МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ВИПУСКНА РОБОТА

на тему:

«Інформаційне та програмне забезпечення ігрової системи жанру казуальних головоломок»

Завідувач
випускаючої кафедри **Довбиш А.С.**

Керівник роботи **Шелехов І.В.**

Студентка групи ІН –62 **Марченко М.А.**

СУМИ 2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютерних наук

Затверджую _____

Зав. кафедри Довбиш А.С.

“_____” _____20__ р.

ЗАВДАННЯ

до випускної роботи

Студентки четвертого курсу, ІН–62 спеціальності “Комп’ютерні науки” денної форми навчання Марченко М.А.

Тема:“**Інформаційне та програмне забезпечення ігрової системи жанру казуальних головоломок** ”

Затверджена наказом по СумДУ

№_____ від _____20__ р.

Зміст пояснювальної записки: 1) аналіз проблеми та постановка задачі; 2) проектування комп'ютерної гри; 3) програмна реалізація.

Дата видачі завдання “_____”_____20__ г.

Керівник випускної роботи _____Шелехов І.В.

Завдання прийняв до виконання _____ Марченко М.А.

РЕФЕРАТ

Записка: 42 стор., 15 рис., 8 табл., 1 додаток, 13 джерел.

Об’єкт дослідження — процес проектування та реалізації інформаційного та програмного забезпечення гіперказуальної ігрової системи

Мета роботи — розробка та програмна реалізація гіперказуальної ігрової системи з підтримкою штучного інтелекту, як супротивника гравця.

Методи дослідження — методи проектування ігрових систем, методи побудови кривих Без’є, методи роботи штучного інтелекту, методи розробки програмного продукту на основі ECS підходу.

Результати — було виконано розробку інформаційного та програмного забезпечення гіперказуальної ігрової системи. В роботі проведено аналіз та порівняння популярних ігрових двигунів, були описані їх слабкі та сильні сторони. Було досліджено методи побудови кривих Без’є третього порядку та розроблений алгоритм їх побудови як розширення редактору Unity. Також було проаналізовано найбільш популярні підходи розробки штучного інтелекту для ігор та вибрано концепт Кінцевих автоматів як найзручніший варіант розробки простих ботів. Було вивчено архітектурний шаблон ECS, на основі якого і було розроблено ігрову систему.

UNITY, ECS, ІГРОВИЙ ДВИГУН, ГІПЕРКАЗУАЛЬНІ ІГРИ,
GAME ARTIFICIAL INTELLIGENCE, КРИВА БЕЗ’Є

ЗМІСТ

ВСТУП

Комп'ютерні ігри – це програми різної складності, чия ціль навчати або розважати людей різних вікових груп. Найцікавіше, що ріст ігрового виробництва співпадає з ростом технологічного прогресу середини XX століття. Разом із винаходом та розвитком комп'ютерів почали з'являтися і комп'ютерні ігри. Поява перших комп'ютерних ігор стартує з 1940 року. Довготривалий час ігри не користувалися попитом по одній причині – не кожен міг мати комп'ютер для ігор вдома. Вже з 1970 років з'являються аркадні автомати, ігрові консолі та домашні персональні комп'ютери.

У 1889 році японець Фусадзіро Ямаути створив першу компанію в світі Marufuku, яка займалася виробництвом ігральних карт Ханафуда. У 1907 році вона змінила ім’я на Nintendo Koppai, яка в подальшому стала найбільшою компанією інтерактивних розваг, в тому числі і відеоігор. Перший прототип комп'ютерної гри було створено в США, в Технологічному Університеті штату Массачусетс. Розроблена гра мала назву Space War і була призначена для двох гравців. ЇЇ суть полягала в тому, що два космічних корабля намагалися збити, за допомогою ракетних пострілів, один в одного.

Через декілька десятиліть, в 1972 році, Нолан Башнелл випустив відому в світі гру під назвою "Pong", суть якої в утриманні м'ячика на ігровому полі за допомогою рухомих бар’єрів по краях екрану. Ця гра стала одним з найважливіших етапів в історії розвитку комп'ютерних ігор. Вже через декілька років була випущена перша ігрова приставка з єдиною грою на ній. Після цього персональні комп'ютери та ігрові приставки продовжили покращуватися та вдосконалюватися, а ігри для них ставали більш технологічними, цікавішими та яскравішими. Зародилася ціла сфера комп'ютерної індустрії, яка за своїми об’ємами наздогнала навіть ринок кіноіндустрії. [1]

За останні роки, утворився гігантський за розміром аудиторії і за обсягом коштів ринок – гіперказуальні ігри. Важливою особливістю ринка є крайня хаотичність, яка характеризується постійною зміною трендів і правил. Кожен день з'являється величезна кількість нових проектів в цьому жанрі. Це обумовлено відносною простотою їх створення.

Ігри даного жанру відрізняються від інших занадто короткою тривалістю ігрової сесії – від двох до п'яти хвилин. Нон-геймер не може постійно грати в ігри, і завданням розробника є за настільки коротку сесію зацікавити та навчити грати. Він повинен встигнути пройти декілька перших рівнів з успішним завершенням і вийти з гри максимально задоволеним. [2]

В рамках даної роботи буде описано створення казуальної гри під назвою "NeonZuma" на движку Unity. Даний продукт буде створений на базі ОС Windows 10, але також в подальшому може бути портований на інші ОС, зокрема слід виділити мобільні платформи, як найпопулярніші серед гіперказуальних ігор.

1 АНАЛІТИЧНИЙ ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Особливості ігрових система жанру казуальної головоломоки

Гіперказуальні ігри – це ігри, в які легко навчитись грати, але вони захоплюють не менше інших ігор, вимагаючи набагато менше ігрового часу.

Гіперказуальні ігри, зазвичай, мають інтуїтивну механіку, яка може залишатися незмінною протягом ігрового процесу або може вимагати більшої спритності, оскільки гра збільшується у складності. Більш складні ігри можуть збільшувати незадоволеність гравців, у яких немає часу, необхідного для отримання гарних навичок в грі, але гіперказуальні ігри усувають цей бар'єр. [3]

Існує кілька категорій гравців: хардкорні, мідкорні і казуальні. Основна орієнтація розробників ігор завжди була спрямована на них. Але усі гравці складають відносно невеликий відсоток від усіх людей всієї планети. Гіперказуальні проекти почали орієнтуватися на так званих нон-геймерів, тобто на більший відсоток людей, які ніколи не грали в ігри. У звичайних категорій гравців є закладені шаблони поведінки в іграх, які допомагають їм пристосуватися до більшості ігрових механік. В цей же час у нон-геймерів вони відсутні. Розробники намагаються використовувати механік подібні до тих, що є в звичайних мобільних додатках, якими користуються мільйони людей, наприклад, WhatsApp, Tinder або Instagram. Головним завданням розробників гіперказуальних ігор є створення геймплея, який підлаштовується під шаблони зрозумілі нон-геймерам. Важливо розуміти, що нон-геймери – це максимально широка аудиторія, відповідно саме вони роблять продукт масовим.

Основою гіперказуальних ігрових продуктів є геймплей. Необхідно відзначити, що тут мається на увазі таке поняття як core gameplay. Людина, яка грає в ці ігри, повторює тисячократно одні і ті ж дії. Геймплей в таких проектах подібний потоку, так як людина концентрується на одній дії. [2]

Якщо подивитися обсяги гіперказуальних ігрових проектів сьогодні, то можна виділити певні механіки та шаблони, які найбільш часто використовуються. Важливо відзначити, що немає єдиного підходу в структуруванні таких механік. Існують різні класифікації в різноманітних джерелах. Ось основні з них:

Механіка часу

Ігри з механікою синхронізації – це обмеження часу на гру, що змушує гравця прискорюватись та грати швидке. Швидкість, з якою рухаються компоненти гри визначає складність гри. При використанні механіки синхронізації ігровий процес може бути дуже коротким, що сприяє гіперказулальному підходу. Розробники ігор також розуміють, наскільки страшним для гравця може бути швидкий програш і змушення починати гру з початку, тому розробники надають гравцеві не одну спробу та намагаються максимально полегшити рестарт гри.

Механіка спритності

Ігри, які використовують механіку спритності змушують вас діяти швидко, зіткнувшись з ігровою небезпекою, гравець мусить діяти швидко, щоб уникнути ігрової смерті та програшу. Класичний приклад механіки спритності – Pac-Man. Ще один приклад механіки спритності – гра Змійка, де грати стає все складніше, коли ви з’їли багато яблук .

Механіка головоломки

Жанр головоломки займає 60% ринку мобільних ігор. Ігри-головоломки створюються для повсякденних розваг через прості розумові завдання. Тетріс – одна з найперших історій успіху в іграх – прекрасний приклад механіки головоломки. По суті, створюється ілюзія рішення якоїсь задачі і ілюзія пазла. Гіперказуальні пазл-ігри дуже легкі і приємні. Прогрес йде швидко і створюється відчуття рішення великої кількості завдань в одиницю часу, що тягне за собою ефект задоволення.

Механіка злиття

В іграх, що використовують механіку злиття, ви зазвичай комбінуєте або замінюєте клітинки в рядку чи стовпці, щоб знищити якомога більше елементів. Наприклад, є багато ігор, в яких гравець повинен міняти місцями деякі ігрові об’єкті, щоб зібрати три або більше, елементів одного типу, тим самим усуваючи ці елементи. Це простий, але захоплюючий геймплей. Бувають моменти, коли знищення однієї лінії елементів спричиняє ще одну, або більше усунень. Це виглядає як деякий ланцюжок усунень, що захоплює гравця ще більше.

Механіка повороту

Це ігри, які вимагають від гравця рухатись по ігровому шляху та уникати всілякі перешкоди під час руху – використовують механіку повороту. В Таких іграх розробники дають ілюзію безмежного руху, але насправді існує лише декілька варіантів руху, з яких і вибирає гравець [3]

Усі ці механіки призначені для того, щоб розважати користувачів та різноманітити геймплей. Я вважаю гіперказуальні ігри "іграми для всіх". Це ігри, які, в першу чергу, фокусуються на розважальному ігровому процесі доступному усім, не особливо концентруючись на звичні для складних ігор механіках та вишукану графіку. Очевидно, що в даний час всі ігри повинні фокусуватися на вмінні розважати гравця, дарувати йому задоволення від гри.

