Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра програмних систем і технологій

УДК 004.925

На правах рукопису

БР.ІПЗ - 31.00.00.00

ВИПУСКНА КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Тема Розробка комп'ютерної гри у жанрі "Аркада"

Спеціальність - 121 “Інженерія програмного забезпечення”

Виконав студент

ІПЗ-43_____/Данило КУШЛЯНСЬКИЙ

(шифр групи)(підпис)(дата)(розшифровка підпису)

Науковий керівник

к.ф.-м.н._____/Сергій ПОЛЯКОВ

(посада) (підпис) (дата) (розшифровка підпису) s

Консультант з питань нормоконтролю

фахівець._____/Тамара ЧАПОВСЬКА

Допускається до захисту

з питань нормоконтролю

Завідувач кафедри

д.т.н., проф._____/Олексій БИЧКОВ

(посада) (підпис) (дата) (розшифровка підпису)

Київ – 2021

Рішенням Екзаменаційної комісії

випускна кваліфікаційна робота студента

_____

захищена з оцінкою

_____

Голова екзаменаційної комісії

д.т.н., проф., Андрій БОНДАРЧУК

Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра програмних систем і технологій

Освітньо-кваліфікаційний рівень бакалавр

Спеціальність 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ:

Завідувач кафедри програмних систем і технологій

_____ (Олексій БИЧКОВ)

„____”_____20___р.

ЗАВДАННЯ

НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

Кушлянському Данилу Олексійовичу

(прізвище, ім’я, по батькові)

1. Тема випускної кваліфікаційної бакалаврської роботи «Розробка комп'ютерної гри у жанрі "Аркада"»

керівник проекту (роботи): Поляков Сергій Анатолійович, к.ф.м.н.

затверджена наказом вищого навчального закладу від 11 листопада 2020р. № 6

2. Строк здачі студентом закінченої роботи_____

3. Вихідні дані до роботи (проекту) Середовище розробки програмного забезпечення Visual Studio 2019 та Unity, теоретичні концепції, математичні засади роботи ігрових компонентів, результати експериментальних досліджень, наукові статті, офіційна документація.

4. Зміст пояснювальної записки (перелік питань, що їх належить розробити)

1.1 Опис концепції та ідеї гри. 1.1.1 Жанр та назва. 1.1.2 Ідея гри 1.2 Аналіз предметної області 1.2.1 Статистика індустрії 1.2.2 Розподіл долі ринку мобільних ігор 1.3 Огляд інструментів розробки 2.1 Варіації трансформацій ігрових об'єктів 2.1.1 Визначення руху та напрямку 2.1.2 Повороти 2.2 Визначення колізій 3.1 Дизайн архітектури 3.2 Скрипт-менеджер 3.3 Івент-функції 3.4 Оптимізація 3.4.1 Coroutines 3.4.2 Object pooling 3.5 Дизайн-документ компонентів гри 3.5.1 Камера, персонаж, управління 3.5.2 Ігровий світ 3.5.3 UI

5. Перелік графічного матеріалу (з точним забезпеченням обов'язкових креслень)

1. Зображення логотипу гри (рис. 1.1, ст.)
2. Зображення зовнішнього вигляду гри (рис. 1.2, ст.)
3. Гістограма глобального ігрового ринку (рис. 1.3, ст.)
4. Координати об'єкта в просторі (рис. 2.1, ст.)
5. Додавання двох векторів (рис. 2.2, ст)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Постановка задачі та технічне завдання	Поляков Сергій Анатолійович		
Математичний опис задачі	Поляков Сергій Анатолійович		
Імплементація технічних рішень та компонентів гри	Поляков Сергій Анатолійович		

7. Дата видачі завдання _____

Керівник_____ (підпис)

Завдання прийняв до виконання _____ (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	**Назви етапів бакалаврської роботи**	**Термін виконання етапів роботи**	**Примітка**
1	Підбір і вивчення літератури	29.11.2020	виконано
2	Аналіз концепцій і паттернів розробки ПЗ	31.12.2020	виконано
3	Вивчення середовища розробки Unity та його можливостей	15.01.2021	виконано
4	Розробка робочого прототипу	1.02.2021	виконано
5	Імплементація кінцевих технічних рішень проекту	1.03.2021	виконано
6	Написання пояснювальної записки	1.05.2021	виконано
7	Затвердження пояснювальної записки	1.06.2021	виконано

Студент – бакалавр _____(підпис)

Керівник роботи _____ (підпис)

АНОТАЦІЯ

Випускна кваліфікаційна бакалаврська робота: 83 с., 34 рис., 2 табл., 4 додат., 10 джерел.

Тема: Розробка комп'ютерної гри у жанрі «Аркада».

Об’єкт дослідження: процес розробки комп’ютерної гри під мобільну операційну систему Android.

Мета роботи: Розробка та створення програмного продукту у вигляді гри, аналіз шаблонів проектування, опис математичних та програмних методів створення ігрових компонентів, розгляд алгоритмів оптимізації програмного забезпечення з використанням різних структур даних.

Предмет дослідження: технології візуалізації 3D графіки, ігровий двигун Unity, математичні засади роботи ігрових сутностей, побудова та проектування архітектури застосунку, методи оптимізації програмного забезпечення.

Результати дослідження: досліджено можливості застосування ігрових двигунів та вплив вибору підходу до розробки на ігрові механіки. Було розглянуто теоретичні та математичні засади, які є основою роботи з ігровими компонентами, а також застосовано декілька підходів оптимізації програмного забезпечення.

Висновок: в результаті досліджень було отримано мобільний програмний застосунок, розроблений на двигуні Unity. Було обґрунтовано прийняті архітектурних рішення. Описано принципи роботи кожного ігрового компоненту та знайдено оптимальні шляхи реалізації ідей та концепцій.

3D ГРАФІКА, ІГРОВИЙ ДВИГУН UNITY, ANDROID, АРКАДА, ІГРОВИЙ КОМПОНЕНТ

ANNOTATION

Final qualifying bachelor's work: 83 pages, 34 figures, 2 tables, 4 appendices, 10 sources.

Subject: Development of a computer game in the genre of "Arcade".

Object of research: the process of developing a computer game for the Android mobile operating system.

Goal of the work: Development and creation of a software product in the form of a game, analysis of design templates, description of mathematical and software methods of creating game components, consideration of software optimization algorithms using different data structures.

Subject of research: 3D graphics visualization technologies, Unity game engine, mathematical principles of game essences, construction and design of application architecture, software optimization methods.

Research results: the possibilities of application of game engines and the influence of the choice of the approach to development on game mechanics are investigated. Theoretical and mathematical principles that are the basis of working with game components were considered, as well as several approaches to software optimization were applied.

Conclusion: The result of the research is a mobile software application developed using the Unity engine. Architectural decisions were explained. The principles of operation of each game component were described and the optimal ways of realization of ideas and concepts were found.

3D GRAPHICS, UNITY GAME ENGINE, ANDROID, ARCADE, GAME COMPONENT

АННОТАЦИЯ

Выпускная квалификационная бакалаврская работа: 83., 34 рис., 2 табл., 4 доп., 10 источников.

Тема: Разработка компьютерной игры в жанре «Аркада».

Объект исследования: процесс разработки компьютерной игры под мобильную операционную систему Android.

Цель работы: разработка и создание программного продукта в виде игры, анализ шаблонов проектирования, описание математических и программных методов создания игровых компонентов, рассмотрение алгоритмов оптимизации программного обеспечения с использованием различных структур данных.

Предмет исследования: технологии визуализации 3D графики, игровой двигатель Unity, математические основы работы игровых сущностей, построение и проектирование архитектуры приложения, методы оптимизации программного обеспечения.

Результаты исследования: исследованы возможности применения игровых двигателей и влияние выбора подхода к разработке на игровые механики. Были рассмотрены теоретические и математические принципы, которые являются основой работы с игровыми компонентами, а также применено несколько подходов оптимизации программного обеспечения.

Вывод: в результате исследований было получено мобильное приложение, разработанное на движке Unity. Было описано принятые архитектурных решения. Описаны принципы работы каждого игрового компонента и найдены оптимальные пути реализации идей и концепций.

3D ГРАФИКА, ДВИГАТЕЛЬ UNITY, ANDROID, АРКАДА, ИГРОВОЙ КОМПОНЕНТ

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПК – персональний комп’ютер.

ААА – (Triple A) – одна з категорій відеоігор, які створюються середніми і великими видавництвами, що, як правило, мають великі бюджети на розробку та маркетинг.

СК – система координат.

ЛСК – локальна система координат.

ПЗ – Програмне забезпечення.

Сеттінг – обстановка або антураж, які створюються різними елементами та визначає світ, в якому відбуваються події гри.

Геймплей – процес взаємодії гравця з механіками гри.

Тайм-кіллер – різновид ігор, які використовуються для того, щоб скоротити час.

One-handed – ігровий досвід, який отримує гравець, використовуючи тільки одну руку.

Казуальна комп’ютерна гра – гра, призначена для широкої аудиторії. Відрізняється простими правилами та не вимагають від гравця особливої посидючості, затрат часу на вивчення або отримання яких-небудь навичок.

Префаб – особливий тип асетів, який дозволяє зберігати весь ігровий об’єкт з усіма компонентами і значеннями властивостей. Виступають в ролі шаблона для створення екземплярів об’єктів на сцені.

Фреймрейт – термін, що є одиницею виміру кадрів в секунду (frames per second).

Спрайт – двовимірне растрове зображення, що може вільно переміщуватися по екрану.

ВСТУП

Актуальність роботи

Актуальність теми дослідження обумовлена новизною ракурсу вивчення проблем гейм-дизайну. Ігровий ринок зростає швидкими темпами з кожним роком та акумулює безліч людей, які готові виділити декілька годин життя на день аркадним іграм. В сучасному світі майже кожен має в кишені смартфон, який може в будь-яку хвилину допомогти скоротити час за допомогою гри. Не дивлячись на те, що ці ігри не надають користувачу складних ігрових механік або складних сюжетних ліній, вони все ж користуються великим попитом та займають провідні позиції в чартах основних платформ цифрової дистрибуції. Простота ігрового процесу робить ігри даного жанру доступними для широкого кола гравців, незалежно від віку та ігрових навиків. Тому вплив аркадного напряму на розвиток індустрії комп’ютерних ігор ж дуже великим.

Зв'язок роботи з науковими програмами, планами, темами

Тема кваліфікаційної роботи відповідає науковому напряму кафедри програмних систем і технологій Факультету Інформаційних технологій Київського національного університету імені Тараса Шевченка. В даній роботі було використано знання та навички отримані під час навчання таким дисциплінам як: комп’ютерна графіка та візуалізація, розробка мультимедійних та ігрових систем, програмування під мобільні платформи, додаткові розділи проектування програмного забезпечення.

Мета і задачі дослідження

Мета: відобразити результати проведених наукових досліджень, провести математичні розрахунки, обґрунтувати прийняті при виконанні роботи структурні, системні та конструкторські рішення. Пояснити принципи роботи проекту та його

окремих компонентів і програмних модулів. Задачею дослідження було емпіричне дослідження роботи з ігровими компонентами через аналіз відповідних літературних джерел та научних публікацій, побудова власного мобільного застосунку використовуючи власні напрацювання та знання, отримані в процесі вивчення роботи двигуна Unity.

Об’єкт дослідження

Об’єктом дослідження є комп’ютерні ігри, шаблони проектування програмного забезпечення, математичні засади, на яких побудована взаємодія з ігровими об’єктами, оптимізаційні техніки.

Предмет дослідження

Предметом дослідження є принципи роботи з ігровими двигунами, математичними об’єктами, робота з скрипт-менеджерами, івент-функціями та іншими архітектурними прийомами.

Методи дослідження

Під час написання програмного модулю використовувалися алгоритми оптимізації, які засновані на роботі з відповідними структурами даних. Дослідження також включало аналіз першоджерел у вигляді офіційної документації, статей. Розробка проводилася в декілька етапів: від прототипування до рефакторингу та кінцевого налаштування.

Порівняння роботи з відомими розв’язаними проблемами

Існує багато варіацій аркадних ігор, які використовують схожі механіки та алгоритм генерування світу, такі як Temple Run, Drop Stack Ball та багато інших. Кожна з яких є унікальною. Концепція нескінченного геймплею є не новою, проте реалізація і позиціонування є досить різним.

Наукова новизна отриманих результатів

Досліджено можливості застосування векторної алгебри при розробці комп’ютерних ігор. Наукова новизна дослідження полягає в тому, що мною було застосовано та розроблено способи вдосконалення та оптимізації програмного забезпечення, які є нетиповими для інших представників жанру.

Практичне значення одержаних результатів

Одержані результати можуть бути використані при вдосконаленні методів оптимізації та пошуку нових підходів в гейм-дизайні, які підштовхнуть індустрію до нових ідей та концепцій.

Особистий внесок студента

Гра є результатом особистого внеску в розвиток технологій розробки комп’ютерних ігор. Основними результатами є: імплементований підхід до перенесення математичних уявлень про простір до їх симулювання в грі, реалізація оптимізаційних технік в прикладній площині.

Публікації та впровадження результатів

Загальна кількість публікацій – 3. За результатами наукових досліджень, проведених у бакалаврській роботі, було підготовлено статтю для доповіді у 8-ій Східно-Європейській конференції Математичні та програмні технології Internet of Everything на тему: «Обджект пулінг як інструмент оптимізації в ігровій розробці», а також було опубліковано статті на теми: «Геймінг як сервіс хмарних обчислень» у 7-ій, «Застосування IoE в охороні здоров’я та медичній сфері» у 3-ій Східно-Європейській конференції Математичні та програмні технології Internet of Everything відповідно.

Структура та обсяг роботи

Робота викладена на 83 сторінках друкованого тексту, який складається із вступу, трьох розділів, висновків, списку використаних джерел (10 найменувань). Робота містить 2 таблиці, 34 рисунки та 4 додатки, обсягом 12 сторінок.

РОЗДІЛ 1
ПОСТАНОВКА ЗАДАЧІ ТА ТЕХНІЧНЕ ЗАВДАННЯ

В даному розділі буде розглянуто основні ідеї та концепції, що спонукали до створення гри та були використані при її розробці. Цілі та методи їх досянення, що були застосовані під час проектування програмного модуля.

Буде проведено аналіз області застосування відеоігор та особливості їх створення на основі мультиплатформенних інструментів та редакторів для розробки – невід’ємної частини процесу розробки будь-якої гри.

1.1 Опис концепції та ідеї гри

Першим кроком в створенні дизайну гри є створення концепції. Концепція повинна включати в себе основну інформацію про гру, причини створення саме такої гри та аналіз ринкової ситуації.

1.1.1 Жанр та назва

Назва повинна повноцінно відображати одну із складових гри: сеттінг, геймплей, механіки та інші особливості. Назвою гри було вибрано «**Roller Ball**», оскільки така назва дає чітко зрозуміти гравцю, про те, з чим він буде мати справу.

Жанр гри – аркада. Аркада - жанр комп'ютерних ігор, що характеризується коротким за часом, але інтенсивним ігровим процесом. У вузькому сенсі аркадними називаються гри для аркадних ігрових автоматів. Ігри, портовані з аркадних автоматів, також прийнято називати аркадними.

Період з кінця 1970-х до середини 1980-х був охарактеризований розквітом аркадних ігор, і його часто називають золотим сторіччям аркадних ігор. Класичні аркади характеризуються наступними властивостями:

- *Гра на одному екрані*. У класичних аркадах весь ігровий процес зосереджений на одному екрані. Перш за все це обумовлено історично, що сталося через технічні обмеження, але в той же час це значно впливало на гейм дизайн. Так, гравці в будь-який момент часу могли бачити весь ігровий світ і приймати рішення, виходячи з повної інформації про його стан.
- *Нескінченна гра.* Потенційно гравці можуть грати в аркаду безкінечну кількість часу, і відповідно, не можуть виграти. Це впливало на те, що гравці робили виклик самі собі - наскільки довго вони зможуть протриматися. Щодо гейм дизайну в аркадах, гравець ніколи не вигравав, і кожна гра закінчувалася поразкою.
- *Ігровий рахунок.* Практично всі класичні аркади включають в себе ігровий рахунок, коли гравець отримує бали за виконання різних цілей або завдань. При цьому типовий час гри середнього гравця становить близько двох хвилин, а у досвідченого гравця до десятків хвилин.
- *Швидке навчання, простий ігровий процес.* Для класичних аркад характерно те, що гравцям легко навчитися геймплею. Разом з тим, якщо гравець гине в аркаді, то це практично завжди відбувається з його вини. В таких іграх немає «спеціальних комбінацій клавіш», які гравець повинен вивчити з документації для того, щоб зробити щось особливе.
- *Відсутність сюжету та історії.* Класичні аркади практично завжди уникали спроб розповісти якусь історію, і дана тенденція продовжується для сучасних аркад. Для ігор жанру завжди було потрібно, щоб гравці швидко зрозуміли що відбувається - це наукова фантастика, війна, спорт або щось ще. Гейм дизайнери класичних аркадних ігор не відчували необхідності в тому , що їм

потрібно наповнювати свої світи чимось і окремо пояснювати гравцям чому вони повинні стріляти, бігти, або виконувати інші дії.

