МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет
Центр заочної, дистанційної та вечірньої форм навчання
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)
_____ грудня 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 - Комп'ютерних наук,
освітньо-професійної програми «Інформатика»
на тему: «Інформаційна технологія проєктування концепції та генерування середовища комп'ютерної ігрової системи»
здобувача групи ІН.мз-21с Ципліна Дмитра Олександровича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Дмитро ЦИПЛІН
(підпис)

Керівник,
в.о. завідувача кафедри,
кандидат технічних наук, доцент Ігор ШЕЛЕХОВ _____
(підпис)

Суми – 2023

Сумський державний університет
Центр заочної, дистанційної та вечірньої форм навчання
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми «Інформатика»
здобувача групи ІН.мз-21с Ципліна Дмитра Олександровича

1. Тема роботи: «Інформаційна технологія проєктування концепції та генерування середовища комп'ютерної ігрової системи»
затверджую наказом по СумДУ від *«01» грудня 2023 року № 0475-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 16 грудня 2023 року*
3. Вхідні дані до кваліфікаційної роботи _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Проєктування ігрової системи. 3) Розробка інформаційного та програмного забезпечення. 4) Аналіз результатів.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
_____
6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 20 ___ р.
Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналіз проблеми предметної області, постановка й формування завдань дослідження*		
2	*Проєктування ігрової системи*		
3	*Розробка інформаційного та програмного забезпечення*		
4	*Аналіз отриманих результатів*		
5	*Оформлення пояснювальної записки до кваліфікаційної роботи*		

Здобувач вищої освіти _____ Керівник _____
(підпис) (підпис)

АНОТАЦІЯ

Записка: 47 стр., 11 рис., 1 додаток, 17 використаних джерел.

Обґрунтування актуальності теми роботи – За останнє десятиліття комп'ютерні ігри постійно стають все більш популярними, в той час як настільні ігри все ще актуальні завдяки локальним спільнотам та конвенціям, присвяченим їм. Поєднуючи переваги обох, можна створювати унікальні ігри, які не могли б існувати інакше.

Об'єкт дослідження — процес проєктування гібридної ігри, яка здатна моделювати фізичні елементи настільної гри у вигляді цифрових елементів комп'ютерної гри.

Мета роботи — Розробити комплекс інформаційного та програмного забезпечення гібридної ігри.

Методи дослідження — методи інформаційного аналізу і синтезу інформаційних ігрових систем.

Результати — запропоновано комплекс моделей, методів та засобів інформаційної технології проєктування гібридної гри, що ефективно моделює фізичні елементи настільної гри у вигляді цифрових елементів комп'ютерної гри. Тестування ефективності інформаційної технології виконано шляхом проєктування і програмної реалізації модифікації гри баусак.

ІНФОРМАЦІЙНА ІГРОВА СИСТЕМА, ІГРОВИЙ ПРОТОТИП, UNITY3D

ЗМІСТ

ВСТУП

Обґрунтування актуальності теми роботи – За останнє десятиліття комп’ютерні ігри постійно стають все більш популярними, в той час як настільні ігри все ще актуальні завдяки локальним спільнотам та конвенціям, присвяченим їм. Поєднуючи переваги обох, можна створювати унікальні ігри, які не могли б існувати інакше.

Об’єкт дослідження — процес проєктування гібридної ігри, яка здатна моделювати фізичні елементи настільної гри у вигляді цифрових елементів комп’ютерної гри.

Мета роботи — Розробити комплекс інформаційного та програмного забезпечення гібридної ігри.

Методи дослідження — методи інформаційного аналізу і синтезу інформаційних ігрових систем.

Гіпотеза. Розробка інструментарію для перетворення популярних настільних ігор в комп’ютерні дозволить значно розширити їх функціонал і створити новий клас гібридних ігор.

Новизна. В роботі запропоновано комплекс інформаційних, методичних, алгоритмічних і програмних засобів для гібридізації настільних ігор шляхом їх повної або часткової діджиталізації.

Структура. Дане робота складається зі вступу, аналітичного огляду, постановки задачі, вибір методу розв’язання поставленої задачі, опису програмного забезпечення інформаційної системи, висновків, списку використаних джерел та додатків.

1 АНАЛІТИЧНИЙ ОГЛЯД І ПОСТАНОВКА ЗАДАЧІ

1.1 Основні визначення

Розглянемо основні типи складових ігор, якими може взаємодіяти користувач.

Фізичний складова означає компонент, з яким може взаємодіяти гравець без залучення програмного забезпечення комп’ютерна. У цій роботі він зазвичай використовується у значенні фізична гра, тобто гра, яка не використовує електроніку, або як фізичний елемент, тобто елемент, до якого можна фізично доторкнутися (на відміну від елемента, показаного на екрані комп'ютера). Цифровий, з іншого боку, використовується для позначення того, що процес взаємодії забезпечується комп'ютером або іншим електронним пристроєм. Гібридний в контексті цієї роботи використовується для позначення поєднання фізичних і цифрових компонентів. Гібридна гра - це гра, яка поєднує фізичні та електронні компоненти, наприклад, гра, яка відбувається на повністю неелектронній дошці, але використовує смартфон для відображення додаткової інформації, пов'язаної з ігровим процесом, або гра, в якій гравець переміщує фізичні ігрові фігури на екрані, який може розпізнавати об'єкти, і який використовує рух і положення фігур як вхідні дані. Метою цієї кваліфікаційної роботи є вивчення аспектів фізичних та цифрових ігор, щоб полегшити створення таких гібридних ігор у значущий спосіб, тобто фізичні та цифрові аспекти доповнюють один одного, і жоден з них не може бути вилучений без зміни способу, в який грається гра[1].

Водночас настільна гра означає будь-яку гру, що грається на столі (або подібній пласкій поверхні), чи то суто фізичну з дотиком до фігур, чи то з використанням технологій, чи то поєднання обох варіантів.

Термін "настільна гра" має різні визначення.[2] Суворе визначення зазвичай включає дошку як основний компонент. Проте воно виключає ігри, які не включають дошку і в які грають виключно за допомогою кубиків або

карток. У цій роботі слово вживається у його нестрогому загальновживаному значенні, що включає всі види фізичних і деякі цифрові нерольових настільних ігор, незалежно від використовуваних компонентів, а також інші ігри в приміщенні, в які можна грати без компонентів і столу, наприклад, шаради

Традиційно настільні ігри визначалися використанням фізичних фігур, а також певними атрибутами, пов'язаними з ігровим процесом. Це дало поштовх до появи цифрового ігрового жанру, який зазвичай називають "цифровими настільними іграми". Ці два формати змішуються таким чином, що штучна межа, яка їх розмежовує, розмивається, і багато цифрових ігор зараз створені так, щоб нагадувати настільні ігри. Популярними є мобільні ігри, що містять атрибути настільних ігор - включаючи покрокову гру, перетасовану колоду ігрових фігур, видиму дошку, поділену на плитки, і прозорі правила без прихованих модифікаторів. Хоча ця робота присвячена створенню гібридних настільних ігор, а не конкретно іграм, що належать до жанру настільних ігор, розгляд цього жанру та його фізичних ігор може дати уявлення про те, які ігрові механіки та властивості уможливлюють фізичні ігри[3].

1.2 Поточний стан та тенденції розвитку гібридних ігрових систем

Першою грою, що поєднує фізичні та цифрові елементи, є PingPongPlus[1]. Над столом для пінг-понгу встановлюється відеопроектор, який за допомогою звуку відстежує, коли м'яч потрапляє на стіл. При цьому використовуються різні доповнення. Деякі доповнення - це просто візуалізація зі звуком, і хоча вони не змінюють ігровий процес пінг-понгу. Інші доповнення передбачають використання столу як засобу для створення візуальних творів мистецтва або звуку. Один з режимів, зокрема, експериментує зі зміною ігрового процесу, використовуючи проектор як джерело світла в темній кімнаті, а місця, в які потрапляють, затемнюються, додаючи ще один стратегічний рівень до гри.

Pirates! [1]- це гра заснована на наближенні/розташуванні, з використанням портативних комп'ютерів у попередньо підготовленій кімнаті. Кожен гравець представляє піратський корабель, а кілька локацій у кімнаті є островами, до яких можна дістатися, просто пройшовши туди пішки. Гравець проти гравця також можливість з’являється, якщо гравці знаходяться поруч один з одним. Таким чином, технологія, що враховує розташування гравців, може покращити багатокористувацькі ігри, які проводяться у фізичному світі

False Prophets [1]– один з прикладів гібридної настільної гри. У грі є спроектована карта, на яку кожен гравець переміщує персонажа, якого може відчути дошка. Окрім загальнодоступної інформації, що проектується, кожен гравець також має портативний комп'ютер для приватної інформації. Основна увага приділяється взаємодії гравців між собою. Використання кишенькового комп'ютера обмежується тим, що не може бути виражено за допомогою руху ігрових фігур. Крім того, приватне спілкування та обмін інформацією не опосередковується ігровою системою.