З огляду на все сказане вище, найбільш зручно розглядати казуальні ігри в більш ширшому значенні. У міру того, як бюджети гіперказуальних ігор збільшуються, а звичні ігрові жанри стають більш простими для нон-геймерів, межа між гіперказуальними і hardcore іграми поступово стирається. Останнім часом ця тема регулярно обговорюється на різних ігрових конференціях. Настане час – коли поняття гіперказуальної гри буде мати зовсім інше значення, ніж зараз, або і зовсім не буде мати ніякого значення, оскільки ми всі будемо займатися розробкою ігор, розрахованих на масового споживача. [4]

1.2 Сучасні програмні середовища розробки ігрових систем

Термін «ігровий двигун» з'явився в середині 1990-х в контексті комп'ютерних ігор жанру шутер, схожих на популярну в той час Doom. Архітектура програмного забезпечення Doom була побудована таким чином, що представляла собою розумно виконаний поділ центральних компонентів гри (наприклад, підсистеми тривимірної графіки, розрахунку фізичних об'єктів, звуковий і інших) і графічних ресурсів, ігрових правил та іншого. Як наслідок, це отримало певну популярність за рахунок того, що можна було створювати ігри з мінімальними трудовитратами, тобто основну кількість, подібних для кожної гри, задач брав на себе ігровий двигун [5].

Більшість перших ігрових двигунів були розроблені і налаштовані для того, щоб запустити одну гру на певній платформі. І навіть найбільш узагальнені багатоплатформні двигуни підходять для побудови ігор лише певного жанру, наприклад шутерів або RPG. Тобто можна більш точно сказати, що ігровий двигун стає оптимальним при його застосуванні для розробки ігор подібного жанру. Даний ефект проявляється від того, що програмне забезпечення являє собою набір компромісів, заснованих на тих припущеннях, якою має бути гра. Наприклад, проектування рендерінга всередині споруд призведе до того, що двигун не буде таким же гарним для відображення відкритих просторів. У першому випадку двигун може використовувати BSP-дерево для відтворення об'єктів, близьких до камери. У той же час для відкритих просторів можуть використовуватися менш точні способи, а також більш активно застосовувати технології відтворення з різним ступенем деталізації, залежно від відстані.[5].

Крім багаторазово використовуваним програмним компонентам, ігрові двигуни надають набір візуальних інструментів для розробки. Ці інструменти зазвичай складають інтегроване середовище розробки для спрощеної, швидкої та явної розробки ігор. Ці ігрові двигуни іноді називають «ігровим підпрограмним забезпеченням», так як, вони скоріше надають гнучку і

багаторазово використовувану програмну платформу з усією необхідною функціональністю для розробки ігрового продукту, скорочуючи витрати, складність і час розробки – всі критичні фактори в висококонкурентній індустрії відеоігор.

Загалом ігровий двигун є розробкою компанії, що займається розробкою ігор і орієнтований на одну або декілька ігор всередині цієї компанії. Але з часом з’явилися компанії які замість розробки ігор вирішили розробляти ігрові двигуни для розробників ігор, кількість яких за останні роки почала збільшуватися. А з часом, такі двигуни почали робити безплатними або частково безплатними і майже кожен вже може почати розробку власної гри, лише ознайомившись з правилами роботи ігрового двигуна. Серед найпопулярніших ігрових двигунів я би виділив Unity, Unreal Engine 4 та Godot, кожен зі своїми перевагами та недоліками.

Unity

Unity – це середовище для розробки комп'ютерних ігор, в якій об'єднані різні програмні засоби, що використовуються при створенні програмного забезпечення – текстовий редактор, компілятор, відладчик і так далі. При цьому, завдяки зручності використання, Unity робить створення ігор максимально простим і комфортним, а мультиплатформенність ігрового двигуна дозволяє розробникам обійняти більшу кількість ігрових платформ і операційних систем. [6]

Як правило, ігровий двигун має багато функціональних можливостей, що дозволяють їх використовувати в різних іграх, наприклад, моделювання фізичних середовищ, динамічні тіні, карти нормалей і багато іншого. На відміну від більшості ігрових двигунів, в Unity є дві основні переваги: наявність візуального середовища розробки і кроссплатформенність. Перший пункт включає не тільки інструменти візуального моделювання, а й інтегроване середовище розробки, автоматичну збірку, що направлено на підвищення продуктивності розробників. Під кроссплатформенністю надаються не тільки місця розгортання (на персональному комп'ютері, на

мобільному пристрої, консолі і т.п.), але і наявність інструментарію розробки (інтегроване середовище може використовуватися під Windows і Mac OS). [7]

Третьою перевагою називають модульну систему компонентів Unity, за допомогою якої відбувається конструювання ігрових об'єктів, які являють собою комбіновані набори функціональних елементів. На відміну від механізмів наслідування, об'єкти в Unity створюються за допомогою об'єднання функціональних компонентів. Такий підхід спрощує створення прототипів, що актуально при розробці ігор. [7]

В якості недоліків приводять обмеження візуального редактора при роботі з багатокомпонентними схемами. Другим недоліком називається відсутність підтримки Unity посилань на зовнішні бібліотеки, роботу з якими програмістам доводиться налаштовувати самостійно, і це також ускладнює командну роботу. Ще один недолік пов'язаний з використанням шаблонів екземплярів (англ. prefabs). З одного боку, ця концепція Unity пропонує гнучкий підхід візуального редагування об'єктів, але з іншого боку, редагування таких шаблонів має багато своїх недоліків.[7] Також, WebGL-версія двигуна, в силу специфіки своєї архітектури (трансляція коду з C# В C++ і далі в JavaScript), має ряд невирішених проблем з продуктивністю, споживанням пам'яті і працездатністю на мобільних пристроях.

Unreal Engine

Як і у випадку Unity, Unreal Engine дозволяє розробляти ігри для більшості операційних систем, консолей і мобільних платформ. Завдяки підтримці різних драйверів рендерінга графіки Direct3D, OpenGL та ін, підтримці різних систем звуку і можливостей для гри через інтернет, Unreal Engine можна використовувати для створення самих різних ігор. До складу Unreal Engine входить і набір засобів розробки (SDK), і редактор.

На відміну від Unity, Unreal Engine має відкритий код програмного забезпечення, який написаний на C++. Якщо говорити про недоліка та переваги Unreal Engine у порівнянні з Unity, потрібно розуміти, що Unity

більше підходить для мобільних і 2D ігор, а UE4 дозволяє створювати високотехнологічну графіку.

У Unreal Engine є примітна система скриптів, так звані Blueprints, які створюються без використання мов програмування. Blueprints – це візуальна система створення скриптів, з його допомогою можна описати все що завгодно – від дій логіки персонажа до процедурної генерації у грі. За допомогою використання Unreal Engine можна створювати і реалістичну графіку при архітектурній візуалізації. [8]

Підводячи підсумки короткого огляду Unreal Engine зазначу, що по більшості, UE дозволяє робити більш складну та гарну графіку, ніж Unity, але і пред'являє більше вимог до навичок розробника. У простих іграх ви не побачите різниці в кінцевому продукті, а створення такого на Unity займе менше часу.

Godot

Завдання Godot – бути максимально простим і самодостатнім середовищем для розробки ігор. Двигун дозволяє розробникам створювати ігри з нуля, не користуючись більш ніякими інструментами, за винятком тих, які необхідні для створення ігрового контенту (елементи графіки, музичні треки і т.д.). Написання коду також не вимагає зовнішніх інструментів.

Архітектура двигуна побудована на основі концепції дерева з успадкованих "сцен". Кожен елемент сцени(вузол), сам може стати повноцінною сценою. Тому при розробці можна змінювати повністю всю архітектуру проекту і працювати з комплексними сценами на рівні простих абстракцій.

Всі ігрові ресурси, від скриптів до графічних ассетів і ігрових сцен, зберігаються в папці проекту як бінарні файли, і не представляють собою складну базу даних проекту. Ресурси, які не є комплексними даними, зберігаються в простих текстових форматах. Ці рішення дозволили значно спростити роботу команд через системи управління версіями.[9]

Порівнюючи і вибираючи середовище розробки для свого проекту, я зупинив свій вибір на Unity. Це обумовлене тим, що Unreal Engine краще підходить для розробки великих проектів з потужною графікою та складною логікою, а розробка гіперказуальної гри займе занадто багато часу; Godot більше підходить для даного проекту, але цей двигун ще досить молодий, та його інструменти знаходяться на стадії розробки, або стабілізування, також по цьому двигуну не так багато навчального матеріалу; Unity ж гарно підходить для простих ігор: завдяки підтримці мови програмування C#, час на реалізацію ігрової логіки витратиться достатньо менше часу, а завдяки гарній мультиплатформеності двигуна, гру можливо буде легко випустити на мобільних та інших платформах.

1.3 Огляд штучного інтелекту в іграх

Ігровий штучний інтелект (англ. artificial intelligence – AI) – набір програмних підходів та шаблонів, які використовуються в комп'ютерних іграх для створення ілюзії інтелекту, керованих комп'ютером. Ігровий AI включає в себе алгоритми теорії управління, робототехніки, комп'ютерної графіки та інформатики в цілому.

Реалізація AI сильно впливає на геймплей, системні вимоги та бюджет гри, і розробники повинні враховувати ці вимоги, намагаючись зробити невимогливий до ресурсів AI. Тому підхід до ігрового AI серйозно відрізняється від розробки AI для інших сфер. В ігровому AI широко застосовуються всілякі спрощення, обмани і емуляції. Наприклад, в шутерах безпомилковий рух і миттєве прицілювання, властиве ботам, не дає жодного шансу гравцю, так що ці здібності штучно знижуються. Але боти повинні робити засідки, діяти командою і т.д., для цього застосовуються контрольні точки, розставленні на рівні.

