МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Центр заочної, дистанційної та вечірньої форм навчання

Кафедра комп'ютерних наук

«До захисту допущено»

В.о. завідувача кафедри

_____________ Ігор ШЕЛЕХОВ

(підпис)

11 січня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 - Комп'ютерних наук,

освітньо-професійної програми «Інформатика»

на тему: «Інформаційна технологія оцінки та адаптації ігрового процесу комп'ютерної ігрової системи»

здобувачки групи ІН.мз-22с Ципліної Ірини Сергіївни

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

_____________ Ірина ЦИПЛІНА

(підпис)

Керівник,
в.о. завідувача кафедри,
кандидат технічних наук, доцент Ігор ШЕЛЕХОВ _____________

(підпис)

Суми – 2024

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

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

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми «Інформатика»
здобувачки групи ІН.мз-22с Ципліної Ірини Сергіївни

1. Тема роботи: «Інформаційна технологія оцінки та адаптації ігрового процесу комп'ютерної ігрової системи»
затверджую наказом по СумДУ від _*«01» грудня 2023 року № 0475-VI*_
2. Термін здачі здобувачем кваліфікаційної роботи _*до 12 січня 2024 року*_
3. Вхідні дані до кваліфікаційної роботи ______________________
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
*1) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Проєктування ігрової системи. 3) Розробка інформаційного та програмного забезпечення. 4) Аналіз результатів.*
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
____
6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

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

КАЛЕНДАРНИЙ ПЛАН

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

Здобувач вищої освіти _________________ Керівник _________________
(підпис) (підпис)

АНОТАЦІЯ

Записка: 44 стр., 9 рис., 1 додаток, 28 використаних джерел.

Обґрунтування актуальності теми роботи – За останні роки ігрова індустрія значно зросла за обсягом та популярністю. Графіка, механіка та геймплей стають все складнішими, а гравці стають більш вимогливими. Оцінка та адаптація ігрового процесу може відігравати ключову роль у створенні захопливих та персоналізованих ігрових досвідів. Сучасні гравці шукають індивідуалізовані враження. Модуль оцінки та адаптації може забезпечити персоналізацію гри, враховуючи вміння, стиль гри та інші характеристики гравця. Загалом, створення інформаційної технології для оцінки та адаптації ігрового процесу має великий потенціал для поліпшення ігрового досвіду, збільшення залученості гравців та створення конкурентних переваг для розробників ігор.

Об'єкт дослідження — Процес оцінки та адаптації ігрового процесу, а саме дослідження методів збору даних про дії гравця в грі, аналіз відомостей про гравців для визначення їхніх вмінь, стратегій та стилів гри, вивчення методів динамічного налаштування рівня складності гри в реальному часі, аналіз персоналізованих завдань та викликів для гравців, дослідження ефективності систем нагородження та їх впливу на мотивацію гравців.

Мета роботи — Розробка інноваційних підходів до оцінки та адаптації ігрового процесу з метою створення більш захопливого та персоналізованого ігрового досвіду для користувачів.

Методи дослідження — Вивчення та аналіз існуючих ігор, де вже використовуються елементи оцінки та адаптації, для визначення плюсів і мінусів існуючих підходів. Використання програмних засобів для створення прототипів модулів оцінки та адаптації для ігрової системи.

Результати — Успішна розробка та імплементація модулів оцінки та адаптації ігрового процесу в ігрову систему. Забезпечення персоналізованого

геймплею для гравців, яка враховує їхні вміння, стиль гри та індивідуальні особливості.

ІНФОРМАЦІЙНА ІГРОВА СИСТЕМА, ІГРОВИЙ ПРОТОТИП, UNITY3D

ЗМІСТ

ВСТУП 7
1 АНАЛІТИЧНИЙ ОГЛЯД І ПОСТАНОВКА ЗАДАЧІ 9
1.1 Основні визначення 9
1.2 Застосування штучного інтелекту в геймінгу 10
1.3 Визначення агента 11
1.4 Рух 11
1.5 Постановка задачі 12
2 ТЕХОЛОГІЯ ОЦІНКИ ТА АДАПТАЦІЇ ІГРОВОГО ПРОЦЕСУ 14
2.1 Прийняття рішень 14
2.1.1 Кінцеві автомати 14
2.1.2 Ієрархічні скінченні автомати 15
2.1.3 Дерева поведінки 17
2.1.4 Типи завдань 17
2.1.5 Оцінка дерева поведінки 18
2.2 Пошук шляху 20
2.2.1 Ієрархічний пошук шляху 21
2.2.2 Кластеризація 21
3 ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ 23
3.1 Опис програмного забезпечення 23
3.2 Пошук шляху 23
3.2.1 Алгоритм А* 24
3.2.2 Модифікація А* для гри 25
3.2.3 Еврістика 26
3.2.4 Побудова шляху 28
3.2.5 Згладжування шляху 28
3.2.6 Навігаційна сітка 28
3.2.7 Сітка 31

3.3 Прийняття рішень 32
3.3.1 Скінчені автомати 32
3.4 Тестування 36
ВИСНОВКИ 38
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 39
ДОДАТОК 42

ВСТУП

Обґрунтування актуальності теми роботи – За останні роки ігрова індустрія значно зросла за обсягом та популярністю. Графіка, механіка та геймплей стають все складнішими, а гравці стають більш вимогливими. Оцінка та адаптація ігрового процесу може відігравати ключову роль у створенні захопливих та персоналізованих ігрових досвідів. Сучасні гравці шукають індивідуалізовані враження. Модуль оцінки та адаптації може забезпечити персоналізацію гри, враховуючи вміння, стиль гри та інші характеристики гравця. Загалом, створення інформаційної технології для оцінки та адаптації ігрового процесу має великий потенціал для поліпшення ігрового досвіду, збільшення залученості гравців та створення конкурентних переваг для розробників ігор.

Об'єкт дослідження — Процес оцінки та адаптації ігрового процесу, а саме дослідження методів збору даних про дії гравця в грі, аналіз відомостей про гравців для визначення їхніх вмінь, стратегій та стилів гри, вивчення методів динамічного налаштування рівня складності гри в реальному часі, аналіз персоналізованих завдань та викликів для гравців, дослідження ефективності систем нагородження та їх впливу на мотивацію гравців.

Мета роботи — Розробка інноваційних підходів до оцінки та адаптації ігрового процесу з метою створення більш захопливого та персоналізованого ігрового досвіду для користувачів.

Методи дослідження — Вивчення та аналіз існуючих ігор, де вже використовуються елементи оцінки та адаптації, для визначення плюсів і мінусів існуючих підходів. Використання програмних засобів для створення прототипів модулів оцінки та адаптації для ігрової системи.

Гіпотеза. Впровадження модуля оцінки та адаптації ігрового процесу в ігрову систему призведе до покращення ефективності гравців та забезпечення персоналізованого геймплею, що призведе до підвищення загального

задоволення гравців від ігрового досвіду.

Новизна. В роботі запропоновано механізми, які враховують індивідуальні вміння та стиль гри кожного гравця, щоб забезпечити унікальний ігровий досвід, а також алгоритми для динамічного налаштування рівня складності гри в залежності від дій гравця під час гри.

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

1 АНАЛІТИЧНИЙ ОГЛЯД І ПОСТАНОВКА ЗАДАЧІ

1.1 Основні визначення

Розглянемо деякі основні визначення, пов'язані з цією областю.

Ігрова система (Game System) - це комплексне програмне та апаратне забезпечення, що створює ігрове середовище. Вона включає в себе елементи гри, такі як графіка, звук, механіка гри, та інші аспекти, які співпрацюють для створення ігрового досвіду[1].

Модуль оцінки (Assessment Module) - це частина ігрової системи, яка відповідає за аналіз та оцінку дій гравців. Вона може враховувати різні параметри, такі як вміння гравця, часові показники, стратегії гри та інші, щоб забезпечити об'єктивну оцінку гравця[1].

Адаптація ігрового процесу - це здатність ігрової системи належним чином змінювати параметри гри в реальному часі для відповіді на дії та вміння гравців. Це може включати в себе зміни рівня складності, генерацію завдань, або інші аспекти геймплею.[1-5]

Персоналізація геймплею - це властивість ігрової системи, яка дозволяє адаптувати гру для конкретного гравця, враховуючи його індивідуальні вподобання, стиль гри та рівень вмінь.[1-5]

Застосування штучного інтелекту в геймінгу передбачає використання алгоритмів та моделей для створення "розумних" ворогів, автоматизації адаптації геймплею, та інших інтелектуальних аспектів.[6,7]

Системи нагородження включають в себе механізми, які надають гравцям винагороду за досягнення певних цілей або успіхів в грі. Це може бути в формі віртуальних нагород, розблокованих рівнів, чи інших стимулів.

Ці терміни відображають основні аспекти та поняття, які можуть бути важливими у дослідженні з інформаційної технології проєктування ігрової системи та модулів оцінки та адаптації ігрового процесу.