STARS [2-5] - це платформа для доповнених настільних ігор. STARS динамічно поєднує різні типи пристроїв взаємодії, такі як смартфони або гарнітури, з інтерактивним ігровим столом. Доповнені настільні ігри STARS надають ряд функцій, таких як динамічні ігрові дошки або приватні канали зв'язку, які виходять за рамки традиційних настільних ігор, але в той же час зберігають динаміку взаємодії, орієнтовану на людину, що робить гру в настільні ігри радісним груповим досвідом. Основними перевагами для ігор STARS є стійкість, складні правила гри та динамічна візуалізація інформації, і зазначають, що використання їхньої системи може допомогти розробникам сконцентруватися на створенні самої гри, замість того, щоб створювати інфраструктуру довкола неї. У самій системі реалізовано кілька можливих режимів введення (настільні пішаки, мова, настільні жести, настільний WIMP смартфони) і кілька режимів виведення (настільний дисплей, настінний

дисплей, дисплей смартфону, аудіо з гучномовця і аудіо з навушників). Як наслідок віртуальний простір розширюється у фізичний світ. При цьому зміни стосуються і процесів розробки дизайну ігор та фізичних інтерфейсів

TARBoard [2-5]- це настільне ігрове середовище, яке використовує камеру під скляним столом у поєднанні з дзеркалом, щоб бачити маркери під картами, викладеними на столі. Окрема камера використовується для управління доповненням гри. При цьому використання камери під столом вирішує проблеми з оклюзією

Ще одна система, призначена для настільного використання (хоча і не зовсім для ігор), - це TViews [4,5]. TViews має дисплей замість проектора і не використовує жодних камер, а лише комбінацію акустичного та інфрачервоного зв'язку для визначення місцезнаходження фізичних елементів на столі.

Паралельно з розробкою апаратно-програмного забезпечення гібридних ігор проводилися і опитування гравців, що їх задоволеності від результатів такої гібридизації. Наприклад [5], при порівнювалося використання знакових (візуально представляють те, що вони означають) і символічних (абстрактних) фігур у настільних іграх, а також те, як вони впливають на веселощі та взаєморозуміння в процесі гри. В результаті виявилося, що жоден з цих типів не є більш придатним для розуміння чи розваги, але класичним конічним фішкам надають перевагу, тому, що вони відповідають темі гри і їхній зовнішній вигляд є впізнаваним. Тим часом символічні фішки мають перевагу в тому, що їх можна використовувати багаторазово.

Цифрові настільні ігри створювалися і для людей похилого віку [5]. При цьому частина програмного забезпечення була призначена для порівняння реакцій гравців під час ігрових сесій зі звичайною настільною грою і гібридною з метою оцінки впливу на такі аспекти, як занурення, потік, афект і виклик.

У Surface-poker [2,3] було додано спеціалізоване обладнання - датчики ЕЕГ з метою з'ясувати, чи вплине демонстрація нервозності іншого гравця на ігровий процес.

1.3 Постановка задачі

Метою кваліфікаційної роботи є розв’язання практичної задачі з розробки комплексу інформаційного та програмного забезпечення гібридної ігри. Для досягнення поставленої мети необхідно виконати такі основні завдання роботи:

1) Аналіз існуючої фізичної гри і визначення основних фізичних компонентів, які можна подати в цифровому вигляді.
2) Розробка концептуальної схеми ігрової системи.
3) Розробка цифрових моделей фізичних компонентів з урахуванням особливостей їх застосування в ігровому процесі.
4) Розробка логічного контенту взаємодії компонентів в ігровому середовищі
5) Визначення набору інструментів взаємодії ігрової системи з гравцями в режимі введення керуючих дії і виведення їх результатів.
6) Програмна реалізація ігрової системи та перевірка її працездатності.

2 ПРОЄКТУВАННЯ ІГРОВОЇ СИСТЕМИ

2.1 Аналіз фізичних компонентів настільних ігор

Хоча настільна гра традиційно була фізичною за своєю природою, комп'ютери, смартфони та планшети вже давно сприйняли настільні ігри як жанр. Ці фізичні та цифрові настільні ігри часто мають спільні риси, які є індикаторами можливості моделювання їх фізичних компонентів у вигляді цифрових аналогів: покроковості гри, перетасування колод ігрових фігур, видимість дошки, розділення її на плитки, забезпечення виконання прозорих правил всіма учасниками гри [6].

Спільною рисою всіх суто фізичних настільних ігор є те, що правила повинні виконуватися гравцями. Це призводить до цікавого наслідку: Зазвичай всі правила повністю відомі і прозорі для кожного гравця в будь-який момент гри, що дозволяє їм приймати обґрунтовані рішення. Зауважте, що це стосується лише самих правил. Ігровий процес може стати менш прозорим через прихований стан (наприклад, коли гравці тримають приховані карти), невідомість реакції опонента, а також через випадковість, що вноситься, наприклад, за допомогою кубиків або прихованих елементів. З іншого боку, багато відеоігор не повідомляють гравцям точних правил, а просто виконують їх за лаштунками. Наприклад, у відеогрі може бути рейтинг броні, і хоча гравець знає, що вищий рейтинг броні допомагає персонажу отримувати менше пошкоджень, він може не знати, як саме розраховується це зменшення.

Щоб гравці могли розуміти і виконувати правила, вони повинні бути обмеженої складності. Хоча, безумовно, існують настільні ігри, які є дуже складними і потребують багато часу для розуміння та гри, більшість з них мають прості формули, невелику кількість чисел, обмежену кількість обчислень між діями та обмежений набір дій. Крім того, оскільки настільну гру не можна легко зберегти, як відеогру, більшість настільних ігор призначені для того, щоб пройти їх від початку до кінця за кілька годин [7].

Окрім ігрової механіки, існують певні компоненти, які асоціюються з фізичними настільними іграми і які часто імітуються і в цифрових настільних іграх: дошка, кубики, картки/карти, покроковий режим. Крім того, оригінальність ігри забезпечується системою складності і правилами визначення переможця.

Дошка [1-5] – це найпоширеніший компонент, від якого і походить назва "настільна гра". На дошці позначені різні зони, що дозволяє гравцям надавати фішкам, які вони ставлять на неї, додаткового значення, наприклад, "ця фішка-робітник зараз збирає зерно" або "ця фішка-рахунок позначає, що мій рахунок дорівнює 100". Одна з головних переваг використання дошки з зонами полягає в тому, що ці зони дозволяють фішкам рухатися в дискретному просторі. Фізичний світ має безперервний простір - однак, на практиці важко обмежити рух без використання інструментів, таких як лінійки, і часто складність "пересунути фігуру на 6 сантиметрів" не потрібна. Натомість настільні ігри часто обмежують рух окремими плитками, сіткою або просто локаціями, з'єднаними графом чи подібною структурою, таким чином дозволяючи гравцям легко переміщатися "на 3 кроки" або "на сусідню плитку". Наявність дошки, розділеної на непомітні проміжки, також полегшує розуміння стану гри і можливостей - наприклад, легко зрозуміти, що дві фігури можуть атакувати одна одну, якщо вони знаходяться на відстані 3 клітинок або менше, просто подивившись на них, але якщо дальність їхньої атаки вказана в сантиметрах, то може знадобитися лінійка. Деякі винятки з правила "переміщення тільки між дискретними просторами" - це ігри, які насправді керуються фізикою ігри, де рух є ключовою механікою, і тому фактично використовують лінійки або подібні пристрої, а також ігри, які взагалі не покладаються на дошку.

У той час як існує багато настільних ігор, де все повністю визначається діями гравців, в інших іграх присутня випадковість - часто за допомогою гральних кубиків. У відеоіграх також використовується випадковість, але

знову ж таки є різниця в прозорості. Наприклад, коли кидається гральний кубик, гравець може бути відносно впевнений, що він має шість граней з однаковою ймовірністю, і точно знає, як буде використаний результат - наприклад, кубик, кинутий у бою, може завдати вказаної кількості шкоди, якщо тільки не випаде шістка, в такому випадку це "критичне попадання", і гравець може кинути другий кубик, додавши до першого результату. У відеоіграх це часто приховано, показуються лише результати, і гравець не може бути впевненим, чому його атака завдала саме таку кількість шкоди. Також часто відчувається різниця в управлінні: У настільних іграх гравець кидає кубик, тим самим нібито визначаючи свою долю; у відеогрі, з іншого боку, гравцеві може здатися, що результат "вирішує" комп'ютер. Хоча часто немає різниці в механіці (оскільки гравець не може контролювати фізичний результат кидання костей, а цифровий генератор випадкових чисел може видавати такі ж "випадкові" результати), досвід може відрізнятися у сприйнятті та відчуттях.

Перетасовані фішки або карти - це інші елементи, які дають гравцеві рандомізований набір варіантів вибору, часто прихований від інших гравців. Багато ігор повністю засновані на картах. Це колекційні карткові ігри, де карти представляють ресурси, юніти, заклинання і подібні елементи, або, де створення власних карткових колод гравцями фактично є частиною гри, і оскільки колода регулярно перетасовується, гравці повинні маніпулювати своїми шансами, додаючи карти в колоду і скидаючи карти з колоди.

У відеоіграх [6,7] зазвичай штучно обмежують швидкість гри, використовуючи швидкість руху та таймери - наприклад, аватар гравця може рухатися лише на 5 ігрових метрів за секунду, або таймер перезарядки зброї - на 1 секунду. Це дозволяє грати одночасно, не зводячи ігровий процес до чистої можливості виконати якомога більше дій за найкоротший проміжок часу. Крім того, відеоігри зазвичай мають різні інтерфейси введення для різних гравців. З іншого боку, у фізичних настільних іграх важко забезпечити дотримання

часових рамок руху та дій, а люди часто грають в одному просторі і фізично заважають один одному [8]. Ці обмеження призводять до того, що настільні ігри зазвичай є покроковими, тобто мають лише одного активного гравця, тоді як інші гравці просто спостерігають або планують свій хід. Якщо гра передбачає одночасні дії, то вони зазвичай обмежені: Наприклад, може бути фаза, коли гравці одночасно приймають рішення про таємну дію, перш ніж кожен одночасно розкриє свою, або гравці діють одночасно у власному обмеженому просторі, не перешкоджаючи іншим гравцям.

