МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВОДНОГО ГОСПОДАРСТВА ТА ПРИРОДОКОРИСТУВАННЯ

“До захисту допущена”
Зав. кафедри комп'ютерних наук
та прикладної математики
д.т.н., професор Турбал Ю. В.
«___» _____ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

«Проектування та розробка гри-симулятора студентського життя»

Виконав: Бондарчук Богдан Олександрович
студент навчально-наукового інституту автоматики, кібернетики та обчислювальної техніки
група ІПЗ-41

_____
(підпис)

Керівник: доц., к.т.н. Жуковський Віктор Володимирович

_____
(підпис)

Рівне – 2023

ЗМІСТ

РЕФЕРАТ 3
ВСТУП 3
РОЗДІЛ 1. АРХІТЕКТУРНІ РІШЕННЯ У ВІДЕОІГРАХ 6
1.1 Історія розвитку ігрових рушіїв 6
1.2 Патерни проектування в іграх 8
РОЗДІЛ 2. ВИБІР СЕРЕДОВИЩА РЕАЛІЗАЦІЇ ТА ПРОЕКТУВАННЯ АРХІТЕКТУРИ ГРИ-СИМУЛЯТОРА 14
2.1. Вибір інструментів для розробки 14
2.2. Розробка концепту гри 23
РОЗДІЛ 3. РОЗРОБКА ГРИ-СИМУЛЯТОРА СТУДЕНТСЬКОГО ЖИТТЯ 29
3.1. Програмування ігрового процесу 29
3.2. Реалізація мініігор 41
ВИСНОВКИ 52
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 53
ДОДАТКИ 56

Національний університет водного господарства та природокористування

(повне найменування вищого навчального закладу)

НН інститут **автоматики, кібернетики та обчислювальної техніки**

Кафедра **комп'ютерних наук та прикладної математики**

Освітньо-кваліфікаційний рівень **бакалавр**

Галузь знань **12 «Інформаційні технології»**

Спеціальність **121 «Інженерія програмного забезпечення»**

Освітня програма **Інтернет речей.**

ЗАТВЕРДЖУЮ

Зав. кафедри

д.т.н., професор Турбал Ю. В.

"____" _____ 2023 року

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Бондарчуку Богдану Олександровичу

1. Тема роботи: *Проектування та розробка гри-симулятора студентського життя*,
   керівник роботи: *доц., к.т.н, Жуковський Віктор Володимирович*, затверджені наказом вищого навчального закладу С №-449 від *" 19 " квітня 2023 р*.
2. Терміни подання роботи студентом: *20 червня 2023 року*.
3. Вихідні дані до роботи: *перелік відкритих архітектурних патернів, офіційна документація рушія Unity.*
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): *Дослідити використання архітектурних шаблонів у сфері*

розробки ігор. Оцінити відомі патерни проектування та можливості їх застосування у іграх. Розробити гру-симулятор студентського життя із використанням загальних архітектурних рішень.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): *рисунки.*
6. Консультанти розділів роботи:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Розділ 1	доц., к.т.н. Жуковський Віктор Володимирович	26.03.2023	14.04.2023
Розділ 2	доц., к.т.н. Жуковський Віктор Володимирович	15.04.2023	28.05.2023
Розділ 3	доц., к.т.н. Жуковський Віктор Володимирович	28.05.2023	17.06.2023

7. Дата видачі завдання: 26 березня 2023 (26 березня 2023 рік)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вивчення літератури за обраною тематикою	26.03.2023 - 05.04.2023	Виконано
2	Ознайомлення із існуючими архітектурними шаблонами	05.04.2023 - 14.04.2023	Виконано
3	Вибір стеку технологій	15.04.2023 -	Виконано

		20.05.2023	
4	Розробка архітектури проекту	20.05.2023 - 28.05.2023	Виконано
5	Написання основної логіки гри-симулятора студентського життя	28.05.2023 - 06.06.2023	Виконано
6	Розробка мініігор	06.05.2023 - 15.06.2023	Виконано
7	Тестування розробленої гри	15.05.2023 - 17.06.2023	Виконано
8	Підготовка звіту до кваліфікаційної роботи	17.05.2023 - 20.05.2023	Виконано
9	Підготовка презентації	20.05.2023 - 21.05.2023	Виконано

Студент _____ Бондарчук Богдан Олександрович
(підпис)

Керівник роботи _____ Жуковський Віктор Володимирович
(підпис)

РЕФЕРАТ

Кваліфікаційна робота: 58 сторінок, 20 рисунків, 20 джерел

Предмет дослідження: шаблонні архітектурні рішення у розробці ігрових додатків та їх практичне застосування.

Мета роботи: Розробка гри-симулятора студентського життя, що буде демонструвати використання шаблонних архітектурних рішень, які є корисними при розробці ігрових додатків.

Актуальність роботи: полягає у необхідності створення та популяризації використання шаблонних архітектурних рішень у сфері розробки ігор, а саме логічних шаблонів для створення і контролю ігрових об’єктів та забезпечення сумісності та зручності логічних взаємодій між логічними частинами гри.

Ключові слова: Unity, архітектурне рішення, патерн, шаблон, C#, візуальний інтерфейс, ініціалізація, сутність, логічні системи, кадр, мінігра.

ВСТУП

Ігрова індустрія сьогодні є різнобічною, відкритою для нових ідей та технологій. Вона постійно зростає та еволюціонує, раз за разом міняючи погляди на процеси розробки ігор. За останні роки ринок відеоігор перетворився на масову індустрію із захмарними щорічними доходами, що вимірюються в сотнях мільйонів доларів. Із появою цифрових технологій ігри змогли вийти на передній план, обійшовши при цьому більш традиційні види дозвілля людей: музику, кіно та книги. У сучасному житті ігри посідають значне місце в житті людей. Вони надають можливість розважитися та зняти стресс або ж навпаки досягнути вершин популярності, обійшовши усіх суперників. Кожен окремий гравець може знайти собі жанр, механіку чи ігровий стиль, який би припав йому до душі.

Ігри давно стали багатофункціональним інструментом і станом на сьогодні виконують не лише розважальну функцію. Для прикладу можна розглянути їх використання у навчальних цілях - ігрові додатки часто використовують як навчальний інструмент, який дозволяє донести до гравця всю необхідну інформацію у простій для сприйняття формі. Наявність величезної кількості ігор та онлайн курсів, що проходять у ігровій формі, доводять ефективність таких методів навчання та підтверджують те, що ігри посідають важливе місце у житті сучасного суспільства.

Конкуренція на ринку між розробниками відеоігор сприяє розвитку індустрії: покращенню існуючих та використання нових технологій. Для утримання існуючих та залучення нових гравців якість ігор повинна бути достатньо високою. Щоб забезпечити стабільний розвиток ігрових продуктів, у індустрії намагаються використати майже кожну наявну та доступну технологію. Створення нових способів відображення графіки, створення реалістичної фізики та вдосконалення ігрових механік сприяють появі більш реалістичних, пропрацьованих та деталізованих віртуальних світів. Кожен аспект розробки ігрових додатків є окремою сферою зі своїми особливостями і має великий потенціал для покращень та досліджень.

Підходи до розробки кожного окремого елементу гри можуть змінюватися із дня у день. При цьому відкриваються нові й нові можливості, які надають розробникам змогу показати свої ідеї світу у максимально повному обсязі. Незважаючи на постійну зміну способів та підходів до розробки, деякі рішення зарекомендували себе дуже корисними і використовуються майже у кожному ігровому додатку у тому чи іншому вигляді. У даній роботі ми познайомимося із частиною із них та спробуємо використати їх на практиці для розробці гри.

РОЗДІЛ 1. АРХІТЕКТУРНІ РІШЕННЯ У ВІДЕОІГРАХ

1.1 Історія розвитку ігрових рушіїв

Індустрія відеоігор з’явилась декілька десятків років тому і відтоді невпинно розвивається. Із кожним роком загальний світовий розвиток технологій надає розробникам ігор нові можливості для реалізації все більшої кількості деталей та можливостей для гравця. Разом із технічної частиною розвиваються підходи до дизайну ігор та їхньої візуальної інтерпретації. Усі ці процеси мають за мету покращення занурення гравця у віртуальний світ, що дозволяє йому відпочити від реальності чи отримати нові відчуття. Завдяки цим можливостям ігрова індустрія приваблює велику кількість людей різного віку, професій та соціального статусу.

Зазвичай першочерговою ціллю розробки гри є завоювання симпатії обширної аудиторії, яка буде генерувати трафік та у перспективі стане джерелом доходу для розробників гри. Оскільки на маркетплейсах щодня з’являються сотні нових ігор серед яких більшість не помічається гравцями, то для того щоб виділитись серед потоку гра повинна бути продуктом достатньо високої якості щоб захоплювати увагу гравця. Якість гри можна легко оцінити за маркетинговими показниками (кількість повторних заходів у гру чи внутрішніх ігрових покупок, тощо). Для забезпечення достатньо високих показників усі аспекти гри повинні працювати чітко та злагоджено, а також доповнювати одне одного. Такий баланс повинен працювати у всіх аспектах гри, включаючи геймдизайн, візуальне сприйняття, вплив на почуття гравця та ігрові механіки. У цій роботі розберемо вище описані взаємодії для ігрових механік.

Ігрова механіка (англ. game mechanics) — набір правил і способів, який реалізує певним чином деяку частину інтерактивної взаємодії гравця та гри. Вся множина ігрових механік гри формує конкретну реалізацію її ігрового процесу [1].

З початку 90-х ігри створювалися для певних платформ. Наприклад, не можна було пограти у гру, написану для певної консолі, використовуючи інший пристрій. Для кожного окремого девайсу потрібен був окремий код і у наслідку

робота програмістів. Станом на сьогодні операційні системи забезпечують певну апаратну абстракцію, тому програми не повинні враховувати можливі особливості операційної системи та різного обладнання, яке можна встановити комп'ютери. До цього кожна гра повинна була окремо обробляти взаємодію із драйверами для будь-якої відеокарти чи звукової карти. В результаті не можна було повторно використовувати для однієї гри код, написаний для іншої гри. У 80-тих роках минулого століття аркадні ігри були дуже популярні, але майже жоден із варіантів коду не мав можливості повторного використання в нових іграх завдяки постійним змінам і прогресу комп'ютерних технологій.

Ігри ставали все більш складними, операційні системи у наслідку також ставали все більш комплексними та менше залежали від апаратного забезпечення. Для ігрових компаній стала відчутною потреба у можливості повторного використання коду з попередніх ігор і все частіше поставало питання - навіщо потрібно писати новий спосіб відмальовки об'єктів чи взаємодії із персонажами якщо можна позичити її у однієї з ігор, що уже були випущені? У 1990-х роках, стали популярними шутери від першої особи, такі як Doom та Quake. Ці ігри мали настільки велику популярність, що виробник Id Software вирішила ліцензувати частину коду гри та продавати її іншим компаніям що займались розробкою ігор як окреме програмне забезпечення. Так з'явився перший ігровий рушій.

Ігровий рушій (англ. Game engine) — програмний рушій, центральна програмна частина будь-якої відеогри, яка відповідає за всю її технічну сторону, дозволяє полегшити розробку гри шляхом уніфікації та систематизації її внутрішньої структури. Важливим значенням рушія є можливість створення багатоплатформових ігор (сьогодні найчастіше одночасно для ПК, PS4 та Xbox One)[2].

Перепродаж основного коду як ігрового рушія був прибутковим бізнесом, оскільки інші ігрові компанії були готові заплатити великі суми грошей за ліцензію на використання рушія. Перевагою для цих компаній стала можливість зосередитися на різних аспектах гри, як от світобудова, арт чи звуковий супровід

замість того щоб витрачати час на написання ігрового коду з нуля, оскільки всі потрібні можливості можна було отримати за допомогою рушія.

Станом на сьогодні доступно багато різних ігрових рушіїв. Частину ігрових рушіїв було розроблено під особливі платформи для роботи із конкретними ОС чи ігровими консолями. У той же час інші ігрові рушії можна використовувати для різних платформ без необхідності у модифікації програм, які використовують код даного рушія. Така можливість є дуже важливою для ігрових компаній, метою яких є публікація ігор на різних платформах.

У наш час ігрові рушії пропонують безліч функцій для розробки ігор, таких як механізми 2D і 3D візуалізації, VFX ефекти, робота з освітленням, звуком та анімаціями та незліченну кількість інших допоміжних функцій що полегшують розробку. Завдяки такому набору функцій ігрові рушії використовуються майже у кожній гра, оскільки в іншому випадку реалізація усіх потрібних механізмів може відібрати надто багато часу, а купівля ліцензії на використання рушія зазвичай є дешевшим варіантом. Звісно є і винятки, оскільки коли дохід від розроблених ігор дуже великий, то розробка власного рушія дозволить зекономити частину коштів. Наявність рушіїв дозволяє програмістам у компаніях, що займаються розробкою ігор, зосереджуватись на реалізації основних ігрових механік. Хоча часто є і потреба у розробці додаткових програм, як от редактора рівнів чи програми роботи із специфічними форматами картинок чи моделей.

1.2 Патерни проектування в іграх

Патерном проектування називають типове архітектурне рішення для вирішення однієї конкретної проблеми що зустрічається при проектуванні логіки різноманітних програм. Патерни являють собою узагальнені високорівневі логічні рішення для певних проблем, при цьому їх реалізації у коді програм зазвичай відрізняються одна від одної, бо є адаптованими під конкретну ситуацію[15]. Існує велика кількість патернів для вирішення різних типів задач. Вони можуть відрізнятися за призначенням, кількістю деталей які охоплюють, складністю реалізації та іншими критеріями.

