КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
імені ТАРАСА ШЕВЧЕНКА
Факультет інформаційних технологій
Кафедра прикладних інформаційних систем

122 «Комп'ютерні науки»
(шифр і назва спеціальності)

«Прикладне програмування»
(назва освітньої програми)

Кваліфікаційна робота бакалавра

на тему: «Комп'ютерна гра для вивчення точних наук»

Виконала _____ _____
(Підпис)
Гавриленко Надія Юріївна
(прізвище, ім'я, по батькові)

Керівник _Плескач Валентина Леонідівна
(прізвище, ім'я, по батькові)

_____ _____
(Резолюція «До захисту»)

Попередній захист:

_____
(Висновок: "До захисту в екзаменаційній комісії ")

Завідувач кафедри _____ Плескач В.Л.
(Підпис) (Прізвище, ініціали) (Дата)

Київ – 2022

ВІДОМІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ БАКАЛАВРА

Складові частини дипломної роботи	Обсяг, арк. 85
Титульний аркуш	1
Календарний план дипломної роботи	1
Відомість дипломної роботи	1
Анотація	1
Анотація (іноземною мовою-англійською)	1
Зміст	1
Перелік скорочень, умовних позначень, термінів	0
Вступ	3
Розділ 1	20
Розділ 2	20
Розділ 3	8
Висновки	2
Перелік використаних джерел	3
Додатки	28

<table>
<tr><td></td><td></td><td></td><td></td><td colspan="3" rowspan="2">ДП XXXX 00.000.00</td></tr>
<tr><td></td><td>ПІБ</td><td>Підп.</td><td>Дата</td></tr>
<tr><td>Розробн.</td><td>Гавриленко Н.Ю.</td><td></td><td></td><td rowspan="4">Відомість дипломної роботи</td><td>Лист</td><td>Листів
85</td></tr>
<tr><td>Керівн.</td><td>Плескач В.Л.</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Н/контр.</td><td>Базилюк А.М.</td><td></td><td></td><td colspan="2" rowspan="2"></td></tr>
<tr><td>Зав.каф.</td><td>Плескач В.Л.</td><td></td><td></td></tr>
</table>

АНОТАЦІЯ

Кваліфікаційна робота бакалавра містить: 85 сторінок, 17 рисунків, 30 використаних джерел. Метою роботи є інтелектуалізація навичок і компетенцій гравців у сфері точних наук за допомогою комп'ютерної гри. Об'єктом дослідження є процеси навчання людини під час проходження комп'ютерної гри. Предмет дослідження – засади, принципи, методи побудови комп'ютерної гри, навчального процесу. Методи дослідження - метод аналізу, описовий метод (проектування алгоритмів роботи програми, ведення документації та формування звіту до виконаної роботи), метод синтезу (підбір та комбінування популярних часткових програмних і теоретичних рішень з метою отримання системи компонентів, яка підходить до виконання поставленого завдання), системний метод. У результаті роботи було розроблено прототип комп'ютерної гри, що має на меті допомогти здобувачам освіти старших класів школи у освоєнні точних наук, в цьому конкретному випадку – фізики. Програмна реалізація виконана на ігровому рушії Unity мовою програмування C#.

Ключові слова: геймдизайн, гра для навчання, головоломка, патерни програмування, розробка ігор на Unity.

ABSTRACT

The bachelor's thesis contains 85 pages, 17 drawings, and 30 used sources. The aim of the work is to intellectualize the skills and competencies of players in the field of exact sciences using computer games for it. The object of research is the processes of human learning while playing a computer game. The subject of research - the principles and methods of building a computer game and the learning process in gaming. Research methods - method of analysis (search for theoretical data, critical consideration, identification of approaches that will be needed in the implementation of the project), descriptive method (design of algorithms, documentation, and reporting on the work performed), synthesis method (selection and combination of popular partial software and theoretical solutions in order to obtain a system of components that is suitable for the task). As a result of the work, a prototype of a computer game was developed, which aims to help high school students in learning exact sciences, in this case - physics. The software implementation is made using the Unity game engine and the C# programming language.

Keywords: game design, learning game, puzzle, programming patterns, game development on Unity.

ЗМІСТ

ВСТУП

За останнє сторіччя технології зробили величезний стрибок вперед і продовжують розвиватися. Сьогодні складно уявити таку сім'ю, яка б не мала вдома ні комп'ютера (чи ноутбука), ні смартфона. І можливості кожного такого пристрою вражають. Але, окрім програмного забезпечення, яке вважається корисним та допомагає в роботі чи повсякденному житті, є величезна кількість програм, ставлення суспільства до котрих загалом неоднозначне – ігор.

Актуальність теми. Хтось назве ігри марною тратою часу, а хтось – відпочинком. Багато людей не бачитиме в них цінності, але не менше людей будуть просто продовжувати грати. Не можна заперечувати, що гравців у сучасному світі дуже багато: це і маленькі діти, і підлітки, і дорослі. Грають на персональних комп'ютерах, на телефонах, на спеціальних пристроях – консолях. Під час та замість відпочинку, вдома та у транспорті. Влаштовують змагання та клуби за інтересами. На перший погляд можна подумати, що така одержимість – це лише зло, вона не несе людині нічого доброго. І справді, все добре лише в міру. Але також правдою є те, що в іграх існує багато менш очевидних моментів, які несуть людині користь, інколи навіть значну. Через ігри може відбуватися соціалізація, тренуватися креативне мислення, здатність приймати рішення, розподіляти ресурси, визначати пріоритети… Цей список можна продовжувати досить довго. Людину складно «відірвати» від цікавої гри. А що, якщо вона буде не лише грати, а й одночасно навчатися? Нікому не нудно та не ліньки грати. А тепер просто уявімо, яким був би світ, якби людям не було нудно або ліньки вчитися. Саме тому важливо досліджувати, що приваблює людей в іграх, застосовувати ці принципи у повсякденному житті та робити такі ігри, які нестимуть щось корисне у життя.

Мета дослідження - інтелектуалізація навичок і компетенцій гравців у сфері точних наук за допомогою комп'ютерної гри.

Завдання дослідження. Для досягнення поставленої мети буде необхідно виконати такі завдання:

- дослідити теоретичні основи побудови комп'ютерних ігор, а саме – засади та принципи, методики, за допомогою яких створюють навчальні ігри;
- висвітлити історію еволюції ігор;
- здійснити аналіз ігрової аудиторії;
- визначити негативні та позитивні тенденції впливу комп'ютерних ігор;
- здійснити аналіз програмно-технічних рішень із побудови навчальних ігор;
- описати основні принципи геймдизайну;
- здійснити огляд можливостей ігрового рушія Unity;
- здійснити вибір технологій і практик проектування комп'ютерних ігор;
- здійснити проектування, реалізацію, тестування, впровадження комп'ютерної гри для навчання;
- описати стек технологій реалізації комп'ютерної гри для навчання;
- висвітлити перспективи розвитку комп'ютерної гри для навчання точним наукам.

Об'єкт дослідження – процеси навчання людини під час проходження комп'ютерної гри.

Предмет дослідження – засади, принципи, методи побудови комп'ютерної гри, навчального процесу.

Методи дослідження є наступними:

- метод аналізу (пошук теоретичних даних, критичний їх розгляд, виявлення тих підходів, які знадобляться при втіленні проекту);
- описовий метод (проектування алгоритмів роботи програми, ведення документації та формування звіту до виконаної роботи);

- метод синтезу (підбір та комбінування популярних часткових програмних і теоретичних рішень з метою отримання системи компонентів, яка підходить до виконання поставленого завдання).

Практичне значення одержаних результатів полягає у тому, що буде розроблений прототип комп’ютерної гри, що має на меті допомогти користувачам (потенційна аудиторія – здобувачі освіти старших класів школи) у освоєнні точних наук.

Структура роботи. Робота складається зі вступу, трьох розділів, поділених на підрозділи, висновку та списку використаних джерел. Загальний обсяг роботи складає 85 сторінок. Список використаних джерел налічує 30 найменувань.

У першому розділі розглянуто історію ігор та досліджено теоретичні принципи, які можуть зробити гру цікавою та успішною, а також нинішній стан ігрової індустрії загалом. У другому розділі увагу сконцентровано на плануванні прототипу гри та виборі та вивченні технологій, які знадобляться при реалізації. У третьому розділі наведено опис реалізацій систем гри та подальші перспективи цієї роботи.

РОЗДІЛ 1

ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ КОМП’ЮТЕРНИХ ІГОР

1.1 Історія еволюції ігор

Коли пересічна людина думає про ігри, вона згадує або дитячі веселощі, або підлітків, які проводять години перед комп’ютером, або, можливо, якісь азартних

гравців в казино. Трохи згодом на думку приходять настільні та спортивні ігри. Здається, що ці три типи людей та ігор кардинально відрізняються. Але вони мають чимало спільних рис та в певному розумінні спільне коріння.

Але чому зв'язок між настільки різними речами важливий? Чому не можна досліджувати сучасні ігри, не враховуючи той шлях, що вони пройшли протягом віків?

У сучасних іграх багато чим керує маркетинг та поточні захоплення суспільства – а це вкрай мінливі речі. Можна покладатись на вдачу, намагаючись створити хіт, а можна досліджувати ті засади, що роблять гру популярною та цікавою завжди. І хоча це дуже абстрактна сфера, в якій кожна людина може зробити свої власні унікальні висновки, певні принципи виділяються досить чітко і усі з ними погодяться.

Неможливо датувати той час, коли ігри виникли. Можливо, вони існували завжди, починаючи з дитячих забав. Адже діти пізнають світ через гру, часто імітуючи якісь ситуації, приближені до реальних життєвих. Це присутнє навіть у тварин: як дитинчата хижаків починають битися, але не наносять один одному шкоди – вони лише імітують реальний бій. Отже, занотуємо, що першочергово ігри виникли як спосіб чомусь навчитися.

Перейдемо до більш «серйозних» розваг, що створені з чіткою метою розробити гру. Спершу були, звичайно, настільні ігри. Це такі ігри, для яких потребувалися спеціальні предмети: дошка та/або якісь фігурки. Часто вони були досить імпровізованими: дошку малювали на землі, заміть фігурок брали боби, тощо.

Найдавнішою грою вважається Королівська гра Ура, в яку грали шумери на Ближньому Сході близько III тисячоліття до н. є. Це певно не найперша гра у світі, але це та, яку вдалося більш-менш датувати. Незважаючи на назву, грали в неї не лише королі, а й взагалі будь-хто: хоча і були красиві набори для цієї гри, було

знайдено і графіті, що означало, що грати в неї могла і звичайна людина, просто намалювавши собі дошку. З часом ця гра отримала зв'язок з релігією: гравці вважали, що через неї можна отримувати пророцтва. Згодом її витіснив прообраз гри в нарди.

Ігри виникали в усіх країнах протягом усієї історії. Часто бувало таке, що деталі відрізнялися, але основна ідея була однією і тією ж. А часто – виникнувши в одній країні, гра робила подорож світом, модифікуючись за цей час, і поверталася «додому» вже іншою, набагато покращеною, витісняючи свій прообраз. Така подорож сталася з грою в шахи.

А ще до того, як з'явилися шахи, в Скандинавії була популярна гра Тафл. Вона імітувала осаду замку.

Але, якщо копнути ще глибше, насправді прообраз шахів з'явився в Індії під назвою Чатуранга. Ціллю гри було знищити армію противника. За однією з легенд, її придумали, щоб втішити царицю, яка була засмучена тим, що її сини воюють за владу. Гра продемонструвала їй, що її діти не вбивають один одного голими руками, а що війна насправді – це скоріше комбінація різних невеликих, часто незалежних або безконтрольних подій, в яких приймає участь велика кількість людей. Пізніше ця гра пройшла через персів, арабів, європейців (де набула вигляду, який нам знайомий зараз) і знову повернулася в Індію.

Майстерну гру в Го настільки поважали в Китаї, що вона ввійшла в список основних мистецтв на одному рівні з живописом та музикою.

Приблизно у VI ст. до н. е. у Греції ігри отримують змагальну складову. Змагались у спорті, поезії, музиці, театрі. На зміну цьому погляду на ігри в Римі виникає нове: ігри стають способом розслабитися та розважитися. У середні часи, у розквіт християнства, азартні ігри починають засуджувати, але це не означає, що в них перестають грати.

У період Високого Середньовіччя інтелектуальні ігри повертаються у Європу, але вже як розвага для аристократів. Це було настільки виразно, що у юнаків-аристократів настільні ігри входять до обов'язкової навчальної програми. Також з'являються настільки «інтелектуальні» ігри, що зрозуміти їх правила було дуже складною задачею; такі ігри слугували скоріше для демонстрації своєї статусності, ніж для гри.

Цікаво, що приблизно до XVII ст. ігри не поділялися на «дитячі» та «дорослі». Лише у цей час такий вид дозвілля переходить до дитячого, або й взагалі, зневажливо, «сільської простоти», і лише з цього часу дітей почали берегти від азартних ігор.

Однак згодом зрозуміли, що утримати дітей та підлітків від «сільської простоти» - рухливих розваг - не вдасться, і навіть почали додавати їх у навчальну програму. Ідею підхопили медики, і звідси з'явився спорт – корисний вид гри. Також виявили, що фізично розвинуті люди будуть краще воювати, і спорт став частиною політичної пропаганди. Цей вид гри схвалювався, а настільні та інші залишалися розвагою для селянства.

Досить скоро спортивні версії з'являються не тільки у рухливих ігор.

Починається епоха монетизація ігор: численні видавництва перепрофілюються на випуск настільних ігор. XX сторіччя можна вважати часом їх розквіту.

Також у XX сторіччі виникають комп'ютерні ігри. І на перший погляд здається, що це щось зовсім нове, оскільки взагалі технології тоді були чимось новим. Але насправді це просто новий формат для все тих же старих ідей та бажань. Наприклад, 1958 року була створена гра Tennis For Two – комп'ютерне представлення звичайного тенісу. Вона була створена на комп'ютері, призначеному для прорахунку траєкторії польоту балістичних ракет – такий комп'ютер добре

вправлявся з необхідністю прорахувати траєкторію тенісного м’ячику. Але для загального доступу така гра явно не підходила, тому зараз її мало хто пам’ятає.

Першою комп’ютерною грою вважають «Spacewar!», що була випущена 1962 року. Саме ця першою набула популярності та змогла стати масовою. В неї могло грати дві людини, кожна з яких мала свій космічний корабель. Завдання – знищити опонента.

Більшість ігор в ті часи мали мінімальну графіку та функціонал, але вже приваблювали гравців. Вони були динамічними. Певні типи ігор були сконцентровані на якісь історії, і гравець повинен був читати та вводити свої дії з клавіатури. Згодом з’являється управління мишкою, і виникає новий жанр Point-and-click, де гравцю більше не потрібно вводити команди вручну (для цього фактично їх вгадуючи), а достатньо клікати на предмети в самій грі. Розвивається графіка. До ігор додається музикальне супроводження.

Більшість популярних на сьогодні жанрів зародилися на самому початку історії комп’ютерних ігор. Наприклад, перший First Person Shooter – «стрілялка від першого лиця» - з’явився у 1973 році під назвою Maze War. Тоді ще не було текстурованої графіки та гравець переміщувався по приміщенню, яке нарисовано «каркасно» - предмети мали лише контури, без заливки. І хоча перша справжня популярна стрілялка з’явилася лише через двадцять років (Doom, 1993), основні риси жанру можна було побачити вже в Maze War та ще кількох іграх, що були після неї.

З моменту виникнення аудиторія комп’ютерних ігор значно змінилася. На початку це були скоріше програмісти-любителі, для яких будо цікавою задачею створити свою гру. Їх аудиторія складалася з малочисельних, але завзятих гравців – зазвичай любителів настільних рольових ігор та стратегій.

На цьому етапі буде доречно поділити аудиторію гравців на три категорії:

- Хардкор (Hardcore) – це люди, для яких ігри є основним хобі. Вони надають перевагу складним об’ємним іграм та не шкодують часу, щоб розібратися в правилах та досягти справжньої майстерності. Саме це відрізняє хардкорні ігри від інших: це не просто (і не завжди) ігри з великим сюжетом, це ігри, в яких потрібно тривалий час відточувати навички самого гравця.
- Кор (Core) – ті гравці, для яких ігри в першу чергу є розвагою. Вони з задоволенням зануряться цікавий ігровий світ, але не витрачають на це занадто багато часу.
- Казуал (Casual) – люди, які використовують ігри в першу чергу як «вбивець часу». Найбільше їх можна побачити в транспорті з якоюсь простою мобільною грою.

Аудиторія відеоігор початково складалася з хардкор-гравців, оскільки потрібно було відчутно заморочитися, щоб створити або пограти в таку гру. Але з моменту виникнення вона увесь час розширювалася.

З 70-х років справа стає більш комерційною: ігри починають приносити прибуток. Продаються ігрові консолі. Аудиторія стає ширшою, складається переважно з студентів та інженерів. Також з’являються ігрові автомати в барах, це дає шанс кор-аудиторії доєднатися до світу відеоігор.

До двохтисячних років популярність відеоігор стрімко зростає і приблизно у цей час розмах сягає таких масштабів, що вже конкурує з кінематографом. І надалі продовжує зростати! Оскільки ігри стають більш доступними, з’являється казуальна аудиторія. А з розповсюдженням смартфонів вона набуває величезного розміру.

Сьогодні для розробки ігор залучено команди з сотень людей, мільйонні бюджети, найновіші технології. Для озвучення персонажів (а часто і для анімації) запрошують відомих акторів. Системні вимоги для найновіших ігор продовжують

зростати, і для багатьох людей бажання пограти в новенький хіт – основна причина покупки нового комп'ютера або ігрової консолі. Навколо окремих ігор гуртуються справжні суб-культури з фанатів, які постійно генерують новий контент за тематикою улюбленого світу.