Перший концепт AI зародився ще в далекому 1956 році, а визначення не було зв'язано напряму з поняттям інтелекту. Раніше система була набагато простішою. Перший бот у відеогрі був введений в 1951р, за 5 років до самого

визначення концепції AI. Він створювався для гри в шахи і представляв собою порівняно простий список шахових алгоритмів. Подальші відеоігри 60-х і 70-х, в більшості своїй, не мали AI, а були створені як ігри для двох гравців. [10]

Евристичні алгоритми ігрового AI використовуються в багатьох галузях всередині гри. Найбільш очевидне застосування ігрового AI можна побачити в контролі неігрових персонажів. Пошук шляху є іншим широко поширеним застосуванням ігрового AI — він завжди застосовується в стратегіях реального часу. Пошук шляху є алгоритмом для визначення, як неігровому персонажу перейти з однієї точки на рівні до іншої: потрібно враховувати ландшафт, перешкоди і т.д.

Концепція непередбачуваного (англ. emergent) AI була недавно реалізована в деяких іграх. Ігрові боти в цих іграх мають здатність "вчиться" з дій, зроблених гравцем, і їх поведінка змінюється відповідно. Насправді ці рішення взяті з обмеженої безлічі рішень, це дійсно часто дає бажану ілюзію інтелекту у грі. Часто розробники намагаються створити AI, який дійсно моделював би роботу людського мозку, але таких прикладів одиниці, тому що ця задача є значно складнішою, ніж емуляція деяких непередбачених для гравця дій.

У 1992 році британський вчений Стів Гранд вирішив спробувати себе в розробці комерційного програмного забезпечення. Він запропонував ідею віртуального домашнього улюбленця, який жив би на робочому столі Windows. Гра повинна була поступово навчатися новим трюкам завдяки внутрішній нейромережі.

Пізніше ідея перетворилася на повноцінну гру про видуманих створінь – норнів. Пухнасте створіння вилуплюється з яйця, а гравець повинен допомогти йому пізнати світ: навчити словам, повторюючи їх по багато разів і показуючи на те, що воно означає, а також змушувати виконувати деякі дії, заохочуючи хорошу поведінку. Проте Норни часто забувають свій досвід і помиляються – це і є величезне досягнення розробника.

В основі штучного інтелекту кожного створіння лежить нейромережа з тисячі нейронів, поділена на кластери. Кожен кластер виконує одне із завдань: почуття, фокусування уваги, пам'ять і прийняття рішень. Норни асоціюють дії зі схваленнями і покараннями і роблять узагальнення на основі попереднього досвіду, що допомагає їм діяти в нових умовах.

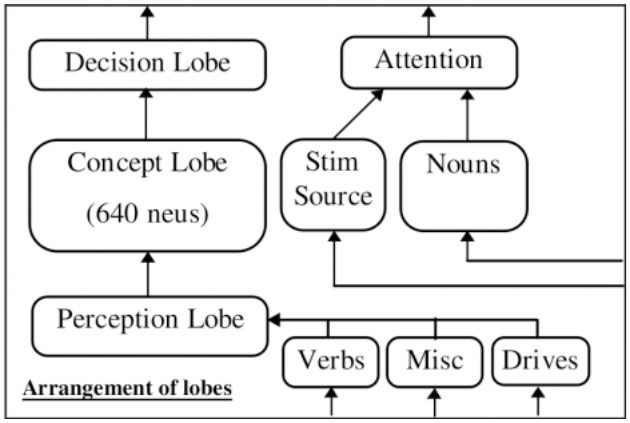

Рисунок 1.1 – Загальна схема мозку норна

Поведінка норнів регулюється гормонами, як у живих істот, тому вони можуть жадати спілкування і розваг, а також відчувати біль, голод, втому — все це має ефект на баланс хімічних елементів в їх крові на спеціальній консолі. [11]

Ще одне основне застосування AI в іграх – це моделювання поведінки гравця, щоб зрозуміти, як взаємодія з грою відчувається окремими гравцями. Складний AI повинен розуміти, що робить гравець і як він себе почуває при цьому. Щоб оцінити ігровий досвід гравця, розробники використовують методи машинного навчання, такі як контрольоване навчання на зразок машин опорних векторів або нейронні мережі, щоб побудувати моделі досвіду гравця. Тренувальні дані тут складаються з деякого аспекту гри або взаємодії гравця з грою, а цілі – це мітки, отримані з деякої оцінки досвіду гравця, зібраної, наприклад, з фізіологічних вимірювань або анкет.

Збільшення залученості гравців – це трохи складніша проблема. Розробники зазвичай виділяють чотири основні моделі гравців, які особливо актуальні для ігрового AI:

- Розвивайте розумних і людиноподібних NPC, щоб краще взаємодіяти

з гравцями;

- Прогнозуйте поведінку гравців, що призведе до поліпшення ігрового тестування та ігрового дизайну;
- Класифікуйте їх поведінку, щоб забезпечити персоналізацію гри;
- Виявлення частих шаблонів або послідовностей дій для визначення того, як гравець діє у грі.

Штучний інтелект стає все більш популярним і широко використовується в ігровій індустрії. Типові характеристики ігор роблять розробку ігор ідеальним місцем для практики і інтеграції методів штучного інтелекту, особливо глибокого навчання і навчання з підкріпленням. Більшість ігор добре відомі; в них відносно легко генерувати і використовувати дані, а стани/дії/нагороди відносно ясні. [12]

1.4 Постановка задачі

Метою роботи є розробка та програмна реалізація ігрової системи, що підходить для гри нон-геймерам та дозволяє змагатися зі штучним інтелектом. Для досягнення поставленої мети необхідно виконати такі завдання:

1. Розробка ігрової концепції:
    a. створення ігрової логіки та правил,
    b. проектування ігрових механік,
    c. створення дизайну гри.
2. Прототипування ігрового бота:
    a. розробка алгоритму аналізу ігрового процесу.
    b. розробка алгоритму взаємодії с ігровим середовищем.
3. Програмна реалізація ігрової системи:
    a. вибір програмного середовища,
    b. імплементація ігрової механіки,
    c. програмна реалізація алгоритмів штучного інтелекту,
    d. реалізація дизайну ігрового середовища,
4. Тестування ігрової системи.

2 ПРОЕКТУВАННЯ ІГРОВОЇ СИСТЕМИ

2.1 Інформаційна модель системи

Гра представляє собою змагання проти штучного інтелекту. Основний геймплей гри – це стрільба різнокольоровими шарами в такі ж різнокольорові шари, що рухаються уздовж певного маршруту. Змагання полягає в тому, що одночасно з вами також грає штучний інтелект, що має такі ж цілі, як і ви. Хто набере найбільшу кількість балів за певний час – той і виграв.

Для опису ігрової системи побудуємо діаграму ігрової логіки. Вона описує основні ігрові взаємодії, що відбуваються в ігровій системі. Так, наприклад, всю логіку можна поділити на 3 основні частини:

- Логіка ігрового меню
- Геймплейна логіка
- Логіка ігрового інтерфейсу

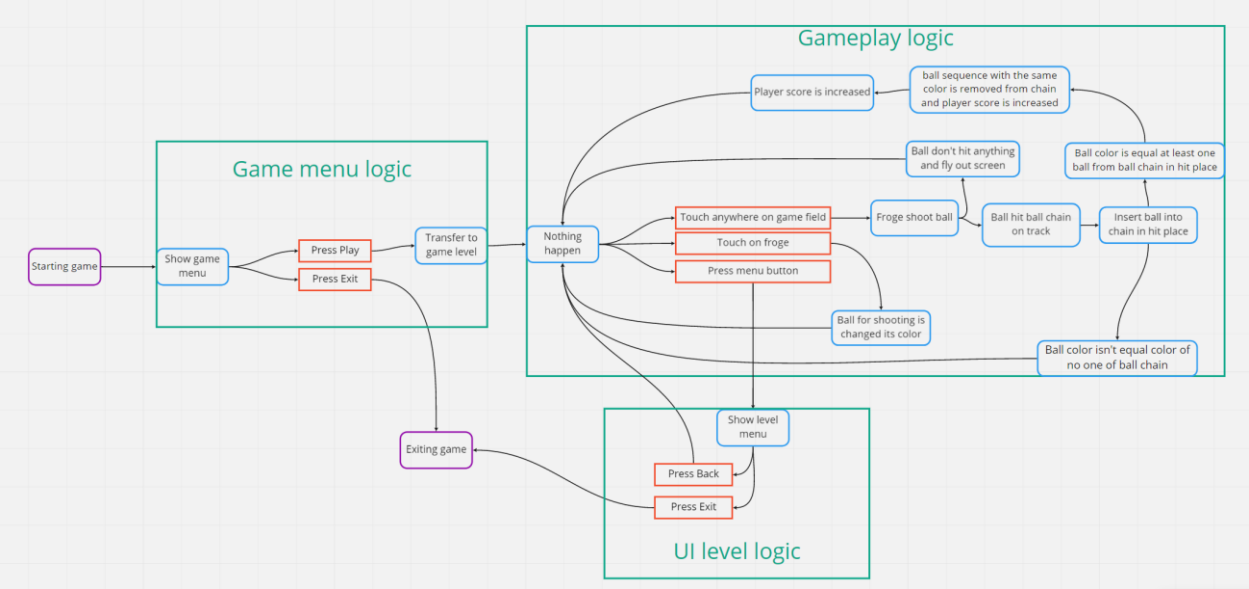

Рисунок 2.1 – Діаграма ігрової логіки

На рисунку 2.1 представлена діаграма як працюватиме логіка ігрової системи. Гравець після входу в ігрове меню може запустити ігровий рівень, або вийти з неї. На ігровому рівні, гравець може змагатися зі штучним інтелектом шляхом описаним на діаграмі. В будь-який момент, гравець може

призупинити гру, визвавши ігровий інтерфейс, де може або вийти з гри, або продовжити її.

2.2 Проектування ігрового середовища

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.

Найскладнішим та найважливішим аспектом ігрового середовища являється шлях, по якому рухаються шари. Для його побудови, ми реалізуємо всередині Unity модуль, що буде дозволяти створювати шлях прямо в редакторі та імпортувати його в гру.