За призначенням виділяють такі типи патернів:

- Структурні
- Поведінкові
- Породжуючі

Із них породжуючі патерни відповідають за створення різних типів сутностей, структурні допомагають забезпечувати доступ до необхідних частин коду та відповідають за ієрархію, а поведінкові описують шаблонні методи вирішення тих чи інших проблем що можуть трапитися розробнику. Зазвичай патерн складається із короткого опису проблеми яку він вирішує та способу вирішення, а іноді додатково вказуються альтернативи чи надається приклад коду із рішенням на одній із мов програмування[15]. Для розробки навіть невеликих ігрових проектів зазвичай застосовуються рішення із усіх категорій. Щоб полегшити собі роботу розробники ігор віднайшли багато шаблонних рішень і усі вони мають своє застосування, проте серед них можна виділити кілька найбільш популярних патернів проектування, які можна відшукати у більшості існуючих ігор. Ознайомимося з ними та виділимо для кожного окремого рішення його переваги та недоліки.

До популярних патернів відносяться:

- Component System
- MVVM (Model-View-ViewModel)
- Observer
- Object Pool
- Singleton

Мабуть найбільш популярним рішенням при розробці ігрових додатків є розділення логіки гри на окремі компоненти. Такий підхід до розробки називається компонентною системою (Component System). Він полягає у тому, що ігрова логіка розділяється по класах-компонентах, які забезпечують певний функціонал і зберігають у собі дані та потрібну логіку. При цьому такі класи можуть бути частково пов’язані між собою. Зазвичай один такий компонент відповідає за певний конкретно визначений функціонал, який при цьому можна

відділити від іншої логіки. Прикладом такої системи є архітектура ECS, Behaviour у рушії Unity та компоненти у Unreal Engine та інших ігрових рушіях. Вони часто використовуються для збірки ігрових об’єктів із різноманітних частинок-компонентів що містять потрібний функціонал.

Підхід компонентної системи використовується у випадках, коли необхідно створювати велику кількість об’єктів із різним набором можливостей. Також його використовують для розбиття великих шматків ігрової логіки на кілька окремих частин і для спрощення реалізації функціоналу, який має працювати із різними частинами коду програми. Перевагами Component System є зменшення кількості зв’язків між класами та забезпечення можливості повторного використання компонентів. При цьому така система є складнішою ніж варіація зі зберіганням логіки у одному місці, тому її краще не використовувати для невеликих ігрових систем.

Часто для коректної роботи частині компонентів потрібно взаємодіяти із іншими компонентами. Для цього є два підходи зі своїми перевагами та недоліками. Першим із них є створення явних зв’язків між логікою, яку необхідно пов’язати між собою. Цей варіант є досить простим та не ускладнює відстеження взаємодій компонентів між собою. Його недоліком є те що при цьому такі компоненти не можна використовувати окремо одне від одного, а їхній функціонал стає складніше розширювати, оскільки необхідно враховувати створені зв’язки. Другим варіантом створення взаємодій між компонентами є використання системи сповіщень. Один із компонентів буде відправляти сповіщення та супутні дані, а інший приймати їх та виконувати необхідні дії. Такий підхід дозволяє прибрати прямі залежності між компонентами та зробити їх автономними. Недоліком такого підходу є те, що стає складно прослідкувати поведінку системи у тій чи іншій ситуації через відсутність прямих зв’язків які б явно вказували на стан тієї чи іншої частини програмного коду. Зазвичай при розробці ігрових продуктів використовуються обидва підходи у парі.

Окремим видом підходу до компонентної системи є Entity Component System. Вона відходить від стандартних парадигм об’єктно-орієнтованого

програмування і є прикладом data-oriented архітектури. Основна її відмінність від звичайної компонентної системи полягає у тому, що набори даних та логіка що відповідає за їхню обробку розміщені у окремих компонентах, які не пов’язані між собою. При використанні такої системи ігрові об’єкти представляють собою просто набори компонентів які визначають їхні властивості та поведінку. Підхід із використанням Entity Component System є відносно новим, проте достатньо популярним завдяки набору переваг які він може надати розробникам. Основною перевагою такого підходу є те, що він дозволяє реалізовувати логіку будь-якої складності найбільш оптимізованим способом. Це забезпечується тим, що логіка використовує із контейнеру, що тримає у собі дані, лише те що їй потрібно для роботи і при цьому не підтягує непотрібних залежностей які можуть вплинути на час що витрачається для доступу до даних. Ще одним приємним моментом є те, що кількість можливих залежностей при використанні такого підходу є мінімальною і це дозволяє спростити архітектуру та роботу з нею.

Далі ознайомимося із MVVM (Model-View-ViewModel) - це патерн, що відноситься до архітектурних та полегшує взаємодію із користувачем за допомогою відокремлення логіки програми від її графічного інтерфейсу. Вперше цей патерн з’явився на публіці у 2005 році у вигляді розширення відомого тоді MVP (Model-View-Presenter).

Основною ідеєю патерну MVVM є відділення логіки що використовується для відображення графічного інтерфейсу від усіх інших логічних частин гри. Він виділяє у собі три сутності: Model, View, ViewModel. Із них Model містить у собі необхідні дані, View відповідає за відображення графічного інтерфейсу, а ViewModel пов’язує ці дві сутності між собою та містить логіку для перетворення та передачі потрібних даних. Перевагою використання MVVM є спрощення взаємодії із інтерфейсом, за рахунок того що він є відокремленим від іншої логіки програми. Реалізація такої логіки є простою, тому патерн часто використовується при розробці програм. Недоліком MVVM є лише те, що він збільшує кількість класів, оскільки для кожного варіанту інтерфейсу потрібно додавати три класи.

Наступним згаданим патерном є Observer(спостерігач). Цей патерн відповідає за створення механізмів підписки[18]. Observer є покращеною версією стандартних систем сповіщень. Такі системи мають у собі дві сутності: сповіщення та отримувач. Отримувачі підписуються на потрібні їм сповіщення, а сповіщення при запуску у потрібний момент надсилаються до потрібних отримувачів. При надсиланні клас сповіщення перевіряє всіх отримувачів, щоб визначити яким із них потрібно надсилати те чи інше сповіщення. Патерн Observer також реалізує систему сповіщень, але при цьому конкретне сповіщення має доступ до інформації про потрібних отримувачів за допомогою динамічного списку і тому не викликає перевірки для усіх інших. Такий патерн застосовується у тій чи іншій формі майже у всіх ігрових проектах, оскільки дозволяє оптимізувати кількість звернень і при цьому забезпечує роботу сповіщень для конкретних отримувачів лише у потрібний їм момент. Єдиним недоліком такої системи сповіщень є те, що отримувачі сповіщаються не у визначеному порядку.

Для роботи із великою кількістю однакових об’єктів використовують патерн Object Pool. Він відповідає за створення та видалення об’єктів. Зазвичай у ігрових рушіях операції зі створення і видалення займають багато процесорного часу, а Object Pool дозволяє оптимізувати такі процеси. При його використанні об’єкти після використання не видаляються, а зберігаються у спеціальному місці і можуть бути перевикористані при потребі[19]. Іноді такі об’єкти створюються наперед ще до потреби у використанні. Перевагою використання пулу об’єктів є економія на операціях створення та видалення. При цьому така економія має зміст лише для об’єктів які часто використовуються, наприклад для візуальних ефектів (VFX) чи різних динамічних об’єктів які час від часу зникають із поля зору гравця. У інших випадках пул об’єктів лише погіршить загальну ситуацію, оскільки зберігання об’єктів для перевикористання лише збільшить загальне навантаження системи якщо вони рідко використовуються.

Останнім із популярних патернів при розробці ігор є Singleton(одинак). Він гарантує те, що клас має лише один екземпляр, і створює для нього точку доступу, яку можна використати у потрібному місці коду[20]. Ці особливості і є його перевагами, оскільки такий підхід до створення об’єкту забезпечує можливість його зручного використання незалежно від місця виклику. Недоліком такого використання об’єктів є те, що вони порушують логіку того що один клас повинен мати одну функцію. Також із класами-одинаками можуть виникати проблеми при використанні асинхронних методів. Патерн Singleton потрібно використовувати лише у особливих випадках для реалізації сутностей який дійсно потрібен тільки один екземпляр. У іншому ж випадку класи-одинаки лише ускладнять роботу із кодом та знижуватимуть його якість.

РОЗДІЛ 2. ВИБІР СЕРЕДОВИЩА РЕАЛІЗАЦІЇ ТА ПРОЕКТУВАННЯ АРХІТЕКТУРИ ГРИ-СИМУЛЯТОРА

2.1. Вибір інструментів для розробки

Для розробки гри-симулятора було використано такі технології:

- Microsoft Visual Studio 2022
- Microsoft Visual Studio Code
- Unity 2021.3.17
- Git + Github
- Atlassian Sourcetree

Розберемо їхні переваги та недоліки.

Visual Studio – створене компанією Microsoft інтегроване середовище для спрощення розробки програмних рішень. Перша версія Visual Studio вийшла у 1997 році і з того часу продукт невпинно розвивається. Сучасна версія цього програмного середовища надає можливість створювати найрізноманітніші програмні рішення: веб-додатки, стандартні програми для Windows із графічним інтерфейсом, консольні додатки, утиліти, бібліотеки чи ігрові додатки, тощо[3].

Visual Studio пропонує можливості для розробки додатків із використанням різних мов програмування: Python, Visual C++, Visual C#, Visual Basic, Visual F#, JavaScript. При цьому середовище також дозволяє працювати із різними сервісами та готовими рішеннями: ASP.NET, Azure, Node.js, .Net Core, .Net Framework, Unity та Unreal, DirectX та багато інших. Visual Studio та частина із вищезгаданих рішень підтримують розробку під різні популярні платформи, хоча основною із них залишається Windows оскільки він теж розроблений компанією Microsoft.

Для спрощення розробки програм у Visual Studio є кілька вбудованих функцій і плагінів та можливість встановлення додаткових (їх зазвичай використовують при розробці Web-додатків). Вбудовані функції середовища включають у себе вбудований Git, підсвічування та доповнення коду. Найбільш корисною із них при розробці гри-симулятора студентського життя стала вбудована у середовища система автозаповнення Intellisense. Вона аналізує код та пропонує варіант для дописання на основі раніше написаного коду:

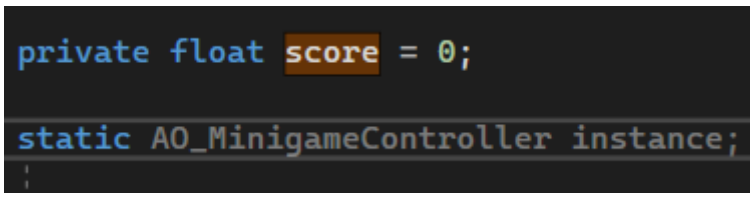

Рис 2.1. Автозаповнення Intellisense

Варіанти коду від Intellisense виділенні напівпрозорим текстом. Крім автоматичного дописування коду Intellisense дозволяє генерувати методи чи поля при їх використанні без оголошення чи навпаки, що особливо корисно при розробці додатків на мові C++. Єдиним виявленим при розробці недоліком Intellisense є те, що при кожному випадковому перериванні аналізу коду ця система перестає коректно працювати і іноді може запропонувати рішення, але видати помилку при генерації. Таке часто трапляється на об'ємних громіздких проектах. В загальному Visual Studio є важким програмним рішенням тому може часто зависати при виконанні деяких із своїх внутрішніх процесів. Такі проблеми трапляються досить часто незалежно від фізичних можливостей пристрою на якому ведеться розробка і є головним недоліком середовища. Інші знайдені недоліки зазвичай стосуються візуального інтерфейсу та не заважають при розробці програм.

Іноді при розробці гри була потреба у маленьких виправленнях і щоб уникнути можливих втрат часу при оновленні внутрішньої інформації Visual Studio вони виконувались за допомогою іншого програмного рішення від Microsoft - редактору Visual Studio Code.

Visual Studio Code є редактором коду створеним компанією Microsoft із частиною можливостей Visual Studio[4]. Він має схожий набір вбудованих функцій та працює в рази швидше від Visual Studio, хоча деякі плагіни можуть сильно його сповільнити. Visual Studio Code має набагато більше можливостей для кастомізації ніж Visual Studio, зручніший термінал та вибір методу запуску програмного проекту. Такі можливості роблять його дуже зручним для розробки Web-додатків, оскільки за допомогою терміналу від може забезпечити усе що потрібно для зручної розробки. Незважаючи на всі переваги при розробці ігор

Visual Studio Code використовується рідко, оскільки він не надає достатньо можливостей для налаштування проектів. При розробці гри-симулятора студентського життя Visual Studio Code зазвичай використовувався для швидких виправлень, пошуку по проекту та пришвидшення роботи із ігровими ресурсами у рушії.

Unity - умовно безкоштовний кросплатформний рушій для розробки та підтримки роботи відеоігор, додатків та навіть анімованих фільмів. Unity дозволяє розробляти ігри та програми різного типу із використанням двовимірної чи тривимірної графіки. Створені із використанням цього рушія відеоігри та програми можуть запускатись на ігрових консолях, пристроях що підтримують VR(віртуальну реальність) чи AR(доповнену реальність), персональних комп'ютерах, у браузерах чи на мобільних пристроях. В загальному додатки розроблені із використанням Unity можуть працювати на 25 платформах[5]. Така кількість можливих платформ для підтримки надає цьому рушію популярності, оскільки один і той же проект можна запустити на великій кількості найрізноманітніших пристроїв при мінімальних затратах часу розробників.

Окрім великого набору можливих платформ Unity також забезпечує захмарну кількість інструментів для розробки ігор. До них відносяться велика кількість базових програмних рішень для забезпечення можливостей відображення та переміщення об'єктів, взаємодії із вхідними даними від гравця чи програвання звуків та музики. Також є більш складні системи, а саме можливість роботи зі світлом, фізикою тіл із різного матеріалу у двовимірному та тривимірному просторах, тривимірними моделями (мешами), забезпечення упаковки ресурсів у різні формати та багато інших можливостей[5].