Як згадувалося раніше, більшість з цих характеристик мають винятки, що забезпечують унікальність гри. Щоб проілюструвати цю тезу, деякі з них будуть згадані тут [1-5].

У суто фізичних іграх правила повинні виконуватися гравцями, але це не обов'язково означає, що всі гравці знають усі правила або що правила не можуть змінюватися несподіваним чином. Це призводить до того, що гравці намагаються виграти проти гри разом, а гра має знайомити їх з правилами під час гри за допомогою системи, що базується на подіях і нагадує підручник. Таким чином, гра змушує гравців приймати рішення, не знаючи повного набору правил, що полегшує початок гри і може посилити напругу і несподіванку в процесі . Реалізація такого механізму можлива для будь-якого виду настільних ігор. Наприклад, карткові ігри, які починаються з дуже простими основними правилами, що змінюються в залежності від дій гравців, торговельні ігри, що мають секретний набір правил для кожного гравця щодо початкового набору товарів та правил, бонусів/штрафів при торгівлі окремими товарами або певними їх комбінаціями [9-10]. При цьому гравці можуть вільно обмінюватися товарами та знаннями правил, які після цього стають доступними і застосованими до всіх гравців одночасно.

Складність гри впливає на її тривалість і визначається складністю правил і кількістю виключень з них, а також механізмами їх динамічної зміни

в ігровому процесі. Ігри на реакцію часто передбачають одночасну гру в одній і тій самій фізичній зоні [11-12]. Інші ігри мають лише певні одночасні фази в спільній фізичній зоні. Це має наслідки для інших гравців: Якщо один гравець бере фігуру, інші гравці більше не можуть її використовувати. Багато комунікативних ігор для вечірок передбачають гру в реальному часі у вигляді зворотного відліку, коли гравці повинні виконати завдання якомога швидше, часто з кількома гравцями одночасно [13-14].

2.2 Аналіз настільної гри «Баусак»

Фактично БАУСАК [1-5] - це цілий набір ігор, в яких вам належить будувати вежі. Незліченні варіанти будівництва гарантують, що кожна конструкція буде унікальною завдяки вибору з 66 будівельних частин, які і є основними фізичними компонентами гри (рис. 2.1).

Рисунок 2.1 – Фізичні компоненти гри

В ході гри гравець випадковим чином отримує одну з частин, яку йому треба розмістити на вже розміщені частини або на основу – частину, яка розміщується першою на ігровий стіл. На рис. 2.2 подано варіант правильного і неправильного розміщення темної частини.

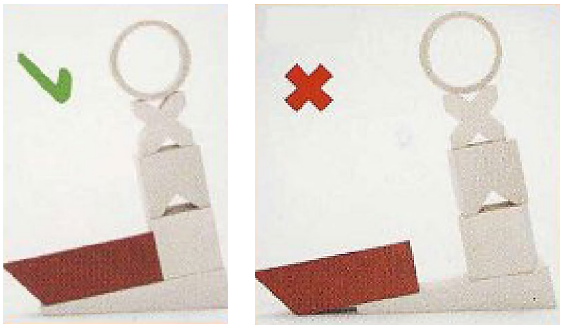

Рисунок 2.2 – Варіанти розміщення частин на основі

Переміщувати вже встановлені частини не можна, за винятком встановлення нової частини, якою можна посунути всі інші.

Існує декілька варіантів гри, в яких формується одна башта або декілька башт (для кожного гравця власна), використовується випадковий вибір частин (однієї на кожному кроці) або необхідно встановити порядок розміщення декількох випадково обраних частин. Крім того, деякі правила застосовують економічні моделі, коли кожна частина «купується» гравцями на аукціоні.

В цій роботі розглянемо класичні правила, що мають назву «Вавилонська вежа» (рис. 2.3), коли гравці по черзі встановлюють по одній частині обраній випадково. Переможцем є той гравець, який останнім встановив частину таким чином, що це не призвело до падіння башти. Нічия оголошується у випадку, коли всі 66 частин було розміщено на башті і вона не впала.

Гра може тривати декілька раундів, в ході яких формуються нові башти. Тоді переможець визначається або як той гравець, хто першим виграв певну кількість раундів, або той, хто виграв максимальну кількість рауднів.

Альтернативною до Баусака є гра Дженга, що починається з вежі, яка вже стоїть на місці, і гравці по черзі знімають фігури і розміщують їх на вершині вежі, поки вона остаточно не впаде, що стає дедалі важче, коли вежа втрачає вирішальні фігури. Переможцем стає гравець, який раніше гравця, чия дія призвела до падіння вежі.

Рисунок 2.3 – Сформована башта

Аналіз гри показує, що Баусак є грою, де задіяна гравітація і є необхідність балансування фігур на інших фігурах.

Для гібридізації таких ігор на необхідно моделювання не лише фізичних частин веж і гравітації, але і тонкими рухів гравців та фізичного зворотного зв'язку. При цьому, для ускладнення або полегшення гри можлива зміна законів гравітації, використанням оригінальних моделей сили тертя тощо.[15]

2.3 Моделювання фізичних частин

Розробка цифрових моделей фізичних компонентів гри виконується у 3DS Max для подальшого експорту у ігровий рушій [16]. При цьому необхідно відтворити набір простих частин у формі паралелепіпедів (рис. 2.4),

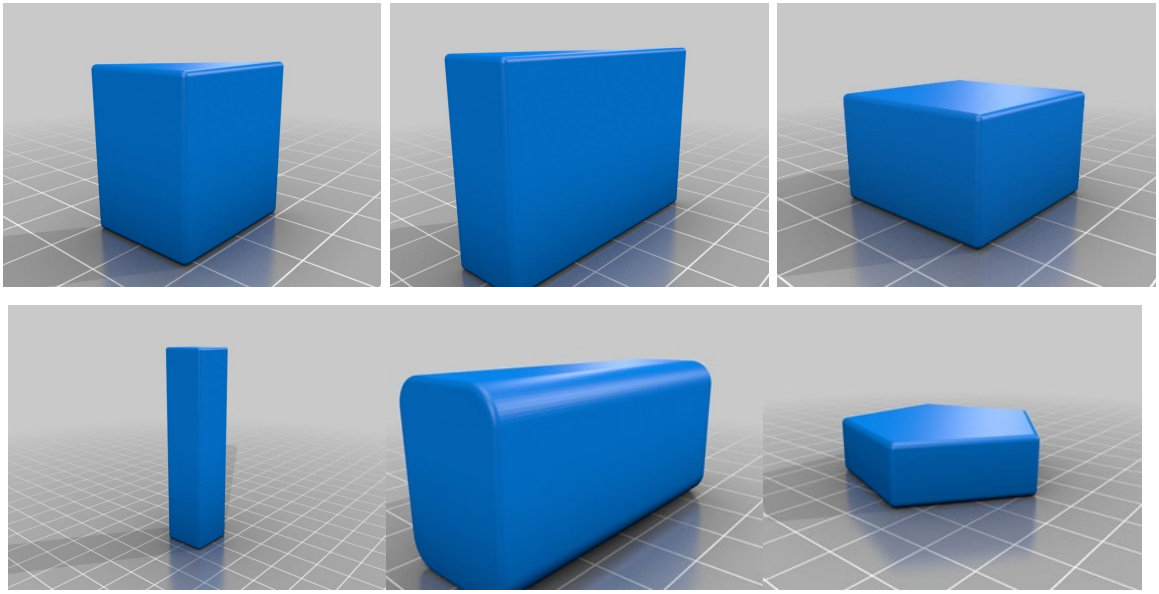

Рисунок 2.4 – Моделі найпростіших чистин вежі

серед яких виділити елементи що можуть бути основою (рис. 2.5),

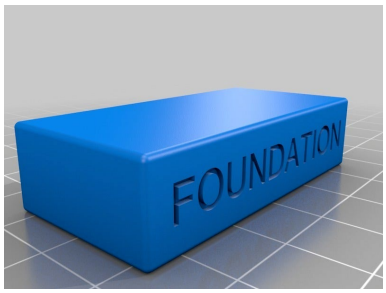

Рисунок 2.5 – Моделі основи вежі

більш складні елементи (рис. 2.6), що мають форму циліндру чи кільця

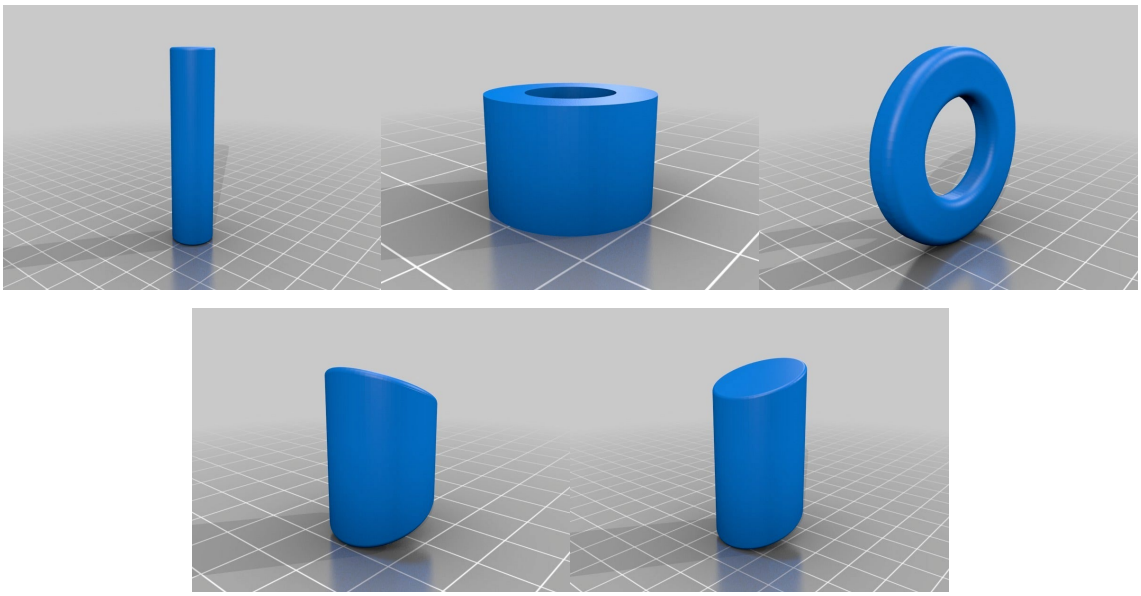

Рисунок 2.6 – Моделі циліндричних та кільцевих чистин вежі

конічні частини (рис.2.7)