Повернемося до питань, поставлених у початку цього параграфу. Протягом тисячоліть ігри невтомно розвивалися, пережили декілька підйомів та падінь, але завжди був якийсь напрям, який неодмінно затягував людей. З появою технологій швидкість цих процесів, як і все в світі, почала не просто зростати, а примножуватись.

Насправді, люди відкрили, що з появою технологій в один пристрій можна «запхнути» усе, що протягом багатьох поколінь відтворювали за допомогою гральних дошок, карт, кубиків, фігурок, словесно, уявно, фізичними зусиллями… «Запхнути» правила, ідеї, свої бажання, філософію, і взагалі цілий світ.

Дійсно, можна зробити програмний аналог будь-якої настільної, азартної чи спортивної гри. Можна надати гравцям таку ж можливість спілкуватися один з одним, роздумувати над своїми діями. Можна перенести спортивні ігри в монітор та дати людині контролер, що дозволить керувати рухами персонажа, а самому не рухатись. А якщо соціуму в якийсь момент не хочеться – не проблема, адже можна грати з комп'ютером! Не потрібно збирати цілу команду людей, щоб пограти. Диво техніки буде в тебе завжди вдома, готове скласти компанію в будь-який момент.

Але просто втілити всі мотиви настільних та спортивних ігор – це ще не все, що зробили відеоігри. У свій час кінематограф став революцією, тому що поєднав літературу (сюжет, сценарій), візуальну естетику (відео) та звук (музичний супровід, ораторське мистецтво, акторська гра). Відеоігри також це все поєднують, але додають дещо унікальне – це вибір, який має гравець. Якщо кінофільми лише показують світ, у якому глядачу хотілося б жити, то відеоігри дають йому можливість справді там опинитися. Ігри передають відчуття, симулюючи умови, в

яких ці відчуття виникають. Ігри передають досвід про ці відчуття. Звісно, це не йде ні в яке порівняння з реальним життєвим досвідом (і гравці мають це пам’ятати). Але це краще, ніж нічого, особливо враховуючи, що не у всіх і не завжди є можливість отримати реальний досвід.

І знов-таки, повертаючись до самого початку. Ігри майже завжди є симуляцією якихось реальних життєвих ситуацій. Це тренування. Це можливість опинитися в умовах, які в реальному житті відтворити проблематично (або навіть не бажано – якщо мова про небезпечні ситуації). Саме тому ігри постійно дають нові враження, новий досвід. Фактично, людина, граючи в ігри, завжди навчається. Складно представити, щоб комусь було цікаво робити одне і те ж абсолютно однаковим чином протягом багатьох годин – тому в іграх такого немає. В цікавій грі складність поступово збільшується, змушуючи гравця розкривати все більше можливостей свого мозку, або реакцію, або координацію рухів. Саме тому можна вважати, що гра – це завжди розвиток. Що робить ставлення суспільства до ігор неоднозначним – так це питання, в якій сфері цей розвиток відбуватиметься. І це вже у владі розробників.

1.2 Аналіз ігрової аудиторії

У минулому розділі ми вже виділили три типи гравців:

- Хардкор – найбільш завзяті, для яких ігри – це серйозне захоплення, котрі вкладають в гру багато зусиль;
- Кор – середні гравці, які полюбляють пограти в ігри, але для них це більше як розвага;
- Казуальні гравці – які надають перевагу нескладним іграм і грають з метою «вбити трохи часу».

У нас час складно знайти дитину чи підлітка, які зовсім не грає в відеоігри. Багато з них будуть відноситися до кор-аудиторії. Люди старшого покоління залучаються менш активно, і часто це «вбивачі часу» в транспорті, або якісь нескладні ігри, що допоможуть відпочити після робочого дня.

Станом на сьогодні ми маємо надзвичайно масову казуальну аудиторію (яка, однак, найгірше монетизується); невелику аудиторію хардкор-гравців (яка приносить основний прибуток індустрії, тому що саме хардкор-гравці готові вкладати в гру гроші, і часто витрачають досить великі суми); та середнього обсягу аудиторію кор-гравців. А також рівень розвитку технологій такий, що буквально будь-хто може створити власну гру. Звідси виникають інді-студії – невеличкі команди з малим бюджетом, а інколи і зовсім без нього. Прості ентузіасти, які хочуть втілити свою ідею.

Прогнозовано, що до 2023 року вартість світового ринку відеоігор перевищить $200 млрд[1]. Але що є найпопулярнішим і чому?

В першу чергу варто згадати великі багатокористувацькі ігри. Цілі та правила можуть значно відрізнятися.

В таких шутерах, як Counter Strike, гравець отримує команду та має перемогти ворожу команду таких же гравців. В цьому є схожі риси з не менш популярною Dota 2, але там у гравців є ще власний замок, який потрібно захистити та знищити

[1] Gaming - Statistics & Facts. URL: https://www.statista.com/topics/1680/gaming/

ворожий. В іграх типу «королівська битва» кожен сам за себе, і завдання – залишитися єдиним «живим» на карті.

Не завжди гра ставить ціль перемогти інших гравців. Мабуть, справедливо сказати, що не завжди гра ставить єдину ціль взагалі. Такі ігри, як Rust, Minecraft, хоч і здаються дуже різними, мають спільну рису: це симулятор виживання, пісочниця, де ти можеш будувати все, що тобі заманеться, та взаємодіяти з іншими гравцями. Вони можуть допомагати тобі, а можуть заважати.

Якщо комусь не до смаку воювати з реальними людьми, існують ігри і для них: це кооперативні, де гравці збираються у команду та, допомагаючи та підтримуючи один одного, йдуть до спільної мети. Якщо гра добре збалансована, то це дозволяє кожному члену команди відчувати себе необхідним та робити внесок в майбутню перемогу.

Онлайн-ігри цікавлять гравців, які шукають соціалізації та взаємодії з іншими людьми. Часто така гра виступає платформою для знайомств та пошуку товаришів за інтересами. Але не менш важлива деталь – це здатність прилюдно заявити про свою майстерність в чомусь (для вмілих гравців) та отримати схвальні відгуки оточуючих, або навіть заздрість (що для деяких людей теж може бути приємним).

Для людей, які не полюбляють активну взаємодію з іншими гравцями, є ігри на одного користувача. В деяких з них присутній розвинутий сюжет, який можна порівняти з захоплюючою книгою, та не менш цікаві та харизматичні персонажі. Такі ігри відносять до жанру інтерактивного кіно: вони мають не дуже складний геймплей, а акцент ставиться на наративі. Яскравий приклад таких ігор – «Life is Strange», «Detroit», «The Last of Us». Часто (але зовсім не обов’язково) такі ігри мають графіку, максимально наближену до реалізму. Більш «дієві», екшенові ігри, які не обділені сюжетною складовою – «Hellblade: Senya’s Sacrifice», «Cyberpunk 2077», серія «The Witcher». Такі ігри налічують 50-80 годин геймплею, їх розробкою займаються величезні студії зі значним бюджетом. Гравці цих ігор

шукають захопливу історію, бажають максимального заглиблення в світ гри, хочуть відчути себе в ролі героя – фактично, переслідують ті ж цілі, що і любителі хорошої художньої літератури.

Щоб зробити хорошу, цікаву (в тому числі і сюжетну) гру, не обов'язково мати величезний бюджет та багато людей в команді. Інді-ігри створені саме такими розробниками. Зазвичай у ядрі такої гри якась незвичайна ідея або її оформлення. Наприклад, в «Celeste» гравець має лише три дії: стрибок, деш (ривок персонажа вперед) та лазіння по стінам. Тим не менш, гра має велику кількість різноманітних рівнів, що дозволить гравцю не тільки протестувати свою спритність, а й розкажуть цікаву історію. «Journey» занурює гравця в медитативну атмосферу, а «Gris» окрім цього, з психологічною точністю проводить гравця через всі стадії переживання втрати. Поширені тут і головоломки. В «Inside» та «Limbo» буде потрібно цікавим чином використовувати предмети на рівні. «Inscryption» - карточна гра з сильним сюжетом, що змушує напружено думати над кожним своїм кроком.

Головне, що об'єднує прихильників інді-ігор – це любов до чогось нового і незвичного.

Варто також згадати ігри-пісочниці, як їх називають. Основний акцент в таких іграх ставиться на вивчення світу та побудову чогось. До таких ігор можна віднести «Minecraft» (хоча ця гра має також і сильну соціальну складову) і «Valheim» (в неї можна грати як одному, так і з командою). Такі ігри дають гравцю простір для творчості, наприклад, можна будувати найрізноманітніші споруди з ресурсів, які вдасться знайти. Часто є ще елемент виживання: гравцю необхідно, окрім збору ресурсів на реалізацію своїх архітектурних ідей, дбати про стан здоров'я персонажа. Наприклад, у «Subnautica» потрібно добувати їжу та створювати нові пристрої, щоб мати змогу більше досліджувати світ та просуватися по сюжету.

Трохи схожими на «пісочниці» є ігри-стратегії з будівництва. Але тут акцент вже не на вивченні світу, а на розумному та своєчасному розподіленні ресурсів, яких зазвичай дуже обмежена кількість. В «Frostpunk» комбінуються елементи розподілення ресурсів та виживання: необхідно стратегічно мислити над тим, які задачі є першочерговими, на які роботи направити людей (котрих не так вже багато!), і в разі помилки – місту загрожує смерть.

Популярними є ігри-симулятори. Відома серія «Need for Speed» - гонки та симулятор водіння. Такого ж жанру «Forza Horizon». «WarThunder» - симулятор управління літаком часів Другої Світової Війни, а «World of Tanks» - танку. Також існують симулятори спортивних ігор. Цікаво, що не завжди симулятор має на меті передати максимально точно схему взаємодії людини з реальним пристроєм. Гравець ніколи на навчиться пілотувати, граючи в «WarThunder». Такі розважальні ігри мають на меті скоріше передати емоції від польоту, які, на думку людини, вона б очікувала відчути, пілотуючи літак. Гра Microsoft Flight Simulator є набагато більше наближеною до реальності: тут відтворено реальну фізику польоту, ландшафти, аеропорти, літаки. Також існують дійсно професіональні програми, що використовують для тренування справжніх пілотів – але це більш специфічний випадок, пересічному гравцю вони можуть здатися занадто складними або нудними.

Жанрів ігор є така велика кількість, що перерахувати їх всі досить складно. Часто ігри поєднують у собі декілька різних жанрів або можуть бути побудовані так, що кожен гравець самостійно вирішуватиме, яка тактика для проходження для нього найзручніша та найцікавіша. Наприклад, багато ігор дають вибір між екшеном та стелсом: можна проходити їх через гучні перестрілки з ворогами, а можна тихенько прокрадатися повз них до цілі, взагалі не вв’язуючись в бій.

Безумовно, кожна людина унікальна та має свої вподобання. Але все ж таки, дуже узагальнюючи, можна виділити чотири категорії гравців. Таку категоризацію

називають моделлю психотипів Бартла[2]. Її можна кратко продемонструвати наступною схемою (Рисунок 1.1).

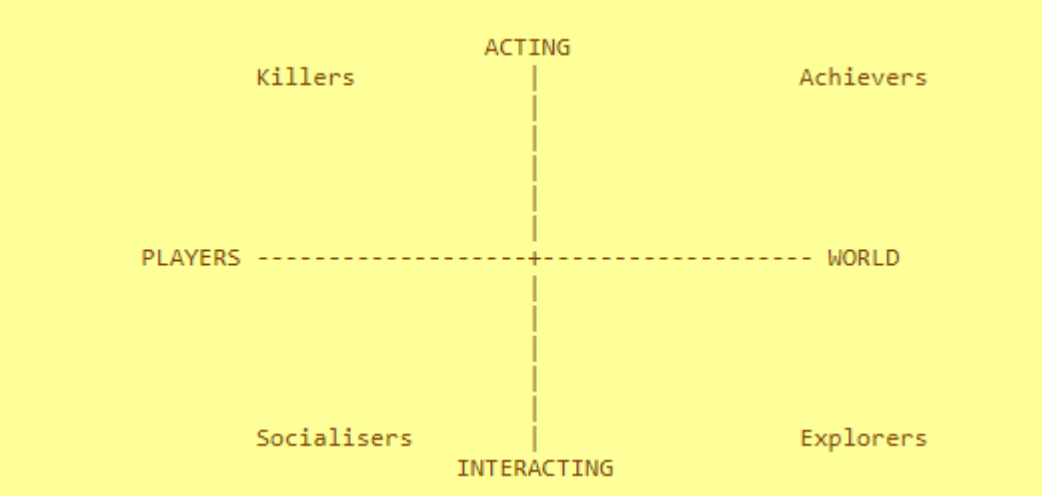

Рисунок 1.1 - Психотипи гравців за Бартлом

Тут на «координатній площині» ми бачимо протилежності: дія – взаємодія та гравці – світ. Залежно від степені, в якій дані види гравців зацікавлені в кожній частині цієї ігрової координатної площини, Бартл виділяє чотири типи гравців:

- Кар'єристи (Achievers) – це ті гравці, які обожнюють накоплювати ігрові ресурси, силу, для них основне в грі – це власний прогрес та ріст. Статистично, цей тип аудиторії найпоширеніший. Варто відмітити, що саме кар'єристи затримуються у грі найдовше (оскільки вони хочуть досягти усіх можливих висот).
- Дослідники (Explorers) – гравці, що максимально зацікавлені у вивченні ігрового світу. Вони прагнуть розгадати всі загадки, люблять, коли в грі багато різних механік та є можливість задіяти свій розум для нестандартних вирішень проблем. Ця категорія гравців теж проводить у

[2] "Hearts, Clubs, Diamonds, Spades: Players Who Suit MUDs", Richard Bartle (1996) https://mud.co.uk/richard/hcds.htm

грі багато часу, досконально вивчаючи її, та не дуже люблять переходити у інші ігри. Також вони не люблять, коли їх порівнюють з іншими гравцями (через рейтинги, масові події) – як бачимо, на схемі вони знаходяться максимально далеко від частині осі «гравці».

- Соціалайзери (Socializers) – гравці, для яких найважливіше – це взаємодія з іншими гравцями. Вони зацікавлені у спілкуванні та прагнуть до популярності. Сама гра для них відходить на другий план – вона є лише середовищем, в якому вони втілюють свої соціальні потреби. Ця категорія гравців погано монетизується, але часто саме соціалайзери активно приводять у гру нових користувачів та утримують старих.
- Кіллери (Killers) – ці гравці хочуть змагань та перемоги над іншими, відчуття якоїсь влади та переваг. Для них є нестерпними монотонні рутинні ігри. Статистично, ця група найменш чисельна, але найкраще монетизується.

Існує також розширена модель психотипів Бартла. Для неї схема буде вже трьохмірною: з додатковою віссю «Свідоме - Несвідоме» (Рисунок 1.2).

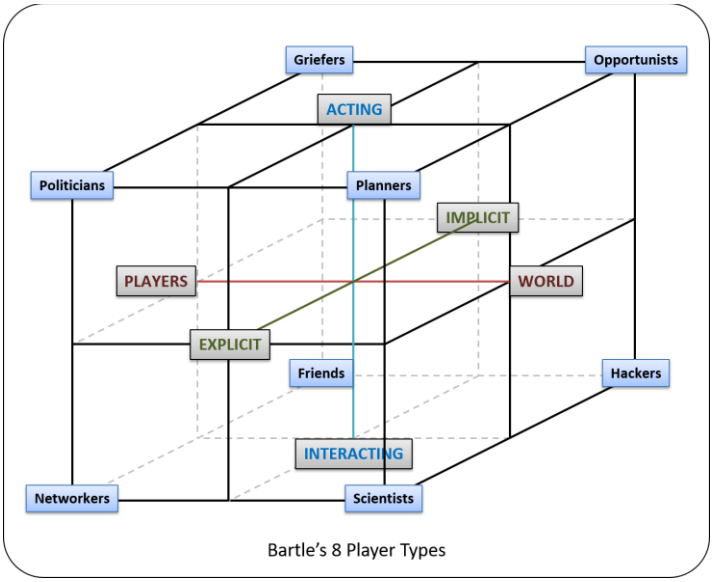

Рисунок 1.2 – Розширена модель Бартла

Тут Кар’єристи розділяються на любителів планувати кожен свій крок та на тих, хто діє по ситуації; Дослідники – на Вчених, які підходять до вивчення світу з певною стратегією, та Хакерів, які діють спонтанно; Соціалайзери – на тих, хто використовує інших гравців для отримання максимальної вигоди, та тих, хто просто дружить; та Кілери – на Політиканів, які користуються своєю силою для цілеспрямованого досягнення нових висот, та Хейтерів, які просто виплескують свій негатив у грі.

Існують і інші способи категоризації гравців. В інших категоризаціях також відіграє роль мотивація гравців – чого вони прагнуть, що шукають у грі. Парадоксально, але не всі гравці дійсно хочуть грати в неї. Хтось приходить за соціальною взаємодією, хтось – за відчуттям власної сили, якого він може досягнути, ламаючи все навкруги.

Людей певного психотипу у «чистому» вигляді практично не існує, зазвичай це комбінація двох-трьох психотипів у певних пропорціях. Тому немає сенсу намагатися виявити психотип гравця. Категоризацію використовують переважно з метою врівноважити активності гравців у самій грі, зрозуміти, на чому варто зробити акцент, якого типу гравців (і їх активностей) потрібно приваблювати до гри більше.

1.3 Негативні та позитивні тенденції впливу комп’ютерних ігор

Так само як фільми та література, ігри значно впливають на наше світосприйняття. На жаль, силу та напрям цього впливу не завжди можна передбачити.