Принцип побудови шляху оснований на кривих Без’є третього порядку. Криві Без’є – це спосіб визначення кривої по опорних точках. Для наочності можна розглядати їх як графік пересування точки від початку до кінця маршруту в залежності від часу руху.

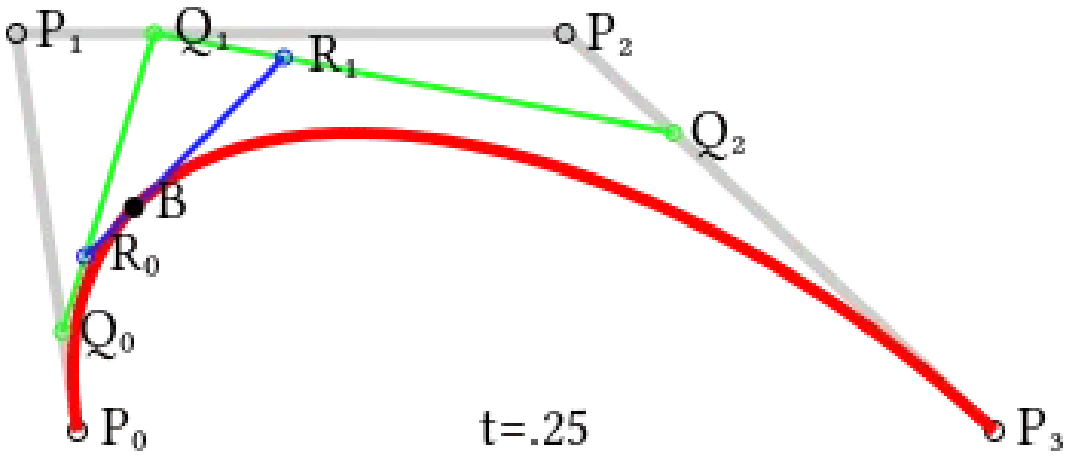

Рисунок 2.2 – Побудова кривої Без’є третього порядку

Криві Без'є третього порядку створюються певними важелями, які виходять з кожної опорної точки кривої. На кінцях таких важелей існує точка управління, що витягує формує переходи кривої подібно магніту.

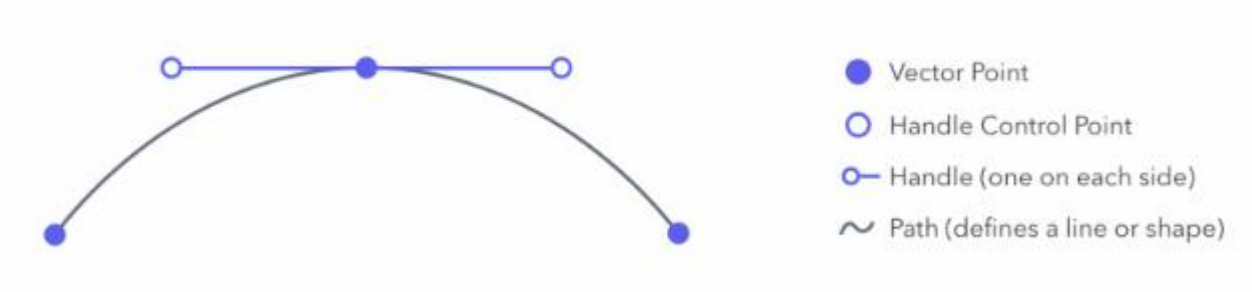

Рисунок 2.3 – Крива Без’є з важелем

Існує декілька видів керування важелями в кривих Без’є:

- "**Mirrored**" (дзеркальний спосіб) є найпростішим способом управління кривою Без’є. В даному підході використовується два важеля, які рівновіддалені від опорної точки і знаходяться під однаковим кутом.
- "**Asymmetric**" (асиметричний) спосіб схожий на Mirrored, так як ручки управління також знаходяться під одним кутом, але края ручки віддалені від опорної точки на різну відстань.
- "**Disconnected**" (розімкнутий спосіб) дозволяє управляти кожною ручкою незалежно від іншої.

Рисунок 2.4 – Візуальне представлення дзеркального, асиметричного та розімкнутого способів управління

Криві Без’є використовуються в багатьох аспектах IT сфери. Вони використовуються в описанні деяких шрифтів, графічних форматів, в 3D графіці та векторній 2D графіці, криві знайшли спосіб використання навіть в CSS – для опису плавності анімації. Але ми, в свою чергу, знайшли нове використання, хоча і досить специфічне, для кривих Без’є.

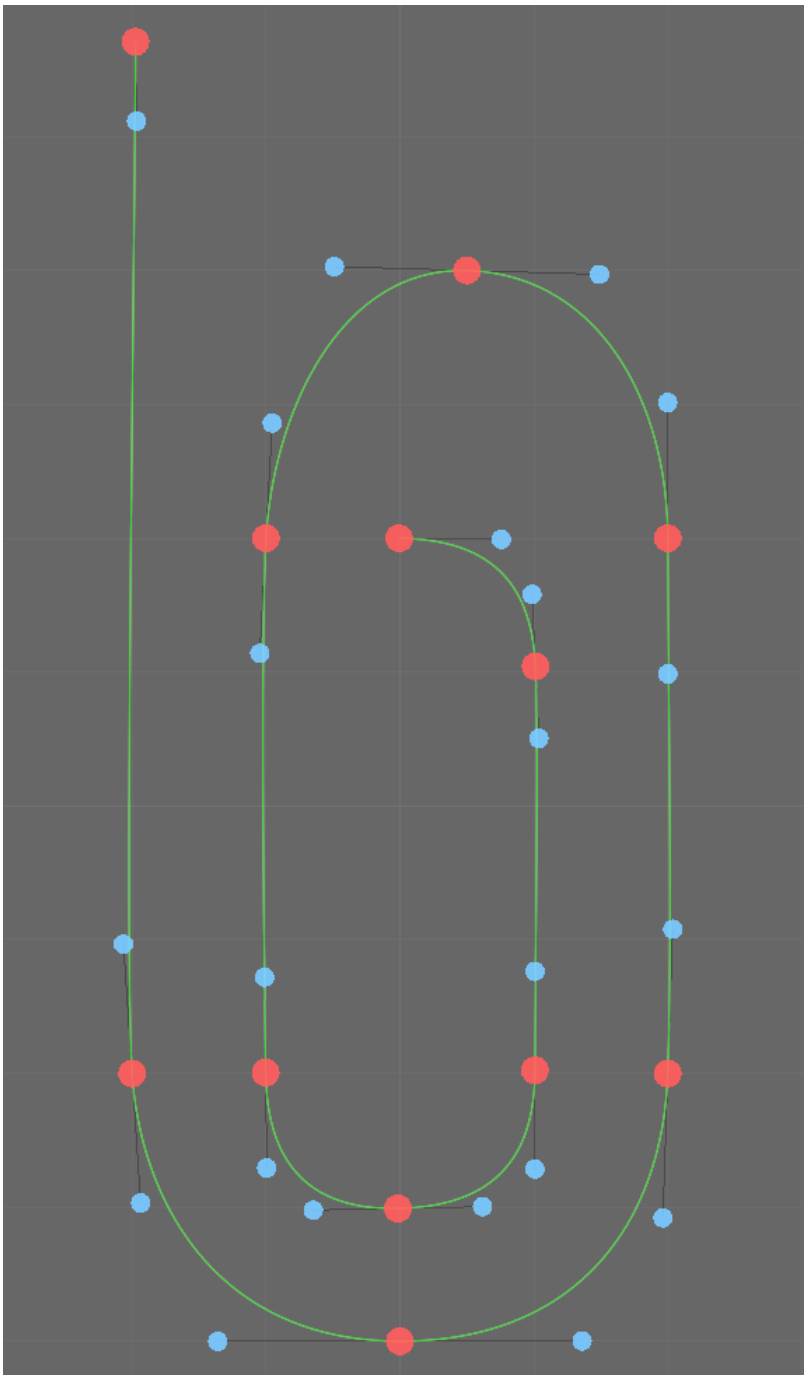

Рисунок 2.5 – Приклад шляху для нашої гри, побудований за допомогою кривих Без’є

2.3Проектування ігрового бота

Штучний інтелект – дуже важливий елемент майже будь-якої одиночної гри. Всі персонажі, керовані комп'ютером, задають атмосферу і роблять максимум, щоб користувач відчув себе частиною їхнього світу. На самому базовому рівні "штучний інтелект" складається з емуляції поведінки інших гравців або сутностей (тобто всіх елементів гри, які можуть діяти або на які можна впливати-від гравців до ракет і датчиків здоров'я), які вони представляють. Ключова концепція полягає в тому, що поведінка моделюється. Іншими словами, AI для ігор є більш "штучним “і менш "інтелектуальним". Ця система може бути простою, як система, заснована на правилах, або складною, як система, призначена для імітації поведінки людини, або прийняття нестандартних рішень.[13]

Існує декілька загальних підходів для реалізації штучного інтелекту в іграх.

- **Дерево рішень** – це система, в якій рішення збудовані у формі дерева і алгоритм повинен обходити його, щоб досягти «листа», що містить остаточне рішення. Кожну частину дерева рішень зазвичай називають "вузлом", тому що для опису подібних структур використовується теорія графів. Кожен вузол може бути одного з двох типів: *вузли рішень* (вибір з двох альтернатив на підставі перевірки якоїсь умови. Кожна альтернатива представлена у вигляді власного вузла), *кінцеві вузли* (виконувана дія, що представляє собою остаточне рішення, прийняте деревом)
- **Кінцеві автомати** – спосіб моделювання і реалізації об'єкта, що володіє різними станами протягом свого життя. Кожен "стан" може представляти фізичні умови, в яких знаходиться об'єкт, набір поведінкових моделей, що вписується в контекст гри.
- **Системи на основі корисності (Utility)** – це такі системи, в яких в розпорядженні агента є безліч дій, і він вибирає виконання

одного на підставі відносної корисності кожної дії. Корисність тут – це довільна міра важливості або бажаності для агента виконання цієї дії.