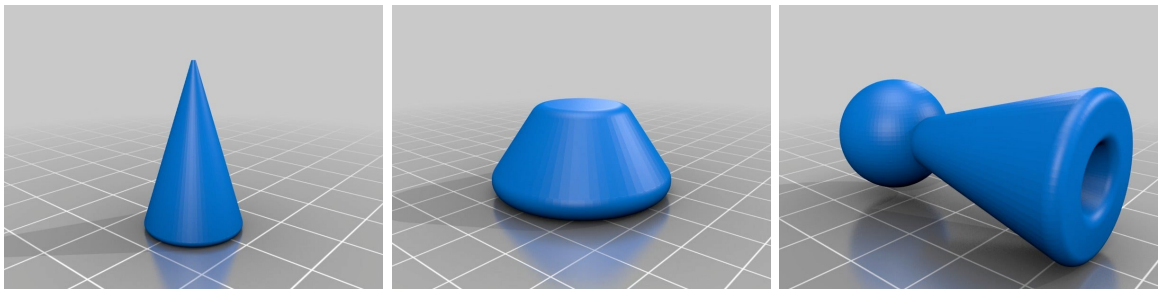

Рисунок 2.7 – Моделі конічних чистин вежі

моделі сферичної та напівсферичної форми (рис. 2.8),

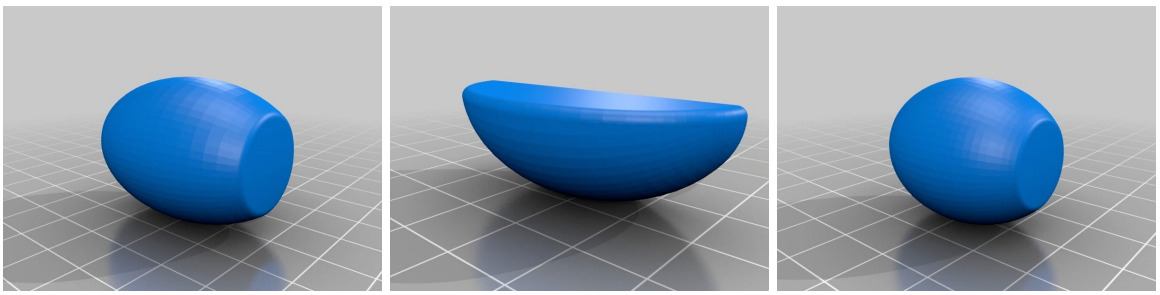

Рисунок 2.8 – Моделі сферичних та напівсферичних чистин вежі

моделі частин з отворами та вирізами (рис.2.9),

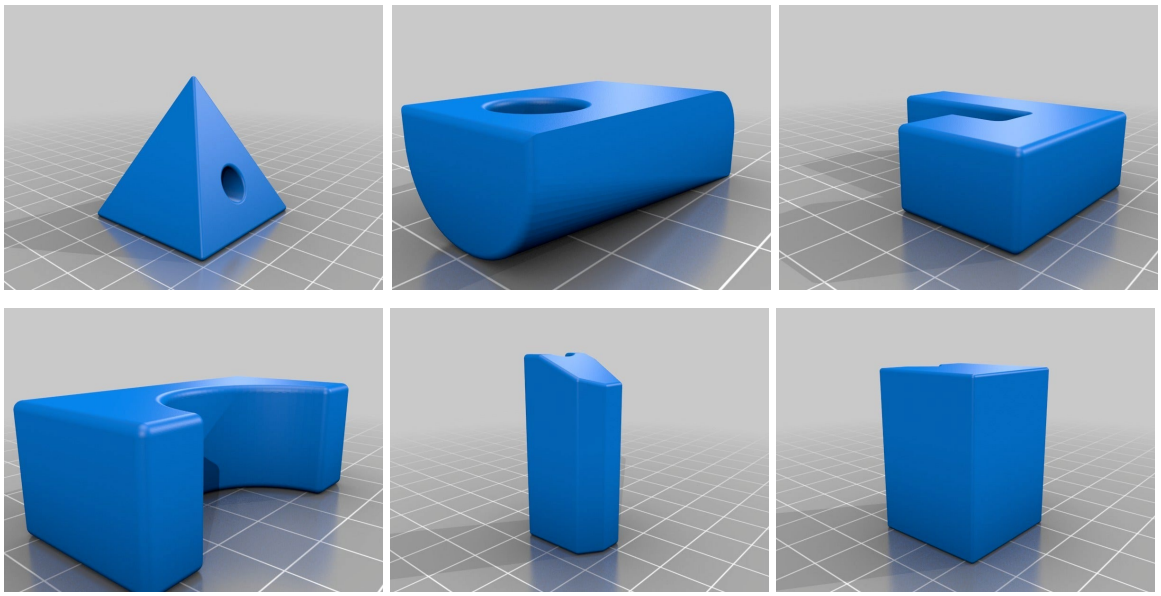

Рисунок 2.9 – Моделі чистин вежі з отворами і вирізами

моделі частин неправильної форми (рис.2.10)

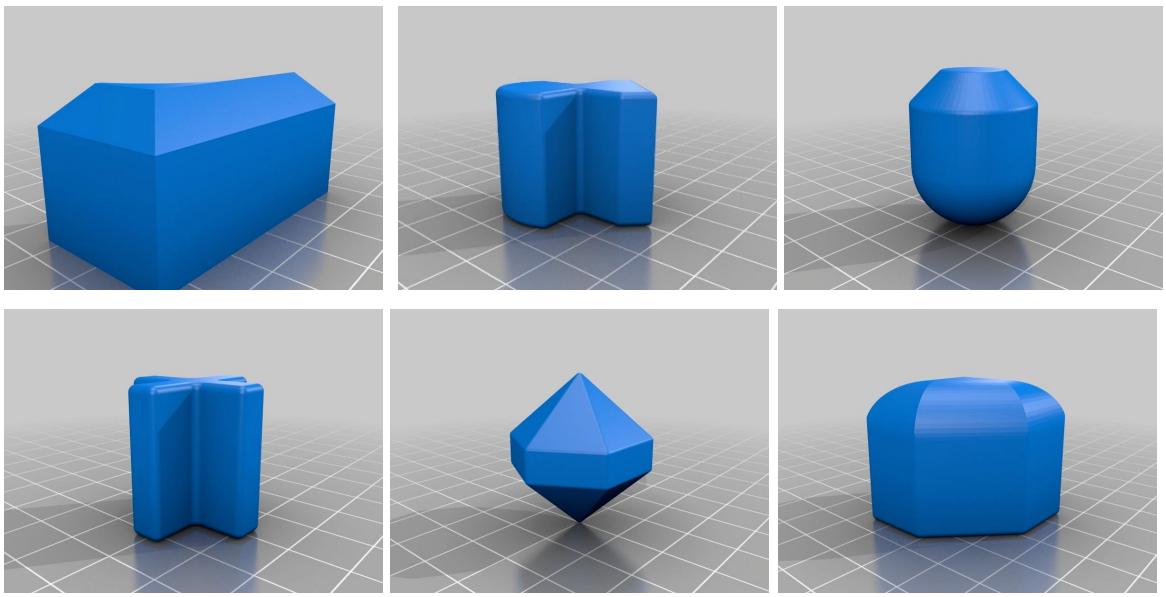

Рисунок 2.10 – Моделі чистин вежі неправильної форми

моделі частин складних для використання при формуванні вежі (рис.2.11)