Основні напрями впливу, яких бояться люди: це жорстокість та залежність. У центрі гри зазвичай лежить якийсь конфлікт, а вирішити конфлікт, не вдаваючись до жорстокості, зазвичай не можна. Шахи не вважаються жорстокою грою, але кожен гравець жертвує своїми фігурами та забирає фігури опонента, і ніхто не

засуджує не комп’ютерні дитячі ігри в піратів чи в війну. Психолог Джерард Джонс у своїй книзі «Вбиваючи монстрів» пише, що певний рівень жорстокості необхідний для здорового розвитку психіки дитини. А в книзі «Mortal Combat» (що своєю назвою відсилає до популярної «жорстокої» гри) автори Маркей та Фрегюсон наводять цікаву думку, що більше за все ігри звинувачують в шкідливому впливі саме люди старшого і похилого віку, котрі мають досить поверхневе знання відеоігор та гравців.

Що вирізняє жорстокість комп’ютерних ігор – так це те, наскільки реалістично вона може виглядати, наскільки детально показано вбивство та передані відчуття від нього. Багато людей хвилюється, що гравець з часом перестане бачити різницю між ігровим світом та реальним. Але, насправді, це не так. Коли людина грає, вона зосереджена на досягненні ігрових цілей, а не на ненависті. Люди пам’ятають, що гра – це гра. Якщо людина не схильна до насилля сама по собі, то жодна комп’ютерна жорстокість не «зламає» її.

Щодо другого популярного страху – залежності – теж все не так прямолінійно, як здається багатьом людям. Залежність від ігор справді зустрічається, але її причини сильно різноманітні. Інколи це сама гра виявляється занадто актуальною для нього; а інколи реальні життєві проблеми штовхають не пошук альтернативної реальності. Не варто звинувачувати ігри в тому, що вони занадто хороші. Але і геймдизайнерам варто пам’ятати на те, що їх гра не має порушувати нормальний порядок життя людини.

Розглянемо тепер, чим саме ігри можуть бути корисними.

Перш за все, люди часто приходять у ігри, щоб виплескувати свої емоції, контролювати та балансувати свій емоціональний стан. В активних іграх з боями можна випустити пар. У легких аркадах можна розвеселитися. У сюжетні ігри можна зануритися, на час забути про власні проблеми, і потім, коли люди повертається до них після гри, вони вже виглядають інакше. Якщо людина відчуває

себе безпорадною або невпевненою, гра може дати їй відчуття успіху. Інколи в іграх можна навіть знайти для себе відповіді чи підтримку. Наприклад, гра Gris в абстрактній формі проводить людину по всім етапам проживання втрати, візуалізуючи емоції та дозволяючи прожити їх. Проживання емоцій – це дуже важлива складова психологічного здоров'я.

Людині, однак, варто пильно слідкувати за тим, щоб гра не погіршувала її емоційний стан – наприклад, зачіпаючи душевні рани, на які зараз тиснути не варто. Але тут вже все в руках гравця, якщо гра не подобається – просто не варто в неї грати.

Не менш поширена користь від гри – це зв'язок між людьми, соціалізація. В сучасному світі вже не рідкі випадки, коли люди знаходили в онлайн-іграх собі нових хороших друзів (з якими продовжили спілкування в реальному житті) або навіть коханих. Крім нових знайомств, ігри також додають способів підтримувати контакт з старими друзями. Граючи з другом, ви створюєте нові спільні способи, нові історії, нові жарти. А ще – це гарний спосіб тренувати навички командної роботи та взаєморозуміння. Наприклад, в Overcooked може грати до чотирьох гравців, час на кожному рівні обмежений, а зробити потрібно дуже багато – це змушує людей швидко координувати свої зусилля та домовлятися так, щоб ніхто в кого не стояв на дорозі та все процеси були максимально оптимізовані.

Люди, які мають труднощі зі спілкуванням в звичайному житті, в іграх можуть почувати себе більш розкутими, сміливішими, гра сама буде підштовхувати їх до спілкування, дасть тему для розмови і створить комфортні умови – завдяки цьому всьому вони зможуть отримати базовий соціальний досвід і, можливо, після цього їм стане легше спілкуватися у реальному житті.

Ігри, в певній мірі, несуть користь і для здоров'я. Звісно, важко уявити, щоб людина стала фізично міцнішою, сидячи за комп'ютером (і дива справді не станеться). Але ігри завжди включають в себе високу розумову активність, а це теж

дуже важливо, особливо у літніх людей. Ігри також тренують мозок розглядати проблему з різних сторін, швидко шукати нестандартне рішення.

Та і навіть фізична активність у іграх вже не є якоюсь фантастикою – з виникненням шлемів віртуальної реальності. Одягаючи такий шлем, людина повністю занурюється у віртуальний світ. Тут немає клавіатури, мишки і геймпада. Все керування здійснюється за допомогою рухів рук, змінення положення голови. Такі ігри часто включають в себе необхідність присідати чи підстрибувати. Прикладом такої гри є Beat Saber, в якій гравцю потрібно, в такт музиці (яка буває дуже швидкою!) збивати куби, що в нього летять, та відхилятися від перешкод. Чим не фізичні вправи? А також тренування реакції, спритності, контролю над тілом. Єдине, про що варто пам’ятати в таких іграх – це щоб навколо було достатньо вільного простору. Також, є невеликий процент людей, які взагалі не можуть грати в VR. Але це досить нова технологія, яка поступово буде покращуватися, тому, можливо, з часом це обмеження зникне.

І, звісно, найважливіша у розрізі теми роботи користь – це можливість навчитися чомусь новому.

Насправді, можна назвати освітню систему грою – але це буде досить погано спроектована гра. В освітній системі не вистачає сюрпризів, що так люблять гравці. Часто не вистачає винагороди. Неправильно побудована крива інтересу.

Адже усім цікаво дізнаватися нове. Більшість людей захоче подорожувати світом, але не всім подобається вивчати географію; люди легко знаходять закономірності у іграх-головоломках, але не здатні освоїти математику; всі в дитинстві задавали тисячу питань про те, як влаштований світ, але більшість відчувають нудьгу на заняттях з фізики та хімії. Справа не в предметах, що вивчаються, а у формі подачі.

Розглянемо кілька напрямів, які є надзвичайно важливими у сфері освіти і з якими ігри справляються напрочуд добре.

- Подача фактів. Для людини взагалі не складно вивчити відомості про ігровий світ (про який вона початково нічого не знає!), запам’ятати правила, за яким він функціонує. Вивчення нових речей в іграх є різноманітним, захоплюючим і завжди супроводжується якоюсь винагородою. Тому людина охоче сприймає нову інформацію; на відміну від навчання в реальному житті, де часто перед нею постає завдання вивчити купу одноманітної інформації одноманітним чином.
- Вирішення проблем. Це невід’ємна частина ігрового процесу. Гравець постійно думає, як йому добитися своєї цілі в грі: як дістатися у конкретну точку, перемогти ворога, побудувати щось. При цьому він використовує знання та навички, отримані раніше у грі. Чи не це саме відбувається на шкільному занятті? Спочатку учень вивчає правила, а потім отримує завдання та шукає спосіб використати ці правила для його вирішення.
- Зв’язок, залежність у системі. Скільки б разів людина не прочитала про те, як все влаштовано та взаємопов’язано, вона не зможе відчути це як слід. Ігри, у свою чергу, надають можливість безпечно пограти з системою: покрутити її, спробувати провести над нею якісь дії та одразу побачити результати. Не менш важливою частиною ігрової симуляції є можливість програти, при чому під час програшу людина може побачити, що саме призвело до цього. Це теж є найважливішим внеском в загальне розуміння системи.
- Ігри постійно підштовхують людину до того, щоб подивитися на ситуацію по-новому, спробувати якесь нове рішення. У навчанні це може бути досить корисним, якщо на меті є знати багато способів вирішення проблеми, а не лише один.
- Ігри викликають цікавість, а цікавість – те запорука того, щоб досконало оволодіти певною навичкою, бути невтомним у її освоєнні.

Створюючи гру, геймдизайнер завжди вкладає в неї певний досвід, який він хоче передати гравцям – певні навички, емоції, ідеї, думки. Це велика відповідальність, адже це може вплинути на людей як позитивно, так і негативно. Ігрова форма справді є дуже ефективною для передачі інформації, при чому інформація сприймається майже завжди несвідомо, сама собою. Саме тому варто бути обережною з нею. Але у хороших руках – це чудовий інструмент, щоб змінити світ у кращу сторону, підвищити якість життя людей.

РОЗДІЛ 2

АНАЛІЗ ПРОГРАМНО-ТЕХНІЧНИХ РІШЕНЬ З ПОБУДОВИ КОМП'ЮТЕРНИХ ІГОР

2.1 Основні принципи геймдизайну

Геймдизайн – це процес створення форми та змісту ігрового процесу (геймплею).

Можна виділити такі основні галузі дизайну:

- Системний дизайн – створення основних правил ігрового світу, так званих систем.
- Контент-дизайн – створення персонажів та конкретних квестів – всього того, що наповнює гру.
- Дизайн рівнів (левел-дизайн) – побудова віртуального оточення, в якому буде діяти гравець, таким чином, щоб це оточення ненав'язливим чином вело гравця до мети.
- Дизайн інтерфейсу – створення інтерфейсу користувача таким, щоб він був зручним і не перетягував на себе увагу з самої гри.

В результаті своєї роботи геймдизайнери формують дизайн-документ, який детально описує всі складові гри, яку планується створити. На основі цього документу приймається рішення про фінансування проекту, склад команди і т.д. Мета команди розробників – максимально наблизитися до задуму геймдизайнера. Але інколи і геймдизайнер має поступатися ідеями, якщо на них не вистачає бюджету, часу або ж технології себе не виправдують.

Дати визначення поняттю гри – завдання практично неможливе, оскільки кожен дизайнер трактує це поняття по-своєму.

Одне з влучних, хоча і дуже абстрактних визначень: «Гра – це вільний рух у закритій системі»[3]. Справді, кожна гра являє собою закриту систему – зі своїми внутрішніми правилами, незалежними від правил реального світу; і гравець, перебуваючи у цій системі, може вільно взаємодіяти з її компонентами. Є лише одна деталь, якої не вистачає цьому визначенню: це мета. Гра не буде грою, якщо гравцю не запропоновано якоїсь конкретної мети, якої він має досягти, якогось конкретного напрямку, в якому він може йти – або декілька на вибір.

Це, фактично, саме те, що відрізняє гру від іграшки. Беручи як приклад дитячі розваги: іграшки в них є лише «інструментами», які може бути цікаво крутити в

[3] «Rules of Play», Katie Salen and Eric Zimmerman, глава 22.

руках, але сама гра починається, коли дитина вигадує для них світ зі своїми правилами.

В основі будь-якої гри лежить певний досвід, який геймдизайнер хотів через неї передати. Досвідом є внутрішнє відчуття якогось процесу. Наприклад, гра про польоти не має піднімати гравця у повітря, але може надати відчуття легкості і свободи в рамках свого ігрового світу, і передати досвід польоту таким чином.

Джесі Шел у книзі «Мистецтво геймдизайну: книга лінз» розглядає велику кількість «лінз», через які варто поглянути на дизайн гри, щоб зробити його найкращим, і «лінза суттєвого досвіду» стоїть найпершою. Автор пропонує дизайнеру поставити собі запитання: що за досвід я хочу передати гравцям? Що для цього досвіду є суттєвим? Як моя гра може передати цю суть?

Ще одна річ, яка робить гру грою – це фан (від англ. fun, веселощі). В ігри грати завжди весело, саме тому вони не набридають людям. Щоб гра залишалася веселою, вона має ставити питання та допомагати знаходити на них відповіді (задовольняти цікавість), з нею має бути приємно взаємодіяти (кожна гра має бути хорошою іграшкою), а також вона має містити для гравця сюрпризи – несподівані, але все ж таки обумовлені світом гри, події.

Якщо говорити менш абстрактно та більш технічно, гру можна поділити на такі складові: естетика, історія, технологія та механіка, та розмістити їх, як це показано на Рисунку.3, в залежності від того, які складові більш та менш видимі гравцю. Такий поділ створено геймдизайнером Джесі Шелом і називаний елементарною тетрадою (Рисунок 2.1).

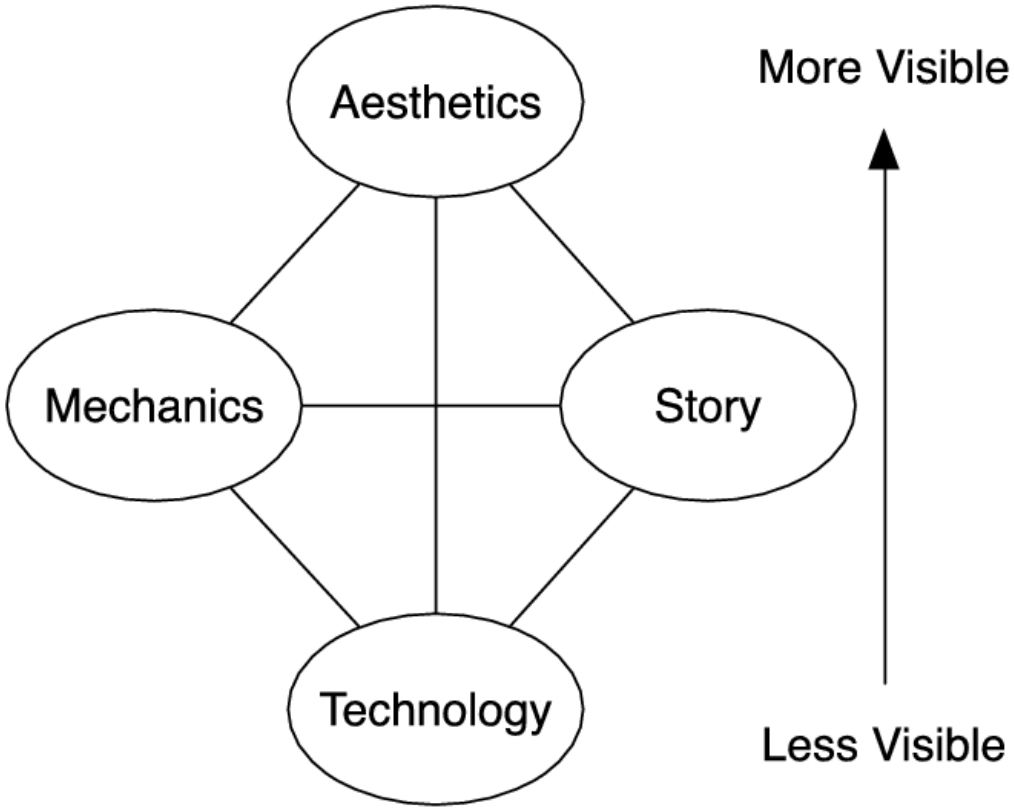

Рисунок 2.1 – Елементарна тетрада

У цій схемі:

- Естетика – все те, що гравець бачить та відчуває під час взаємодії з грою. Це графіка, музика, звуки. Якщо мова йде не про комп’ютерну, а про настільну гру, то до цього додаються ще тактильні відчуття від того, з якого матеріалу вона зроблена, її запах. Ця складова є найбільш помітною для користувача, вона формує початкове враження від гри, а також сприяє зануренню в атмосферу ігрового світу.
- Історія - це послідовність подій, які відбуваються в грі. Гра може бути як лінійною, коли гравець, умовно, йде по єдиній доріжці до фіналу, так і містити велику кількість роздоріжь, кожне з яких може розкрити більше контенту та мати інший розв’язок. Історія дуже тісно взаємодіє з естетикою та механікою: вони мають підсилювати її. Гравець помічає

історію після того, як занурюється в гру, він має сприймати її, тому вона є проміжною між «видимими» і «невидимими».

- Механіка – це правила на процеси гри; це те, що гравцю доводиться робити під час гри. Якщо в грі є мета, то механіки – це методи її досягнення. Механіки мають підсилювати історію гри. Фактично, механіки – це те, що передає гравцю відчуття досвіду, а історія – це те, що називає ці відчуття, дає їм контекст та пояснення. Механіка не є невидимою для гравця, проте більшість людей, граючи, про неї зовсім не задумується, і це є правильним: дії в грі мають бути природними та інтуїтивно зрозумілими.
- Технологія – «внутрішня» складова гри; те, що зв’язує все інше разом та змушує працювати. Зрозуміло, що технологія повністю прихована від користувача, але в жодному разі вона не є менш важливою за решту елементів. Правильний вибір технології дасть можливість максимально розкрити інші складові гри.

У грі має бути тема та ідея. Тема – це те, про що гра; ідея – це основна думка, яку розробники гри хочуть донести до гравця. Ці поняття перетинаються з основами сценарної майстерності, які важливі для створення книг та фільмів; оскільки і там, і там – є свій світ, своя історія та самовираження автора (чи колективу авторів).

Також гра має містити якусь проблему, яку гравець буде намагатися вирішити, і, якщо присутній сюжет, то конфлікт.

При розробці гри дуже важливо собі уявляти, які люди будуть грати в неї – свою цільову аудиторію. І не просто уявляти, а знати настільки досконало, настільки це можливо.

Основні характеристики цільової аудиторії – це вік та стать. Діти молодшого віку, школярі, підлітки, молоді люди, люди середнього та похилого віку; жінки та чоловіки. Хоча всі люди унікальні та не можна так просто класифікувати їх, все ж

у представників кожної з цих груп є дещо спільне в інтересах та поглядах на життя. Наприклад, малим дітям не потрібно давати складні головоломки – неможливість утримати на них увагу та самостійно знайти рішення засмутить їх; тоді як вже в віці 10-13 років дитина здатна приділяти своєму захопленню багато уваги та серйозно думати про нього.

Якщо говорити про відмінності у вподобаннях, які властиві певній статі, то чоловікам подобається влада та сила, конкуренція, просторові головоломки та методи проб і помилок. Тоді як жінкам, частіше – емоції, реальний світ, піклування про когось, діалоги, навчання через наявність прикладу. Варто відмітити, що такий розподіл є лише приблизним, але це краще, ніж нічого, коли є необхідність побудувати портрет цільової аудиторії.