Редактор рушія Unity є доступним на платформах Windows та Mac OS та і складається із певного набору вікон які можна розташувати так як було б зручно розробнику. Кожне вікно редактора відповідає за певну частину проекту та дозволяє її налаштовувати.

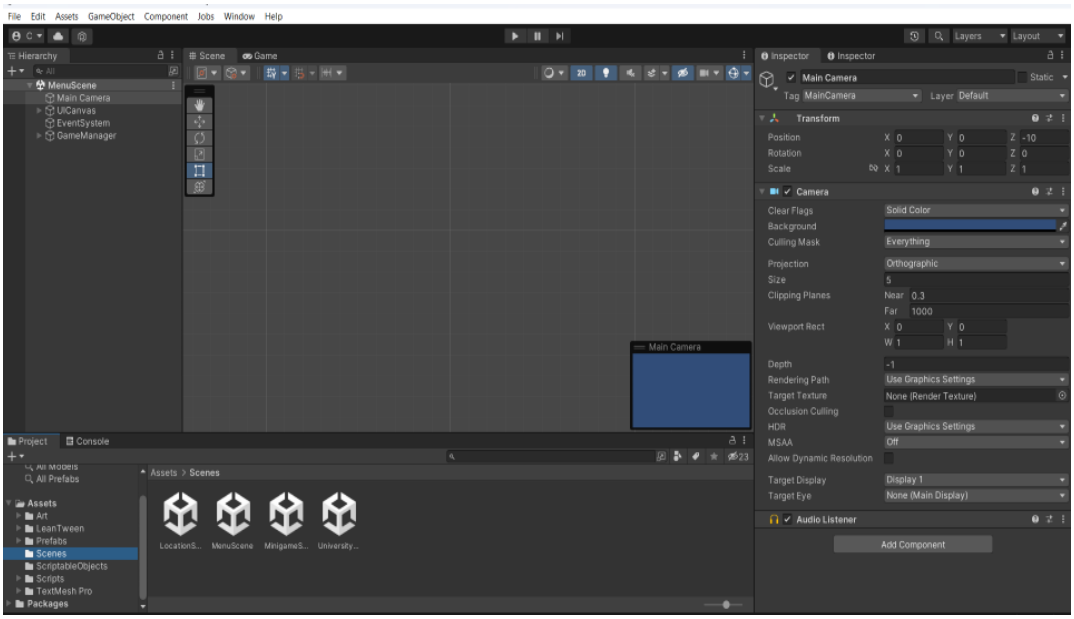

Рис 2.2. Редактор Unity

Проект у Unity представляє собою певний набір ігрових сцен та зв'язків між ними. Ігрова сцена представляє собою окремий файл із певним набором об'єктів (GameObject), та налаштувань. Зазвичай кількість сцен залежить від розміру проекту. Маленьким проектам може вистачити однієї ігрової сцени, а великі можуть використовувати десятки таких сцен. У більшості випадків одночасно активною є лише одна сцена, але іноді використовуються одразу декілька: одна основна ігрова сцена та необхідна кількість додаткових.

Об'єкти можуть мати візуальне відображення на сцені (спрайти, 3d моделі), а іноді бувають пустими і являють собою точку прив'язки компонентів що відповідають за ігрову логіку (Behaviour). GameObject обов'язково має у собі компонент Transform, що відповідає за його положення у просторі, а все інше не є потрібним та задається лише при потребі[5]. При цьому об'єкти можна зберігати як окремі файли що називаються префабами або ассетами. Те ж саме стосується і файлів налаштувань.

Рушій Unity є простим у використанні та від початку не потребує додаткових налаштувань, можна просто створити проект по одному із доступних шаблонів та відразу розпочати роботу. Єдиним налаштуванням яке рекомендовано встановити відразу є вибір стандартного для рушія середовища розробки у розділі External Tools меню із налаштуваннями уподобань

користувача. Unity повністю підтримує роботу із Visual Studio, Visual Studio Code та XCode, тому зазвичай проекти запускаються за допомогою них.

Останньою із вагомих переваг рушія Unity є вбудований маркетплейс із асетами та плагінами для розробки - Unity Asset Store. За допомогою нього можна інтегрувати готові ресурси або програмні рішення у свою гру одним кліком. При цьому варто враховувати те що деякі із них можуть використовувати багато пам'яті пристрою чи його обчислювальних ресурсів, що негативно впливає на ігровий досвід гравців. Найбільш ця проблема помітна при створенні мобільних ігор, тому доводиться або відмовлятись від деяких рішень або шукати обхідні шляхи. Ними можуть бути наприклад LOD механізм для спрощення рендеру об'єктів чи стискання та додаткове завантаження потрібних грі ресурсів для зменшення фінального розміру розробленого додатку. Для розробки гри-симулятора студентського життя було встановлено безкоштовний плагін LeanTween, який надає можливість більш зручного способу запуску коду із затримкою у порівнянні із кількома стандартними рішеннями що реалізовані у Unity.

Основними перевагами Unity є описані вище простота, кросплатформність, широкі можливості кастомізації та інтеграції зовнішніх рішень у проекти. Крім них він має і декілька недоліків. Першим із них є його потреба у обчислювальних ресурсах, можливості пристрою та кількість файлів у проекті напряму впливають на час завантаження та перекомпіляції при внесенні змін, тому іноді одне лише завантаження проекту може віднімати кілька хвилин часу. Другим вагомим недоліком Unity є великий розмір готового проекту на мобільних платформах. Повністю пустий проект при збірці займає близько 20 мегабайт пам'яті пристрою, в той час як частина простіших рушіїв, таких як Defold чи Godot можуть вмістити у вдвічі менший обсяг готову гру із усіма ресурсами та кодом.

Для програмування ігрової логіки із використанням Unity зазвичай використовується мова C#. Інші мови мають обмежену підтримку або узагалі не підтримуються рушієм.

C# - об’єктно-орієнтована мова програмування розроблена командою інженерів Microsoft і стандартизована у організаціях ECMA та ISO. Вона має безпечну строгу статичну типізацію. C# ввібрав у себе більшість переваг від уже існуючих мов програмування і у першій версії нагадував собою розширену версію Java із можливостями C++ та деякими новими унікальними функціями[7]. C# має простий синтаксис у порівнянні із C++ чи Java що робить його простим для ознайомлення та достатньо зрозумілим для початківців. Для роботи із технологією C# існує декілька компіляторів, основним із яких вважається стандартна реалізація від компанії Microsoft. Крім нього існують і інші компілятори C#, більша частина яких використовують бібліотеки класів .NET та рішення із фреймворку Common Language Infrastructure[7]. Можливості C# залежать від реалізації CLR (Common Language Runtime) - віртуальної машини із пакету .NET, яка виконує код мови[9]. Незважаючи на відмінності більшість реалізацій мови схожі та забезпечують увесь потрібний мові функціонал та не обмежують стандартної реалізації.

Основними перевагами мови C# є:

- Простота
- Універсальність
- Модульність
- Стабільний розвиток технології
- Широкий спектр функціоналу
- Система пакетів NuGet

Перша версія мови C# з’явилась разом із технологією .NET Framework та була її частиною. **.NET Framework** - створена компанією Microsoft платформа для створення кросплатформенних додатків. Метою її розробки було відв’язування продуктів компанії Microsoft від платформи Windows та відхід від мовноорієнтованої розробки.

.NET Framework наслідував ідеї що були реалізовані у Java. Технологія .NET поділяється на дві частини: середовище розробки та віртуальну машину (CLR), яка виконує код[8]. .NET надає можливість роботи із модулями що

написані на різних мовах та забезпечує їхню сумісність. За рахунок цієї переваги за допомогою .NET можна розробляти різноманітні проекти різної складності та призначення: маленькі мобільні додатки, компілятори, глобальні веб-платформи та багато інших типів додатків.

Останньою із використаних технологій був Git. **Git** - система контролю версій файлів. Він використовується для спільної роботи та керування версіями файлів. Ця система надає гнучкі інструменти для зручної розробки проекту, основою яких є робота з гілками. За рахунок своєї надійності, ефективності та функціональності Git є найпопулярнішою системою контролю версій[10]. Використання Git є майже обов'язковим при розробці проектів у команді, але при цьому він є достатньо корисним і для розробників-одинаків.

Git зберігає всі дані файлів у проекті в окрему теку і дозволяє відслідкувати їхню історію змін за допомогою виконання команди git blame у консолі. При цьому ми побачимо хеш-коди коммітів у яких були зроблені останні зміни для кожного рядку файлу, а також автора змін та час їх внесення.

```
$ git blame Assets/Scripts/GameManager.cs
c7686220 (Chesst1k 2023-01-27 21:03:54 +0200  1) using UnityEngine;
c7686220 (Chesst1k 2023-01-27 21:03:54 +0200  2) using UnityEngine.EventSystems;
c7686220 (Chesst1k 2023-01-27 21:03:54 +0200  3)
c7686220 (Chesst1k 2023-01-27 21:03:54 +0200  4) public class GameManager : Mono
Behaviour
c7686220 (Chesst1k 2023-01-27 21:03:54 +0200  5) {
c7686220 (Chesst1k 2023-01-27 21:03:54 +0200  6)     public static GameManager I
nstance { get; private set; }
c7686220 (Chesst1k 2023-01-27 21:03:54 +0200  7)     public AudioManager AudioMa
nager { get; private set; }
c7686220 (Chesst1k 2023-01-27 21:03:54 +0200  8)     public GameResources GameRe
sources { get; private set; }
0fabc521 (Chesst1k 2023-02-15 23:07:07 +0200  9)     public UIManager UIManager
{ get; private set; }
3e4ea8a9 (Chesst1k 2023-03-01 23:15:57 +0200 10)     public QuestsManager QuestM
anager { get; private set; }
9f01421f (chesst1k 2023-05-04 00:05:47 +0300 11)     public MinigamesManager Min
igamesManager { get; private set; }
fcd10cd0 (chesst1k 2023-05-04 23:57:29 +0300 12)     public ActionsQueue Actions
Queue { get; private set; }
293842c3 (chesst1k 2023-05-08 23:57:17 +0300 13)     public GameInfo GameInfo {
get; private set; }
c7686220 (Chesst1k 2023-01-27 21:03:54 +0200 14)
:
```

Рис 2.3. Команда git blame

Git надає можливість відновити стан файлу на момент кожної внесеної до репозиторію зміни. Така можливість дозволяє сміливо експериментувати, оскільки якщо щось піде не так то завжди можна повернутись назад до моменту

коли усе працювало правильно. Зазвичай всі файли git зберігаються на віддаленому сервері, доступ до якого забезпечується за допомогою одного із доступних протоколів, найпопулярнішим із яких є SSH. Щоб відправити зміни на сервер використовують команду git push. Також git надає можливість зберегти файли локально за допомогою команди git stash і потім використати їх у потрібний момент. Для розробки гри-симулятора студентського життя було використано сервіс Github - один із найбільш популярних веб-сервісів для командної розробки програмних рішень. Він базується на Git та є безкоштовним для проектів із відкритим кодом. На Github було створено репозиторій для розроблюваної гри, доступ до якого здійснюється за допомогою встановленого Git із використанням SSH ключа.

Після установки Git пропонує для роботи консоль (стандартну або Git Bash), а також додаткову установку Git GUI для роботи з візуальним інтерфейсом Git.

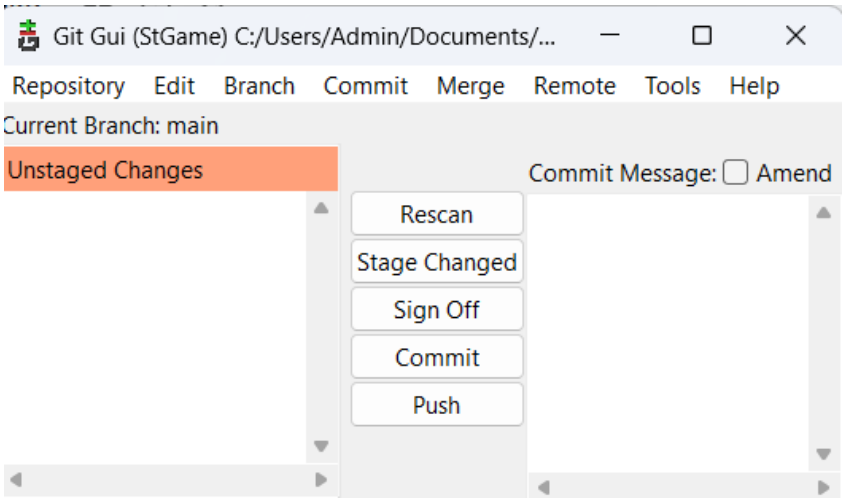

Рис 2.4. Git GUI

Для роботи із Git розроблено багато варіацій клієнтів із різними візуальними інтерфейсами, кожен із яких має свій інтерфейс. Таких Git клієнтів є досить багато і усі вони мають за мету спрощення роботи із технологією. Для мене із них найбільш зручним для роботи із невеликим проектом став продукт Sourcetree від компанії Atlassian.

Sourcetree є безкоштовним Git клієнтом для платформ Windows та Mac OS. Він має зручний та зрозумілий інтерфейс тому є чудовим вибором для

початківців. Найбільш вагомою перевагою Sourcetree над іншими Git клієнтами є те, що він надає можливість візуально відображати зміни та відстежувати що і коли було змінено у файлі, розширюючи функціонал стандартної команди git log, яка у звичайному Git лише показує історію коммітів у який була присутньою зміна відстежуваного файлу.