1.1.2 Ідея гри

Як вже було вказано вище, гра належить до категорії аркад. Тематика передбачає наступне: гравець керує м’ячем, який завжди рухається в 2 напрямах: вліво або направо. Гравець керує м’ячем через тапи по екрану, оскільки гра розроблюється під мобільну платформу Android на мові програмування C# в використанням Unity двигуна. Логотип у вигляді іконки (Рис.1.1) :

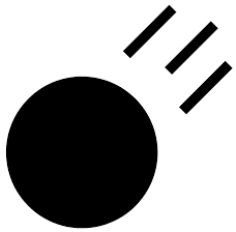

Рисунок 1.1

Геймплей за допомогою тапів (торкань) по екрану є інтуїтивним для гравця та допоможе охопити якомога більшу групу гравців. Ціль – пройти якомога далі по падаючим платформам, не вилетівши за їх межі. В процесі прогресії, гравець набирає бали, за кожну підібрану монетку, яка є окремим ігровим об’єктом та одним із ключових елементів механіки (Рис. 1.2):

Рисунок 1.2

1.2 Аналіз предметної області

В цілому, мобільні ігри користуються великою популярністю. Багато людей грають в ігри на своєму мобільному пристрої, будь то Android або iOS. Більшість ігор потребують участі двох рук користувача, або окремого контролера, проте багато ігрових проектів надають користувачам one-handed досвід, до дає змогу грати в гру будь-де, будь-коли. Таким чином таку гру можна віднести до категорії тайм-кіллерів.

1.2.1 Статистика індустрії

Інфографіка ринку мобільних ігор засвідчує, що зараз він характеризується різноманітністю ігрових жанрів, які задовольняють не менш різноманітну групу геймерів у всьому світі. За даними Sensor Tower, в третьому кварталі 2010 року кількість видавців ігрових проектів в Apple Store і Google Play склало 108 000 з 793

000, що становить 15 відсотків. App Annie прогнозує, що в 2021 році споживачі будуть разом витрачати 674 млрд. годин на геймінг на мобільних пристроях в порівнянні з 558 млрд. В 2020 р

Сьогодні на мобільні ігри припадає 33% всіх завантажень програм, 74% витрат, пов'язаних з всередині ігровими процесами і 10% всього часу, проведеного в додатку. До кінця 2021 року майже ⅓ населення Землі (2,4 млрд.) Гратимуть в мобільні ігри. Більше 50% користувачів мобільних додатків грають в ігри, що робить цю категорію додатків такою ж популярною, як музичні програми, такі як Spotify і Apple Music.

1.2.2 Розподіл долі ринку мобільних ігор

Збільшення доходів від мобільних ігор продовжить випереджати зростання доходів від продажу ПК ігор в наступному році, що в кінцевому підсумку призведе до скорочення частки ринку ігор для ПК до 20% в 2022 році. Крім цього, дохід від мобільних ігор також випереджає дохід від ігор на консолях. Ігрове аналітичне агентство Newzoo дає оцінку майбутнього положення ринку ігор по платформам в 2021 році. Наприклад, ігри на ПК будуть займати не більше 20% всіх доходів, в той час як ігри на консолях матимуть частку в 30%. До кінця 2022 року мобільні ігри будуть займати близько 50% всього ринку. (Рис.1.3).

Рисунок 1.3

Які сучасні тренди мобільної ігрової індустрії?

- Хмарний геймінг
- Геймінг в доповненій реальності
- Крос-платформенний геймінг
- Блок-чейн геймінг
- Гіпер-казуальний геймінг
- Змагальний мультиплеєрний геймінг

1.3 Огляд інструментів розробки

Основними інструментами розробки гри стали: редактор Unity, високорівнева мова програмування C# та середовище розробки Visual Studio 2019. Простіше кажучи, Unity - найпопулярніший у світі ігровий движок. Він поєднує в собі тонну функцій і є досить гнучким, щоб зробити майже будь-яку гру. Завдяки

особливим кросплатформенним функціям Unity користується популярністю як серед розробників міні-проектів, так і серед студій AAA рівня. Він використовувався для створення таких ігор, як Pokemon Go, Heathstone, Rimworld, Cuphead та багато інших. Програмісти цінять Unity через наявність можливості писати скрипти на C# та вбудованій інтеграції Visual Studio. Unity також пропонує JavaScript як альтернативну мову та MonoDevelop як IDE для тих, хто хоче щось менш навантажене, ніж Visual Studio. Unity також постачається з потужними анімаційними інструментами, які спрощують створення власних 3D-роликів або створення 2D-анімації з нуля. В Unity можна анімувати майже все.

До переваг даного ігрового движку можна віднести:

1. Unity використовує компонентний підхід до розробки ігор, який засновано на взаємодії з префабами. За допомогою префабів геймдизйнери та розробники можуть ефективніше будувати об’єкти та середовища та швидше масштабуватись.
2. Система префабів полегшує повторне використання коду та ресурсів інших проектів та їх редагування для нових цілей.
3. За допомогою інструментів редактора Unity присутня можливість одночасно обробляти рівзні види маніпуляторів, таких як: комп'ютерні миші, клавіатури, джойстики, тач-дисплеї і багато інших.
4. Існує також досить потужна підтримка хмарних рішень для багатокористувацьких ігор із хостингом на сервері та масштабованим встановленням зв’язків, що робить його універсальним рішенням для мультиплеєрного геймлею.
5. Але головна причина вибрати Unity - величезна бібліотека ресурсів, яка доступна кожному. Бібліотека містить тисячі моделей, сценаріїв, сцен, матеріалів та багато іншого.

1.4 Висновки до розділу

В першому розділі було обґрунтовано вибір жанру для створення проекту, охарактеризовано властивості, які притаманні представникам даного жанру. Ідея гри – створити гіперказульну аркаду, яка не зможе дати гравцю відчути себе непереможним, проте не буде навантажувати і відлякувати складними ігровими механіками, які формують високий поріг входження аудиторії.

В даному розділі було обґрунтовано вибір ігрового движку, який є основою будь-якої гри та інших інструментів, які дають можливість розробляти мультиплатформенні проекти на мові C#.

Також було проведено аналіз ринку мобільних додатків. Виходячи з даних аналізу, можна зробити висновок про те, що ігри займають велику долю всього ринку, який тільки збільшується у масштабах.

РОЗДІЛ 2
МАТЕМАТИЧНИЙ ОПИС ЗАДАЧІ

В даному розділі буде описано принципи роботи з математичними об’єктами, такими як: матриці, вектори, кватерніони. В свою чергу ці базові сутності є складовими механіками гри, такими як: рух, обертання, переміщення, масштабування, пересічення, визначення колізій ігрових об’єктів. Розуміння принципів симуляції природніх явищ дає можливість розробнику фантазувати та створювати реалістичні поведінкові моделі.

2.1 Варіації трансформацій ігрових об'єктів

Векторна арифметика є фундаментальною для багатьох аспектів комп’ютерного програмування, таких як графіка, фізика та анімація, і корисно її глибоко зрозуміти, щоб отримати максимальну користь від Unity

Математичні засади – основа, на якій формується фізична взаємодія ігрових об’єктів на сцені. Кожен компонент, який відрисовано в сцені, має специфічні властивості та поведінку, які обумовлені його параметрами, які задав гейм дизайнер. Об’єкти можна обертати, масштабувати, рухати. Всі ці властивості доступні завдяки симуляції математичних моделей. Чудова частина Unity полягає в тому, що в движок входить надійна математична бібліотека, тому там є найпоширеніші математичні функції. Однак, розробнику необхідне розуміння принципів їх роботи.

Трансформація використовується для зберігання таких станів об’єктів, як: позиція, обертання, масштаб та задаються координатами відповідно до простору, в якому перебуває об’єкт (Рис.2.1):

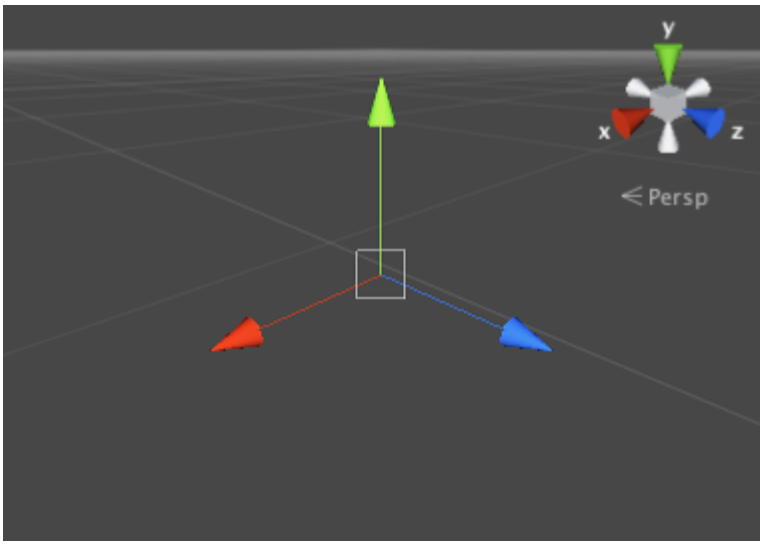

Рисунок 2.1

Трансформації визначаються властивостями, наведеними в таблиці 2.1:

Таблиця 2.1

Властивість	Функція
Position	Положення об’єкту трансформації в координатах X, Y та Z.
Rotation	Обертання об’єкту трансформації навколо осей X, Y та Z, вимірюване в градусах.
Scale	Масштаб перетворення по осях X, Y та Z. Значення "1" - це початковий розмір (розмір, при якому було імпортовано об’єкт).

Масштаб трансформації визначає різницю між розміром сітки у додатку для моделювання та розміром сітки в Unity. Розмір сітки в Unity (а отже, і масштаб Transform) дуже важливий, особливо під час фізичного моделювання. За замовчуванням фізичний движок передбачає, що одна одиниця у світовому просторі відповідає одному метру.

Є три фактори, які можуть вплинути на масштаб об’єкта:

- Розмір сітки у програмі 3D-моделювання.
- Налаштування коефіцієнта масштабування сітки в налаштуваннях імпорту об’єкта.
- Значення масштабу компонента трансформації.

В ідеалі, не слід налаштовувати масштаб вашого об'єкта в компоненті трансформації. Найкращий варіант - створити свої моделі в реальному масштабі, щоб не довелося змінювати масштаб вашого Transform. Деякі оптимізації відбуваються залежно від розміру імпорту, і створення екземпляра об’єкта, який має скориговане значення масштабу, що може знизити продуктивність.

2.1.1 Визачення руху та напрямку

В іграх вектори використовуються для зберігання розташування, напрямків і швидкостей.
Нижче наведено приклад двомірного вектора:

$$\bar{A}\,(x_a\,;y_a) \tag{2.1}$$

Де x_a — значення координат по осі X;

y_a- значення координат по осі Y.

Тривимірний вектор додає нову вісь – z:

$$\bar{A}\,(x_a\,;y_a\,;\,z_a\,) \tag{2.2}$$

Проте для того, щоб задавати необхідні параметри об’єктам, слід проводити арифметичні операції з векторами.

***Додавання для тривимірного простору*:**

$$\bar{A} + \bar{B} = (a_x + b_x\,; a_y\ +\ b_y\,;\ a_z\ +\ b_z\,) \qquad (2.3)$$

Де $\bar{A}$ та $\bar{B}$ – вектори, які підлягають операції додавання;

$a_{x,y,z}$ – координати вектора $\bar{A}$;

$b_{x,y,z}$ – координати вектора $\bar{B}$;

Якщо перший вектор прийняти за точку в просторі, то другий можна інтерпретувати як зміщення або «стрибок» з цієї позиції для того, щоб знайти точку на 5 одиниць вище над місцем на поверхні, наприклад.

Якщо вектори представляють сили, то більш інтуїтивно представляти їх як напрямок і величину прикладання сили. Додавання двох векторів сили дає новий вектор, еквівалентний поєднанню сил. Ця концепція часто корисна при застосуванні сил з кількома окремими компонентами, що діють одночасно.

Таким чином працює принцип суперпозиції векторів (Рис. 2.2):

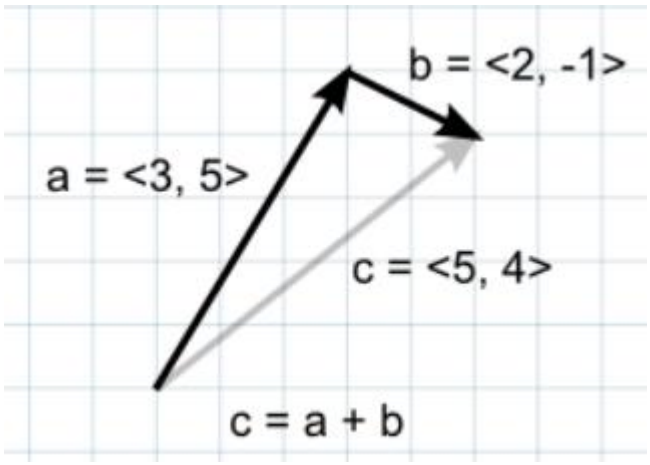

Рисунок 2.2

У коді це б виглядало наступним чином:

```
var pointInAir = pointOnGround + new Vector2(0, 0, 5);
```

Віднімання

Віднімання вектору найчастіше використовується для отримання напрямку та відстані від одного об’єкта до іншого (Рис.2.3):

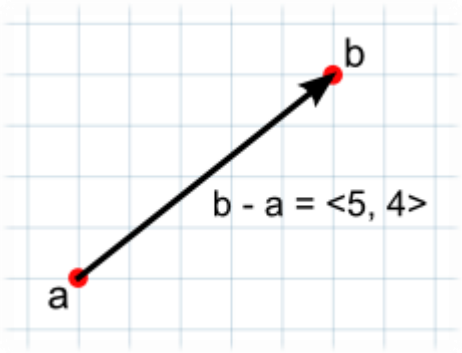

Рисунок 2.3

$$\bar{A} + \bar{B} = (a_x - b_x\,; a_y\ -\ b_y\,;\ a_z\ -\ b_z\,) \qquad (2.4)$$

Де $\bar{A}$ та $\bar{B}$ – вектори, які підлягають операції віднімання;

$a_{x,y,z}$ – координати вектора $\bar{A}$;

$b_{x,y,z}$ – координати вектора $\bar{B}$;

Множення та ділення на скаляр

Скаляр відображає лише величину, в той час як вектор відображає як величину, так і напрямок. Помноживши вектор на скаляр, отримується вектор, який вказує в тому ж напрямку, що і початковий. Однак величина нового вектору дорівнює початковій величині, помноженій на скалярне значення:

$$\lambda \cdot \bar{A} = (\overline{\lambda a_1; \lambda a_2}) \qquad (2.5)$$

Де $\bar{A}$ – вектор, які підлягають операції множення на скаляр;

λ – скалярна величина ;

$a_{x,y,z}$ – координати вектору $\bar{A}$;

Так само скалярний поділ ділить величину вихідного вектору на скаляр. Ці операції корисні, коли вектор представляє зміщення руху або силу. Вони дозволяють змінювати величину вектору, не впливаючи на його напрямок.

Скалярний добуток

Коли вектору вказують в одному напрямку, то їх скалярний добуток більше нуля. Коли вони перпендикулярні один одному, то скалярний добуток дорівнює нулю (Рис. 2.4):

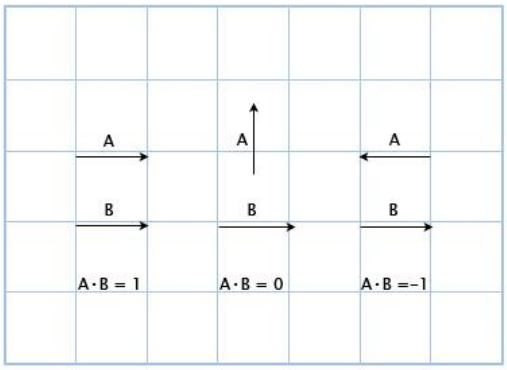

Рисунок 2.4

В основному, за допомогою скалярного добутку векторів можна розрахувати, скільки їх вказує в одному напрямку. І це дуже корисна властивість. Скалярний добуток приймає два вектори і повертає скаляр. Цей скаляр дорівнює величинам двох векторів, помножених разом, і результат, помножений на косинус кута між векторами:

$$\bar{A} \cdot \bar{B} = |\bar{A}| \cdot |\bar{B}| \cdot \cos\varphi \qquad (2.6)$$

Де $|\bar{A}|$ та $|\bar{B}|$ – модулі (довжини) векторів, які підлягають операції скалярного множення;

φ – кут між векторами;

У випадку, якщо вектори задані координатами:

$$\bar{A} \cdot \bar{B} = (a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z) \quad (2.7)$$

Де $\bar{A}$ та $\bar{B}$ – вектори, які підлягають операції множення;

$a_{x,y,z}$ – координати вектора $\bar{A}$;

$b_{x,y,z}$ – координати вектора $\bar{B}$;

В коді це виглядає наступним чином:

```
var fwdSpeed = Vector3.Dot(rigidbody.velocity, transform.forward);
```

Напрямок може бути будь-яким, але вектор напрямку завжди повинен бути нормалізований для цього розрахунку. Це дозволяє уникнути повільної в обчисленнях операції з квадратним коренем, яка бере участь у пошуку величини.