Також для цього портрету варто дослідити, які ігри полюбляють потенційні гравці розроблюваної гри, що саме їм більш за все в цих іграх подобається; чим вони можуть займатися в житті. Чим більше інформації – тим більшу кількість елементів, потенційно цікавих цільовій аудиторії, можна буде додати в гру. Але, варто відмітити, що разом з цим гра може ставати і більш «вузьконаправленою». Тому дуже важливо балансувати.

Існують чотири властивості людського розуму, які в принципі роблять ігри можливими. Варто розуміти та враховувати їх:

- Моделювання. Наш розум не працює з реальністю, як вона є, лише з її моделями. Це спосіб зробити реальність доступною та зрозумілою. Коли ми працюємо за комп’ютером, ми не задумуємося про послідовність електричних імпульсів, які роблять це можливим. Ми просто сприймаємо комп’ютер як обгортку для цього, а кнопки клавіатури – як посередників. Так само і гра виділяє лише найважливіше для поточного моменту, і дозволяє сконцентруватися на ньому. Тому, часто, ігри настільки приємні для нашого мозку: всі

складності просто відсічені. Дана властивість може бути вкрай важливою для побудови будь-яких ігор для навчання, адже є можливість зосередитися конкретно на темі, що вивчається, не задумуючись про все інше.

- Зосередження. Це властивість нашого мозку фокусувати свою увагу на тому конкретному елементі, який нас зараз цікавить. Точно так же, як ми можемо зосередитися на голосі друга, навіть коли знаходимося з ним с шумному місці, де купа людей, що розмовляють. Все решта стане лише білим шумом. Оскільки нам цікаве саме те, що скаже друг. Так само завдання гри – бути достатньо цікавою, щоб гравець зосередився на ній і перестав звертати увагу на весь оточуючий світ. Тоді гравець переходить у стан «потоку»: коли він, не відволікаючись, не відчуваючи плину часу, продовжує вирішувати поставлені грою завдання. Цей стан супроводжується високим рівнем задоволення людини, тому так важливо намагатися створити його в процесу гри. Для цього є декілька основних правил: має бути чітка ціль – гравець має постійно розуміти, куди йому рухатися, що йому спробувати далі; відсутність відволікаючих чинників; швидка відповідь на дії гравця; правильна кількість напруги у грі. Відносно останнього пункту варто одразу уточнити: якщо напруги занадто мало, то гравець відчуває нудьгу; якщо її занадто багато, він втомлюється, відчуває тривогу, і це виводить його зі стану задоволення. Напругу можна також назвати відношенням навичок гравця до складності поставлених завдань. Завдання мають бути лише трохи складнішими, ніж наявні навички: тоді гравець зможе відчувати якийсь прогрес, розвиток, і при цьому не застрягати на вирішенні проблеми на занадто довгий час. Описане явище називають станом потоку, його наглядно ілюструє Рисунок 2.2.

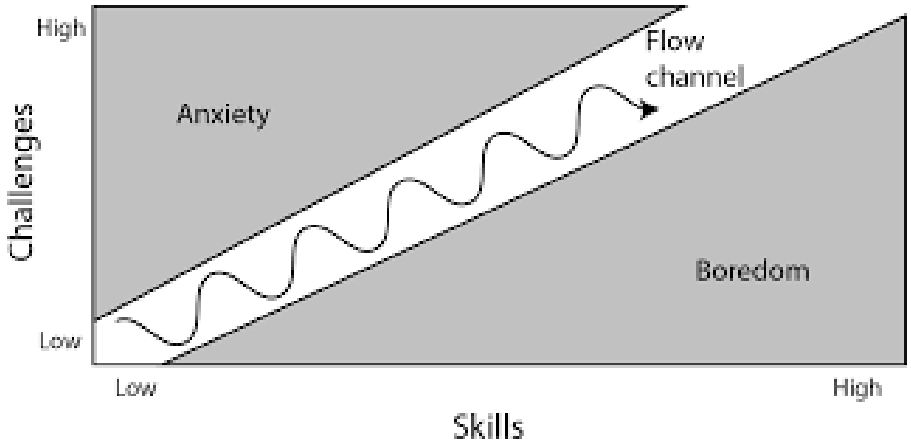

Рисунок 2.2 – Канал потоку

- Співчуття. Мабуть, це базова річ, яка відрізняє людину від тварини – можливість відчувати чужий біль та чужу радість як свої власні. Людині для цього не потрібно навіть особливо замислюватися. І це базова властивість, завдяки котрій художні твори знаходять відклик в людській душі. Читаючи про вигаданих героїв, або дивлячись на них, або граючи за них, ми відчуваємо їх емоції так, наче вони наші власні. Людей завжди цікавила можливість побачити світ чужими очима, і завдяки співчуттю до хай навіть нереальних істот це стає можливим.
- Уява. Це надзвичайно важлива властивість, котра, в геймдизайні, працює в двох напрямках. По-перше, уява дозволяє заповнити будь-які «пробіли» в світі гри, і таким чином розробникам не потрібно висвітлювати абсолютно кожну дрібницю (якби вони це робили, то гра була б інформаційно перенасичена!). По-друге, завдяки уяві гравець може уявити шлях до рішення поставленою грою проблеми, що, зрештою, і складає суть гри. Майстерність геймдизайнера починається там, де він розуміє, які деталі конче необхідно показати для цілісності світу, а які можна залишити на розсуд гравця; які шляхи до вирішення проблеми показати більш явно, а які залишити на роздуми.

Ще один цікавий принцип, який лежить в основі того, щоб захопити увагу гравця – це крива інтересу[4]. Вона зображена на Рисунку 2.3. Розглянемо її детальніше.

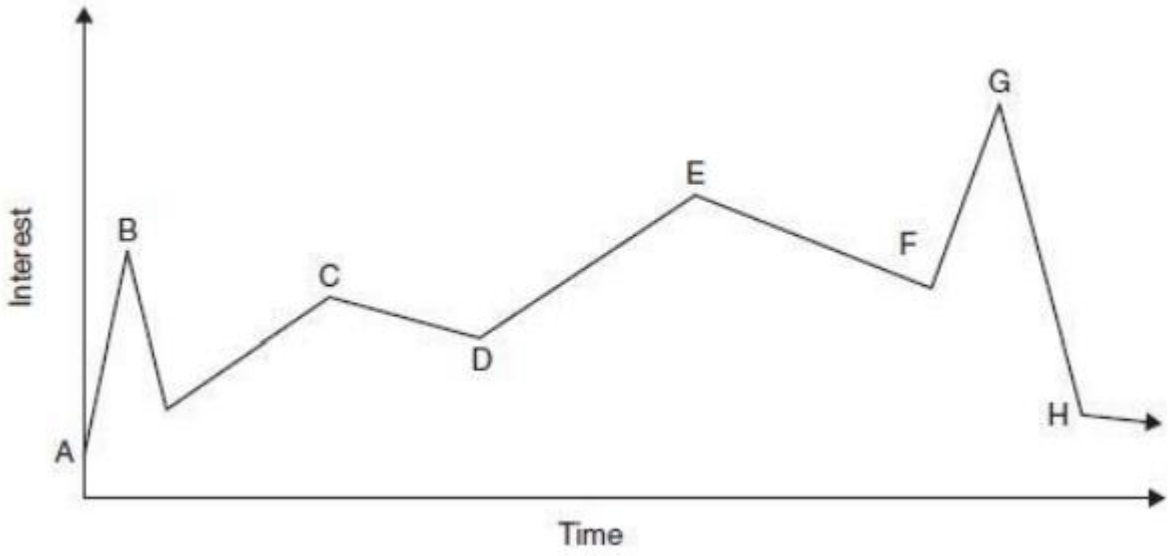

Рисунок 2.3. – Крива інтересу

Ця лінія представляє собою напруженість, цікавість подій, що відбуваються в грі, залежно від проведеного в грі часу.

Починається все з точки А. На цьому рівні гравець тільки доєднався до гри. Він зацікавлений зовсім трохи (адже він ще нічого не знає про світ гри), але достатньо, щоб почати грати.

Досить швидко ми переміщуємося до точки В. Це вступ в сюжет і ігровий процес, який має бути достатньо яскравим, щоб захопити увагу гравця і переконати його, що попереду ще багато цікавого.

[4] Jesse Schell. The Art of Game Design: A book of lenses. Глава 14.

Далі у точках від C до F ми бачимо, що напруженість та інтерес то зростає, то падає, але загалом – підвищується. Спади дають гравцю можливість відпочити та зробити перерву, тоді як зростання не дозволяють закинути гру зовсім.

Нарешті, у точці G інтерес сягає максимального рівню – це кульмінація, як і в літературному творі. Далі за нею слідує падіння інтенсивності – логічне завершення.

Крива інтересу є фрактальною. Якщо наша гра складається з окремих квестів, то ці квести розташовані відповідно до кривої інтересу; але всередині завдання кожного окремого квесту розташовані теж відповідно до неї. І навіть всередині завдання можна вмістити невеличку криву інтересу, яка зробить його виконання захопливим, а не нудним.

Геймдизайн є дуже неоднозначною галуззю, яка у багато чому суміжна з мистецтвом. А у мистецтві, як відомо, правил немає. Так само і тут: немає конкретної інструкції, як зробити хорошу гру, яка захопить увагу людей. Але, все ж, є основні принципи – які ми і розглянули – що працюють в більшості хороших ігор. Кожен геймдизайнер сам вирішує, в якій мірі ним слідувати, а в якій – робити на власний розсуд.

2.2 Огляд і можливості ігрового рушія Unity

Серед ігрових рушіїв Unity займає далеко не останнє місце. Його використовують і великі розробники, і (набагато частіше) невеликі незалежні студії (інді-студії).

Unity – це середовище для розробки комп'ютерних ігор, в якій об'єднані різні програмні засоби, що використовуються при створенні програмного забезпечення. Він поєднує в собі текстовий редактор, компілятор, відладчик і т.д. Можливості Unity досить широкі, при чому включають інструменти для розширення самими розробниками. При цьому, завдяки зручності використання, Unity робить створення ігор максимально простим і комфортним, завдяки чому його часто обирають програмісти-початківці для легкого вступу в світ програмування ігор, або ж студії для створення невеликих проектів. Мультиплатформенність двигуна дозволяє охопити якнайбільшу кількість ігрових платформ та операційних систем: і телефони (iOS, Android), і ПК, і навіть консолі (PlayStation, xBox – але лише якщо у вас є акаунт розробника).

Простота використання Unity у багато чому пов’язана з тим, що в ньому використовується компонентно-орієнтований підхід. Згідно цьому підходу, розробник створює об’єкти, що представляють конкретні об’єкти в грі, і до них додає різні компоненти: наприклад, об’єкт головного героя і його візуальне відображення.

Друга перевага рушія – це наявність величезної бібліотеки асетів та плагінів, за рахунок чого можна значно прискорити процес розробки гри. Більшість асетів знаходиться на Unity Asset Store, звідки їх можна імпортувати. Також можна створити свій плагін в Unity та експортувати його в інші проекти або в магазин для продажу. За допомогою асетів можна додавати в гру цілі заготовки – рівні, ворогів, патерни поведінки штучного інтелекту тощо. Таким чином, розробляти ігри на Unity можна взагалі не пишучи код – існують асети для візуального скриптінку. Але все ж таки для великих складних проектів код краще все-таки писати.

Unity підтримує всі основні графічні технології. Він працює з DirectX та OpenGL, з усіма сучасними ефектами, включаючи новітню технологію трасування

променів у реальному часі. В рушій вбудована фізика твердих тіл та ragdoll, колізії, система Level of Detail, можливість працювати з анімаціями.

І одна з найважливіших переваг рушія в тому, що він доступний безплатно (хоч і з деякими обмеженнями, при чому вони стосуються скоріше публікації гри, ніж функціоналу при самій розробці).

Більшість того, що не доступне в Unity за замовчуванням, можна знайти в Unity Asset Store. Це величезний магазин інструментів та асетів, які створюються сторонніми розробниками: будь-хто може створити такий асет. Основні категорії:

- 3D, 2D візуальні асети – спрайти, меші, анімації;
- Add-Ons – розширення для гри, наприклад, інструменти розпізнавання голосу;
- Аудіо асети;
- Essentials та Templates - невелика колекція пакетів, які містять в собі все, щоб створити простеньку гру на певну тематику;
- Інструменти для полегшення роботи в самому рушії;
- Візуальні ефекти

Деякі з асетів є платними.

На Unity зроблені такі популярні ігри як Rust, Ori and the Blind Forest, Firewatch, Inside, Subnautica, Superhot, Beat Saber.

Тепер розглянемо основи того, як будується гра на Unity.

Гра складається зі сцен. Сцена – це найкрупніша структура одиниця вашої гри (з технічної точки зору). Часто це – окремо взятий рівень. Сцена містить об’єкти, які знадобляться нам в межах цього рівню. Матимемо на увазі, що ці об’єкти знаходяться в оперативній пам’яті, коли сцена активна, тому правильна робота зі сценами надзвичайно важлива для того, щоб створити оптимізовану програму. Існують більш складні організації сцен, коли одночасно завантажені дві, три, а то і більше (дуже рідко). В таких випадках рівні будуються цією комбінацією сцен.

Наприклад, у нас може бути одна сцена, що містить користувацький інтерфейс, який постійно присутній на екрані, а на інших сценах буде сам рівень, по якому пересувається персонаж. В такому разі сцена з інтерфейсом буде постійно завантажена, а сцени з рівнями будуть то завантажуватися, то вивантажуватися. Це також допоможе уникнути «блимання» екрану при переході на новий рівень. Таким чином можна навіть побудувати ілюзію «безшовності»: тобто гравець не буде бачити екран завантаження взагалі.

Інше базове поняття Unity – це GameObject, ігровий об’єкт. Це також один з базових елементів, але меншого порядку, ніж сцена. Об’єктами бувають і «фізичні» сутності в грі, як, наприклад, будь-які предмети, які гравець бачить на екрані – дерево, стіл, будинок; так і абстрактні, що існують лише «за кулісами» гри та забезпечують функціонування та взаємодії всіх її складових: менеджер звуку, різноманітні контролери.

Всі об’єкти складаються з компонентів. Кожен компонент має своє призначення та функції. Навішуючи на об’єкт компоненти, ми додаємо йому властивості.

Базовий компонент, який є у будь-якого об’єкта – це компонент Transform, що відображає позицію і поворот об’єкта на сцені (у ігрових координатах). Об’єкти можуть мати дочірні об’єкти. Позиція та поворот дочірнього об’єкта завжди враховує ці значення батьківського. Звідси випливають також поняття світових та локальних координат: в першому випадку йдеться про результуючу величину відносно центру світу, а в другому – відносно батьківського об’єкту.

На Рисунку 2.4 можна побачити об’єкт камери, а на Рисунку 2.5 – приклад ієрархії об’єктів в Unity.

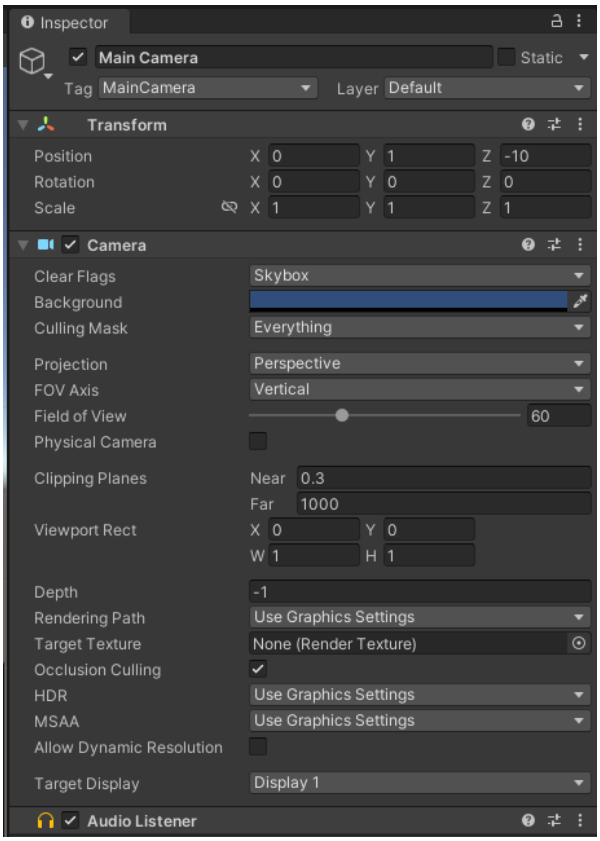

Рисунок 2.4 – Об’єкт гральна камера

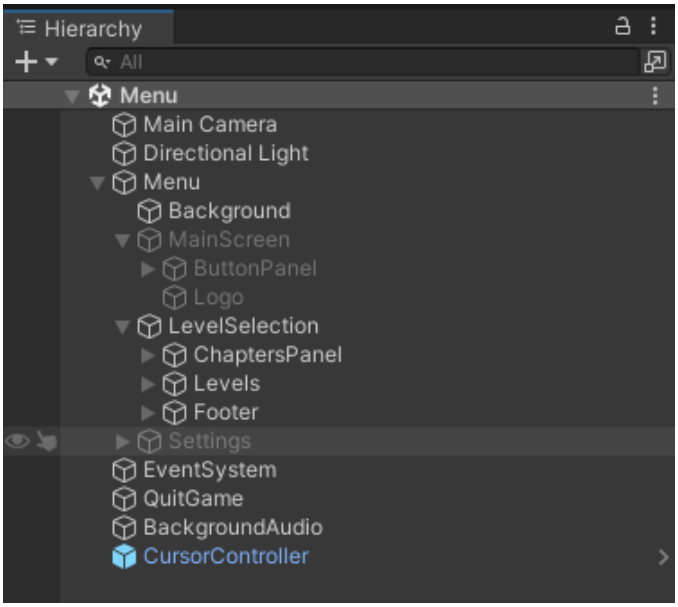

Рисунок 2.5 – Ієрархія об’єктів на сцені

Окрім компоненту Transform, який є обов'язковим, можна навести такі приклади компонентів, які часто використовуються в іграх:

- Rigidbody – компонент імітує фізику твердого тіла; з таким компонентом об'єкт матиме масу, тертя і т.д. Його взаємодія з іншими фізичними об'єктами буде обраховуватися додатково і Unity буде прагнути імітувати її максимально реалістично. Проте, варто зазначити, що це достатньо ресурсовимогливі операції, тому «з коробки» вони є дещо спрощеними; до того ж, з коду складно на них якось впливати. Якщо розробляється гра, в якій потрібна дійсно реалістична фізична взаємодія, то розробники зазвичай пишуть свої власні реалізації для цього, не задіюючи фізику Unity.
- Collider (box, cylinder, sphere, варіації для 2D), колайдер – компонент, який додає невидимий меш об'єкту, який реагує на зіткнення з іншими колайдерами. Можуть бути колайдери для чисто фізичної взаємодії (щоб об'єкти не проходили один через одного), а можуть бути тригери – такі колайдери будуть проходити один через одного, але при цьому надсилатимуть повідомлення про це в скрипт-обробник.
- Sprite та Image – схожі компоненти, які мають різне призначення. Перший – для розміщення 2D картинок на самій сцені. Другий – лише в межах UI, під батьківським компонентом Canvas, що контролює весь UI.
- Audio Listener – в базовому вигляді супроводжує компонент Camera. Цей компонент забезпечує, що музика, яку включають компоненти AudioSource, буде почута. На сцені може бути лише один AudioListener.

