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 на Unity»

Виконав _____
(Підпис)
Майсак Михайло Костянтинович
(прізвище, ім'я, по батькові)

Керівник Сайко Володимир Григорович
(прізвище, ім'я, по батькові)

_____
(Резолюція «До захисту»)

Попередній захист:

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

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

Київ – 2022

Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра прикладних інформаційних систем

Назва теми: «Комп'ютерна гра на Unity»

Освітня програма: Прикладне програмування

Спеціальність: Комп'ютерні науки

ПІБ	Підпис
Майсак Михайло Костянтинович	

Назва роботи українською та англійською мовами

Комп'ютерна гра на Unity Computer game developed on the Unity game engine

Мета кваліфікаційної роботи бакалавра, завдання

Мета кваліфікаційної роботи бакалавра: Розроблення комп'ютерної гри на рушії Unity для підвищення ефективності технічного дизайну.

План роботи:

Підходи до дизайну ігор. Особливості технічного дизайну ігор.

Аналіз інструментів для розробки ігор. Проектування програмної логіки гри.

Реалізація комп'ютерної гри на рушії Unity.

ПІБ, ступінь, звання наукового керівника роботи:

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ БАКАЛАВРА

№з/п	Назва етапів кваліфікаційної роботи бакалавра	Термін виконання етапів кваліфікаційної роботи бакалавра	Відмітка про виконання
1.	Вибір теми та наукового керівника кваліфікаційної роботи бакалавра	09.10.2021	виконано
2.	Видача завдання кваліфікаційної роботи бакалавра	19.10.2021	виконано
3.	Настановча групова співбесіда з питань кваліфікаційної роботи бакалавра	21.10.2021	виконано
4.	Затвердження плану кваліфікаційної роботи бакалавра	25.10.2022	виконано
5.	Підбір та вивчення літературних та інших джерел з теми дослідження	01.11.2022	виконано
6.	Підготовка і подання науковому керівнику першого варіанту І розділу роботи	21.12.2022	виконано
7.	Підготовка і подання науковому керівнику першого варіанту ІІ розділу роботи	31.01.2022	виконано
8.	Підготовка і подання науковому керівнику першого варіанту ІІІ розділу роботи	30.03.2022	виконано
9.	Подання роботи у першому варіанті	28.04.2022	виконано
10.	Оформлення пояснювальної записки кваліфікаційної роботи бакалавра	03.05.2022	виконано
11.	Подання кваліфікаційної роботи бакалавра на попередній захист	**23.05.2022**	виконано
12.	Врахування зауважень керівника і подання роботи в остаточному варіанті (з відповідним висновком про допуск) на кафедру	27.05.2022	виконано
13.	Затвердження роботи в цілому (підготовка письмового відгуку керівника, письмова рецензія на бакалаврської роботу)	10.06.2022	виконано
14.	Захист кваліфікаційної роботи бакалавра	22.06.2022 23.06.2022 24.06.2022	виконано

Здобувач вищої освіти_____ Керівник _____

ВІДОМІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ БАКАЛАВРА

Складові частини дипломної роботи	Обсяг, арк.
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	1
Вступ	3
1	20
2	20
3	6
Висновки	1
Перелік використаних джерел	3
Додатки	45

<table>
<tr><td></td><td></td><td></td><td></td><td colspan="3" rowspan="2">ДП ХХХХ 00.000.00</td></tr>
<tr><td></td><td>ПІБ</td><td>Підп.</td><td>Дата</td></tr>
<tr><td>Розроб н.</td><td>Майсак М.К.</td><td></td><td></td><td rowspan="4">Відомість дипломної роботи</td><td>Лист</td><td>Листів</td></tr>
<tr><td>Керівн .</td><td>Сайко В.Г.</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Н/конт р.</td><td>Базилюк А.М.</td><td rowspan="2"></td><td></td><td colspan="2" rowspan="2"></td></tr>
<tr><td>Зав.ка ф.</td><td>Плескач В.Л.</td><td></td></tr>
</table>

АНОТАЦІЯ

Дипломна робота: 105 с., 13 рис., 3 табл., 34 джерела, 4 додатки.

Ця дипломна робота присвячена проектуванню та розробці комп'ютерної гри на рушії Unity.

Метою дипломної роботи є розроблення комп'ютерної гри на рушії Unity для підвищення ефективності технічного дизайну.

Завдання дослідження:

Аналіз та систематизація підходів до технічного дизайну ігор та розробка власної гри на рушії Unity.

Для досягнення поставленої мети треба вирішити такі **завдання**:

- Проаналізувати та систематизувати теоретичні засади та технічні підходи для розробки ігор
- Здійснити аналіз програмно-технологічних рішень для розробки ігор.
- Спроектувати та реалізувати комп'ютерну гру відповідно до поставлених вимог та з урахуванням попередніх досліджень.

Об'єкт дослідження.

Процеси створення комп'ютерних ігор.

Предмет дослідження.

Програмно-технічні засади підвищення ефективності технічного дизайну та розробки ігор на Unity.

Методи дослідження.

UML-моделювання, аналогія залучені в процесі проектування та розробки власної гри, метод порівняння, що застосовано для аналізу наявних ресурсів та технічних рішень для розробки ігор, метод індукції та синтезу для виділення власних підходів до вирішення поширених завдань розробки комп'ютерної гри. Описовий метод використано при описі алгоритмів, шляхів розробки ігрової логіки і основних інструментів для розробки.

Ключові слова: відеогра, Unity, C#, Геймдев.

ABSTRACT

Thesis: 105 pages, 13 figures, 3 tables, 34 sources, 4 appendices.

This thesis is devoted to the design and development of a computer game with Unity Engine.

The purpose of the thesis is to develop a computer game on the Unity engine in order to increase the efficiency of technical design.

To achieve this goal you need to solve the following **tasks**:

- To analyze and systematize theoretical foundations and technical approaches for game development
- To carry out the analysis of software and technology solutions for game development.
- To design and implement a computer game in accordance with the requirements and taking into account previous research.

Object of study.

Game development processes.

Subject of study.

Software and hardware principles to increase the efficiency of technical design and game development at Unity.

Research methods.

UML modeling, analogy involved in the design and development of my game, comparison method used to analyze available resources and technical solutions for game development, induction and synthesis method used to identify own approaches for solving common problems of game development. The descriptive method was used to describe algorithms, ways to develop game logic and the main game development tools.

Keywords: videogame, Unity, C#, Gamedev.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

API – Applied programming interface;

VFX – Visual effects;

FPS – frames per second.

OOP – Object-oriented programming

ECS – Entity Component System

FRP – Functional Reactive Programming

CCD – Continuous Collision Detection

K-D Tree – K-Dimensional Tree

Відмовостійкість (fault tolerance) – здатність програмної системи продовжувати коректну роботу попри виникнення помилок у її окремих блоках або суміжних системах.

Ігровий ассет (game asset) – цифровий об’єкт, набір однотипних даних, який є повністю або частково самостійним відображенням певного ігрового контенту.

ВСТУП

Перша комп’ютерна гра Nimatron була розроблена і запущена на електронному пристрої ще у 1940 році. Новий проект швидко захопив увагу аудиторії на виставці, тож у новоутворену сферу почали інвестувати. Вже у 1952 році Александром Дугласом були написані хрестики-нулики для комп'ютера IBM, а у 1958му Вільям Гігінботам продемонстрував власну відеогру в теніс. З цього моменту почала розвиватись нова гілка у сфері створення програмних продуктів – розробка комп’ютерних відеоігор. Новоутворена сфера почала швидко розвиватись та позичати багато моделей та принципів із сектору розробки звичайного програмного забезпечення. Проте з часом почали відокремлюватись особливі риси ігрової розробки, і відповідно певні підходи ставали більш ефективними за своїх конкурентів. Більше того, деякі підходи, які не вважались популярними для використання у процесах розробки програмного забезпечення, знайшли нове життя у розробці відеоігор.

Актуальність дослідження: Протягом останніх 40 років відеоігри займають важливе місце у секторі розваг та освіти. І відповідно база знань технічного характеру отримала дуже стрімкий розвиток. Починає зароджуватись зворотна тенденція – спроби інтегрувати деякі методології розробки комп’ютерних ігор у розробку ПЗ. Створення ігрового продукту за означенням передбачає велику динаміку змін, високу складність завдань та проблем, комплексну кодову базу. Методології, вкарбовані у таких умовах, повинні бути дослідженими на глибокому рівні.

Мета дослідження: Розроблення комп’ютерної гри на рушії Unity для підвищення ефективності технічного дизайну.

Завдання дослідження:

- Дослідити методи технічного дизайну відеоігор.
    - Дослідити основні парадигми програмування що використовуються у розробці відеоігор.
    - Визначити окремі поширені завдання що виникають під час

розробки відеоігор та дослідити найефективніші підходи для їх вирішення.

- Дослідити інструментарій для створення ігор.
  - Вивчити необхідні базові інструменти, які використовуються при розробці ігор.
  - Дослідити можливості різних ігрових рушіїв.
  - Визначити основні особливості роботи на Unity. В тому числі і основні небезпеки та недоліки, які потрібно мати на увазі при розробці гри.
- Спроектувати основні елементи гри.
  - Визначити концепцію гри.
  - Спроектувати правила гри, геймплейні цикли та ін.
- Реалізувати гру.
  - Спроектувати основну програмну логіку гри.
  - Написати основні механіки, контроллери та ін.
  - Інтегрувати необхідний візуал у гру.
  - Створити необхідні VFX.
  - Реалізувати UI.
  - Завершити об’єднання всіх компонентів гри.
  - Протестувати ігровий проект у середовищі рушія та виправити критичні несправності.
  - Зібрати проект під цільові платформи
  - Протестувати проект на target платформах та виправити критичні несправності

Об’єкт дослідження: Процеси створення комп’ютерних ігор.

Предмет дослідження: Програмно-технічні засади підвищення ефективності технічного дизайну та розробки ігор на Unity.

Методи дослідження: Для реалізації поставлених задач та для формування звіту цього процесу, були використані наступні методи:

- Метод аналізу та декомпозиції використано при розгляді технічного дизайну інших відеоігор.
- Метод індукції та синтезу при виділенні власних підходів до вирішення поширених завдань розробки комп’ютерної гри.
- Метод наукового моделювання використано при формуванні схем геймплейних циклів, діаграм архітектури класів та схем UI.
- Описовий метод використано при описі алгоритмів та шляхів розробки ігрової логіки, описі основних інструментів для розробки.

Практичне значення одержаних результатів: Вклад у систематизацію знань у сфері технічного дизайну відеоігор та демонстрація практичного використання систематизованих методів. Підвищення ефективності створення відеоігор, вирішення поширених складних завдань.

Структура роботи:

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, розподілених на підрозділи та висновок.

РОЗДІЛ 1 ТЕОРЕТИЧНІ ЗАСАДИ ТА ТЕХНІЧНІ ПІДХОДИ ДЛЯ РОЗРОБКИ ВІДЕОІГОР

1.1 Використання моделей та парадигм програмування у розробці відеоігор

Комп’ютерна гра в загальній площині не відрізняється від будь якого іншого програмного продукту, тому можна констатувати, що при розробці ігор використовуються ті ж самі загально прийняті моделі та парадигми програмування. З іншого боку, на відміну від більшості проектів написаних для сучасних комп’ютерів, програмний устрій відеоігор ускладнений високим рівнем контролю в реальному часі від користувача, великою кількістю складних математичних симуляцій та високим ступенем розгалуженості системи. Тому за всю 70-ти річну історію комп’ютерних ігор природнім шляхом виокремились найбільш оптимальні моделі та парадигми для побудови таких комплексних програм. Розглянемо кожну більш докладно.

Object Oriented Programming

Найбільш популярна модель побудови будь яких програмних продуктів. Вона добре підходить для створення проектів високої складності та великого обсягу, зокрема комп’ютерних ігор. Організація кодової бази навколо об’єктів у вигляді набору даних та методів які їх опрацьовують, надає велику кількість переваг, а саме:

- Відсутність дублікації.

  Можливість повторного використання структур даних та методів роботи з ними.

- Високий ступінь захищеності даних і в результаті – низький ризик порушення їх структурної та логічної цілісності.

- Здатність до розширення системи.

  Нові об’єкти можуть взаємодіяти з інтерфейсом інших об’єктів.

- Напрямки розширення системи можуть обмежуватись розробником.

Це підвищує ефективність командної розробки, адже будь який наступний розробник буде пристосовувати свій код під обмеження попередньо створеної вищої сутності і таким чином зазвичай вся команда рухається в одному напрямі.

- Довга історія розвитку та значне розповсюдження.

  За останні 60 років під егідою ООП народилась величезна кількість парадигм, принципів та патернів проектування. Ці стандарти значно спрощують створення програмного застосунку з кодовою базою високої якості.

Entity Component System

На відміну від об’єктно орієнтованого підходу ця модель програмування не так відома на загальному ринку. Проте ECS підхід посів дуже важливе місце у ігровій розробці. В основі цієї моделі лежить Компонент (Component), набір компонентів представляє собою Сутність (Entity), сутностями та компонентами оперують Системи (System). Така модель дозволяє декомпозувати ігрову логіку на дрібні самодостатні блоки. Типовими техніками в ECS є створення невеликих струкутр даних, які відображають певну властивість сутності, написання модулярних систем для підтримки окремих функцій проекту та динамічна зміна сутностей під час роботи програми. Аби визначити чому саме ECS так добре підходить для розробки ігрових проектів, розглянемо основні переваги цієї моделі:

- Гарантія низької зв’язності окремих систем.
- Високий ступінь декомпозиції логіки проекту на рівні даних
- Гнучкість.

  Можна вільно додавати та видаляти системи, компоненти та сутності, надто не переймаючись про залежності.

- Розміщення сутностей у пам’яті.

ECS вважається моделлю з data-driven підходом. Модулярна організація даних, єдине сховище та швидкий доступ до них в ECS значно оптимізує використання оперативної пам'яті.

- Схильність до паралельних обчислень.

  Низька зв'язність та швидкий і простий для розуміння потік даних робить ECS підхід майже ідеальним для паралельного виконання багатьох завдань.

- Висока сумісність з іншими методологіями. В ECS проекти легко впроваджуються інші техніки, такі як Dependency Injection, Inversion of Control або глобальний Command Pattern.

Functional Reactive Programming

Ще одна досить вживана у розробці ігор парадигма програмування. Вона представляє собою поєднання функціонального та реактивного програмування. В основі лежать характерні риси обох парадигм: «чисті функції», lazy data flow, event stream та referencial trancparency. Системи в реактивному програмуванні повинні наслідувати наступні принципи:

- Керованість за допомогою інструкцій та запитів.

  Система повність працює з потоком інструкцій (нерідко асинхронним) у вигляді event stream або прямих запитів.

- Відгук у реальному часі.

  Система повинна швидко реагувати на запити.

- Стійкість.

  Система повинна залишатись активною та зберігати здатність обробляти запити навідь після виникнення помилок.

- Гнучкість.

Система повинна мати здатність до розширення та перегрузок і при цьому зберігати можливість обробляти запити.

Хоча ці критерії виглядають досить природніми для будь якого сервісу, написати систему на реактивній моделі програмування досить складно. Кожен вузол обробки запитів такої системи повинен бути асинхронним неблокованим модулем з високою відмовостійкістю. При розробці ігор реактивний підхід активно використовується для створення комплексних графічних інтерфейсів. Проаналізуємо унікальні риси цієї моделі, аби визначити причини використання реактивного програмування у іграх з UI системами великого обсягу:

- Представлення даних у вигляді «потоків», тобто повідомлень що розміщені у площині часу.

  Такий підхід дозволяє не очікувати увесь необхідний обсяг вхідних даних, а працювати з тими даними що вже надійшли, додатково очікуючи нових.

- Чисті функції. Immutability (незмінність).

  Чисті функції – це функції, результат виконання котрих залежить лише від вхідних параметрів, а не від стану системи. Такі функції не створюють так званих «побічних ефектів», тобто додаткових змін у загальній системі. Це значно знижує ризик виникнення помилок та критичних несправностей у роботі програми. Реактивні функції (map, filter, scan) не змінюють сам потік даних, а створюють новий на основі зразка, додаючи відповідні властивості або змінюючи структуру упорядкування потоку даних.

- Push-pull propagation. Властивість Backpressure.

  Коли змінюється стан певної сутності, повідомлення про цю подію розповсюджується по всій системі за допомогою потоків (reactive streams). Pull алгоритми визначають активного споживача, який

постійно перевіряє, чи джерело готове віддати оновлені дані. Push алгоритми визначають пасивного споживача, який отримує оновлені дані від джерела, як тільки джерело готове до цього. Push-pull алгоритми є гібридною концепцією, в якій споживач отримує коротке повідомлення від джерела про те що стан системи змінився, і після цього робить запит до джерела на отримання даних. Таким чином реалізується так звана властивість backpressure, коли джерело надає інформацію тільки в тому випадку, коли споживач готовий її отримати та обробити. Це досить ефективний спосіб комунікації між окремими системами та організації data flow.

- Використання ефективних структур даних.

  Архітектура системи, що використовує модель реактивного програмування, часто організована у вигляді дерев або графів. Таким чином можна значно підвищити зручність подальшого розширення системи, структурованість та ефективність алгоритмів розповсюдження даних по системі. У поєднанні чистими функціями, ця властивість стаж ще більш потужною, адже дозволяє реактивним системам наблизитись до швидкості виконання систем з імперативною моделлю. Використовуючи властивості чистих функцій, на етапі компіляції окремі блоки програми замінюються на конкретні значення, таким чином розпаковуючи дерево у чергу прямих інструкцій.

- Асинхронна складова.

  Системи на реактивній моделі завжди рекомендується створювати з підтримкою асинхронного виконання операцій. В самому підході для цього закладене необхідна незалежність між потоками даних, розподіл їх у часі, неблокованість потоку та захищеність від змін.

- Реактивні розширення.

Модель реактивного програмування може використовуватись багатьма способами. Є чисте використання, яке вважається загалом не доцільним при ігровій розробці, через складність побудови і підтримки чистих реактивних систем. Тому великий обсяг використання реактивної моделі припадає на сектор розширень для інших моделей програмування. Загалом такі розширення позначаються кодовим постфіксом Rx. Існує безліч різних фреймворків, таких як UniRx, RxJava, які дозволяють використовувати асинхронні підходи основані на реактивних потоках даних разом з ООР або ECS. Також у поєднанні з реактивними розширеннями добре працює паттерн Observer (адже він є фактичною аналогією реактивних ідей в ООР моделі) та робота з даними через запити.