2.1.2 Повороти

Повороти в 3D-іграх зазвичай представляються одним із декількох способів: *матриці, кватерніони або кути Ейлера*. Кожен має своє власні переваги та недоліки. Unity використовує Кватерніони за замовчуванням, але показує значення еквівалентних кутів Ейлера в інспекторі. Кватерніони і матриці повороту можуть бути використані як для виконання просторового повороту векторів, так і для опису взаємної орієнтації систем координат. Зокрема, для опису положення цільової системи координат відносно вихідної: як потрібно повернути вихідну, щоб її осі збіглися з однойменними осями цільової. Матриця повороту (вона ж матриця

орієнтації, для випадку опису взаємної орієнтації систем координат) може бути отримана з нормованого кватерніона орієнтації по відомим з літератури формулам.

Нехай вектор що підлягає повороту задається величинами його проекцій на три ортогональні осі початкової СК:

$$v = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{2.8}$$

Матриці

Поворот з використанням матриці: матриця повороту *M* множиться на вихідну матрицю-стовпець. На виході маємо іншу, результуючу матрицю-стовпець:

$$\begin{bmatrix} v_{x+} \\ v_{y+} \\ v_{z+} \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} v_{x-} \\ v_{y-} \\ v_{z-} \end{bmatrix} \tag{2.9}$$

Числа в результуючої матриці-стовпці, отриманої будь-яким з цих способів - це проекції вектора в його новому, повернутому положенні, на осі все тієї ж вихідної системи координат Щоб додати новий ефект ми можемо перемножити матриці або ми можемо інвертувати матрицю, щоб отримати прямо протилежне положення об'єкта.

Кватерніони

Це – альтернативний варіант обертання, заснований на векторі і куті (axis-angle rotation), який існує в просторі (Рис. 2.5).

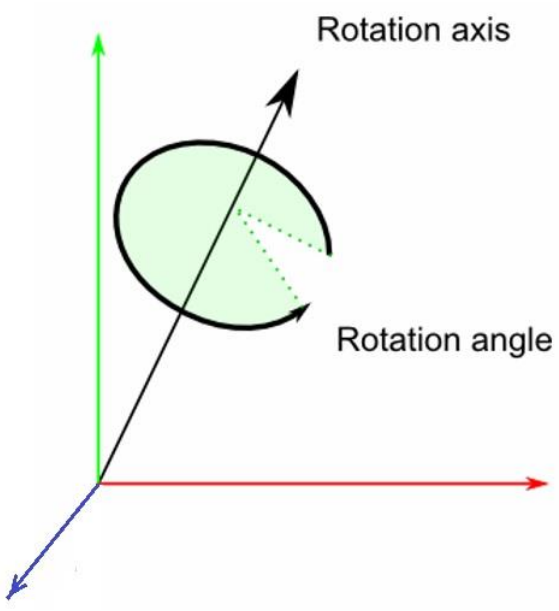

Рисунок 2.5

У фізиці, таким чином зберігають кутову швидкість. Напрямок вектора збігається з напрямком осі обертання, а довжина вектора дорівнює швидкості (в радіанах в секунду). Кватерніон (судячи з назви) являє собою набір з чотирьох параметрів, які визначають вектор і кут обертання навколо цього вектора. По суті таке визначення нічим не відрізняється від Axis Angle уявлення обертання. Відмінності лише в способі подання. У кватерніоні параметри одиничного вектора множиться на синус половини кута повороту. Четвертий компонент - косинус половини кута повороту.

$$q = \left[V * \sin\frac{\alpha}{2}, cos\frac{a}{2}\right] \quad (2.10)$$

Де V – вектор осі;

$\frac{\alpha}{2}$ – половина кута повороту (Див. (Рис.2.5));

Кути Ейлера

Кути Ейлера представлені трьома значеннями кутів для X, Y та Z, які застосовуються послідовно. Щоб застосувати обертання Ейлера до певного ігрового об’єкту, кожне значення обертання застосовується по черзі як обертання навколо

своєї відповідної осі. Від вибору осей і послідовності обертання залежить кінцевий результат. На малюнках відображена наступна послідовність обертання щодо осей ЛСК:

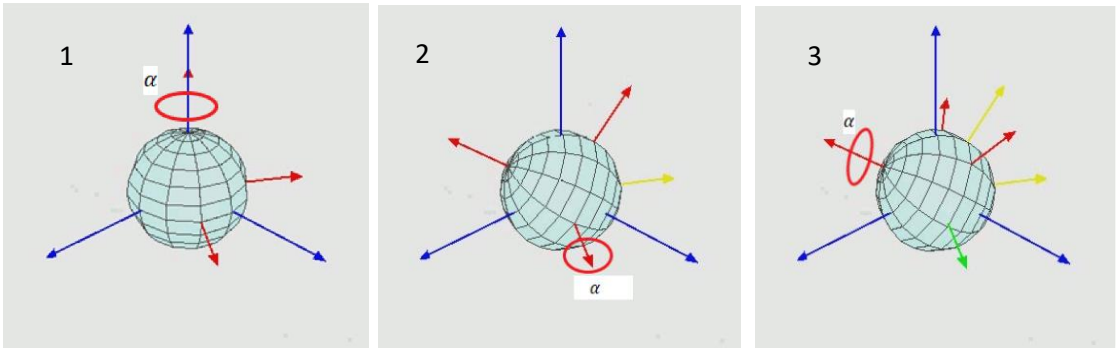

Рисунок 2.6

2.2 Визначення колізій

Означення колізій стосується виявлення пересічень між двома або більше об’єктами в ігровій сцені. Колізія є основним елементом взаємодії елментів гри, адже вона додає реалістичності геймпею.

Для перевірки перетину об’єктів може бути використаний Raycast. Тобто будується промінь з початкової до кінцевої точки. Unity визначить - перетинає промінь що-небудь, чи ні, і якщо перетинає, то також дасть інформацію про об'єкт і точку перетину.

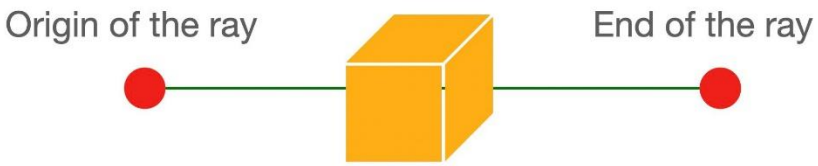

Рисунок 2.7

Raycast передає уявний «лазерний промінь» вздовж шляху, поки він не потрапляє в колайдер на сцені. Потім повертається інформація про об’єкт та точку, яка потрапила в об’єкт RaycastHit. Це дуже корисний спосіб знайти об’єкт на основі його зображення на екрані. Наприклад в коді таким чином перевіряється чи перебуває м’яч на поверхні чи ні (Рис 2.8):

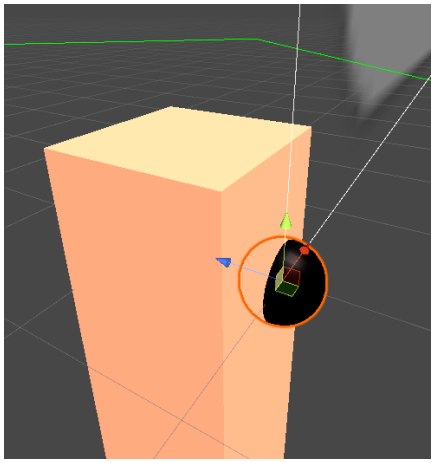

Рисунок 2.8

```
Ray ray = new Ray(origin, Vector3.down);

return Physics.Raycast(ray, Main.Get<BallConfig>().GroundLayer);
```

Для того, щоб дізнатися, де площина перетинається з площиною, необхідно скористатися скалярним добутком. Важливим елементом фізичного представлення

компонента в грі є колайдер. Компоненти колайдера визначають поведінку ігрового об’єкта під час фізичних зіткнень. Колайдер не обов’язково повинен мати однакову форму з об’єктом (Рис. 2.9). Груба апроксимація сітки часто є більш ефективною і не відрізняється в ігровому процесі.

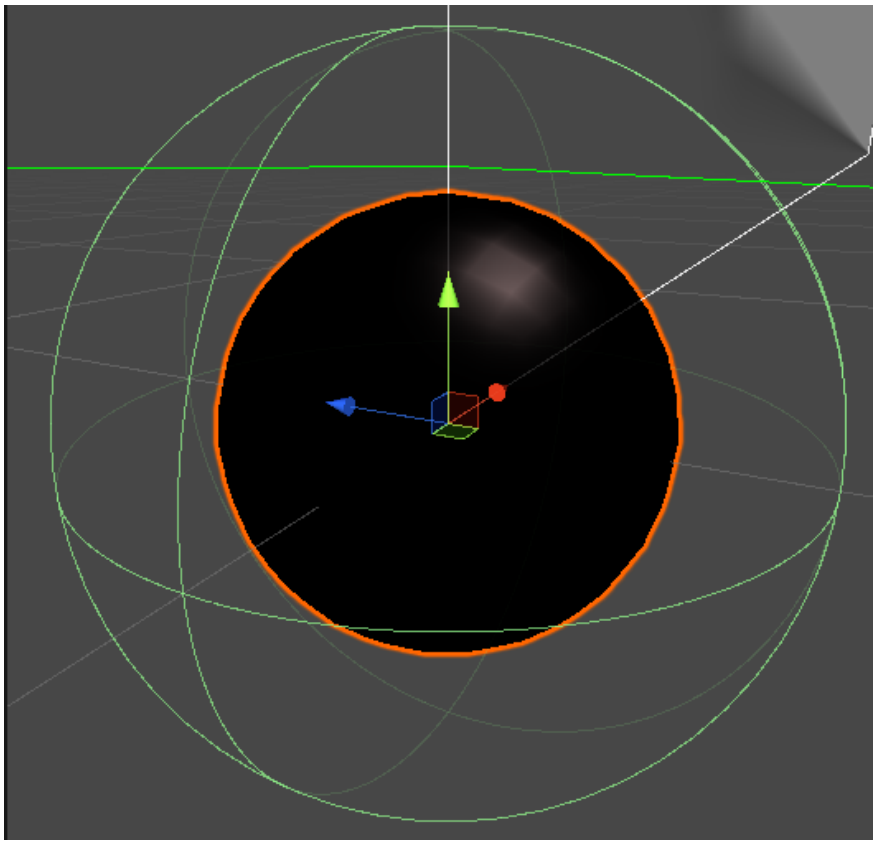

Рисунок 2.9

Найпростіші колайдери - це кубічний, сферичний та капсульний колайдер. Колайдери взаємодіють між собою по-різному, залежно від того, як налаштовані їх компоненти Rigidbody. Існує три типи колайдерів:

- Статичний колайдер

Статичний колайдер – це компонент, який не має Rigidbody. Статичні колайдери в основному використовуються для геометрії рівня, який завжди залишається на одному місці і ніколи не рухається. Вхідні об'єкти з твердим тілом стикаються зі статичними колайдерами, але не рухають їх.

- Колайдер Rigidbody

Це компонент, що застосовується до некінематичного твердого тіла. Колайдери з твердим тілом повністю імітуються фізичним двигуном і можуть реагувати на зіткнення та сили. Вони можуть зіткнутися з іншими об'єктами (включаючи статичні колайдери) і є найбільш часто використовуваною конфігурацією колайдерів в іграх.

- Кінематичний колайдер Rigidbody

Об’єкти з таким типом колайдерів не будуть реагувати на зіткнення та сили, які на нього прикладені. Кінематичні тверді тіла повинні використовуватися для колайдерів, які можна час від часу переміщати або вимикати / вмикати, але які в іншому випадку повинні поводитися як статичні колайдери.

2.3 Висновки до розділу

Розуміння принципів роботи з математичними сутностями під час симулювання ігрового процесу є пріоритетним завданням розробника. В цьому розділі було детально описано закономірності та правила використання векторної арифметики, роботи з матрицями та кватерніонами, які формують основні механіки поведінки об’єктів на сцені, такі як: визначення напрямків, поворотів, рухів.

Було розглянуто принципи роботи з різними фізичними компонентами. Unity допомагає імітувати фізику у проекті, для того, щоб об’єкти правильно прискорювалися та реагували на зіткнення, була присутня сила тяжіння та інші сили. Unity пропонує різні конфігурації фізичного двигуна, які можна використовувати відповідно до потреб проекту.

РОЗДІЛ 3

ІМПЛЕМЕНТАЦІЯ ТЕХНІЧНИХ РІШЕНЬ ТА КОМПОНЕНТІВ ГРИ

У даному розділі буде описано особливості архітектурних та технічних рішень, які були прийняті під час розробки гри. Опис компонентів гри, який включає в себе всі геймоб’єкти, скрипти, префаби, елементи графічного інтерфейсу, анімації, робота за камерою за ігровим середовищем. Буде розглянуто прийоми, які були використані задля забезпечення оптимізації та стабільного фреймрейту.

3.1 Дизайн архітектури

В Unity найкраще використовувати компонентно-орієнтований підхід, тобто композицію - формування цілого із частин.

Компоненти - це сценарії, які можна додати до ігрового об’єкту; їх можна розробити та реалізувати різними способами. Створення функцій гри за допомогою підкомпонентів (а не успадкування від об’єкта), як правило, є більш ефективним та простим. Саме це робить його дизайном на основі “компонентів”. Комбінування на накладання різних компонентів дає можливість досягти бажаних властивостей від об'єкту:

Наступна таблиця добре відобразить принципи роботи такого архітектурного рішення на прикладі двох ігрових об'єктів – камери (Рис.3.1) та монети (Рис.3.2). Вони обидва складаються з наступних компонентів:

Таблиця 3.1

Main Camera	Coin
Audio Listener component	Coin Pickup component
Camera Controller component	Sphere Collider component

Color Shifting component	Move and Rotate component

Продовження таблиці 3.1

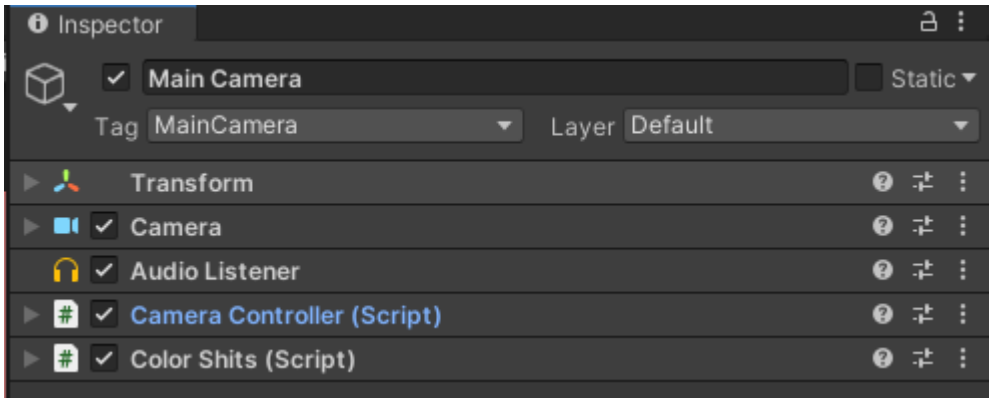

Рисунок 3.1

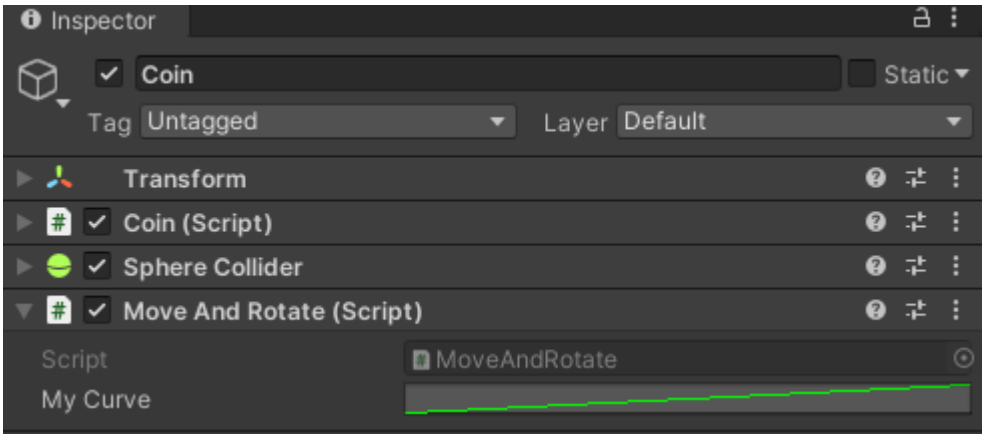

Рисунок 3.2

Основна ідея шаблону проектування «Компонент» полягає в угрупованні тісно пов'язаних функцій і даних в один клас і, одночасно, в збереженні класів якомога дрібнішими і спеціалізованими.