1.2 Застосування штучного інтелекту в геймінгу

На сьогоднішній день штучний інтелект є активно досліджуваною областю в галузі інформаційної технології геймінгу. ШІ може бути використаний дуже різними способами і також з дуже різними цілями на увазі; отже, область ШІ є обширною не тільки загалом, але і в розробці відеоігор, яка вже є досить вузькою галуззю. ШІ може бути використаний не тільки для самої гри, але і під час процесу розробки.[6]

Можливо, найочевиднішим використанням ШІ в іграх є покращення поведінки не-гравецьких персонажів (НПС) для зроблення їх реактивними, адаптивними та інтелектуальними. Гравцям важливо бачити їхню поведінку як значущу, щоб вони вірити, що НПС є розумними. Отже, тут виникає термін "штучний інтелект".

Також існують інші способи використання ШІ в іграх. Він може використовуватися для динамічного контролю рівня складності гри; вивчати поведінку гравця під час гри та адаптуватися до неї; генерувати вміст гри, такий як карти, рівні, текстури, правила гри, сюжети, квести, музика, предмети, зброя, транспортні засоби або навіть персонажі та їх візуальні виведення, такі як анімації.[7]

Слід відзначити, що термін "штучний інтелект" в ігровій індустрії зазвичай не представляє собою "справжнього ШІ", оскільки ці техніки зазвичай лише автоматизовані обчислення на фіксованому та обмеженому наборі вхідних даних, які обирають відповідь із фіксованого та обмеженого набору відповідей. Насправді воно лише створює ілюзію "справжнього ШІ" для конкретного середовища.

Рух, прийняття рішень та стратегія - це три основи гри в моделі ШІ, яку використовується в "Штучний інтелект для ігор" Міллінгтоном та Фанге, яка ілюструється на Рисунку 1.1 і яку ми також використовуємо. Навколо них є додаткова інфраструктура, яка нас особливо не стосується, так само як і

стратегія. Стратегія працює на основі кількох персонажів, команди або цілого боку, на відміну від руху та прийняття рішень, які діють на рівні окремого персонажу і про це ми розповімо пізніше.

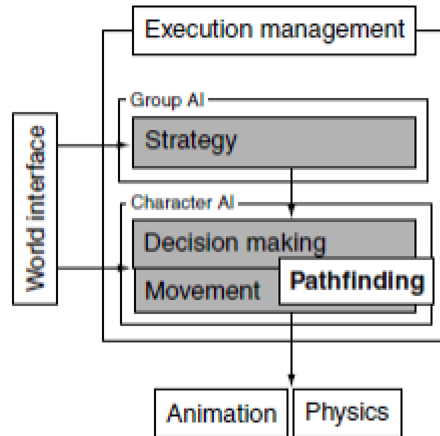


Рисунок 1.1: Модель ШІ

1.3 Визначення агента

Ми часто говоримо про персонажів у іграх, але НПС не обов'язково має бути персонажем як таким. Існує інше значення абревіатури НПС, а саме - некерований клас (NPC - non-player class). НПС також може бути транспортним засобом, твариною, створінням чи чимось іншим. Тому ми визначаємо і використовуємо термін "агент".[6-10]

Агент - це штучно інтелектуальна сутність, яка проявляє виткані поведінкові патерни, що не є випадковими і можуть розглядатися як розумні.

1.4 Рух

Існує лише декілька ігор, в яких агенти не рухаються, що означає, що більшість ігор використовують рух якимось чином. Це може бути від дуже простого руху, схожого, якщо не ідентичного, на деякі класичні ігри, такі як Pac-Man або Mario Bros., до складного, як у багатьох сучасних іграх, наприклад, у "Відьмак 3".

Найпростіший тип руху, схожий на той, що в Pac-Man, - це кінематичний рух. Це означає, що агент не враховує прискорення та сповільнення. У

старіших іграх агенти зазвичай мають лише дві швидкості: нерухомість та максимальна швидкість, часто біг, іноді з третьою швидкістю для ходьби (або повільного руху) серед них. Алгоритми кінематичного руху, як правило, потребують знати лише поточне положення агента, а вивід алгоритму - це напрямок для руху.

Коли агент отримує напрямок для руху до своєї цілі, він одразу починає рухатися з цільовою швидкістю, і по досягненню місця призначення відразу зупиняється. Просто, але не дуже реалістично.

Оскільки кінематичний рух не виглядає життєво із часом з'явилася потреба в більш реалістичному русі, розробники почали впроваджувати динамічний рух, який також враховує поточний рух агента. Алгоритм для динамічного руху повинен знати поточні швидкості агента, крім його поточного положення. Він генерує сили, які змінюють швидкість агента.

Припустимо, порівняємо кінематичний і динамічний алгоритми для агента, який переміщується з одного місця на інше. У такому випадку кінематичний алгоритм дає агенту напрямок для руху, і коли агент досягає своєї цілі, він просто зупиняється, оскільки він більше не отримує напрямку для руху. З іншого боку, динамічний алгоритм спочатку повинен повернути агента в потрібному напрямку, а потім прискорити його до цільової швидкості. Перед тим, як агент досягне місця призначення, алгоритм повинен поступово сповільнювати його, зменшувати швидкість до зупинки точно на місці призначення.

1.5 Постановка задачі

Метою кваліфікаційної роботи є розв’язання практичної задачі з розробки комплексу інформаційного та програмного забезпечення здатної до адаптації до дій гравця ігрової системи. Для досягнення поставленої мети необхідно виконати такі основні завдання роботи:

1. Аналіз Існуючих Робіт:

Провести літературний огляд існуючих підходів до інтеграції модулів оцінки та адаптації в ігрові системи. Визначити ключові аспекти та відмінності підходів, що використовуються в галузі.

2. Проєктування Модуля Оцінки:

Розробити алгоритми та методи оцінки гравців, враховуючи різні аспекти, такі як вміння, стратегії гри та часові показники. Вивчити можливості використання технік машинного навчання для поліпшення точності оцінки.

3. Реалізація Модуля Адаптації:

Розробити алгоритми динамічної адаптації геймплею в реальному часі, використовуючи зібрані дані оцінки гравців. Забезпечити можливість персоналізації геймплею згідно з результатами оцінки.

4. Інтеграція в Ігрову Систему:

Розробити інтеграцію розробленого модуля в існуючу ігрову систему або створити прототип для тестування.

5. Аналіз Результатів та Висновки:

Провести аналіз отриманих результатів експериментів.

Зробити висновки щодо ефективності та можливостей подальшого вдосконалення.

2 ТЕХОЛОГІЯ ОЦІНКИ ТА АДАПТАЦІЇ ІГРОВОГО ПРОЦЕСУ

2.1 Прийняття рішень

Те, що люди загалом уявляють, коли думають про ШІ в відеоіграх, маючи лише базові знання на тему, - це сутність, яку приводить в дію комп'ютер та приймає рішення щодо того, що їй слід робити. Вони не розуміють, що виконання цих рішень також є частиною ШІ.[7]

Існують кілька основних підходів до систем прийняття рішень. Кожен має свої плюси та мінуси, і часто використовується більше ніж один.

2.1.1 Кінцеві автомати

Кінцеві автомати (КА), разом із їхніми варіаціями, такими як ієрархічні кінцеві автомати (ІКА), були домінуючими в системах прийняття рішень ШІ в відеоіграх до середини 2000-х років. КА - це графи, що складаються зі станів та переходів. Також існують дії, які пов'язані із конкретними станами.[8-10]

Агент починає в певному стані. У кожному стані агент виконує набір дій. Коли виконуються певні умови, агент переходить з поточного стану в новий, де він виконує інший набір дій. Наприклад, охоронець може мати стани "на оберіг", "бій" і "втікання". Агент починає в стані "на оберіг", можливо, патрулюючи або просто стежачи за місцем. Це залежить від дій, пов'язаних із станом. Коли він бачить маленького ворога, він переходить в стан "бій" і воює з малим ворогом. Якщо він програє бій, він переходить в стан "втікання" і намагається втекти. Якщо йому вдається втекти, він повертається до стану "на оберіг". Існує ще один перехід з цього стану. Коли агент бачить великого ворога, він не намагається воювати і втікає негайно, що означає, що він переходить в стан "втікання". Цей приклад ілюструється на Рисунку 2.2.