В результаті проведених досліджень стає зрозуміло, що не можна ставити на меті пошук найкращої моделі програмування ігрового застосунку задля її подальшого використання у проектуванні всього додатку. Важливим є предметне порівняння кожного аспекту різних моделей та парадигм, та виявлення їх сильних сторін. В результаті найефективнішим сучасним підходом для розробки ігрового проекту є застосування мультипарадигменного програмування. Після власного власного аналізу найкращих моделей у ігровій розробці можна виокремити наступні висновки:

- Об’єктно-орієнтований підхід варто застосовувати для створення центральної архітектурної ланки, високого рівня абстракцій. Ця модель має грану сумісність з будь якими іншими парадигмами, та забезпечить стійкий каркас майбутнього додатку. Також цей підхід гарно себе демонструє при розробці невеликих UI систем. Загалом ООР модель має достатню потужність для того щоб за її допомоги реалізувати усі частини програмної логіки. Втім це не доцільно при збільшенні розміру відеогри, або при ускладненні зв’язків в ігровій

логіці. У таких випадках краще відійти від об’єктно-орієнтованої парадигми на користь інших моделей.

- Entity Component System наразі є найважливішою моделлю для вивчення та використання у розробці відеоігрових проектів. Модульність, здатність до швидких змін, висока ефективність обчислення, ефективне збереження та передача даних і великий ступінь сумісності з багатьма іншими ефективними техніками програмування, роблять з цієї парадигми дуже вдалий інструмент для роботи з такою динамічною і комплексною системою, як відеогра.
- У свою чергу реактивні розширення необхідно використовувати, якщо у грі потрібно підтримувати широкий спектр користувацьких налаштувань перед основним геймплеєм, або якщо сам менеджмент або налаштування та експерименти з ігровими сутностями в хабі і є важливою складовою гри. В таких випадках необхідно розробити графічний інтерфейс великої складності. Реактивний підхід значно спростить написання складних і багатошарових UI систем, дозволить обробляти одразу декілька запитів від гравця та підвищить відмовостійкість системи в цілому. Ці аспекти напряму впливають на ігровий досвід, тому багато ігрових рушіїв мають свої власні рішення у сфері реактивного програмування.

2.1 Ефективні підходи до вирішення складних завдань ігрової розробки

Кожен ігровий проект має ряд схожих і досить складних проблем, які необхідно вирішити у процесі розробки. Декомпозуємо основні з них, та підберемо найкращі рішення.

Віртуальний простір

На початку розробки будь якої гри проектуються три основних аспекти майбутнього додатку: віртуальний простір, core-механіки та програмне представлення ігрового контроллеру (спосіб взаємодії гравця з грою). Зупинимося на першому.

Віртуальний простір зазвичай має представлення у вигляді окремих ігрових рівнів різного розміру або єдиного відкритого ігрового світу з ілюзією відсутності розмежувань. Існують три основні підходи до створення віртуального ігрового простору:

- Процедурна генерація
- Власноручне конструювання
- Гібридний підхід

Для процедурної генерації необхідно створити алгоритм, який як правило є послідовним виконанням декількох інших алгоритмів. Алгоритми процедурної генерації мають в основі випадкові, а точніше - псевдо випадкові числа. Існує два основних підходи до процедурної генерації:

1. Так званий «запечений» варіант. Коли простір генерується один раз, при першому запуску алгоритму, а потім зберігається у пам’яті пристрою. Зазвичай створюються спеціальні інструменти гібридного підходу, які дозволяють згенерувати випадковий віртуальний простір певного розміру в якості основи, після чого розробник власноруч допрацьовує згенерований ассет.

В якості прикладу розглянемо генерацію кімнат, або так званих «данжей». Така структура ігрового простору дуже поширена у іграх з процедурною генерацією.

В основу алгоритму для генерації кімнат, з’єднаних коридорами, зазвичай покладають контрольовану множину випадкових значень. Для цього

використовується нормальний розподіл випадкової величини (розподіл Гауса) або генератор псевдо-випадкових чисел Парка-Міллера. Ці методи досить швидко генерують послідовність псевдо-випадкових значень на достатньо великому діапазоні. Значення , для генерації кімнат та коридорів, отримані з генератору, використовуються у створенні набору довільних прямокутників. Додатково вводиться обмеження на співвідношення сторін, аби мати більше контролю над формою майбутніх кімнат. Після генерації прямокутників їх необхідно поєднати графом у вигляді мінімального остовного дерева. Ребра графу будуть слугувати проходами між кімнатами. Щоб цього досягти, спочатку застосовується метод тріангуляції Делоне або розбиття Вороного з подальшим використанням алгоритму Прима з метою виокремити мінімальне остовне дерево.

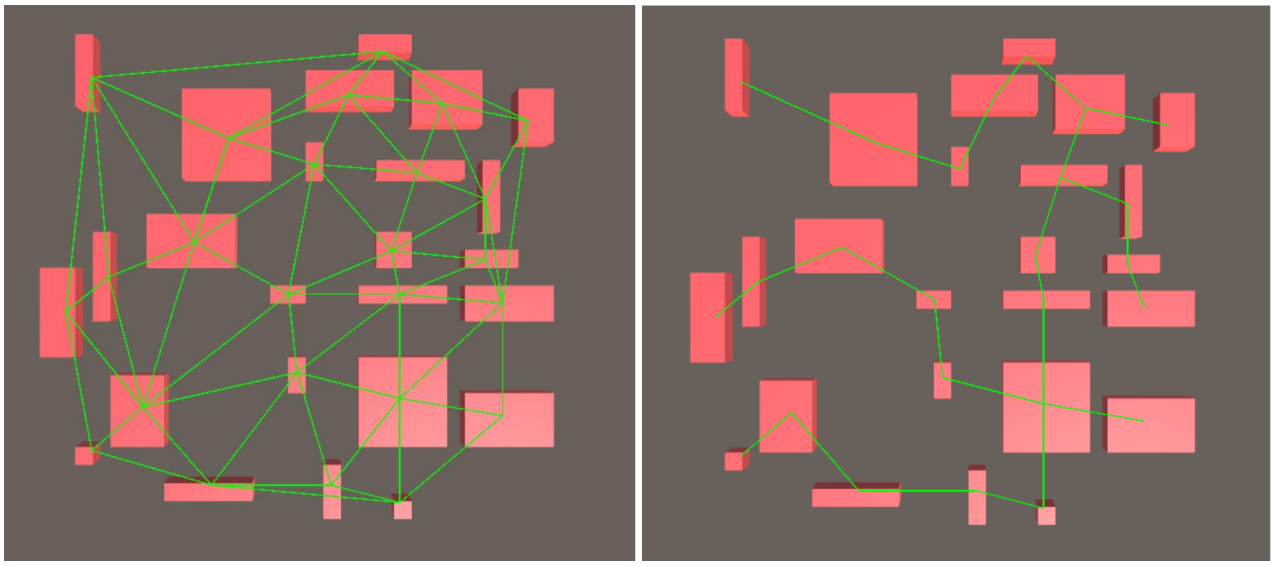


Рисунок 1.1 - Генерація випадкової мережі кімнат

До отриманого дерева можна додати випадкову кількість векторів з тріангульованого графу, аби отримати декілька зациклених шляхів.

На наступному кроці необхідно згенерувати проходи на основі створеного графу. Для цього добре підходить алгоритм пошуку шляху A*. Налаштуємо граф так, щоб шлях був дешевшим крізь малі кімнати (майбутні коридори) та дорожчим крізь великі кімнати.

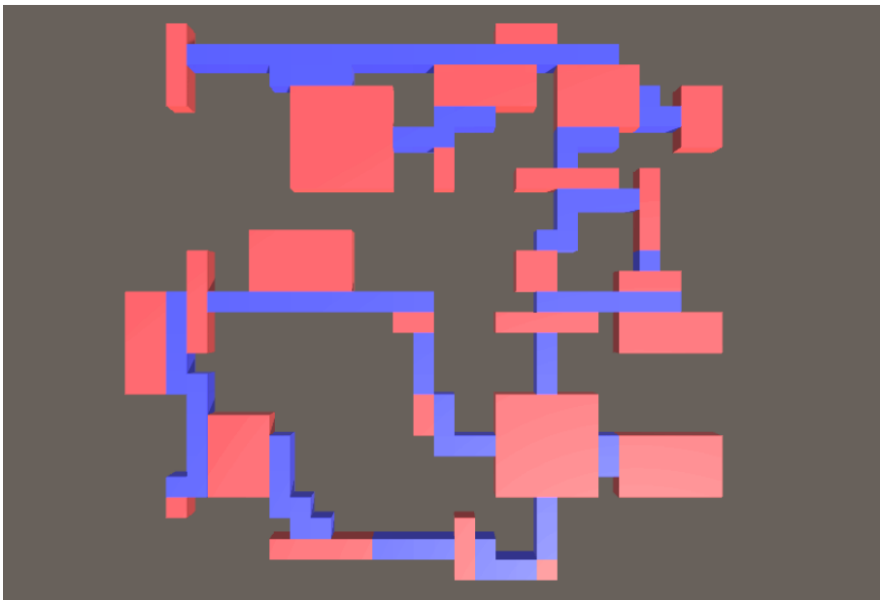


Рисунок 1.2 - Згенерована мережа кімнат

У тривимірному просторі тріангуляція Делоне теж має варіанти рішень з використанням сфер замість кіл, алгоритм пошуку шляху також може бути адаптований для руху у додатковій площині координат. Таким чином можна створювати незліченну кількість унікальних комбінацій кімнат та коридорів, які слугуватимуть ігровим полем.

2. Другим варіантом процедурної генерації, є генерація у реальному часі з використанням контрольованих випадкових чисел для отримання сталих результатів при однакових вхідних параметрах. Такий вид генерації прибирає головний недолік попереднього типу – використання великого обсягу пам’яті. Проте зі зростом бажаної складності ігрового рівня, значно зростає обчислювальна складність алгоритму і відповідно навантаження на CPU.

Поширеним та ефективним підходом до реалізації другого типу процедурної генерації, є використання позиції ігрового персонажа у просторі в якості зерна для алгоритму генерації навколишнього середовища. Таким чином досягається випадковість згенерованих даних у різних частинах простору, але генерація для тої самої позиції залишається сталою. В основу можна знову покласти генератор псевдо випадкових чисел Парка-Міллера. Але на цей раз

визначити невеликий діапазон генерованих значень, та передавати поточну позицію гравця в якості зерна для генерації. В результаті можна отримати середовище абсолютно випадкового вигляду, яке генерується у реальному часі, займає умовно 100 мб оперативної пам’яті, та залишається стабільним при поверненні гравця на попередні позиції.

В якості підсумку, варто відзначити, що важливою складовою при генерації випадкового віртуального простору є встановлення обмежень та підбір граничних значень, які в результаті стають інструментами налаштування для отримання бажаних результатів.

Самостійне конструювання рівнів, це екстенсивний варіант дизайну ігрового середовища. Але цей підхід значно швидше впроваджується, надає більший контроль над процесом створення ігрового поля та дозволяє розробляти складніші рівні з більш виваженим дизайном.

Для власноручного створення ігрового середовища, рушій Unity має вбудований інструментарій. Відкритість кодової бази рушія дозволяє доповнювати свій проект власними інструментами. До того ж суспільством розробників Unity вже були створені користувальницькі розширення для дизайну рівнів, такі як Polybrush. Сам Юніті окрім засобів безпосереднього редагування полігонів додатково має методи збереження створених ресурсів у вигляді префабів (prefab) або сцен (scene). Префаби у свою чергу мають деякі властивості, що притаманні об’єктам у об’єктно-орієнтованій парадигмі. Вони можуть мати нащадків, так званих prefab variant, що створюються на основі звичайного префабу, котрий виступає у ролі зразка (template). Завдяки цьому можна легко змінювати спільні елементи у багатьох взаємопов’язаних ассетах. При розміщенні префаба на ігрову сцену створюється його копія тільки в рамках цієї сцени (аналог паттерна Прототип), що запобігає розповсюдженню дублікатів у ресурсах проекту.

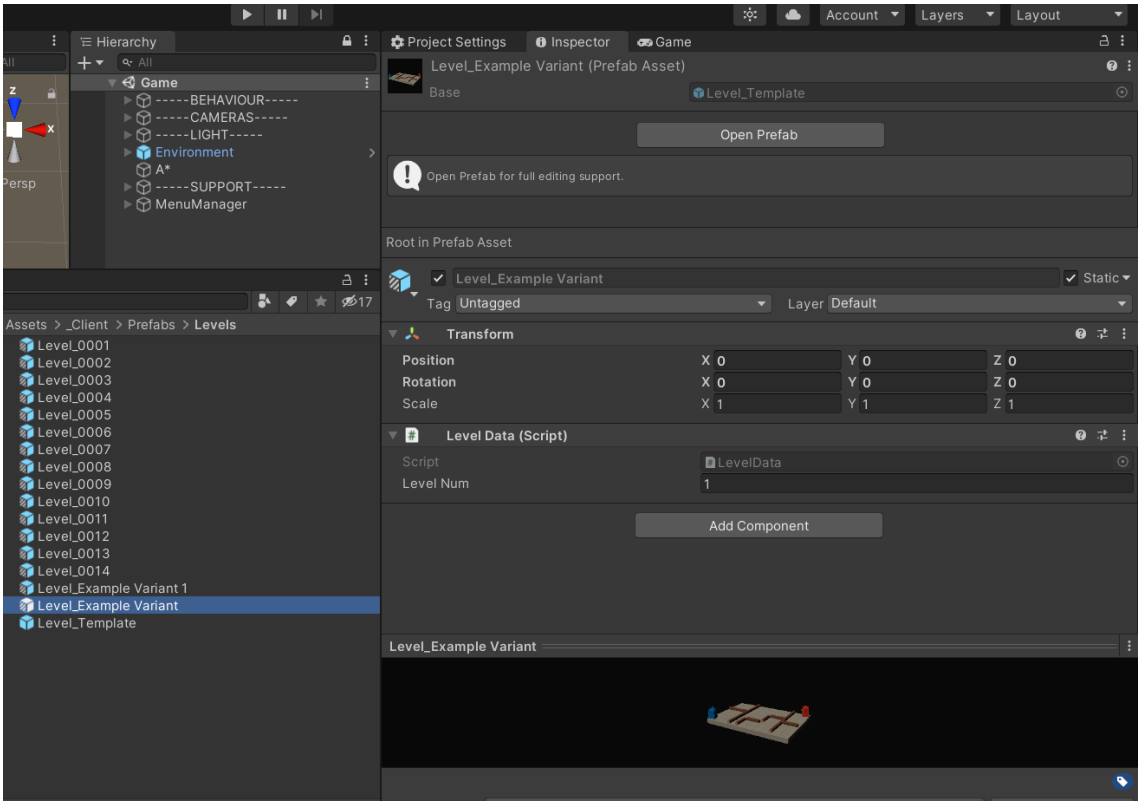


Рисунок 1.3 - Власноруч створені Prefab Variant ігрових рівнів

Для того щоб впевнено оперувати графічними об’єктами в Unity, при конструюванні рівінів, додатково потрібно мати мінмальні знання у області кватерніонів.

Collision detection in runtime. Симуляція фізики у відеогрі.

Перед тим як перейти до core-механік, необхідно розглянути ще одну нетривіальну проблему - симуляція фізичної взаємодії об’єктів між собою. На практиці багато рушіїв мають у собі додатковий дочірній рушій для симуляції фізики. Туди входять вбудовані алгоритми та інструменти для імітації різних фізичних явищ реального світу (симуляція тканини, симуляція води, симуляція тертя, гравітації, ваги, пружини, прискорення). На практиці у більшості випадків можна обійтись без значної кількості запропонованих фізичних симуляцій, окрім симуляції зіткнень (collision detection).

Unity має власну систему колізій, основану на фізичному рушії Unity Physics. В останніх версіях команда Unity додатково впроваджує новий рушій

для більш складної симуляції фізики – Havok Physics, який буде працювати на DOTS парадигмі, що є імплементацією ECS підходу в Unity.

Важливо знати, що фізика в Unity поділена на 2d та 3d сектори. Для роботи Unity Physics необхідно, щоб на об’єктах був компонент Rigidbody або Rigidbody2D, який є маркером того що об’єкт є частиною певного фізичного світу. Rigidbody додатково надає перелік параметрів для налаштування фізичних властивостей об’єкта (маса, опір, гравітація, тип колізій).

Після закріплення Rigidbody на об’єкті необхідно додати ще один компонент – Collider. Цей компонент має безліч версій, є можливість налаштувати форму, розмір і призначення колізії.

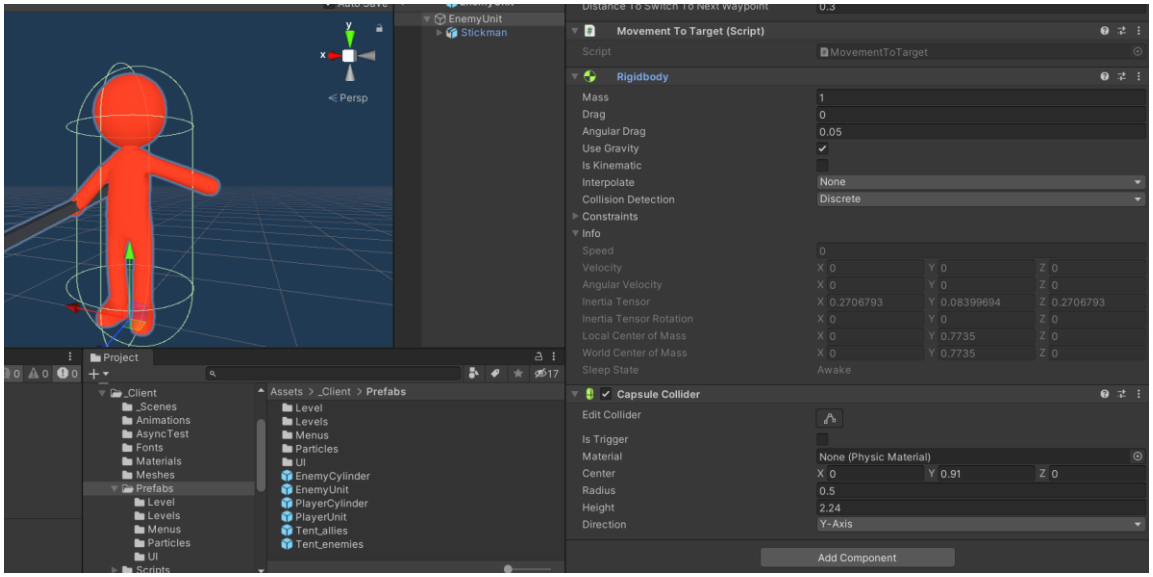


Рисунок 1.4 - Компонент Capsule Collider на об’єкті юніта

Після цих приготувань будь який об’єкт з колайдером та Rigidbody буде мати пасивні фізичні властивості. Додатково можна викликати функції фізичного рушія із скриптів, аби симулювати активні фізичні властивості.

Такої симуляції достатньо для розробки проект з маленькою кількістю фізичних агентів і великим запасом обчислювальної потужності. Кожен колайдер, це додаткові полігони, кожен Rigidbody це надлишкові розрахунки фізичних взаємодій різної складності. Тому на даному етапі Unity не здатен

підтримувати адекватну кількість FPS, якщо у сцені задіяна велика кількість об’єктів з підтримкою симуляції фізики.