Всі компоненти, що підключаються до ігрових об'єктів в Unity, написані із застосуванням цього шаблону проектування. Кожен ігровий об'єкт в Unity – це дуже маленький клас, який може служити контейнером для безлічі компонентів, кожен з яких вирішує конкретну - незалежно від інших компонентів задачу. наприклад:

- Компонент *Transform* відповідає за координати, поворот, масштабування і місце розташування в ієрархії.

- Компонент *Rigidbody* вирішує завдання, пов'язані з рухом і моделюванням законів фізики.
- Компоненти *Collider* підтримують можливість виявлення зіткнень і визначають форму і властивості зіткнень.

Створення окремих компонентів також спрощує розширення їх можливостей в майбутньому: завдяки відділенню компонента Collider від Rigidbody ви легко зможете додати новий вид колайдера, наприклад конічний колайдер ConeCollider, і Rigidbody зможе використовувати його без будь-яких змін в коді Rigidbody.

Які переваги? Перш за все, слідуючи компонентно-орієнтованому стилю, формуються більш прості й короткі класи. Короткі скрипти простіше писати, їх простіше використовувати повторно, і вони простіші в налаштуванні.

Таким чином, камера має скрипт, який відповідає за рухи, скрипт, який змінює бекграунд, робить його приймачем звуку, а монетка підбирається, має фізичний колайдер, рухається та обертається.

Проте об’єктно-орієнтована та компонентно-орієнтована парадигми не взаємовиключні, хоч іноді можуть стикатися. Прикладом цього є базовий клас **Entity**, від якого наслідуються інші класи **(**Рис. 3.3**):**

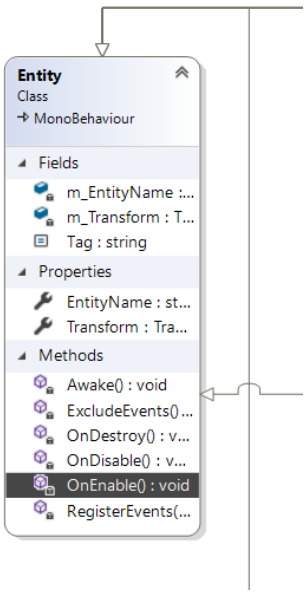

Рисунок 3.3

Загальна діаграма класів з усіма методами та полями виглядає наступним чином на Рис. 3.4.

Однак найскладніше в архітектурі на основі компонентів – це вибір з безлічі способів реалізації. Про один з таких буде йти мова в наступному підрозділі.

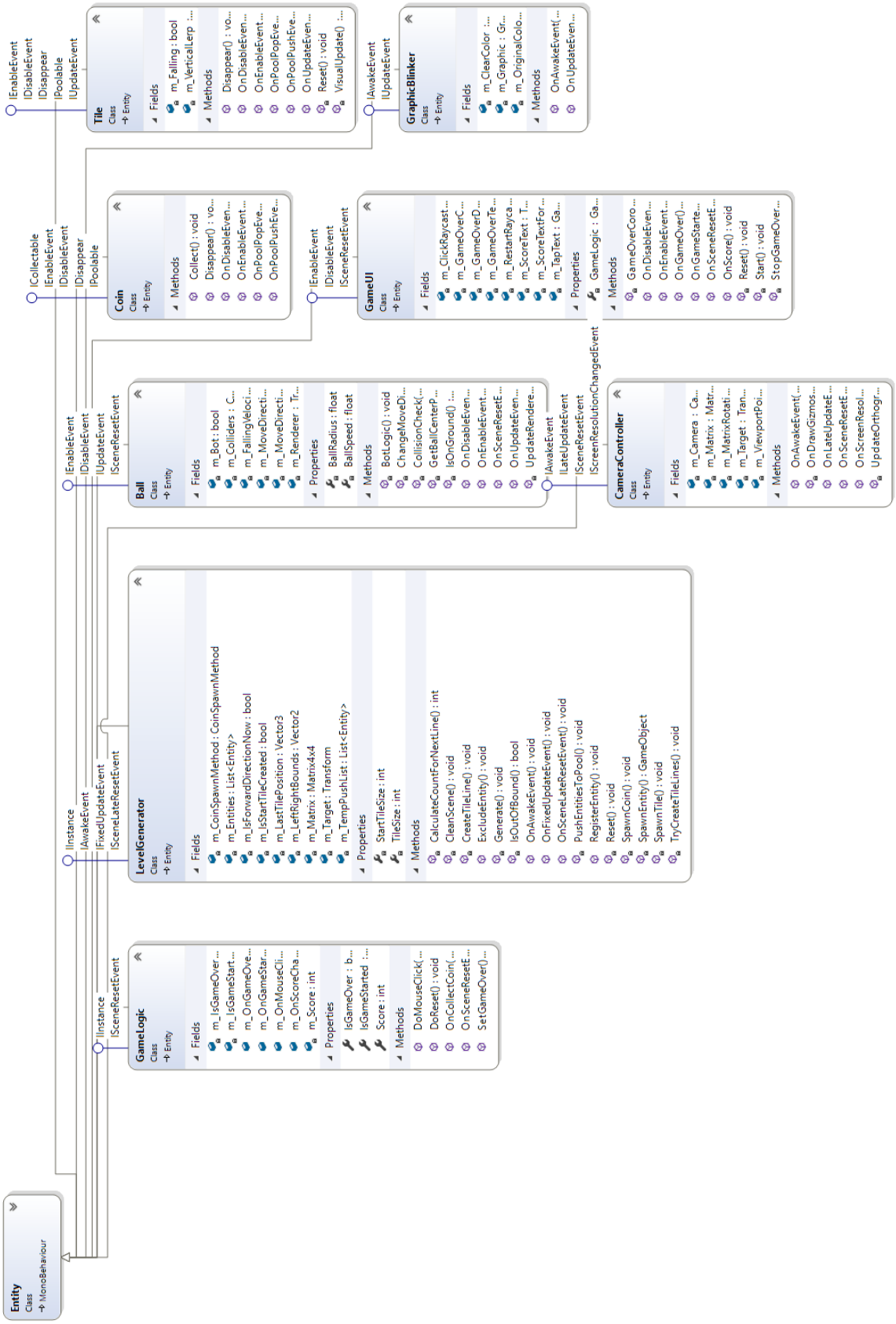
IEnableEvent
IDisableEvent
IDisappear
IPoolable
IUpdateEvent
Tile
Class
Entity
Fields
m_Falling : bool
m_VerticalLerp :...
Methods
Disappear() : vo...
OnDisableEven...
OnEnableEvent...
OnPoolPopEve...
OnPoolPushEve...
OnUpdateEven...
Reset() : void
VisualUpdate() :...
IAwakeEvent
IUpdateEvent
GraphicBlinker
Class
Entity
Fields
m_ClearColor :...
m_Graphic : Gr...
m_OriginalColo...
Methods
OnAwakeEvent(...
OnUpdateEven...
ICollectable
IEnableEvent
IDisableEvent
IDisappear
IPoolable
Coin
Class
Entity
Methods
Collect() : void
Disappear() : vo...
OnDisableEven...
OnEnableEvent...
OnPoolPopEve...
OnPoolPushEve...
IEnableEvent
IDisableEvent
ISceneResetEvent
GameUI
Class
Entity
Fields
m_ClickRaycast...
m_GameOverC...
m_GameOverD...
m_GameOverTe...
m_RestartRayca...
m_ScoreText : T...
m_ScoreTextFor...
m_TapText : Ga...
Properties
GameLogic : Ga...
Methods
GameOverCoro...
OnDisableEven...
OnEnableEvent...
OnGameOver()...
OnSceneResetE...
OnScore() : void
Reset() : void
Start() : void
StopGameOver...
IEnableEvent
IDisableEvent
IUpdateEvent
ISceneResetEvent
Ball
Class
Entity
Fields
m_Bot : bool
m_Colliders : C...
m_FallingVeloci...
m_MoveDirecti...
m_MoveDirecti...
m_Renderer : Tr...
Properties
BallRadius : float
BallSpeed : float
Methods
BotLogic() : void
ChangeMoveDi...
CollisionCheck(...
GetBallCenterP...
IsOnGround() :...
OnDisableEven...
OnEnableEvent...
OnSceneResetE...
OnUpdateEven...
UpdateRendere...
IAwakeEvent
ILateUpdateEvent
ISceneResetEvent
IScreenResolutionChangedEvent
CameraController
Class
Entity
Fields
m_Camera : Ca...
m_Matrix : Matr...
m_MatrixRotati...
m_Target : Tran...
m_ViewportPoi...
Methods
OnAwakeEvent(...
OnDrawGizmos...
OnLateUpdateE...
OnSceneResetE...
OnScreenResol...
UpdateOrthogr...
IInstance
IAwakeEvent
IFixedUpdateEvent
ISceneLateResetEvent
LevelGenerator
Class
Entity
Fields
m_CoinSpawnMethod : CoinSpawnMethod
m_Entities : List<Entity>
m_IsForwardDirectionNow : bool
m_IsStartTileCreated : bool
m_LastTilePosition : Vector3
m_LeftRightBounds : Vector2
m_Matrix : Matrix4x4
m_Target : Transform
m_TempPushList : List<Entity>
Properties
StartTileSize : int
TileSize : int
Methods
CalculateCountForNextLine() : int
CleanScene() : void
CreateTileLine() : void
ExcludeEntity() : void
Generate() : void
IsOutOfBound() : bool
OnAwakeEvent() : void
OnFixedUpdateEvent() : void
OnSceneLateResetEvent() : void
PushEntitiesToPool() : void
RegisterEntity() : void
Reset() : void
SpawnCoin() : void
SpawnEntity() : GameObject
SpawnTile() : void
TryCreateTileLines() : void
IInstance
ISceneResetEvent
GameLogic
Class
Entity
Fields
m_IsGameOver...
m_IsGameStart...
m_OnGameOve...
m_OnGameStar...
m_OnMouseCli...
m_OnScoreCha...
m_Score : int
Properties
IsGameOver : b...
IsGameStarted :...
Score : int
Methods
DoMouseClick(...
DoReset() : void
OnCollectCoin(...
OnSceneResetE...
SetGameOver()...
Entity
Class
MonoBehaviour

Рисунок 3.4

3.2 Скрипт-менеджер

В Unity немає головної функції, оскільки все зав’язано на роботі з компонентами. Простим рішенням є створити порожній ігровий об’єкт із прикріпленим скриптом менеджера.

У грі присутні глобальний об'єкт (менеджер), який буде перебувати в грі завжди і може бути доступним з будь-якого скрипта, що може бути корисно для створення класів генерації рівнів, контролю камери, контролю графічного інтерфейсу.

Крім менеджера в грі будуть використовуватися і інші об'єкти: інтерфейси, ігрові компоненти і об'єкти ігрового світу. Всі ці об'єкти будуть щільно взаємодіяти з менеджером для досягнення кінцевої мети.

Таким класом-менеджером в випадку моєї гри є клас **GameLogic**, який, як видно з назви відповідає за флоу – логіку (Рис. 3.5):

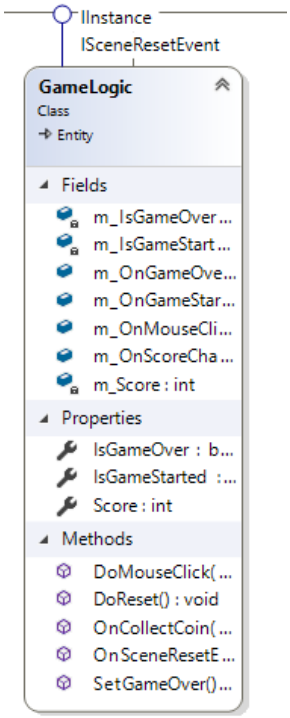

Рисунок 3.5

Клас складається з приватних булевих полів, які відображають поточний статус гри, рахунок, тапи та їх публічні властивості та методи, які перевіряють виконання ключових ігрових умов.

3.3 Івент-функції

Робота скриптів в Unity відрізняється від постійного зациклювання коду програми, до поки вона не буде виконана. Натомість Unity передає контроль скрипту почергово через виклик необхідних функцій, які в ньому ініціалізовані. Як тільки функція закінчила своє виконання, контроль передається назад до Unity. Ці функції відомі як event функції, оскільки вони викликаються двигуном у відповідь до подій під час геймплею. Unity використовує неймінг-схему, яка ідентифікує функцію для її виклика. Наприклад, під час створення срипт-компоненту, за замовчуванням створюється 2 методи: Update (викликається під час кожного оновлення кадру), Start (викликається до першого оновлення кадру, ініціалізації). Цей ігровий двигун надає набагато більше внутрішніх функцій. Блок-схема життєвого циклу скрипта складається з десятків функцій, проте ці (Рис. 3.6) найчастіше вживані:

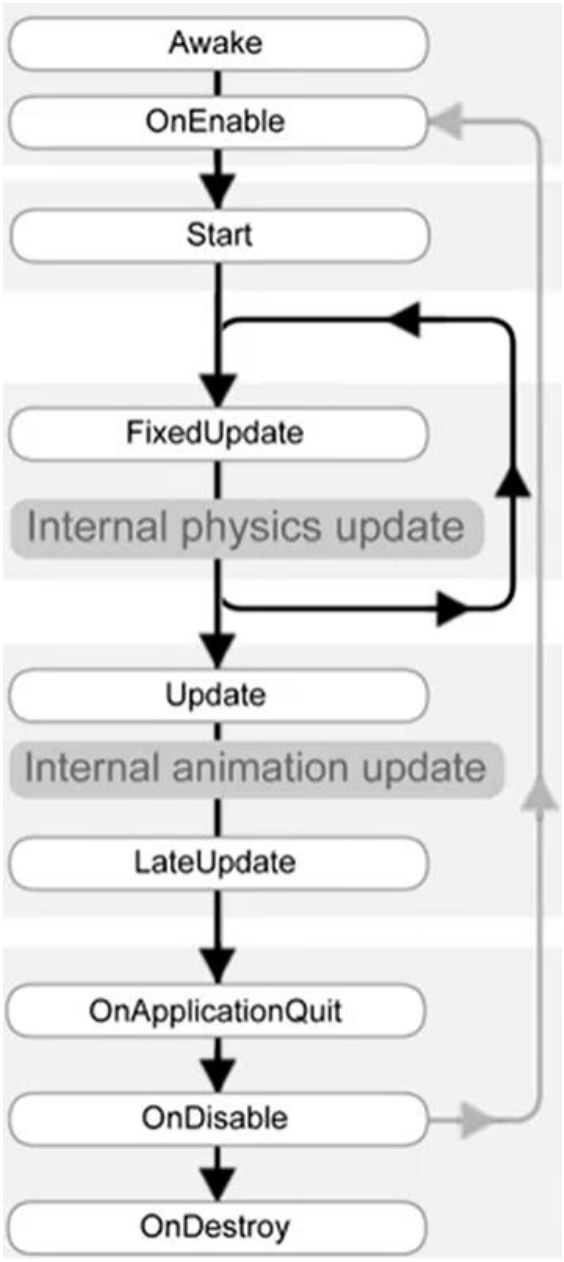

Рисунок 3.6

- *Awake*: викликається для кожного об'єкта в сцені під час завантаження сцени. Хоча Awake та Start викликаються у довільному порядку, Awake повинен закінчитися до початку першого Start. Це означає, що код у функції Start може використовувати інші ініціалізації, попередньо виконані у фазі Awake.
- *OnEnable*: (викликається, лише якщо об’єкт активний): Ця функція викликається відразу після того, як об’єкт активований, тобто коли

створюється екземпляр, наприклад, завантажується рівень або ігровий об'єкт. Зверніть увагу, що для об’єктів, доданих до сцени, функції Awake та OnEnable для всіх сценаріїв буде викликано до того, як для будь-якого з них буде викликано Start, Update, тощо

- *Start*: викликається перед першим оновленням кадру або фізики на об'єкті.
- *FixedUpdate:* іноді викликається навіть частіше, ніж Update. Його можна викликати кілька разів за кадр, якщо частота кадрів низька, і може не викликатися між кадрами, якщо частота кадрів висока. Усі фізичні розрахунки та оновлення відбуваються відразу після FixedUpdate.
- *Update:* викликається один раз на кадр. Це основна функція для оновлення кадру.
- *LateUpdate:* викликається один раз на кадр після завершення Update. Будь-які обчислення, які виконуються в Update, будуть завершені, до того, як почнеться LateUpdate.
- *OnApplicationQuit:* Ця функція викликається на всіх ігрових об'єктах до виходу програми. У редакторі вона викликається, коли користувач зупиняє режим відтворення.
- *OnDisable:* Ця функція викликається, коли об'єкт деактивовано або він стає неактивним.
- *OnDestroy*: викликається при знищенні ігрового об'єкту.