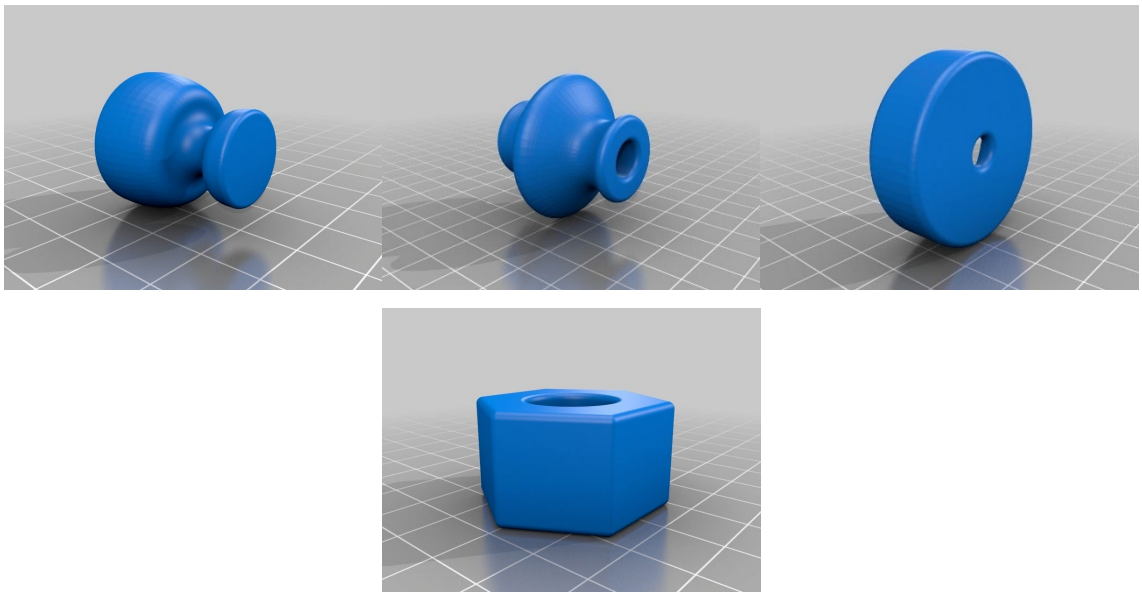

Рисунок 2.11 – Моделі найскладніших чистин вежі

Текстурування частин виконується у редакторі матеріалів 3DS Max або функціями інтегрованого засібу розроблення ігор. Для подальшого використання моделі експортують. У запропонованих моделях не складна геометрія, що дозволяє залишити їх низькополігональними у форматі 3DS Max. Але для складних моделей використовується спеціалізований форматом fbx.

3 ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Моделювання гравітації в Unity

Unity [6-17] - популярний інтегрований засіб розроблення ігор, що дає змогу створювати тривимірні візуальні сцени та додавати різні ефекти. Одним із важливих аспектів розробки ігор є реалістичне моделювання фізики об'єктів в ігровому світі.

Значна частина роботи з ігровим рушієм Unity передбачає знайомство з Unity Rigidbody [17] та його різноманітними властивостями. Rigidbody в Unity дозволяє вашим GameObjects діяти під контролем фізики.

Це дозволяє вам взаємодіяти з фізикою ваших об'єктів і візуалізувати, як Unity намагається імітувати фізику реального світу

Розглянемо, як додати фізику об'єкту в Unity, використовуючи компонент Rigidbody. Rigidbody додає об'єкту фізичні властивості, як-от маса, швидкість, обертання та гравітація. Це дає змогу об'єкту взаємодіяти з іншими об'єктами та оточенням.

Кожний 3D об'єкт, наприклад, куб або сфера, повинен мати компонент Rigidbody, щоб йому було додано фізичні властивості. Після додавання Rigidbody можливе налаштувати параметрів маса та гравітація. Маса визначає, наскільки сильно об'єкт буде взаємодіяти з іншими об'єктами, а гравітація вказує, чи буде об'єкт підданий силі тяжіння Землі або іншій силі гравітації.

Якщо компонент Rigidbody будуть мати декілька об'єктів, то вони стикатимуться, відскакуватимуть і взаємодіятимуть один з одним відповідно до правил фізики. Що надасть змогу створювати реалістичні симуляції та цікаві ігрові механіки.

Під час роботи над фізичною поведінкою об'єктів у Unity необхідно налаштувати кілька компонентів, щоб вони правильно взаємодіяли з фізичним рушієм. Розглянемо основні компоненти, які слід налаштувати для досягнення бажаної фізичної поведінки об'єктів.

1. Компонент Rigidbody визначає фізичні властивості об'єкта, такі як маса, тертя і сила гравітації.

2. Компонент Collider визначає форму і розміри об'єкта, а також бере участь в обробці зіткнень. Unity надає кілька типів колайдерів, таких як Box Collider, Sphere Collider і Capsule Collider.

3. Компонент Joint дає змогу з'єднувати об'єкти разом і керувати їхньою фізичною взаємодією. Unity пропонує кілька типів з'єднань, таких як Fixed Joint, Hinge Joint і Configurable Joint.

4. Компонент Physics Material визначає фізичні властивості поверхні колайдера, як-от тертя та пружність.

5. Компонент Raycast дає змогу визначити, чи перетинає промінь колайдер або поверхню об'єкта. Unity надає можливість виконувати променеві операції не тільки в режимі редактора, а й під час гри.

Таким чином, в Unity можна застосовувати фізичні матеріали до об'єктів, щоб задати їм певні властивості та поведінку під час взаємодії з фізикою.

Фізичний матеріал являє собою набір параметрів, які визначають, як об'єкт поводитиметься під час зіткнень, тертя та інших фізичних явищ. Залежно від обраного матеріалу, об'єкт може бути слизьким, твердим, м'яким або мати інші властивості.

Залежно від налаштувань фізичного матеріалу, об'єкти можуть поводитися по-різному під час взаємодії з іншими об'єктами. Наприклад, якщо в об'єкта є слизький фізичний матеріал, то він буде зісковзувати по плоских поверхнях. Якщо в об'єкта є пружний матеріал, то він може відскакувати від інших об'єктів.

Ця властивість Use Gravity визначає, чи впливає гравітація на ваш ігровий об'єкт, чи ні. Якщо вона має значення false, то Rigidbody поводитиметься так, ніби перебуває у відкритому космосі. В інтерфейсі unity ви можете переміщати ваш об'єкт у заданій площині за допомогою клавіш зі

стрілками. Якщо гравітацію увімкнено, об'єкт впаде, щойно перетне межу площини. Якщо ж гравітацію вимкнено, то об'єкт продовжить свій шлях безперервно.

Властивість Mass використовується для визначення маси вашого об'єкта. За замовчуванням значення зчитується в кілограмах. Різні тверді тіла з великою різницею у масі можуть призвести до нестабільної роботи фізичної симуляції. Взаємодія між об'єктами різної маси відбувається так само, як і в реальному світі. Наприклад, під час зіткнення об'єкт з більшою масою більше штовхає об'єкт з меншою масою. Поширеною помилкою, яка панує у свідомості користувачів прокату, є те, що важкі об'єкти падають швидше, ніж легкі. У навколишньому світі це не так, швидкість падіння диктується факторами гравітації та опору.

Властивість Drag можна інтерпретувати як величину опору повітря, яка впливає на об'єкт при русі з боку сил. Коли значення лобового опору дорівнює 0, об'єкт не зазнає жодного опору і може рухатися вільно.

З іншого боку, якщо значення лобового опору дорівнює нескінченності, то рух об'єкта миттєво зупиняється. По суті, опір використовується для уповільнення об'єкта. Чим вище значення опору, тим повільнішим стає рух об'єкта.

Властивість Add Force використовується для додавання сили до Rigidbody. Сила завжди прикладається безперервно вздовж напрямку вектора сили. Крім того, вказавши режим сили, користувач може змінити тип сили на прискорення, зміну швидкості або імпульс.

Користувачі повинні пам'ятати, що силу можна застосувати лише до активного жорсткого тіла. Якщо GameObject неактивний, то AddForce на нього не вплине. Крім того, тіло також не повинно бути кінематично неактивним. Після застосування сили, стан Rigidbody за замовчуванням встановлюється на

пробудження. Величина опору впливає на швидкість руху об'єкта під дією сили.

Властивість кутовий опір Angular Drag можна інтерпретувати як величину опору повітря, що впливає на об'єкт при обертанні з моменту 0. Як і у випадку з Drag, встановлення значення Angular Drag рівним 0 усуває опір повітря і тут.

Однак, встановивши значення Angular Drag на нескінченність, ви не зможете зупинити обертання об'єкта. Кутовий опір має на меті лише сповільнити обертання об'єкта. Чим більше значення Angular Drag (Кутового перетягування), тим повільнішим стає обертання об'єкта.

Властивість Is Kinematic (Кінематичний). Коли властивість Is Kinematic (Кінематичний) увімкнено, об'єкт більше не керуватиметься фізичним рушієм. Сили, з'єднання або зіткнення перестануть впливати на Rigibody. У цьому стані ним можна маніпулювати лише за допомогою його трансформації.

Удар по об'єктах у стані Кінематичний не призводить до зміни їхнього стану, оскільки вони більше не можуть обмінюватися силами. Ця властивість стає в нагоді, коли ви хочете переміщати платформи або анімувати Rigidbody з приєднаним HingeJoint'ом.

Властивіст Інтерполяція (Interpolate). Користувачам рекомендується використовувати інтерполяцію у ситуаціях, коли потрібно синхронізувати графіку з фізикою GameObject'a.

Оскільки графіка єдності обчислюється у функції оновлення, а фізика - у функції фіксованого оновлення, іноді вони не синхронізуються. Щоб виправити це відставання, використовується інтерполяція (Interpolate).

Властивість Collision Detection використовується для запобігання проходженню об'єктів, що швидко рухаються, через інші об'єкти без виявлення зіткнень. Для найкращих результатів користувачам рекомендується встановити це значення у CollisionDetectionMode.ContinuousDynamic для об'єктів, що

швидко рухаються. Щодо інших об'єктів, з якими ці об'єкти мають зіштовхуватися, ви можете встановити для них значення CollisionDetectionMode.Continuous.

Властивість обмеження (Constraints) використовуються для накладання обмежень на рух твердого тіла. Вони визначають, які ступені свободи дозволені для симуляції жорсткого тіла. За замовчуванням встановлено значення RigibodyConstraints.None.

