ДВНЗ «Донецький національний технічний університет»

Факультет комп'ютерно-інформаційних технологій та автоматизації
(повне найменування інституту, назва факультету)

Кафедра прикладної математики та інформатики
(повна назва кафедри)

«До захисту допущено»
в.о. завідувача кафедри

_____ Наталія МАСЛОВА
(підпис) (ім'я та прізвище)

"___"_____2022р.

Випускна кваліфікаційна робота

магістра
(освітньо-кваліфікаційний рівень)

на тему: «Дослідження методів розробки ігрового додатку засобами Unity»

Виконала: студентка 2 курсу, групи КНм-21
(шифр групи)

спеціальності (напряму підготовки) 122 Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)

Кроха А.В _____
(прізвище та ініціали) (підпис)

Керівник доц. каф. ПМІ, к.т.н., доц. Ірина НАЗАРОВА _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент доц. каф. ЕТ, к.т.н., доц. Віталій ШАМАЄВ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент доц. каф. ЕІ, к.ф.-м.н., доц. Олена ЛЮБИМЕНКО _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Нормоконтроль
к.т.н., доц., доц.каф.ПМІ

_____ Ірина Назарова
(підпис)

Засвідчую, що у цій випускній кваліфікаційній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Луцьк – 2022 р.

ДВНЗ «Донецький національний технічний університет»
Кафедра прикладної математики і інформатики
Освітній ступінь Магістр
Спеціальність 122 Комп'ютерні науки

ЗАТВЕРДЖУЮ
в.о. завідувача кафедри ПМІ
_____ /Н. О. Маслова/
«____»_____2022 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Крохі Анні Василівні

1. Тема роботи: «Дослідження методів розробки ігрового додатку засобами Unity»
Керівник роботи доцент каф. ПМІ, к.т.н., доцент Назарова І.А

затверджені наказом від " 28 " вересня 2022 року № 446

2. Строк подання студентом роботи "05" грудня 2022 року

3. Вихідні дані до роботи результати науково-дослідної роботи, матеріали переддипломної практики, інтернет-ресурси, документація Unity.

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1) аналітичний огляд, 2) процес проектування та інструменти 3) зміст гри , 4) опис роботи та тестування

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) графічний матеріал, у кількості, достатній для розкриття сутності і демонстрації основних результатів

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 30 вересня 2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Приміт ка
	Огляд літератури з предметної області	01.09.2022 – 16.10.2022	
	Збір інформації, вивчення статистики	17.10.22-31.10.2022	
	Написання 1-го розділу	02.11.2022 - 16.11.2022	
	Написання 2-го розділу	17.11.2022 - 25.11.2022	
	Проектування та тестування програмного продукту	25.12.2022 – 01.12.2022	
	Написання 3-го розділу	05.12.2022 – 10.12.2022	
	Написання 4- го розділу	10.12.2022 – 12.12.2022_	
	Оформлення пояснювальної записки	02.12.22-13.12.22	
	Нормоконтроль пояснювальної записки, її перевірка антиплагіатною системою	13.12.22-19.12.22	
	Захист роботи	22.12.22	

Студент _____ Кроха А.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Назарова І.А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кроха А.В. Дослідження методів розробки ігрового додатку засобами Unity / Випускна кваліфікаційна робота на здобуття освітнього ступеня «магістр» за спеціальністю 122 «Комп'ютерні науки». – ДВНЗ ДонНТУ, Луцьк, 2022.

Випускну магістерську роботу присвячено дослідженню існуючих ігрових движків, методів ігрової індустрії та розробці гри на основі ігрового двигуну Unity.

Актуальність роботи: геймдев в наш час є досить популярною і швидко розвивинутою індустрією. Для цієї індустрії потребується різноманіття професій, у тому числі одну із головних ролей відіграють програмісти.

Об'єктом дослідження є методи та алгоритми ігрової індустрії.

Предмет дослідження – розробка гри та дослідження роботи ігрових двигунів на прикладі Unity.

Мета роботи полягає у проектуванні і розробці комп'ютерної гри у жанрі Tower Defence.

Завдання роботи: дослідити область ігрової індустрії, оглянути види та принципи роботи ігрових движків і розробити гру на Unity.

КЛЮЧОВІ СЛОВА: Unity, ігрові двигуни, ігрова індустрія, ігрові двигуни, розробка ігор, гра на Unity, Tower Defence.

Список публікацій здобувача:

1. Кроха А.В. Розвиток ігрової індустрії та її перспективи // III Міжнародна студентська наукова конференція «Цифровізація науки та сучасні тренди її рохвитку» / зб. доповідей матеріали III Міжнародної студентської наукової конференції, м. Умань, 11 листопада, 2022 рік / ГО «Молодіжна наукова ліга». – Вінниця: ГО «Європейська наукова платформа», 2022. – 304с.

Krokha A.V. Researching of methods for developing a game on using Unity / Graduation qualification work for obtaining an educational degree "Master" in the specialty 122 "Computer Science". – DVNZ DonNTU, Lutsk, 2022.

The final master's thesis is devoted to researching existing game engines, researching the game industry and developing a game using the Unity game engine.

The relevance of the work: nowadays, game development is a fairly popular and rapidly developing industry. This industry requires a variety of professions, including one of the main roles played by programmers.

The object of research is the game engine Unity and the game industry.

The subject of research is game development and study of game engines using Unity as an example.

The purpose of the work is to design and develop a computer game in the Tower Defense genre.

Job tasks: research the field of the game industry, review the types and principles of game engines and develop a game on Unity.

KEYWORDS Unity, Game Engines, Game Industry, Game Engines, Game Development, Unity Game, Tower Defense.

List of publications of the acquirer:

1. Krokha A.V. The development of the gaming industry and its prospects // III International Student Scientific Conference "Digitalization of Science and Modern Trends in its Development" / coll. reports, materials of the III International Student Scientific Conference, Uman, November 11, 2022 / NGO "Youth Scientific League". — Vinnytsia: NGO "European Scientific Platform", 2022. – 304

ЗМІСТ

ВСТУП

Сьогодні ігри – це панівна багатомільярдна індустрія, яка конкурує з Голлівудом за масштабом та популярністю. І програмне забезпечення, яке лежить в основі цих, тепер уже повсюдно поширених тривимірних світів, називається ігровими двигунами. До них відносяться Unreal Engine 4 (компанія Epic Games), Source (компанія Valve), CRYENGINE® 3 (компанія Crytek), Frostbite™ (компанія DICE (Electronic Arts)) та Unity. Ці ігрові двигуни стали повнофункціональними засобами розробки програмного забезпечення багаторазового використання, які можуть ліцензуватися та застосовуватися для розробки практично будь-якої гри.

У той час як ігрові двигуни істотно різняться в деталях архітектури та реалізації, завдяки як двигунам з громадською ліцензією, так і їх комерційним аналогам з'явилися відомі патерни. Практично всі ігрові двигуни містять подібні набори головних компонентів, включаючи двигун рендеринга, двигун колізій і фізики, об'єктну модель ігрового світу, системи анімації, аудіо, штучного інтелекту і т. д. У кожному з цих компонентів також починає з'являтися відносно невелика кількість варіантів дизайну, поступово стають стандартними.

Існує безліч книг, які детально описують окремі підсистеми ігрових движків, наприклад тривимірну графіку.

Індустрія розваг є однією із найпопулярніших напрямків. Індустрія розваг це фільми, серіали, музика і звісно туда входять комп'ютерні та мобільні ігри. Кожен напрямок має вплив один на одного.

На сьогоднішній день гейм індустрія має великий вплив на життя людей та на інші індустрії розваг. Взагалі індустрія розваг зараз грає не останню роль в повсякденному житті. Гейм Індустрія має величезний вплив на інші напрямки. По сюжетам ігр регулярно створюються фільмі і сериали.

Створення якогось великого проекту-гри це дуже багато ресурсів, це велика кількість спеціалістів з різних сфер. Для гри потрібен сюжет, музика, графіка і т.д. Зараз створення гри не такий складний процес завдяки різноманітності інструментів таких як ігрові движки. Раніше ігрові двигуни були лише під компаніями які створювали ігри і не мали вільного доступу. На даний момент є два найпопулярніших ігрових двигуна у вільному доступі - Unreal Engine і Unity. Кожен з них має свої особливості застосування та специфіку. Але ці інструменти значно полегшують роботу гейм девелоперів, які не мають багато ресурсів на створення якогось дуже масштабного проекту. Тож створення ігор стало більш доступним та приємним. Кожен може спробувати створити свою власну гру. З'ясувалось, що усі ігри мають спільні моменти такі як: графіка, фізика, ігрові механіки. Тому почали створюватися бібліотеки для ігор.

Ігрова індустрія є актуальна і набуває все більше популярності і розвитку. Річний обсяг геймдев-індустрії оцінюють у 100 мільярдів. Аудиторія геймерів є більш ніж 2.6 мільярдів (населення Землі становить приблизно 7 мільярдів), тобто це майже половина населення Землі, а з роками і розвитком технологій аудиторія буде рости в геометричній прогресії. Гейм індустрія в Україні не є дуже розвинутою, вона застрягла на критичному рівні. Масштабні іноземні компанії мають таку практику як відкривати свої офіси, наприклад, компанія Ubisoft створювала офіс в Одесі. Але власних компаній з геймдеву невелика кількість в Україні. Для створення великого проекту потрібно багато різноманітних спеціалістів і список знизу є неповним з тих, що потребується. Наприклад, для однієї з найпопулярніших ігор Grand Theft Auto було задіяно десь 1000 працівників, а обіг бюджету становив навіть більше ніж у фільма, який нагородили за найкращі спецефекти.

Приблизний список спеціалістів для великого проекту:

-програмісти архітектури гри;

-програмісти ігрових фішок і режимів;

-програмісти інструментів;

-левел дизайнери;

-скриптери;

-моделери;

-аніматори;

-художнки;

-творці концепт арту;

-творці текстур;

- UX/UI дизайнери;

-ліцеві аніматори;

-саунд дизайнери;

-сценаристи;

- локалізатори;

- актор мокапу;

- актор озвучки;

- тестувальники;

- менеджери проекту;

- продюсери.

Ігрові платформи можно віділити на:

-персональні комп’ютери на базі Windows, Linux, Mac/OS X;

- ігрові консолі;

-мобільні пристрої;

-універсальні веб-платформи, соціальні мережі;

-аркадні автомати;

-інноваційні платформи віртуальної реальності.

1 ПРЕДМЕТНА ОБЛАСТЬ. ІГРОВІ ДВИГУНИ

1.1 Історія ігрових двигунів

Коли почали створюватися ігри стало зрозуміло, що кожна гра має спільні компоненти, навіть при різних апаратних платформах. А перші ігри були на величезних ігрових автоматах.

Спільна функціональність ігор - графічні рішення, розрахунки фізики і інше стали формуватися в окремі бібліотеки і лише потім з бібліотек перетворилися в ігрові двигуни. Ці складності були пов’язані з тим, що були великі відмінності програмно-апаратних платформ і невизначеність у самих іграх, бо жанри і типи ігор ще треба було створити, при тому що перші ігри були текстовими. Саме через перші адвенчури і платформери з’явилися перші ігрові двигуни, особливо з розвитком графіки - гарним прикладом можно назвати Adventure Game Interpreter (AGI). При розробці King’s Quest в 1984 році програмісти Sierra On-Line зіштовхнулися з незручностями низькорівневої розробки і розробили набір рішень, яким став AGI. Всього на ньому було випущено 14 різних ігор за 5 років на 7 різних платформах, тому поняття “кросплатформеність” було важливим вже в ті часи. Однак перші двигуни були вузькопрофільними під якийся конкретний жанр.

Знаменним моментом став вихід гри Doom від компанії id Software. Хоча при її розробці використовувались наробки двигуна Wolfenstein 3D, з точки зору можливостей то був справжній технологічний прорив. В той час відеопроцесори були не спроможні ефективно працювати з тривимірною графікою, світлом, затінням, наложенням текстур і іншого самостійно.

Тому Doom engine (перша версія id Tech) була не насправді тримірним, а псевдо тривимірним. Важливо що технічна складаюча, цієї гри завдала

стандарт для того, що могло називатися ігровим двигуном. А саме движок Doom був модульним, який складав із себе набір із підсистем, в якому кожна є чітко виділений слой, який відповідав за обробку своєї порції даних. У результаті використання його у різних іграх стало набагато простіше.

Термін «ігровий двигун» з’явився у середині 1990-х в контексті комп'ютерних ігор жанра шутер від першого обличчя, схожих на популярну в той час Doom. Архітектура програмного забезпечення Doom було побудоване таким чином, що представляла собою розумне і добре виконання розподілення центральних компонентів ігри (наприклад, підсистеми тривимірної графіки, розрахунку зіштовхнення об’єктів, звукої та інших) і графічних ресурсів, ігрових світів, формуюче досвід ігрока, ігрові правила та інше. Як слід, це отримало певну цінність за рахунок того, що почали створюватися ігри з мінімальними змінами, коли при наявності ігрового двигуна компанії створювали нову графіку, зброю, персонажів, правил ігри та таке інше.

Розподілення між грою та ігровим двигуном часто невизначене. Деякі двигуни мають разумне та ясне розділення, в той же час інші практично неможливо відділити від ігри. Наприклад, в грі двигун може «знати» про те, як мальовати дугу, в той же час інший двигун може працювати з іншим рівнем абстракції, і в ньому дуга буде приватним випадком параметрів викликаних функцією. Одним з ознаків ігрового двигуна є застосування архітектури управління даними. Це визначається тим, що якщо гра містить жорстко фіксовані дані, які впливають на логіку, правила гри, малювання об’єктів і тому подобне, то становиться складно застосовувати дане програмне забезпечення в різних іграх.

Більшість ігрових двигунів розроблено і налаштоване для того, щоб запустити певну гру на певній платформі. І навіть найбільш узагальнені багатоплатформені двигуни підходять для побудування ігор певного жанру, наприклад, шутерів першого обличчя або гонок. В даному контексті можна

більш акуратно сказати, що ігровий двигун становиться не оптимальним при його застосуванні не для тої гри або тої платформи, для якої разроблений. Даний ефект проявляється від того, що ПЗ представляє собою набір компромісів, заснованих на тих припущеннях, якою повина бути гра. Наприклад, проектування рендерінгу всередині будівель призведе до того, що двигун, скоріш за все, не буде таким же добрим для відкритих просторів. В першому випадку двигун може використовувати BSP-дерево для відмальовки об’єктів, близьких до камери. В той же час для відкритих просторів можуть використовуватися менш точні способи, а також більш активно застосовуються технології відмальовки з різною ступінню деталізації, коли більш далекі об’єкти промальовуються менш чітко, так як займають меншу кількість пікселів.

Ігровий двигун представляє із себе своєрідну вузькоспеціалізовану операційну систему, оскільки включає всі модулі останнім. До нього входять: система управління пам'яттю, графічна підсистема, система вводу, штучний інтелект, редактор ігрових рівнів та інше. Крім того ядро двигуна може демонструвати особливий підхід до роботи з файлами - файлову систему, а також засоби роботи з багатопоточністю. Сучасні ігрові двигуни також включають інтерпретатор скриптової мови, який розрахований на написання ігрової логіки, а нерідко і повністю візуальний її редактор. Його використання дозволяє утриматися від написання низькорівневих команд та інструкцій, а зосередити увагу на геймплеї. На цьому складаючі двигуна не обмежуються, їх може бути як більше так і менше.

Ігровий двигун впершу чергу створюєтьсяу цілях зпрощення та прискорення розробки. Тому включає засоби для створення ігрового світу-левел-моделінгу, імпорту об’єктів, текстурування, завантаження та анімації персонажів, створення візуальних ефектів, настройки фізики та іншого. Другою важливою ціллю розробки движка є кросплатформеність або платформна незалежність розробляємої гри. Тобто можливість її запуску з

мінімально можливими змінами. Зовсім без змін зробити запуск гри не вдасця через апаратні відмінності, у тому числі: розмір екрану, засоби та способи управління та інше.

Ймовірно, архітектура, керована даними, – це те, що відрізняє ігровий движок від програмного забезпечення, яке є грою, але не є движком. Коли гра містить жорстко запрограмовану логіку або правила гри або задіює спеціальний код для відображення певних типів ігрових об'єктів, стає важко або неможливо повторно використовувати це програмне забезпечення для створення іншої гри. Ймовірно, нам слід зарезервувати термін «ігровий двигун» для програмного забезпечення, яке розширюється і може застосовуватися як основа для багатьох ігор без значних модифікацій.

Зрозуміло, що таке визначення не має на увазі поділу на чорне та біле. Шкала багаторазового використання, на якій знаходиться кожен двигун, представлена на рисунку 1.1.

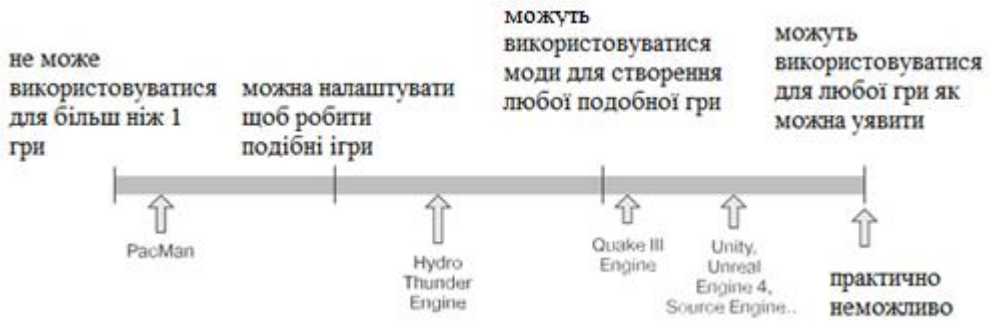

Рисунок 1.1 – Розташування відомих ігор/рухів на шкалі багаторазового використання

Можна подумати, що ігровий движок повинен бути чимось схожим на Apple QuickTime або Microsoft Windows Media Player: бути універсальним програмним забезпеченням, здатним відтворювати практично будь-який ігровий контент, який тільки можна собі уявити. Однак, цей ідеальний варіант ще не досягнутий і, ймовірно, ніколи не буде досягнутий. Більшість ігрових

двигунів розроблені та ретельно налаштовані для запуску конкретної гри на конкретній апаратній платформі. І навіть найуніверсальніші крос-платформні движки дійсно підходять тільки для створення ігор в одному певному жанрі, таких як шутери від першої особи або гонки. Можна з упевненістю сказати, що чим більш універсальним є ігровий двигун або компонент проміжного програмного забезпечення, тим меншим він є оптимальним для запуску конкретної гри на конкретній платформі.