Unity налічує десятки вбудованих компонентів різних призначень.

Кожен компонент має скрипт, що власне і реалізовує всі функції компоненту. Всі такі скрипти є нащадками класу Component. Можна створювати і власні

компоненти, але вони, скоріше за все, будуть нащадками MonoBehaviour (який є нащадком Component та має додаткові функції). У MonoBehaviour є декілька корисних функцій, таких як: Awake, Start (викликаються на самому початку життєвого циклу компонента), Update, FixedUpdate, LateUpdate (викликаються кожен кадр), OnDisable, OnDestroy (викликаються, коли об’єкт деактивовують або знищують). Важлива властивість об’єктів MonoBehaviour: можливість запускати так звані «корутини». Це функції, які імітують асинхронну роботу. Вони не є по-справжньому асинхронними, але можна призупинити виконання функції в певному місці та повернутися до її виконання пізніше, наприклад, в наступному кадрі, чи через декілька секунд. «Всередині» корутина є, насправді, просто об’єктом IEnumerator.

Ще одна важлива складова гри – це ресурси. Це, фактично, усе, окрім ігрової логіки. До ресурсів, або асетів, них відносяться спрайти (картинки), аудіофайли, меші (3D моделі), відеофайли та ін.

Отже, таким чином і будується гра: створюється сцена, на якій розміщені об’єкти, які складаються з компонентів. Далі організовується перехід на інші сцени і вивантаження поточної, якщо це необхідно. Залежно від розміру проекту, дана схема може бути і значно складнішою, але в базовому вигляді все зводиться до цього.

2.3 Технології та практики проектування комп’ютерних ігор

Хоча в Unity теоретично можливо створити гру засобами visual-scripting, не пишучи коду (і навіть є успішні приклади таких ігор – популярний платформер Hollow Knight майже повністю зроблений так, хоча розробники все ж і написали деякі скрипти), все ж таки це не дуже популярний підхід, і написання коду в Unity не таке вже й складне та значно розширює можливості.

Саме тому загальні підходи ООП цілком застосовні для Unity, як і відомі шаблони, або патерни, програмування (і, на додачу – декілька таких, які популярні лише для програмування ігор). Розглянемо основні.

Під час програмування часто використовуються патерни, оскільки це полегшує вирішення часто виникаючих задач певного типу, а також, якщо люди працюють в команді (а це відбувається дуже часто), то іншій людині буде легше прочитати чужий код, якщо вона впізнає знайомий патерн.

ECS (Entity Component System) – архітектурний шаблон, за яким будується більшість ігор. Це дещо глобальне, що не обмежується парою скриптів. Це скоріше підхід до архітектури всього проекту. Компоненти цього шаблону: система – основний контролер певних Entity, який здійснює управління ними, своєчасне оповіщення і т.д.; Entity – структурна одиниця системи, що зберігає дані про свій стан і керує викликами конкретних компонентів; Component – низкорівневий скрипт, що виконує конкретні дії, пов’язані з Entity, до якої він прикріплений. Unity побудований на цій архітектурі, оскільки при створенні ігор на цьому рушії неможливо обійтися без об’єктів (сутностей) та компонентів.

MVC (Model-View-Controller) – теж дещо більше, ніж просто патерн; це схема розділення та взаємодії даних програми та її логіку. Відповідно до неї, все поділяється на Модель – дані, Вид – візуальне відображення, та Контролер – те, що знаходиться посередині та допомагає двом іншим компонентам спілкуватися між собою (Рисунок 2.6)

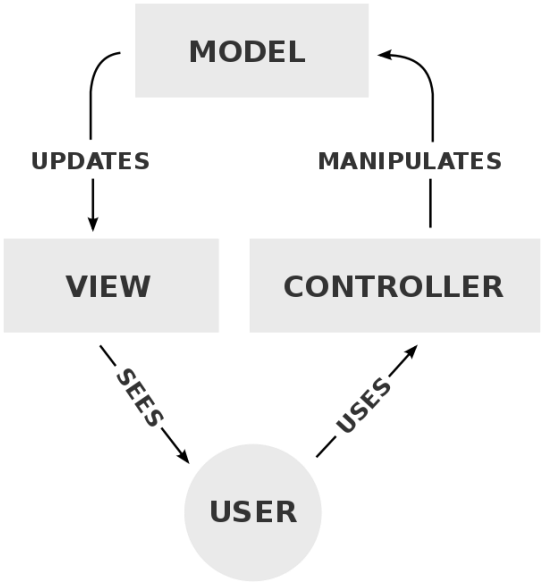

Рисунок 2.6 – Спрощена схема патерну MVC

Решта патернів розв’язують більш локальні завдання.

Одні з найпоширеніших патернів у розробці ігор – це Singleton та Object Pool. Обидва вони використовувалися при виконанні завдання.

Singleton — це породжуючий патерн проектування, який гарантує, що клас має лише один екземпляр, і надає до нього глобальну точку доступу (Рисунок 2.7).

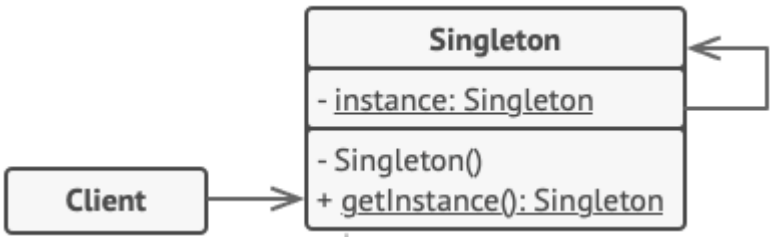

Рисунок 2.7 – Схема патерну Singleton

В Unity зазвичай існує велика кількість окремих об’єктів. Деякі з них представляють важливі загальні системи, доступ до котрих потрібен багатьом іншим. Передавати посилання на них, якось їх шукати може бути дуже складним та

не оптимізованим завданням. Саме тому Singleton є зручним та простим рішенням.

Однак він має в собі також і небезпеку. По-перше, зловживання ним призводить до порушення принципу єдиної відповідальності. По-друге, неуважність може призвести до того, що сам Unity-об’єкт видалиться, а статичне посилання на екземпляр залишиться, цим самим заважаючи очистити пам’ять та створюючи подальші проблеми, якщо буде створено новий екземпляр. Через такі небезпеки та невміле використовування Singleton часто відносять до антипаттернів. Патерн об’єктний пул – породжуючий шаблон проектування, згідно з яким існує набір ініціалізованих і готових до використання об'єктів. Коли системі потрібен об'єкт, він створюється, а береться з пула. Коли об'єкт більше не потрібен, він не знищується, а повертається в пул (Рисунок 2.8)

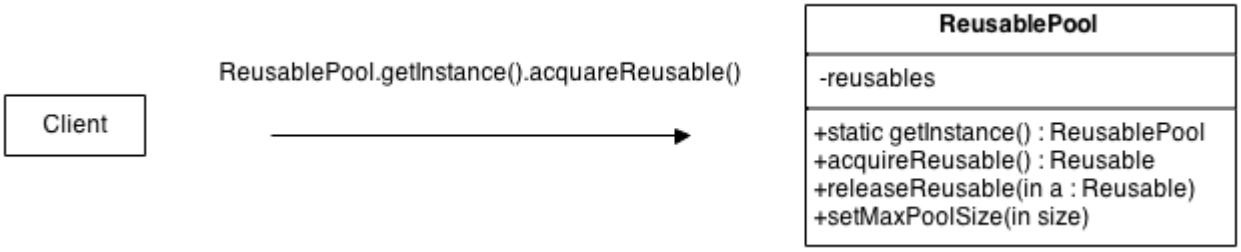

Рисунок 2.8 – Схема патерну Object Pool

Основна причина широкого використання такого патерну в Unity – це турбота про оптимізацію пам’яті. В іграх часто трапляється така ситуація, що створюється багато ідентичних об’єктів (найпростіший приклад – набої в зброї). Операція створення та видалення об’єкту в Unity є значно дорожчою, чим його деактивація. Тому набагато краще просто приховувати об’єкти, які наразі не потрібні, і потім діставати їх та переналаштовувати.

В багатьох випадках патерн Object Pool поєднується з патерном Singleton, оскільки скоріше за всі в програмі буде лише один об’єкт, який займається створенням об’єктів кожного типу.

Патерн Observer теж вирішує багато проблем в іграх. Він підходить, коли потрібно оповістити багато об’єктів про зміну стану одного об’єкту. Об’єкти, що хочуть отримати оповіщення, «підписуються» на подію (Рисунок 2.9).

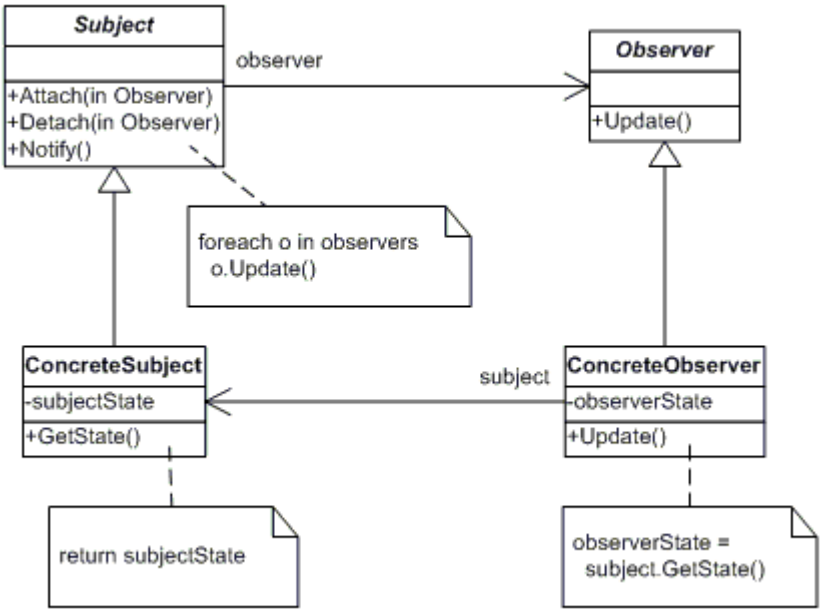

Рисунок 2.9 – Схема патерну Observer

У програмуванні ігор часто саме цей патерн мають на увазі, коли говорять про Event System.

Strategy – патерн, що вирішує проблему зміни поведінки класів залежно від вхідних даних. Проста ситуація: гра має велику кількість квестів, котрі мають між собою спільні риси, особливо початок і кінець, але сам хід квесту відрізняється. Дуже зручно підмінювати в основному класу реалізації поведінки залежно від поточного квесту (Рисунок 2.10).

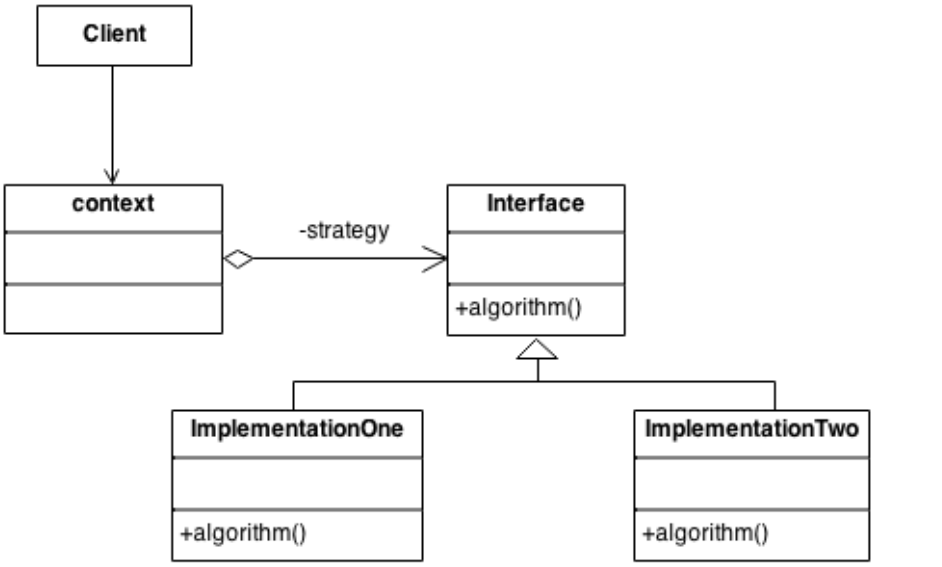

Рисунок 2.10 – Схема патерну Strategy

Патернів програмування існує велика кількість. Важливо вивчати їх, тому що вони створені для вирішення великої кількості загальнопоширених задач: не знаючи патернів, можна раз за разом «винаходити велосипед». Також важливо розглядати патерни та задачі, що підлягають вирішенню, з різних боків, щоб зрозуміти, що у даному випадку буде найбільш доречним. Зловживання патернами, їх використання у непідходящому місці так же погано, як і не використовувати їх зовсім.

РОЗДІЛ 3

ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ, ВПРОВАДЖЕННЯ КОМП'ЮТЕРНОЇ ГРИ ДЛЯ НАВЧАННЯ

3.1 Постановка задачі щодо побудови комп'ютерної гри

Темою роботи є комп'ютерна гра для вивчення точних наук.

Цільовою аудиторією такої гри є здобувачі освіти середньої та старшої школи.

В якості точної науки, що буде висвітлена через гру, обрана фізика. В грі будуть відображені теми, що виносяться в програму Зовнішнього Незалежного Оцінювання (ЗНО). Гра має на меті дати гравцю інтуїтивне відчуття кожної теми, розуміння того, як працюють фізичні закони та де їх можна застосувати.

Для глибшого занурення в ігровий світ (і, відповідно, фокусу на предметі, що вивчається), в грі присутній фентезійний сюжет. Короткий опис є таким: одного ранку головний герой, чотирнадцятирічний Нейдан, прокидається і розуміє, що магія покинула світ; проте він знаходить інший спосіб творити дива – силою науки. Метафорично даний сюжет ілюструє шлях людства від моменту, коли віра в абсолютну силу давніх божеств почала падати, до розквіту науки. Вік головного героя обрано відповідно до віку цільової аудиторії – учнів середньої школи. Таким чином потенційним гравцям буде легше уявити себе на його місці, легше побачити світ його очима. Також варто зазначити, що не просто так обрана паралель між магією та наукою: в дитинстві всі мріяли про можливість творити дива, тож ідея гри в тому, що кожен здатен на це, якщо буде старанно вивчати закони природи.

У грі також присутня можливість пропустити сюжет, якщо, наприклад, гравець хоче перепройти рівень, фокусуючись лише на ігрових механіках.

Гра реалізована на ігровому рушії Unity та мові програмування C#. Це буде 2D гра. Основний жанр – головоломка.

Гравець має можливість обирати порядок рівнів і змінювати його в будь-який момент: один з варіантів – відповідно до програми ЗНО, інший – за розділами фізики.

В рамках даної роботи реалізовується демо-версія гри, в яку входить зав’язка сюжету та один рівень.

Зав’язка показана через інтро-відео.

Ігровий рівень присвячений фізичному закону про момент сили. Тема обрана відповідно до кривої інтересу: вона не є найпростішою (оскільки гра має починатися з чогось, що зачепить гравця, інакше йому одразу стане нудно), але і не є занадто складною для того, щоб з неї почати, і до того ж є підходящою в рамках сюжету: оскільки левітація предметів – одне з найпоширеніших «заклять», і саме фізичний закон про момент сили дозволяє підіймати предмети меншими зусиллями. Звісно, це не єдина можлива відповідь на те, з імітації якого закляття вчорашній чаклун міг би почати. Але ця відповідь цілком виконує функції, які від неї потрібні сюжету та дизайну, тому вона нічим не гірша за інші.

Після перегляду інтро-відео гравець бачить важіль та короткий туторіал, який демонструє, як він працює. Після цього гравцю запропоновано кілька завдань з урівноваження важелю. Складність завдань поступово зростає та підштовхує до глибшого розуміння механізму. В першому завданні необхідно повісити аналогічний вантаж з іншого боку важелю симетрично; в другому – зрозуміти, що легший вантаж треба повісити на більш довгому плечі; в третьому – розмістити вантажі різної ваги на плечах важелю так, щоб він опинився в рівновазі – це підштовхне до розуміння принципу додавання моментів сил.