У Constraints є два режими - Freeze Position (Фіксоване положення) та Free Rotation (Вільне обертання). У той час як Freeze Position обмежує переміщення жорсткого тіла по осях X, Y та Z, Freeze Rotation обмежує їх обертання навколо цих осей.

```
using UnityEngine;
public class PhysicsController : MonoBehaviour
{
public float gravity = 9.8f;
public float friction = 0.1f;
private Rigidbody rb;
private void Start()
{
rb = GetComponent();
}
private void Update()
{
rb.AddForce(Vector3.down * gravity);
rb.AddForce(-rb.velocity.normalized * friction);
}
}
```

Крім того, в Unity можна створювати скрипти, які дають змогу керувати фізичними властивостями об'єктів, такими як гравітація, тертя, сила тощо. Ці скрипти можуть бути застосовані до будь-якого ігрового об'єкта і дають змогу створювати фізичні ефекти в грі.

У цьому прикладі до об'єкта застосовуватиметься сила гравітації, спрямована вниз, а також сила тертя, спрямована проти поточного напрямку руху об'єкта.

Щоб використовувати цей скрипт, необхідно додати його до ігрового об'єкта в Unity і налаштувати значення змінних "gravity" і "friction". Після цього об'єкт рухатиметься під впливом гравітації та тертя.

Таким чином, створення скриптів дає змогу додавати та налаштовувати різні фізичні властивості об'єктів у Unity, що відкриває широкі можливості для створення інтерактивної та реалістичної ігрової механіки.

3.2 Короткий опис програмної реалізації

Ми розробимо невелику повноцінну гру у форматі 2D. Сама гра буде аркадою, яка, як і звичайно для аркад, захоплює процесом гри, але при цьому сам процес буде досить простим.

Розробка починалася з чіткого формульовання концепції гри. Визначалися основні правила, цілі, механіка гри та особливості геймплею. Це включало в себе визначення типу кубів, розміру вежі, правил її побудови та правил перемоги. Перемоги в грі яктакової не існує. Це підтримує інтерес до аркади – створити якнайвищу вежу.

Опис програмної реалізації:

3.2.1 Rotate Camera

Змінні:

- **speed**: Визначає швидкість обертання.
- **_rotator**: Посилання на компонент Transform об'єкта, до якого прикріплений цей скрипт.

Метод Start:

- Отримує компонент Transform об'єкта, до якого прикріплений цей скрипт, і призначає його змінній **_rotator**.

Метод Update:

- Виконується кожен кадр.
- Обертає об'єкт навколо осі Y залежно від змінної **speed**, помноженої на **Time.deltaTime**.
- **Time.deltaTime** гарантує, що швидкість обертання залишається постійною незалежно від кадрової частоти.

Цей скрипт буде безперервно обертати об'єкт, до якого він прикріплений, навколо осі Y зі швидкістю, визначеною змінною **speed**. Цей скрипт призначений для обертання камери і він прикріплений до об'єкта камери у сцені Unity.

3.2.2 Canvas Buttons

Метод RestartGame перезавантажує поточну сцену, використовуючи SceneManager.LoadScene та SceneManager.GetActiveScene().buildIndex.

Коли цей метод викликається (наприклад, при натисканні кнопки на канвасі), він перезавантажить поточну сцену, використовуючи індекс будівлі, що дозволить почати гру спочатку.

Це корисний метод для ігор або додатків, де потрібно мати можливість перезавантаження гри без виходу з неї.

3.2.3 CubeToCreate

Ці методи використовуються для ініціалізації ресурсів перед початком гри (у випадку **Start**) та для оновлення чогось кожен кадр (у випадку **Update**).

3.2.4 Explode Cubes

Скрипт містить метод **OnCollisionEnter**, який спрацьовує при зіткненні об'єкта, до якого прикріплений цей скрипт, з іншим об'єктом у сцені.

Отже, коли об'єкт з тегом "Cube" зіткнеться з об'єктом, до якого прикріплений цей скрипт, буде виконано наступні дії:

1. Усі дочірні об'єкти знайденого "куба" отримують компонент Rigidbody, тоді вони отримують силу вибуху та відокремлюються від батьківського об'єкту.
2. Кнопка перезавантаження (**restartButton**) стає активною.
3. Позиція головної камери зсувається назад уздовж вектора Z на 3 одиниці.
4. Знищується об'єкт, з яким відбулося зіткнення.
5. Змінна **_collisionSet** встановлюється в **true**, щоб уникнути подальших зіткнень.

3.2.5 GameController

Скрипт, **GameController**, відповідає за керування різними аспектами гри. Розглянемо деякі ключові елементи цього коду:

1. **Розміщення кубів:**
    - Є змінна **cubeToPlace**, яка вказує на позицію, де буде створюватися новий куб.
    - Метод **SpawnPositions** перевіряє можливі позиції для розміщення нового куба, враховуючи, що вони не зайняті іншими кубами або не знаходяться поза межами гри.
2. **Керування камерою:**
    - Позиція камери змінюється в методі **MoveCamChangeBg** в залежності від розміщення кубів у грі.
    - Також змінюється колір фону камери в залежності від розташування кубів.
3. **Управління грою:**
    - Є логіка для перевірки поразки гравця, що спрацьовує, якщо куби рухаються занадто швидко.

- Відбувається створення першого куба, яке видаляє початковий екран.

4. **Інші елементи:**
    - Використовується структура **CubePos**, що зберігає координати куба в просторі.

3.3 Тестування

Після створення прототипу проводилися тестування гри. Це дозволило виявити можливі помилки, баланс гри та отримати фідбек від гравців для покращення геймплею. Наступні рисунки відображають кожен етап тестування гри.

На рис. 3.1 подано розташування елементів після встановлення основи (білий колір частини) та перших ходів двох гравців (синій та червоний колір частин)

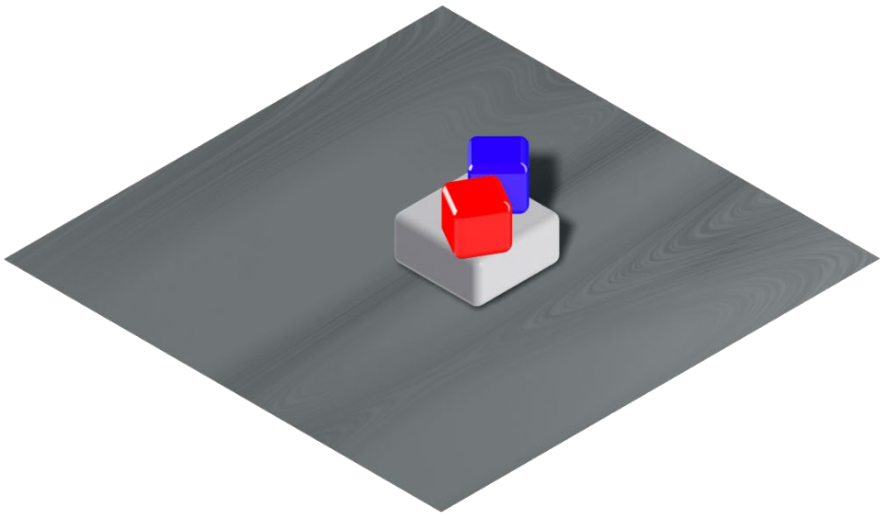

Рисунок 3.1 - Початок гри

На рис. 3.2 подано розташування елементів після першого і другого ходів всіх трьох гравців (синій, червоний та зелений колір частин).

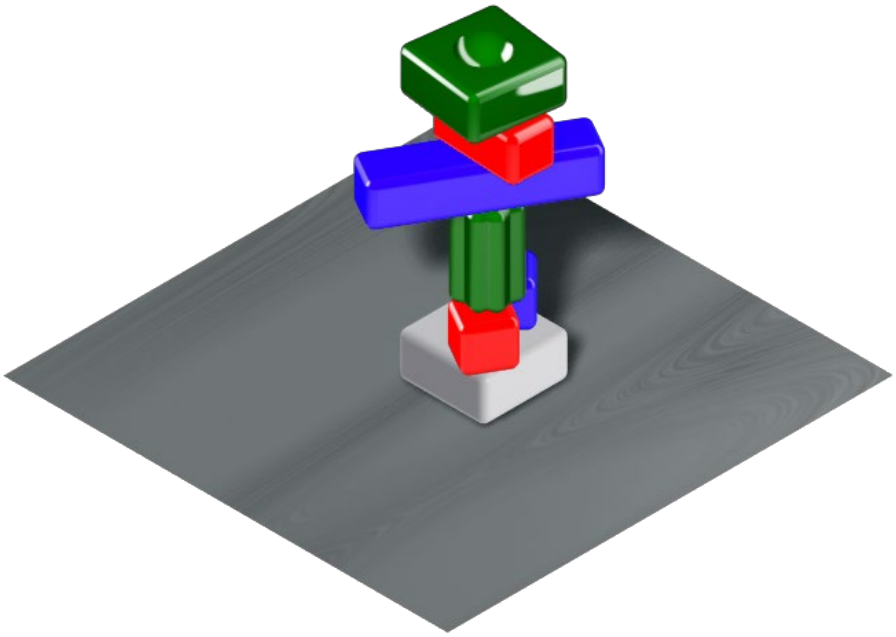

Рисунок 3.3 – Встановлено 7 частин

На рис. 3.4 подано розташування елементів після вдалого ходу «синього» гравця і ризикованого ходу «червоного».