В нашому ігровому середовищі буде реалізований ігровий бот-суперник на основі підходу кінцевих автоматів. Цей підхід дозволяє дуже чітко простежити та встановити зміну станів штучного інтелекту, що нам і потрібно, а його головним мінусом є складність реалізації для складних та громіздких рішень, що нам не загрожує, так як ми розробляємо гіперказуальну гру, де весь гемплей є досить примітивним, і штучний інтелект має відповідати цій концепції.

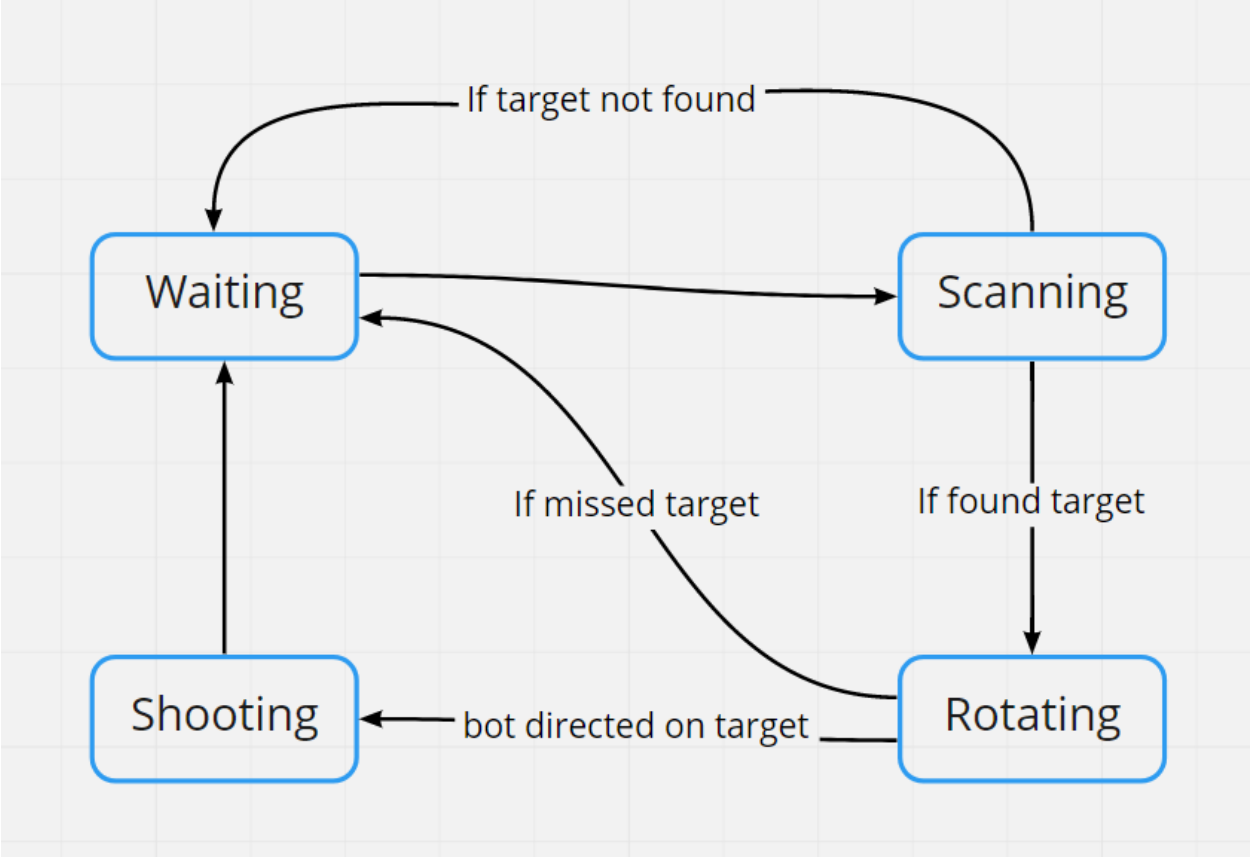

Рисунок 2.6 – Схема штучний інтелекту бота в моєму проекті

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІГРОВОЇ СИСТЕМИ

3.1 Вибір та конфігурування програмного середовища

Для розробки даного ігрового середовища був обраний ігровий двигун Unity, через його легкість 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 – він має гарну інтеграцію з Unity.

В нашому випадку, ми маємо головну сцену всередині Unity (для різних проектів, їх може бути різна кількість, для нашого проекту достатньо одної). 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 визовів на різні випадки.

Для нашої гри ми 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 представлений зовнішній вигляд редактора Unity.

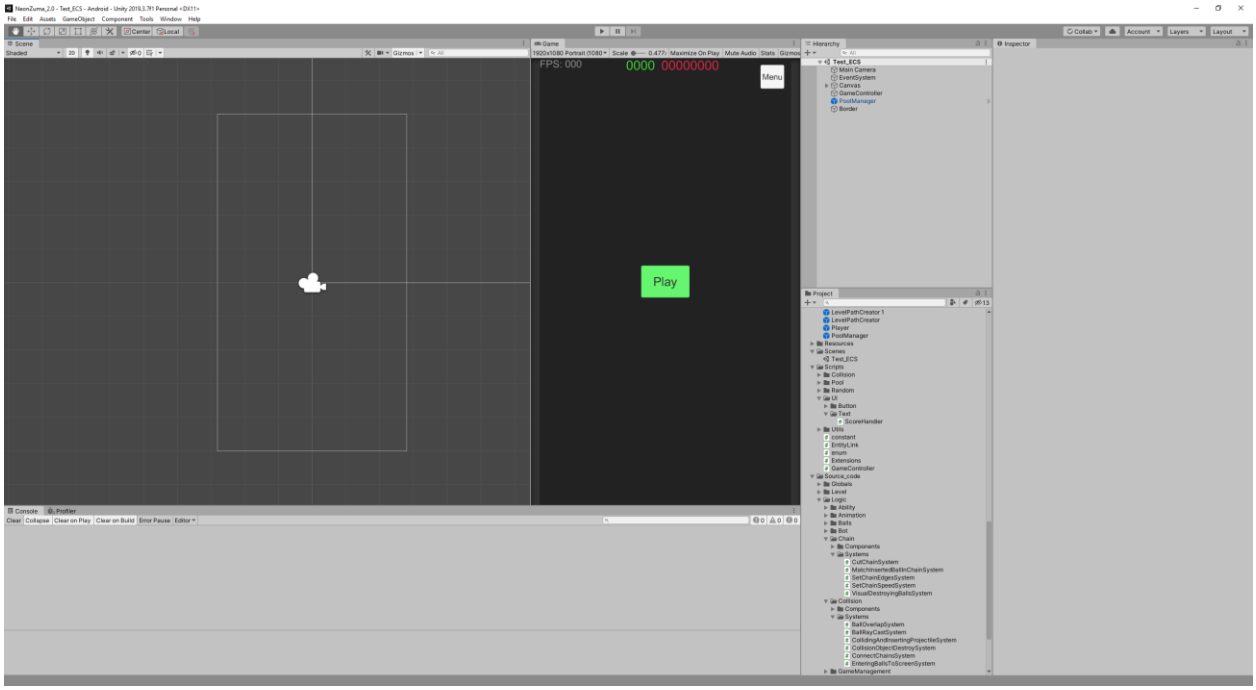

Рисунок 3.1 – Редактор Unity

На рисунку 3.2 представлений зовнішній вигляд середовища розробки Visual Studio 2017,.який був використаний для написання коду

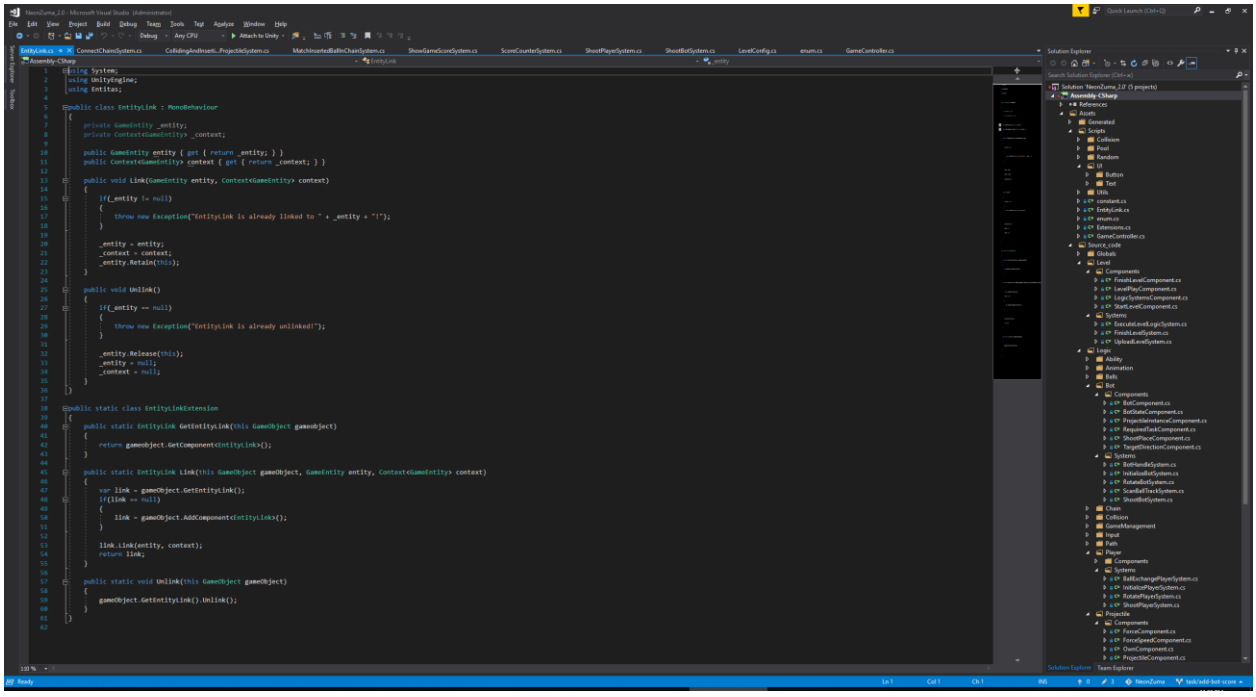

Рисунок 3.2 – Середовище розробки Visual Studio 2017

3.2 Короткий 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
ExecuteLevelLogic	Execute	Логіка виконання систем, що являє собою ігрову логіку рівня.