У випадках, коли необхідно симулювати колізії великої кількості агентів одночасно, варто використовувати власні рішення, тому розглянемо можливі варіанти:

1. Використання Physics.Raycast

Цей підхід є найшвидшим і найпростішим для симуляції примітивних колізій між об’єктами. Використовуючи метод з того самого рушія Unity Physics, можна значно оптимізувати розрахунки фізики, але втратити у якості симуляції. Функціонал методу Physics.Raycast дозволяє провести вектор заданої довжини та форми у конкретному напрямку, та отримати результат його перетину з іншими колайдерами. Є певні налаштування, наприклад можна отримати всі пересічення на шляху, а можна зробити запит лише на перше влучання у колайдер. Існує опція вести не вектор точок а вектор сфер, або інших об’ємних фігур, з метою збільшення площі перетину. На основі цього симуляцію колізій можна реалізувати у такий спосіб:

- На кожному кадрі фізичного циклу робити Raycast капсули, схожої на об’єкт за розміром, від самого об’єкта у напрямку його руху на задану відстань.
- Прорахувати квадрат дистанції до найближчого перетину з іншим колайдером.
- Якщо отримане значення є меншим за квадрат прийнятного відхилення, то викликати функцію зупинки поточного об’єкта.

Незважаючи на простоту даного рішення, воно добре працює для колізій об’єкта з віртуальним простором (аватару гравця зі стінами, бордюрами, деревами та іншими об’єктами оточення). Додатковою оптимізацією цього

підходу є налаштування матриці шарів у налаштуваннях фізики рушія (Edit -> Project Settings -> Physics).

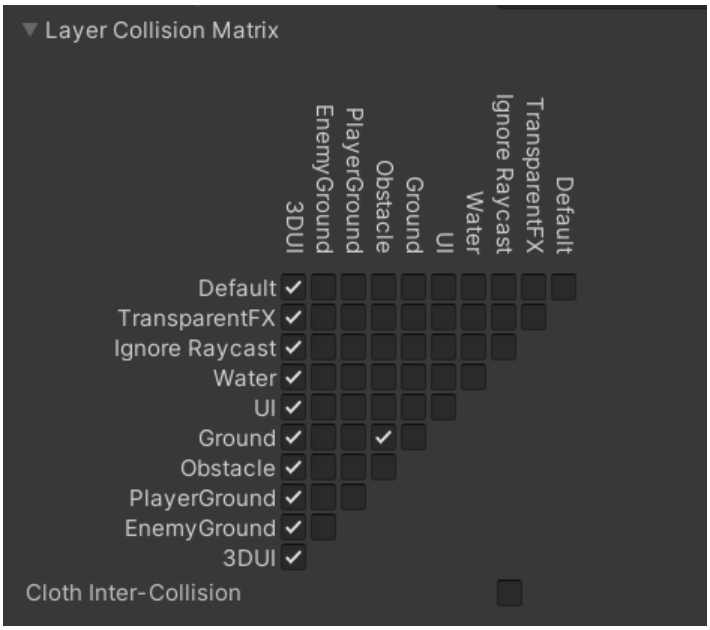


Рисунок 1.5 - Налаштування Layer Collision Matrix

Використовуючи цей інструмент можна налаштовувати фізичну взаємодію окремих шарів. На будь який шар можна покласти будь яку кількість об’єктів. Unity здатен підтримувати до 25 користувацьких шарів.

Варто відзначити недоліки підходу з Raycast:

- На великій швидкості виникає так званий Tunneling. Явище, при якому об’єкт рухається настільки швидко, що проходить крізь інші колайдери між двома кадрами. Подовження Raycast в залежності від швидкості об’єкта лише частково приховує цю проблему.
- У рантаймі досі пристуня велика кількість колайдерів, які створюють значне навантаження на CPU та оперативну пам’ять.
- Відсутність пасивної фізики. Прорахування колізій поза напрямком руху об’єкта значно складніше і дорожче прорахувати, використовуючи Raycast.

2. Continuous Collision Detection

Цей підхід варто виокремити, тому що він вирішує проблему Tunneling. Завдяки використанню лінійної інтерполяції можна прорахувати точну позицію, де має опинитись тіло під час колізії, та використати цю позицію для корегування руху об’єкта. CCD підхід може стати прямим вдосконаленням Raycast системи. Для цього необхідно:

- Визначити координати точки на відстані радіуса колайдера нашого об'єкта від точки перетину Raycast, у напрямку об'єкта.
- Прорахувати точний момент колізії за допомогою рівняння лінійної інтерполяції, та передати цей час для наступного прорахунку позиції об'єкта.

CCD може бути застосований з багатьма іншими підходами. Цей метод має досить високу обчислювальну складність, і його чисте використання при збільшенні об’єктів призведе до зростання навантаження на CPU у геометричній прогресії. Тому CCD слід використовувати тільки у поєднанні з менш точним та дешевим collision detection алгоритмом та тільки за умови, що у грі є об’єкти, що рухаються надто швидко для покадрової перевірки звичайними методами. Також необхідно якомога рідше прораховувати CCD, тобто тільки перед самою колізією.

3. Spatial Hashing

Цей підхід полягає у розділення простору на блоки (chunks), та замість підрахунку конкретних позицій кожного об’єкта відносно позицій кожного іншого об’єкта, у обчислення використовуються лише сектори простору, та перевіряється наявність об’єктів в цих секторах.

Spatial Hashing має багато різних реалізацій. Найпростішим є звичайне розділення простору на решітку, та відстеження, в якій клітині знаходиться

кожен об’єкт у кожний момент часу. Втім такий підхід призводить до проблем у підрахунку на межах між клітинами. Одним з варіантів вдосконалення цього методу є використання шестикутників замість клітин для розділення простору. У такий спосіб значно зменшується ймовірність невизначеності на кордонах чанків.

Втім найбільш ефективним підходом для прорахування колізій є поділ простору з використанням структур даних, які підтримують ефективний пошук потрібних значень. Одним з варіантів є використання K-D дерев. Дерево заповнюється позиціями об’єктів. При додаванні до дерева, об’єкт розміщується відносно медіани своєї позиції вздовж певної осі в залежності від поточної кратності 1 – x, 2 – y, 3 – z (для двовимірної колізії 1– x та 2– z). В результаті простір поділяється на блоки довільного об’єму, які зберігаються у структурі дерева. Кожен вузол має рівень, причому найближчі елементи мають найменшу різницю у рівнях. В результаті заповнення дерева, вузли що знаходяться на одному рівні – і є головним кандидатом на колізію. Далі вона вже перевіряється прямим порівнянням або зі застосуванням CCD.

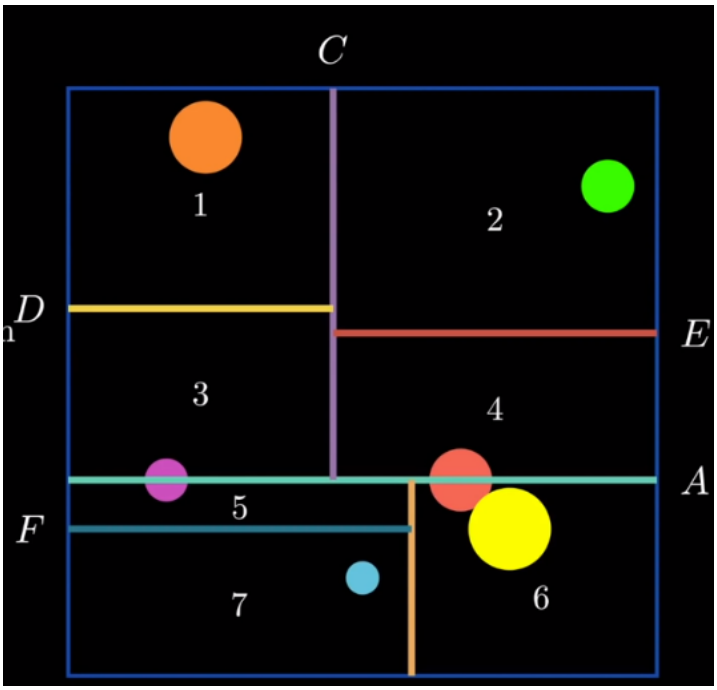


Рисунок 1.6 - Принцип розділення простору по K-D Tree

Ефективність використання KD Tree значно знижується, якщо об’єкти мають надто різноманітну форму і сильно деталізовані і складні площини

дотику.

Вирішити цей недолік може схожий метод, який замість KD Tree використовує Bounding Box. Насправді це і є ті самі Unity колайдери, від яких ми відійшли на початку дослідження. Дійсно, для складних форм більш ефективного методу перевірки на колізію ще не винайдено. Проте власна реалізація алгоритму, що використовує Bounding Box підхід, може бути набагато швидшою ніж аналог Unity. Адже спеціалізовані рішення завжди виграють у швидкості у порівнянні з загальними. Існує декілька варіантів реалізації Bounding Box Spatial Hashing. За основу береться обмеження простору коробками (колайдерами) навколо крайніх та центральних об’єктів і занесення цих коробок у дерево. Таким чином поступово коробки звужуються, і коли виокремлення нової коробки не має сенсу (вона не буде містити об’єкту), заповнення дерева завершується. У отриманому бінарному дереві кожна пара листів одного вузла є потенційною колізією.

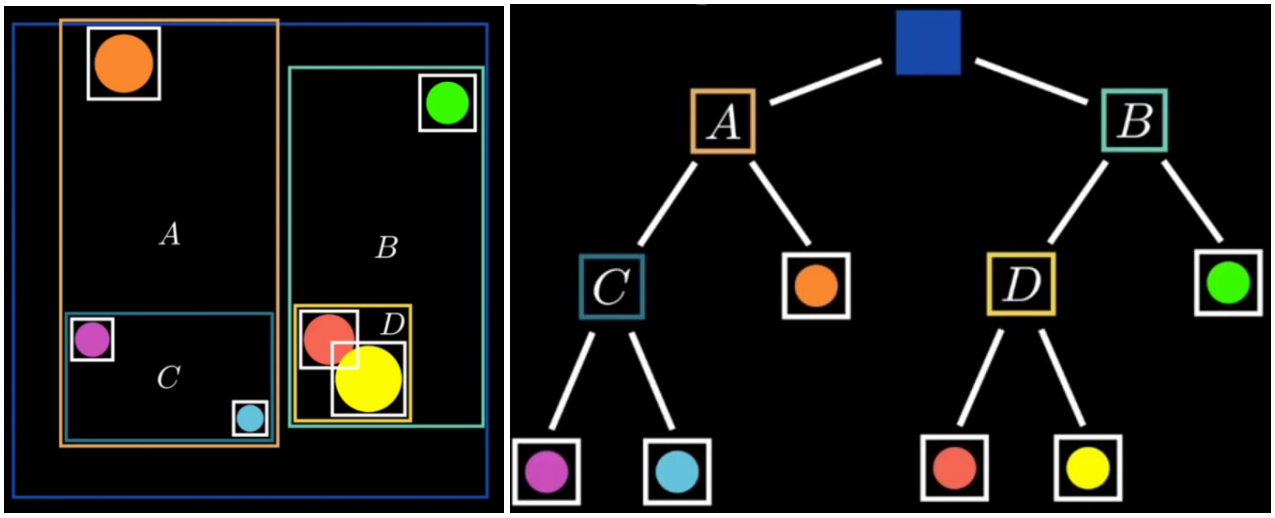


Рисунок 1.7 - Принцип розділення простору через Bounding Volumes

В результаті можна остаточно виокремити структуру будь якого ефективного підходу до пошуку та симуляції колізій:

1. Загальний розділ простору для пошуку кандидатів на колізію.

   Цей етап має використовувати загальні методи оптимізації, та швидкі методи розподілу наявного простору на підблоки. Найефективнішими на цьому етапі себе проявляють методи, засновані на K-D Tree або Bounding Values.

2. Визначення об’єктів що зіткнулись.

   Якщо використовувати деревоподібні структури, то цей етап полягає у простому порівнянні позицій сусідніх листів дерева. В інших випадках можна застосувати порівняння між сусідніми чанками або сортування.

3. Точне моделювання визначених колізій.

   У випадку необхідності, на цьому етапі прораховуються уточнюючи координати для коректного відображення колізії, та подальшої поведінки об’єктів. Найнадійнішим методом на цьому етапі є застосування CCD.

РОЗДІЛ 2 АНАЛІЗ ПРОГРАМНО-ТЕХНОЛОГІЧНИХ РІШЕНЬ ДЛЯ

РОЗРОБКИ ВІДЕОІГОР

2.1 Необхідний набір інструментів для створення ігрового застосунку.

Розробка ігрового проекту з точки зору організації та побудови процесів роботи має багато спільних рис із розробкою комплексних веб-сервісів або мобільних застосунків.

В основу закладається концепція

Концепція розвивається до дизайн документу

Визначаються основні модулі майбутнього додатку та починається формування перших технічних вимог.

Початок паралельної роботи над першим прототипом та подальшим удосконаленням дизайн документу.

Вихід на ітеративний процес із застосуванням Agile моделі

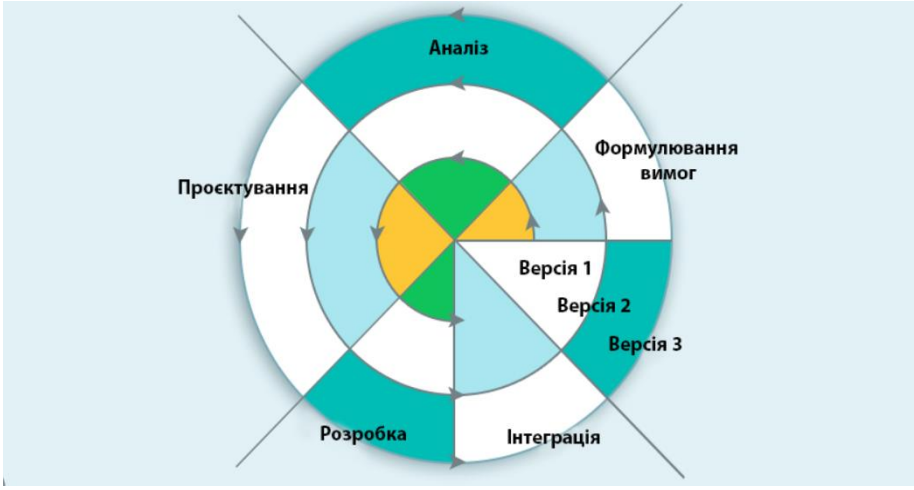


Рисунок 2.1 - Ітеративний процес розробки

Для комфортного та ефективного циклу розробки необхідно вміти користуватись наступними інструментами:

- Система управління проектами
- Система візуальної дошки для проектування
- Система контролю версій
- IDE
- Ігровий рушій
- Електронний або фізичний записник

Проведемо порівняльний аналіз поширених систем управління:

Таблиця 2.1 – Порівняння систем управління проектами

	JIRA	Asana	Trello
Підтримка Agile методологій	Наявна	Наявна	Частково пристуня
Календарний планер	Наявний	Наявний	Наявний
Інтеграція з Git системами	Наявна	Наявна	Тільки з Git Hub
Аналітика продуктивності	Наявна	Наявна при встановленні додаткового розширення	Наявна при встановленні додаткового розширення
Можливість створювати власні шаблони	Наявна	Наявна	Відсутня
Розширюваність системи	Має Atlassian SDK який дозволяє	Відсутня	Теж є частиною Atlassian сімейства.

	створювати власні рішення		
Система ролей у команді	Наявна	Наявна	Наявна у спрощеному варіанті
Інтеграція з іншими сервісами	Має інтеграцію з власними сервісами групи Atlassian (Confluence, Bitbucket)	Наявна інтеграція з хмарними сервісами	Має інтеграцію з власними сервісами групи Atlassian
Ведення документації	Дозволяє вести документацію у потужному сервісі Confluence, та інтегрувати зручні посилання до задач	Дозволяє колективно вести документи	Теж має засоби для поєднання з базою документації в Confluence.
Система безпеки	Високий рівень Ролі, шифрування, буфер IP адрес	Середній рівень Захищені сервери Неможливість локального запуску	Середній рівень Постійні бекапи, готові розширення для безпеки

Підтримка української локалізації	Наявна	Відсутня	Наявна
Унікальні особливості	Можливість інтеграції з Discord Можливість повної кастомізації функцій сервісу через SDK	Інтеграція з Zoom та Jira Cloud Багато варіантів інтеграції з різними хмарними сервісами	Можливість отримати усю велику потужність сервісів Atlassian у облегшеному інтерфейсі
Середня ціна	7.50$	11$	5$

Серед проаналізованих сервісів управління проектами JIRA найбільше підходить досвідченим командам. Ті, хто не любить а боїться такої вискої функціональності та перевантажених інтерфейсів, можуть почати з Trello, та поступово інтегруватись в JIRA. Серед систем управління від українських розробників наразі нажаль немає конкурентно спроможного продукту. Найуспішніший проект від української команди у цій галузі – Casual, зараз повність орієнтований на ринок США і не має тої функціональної повноти, яку мають сервіси, розглянені вище.

Перед початком розробки ігрового проекту, окрім підготовки системи управління проектами, необхідно створити репозиторій у git або subversion системі. У вільному доступі наявна велика кількість веб-сервісів, desktop клієнтів та інших рішень для зручного використання системи контролю версій. В решті решт багато хто використовую звичайну командну строку або GitBash. Порівняємо найпопулярніший клієнт GitHub та мій особистий вибір – SmartGit:

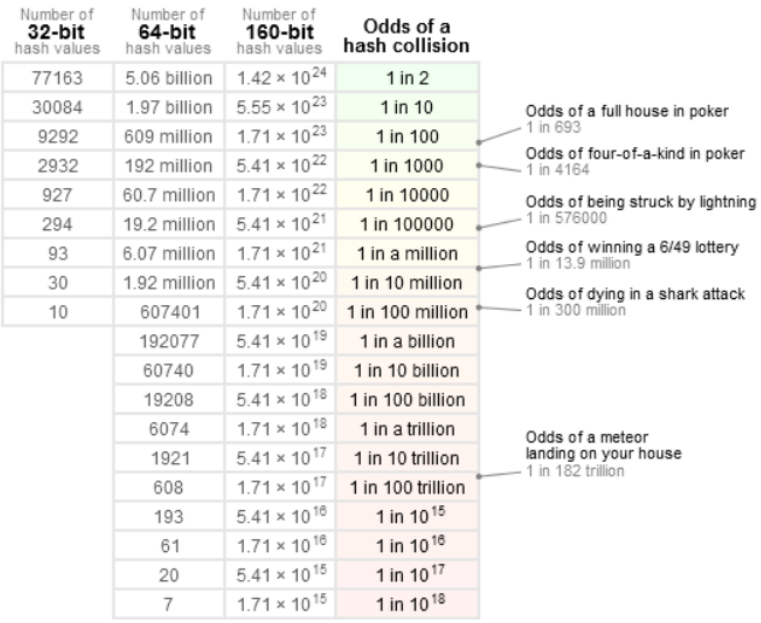
Таблиця 2.2 – Порівняння GitHub клієнтів

	GitHub Desktop	SmartGit
Інтеграція з GitHub/GitLab/Bitbucket	Наявна	Наявна
Основні команди git у графічному інтерфейсі	Наявні	Наявні
Відображення дерева версій	Наявне	Наявне
Навігація між версіями по графу	Наявна	Наявна
Перегляд різниці у прогресі між двома гілками	Наявна	Наявна
Поле командної строки	Відсутнє	Наявне
Автоматичне вирішення конфліктів	Наявне	Наявне
Власноручне вирішення конфліктів	Наявне	Наявне
Side-to-side порівняння версій	Відсутнє	Наявне
Інтеграція git-flow	Наявна	Наявна
Відображення повідомлень з консолі	Відсутнє	Наявне

Фільтри та розширений пошук за ключовими словами	Обмежений функціонал стандартних фільтрів	Наявні у повній мірі
Прив’язка клавіш через Vim	Відсутня	Наявна
Код ревью	Наявне	Наявне
Унікальні особливості	Добер складений мінімалістичний інтерфейс	Можна відстежити повний ланцюг змін версії файлу або області коду
Ціна	Безкоштовний для open-source проектів	Безкоштовний для всіх проектів, які не заробляють більше 100 000 долларів на рік

Наведена таблиця є аргументом мого вибору git-клієнту на користь SmartGit.