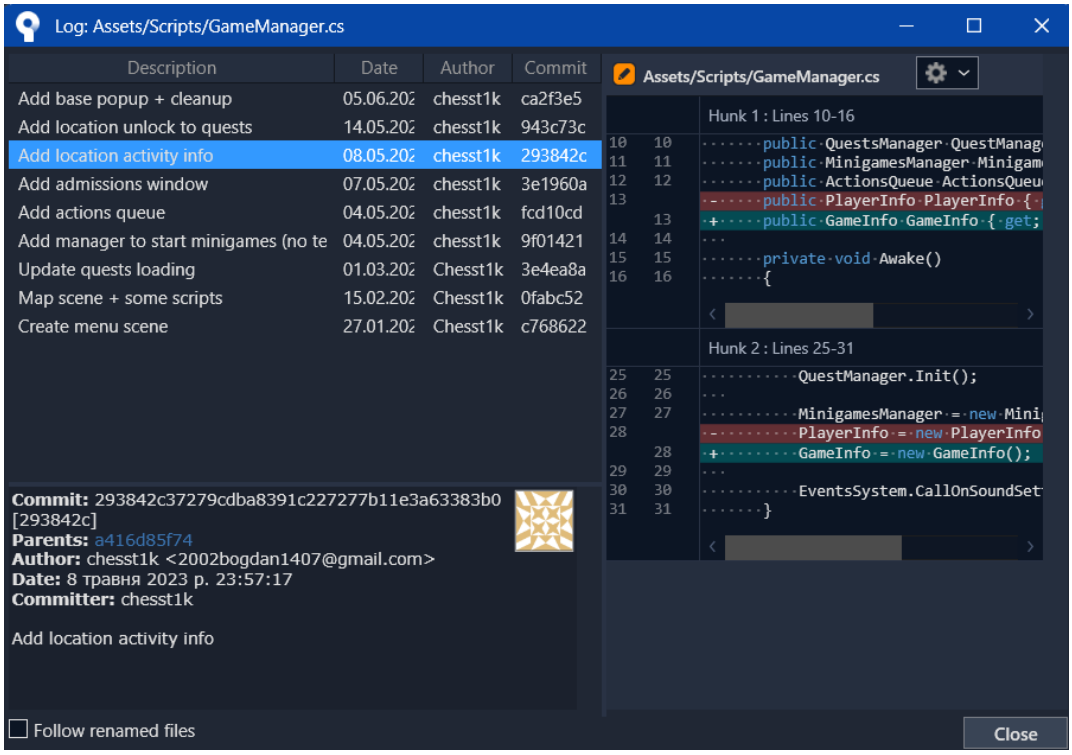

Рис 2.5. Sourcetree Log

Окрім описаних вище переваг у Sourcetree є певні недоліки. Найбільш помітними із них є неправильна робота підключень до репозиторіїв при використанні декількох SSH ключів, сильні зависання при виконанні git команд у репозиторіях із великою кількістю файлів або при прийнятті об'ємних змін до робочої копії. Для невеликих репозиторіїв ці проблеми не суттєві, тому Sourcetree чудово підходить для нашої гри-симулятора.

2.2. Розробка концепту гри

Гра-симулятор буде розповідати про буденні ситуації у житті студента після вступу до університету. Щоб продемонструвати різноманіття можливих ігрових механік, було вирішено розробити гру як набір пов'язаних між собою мініігор. Розроблені мініігри будуть пов'язані між собою сюжетом, який в свою чергу потребуватиме їх проходження для розвитку історії. Сюжетна лінія гри

буде обертатись навколо персонажа-студента та включатиме в себе вибір спеціальності та вступ до університету НУВГП. Далі новому студентові доведеться познайомитися із системою коридорів у корпусі, а також підготуватись до свого першого екзамену та успішно здати його. Як основну ігрову локацію використаємо макет НУВГП. Щоб реалізувати такий проект доведеться створити декілька мініігор, систему для керування сюжетом та логіку для взаємодії гравця із ними.

Для розробки гри-симулятора студентського життя створимо проект за допомогою обраного рушія Unity. Спочатку підготуємо спрощений варіант для сюжетної лінії, щоб визначити які допоміжні елементи знадобляться для її реалізації у розроблюваній грі-симуляторі.

Для реалізації обраного проекту буде достатньо трьох ігрових сцен, а саме: сцени яку показуватимемо при запуску, основної ігрової сцени, та сцени для роботи із мініграми. Будемо починати гру зі сцени зі стартовим меню з кнопками для старту, виходу з гри та відкриття вікна із налаштуваннями, у якому можна буде змінити гучність звуку та музики або вимкнути їх. При натисканні на кнопку старту гри відбуватиметься перехід до основної ігрової сцени, на якій будуть відбуватися взаємодії із головною логікою гри.

Після завершення завантаження основної ігрової сцени спочатку покажемо гравцю стартовий діалог із коротким поясненням світобудови розроблюваної гри, у якому направимо його до виконання наступного завдання. Було вирішено зібрати деякі дані про гравця, тому після першого діалогу потрібно показати вікно реєстрації із запитом на введення визначених даних. Щоб пояснити такий запит у контексті сюжету гри додамо окрему локацію для приймальної комісії. Локацій у грі знадобиться декілька, але на потрібний момент сюжету доведеться вимкнути усі крім потрібної щоб не заплутувати гравця. Перехід на локацію буде відбуватись за допомогою кнопки, яка при натисканні відкриватиме спливаюче вікно із пропозицією переходу до приймальної комісії. Щоб зекономити час та ресурси було вирішено замість повноцінної ігрової сцени для візуального відображення потрібної локації показувати окреме вікно на повний екран, яке б

створило у гравця необхідне відчуття переходу і при цьому виконувало функцію збору даних за допомогою полів. Для забезпечення можливості введення інформації використаємо компонент Input Field, реалізований у рушії Unity. Ввід даних у вікні буде обов'язковим, тому для нього вирішено вимкнути можливість закриття за допомогою кнопки та кліку поза вікном, які є необхідними для більшості вікон для коректної роботи та в будь-якому випадку буде розробленою для базової реалізації вікна.

Після завершення вводу даних та натискання кнопки підтвердження сховаємо вікно реєстрації та покажемо іще один діалог у якому відмітимо прогрес гравця. Під час показу діалогу додатково оновимо дані про доступні йому локації.. Для цього приховаємо локацію приймальної комісії та відкриємо доступ до наступних локацій. У діалозі вкажемо на відкриття нових локацій та дамо гравцеві нове завдання, що полягатиме у відвідуванні усіх відкритих локацій та виконанні додаткових завдань у них.

На даному етапі будуть доступними всі локації із мініігами, яких буде всього три. Було вирішено створити кілька мініігор, які своєю логікою нагадуватимуть колись популярні ігри зі старих пристроїв, а саме:

1. Memory Matching
2. Whack A Mole
3. Avoid Obstacles

Кожна із цих ігор має просту, але унікальну ігрову механіку. У результаті такий набір мініігор з однієї сторони дозволить нам продемонструвати більше архітектурних можливостей при розробці ігор зі сторони коду, а з іншої забезпечить різноманіття ігрових механік для гравця, що дозволить зацікавити його та утримувати у грі протягом якомога довше.

Щоб реалізувати перехід гравця на нові локації використаємо ту ж логіку що й для локації приймальної комісії із кнопкою на ігровій сцені та спливаючим вікном. При цьому після підтвердження переходу у цьому вікні будемо показувати додаткові діалоги для надання короткого опису локації та сюжетної ситуації у яку нас поставить мінігра. Після закриття діалогу буде відбуватись

перехід до локації з мінігрою, яка на відміну від приймальної комісії являтиме собою окрему ігрову сцену з додатковою логікою. Якщо гравцю вдасться перемогти у мінігрі то при поверненні до основної сцени буде показано ще один додатковий діалог для відзначення успішного завершення визначеного етапу гри. При цьому завершена локація позначатиметься як пройдена та буде відключена візуально, щоб гравець не мав можливості проходити один і той же шматок сюжету знову і знову.

Усі мініігри будуть мати багато спільних елементів, але при цьому кожна окрема мінігра потребуватиме розробки особливої логіки. До спільних рис мініігор віднесемо логіку запуску, перевірку умов перемоги, перезапуск при програші та нарахування очок за правильно виконані дії. Проаналізуємо усі три мінігри.

Почнемо із логіки мінігри Memory Matching. Memory Matching (ігри на відповідність) - це жанр ігор у яких гравці повинні встановлювати відповідності між певними елементами. Ними можуть бути картинки, слова, картки, звуки чи будь-які інші об’єкти які можна поєднати[12]. Такі об’єкти зазвичай є відкритими гравцю, як наприклад при грі у маджонг. Також вони можуть бути частково відкритими, як при грі у доміно у якій відкритою є лише одна із двох фішок які необхідно поєднати. Крім цього елементи для поєднання бувають прихованими. У такому випадку гравцям надається певний час щоб запам’ятати розміщення елементів, а по його завершення вони приховуються і потрібно згадати розміщення відповідних елементів по пам’яті.

Для розробки потрібної нам мінігри використаємо останній згаданий варіант із повністю прихованими елементами для запам’ятовування. Він найкраще підійде у нашому випадку оскільки його реалізація є простішою за інші та дозволяє виключити наявність випадковостей яка часто дратує гравців.

Як елементи для запам’ятовування було вирішено використати інтуїтивно зрозумілі картки. Для демонстраційного рівня знадобиться наперед створене поле із розміщеними на ньому об’єктами картками. Крім готового рівня задамо також набір картинок для карток. Ці картинки будемо встановлювати випадково

при старті та перезапуску рівня, щоб гравцеві доводилось запам’ятовувати позиції карток заново після кожної невдалої спроби. Оскільки на рівні мінігри не буде великої кількості елементів для запам’ятовування, дамо гравцеві три шанси на помилку. Трьох спроб є достатньо щоб продовжити гру при випадковому виборі неправильної комбінації карток, але водночас достатньо мало для того щоб забрати у гравця можливість переглянути усі елементи поля за рахунок них.

Для перемоги у грі гравцеві потрібно буде зібрати всі картки на полі, а для поразки тричі помилитися.

Наступною грою є Whack A Mole - це колись популярна аркадна гра, що з’явилась у Японії у 1975 році під назвою Mogura Taiji. У цій версії гри ігрове поле представляло собою плоску поверхню із ямками, у яких були заховані різні об’єкти (зазвичай фігурки кротів). Раз у певний проміжок часу один чи кілька кротів вилазили із ямок на короткий час і гравцеві потрібно було влучити по них спеціальним молотком. Якщо гравець встигав це зробити до того як кріт сховається назад до ями, то йому нараховувались ігрові очки. Ціллю гри зазвичай було якомога швидше набрати певну кількість очок або навпаки за визначений час набрати максимально можливу кількість очок[13].

Щоб розробити гру такого типу знадобиться лише підготувати ігрове поле у вигляді статичного фону та кілька об’єктів по яких потрібно буде натиснути гравцю за певний проміжок часу.

При натисканні по об’єкту-кроту будемо додавати гравцеві одне ігрове очко, якщо ж гравець не встигне натиснути, то зніматимемо також одне очко. Для відстежування натискань використаємо компоненти для взаємодії із фізикою, а саме RaycastHit та Collider, що реалізовані у рушії Unity. Щоб перемоги у даній мінігрі гравцеві потрібно буде набрати певну визначену кількість очок, а для поразки довести кількість очок до від’ємної. При перезапуску мінігри будемо очищати поле та створювати кротів заново.

Останньою із списку мініігор які було вирішено реалізувати є гра на основі механіки Avoid Obstacles - уникнення перешкод. Така механіка є присутньою у багатьох іграх у різному вигляді. Зазвичай у них є об’єкт-персонаж, яким

напряму або за допомогою команд може керувати гравець. У найбільш популярному варіанті використання механіки гравцеві потрібно провести персонажа від початкової точки до пункту призначення або якомога далі, уникаючи при цьому можливих динамічних перешкод що з'являтимуться на його шляху та оминаючи статичні об'єкти. Другим популярним варіантом реалізації механіки є ситуація, коли гравець може переміщатись у обмеженому просторі, а до нього рухаються предмети від яких потрібно ухилятись. На основі такої механіки зародився жанр Bullet Hell. Окрім цього вона часто є частиною інших ігрових механік у популярних іграх[14].

Щоб реалізувати необхідну мінігру було вирішено використати варіант на основі механіки Bullet Hell. Для цього нам знадобиться створити персонажа та логіку для управління ним для гравця. Крім цього потрібно забезпечити створення перешкод та налаштувати їх рух до гравця.

Для забезпечення руху персонажа використаємо систему вводу рушія Unity, яка надає можливість зчитувати те що вводить користувач. Вона є достатньо зручною та має можливість задання різних типів вводу що є корисним при розробці ігор та програм для мобільних пристроїв. При отриманні вводу від гравця у конкретному кадрі будемо рухати його по потрібній осі, яку визначимо за допомогою системи вводу. Крім руху персонажа нам також буде необхідно зчитувати його колізії із об'єктами-перешкодами. Для цього використаємо згадану вище реалізацію фізики у рушії. При колізії із перешкодою гравець програє гру, тому знадобиться також реалізувати передачу інформації про програш у логіку мінігри.

