос Х. Unity 5.x. Программирование искусственного интеллекта в играх: пер. с англ. Р. Н. Рагимова. - М.: ДМК П пресс, 2017. -272 с.:

43. Парр Э. Програмируемые контроллеры. Бином, 2007. 41 с.

44. Пол Дейтел. Android для разработчиков. М.:Питер, 2016. 512 с.

45. Популярний портал розробників ігор. URL: <http://www.gamedev.net/>  
( Дата звернення: 27.11.2020).

46. Портал ігрових новин . URL: <http://www.3dnews.ru/games/622071>.  
(Дата звернення: 29.11.2020).

47. Портал ігрових новин. URL: [http://www.sk- gaming.com/content/4811](http://www.sk-gaming.com/content/4811).  
(Дата звернення: 27.11.2020).

48. Роберт К. Мартин. Чистый код: создание, анализ и рефакторинг. – М.: Питер, 2016. – 464 с.

49. Роллингз Э. Проектирование и архитектура игр: пер. с англ. / Э. Роллингз, Д. Моррис. – М.: Вильямс, 2006.- 1040с.

50. Стив Макконнелл. Совершенный код. Мастер-класс. – М.:Питер, 2016. – 896 с.

51. Техническая коллекция Schneider Electric. Выпуск 16, 2006. 206 с.
52. ТОП10 мов програмування у 2020. Назва з екрану. URL: <http://apers.kpi.ua/top10-mov-programuvania-2020>. (Дата звернення: 27.11.2020).
53. Хокинг, Джозеф. Unity — в действии. Мультиплатформенная разработка на C# / Джозеф Хокинг. — К. : Питер, 2016. — 336 с.
54. Чад Фаулер. Программист-фанатик. —М.:Питер, 2016. — 208 с.
55. Эрих Гамма. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — М.:*Microsoft*, 2016. — 366 с.



## ДОДАТОК

## Код GameplayController.cs

```
using System;
using System.IO;

using NLog;
using UnityEngine;
using Entitas;

public class GameController : MonoBehaviour
{
    public bool isDebug = false;
    public LevelConfig config;

    private Systems _systems;

    private static string tempFolder;
    private NLog.Logger logger;

    void Start()
    {
        InitializeLogger();
        if (isDebug)
        {
            logger.Trace("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
        }

        Contexts contexts = Contexts.sharedInstance;

        InitializeSingletonComponents(contexts);
        _systems = CreateSystems(contexts);
        _systems.Initialize();
    }

    void Update()
    {
        try
        {
            _systems.Execute();
            _systems.Cleanup();
        }
        catch (Exception ex)
        {
            logger.Error(ex, "Failed to process update frame");
        }
    }

    private void OnDestroy()
```

```

{
    _systems.TearDown();
}

private void InitializeSingletonComponents(Contexts contexts)
{
    contexts.global.SetLevelConfig(config);
    contexts.global.isDebugAccess = isDebug;
    contexts.manage.SetLogicSystems(CreateLogicSystems(contexts));
}

private Systems CreateSystems(Contexts contexts)
{
    return new Feature("Game")
        .Add(new UpdateDeltaTimeSystem(contexts))

        //Level
        .Add(new UploadLevelSystem(contexts))
        .Add(new ExecuteLevelLogicSystem(contexts))           // here locates exe
cution of Logic systems
        .Add(new FinishLevelSystem(contexts))

        // Log recording
        .Add(new RecordLogMessageSystem(contexts))

        //CleanUp
        .Add(new DestroyInputEntityHandleSystem(contexts))
        .Add(new DestroyGameEntityHandleSystem(contexts))
        .Add(new DestroyManageEntityHandleSystem(contexts))
        ;
}

private Systems CreateLogicSystems(Contexts contexts)
{
    return new Feature("Logic")
        //Initialization
        .Add(new InitializePathSystem(contexts))
        .Add(new InitializePlayerSystem(contexts))
        .Add(new InitializeBotSystem(contexts))

        //RayCasting
        .Add(new BallRayCastSystem(contexts))
        //Overlapping
        .Add(new BallOverlapSystem(contexts))

        //Counter
        .Add(new TickCountersSystem(contexts))

        //Animation finishing
        .Add(new FinishMoveAnimationSystem(contexts))