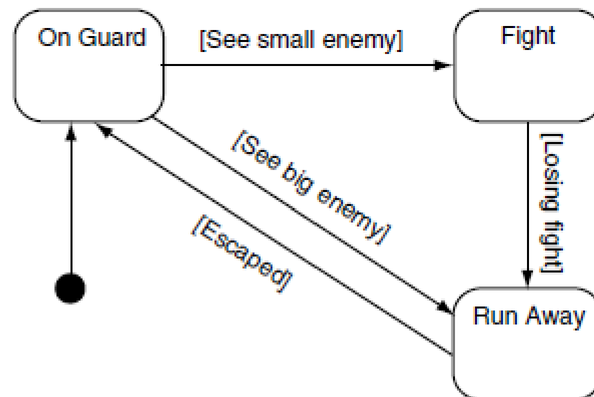


Рисунок 2.1 - Кінцевий автомат

Ми можемо зрозуміти, що цей КА не є ідеальним. Є лише один перехід зі стану боротьби, отже, що відбудеться, якщо агент вбиває маленького ворога? Це залежить від реалізації, але, ймовірно, він залишиться на місці і нічого не робитиме до тих пір, поки не помре або гра не закінчиться. Також є друга проблема. Якщо агент бачить великого ворога, перебуваючи в стані боротьби, нічого не відбудеться до того моменту, поки він не почне втрачати боротьбу, хоча ймовірно, йому слід спробувати втекти негайно.[9]

2.1.2 Ієрархічні скінченні автомати

Скінченні автомати можуть бути дуже складними, якщо існує багато станів або якщо у нас є поведінка, яка відбувається незалежно від поточного стану. Розгляньте поведінку робота-прибиральника як приклад. У початковому стані "пошук" робот шукає сміття. Якщо він бачить сміття, він переходить у стан "напрямок до сміття" і рухається, щоб зібрати його. Після збору сміття він переходить у стан "напрямок до компактора" і рухається до компактора, щоб викинути сміття. Після викидання сміття він повертається до початкового стану "пошук".[11-15]

Тепер уявіть, що робот працює на батареях і повинен заряджатися на станції зарядки, коли заряд батареї низький. Якщо б ми реалізували це за

допомогою традиційного скінченного автомата, це виглядало б як на рисунку 2.2, що є досить складним.

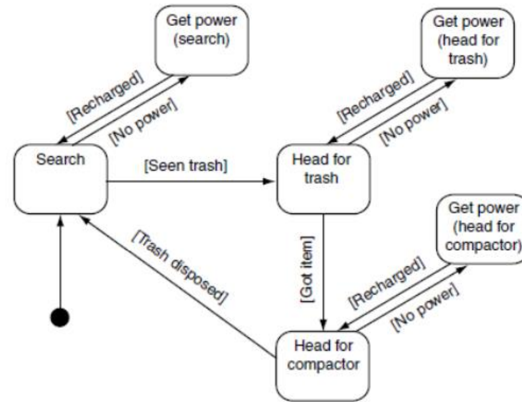


Рисунок 2.2 - Кінцевий автомат для прибирального робота

Однак існують ієрархічні скінченні автомати [12,13], які вирішують цю конкретну проблему. Вони створюють ієрархію скінченних автоматів, які можуть бути з'єднані між собою. Ми навіть можемо встановлювати переходи між ІСА та станами звичайного скінченного автомата або внутрішнім станом ІСА та станом іншого скінченного автомата або ІСА. Приклад робота-прибиральника у вигляді ІСА ілюстровано на рисунку 2.3. Перехід між станом ІСА та іншим скінченим автоматом був доданий, щоб показати додатковий функціонал: якщо робот не знаходить сміття, він переходить до заряджання своєї батареї.

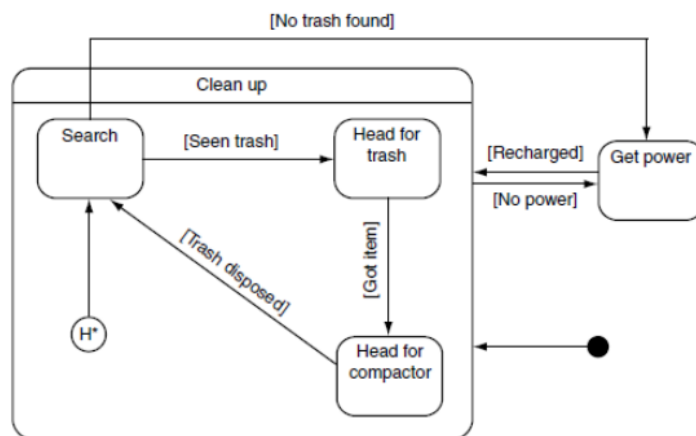


Рисунок 2.3 - Ієрархічний кінцевий автомат для прибирального робота

2.1.3 Дерева поведінки

Дерева поведінки (BT) є важливим інструментом для проектування штучного інтелекту. На відміну від скінчених автоматів (FSM), які орієнтовані на стан, дерева поведінки орієнтовані на завдання. Вони дозволяють дизайнерам створювати дуже складні завдання, об'єднуючи кілька простих завдань. Однією з їх найважливіших переваг є легкість відділення дизайну від програмування, що дозволяє легко проектувати дерево поведінки за допомогою проміжного програмного забезпечення, не потребуючи жодних знань з програмування від дизайнерів штучного інтелекту.[14]

2.1.4 Типи завдань

Дерева поведінки, як із самої назви, зображують у вигляді напрямлених ациклічних дерев. В завджи існує один кореневий вузол у дереві, який не має батьківського вузла, та принаймні один дочірній вузол. Кожен листовий вузол має один батьківський вузол та нульову кількість дочірніх вузлів. Кожен вузол всередині дерева має один батьківський вузол та один або більше дочірніх вузлів. Ми розрізняємо чотири основні типи завдань. Є композити та декоратори, які завжди знаходяться всередині дерева, і умови та дії, які займають листові вузли.

Як пишуть [15], умови тестують певну властивість гри. Може бути різноманітні тести; наприклад, можна перевірити, чи відчинена дверь, або перевірити, чи у нашого персонажа є ключ. Кожен тест повинен бути реалізований як окреме завдання.

Дії, з іншого боку, змінюють стан гри. Прикладом дії може бути відчинення двері, рух або відтворення анімації. Знову ж таки, кожну дію слід реалізувати окремо.

Дії та умови схожі за функціоналом на ті в FSM. Головна відмінність полягає в тому, що в деревах поведінки для всіх них існує один загальний

інтерфейс, що означає, що їх можна використовувати в дереві поведінки без необхідності знати, що ще присутнє в дереві.

Композитні вузли можуть мати кілька дочірніх вузлів, завдань, за якими вони ведуть облік. Є кілька типів композитних вузлів, найважливішими з яких є селектори та послідовності.

Селектор виконує свої дочірні вузли в зазначеному порядку, зазвичай зліва направо або зверху донизу в дереві, до тих пір, поки один з них не успіє. Коли його дочірній вузол успіхає, селектор завершується та повертає успіх. Якщо всі дочірні вузли невдаються, селектор також зазнає невдачі.[17]

Послідовність працює дуже схоже на селектор. Критична відмінність полягає в тому, що в послідовності всі дочірні вузли повинні успіти, щоб послідовність успіла. Якщо хоча б один з дочірніх вузлів невдається, послідовність також зазнає невдачі.

Декоратори мають одного дочірнього вузла і змінюють його поведінку якимось чином.

На відміну від композитів, які мають лише кілька типів, є багато типів декораторів, так само, як і умов і дій. Декоратори можуть обмежувати кількість разів, коли виконується завдання, тримати його виконування завжди, інвертувати його значення повернення, перевіряти, чи є доступні необхідні ресурси для виконання завдання і так далі. Один з типів декораторів - це фільтр.

Фільтр вирішує, чи може дочірнє завдання виконуватися. Якщо він дозволяє дочірньому виконати завдання, він повертає те, що повернуло дочірнє завдання. У випадку, якщо він не дозволяє дочірньому виконувати завдання, він повертає невдачу.

2.1.5 Оцінка дерева поведінки

На рисунку 2.3 ми маємо приклад дерева поведінки (BT), і ми використовуватимемо його, щоб показати, як відбувається оцінка BT. Символ

стрілки вузла символізує послідовність, а вузол з питанням - селектор. Інші вузли - це дії, за винятком вузла "Двері відчинено?", який є умовою.

Під час оцінки BT його завдання виконуються в зазначеному порядку, у нашому випадку - зверху вниз і зліва направо. Кожне завдання після оцінки повертає свій статус. Зазвичай використовуються такі статуси, як успіх, невдача та виконання. Інші статуси можуть використовуватися, наприклад, для допомоги при відлагодженні або сигналізації певної поведінки.

Коли завдання вдається, воно повертає успіх, і якщо воно зазнає невдачі, воно повертає невдачу. Якщо завдання ще не завершило свою оцінку, воно повертає виконання, і коли BT запускається наступного разу, воно починає оцінку з цього вузла. Таким чином, глядя на наш приклад, коли персонаж рухається до дверей, BT не повторює питання, чи відчинено двері, але чекає, поки дія "Рухатися (до дверей)" поверне або успіх, або невдачу. Після цього воно продовжує оцінку.

Детально розглядаючи приклад BT, ми бачимо два способи його виконання. Обидва залежать від статусу дверей у питанні. Якщо двері відчинені, умова вдається, що призводить до успіху селектора, і коренева послідовність продовжується дією "Рухатися (у кімнату)". Якщо навпаки двері зачинені, умова не вдається, і ми продовжуємо оцінку другої (нижньої в дереві) послідовності. Спочатку агент рухається до дверей, а потім відчиняє їх. Послідовність вдається, селектор вдається, і агент рухається у кімнату. Після цього коренева послідовність вдається, що означає, що вся BT вдається. Якщо на будь-якому етапі будь-яка дія зазнає невдачі, вся BT також невдається.