Таблиця 3.2 – Опис реактивних систем контексту Level

Reactive System	Types	Entity	Triggers	Description
FinishLevel	✕	Manage	FinishLevel	Логіка закінчення ігрового рівня та його розвантаження, а також виклик усіх подій, підписаних на закінчення рівня
UploadLevel	✕	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
FinishLevel	Manage	☑	☑	🏳	Флаг о том, що рівень завершено
LevelPlay	Manage	☑	☑	🏳	Флаг того, що логіка ігрового рівня виконується, якщо вимкнути – то виконання ігрових систем призупиниться
LogicSystems	Manage	☑		Systems	Компонент зберігає системи, що відносяться до ігрового рівня
StartLevel	Manage	☑	☑	🏳	Флаг о том, що рівень почався

Таблиця 3.4 – Опис загальних систем контексту Animation

Reactive System	Types	Entity	Triggers	Description
FinishMoveAnimation	TearDown	Game	AnimationDone	Логіка обробки закінчення анімації та виклик пост подій
MoveAnimationControl	TearDown	Game	MoveAnimation	Логіка запуска та переривання анімації руху об’єктів
ScaleAnimationControl	TearDown	Game	ScaleAnimation	Логіка запуска та переривання анімації масштабування об’єктів

Таблиця 3.5– Опис компонентів контексту Animation

Component	Contexts	Unique	Event	Fields	Description
MoveAnimation	Game			float Vector3 Action	Дані для запуска анімації руху об’єкта
AnimationDone	Game			🏳	Флаг о том, що анімація закінчена
AnimationInfo	Game			List	Лист подій, що повинні бути оброблені по завершенню анімації
ScaleAnimation	Game			float float Action	Дані для запуска анімації масштабування об’єкта

Таблиця 3.6 – Опис загальних систем контексту Balls

System	Types	Description
CheckAndSpawnBall	• Execute • Initialize • TearDown	Логіка перевірки можливості появлення нового шару, а також сама логіка створення
UpdateBallDistanceBySpeed	• Execute • Initialize • TearDown	Логіка оновлення позиції шарів відносно швидкості, тобто рух по треку

Таблиця 3.7 – Опис реактивних систем контексту Balls

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

Таблиця 3.8 – Опис компонентів контексту Balls

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

Частину програмного коду наведено в додатку. Весь код проекту можна знайти на даному репозиторії - https://github.com/Markmax2304/NeonZuma (даний проект являє собою open source)

3.3 Тестування програмного забезпечення

Для тестування ігрової системи було запущено декілька ігрових сесій, ціль яких виявити якомога більше багів та помилок розробки. На рисунку 3.3 подано вид ігрового меню після запуску додатку.

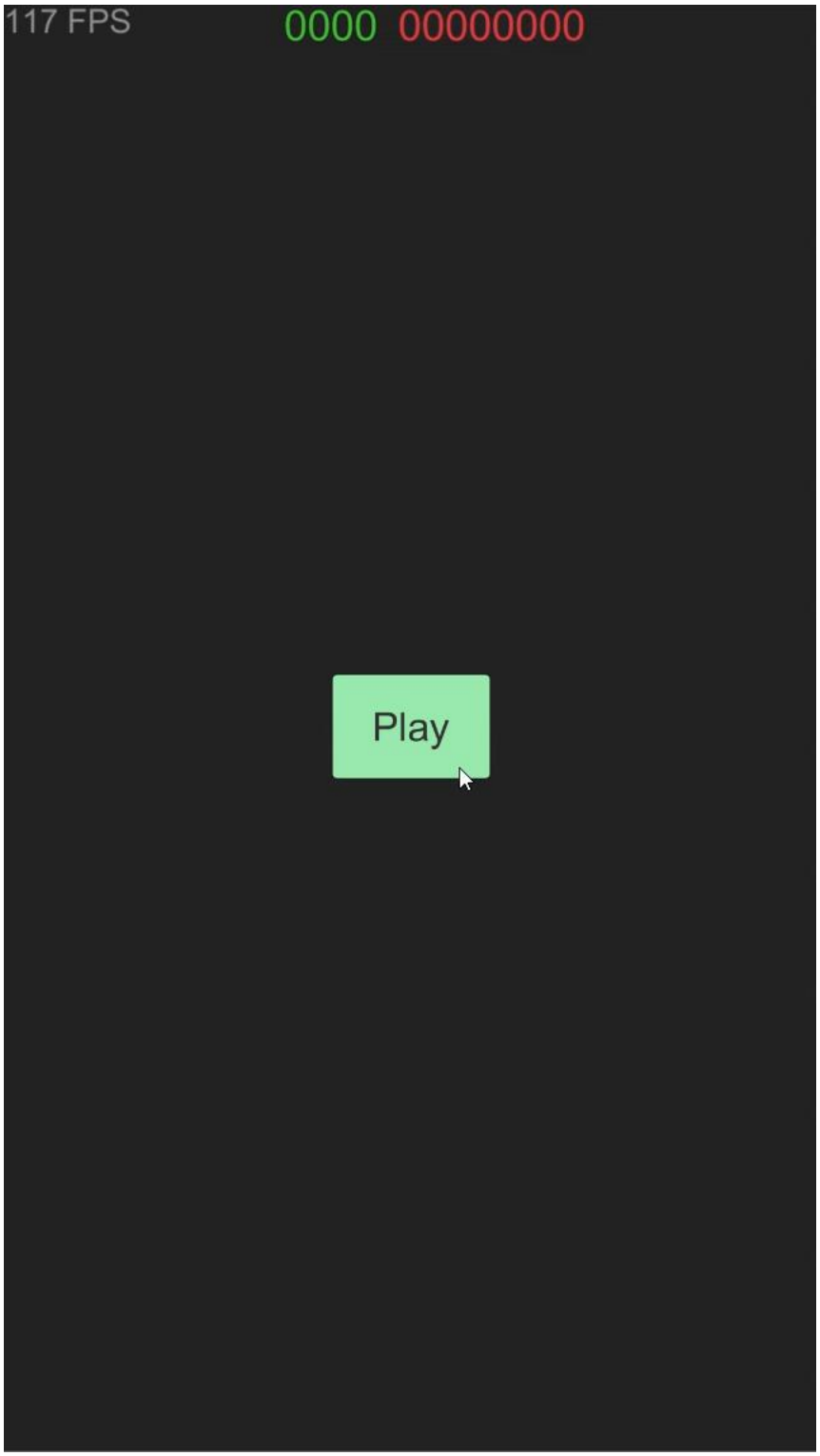

Рисунок 3.3 – Екран гри на початку гри.

Нажимаємо кнопку Play та починаємо ігрову сесію(рисунок 3.4).

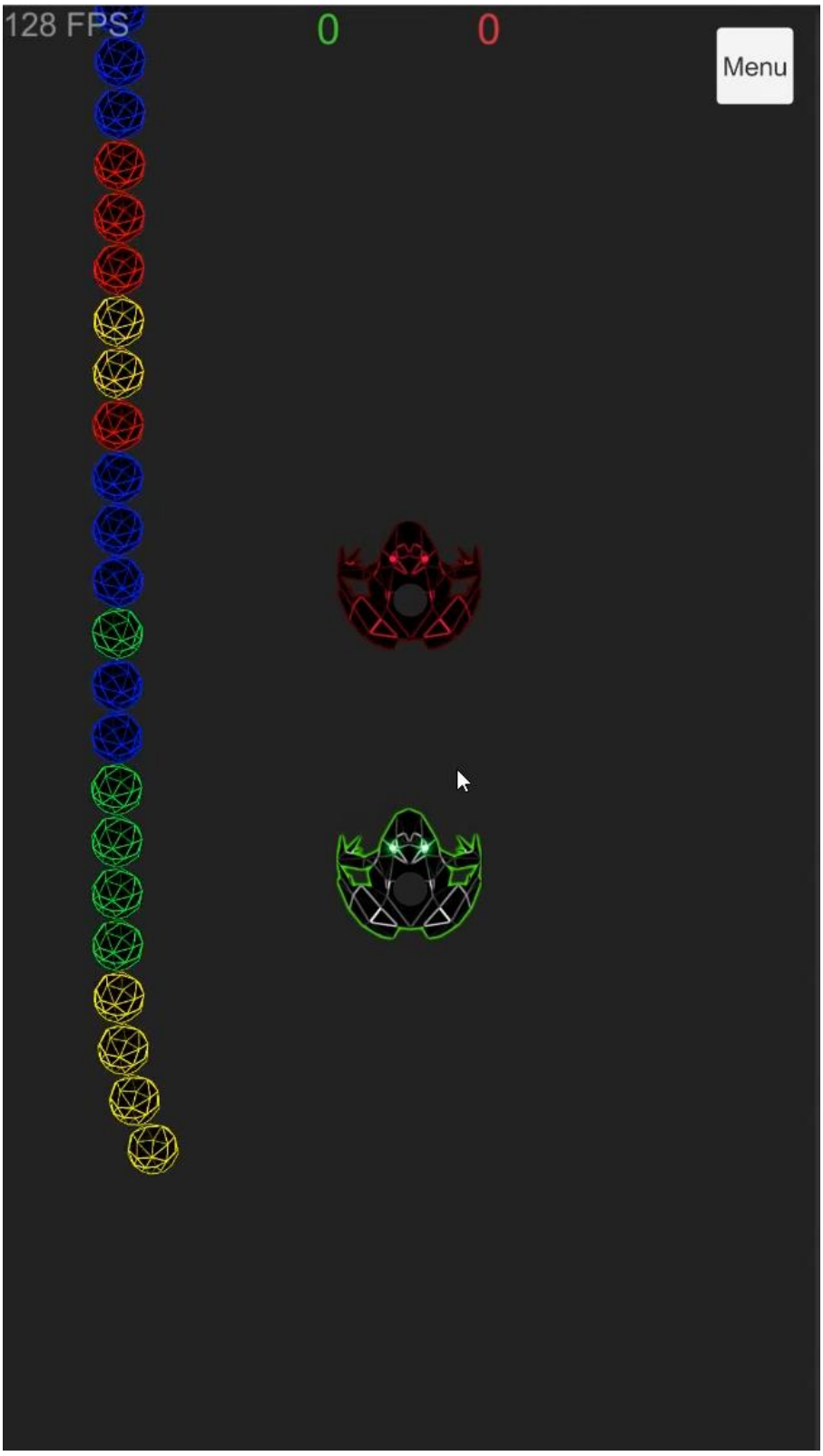

Рисунок 3.4 – Початок ігрового рівня.