Так відбувається тому, що розробка будь-якого ефективного компонента програмного забезпечення незмінно тягне за собою компроміси, засновані на припущеннях про те, як використовуватиметься програмне забезпечення, та/або про цільове обладнання, на якому воно працюватиме. Наприклад, двигун рендерингу, створений для роботи із закритими приміщеннями, ймовірно, не дуже підійде для рендерингу великих відкритих просторів. Двигун для приміщень може задіяти систему порталів або дерево з двійковим розбиттям простору (binary space partitioning, BSP), щоб гарантувати, що предмети не будуть перетинатися стінами або об'єктами, що знаходяться ближче до камери. А двигун для відкритих просторів може використовувати менш точний механізм накладання або взагалі не використовувати його. Але він, ймовірно, активно задіює техніки рівня деталізації (level-of-detail, LOD), щоб гарантувати, що віддалені об'єкти відображаються мінімальною кількістю трикутників, а для об'єктів, що знаходяться близько до камери, використовується мережа трикутників, що реалізує більшу роздільну здатність.

Поява більш швидкого комп'ютерного обладнання та спеціалізованих відеокарт поряд з дедалі ефективнішими алгоритмами рендерингу та структурами даних поступово стирає різницю між графічними движками для різних жанрів. Наприклад, тепер можна використовувати двигун для шутера від першої особи для створення стратегічної гри. Однак компроміс між узагальненням та оптимізацією все ще існує. Гру завжди реально зробити

більш вражаючою, налаштувавши двигун під вимоги та обмеження конкретної ігрової та/або апаратної платформи.

Розвиток ігрових двигунів відбувався разом чи під впливом розвитку апаратних і програмних платформ, разом чи з появою нових ігрових жанрів і зі зміною смаку у користувачів. Коротше кажучи, з розвитком ігрової індустрії в цілому.

1.2 Генезис графічних систем

У середині 90-х після появи відеопроцесорів, здатних обробляти тривимірну графіку стали з’являтися програмні інтерфейси, які спрощували її розробку. У слід за кросплатформенним Open GL у складі DirectX вийшов Direct3D для Windows. Ці два візуалізатора набагато років випередили способи графічного виводу в іграх.

У 1996 році вийшла гра Quake на Quake engine. Цей двигун зробив величезний вплив на ігрову індустрію.

Нижче приведено малюнок дерево двигунів заснованих на Quake.

Майже до кінця десятиріччя наринку проміжкового програмного забезпечення для ігор майже одноосібно завдавала ритм id Software.

Однак в 1998 році компанія Epic Games випустила успішну гру Unreal на однойменному двигуні – зі справжнім технічним проривом по рівню графіки. Головним програмістом движку став засновник Epic Тім Суіні. Тім на рівни з Кермаком є найбільш визначними фігурами в історії ігрових двигунів ігрової індустрії – і Unreal Engine у його 3 і 4 версіях дуже популярні і зараз. Рік потому от Epic вийшла ще більш популярна гра Unreal Tournament.

В той самий час конкуруюча компанія-розробник – id Software випустила мультиплеєрну гру Quake 3 Arena (id Tech 3 двигун), рівно як Unreal Tournament включала мережеві баталії.

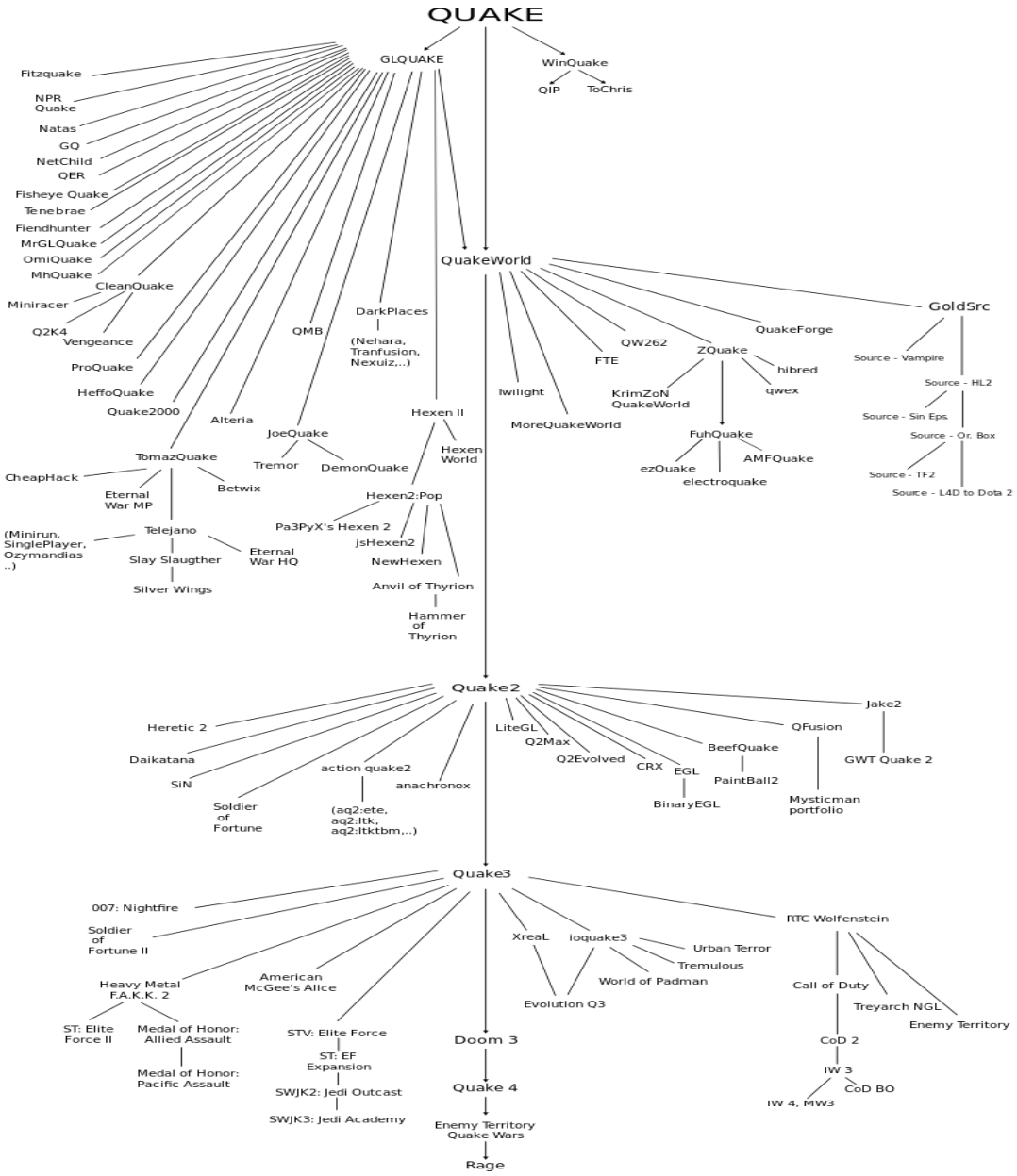

Рисунок 1.2 – Дерево двигунів, заснованих на Quake Engine

Ці дві гри стали фундаментальними в індустрії визначивши її розвиток на роки вперед. На ринку не було так багато геймерів. Тому продукція не була дуже дорогою і флагманські двигуни ліцензувалися тільки великими компанями розробників.

Ситуація почала змінюватися в середині першого десятиліття 21-го віку. Тоді, на ринку у вільному доступі стало з’являтися більша кількість засобів для розробки ігор. Бізнес проміжного ПЗ став набирати оберти.

Спочатку ринок високорівневої прослойки над графічним API. У той же час відрізняючись від ігрових повною відсутністю внутроігрових редакторів.

Потім на ринок прийшли повноцінні ігрові двигуни по цінам доступних для невеликої інді-команди розробників, серед них: Torque 3D, Unity 3D і багато інших. Навіть зарекомендовані як флагманські двигуни - наприклад, Cry Engine від Crytek і раніше згаданий Unreal Engine - стали використовувати більш доступну цінову політику і стали доступні починаючим розробникам.

Важливим трендом ігрової індустрії стали казуальні ігри. Ці ігри по своїй суті не складні, але кольорові, які не потребували багато взаємодії з клавіатурою і мишею головоломки з технічної точки зору були простіше ніж тривимірні хардкорні шутери, тому для їх розробки не потрібно сильної модифікації універсальних двигунів. Проте в індустрії з'явилися нові гравці, такі як: Torque Game Builder, HGE та інші.

В той час завдяки World of Warcraft, в ігровій індустрії стали дуже популярні MMORPG – а паралельно більшість жанрів все більш робили ставку на мультиплеєр. Цілий ряд двигунів не змогли зробити користувачам нову функціональність для клієнт-серверних додатків, тому вони стали неактуальними. Інші двигуни адаптовані для мультиплеєру шляхом розробки для них серверних рішень, так і для Unity 3D були розроблені Photon SmartFox. Третій тип універсальних двигунів, з початку являвся клієнт-серверним і не почув ніяких змін. До них відноситься Torque 3D. Також, на ринку з'явилися нові двигуни, які робилися для глобальних многокористувацьких ігор, наприклад, HeroEngine, BigWorld, об'єднуючі масштабування під тисячі гравців серверні рішення і доступний конкретному гравцю клієнт.

На ринці ще з 90-х існували браузерні ігри, а потім другий подих їм дали соціальні мережі, необхідність створювати ігри для браузера не залишилась непоміченою. Розробники універсальних двигунів, наприклад Torque 2D/3D, Unity 3D відреагували на це доволі швидко і випустили плагіни

для браузерів, які дозволили відображати графіку прямо у вікні останніх. Спочатку здобув популярність візуалізатор на основі технології Flash, но по цілій низці причин ця технологія втрачає все більшу долю на ринці. Тому зараз для візуалізації у вебі частіше використовується бібліотека для мови JavaScript – WebGL, яка дозволяє створювати інтерактивну 3D-графіку. Однак через недоліки мови, таких як відсутність многопоточності, бібліотека не може повноцінно задовольнити потрібності розробників ігор. Їй на зміну прийшов консорціум W3C (куди входять Microsoft, Google, Mozilla та інші) розробляється новий низькорівневий бінарний компілюючий формат WebAssembly.

Під кінець 21-го століття дуже швидко розвивалися мобільні технології. Несподівано з’явилися мобільні пристрої по потужності схожі з комп’ютером середньої цінової категорії і здатні запускати потужні ігрові додатки з усіма спецефектами, якими володіли низькорівневі графічні інтерфейси. На що розробники ігрових двигунів створили спеціалізований конвертерів, які створюють нативний для конкретного обладнання код (як наприклад Unity 3D), а в інших – модернізували свої продукти для кросплатформеності (наприклад Torque 2D, Cocos 2DX). Також, з’явилися нові гравці, пропонуючі кросплатформені двигуни для всього парку мобільних пристроїв, які виконуються зі швидкістю нативного коду. Приклади таких: Corona SDK, Marmelade SDK, AGK (App Game Kit).

Виник цілий ряд кростплатформених движків, які дозволяють розробляти гру при мінімальному знанню програмування. Прикладами є Construct 2 і GameMakerPro. Використовуючи готові рішення і візуальні редактори, можно швидко - іноді за декілька годин - можно створювати прості ігри. Ця особливість розповсюдження на мобільному ринку, де є розповсюдженим free2play моделі і коротка ігрова сесія зробили “прості” ігри успішним жанром.

1.3 Новинки ігрової індустрії

Низькорівневі програмні інтерфейси: Open GL, DirectX розвиваються відповідно з відеоадаптерами. Раз в 1-2 роки з’являються нові версії, які пітримують і дають прикладним програмістам (розробникам двигунів) реалізувати всю функціональність заліза. DirectX вже досяг 12-ї версії. З іншого боку на зміну OpenGL прийшов Vulcan - новий кросплатформений графічний api, який розробляє консорціум Khronos Group, куди входять розробники заліза та софту.

VR. Останній на теперішній час тренд ігрової індустрії - віртуальна/доповнена реальність. Більшість сучасних ігрових двигунів вже мають підтримку цієї технології, серед них: Torque 3D, Unity 3D, Unreal Engine 4. Розроблено і багато інших сторонніх розширень, таких як Vuforia Unity Extension. Щоби реалізувати підтримку окулярів віртуальної реальності розробниками двигунів потрібно не лише додати візуалізацію на другий екран (для другого ока) з добрим от першого обличчя змістом (так як, перше і друге око може мати разні сцени), але також і додати підтримку управління з нових приладів вводу, які відмінні для різних гарнітур віртуальної реальності і пока не стандартизовані.

За роки існування існування ігрової індустрії в ній з’явилися 5 найбільш типових ігор з точки зору ігрових двигунів:

1) однокористувацькі;

2) багатокористувацькі;

3) ігри для соціальних мереж;

4) мобільні ігри;

5) ігри для VR/AR/

Окрім того, існують інші платформи – від смарт ТВ до ігрових автоматів. Для розробки кожного типу є певний набір движків, тому що з технічної сторони між усіма типами ігор є певні відмінності. На ринку зараз представлені десятки двигунів на будь-який смак: кросплатформені і

спеціалізовані, потребуючі активної роботи з початковим кодом двигуна і доступні без знань програмування взагалі, з різною продуктивністю, якістю документації і ціною.

1.4 Види ігрових рушіїв

Ігровий двигун – це базове програмне забезпечення комп'ютерної гри. Розділення ігри та ігрового двигуна часто розпливчасте і не завжди студії проводять чітку грань між ними. Але загалом термін "ігровий двигун" застосовується до того програмного забезпечення, яке здатне на повторне використання і розширення і тим самим може бути розглянуте як основа для розробки багатьох інших ігор без суттєвих змін.

Вперше визначення ігрового двигуна з'явилося в середині 90-х років, коли почали з'являтися ігри, схожі на головний шутер того часу – Doom. В той же час у вільному доступі почали з'являтися ігрові двигуни, на основі яких звичайні користувачі могли спробувати написати власні ігри.

З тих пір ігрові двигуни ставилися все більш складними технічно, довгими і насиченими за своїм програмним кодом. І при цьому як і напочатку існування вони містять у собі жорстко фіксовані дані:

-ігрову логіку;

-фізику об'єктів;

-правила відрисування об'єктів;

-геймплей в цілому.

Поверх двигуна прописуються всі інші складаючі гри і їх немало. Тому навіть при використанні одного й того ж двигуна віртуальні світи виходять зовсім інші. При цьому деякі обмеження все ж існують. Наприклад, один і той же двигун не можуть використовуватися для стратегій і екшенів, рпг і тактик.

Зазвичай один двигун заточений для ігор одного або пари суміжних жанрів.

Крім дорогих движків, які використовуються крупними компаніями, існують багато безоплатних аналогів, які можно запросто скачати.

На відміну від ігор, двигуни не змінюють один одного так часто. Так що деякі з них використовуються уже майже десять років. Приклад таких: RAGE, CryEngine, Naughty Dog Game Engine, Avalanche Engine, IW Engine, Anvil Engine, Geo-Mod Engine, The Dead Engine і звісно Unreal Engine.

Один з найпопулярніших рушіїв – 4A Engine представлено на рис. 1.4.

Рисунок 1.4 – Ігровий руший 4A Engine

Один із популярніших двигунів для шутерів і екшенів, написаний українськими розробниками – вихідцями із GSC Game World. Ця платформа використовується тільки для внутрішніх потреб компанії і не доступна для інших розробників ні на якій основі. З технічної точки зору вона представляє із себе покращенний X-Ray з дороблленним PhysX, теселяцією для покращення графіки, а також повного зруйнування об’єктів. Двигун забезпечує тривимірне позиціювання звуку, повністю динамічне освітлення, безліч умов бою, відмінні умови для скриптування. Окрім того, в ньому є система аналізу топології штучного інтелекту, можливість наділити

персонажів зором, слухом та іншими відчуттями, а також завдати їх групову поведінку. Взагалі, на основі цього двигуна можна писати дійсно складні шутери так екшени.

Anvil – це ще один двигун написаний тільки для внутрішнього користування в грі Assassins Creed (рис. 1.5).

Рисунок 1.5 – Ігровий руший Anvil

В основі використовується код C++, з одмалюванням в ZBrush і 3ds Max і створенням фізики в менш відомому Havok. Двигун не самий легкий, а потому потребуючий систем з гарними ресурсами в плані продуктивності, але він забезпечує відмінну картинку з деталізованою анімацією, реалістичними погодними умовами, великою кількістю персонажів в одній сцені (до 3 тисяч), а також активним і достатньо розумним NPC як просто в грі, так і під час боїв.

Creation Engine – один із нових движків, створений студією Bethesda. Його роботу можна побачити в Fallout 76 або Skyrim (рис. 1.6). Перше і одне з основних переваг цієї платформи - підтримка великих локацій з детальною промальовкою всіх об’єктів на ній, а також можливістю вільного швидкого переміщення, без додаткової підгрузки. Окрім того, розробники Creation Engine приділили увагу штучному інтелекту, зробивши його майже довершенним. Ефекти погоди також, такі як вода та сніг виглядають дуже реалістично, а також анімації і заснованованому на фізиці рендерінгу.

Рисунок 1.6 – Ігровий руший Creation Engine

CryEngine 4 (рис. 1.7): німецька студія Crytek продовжує обновляти і робити сучаснішим своє ігрове ядро, при тому як і попередні версії, CryEngine 4 розповсюджується майже безкоштовно – з мінімальною оплатою.