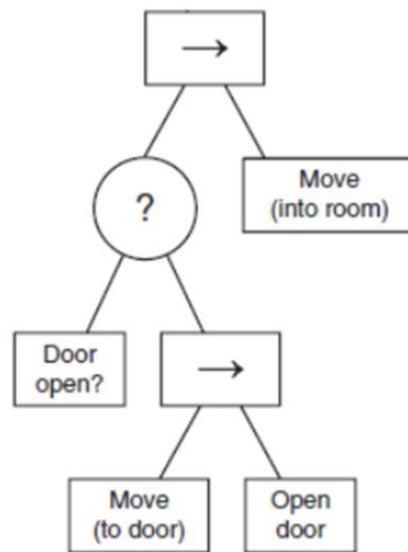


Рисунок 2.3 - Дерево поведінки

2.2 Пошук шляху

Пошук шляху розташований між рухом та прийняттям рішень у нашій моделі. Це тому, що пошук шляху може служити як для переміщення агентів, так і для планування або прийняття рішень.[19-21]

В основному алгоритми пошуку шляху працюють однаково. Їм потрібні два вхідні параметри - початок та кінцевий пункт маршруту, для обчислення маршруту через мапу, а також потрібна карта гри в підходящій структурі даних, яка є направленим невід'ємним зваженим графом для улюблених алгоритмів. Ця частина алгоритмів пошуку шляху вписується під рух. Однак, якщо алгоритм пошуку шляху не лише обчислює шляхи, але й вирішує, куди рухатися далі, він також підходить під прийняття рішень, тому він знаходиться між рухом та прийняттям рішень у моделі штучного інтелекту.

Алгоритмом вибору для пошуку шляху є A*, який використовується у більшості випадків. Алгоритм A* насправді є класом алгоритмів, які використовують різні евристики та правила розгалуження. Він базується на алгоритмі Дейкстри, поверх якого він використовує евристику для оцінки відстані до пункту призначення.[22]

В той час як алгоритм Дейкстри вибирає наступний вузол з найменшою відстанню від початку, A* вибирає вузол з найменшою евристичною відстанню, яка є найменшою відстанню від початку плюс оцінена відстань до пункту призначення. Алгоритм A* був доведений оптимальним [34] (* позначає оптимальність) для допустимих евристик. Евристика є допустимою, якщо вона ніколи не переоцінює відстань до пункту призначення для кожного вузла. Одна з часто використовуваних евристик - це євклідова відстань, яка є допустимою евристикою.

2.2.1 Ієрархічний пошук шляху

Як ми вже зазначили, алгоритм A* є оптимальним. Однак він може бути дуже витратним з обчислювальної точки зору для великих карт і може суттєво негативно впливати на продуктивність, наприклад, у стратегіях реального часу. Існує ієрархічний пошук шляху, який призначений для вирішення цієї проблеми.[23]

2.2.2 Кластеризація

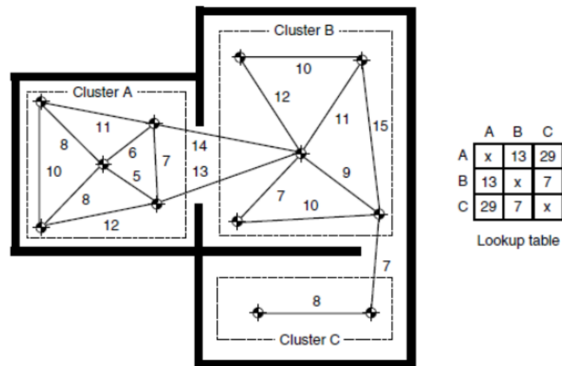
У пошуку шляху процес кластеризації полягає в тому, що ми ділимо вузли пошуку шляху на групи, які називаються кластерами. Як пишуть Міллінгтон і Фанге , "вузли в кластері представляють деякий регіон рівня, який має високий рівень взаємодії". У дизайні гри рівня цілком типово мати один кластер на одну кімнату. Кластеризація може бути як ручною, так і автоматичною. Після того, як у нас є кластери, нам потрібна таблиця пошуку з вартостями шляхів між окремими кластерами. Таблиця пошуку складається з найменших можливих довжин шляху між парами кластерів.

Кластерна евристика дає нам довжини шляхів між кластерами. Якщо як початковий вузол, так і вузол цілі знаходяться в одному кластері, ми використовуємо реальні вартості шляху для отримання шляху.[24]

Складні області із меншими кластерами, наприклад, багатоквартирний будинок, добре підходять для кластерної евристики, оскільки вона зазвичай

драматично покращує продуктивність порівняно з простими евристиками, такими як евклідова відстань. Однак таблиця пошуку може бути величезною, вимагаючи як пам'яті, так і часу для попереднього обчислення.[25-27]

Кластерна евристика ілюстрована на рисунку 2.4.



	A	B	C
A	x	13	29
B	13	x	7
C	29	7	x

Рисунок 2.4: Евристичний метод кластеризації

3 ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Опис програмного забезпечення

Основою нашого проекту був гральний рушій Unity. Ми використовували його версію 2019 LTS, яку оновлювали протягом розробки проекту до версії 2019.4.26f1 LTS. Мовою програмування для розробки в Unity є C#, хоча сам рушій працює на C++. Unity надає багатий набір інструментів та середовище не лише для розробки ігор, а й включає магазин активів з постійно зростаючою бібліотекою активів, розроблених в основному спільнотою. Ми використовували два зовнішні активи. Перший - це Bolt, візуальний інструмент для скриптів, який ми використовували для візуалізації МКС (Машини Кінцевих Станів). Другий - це Bonsai Behaviour Tree, фреймворк для дерев поведінки з графічним редактором, який ми використовували для реалізації та візуалізації БД (Блокових Діаграм). Для редагування коду ми використовували Microsoft Visual Studio.

3.2 Пошук шляху

Ми розпочинаємо з реалізації алгоритму A*, потім переходимо до представлення світу навміш та завершуємо представленням світу у вигляді сітки.

Алгоритм пошуку шляху, який ми реалізували для нашої роботи, - це A*. Ми також реалізували модифіковану версію, яку використовували для пошуку шляху в нашій грі, щоб заборонити привидам повертати назад через обчислений шлях.

Алгоритм A* - лише одна частина алгоритму обчислення шляху, хоча й найважливіша. Інші частини - це евристичні алгоритми, які використовуються A*, алгоритм, який конструює шлях з даних, обчислених A*, і алгоритм згладжування шляху.

Робочий процес виглядає наступним чином. Спочатку ми викликаємо A*, щоб розрахувати шлях на нашому графі від вузла старту до вузла цілі з

вибраною евристикою. Потім ми відновлюємо шлях від вузла цілі до вузла старту, і, нарешті, ми викликаємо алгоритм згладжування шляху для побудованого шляху. Весь цей процес виконується методом FindPath у нашій реалізації, який ми описуємо далі.

Ми використовуємо цей процес пошуку шляху в усіх частинах нашої реалізації, або, іншими словами, у всіх реалізованих сценах.

3.2.1 Алгоритм A*

Тут ми описуємо нашу реалізацію алгоритму A*, який знаходиться у скрипті AStarSearch. Ми також реалізуємо евристичні функції у цьому скрипті.

Вхідними параметрами для алгоритму є навігаційний граф, який ми шукаємо, початковий вузол та вузол цілі пошуку, вибрана евристика та вибраний тип руху (для сіток). Окрім параметрів методу, наша реалізація також працює з двома словниками (cameFrom та costSoFar), множиною (closedNodes) та пріоритетною чергою (openNodes). Ми використовуємо ці об'єкти поза цим методом, тому ми повинні очищати їх на початку нового пошуку. Нижче наведено опис алгоритму A*, представленого у лістингу 2.1.

Ми розпочинаємо, вставляючи початковий вузол у нашу пріоритетну чергу та встановлюючи його значення costSoFar на нуль. Потім починається основний цикл пошуку методу і працює, поки в черзі пріоритетів є вузли. Під час циклу спочатку ми видаляємо вузол з черги пріоритетів, називаємо його поточним вузлом, додаємо його до множини закритих вузлів і перевіряємо, чи є поточний вузол вузлом цілі, в такому випадку ми завершуємо роботу методу. Це відбувається на рядках 11-14 у коді. Потім ми переходимо до всіх сусідніх вузлів поточного вузла у циклі foreach на рядку 16.

У кожній ітерації циклу foreach ми обчислюємо новий витрати від початкового вузла до сусіднього вузла на рядку 17. Ми називаємо сусідній вузол "next" у коді. Якщо нові витрати менше, ніж поточні витрати для сусіднього вузла (costSoFar), ми перевіряємо, чи сусідній вузол є в черзі

пріоритетів або в множині закритих вузлів і видаляємо його. Потім, на рядках 27-32, якщо сусідній вузол не знаходиться в нашій черзі пріоритетів або множині закритих вузлів, ми встановлюємо йому значення costSoFar на нові витрати і значення cameFrom на поточний вузол. Ми також обчислюємо пріоритет вузла за допомогою нових витрат і евристичних витрат, і додаємо сусідній вузол в нашу чергу пріоритетів з обчисленим пріоритетом. Значення евристичних витрат ми отримуємо з обчислення, яке ми описуємо пізніше.