На прикладі того ж класу **`GameLogic`** розглянемо роботу та умови виклику кожного методу.

`SetGameOver()` – перевіряє стан гри та є його флагом.

`DoMouseClick()` – перевіряє чи натиск на экран, та є його флагом

`DoReset()` – оновлює головну сцену

`OnCollectCoin()` – оновлює кількість очок після того, як гравець підбирає монетку.

`OnSceneResetEvent()` – оновлює показник кількості очок після оновлення головної сцени.

3.4 Оптимізація

Розробка ігор - це складна галузь, у якій використовується багато технік для забезпечення якомога більш плавної та стабільно роботи на девайсах кінцевих користувачів, особливо якщо мова йде про мобільний телефон, де обсяг оперативної пам’яті значно нижчий у порівнянні з ПК чи іншою ігровою платформою.

Геймплей гри було розроблено з врахуванням та націленням на слабке апаратне забезпечення смартфону, тому було прийнято застосувати такі оптимізаційні рішення, як корутини (співпрограми) та обджект пулінг, про які піде мова в наступних підрозділах.

3.4.1 Coroutines

Механізм підтримки співпрограм в C # дозволяє методу призупинитися посеред процесу обчислень, дати можливість виконатися іншим процесам і потім продовжити роботу з місця призупинення.

Співпрограми часто використовуються в Unity для виконання тривалих обчислень (які, якщо їх не при зупиняти періодично, можуть створити враження, що гра зависла).

Співпрограми можна також використовувати в ролі таймерів для завдань, які повинні виконуватися через певні інтервали часу. Багато завдань у грі потрібно виконувати періодично, і найбільш очевидний спосіб зробити це - включити їх у функцію `Update`. Однак ця функція зазвичай викликається багато разів на секунду. Коли перевірку не потрібно повторювати так часто, можна помістити його в програму, щоб регулярно отримувати оновлення, але не кожен кадр.

У випадку моєї гри корутини були використані для перевірки гри на стан гейм оверу (кінця гри), тобто коли гравець вилітає за межі платформи.

Для цього було створено змінну дельти часу:

```
private float m_GameOverDelay = 1.0f;
```

Змінна дорівнювала 1 секунді та використовувалася як параметр для інтструкції `yield`, яка дасть змогу відновити виконання процесу через 1 секунду. Для прикладу, можливі варіанти використання з іншими інструкціями `yield`:

`yield return null` - відновлює виконання при першій можливості;

`yield return new WaitForSeconds(10)` – чекає 10 секунд;

`yield return new WaitForEndOfFrame()` – чекає наступного кадру;

`yield return new WaitForFixedUpdate()` – чекає наступного виклику `FixedUpdate`;

3.4.2 Object pooling

Пулінг - це техніка оптимізації продуктивності, яка полягає у повторному використанні об’єктів замість того, щоб створювати та знищувати їх кожного разу, коли вони потрібні. Детально механізм роботи даної техніки оптимізації було розглянуто в моїй статті «OBJECT POOLING AS A GAME DEVELOPMENT OPTIMIZATION INSTRUMENT» статті 8-ї Східно-Європейської конференції Математичні та програмні технології Internet of Everything.

Об'єкти та їх скрипти змінюють свої стани з активованого на деактивований, коли використовується пул. Дивлячись на ЖЦ скрипта, можна допустити, що буде доцільно мати код в методі Start(). Однак Start() викликається тільки після активування скрипта в перший раз.

Якщо перенести код зі Start() до OnEnable() то можна переконатися в тому що код виконується під час повторного використання об'єкту

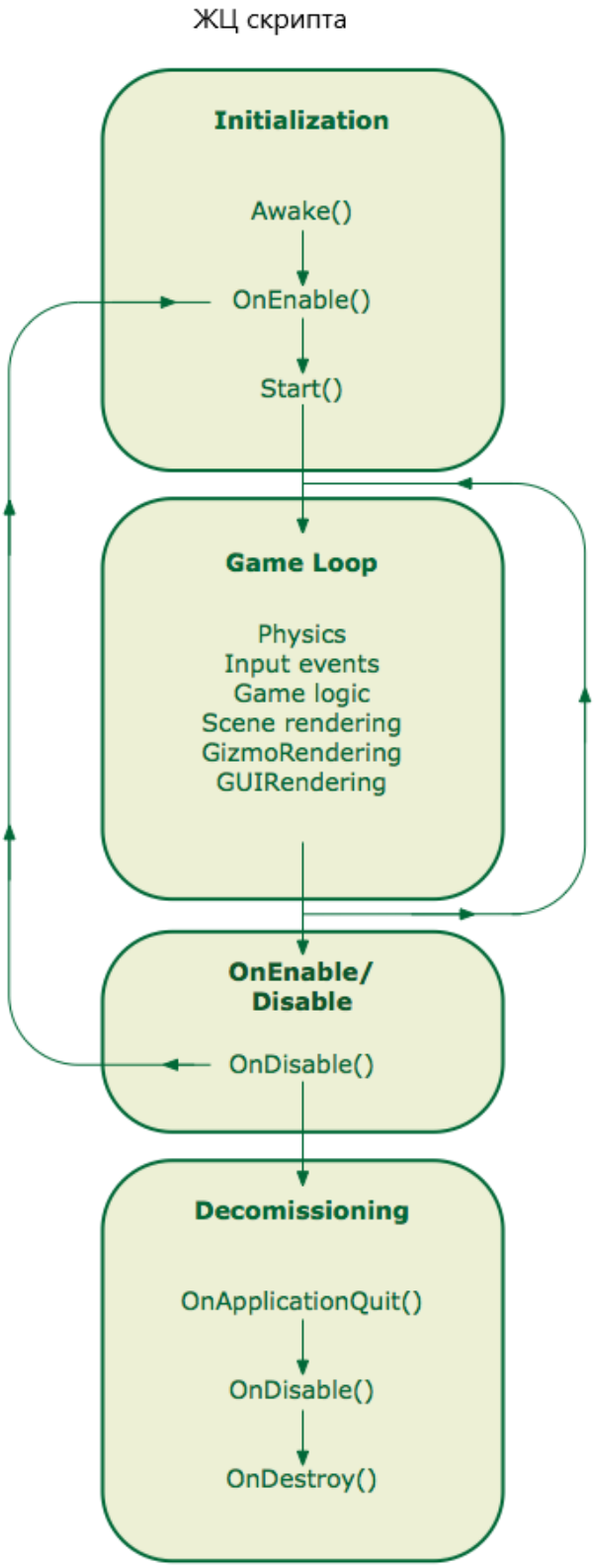

Рисунок 3.7

Багаторазове використання преінстанційонваних об'єктів лежить в основі роботи даної техніки. В моєму випадку об'єкти генеруються до ініціалізації ігрової сцени та використовуються в процесі гри, не потребуючи:

- створювати екземпляри та знищувати ігрові об'єкти – платформи.
- часто виділяти та звільняти об'єкти, що зберігаються в купі (замість повторного їх використання).

Принцип роботи описано на Рис.3.8:

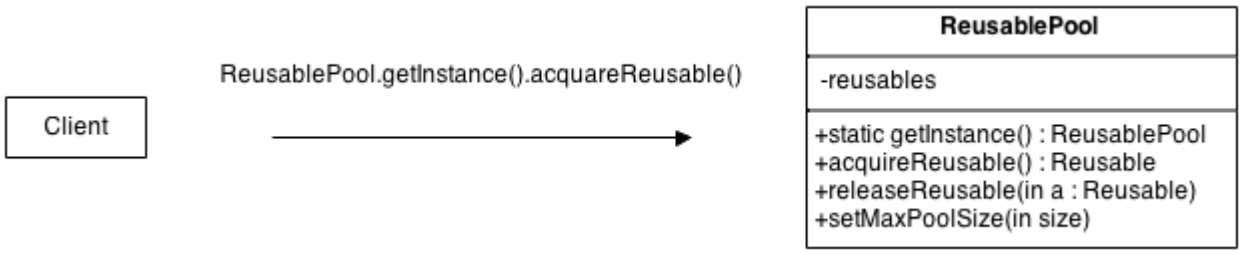

Рисунок 3.8

У моїй грі було використано цей принцип для генерації рівня з платформ. До конфігу генерації рівня було додано декілька змінних, які зберігають значення розмірності платформ:

- `private int m_StartTileSize` – розмір початкової платформи в блоках
- `private int m_TileSize` – розмір платформи, що додається в блоках
- `private int m_MaxTilesPerDirection` – максимальна кількість блоків, з яких може бути побудована платформа
- `private float m_BufferPosition` – буфер згенерований блоків
- `private float m_DisappearPosition` – через скільки блоків платформа може знову бути додана до буферу.

Кожен компонент гри, в тому числі і скрипт генерації рівня буде розглянуто в наступному підрозділі.

3.5 Дизайн-документ компонентів гри

В даному підрозділі буде розібрано різноманітні компоненти, з яких складається гра. Більш того, це ті елементи, з якими саме гравець буде взаємодіяти. Ця взаємодія буде формувати унікальний користувацький ігровий досвід, саме тому вони є дуже важливими. Правильне розуміння балансу гри, позиціювання та взаємозв'язків між елементами формує ігрові механіки. Для кращого розуміння призначення кожного компоненту, в дизайн-документах зазвичай їх розбивають на категорії за призначенням. В дизайн-документі всі елементи повинні бути чітко описані: наприклад, елементи, якими займаються художники, - текстури, спрайти, анімації, шрифти, візуальні ефекти і так далі. А програміст з дизайн-документа повинен зрозуміти не тільки який код він буде писати, а й як йому додавати в гру спрайт або 3D-моделі. Мною було виділено основні категорії: 3C, Game World, UI – кожна з яких буде описана нижче.

3.5.1 3C

Акронім розшифровується як «Персонаж, елементи керування, камера» (Character, Control, Camera). Багато хто використовує його в індустрії розробки ігор, щоб швидко описати складні взаємозалежні взаємозв'язки між контролем гравця та реакціями персонажа на екрані та тим, як цей персонаж рендериться у кадрі під час гри. Іншими словами, 3C відповідає за ігровий досвід; як гравець відчуває управління гри, як проходить взаємодія з основними механіками гри.

Почемо з першого з 3-х «С» - **Character**. Персонажем, тобто об’єктом, який керує гравець є м’яч – сфера. В ієрархії він виглядає наступним чином:

Рисунок 3.9

До його компонентів входять:

- **`ParticleSystem`** – додає ефект падаючого пилу під час руху. Обмежена 1000 частинок, які генеруються на протязі 5 секунд у формі напівсфери, піддаються гравітації.

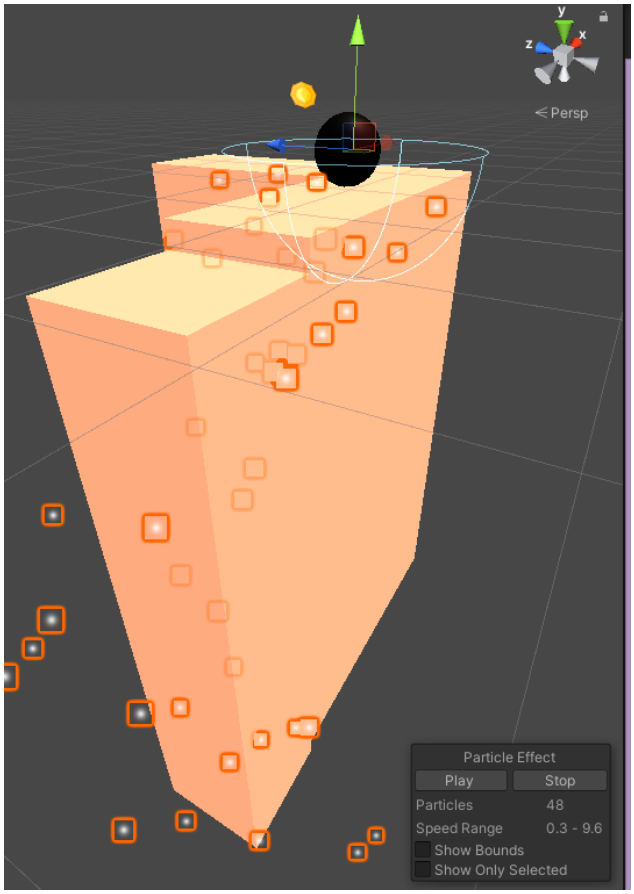

Рисунок 3.9

- **`Sphere`** – містить в собі властивості шейдерів та матеріалів, які накладаються на об’єкт на формують його візуальне відображення
- Сам **`Ball`** містить в собі одніменний скрип `Ball, BallConfig.`

`BallConfig` представляє собою клас, який містить в собі приватні поля та властивості, які відповідають за швидкість, радіус та рівень поверхні: `m_BallSpeed, m_BallRadius, m_GroundLayer`.

Ці змінні є відкритими для модифікації з самого редактору та їх можна редагувати на льоту.

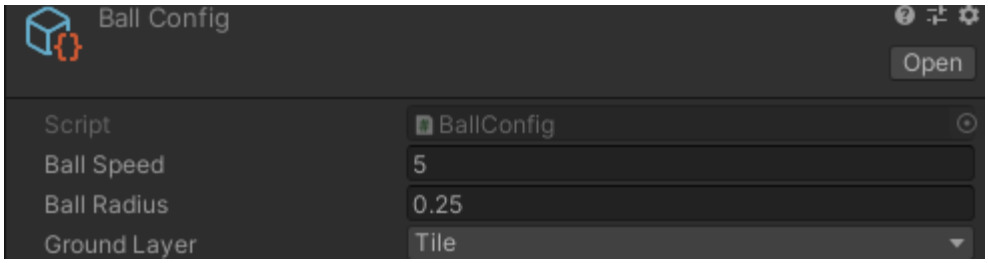

Рисунок 3.10

В скрипті **`Ball`** присутні такі *змінні*:

`m_MoveDirectionIndex` – індекс напрямку руху;
`m_FallingVelocity` – прискорення вільного падіння;
`[] m_Colliders` – масив колайдерів;
`m_Bot` – режим боту (автопілоту);

Наступним невід'ємним елементом 3С є **Control**, тобто способи управління персонажем. Присутні наступні *методи*.

Метод зміни напрямку руху:

```
m_MoveDirectionIndex = (m_MoveDirectionIndex + 1) % m_MoveDirections.Length;
```

де `MoveDirections` – масив напрямів: [Vector3.right, Vector3.forward]

По натисканню на екран змінюється напрям руху м'ча:

```
Main.Get<GameLogic>().m_OnMouseClick += ChangeMoveDirection;
Main.Get<GameLogic>().m_OnMouseClick -= ChangeMoveDirection;
```

Присутній метод перевірки на кінець гри:

```
{if (Transform.position.y < -1000.0f) // TODO: Min height position{return;}
m_FallingVelocity += Physics.gravity * Time.deltaTime;
Transform.position += m_FallingVelocity * Time.deltaTime;
```

В такому випадку, якщо гра закінчена, тобто відсутня колізія з платформою і позиція м’яча не менше 1000 по осі Y, то м’ячу задається нове значення position, яке є добутком прискорення падіння на інтервал часу з останнього оновлення кадру.

Метод, що описує логіку роботи Bot-режиму – **`BotLogic`**():

```
if (!IsOnGround(GetBallCenterPosition() + m_MoveDirections[m_MoveDirectionIndex] *
Main.Get<LevelGeneratorConfig>().TileSize * 0.5f))
ChangeMoveDirection();
```

Як тільки м’яч пересікає половину останнього блока платформи, він, автоматично змінює напрямок руху на протилежний.

Метод перевірки колізії з платформою – **`IsOnGround`**(), який формується на побудові променів з джерела, тобто, м’яча:

```
Ray ray = new Ray(origin, Vector3.down);
return Physics.Raycast(ray, Main.Get<BallConfig>().GroundLayer);
```

Метод оновлення сцени

```
Transform.position = Vector3.zero;
m_MoveDirectionIndex = 0;
m_FallingVelocity = Vector3.zero;
```

Обнуляє змінnні, що впливають на геймплей.

Ще одним обов’язковим елементом є **Camera.** В багатьох випадках, робота з камерою визначає ігрові механіки та навіть жанри проектів, наприклад: FPS – шутер від першої особи.

Ортографічна камера - камера з ортографічною проекцією. При цьому способі проектування, розмір об'єкта у видимій зображенні залишається постійним, незалежно від відстані між ним і камерою. Між іншим, це може бути корисним при відображенні двомірних (2D) сцен і елементів інтерфейсу користувача (UI).

```
Orthographic_Camera( left, right, top, bottom, near, far )
```

`left` - ліва площина відсікання області видимості камери.

`right` - права площина відсікання області видимості камери.