Варто підкреслити, що до цього моменту жодних обчислень та формул показано не було. Ідея в тому, щоб гравець дійшов до інтуїтивного розуміння того, як приблизно працює ця система.

Зрештою, гравець має скласти формулу рівноваги важелю, використовуючи для цього надані йому змінні, які він може перетягувати по екрану. Також він має наявним важіль та вантажі до нього, на випадок, якщо він захоче ще пограти з системою.

Після завершення цього завдання гра привітає гравця з успішним проходженням рівнів та надасть пояснення всіх подій, що відбувалися на ньому: визначення та формули.

Таким чином, гравець починає з практичної моделі закону, і лише після ознайомлення з нею бачить математичне підґрунтя: завдяки цьому теорія одразу ж пов’язується з конкретним візуальним прикладом та емоціями й відчуттями, які гравець пережив, вирішуючи надані завдання.

Протягом гри присутня спокійна, але не занадто повільна музика на фоні, яка занурює гравця в зосереджену, але не напружену атмосферу, сприятливу для того, щоб в своєму темпі розмірковувати над задачею. Через меню налаштувань можна змінювати гучність звуку.

В грі також реалізована система збереження прогресу, завдяки котрій можна приділяти грі стільки часу, скільки буде зручно: завжди можна буде повернутися до неї пізніше.

Гра побудована таким чином, що в подальшому її, теоретично, буде легко масштабувати і додавати нові рівні.

3.2 Стек технологій реалізації комп'ютерної гри для навчання

Для реалізації проекту використовувався рушій Unity, декілька завантажених з Unity Asset Store пакеджів (графіка та звук), пакедж TextMeshPro для якісного відображення тексту на екрані, інші асети, знайдені у вільному доступі в Інтернеті. Використані пакеджі, разом з пакеджами Unity за замовчуванням, представлені на Рисунку 3.1. Тут відсутні пакеджі, які містять тільки ресурси, тому що в них немає логіки.

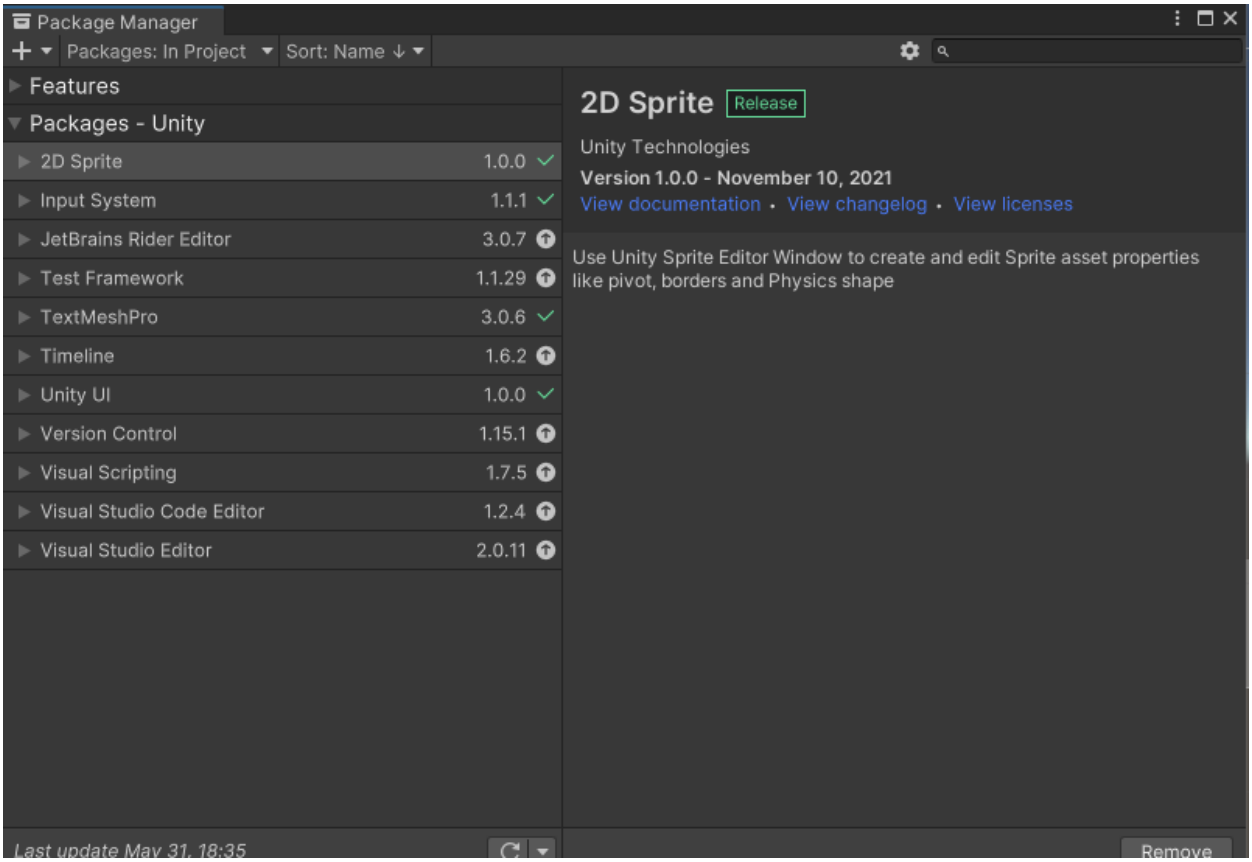

Рисунок 3.1 – Unity Package Manager

У грі присутня одна сцена для меню, далі вона змінюється сценою для кожного окремого рівня.

У коді задіяні патерни програмування, які ми вже розглядали в попередньому розділі.

Патерн Singleton реалізовують LevelLaucher, Quit, MenuWindowManager, LevelQuestManager, BackgroundMusicManager та інші. Патерн Object Pool реалізовує WeightPool, що займається створенням вантажів для розміщення на важелі і крючків для них, таким чином маючи навіть два пули.

Клас LevelQuestManager в певній мірі реалізовує паттерн Strategy. Він має поле для поточного квесту та здійснює певні дії над кожним квестом (всі класи квестів поєднані спільним інтерфейсом): такі як виконання функції Run на початку та функції QuestUpdate в кожному кадрі. LevelQuestManager також містить список усіх квестів в порядку їх виконання та здійснює перемикання на наступний квест, коли отримає відповідний сигнал. Схематично його роботу можна побачити на Рисунку 3.2.

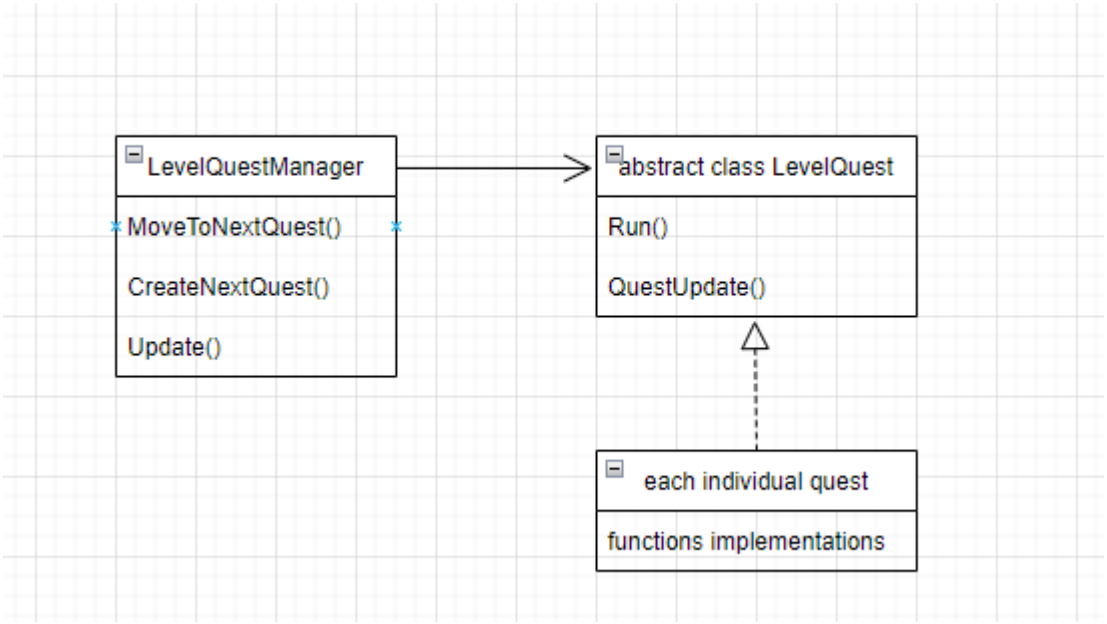

Рисунок 3.2 – Схема системи квестів

Також для проекту мені довелося відмовитися від використання вбудованої фізики Unity по декільком причинам. По-перше, вона погано реалізована для випадку 2D гри; по-друге, в ній досить складно добитися «круглих» величин, і

взагалі контролювати об’єкти – вони довго знаходяться в русі після поштовху, або, навпаки, в силу неочевидних для гравця причин не рухаються, коли мали б. Незважаючи на те, що це гра про фізичні закони, для цілей простоти та ясності варто трохи знехтувати точністю. Тому весь обрахунок руху прописаний в коді (Рисунок 3.3).

Рисунок 3.3 – Фрагмент коду важеля, який перераховує баланс моментів сил

Між об’єктами важеля, вантажів та крюків, на які вони вішаються, утворений взаємозв’язок, продемонстрований на наступній діаграмі (Рисунок 3.4).

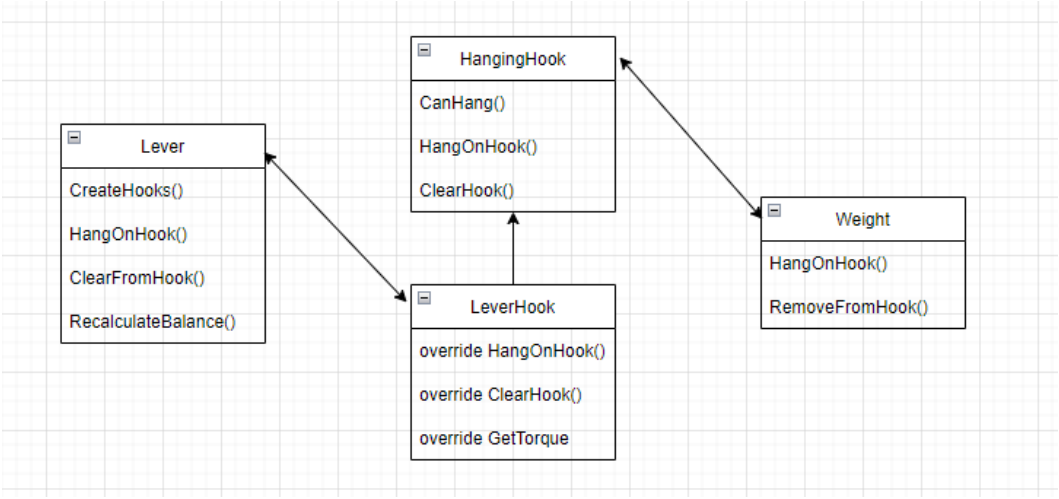

Рисунок 3.4 – Система важелю

Зрештою, архітектура програми передбачає розширення функціоналу, додавання нових рівнів по мірі розвитку сюжетної складової. Використання патернів програмування також робить її більш зрозумілою іншим людям, якщо такі почнуть роботу над проектом – адже розширення проекту майже завжди включає в себе і розширення команди, що над ним працює.

3.3 Перспективи розвитку комп'ютерної гри для навчання точним наукам

Реалізовано альфа-версію вертикального зрізу гри – це невелика частина гри, яка дає максимальне уявлення про кількість, склад механік, атмосферу, ідею гри, часто візуальну складову. Вертикальний зріз роблять для проекту, коли показують його перед потенційними інвесторами. Очевидно, що можна розвивати ідею далі в повноцінну гру, яка охопить та доступно пояснить основні теми з фізики на рівні здобувачів освіти середньої та старшої школи.

Хоча реалізовано лише один рівень, в меню присутні плейсхолдери для решти рівнів, що дає уявлення про задуманий об'єм гри. Подальше додавання рівнів не є складним.

Реалізовано вступний сюжетний ролик, базові механіки першої теми (момент сили), пояснення того, як вони працюють.

В подальшому до гри, окрім решти рівнів та логічного завершення сюжету, можна додати систему досягнень. Відомо, що такі системи мотивують багатьох користувачів добиватися більшого в іграх: ідеальне проходити рівні, уважніше вивчати їх. Це добре компонується з основною ціллю гри.

Світ гри побудований таким чином, що, якщо завершена версія даної гри буде успішна, можна робити продовження у вигляді наступних частин гри, присвяченим іншим наукам (математика, хімія). Завдяки цьому школярі отримають хорошу підтримку у опануванні наук, які, деколи, вважаються найскладнішими у шкільній програмі.

В перспективі, при дуже детальному розробленню гри вона може значно допомогти з підготовкою до шкільних іспитів, але на даному етапі вона скоріше являє собою підтримку школярам та інструмент підвищення інтересу до предмету.

ВИСНОВКИ

У межах цього бакалаврського дослідження було виявлено вплив комп’ютерних ігор на людей та суспільство, зокрема у сфері навчання.

Мета виконана в повному обсязі: створено прототип комп’ютерної гри, яка здійснює інтелектуалізацію навичок та компетенцій гравців у сфері точних наук.

Окрім того, виконано усі завдання дослідження:

- досліджено теоретичні основи побудови комп’ютерних ігор, а саме – засади та принципи, методики, за допомогою яких створюють навчальні ігри;
- висвітлено історію еволюції ігор;
- здійснено аналіз ігрової аудиторії;
- визначено негативні та позитивні тенденції впливу комп’ютерних ігор;
- здійснено аналіз програмно-технічних рішень із побудови навчальних ігор;
- описано основні принципи геймдизайну;
- здійснено огляд можливостей ігрового рушія Unity;
- здійснено вибір технологій і практик проектування комп’ютерних ігор;
- здійснено проектування, реалізацію, тестування, впровадження комп’ютерної гри для навчання;
- описано стек технологій реалізації комп’ютерної гри для навчання;
- висвітлено перспективи розвитку комп’ютерної гри для навчання точним наукам.

Під час виконання роботи було використано всі заявлені на початку методи дослідження:

- метод аналізу (пошук теоретичних даних, критичний їх розгляд, виявлення тих підходів, які знадобляться при втіленні проекту) – при дослідженні принципів геймдизайну;

- описовий метод (проектування алгоритмів роботи програми, ведення документації та формування звіту до виконаної роботи) – при плануванні реалізації практичної частини роботи;
- метод синтезу (підбір та комбінування популярних часткових програмних і теоретичних рішень з метою отримання системи компонентів, яка підходить до виконання поставленого завдання) – при розв’язуванні конкретних завдань, що виникали при створенні практичної частини.

У результаті розроблено прототип комп’ютерної гри, що має на меті допомогти здобувачам освіти старших класів школи у освоєнні точних наук, в даному конкретному випадку – фізики.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Джессі Шелл. Геймдізайн. Як створити гру, в яку гратимуть усі. Альпіна паблішер, 2019. 640 с.
2. Документація Microsoft .NET. URL: https://docs.microsoft.com/en-us/dotnet/ (дата звернення: 25 січня 2022 року)
3. Документація Unity3D. URL: https://docs.unity3d.com/ScriptReference (дата звернення: 20 березня 2022 року)
4. Unity Learn. URL: https://learn.unity.com/ (дата звернення: 23 березня 2022 року)
5. Статистичні дані щодо розмірів ігрової індустрії. URL: https://www.statista.com/topics/1680/gaming/ (дата звернення: 10.01.2022)
6. «Фізика» підручник для 7 класу закладів загальної середньої освіти / Бар'яхтар В. Г. та ін. Харків, видавництво «Ранок», 2020. 255 с.
7. Design Patterns: Elements of Reusable Object-Oriented Software 1st Edition / Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John. Addison-Wesley Professional, 1994. 416 с.
8. Eric Freeman. Head First Design Patterns (A Brain Friendly Guide). O'Really Media, 2004. 694 с.
9. Eric Lengyel. Foundations of Game Engine Development, Volume 1: Mathematics. Terathon Sofrware, 2016. 200 с.
10. Francesco Sapio. Unity UI Cookbook. Packt Publishing, 2015. 284 с.
11. Jeffrey Richter. CLR via C# (Developer Reference). Macrosoft Press, 2012. 892 с.
12. Hearts, Clubs, Diamonds, Spades: Players Who Suit MUDs. Richard Bartle (1996). URL: https://mud.co.uk/richard/hcds.htm (дата звернення: 20.04.2022)
13. Jared Halpern. Developing 2D Games with Unity: Independent Game Programming with C#. Apress, 2018. 405 с.

14. Joe Hocking. Unity in Action: Multiplatform Game Development in C# with Unity 5. Manning Publications, 2015. 352 c.
15. Joesph Hocking. Unity in Action, Second Edition: Multiplatform game development in C#. Manning Publications, 2018. 400 c.
16. Joris Dormans. Game Mechanics: Advanced Game Design. Addison-Wesley Professional, 2008. 464 c.
17. Joseph Albahari. C# 7.0 Pocket Reference. O'Reilly, 2017. 240 c.
18. Jeremy Gibson Bond. Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C#. Addison-Wesley, 2014. 944 c.
19. Jesse Schell. The Art of Game Design: A book of lenses. CRC Press, 2008. 520 c.
20. Katie Salen and Eric Zimmerman. Rules of Play - Game Design Fundamentals. The MIT Press Cambridge, 2004. 694 c.
21. Matt Smith. Unity 5.x Cookbook: More than 100 solutions to build amazing 2D and 3D games with Unity. Packt Publishing, 2015. 570 c.
22. Nicolas Lovell. The Pyramid of Game Design: Designing, Producing and Launching Service Games. A K Peters/CRC Press, 2019. 340 c.
23. Paris Buttfield–addis, Jonathon Manning, Tim Nugent. Unity Game Development Cookbook: Essentials for Every Game. O'Reilly, 2019. 400 c.
24. Patrick M. Markley. Moral Combat: Why the War on Violent Video Games Is Wrong. BenBella Books, 2017. 256 c.
25. Raph Koster. Theory of Fun. O'Reilly Media, 2013. 300 c.
26. Robert Nystrom. Game Programming Patterns, 2014. 354 c.
27. Robert Martin. Clean Code. Pearson, 2008, 464 c.
28. Steve Swink. Game Feel: A Game Designer's Guide to Virtual Sensation. Routlenge, 2008. 376 c.