Рисунок 1.7 – Ігровий руший CryEngine 4

При цьому по своїм можливостям він ніскільки не обмеженний. Роботу цього двигуна можно побачити в серіях ігор FarCry.

CryEngine 4 забезпечує відображення великих безшовних локацій, інверсної кінематики, транспорту і персонажів, відмінну імітацію не твердих об’єктів, підлаштовані параметри штучного інтелекту, звучання формата 5.1, а також безліч інших переваг. Взагалі, це відмінна платформа для розробників з досвідом і для починаючих девелоперів.

id Tech – цей двигун є один із найвідоміших, уже 7 версія і на протязі всього існування розповсюджується на безоплатній основі (рис. 1.8).

Рисунок 1.8 – Ігровий руший id Tech

При цьому не варто думати, що цей движок обмежений по функціоналу або простий - на його основі створені такі хіти, як Wolfenstein, Quake, Rage, Doom. В цьому двигуні якісно зробленні отображення текстур, є окремий поток для обробки кожного елементу двигуна і є навіть затемнення участків у кадрі.

Frostbite (рис. 1.9) – це двигун від компанії Electronic Arts. Він використовується в багатьох іграх цієї студії, включаючи Battlefield, FIFA, DragonAge, PayBack.

Рисунок 1.9 – Ігровий руший id Frostbite

Ця платформа універсальна, бо на основі неї написані різні жанри ігор РПГ, гонки, спортивні симулятори і так далі.

В цьому двигуні складно знайти недоліки, а серед переваг можна виділити: безліч пост-ефектів, відмінне руйнування об’єктів, тривимірні і двовимірні спецефекти, реалістичні текстури і навіть окремий ігровий редактор для роботи зі шейдерами і малими деталями.

IW Engine (рис.1.10). Ця платформа використовується для серії шутерів Call of Duty, і це один із небагатьох двигунів в якому використовується система невагомості. Окрім того, це ядро дозволяє використовувати змінювані текстури, безліч спецефектів, димку та інші динамічні погодні умови, створення багатослойних текстів, імітувати контузію персонажу, налаштовувати перегрів стволу в залежності від скорості стрільби або температури навколо персонажу та багато іншого.

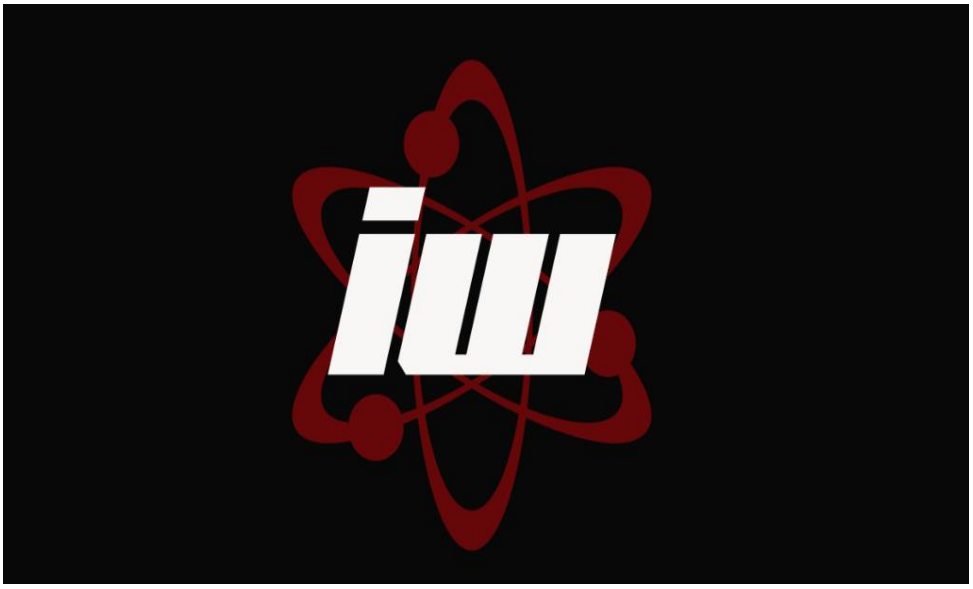

Рисунок 1.10 – Ігровий руший IW Engine

При цьому двигун відмінно оптимізований, так що відрізняється великою продуктивністю і вимагає не дуже багато ресурсів.

Rage Engine (рис. 1.11). Цей двигун був написаний командою розробників Rockstar Games для власного використання – Rage Engine.

Рисунок 1.11 – Ігровий руший Rage Engine

При чому більшість компонентів цього коду написана з нуля робітниками компанії, а взагалі, двигун описує майже всі складаючі гри: в ньому є звукова, графічна, анімаційна і мережева складаючі, штучний інтелект, власна скриптова мова і модулі для роботи онлайн.

Як і в більшості сучасних ігрових ядрах, тут є підтримка великих локацій без дозагрузки, включаючи захід в розташуванні на них будівлі, підтримка динамічної погоди і навіть безліч видів транспорту.

Source – цей двигун не такий відомий як ті що були описані до цього, на відмінну компанії, яка його створила. Це компанія Valve, яка написала такі хіти, як CS, Portal і Half Life.

Рисунок 1.12 – Ігровий руший Source

Важливою складаючою цієї платформи є відмінно відпрацьована ліцева анімація, просунутий штучний інтелект, який дозволяє противникам збиратися в групи, а також якісна робота зі шейдерними ефектами. Крім того, в цьому коді передбачена робота з динамічними джерелами світла, включаючи автоматичне затемнення, відмінне руйнування об’єктів і крута кінематографічна фізика.

Unreal Engine – вне залежності від версій – найбільш популярна платформа для розробників-початківців. Він розповсюджується безкоштовно і використовувати його можно до тих пір пока чистий дохід на іграх не досягне 3000 доларів.

Рисунок 1.12 – Ігровий руший Unreal Engine

При цьому двигун дуже якісний та ідеально підходить для створення шутерів і екшенів, в ньому написано безліч важливих моментів, є доступ в магазин контенту, звідки можна брати необхідні додаткові компоненти і при цьому не вникати в код. Взагалі цей движок з якого можна починати розробку власної гри.

Unity (рис. 1.13). Це міжплатформена середа розробки комп’ютерних ігор, розроблена американською компанією Unity Technologies. Unity дозволяє створювати додатки, працюючі на більш ніж 25 різних платформах, до яких належать: персональні комп’ютери, ігрові консолі, мобільні прилади, інтернет додатки та інші. Unity була випущена в 2005 році і з того часу йде постійний розвиток.

Рисунок 1.13 – Ігровий руший Unity

2 ДОСЛІДЖЕННЯ МЕТОДІВ ТА АЛГОРИТМІВ РОЗРОБКИ ІГРОВИХ ДОДАТКІВ

2.1 Спеціалізація жанрів

Як правило, ігрові двигуни спеціалізовані в рамках жанру комп'ютерних ігор. Так, двигун, спроектований для двовимірного файтінгу на боксерському рингу, буде значно відрізнятися від двигуна для масової багатокористувацької гри, шутера від першого обличчя або стратегій у реальному часі. Але в той же час двигуни мають значні спільні частини – всі тривимірні гри, незважаючи на жанр, потребують взаємодії гравця за допомогою клавіатури, геймпада і/або миші, деяку форму тривимірного рендерінгу, засоби індикації, як на лобовому склі (наприклад, печать тексту поверх графічного зображення), звукову систему і багато іншого. Так, двигун Unreal Engine, недивлячись на те, що був спроектований для шутера от першого обличчя, успішно використовується для створення ігор в безліч інших жанрів, таких як шутер від третього обличчя Gears of War, пригодницька рольова гра Grimm, або футуристична гонка Speed Star.

Ігри від першої особи. Історичні шутери від першого обличчя відносяться до ігор, які найбільш технологічно складні, так як їм необхідно представляти гравцю ілюзію тривимірного світу, та робити це для активних дій у реальному часі. Двигуни шутерів від першого обличчя більш звертають увагу на такі технології, як ефективний рендерінг тривимірних світів чутлива ігрова механіка контролю і прицілювання, висока точність анімації зброї і рук керуючого ігроком персонажу, широкий спектр ручного озброєння, «вибачаюча» модель руху ігрока і його зіштовхнення з перепонами, висока якість анімації та штучного інтелекту не ігрових персонажів. При цьому характерна мала масштабуєма в багатокористувацьких іграх (типічна підтримка до 64 гравців) і повсемісно орієнтація на ігровий процес

deathmatch. Графічні двигуни ігор даного жанру використовують ряд оптимізацій в залежності від поточного оточення ігрока, але разом з тим пред'являється вимпоги по анімації персонажу, аудіо та музиці, динаміки твердого тіла, кінематиці та іншим технологіям.

До жанру шутера від першої особи належать такі ігри, як Quake, Unreal Tournament, Half-Life, Battlefield, Destiny, Titanfall та Overwatch (рис. 1.2). Історично вони передбачали відносно повільне піше пересування у потенційно великому, але переважно коридорному світі. Проте в сучасних FPS дія відбувається в різних віртуальних середовищах, включаючи великі відкриті і закриті приміщення. Сучасна механіка переміщення в FPS може включати пересування пішки, на наземних транспортних засобах, як рейкових, так і ні, суднах на повітряній подушці, човнах і літаках.

Ігри від першої особи, як правило, одні з найбільш технологічно складних у створенні. З ними конкурують тільки шутери від третьої особи, ігри-платформи і розраховані на багато користувачів ігри. Це викликано тим, що FPS прагнуть дати гравцям ілюзію занурення у детальний, гіперреалістичний світ. Не дивно, що багато великих технологічних інновацій ігрової індустрії виникли з ігор цього жанру.

Відмінними рисами двигунів FPS є:

- ефективний рендеринг великих тривимірних віртуальних світів;
- чуйна механіка керування камерою/прицілюванням;
- високоякісна анімація віртуальної зброї гравця;
- широкий асортимент потужної зброї у руках;
- нестрога модель руху персонажа та колізій, яка часто надає цим іграм відчуття плавання;
- високоякісна анімація та штучний інтелект для неігрових персонажів (non-player characters, NPC) – ворогів та союзників гравця;

– невеликі можливості розрахованої на багато користувачів онлайн-ігри (зазвичай з підтримкою від 10 до 100 одночасних гравців), а також особливий ігровий режим "смертельний бій" (death match).

Рисунок 2.1

Технологія рендерингу, яка використовується в шутерах від першої особи, майже завжди гранично оптимізована і ретельно налаштована на конкретний тип середовища, що візуалізується. Наприклад, двигуни ігор у закритих приміщеннях типу «обхід підземель» часто використовують дерева двійкового розбиття простору або портальні системи рендерингу. А у FPS-іграх на відкритій місцевості застосовуються інші види оптимізації, такі як відключення рендерингу об'єктів, невидимих в даний момент, або автономний поділ ігрового світу на сектори з ручним або автоматичним визначенням того, які цільові сектори видно з кожного вихідного сектора.

Звичайно, занурення гравця в гіперреалістичний ігровий світ потребує значно більшого, ніж просто оптимізована високоякісна графічна технологія. Анімація персонажів, аудіо та музика, фізика твердих тіл, ігрова кінематика

та безліч інших технологій у FPS мають бути найпередовішими. Так що вимоги до цього жанру, одні з найсуворіших і найширших в індустрії ігор.

Платформери та інші ігри від третьої особи.

Двигуни платформерів звертають більше уваги на анімацію персонажу і його аватару, і при цьому їм не потрібно тієї реалістичності, яка присуща тривимірним шутерам. Для платформерів характерне застосування ряду технологій: безліч способів переміщення (рухаючі платформи, сходи, мотузки, підпори та інші), елементи із головоломок, використання слідкуючої за персонажем камери від третього обличчя, рендерінг декількох слоїв геометрії в поєднанні з системою зіштовхнення об’єктів, та інші.

«Платформер» — це термін, що застосовується до екшен-ігор від третьої особи, де стрибок із платформи на платформу є основною механікою ігрового процесу. Типові ігри епохи 2D - Space Panic, Donkey Kong, Pitfall! та Super Mario Brothers. Epa 3D породила такі платформери, як Super Mario 64, Crash Bandicoot, Rayman 2, Sonic the Hedgehog, серії Jak та Daxter (рис. 1.3), серії Ratchet & Clank та Super Mario Galaxy.

З точки зору технологічних вимог платформери, як правило, можуть бути об'єднані із шутерами від третьої особи та екшен/пригодницькими іграми від третьої особи, такими як Just Cause 2, Gears of War 4 (рис. 1.4), серіями Uncharted, Resident Evil, The Last of Us, Red Dead Redemption 2 та багатьма іншими.

Ігри від третьої особи мають багато спільного зі шутерами від першої особи, але в них набагато більше уваги приділяється здібностям та пересуванням головного персонажа. Крім того, для аватара гравця потрібна високоякісна анімація всього тіла, на відміну від дещо менш обтяжливих вимог до анімації «плаваючих рук» у типовій грі FPS. Тут важливо відзначити, що майже всі шутери від першої особи мають розрахований на багато користувачів онлайн-компонент, тому на додаток до зброї тут повинен бути представлений аватар гравця на весь зріст. Проте у FPS реалістичність

цих аватарів зазвичай непорівнянна з реалістичністю неігрових персонажів, її також не можна порівнювати з точністю аватара гравця у грі від третьої особи.

У платформер головний герой часто схожий на героя мультфільму і не особливо реалістичний, часто навіть виконаний в невисокій роздільній здатності. А у шутерах від третьої особи людський персонаж часто дуже реалістичний. В обох випадках персонаж гравця зазвичай має дуже багатий набір дій та анімацій.

Деякі з технологій, спеціально призначених для використання в іграх у цьому жанрі, включають:

- рухомі платформи, сходи, канати, грати та інші цікаві способи пересування;
- схожі на головоломки елементи довкілля;
- наступну за персонажем камеру, яка залишається сфокусованою на ньому, її обертання зазвичай контролює гравець за допомогою правої ручки джойстика (на консолі) або миші (на ПК) (зверніть увагу: існує ряд популярних шутерів від третьої особи для ПК, а ось платформери призначені майже виключно для консолей);
- складну систему контролю положення камери, через яку точка огляду ніколи не «потопає» в геометрії фону або динамічних об'єктах на передньому плані.

Файтинги. Файтинги орієнтовані на багату анімацію, точність ударів, можливість завдання складних комбінацій за допомогою кнопок і/або джойстика і тому подобне. Анімаціоні персонажі пред'являють вимоги двигунам за високої деталізації, для двигунів забезпечують можливіть змінення і додавання спецефектів (шрамів після ударів, піт і тому інше), а також предоставляються можливості симуляції зачіски, одягу та інших елементів.

Файтинги – це бійки для двох гравців, у яких персонажі, схожі на людей, б'ють один одного на подібній до рингу. Жанр характеризується такими іграми, як Soul Calibur та Tekken 3. Сторінки "Вікіпедії" en.wikipedia.org/wiki/Fighting_game та https://ua.wikipedia.org/wiki/Файтінг містять огляд цього жанру.

Традиційно ігри у жанрі файтинг орієнтовані:

- на багатий набір бойових анімацій;
- точне виявлення попадання;
- систему введення користувача, здатну виявляти складні комбінації натискання кнопок;
- досить статичні фони, крім анімованих глядачів.

Оскільки тривимірний світ у цих іграх невеликий, а камера постійно зосереджена на дії, історично склалося так, що тут практично не було необхідності у розподілі світу або відключенні рендерингу невидимих об'єктів. Ніхто також не очікує використання в них просунуті тривимірні моделі поширення звуку.

Автосимулятори. Автосимулятори можуть бути різними і тут є низка підджанрів. Графіка таких ігор орієнтована на «коридорність» та кільцеві треки, і тому двигуни більше звертають увагу на деталізацію машин, треку та безпосереднє оточення. Як наслідок, використовуються технології для рендерингу далеких фонових об'єктів (відображаються двовимірно), трек часто поділяється на кілька секторів, усередині яких проводиться оптимізація з рендерингу. У разі руху тунелями або іншими «тісними» місцями використовуються техніки для того, щоб камера з виглядом від третьої особи не перетиналася з фоновою геометрією. Використовувані структури даних і штучний інтелект орієнтуються вирішення завдань машин неігрових персонажів, як-от пошук шляху та інших технічних проблем.

Стратегії. У стратегій реального часу немає високих вимог до графіки і тому двигун орієнтується те що, що відображає юнітів в низькому дозволі, та

заодно він може бути здатний працювати з великою кількістю юнітів одночасно. Окремі особливості є в інтерфейсу взаємодії гравця та елементів управління, до яких входять інструменти роботи з групами юнітів (виділення за площею, управління) та ряд меню та панелей інструментів, що містять команди управління, елементи спорядження, вибір типів юнітів та будівель тощо.

Онлай ігри. Масові розраховані на багато користувачів ігри вимагають наявності великого ігрового світу і можливості одночасної присутності і взаємодії великого числа гравців. Локальні завдання, що вирішуються двигуном, схожі на ті, що є в іграх інших жанрів, але особливістю жанру є орієнтація та опрацювання програмного забезпечення серверів, які повинні зберігати стан світу, керувати підключенням та відключенням гравців, надавати внутрішньоігрові чати, способи взаємодії голосом і так далі.

2.2 Розробники

Розробники проектують та створюють програмне забезпечення, що змушує гру та інструменти працювати. Вони часто поділяються на дві групи: розробники середовища виконання (працюють над двигуном та грою як такий) та розробники інструментів (створюють офлайнові інструменти, що дозволяють іншим розробникам діяти більш ефективно). Члени обох груп мають різні спеціальності. Деякі з них зосереджуються на одній певній системі движка, наприклад, рендерингу, штучному інтелекті, аудіо або колізії та фізиці, деякі – на програмуванні геймплея та скриптингу. А інші вважають за краще працювати на системному рівні і не надто вникають у те, як насправді працює гра. Деякі розробники – універсали, майстри на всі руки, які можуть перейти на роботу над будь-якою проблемою, яка потребує вирішення під час розробки.