`top` - верхня площина відсікання області видимості камери.

`bottom` - нижня площина відсікання області видимості камери.

`near` - ближня площина відсікання області видимості камери.

`far` - далека площина відсікання області видимості камери.

Разом вони визначають область перегляду камери (у вигляді усіченої піраміди)

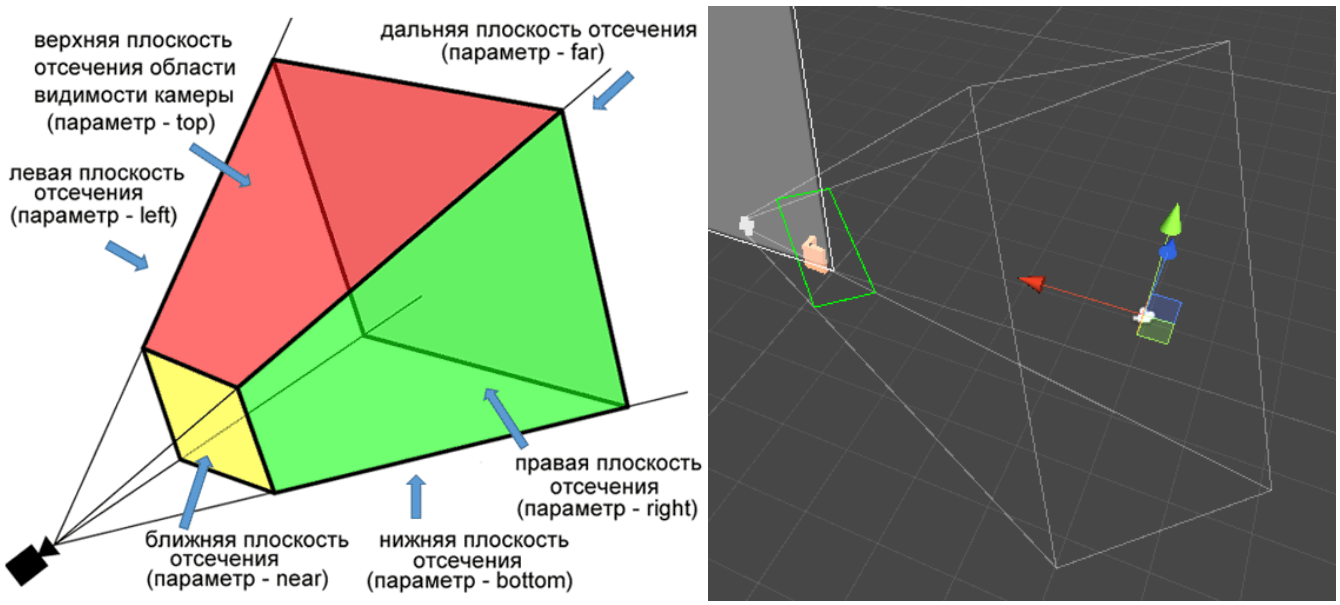

Рисунок 3.11

Скрипт **CameraControler** вміщує наступні *поля*:

`private Transform m_Target` – об'єкт, за яким закріплена камера;

`private Quaternion m_MatrixRotation` – матриця повороту;

Масив точок для фіксування камери:

```
private static readonly Vector2[] m_ViewportPoints = new Vector2[]
        {
new Vector2(0.0f, 0.0f),
new Vector2(0.0f, 1.0f),
new Vector2(1.0f, 0.0f),
new Vector2(1.0f, 1.0f),
new Vector2(0.0f, 0.0f),
new Vector2(1.0f, 0.0f),
new Vector2(0.0f, 1.0f),
new Vector2(1.0f, 1.0f)
};
```

Після кожного виклику функції Update викликається функція **OnLateUpdate**. Виконуються перевірки на стан гри:

```
if (Main.Get<GameLogic>().IsGameOver)
{return;}
```

А також присвоюється значення точки закріплення камери:

```
if (m_Target != null)
{
Vector3 cameraTargetPosition =m_Matrix.inverse.MultiplyPoint3x4(m_Target.position);
cameraTargetPosition = m_MatrixRotation * new Vector3(0.0f, cameraTargetPosition.y,
cameraTargetPosition.z);
Transform.position = Transform.rotation * new Vector3(0.0f, 0.0f,
Main.Get<CameraControllerConfig>().CameraOffset) + cameraTargetPosition;
}
```

В редакторі компонент виглядає наступним чином:

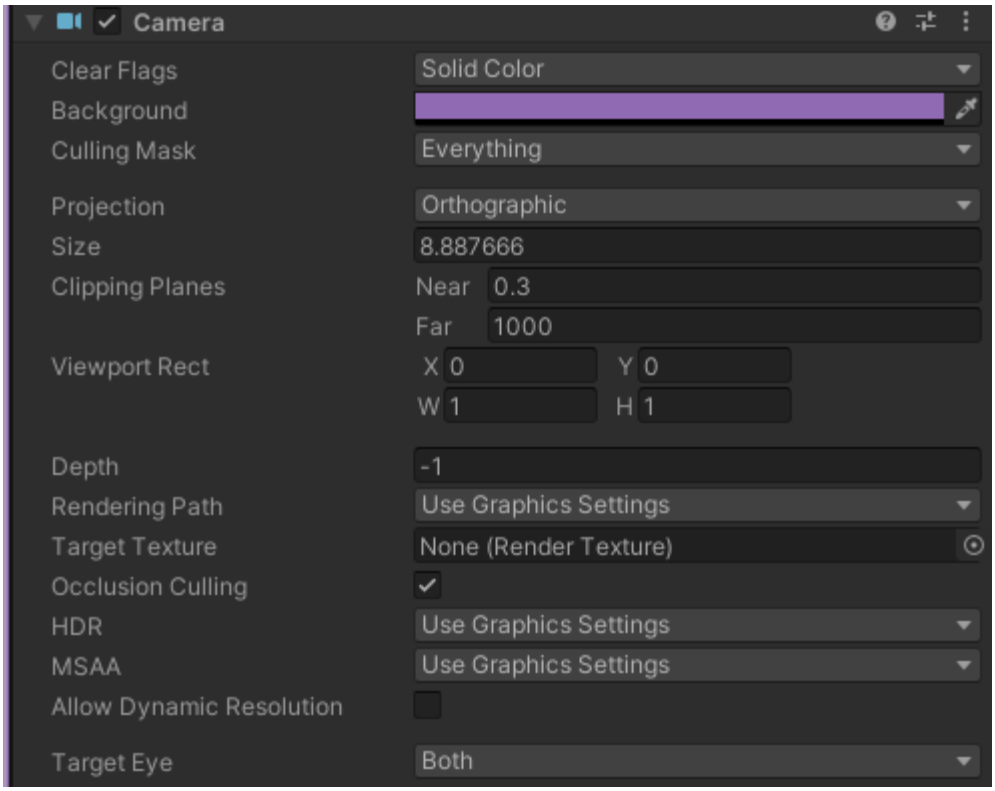

Рисунок 3.12

Також присутній скрипт **`ColorShift`**, який створю градієнтний фон для камери:

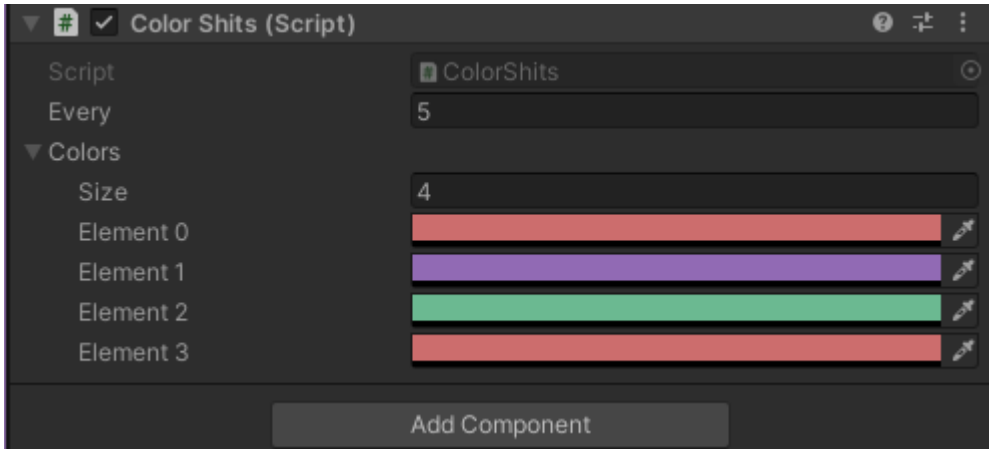

Рисунок 3.13

```
if (colorstep < every)
{
        lerpedColor = Color.Lerp(colors[i], colors[i + 1], colorstep);
        this.GetComponent<Camera>().backgroundColor = lerpedColor;
        colorstep += 0.01f;  //The lower this is, the smoother the transition, set
it yourself
```

```
}
else
    {
    colorstep = 0;
          if (i < (colors.Length - 2))
               {
                  i++;
                }
           else
               {
                  i = 0;
                 }
      }
```

За допомогою лінійної інтерполяції формується градієнтний перехід від одного кольора до наступного, зі збільшенням кроку. Як тільки крок стає рівним бажаній тривалості кольору, відбувається перехід до наступного кольору.

Як можна бачити з Рис.3.13 в редакторі присутня можливість закастомізувати будь-які кольори, використовуючи зручний вбудований колор-пікер, або ввести код кольору власноруч.

Отже, в цьому підрозділі було описано компонент 3С, який складається з камери, управління та персонажу. Скрипти та їх відображення в редакторі, яке дає змогу модифікувати значення та льоту.

3.5.2 Game World

До елементів ігрового світу можна віднести об’єкти, з якими взаємодіє персонаж та правила цієї взаємодії. До таких елементів можна віднести: платформи та алгоритми їх генерації, монетки та алгоритми їх генерації, генерація рівня та ігрова логіка.

Першим ігровим компонентом, який буде розглянуто – **`GameLogic`.** В ньому міститься однойменний скрипт. Як вже було зазначено, в попередньому розділі він

є пустим ігровим об’єктом, який містить в собі компонент-менеджер у вигляді скрипта. В ньому містяться головні функції, які відповідають за ігрові механіки.

Присутні *поля*:

`private bool m_IsGameStarted` – чи почалася гра (за замовчуванням - false) ;

`private bool m_IsGameOver` – чи закнічилася гра (за замовчуванням - false) ;

`private int m_Score` – показник рахунку гравця

Присутні *методи* встановлення статусу кінця гри:

```
public void SetGameOver()
        {
            if (m_IsGameOver)
            {
                return;
            }
            m_IsGameOver = true;
            if (m_OnGameOver != null)
            {
                m_OnGameOver.Invoke();
            }
        }
```

Метод підбирання монетки:

```
public void OnCollectCoin()
        {
            m_Score++;
            if (m_OnScoreChanged != null)
            {
                m_OnScoreChanged.Invoke();
            }
        }
```

А також присутні методи оновлення сцени та обробки доторкань.

Наступним ігровим елементом, який слід більш детально розгллянути, це **`Tile`** – платформа. Платформа представляє собою вертикальний, прямокутний паралелепіпед.(Рис.3.14)

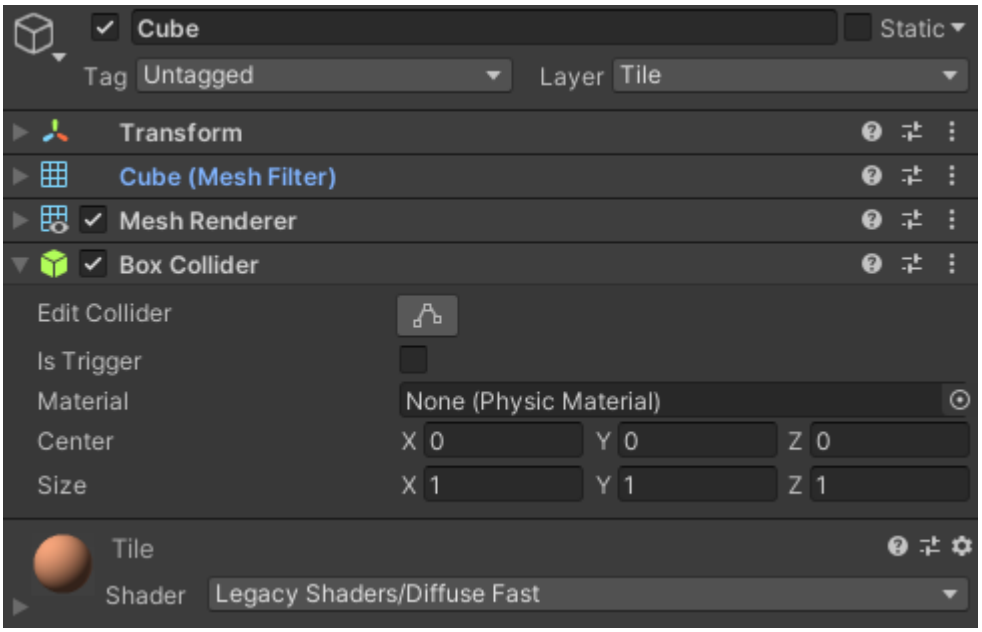

Рисунок 3.14

Складється він з фізичного колайдеру, для колізій, шейдеру та кубічної сітки, яка окреслює форму платформи (Рис.3.15).

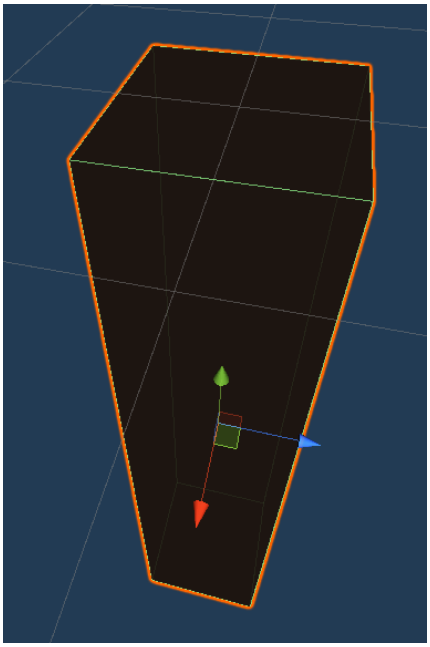

Рисунок 3.15

Найцікавіший аспект роботи – це генерація шляху з цих платформ. Як вже було зазначено найважливішим аспектом буде робота з пулом. Пул являє собою словник, де key: назва об’єкту, а value: список типу IPoolable, з якого виймаються та до якого додаються об’єкти:

```
private Dictionary<string, List<IPoolable>> m_Pool = new Dictionary<string, List<IPoolable>>();
```

Додавання до пулу

Спочатку об'єкти активується та реєструється. Під реєстрацією мається на увазі додавання сутності до черги платформ. Початкова кількість платформ дорівнює 25. Для того, щоб уникнути спустошування пулу, необхідно поставити умову, що `BufferPosition` > `MaxTilePerCreation`. Це робиться для того, щоб під час першої генерації перша лінія не використає всі доступні преінстанційовані платформи, які будуть необхідні для генерації наступних.

```
Main.Get<LevelGenerator>().RegisterEntity(this);
if (!m_Entities.Contains(entity) && entity is IPoolable)
```

```
{
    m_Entities.Add(entity);
}
```

Наступним викликається метод **Generate**. Він в свою чергу викликає метод **SpawnTile**, який створює об’єкт платформи та присвоює йому позицію за замовчуванням та зберігає його останні координати, на основі яких будує зразу задану кількість наступних платформу, які прилягають до першої. Необхідно більше детально розібрати процес створення об’єкта платформи.

Цим займається універсальний метод **SpawnEntity**, який приймає в якості параметра префаб об’єкту, яких необхідно створити (в даному випадку це платформа, але він також використовується для генерації монет). Він витягує об’єкт з пулу через метод **Pop** і повертає його, якщо він існує.

```
GameObject tileObject = SpawnEntity(Main.Get<LevelGeneratorConfig>().TilePrefab);
Entity entity = ObjectPooler.Pop<Entity>(entityPrefab.EntityName);
```

Об’єктом, що генерується стає останній в списку елемент по індексу:

```
int lastIndex = list.Count - 1;
IPoolable poolableObject = list[lastIndex];
list.RemoveAt(lastIndex);
```

Далі виконується метод **TryCreateTileLines**, який намагається побудувати лінію з платформ, перевіряючи координати останньої поставленої платформи. Лінія формується рандомно від одного до `MaxTilePerCreation`, який можна відредагувати в редакторі (за замовчуванням - 5).

Наступним виконується метод **`CreateTileLines`**, який спаунить платформу на оновленому місці останньої платформи. А також, змінює напрямок побудови платформ:

```
m_IsForwardDirectionNow = !m_IsForwardDirectionNow;
```

Напрямок побудови платформ також змінюється, коли досягається край видимості камери.