Коли персонажа буде створено, знадобиться також розробити алгоритм для руху перешкод. Спробуємо створити у гравця відчуття того, що персонаж рухається вперед із певною швидкістю при цьому не рухаючи самого персонажа щоб логіка його переміщення вперед по ігровому полю не перетиналася із рухом, який буде задавати користувач при керуванні персонажем. Щоб створити ілюзію руху нам знадобиться рухати фон та об'єкти із перешкодами у сторону гравця (тут повторюємо логіку жанру Bullet Hell). Для цього нам знадобиться створити

фон та перешкоди та задати їм переміщення з певною швидкістю. Щоб спростити реалізацію такої задачі було вирішено використати логіку одного із згаданих раніше патернів - пулу об’єктів (Object Pool). Створимо фон та перешкоди на ньому як один заготований об’єкт що представлятиме собою невелику ігрову зону та будемо створювати його перед персонажем так, щоб це не було помітним для гравця. Щоб визначити чи бачить гравець об’єкт використаємо його позицію та компонент камери рушія, який є наявним завжди. При цьому об’єкти через які гравець уже пробіг будемо використовувати знову і знову, переміщаючи їх так, щоб вони з’являлись перед гравцем у невидимій для нього зоні ігрової сцени. Під час переміщення пройденої зони додатково будемо міняти розміщення перешкод на ній, щоб гравцю не здавалось що він пробігає одні і ті ж частинки гри знову і знову.

Щоб гру можна було закінчити, будемо нараховувати гравцю очки із плином часу та оголосимо про його перемогу при досягненні певної кількості очок.

У підсумку маємо потребу у розробці таких елементів гри:

- Мініігри
- Система для контролю сюжету (включатиме у себе також контроль прогресу гравця)
- Клас-менеджер для забезпечення доступу
- Система сповіщень
- Логіка контролю ігрових ресурсів
- Логіка взаємодії із користувачем
- Система для відображення вікон
- Черга подій

РОЗДІЛ 3. РОЗРОБКА ГРИ-СИМУЛЯТОРА СТУДЕНТСЬКОГО ЖИТТЯ

3.1. Програмування ігрового процесу

Більшість звернень до різних типів сутностей, що реалізовані у проекті, відбуваються за допомогою класу GameManager. У ньому реалізований патерн Singleton(Одинак). GameManager надає можливість доступу до основного функціоналу проекту, а саме підкласів що відповідають за контроль певного аспекту гри. Вони записуються у GameManager при його ініціалізації, що відбувається при створенні ноди із класом на першій ігровій сцені.

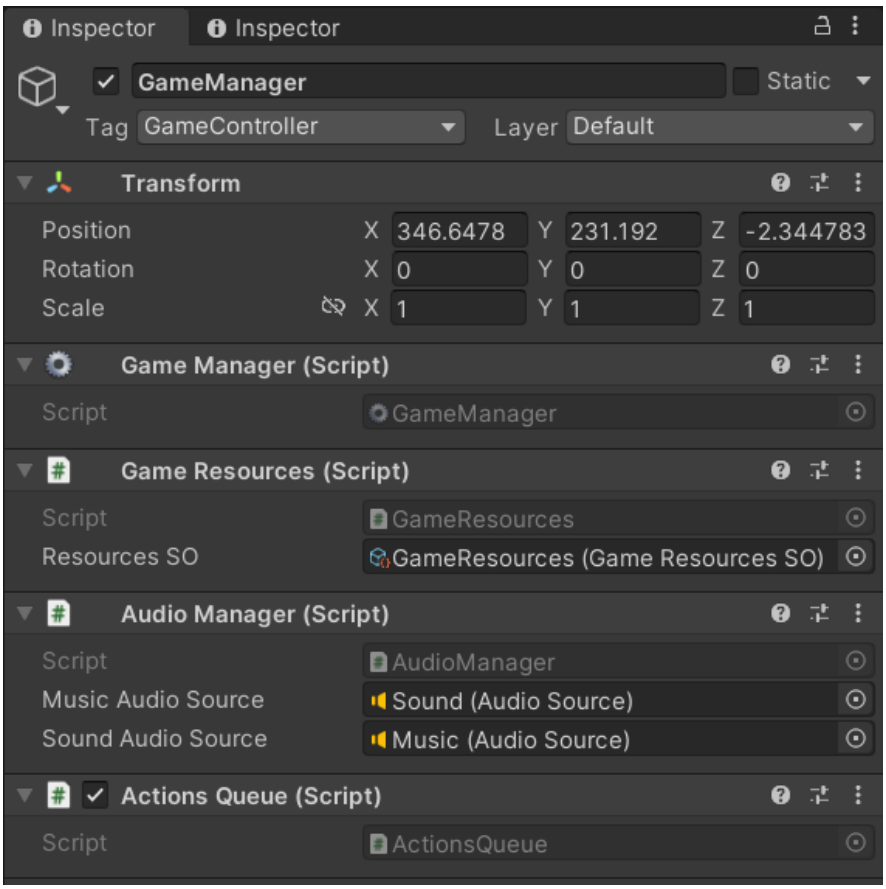

Рис 3.1. Нода GameManager

При створенні усіх об’єктів ігрової сцени що наслідуються від класу UnityEngine.MonoBehaviour, який є вбудованою частиною рушія Unity, рушій викликає у них метод Awake(). GameManager використовує цей метод для ініціалізації та реалізації патерну Singleton. У ньому встановлюється Instance - публічна статична змінна через яку відбуваються всі подальші звернення до класу у програмі та властивість dontDestroyOnLoad, яка сигналізує рушію про те,

що об’єкт не потрібно знищувати при переході між сценами (за замовчуванням при вивантаженні сцени всі об’єкти-ноди знищуються). Далі GameManager бере створює всі необхідні класи або дістає їх за допомогою методу GetComponent, який дозволяє отримати доступ до класу екземпляр якого створений на підготовленій ноді. Окрім цього при потребі GameManager може викликати ініціалізацію доступних для нього компонентів програми (наприклад для функціоналу квестів).

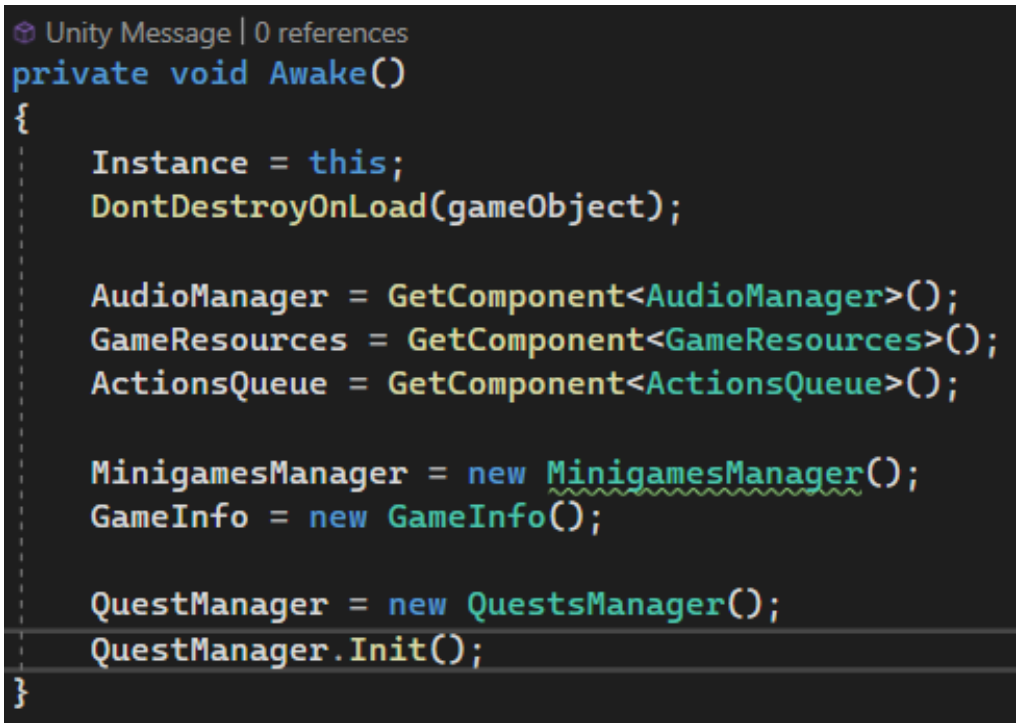

Рис 3.2. Ініціалізація GameManager

Окрім GameManager у проекті реалізована іще одна сутність, доступ до якої можна отримати із будь-якого місця у проекті, а саме EventSystem.

EventSystem являє собою один із варіантів реалізації системи сповіщень. Клас EventSystem містить у собі оголошення делегату EventHandler що приймає у себе дані у вигляді екземпляру класу EventData, статичні екземпляри цього делегату та статичні методи, що реалізують у собі виклики до відповідних екземплярів EventHandler.

```
18 references
public class EventsSystem
{
    public delegate void EventHandler(EventData data);

    public static event EventHandler OnSoundSettingsUpdateNeeded;
    0 references
    public static void CallOnSoundSettingsUpdateNeeded(EventData data = null)
    {
        if (OnSoundSettingsUpdateNeeded != null)
        {
            OnSoundSettingsUpdateNeeded(data);
        }
    }
```

Рис 3.3. Клас EventSystem

Клас EventData використовується для передачі потрібних даних із однієї частини програми у іншу. EventData містить у собі мапу, у якій ключем є рядок типу string, а значенням об’єкт типу object. Оскільки object є базовим типом, від якого успадковані усі інші можливі типи, то у мапу можна об’єкт будь-якого потрібного типу від базових (наприклад символьний тип - char) до найбільш комплексних.

Описаний вище підхід до реалізації сповіщень зазвичай використовується у маленьких іграх, оскільки дозволяє прискорювати розробку за рахунок відкидання потреби у сортуванні типів сповіщень та даних що передаються у них. Недоліком такого варіанту реалізації є те, що при збільшенні кількості можливих сповіщень у них легко заплутатись через те, що кожне сповіщення вимагає створення окремого екземпляру EventHandler, статичного методу для виклику та унікального програмного рішення для обробки даних, які потрібно передати та опрацювати.

Як було описано вище - доступ до всіх основних функцій відбувається через GameManager, під час ініціалізації якого вони створюються або просто записуються за допомогою GetComponent. Єдиним винятком із цього процесу є клас UIManager що відповідає за взаємодію із інтерфейсом. UIManager зберігає у собі посилання на Canvas - один із вбудованих у рушій елементів, який є необхідним для роботи із інтерфейсом. Для кожної ігрової сцени такий Canvas створюється окремо, тому було прийняте рішення створювати UIManager для

кожного такого екземпляру візуального інтерфейсу. При створенні UIManager записує себе у GameManager, тому можна отримати до нього доступ на кожній із сцен, які потребують відображення ігрового інтерфейсу.

Для реалізації інтерфейсу для взаємодії із користувачем знадобилось два типи елементів. Першим із них є вікна, що відкриваються на весь екран та дозволяють виконати багато дій, як наприклад вікно реєстрації, куди гравець вводить ім’я (рис 3.4) чи вікно для показу діалогів.

Рис 3.4. Вікно вступу

Другим типом UI стали маленькі спливаючі вікна (далі - попапи). Вони вміщують у себе мінімальну кількість контенту і тому використовуються переважно для виконання простих дій, для прикладу - спливаюче вікно для переходу в локацію.

Рис 3.5. Попап локації

UIManager містить у собі реалізовані методи для створення вікон та попапів у об’єкті, на якому міститься Canvas. Подальша робота із створеними об’єктами виконується у місці виклику методів.

Базовий клас попапу (BasePopupScript) містить у собі метод для задання позиції та показу і приховування об’єкту. Для базового класу вікна (BaseWindow) аналогічно реалізовані методи показу і приховування, а позиція задана статично за допомогою константи. Також для вікон реалізовані кнопка закриття та логіка, яка надає можливість закривати вікно при натисканні поза його межами.

Для забезпечення своєї роботи гра використовує велику кількість ресурсів різних типів. Доступ до цих ресурсів потрібен у більшості класів проекту і щоб полегшити його було створено клас GameResources. Він зберігає у собі велику кількість контейнерів типу ключ-значення для ресурсів різних типів. Для кожного типу ресурсів є окрема ініціалізація, де у контейнери записуються ресурси та ключі до них. Також реалізовано методи для отримання посилання на ресурс із використанням конкретного ключа.

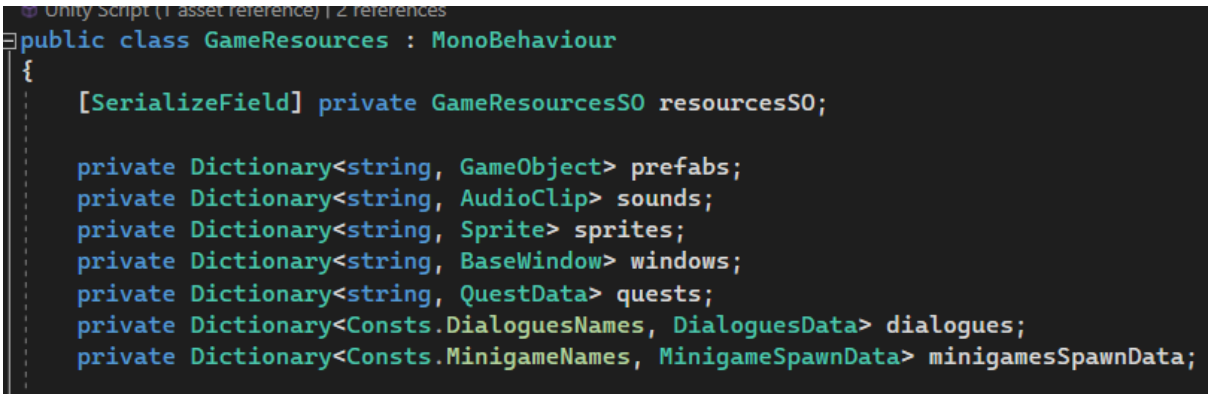

Рис 3.6. GameResources

У розробленій грі усі ресурси одного типу зберігаються разом. Такий варіант підходить для маленьких ігор, оскільки кількість ресурсів до яких потрібен доступ зі сторони коду у них достатньо мала. У великих іграх зазвичай ресурси більш диверсифіковані, бо інакше у них було б надто складно орієнтуватись.