В кінці або при першому виході ми повертаємо об'єкт, який містить множину закритих вузлів і чергу пріоритетів відкритих вузлів.

3.2.2 Модифікація A* для гри

У грі Pac-Man привиди ніколи не повертаються, рухаючись, крім випадків, коли це спричинене. Через це нам довелося змінити алгоритм A* для досягнення цього поведінки. Тут ми описуємо модифікації та їх місце в алгоритмі з Лістингу 2.1.

Ми додали новий параметр до методу, який повідомляє нам напрямок агента, і новий словник stepsSoFar для зберігання кількості попередніх вузлів на поточному найкоротшому шляху.

Стосовно нового словника, ми очищуємо його на початку, і перед циклом while ми вставляємо значення нуля для початкового вузла. Після рядка 30 в оригінальному алгоритмі ми вставляємо рядок з Лістингу 2.2.

Ще одна модифікація відбувається на рядку 13, де ми змінюємо умову так, щоб вона була такою ж, як у Лістингу 2.3. Раніше, якщо початковий вузол і цільовий вузол були однаковими, алгоритм завершувався негайно. Ця модифікація забороняє таку поведінку.

Остання модифікація алгоритму - це фрагмент коду у Лістингу 2.4, вставлений після рядка 18 у вихідному алгоритмі. Перша умова призводить до того, що алгоритм пропускає сусідній вузол, якщо це перший крок і сусідній вузол знаходиться за агентом. На наступних кроках всі вузли за агентом

завжди знаходяться в множині закритих вузлів, і їх costSoFar завжди менше нових витрат, тому вузол cameFrom не може бути оновлений. Саме тому ми використовуємо другу умову, яка є нашим раннім виходом. Якщо сусідній вузол є цільовим вузлом, і ми зробили принаймні два кроки, ми знову досягли цільового вузла. Ми змінюємо значення cameFrom сусіднього вузла на поточний вузол, додаємо поточний вузол до множини закритих вузлів і повертаємося.

3.2.3 Еврістика

Ми реалізували п'ять евристик, які ми коротко представили в попередньому розділі, і тут ми пишемо про їх реалізацію. Ми базуємо реалізацію чотирьох відомих евристик на псевдокоді від Патела.

Евристика відстані Мангеттену є стандартною евристикою для квадратних сіток з чотиризначним рухом, і наша реалізація показана в Лістингу 2.5. Константа D, як правило, встановлюється як найкоротший можливий шлях між двома вузлами.

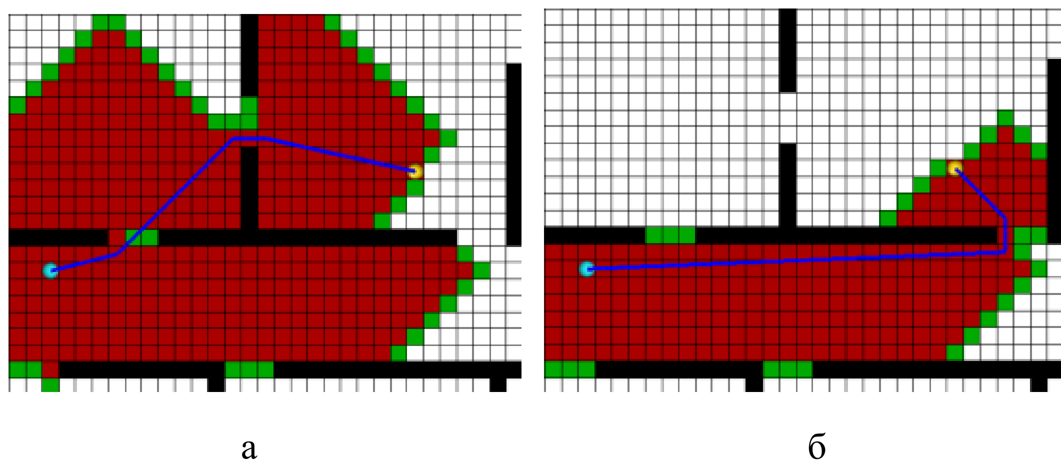
У випадку, якщо маємо сітку з восьмивекторним рухом, нам потрібна евристика діагональної відстані, яка розраховується з діагональним рухом.

Тут D є таким самим, як у відстані Мангеттену, і D2 - найкоротша діагональна відстань між вузлами. Дивіться Лістинг 2.6.

Евклідова евристика використовує відстань прямої лінії, що робить її ідеальною для використання, коли ми можемо рухатися у всіх напрямках. Тут ми використовуємо D = 1, оскільки наша функція вартості в navmeshes розраховується з реальними відстанями, так само як і ця евристика. Дивіться Лістинг 2.7.

Нульова евристика завжди повертає 0, і ми включили її для порівняння. При використанні з A*, всі вузли мають однаковий пріоритет під час обходу. Таким чином, алгоритм веде себе як алгоритм пошуку в ширину.

Ми реалізували кластерну евристику, яка сильно залежить від значення множника кластеризації, яке може налаштовуватися користувачем і змінює вагу вартості евристики у процесі пошуку шляху в порівнянні з реальною вартістю шляху до цього моменту. Під час тестування ми виявили, що розумний інтервал для множника знаходиться між одиницею і п'ятдесятьма, що приблизно відповідає довжині найбільшого можливого шляху в будь-якому з наших кластерів.



а б

Рисунок 3.1 - Евристичний метод кластеризації, (а) множник встановлено на 20; вартість шляху - 20, 456 відвіданих вузлів, (б) множник встановлено на 50; вартість шляху - 31, 249 відвіданих вузлів.

Коли множник дорівнює одиниці, евристичний метод кластеризації в основному не враховується, і результат подібний до використання нульової евристики. Коли множник дорівнює п'ятдесяти, відбувається навпаки, і вартість евристичного методу кластеризації стає вирішальною.

Наш метод кластеризації є базовим, і при множнику, рівному п'ятдесяти, він не завжди надає найкоротших шляхів. Однак якщо ми встановимо множник рівним двадцять, ми отримаємо найкоротші шляхи за рахунок збільшення кількості відвідуваних вузлів. Рисунки 3.1а та 3.1б ілюструють це порівняння.

Як вже було сказано в попередньому розділі, ми завантажуємо вартості шляхів між кластерами з файлу. Ми зберігаємо ці значення у двовимірному масиві clusterCosts, і кожен вузол містить індекс свого кластера в масиві. Масив утворює симетричну матрицю. Дивіться код евристичного методу кластеризації у Лістингу 2.8.

3.2.4 Побудова шляху

Після завершення алгоритму A*, нам потрібно побудувати шлях із словника cameFrom. Ми починаємо з цільового вузла і ітеруємо, поки не досягнемо початкового вузла, додаючи кожен вузол до стеку. Код для цього наведений у Лістингу 5.9.

3.2.5 Згладжування шляху

Наш алгоритм згладжування шляху працює за принципом "лінії видимості". Ми починаємо з позиції агента і ітеруємося по точках шляху. Якщо точка не знаходиться в поточній лінії видимості, ми вставляємо попередню точку в тимчасовий стек, встановлюємо її як нову початкову точку для променевого кастингу та продовжуємо ітерацію. Коли ми досягаємо останньої точки шляху, ми вставляємо її в тимчасовий стек. Точки шляху в тимчасовому стеку знаходяться у зворотньому порядку, тому ми повертаємо новий стек, знову в зворотньому порядку, для отримання правильного порядку. Дивіться Лістинг 2.10. ----

3.2.6 Навігаційна сітка

Спочатку ми реалізували сцени навмешу, і інші частини реалізації базуються на навмеш-частині, тому ми спочатку описуємо навмеш-частину реалізації.

3.2.6.1 Сценарій NavmeshAIController

Маршрутизація в навмеші переважно визначається сценарієм під назвою NavmeshAIController, який ми можемо знайти в ієрархії, приєднаній до ігрового об'єкта, названого GameManager. NavmeshAIController

успадковується від MonoBehaviour. Сценарій містить кілька публічних атрибутів, які можна налаштовувати в редакторі Unity. Один з них - перемикач для кластеризації, наступний - файл, що містить вартості кластеризації для сцени, третій - перемикач для базової навмеші, і четвертий - перемикач для візуалізації графічного інтерфейсу користувача. Робочий процес сценарію є наступним.

У функції Awake ми розбираємо витрати на рух між кластерами з наданого файлу, якщо увімкнена кластеризація. Потім ми будуємо наш граф навмешу з навмешу, створеного Unity. Для цього ми отримуємо дані з navmesh Unity за допомогою статичного методу NavMesh.CalculateTriangulation. Оскільки дані містять деякі маленькі трикутники, що викликають проблеми, було необхідно спростити меш за допомогою методу SimplifyMeshTopology. Ми також завантажуємо кластери, які присутні на сцені під час побудови графа навмешу та призначаємо їх окремим трикутникам у нашому навмеші.