На схемі можна бачити спрощений процес витягання з пулу (Рис.3.16):

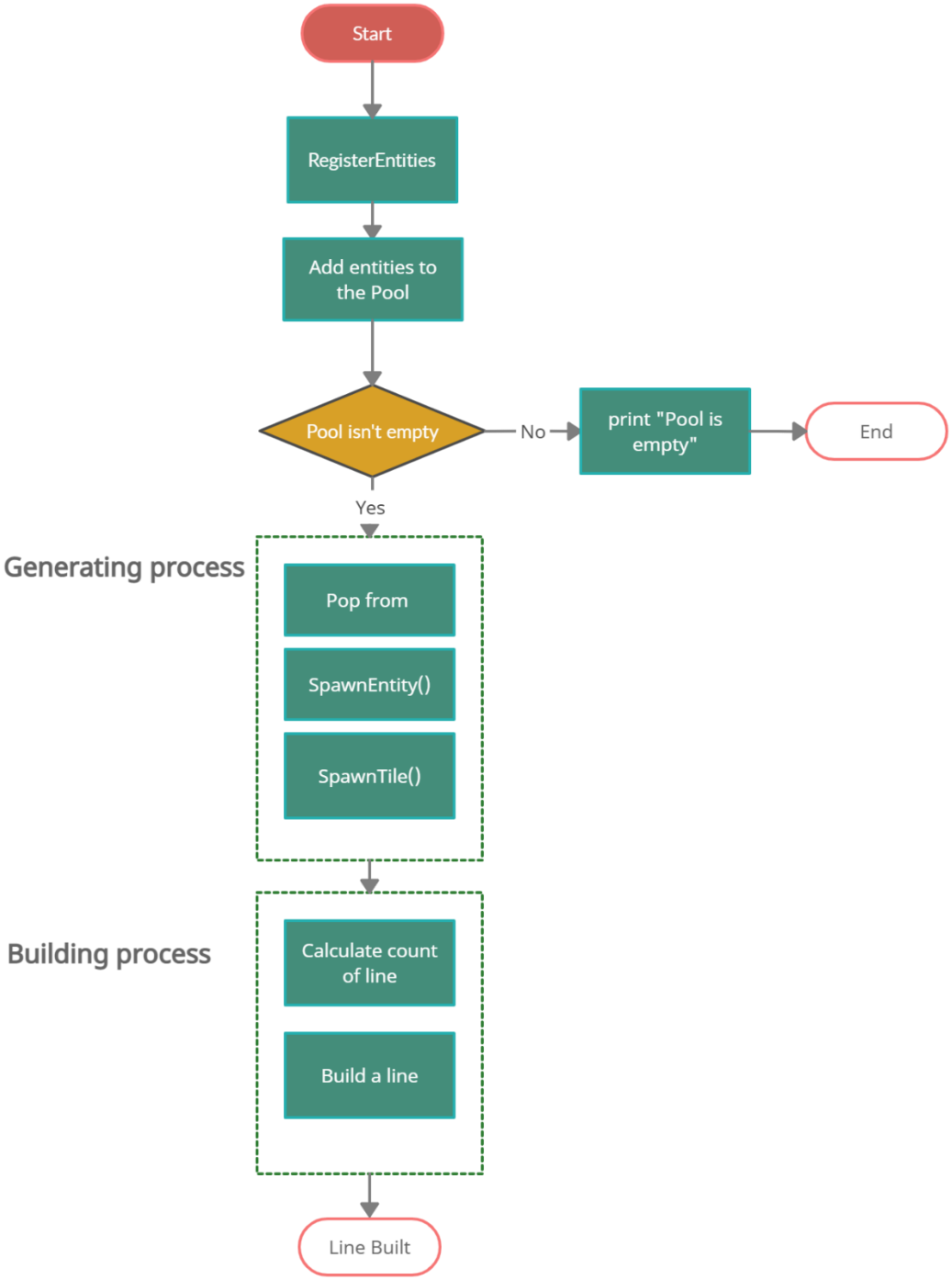

Рисунок 3.16

Витягання з пулу

Як тільки м’яч проходить через платформу і відстань від м’яча більша ніж значення змінної `DisappearPosition`, її координати по осі Y (по вертикалі у 3D

просторі) змінюються на -15, та за допомогою функції лінійної інтерполяції плавно падають вниз задану кількість часу:

```
Vector3 position = Transform.position;
            position.y = Mathf.Lerp(Main.Get<TileConfig>().TileMinPosition, 0.0f,
Main.Get<TileConfig>().TilePositionCurve.Evaluate(m_VerticalLerp));
            Transform.position = position;
```

Величину кроку можна задати використовуючи криву, яка відображає взаємозв'язок між часом, початковим значенням у вигляді 0 та бажаним кінцевим значенням у вигляді 1. Таким чином виглядає крива (Рис.3.17):

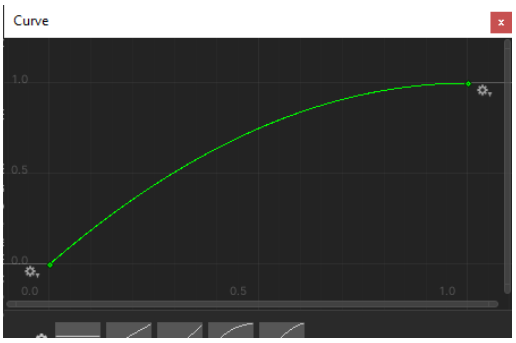

Рисунок 3.17

```
if (m_Falling)
            {
                if (m_VerticalLerp > 0.0f)
                {
                    m_VerticalLerp = Mathf.Clamp01(m_VerticalLerp - Time.deltaTime
/ Main.Get<TileConfig>().TileFallingTimeLength);
                    VisualUpdate();
                }
            }
```

Якщо часу, виділеного на падіння, буде занадто багато, а м’яч продовжить рухатися, то платформа буде примусово додана до пулу.

```
if (m_VerticalLerp == 0.0f)
     {
          ObjectPooler.Push(EntityName, this);
      }
```

Ці значення (час падіння, мінімальна позиція по осі Y) можна також кастомізувати в редакторі (Рис.3.18):

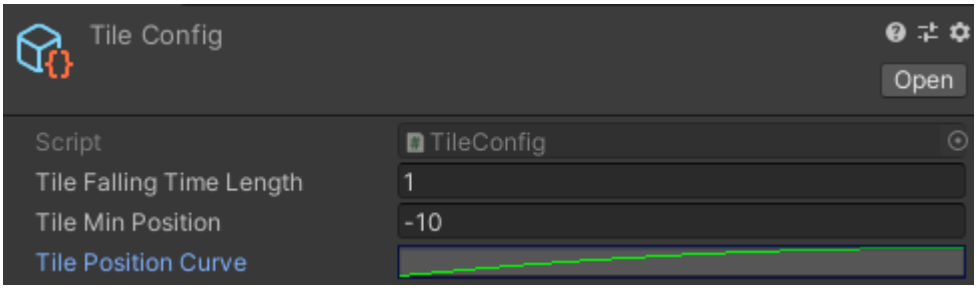

Рисунок 3.18

На відміну від генерації платформ за допомогою `Pop` метода, для додавання в пул використовується `Push` метод, який додає об'єкт на ігрову сцену:

```
m_Pool[entityName].Add(poolableObject);
poolableObject.OnPoolPushEvent();
```

Процес додавання платформи назад до пулу можна бачити на схемі нижче (Рис.3.19):

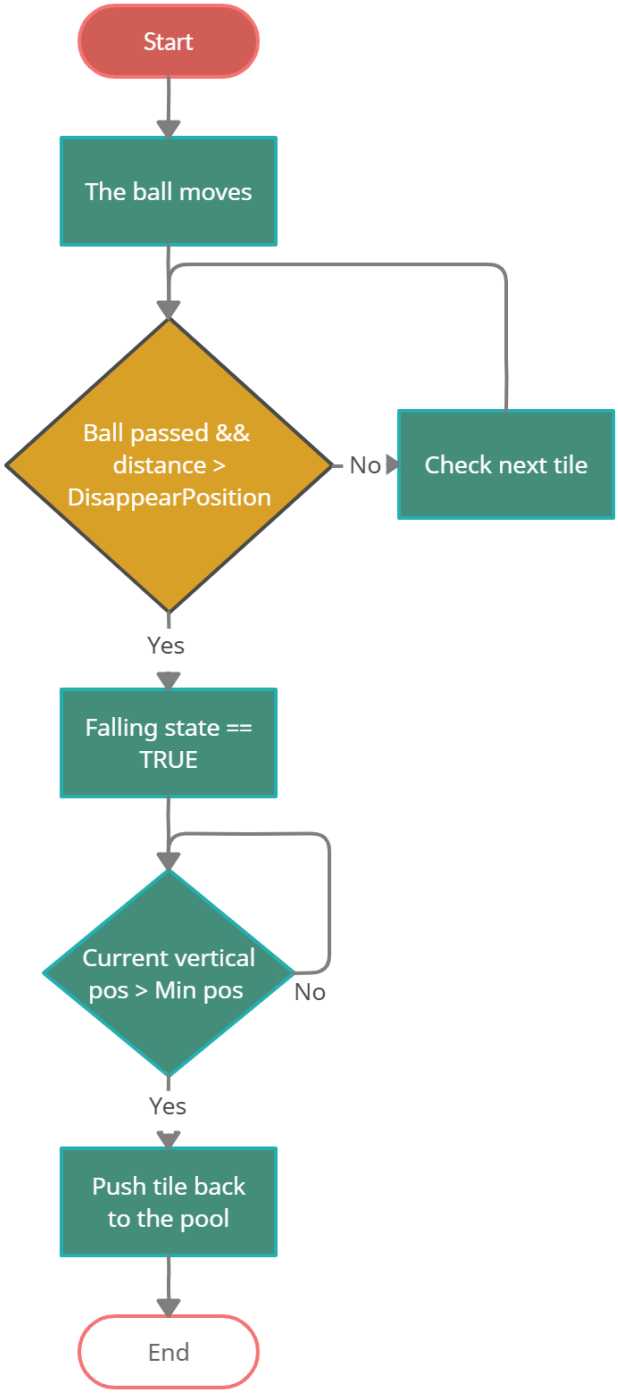

Рисунок 3.19

Наступним ігровим об’єктом є монетка – **`Coin`**. Як тільки перша платформа створюється, створюється і перша монетка, яку може підібрати гравець. В цілому, процес генерації монет схожий на аналогічний процес з платформами, адже метод, який додає сутності до пулу `SpawnEntity`, є універсальним, а отже монети також додаються і витягуються з пулу. Проте для того, щоб зробити процес спауну монет

більш випадковим, було створено абстрактний клас **CoinSpawnMethod** з методом, який перевизначається у двох похідних класах: **CoinSpawnMethodRandom**, **CoinSpawnMethodSequence.**

Перший метод додає монетки на випадкову позицію, якщо рандомне число було менше ніж заданий шанс:

```
return Random.value <= m_CoinSpawnChance;
```

Другий метод додає монети в ігровий світ постійно, виходячи з кількості вже побудованих платформ та індексу останної платформи:

```
int blockIndex = (m_TilesCount / m_CoinSpawnOrderCount) % m_CoinSpawnOrderCount;

m_TilesCount++;

m_TileLocalIndex = (m_TileLocalIndex + 1) % m_CoinSpawnOrderCount;
```

Монета є зарахованою у тому випадку, коли відбулася колізія з м’ячем.

В цьому підрозділі було детально описано роботу ігрових об’єктів, які відносять до елементів ігрового світу. В наступному підрозділі буде розібрано UI складову гри.

3.5.3 UI

UI – це користувацький інтерфейс, який представляє собою графічну структуру гри. В Unity розмістити UI елементи можна на канвасі, який є свого роду полотном. Канвас також є ігровим об’єктом, а всі графічні елементи, які на ньому розміщуються, унаслідуються від нього. Ієрархія канвасу моєї гри наступна (Рис.3.20):

Рисунок 3.20

Score Canvas складається з прозорої панелі, та текстового компоненту, який відображає поточну кількість очок, набраних гравцем. UI Canvas має в собі два тестових компоненти: Tap Text, Game Over. Другий за замовчуванням деактивований, адже відображає текст, який показується гравцю після його програшу (Рис.3.21).

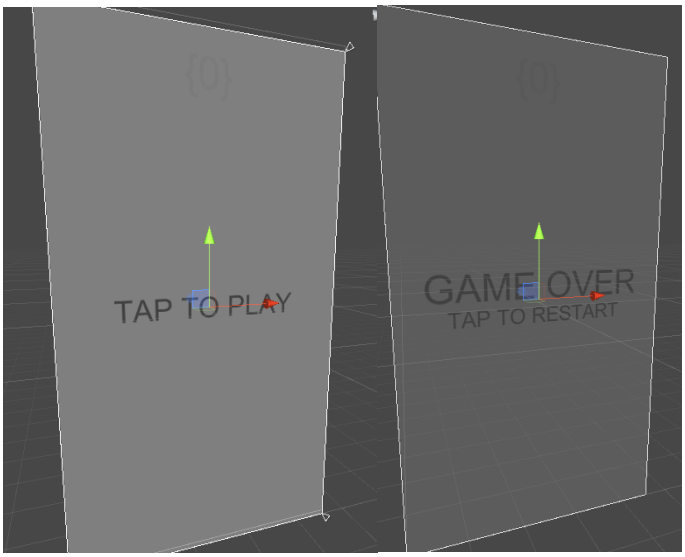

Рисунок 3.21

На компоненті TapTap Text присутній скрипт, який додає ефект мерехтіння напису:

```
float lerp = Mathf.Sin(Time.unscaledTime * Mathf.PI) * 0.5f + 0.5f;m_Graphic.color = Color.Lerp(m_ClearColor, m_OriginalColor, lerp);
```

3.6 Висновки до розділу

В цьому розділі було проведено огляд архітектури та складових компонентів програмної реалізації гри, яка складає собою дизайн-документ, який описує кожен ігровий об’єкт. Було приведено опис застосованої техніки оптимізації у вигляді з використанням пулу об’єктів. Це дозволило уникнути зайвих створень та знищень об’єктів та підвищити продуктивність роботи гри на мобільному девайсі.

ВИСНОВКИ

В цій кваліфікаційній роботі було описано концепції та ідею гри, яка відноситься до жанру аркадних мобільних ігор. Проведений мобільних застосунків показав, що гіперказуальні ігри демонструють тенденцію до росту в контексті долі на ринку. Ціллю проектування ігрового процесу стало охоплення якомога більшої кількості категорій гравців, яким прийшовся до вподоби інтуїтивний геймплей. Roller ball є характерним представником свого жанру, який не навантажує і відлякує складними ігровими механіками, які формують високий поріг входження аудиторії. Ключовим було створення атмосфери непереможності гри за рахунок інтенсивного ігрового процесу та особливостей генерації ігрового світу. Також було розглянуто інструменти розробки, обґрунтовано вибір ігрового движку, який став основою розроблюваної гри.

Основою роботи з 3D графікою є розуміння математичних засад, на яких побудована взаємодія ігрових об’єктів зі світом. В даній роботі було досліджено роботу з симуляціями природніх явищ, таких як гравітація та фізичні зіткнення, робота зі світлом та проекціями в контексті додавання їх на ігрову сцену. Векторна арифметика лежить в основі розуміння принципів роботи евклідового простору. В основі гри лежать різни види трансформацій об’єктів: визначення напрямків, поворотів, рухів. Завдяки широкій бібліотеці застосувань Unity вдалось уникнути написання недосконалих програмних компонентів, які б не раціонально використовували обчислювальні потужності смартфону.

Цілі та методи їх досягнення, опираючись на жанрові особливості, є стали опорами для дизайну архітектури програмного забезпечення. Компонентно-орієнтований підхід, що було вибрано для реалізації в даному проекті виявився найбільш оптимальним, адже вдалося досягти тісного групування споріднених функцій в межах одного класу, який відповідальний лише за одну сутність. В межах

такого підходу доцільно було використати скрипт-менеджер, який перевіряє на виконання основні ігрові умови. Враховуючи специфіку гри, та цільову платформу, було доцільно вдатися до застосування оптимізаційних технік, які б зробили ігровий досвід користувача більш приємним та комфортним. Серед таких, можна виділити обджект пулінг, який дозволив уникнути постійних створень та знищень об’єктів за необхідності та дав можливість їх повторного використання, що дуже важливо для мобільної гри. Результати наукового дослідження були опубліковані в збірнику матеріалів 8-ї Східно-Європейської конференції «Математичні та програмні технології Internet of Everything», яка проходила на базі кафедри програмних систем і технологій. Призначення кожного компоненту гри було розглянуто в окремому підрозділі, який окреслив їх поведінку та взаємозв’язки з іншими компонентами.