29. Tracy Fullerton. Game Design Workshop: A Playcentric Approach to Creating Innovative Games, Third Edition. Routledge, 2014. 536 c.
30. Tynan Sylvester. Designing Games: A Guide to Engineering Experiences. O'Reilly Media, 2013. 416 c.

ДОДАТКИ

Додаток А

Source Code проекту

```
using System;
using UnityEngine;
public class HangingHook : MonoBehaviour
{
    [SerializeField, ReadOnly] private Rigidbody hangingObject;
    [SerializeField] protected CapsuleCollider collider;
    [SerializeField] protected Transform hangingPoint;
    [SerializeField, ReadOnly] protected float hangingMass = 0;

    public bool CanHang() => hangingMass == 0;
    private void Start()
    {
        ChangeLayer(Contants.IGNORE_RAYCAST_LAYER);
    }
    public virtual void HangOnHook(Rigidbody rb)
    {
        rb.transform.position = hangingPoint.position;
        rb.transform.parent = hangingPoint;
        hangingMass = rb.mass;
        // ChangeLayer(Contants.IGNORE_RAYCAST_LAYER);
    }
    public virtual void ClearHook()
    {
        hangingMass = 0;
        // ChangeLayer(Contants.DEFAULT_LAYER);
    }

    private void ChangeLayer(int layer)
    {
        gameObject.layer = layer;
        collider.gameObject.layer = layer;
    }
}

using System;
using System.Collections.Generic;
using UnityEngine;
public class Lever : MonoBehaviour
{
      [SerializeField] private LeverCenter center;
      [SerializeField] private int leftHooksCount;
      [SerializeField] private GameObject leftHookEndpoint;
      [SerializeField] private int rightHooksCount;
      [SerializeField] private GameObject rightHookEndpoint;
      [SerializeField] private GameObject hookParent;
      [SerializeField] private GameObject leverHookPrefab;
```

```
      [SerializeField, ReadOnly] private List<LeverHook> activeHooksLeft = new
List<LeverHook>();
      [SerializeField, ReadOnly] private List<LeverHook> activeHooksRight = new
List<LeverHook>();
      private Dictionary<int, LeverHook> hooks = new Dictionary<int,
LeverHook>();
      public Action OnBalanceChanged;
      public int currentBalance;

      private void Start()
      {
            CreateHooks();
      }
      private void CreateHooks()
      {
            float centerPos = center.positionX;
            float leftLen = Mathf.Abs(leftHookEndpoint.transform.position.x -
centerPos);
            float rightLen = Mathf.Abs(rightHookEndpoint.transform.position.x -
centerPos);
            float stepLeft = leftLen / leftHooksCount;
            float stepRight = rightLen / rightHooksCount;
            float step = stepLeft < stepRight ? stepLeft : stepRight;

            CreateHooksOnSide(leftHooksCount, step, -1);
            CreateHooksOnSide(rightHooksCount, step, 1);
      }
      private void CreateHooksOnSide(int count, float step, int sideSign)
      {
            for (int i = 1; i <= count; i++)
            {
                  float armLength = step * i;
                  float position = center.positionX + armLength * sideSign;
                  GameObject hookGO = Instantiate(leverHookPrefab,
hookParent.transform);
                  LeverHook leverHook = hookGO.GetComponent<LeverHook>();
                  leverHook.Setup(this, position, armLength, step * 0.75f, i);
                  hooks[i * sideSign] = leverHook;
            }
      }
      private void OnEnable()
      {
            RecalculateBalance();
      }
      public void AddHangingMass(LeverHook hook)
      {
            int hookSign = GetHookSign(hook.transform.position.x);
            if (hookSign == -1)
            {
                  activeHooksLeft.Add(hook);
            }
            else
            {
                  activeHooksRight.Add(hook);
            }
            RecalculateBalance();
```

```
	}
	public void RemoveHangingMass(LeverHook hook)
	{
		int hookSign = GetHookSign(hook.transform.position.x);
		bool removed = hookSign == -1 ? activeHooksLeft.Remove(hook) :
activeHooksRight.Remove(hook);
		if (!removed)
		{
			Debug.LogError("Was trying to remove hook that wasn't in the
list");
			return;
		}

		RecalculateBalance();
	}
	private List<LeverHook> GetSideList(float position) =>
		GetHookSign(position) == -1 ? activeHooksLeft : activeHooksRight;
	private int GetHookSign(float position) =>
		position < center.positionX ? -1 : 1;
	public void RecalculateBalance()
	{
		float leftTorque = GetSideTotalTorque(activeHooksLeft);
		float rightTorque = GetSideTotalTorque(activeHooksRight);
		int rotationDirection;
		float torque = 0;
		if (leftTorque > rightTorque)
		{
			rotationDirection = -1;
			torque = leftTorque;
		}
		else if (leftTorque < rightTorque)
		{
			rotationDirection = 1;
			torque = rightTorque;
		}
		else
		{
			rotationDirection = 0;
			torque = 0;
		}
		center.Rotate(rotationDirection, torque);
	}
	private float GetSideTotalTorque(List<LeverHook> side)
	{
		float torque = 0;
		foreach (LeverHook hook in side)
		{
			torque += hook.GetTorque();
		}
		return torque;
	}
	public LeverHook GetHookByIndex(int index) =>
		hooks[index];
	public void RemoveAllWeights()
	{
		Weight[] allWeights = GetComponentsInChildren<Weight>();
```

```
            foreach (Weight weight in allWeights)
            {
                  weight.RemoveFromHookInstant();
            }
      }
}

using System.Collections;
using UnityEngine;
public class LeverCenter : MonoBehaviour
{
    [SerializeField] private float speedMultiplier = 10f;
    [SerializeField] private float rotationLimit;
    [SerializeField] private AnimationCurve curve;
    [SerializeField, ReadOnly] private Quaternion targetRotation;

    public float positionX => transform.position.x;

    private bool isRotating = false;
    private float force;
    public void Rotate(int direction, float force)
    {
        this.force = force;
        float newRotation = 45 * direction * -1;
        newRotation = Mathf.Clamp(newRotation, -rotationLimit, rotationLimit);
        targetRotation = Quaternion.Euler(0, 0, newRotation); //new Vector3(0,
0, newRotation);
        if (!isRotating)
        {
            StartCoroutine(DoRotation());
        }
    }
    private IEnumerator DoRotation()
    {
        isRotating = true;
        while (!Mathf.Approximately(Mathf.Abs(transform.rotation.z -
targetRotation.z), 0))
        {
            // float speed = force * curve.Evaluate(Time.deltaTime) *
speedMultiplier;
            transform.rotation = Quaternion.RotateTowards(transform.rotation,
targetRotation, 10f * Time.deltaTime);
            yield return null;
        }
        transform.rotation = targetRotation;
        isRotating = false;
    }
}

using System;
using UnityEngine;
using TMPro;
[Serializable]
public class LeverHook : HangingHook
{
```

```
    [SerializeField, ReadOnly] private float armLength;
    [SerializeField] private TextMeshProUGUI label;
    private Lever lever;
    public void Setup(Lever lever, float position, float armLength, float
radius, int index)
    {
        this.lever = lever;
        this.armLength = armLength;
        transform.position = new Vector3(position, transform.position.y,
transform.position.z);
        collider.radius /= transform.lossyScale.x;
        collider.height = radius / transform.lossyScale.x;
        label.text = index.ToString();
    }
    public override void HangOnHook(Rigidbody rb)
    {
        base.HangOnHook(rb);
        lever.AddHangingMass(this);
    }
    public override void ClearHook()
    {
        base.ClearHook();
        lever.RemoveHangingMass(this);
    }
    public float GetTorque() =>
        hangingMass * armLength;
}

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;
public class Weight : MonoBehaviour, IMouseMovable
{
      [SerializeField] private AudioClip hangOnHookAudio;
      [SerializeField] private AudioClip removeFromHookAudio;
      [SerializeField] TextMeshProUGUI weightLabel;
      private bool isDragged = false;
      private List<HangingHook> hooksNearby = new List<HangingHook>();
      public HangingHook currentHook = null;
      private Rigidbody rb;
      public float Mass => rb.mass;
      private Coroutine flyToHookCoroutine;
      private void Awake()
      {
            rb = GetComponent<Rigidbody>();
      }
      private void Start()
      {
            weightLabel.text = rb.mass.ToString();
      }
      public void OnMouseDown()
      {
            if(flyToHookCoroutine != null)
        {
```

```
                StopCoroutine(flyToHookCoroutine);
        }
            RemoveFromHook();
            isDragged = true;
            rb.isKinematic = true;
            transform.rotation = Quaternion.identity;
            StartCoroutine(OnMouseHold());
      }
      private IEnumerator OnMouseHold()
      {
            while (isDragged)
            {
                  Vector2 mousePos = MouseInput.GetMousePoint();
                  transform.position = new Vector3(mousePos.x, mousePos.y,
transform.position.z);

                  yield return null;
            }
      }
      public void OnMouseUp()
      {
            isDragged = false;
            HangingHook hook = GetClosestDetectedHook();
            if (hook == null)
            {
                  StartFlyToEmptyHook();
                  //rb.isKinematic = false;
                  Debug.Log("No hook");
                  return;
            }
            Debug.Log("Put on hook " + hook.name);
            HangOnHook(hook);
      }
      private void OnTriggerEnter(Collider other)
      {
            HangingHook hook =
other.transform.parent?.GetComponent<HangingHook>();
            if (hook)
            {
                  hooksNearby.Add(hook);
            }
      }


      private void OnTriggerExit(Collider other)
      {
            HangingHook hook =
other.transform.parent?.GetComponent<HangingHook>();
            if (hook)
            {
                  hooksNearby.Remove(hook);
            }
      }
      void Update()
    {
            if(currentHook != null)
```

```
        {
                Vector3 directionUp =
transform.InverseTransformDirection(Vector3.up);
                transform.rotation *= Quaternion.FromToRotation(Vector3.up,
directionUp);
        }
    }
      public void HangOnHook(HangingHook hook)
      {
            hook.HangOnHook(rb);
            currentHook = hook;
            rb.isKinematic = true;
            PlayHangOnSound();
      }
      public void RemoveFromHook()
      {
            if (currentHook != null)
            {
                  transform.parent = null;
                  currentHook.ClearHook();
                  currentHook = null;
                  PlayRemoveFromHookSound();
            }
      }
      // private void OnDisable()
      // {
      //    RemoveFromHookInstant();
      // }

      public void RemoveFromHookInstant()
      {
            if (currentHook != null)
            {
                  RemoveFromHook();
            }
            HangOnHook(WeightPool.Instance.FindFreeActiveHook());
      }
      private void StartFlyToEmptyHook()
    {
            HangingHook hook = WeightPool.Instance.FindFreeActiveHook();
            if(hook != null)
        {
                  rb.isKinematic = true;
                  StartCoroutine(FlyToHook(hook));
        }
    }
      IEnumerator FlyToHook(HangingHook hook)
    {
            float distance = float.MaxValue;
            float step = 0.5f;
            while(distance > step)
        {
                  transform.Translate((hook.transform.position -
transform.position).normalized * step);
                  yield return new WaitForFixedUpdate();
```

```
                distance = Vector3.Distance(hook.transform.position,
transform.position);
            }
            HangOnHook(hook);
    }
      private HangingHook GetClosestDetectedHook()
      {
            if (hooksNearby.Count == 0)
                  return null;
            if (hooksNearby.Count == 1 && hooksNearby[0].CanHang())
                  return hooksNearby[0];
            hooksNearby = hooksNearby.FindAll(hook => hook != null);
            HangingHook closestHook = null;
            float closestDistance = Single.MaxValue;
            foreach (HangingHook hook in hooksNearby)
            {
                  float distance = Vector3.Distance(transform.position,
hook.transform.position);
                  if (distance < closestDistance && hook.CanHang())
                  {
                        closestHook = hook;
                        closestDistance = distance;
                  }
            }
            return closestHook;
      }
      public void MoveToHook(HangingHook hook)
      {
            RemoveFromHook();
            StartCoroutine(FlyToHook(hook));
      }
      private void PlayHangOnSound()
      {
            SoundEffectsManager.Instance?.PlayEffect(hangOnHookAudio);
      }

      private void PlayRemoveFromHookSound()
      {
            SoundEffectsManager.Instance?.PlayEffect(removeFromHookAudio);
      }
}

using System;
using System.Collections.Generic;
using UnityEngine;
public class WeightPool : MonoBehaviour
{
    public static WeightPool Instance;
    [SerializeField] private float poolWidth;
    [SerializeField] private GameObject hookPrefab;
    [SerializeField] private Transform hooksParent;
    [Serializable]
    public class WeightPoolEntry
    {
        public GameObject weightPrefab;
        public int requiredInstances;
```

```
    }
    public List<WeightPoolEntry> poolPlan;
    private Dictionary<GameObject, List<GameObject>> pool = new
Dictionary<GameObject, List<GameObject>>();
    private Dictionary<GameObject, List<GameObject>> activeObjects = new
Dictionary<GameObject, List<GameObject>>();
    private List<HangingHook> activeHooks = new List<HangingHook>();
    private List<HangingHook> hooksPool = new List<HangingHook>();
    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
    }
    private void Start()
    {
        ResetPool();
    }
    [ContextMenu("ResetPool")]
    public void ResetPool()
    {
        UnitePoolAndActiveObjects();
        ClearActiveHooks();
        foreach (WeightPoolEntry entry in poolPlan)
        {
            List<GameObject> activeList = new List<GameObject>();
            while (activeList.Count < entry.requiredInstances)
            {
                GameObject pooledObject = null;
                if (pool.ContainsKey(entry.weightPrefab) &&
pool[entry.weightPrefab].Count > 0)
                {
                    pooledObject = pool[entry.weightPrefab][0];
                    pool[entry.weightPrefab].RemoveAt(0);
                }
                else
                {
                    pooledObject = CreateWeight(entry.weightPrefab);
                }
                SetupWeight(pooledObject);
                activeList.Add(pooledObject);
            }
            activeObjects[entry.weightPrefab] = activeList;
        }

        DisableInactiveObjects();
        PlaceHooks();
        DisablePolledHooks();
    }
    private void UnitePoolAndActiveObjects()
    {
        foreach (KeyValuePair<GameObject,List<GameObject>> entry in
activeObjects)
        {
            if (pool.ContainsKey(entry.Key))
```

```
            {
                pool[entry.Key].AddRange(entry.Value);
            }
            else
            {
                pool[entry.Key] = entry.Value;
            }
        }
        activeObjects.Clear();
    }
    private GameObject CreateWeight(GameObject prefab)
    {
        return Instantiate(prefab);
    }
    private void SetupWeight(GameObject weightGO)
    {
        weightGO.SetActive(true);
        HangingHook hook = GetHook();
        hook.gameObject.SetActive(true);
        activeHooks.Add(hook);
        Weight weight = weightGO.GetComponent<Weight>();
        weight.RemoveFromHook();
        weight.HangOnHook(hook);
    }
    private void DisableInactiveObjects()
    {
        foreach (KeyValuePair<GameObject,List<GameObject>> entry in pool)
        {
            foreach (GameObject o in pool[entry.Key])
            {
                Weight weight = o.GetComponent<Weight>();
                weight.RemoveFromHook();
                o.SetActive(false);
            }
        }
    }
    private HangingHook GetHook()
    {
        HangingHook hook = null;
        if (hooksPool.Count > 0)
        {
            hook = hooksPool[0];
            hooksPool.RemoveAt(0);
        }
        else
        {
            hook = Instantiate(hookPrefab,
hooksParent).GetComponent<HangingHook>();
        }
        return hook;
    }
    public HangingHook FindFreeActiveHook()
    {
        foreach (HangingHook hook in activeHooks)
        {
            if (hook.CanHang())
```

```
            {
                return hook;
            }
        }
        return null;
    }
    private void SetupHook(HangingHook hook)
    {
    }
    private void DisablePolledHooks()
    {
        foreach (HangingHook hangingHook in hooksPool)
        {
            hangingHook.gameObject.SetActive(false);
        }
    }
    private void PlaceHooks()
    {
        int hooksCount = activeHooks.Count;
        float step = poolWidth / hooksCount;
        float borderLeftPoint = hooksParent.position.x - (poolWidth / 2);
        for (int i = 0; i < activeHooks.Count; i++)
        {
            Vector3 pos = activeHooks[i].gameObject.transform.position;
            pos.x = borderLeftPoint + step * i;
            activeHooks[i].gameObject.transform.position = pos;
        }
    }
    private void ClearActiveHooks()
    {
        hooksPool.AddRange(activeHooks);
        foreach (HangingHook hangingHook in activeHooks)
        {
            hangingHook.ClearHook();
        }
        activeHooks.Clear();
    }
}

using System;
using UnityEngine;
public abstract class LevelQuest : MonoBehaviour
{
    public Action OnQuestFinished;
    public abstract void Run();
    public abstract void QuestUpdate();

}

using System;
using System.Collections.Generic;
using UnityEngine;
public class LevelQuestManager : MonoBehaviour
{
    public static LevelQuestManager Instance;
```

```
[SerializeField] private Transform spawnParent;
[SerializeField] private List<GameObject> questSequence;
private LevelQuest activeQuest;
private void Awake()
{
    if (Instance == null)
    {
        Instance = this;
    }
}
private void OnDestroy()
{
    if (Instance == this)
        Instance = null;
}
void Update()
{
    if (activeQuest != null)
    {
        activeQuest.QuestUpdate();
    }
}
public void MoveToNextQuest()
{
    if (activeQuest != null)
    {
        activeQuest.OnQuestFinished -= MoveToNextQuest;
    }

    activeQuest = CreateNextQuest();
    if (activeQuest == null)
    {
        Debug.Log("Failed to select next quest");
        return;
    }

    activeQuest.OnQuestFinished += MoveToNextQuest;

    activeQuest.Run();
}
private LevelQuest CreateNextQuest()
{
    LevelQuest quest = null;
    if (questSequence.Count > 0)
    {
        GameObject questGO = Instantiate(questSequence[0], spawnParent);
        quest = questGO.GetComponent<LevelQuest>();
        if (quest == null)
        {
            Debug.LogError("Error in quest creating");
            return null;
        }
        questSequence.RemoveAt(0);
    }
    return quest;
}
```

```
}

using System.Collections.Generic;
using System.Linq;

public class LeverQuestTools
{
	public static List<Weight> GetAvailableWeights() =>
		WeightPool.Instance?.GetComponentsInChildren<Weight>().ToList();

	public static Weight FindWeight(float mass, List<Weight> weights)
	{
		foreach (Weight weight in weights)
		{
			if (weight.Mass == mass)
			{
				return weight;
			}
		}

		return null;
	}
}
public class FinalExplanation : LevelQuest
{
	public override void Run()
	{
	}
	public override void QuestUpdate()
	{
	}
	public void ExitToMenu()
	{
		LevelLauncher.LoadMenu();
	}
}

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Video;
public class IntroPlotSequence : MonoBehaviour
{
	[SerializeField] private bool debugDisablePlot = true;
	[SerializeField] private VideoPlayer video;
	private void Awake()
	{
		bool hidePlot = !SaveSystem.IsPlotEnabled || debugDisablePlot;
		if (hidePlot)
		{
			EndIntro();
			return;
		}
		StartCoroutine(StartSequence());
	}
	private IEnumerator StartSequence()
```

```
      {
            video.Play();
            yield return null;

            while (video.isPlaying)
            {
                  yield return null;
            }
            EndIntro();
      }
      public void EndIntro()
      {
            gameObject.SetActive(false);
            LevelQuestManager.Instance?.MoveToNextQuest();
      }
}
using System;
[Serializable]
public class SaveData
{
}

using System;
using UnityEngine;
public class SaveSystem : MonoBehaviour
{
    private const string gameStartedKey = "gameStarted";
#region USER SETTINGS
    private const string plotEnabledKey = "plotEnabled";
    private const string alphabeticSort = "alphabeticSort";
    private const string backgroundVolumeKey = "backgroundVolumeKey";
    private const string soundEffectsVolumeKey = "backgroundVolumeKey";
    #endregion
#region LEVEL_PROGRESS
    private const string levelUnlockedKeyPattern = "levelfinished_{0}";
    private const string alwaysUnlockedLevelIndex = "c1_l1";
#endregion
    public static bool IsGameStarted
    {
        get => GetSavedBool(gameStartedKey, false);
        set => SetSavedBool(gameStartedKey, value);
    }

    public static bool IsPlotEnabled
    {
        get => GetSavedBool(plotEnabledKey, true);
        set => SetSavedBool(plotEnabledKey, value);
    }
    public static bool IsAlphabeticalLevelSort
    {
        get => GetSavedBool(alphabeticSort, false);
        set => SetSavedBool(alphabeticSort, value);
    }

    public static float BackgroundVolumeLevel
    {
```

```
        get => GetSavedFloat(backgroundVolumeKey, 0.5f);
        set => SetSavedFloat(backgroundVolumeKey, value);
    }

    public static float SoundEffectsVolumeLevel
    {
        get => GetSavedFloat(backgroundVolumeKey, 0.5f);
        set => SetSavedFloat(backgroundVolumeKey, value);
    }
    public static bool IsLevelUnlocked(string levelIndex)
    {
        if (levelIndex == alwaysUnlockedLevelIndex)
            return true;

        string levelUnlockedKey = String.Format(levelUnlockedKeyPattern,
levelIndex);
        return Convert.ToBoolean(PlayerPrefs.GetInt(levelUnlockedKey, 0));
    }
    private static bool GetSavedBool(string key, bool defaultValue = false) =>
        Convert.ToBoolean(PlayerPrefs.GetInt(key,
Convert.ToInt32(defaultValue)));
    private static void SetSavedBool(string key, bool value) =>
        PlayerPrefs.SetInt(key, Convert.ToInt32(value));

    private static float GetSavedFloat(string key, float defaultValue = 0f) =>
        PlayerPrefs.GetFloat(key, defaultValue);
    private static void SetSavedFloat(string key, float value) =>
        PlayerPrefs.SetFloat(key, value);
}

using UnityEngine;
public class ReadOnlyAttribute : PropertyAttribute
{

}

using TMPro;
using UnityEngine;
using UnityEngine.UI;
public class ChapterButton : MonoBehaviour
{
    [SerializeField] public ChapterUI chapterUI;
    [SerializeField] public Button button;
    [SerializeField] private TMP_Text chapterTitle;
    [SerializeField] private GameObject selectedView;
    [SerializeField] private GameObject deselectedView;

    public void SetupButton(ChapterUI chapterUI)
    {
        this.chapterUI = chapterUI;
        chapterTitle.text = chapterUI.Title;
    }
    public void SelectThisChapter()
    {
        selectedView.SetActive(true);
        deselectedView.SetActive(false);
```

```
    }
    public void DeselectThisChapter()
    {
        selectedView.SetActive(false);
        deselectedView.SetActive(true);
    }
}

using System.Collections.Generic;
using UnityEngine;
public class ChaptersUIManager : MonoBehaviour
{
    public static ChaptersUIManager Instance;
    [SerializeField] private LevelUIManager levelUIManager;
    [SerializeField] private List<ChapterUI> chaptersPlot;
    [SerializeField] private List<ChapterUI> chaptersAlphabet;
    [SerializeField] private GameObject chapterButtonPrefab;

    private ChapterButton currentSelectedChapter;
    private List<ChapterButton> chapterPool = new List<ChapterButton>();

    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
    }
    private void OnDestroy()
    {
        if (Instance == this)
        {
            Instance = null;
        }
    }
    void Start()
    {
        SetChaptersSort(SaveSystem.IsAlphabeticalLevelSort);
    }
    public void SetChaptersSort(bool isAlphabetical)
    {
        if (isAlphabetical)
            InitializeChapters(chaptersAlphabet);
        else
            InitializeChapters(chaptersPlot);
    }
    private void InitializeChapters(List<ChapterUI> chapters)
    {
        currentSelectedChapter = null;
        foreach (ChapterButton button in chapterPool)
        {
            button.gameObject.SetActive(false);
        }
        foreach (ChapterUI chapter in chapters)
        {
            CreateChapterButton(chapter);
```

```
        }
    }
    private void CreateChapterButton(ChapterUI chapterUI)
    {
        ChapterButton chapterButton = GetChapterButton();
        chapterButton.SetupButton(chapterUI);
        chapterButton.button.onClick.AddListener(() =>
SelectChapter(chapterButton));
        if (currentSelectedChapter == null)
        {
            SelectChapter(chapterButton);
        }
        else
        {
            chapterButton.DeselectThisChapter();
        }
    }
    private ChapterButton GetChapterButton()
    {
        foreach (ChapterButton button in chapterPool)
        {
            if (!button.gameObject.activeInHierarchy)
            {
                button.gameObject.SetActive(true);
                return button;
            }
        }
        GameObject go = Instantiate(chapterButtonPrefab, transform);
        ChapterButton chapterButton = go.GetComponent<ChapterButton>();
        chapterPool.Add(chapterButton);
        return chapterButton;
    }
    private void SelectChapter(ChapterButton caller)
    {
        if (currentSelectedChapter)
            currentSelectedChapter.DeselectThisChapter();

        caller.SelectThisChapter();
        currentSelectedChapter = caller;

        levelUIManager.CreateLevelsForChapter(caller.chapterUI);
    }
}
using UnityEngine;
[CreateAssetMenu(fileName = "ChapterUI", menuName =
"ScriptableObjects/ChapterUI", order = 1)]
public class ChapterUI : ScriptableObject
{
    public string Title;
    public LevelUI[] Levels;
}

using TMPro;
using UnityEngine;
using UnityEngine.UI;
public class LevelButton : MonoBehaviour
```

```
{
    public string Index;
    public TMP_Text levelTitle;
    public Button button;
    public void Setup(LevelUI level)
    {
        Index = level.Index;
        levelTitle.text = level.Title;
        button.interactable = SaveSystem.IsLevelUnlocked(Index);
    }

    public void SendLevelLoadRequest()
    {
        LevelLauncher.LoadLevel(Index);
    }
}
```

```
using System;
[Serializable]
public class LevelData
{
    public string levelCode;
    public string sceneName;
    public const string plotTextPattern = "Story_c{0}_l{1}";
}
```

```
using System;
using UnityEngine;
using UnityEngine.UI;
public class LevelSortTick : MonoBehaviour
{
      [SerializeField] private Toggle sortToggle;
      private void Start()
      {
            sortToggle.isOn = SaveSystem.IsAlphabeticalLevelSort;
            sortToggle.onValueChanged.AddListener(SwitchSort);
      }
      private void SwitchSort(bool value)
      {
            sortToggle.isOn = value;
            SaveSystem.IsAlphabeticalLevelSort = value;
            ChaptersUIManager.Instance.SetChaptersSort(value);
      }
}
```

```
using System;
using UnityEngine;
[Serializable]
public class LevelUI
{
    public string Index;
    public string Title;
    public Sprite finishedIcon;
}
```