У функції Start ми завантажуємо всі агенти, що присутні на сцені, керовані нашим сценарієм NavmeshAgent, та ініціалізуємо їх.

Ми робимо дві речі в функції Update. Ми перевіряємо, чи натиснута клавіша "Пробіл", що призводить до паузи гри, і викликаємо метод Navigate для кожного агента, який ми описуємо пізніше.

У функції OnDrawGizmos ми малюємо базову навмешу, і у функції OnGUI ми рендеримо інформацію про вартість шляху та різні кількості вузлів, про які ми розповідали в попередньому розділі.

3.2.6.2 Сценарій NavmeshAgent

Сценарій NavmeshAgent керує штучним інтелектом (ШІ) агента в сцені і виконує пошук шляху. Він також успадковується від MonoBehaviour, оскільки нам потрібно приєднати його до об'єктів агентів і малювати візуалізації у функціях OnDrawGizmos або OnDrawGizmosSelected. Ми використовуємо OnDrawGizmos, якщо в сцені є один агент, і

OnDrawGizmosSelected, якщо в сцені є кілька агентів, щоб мати активну візуалізацію лише для одного агента одночасно. Візуалізація включає в себе відкриті та закриті вузли та розрахований шлях.

В іншому випадку робочий процес цього сценарію наступний. Є два публічних методи, обидва з яких викликаються з NavmeshAIController. Метод Initialize викликається у функції Start NavmeshAIController і створює новий об'єкт AStarSearch для нашого навмешу. Метод Navigate викликається кожен раз під час оновлення через NavmeshAIController, і є код управління для обмеження того, як часто ми розраховуємо новий шлях. Ми викликаємо метод FindPath з Navigate.

Метод FindPath викликає два інші основні методи. Перший - це метод UpdateAStar, який ефективно розраховує шлях, оновлюючи наш граф навігації за допомогою алгоритму A* та вибраної евристики. Ми реалізували його в окремому сценарії AStarSearch разом з евристиками. Другий - це метод ApplyPathSmoothing, який виконує алгоритм згладжування шляху нашого розрахованого шляху.

Є одна важлива деталь в нашому навмешевому пошуку шляху. Наш пошук шляху навмешу досить базовий і створює шляхи тільки на центрах трикутників, на відміну від кращих реалізацій, які також використовують центри країв та вершин трикутника. Однак це має свою мету. Це дозволяє нам показати, що ми можемо отримувати відмінні шляхи, використовуючи алгоритм згладжування шляху, поєднаний з частим перерахуванням шляху, навіть коли використовується такий простий та грубий навмеш. Крім того, шляхи практично непридатні без алгоритму згладжування шляху, на відміну від використання високорозвиненої системи навмешу без згладжування шляху. Звісно, часто неможливо так часто розраховувати нові шляхи через те, що це дуже витратний за ресурсами підхід, що робить цей підхід в межах розумного.

Щодо кластерів, вони створюються вручну на сцені за допомогою колайдерів.

3.2.7 Сітка

Як ми вже зазначили раніше, наш пошук шляху за сіткою базується на нашому навмешевому пошуку шляху. Він використовує той самий клас AStarSearch для розрахунку шляху. Сценарії GridAIController та GridAgent базуються на їхніх навмешевих аналогах та містять ту саму функціональність, за винятком функціональності, що специфічна для навмешу. Сценарії також містять функціональність, специфічну для сітки, на яку ми фокусуємося тут.

3.2.7.1 Сценарій GridAIController

Як ми вже казали, сценарій GridAIController базується на сценарії NavmeshAIController. Він робить ті самі речі, але для сітки, а не для навмешу. Він динамічно створює сітку пошуку шляху при запуску гри. Також містить додаткові публічні атрибути.

У функції Awake ми розбираємо витрати на рух між кластерами з наданого файлу, якщо увімкнена кластеризація. Далі ми завантажуємо збережену сітку з непрохідними вузлами, які представляють стіни.

У функції Start ми завантажуємо кластери, що присутні на сцені, і призначаємо їх окремим вузлам нашої сітки. Далі ми завантажуємо всі агенти, які присутні на сцені, керовані нашим сценарієм GridAgent, та ініціалізуємо їх. Ми також створюємо ігрові об'єкти на місці всіх вузлів сітки, і кожен вузол встановлюється як прохідний або непрохідний.

У функції Update функціональність з NavmeshAIController зберігається та розширюється. Якщо атрибут DevelopmentMode увімкнено, доступні три гарячі клавіші. Одна служить для зміни грид-плану. Вона додає або видаляє стіну у позиції курсора. Ще дві працюють разом. Одна зберігає поточну сітку в файл, вказаний в атрибуті SavedGridName, тоді як інша завантажує сітку з вказаного файлу.

У функції OnDrawGizmos ми малюємо сітку, і якщо гра відтворюється, ми малюємо стіни.

Функція OnGUI ідентична тій, яка є в NavmeshAIController.

3.2.7.2 Сценарій GridAgent

Сценарій GridAgent функціонально ідентичний своєму аналогу NavmeshAgent, за винятком двох нових публічних аргументів. Один вказує тип руху в сітці, чотири або вісім напрямків, і інший перемикає малювання сітки, яка повинна бути відображена поверх відвіданих вузлів.

3.3 Прийняття рішень

У цьому розділі ми розповідаємо про реалізацію гри, інспірованої грою Pac-Man. Спочатку ми описуємо версію зі скінченими автоматами (FSM) та візуалізацією, а потім версію з деревами поведінки (BT) та візуалізацією. Ми базуємося на реалізації пошуку шляху за сіткою для обох версій, яка обробляє оточення і надає нам базові класи для сценаріїв агента сітки та керівника ШІ сітки, які ми розширюємо. Сцени зі скінченими автоматами та деревами поведінки мають чотири агенти (привиди) на сцені, тоді як усі сцени з пошуком шляху мають одного агента на сцені.

3.3.1 Скінчені автомати

Перша частина реалізації Скінчених автоматів була створення гри. Спочатку ми створили лабіринт, що було швидким процесом завдяки нашій готовій реалізації пошуку шляху за сіткою. Потім ми створили сценарії для збору предметів та їх контролера. Після цього ми адаптували основні сценарії для пошуку шляху за сіткою та створили візуалізацію для Скінчених автоматів. Нарешті, ми модифікували алгоритм A*, щоб завадити агентам поворачувати назад, але про це ми вже писали в частині про пошук шляху цього розділу.

Сценарій PickupManager

Сценарій PickupManager керує всім, що стосується збору предметів. Як і GridAIController, він прикріплений до ігрового об'єкта, який називається PickupManager. У нього є кілька публічних атрибутів, серед яких важливі

SpawnInterval - інтервал між окремими появами предметів, та BoosterPickupProbability - ймовірність того, що замість звичайного збору балів випаде бустер. Сценарій контролює створення нових предметів.

Сценарій Pickup

Сценарій Pickup, прикріплений до ігрового об'єкта, представляє собою один зі спавнених об'єктів збору об'єктів будь-якого типу. Коли гравець отримує збір, він нагороджується певною кількістю балів. Якщо це бустер, він також викликає подію для привидів, щоб змінити їх стан на сполошений.

Сценарій GridAIControllerFSM

Сценарій GridAIControllerFSM базується на своєму сітковому аналогу, але виключає кластери, які нам тут не потрібні. Він містить два нових важливих публічних атрибути: перемикач для режиму невразливості та перемикач для початку з паузи, який є частиною агента в сценах сітки.

У функції Awake ми завантажуємо збережену сітку з непрохідними вузлами, які представляють стіни.

Функція Start завантажує всіх агентів, що присутні на сцені, керовані нашим сценарієм GridAgent, та ініціалізує їх. Ми також створюємо ігрові об'єкти на місці всіх вузлів сітки, і кожен вузол встановлюється як прохідний або непрохідний. Тут також ініціалізується PickupManager.

Функція Update (роз)пускає гру при натисканні клавіші "пробіл" і має ту ж функціональність щодо режиму розробки, що і GridAIController. Вона також містить скидання гри. Коли привид ловить гравця, гра закінчується, і гравець може натискати "пробіл", щоб скинути гру. Візуалізація пошуку шляху в функції OnDrawGizmos ідентична тій, що в GridAIController.

Функція OnGUI відображає інформацію про рахунок гравця, час гри, стани всіх привидів і чи наразі гра призупинена. Вона також показує повідомлення про завершення гри "Game Over", коли гра закінчується.

Сценарій GridAgentFSM

Незважаючи на те, що ми базуємо сценарій GridAgentFSM на сценарії GridAgent, він є значно відмінним, оскільки використовує FSM для поведінки агента. Є кілька нових відкритих атрибутів, хоча всі вони призначені лише для конфігурації. Крім того, відкриті атрибути heuristic, movementType і applyPathSmoothing більше не піддаються налаштуванню користувачем. Оскільки гра має жорстке середовище, ми використовуємо евристичну відстань Мангеттену з чотиризмірним рухом і без вирівнювання шляху.