Отже, аркадні ігри займають велику нішу на сучасному ігровому ринку, а отже, користуються великим попитом. Ігри все більше впливають на сучасну людину, а через неї – на все суспільство.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Geig M. Unity Game Development in 24 Hours / Mike Geig., 2013. – (Third).
2. Robert Nystrom. Game Programming Patterns / Robert Nystrom.. – 354 с. – (Genever Benning).
3. Katie Salen Tekinbas. Rules of Play: Game Design Fundamentals / Katie Salen Tekinbas, Eric Zimmerman.. – 688 с. – (ISBN-13: 978-0262240451).
4. Unity User Manual 2020.3 (LTS) [Електронний ресурс] // Unity Technologies. – 2020. – Режим доступу до ресурсу: https://docs.unity3d.com/Manual/index.html.
5. Mark Placzek. Object Pooling in Unity [Електронний ресурс] / Mark Placzek // raywenderlich.com. – 2016. – Режим доступу до ресурсу: https://www.raywenderlich.com/837-object-pooling-in-unity.
6. Alfred. Infographic: Mobile Game Market Trends 2020 [Електронний ресурс] / Alfred // Dci. – 2020. – Режим доступу до ресурсу: https://www.dotcominfoway.com/blog/infographic-mobile-game-market-trends-2020/#gref.
7. DyadichenkoGA. Работа с EventSystem в Unity. Базовые вещи в работе с UI [Електронний ресурс] / DyadichenkoGA // Habr. – 2018. – Режим доступу до ресурсу: https://habr.com/ru/post/359106/.
8. Tracy Fullerton. Game Design Workshop / Tracy Fullerton., 2004. – 480 с.
9. Jesse Schell. The Art of Game Design / Jesse Schell., 2008. – 520 с. – (1st). – (ISBN-13: 978-0123694966).
10. Jeremy Gibson Bond. Introduction to Game Design, Prototyping, and Development / Jeremy Gibson Bond., 2014. – (Third). – (ISBN-13: 978-0321933164).

ДОДАТКИ

Додаток А

LevelGenerator.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

#pragma warning disable CS0649

namespace TapTap
{
    [DefaultExecutionOrder(ExecutionOrder.LevelGenerator)]
    public class LevelGenerator : Entity, IInstance, IAwakeEvent, IFixedUpdateEvent,
ISceneLateResetEvent
    {
        [SerializeField]
        private Transform m_Target;

        private Matrix4x4 m_Matrix;

        private Vector2 m_LeftRightBounds;

        private Vector3 m_LastTilePosition;

        private bool m_IsStartTileCreated;

        private bool m_IsForwardDirectionNow;

        private CoinSpawnMethod m_CoinSpawnMethod;

        private List<Entity> m_Entities = new List<Entity>();

        private List<Entity> m_TempPushList = new List<Entity>();

        private int TileSize
        {
            get { return Main.Get<LevelGeneratorConfig>().TileSize; }
        }

        private int StartTileSize
        {
            get { return Main.Get<LevelGeneratorConfig>().StartTileSize; }
        }

        public void RegisterEntity(Entity entity)
        {
            if (entity == null)
            {
                Debug.LogError("Can't register entity! Entity is null!");
                return;
            }

            if (!m_Entities.Contains(entity) && entity is IPoolable)
```

```
            {
                m_Entities.Add(entity);
            }
            else
            {
                Debug.LogError(string.Format("Entity {0} is not poolable!", entity), entity);
            }
        }

        public void ExcludeEntity(Entity entity)
        {
            int index = m_Entities.IndexOf(entity);
            if (index != -1)
            {
                m_Entities.RemoveAt(index);
            }
        }

        public void OnAwakeEvent()
        {
            m_CoinSpawnMethod = GetComponent<CoinSpawnMethod>();

            m_Matrix = Matrix4x4.TRS(Vector3.zero, Quaternion.Euler(new Vector3(0.0f, 45.0f,
0.0f)), Vector3.one);

            float boundSize = Main.Get<CameraControllerConfig>().HorizontalSpace / 2.0f;
            m_LeftRightBounds = new Vector2(-boundSize, boundSize);

            Generate();
        }

        private void CleanScene()
        {
            for (int i = 0; i < m_Entities.Count; i++)
            {
                m_TempPushList.Add(m_Entities[i]);
            }

            PushEntitiesToPool();
        }

        private void Generate()
        {
            SpawnTile();
            m_IsStartTileCreated = true;

            m_LastTilePosition = new Vector3(StartTileSize / 2.0f + TileSize / 2.0f, 0.0f,
0.0f);
            SpawnTile();

            TryCreateTileLines();
        }

        private void Reset()
        {
            m_LastTilePosition = Vector3.zero;
            m_IsStartTileCreated = false;
            m_IsForwardDirectionNow = false;
        }
```

```
        public void OnFixedUpdateEvent()
        {
            if (Main.Get<GameLogic>().IsGameOver)
            {
                return;
            }

            TryCreateTileLines();

            // Destroy objects behind.

            float targetZ = m_Matrix.inverse.MultiplyPoint3x4(m_Target.position).z;
            float bufferZ = targetZ - Main.Get<LevelGeneratorConfig>().BufferPosition;
            float disappearZ = targetZ - Main.Get<LevelGeneratorConfig>().DisappearPosition;

            for (int i = 0; i < m_Entities.Count; i++)
            {
                float posZ =
m_Matrix.inverse.MultiplyPoint3x4(m_Entities[i].Transform.position).z;

                if (m_Entities[i] is IDisappear && posZ <= disappearZ)
                {
                    (m_Entities[i] as IDisappear).Disappear();
                }

                if (posZ <= bufferZ)
                {
                    m_TempPushList.Add(m_Entities[i]);
                }
            }

            PushEntitiesToPool();
        }

        private void PushEntitiesToPool()
        {
            for (int i = 0; i < m_TempPushList.Count; i++)
            {
                ObjectPooler.Push(m_TempPushList[i].EntityName, m_TempPushList[i] as IPoolable);
            }

            m_TempPushList.Clear();
        }

        private void TryCreateTileLines()
        {
            float targetZ = m_Matrix.inverse.MultiplyPoint3x4(m_Target.position).z +
Main.Get<LevelGeneratorConfig>().BufferPosition;

            while (m_Matrix.inverse.MultiplyPoint3x4(m_LastTilePosition).z <= targetZ)
            {
                CreateTileLine();
            }
        }

        private void CreateTileLine()
        {
            int count = CalculateCountForNextLine();

            Vector3 tempPos = m_LastTilePosition;
```

```
            for (int i = 1; i < count + 1; i++)
            {
                m_LastTilePosition = m_IsForwardDirectionNow ? tempPos + Vector3.forward * i *
TileSize : tempPos + Vector3.right * i * TileSize;
                SpawnTile();
            }

            m_IsForwardDirectionNow = !m_IsForwardDirectionNow;
        }

        private int CalculateCountForNextLine()
        {
            int maxCount = 0;

            for (int i = 1; i < Main.Get<LevelGeneratorConfig>().MaxTilesPerDirection + 1; i++)
            {
                Vector3 checkPosition = m_LastTilePosition + (m_IsForwardDirectionNow ?
Vector3.forward : Vector3.right) * i * TileSize;

                if (IsOutOfBound(checkPosition))
                {
                    break;
                }
                else
                {
                    maxCount = i;
                }
            }

            if (maxCount < 1)
            {
                return 0;
            }

            return Random.Range(1, maxCount + 1);
        }

        private GameObject SpawnEntity(Entity entityPrefab)
        {
            Entity entity = ObjectPooler.Pop<Entity>(entityPrefab.EntityName);

            if (entity != null)
            {
                return entity.gameObject;
            }
            else
            {
                GameObject entityObject = Instantiate(entityPrefab.gameObject);
#if UNITY_EDITOR
                entityObject.name = System.Guid.NewGuid().ToString();
#endif
                return entityObject;
            }
        }

        private void SpawnTile()
        {
            m_LastTilePosition.y = 0.0f;

            GameObject tileObject = SpawnEntity(Main.Get<LevelGeneratorConfig>().TilePrefab);
```

```
            Transform tileTransform = tileObject.GetComponent<Transform>();
            tileTransform.position = m_LastTilePosition;
            tileTransform.rotation = Quaternion.identity;
            tileTransform.localScale = m_IsStartTileCreated ? new Vector3(TileSize, 1.0f,
TileSize) : new Vector3(StartTileSize, 1.0f, StartTileSize);

            if (m_IsStartTileCreated)
            {
                SpawnCoin();
            }
        }

        private void SpawnCoin()
        {
            if (m_CoinSpawnMethod != null && m_CoinSpawnMethod.CanSpawnCoin())
            {
                GameObject coinObject =
SpawnEntity(Main.Get<LevelGeneratorConfig>().CoinPrefab);

                Transform coinTransform = coinObject.GetComponent<Transform>();
                coinTransform.position = m_LastTilePosition;
                coinTransform.rotation = Quaternion.identity;
            }
        }

        private bool IsOutOfBound(Vector3 position)
        {
            if (m_Matrix.inverse.MultiplyPoint3x4(position + new Vector3(-TileSize / 2.0f, 0.0f,
TileSize / 2.0f)).x < m_LeftRightBounds.x)
            {
                return true;
            }

            if (m_Matrix.inverse.MultiplyPoint3x4(position + new Vector3(TileSize / 2.0f, 0.0f,
-TileSize / 2.0f)).x > m_LeftRightBounds.y)
            {
                return true;
            }

            return false;
        }

        public void OnSceneLateResetEvent()
        {
            Reset();

            CleanScene();

            Generate();
        }
    }
}
```

Додаток Б

ObjectPooler.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace TapTap
{
    public class ObjectPooler : IInstance
    {
        private Dictionary<string, List<IPoolable>> m_Pool = new Dictionary<string,
List<IPoolable>>();

        public static T Pop<T>(string entityName) // Shortcut
        {
            return Main.Get<ObjectPooler>().PopFromPool<T>(entityName);
        }

        public static bool Push(string entityName, IPoolable poolableObject) // Shortcut
        {
            return Main.Get<ObjectPooler>().PushToPool(entityName, poolableObject);
        }

        public bool TryRemoveFromCollection(string entityName, IPoolable poolableObject)
        {
            if (poolableObject == null || string.IsNullOrEmpty(entityName))
            {
                return false;
            }

            if (!m_Pool.ContainsKey(entityName))
            {
                return false;
            }

            int index = m_Pool[entityName].IndexOf(poolableObject);

            if (index == -1)
            {
                return false;
            }

            m_Pool[entityName].RemoveAt(index);

            return true;
        }

        public T PopFromPool<T>(string entityName)
        {
            if (string.IsNullOrEmpty(entityName))
            {
                return default(T);
            }

            if (!m_Pool.ContainsKey(entityName))
            {
                return default(T);
```

```
            }

            List<IPoolable> list = m_Pool[entityName];

            if (list.Count < 1)
            {
                return default(T);
            }

            int lastIndex = list.Count - 1;
            IPoolable poolableObject = list[lastIndex];
            list.RemoveAt(lastIndex);

            poolableObject.OnPoolPopEvent();

            return (T)poolableObject;
        }

        public bool PushToPool(string entityName, IPoolable poolableObject)
        {
            if (poolableObject == null)
            {
                Debug.LogError("Can't push object to pool! Object is null!");

                return false;
            }

            if (string.IsNullOrEmpty(entityName))
            {
                Debug.LogError("Can't push object to pool! <Entity Name> is null or empty!",
poolableObject as Object);

                return false;
            }

            if (!m_Pool.ContainsKey(entityName))
            {
                m_Pool.Add(entityName, new List<IPoolable>());
            }

            if (m_Pool[entityName].Contains(poolableObject))
            {
                return false;
            }

                m_Pool[entityName].Add(poolableObject);

                poolableObject.OnPoolPushEvent();

            return true;
        }
    }
}
```

Додаток В

Ball.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

#pragma warning disable CS0649

namespace TapTap
{
    public class Ball : Entity, IEnableEvent, IDisableEvent, IUpdateEvent, ISceneResetEvent
    {
        [SerializeField]
        private bool m_Bot;

        [Header("Render")]

        [SerializeField]
        private Transform m_Renderer;

        private int m_MoveDirectionIndex;

        private Vector3 m_FallingVelocity;

        private Collider[] m_Colliders = new Collider[10];

        private float BallSpeed
        {
            get { return Main.Get<BallConfig>().BallSpeed; }
        }

        private float BallRadius
        {
            get { return Main.Get<BallConfig>().BallRadius; }
        }

        private readonly Vector3[] m_MoveDirections =
        {
            Vector3.right,
            Vector3.forward
        };

        public void OnEnableEvent()
        {
            UpdateRenderer();

            Main.Get<GameLogic>().m_OnMouseClick += ChangeMoveDirection;
        }

        public void OnDisableEvent()
        {
            if (Main.Get<GameLogic>() != null)
            {
                Main.Get<GameLogic>().m_OnMouseClick -= ChangeMoveDirection;
            }
        }
```

```
        private void ChangeMoveDirection()
        {
            m_MoveDirectionIndex = (m_MoveDirectionIndex + 1) % m_MoveDirections.Length;
        }

        public void OnUpdateEvent()
        {
            if (!Main.Get<GameLogic>().IsGameStarted)
            {
                UpdateRenderer(); // HACK: Scene Reset

                return;
            }

            if (Main.Get<GameLogic>().IsGameOver)
            {
                if (Transform.position.y < -1000.0f) // TODO: Min height position
                {
                    return;
                }

                m_FallingVelocity += Physics.gravity * Time.deltaTime;
                Transform.position += m_FallingVelocity * Time.deltaTime;
            }
            else
            {
                CollisionCheck();

                if (!IsOnGround(GetBallCenterPosition()))
                {
                    Main.Get<GameLogic>().SetGameOver();
                }
            }

            Transform.position += m_MoveDirections[m_MoveDirectionIndex] * BallSpeed *
Time.deltaTime;

            UpdateRenderer();

            if (m_Bot)
            {
                BotLogic();
            }
        }

        private void BotLogic()
        {
            if (!IsOnGround(GetBallCenterPosition() + m_MoveDirections[m_MoveDirectionIndex] *
Main.Get<LevelGeneratorConfig>().TileSize * 0.5f))
            {
                ChangeMoveDirection();
            }
        }

        private void UpdateRenderer()
        {
            m_Renderer.localPosition = new Vector3(0.0f, BallRadius, 0.0f);
            m_Renderer.localScale = Vector3.one * BallRadius * 2.0f;
        }
```

```
        private Vector3 GetBallCenterPosition()
        {
            return Transform.position + Vector3.up * BallRadius;
        }

        private bool IsOnGround(Vector3 origin)
        {
            Ray ray = new Ray(origin, Vector3.down);

            return Physics.Raycast(ray, Main.Get<BallConfig>().GroundLayer);
        }

        private void CollisionCheck()
        {
            int count = Physics.OverlapSphereNonAlloc(GetBallCenterPosition(), BallRadius,
m_Colliders);

            for (int i = 0; i < count; i++)
            {
                if (m_Colliders[i].CompareTag(Entity.Tag))
                {
                    ICollectable collectable = m_Colliders[i].GetComponent<ICollectable>();

                    if (collectable != null)
                    {
                        collectable.Collect();
                    }
                }
            }
        }

        public void OnSceneResetEvent()
        {
            Transform.position = Vector3.zero;

            m_MoveDirectionIndex = 0;

            m_FallingVelocity = Vector3.zero;
        }
    }
}
```

Додаток Г

GameLogic.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

namespace TapTap
{
    [DefaultExecutionOrder(ExecutionOrder.GameLogic)]
    public class GameLogic : Entity, IInstance, ISceneResetEvent
    {
        private bool m_IsGameStarted = false;

        private bool m_IsGameOver = false;

        private int m_Score;

        public UnityAction m_OnMouseClick;

        public UnityAction m_OnGameStarted;

        public UnityAction m_OnGameOver;

        public UnityAction m_OnScoreChanged;

        public bool IsGameStarted { get { return m_IsGameStarted; } }

        public bool IsGameOver { get { return m_IsGameOver; } }

        public int Score { get { return m_Score; } }

        public void SetGameOver()
        {
            if (m_IsGameOver)
            {
                return;
            }

            m_IsGameOver = true;

            if (m_OnGameOver != null)
            {
                m_OnGameOver.Invoke();
            }
        }

        public void DoMouseClick()
        {
            if (!m_IsGameStarted)
            {
                m_IsGameStarted = true;

                if (m_OnGameStarted != null)
                {
                    m_OnGameStarted.Invoke();
                }
```

```
                return;
            }

            if (m_IsGameOver)
            {
                return;
            }

            if (m_OnMouseClick != null)
            {
                m_OnMouseClick.Invoke();
            }
        }

        public void DoReset()
        {
            Main.Get<UpdateSystem>().ResetScene();
        }

        public void OnCollectCoin()
        {
            m_Score++;

            if (m_OnScoreChanged != null)
            {
                m_OnScoreChanged.Invoke();
            }
        }

        public void OnSceneResetEvent()
        {
            m_IsGameStarted = false;

            m_IsGameOver = false;

            m_Score = 0;

            if (m_OnScoreChanged != null)
            {
                m_OnScoreChanged.Invoke();
            }
        }
    }
}
```