Git системи не мають жодних недоліків і в будь якій ситуації підвищують ефективність та надійність розробки проекту. Єдина відома небезпека при використанні систем контролю версій на базі git – це Hash collision. Для диференціювання версій git використовує хеш-ключі, це дозволяє значно облегшити навантаження на клієнт-серверну взаємодію та пам’ять хмари. Проте існує ймовірність так званої колізії хешів, коли дві версії мають однаковий хеш ключ, та відповідно оперуються системою однаково, що призводить до деградації усього сховища та критичних помилок. Ймовірність цього явища дуже мала, тому git-системи контролю версій досі вважаються одними з найефективніших та найбезпечніших.



Number of **32-bit** hash values	Number of **64-bit** hash values	Number of **160-bit** hash values	Odds of a hash collision
77163	5.06 billion	1.42×10^{24}	1 in 2
30084	1.97 billion	5.55×10^{23}	1 in 10
9292	609 million	1.71×10^{23}	1 in 100
2932	192 million	5.41×10^{22}	1 in 1000
927	60.7 million	1.71×10^{22}	1 in 10000
294	19.2 million	5.41×10^{21}	1 in 100000
93	6.07 million	1.71×10^{21}	1 in a million
30	1.92 million	5.41×10^{20}	1 in 10 million
10	607401	1.71×10^{20}	1 in 100 million
	192077	5.41×10^{19}	1 in a billion
	60740	1.71×10^{19}	1 in 10 billion
	19208	5.41×10^{18}	1 in 100 billion
	6074	1.71×10^{18}	1 in a trillion
	1921	5.41×10^{17}	1 in 10 trillion
	608	1.71×10^{17}	1 in 100 trillion
	193	5.41×10^{16}	1 in 10^{15}
	61	1.71×10^{16}	1 in 10^{16}
	20	5.41×10^{15}	1 in 10^{17}
	7	1.71×10^{15}	1 in 10^{18}

Рисунок 2.3 - Залежність ймовірності hash collision від довжини хешу

Як бачимо з наведених підрахунків, чим більша довжина хеш-ключа, тим менша ймовірність виникнення колізій. При використанні хеша довжиною в 160 біт подія колізії майже неможлива, приблизно 1 на мільярд випадок. В git системах використовуються хеши довжиною 256 біт.

2.2 Порівняльний аналіз ігрових рушіїв.

За 70 років розвитку відеоігрової промисловості, суспільство створило безліч готових рішень для використання в якості базису для подальшої розробки ігор. Ці рішення варуються від окремих бібліотек або фреймворків до спеціального програмного забезпечення. Існують навіть ігри для створення інших ігор, що побуджує молоде покоління до креативності та раннього заохочення до роботи у цій сфері.

Найпотужнішим інструментом для створення відеоігрових проектів є ігровий рушій. Рушії відрізняються підходом до розробки, інтерфейсом, орієнтованістю на певну мову програмування, засобами збереження та контролю інформації, підтримкою платформ та ін.

Багато ігрових студій створюють васні рушії, аби задовільнити власні потреби. У польської CdProjectRed власний рушій RedEngine, у німецько-турецької Crytek – CryEngine, а у загальновідомої Valve – власний Source Engine. Проте існують і загально доступні рушії. Безперечними лідерами серед них є Unity Engine від Unity Technologies та Unreal Engine від Epic Games. Проаналізуємо обидва рушія, для цього будемо використовувати останні версії обох.

Таблиця 2.3 – Порівняння ігрових рушіїв

	Unity	Unreal Engine
Рік виходу у загальний доступ	2005	1998
Середня частота оновлень	Декілька разів на рік	Приблизно раз в 6 років
Цільова платформа	Повністю мультиплатформений рушій. Є можливість зібрати проекти під PC, Mobile, Web, PS, Xbox, Mac, Nintendo Switch	Підтримує усі головні платформи, але пріоритетною завжди була PC
Мова програмування	Основна – C# Можна встановити додаткові плагіни для програмування на JavaScript, C++ та навіть Python.	Основні мови – C++ та власна мова Blueprint.

Вихідний код	Надає open-source доступ тільки після придбання.	Повністю open-source
Поріг входу	Низький	Достатньо високий, через необхідність вивчати Blueprint
Графічний інтерфейс	Зручний та компактний. Інтуїтивне проектування.	Зручний, але треба витратити час на вивчання.
Інструментарій для 2D ігор	Присутні окремі потужні екосистеми для 2D ігор. Від спеціального фізичного рушія до докремого Render Pipeline.	Також має інструментарій для створення 2D проектів, проте значно менший ніж у Unity.
Інструментарій для 3D ігор	Має повний набір інструментів для створення 3D ігор від початку до кінця.	Має абсолютно повний набір інструментів для створення 3D ігор. Містить більше вузько направлених інструментів, ніж Unity.
AR/VR	Має підтримку окремих плагінів для комфортнгої розробки AR та VR ігор. Наприклад плагін Vuforia.	Має власні потужні AR та VR фреймворки для створення ігор у цій сфері.

Інструменти для оптимізації	Має достатній набір інструментів для аналізу слабких міст та їх вдосконалення. Є інструменти для автоматичної оптимізації графіки, скриптів та використання пам’яті.	Має повний набір інструментів для аналізу слабких міст та їх вдосконалення. Містить надпотужний набір інструментів для оптимізації графіки різними методами (LOD, Defeaturing, Proxy Geometry)
Рендерінг	Має перелік рендер пайплайнів на вибір, в залежності від специфіки проекту. Сам рендерінг досить повільний, проте не критично. В останніх версіях додатково з’явилась можливість розширювати наявні та створювати власні рендер пайплайни.	Має перелік рендер пайплайнів на вибір. Унікальною рисою є варіант рендер пайплайну для кінофільмів. Ренжерінг значно швидший та оптимальніший ніж в Unity. Має потужну систему Post Procrssing. Уся графічна система розширювана та відкрити до користувацьких налаштувань.
Анімації	Містить власний комплексний інструмент для створення і	Має значно потужніший інструментарій для анімації. На базі Unreal

	контролю анімацій, в основі якого лежить FSM. Є можливість створювати бленди та дуже тонко налаштовувати анімаційні кліпи.	Engine останніх версій створюється дуже багато цифрових продуктів навіть для мультфільмів та фільмів.
Світло	Має тонкі налаштування світла у сцені. Є можливість для власних рішень. Світло можна запікати та використовувати інші поширені методи оптимізації. Має потужний інструментарій світла для 2D проектів.	Як і в інших сферах графіки, у світлі Unreal Engine переважає Unity. Обїємне світло, підтримка різних видів Phyics based освітлення, високий рівень оптимізації та прозорий процес налаштування створюють враження необмеженого потенціалу у графіці.
Програмування графічного процесора	Unity має власну UnityShaderLab та екосистему для створення інструкцій графічному процсору різних видів. Також є можливість використання мов HLSL, GLSL та ін.	Має власну зручну та потужну мову для написання шейдерів. Присутня зручна інтеграція з C++. Загалом програмування для графічного процесора в Unreal Engine вважається більш

		оптимізованим та зручним у використанні.
VFX	У наявності є набір інструментів для створення VFX. А саме Particle System, графічний редактор ShaderLab та багато плагінів, доступних у Asset Store.	Має дуже функціональну систему для створення Particle. Самі партікли мають декілька варіантів представленя кожен з яких досконало оптимізований для вирішення певних завдань (від створення диму та води до генерування власних партіклів)
Підтримка графічних API	Має абсолютну більшість інтеграцій з різноманітними графічними API. DiretX, Metal, Vulkan, OpenGl. За рахунок цього має можливість підтримувати багатоплатформеність.	Є підтримка усіх основних графічних API. Але має складніший процес впровадження сторонніх API.
Інструменти клієнт-серверної взаємодії	Для Unity були створені ряд ефективних плагінів для реалізації різних завдань, таких як peer-	Має власну надпотужну систему для реалізації мультиплеєру. Реалізовані ефективні

	to-peer мультиплеєр та ін. Прикладом є використання Photon або Colyseus.	рішення ряду проблем, від симуляції фізики у мультиплеєрних іграх до підбору та контролю підключення коієнтів.
Підтримка медіа форматів	Наявна зручна підтримка всіх поширених медіа форматів	Наявна зручна підтримка всіх поширених медіа форматів
Asset store	Має великий asset store з понад 31000 готових рішень та удосконалень.	Має достатньо великий asset store з понад 10000 готових рішень та удосконалень.
Ком’юніті	Має величезне ком’юніті, що нараховує понад 63% від загальної кількості розробників ігор.	Має велике ком’юніті за рахунок довгої історії та якісно надаваних послук. Кількість оцінюється між 13% та 20% від загальної кількості розробників ігор.
Технічна підтримка	Наявна якісна технічна підтримка 24/7 та зручний bug tracking по версіям.	Наявна досить якісна технічна підтримка 24/7, проте за статистикою вона програє Unity/
Performance	Мова C# добре підходить для розробки	Мова C++ прекрасно підходить для складних

	ігор та значно легша у застосуванні до вирішення проблем ігрової розробки. Сам рушій дуже добре оптимізований для створення проектів під мобільні платформи.	математичних розрахунків, що робить її дуже вдалим вибором для розробки складних ігор для потужних машин. Вважається що Unreal Engine найкращий вибір серед загальнодоступних рушіїв для розробки під PC платформи
Інтеграція з гіт	Наявна	Наявна
Інтеграція з IDE	Наявна з усіма IDE що підтримують C#	Наявна з усіма IDE що підтримують C++
Власна білд-машина	Існує власна білд машина Unity Cloud Build. Вона має увесь необхідний функціонал для паралельної сборки, відлову та виправлення помилок, підтримки версіонності, автоматизації процесів та домить зручна у використанні.	Відсутня Але може бути інтегрована з використанням стороннього плагіна або ПЗ.
Документація	Наявна у повному обсязі. Додатково є багато навчальних	Наявна у повному обсязі.

	матеріалів від розробників на офіційній платформі Udemy. Навчання на платформі безкоштовне, гейміфіковане та має багато проектів у якості прикладу.	
Унікальні особливості	Має власні хмарне сховище, інструменти аналітики та crashlytics. Розробляє інноваційну систему DOTS в якості основного підходу до розробки кодової бази ігрових проектів, в основі якого ECS модель та паралельні обчислення.	Надпотужний інструментарій для створення графіки. Вдале поєднання C++ та Blueprint, що дозволяє створювати дуже масштабні ігрові проекти без уповільнення у темпі розробки.
Use Cases	Cuphead, Ori and the blind forest, Monument Valley 1-2, Hollow Knight	Hellblade: Senua’s Sacrifice, Fortnite, Batman, Star Wars, Stalker, Gears, Bioshock
Цінова політика	75$ в місяць за pro версію	5% з продажів розробленого проекту

Для свого проекту я обрав Unity Engine, адже наразі віддаю перевагу мові C# та високій швидкості розробки. Докладний аналіз рушіїв розробки не тільки

дозволяє обрати кращий для себе інструмент, а й одразу визначити, які аспекти розробки можуть бути найскладнішими. Такий підхід надає змогу одразу оцінити приблизний час виконання окремих стадій розробки, та допоможе оптимізувати цілком увесь процес. Тому далі я наведу більш докладний аналіз переваг та недоліків обраного рушія:

Переваги:

- Дуже потужна система кросплатформеності.
- Гнучка графічна компонентно-орієнтована система для розробки ігрової логіки.
- Велика відкритість до розширення. Є інструменти для розширення функціоналу візуального середовища, функціоналу систем рендерінгу, функціонал будування проекту під платформи та багатьох інших ситем. У платній версії рушій має повністю відкритий код.
- Ефективна оптимізація розміру додатку. Unity має довгий перелік методів, які можуть значно оптимізувати кількість пам’яті, яку займає ваш додаток. На вибір є декілька методів стиснення графічних асетів. Пристунє автоматичне виключення із зборки невживаних асіетів, та тих, які стосуються тільки Unity Editor. Є підтримка зручних директив для динамічної компіляції файлів. Додатково Unity надає потужний інструмент для розділення проекту на окремі бандли, що дозволяє динамічно додавати та прибирати контент вже після вивантаження у магазини.
- Власні рішення у симуляції фізики. Це значно прискорює створення маленьких проектів.
- Вбудований Profiler для збору та аналізу статистики по технічним параметрам під час роботи вашого проекту. Profiler дозволяє побачити доволі докладні дані про швидкість виконання вашого коду, процесу рендерінгу, мережевих процесів і використання пам’яті. Цей інструмент значно полегшує виявлення слабких місць,

визначення перешкод та проведення оптимізації.

- Інструмент ShaderGraph, який дозволяє створювати шейдери оперуючи компонентами у візуальному середовищі, тобто без написання коду.
- Величезний обсяг додаткових інструментів, систем та корисних даних, які можна завантажити з єдиного середовища UnityAssetStore. Інші користувачи мають змогу створювати і завантажувати у AssetStore власні рішення, та поширювати їх безкоштовно або у якості товару з фіксованою ціною.
- Підтримка нових версій C#. Наразі найновіша підтримувана версія – C# 9.0
- Зручне та оптимальне рішення для збереження, налаштування та передачі даних у вигляді ScriptableObject.
- Дуже велика та активне ком’юніті. Не існує проблеми, яка б не була вже кимось опрацьована або вирішена. Багато інформації можна знайти на офіційних форумах, які досить зручно побудовані. Також Unity Trchnologies мають активну команду розробників, яка нараховує близько 900 осіб, має якісний відділ підтримки та постійно комунікує на тих самих форумах.

Недоліки:

- Великий ступінь автоматизації у багатьох випадках призводить до надлишкових рішень, які обтяжують проект. Яскравим прикладом є MonoBehaviour. Це наданий Unity базовий клас для будь якого об’єкту що має бути представлений на сцені, та повинен взаєиодіяти з вбудованими в Unity функціями. Цей клас містить надто багато вбудованих можливостей, які доволі важко відключити. Наприклад довга серія вбудованих відгуків на event-функції, які тобі можуть бути і не потрібними, значно сповільнюють хід виконання програми. Тому кількість нащадків MonoBehaviour розробники намагаються мінімізувати, натомість створюючи власні рішення.

- Відсутність нормальної підтримки багатопоточності. Увесь життєвий цикл будь якого об’єкту в Unity протікає у єдиному main потоці. Наразі Unity розвиває системи для підтримки параллельного програмування у вигляді DOTS системи, яка вже продемонструвала свою значну потужність у сфері оптимізації обчислень. При цьому самі користувачі вже давно запропонували свої рішення для використання стандартних практик паралельного програмування у C#. Наприклад плагін UniTask, який позиціонується як zero-allocation async tasks.
- Використання так званого stop-the-world garbage collector. Цей збирач сміття під час виконання своїх дій повністю зупиняє роботу проекту. Тому розробники на Unity дуже уважно контролюють потік даних у своїх системах і намагаються розподілити збір сміття на декілька кадрів.
- Дуже стрімкий розвиток рушія призводить до майже постійної неактуальності версії вашого проекту, тому робота з Unity потребує доволі частого перенесення проекту на нові версії рушія.
- Unity написаний частково на C++ та частково на JavaScript. При цьому основною мовою програмування у рушії є C#. В результаті, написаний розробником код на C# при кінцевому будуванні проекту проходить комплексний процес інтерпритації у C++ з використанням рефлексії. Це не має серйозного прямого впливу на швидкість отриманого додатку, проте досвідчений розробник повинен знати тонкощі інтерпритації C# у C++, аби використати цей фактор для мікрооптимізацій.

Рушій для створення відеоігр, це дуже комплексна програма, яка потребує глибокого вивчення. Висока експертиза та знання в області використання ігрових рушіїв високо ціниться на ринку праці, адже цей фактор є показником великого досвіду. Будь який розробник мусить спробувати розробити невеличкий pet-

project на декількох різних рушіях, адже це значно поглиблює розуміння того як працюють ці програми та підвищує ефективність розробки відеогри за допомогою будь якого ПЗ.

Під час розробки за допомогою Unity я буду використовувати усі можливості потужної мови C#, зручні інструменти рушія для створення анімацій та VFX і підтримувати баланс між чистими класами та класами MonoBehaviour.

РОЗДІЛ 3 ПРОЕКТУВАННЯ ТА РОЗРОБКА КОМП'ЮТЕРНОЇ ГРИ НА РУШІЇ UNITY

3.1 Постановка технічних та функціональних вимог

В рамках дипломного проекту було обрано розробити покрокову гру казуального жанру, яка може працювати на декількох платформах.

Опишемо модель гри по її основних структурних блоках:

- Віртуальний простір – рівень у вигляді пласкої мапи з брусками в якості перепон. Бруски обмежують певні території. На двох таких територіях стоять початкові позиції гравців.
- Головна механіка – покроковий процес, де кожний гравець на своєму кроці прибирає брусок з метою захопити більшу територію. Коли між територіями гравців не залишається перепон – іде зіткнення та підрахунок результатів.
- Умови перемоги – перемагає той гравець, який захопив більшу територію до моменту зіткнення з опонентом.

Функціональні вимоги:

- **Зручний дизайнер рівнів**

Гра буде мати декілька рівнів, потенційно необмежену кількість. Тому необхідно забезпечити легке конструювання рівня. Важлива стандартизація метрик та структурних блоків рівня.

- **Ігровий UI**

Гравець повинен орієнтуватися у грі. Необхідно відображати поточний прогрес (номер рівня), інформацію про результат ігрової сесії (перемога або поразка).

- **Покрокова система**

Необхідно забезпечити покрокову систему гри. Під час кроку опонента гравець не повинен мати можливість зробити власний крок або завадити іншим чином.