Старших розробників іноді просять взяти на себе технічну керівну роль. Провідні розробники зазвичай продовжують проектувати та писати код,

а також допомагають керувати роботою команди, приймають рішення щодо загального технічного спрямування проекту та іноді навіть безпосередньо керують іншими співробітниками.

У деяких компаніях також є один або кілька технічних директорів, робота яких зводиться до ведення одного або декількох проектів на високому рівні, інформування команд про потенційно можливі технічні проблеми, майбутні події в індустрії, нові технології і т. д. Найвища посада, пов'язана з програмуванням в ігровій студії, CTO (якщо вона взагалі є). Роль CTO – бути свого роду технічним директором для всієї студії і виконувати ключову роль у компанії.

2.3 Фахівці творчих професій

У ігровій промисловості говорять: «Контент – це головне». Художники створюють весь візуальний та аудіоконтент гри, і якість їхньої роботи може буквально зробити чи знищити її. Для роботи потрібні художники та дизайнери усіх напрямків.

Творці концепт-артів створюють скетчі та замальовки, що дозволяють команді побачити результат розробки гри. Вони починають раніше за інших – на стадії розробки концепції, але зазвичай забезпечують візуальну спрямованість проекту протягом усього життєвого циклу. Як правило, скріншоти, зроблені в процесі гри, мають дивну схожість із творами концептуального мистецтва.

Розробники 3D-моделей (3D-моделери) створюють тривимірну геометрію для всього у віртуальному світі гри. Сюди зазвичай входять розробники моделей переднього та заднього плану. Ті, хто працює над переднім планом, створюють об'єкти, персонажів, транспорт, зброю – все, що наповнює ігровий світ. Розробники моделей заднього плану створюють геометрію статичного фону – рослинність, будівлі, мости тощо.

Художники за текстурами створюють двовимірні зображення, які називаються текстурами, які накладаються на поверхні 3D-моделей для забезпечення деталізації та реалізму.

Фахівці з освітлення створюють в ігровому світі джерела світла, як статичні, так і динамічні, працюють з кольором, інтенсивністю та напрямом освітлення, щоб забезпечити максимальну художність та емоційну дію кожної сцени.

Аніматори надають рухомим персонажам та об'єктам гри динамічності. Вони буквально виступають акторами у виробництві гри, як це відбувається у кіновиробництві з комп'ютерною графікою. Однак ігровий аніматор повинен мати унікальний набір навичок, щоб виконати анімацію, яка бездоганно узгоджуватиметься з технологічними особливостями ігрового двигуна.

Актори захоплення руху зазвичай використовуються для забезпечення грубого набору даних про рух, які потім обробляють аніматори, перш ніж інтегрувати їх у гру.

Звукорежисери працюють разом із розробниками, щоб створити та змішати звукові ефекти та музику у грі.

Актори озвучення наділяють персонажів голосами у більшості ігор.

У багатьох іграх є один або кілька композиторів, які складають оригінальний звуковий супровід для гри.

Як і розробників, старших художників часто закликають стати тимлідерами. Деякі ігрові команди мають одного (або кількох) арт-директора, який є головним художником, який керує всією грою та забезпечує впорядкованість діяльності всіх членів групи.

Як і розробників, старших художників часто закликають стати тимлідерами. Деякі ігрові команди мають одного (або кількох) арт-директора, який є головним художником, який керує всією грою та забезпечує впорядкованість діяльності всіх членів групи.

Геймдизайнери. Робота геймдизайнера полягає в розробці інтерактивної складової взаємодії користувача з грою, званої геймплеєм. Різні дизайнери працюють на різних рівнях деталізації. Деякі (як правило, старші) геймдизайнери діють на макрорівні, визначаючи сюжет гри, загальну послідовність розділів або рівнів, а також основні цілі та завдання гравця. Інші дизайнери працюють над певними рівнями або географічними областями всередині віртуального світу гри, створюючи шари геометрії статичного фону, визначаючи, коли і звідки з'являться вороги, розміщуючи такі запаси, як зброя та відновлюючі здоров'я аптечки, розробляючи елементи внутрішньоігрових загадок тощо. У той же час інші дизайнери працюють на більш високому технічному рівні, тісно взаємодіючи з розробниками геймплею, та/або пишуть код (як правило, високорівневою мовою скриптингу). Багато геймдизайнерів у минулому були розробниками, які вирішили грати більш активну роль у визначенні того, як гра виглядатиме.

Деякі ігрові команди наймають одного або кількох ігрових письменників. Робота цього фахівця може змінюватись від співпраці зі старшими геймдизайнерами з метою конструювання сюжету всієї гри до написання окремих ліній діалогів.

Деякі старші дизайнери відіграють роль менеджерів. У багатьох ігрових командах є геймдиректор, робота якого полягає у спостереженні за всіма аспектами геймдизайну, допомоги в управлінні термінами та забезпеченні того, щоб діяльність усіх дизайнерів була впорядкована протягом усієї роботи над продуктом. Старші дизайнери іноді виростають у продюсерів.

Продюсери. Роль продюсерів у різних студіях визначається по-різному. В одних ігрових компаніях їхня робота полягає в управлінні термінами та людськими ресурсами. В інших продюсер є старшим геймдизайнером. По-третє, він є сполучною ланкою між командою розробників і підрозділами компанії (фінансовими, правовими, маркетинговими тощо). У деяких невеликих студіях взагалі немає продюсерів. Наприклад, у студії Naughty Dog

буквально кожен, включаючи обох співпрезидентів, відіграє провідну роль у створенні гри, управління командами та ділові обов'язки поділені між старшими фахівцями.

Інший персонал. Групу людей, які безпосередньо створюють гру, як правило, підтримує ключова команда технічного персоналу. До неї входять виконавче керівництво студії, маркетинговий відділ (або команда, що кооперується із зовнішньою маркетинговою групою), адміністративний штат та ІТ-відділ, робота якого полягає в тому, щоб купити, встановити та налаштувати апаратне та програмне забезпечення та надавати технічну підтримку співробітникам.

Видавці та студії. Маркетингом, випуском та розповсюдженням гри зазвичай займається видавець, а не сама студія. Видавець – це, як правило, велика корпорація, така як Electronic Arts, THQ, Vivendi, Sony, Nintendo та ін. Багато ігрових студій не афілійовані з певним видавцем. Вони продають ігри, що виробляються, будь-якому видавцеві, який запропонував більш вигідні умови. Інші студії працюють тільки з певним видавцем, або уклавши з ним довгостроковий контракт або будучи його дочірньою компанією. Наприклад, ігрові студії THQ управляються незалежно від видавця THQ, але вони належать йому та повністю контролюються ним. Видавець Electronic Arts має тісніші стосунки зі студіями і безпосередньо керує ними. Розробники першого ешелону – це ігрові студії, якими володіють безпосередньо виробники консолей (Sony, Nintendo та Microsoft). Наприклад, Naughty Dog – розробник першого ешелону Sony. Ці студії створюють ігри ексклюзивно для ігрових консолей, що випускаються батьківською компанією.

2.4 Відеоігри як м'яка симуляція реального часу

Більшість дво- та тривимірних відеоігор є прикладами того, що розробники назвали б м'якою комп'ютерною симуляцією на основі

інтерактивної взаємодії агентів у реальному часі. Давайте розберемо цю фразу частинами, щоб краще зрозуміти, що це означає.

У більшості відеоігор деяка частина реального або уявного світу моделюється математично, тому ним можна керувати за допомогою комп'ютера. Модель є наближенням та спрощенням реальності (навіть якщо це уявна реальність), тому що зовсім непрактично переносити до неї кожну деталь аж до атомів чи кварків. Отже, математична модель є симуляцією реального чи уявного ігрового світу. Наближення та спрощення – два найпотужніші інструменти розробника ігор. При вмілому використанні навіть дуже спрощена модель іноді майже не відрізняється від реальності.

Симуляція на основі агентів – це така, в якій взаємодіє низка різних сутностей, відомих як агенти. Під цей опис підходить більшість тривимірних комп'ютерних ігор, де агентами є автомобілі, герої, вогняні кулі, точки сили тощо. Усі інтерактивні відеоігри є тимчасовими симуляторами. Це означає, що модель віртуального ігрового світу динамічна – стан ігрового світу згодом змінюється у міру розвитку подій та історії гри. Відеогра повинна реагувати на непередбачувані вхідні дані від гравця (гравців) — інтерактивні тимчасові симуляції. Нарешті, більшість відеоігор представляють свої історії та реагують на дії гравців у реальному часі, перетворюючи їх на інтерактивні симуляції у реальному часі. Винятком може бути категорія покрокових ігор, таких як комп'ютеризовані шахи або покрокові стратегії. Але навіть ці типи ігор зазвичай надають користувачеві деяку форму графічного інтерфейсу користувача в середовищі виконання. Таким чином, для цілей цієї книги ми припускатимемо, що всі відеоігри мають як мінімум деякі обмеження, пов'язані з роботою в реальному часі. В основі будь-якої системи реального часу лежить концепція граничного значення (deadline). Наочним прикладом є вимога, щоб екран оновлювався не менше 24 разів на секунду, щоб створити ілюзію руху. (Більшість ігор оновлюють екран із частотою 30 або 60 кадрів на секунду, тому що це кратно частоті оновлення монітора NTSC.) Звичайно,

у відеоіграх багато інших граничних значень. Симуляція фізики може вимагати поновлення 120 разів на секунду, щоб залишатися стабільною. Система штучного інтелекту персонажа може знадобитися «думати» принаймні раз на секунду, щоб персонаж не здався дурним. Аудіобібліотека може викликатися не рідше одного разу на 1/60 щоб аудіобуфери були заповнені і не було чутного заїкуватість.

"М'яка" система реального часу – це система, в якій недотримання термінів не є катастрофічним. Отже, всі відеоігри — це м'які системи реального часу: якщо вимоги до частоти кадрів не дотримуються, то з гравцем, як правило, нічого не трапляється. Порівняйте це з жорсткою системою реального часу, коли пропущений термін може означати серйозну травму або навіть смерть оператора. Система авіоніки на гелікоптері або система керування стрижнем на атомній електростанції – це приклади жорстких систем реального часу.

Математичні моделі бувають аналітичними чи чисельними. Наприклад, аналітична (замкнута) математична модель твердого тіла, що падає з постійним прискоренням під дією сили тяжіння зазвичай записується наступним чином:

$$y(t) = \frac{1}{2}gt^2 + v_0 t + y_0. \tag{2.1}$$

Аналітична модель може бути обчислена для будь-яких значень її незалежних змінних, таких як час t у рівнянні (2.1), лише з урахуванням початкових умов v0 та y0 та постійної g. Коли такі моделі можна створити, користуватись ними дуже зручно. Проте багато математичних завдань немає рішення в аналітичному вигляді. А у відеоіграх, де введення користувача непередбачуване, не можемо покладатися на аналітичне моделювання всієї гри.

Чисельна модель руху того ж твердого тіла під дією сили тяжіння може бути виражена таким чином:

$$y(t+\Delta t) = F\left(y(t), \dot{y}(t), \ddot{y}(t)\ldots\right). \quad (2.2)$$

Отже, висота твердого тіла в наступний момент часу $(t + \Delta t)$ може бути знайдена як функція висоти та її похідної від часу першого, другого і, можливо, більш високого порядку в поточний момент часу t (2.2). Чисельне моделювання зазвичай виконується шляхом багаторазових обчислень, щоб визначити стан системи на кожному дискретному тимчасовому кроці. Ігри працюють так само. Основний ігровий цикл запускається багаторазово, і під час кожної ітерації різні ігрові системи, такі як штучний інтелект, ігрова логіка, фізична симуляція та ін, отримують можливість розрахувати або оновити свій стан для наступного дискретного тимчасового кроку. Потім результати візуалізуються шляхом відображення графіки, відтворення звуку та, ймовірно, створення інших вихідних даних, таких як віддача на джойстиці.

3 РОЗРОБКА ІГРОВОГО ДОДАТКУ НА БАЗІ UNITY ЯК ГОЛОВНОГО ІНСТРУМЕНТУ СТВОРЕННЯ ІГОР

3.1 Огляд можливостей Unity

Перша версія Unity з'явилася в 2005 році, коли ігровий двигун був анонсований на Worldwide Developers Conference. Напочатку Unity була створена виключно для комп'ютерів Mac, але потім у серпні вийшло оновлення, яке дозволяло працювати під Windows. В наступних версіях поступово додавались нові платформи і розгортання: міжплатформений веб-плеєр в 2006-м, iPhone в 2008-м, Android в 2010-м, і далі на ігрових консолях Xbox і Playstation.

Є можливість створювати додатки для запуску у браузерах за допомогою спеціального підключаємого модулю Unity (Unity Web Player), а також за допомогою реалізації технології WebGL. Раніше була експериментальна підтримка реалізації проектів в рамках модулю Adobe Flash Player, але пізніше команда розробників Unity прийняла складне рішення відмовитися від цього. У грудні 2009 року Gamasutra назвав Unity одним із самих визначних учасників на ринку ігрових компаній. Безкоштовна версія Unity має деякі обмеження, але для неї є можливість розповсюдження гри при умовах, щорічний дохід з гри не перевищуватиме 100 тисяч доларів.

Таблиця 3.1 – Порівняння форматів ліцензій

Тип ліцензії	Дохід компанії в рік	Багатокористу-вацькі функції	Збірка в хмарному сховищі	Темна тема	Звіт про продуктив-ність
Personal	До 100 000$	20 CCUs	Стандартна	Так	Ні

Продовження таблиці 3.1

Тип ліцензії	Дохід компанії в рік	Багатокористу-вацькі функції	Збірка в хмарному сховищі	Темна тема	Звіт про продуктив-ність
Plus	До 200 000$	50 CCUs	Приорітетна	Так	Так
Pro	Необмежений	200 CCUs	Одночасна	Так	Так
Enterprise	Необмежений	Користувацький мультиплеер	Виділення ресурсів	Так	Так

Таблиця 3.2

Тип ліцензії	Преміум підтримка	Доступ до початкового коду	Екран привітання	Ціна
Personal	Ні	Ні	«Made With Unity» і не обов'язкова анімація	0$
Plus	Ні	Ні	«Made With Unity» і/або не обов'язкова анімація	399$/рік 40$/місяць
Pro	Так	Ні	«Made With Unity» і/або не обов'язкова анімація	1800$/рік 150$/місяць
Enterprise	Так	Так	«Made With Unity» і/або не обов'язкова анімація	200$

На Unity написані тисячі ігор, додатків, візуалізації математичних моделей, які захоплюють безліч платформ і жанрів. При цьому Unity використовується як великими компаніями, так і невеличкими студіями.

Редактор Unity має простий Drag&Drop інтерфейс, а також встановленням плагінів KALI, який легко налаштовувати, який складається з різних вікон, завдяки чому можна робити відладку прямо під час гри в редакторі. Двигун використовують для написання скриптів C#. Раніше підтримувалось Boo також і модифікації JavaScript, відома як Unity.

Script. Розрахунки фізики робить фізичний двигун PhysX від NVDIA для 3D і Box2D для 2D. Графічний API – DirectX (на даний момент DX 11, підтримується DX 12).

Проект в Unity поділяється на сцени (рівні) – окремі файли, які містять свої ігрові світи зі своїм набором об’єктів, сценаріїв, налаштувань. Сцени можуть містити у собі як об’єкти, так і пусті ігрові об’єкти – об’єкти, які не мають моделі. Об’єкти, в свою чергу, містять набори компонентів з якими і взаємодіють скрипти. Також у об’єктів є назва (в Unity дозволяється наявність двух і більш об’єктів з однаковими назвами в одній сцені), може бути тег (мітка) і слой, на якому повинен відображатися. Так у будь-якого об’єкта на сцені обов’язково присутній компонент Transform - він зберігає у собі координати місцезнаходження, звороту і розмірів об’єкту по трьом осям.

Також в Unity підтримується фізика твердих тіл і тканин, а також фізуки типу Ragdoll. В редакторі мається система наслідування об’єкті; дочірні об’єкти будуть повторювати усі позиції, звороту і масштабу батьківського об’єкту.

Скрипти в редакторі прикріпляються до об’єктіві у виді окремих компонентів. При імпорті текстур в Unity можно генерувати alpha-канал, mip-рівні, normal-map, light-map, карту відображень, однак безпосередньо на модель текстуру прикріпити не можна - буде створено матеріал, якому будет назначений шейдер, а потім матеріал прикріпиться до моделі. Редактор Unity

має компонент для створення анімації, але також анімацію можна створити попередньо в 3D-редакторі та імпортувати разом з моделлю, а потім розбити на файли.

Unity 3D підтримує систему Level Of Detail, суть якого в тому, щоби на дальній відстані від ігрока високо деталізовані моделі замінялись на менш деталізовані і навпаки, а також систему Occlusion culling, суть якої в тому, щоби у об'єктів, не потрапляючих в поле зору камери не візуалізувалась геометрія і колізія, що знижує нагрузку на центральний процесор і дозволяє оптимізувати проект. При компіляції проекту створюється виконуваний .exe файл гри (для), а в окремій папці – дані гри (включаючи всі ігрові рівні і динамічні бібліотеки).

Двигун підтримує безліч популярних форматів. Моделі, звуки, текстури, матеріли, скрипти можна пакувати в формат .unitypackage і передавати іншим розробникам або викладати у вільний доступ. Цей же формат використовується во внутрішньому магазині Unity Asset Store, в якому розробники можуть безкоштовно і за гроші викладати у вільний доступ різні елементи, які потрібні при створенні гри. Щоби використовувати Unity Asset Store, необхідно мати аккаунт розробника Unity.

UNet (бібліотека для реалізації мультиплеера в іграх на Unity) була видалена, починаючи з версії 2018.4; рішення "із коробки" для мультиплеера відсутнє. Також можна використовувати підходящий користовачу спосіб контролю версій. Наприклад, Tortoise SVN, Git або Source Gear.

В Unity входить Unity Asset Server – інструментарій для сумісної розробки на базі Unity, який є доповненням, що добавляє контроль версій і ряд інших серверних рішень.