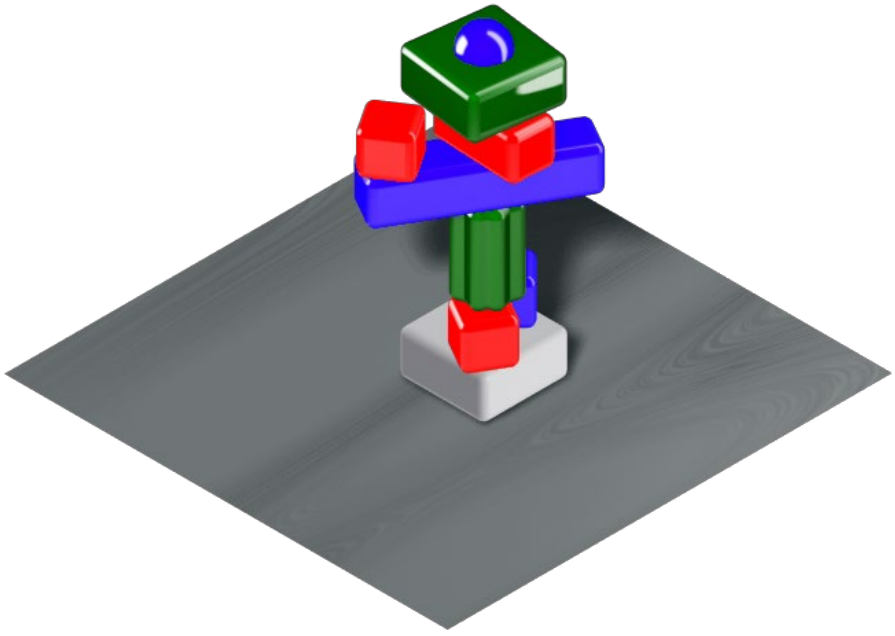

Рисунок 3.4 – Встановлено 9 частин

На рис. 3.5 подано розташування елементів після невдалого «червоного» гравця.

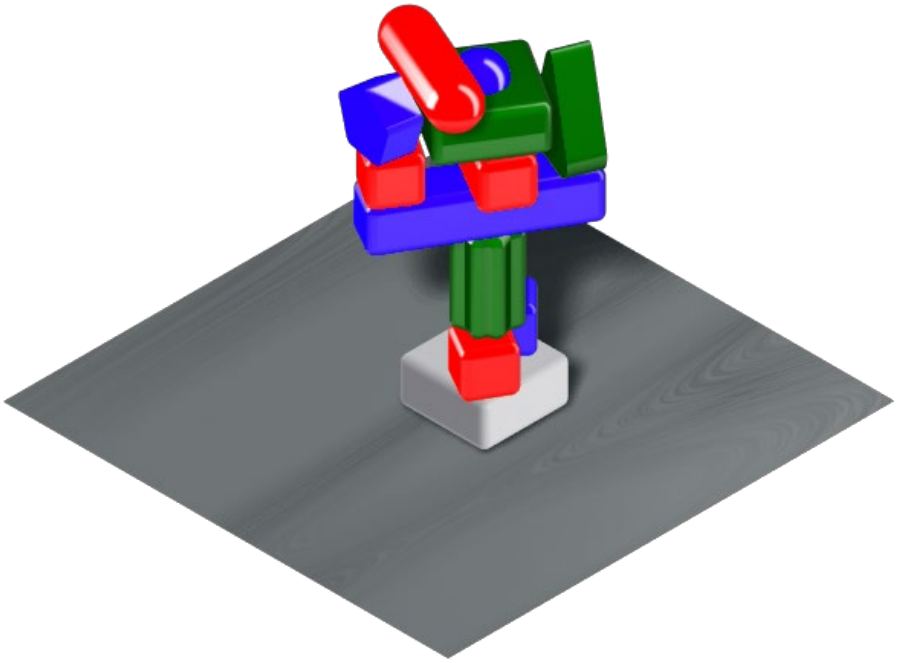

Рисунок 3.5 – Встановлено 12 частин

На рис. 3.6 подано результат дії гравітації на сформовану вежу, останній (червоний) елемент якої було розташовано не правильно.

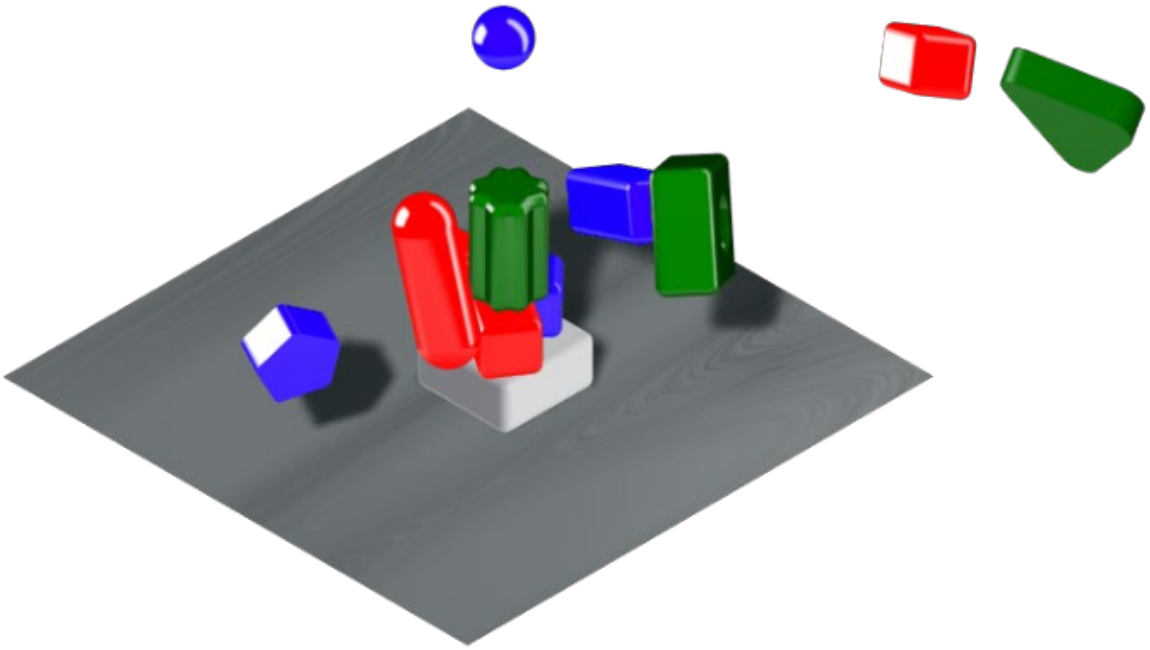

Рисунок 3.3 – Кінець гри

Таким чином, результат тестування показав коректність реалізації логічного контенту гри та відповідність цифрових моделей частин вежі їх фізичним аналогам.

ВИСНОВКИ

В кваліфікаційній роботі було розв’язано практичної задачі з розробки комплексу інформаційного та програмного забезпечення гібридної ігри. Для досягнення поставленої виконано такі основні завдання роботи:

1) Проведено аналіз існуючої фізичної гри і визначення основних фізичних компонентів, які можна подати в цифровому вигляді.

2) Розроблено концептуальну схему ігрової системи.

3) Розроблено цифрові моделі фізичних компонентів з урахуванням особливостей їх застосування в ігровому процесі.

4) Розроблено логічний контент взаємодії компонентів в ігровому середовищі

5) Визначено набір інструментів взаємодії ігрової системи з гравцями в режимі введення керуючих дії і виведення їх результатів.

6) Програмно реалізовано ігрову систему та проведено перевірка її працездатності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. BoardGameGeek is an online resource and community that aims to be the definitive source for board game and card game.- content.https://boardgamegeek.com/browse/boardgamecategory
2. Board Games .- https://www.amazon.com/Board-Games/b?ie=UTF8&node=166225011
3. Slack J. The Board Game Designer's Guide to Careers in the Industry. - CRC Press, 2023. – 124 p.
4. Daniels Jesse Terrance Make Your Own Board Game: Designing, Building, and Playing an Original Tabletop Game. - Storey Publishing, LLC, 2022.
5. Hinebaugh Jeffrey P. More Board Game Education: Inspiring Students Through Board Games. - Rowman & Littlefield Publishers, 2019.
6. Braun Anna, Rizzo Raffael. XR Development with Unity: A beginner's guide to creating virtual, augmented, and mixed reality experiences using Unity .- Packt Publishing, 2023. - 284 p
7. Brusca Victor G. Advanced Unity Game Development: Build Professional Games with Unity, C#, and Visual Studio .- Apress Media LLC, 2022. -363 p.
8. Buttfield-Addison Paris, Manning Jon, Nugent Tim. Unity Development Cookbook: Real-Time Solutions from Game Development to AI .- 2nd Edition. - O’Reilly Media, Inc., 2023. - 430 p.
9. Rawat Puneet Singh. Hands-On Unity Application Development: Unlock the power of Unity3D for non-gaming applications .- BPB Publications, 2023. - 294 p.
10. Sung K., Smith G. Basic Math for Game Development with Unity 3D: A Beginner's Guide to Mathematical Foundations .- 2nd Edition. - Apress, 2023. - 456 p.