- **Опонент керований комп’ютером.**

Людина повинна мати змогу грати з комп’ютером. При цьому необхідно спроектувати програму так, щоб комп’ютерного опонента було легко замінити на опонента-гравця, адже у гри є онлайн потенціал.

- **Підказки у процесі гри**

В казуальних іграх краще впроваджувати навчання механікам одразу в ігровий процес. Тому необхідно реалізувати підказки, якщо гравець довго не може зробити свій хід.

- **Забезпечення 30-60 FPS**

Дослідження показують, що повний цикл обробки інформації у людини займає 240мс. Умовно людське сприйняття інформації можна поділити на 3 процесори: перцептивний, когнітивний та моторний.

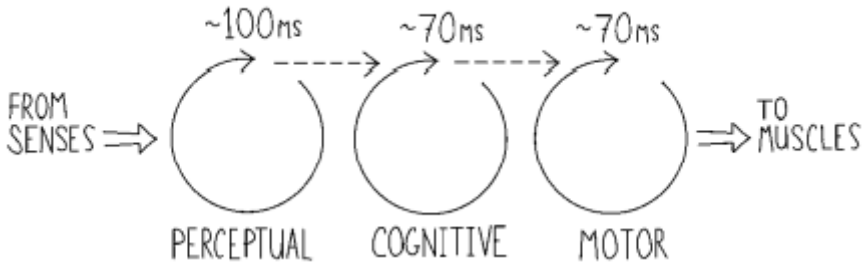


Рисунок 3.1 - Умовні процесори людського сприйняття

Також відомо що людина вже помічає підвисання гри якщо частота кадрів нижче ніж 30 за секунду. Допустимі три кадри затримки при 30fps займуть 3/30*1000 = 100 мс, тобто на практиці бар’єр не 240мс а 100 мс. Причина у тому, що усі три умовні ділянки обробки інформації (сприйняття – 100мс, усвідомлення – 70мс, моторика – 70мс) працюють не послідовно а одночасно. Через це, коли усвідомлення першого блоку інформації завершилось, сприйняттю другого залишилось не 100мс, а лише 30мс. Тому необхідно забезпечити стабільні 30-60 fps.

- **Мультиплатформеність**

У сучасному світі великий обсяг ігрового ринку припадає саме на мобільні платформи, особливо у казуальному секторі. Через це необхідно аби розроблюваний додаток працював як на комп’ютері, так і на телефонах різних поколінь. В рамках практики орієнтуємось на платформи Android.

- **Додаткові відчуття від додатку**

Значну роль в успіху казуальної гри має додатковий полішинг застосунку. Обраний мною додаток повинен містити візуальні ефекти та застосовувати вібрацію на мобільних платформах.

3.2 Імплементація основних модулів гри у відповідності до поставлених вимог.

Під час проходження практики мені вдалося досягти зручного конструювання рівнів гри, імплементувати основні механіки, запустити проект на комп’ютері та оптимізувати його під мобільні платформи. Також вдалося застосувати вібраційні властивості телефонів.

Конструювання рівнів

В основі конструювання рівнів буде клітина. Клітина одночасно послужить і блоком будівництва рівня і чанком для гравців.

```
1 asset usage  21 usages  DrMichaelUm *  2 exposing APIs
public class GroundCell : MonoBehaviour
{
    [ReadOnly]
    public GroundCellType Type = GroundCellType.Free;  Unchanged

    3 usages  DrMichaelUm *
    public void MarkAs(GroundCellType type)
    {
        Type = type;
    }

    1 usage  DrMichaelUm *
    public bool IsFreeAndReachable(NavGraph graph, Vector3 from) =>
        Type == GroundCellType.Free && IsReachable(graph, from);

    2 usages  DrMichaelUm *
    public bool IsReachable(NavGraph graph, Vector3 from)
    {
        GraphNode fromNode = graph.GetNearest(from, NNConstraint.Default).node;
        GraphNode toNode = graph.GetNearest(transform.position, NNConstraint.Default).node;

        return PathUtilities.IsPathPossible(fromNode, toNode);
    }
}
```

Рисунок 3.2 - Програмне представлення клітини

На деяких клітинах будуть розміщатись бар’єри. Бар’єр обмежує певну область. Прибравши бар’єри кожен гравець може захопити територію, яку він обмежував.

І останнім елементом є тенти гравців, які позначають початкові клітини, та відповідно початкову територію.

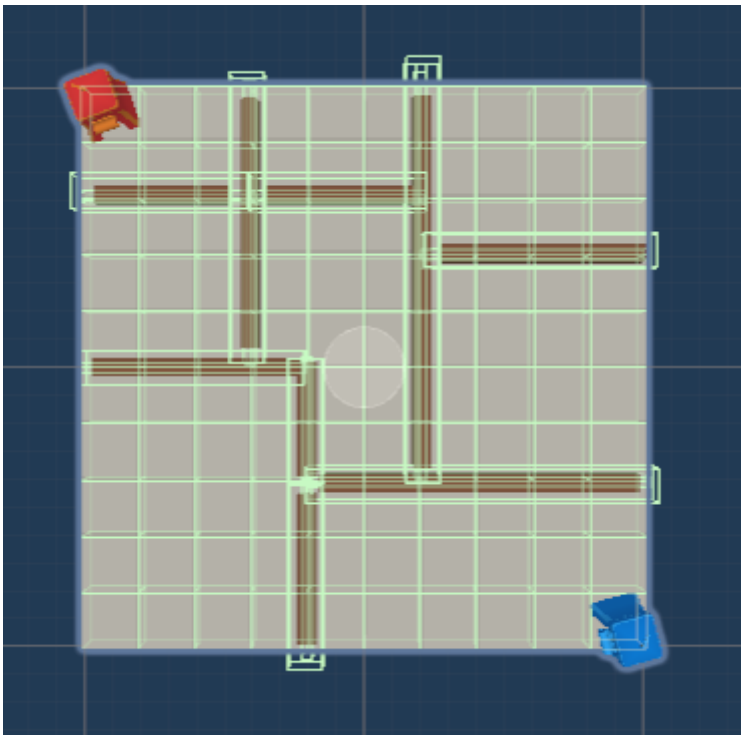


Рисунок 3.3 - Приклад готового рівня в середовищі Unity

Механіка захоплення клітин

Основна механіка – прибирання бар’єрів для захоплення нових клітин. Задля її роботи спочатку реалізуємо клас PlayerInput, який буде відповідати за зчитування вхідного контролю гравця та прибирати блоки. Для того щоб визначити на який блок гравець хотів прибрати використаємо технологію Raycast та інтерполяцію координат на площину перспективної камери.

Після того як бар’єр прибраний програма аналізує нові відкриті клітини, і якщо вони мають статус незайнятих, то отримують статус захоплених гравцем. Цей процес працює за допомогою алгоритму A* для «пошуку шляху». Після відкриття нової області ми обходимо по графу всі клітини та перевіряємо їх статус. Поле гри має фіксовану площину тому граф можна налаштувати заздалегідь для більшої оптимізації.

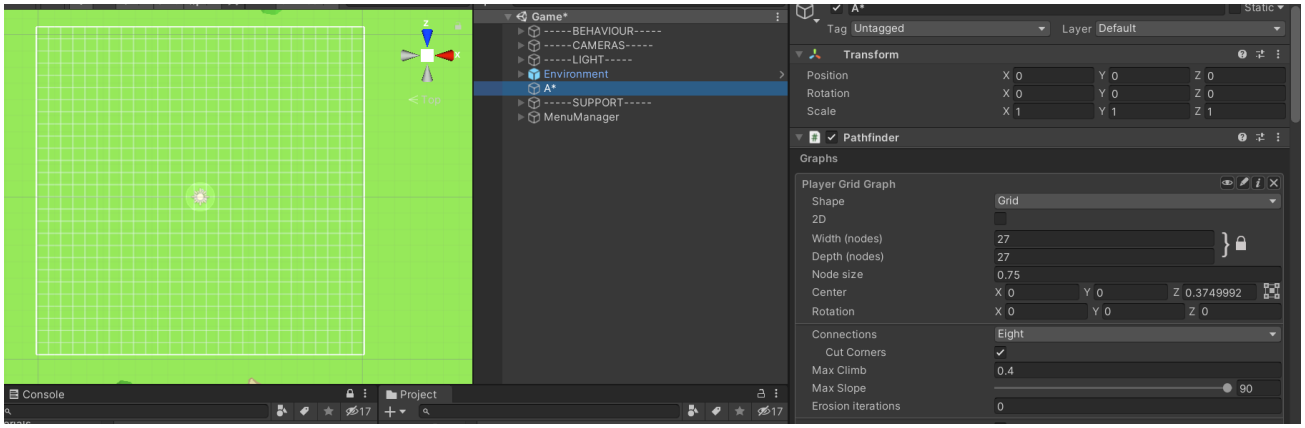


Рисунок 3.4 - Сітка для алгоритму A*

Оптимізація під мобільні платформи

По захопленій території розходяться велика кількість юнітів гравця, а під час зіткнення з опонентом всі ці юніти мають битись. На кінцевих етапах партії на полі може знаходитись до 200 юнітів, тому перебирати прямим проходом 40000 операцій для визначення цілей раз на кілька кадрів буде недоцільно. Наші чанки у вигляді клітин могли б спростити ситуацію, але пошук найближчих ворожих клітин теж досить коштовна операція. А після визначення цілей юніт повинен ще й бігти в задану позицію.

Для вирішення цієї проблеми вдало підходить метод, який я розглянув раніше – KDTree (k-dimentional tree). Під час його аналізу було визначено, що його основна область використання - системи колізій в іграх. Проте головне абстрактне завдання, яке виконує алгоритм - пошук двох найближчих сусідів, тож я використаю його для пошуку найближчого ворожого юніта. У саме дерево я буду додавати класи юнітів, а дерево буде оперувати їх позиціями у тривимірному просторі.

Пошук найближчого сусіда по KDTree має обчислювальну складність O(nlog(n)), що значно прискорить пошук для юнітів. Додавання елементу до дерева теж не є сильно коштовною операцією, його обчислювальна складність O(h), де h – висота дерева. Тобто чим більше юнітів в одному вузлі, тим швидше буде працювати дерево. Ця залежність добре вписується в мій проект адже юніти досить тісно стоять і з великою вірогідність будуть додаватись під один вузол.

Покращення відчуття від додатку

Основними засобами для покращення відчуття від ігор є анімації, візуальні ефекти, стерео звук, застосування вібровідгуку та інших унікальних для платформи властивостей (наприклад опір тригерів на джойстику PS5).

Для створення VFX в Unity є певний набір інструментарію. Я буду використовувати Particle System, цей інструмент дозволяє створити анімовані візуальні ефекти на основі система частинок. Мені необхідно було створити анімоване конфетті, яке б сипалось згори, якщо гравець переміг. Для цього мені потрібно створити матеріал на основі текстури конфетті, один об’єкт з таким матеріалом і буде нашою частинкою (particle). Далі створимо об’єкт з компонентом ParticleSystem, де будемо оперувати такими частинами.

Тепер я можу виставити кількість частинок, які будуть з’являтись кожну секунду або кадр, форму їх розповсюдження, розмір, траєкторію руху. До того ж всі ці параметри можна рандомізувати та анімувати.

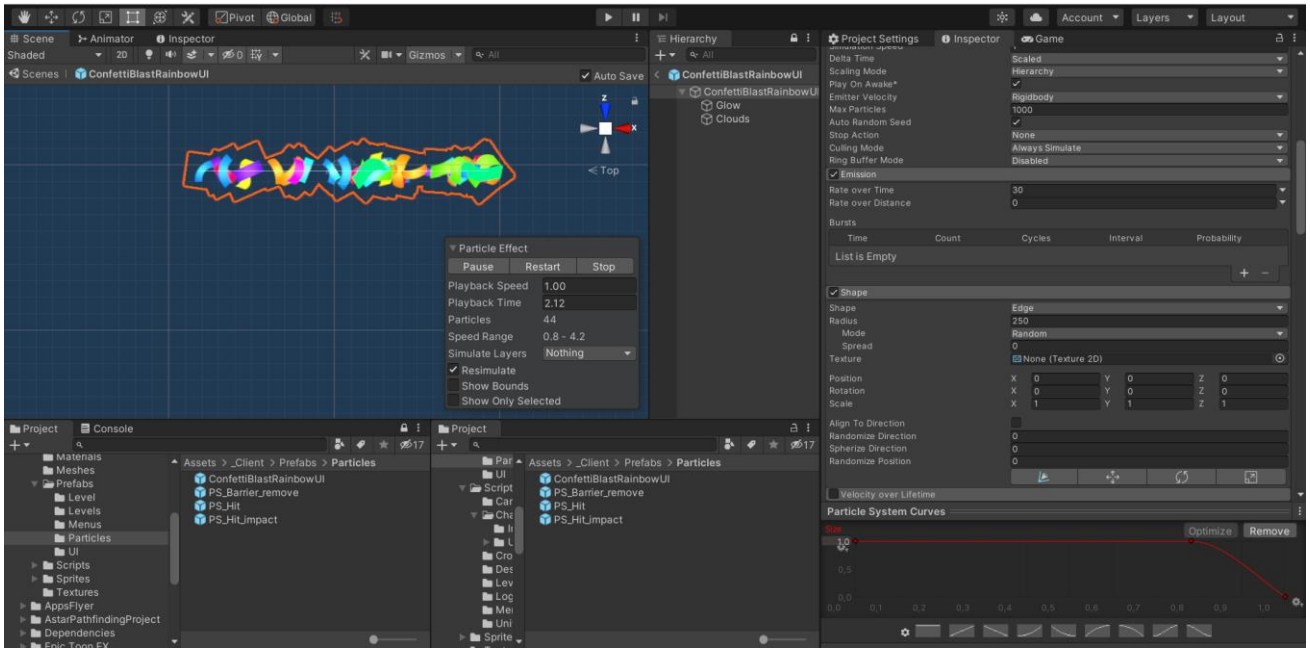


Рисунок 3.5 - Налаштування Particle System

ВИСНОВОК

Отже, у результаті виконання кваліфікаційної роботи бакалавра була створена комп’ютерна гра на рушії Unity. На етапі підготовки до розроблення додатку було проведено перелік досліджень, а саме:

- Були досліджені найефективніші моделі побудови програмних систем у відеоігровій розробці.
- Були проаналізовані найпоширеніші проблеми та завдання, що зустрічаються у процесі розробки комп’ютерних ігор. Для кожної проблеми я визначив перелік можливих рішень та виокремив найефективніші з них.
- На другому етапі підготовки я затвердив перелік необхідних інструментів для створення відеогри. Додатково були проведені дослідження найважливіших з них – систем управління проектами, систем контролю версій та ігрових рушіїв.
- Я виконав порівняльний аналіз двох найпопулярніших ігрових рушіїв – Unity та Unreal Engine. Після цього я провів глибоке дослідження переваг та недоліків Unity Engine, який я використовував для розробки своєї гри.

Проведені дослідження допомогли мені скорегувати технічне завдання, в залежності від наявного інструментарію рушія, обмежитись певним стеком технологій для створення власної гри і прискорити розробку. В результаті мені вдалося реалізувати основні механіки, розробити досить зручний і лаконічний UI, використати велику кількість інструментів Unity для покращення системи контролю гравця та удосконалення зовнішнього вигляду додатку. Також я зміг використати декілька зовнішніх рішень, які представлені у Unity Asset Store, для покращення гри, таких як плагін для інтеграції вібровідгуків телефона.

Розроблена гра працює на комп’ютері з операційною системою Windows та телефонах з операційною системою Android.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Хокінг, Джозеф. Unity — у дії. Мультиплатформенна розробка на C# : - 2ге вид., 2016. — 336 с.
2. Design Patterns: elements of reusable object-oriented software. Erich Gamma, Richard Helm, Ralph Jonson, John Vissides. Addison-Wesley, Reading, MA. NEXTSTEP General Reference: Release 3, Volumes 1 and 2, 1994. – 431 с.
3. Simon LH. SOLID Principles: Explanation and examples [Електронний ресурс], URL: https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4
4. Wolf M. J. P. Encyclopedia of Video Games: The Culture, Technology, and Art of Gaming. Ed. by Mark J. P. Wolf. Santa Barbara, – CA: Greenwood, 2012. – 740 с.
5. Continuous collision detection (CCD) [Електронний ресурс], URL: https://docs.unity3d.com/Manual/ContinuousCollisionDetection.html
6. Swink, Steve. Game feel: a game designer’s guide to virtual sensation. ISBN: 978-0-12-374328-2 / Printed in the United States of America. – 358 с.
7. Патерни/шаблони проектування [Електронний ресурс], URL: https://refactoring.guru/uk/design-patterns
8. Five Things to Know About Reactive Programming [Електронний ресурс], URL: https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming
9. Cleaner code with functional programming [Електронний ресурс], URL: https://web.archive.org/web/20200728013926/https://wimvanderbauwhede.github.io/articles/decluttering-with-functional-programming/
10. Inversion of Control Containers and the Dependency Injection pattern [Електронний ресурс],

URL: https://martinfowler.com/articles/injection.html

11. Institute for Advanced Study Homotopy Type Theory: Univalent Foundations of Mathematics. Princeton, 2013. — 603 с.
12. Benjamin C. Pierce Types and Programming Languages. The MIT Press,

2002. – 648 с.

13. EMS Press "Normal Distribution", Encyclopedia of Mathematics, 2001 [Електронний ресурс],

URL: https://encyclopediaofmath.org/wiki/Main_Page

14. Delaunay Triangulation, [Електронний ресурс], URL: https://mathworld.wolfram.com/DelaunayTriangulation.html
15. Franco P. Preparata and Michael I. Shamos Computational Geometry: An Introduction. Springer, 1985. – 390 с.
16. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein Introduction to Algorithms. MIT Press, 1990. – 1312 с.
17. Circumsphere, [Електронний ресурс], URL: https://mathworld.wolfram.com/Circumsphere.html
18. Ben Carter. The Game Asset Pipeline. — 1-ше вид. — Charles River Media, 2004. — 302 с.
19. Pierre L`ecuyer Efficient and Portable Combined Random Number Generators. Communications of ACM, 1988 p. 742-774 с.
20. Donald Knuth The Art of Computer Programming Vol-2. Addison-Wesley, 1968, 10-26 с.
21. Unity Physics Documentation, [Електронний ресурс], URL: https://docs.unity3d.com/ru/530/Manual/PhysicsSection.html
22. Broad Phase Collision Detection – Bounding Volume Hierarchies, [Електронний ресурс], URL: https://thegeneralsolution.wordpress.com/2011/12/13/broad-phase-collision-detection-bounding-volume-hierarchies-1/
23. Continuous Collision Detection (Background Information), [Електронний ресурс], URL: https://digitalrune.github.io/DigitalRune-Documentation/html/138fc8fe-c536-40e0-af6b-0fb7e8eb9623.htm
24. Lyssa Adkins Coaching Agile Teams: A Companion for Scrum Masters, Agile Coaches, and Project Managers in Transition. Addison-Wesley, 2010 p. – 352 с.