3.2 Переваги і недоліки Unity

Головними перевагами двигуна Unity є присутність візуальної середи розробки, міжплатформної підтримки і модульної системи компонентів. До

недоліків відносять складності при взаємодії з багатокомпонентними схемами і складнощі при підключені зовнішніх бібліотек.

Як правило, ігровий двигун надає безліч функціональних можливостей, які дозволяють задіяти їх у різних іграх, до яких входять моделювання фізичних серед, карти нормалій, динамічні тіні і багато іншого. На відміну від багатьох ігрових двигунів, у Unity є дві основних переваги: присутність візуальної середи розробки та міжплатформенної підтримки. Перший фактор включає не тільки інструментарій візуального моделювання, але також інтегровану середе, ланцюг сбірки, що направлено на підвищення продуктивності розробників, зокрема, етапів створення прототипів і тестування. Під міжплатформеною підтримкою представляється не тільки місця розвертання (установка на персональному комп'ютері, на мобільному приладі, на консолі і т.д), але і наявність інструментарія розробки (інтегрована середа може бути використовуватися під Windows і Mac OS).

Третьою перевагою називається модульна система компонентів Unity, за допомогою якої відбувається конструювання ігрових об'єктів, коли останні представляють собою комбіновані пакети функціональних елементів. На відміну від механізмів наслідування, об'єкти Unity створюються за допомогою об'єднання функціональних блоків, а не поміщення до вузлів дерева наслідування. Такий підхід полегшує створення прототипів, що є актуальним при розробці ігор.

У якості недоліків приводяться обмеження візуального редактору при роботі з багатокомпонентними схемами, коли у складних сценах візуальна робота затруднюється. Другим недоліком є відсутність підтримки Unity посилань на зовнішні бібліотеки, роботу з якими програмістам приходиться налаштовувати самостійно, а також заважає продуктивній роботі команді. Ще один недолік пов'язаний з використанням шаблонів екземплярів. З одного боку, ця концепція Unity пропонує гнучкий підхід візуального редагування об'єктів, але з іншого боку, редагування шаблонів стає складнішим. Також

WebGL-версія двигуна, в силу специфікації своєї архітектури (трансляція коду з C# в C++ і надалі в JavaScript), має ряд невирішених проблем з продуктивністю, використанням пам’яті і працездатністю на мобільних пристроях.

3.3 Ігри на Unity

На Unity написані сотні ігор, додатків і симуляцій, Unity використовується великими компаніями (наприклад) так і при створенні інді-ігор. Комп’ютерні ігри на Unity охоплюють безліч платформ і жанрів, характерними прикладими яких є:

- Guns of Icarus Online, Gone Home – шутер від першого обличчя і квест від першого обличчя, які створені незалежними студіями – для персональних комп'ютерів;
- Dead Trigger, Bad Piggies, Tyrant Unleashed – шутер від першого обличчя, головоломка і колекційна карткова гра – для мобільних пристроїв;
- Assault Android Cactus, The Golf Club – аркаднийй шутер і спортивний симулятор – для игрових консолей.

3.4 Етапи створення гри і огляд Unity двигуна

Unity у вільному доступі для скачування і є безкоштовним до тих пір поки дохід з ігор не стане вище певної суми. Unity можна завантажии у декілька версіях перша безкоштовна і її функціоналу досить до розробки простих ігор (цю версію використано у магістерській роботі). Коли скачується файл з офіційного сайту – це лише хаб у якому можна завантажити потрібну версію самого двигуна і під свої запити. Можна скачати стабільну версію, новішу або навіть бета, альфа версії, але це не рекомендовано. Щодо запитів, то на Unity можна розробляти ігри на пк, андроїд, браузерні ігри, IOS та інше.

Принцип гри цього жанру дуже простий. Можу бути дві ітогові кінцівки або програш або перемога. Основна структура це карта, шлях до цілі, ціль, башні і вороги. Мета не дати ворогу дойти до цілі за допомогою юнітів – башень, які розміщуються на карті. Якщо ворог дойде до цілі це програш, якщо не дойде – це перемога. Може бути підвищення складності оборони цілі – складніші рівні після перемоги. Інтерфейс хабу Unity наведено на рисунку 3.1.

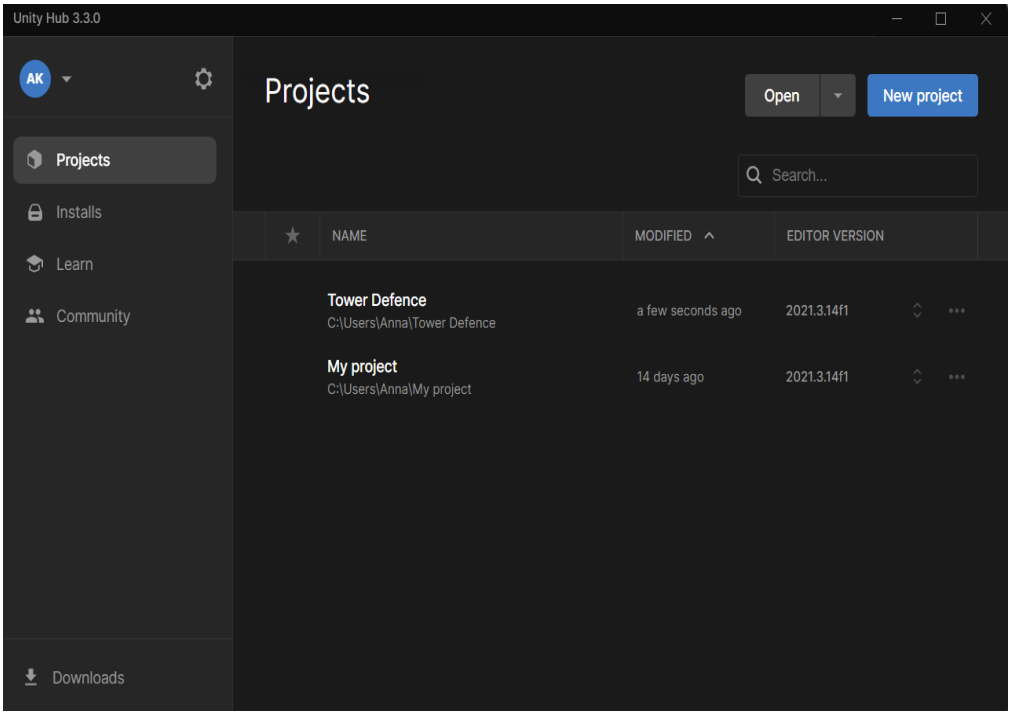

Рисунок 3.1 – Інтерфейс хабу Unity

В хабі – менеджмент створенних проектів, інсталяція пакетів та версій, які потребуються, вкладка з обученням та ком’юніті.

Інтерфейс Unity наведено на рисунку 3.2. Верхня панель інструментів – в ній знаходяться стандартні вкладки File, Edit, Help, як у багатьох інших інтерфейсах, а також вкладки Assets, GameObject, Components та Window.

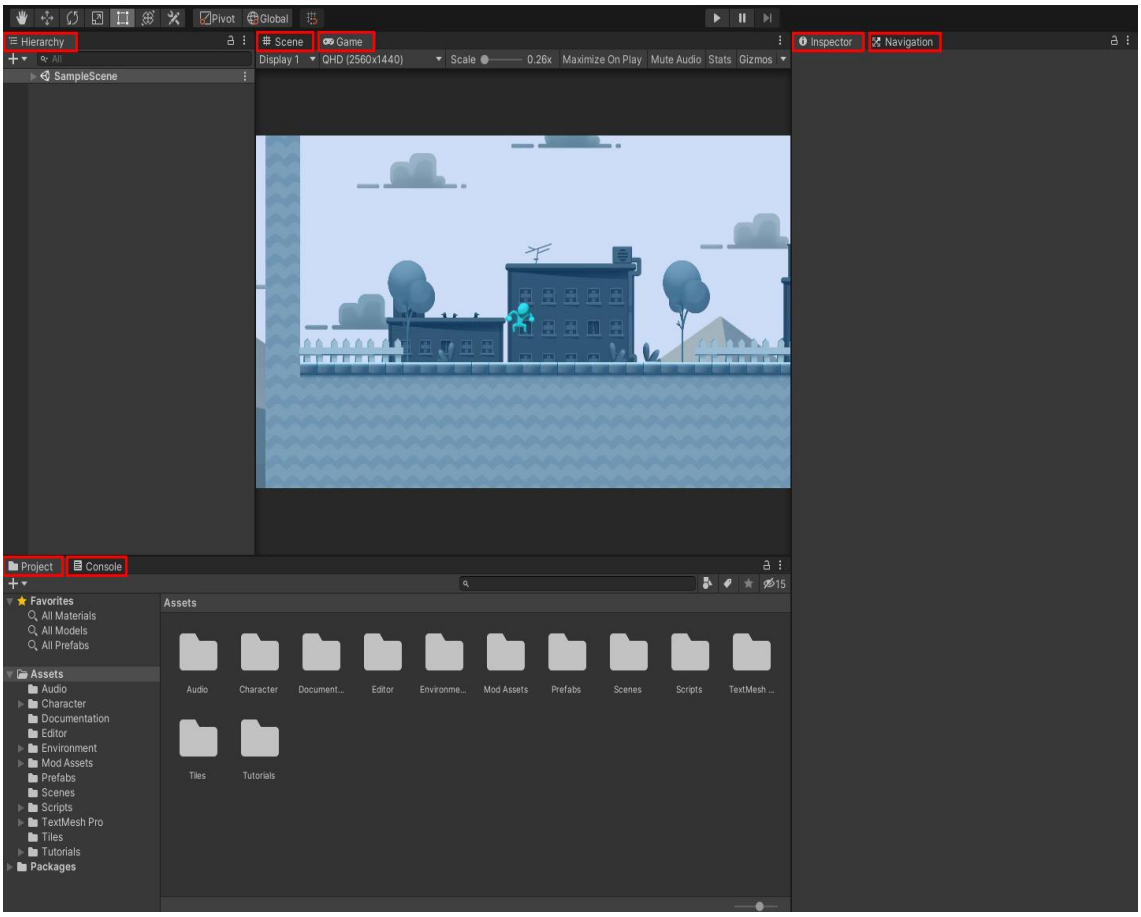

Рисунок 3.2 – Інтерфейс Unity

Scene – вікно сцени, в якому вибудовується ігровий простір (елементи ігрового світу, текстури, фігурки персонажів та інше).

Games - це вікно гри, в якому можна подивитися очима користувача, як рухатимуться елементи та працюватимуть ігрові механіки.

Hierarchy - вікно ієрархії, в ньому перераховано список всіх елементів (GameObject), які розміщені у вікні Scene.

Project – це система папок, у яких зберігаються ассети за категоріями (текстури, шрифти, звуки тощо).

Inspector – вікно для зміни елементів гри, їх розміру, кольору, положення у просторі та інших характеристик.

Основні базові поняття для будь-якої гри.

Геймплей – це загальне поняття взаємодії гравця з ігровим світом, яке визначає його дії (бігти вперед, долати перешкоди, стріляти по мішеням, обганяти інших) та цілі (прийти першим до фінішу, вибити 10 з 10, перемогти ворога в бою, зібрати як можна більше монет). Геймплей безпосередньо пов'язаний з жанром гри, тому що у кожного є специфічний набір правил і механік.

Ігрові механіки – конкретні елементи взаємодії з грою, які входять до складу геймплей. Стрілянина - це одна механіка, бій на мечах – інша, гонка – третя. Одна гра може поєднувати у собі десятки таких механік.

Сюжет - це розвиток дії у грі; він однаково важливий і для масштабних ААА проектів, і для невеликих, але глибоких інді-ігор. Сюжет має затягнути гравця, розповісти йому історію, а також розвивати персонажів, щоб вони не залишалися однобокими та розкривалися для гравця з нових сторін.

Персонажі – у них важливі і дизайн, і характер. Вдало опрацьований персонаж має відомі особливості поведінки, цікавою історією, а ще для повного занурення він повинен мати щось спільне з гравцем, що зачепить його і змусить співпереживати. На цю тему Unity розробили гайд "П'ять типів привабливих ігрових персонажів", щоб у новачків виходило зробити ігрового персонажа правдоподібним.

Дизайн рівнів – це зовнішній вигляд гри, колірні рішення, загальна стилістика об'єктів, фону, персонажів, предметів, що створює певний настрій.

Баланс - це співвідношення характеристик різних об'єктів, він також відповідає за захопленість гравця. Наприклад, якщо меч у грі може наносити об'єкту 3 одиниці шкоди, а об'єкт має всього 3 HP (hit points — величина, що позначає максимальну шкоду), його можна знищити з першого разу, і грати буде занадто легко. Якщо об'єкт має 30 HP, гравцеві доведеться завдати 10 ударів, щоб його знищити. Таке вже підходить швидше для боса, наприклад, на першому чи другому рівні. Розробнику важливо грамотно розподілити ці величини, щоб гра була захоплюючою та кидала гравцю виклики.

Ассет (Asset) – готовий компонент, який можна використовувати для створення своїх проектів. Це може бути елемент інтерфейсу у грі, текстура, фігурка персонажа, шрифт чи звук. Придбати ассети або завантажити безкоштовно деякі з них можна на Unity Asset Store.

Ігровий об'єкт (GameObject) – це будь-який асет, який використовується в ігровій сцені. Наприклад, зображення монетки, її зовнішній вигляд — це ассет, а п'ять монет, які має підібрати персонаж у процесі проходження рівня — це п'ять ігрових об'єктів. Сам персонаж також стане ігровим об'єктом.

Компоненти (Components) - частина ігрового об'єкта, що відповідає за його поведінку в процесі гри: переміщення або реакцію певні тригери.

Скрипт (Script) — код C#, в якому прописані конкретні умови роботи компонента.

Робота зі скриптами.

За поведінку ігрових об'єктів відповідають приєднані до них компоненти (Components). Базовий компонент будь-якого об'єкта Transform, він відповідає за положення елемента у вікні Scene, можливість повертати і масштабувати його. До базового компонента можна додати, наприклад, Renderer, який змінює колір, або RigidBody, який відповідає за масу та фізику об'єкта. Але крім базових компонентів, об'єктам можна задавати особливі умови, і для цього використовуються скрипти.

Створити новий скрипт можна у вікні Project, клацнувши мишкою на Assets -> Create -> C# Script.

Подвійний клік миші скрипт відкривається в текстовому редакторі. Скрипти, як і все інше в Unity, прописуються на C#, так що для створення складних проектів розробникам все ж таки доведеться освоїти цю мову програмування.

Базові елементи скриптів - це:

1) using - елемент у коді, який підключає бібліотеки;

2) public class — у цьому рядку зазвичай прописаний клас MonoBehaviour, він містить набір функцій, необхідних роботи скрипта;

3) void — ті функції, з допомогою прописуються дії, які у грі.

Розглянемо, наприклад, функцію start. Будь-яка дія в ній відбудеться лише один раз, коли запуститься гра. Пропишемо тут print (“Hi”).

І можна побачити, що у консолі це слово виводиться один раз.

Функція update — повторювана, її можна використовувати, наприклад, пересування об'єкта. Для цього в скрипті визначається змінна int i = 0, вона виводиться на екран за допомогою функції print (i) і збільшується на одну одиницю за кожен крок за допомогою i++.

У консолі можна буде помітити, що апдейт справді спрацьовує кожен кадр і об'єкт, до якого застосовано цей скрипт, плавно рухається (рис. 3.3).

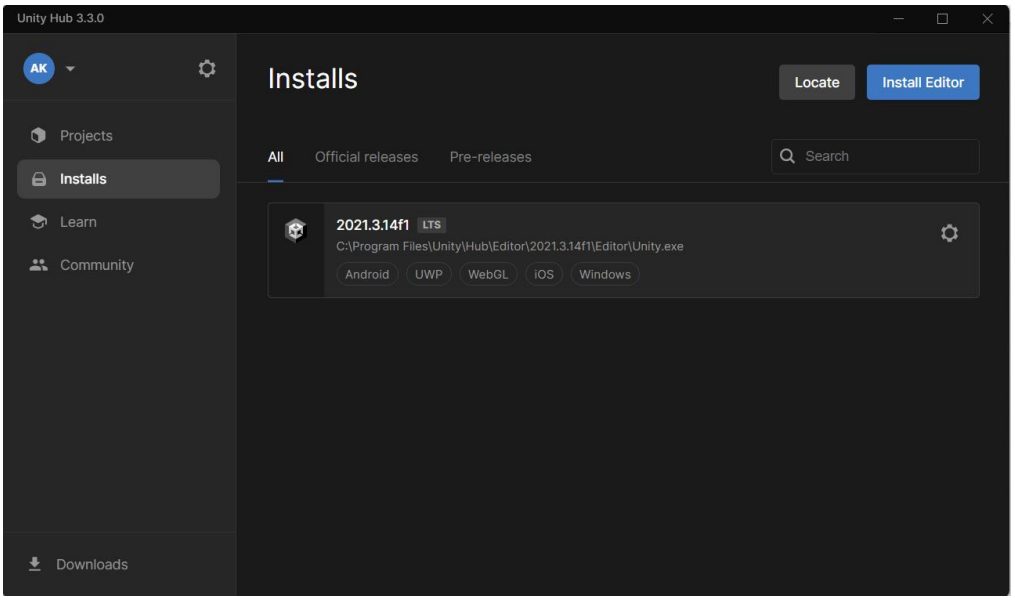

Рисунок 3.3 – Заінстастальвання версії Unity

Тестування.

При тестуванні гри виявився баг на спавні башні на 3 рівні. Друга башня синьої ячейки, вогонь синій, башня біла, але вона не б’є ворогів. З усіма останніми рівнями багів помічено не було.