Бот під управлінням штучного інтелекту одразу починає діяти і першим пострілом досягає поставленої задачі (рисунок 3.5).

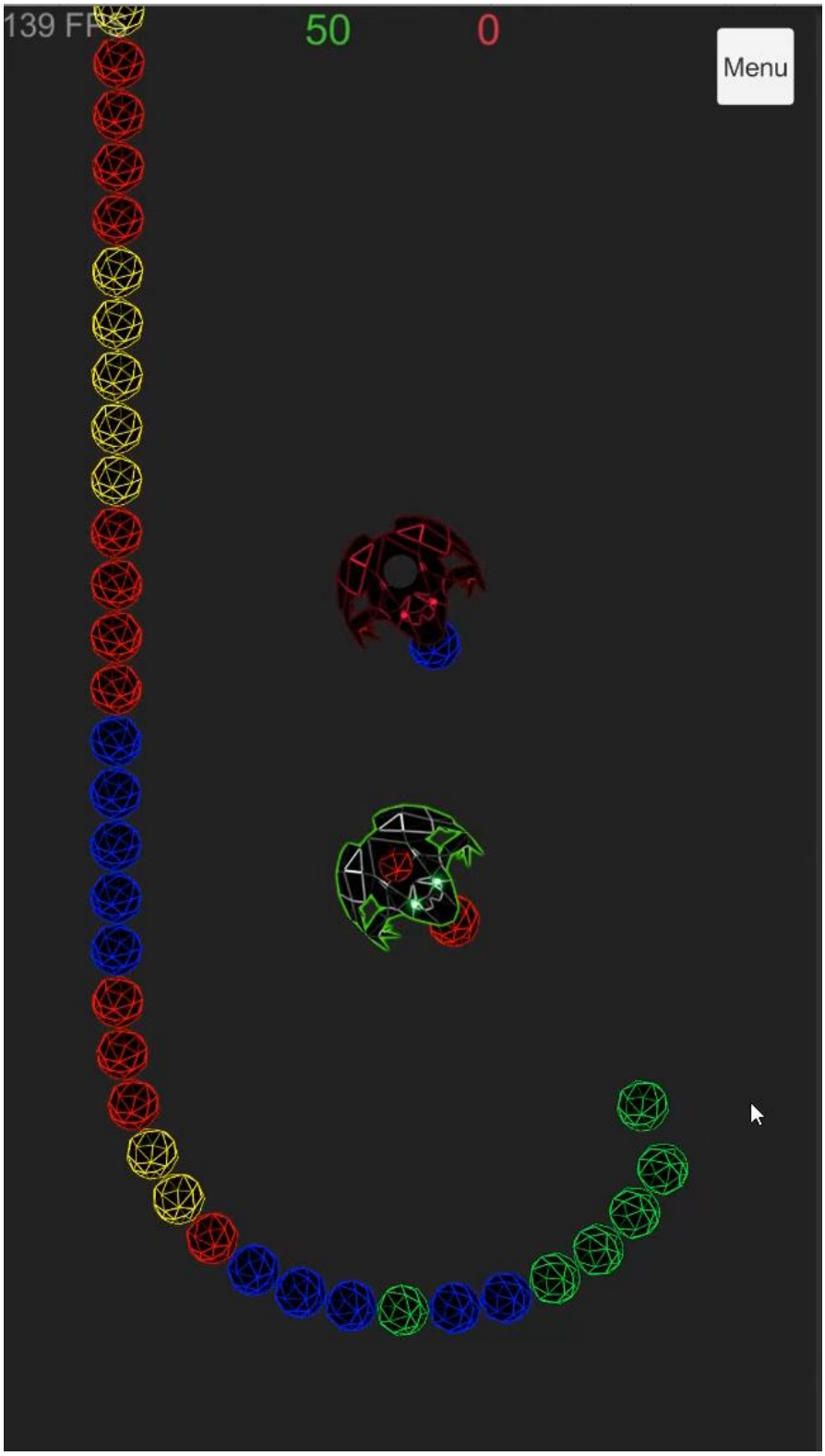

Рисунок 3.5 – Перші дії боту.

Після деякого часу гри ми можемо бачити рахунок як гравця, так і бота в верху ігрового екрану (зеленим – гравець, червоним – бот) (рисунок 3.6).

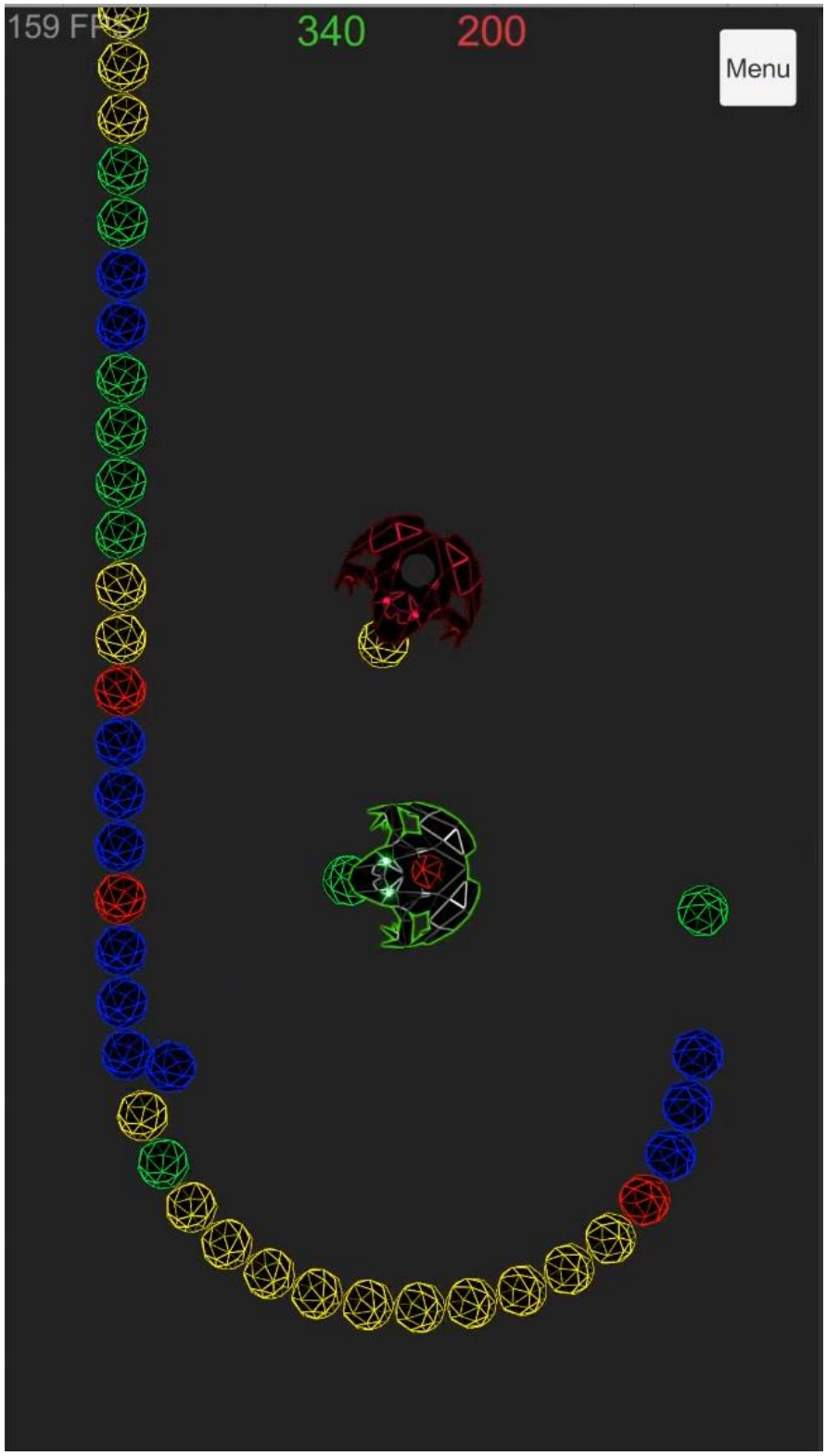

Рисунок 3.6 – Ігровий процес через деякий час гри

Ігровий процес продовжується до тих пір, поки шари не досягнуть кінцевої точки шляху та гра не буде завершена(на даному етапі, ще немає підтримки

інших режимів гри – але планується в подальшій розробці). Завершення гри можна бачити на рисунку 3.7 та 3.8