25. Abramson, M.; Moser, W. O. J. More Birthday Surprises. The American Mathematical Monthly, 1970 p. – 856 – 858 c.
26. Jeff Preshing Hash Collision Probabilities. [Електронний ресурс], URL: https://preshing.com/20110504/hash-collision-probabilities/
27. Jon Loeliger and Matthew McCullough Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development. O'Reilly Media, 2012 p. – 456c.
28. Game engine comparison guide for 2021. [Електронний ресурс], URL: https://www.evercast.us/blog/unity-vs-unreal-engine
29. Made with Unity. [Електронний ресурс], URL: https://unity.com/ru/madewith
30. Adding Global Shaders to Unreal Engine. [Електронний ресурс], URL: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Rendering/ShaderDevelopment/AddingGlobalShaders/#:~:text=Unreal%20Engine%20uses%20Unreal%20Shader,the%20Plugin%2FShaders%20folder%20instead.
31. Unreal Build System. [Електронний ресурс], URL: https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/UnrealBuildSystem/
32. Procedural Generation: Programming The Universe. [Електронний ресурс], URL: https://www.youtube.com/watch?v=ZZY9YE7rZJw
33. Andre Medeiros The introduction to Reactive Programming. [Електронний ресурс], URL: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
34. Richard Lord Why use an Entity Component System architecture for game development? [Електронний ресурс], URL: https://www.richardlord.net/blog/ecs/why-use-an-entity-framework.html

ДОДАТКИ

ДОДАТОК А

```
using System.Collections.Generic;
using System.Threading.Tasks;
using CrowdT.Level;
using Pathfinding;
using UnityEngine;

namespace CrowdT
{
    public abstract class UnitsSpawner : MonoBehaviour
    {
        [SerializeField] private SingleUnit unitPrefab;

        [SerializeField] private float spawnInterval;

        private LevelController _levelController;

        private void Awake()
        {
            _levelController = FindObjectOfType<LevelController>();
        }

        public async void Spawn(int amount, Vector3 targetPos)
        {
            for (var i = 0; i != amount; ++i)
            {
                var cooldown = 0f;

                while (cooldown < spawnInterval)
                {
                    cooldown += Time.deltaTime;
                    await Task.Yield();
                }

                var unit = Instantiate(unitPrefab, transform.position,
Quaternion.identity);
                _levelController.crowdController.RegisterUnit(unit);

                unit.pathData.path = MakePath(new
Vector3(transform.position.x, 0, transform.position.z), targetPos);
            }
        }

        // calculates a path and copies it from A*PFP's vectorPath to a
FixedList4096
        private List<Vector3> MakePath(Vector3 fromPos, Vector3 toPos)
        {
            var destPath = new List<Vector3>();

            var unitGraph = GetUnitGraph();

            var fromNode = unitGraph.GetNearest(fromPos,
NNConstraint.Default).node;
            var toNode = unitGraph.GetNearest(toPos,
NNConstraint.Default).node;
```

```
            if (PathUtilities.IsPathPossible(fromNode, toNode))
            {
                ABPath path = null;
                path = ABPath.Construct((Vector3) fromNode.position,
(Vector3) toNode.position);
                AstarPath.StartPath(path);
                path.BlockUntilCalculated();

                for (var i = 0; i != path.vectorPath.Count; ++i)
                {
                    destPath.Add(path.vectorPath[i]);
                }
            }

            return destPath;
        }

        protected abstract NavGraph GetUnitGraph();
    }
}
```

Клас, що дозволяє створювати юнітів на ігровому полі

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Sirenix.OdinInspector;
using UnityEngine;

namespace CrowdT
{
    [Serializable]
    public class PathData
    {
        public List<Vector3> path;
        public int waypointIndex;
        public bool reachedEndOfPath;
    }

    public class SingleUnit : MonoBehaviour
    {
        [SerializeField] private float pathSpeed;
        [SerializeField] private float battleSpeed;

        [HideInInspector] public int health = 1;
        [SerializeField] private int damage;
        [ReadOnly] public bool isDead;

        [SerializeField] protected SingleUnitView view;

        [ReadOnly] public PathData pathData;

        [ReadOnly] public SingleUnit targetUnit;
        [ReadOnly] public SingleUnit attackedBy;

        [HideInInspector] public bool targetsExist = true;
```

```
        private MovementByPath _movementByPath;
        private MovementToTarget _movementToTarget;
        [HideInInspector] public CrowdController crowdController;

        private void Awake()
        {
            _movementByPath = GetComponent<MovementByPath>();
            _movementToTarget = GetComponent<MovementToTarget>();
        }

        private void OnEnable()
        {
            _movementByPath.OnStart += () => view.SetIdle(false);
            _movementByPath.OnComplete += () =>
crowdController.unitsOccupiedTheirCellsAmount++;
            _movementByPath.OnComplete += () => view.SetIdle(true);
            _movementToTarget.OnComplete += Attack;
        }

        private async void Start()
        {
            await _movementByPath.Move(pathData, pathSpeed);
        }

        public void FindTarget()
        {
            if (isDead) return;

            targetUnit = FindClosestTarget();

            if (!targetUnit)
            {
                targetsExist = false;

                return;
            }

            targetUnit.attackedBy = this;

            StartCheckingForTargetAlive();
            MoveToTargetUnit();
        }

        protected virtual SingleUnit FindClosestTarget() => null;

        private async void StartCheckingForTargetAlive()
        {
            while (targetsExist && !targetUnit.isDead)
            {
                await Task.Yield();
            }

            view.SetIdle(true);

            if (targetsExist)
                FindTarget();
        }
```

```
private async void MoveToTargetUnit()
{
    if (targetUnit == null) return;

    view.SetIdle(false);

    await _movementToTarget.MoveTo(targetUnit, battleSpeed);
}

private async void Attack()
{
    view.SetIdle(true);
    view.SetAttack();

    const int delayBetweenHits = 1;

    while (!isDead)
    {
        await new WaitForSeconds(delayBetweenHits);

        if (targetUnit)
            Hit();
    }

    FindTarget();
}

protected virtual void Hit()
{
    targetUnit.TakeDamage(damage);
}

private void TakeDamage(int amount)
{
    health -= amount;
    if (health <= 0)
        Die();
}

private void Die()
{
    if (isDead) return;

    view.SetRandomDeath();

    isDead = true;

    crowdController.UnregisterUnit(this);
}

public void Win()
{
    view.SetRandomDance();
}

private void OnDisable()
{
    _movementByPath.OnStart -= () => view.SetIdle(false);
```

```
            _movementByPath.OnComplete -= () => view.SetIdle(true);
            _movementByPath.OnComplete -= () =>
crowdController.unitsOccupiedTheirCellsAmount++;
            _movementToTarget.OnComplete -= Attack;
        }
    }
}
```

Клас логіки юніта

```
using Sirenix.OdinInspector;
using Unity.Mathematics;
using UnityEngine;
using Random = UnityEngine.Random;

namespace CrowdT
{
    public class SingleUnitView : MonoBehaviour
    {
        [SerializeField] private Renderer bodyRenderer;
        [SerializeField] private Material deathMaterial;

        [SerializeField] private Renderer weapon;

        [BoxGroup("HitParticles")]
        [SerializeField] private ParticleSystem hitParticles;
        [BoxGroup("HitParticles")]
        [SerializeField] private Transform hitParticlesSpawner;

        [BoxGroup("HitImpactParticles")]
        [SerializeField] private ParticleSystem hitImpactParticles;
        [BoxGroup("HitImpactParticles")]
        [SerializeField] private Transform hitImpactParticlesSpawner;

        private Animator _anim;

        private static readonly int Idle = Animator.StringToHash("Idle");
        private static readonly int Attack =
Animator.StringToHash("Attack");
        private static readonly int Death =
Animator.StringToHash("Death");
        private static readonly int DeathVariant =
Animator.StringToHash("DeathVariant");
        private static readonly int Dance =
Animator.StringToHash("Dance");
        private static readonly int DanceVariant =
Animator.StringToHash("DanceVariant");

        private void Awake()
        {
            _anim = GetComponent<Animator>();
        }

        public void SetIdle(bool value)
        {
            _anim.SetBool(Idle, value);
        }
```

```
        public void SetAttack()
        {
            _anim.SetTrigger(Attack);
        }

        public void SetRandomDeath()
        {
            var variant = Random.Range(0, 3);
            _anim.SetInteger(DeathVariant, variant);
            _anim.SetTrigger(Death);

            ChangeColorToDeath();
        }

        private void ChangeColorToDeath()
        {
            bodyRenderer.material = deathMaterial;

            weapon.gameObject.SetActive(false);
        }

        public void PlayHit()
        {
            var particle = PoolParty.Instantiate(hitParticles,
hitParticlesSpawner.position, quaternion.identity);
            PoolParty.Destroy(particle,
hitParticles.main.startLifetimeMultiplier);
        }

        public void PlayHitImpact()
        {
            var particle = PoolParty.Instantiate(hitImpactParticles,
hitImpactParticlesSpawner.position, quaternion.identity);
            PoolParty.Destroy(particle,
hitImpactParticles.main.startLifetimeMultiplier);
        }

        public void SetRandomDance()
        {
            var variant = Random.Range(0, 3);
            _anim.SetInteger(DanceVariant, variant);
            _anim.SetTrigger(Dance);
        }

        public void SetDance(int variant)
        {
            _anim.SetInteger(DanceVariant, variant);
            _anim.SetTrigger(Dance);
        }

        public float GetNormalizedTime (int layer = 0)
        {
            return _anim.GetCurrentAnimatorStateInfo
(layer).normalizedTime % 1f;
        }
    }
}
```

Клас візуалу юніта

```
using CrowdT.Level;
using UnityEngine;

namespace CrowdT
{
    public class CrowdController : MonoBehaviour
    {
        public static int TotalPlayerUnitsHitsDuringLastSecond;

        public int unitsPerCell;
        public Vector3 unitsOffset;

        public KdTree<SingleUnit> PlayerUnits = new KdTree<SingleUnit>();
        public KdTree<SingleUnit> EnemyUnits = new KdTree<SingleUnit>();

        [HideInInspector]
        public int unitsOccupiedTheirCellsAmount;
        [HideInInspector]
        public int playerUnitsAliveAmount;
        [HideInInspector]
        public int enemyUnitsAliveAmount;

        private LevelController _levelController;

        private void Awake()
        {
            _levelController = FindObjectOfType<LevelController>();

            TotalPlayerUnitsHitsDuringLastSecond = 0;
        }

        public void RegisterUnit(SingleUnit unit)
        {
            if (unit is PlayerUnit)
            {
                PlayerUnits.Add(unit);
                playerUnitsAliveAmount++;
            }

            else if (unit is EnemyUnit)
            {
                EnemyUnits.Add(unit);
                enemyUnitsAliveAmount++;
            }

            unit.crowdController = this;
        }

        public void UnregisterUnit(SingleUnit unit)
        {
            if (unit is PlayerUnit)
            {
                playerUnitsAliveAmount--;
            }
```

```
            else if (unit is EnemyUnit)
            {
                enemyUnitsAliveAmount--;
            }

            _levelController.TryEndCrowdBattle();
        }

        public void StartBattle()
        {
            for (var i = 0; i != PlayerUnits.Count; ++i)
            {
                PlayerUnits[i].FindTarget();
            }

            for (var i = 0; i != EnemyUnits.Count; ++i)
            {
                EnemyUnits[i].FindTarget();
            }
        }

        public void IncreaseWinnerHealth(Character winner)
        {
            // This method need to be ensure that winner units will be
alive at the end of the battle
            // Because for now truly defeated units (in cells
calculating) can win in the battle due to simple crowd battle logic.
            // Simple crowd battle logic means that multiple units can
attack single unit.
            if (winner is Player)
            {
                for (var i = 0; i != PlayerUnits.Count; ++i)
                    PlayerUnits[i].health = 10;

                for (var i = 0; i != EnemyUnits.Count; ++i)
                    EnemyUnits[i].health = 2;
            }

            else if (winner is Enemy)
            {
                for (var i = 0; i != EnemyUnits.Count; ++i)
                    EnemyUnits[i].health = 10;

                for (var i = 0; i != PlayerUnits.Count; ++i)
                    PlayerUnits[i].health = 2;
            }
        }

        public void PlayPlayerCrowdVictory()
        {
            for (var i = 0; i != PlayerUnits.Count; ++i)
            {
                if (!PlayerUnits[i].isDead)
                    PlayerUnits[i].Win();
            }
        }

        public bool HaveAllUnitsOccupiedCells() =>
```

```
            (unitsOccupiedTheirCellsAmount) >= (PlayerUnits.Count +
EnemyUnits.Count);
    }
}
```

Клас, що відповідає за групову поведінку юнітів на полі

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

namespace CrowdT
{
    public class KdTree<T> : IEnumerable<T> where T : Component
    {
        protected KdNode Root;
        protected KdNode Last;
        private int _count;
        private readonly bool _just2D;
        private float _lastUpdate = -1f;
        protected KdNode[] Open;

        public int Count => _count;

        public bool IsReadOnly => false;

        public float AverageSearchLength { protected set; get; }
        public float AverageSearchDeep { protected set; get; }

        /// <summary>
        /// create a tree
        /// </summary>
        /// <param name="just2D">just use x/z</param>
        public KdTree(bool just2D = false)
        {
            _just2D = just2D;
        }

        public T this[int key]
        {
            get
            {
                if (key >= _count)
                {
                    throw new ArgumentOutOfRangeException();
                }

                KdNode current = Root;

                for (int i = 0; i < key; i++)
                {
                    current = current.next;
                }

                return current.component;
            }
        }
```

```
/// <summary>
/// add item
/// </summary>
/// <param name="item">item</param>
public void Add(T item)
{
    Add(new KdNode() {component = item});
}

/// <summary>
/// batch add items
/// </summary>
/// <param name="items">items</param>
public void AddAll(List<T> items)
{
    foreach (T item in items)
    {
        Add(item);
    }
}

/// <summary>
/// find all objects that matches the given predicate
/// </summary>
/// <param name="match">lamda expression</param>
public KdTree<T> FindAll(Predicate<T> match)
{
    var list = new KdTree<T>(_just2D);

    foreach (T node in this)
    {
        if (match(node))
        {
            list.Add(node);
        }
    }

    return list;
}

/// <summary>
/// find first object that matches the given predicate
/// </summary>
/// <param name="match">lamda expression</param>
public T Find(Predicate<T> match)
{
    KdNode current = Root;

    while (current != null)
    {
        if (match(current.component))
        {
            return current.component;
        }

        current = current.next;
    }
```

```
    return null;
}

public void Remove(int i)
{
    var list = new List<KdNode>(GetNodes());
    list.RemoveAt(i);
    Clear();

    foreach (KdNode node in list)
    {
        node._oldRef = null;
        node.next = null;
    }

    foreach (KdNode node in list)
    {
        Add(node);
    }
}

/// <summary>
/// Remove at position i (position in list or loop)
/// </summary>
public void RemoveAt(int i)
{
    var list = new List<KdNode>(GetNodes());
    list.RemoveAt(i);
    Clear();

    foreach (KdNode node in list)
    {
        node._oldRef = null;
        node.next = null;
    }

    foreach (KdNode node in list)
    {
        Add(node);
    }
}

/// <summary>
/// remove all objects that matches the given predicate
/// </summary>
/// <param name="match">lamda expression</param>
public void RemoveAll(Predicate<T> match)
{
    var list = new List<KdNode>(GetNodes());
    list.RemoveAll(n => match(n.component));
    Clear();

    foreach (KdNode node in list)
    {
        node._oldRef = null;
        node.next = null;
    }
```

```
    foreach (KdNode node in list)
    {
        Add(node);
    }
}

/// <summary>
/// count all objects that matches the given predicate
/// </summary>
/// <param name="match">lamda expression</param>
/// <returns>matching object count</returns>
public int CountAll(Predicate<T> match)
{
    int count = 0;

    foreach (T node in this)
    {
        if (match(node))
        {
            count++;
        }
    }

    return count;
}

/// <summary>
/// clear tree
/// </summary>
public void Clear()
{
    //rest for the garbage collection
    Root = null;
    Last = null;
    _count = 0;
}

/// <summary>
/// Update positions (if objects moved)
/// </summary>
/// <param name="rate">Updates per second</param>
public void UpdatePositions(float rate)
{
    if (Time.timeSinceLevelLoad - _lastUpdate < 1f / rate)
    {
        return;
    }

    _lastUpdate = Time.timeSinceLevelLoad;

    UpdatePositions();
}

/// <summary>
/// Update positions (if objects moved)
/// </summary>
public void UpdatePositions()
```

```
{
    //save old traverse
    KdNode current = Root;

    while (current != null)
    {
        current._oldRef = current.next;
        current = current.next;
    }

    //save root
    current = Root;

    //reset values
    Clear();

    //readd
    while (current != null)
    {
        Add(current);
        current = current._oldRef;
    }
}

/// <summary>
/// Method to enable foreach-loops
/// </summary>
/// <returns>Enumberator</returns>
public IEnumerator<T> GetEnumerator()
{
    KdNode current = Root;

    while (current != null)
    {
        yield return current.component;
        current = current.next;
    }
}

/// <summary>
/// Convert to list
/// </summary>
/// <returns>list</returns>
public List<T> ToList()
{
    var list = new List<T>();

    foreach (T node in this)
    {
        list.Add(node);
    }

    return list;
}

/// <summary>
/// Method to enable foreach-loops
/// </summary>
```

```
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }

        protected float Distance(Vector3 a, Vector3 b)
        {
            if (_just2D)
            {
                return (a.x - b.x) * (a.x - b.x) + (a.z - b.z) * (a.z -
b.z);
            }
            else
            {
                return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y -
b.y) + (a.z - b.z) * (a.z - b.z);
            }
        }

        protected float GetSplitValue(int level, Vector3 position)
        {
            if (_just2D)
            {
                return level % 2 == 0 ? position.x : position.z;
            }
            else
            {
                return level % 3 == 0 ? position.x : level % 3 == 1 ?
position.y : position.z;
            }
        }

        private void Add(KdNode newNode)
        {
            _count++;
            newNode.left = null;
            newNode.right = null;
            newNode.level = 0;
            KdNode parent =
FindParent(newNode.component.transform.position);

            //set last
            if (Last != null)
            {
                Last.next = newNode;
            }

            Last = newNode;

            //set root
            if (parent == null)
            {
                Root = newNode;

                return;
            }

            float splitParent = GetSplitValueInternal(parent);
```

```
            float splitNew = GetSplitValue(parent.level,
newNode.component.transform.position);

            newNode.level = parent.level + 1;

            if (splitNew < splitParent)
            {
                parent.left = newNode; //go left
            }
            else
            {
                parent.right = newNode; //go right
            }
        }

        private KdNode FindParent(Vector3 position)
        {
            //travers from root to bottom and check every node
            KdNode current = Root;
            KdNode parent = Root;

            while (current != null)
            {
                float splitCurrent = GetSplitValueInternal(current);
                float splitSearch = GetSplitValue(current.level,
position);

                parent = current;

                if (splitSearch < splitCurrent)
                {
                    current = current.left; //go left
                }
                else
                {
                    current = current.right; //go right
                }
            }

            return parent;
        }

        /// <summary>
        /// Find closest object to given position
        /// </summary>
        /// <param name="position">position</param>
        /// <returns>closest object</returns>
        public T FindClosest(Vector3 position)
        {
            return FindClosestInternal(position);
        }

        /// <summary>
        /// Find close objects to given position
        /// </summary>
        /// <param name="position">position</param>
        /// <returns>close object</returns>
        public IEnumerable<T> FindClose(Vector3 position)
```

```
        {
            var output = new List<T>();
            FindClosestInternal(position, output);

            return output;
        }

        protected T FindClosestInternal(Vector3 position, List<T>
traversed = null)
        {
            if (Root == null)
            {
                return null;
            }

            float nearestDist = float.MaxValue;
            KdNode nearest = null;

            if (Open == null || Open.Length < Count)
            {
                Open = new KdNode[Count];
            }

            for (int i = 0; i < Open.Length; i++)
            {
                Open[i] = null;
            }

            int openAdd = 0;
            int openCur = 0;

            if (Root != null)
            {
                Open[openAdd++] = Root;
            }

            while (openCur < Open.Length && Open[openCur] != null)
            {
                KdNode current = Open[openCur++];

                if (traversed != null)
                {
                    traversed.Add(current.component);
                }

                float nodeDist = Distance(position,
current.component.transform.position);

                if (nodeDist < nearestDist)
                {
                    //todo: Quick fix
                    if (current.component is SingleUnit {isDead: false})
                    {
                        nearestDist = nodeDist;
                        nearest = current;
                    }
                }
```

```
                float splitCurrent = GetSplitValueInternal(current);
                float splitSearch = GetSplitValue(current.level,
position);

                if (splitSearch < splitCurrent)
                {
                    if (current.left != null)
                    {
                        Open[openAdd++] = current.left; //go left
                    }

                    if (Mathf.Abs(splitCurrent - splitSearch) *
Mathf.Abs(splitCurrent - splitSearch) < nearestDist &&
                        current.right != null)
                    {
                        Open[openAdd++] = current.right; //go right
                    }
                }
                else
                {
                    if (current.right != null)
                    {
                        Open[openAdd++] = current.right; //go right
                    }

                    if (Mathf.Abs(splitCurrent - splitSearch) *
Mathf.Abs(splitCurrent - splitSearch) < nearestDist &&
                        current.left != null)
                    {
                        Open[openAdd++] = current.left; //go left
                    }
                }
            }

            AverageSearchLength = (99f * AverageSearchLength + openCur) /
100f;

            if (nearest != null)
            {
                AverageSearchDeep = (99f * AverageSearchDeep +
nearest.level) / 100f;

                return nearest.component;
            }

            return null;
        }

        private float GetSplitValueInternal(KdNode node)
        {
            return GetSplitValue(node.level,
node.component.transform.position);
        }

        private IEnumerable<KdNode> GetNodes()
        {
            KdNode current = Root;
```

```
            while (current != null)
            {
                yield return current;
                current = current.next;
            }
        }

        protected class KdNode
        {
            internal T component;
            internal int level;
            internal KdNode left;
            internal KdNode right;
            internal KdNode next;
            internal KdNode _oldRef;
        }
    }
}
```