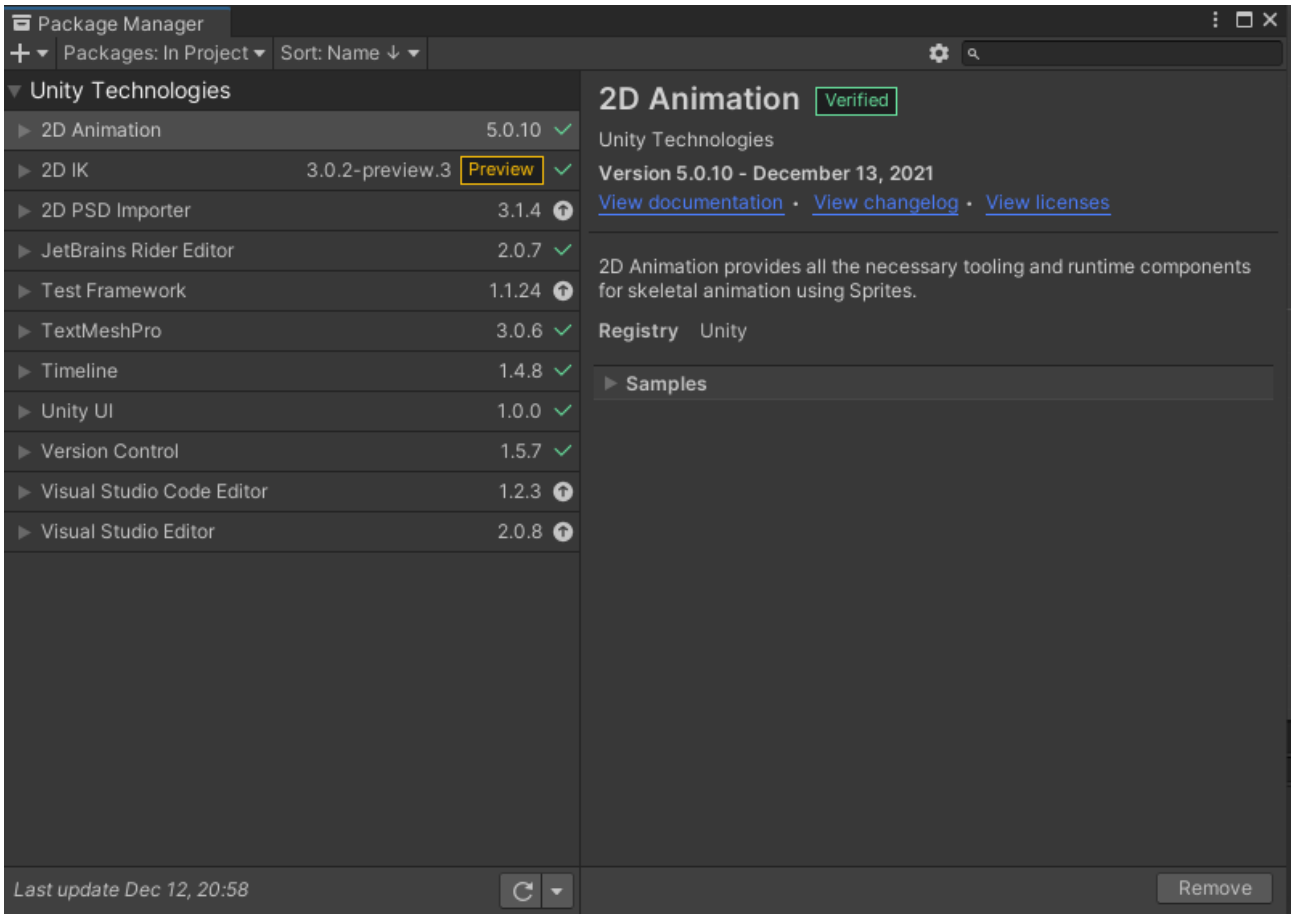

Рисунок 3.5 – Використані пакети

Для гри застосовані такі пакети як 2D Animation, 2D IK, 2PSDD Importer.

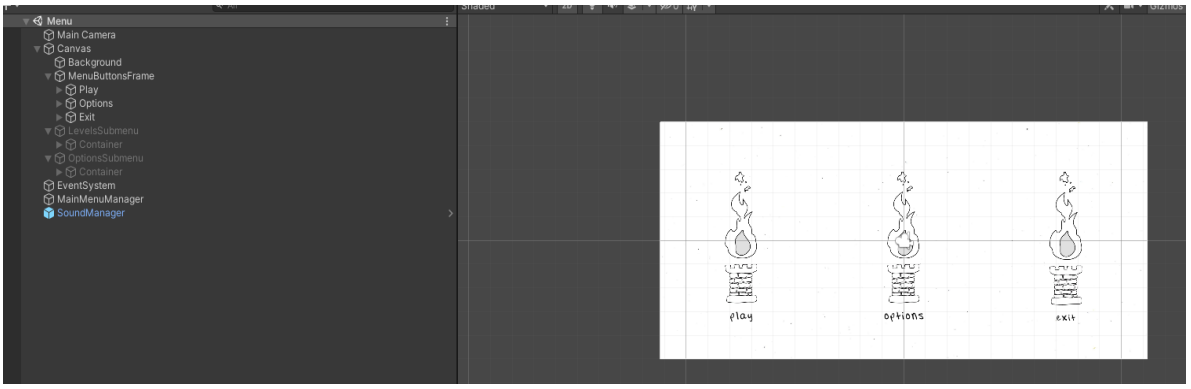

Рисунок 3.6 – Ієрархія сцени Menu

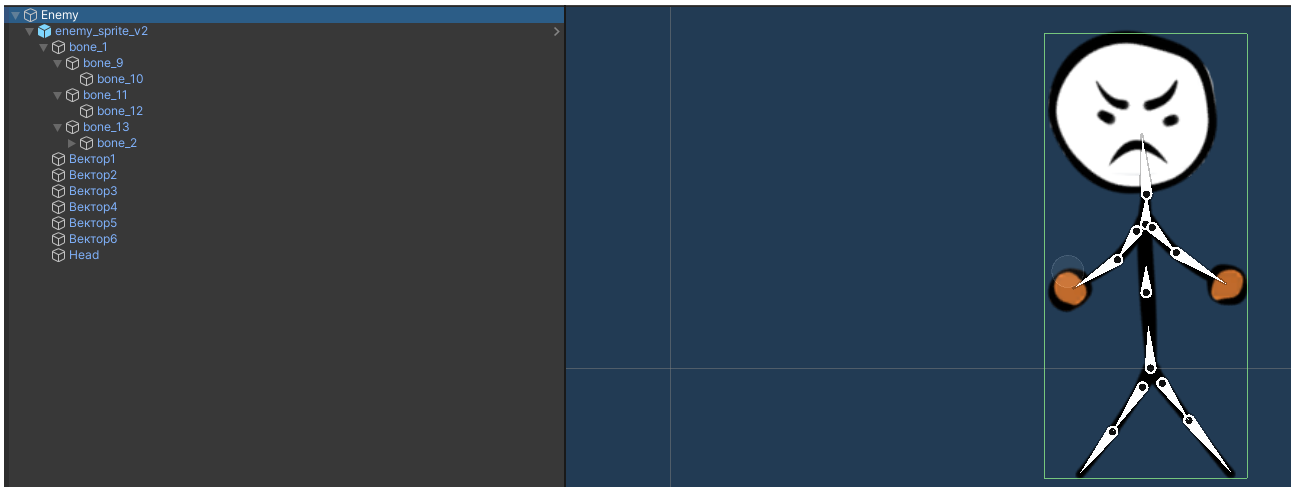

Рисунок 3.7 – Ієрархія префабу Enemy із розставленими кістками

В ігрі головним ворогом є злі чоловічки. Їх є п’ять види: білі, сині, жовті, красні і зелені. Кожен більш чутливий до свого кольору башні і отримують більший урон. На мапі є спеціальні ячейки для установки башень і на ячейках є колір, які башні встановлюються.