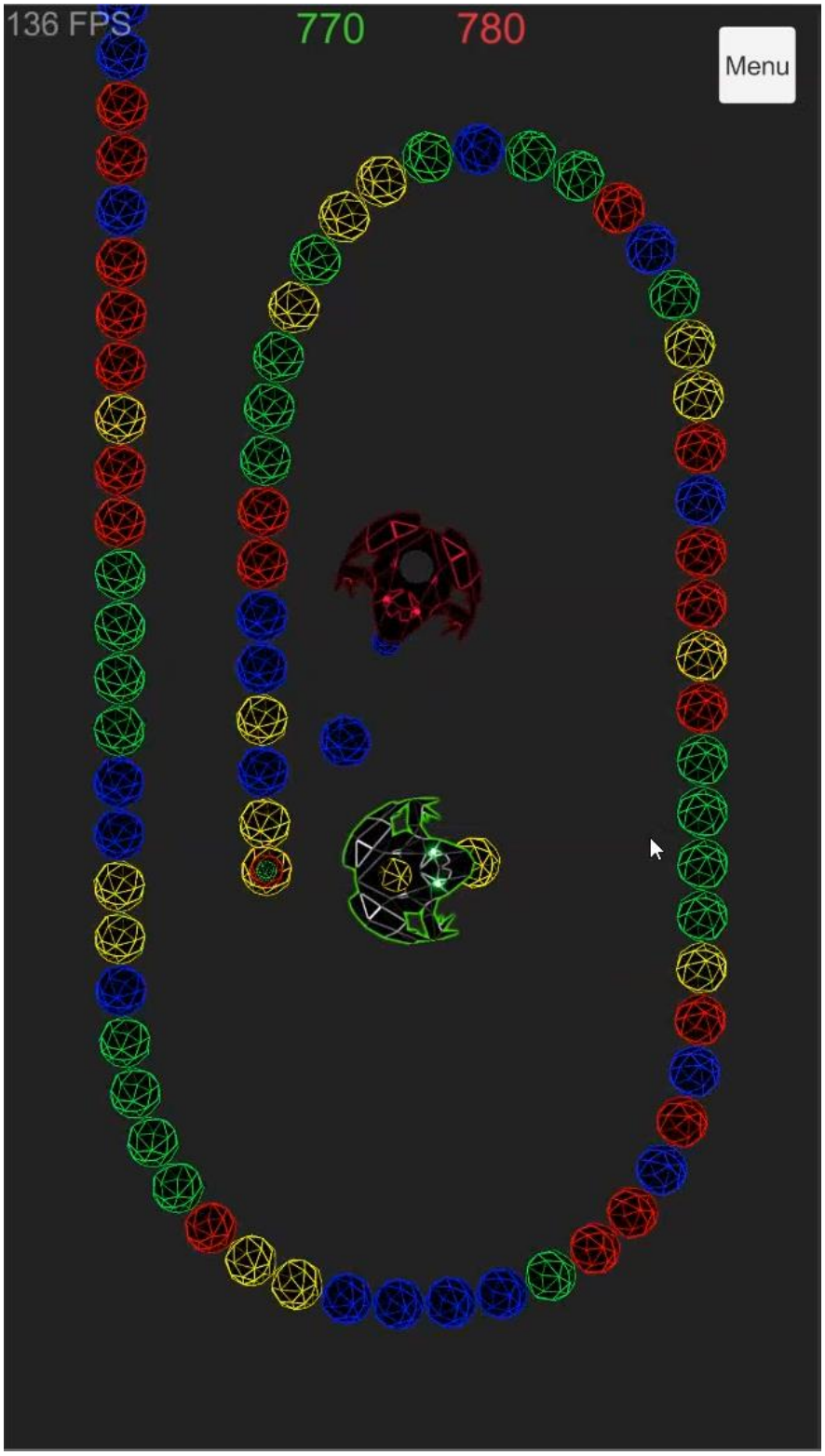

Рисунок 3.7 – Початок кінця гри та знищення усіх шарів

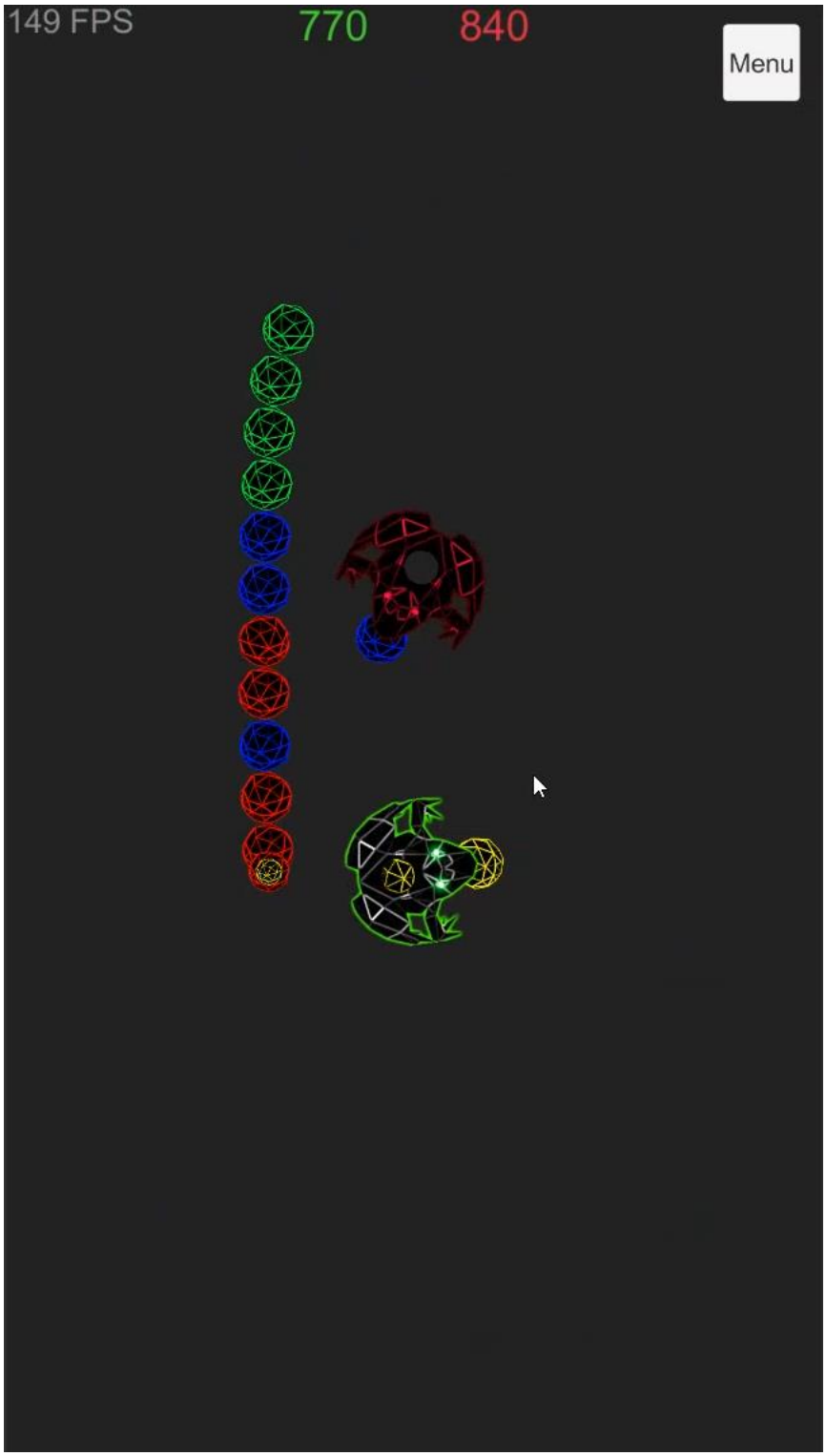

Рисунок 3.8 – Закінчення знищення ігрових шарів

Після програшу гравець може порівняти свої результати та почати нову ігрову сесію.

ВИСНОВКИ

В випускній роботі було виконано розробку інформаційного та програмного забезпечення ігрової системи. В роботі проведено аналіз найбільш популярних ігрових рушіїв, були описані їх слабкі та сильні сторони. На основі проведеного аналізу було розроблено ігрову концепцію, створено ігрову логіку та правила, спроектовано ігрові механіки та прототип дизайну гри. Для генерування ігрового середовища і оцінки ігрового процесу запропоновано використати криві Без’є для генерування шляху. Для програмної реалізації ігрової системи використовувалося середовище розробки Unity, 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 закрите бета-тестування.

СПИСОК ЛІТЕРАТУРИ

1. Комп’ютерні ігри – Режим доступу: https://www.i-igrushki.ru/igrushkapedia/kompyuternye-igry.html
2. Гіперказуальне направлення ігор – Режим доступу: https://hsbi.hse.ru/articles/giper-kazualnoe-napravlenie-igr/
3. Hyper-Casual Games: Mobile Gamings Greatest Genre – Режим доступа: https://clevertap.com/blog/hyper-casual-games/
4. Глубинні цінності індустрії гіперказуальних ігор – Режим доступу: http://dailytelefrag.ru/articles/read.php?id=45312
5. Jason Gregory: Game Engine Architecture – CRC Press, 2016 р. – 864с.
6. Движок Unity – особливості, переваги і недоліки – Режим доступу: https://cubiq.ru/dvizhok-unity/
7. Хокинг Джозеф: Unity – в действии. Мультиплатформенная разработка на C# - СПб: Питер, 2016. – 336 с.
8. Движок Unreal Engine – Режим доступу: https://www.rendertimes.ru/dvizhok-unreal-engine/
9. Godot documentation – Режим доступу: https://docs.godotengine.org/en/stable/about/introduction.html
10. История развития ИИ в играх: эволюция, алгоритмы, хардкор – Режим доступу: https://stopgame.ru/blogs/topic/93248
11. Лучший искусственный интеллект в играх, или Почему ИИ — это подделка – Режим доступу: https://www.igromania.ru/article/29712/Luchshiy_iskusstvennyy_intellekt_v_igrah_ili_Pochemu_II-yeto_poddelka.html
12. Artificial Intelligence in Games – Режим доступу: https://medium.com/aifrontiers/an-overview-of-artificial-intelligence-for-video-games-f491229c0e7d

13. Designing Artificial Intelligence for Games – Режим доступу: https://software.intel.com/content/www/us/en/develop/articles/designing-artificial-intelligence-for-games-part-1.html

ДОДАТОК

Для огляду представлен клас GameController який являє собою головний клас проекту та вхідну точку виконання програмного продукту. Він реалізує логіку конфігурування усієї логіки проекту.

```
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\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 execution 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))


            //RayCasting
            .Add(new BallRayCastSystem(contexts))
            //Overlaping
            .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))
                .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))

                //Ability
                .Add(new InputAbilitySystem(contexts))
                .Add(new InvokingAbilitySystem(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))
                ;
        }
}
```