Реалізація KD-Tree для координат у тривимірному просторі. Використовується у CrowdController.

ДОДАТОК Б

```
using Pathfinding;
using Sirenix.OdinInspector;
using UnityEngine;

namespace CrowdT.Level
{
    public enum GroundCellType
    {
        Free,
        Obstacle,
        Player,
        Enemy
    }

    public class GroundCell : MonoBehaviour
    {
        [ReadOnly]
        public GroundCellType Type = GroundCellType.Free;

        public void MarkAs(GroundCellType type)
        {
            Type = type;
        }

        public bool IsFreeAndReachable(NavGraph graph, Vector3 from) =>
            Type == GroundCellType.Free && IsReachable(graph, from);

        public bool IsReachable(NavGraph graph, Vector3 from)
        {
            GraphNode fromNode = graph.GetNearest(from,
NNConstraint.Default).node;
            GraphNode toNode = graph.GetNearest(transform.position,
NNConstraint.Default).node;

            return PathUtilities.IsPathPossible(fromNode, toNode);
        }
    }
}
```

Клас – представлення одного чанку на полі

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

namespace CrowdT.Level
{
    public class GroundGrid : MonoBehaviour
    {
        [SerializeField] private Color playerCellsColor;
        [SerializeField] private Color enemyCellsColor;

        public GroundCell[] grid;
        [HideInInspector] public GroundCell playerCell;
        [HideInInspector] public GroundCell enemyCell;
```

```
        [HideInInspector] public int newOpenedCellsAmount;
        [HideInInspector] public int playerCellsAmount;
        [HideInInspector] public int enemyCellsAmount;

        private void Awake()
        {
            InitGroundGrid();
        }

        private void InitGroundGrid()
        {
            grid = FindObjectsOfType<GroundCell>();
        }

        public void ColorCellsByType(GroundCellType type, params
GroundCell[] cells)
        {
            if (type == GroundCellType.Player)
                ColorCells(playerCellsColor);

            else if (type == GroundCellType.Enemy)
                ColorCells(enemyCellsColor);

            void ColorCells(Color color)
            {
                for (var i = 0; i != cells.Length; ++i)
                {
                    cells[i].GetComponent<Renderer>().material.color =
color;
                }
            }
        }

        public GroundCell[] GetReachableCells(NavGraph graph, Vector3
from)
        {
            var reachableCells = new List<GroundCell>();

            for (var i = 0; i != grid.Length; ++i)
            {
                if (grid[i].Type != GroundCellType.Free) continue;

                if (!grid[i].IsFreeAndReachable(graph, from)) continue;

                reachableCells.Add(grid[i]);
            }

            return reachableCells.ToArray();
        }

        public GroundCell[] GetNewOpenedCellsIn(GroundCell[] cells)
        {
            var newCells = new List<GroundCell>();

            for (var i = 0; i != cells.Length; ++i)
            {
                if (cells[i].Type == GroundCellType.Free)
                    newCells.Add(cells[i]);
```

```
            }

            newOpenedCellsAmount = newCells.Count;

            return newCells.ToArray();
        }

        public bool IsBarrierBetweenArmies()
        {
            var totalGraph = AstarPath.active.data.graphs[2];
            totalGraph.Scan();

            return !playerCell.IsReachable(totalGraph,
enemyCell.transform.position);
        }

        public bool AreThereFreeCells()
        {
            for (var i = 0; i != grid.Length; ++i)
            {
                if (grid[i].Type == GroundCellType.Free)
                    return true;
            }

            return false;
        }
    }
}
```

Клас, що поєднує окремі клітини чанків у систему сітки

```
using Unity.Mathematics;
using UnityEngine;

namespace CrowdT.Level
{
    public class Barrier : MonoBehaviour, IObservable
    {
        [SerializeField]
        private ParticleSystem destroyParticle;
        [SerializeField]
        private float destroyParticlesOffset;

        private readonly List<IObserver> _observers = new
List<IObserver>();

        private BarrierGroup _group;
        private BarrierController _barrierController;

        private void Awake()
        {
            RegisterObserver(FindObjectOfType<PlayerInput>());

            _group = GetComponentInParent<BarrierGroup>();
            _barrierController = FindObjectOfType<BarrierController>();
        }

        public void RemoveWithNotification()
```

```
        {
            NotifyObservers();

            if (_group)
                RemoveGroup();
            else
                Remove();
        }

        private void RemoveGroup()
        {
            _group.RemoveTheGroupCompletely();
        }

        public void Remove()
        {
            SpawnParticles();

            _barrierController.Remove(this);

            Destroy(gameObject);
        }

        private void SpawnParticles()
        {
            if (destroyParticlesOffset == 0)
            {
                Debug.LogError("DestroyParticlesOffset is 0!");

                return;
            }

            var barrierLength = transform.localScale.z;

            var particlesAmount = (int) (barrierLength /
destroyParticlesOffset);

            var spawnPos = transform.position;

            spawnPos -= transform.forward * (barrierLength / 2 -
destroyParticlesOffset / 2);

            for (var i = 0; i != particlesAmount; ++i)
            {
                var particle = PoolParty.Instantiate(destroyParticle,
spawnPos, quaternion.identity);

                spawnPos += transform.forward * destroyParticlesOffset;

                PoolParty.Destroy(particle,
destroyParticle.main.startLifetimeMultiplier);
            }
        }

        public void RegisterObserver(IObserver observer)
        {
            _observers.Add(observer);
        }
```

```
        public void RemoveObserver(IObserver observer)
        {
            _observers.Remove(observer);
        }

        public void NotifyObservers()
        {
            for (var i = 0; i != _observers.Count; ++i)
            {
                _observers[i].UpdateObserver();
            }
        }
    }
}
```

Клас – представлення бар’єру на полі

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

namespace CrowdT.Level
{
    public class BarrierGroup : MonoBehaviour
    {
        private List<Barrier> _barriers;

        private void Awake()
        {
            _barriers = GetComponentsInChildren<Barrier>().ToList();
        }

        public void RemoveTheGroupCompletely()
        {
            for (var i = _barriers.Count - 1; i != -1; --i)
            {
                _barriers[i].Remove();
                _barriers.RemoveAt(i);
            }
        }
    }
}
```

Клас, що дозволяє групувати бар’єри і таким чином створювати їх складні форми

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

namespace CrowdT.Level
{
    public class BarrierController : MonoBehaviour
    {
        public List<Barrier> barriers;

        private void Awake()
```

```
        {
            barriers = GetComponentsInChildren<Barrier>().ToList();
        }

        public void RemoveAll()
        {
            Feedbacks.Play("SimpleRemoveBarrier", transform.position,
Vector3.zero);

            for (var i = barriers.Count - 1; i != -1; --i)
            {
                barriers[i].Remove();
            }
        }

        public void Remove(Barrier barrier)
        {
            barriers.Remove(barrier);
        }
    }
}
```

Клас, що дозволяє контролювати бар’єри на полі

```
using UnityEngine;

namespace CrowdT.Level
{
    public class GroundObstacle : MonoBehaviour
    {
        private enum ObstacleType
        {
            Obstacle,
            PlayerTent,
            EnemyTent
        }

        [SerializeField] private ObstacleType type;

        private void Start()
        {
            SetCellTypeBelow();
        }

        private void SetCellTypeBelow()
        {
            if (!Physics.Raycast(transform.position, Vector3.down, out
var hit)) return;

            var cell = hit.collider.GetComponent<GroundCell>();

            if (!cell) return;

            var groundGrid =
FindObjectOfType<LevelController>().groundGrid;

            switch (type)
            {
```

```
                case ObstacleType.Obstacle:
                    cell.MarkAs(GroundCellType.Obstacle);

                    break;

                case ObstacleType.PlayerTent:
                    cell.MarkAs(GroundCellType.Player);
                    groundGrid.ColorCellsByType(GroundCellType.Player,
cell);
                    groundGrid.playerCell = cell;
                    cell.gameObject.layer =
LayerMask.NameToLayer("PlayerGround");

                    break;

                case ObstacleType.EnemyTent:
                    cell.MarkAs(GroundCellType.Enemy);
                    groundGrid.ColorCellsByType(GroundCellType.Enemy,
cell);
                    groundGrid.enemyCell = cell;
                    cell.gameObject.layer =
LayerMask.NameToLayer("EnemyGround");

                    break;
            }
        }
    }
}
```

Клас – представлення перепони на полі. В проекті наразі сюди відносяться і палатки гравців.

```
using UnityEngine;

namespace CrowdT.Level
{
    public class TentView : MonoBehaviour
    {
        [SerializeField] private Animator anim;

        private static readonly int IsClose =
Animator.StringToHash("IsClose");

        public void OpenDoor()
        {
            anim.SetBool(IsClose, false);
        }

        public async void CloseDoor()
        {
            await new WaitForSeconds(1f);

            anim.SetBool(IsClose, true);
        }
    }
}
```

Клас, що відповідає за візуал палаток гравців

```
using System;
using System.Threading.Tasks;
using UnityEngine;

namespace CrowdT.Level
{
    public class LevelController : MonoBehaviour
    {
        [HideInInspector] public GroundGrid groundGrid;
        [HideInInspector] public CrowdController crowdController;
        [HideInInspector] public BarrierController barrierController;

        [HideInInspector] public Player player;
        [HideInInspector] public Enemy enemy;

        private Character _winner;

        public static bool IsEnd;

        public static bool CrowdBattleStarted = false;

        public Action OnLevelStart;
        public Action OnPlayerStartPlaying;
        public Action OnLevelEnd;
        public Action OnLevelSuccess;
        public Action OnLevelFail;

        private void Awake()
        {
            IsEnd = false;
            CrowdBattleStarted = false;
        }

        private void OnEnable()
        {
            OnPlayerStartPlaying += PlayerStartPlaying;
        }

        private void Start()
        {
            groundGrid = FindObjectOfType<GroundGrid>();
            crowdController = FindObjectOfType<CrowdController>();
            barrierController = FindObjectOfType<BarrierController>();

            player = FindObjectOfType<Player>();
            enemy = FindObjectOfType<Enemy>();

            MenuManager.Instance.HideAllMenusImmediately();
            MenuManager.Instance.MenuList.GameWindow.Show();

            OnLevelStart?.Invoke();
        }

        private void PlayerStartPlaying()
        {
            FirstPlayerTurnOnLevelMadeTracker.IsMade = true;
        }
```

```
        public async Task TryEndGame()
        {
            if (IsEnd) return;

            if (groundGrid.IsBarrierBetweenArmies()) return;

            IsEnd = true;

            while (!crowdController.HaveAllUnitsOccupiedCells())
                await Task.Yield();

            _winner = GetWinner();

            CrowdBattleStarted = true;
            barrierController.RemoveAll();
            crowdController.IncreaseWinnerHealth(_winner);
            crowdController.StartBattle();
        }

        private Character GetWinner()
        {
            if (groundGrid.playerCellsAmount >
groundGrid.enemyCellsAmount)
                return player;

            return enemy;
        }

        public void TryEndCrowdBattle()
        {
            if (crowdController.enemyUnitsAliveAmount == 0 ||
crowdController.playerUnitsAliveAmount == 0)
            {
                if (_winner == player)
                {
                    crowdController.PlayPlayerCrowdVictory();

MenuManager.Instance.MenuList.VictoryWindow.ShowWithDelay();

                    OnLevelSuccess?.Invoke();
                }

                else if (_winner == enemy)
                {

MenuManager.Instance.MenuList.LoseWindow.ShowWithDelay();

                    OnLevelFail?.Invoke();
                }

                OnLevelEnd?.Invoke();
            }
        }

        private void OnDisable()
        {
            OnPlayerStartPlaying -= PlayerStartPlaying;
        }
```

```
    }
}
```

Клас, що відповідає за контроль ігрового поля

```
using JetBrains.Annotations;
using Sirenix.OdinInspector;
using UnityEngine;
using UnityEngine.SceneManagement;

namespace CrowdT.Level
{
    public class LevelsLoader : MonoBehaviour
    {
        [SerializeField] public GameObject[] levels;

        private void Awake()
        {
            Save.Load();

            var realLevelId = Save.data.level % levels.Length;
            Load(realLevelId);
        }

        private void Load(int levelID)
        {
            Instantiate(levels[levelID]);
        }

        [Button("Load Level")]
        public void LoadLevel(int levelID)
        {
            Save.data.level = levelID;
            Save.Store();


SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
        }

        [UsedImplicitly]
        public static void MoveToNextLevel()
        {
            IncreaseLevel();

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
        }

        private static void IncreaseLevel()
        {
            var currentLevel = Save.data.level;
            currentLevel++;
            Save.data.level = currentLevel;
            Save.Store();
        }

        [UsedImplicitly]
        public static void RestartLevel()
        {
```

```
SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
         }
     }
}
```

Клас, що дозволяє завантажувати та вивантажувати рівні гри

```
namespace CrowdT
{
    using UnityEngine;

    [System.Serializable]
    public class Data
    {
        public bool firstAppLaunch = false;
        public int level = 0;
        public int coins = 0;
        public float volume = 0.5f;
    }

    public static class Save
    {
        public static Data data;
        static string keyName = "data";

        // Convert the data into PlayPrefs as XML
        public static void Store()
        {
            PlayerPrefs.SetString(keyName, Serializer.Serialize<Data>(data));
        }

        // Load the data from the PlayerPrefs and parse it into the data object
        public static void Load(System.Action whenLoad = null)
        {
            if (PlayerPrefs.HasKey(keyName))
            {
                data =
Serializer.Deserialize<Data>(PlayerPrefs.GetString(keyName));
            }
            else
            {
                data = new Data();
                Store();
            }

            if (whenLoad != null) whenLoad.Invoke();
        }

        // Delete the save
        public static void Delete()
        {
            DeleteKey(keyName);
        }

        // Delete the given key inside the PlayerPrefs : DEVELOPER ONLY
        public static void DeleteKey(string key)
        {
            if (PlayerPrefs.HasKey(key))
            {
                PlayerPrefs.DeleteKey(key);
            }
        }
```

```
    }
}
```

Клас, що дозволяє зберігати прогрес по ігровому рівню

```
using System.Xml.Serialization;
using System.IO;

namespace CrowdT
{
    public static class Serializer
    {
        // Serialize <T> any type
        // Convert <T> any type of data into a string
        public static string Serialize<T>(this T toSerialize)
        {
            XmlSerializer xml = new XmlSerializer(typeof(T));
            StringWriter writer = new StringWriter();
            xml.Serialize(writer, toSerialize);

            return writer.ToString();
        }

        // De-serialize
        public static T Deserialize<T>(this string toDeserialize)
        {
            XmlSerializer xml = new XmlSerializer(typeof(T));
            StringReader reader = new StringReader(toDeserialize);

            return (T) xml.Deserialize(reader);
        }
    }
}
```

Клас – помічник, що дозволяє серіалізувати та десеріалізувати будь які дані через формат Xml

```
using Sirenix.OdinInspector;
using UnityEngine;

namespace CrowdT
{
    public class SavesController : MonoBehaviour
    {
        public void Start()
        {
            Save.Load();
        }

        [Button]
        public void DeleteSave()
        {
            Save.Delete();
            Save.Load();
            Start();
        }
    }
}
```

Клас, що дозволяє контролювати процес збереження та завантаження з UnityEditor

ДОДАТОК В