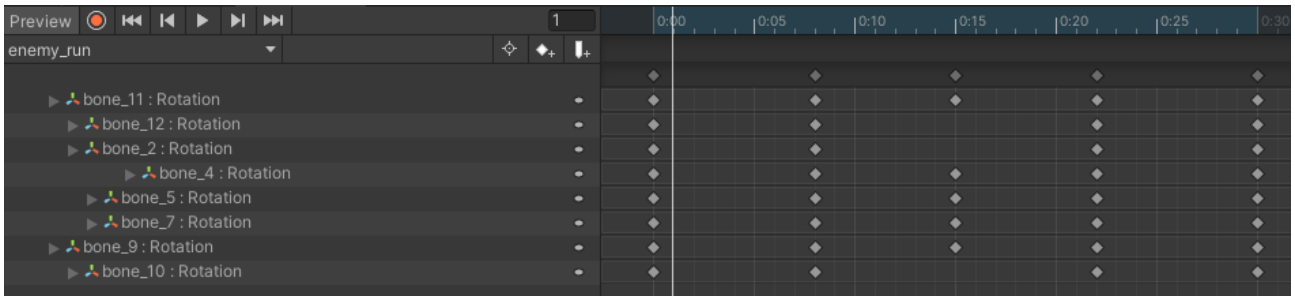

Рисунок 3.8 – Анімація бігу ворога

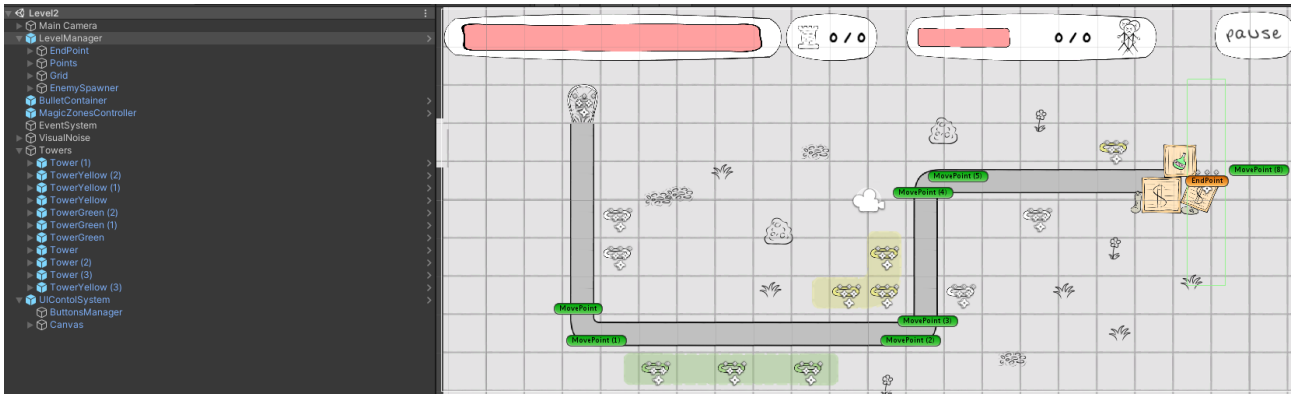

Рисунок 3.9 – Ієрархія одного з рівнів

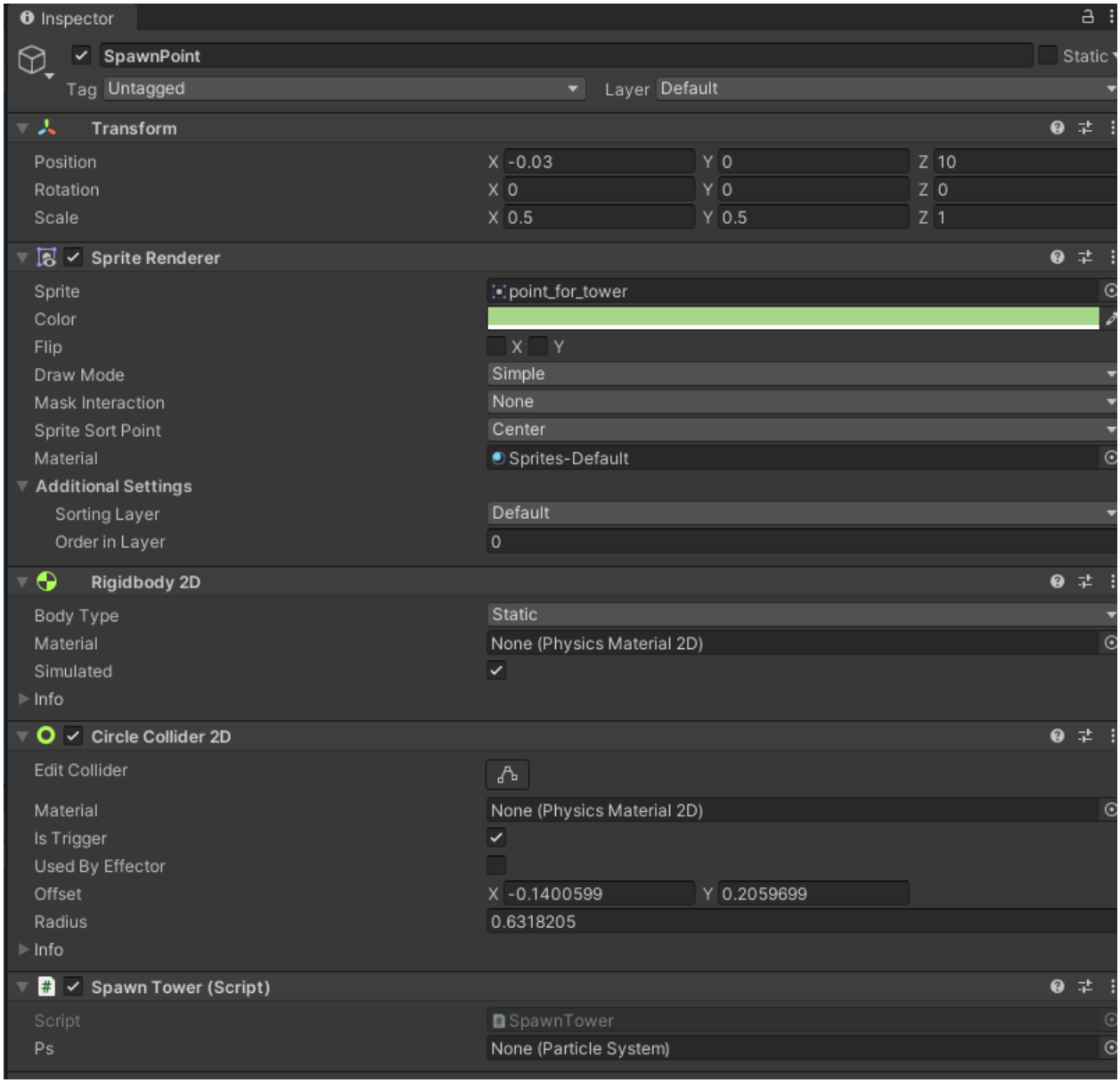

Рисунок 3.10 – Компоненти, які прикріплені до точки спавну башні

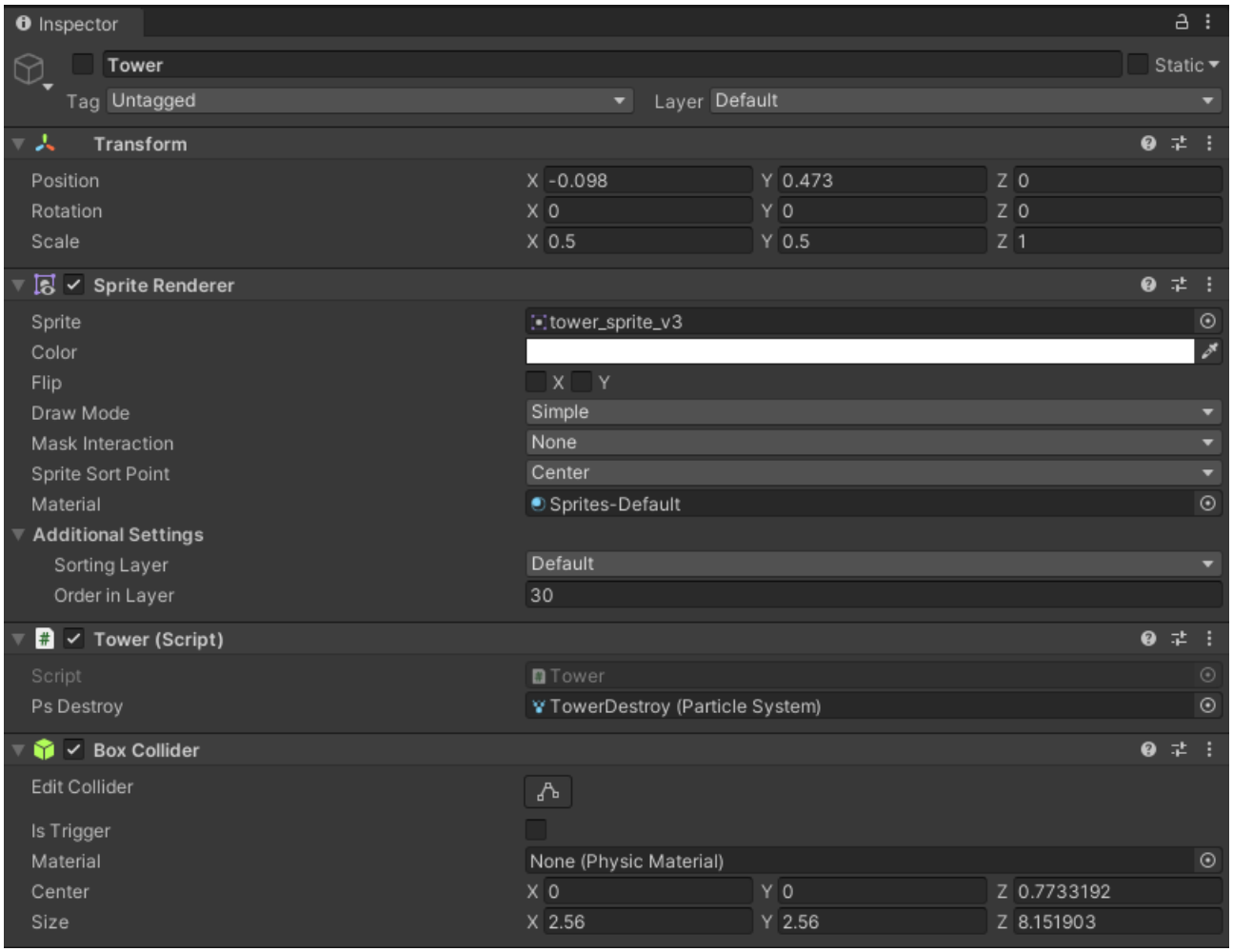

Рисунок 3.11 – Компоненти, прикріплені до об’єкта башні

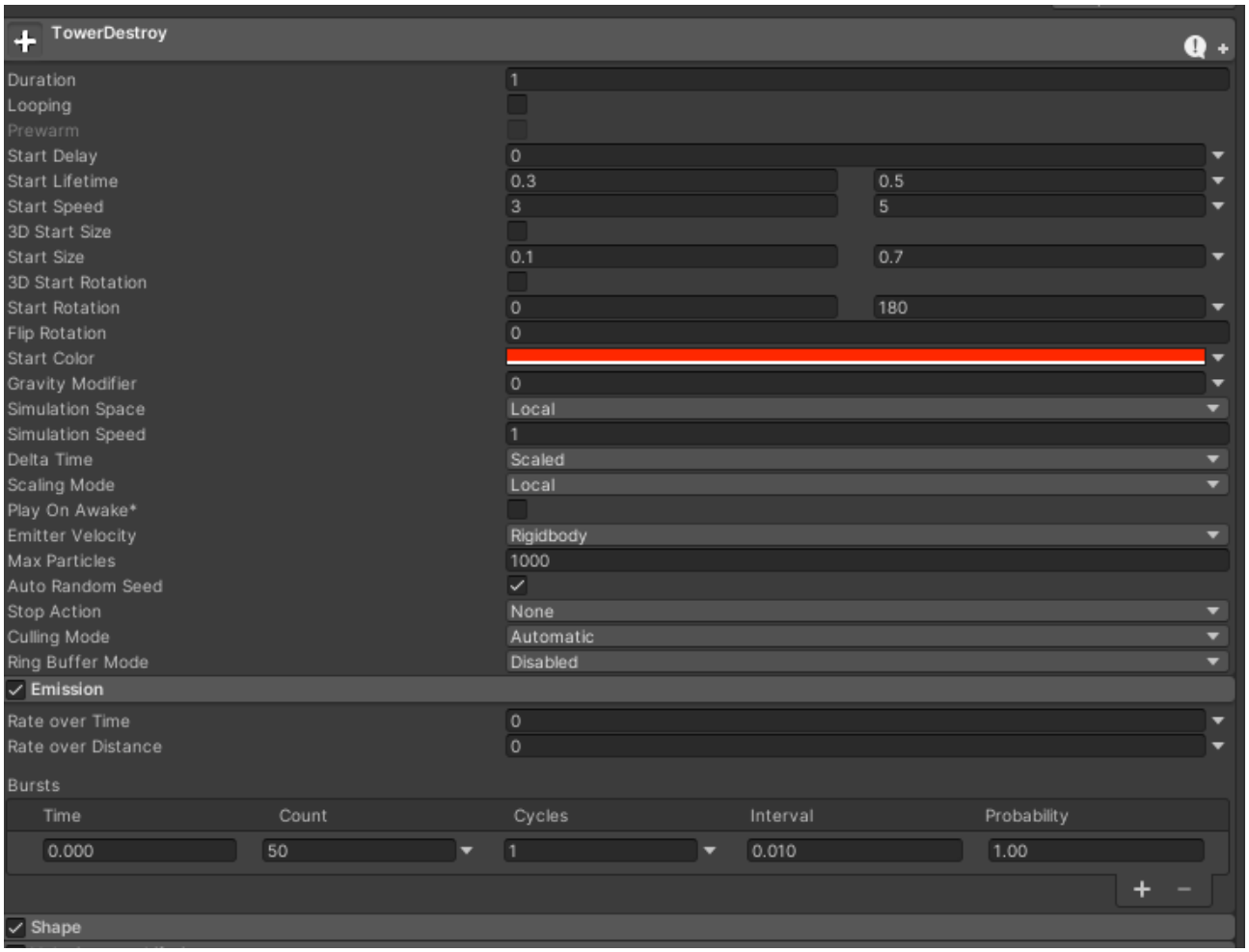

Рисунок 3.12 – Основні налаштування ParticleSystem частинок знищення башні

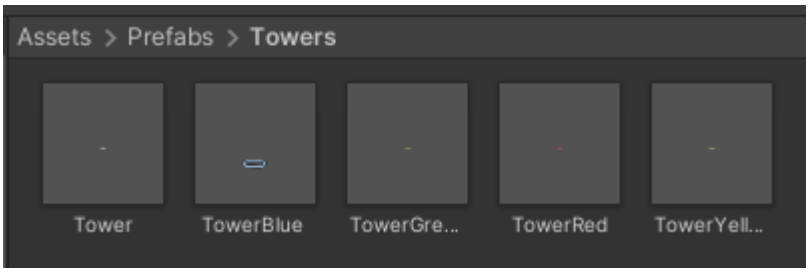

Рисунок 3.13 – Префаби заготовлених башень

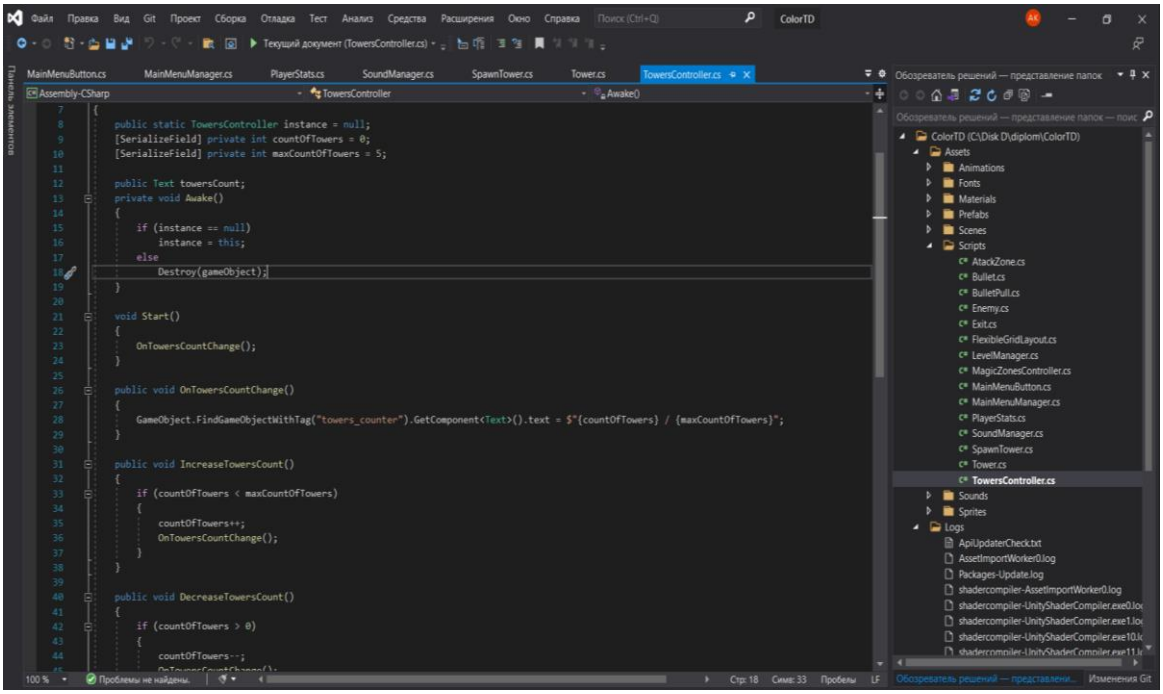

Рисунок 3.14 – Налаштування імпорту графічних ресурсів для ворогів (імпорт відбувається з файлу Photoshop)

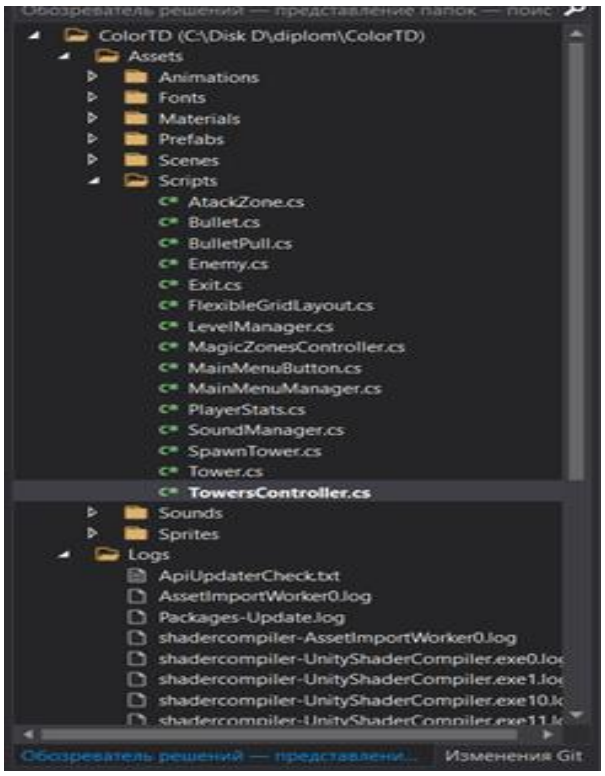

Рисунок 3.15 – Скрипти, які використовуються у грі

1. Видання «Курс Unreal Engine 4 – создание игр».
2. Видання «Разработка игр на Unreal Engine 4 за 24 часа».
3. Список літератури та покрокових гайдів з сайту http://crydev.ru/.
4. Видання «Геймдизайн. Как создать игру, в которую будут играть все».
5. Видання «Курс Unity для начинающих».
6. Курси для початківців з розробки комп’ютерної гри на сайті https://vse- kursy.com/.
7. Онлайн курс навчання в онлайн університеті Skillbox, а саме за наступним посиланням з розробки світу комп’ютерної гри: https://skillbox.ru/media/gamedev/kak_sozdat_zhivoy_igrovoy_mir/.
8. Видання «Курс создания 2D-игр в Unity»
9. Видання «Книга комп’ютерна программа PYTHON для дітей Веселий вступ до програмування»
10. Нвчальне видання з поглибленим вивченням програмування «Язык программирования C++. Лекции и упражнения»
11. Видання «Самоучитель 3ds Max 2014» для початкових знань для розробки 3D моделі
12. Більш детальний посібник з розробки візуальної ігрової моделі за посиланням «https://3d-expo.ru/article/programmy-dlya-sozdaniya-3d-modeley»
13. Більш детальний посібник з розробки візуальної ігрової моделі за посиланням «https://pixlpark.ru/faq/templates/3d-models»

Детальна розповідь про сучасну ігрову індустрію, підводні камні які можуть чекати на етапі розробки та величезна кількість«інсайдерської» інформації від працівників великих студій у виданні «Время игр!: от Parkan до World of Tanks»

14. Ідеї та плани, щоб розробити гру, яку буде грати величезна кількість людей у виданні «Геймдизайн. Как создать игру, в которую будут играть все»

15. Один із мов програмування, яка може допомогти у написанні своєї стрічки коду для величезної кількості завдать в середині гри у виданні «Геймдизайн. Как создать игру, в которую будут играть все»

Додаток А

Зауваження нормоконтролера

Таблиця А.1 – Зауваження нормоконтролера

Сторінка	Позначення	Зміст зауваження
ПЗ		Неплотне заповнення сторінок
ПЗ		Кожен розділ повинен починатися із службового слова РОЗДІЛ та закінчуватися висновками за розділом, нема висновків за роботою
С.61-62		Формули не в редакторі формул - рисунком.
С.61-62		Перелік використаної літератури не за вимогами ДСТУ.

Додаток Б

Лістинг програми

AtackZone.cs

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

public class AtackZone : MonoBehaviour
{
    List<GameObject> enemies;
    IEnumerator shootController;

    public float cooldown = 1;

    public float koefInZone = 2;
    public float koefColorToSameColor = 1;
    public float koefBlackToColor = 0.8f;
    public float koefColorToOtherColor = 0f;

    public float damage = 35;

    public bool isInZone = false;

    public string curColor;

    void Start()
    {
        enemies = new List<GameObject>();
        shootController = null;
    }

    private void OnTriggerEnter(Collider collision)
    {
        if (collision.CompareTag("Enemy"))
        {
            enemies.Add(collision.gameObject);
            if (shootController == null)
            {
                shootController = Shooting();
                StartCoroutine(shootController);
            }
        }
    }

    private void OnTriggerExit(Collider collision)
    {
        if (collision.CompareTag("Enemy"))
        {
            if (enemies.Count > 0)
                enemies.Remove(collision.gameObject);

            if (enemies.Count == 0)
            {
                StopCoroutine(shootController);
                shootController = null;
            }
        }
    }
```

```
    IEnumerator Shooting()
    {
        while (enemies.Count != 0)
        {
            yield return new WaitForSeconds(cooldown);
            Shoot();
        }
    }

    void Shoot()
    {
        float minDistance = 100;
        Vector2 minDistancePosition = Vector2.zero;
        Transform target = null;

        foreach (var enemy in enemies)
            if (enemy != null && minDistance > Vector3.Distance(enemy.transform.position,
transform.position))
            {
                minDistance = Vector3.Distance(enemy.transform.position,
transform.position);
                minDistancePosition = enemy.transform.position;
                target = enemy.transform;
            }

        if (minDistancePosition != Vector2.zero)
        {
            minDistancePosition = (minDistancePosition -
(Vector2)transform.position).normalized;

            GameObject bulletGO =
GameObject.FindGameObjectWithTag("BulletContainer").GetComponent<BulletPull>().getBullet(
);
            bulletGO.transform.position = transform.position;

            Bullet bul = bulletGO.GetComponent<Bullet>();

            if (curColor == "white")
                bul.setDamage(damage * koefBlackToColor);
            else
            {
                if (curColor == target.GetComponent<Enemy>().getColor())
                    bul.setDamage(damage * koefColorToSameColor);
                else
                    bul.setDamage(damage * koefColorToOtherColor);
            }

            if (isInZone)
                bul.setDamage(bul.getDamage() * koefInZone);

            bul.setTarget(target);
            bulletGO.GetComponent<SpriteRenderer>().color =
transform.parent.GetComponent<SpriteRenderer>().color;
            bulletGO.SetActive(true);
            SoundManager.instance?.ShootPlay();
        }
    }

    public void ResetState()
    {
        enemies.Clear();
        shootController = null;
    }
}
```

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

public class AtackZone : MonoBehaviour
{
    List<GameObject> enemies;
    IEnumerator shootController;

    public float cooldown = 1;

    public float koefInZone = 2;
    public float koefColorToSameColor = 1;
    public float koefBlackToColor = 0.8f;
    public float koefColorToOtherColor = 0f;

    public float damage = 35;

    public bool isInZone = false;

    public string curColor;

    void Start()
    {
        enemies = new List<GameObject>();
        shootController = null;
    }

    private void OnTriggerEnter(Collider collision)
    {
        if (collision.CompareTag("Enemy"))
        {
            enemies.Add(collision.gameObject);
            if (shootController == null)
            {
                shootController = Shooting();
                StartCoroutine(shootController);
            }
        }
    }

    private void OnTriggerExit(Collider collision)
    {
        if (collision.CompareTag("Enemy"))
        {
            if (enemies.Count > 0)
                enemies.Remove(collision.gameObject);

            if (enemies.Count == 0)
            {
                StopCoroutine(shootController);
                shootController = null;
            }
        }
    }

    IEnumerator Shooting()
    {
        while (enemies.Count != 0)
        {
            yield return new WaitForSeconds(cooldown);
            Shoot();
```

```
        }
    }

    void Shoot()
    {
        float minDistance = 100;
        Vector2 minDistancePosition = Vector2.zero;
        Transform target = null;

        foreach (var enemy in enemies)
            if (enemy != null && minDistance > Vector3.Distance(enemy.transform.position,
transform.position))
            {
                minDistance = Vector3.Distance(enemy.transform.position,
transform.position);
                minDistancePosition = enemy.transform.position;
                target = enemy.transform;
            }

        if (minDistancePosition != Vector2.zero)
        {
            minDistancePosition = (minDistancePosition -
(Vector2)transform.position).normalized;

            GameObject bulletGO =
GameObject.FindGameObjectWithTag("BulletContainer").GetComponent<BulletPull>().getBullet(
);
            bulletGO.transform.position = transform.position;

            Bullet bul = bulletGO.GetComponent<Bullet>();

            if (curColor == "white")
                bul.setDamage(damage * koefBlackToColor);
            else
            {
                if (curColor == target.GetComponent<Enemy>().getColor())
                    bul.setDamage(damage * koefColorToSameColor);
                else
                    bul.setDamage(damage * koefColorToOtherColor);
            }

            if (isInZone)
                bul.setDamage(bul.getDamage() * koefInZone);

            bul.setTarget(target);
            bulletGO.GetComponent<SpriteRenderer>().color =
transform.parent.GetComponent<SpriteRenderer>().color;
            bulletGO.SetActive(true);
            SoundManager.instance?.ShootPlay();
        }
    }

    public void ResetState()
    {
        enemies.Clear();
        shootController = null;
    }
}
```

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

public class BulletPull : MonoBehaviour
{
    public GameObject bullet;
    public int startBulletCount;

    List<GameObject> bullets;

    void Start()
    {
        bullets = new List<GameObject>();
        for (int i = 0; i < startBulletCount; ++i)
            bullets.Add(Instantiate(bullet, transform));
    }

    public GameObject getBullet()
    {
        if (bullets.Count == 0)
            bullets.Add(Instantiate(bullet, transform));

        GameObject bulletGO = bullets[0];
        bullets.RemoveAt(0);
        return bulletGO;
    }

    public void addBullet(GameObject bulletGO)
    {
        bullets.Add(bulletGO);
    }
}
```

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

public class BulletPull : MonoBehaviour
{
    public GameObject bullet;
    public int startBulletCount;

    List<GameObject> bullets;

    void Start()
    {
        bullets = new List<GameObject>();
        for (int i = 0; i < startBulletCount; ++i)
            bullets.Add(Instantiate(bullet, transform));
    }

    public GameObject getBullet()
    {
        if (bullets.Count == 0)
            bullets.Add(Instantiate(bullet, transform));

        GameObject bulletGO = bullets[0];
        bullets.RemoveAt(0);
        return bulletGO;
    }

    public void addBullet(GameObject bulletGO)
    {
        bullets.Add(bulletGO);
    }
}
```