Щодо реалізації FSM, ми побудували свій власний FSM. Ми реалізували кожний стан з переходами в індивідуальному сценарії, що базується на базовому стані, представленому абстрактним класом. Ми запускаємо FSM в сценарії GridAgentFSM в функції Initialize. FSM викликає функції, реалізовані в агенті. Головна з них - TransitionToState, яка виконує перехід в інший стан, Navigate, яка обчислює новий шлях і рухає агента, та ReverseDirection, яка викликається під час кожного переходу для зміни напрямку агента.

Коли агент потрапляє в стан страху, він також викликає функцію, яка обчислює ціль для втечі, і OnTriggerEnter перевіряє зіткнення з гравцем. Ми також реалізували подію, яка спричиняє перехід в стан страху, коли гравець отримує підсилювач. Нарешті, ми додали цілі для привидів в стані розкиду до сцени.

Сценарій AStarDirSearch

Як описано раніше, ми модифікували алгоритм A* для досягнення двох цілей. Перше - зупинити привидів від поворотів навколо себе, і друге - повертати шлях навіть тоді, коли вузол-ціль є поточним вузлом. У цьому випадку привид зробить коло і повернеться на початок/позначку цілі. Єдиний випадок, коли привид повинен повертати, - це перехід від одного стану до іншого.

Візуалізація машиною скінчених станів

Ми візуалізували FSM, використовуючи Unity-пакет Bolt. Результатуючий граф не містить жодної логіки руху FSM і служить лише для візуалізації. Логіка руху знаходиться в наших сценаріях, як ми написали раніше. Візуалізація FSM показана на малюнку 4.5.

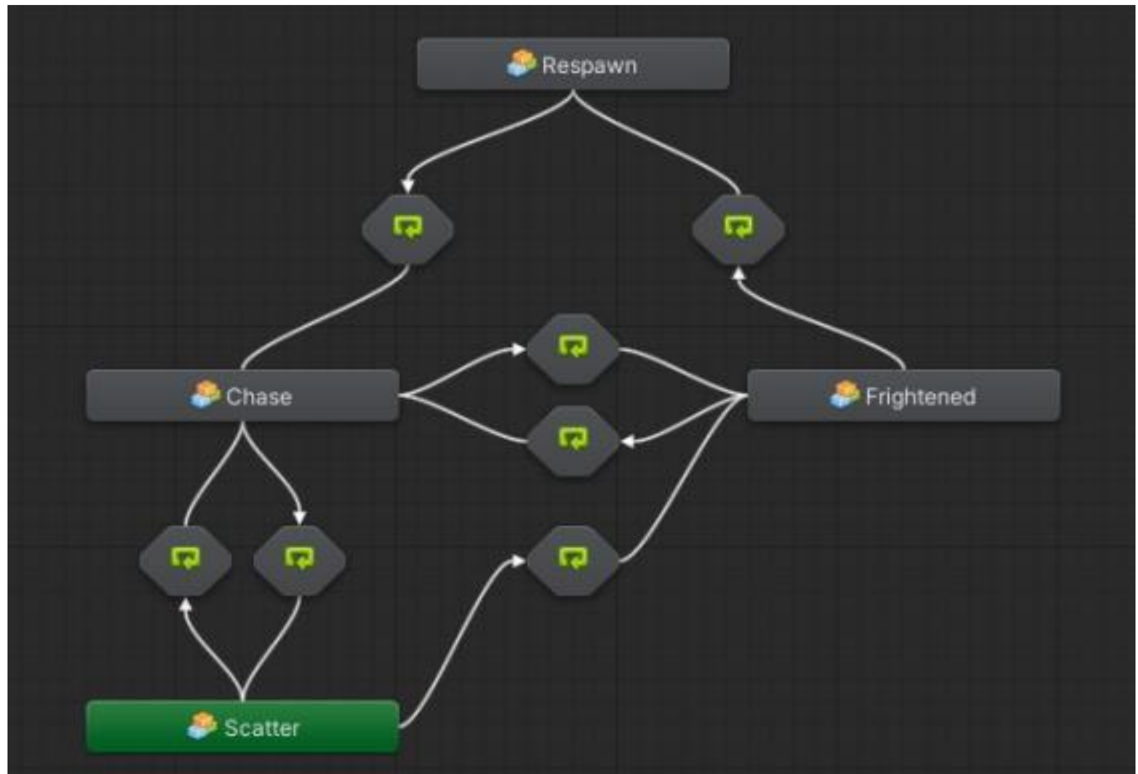


Рисунок 3.2 - Ghost FSM

Сценарій Behavior Trees (Дерева поведінки)

Частина нашого BT використовує реалізацію пошуку шляху за сіткою, що використовується в частині FSM нашої роботи з невеликими модифікаціями. Ми базуємо частину BT, а також візуалізацію на фреймворку Bonsai Behaviour Tree. Ми реалізовуємо наші завдання та декоратор для використання з цим фреймворком і модифікуємо наші сценарії, використовані в частині FSM нашої роботи.

Сценарій GameManagerBT

Сценарій GameManagerBT має ту ж функціональність, що й сценарій GridAIControllerFSM, і ми розширюємо його наступним чином. У функції Start ми ініціалізуємо чорну дошку кожного агента BT. Це дуже важливо, оскільки

інакше ми не могли б отримати доступ до атрибутів агента та викликати його методи з BT. Потім ми ініціалізуємо самі агенти. У функції Update ми викликаємо метод Move кожного агента. Цей метод виконує лише сам рух, тоді як ми вирішуємо, куди йти в BT.

Сценарій GridAgentBT

Сценарій GridAgentBT базується на сценарії GridAgentFSM, і деяка його функціональність є однаковою. Малювання візуалізації пошуку шляху ідентичне. Метод Initialize також функціонально ідентичний. Однак він додатково реєструє подію, яка повідомляє нашому агентові, коли гравець отримує підсилювач, щоб агент переходив у стан страху. Цей сценарій також обробляє зіткнення з гравцем відповідно до стану агента.

3.4 Тестування

Проектування цієї частини було в основному пов'язане з грою. Як вже було сказано, наша гра базується на оригінальній грі Pac-Man, але без деталей. На рисунку 3.3 представлено нашу готову гру.

Рисунок 3.3 - Готова гра

"The Pac-Man Dossier" містить всі подробиці і є матеріалом, на основі якого ми розробляли свою реалізацію. Дизайн візуалізації FSM/BT був

обмежений, оскільки ми вирішили використовувати існуючі рішення та реалізувати їх у нашій програмі. Обидві системи використовували класичні типи візуалізації. Ми будували гру нашою системою пошуку шляху за графом-сіткою, тому візуалізації A* також були доступні у гральних сценах.

Також на рисунках 3.4 і 3.5 представлена поведінка NPC, як реакція на дії гравця.

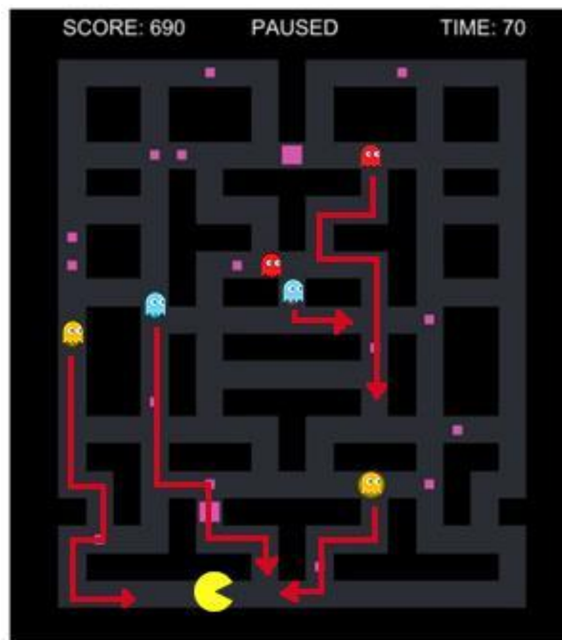


Рисунок 3.4 - Поведінка NPC, як реакція на дії ігрока

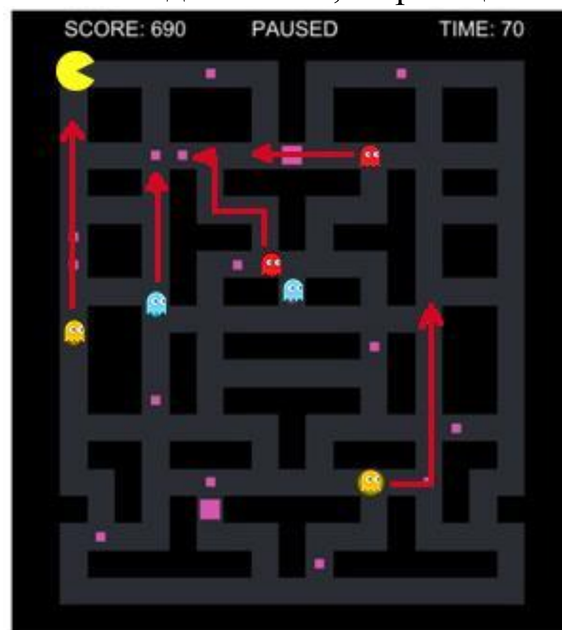


Рисунок 3.5 - Поведінка NPC, як реакція на дії ігрока після зміни його положення на ігровому полі

ВИСНОВКИ

В кваліфікаційній роботі було розв’язано практичної задачі з розробки комплексу інформаційного та програмного забезпечення ігрової системи, що здатна до оцінювання і адаптації до дій гравця. Для досягнення поставленої виконано такі основні завдання роботи:

1. Проведено літературний огляд існуючих підходів до інтеграції модулів оцінки та адаптації в ігрові системи. Визначено ключові аспекти та відмінності підходів, що використовуються в галузі.