```
using System.Threading.Tasks;
using CrowdT.Level;
using UnityEngine;

namespace CrowdT
{
    public abstract class Character : MonoBehaviour
    {
        [SerializeField] private UnitsSpawner _unitsSpawner;

        protected Character characterEnemy;

        private CharacterInput _input;
        private TentView _view;

        protected LevelController levelController;

        protected virtual void Awake()
        {
            _input = GetComponent<CharacterInput>();
            _view = GetComponentInChildren<TentView>();

            levelController = FindObjectOfType<LevelController>();
        }

        private async void Start()
        {
            await Task.Yield(); // Delay to wait while everything is
initialized
            FindAndTakeFreeCells();
        }

        private async void FindAndTakeFreeCells()
        {
            await new WaitForSeconds(.01f); // Delay to wait to
initialize units path

            UpdatePathfindingGrid();

            this.TakeCells();
            characterEnemy.TakeCells();

            UpdatePathfindingGrid();

            await new WaitForSeconds(.2f); // Delay to wait while all
spawned units will be registered

            await levelController.TryEndGame();
        }

        public async Task ExecuteTurn()
        {
            _input.StartTurn();

            do
```

```
            {
                await Task.Yield();
            }
            while (await _input.IsExecuted() == false);

            if (!LevelController.CrowdBattleStarted)
            {
                FindAndTakeFreeCells();
            }
        }

        private void UpdatePathfindingGrid()
        {
            AstarPath.active.data.graphs[0].Scan();
            AstarPath.active.data.graphs[1].Scan();
        }

        private void TakeCells()
        {
            var allReachableCells = GetReachableCells();

            var newCells =
levelController.groundGrid.GetNewOpenedCellsIn(allReachableCells);

            if (newCells.Length == 0) return;

            SetCellsType(newCells);

            ColorCellsByType(newCells);

            SpawnUnits(newCells);
        }

        protected abstract GroundCell[] GetReachableCells();

        protected abstract void SetCellsType(GroundCell[] cells);

        protected abstract void ColorCellsByType(GroundCell[] cells);

        private void SpawnUnits(GroundCell[] targetCells)
        {
            _view.OpenDoor();

            for (var i = 0; i !=
levelController.groundGrid.newOpenedCellsAmount; ++i)
            {
                var previousRandOffset = Vector3.zero;

                for (var j = 0; j !=
levelController.crowdController.unitsPerCell; ++j)
                {
                    var randomOffset = GetRandomCellOffset();

                    while (UnitsAreTooClose())
                        randomOffset = GetRandomCellOffset();

                    previousRandOffset = randomOffset;
```

```
                    var targetPos = targetCells[i].transform.position +
randomOffset;

                    _unitsSpawner.Spawn(1, targetPos);

                    bool UnitsAreTooClose()
                    {
                        return (previousRandOffset -
randomOffset).sqrMagnitude <

levelController.crowdController.unitsOffset.x *

levelController.crowdController.unitsOffset.x;
                    }
                }
            }

            _view.CloseDoor();
        }

        private Vector3 GetRandomCellOffset()
        {
            var randomXOffset = Random.Range(-
levelController.crowdController.unitsOffset.x,

levelController.crowdController.unitsOffset.x);

            var randomZOffset = Random.Range(-
levelController.crowdController.unitsOffset.z,

levelController.crowdController.unitsOffset.z);

            var randomOffset = new Vector3(randomXOffset, 0,
randomZOffset);

            return randomOffset;
        }
    }
}
```

Клас будь якого гравця

```
using System.Threading.Tasks;
using CrowdT.Level;
using UnityEngine;

namespace CrowdT
{
    public class PlayerInput : CharacterInput
    {
        [SerializeField] private Barrier barrierToRemoveInTutorial;
        private Vector3 _barrierToRemoveInTutorialPos;

        private TurnBasedBattleSystem _turnBasedBattleSystem;
        private LevelController _levelController;

        private Camera _mainCam;
```

```
        private const string _PLAYER_REMOVE_BARRIER_FEEDBACK_KEY_ =
"PlayerRemoveBarrier";

        private void Start()
        {
            view = FindObjectOfType<PlayerInputView>();

            if (barrierToRemoveInTutorial)
                _barrierToRemoveInTutorialPos =
barrierToRemoveInTutorial.transform.position;

            _turnBasedBattleSystem =
FindObjectOfType<TurnBasedBattleSystem>();
            _levelController = FindObjectOfType<LevelController>();

            _mainCam = Camera.main;
        }

        private void Update()
        {
            CheckForClickOnBarrier();
        }

        private void CheckForClickOnBarrier()
        {
            if (Input.GetMouseButtonDown(0) &&
_turnBasedBattleSystem.state == TurnState.PlayerTurn)
            {
                var ray = _mainCam.ScreenPointToRay(Input.mousePosition);

                if (!Physics.Raycast(ray, out RaycastHit hit))
                    return;

                OnBarrierClick(ref hit);
            }
        }

        private void OnBarrierClick(ref RaycastHit hit)
        {
            var barrier = hit.collider.GetComponent<Barrier>();

            if (barrier)
            {
                barrier.RemoveWithNotification();
                Feedbacks.Play(_PLAYER_REMOVE_BARRIER_FEEDBACK_KEY_,
transform.position, Vector3.zero);
            }
        }

        public override void StartTurn()
        {
            base.StartTurn();

            TryShowTutorial();
        }

        private void TryShowTutorial()
        {
```

```
            // show the tutorial on the first player turn of the first
level of the session
            if (!FirstPlayerTurnOnLevelMadeTracker.IsMade)
            {
                view.Show(_barrierToRemoveInTutorialPos);
            }
        }

        public override async Task<bool> IsExecuted()
        {
            if (!turnMade) return false;

            turnMade = false;

            if (!FirstPlayerTurnOnLevelMadeTracker.IsMade)
            {
                view.Hide();

                _levelController.OnPlayerStartPlaying?.Invoke();
            }

            return true;
        }

        public override void UpdateObserver()
        {
            if (_turnBasedBattleSystem.state == TurnState.PlayerTurn)
                base.UpdateObserver();
        }
    }
}
```

Клас, що відповідає за обробку контролю гравця

```
using DG.Tweening;
using UnityEngine;
using UnityEngine.UI;

namespace CrowdT
{
    public class CharacterInputView : MonoBehaviour
    {
        public float delayBeforeFadeIn;
        public float fadeInDuration;
        public float delayBeforeTouch;
        public float delayBeforeFadeOut;
        public float fadeOutDuration;

        [SerializeField] private GameObject hand;
        private Animator _animator;
        private Image _handImage;

        [SerializeField] private Image tapCircle;
        private Animator _tapCircleAnimator;

        private RectTransform _rectTransform;

        private RectTransform _canvasRect;
```

```
        private void Awake()
        {
            _rectTransform = GetComponent<RectTransform>();
            _animator = GetComponent<Animator>();
            _tapCircleAnimator = tapCircle.GetComponent<Animator>();

            _handImage = hand.GetComponent<Image>();

            _canvasRect =
MenuManager.Instance.MenuList.GameWindow.GetComponent<RectTransform>();
        }

        public void Show(Vector3 worldPos)
        {
            Vector2 viewportPosition =
Camera.main.WorldToViewportPoint(worldPos);

            var worldObjectScreenPosition =
                new Vector2(((viewportPosition.x *
_canvasRect.sizeDelta.x) - (_canvasRect.sizeDelta.x * 0.5f)),
                            ((viewportPosition.y *
_canvasRect.sizeDelta.y) - (_canvasRect.sizeDelta.y * 0.5f)));

            _rectTransform.anchoredPosition = worldObjectScreenPosition;

            Show();
        }

        private async void Show()
        {
            await new WaitForSeconds(delayBeforeFadeIn);

            if (IsShouldBeAborted()) return;

            PlayAppearanceEffect();
        }

        protected virtual bool IsShouldBeAborted() => false;

        private async void PlayAppearanceEffect()
        {
            _handImage.DOFade(1, fadeInDuration);;
            tapCircle.DOFade(1, fadeInDuration);

            await new WaitForSeconds(delayBeforeTouch);

            _animator.enabled = true;
            _tapCircleAnimator.enabled = true;
            _animator.Rebind();
            _tapCircleAnimator.Rebind();
        }

        public async void Hide()
        {
            _animator.enabled = false;
            _tapCircleAnimator.enabled = false;
```

```
            await new WaitForSeconds(delayBeforeFadeOut);

            tapCircle.DOFade(0, fadeOutDuration);
            _handImage.DOFade(0, fadeOutDuration);
        }
    }
}
```

Клас, що відповідає за візуальні підказки.

ДОДАТОК Г

```
using Sirenix.OdinInspector;
using System;
using UnityEngine;
using UnityEngine.Events;
using UnityEngine.UI;

namespace Menus
{
    [RequireComponent(typeof(Canvas))]
    [RequireComponent(typeof(CanvasScaler))]
    [RequireComponent(typeof(GraphicRaycaster))]
    public class Menu : MonoBehaviour
    {
        #region Enums

        public enum States
        {
            NotInit,
            Hidden,
            Hide,
            Shown,
            Show
        }

        #endregion

        #region Properties

#if UNITY_EDITOR
        [UnityEditor.MenuItem("GameObject/Menus/Manager", false, 1)]
        public static void CreateMenuManager()
        {
            GameObject menuManagerGameObject = new
GameObject(typeof(MenuManager).Name);

UnityEditor.Undo.RegisterCreatedObjectUndo(menuManagerGameObject, "Create
" + typeof(MenuManager).Name);
            menuManagerGameObject.AddComponent<MenuManager>();
            UnityEditor.Selection.activeGameObject =
menuManagerGameObject;
        }

        [Button("Show"), FoldoutGroup("Debug", order: 100),
DisableIf("DisableIfEditorShowMenu")]
        private void EditorShowMenu()
        {
            Show();
        }
        private bool DisableIfEditorShowMenu()
        {
            return state != States.Hidden;
        }

        [Button("Hide"), FoldoutGroup("Debug", order: 100),
DisableIf("DisableIfEditorHideMenu")]
```

```
        private void EditorHideMenu()
        {
            Hide();
        }
        private bool DisableIfEditorHideMenu()
        {
            return state != States.Shown;
        }
#endif

        public string ID => gameObject.name;

        protected States state = States.NotInit;
        /// <summary>
        /// Return the state of the menu :
        /// Shown -> displayed,
        /// Show -> transition to be shown,
        /// Hidden -> not displayed,
        /// Hide -> transition to be hidden
        /// </summary>
        public States State => state;

        /// <summary>
        /// Return true when the menu is displayed, including during the
transitions in and out
        /// </summary>
        public bool IsDisplayed => state != States.NotInit || state !=
States.Hide;

        /// <summary>
        /// Return true when the menu is shown or during the in
transition
        /// </summary>
        public bool IsShownOrAlmost => state == States.Show || state ==
States.Shown;

        /// <summary>
        /// Return true when the menu is hidden or during the out
transition
        /// </summary>
        public bool IsHiddenOrAlmost => state == States.Hide || state ==
States.Hidden;

        [SerializeField, FoldoutGroup("Basics")]
        protected Transition transitionIn = null;

        [SerializeField, FoldoutGroup("Basics")]
        protected Transition transitionOut = null;

        [SerializeField, FoldoutGroup("Basics")]
        protected bool showAtInit = false;

        [SerializeField, FoldoutGroup("Basics")]
        protected Body body = null;
        public Body Body => body;

#if UNITY_EDITOR
        public void EditorSetBody(Body _body)
```

```
        {
            body = _body;
        }
#endif

        [SerializeField, FoldoutGroup("Events")]
        public UnityEvent OnShowStartEvent = null;

        [SerializeField, FoldoutGroup("Events")]
        public UnityEvent OnShowEndEvent = null;

        [SerializeField, FoldoutGroup("Events")]
        public UnityEvent OnHideStartEvent = null;

        [SerializeField, FoldoutGroup("Events")]
        public UnityEvent OnHideEndEvent = null;

        #endregion

        #region Methods

        protected virtual void OnShowStart() { }
        protected virtual void OnShowEnd() { }
        protected virtual void OnHideStart() { }
        protected virtual void OnHideEnd() { }

        public virtual void Initialize()
        {
            if (showAtInit)
                ShowImmediately();
            else
                HideImmediately();
        }

        /// <summary>
        /// Show he menu without transition
        /// </summary>
        public void ShowImmediately(Action callback = null, bool
closeShownMenus = false)
        {
            if(closeShownMenus)
                MenuManager.Instance.HideAllMenusImmediately();

            state = States.Shown;
            gameObject.SetActive(true);
            OnShowStart();
            OnShowStartEvent?.Invoke();
            callback?.Invoke();
            OnShowEnd();
            OnShowEndEvent?.Invoke();
        }

        /// <summary>
        /// Show the menu with transition
        /// </summary>
        public void Show(Action callback = null, bool closeShownMenus =
false, Transition transition = null, bool autoBlock = true)
        {
```

```
        if (closeShownMenus)
            MenuManager.Instance.HideAllMenus();

        if (state != States.Hidden)
        {
            Debug.Log("The menu is already shown");
            OnShowStart();
            OnShowStartEvent?.Invoke();
            callback?.Invoke();
            OnShowEnd();
            OnShowEndEvent?.Invoke();
            return;
        }

        // Get transition by priority
        if (!transition)
        {
            if (transitionIn)
                transition = transitionIn;
            else
                transition = MenuManager.Instance.TransitionIn;
        }

        // If any transition found show immediately
        if (!transition)
            ShowImmediately(callback);
        else
        {
            state = States.Show;
            OnShowStart();
            OnShowStartEvent?.Invoke();
            gameObject.SetActive(true);
            if (autoBlock)
                MenuManager.Instance.Block();
            transition.Play(this, ()=>
            {
                state = States.Shown;
                if (autoBlock)
                    MenuManager.Instance.Unblock();
                callback?.Invoke();
                OnShowEnd();
                OnShowEndEvent?.Invoke();
            });
        }
    }

    /// <summary>
    /// Hide the menu without transition
    /// </summary>
    public void HideImmediately(Action callback = null)
    {
        state = States.Hidden;
        gameObject.SetActive(false);
        OnHideStart();
        OnHideStartEvent?.Invoke();
        callback?.Invoke();
        OnHideEnd();
        OnHideEndEvent?.Invoke();
```

```
        }

        /// <summary>
        /// Hide the menu with transition
        /// </summary>
        public void Hide(Action callback = null, Transition transition =
null, bool autoBlock = true)
        {
            if (state != States.Shown)
            {
                Debug.Log("The menu is already hidden");
                OnHideStart();
                OnHideStartEvent?.Invoke();
                callback?.Invoke();
                OnHideEnd();
                OnHideEndEvent?.Invoke();
                return;
            }

            // Get transition by priority
            if (!transition)
            {
                if (transitionOut)
                    transition = transitionOut;
                else
                    transition = MenuManager.Instance.TransitionOut;
            }

            // If any transition found hide immediately
            if (!transition)
                HideImmediately(callback);
            else
            {
                state = States.Hide;
                OnHideStart();
                OnHideStartEvent?.Invoke();
                if (autoBlock)
                    MenuManager.Instance.Block();
                transition.Play(this, () =>
                {
                    state = States.Hidden;
                    gameObject.SetActive(false);
                    transition.SetToFrom(this);
                    if (autoBlock)
                        MenuManager.Instance.Unblock();
                    callback?.Invoke();
                    OnHideEnd();
                    OnHideEndEvent?.Invoke();
                });
            }
        }

        #endregion
    }
}
```

Базовий клас для всіх інтерактивних UI вікон

```
using CrowdT;
using CrowdT.Level;
using Menus;
using UnityEngine;

namespace Menus
{
   public class LoseWindow : Menu
   {
      #region Properties

      [SerializeField] private float delayBeforeShow;
      [SerializeField] private GameObject dancingUnits;

      #endregion

      #region Menu Methods

      // Call at the start of the show animation
      protected override void OnShowStart()
      {
         base.OnShowStart();
      }

      // Call at the end of the show animation
      protected override void OnShowEnd()
      {
         base.OnShowEnd();
      }

      // Call at the start of the hide animation
      protected override void OnHideStart()
      {
         base.OnHideStart();
      }

      // Call at the end of the hide animation
      protected override void OnHideEnd()
      {
         base.OnHideEnd();
      }

      #endregion

      private void OnEnable()
      {
         GetComponent<Canvas>().worldCamera =
FindObjectOfType<UICamera>().GetComponent<Camera>();
         Instantiate(dancingUnits, Vector3.zero, Quaternion.identity);
      }

      public async void ShowWithDelay()
      {
         await new WaitForSeconds(delayBeforeShow);
         Show();

         Feedbacks.Play("DisplayLoseWindow", transform.position,
Vector3.zero);
```

```
        }

        public void ClickToContinue()
        {
            LevelsLoader.RestartLevel();
        }
    }
}
namespace Menus
{
    public partial class MenuList
    {
        [SerializeField]
        private LoseWindow loseWindow;
        public LoseWindow LoseWindow => loseWindow;
    }
}
```

Клас – представлення вікна поразки

```
using CrowdT;
using CrowdT.Level;
using Menus;
using TMPro;
using UnityEngine;

namespace Menus
{
    public class VictoryWindow : Menu
    {
        [SerializeField] private TMP_Text levelNumText;
        [SerializeField] private float delayBeforeShow;
        [SerializeField] private ParticleSystem confettiParticles;
        [SerializeField] private Vector3 confettiSpawnPos;

        [SerializeField] private GameObject dancingUnits;

        #region Properties

        #endregion

        #region Menu Methods

        protected override void OnShowStart()
        {
            base.OnShowStart();
        }

        // Call at the end of the show animation
        protected override void OnShowEnd()
        {
            base.OnShowEnd();
        }

        // Call at the start of the hide animation
        protected override void OnHideStart()
        {
            base.OnHideStart();
        }
```

```
      // Call at the end of the hide animation
      protected override void OnHideEnd()
      {
         base.OnHideEnd();
      }

      #endregion

      private void OnEnable()
      {
         GetComponent<Canvas>().worldCamera =
FindObjectOfType<UICamera>().GetComponent<Camera>();
         MenuManager.Instance.MenuList.GameWindow.Hide();
         Instantiate(dancingUnits, Vector3.zero, Quaternion.identity);
         levelNumText.text = $"LEVEL {Save.data.level + 1} COMPLETED";
      }

      public async void ShowWithDelay()
      {
         await new WaitForSeconds(delayBeforeShow);
         Show();

         // Instantiate(confettiParticles,
GetComponent<RectTransform>().position + confettiSpawnPos,
Quaternion.identity, transform);

         confettiParticles.gameObject.SetActive(true);/* = 0;*/
         confettiParticles.Play();

         Feedbacks.Play("DisplayVictoryWindow", transform.position,
Vector3.zero);
      }

      public void ClickToContinue()
      {
         LevelsLoader.MoveToNextLevel();
      }

      private void OnDisable()
      {
         confettiParticles.gameObject.SetActive(false);
      }
   }
}
namespace Menus
{
   public partial class MenuList
   {
      [SerializeField]
      private VictoryWindow victoryWindow;
      public VictoryWindow VictoryWindow => victoryWindow;
   }
}
```

Клас – представлення вікна перемоги