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

public class Exit : MonoBehaviour
{
    public PlayerStats _playerStats;
    private MainMenuManager mainMenuManager;
    private ParticleSystem ps;
    private LevelManager levelManager;

    private void Start()
    {
        ps = GetComponentInChildren<ParticleSystem>();
        mainMenuManager =
GameObject.FindGameObjectWithTag("buttons_manager").GetComponent<MainMenuManager>();
        levelManager =
GameObject.FindGameObjectWithTag("GameController").GetComponent<LevelManager>();
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Enemy"))
        {
            SoundManager.instance?.BaseHurtPlay();
            Enemy enemy = other.GetComponent<Enemy>();
            _playerStats.TakeDamage(enemy.GetDamage());
            Destroy(other.gameObject);
            levelManager.DecreaseEnemies();
            Win();
            ps.Play();
        }
    }

    public void Win()
    {
        if (_playerStats.GetHP() > 0 && levelManager.GetEnemyCount() == 0)
            mainMenuManager.Win();
    }
}
```

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

public class FlexibleGridLayout : LayoutGroup
{
    public enum FitType
    {
        Uniform,
        Width,
        Height,
        FixedRows,
        FixedColumns
    }

    public FitType fitType;

    public int rows;
    public int columns;
    public Vector2 cellSize;
    public Vector2 spacing;

    public bool fitX;
    public bool fitY;

    public override void CalculateLayoutInputVertical()
    {
        base.CalculateLayoutInputHorizontal();

        if(fitType == FitType.Width || fitType == FitType.Height || fitType ==
FitType.Uniform)
        {
            fitX = true;
            fitY = true;
            float sqrRt = Mathf.Sqrt(transform.childCount);
            rows = Mathf.CeilToInt(sqrRt);
            columns = Mathf.CeilToInt(sqrRt);
        }

        if(fitType == FitType.Width || fitType == FitType.FixedColumns || fitType ==
FitType.Uniform)
        {
            rows = Mathf.CeilToInt(transform.childCount / (float)columns);
        }
        if (fitType == FitType.Height || fitType == FitType.FixedRows || fitType ==
FitType.Uniform)
        {
            columns = Mathf.CeilToInt(transform.childCount / (float)rows);
        }

        float parentWidth = rectTransform.rect.width;
        float parentHeight = rectTransform.rect.height;

        float cellWidth = (parentWidth / (float)columns) - ((spacing.x / (float)columns)
* (columns - 1)) - (padding.left / (float)columns) - (padding.right / (float)columns);
        float cellHeight = (parentHeight / (float)rows) - ((spacing.y / (float)rows) *
(rows - 1)) - (padding.top / (float)rows) - (padding.bottom / (float)rows);

        cellSize.x = fitX ? cellWidth : cellSize.x;
        cellSize.y = fitY ? cellHeight : cellSize.y;
```

```
        int columnCount = 0;
        int rowCount = 0;

        for(int i = 0; i < rectChildren.Count; i ++)
        {
            rowCount = i / columns;
            columnCount = i % columns;

            var item = rectChildren[i];

            var xPos = (cellSize.x * columnCount) + (spacing.x * columnCount) +
padding.left;
            var yPos = (cellSize.y * rowCount) + (spacing.y * rowCount) + padding.top;

            SetChildAlongAxis(item, 0, xPos, cellSize.x);
            SetChildAlongAxis(item, 1, yPos, cellSize.y);
        }

    }

    public override void SetLayoutHorizontal()
    {

    }

    public override void SetLayoutVertical()
    {

    }
}
```

```
using UnityEngine.UI;

[System.Serializable]
public class EnemyClass
{
    public int count;
    public GameObject prefab;
    public Color color;
    public float enemySpeed = 5f;
    public float enemyHP = 100f;
    public int damage = 1;

    public float delayBetweenSpawn = 0.2f;
    public float delayToNextEnemyGroup = 1f;
}

[System.Serializable]
public class Wave
{
    public List<EnemyClass> enemies = new List<EnemyClass>();
    public float timeToNext = 60f;
}
public class LevelManager : MonoBehaviour
{
    public List<Wave> waves = new List<Wave>();
    private int currentWave = 0;
    private float _timer;
    public Transform startPoint;
    public List<Transform> levelPath;

    private int enemyCount = 0;
    private ParticleSystem portalParticles;
    private Image waveImage;
    private Text waveCountText;
    private float currentWaveTick = 0f;
    private Exit exit;

    void Start()
    {
        GameObject waveContainer = GameObject.FindGameObjectWithTag("WaveContainer");
        waveImage =
GameObject.FindGameObjectWithTag("WaveIndicator").GetComponent<Image>();
        waveCountText = waveContainer.GetComponentInChildren<Text>();
        exit = GameObject.FindGameObjectWithTag("exit").GetComponent<Exit>();

        if (!SoundManager.instance.GameMusicPlaying())
            SoundManager.instance.GameMusicPlay();

        portalParticles = startPoint.GetComponentInChildren<ParticleSystem>();
        _timer = waves[currentWave]?.timeToNext ?? 60f;
        for (int i = 0; i < waves.Count; ++i)
        {
            for (int j = 0; j < waves[i].enemies.Count; ++j)
                enemyCount += waves[i].enemies[j].count;
        }

        StartCoroutine(MakeWave());
    }

    void Update()
    {
        if (_timer > 0f)
```

```
        {
            _timer -= Time.deltaTime;
        }
        else
        {
            if (waves.Count != currentWave)
            {
                _timer = waves[currentWave]?.timeToNext ?? 60f;
                StartCoroutine(MakeWave());
            }
        }
    }

    private IEnumerator MakeWave()
    {
        Wave wave = waves[currentWave];
        List<EnemyClass> enemies = waves[currentWave].enemies;
        waveCountText.text = $"{currentWave + 1} / {waves.Count}";
        waveImage.fillAmount = 1f;
        waveImage.color = enemies[0].color;

        int currentEnemies = 0;
        for (int j = 0; j < enemies.Count; ++j)
            currentEnemies += enemies[j].count;
        currentWaveTick = 1f / currentEnemies;

        foreach (var enemy in enemies)
        {
            portalParticles.Play();
            waveImage.color = enemy.color;
            yield return new WaitForSeconds(0.3f);
            for (int i = 0; i < enemy.count; i++)
            {
                SoundManager.instance?.EnemySpawnPlay();
                Enemy spawned = Instantiate(enemy.prefab, startPoint.position,
Quaternion.identity).GetComponent<Enemy>();
                spawned.SetLevelPath(levelPath);
                spawned.SetStats(enemy.damage, enemy.enemySpeed, enemy.enemyHP);

                waveImage.fillAmount = waveImage.fillAmount - currentWaveTick;

                yield return new WaitForSeconds(enemy.delayBetweenSpawn);
            }

            portalParticles.Stop();
            yield return new WaitForSeconds(enemy.delayToNextEnemyGroup);
        }
        currentWave++;
    }

    public void DecreaseEnemies()
    {
        if (enemyCount > 0)
        {
            enemyCount -= 1;
        }

        if (enemyCount <= 0)
            exit.Win();
    }

    public int GetEnemyCount()
    {
        return enemyCount;
```

```
    }
}
```

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

[System.Serializable]
public class MagicZone
{
    public List<GameObject> towers = new List<GameObject>();
    public bool wasJoined = false;
}

public class MagicZonesController : MonoBehaviour
{
    public List<MagicZone> magicZones;

    private void Start()
    {
    }

    public void CheckZones()
    {
        for(int i = 0; i < magicZones.Count; ++i)
        {
            bool allTowerActive = false;
            for (int j = 0; j < magicZones[i].towers.Count; ++j)
            {
                if(magicZones[i].towers[j].transform.GetChild(1).gameObject.activeSelf)
                    allTowerActive = true;
                else
                {
                    allTowerActive = false;
                    break;
                }
            }

            if(allTowerActive && !magicZones[i].wasJoined)
            {
                magicZones[i].wasJoined = true;
                //print("is combination");
                for (int j = 0; j < magicZones[i].towers.Count; ++j)
                {

magicZones[i].towers[j].transform.GetChild(1).GetChild(1).GetComponent<AtackZone>().isInZ
one = true;

magicZones[i].towers[j].transform.GetChild(1).GetComponent<SpriteRenderer>().color =

magicZones[i].towers[j].transform.GetChild(1).GetChild(0).GetComponent<SpriteRenderer>().
color;
                }
            }
            else
            {
                if (!allTowerActive && magicZones[i].wasJoined)
                {
                    magicZones[i].wasJoined = false;
                    //print("combination died");
                    for (int j = 0; j < magicZones[i].towers.Count; ++j)
                    {

magicZones[i].towers[j].transform.GetChild(1).GetChild(1).GetComponent<AtackZone>().isInZ
one = false;
```

```
magicZones[i].towers[j].transform.GetChild(1).GetComponent<SpriteRenderer>().color =
Color.white;
                    }
                }
            }
        }
    }
}
```

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

public class MainMenuButton : MonoBehaviour, IPointerEnterHandler, IPointerExitHandler,
IPointerClickHandler
{
    public AudioClip hover;
    public AudioClip click;
    public void OnPointerEnter(PointerEventData pointerEventData)
    {
        SoundManager.instance.ButtonPlay(hover);
    }
    public void OnPointerExit(PointerEventData pointerEventData)
    {
        SoundManager.instance.ButtonStop();
    }
    public void OnPointerClick(PointerEventData pointerEventData)
    {
        SoundManager.instance.ButtonPlay(click);
    }
}
```

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

public class MainMenuManager : MonoBehaviour
{
    bool isPaused = false;
    public GameObject loseText;
    public GameObject winText;
    public GameObject nextLevelText;
    public GameObject currLevelText;

    public Slider volume;

    private float min = 1f;
    private float max = 0f;
    private float _biuldIndexOffset = 3;
    private string _levelCounterDecorator = "level : ";

    private void Start()
    {
        Time.timeScale = 1;
        if (volume)
        {
            volume.minValue = max;
            volume.maxValue = min;
            volume.value = PlayerPrefs.GetFloat("volume", max);
        }

        if (nextLevelText != null)
        {
            if (SceneManager.sceneCountInBuildSettings ==
SceneManager.GetActiveScene().buildIndex + 1)
                nextLevelText.SetActive(false);
            else
                nextLevelText.SetActive(true);
        }
    }

    private IEnumerator Delay(float delay, GameObject show, System.Action func)
    {
        yield return new WaitForSeconds(delay);
        Pause();
        show.SetActive(true);
        func();
    }

    public void Play(string sceneName)
    {
        SoundManager.instance.MenuMusicStop();
        SceneManager.LoadScene(sceneName);
    }

    public void Exit()
    {
        Application.Quit();
    }

    public void Pause()
    {
```

```
        if (isPaused)
        {
            isPaused = false;
            Time.timeScale = 1f;
            return;
        }

        Time.timeScale = 0f;
        isPaused = true;

    }

    public void Menu()
    {
        SoundManager.instance.GameMusicStop();
        SoundManager.instance.MenuMusicPlay();
        SceneManager.LoadScene("Menu");
    }

    public void Restart()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    public void Lose()
    {
        StartCoroutine(Delay(1f, loseText, () => { SoundManager.instance?.LosePlay();
}));
    }

    public void Win()
    {
        StartCoroutine(Delay(1f, winText, () => { SoundManager.instance?.WinPlay(); }));
    }

    public void NextLevel()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }

    public void OnChangeSlider(Slider slider)
    {
        PlayerPrefs.SetFloat("volume", slider.value);
        SoundManager.instance.ChangeVolume(slider.value);
    }

    public void OnLevelChange()
    {
        var levelIndex = (SceneManager.GetActiveScene().buildIndex -
_biuldIndexOffset).ToString();

        if (levelIndex == "0")
            levelIndex = "tutorial".ToString();

        currLevelText.GetComponent<Text>().text = _levelCounterDecorator + levelIndex;
    }
}
```

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

public class PlayerStats : MonoBehaviour
{
    private int _hp = 10;
    private int _maxHP;

    private MainMenuManager mainMenuManager;
    private Image healthBar;

    private void Start()
    {
        mainMenuManager =
GameObject.FindGameObjectWithTag("buttons_manager").GetComponent<MainMenuManager>();
        healthBar = GameObject.FindGameObjectWithTag("hp").GetComponent<Image>();

        _maxHP = _hp;
        healthBar.fillAmount = _maxHP / _hp;
    }

    public void TakeDamage(int damage)
    {
        if (_hp - damage <= 0)
        {
            _hp = 0;
            healthBar.fillAmount = 0;
            mainMenuManager.Lose();
            return;
        }

        _hp -= damage;
        healthBar.fillAmount = (float)_hp / _maxHP;
    }

    public float GetHP()
    {
        return _hp;
    }

}
```

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

public class SoundManager : MonoBehaviour
{
    public static SoundManager instance;
    [Header("Music")]
    public AudioSource menuMusic;
    public AudioSource gameMusic;

    [Header("Sounds")]
    public AudioSource enemySpawn;
    public AudioSource towerBuild;
    public AudioSource towerDestroy;
    public AudioSource shoot;
    public AudioSource buttonsSource;
    public AudioSource baseHurt;
    public AudioSource win;
    public AudioSource lose;
    public AudioSource noMoreTowers;

    private List<AudioSource> sfx = new List<AudioSource>();

    private void Awake()
    {
        if (instance == null)
        {
            instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
        DontDestroyOnLoad(this);
        AudioSource[] sfxSource =
        { baseHurt, buttonsSource, shoot, towerDestroy,
            towerBuild, enemySpawn, lose, win, noMoreTowers,
            gameMusic, menuMusic
        };
        sfx.AddRange(sfxSource);
    }

    private void Start()
    {
        //  play music
        string currentScene = SceneManager.GetActiveScene().name;
        if (currentScene == "Menu")
        {
            //  play menu music
            menuMusic.Play();
        }
    }

    public bool GameMusicPlaying()
    {
        return gameMusic.isPlaying;
    }

    public void GameMusicPlay()
    {
```

```
    gameMusic.Play();
}

public void GameMusicStop()
{
    gameMusic.Stop();
}

public void MenuMusicPlay()
{
    menuMusic.Play();
}

public void MenuMusicStop()
{
    menuMusic.Stop();
}

public void EnemySpawnPlay()
{
    enemySpawn.Play();
}
public void TowerBuildPlay()
{
    towerBuild.Play();
}
public void TowerDestroyPlay()
{
    towerDestroy.Play();
}
public void ShootPlay()
{
    shoot.Play();
}
public void ButtonPlay(AudioClip clip)
{
    if (!clip)
    {
        buttonsSource.Stop();
        return;
    }

    buttonsSource.clip = clip;
    buttonsSource.Play();
}
public void ButtonStop()
{
    buttonsSource.Stop();
}
public void BaseHurtPlay()
{
    baseHurt.Play();
}

public void WinPlay()
{
    Time.timeScale = 0;
    win.Play();
}

public void LosePlay()
{
    Time.timeScale = 0;
    lose.Play();
```

```
    }

    public void NoMoreTowersPlay()
    {
        noMoreTowers.Play();
    }

    public void ChangeVolume(float value)
    {
        foreach (var sfxElement in sfx)
            sfxElement.volume = 1f - value;
    }
}
```

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

public class SpawnTower : MonoBehaviour
{
    public ParticleSystem ps;
    private void Start()
    {
        ps = transform.parent.GetComponentInChildren<ParticleSystem>();
    }
    private void OnMouseDown()
    {
        if (Time.timeScale == 0f)
            return;

        if (TowersController.instance.CanSpawnTower())
        {
            ps.Play();
            TowersController.instance.IncreaseTowersCount();
            transform.parent.GetChild(1).gameObject.SetActive(true);
            gameObject.SetActive(false);
            SoundManager.instance?.TowerBuildPlay();

GameObject.FindGameObjectWithTag("MagicZone").GetComponent<MagicZonesController>().CheckZ
ones();
            return;
        }
        SoundManager.instance?.NoMoreTowersPlay();

    }
}
```

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

public class Tower : MonoBehaviour
{
    public ParticleSystem psDestroy;
    private AtackZone atackZone;

    private void Start()
    {
        atackZone = GetComponentInChildren<AtackZone>();
    }

    private void OnMouseDown()
    {
        if (Time.timeScale > 0f)
        {
            SoundManager.instance?.TowerDestroyPlay();
            psDestroy.Play();
            TowersController.instance.DecreaseTowersCount();
            transform.parent.GetChild(0).gameObject.SetActive(true);
            gameObject.SetActive(false);

GameObject.FindGameObjectWithTag("MagicZone").GetComponent<MagicZonesController>().CheckZ
ones();
            atackZone.ResetState();
        }
    }
}
```

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

public class TowersController : MonoBehaviour
{
    public static TowersController instance = null;
    [SerializeField] private int countOfTowers = 0;
    [SerializeField] private int maxCountOfTowers = 5;

    public Text towersCount;
    private void Awake()
    {
        if (instance == null)
            instance = this;
        else
            Destroy(gameObject);
    }

    void Start()
    {
        OnTowersCountChange();
    }

    public void OnTowersCountChange()
    {
        GameObject.FindGameObjectWithTag("towers_counter").GetComponent<Text>().text =
$"{countOfTowers} / {maxCountOfTowers}";
    }

    public void IncreaseTowersCount()
    {
        if (countOfTowers < maxCountOfTowers)
        {
            countOfTowers++;
            OnTowersCountChange();
        }
    }

    public void DecreaseTowersCount()
    {
        if (countOfTowers > 0)
        {
            countOfTowers--;
            OnTowersCountChange();
        }
    }

    public bool CanSpawnTower()
    {
        return countOfTowers < maxCountOfTowers;
    }
}
```

Додаток В

Роздатковий матеріал