11. Tykoski S. Mastering Game Design with Unity 2021: Immersive Workflows, Visual Scripting, Physics Engine, GameObjects, Player Progression, Publishing, and a Lot More .- BPB Publications, 2023. - 394 p.
12. Borromeo N.A. Hands-On Unity 2022 Game Development: Learn to use the latest Unity 2022 features to create your first video game in the simplest way possible .- 3rd ed. - Birmingham: Packt Publishing, 2022. - 712 p.
13. Chen J. Game Development with Unity for .NET Developers .- Packt Publishing, 2022. - 584 p.
14. Davis A., Baptiste T., Craig R. Unity 3D Game Development .- Packt, 2022. - 370 p
15. Lanzinger F. 3D Game Development with Unity .- CRC Press, 2022. - 415p
16. Alam Asadullah. Unity: Beginner to Advanced - Complete Course: Master Video Game Development from the Ground Up Using Unity and C# . - Independently published, 2023. - 249 p.
17. Alda Álvaro. Beginner's Guide to Unity Shader Graph: Create Immersive Game Worlds Using Unity's Shader Tool.- Apress Media LLC, 2023. - 460 p

ДОДАТОК

Scripts:

1. CanvasButtons // рестарт гри
2. cubeToCreate //створення нової гри та оновлення ігрового поля
3. ExplodeCubes //умови та ігролад при програші
4. GameController //головний код програми (ігролад)
5. RotateCamera //обертання камери

Canvas Buttons

```
using UnityEngine.SceneManagement;
using UnityEngine;
using UnityEditor.SearchService;

public class CanvasButtons : MonoBehaviour
{
    public void RestartGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
}
```

cubeToCreate

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

public class cubeToCreate : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

ExplodeCubes

```
using UnityEngine;

public class NewBehaviourScript : MonoBehaviour
{
    public GameObject restartButton;
    private bool _collisionSet;
    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag == "Cube" && !_collisionSet)
        {
            for (int i = collision.gameObject.transform.childCount - 1; i>=0;
i--)
            {
                Transform child = collision.gameObject.transform.GetChild(i);
                child.gameObject.AddComponent<Rigidbody>();

child.gameObject.GetComponent<Rigidbody>().AddExplosionForce(70f, Vector3.up,
5f);
                child.SetParent(null);
            }
            restartButton.SetActive(true);
            Camera.main.gameObject.transform.position -= new Vector3(0, 0,
3);
            Destroy(collision.gameObject);
            _collisionSet = true;
        }
    }
}
```

GameController

```
using System;
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.EventSystems;
using UnityEngine.Rendering.PostProcessing;
using Unity.VisualScripting;

public class GameController : MonoBehaviour
{
   private CubePos nowCube = new CubePos(0, 1, 0);
   public float cubeChangePlaceSpeed = 0.5f;
   public Transform cubeToPlace;

    public GameObject cubeToCreate, allCubes;
    private Rigidbody allCubesRb;

    public Color[] bgColors;
    private Color toCameraColor;

    private bool isLoose, firstCube;
    public GameObject[] canvasStartPage;
    private float camMoveToYPos, camMoveSpeed = 2f;

    private List<Vector3> allCubesPositions = new List<Vector3>
    {
        new Vector3(0, 0, 0),
        new Vector3(1, 0, 0),
        new Vector3(-1, 0, 0),
        new Vector3(0, 1, 0),
        new Vector3(0, 0, 1),
        new Vector3(0, 0, -1),
        new Vector3(1, 0, 1),
```

```
        new Vector3(-1, 0, -1),
        new Vector3(-1, 0, 1),
        new Vector3(1, 0, -1),
    };
    private int precCountMaxHoryzontal;
    private Transform mainCamera;
    private Coroutine showCubePlace;


    private void Start()
    {
        toCameraColor = Camera.main.backgroundColor;
        mainCamera = Camera.main.transform;
        camMoveToYPos = 5.9f + nowCube.y - 1f;


        allCubesRb = allCubes.GetComponent<Rigidbody>();
        showCubePlace = StartCoroutine(ShowCubePlace());

    }

    private void Update()
    {



        if (Input.GetMouseButtonDown(0) && cubeToPlace != null &&
allCubes!=null && !EventSystem.current.IsPointerOverGameObject())

        {
            if (!firstCube)
            {
                firstCube = true;
                foreach (GameObject obj in canvasStartPage)
                {
```

```
                    Destroy(obj);
                }
            }
            GameObject newCube =  Instantiate(
                cubeToCreate,
                cubeToPlace.position,
                Quaternion.identity) as GameObject;
            newCube.transform.SetParent(allCubes.transform);
            nowCube.setVector(cubeToPlace.position);
            allCubesPositions.Add(nowCube.getVector());

            allCubesRb.isKinematic = true;
            allCubesRb.isKinematic = false;
            SpawnPositions();
            MoveCamChangeBg();
        }

        if (!isLoose && allCubesRb.velocity.magnitude > 0.1f)
        {
            Destroy(cubeToPlace.gameObject);
            isLoose = true;
            StopCoroutine(showCubePlace);
        }
        mainCamera.localPosition =
Vector3.MoveTowards(mainCamera.localPosition, new
Vector3(mainCamera.localPosition.x, camMoveToYPos,
mainCamera.localPosition.z), camMoveSpeed* Time.deltaTime);

        if(Camera.main.backgroundColor != toCameraColor)
            Camera.main.backgroundColor =
Color.Lerp(Camera.main.backgroundColor, toCameraColor, Time.deltaTime/1.5f);
    }



    IEnumerator ShowCubePlace()
```

```
    {
        while (true)
        {
            SpawnPositions();

            yield return new WaitForSeconds(cubeChangePlaceSpeed);
        }
    }

    private void SpawnPositions()
    {
        List<Vector3> positions = new List<Vector3>();
        if (IsPositionEmpty(new Vector3(nowCube.x + 1, nowCube.y, nowCube.z))
            && nowCube.x + 1 !=cubeToPlace.position.x)
            positions.Add(new Vector3(nowCube.x + 1, nowCube.y, nowCube.z));
        if (IsPositionEmpty(new Vector3(nowCube.x - 1, nowCube.y, nowCube.z))
            && nowCube.x - 1 != cubeToPlace.position.x)
            positions.Add(new Vector3(nowCube.x - 1, nowCube.y, nowCube.z));
        if (IsPositionEmpty(new Vector3(nowCube.x, nowCube.y + 1, nowCube.z))
            && nowCube.y + 1 != cubeToPlace.position.y)
            positions.Add(new Vector3(nowCube.x, nowCube.y + 1, nowCube.z));
        if (IsPositionEmpty(new Vector3(nowCube.x, nowCube.y - 1, nowCube.z))
            && nowCube.y - 1 != cubeToPlace.position.y)
            positions.Add(new Vector3(nowCube.x, nowCube.y - 1, nowCube.z));
        if (IsPositionEmpty(new Vector3(nowCube.x, nowCube.y, nowCube.z + 1))
            && nowCube.z + 1 != cubeToPlace.position.z)
            positions.Add(new Vector3(nowCube.x, nowCube.y, nowCube.z + 1));
        if (IsPositionEmpty(new Vector3(nowCube.x, nowCube.y, nowCube.z - 1))
            && nowCube.z - 1 != cubeToPlace.position.z)
            positions.Add(new Vector3(nowCube.x, nowCube.y, nowCube.z - 1));

        if (positions.Count > 1)
            cubeToPlace.position = positions[UnityEngine.Random.Range(0,
positions.Count)];
        else if (positions.Count ==0)
```

```
            isLoose = true;
        else
            cubeToPlace.position = positions[0];

    }

    private bool IsPositionEmpty(Vector3 targetPos)
    {
        if(targetPos.y ==0)
            return false;
        foreach(Vector3 pos in allCubesPositions)
        {
            if (pos.x == targetPos.x && pos.y == targetPos.y && pos.z ==
targetPos.z)
                return false;
        }
        return true;
    }

    private void MoveCamChangeBg()
    {
        int maxX = 0, maxY = 0, maxZ = 0, maxHoryzontal;

        foreach (Vector3 pos in allCubesPositions)
        {
            if (Mathf.Abs(Convert.ToInt32(pos.x)) > maxX)
                maxX = Convert.ToInt32(pos.x);
            if (Convert.ToInt32(pos.y) > maxY)
                maxY = Convert.ToInt32(pos.y);
            if (Mathf.Abs(Convert.ToInt32(pos.z)) > maxZ)
                maxZ = Convert.ToInt32(pos.z);
        }
        camMoveToYPos = 5.9f + nowCube.y - 1f;
        maxHoryzontal = maxX > maxZ ? maxX : maxZ;
        if (maxHoryzontal %3 == 0 && precCountMaxHoryzontal != maxHoryzontal)
```

```
        {
            mainCamera.localPosition -= new Vector3(0, 0, 2f);
            precCountMaxHoryzontal = maxHoryzontal;
        }
        if(maxY >=5)
            toCameraColor = bgColors[1];
        else if (maxY >= 4)
            toCameraColor = bgColors[0];
        else if (maxY >= 2)
            toCameraColor = bgColors[2];
    }
}

struct CubePos
{
    public int x, y, z;

    public CubePos(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public Vector3 getVector()
    {
        return new Vector3(x, y, z);
    }
    public void setVector(Vector3 pos) {
        x = Convert.ToInt32 (pos.x);
        y = Convert.ToInt32 (pos.y);
        z = Convert.ToInt32 (pos.z);
    }
}
```

RotateCamera

```
using UnityEngine;

public class RotateCamera : MonoBehaviour
{
    public float speed = 5f;
    private Transform _rotator;

    private void Start()
    {
        _rotator = GetComponent<Transform>();
    }

    private void Update()
    {
        _rotator.Rotate(0, speed*Time.deltaTime, 0);

    }
}
```