Клас GameResources допомагає працювати із двома видами ресурсів. Перший із них представляє собою класи рушія, які є загальними та використовуються майже у всіх проектах. Це можуть бути картинки, звуки, 3d моделі, об’єкти типу GameObject чи інші ресурси, правильна обробка яких реалізована у рушії. Другим типом ресурсів є усі інші типи, реалізовані на стороні проекту. Зазвичай ними можуть бути будь-які класи та структури даних, які мають атрибут System.SerializableAttribute і всі елементи яких є простими типами або також мають необхідний атрибут (як наприклад більшість класів рушія Unity).

Ресурси додаються до GameResources за допомогою GameResourcesSO, який наслідує клас UnityEngine.ScriptableObject. У ньому створено масив для кожного потрібного типу ресурсів. GameResourcesSO використовує реалізований у рушії атрибут для роботи із об’єктом класом у вікні рушія UnityEngine.CreateAssetMenuAttribute, який дозволяє редагувати об’єкт класу прямо у редакторі Unity. Для GameResourcesSO було створено окремий об’єкт у рушії, який потім підтягується до основного класу GameResources за посиланням.

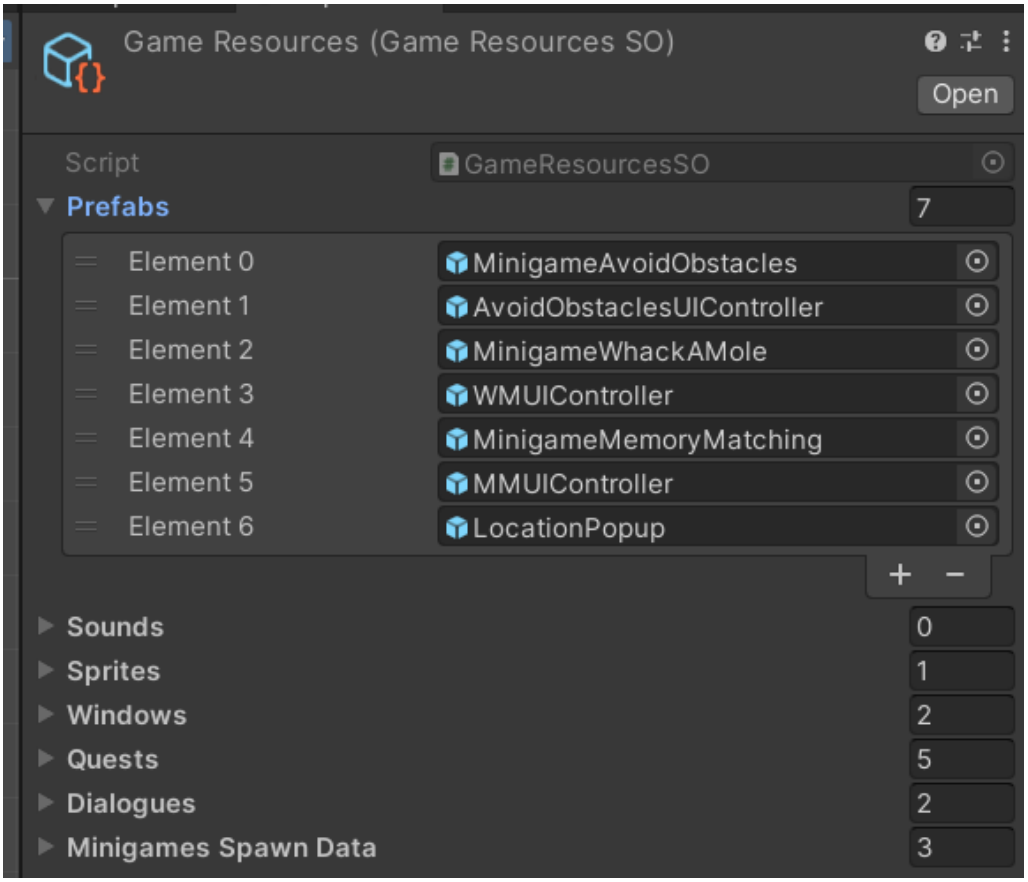

Рис 3.6. Ассет GameResources

Для реалізації сюжетної лінії був створений окремий функціонал, основу якого складають класи: Quest, QuestManager та QuestAction. Сюжетна лінія представляє собою список квестів, кожен із яких складається із декількох частин. Кожна частина в свою чергу є певним набором дій, які одночасно виконуються.

Спочатку розберемо принцип роботи базової квестової дії - QuestAction. Вона являє собою абстрактний клас, що містить у собі потрібний йому набір даних у вигляді списку пар рядків та методи для запуску та завершення дії, тип дії відповідно до якого вона створюється із коду, а також абстрактний метод у якому класи-нащадки виконують обробку даних.

```
17 references
public abstract class QuestAction
{
    protected List<StringStringPair> ActionData;

    2 references
    public bool IsCompleted { get; private set; }

    6 references
    public QuestAction(List<StringStringPair> actionData)
    {
        ActionData = actionData;

        SetupData();
    }

    4 references
    public virtual void StartAction()...

    6 references
    public void Complete()
    {
        IsCompleted = true;
        GameManager.Instance.QuestManager.UpdateQuestsStates();
    }

    6 references
    public abstract QuestActionType GetActionType();

    7 references
    protected abstract void SetupData();
}
```

Рис 3.7. Базовий клас квестової дії

При створенні кожен QuestAction викликає обробку даних та метод для старту дії. Також там задаються умови при яких дія вважається завершеною. Умови завершення є унікальними для кожного із класів-нащадків QuestAction, тому дії можуть чекати конкретних подій (наприклад переходу між ігровими сценами) або завершуватись відразу після старту (наприклад дія що встановлює статус локації).

Quest є сюжетною одиницею, яка відповідає певну частину історії. Для прикладу візьмемо один із квестів що пов’язані із проходженням мінігри. Структура квесту буде такою:

1. Показ вступного діалогу що пояснює завдання.
2. Очікування переходу на ігрову сцену, на якій відбувається взаємодія із мінііграми.
3. Запуск потрібної мінігри.
4. Очікування повідомлення про успішне проходження потрібної мінігри гравцем.
5. Очікування переходу на основну ігрову сцену.
6. Показ завершального діалогу.
7. Блокування локації із пройденою мінігрою.

Для реалізації вище описаного квесту знадобиться розбити цей список дій на окремі частини, оскільки гравцю буде незручно одночасно грати у мінігру та читати діалог. При цьому частину дій зручно запускати одночасно. Для прикладу - можна блокувати пройдену локацію не чекаючи завершення діалогу, оскільки це ніяк не вплине на сприйняття гравця.

Кожна із частин квесту містить у собі список дій, які необхідно виконати для переходу до наступної частини. При старті частини квесту Quest створює та запускає усі QuestAction, що містяться у даній частині, за допомогою методу який створює потрібну дії відповідно до її типу. Кожна із дій, клас якої є успадкованим від QuestAction має свою унікальну умову завершення та методи для її обробки. При завершенні кожен QuestAction викликає оновлення даних квесту у QuestManager, а також перевірку чи мають усі дії у активній частині квесту статус Completed. Якщо при такій перевірці усі дії позначено як виконані, то Quest повторює цей процес для своєї наступної частини.

Клас Quest зберігає у собі заготований набір даних (QuestData), що містить його назву, назви квестів які потрібно завершити для розблокування даного, список частин квесту у вигляді масиву QuestAction, а також змінну що відповідає за відключення квесту. Якщо квест позначено як відключений, то при перевірці статусу він не враховується. Описаний вище набір даних є статичним, оскільки він задається один раз при створенні ресурсу квесту та не має можливості змінюватись при проходженні гри. Також клас Quest зберігає у собі деякі

динамічні дані, якими потім користується QuestManager. До цих даних відносяться статус квесту та його активна стадія.

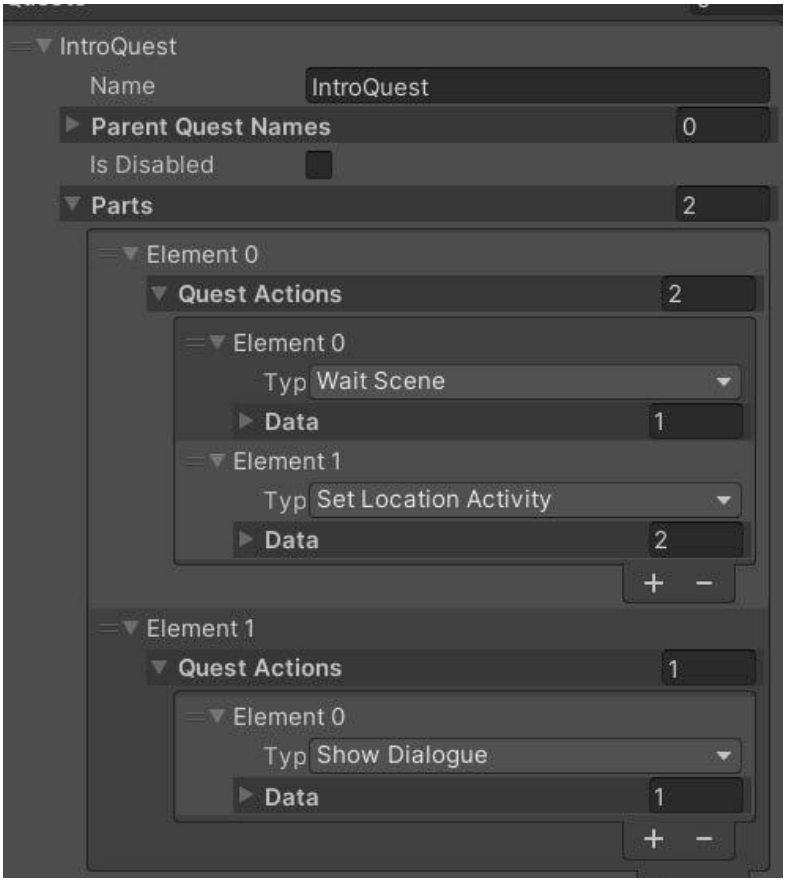

Рис 3.8. Дані вступного квесту

Ядром квестової системи є клас QuestManager, що містить у собі список квестів та методи для роботи із ними. Він створюється при ініціалізації класу GameManager та при створенні завантажує усі дані квестів за допомогою GameResources, що містить у собі окрему колекцію для даних квестів. Такий варіант збереження чудово підходить для невеликих проектів, але у більшості ігор сюжетні лінії потребують створення окремих ресурсів для збереження даних сюжету (зазвичай кожному квесту відповідає окремий файл). При цьому проблем із завантаженням даних сюжету не виникає навіть у маштабних проектах, оскільки необхідні ресурси достатньо компактні та мають просту і зрозумілу структуру.

```
1 reference
public void UpdateState()
{
    if (IsCompleted) {
        return;
    }

    if (questData.IsDisabled) {
        CompleteQuest();
        return;
    }

    if (!IsStarted) {
        if (CanRunQuest()) {
            StartQuest();
        }
        return;
    }
    else {
        TryStartNextPart();
    }
}
```

Рис 3.9. Оновлення стану квесту

При завантаженні даних квестів QuestManager створює екземпляри класу Quest та викликає у них оновлення статусу. При такому оновленні кожен з квестів перевіряє свій теперішній статус та змінює його при потребі. Якщо квест не був запущений, то перевіряється можливість запуску. Для активних квестів перевіряються факти виконання усіх QuestAction що були запущені у активній стадії. Завершені квести не оновлюють свій статус.

Архітектурне рішення що забезпечує собою роботу квестової системи частково нагадує собою логіку роботи патерну “Ланцюжок обов’язків”. Воно забезпечує послідовне виконання квестів та їхніх частин, зв’язуючи ці невеликі структурні елементи в один цільний сюжет. Виразною особливістю системи квестів є те, що вона використовує для роботи наперед задані дані і цим забезпечує автономність системи. Проте іноді виникає потреба запустити деякі дії відокремлено від сюжету. Як приклад розглянемо дії при натисканні гравця на кнопку із попапом локації. За допомогою цього попапу запускаються різні

мініігри, при цьому спочатку відбувається очікування переходу на потрібну сцену а тоді вже створення мінігри. Для такої логіки можна використати варіант реалізації зі створенням квесту, але він потребуватиме створення квестів для кожної із мініігор та реалізації логіки для перезапуску квесту після завершення і тому не є достатньо зручним та очевидним. Щоб вирішити цю проблему було створено аналог квестової системи - клас ActionsQueue.

Логіка ActionsQueue є простішою ніж у квестової системи, оскільки не потребує задання та обробки даних із файлів ресурсів. Для реалізації було створено однойменний клас-менеджер ActionsQueue та базовий абстрактний клас дії QueueAction. Аналогічно до класу QuestAction у QueueAction наявні поля що визначають статус дії та методи для старту та завершення дії. Базовий клас QueueAction не зберігає у собі жодних даних, але для більшості класів нащадків вони потрібні (наприклад щоб знати яке вікно показати чи яку гру запустити) та задаються за допомогою конструктора класу.