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

```
public class LevelUIManager : MonoBehaviour
{
      [SerializeField] private GameObject levelButtonPrefab;

      private List<GameObject> levelIconPool = new List<GameObject>();
      public void CreateLevelsForChapter(ChapterUI chapter)
      {
            if (chapter.Levels.Length < levelIconPool.Count)
            {
                  for (int i = levelIconPool.Count - 1; i >=
chapter.Levels.Length; i--)
                  {
                        levelIconPool[i].SetActive(false);
                  }
            }

            for (int i = 0; i < chapter.Levels.Length; i++)
            {
                  GameObject go;
                  if (i >= levelIconPool.Count)
                  {
                        go = Instantiate(levelButtonPrefab, transform);
                        levelIconPool.Add(go);
                  }
                  else
                  {
                        go = levelIconPool[i];
                        go.SetActive(true);
                  }
                  LevelButton levelButton = go.GetComponent<LevelButton>();
                  levelButton.Setup(chapter.Levels[i]);
            }
      }
}

using UnityEngine;
using UnityEngine.UI;
public class LoadGameButton : MonoBehaviour
{
    private Button button;
    private void Start()
    {
        button = GetComponentInChildren<Button>();
        button.interactable = SaveSystem.IsGameStarted;
        button.onClick.AddListener(() => SaveSystem.IsGameStarted = true);
        button.onClick.AddListener(() =>
MenuWindowsController.Instance.EnableLevelSelect());
    }
}

using UnityEngine;
public class MenuWindowsController : MonoBehaviour
{
    public static MenuWindowsController Instance;

    [Header("Windows")]
```

```
    [SerializeField] private GameObject mainScreen;
    [SerializeField] private GameObject levelSelect;
    [SerializeField] private GameObject settings;

    private void Start()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        DisableAllWindows();
        EnableMainScreen();
    }
    private void OnDestroy()
    {
        if (Instance == this)
        {
            Instance = null;
        }
    }
    public void EnableMainScreen()
    {
        DisableAllWindows();
        mainScreen.SetActive(true);
    }

    public void EnableLevelSelect()
    {
        DisableAllWindows();
        levelSelect.SetActive(true);
    }
    public void EnableSettings()
    {
        DisableAllWindows();
        settings.SetActive(true);
    }

    private void DisableAllWindows()
    {
        mainScreen.SetActive(false);
        levelSelect.SetActive(false);
        settings.SetActive(false);
    }
}

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class NewGameButton : MonoBehaviour
{
    private Button button;
    private void Start()
    {
        button = GetComponentInChildren<Button>();
        button.onClick.AddListener(() => SaveSystem.IsGameStarted = true);
```

```
        button.onClick.AddListener(() =>
MenuWindowsController.Instance.EnableLevelSelect());
    }
}

using UnityEngine;
using UnityEngine.UI;
public class SettingsUIManager : MonoBehaviour
{
    [SerializeField] private Slider backgroundVolume;
    [SerializeField] private Slider effectsVolume;
    private void Start()
    {
        backgroundVolume.value = SaveSystem.BackgroundVolumeLevel;
        backgroundVolume.onValueChanged.AddListener(ChangeBackgroundVolume);
        effectsVolume.value = SaveSystem.BackgroundVolumeLevel;
        effectsVolume.onValueChanged.AddListener(ChangeEffectsVolume);
    }
    private void ChangeBackgroundVolume(float newValue)
    {
        SaveSystem.BackgroundVolumeLevel = newValue;
        BackgroundAudioManager.Instance.SetVolume(newValue);
    }

    private void ChangeEffectsVolume(float newValue)
    {
        SaveSystem.SoundEffectsVolumeLevel = newValue;
        SoundEffectsManager.Instance.SetVolume(newValue);
    }
}

using UnityEngine;
using UnityEngine.UI;
public class StoryModeTick : MonoBehaviour
{
    [SerializeField] private Toggle toggle;
    private void Start()
    {
        toggle.onValueChanged.AddListener(SaveNewValue);
    }
    private void OnEnable()
    {
        toggle.isOn = SaveSystem.IsPlotEnabled;
    }
    private void SaveNewValue(bool newValue)
    {
        SaveSystem.IsPlotEnabled = newValue;
    }
}

using System;
using TMPro;
using UnityEngine;
using UnityEngine.UI;
public class TutorialPanel : MonoBehaviour
{
```

```
    public static TutorialPanel Instance;
    public Action OnContinueButton;
    [SerializeField] private Image bg;
    [SerializeField] private TMP_Text text;
    [SerializeField] private Button continueButton;
    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        HideTutorial();
        HideContinueButton();
    }
    private void OnDestroy()
    {
        if (Instance == this)
        {
            Instance = null;
        }
    }
    public void SetText(string text)
    {
        this.text.text = text;
    }
    public void HideTutorial()
    {
        gameObject.SetActive(false);
    }
    public void ShowTutorial()
    {
        gameObject.SetActive(true);
    }
    public void ShowContinueButton()
    {
        continueButton.gameObject.SetActive(true);
    }
    public void HideContinueButton()
    {
        continueButton.gameObject.SetActive(false);
    }
    public void OnContinue()
    {
        OnContinueButton?.Invoke();
    }
}

using System;
using System.Collections.Generic;
using UnityEngine;
public class BackgroundAudioManager : MonoBehaviour
{
    public static BackgroundAudioManager Instance;

    [SerializeField] private List<AudioClip> backgroundMusic;
    private int playingClipIndex = -1;
```

```
    private AudioSource audioSource;

    private void Start()
    {
        if (Instance == null)
        {
            Instance = this;
        }

        audioSource = GetComponent<AudioSource>();
        audioSource.volume = SaveSystem.BackgroundVolumeLevel;
    }

    private void OnDestroy()
    {
        if (Instance == this)
        {
            Instance = null;
        }
    }

    private void Update()
    {
        if (!audioSource.isPlaying)
        {
            audioSource.clip = SelectNextClip();
            // audioSource.PlayOneShot(SelectNextClip());
            audioSource.Play();
        }
    }
    private AudioClip SelectNextClip() =>
        backgroundMusic[++playingClipIndex % backgroundMusic.Count];
    public void SetVolume(float value)
    {
        audioSource.volume = value;
    }
}

using UnityEngine;
public static class Contants
{
    public static int DEFAULT_LAYER => LayerMask.NameToLayer("Default");
    public static int IGNORE_RAYCAST_LAYER => LayerMask.NameToLayer("Ignore
Raycast");
}

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class CursorController : MonoBehaviour
{
    [SerializeField] Texture2D idleCursorTexture;
    [SerializeField] Texture2D clickedCursorTexture;
    // Start is called before the first frame update
    void Awake()
    {
```

```
        if (FindObjectsOfType<CursorController>().Length > 1)
        {
            Destroy(this.gameObject);
        }
        else
        {
            DontDestroyOnLoad(this);
        }
    }
    // Update is called once per frame
    void Update()
    {
        if(Input.GetMouseButton(0))
        {
            Cursor.SetCursor(clickedCursorTexture, Vector2.zero,
CursorMode.Auto);
        }
        if(Input.GetMouseButtonUp(0))
        {
            Cursor.SetCursor(idleCursorTexture, Vector2.zero, CursorMode.Auto);
        }
    }
}

using System;
using UnityEngine;
using UnityEngine.SceneManagement;
public class LevelLauncher : MonoBehaviour
{
    public static LevelLauncher Instance;

    private const string sceneNameFormat = "Level_{0}";
    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
    }
    private void OnDestroy()
    {
        if (Instance == this)
            Instance = null;
    }
    public static void LoadMenu()
    {
        SceneManager.LoadScene("Menu");
    }
    public static void LoadLevel(string levelIndex)
    {
        SceneManager.LoadScene(GetSceneName(levelIndex));
    }
    private static string GetSceneName(string levelIndex) =>
        String.Format(sceneNameFormat, levelIndex);
}
using UnityEngine;
```

```
public class QuitGame : MonoBehaviour
{
    public static QuitGame Instance;
    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
    }
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            Quit();
        }
    }
    public void Quit()
    {
        #if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
        #endif
        Application.Quit();
    }
}
```