```

```

//Collision actions
.Add(new EnteringBallsToScreenSystem(contexts))
.Add(new CollisionObjectDestroySystem(contexts)) //разобраться с
ЭТИМИ КОЛЛИЗИЯМИ
.Add(new ExplodeBallSystem(contexts))
.Add(new ConnectChainsSystem(contexts))
.Add(new CollidingAndInsertingProjectileSystem(contexts))

//Destroying balls in chain
.Add(new MatchInsertedBallInChainSystem(contexts))
.Add(new VisualDestroyingBallsSystem(contexts))
.Add(new CutChainSystem(contexts))

//Spawn
.Add(new CheckAndSpawnBallSystem(contexts))

//Movement
.Add(new UpdateBallDistanceBySpeedSystem(contexts))
.Add(new ChangeBallPositionOnPathSystem(contexts))

//Updating
.Add(new SetChainEdgesSystem(contexts))
.Add(new SetChainSpeedSystem(contexts))

//Animation beginning
.Add(new MoveAnimationControlSystem(contexts))
.Add(new ScaleAnimationControlSystem(contexts))

//Colors
.Add(new UpdateColorBallSystem(contexts))
.Add(new CountBallColorsSystem(contexts))

//Score
.Add(new ScoreCounterSystem(contexts))
.Add(new ShowGameScoreSystem(contexts))

//Ability
.Add(new InputAbilitySystem(contexts)) // this system just for te
st and imitating ability invoking behaviour
.Add(new InvokingAbilitySystem(contexts))

//AI Bot
.Add(new BotHandleSystem(contexts))
.Add(new ScanBallTrackSystem(contexts))
.Add(new RotateBotSystem(contexts))

```

```

        .Add(new ShootBotSystem(contexts))

        //Input
        .Add(new TouchHandleSystem(contexts))
        //Player
        .Add(new RotatePlayerSystem(contexts))
        .Add(new ShootPlayerSystem(contexts))
        .Add(new BallExchangePlayerSystem(contexts))
        .Add(new UpdatePointerLengthSystem(contexts))
        //Shooting
        .Add(new ShootingForceSystem(contexts))

        //GameOver
        .Add(new GameEndProcessSystem(contexts))
        ;
    }

    #region Logger Methods

    private void InitializeLogger()
    {
        try
        {
            if (LogManager.Configuration != null)
                return;

            var target1 = new NLog.Targets.FileTarget();
            target1.FileName = Path.Combine(GetTempFolder(), "Debug.log");
            target1.KeepFileOpen = false;
            target1.Layout = "${longdate}|${level:uppercase=true}|${logger}|${message}|${exception:format=ToString}";
            var target2 = new NLog.Targets.DebuggerTarget();

            LogManager.Configuration = new NLog.Config.LoggingConfiguration();
            LogManager.Configuration.AddTarget("logFile", target1);
            LogManager.Configuration.AddTarget("debug", target2);
            LogManager.Configuration.LoggingRules.Add(new NLog.Config.LoggingRule(
                "*", LogLevel.Trace, target1));
            LogManager.Configuration.LoggingRules.Add(new NLog.Config.LoggingRule(
                "*", LogLevel.Trace, target2));
            LogManager.ReconfigExistingLoggers();
        }
        catch (Exception)
        {
        }
        finally
        {
            logger = LogManager.GetCurrentClassLogger();
        }
    }
}

```

```
private static string GetTempFolder()
{
    if (string.IsNullOrEmpty(tempFolder))
    {
        var appData = Environment.GetFolderPath(Environment.SpecialFolder.Appli
cationData);
        tempFolder = Path.Combine(appData, "NeonZuma");

        if (!Directory.Exists(tempFolder))
        {
            Directory.CreateDirectory(tempFolder);
        }
    }

    return tempFolder;
}

#endregion
}
```