Механізм ActionsQueue працює за допомогою вбудованого у рушій механізму корутин. Корутини дозволяють виконувати код із затримкою чи після виконання певної умови. При ініціалізації ActionsQueue запускає головну корутину MainCoroutine, яка в свою чергу у вкладеній корутині намагається доступитись до першої дії у черзі. Якщо така дія існує, то вона запускається і далі корутина очікує її завершення і видаляє її із черги. MainCoroutine є рекурсивною, тому викликає себе ж при завершенні. Така логіка дозволяє нам додавати дії до ActionsQueue і бути при цьому впевненими що вони нічого не зламають.

```
2 references | chesst1k, 32 days ago | 1 author, 1 change
IEnumerator MainCoroutine()
{
    yield return StartCoroutine(StartNextAction());
    yield return StartCoroutine(MainCoroutine());
}

1 reference | chesst1k, 30 days ago | 1 author, 2 changes
IEnumerator StartNextAction()
{
    if (actions.Count == 0) {
        yield return new WaitForEndOfFrame();
        yield break;
    }
    actions[0].StartAction();
    yield return new WaitUntil(() => { return actions[0].IsCompleted; });
    actions.RemoveAt(0);
}
```

Рис 3.10. Корутини ActionsQueue

3.2. Реалізація мініігор

Основою розробленої гри-симулятора студентського життя є мініігри, розберемо загальну логіку їхньої роботи та кожну мінігру окремо.

```
Unity Script | 3 references | chesst1k, 18 hours ago | 1 author, 1 change
public abstract class BaseMinigameController : MonoBehaviour
{
    protected bool gameOver = false;
    protected bool gameStarted = false;
    protected bool win = false;

    7 references | chesst1k, 18 hours ago | 1 author, 1 change
    protected abstract void StartGame();

    6 references | chesst1k, 18 hours ago | 1 author, 1 change
    protected abstract void RestartGame();

    6 references | chesst1k, 18 hours ago | 1 author, 1 change
    protected abstract void WinGame();

    8 references | chesst1k, 18 hours ago | 1 author, 1 change
    protected abstract void UpdateUI();

    6 references | chesst1k, 18 hours ago | 1 author, 1 change
    protected virtual void OnKeyInput() { }

}
```

Рис 3.11. BaseMinigameController

Базовий клас для управління мінігрою успадкований від UnityEngine.MonoBehaviour та містить у собі оголошення для кількох абстрактних методів, які необхідні кожній із мініігор для правильної роботи. До них відносяться старт, перезапуск, виграш гри, а також обробка вхідних взаємодій від гравця та оновлення візуальної складової гри. Також у BaseMinigameController було винесено кілька змінних що відповідають за статус гри та рахунок гравця, оскільки вони використовувались при реалізації кожної із мініігор. Для коректної роботи кожної реалізованої мінігри їй необхідні методи та поля із базового класу. Окрім них у кожної мінігри є свій клас для відображення UI та кілька допоміжних скриптів, якщо вони необхідні для реалізації потрібної логіки.

У проекті реалізовано кілька мініігор, кожна із яких має свою унікальну логіку:

4. Memory Matching
5. Whack A Mole
6. Avoid Obstacles

Спочатку ознайомимося із реалізацією мінігри Memory Matching - гри у якій гравець повинен за певний час запам’ятати розташування картинок, після чого вони приховуються і гравцю потрібно знайти однакові.

Для коректної роботи гра використовує три скрипти - MM_MinigameController для основної логіки, MM_UIController для відображення даних гри у інтерфейсі користувача та MM_CardScript що визначає поведінку конкретної картки. MM_MinigameController зберігає у своєму префабі посилання на префаб із контролером візуального інтерфейсу (MM_UIController), контейнер куди будуть інстантіюватись картки із картинками які потрібно буде поєднувати, спрайт закритої картки та масив спрайтів для відкритих карток. Рушій Unity автоматично надає можливість додавання посилань на поля класу, які оголошено публічними і типи яких є успадкованими від вбудованого класу UnityEngine.MonoBehaviour. У нашому випадку для можливості додавання

посилань потрібно додавати полям класу окремий атрибут UnityEngine.SerializeField.

Крім вище описаних посилань клас MM_MinigameController використовує кілька констант та допоміжних полів, а саме:

- Прапорець відповідальний за очікування.
- Кількість спроб що залишились на вгадування (аналог життів із більшості популярних ігор)
- Список карт на полі
- Початкова кількість спроб
- Кількість очок необхідна для виграшу
- Актуальна кількість очок

Оскільки усі створені мініігри мають лише один демонстраційний рівень то частина значень що використовуються у розрахунках є записаними як константи (наприклад кількість очок необхідних для перемоги). При потребі у створенні рівнів із різними граничними значеннями знадобиться створити додаткове рішення для серіалізації даних рівнів.

```
Unity Script (1 asset reference) | 2 references | chesst1k, 6 hours ago | 1 author, 6 changes
public class MM_MinigameController : BaseMinigameController
{
    [Header("Links")]
    [SerializeField] private MM_UIController uIControllerPrefab;
    [SerializeField] private Transform cardsContainer;
    [SerializeField] private Sprite[] frontSprites;
    [SerializeField] private Sprite backSprite;

    private MM_UIController uIController;
    private List<GameObject> cards = new List<GameObject>();

    private bool wait = false;
    private int score = 0;
    private int tryCount = 0;

    const int SCORE_TO_WIN = 10;
    const int TRIES_COUNT = 3;
```

Рис 3.12. MM_MinigameController

Оскільки гра Memory Matching має невелику кількість елементів, то для поля із картками був використаний UI інтерфейс. MM_UIController зберігає у собі посилання на створене поле та список позицій карток, які потім записуються до основного менеджера гри при його ініціалізації. При створенні MM_MinigameController у свою чергу створює префаб із контролером UI за допомогою класу UIManager. Після цього зі створеного префабу береться список позицій карт та записується до основного менеджера гри. Таким чином при ініціалізації ми використовуємо клас MM_UIController як джерело статичних даних для створення мінігри. Такий підхід підходить для розробки прототипів чи при демонстрації ігрових механік як у нашому випадку, проте сильно обмежує можливості для розробки рівнів. У випадку необхідності створення кількох рівнів заготовані поля із картами краще зберігати окремими ресурсами у вигляді або цільного поля або списку параметрів за якими можна відтворити початкове поле.

Після запису у MM_MinigameController даних про розміщення карток на екрані MM_UIController використовується лише за прямим призначенням, а саме - відображає дані якими потрібно поділитися із гравцем для надання йому достатньої для нормального сприйняття гри кількості інформації. MM_MinigameController після отримання даних про розміщення карти вимикає відображення ігрового поля та запускає алгоритм, який вибирає картинку для кожної картки за допомогою вбудованого класу System.Random:

```
var rnd = new System.Random();
        cards = cards.OrderBy(a => rnd.Next()).ToList();
        int cardValue = 0;
        for (int i = 0; i < cards.Count; i++)
        {
          if (i % 2 == 0) {
            cardValue = UnityEngine.Random.Range(0, frontSprites.Length);
          }

          var cardScript = cards[i].GetComponent<MM_CardScript>();
          cardScript.SetupGraphics(frontSprites[cardValue], backSprite);
          cardScript.CardValue = cardValue;
```

}

Скрипт що контролює картку містить у собі посилання на спрайти обох сторін картки і може запам’ятовувати якою стороною вона показана гравцеві. Коли MM_MinigameController вибирає картинку для лицевої сторони картки , вона разом із наперед записаним у контролем спрайтом сорочки записуються у скрипт картки і вона оновлює своє відображення. Крім цього картка зберігає номер картинки для простішого порівняння співпадінь у процесі гри.

Після завершення ініціалізації у першому кадрі гра помічає, що вона ще не почалась і виводить на екран текст із проханням до гравця натиснути на кнопку для запуску гри. Після натискання потрібної кнопки і мінігри викликається один із базових методів - StartGame(), який оновлює її статус, встановлює усі актуальні при старті дані, візуально відображає картки які до цього були вимкненими та запускає показ карт гравцю для запам’ятовування. При показі карт гравцю показується текст, який просить запам’ятати картки, а самі картки перевертаються лицевою стороною до гравця, щоб він мав час їх роздивитись та запам’ятати. Виглядає це ось так:

Рис 3.13. Старт мінігри Memory Matching

Коли картки показались гравцю, гра переходить у статус очікування за допомогою згаданого вище прапорця а також запускає відкладену дію.

Використовуючи можливості рушія Unity можна двома способами запускати код із затримкою. Першим із них є вищезгадані методи-корутини. Можна створити метод-корутину який буде очікувати потрібний час за допомогою команди yield return new WaitForSeconds(time). Іншим варіантом буде створення додаткової змінної-таймеру, якій буде задаватись значення часу який потрібно очікувати у секундах. Далі у методі Update або FixedUpdate потрібно віднімати пройдений час і коли таймер досягне нуля - виконати необхідний код. Обидва ці методи потребують щоб клас був успадкований від UnityEngine.MonoBehaviour та створення додаткових методів або змінних. Щоб уникнути необхідності додавання коду для підтримки такої логіки було використано плагін LeanTween. Він надає потрібну нам можливість виконання коду із затримкою за допомогою одного із вбудованих методів - delayedCall. Цей метод приймає у параметри значення часу та лямбда-функцію у якій буде необхідний код:

```
LeanTween.delayedCall(3f, () =>
    {
      wait = false;
      foreach (var card in cards)
      {
        var cardScript = card.GetComponent<MM_CardScript>();
        cardScript.Flip(false);
      }
      uIController.SetPressToStartText("");
    });
```

Коли час, відведений для запам’ятовування карток гравцем, добігає кінця, картки перевертаються назад за допомогою методу Flip і стартує основна фаза гри. У ній гравцю надається можливість клікати по картках. При кліку картка перевертається лицевою стороною до гравця, при цьому це не працює у зворотну сторону. Коли гравець перевертає дві картки MM_MinigameController запускає перевірку співпадіння їхніх значень. Якщо співпадіння було успішним, то об’єкти карток вимикаються із затримкою (щоб гравець побачив успішне порівняння) і гравцю додається одне очко. У іншому випадку віднімається одна спроба вгадування і картки перевертаються назад із мінімальною затримкою, щоб гравець мав достатньо часу на усвідомлення помилки.

Коли кількість спроб у гравця стає рівною нулю то гра вважається програною. У такому випадку поле блокується і на екран виводиться повідомлення із пропозицією спробувати ще раз. Якщо гравець вирішує погодитись на другу спробу то викликається метод RestartGame() у якому усі значення прогресу скидаються а картки вимикаються. Далі із методу рестарту запускається метод StartGame(), який знову проводить генерацію поля і мінігра починається спочатку. При успішному проходженні мінігра відправляє сповіщення для квестової системи про виграш за допомогою EventSystem та відбувається перехід на основну ігрову сцену.

Далі ознайомимося з найпростішою із реалізованих мініігор - Whack A Mole. Це гра у якій гравцю потрібно натискати на об’єкти що з’являються на екрані щоб набирати очки. Для реалізації такої мінігри знадобилося всього два класи - основний менеджер WM_MinigameController та допоміжний клас WM_UIController для відображення інтерфейсу.

WM_MinigameController містить у собі посилання на контролер UI аналогічно до Memory Matching. Крім нього він зберігає посилання на масив префабів об’єктів що будуть з’являтися на полі і об’єкт-контейнер для них, а також масив наперед заданих позицій у яких можуть з’являтися об’єкти. Для зручного налаштування балансу до редактору також була винесена змінна що відповідає за затримку між появами об’єктів на полі за допомогою UnityEngine.SerializeField.

Запуск мінігри Whack A Mole реалізований простіше ніж у Memory Matching. Грі не потрібен додатковий набір даних оскільки усе задано посиланнями у префабі мінігри. При ініціалізації WM_MinigameController лише інстантіює префаб із скриптом для обробки інтерфейсу, а далі відразу пропонує гравцю запустити гру натисканням кнопки. На відміну від Memory Matching для забезпечення функціоналу мінігри, а саме відстежування натискань, не використовувався функціонал, який надає UI систему рушія. Замість візуального інтерфейсу було використано 2D фізику, що вважається стандартним рішенням

для реалізації обробки кліків гравця по об’єктах які не відносяться до ігрового інтерфейсу.

Основна логіка гри реалізована у методах FixedUpdate() та Update(). Перший із них керує таймером, відповідальним за створення об’єктів. Для цього використовується додаткова змінна, яка у кожному фіксованому кадрі віднімає пройдений час. Коли таймер доходить до нуля на полі у одній із заздалегідь вказаних точок з’являється об’єкт на який потрібно натиснути.

Метод Update() перевіряє чи натиснув гравець на об’єкт за допомогою вбудованого у рушій функціоналу, а саме: UnityEngine.Input щоб відслідкувати факт натискання клавіші чи кнопки та UnityEngine.Physics2D із його функціоналом кидання променів для визначення того, чи влучив гравець по колайдеру. Такий механізм є менш затратним з точки зору продуктивності у порівнянні з аналогами, але потребує наявність одного із доступних фізичних колайдерів на об’єкті по якому відстежуються кліки.

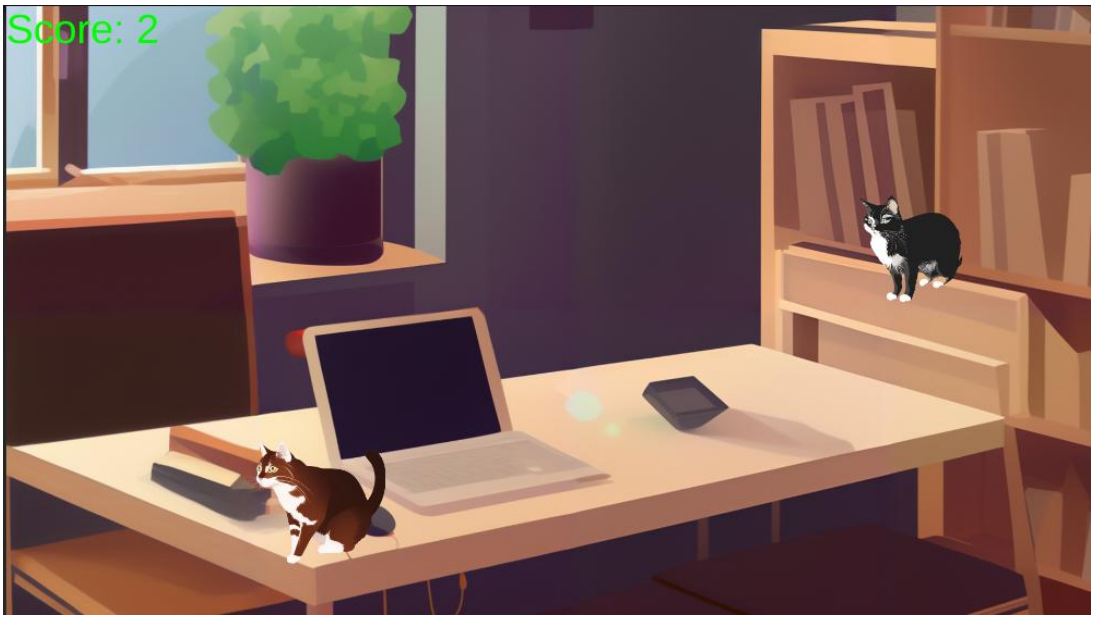

Рис 3.14. Гра Whack A Mole

Коли гравцю вдається натиснути на колайдер потрібного об’єкту, то WM_MinigameController викликає для нього метод OnHit. У цьому методі гравцю додається одне очко до рахунку а сам об’єкт знищується. Якщо ж гравець не встиг знищити один об’єкт і з’явився другий, то йому навпаки списується одне очко із рахунку, при цьому об’єкти залишаються на полі. Коли гравець набирає

потрібну кількість очок, запускається логіка виграшу мінігри, яка працює аналогічно до Memory Matching. У випадку коли кількість очок гравця доходить до нуля гра вважається програною і пропонує гравцю запустити нову спробу. При перезапуску мінігра лише знищує створені об’єкти та очищає список для них, а також обнулює таймер.

Третьою реалізованою мінігрою є Avoid Obstacles. Її концепція полягає у тому, що гравець бере на себе контроль персонажа та повинен уникати перешкод які у нього летять. При цьому із часом швидкість появи перешкод зростає ускладнюючи гру. Аналогічно до попередньо описаних мініігор Avoid Obstacles має основний клас менеджер AO_MinigameController та клас для відображення візуального інтерфейсу AO_UIController. Крім цих двох класів також було створено два допоміжних класи для створення тайлів із перешкодами та контролю гравця. AO_UIController працює аналогічно до контролера UI із гри Whack A Mole, оскільки для мінігри у якій потрібно лише уникати перешкод достатньо лише візуалізації очок набраних гравцем та виведення тексту для вказівок.

Наслідуючи приклад раніше згаданих мініігор AO_MinigameController починає свою роботу із процесу ініціалізації. Спочатку інстантіюється префаб із контролером UI та кешується посилання на головну ігрову камеру для подальшого використання при створенні перешкод. Далі викликається метод у якому оголошується підписка на івент що сповіщає про програш гри. Вона є необхідною, оскільки момент програшу визначається за допомогою класу управління гравцем (AO_Player) який не містить у собі посилання на основний клас-менеджер мінігри.

Після створення підписок та кешування даних відбувається генерація перешкод. Вона використовує наперед заготований префаб тайлу що містить у собі частину рівня яка буде повторюватись знову і знову. Зазвичай у таких тайлах містяться спрайти фону які можна безшовно з’єднати із таким же спрайтом, оскільки логіка гри передбачає те що вони будуть повторюватись. Крім фону у префабі є усі об’єкти із перешкодами які можу зустрітися гравцеві при

проходженні гри. У нашому варіанті реалізації такі перешкоди мають компонент для візуального відображення (спрайт та іноді анімацію для нього) та фізичний 2d колайдер для відстежування колізій із гравцем. Іноді фон та перешкоди зберігають окремо щоб було простіше урізноманітнити гру, проте зазвичай рішення із створенням тайлів є достатньо для комфортного створення рівнів. Для спрощення роботи із тайловим об'єктом для нього було створено окремий клас AO_Tile який має у собі посилання на всі об'єкти-перешкоди, додаткові об'єкти у точках початку та завершення тайлу по горизонталі (оскільки мінігра має горизонтальну орієнтацію) та вміє вмикати випадкову перешкоду за допомогою UnityEngine.Random при цьому вимикаючи усі інші. Для нашого тайлу достатньо однієї перешкоди для підтримки комфортної гри, але для тайлів великого розміру можна вмикати декілька.

AO_MinigameController має посилання на необхідний тайл із перешкодами та винесені у редаткор налаштування для створення тайлових об'єктів. При початковій генерації створюється наперед задана кількість об'єктів-тайлів. Для кожного із них спочатку визначається стартова позиція, далі у цій позиції створюється необхідний об'єкт. Тоді якщо на тайлі не повинно бути перешкод (зазвичай для перших кількох тайлів вони відсутні) то він вимикає усі наперед задані та створені перешкоди. У іншому випадку AO_Tile вмикає одну випадкову перешкоду. Після завершення генерації тайл додається до загального списку тайлів та оновлює значення позиції останнього тайлу щоб можна було визначити де має бути наступний.

При ініціалізації гра створює п'ять тайлів. Такої кількості достатньо щоб заповнити увесь простір який може побачити гравець та задати кілька тайлів наперед. Операції створення та знищення об'єктів тайлів потребують багато обчислювальних ресурсів, тому відповідно до логіки патерну Object Pool для економії використовуються лише створені при запуску мінігри тайли. На такій кількості об'єктів вплив цього способу економії ресурсів може бути непомітним на більшості пристроїв, оскільки зазвичай застосування Object Pool є потрібним для візуальних ефектів (VFX) які створюють багато простих об'єктів кожен із

яких потребує операцій створення та видалення. При цьому Object Pool є простим у реалізації і додатково спрощує відстеження роботи об’єктів, тому є корисним і у нашому випадку. У даному варіанті реалізації мінігри гравець залишається статичним а фон із перешкодами рухається на нього за допомогою методу MoveTiles.

```
void MoveTiles()
{
  if (mainCamera.WorldToViewportPoint(spawnedTiles[0].GetEndPoint().position).x < 0)
  {
    AO_Tile tileTmp = spawnedTiles[0];
    spawnedTiles.RemoveAt(0);
    tileTmp.transform.position = spawnedTiles[spawnedTiles.Count - 1].GetEndPoint().position
        - tileTmp.GetStartPoint().localPosition;
    tileTmp.EnableRandomObstacle();
    spawnedTiles.Add(tileTmp);
  }
}
```

Коли тайл при русі виходить за межі камери з лівої сторони то контролер переміщає його вперед у кінець черги тайлів та викликає нову генерацію перешкод. Альтернативний варіант із статичним фоном теж можливий, але є складнішим для відстеження у редакторі та потребує додаткової логіки із переміщенням камери.

Рис 3.15. Тайл із персонажем

При переміщенні тайлів у методі Update() гравцю також додаються очки за пройдений час. Коли гравець набирає певну наперед задану норму очок, то збільшується швидкість їх набору, а також швидкість із якою рухаються перешкоди. Такий підхід дозволяє ускладнювати гру із часом і дозволяє упевнитись що гравець не зможе грати дуже довго, оскільки обмеження по швидкості не встановлене і у якийсь момент гравець не встигне ухилитися від перешкоди. Для перемоги у мінігрі потрібно набрати деяку наперед задану кількість очок. У випадку якщо ж контролер гравця відстежив колізію із перешкодою за допомогою методу OnCollisionEnter2D() що викликається рушієм при зіткненні колайдерів, то він надсилає повідомлення про поразку і гра зупиняється та пропонує почати спочатку. При перезапуску гри тайли генеруються заново.

ВИСНОВКИ

У даній роботі було розроблено гру-симулятор студентського життя та продемонстровано використання патернів проектування при розробці ігрових додатків. Було досліджено найбільш поширені та популярні патерни: Component System, ECS, MVVM, Observer, Singleton та Object Pool. Для кожного із них було описано можливості, переваги та недоліки, а також типові ситуації що потребують використання патерну для вирішення проблеми. Також було продемонстровано можливості ігрового рушія Unity у різних аспектах розробки ігор.

Проект дозволяє ознайомитися із корисними для розробки ігрових додатків архітектурними рішеннями, логікою їх роботи та реалізацією із використанням рушія Unity. У проекті було реалізовано декілька популярних у ігровій індустрії патернів проектування: MVVM, Observer, Singleton та Object Pool. Крім цього було розроблено та реалізовано додаткові архітектурні рішення: сюжетну лінію, чергу подій, систему роботи із ресурсами та систему сповіщень.

У розробленій грі продемонстровано логіку реалізації різноманітних рішень для більшості аспектів розробки ігор та їхню взаємодію між собою у типових ситуаціях. Для зручного дослідження ігрової логіки було розроблено набір мініігор. Кожна із них демонструє ті чи інші архітектурні можливості, які можна використати при розробці різноманітних варіацій ігрової логіки незалежно від жанру чи особливостей.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ігрова механіка [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%86%D0%B3%D1%80%D0%BE%D0%B2%D0%B0_%D0%BC%D0%B5%D1%85%D0%B0%D0%BD%D1%96%D0%BA%D0%B0 (дата звернення 27.03.2023). - Назва з екрана.
2. Ігровий рушій [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%86%D0%B3%D1%80%D0%BE%D0%B2%D0%B8%D0%B9_%D1%80%D1%83%D1%88%D1%96%D0%B9 (дата звернення 28.03.2023). - Назва з екрана.
3. Visual Studio [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Visual_Studio (дата звернення 15.04.2023). - Назва з екрана.
4. Visual Studio Code [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Visual_Studio_Code (дата звернення 15.04.2023). - Назва з екрана.
5. Unity [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Unity_(%D1%96%D0%B3%D1%80%D0%BE%D0%B2%D0%B8%D0%B9_%D1%80%D1%83%D1%88%D1%96%D0%B9) (дата звернення 17.05.2023). - Назва з екрана.
6. Використання рушія Unity у мобільному геймдеві [Електронний ресурс] – Режим доступу до ресурсу : https://vokigames.com/ua/vikoristannya-rushiya-unity-u-mobilnomu-gejmdevi-mozhlivosti-perevagi-ta-nedoliki/ (дата звернення 17.05.2023). - Назва з екрана.
7. C# [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/C_Sharp (дата звернення 18.05.2023). - Назва з екрана.
8. .NET Framework [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/.NET_Framework (дата звернення 18.05.2023). - Назва з екрана.

9. Common Language Runtime [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Common_Language_Runtime (дата звернення 18.05.2023). - Назва з екрана.
10. Git [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Git (дата звернення 20.05.2023). - Назва з екрана.
11. Github [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/GitHub (дата звернення 20.05.2023). - Назва з екрана.
12. Matching Game [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Matching_game (дата звернення 06.05.2023). - Назва з екрана.
13. Whack A Mole [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Whac-A-Mole (дата звернення 10.05.2023). - Назва з екрана.
14. Bullet Hell [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Bullet_hell (дата звернення 12.05.2023). - Назва з екрана.
15. Патерни проектування [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns (дата звернення 05.04.2023). - Назва з екрана.
16. MVVM у Android [Електронний ресурс] – Режим доступу до ресурсу: https://dou.ua/forums/topic/33022/ (дата звернення 08.04.2023). - Назва з екрана.
17. MVVM [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Model-View-ViewModel (дата звернення 08.04.2023). - Назва з екрана.
18. Observer pattern [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns/observer (дата звернення 10.04.2023). - Назва з екрана.

19. Object pool [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Object_pool_pattern (дата звернення 12.04.2023). - Назва з екрана.
20. Singleton [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns/object (дата звернення 12.04.2023). - Назва з екрана.

ДОДАТКИ

Додаток А

Клас ActionsQueue

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

public class ActionsQueue : MonoBehaviour
{
   private List<QueueAction> actions = new List<QueueAction>();

   void Start()
   {
      StartCoroutine(MainCoroutine());
   }

   public void AddAction(QueueAction action)
   {
      actions.Add(action);
   }

   IEnumerator MainCoroutine()
   {
      yield return StartCoroutine(StartNextAction());
      yield return StartCoroutine(MainCoroutine());
   }

   IEnumerator StartNextAction()
   {
      if (actions.Count == 0) {
         yield return new WaitForEndOfFrame();
         yield break;
      }
```

```
        actions[0].StartAction();
        yield return new WaitUntil(() => { return actions[0].IsCompleted; });
        actions.RemoveAt(0);
    }
}
```

Додаток Б

Клас WaitMinigameWinAction

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

public class WaitMinigameWinAction : QuestAction
{
    private Consts.MinigameNames minigameName;

    public WaitMinigameWinAction(List<StringStringPair> actionData) : base(actionData)
    {
        EventsSystem.OnWinMinigame += OnWinMinigame;
    }

    private void OnWinMinigame(EventData eventData)
    {
        if (eventData != null && eventData.Data.ContainsKey("MinigameName"))
        {
            var inputName =
(Consts.MinigameNames)System.Enum.Parse(typeof(Consts.MinigameNames),
eventData.Data["MinigameName"].ToString());
            if (minigameName == inputName)
            {
```

```
            Complete();
        }
    }
}

public override QuestActionType GetActionType()
{
    return QuestActionType.WaitScene;
}

protected override void SetupData()
{
    int nameIndex = ActionData.FindIndex(0, (StringStringPair pair) => { return pair.Key ==
"MinigameNames"; });
    if (nameIndex >= 0)
    {
        object name;
        if (System.Enum.TryParse(typeof(Consts.MinigameNames), ActionData[nameIndex].Value,
out name))
        {
            minigameName = (Consts.MinigameNames)name;
        }
    }
}
}
```