2. Розроблено алгоритми та методи оцінки гравців, враховуючи різні аспекти, такі як вміння, стратегії гри та часові показники. Вивчено можливості використання технік машинного навчання для поліпшення точності оцінки.

3. Розроблено алгоритми динамічної адаптації геймплею в реальному часі, використовуючи зібрані дані оцінки гравців, що забезпечує можливість персоналізації геймплею згідно з результатами оцінки.

4. Виконано інтеграцію розробленого модуля в існуючу ігрову систему або створити прототип для тестування.

5. Проведено аналіз працездатності ігрової системи, ефективності запропонованих рішень та можливостей подальшого вдосконалення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Amano Yoshitaka, Lacoste Raphael, van Dijk Jesse. The Art of Gaming. - 2nd Edition. — Future Publishing, 2021. — 148 p.
2. Barney Christopher. Pattern Language for Game Design. - CRC Press, 2021. - 502 p.
3. Brusca Victor. Introduction to Video Game Engine Development: Learn to Design, Implement, and Use a Cross-Platform 2D Game Engine. - Apress, 2021. — 391 p.
4. Cai Y., Cao Q. (Eds.). When VR Serious Games Meet Special Needs Education: Research, Development and Their Applications. - Springer, 2021. — 220 p.
5. Cooper K. (Ed.). Software Engineering Perspectives in Computer Game Development.- Chapman and Hall/CRC, 2021. — 313 p.
6. D’Amato James. The Ultimate RPG Game Master's Worldbuilding Guide: Prompts and Activities to Create and Customize Your Own Game World.- Adams Media, 2021. — 272 p.
7. Keith Clinton. Agile Game Development: Build, Play, Repeat . -2nd Edition. — Pearson Eduction, Inc., 2021. — 572 p.
8. Sobota Branislav (ed.) Computer Game Development. - ITexLi, 2022. — 166 p.
9. Soderman Braxton. Against Flow: Video Games and the Flowing Subject. - The MIT Press, 2021. — 328 p.
10. Zizo Rizk. C++ Game Programming: New Book Learn C++ from scratch and start build your very own new games step by step. - Independently published, 2021. — 328 p.

11. Yeung D., Luckraz S., Leong C.K. (Eds.) Frontiers in Games and Dynamic Games: Theory, Applications, and Numerical Methods. - Samsung, 2020. — 243 p.
12. Sanders Benjamin. Foundations of Videogame Programming: Code Repository . - Independently published, 2020. — 542 p.
13. Dillon Roberto (Editor). The Digital Gaming Handbook. - CRC Press, 2020. — 420 p
14. Despain Wendy. Professional Techniques for Video Game Writing. - 2nd Edition. — Chapman and Hall/CRC, 2020.
15. Heussner Tobias. The Advanced Game Narrative Toolbox .-CRC Press, 2019. — 232 p.
16. Weilkiens Tim. The New Engineering Game: Strategies for smart product engineering. - Packt Publishing, 2019. — 104 p.
17. Aslam H., Brown J.A. Affordance Theory in Game Design: A Guide Toward Understanding Players. - Morgan & Claypool, 2020. — 113 p.
18. Fields T. Game Development 2042: The Future of Game Design, Development, and Publishing. - CRC Press, 2023. — 331 p.
19. Karpouzis K., Yannakakis G.N. (Eds.) Emotion in Games: Theory and Praxis.- Springer, 2016. — 344 p.
20. Korn Oliver, Lee Newton. Game Dynamics .- Springer, 2017. — 176 p.
21. Alam Asadullah. Unity Physics Mastery: Introducing Gravitation and Rotation for Game Developers: Craft Dynamic Rigid Body Motion with C# in Unity 3D - A Comprehensive Guide for Beginners . - Independently published, 2023. — 298 p.
22. Alves Claudia. Unity 3d: Build, customize, and optimize professional games using unity 3d . - 2nd Edition. — Independently published, 2021. — 232 p.
23. Barrera Ray. Unity AI Game Programming .- Packt Publishing, 2015. — 237 p.

24. Buttfield-Addison P., Manning J., Nugent T. Unity Development Cookbook: Real-Time Solutions from Game Development to AI .- 2nd edition. — O'Reilly Media, 2023. — 433 p.
25. Cossu Sebastiano M. Beginning Game AI with Unity: Programming Artificial Intelligence with C#.- Apress Media LLC, 2021. — 161 p.
26. Felicia P. Unity 5 from Proficiency to Mastery. Artificial Intelligence .- Patrick Felicia, 2017. — 375 p.
27. Palacios J. Unity 2018 Artificial Intelligence Cookbook - 2nd Edition .- Packt Publishing, 2018. — 334 p.
28. Pêcheux Mina. AI Programming: Boost your Unity/C# .- Independently published, 2023. — 276 p.

ДОДАТОК

Лістинги:

2.1 A* algorithm

2.2 StepsSoFar incrementation

2.3 No immediate exit condition

2.4 One modiication of the A* algorithm

2.5 Manhattan distance heuristic

2.6 Diagonal distance heuristic

2.7 Euclidean distance heuristic

2.8 Clustering heuristic

2.9 Path construction algorithm

Лістинг 2.1

```
1 HashSet <Tuple <Node , NodeType > > UpdateAStar (
NavigationGraph graph , Node startNode , Node goalNode ,
Heuristic heuristic , MovementType movementType ) {
2 cameFrom . Clear () ;
3 costSoFar . Clear () ;
4 closedNodes . Clear () ;
5 openNodes . Clear () ;
6
7 openNodes . Enqueue ( startNode , 0) ;
8 costSoFar [ startNode ] = 0;
9
10 while ( openNodes . Count > 0) {
11 Node current = openNodes . Dequeue () ;
12 closedNodes . Add( current ) ;
13 if ( current == goalNode )
14 return VisitedNodes ( closedNodes , openNodes ) ;
15
16 foreach ( Node next in current . Neighbors ( movementType )
) {
17 double newCost = costSoFar [ current ]
18 + graph . Cost ( current , next ) ;
19 if ( openNodes . Contains ( next )
20 && newCost < costSoFar [ next ]) {
21 openNodes . Remove ( next ) ;
22 }
23 if ( closedNodes . Contains ( next )
24 && newCost < costSoFar [ next ]) {
25 closedNodes . Remove ( next ) ;
26 }
27 if (! openNodes . Contains ( next )
28 && ! closedNodes . Contains ( next ) ) {
29 costSoFar [ next ] = newCost ;
30 cameFrom [ next ] = current ;
31 double priority = newCost + ComputeHeuristic (
heuristic , next , goalNode , clusterCosts ) ;
32 openNodes . Enqueue (next , priority ) ;
33 }
34 }
35 }
36 return VisitedNodes ( closedNodes , openNodes ) ;
```

Лістинг 2.2

```
stepsSoFar [ next ] = stepsSoFar [ current ] + 1;
```

Лістинг 2.3

```
if ( current == goalNode } && costSoFar [ current ] >= 1)
```

Лістинг 2.4

```
One modification of the A* algorithm
1 if ( stepsSoFar [ current ] < 1 &&
2 WrongDirection ( startNode , next , startDir ) )
3 continue ;
4 if ( next . Equals ( goalNode ) &&
5 stepsSoFar [ current ] >= 2) {
6 cameFrom [ next ] = current ;
```

```
7 closedNodes . Add ( next ) ;
8 return VisitedNodes ( closedNodes , openNodes );
9 }
```

Лістинг 2.5

```
1 int D = 1;
2 double dx = Math .Abs( goal . Position .x
3 - next . Position .x) ;
4 double dz = Math .Abs( goal . Position .z
5 - next . Position .z) ;
6 return D * (dx + dz) ;
```

Лістинг 2.6

```
1 int D = 1;
2 double D2 = Math . Sqrt (2) ;
3 double dx = Math .Abs( goal . Position .x
4 - next . Position .x) ;
5 double dz = Math .Abs( goal . Position .z
6 - next . Position .z) ;
7 return D * (dx + dz) + (D2 - 2 * D)
8 * Math . Min (dx , dz) ;
```

Лістинг 2.7

```
1 int D = 1;
2 double dx = Math .Abs( goal . Position .x
3 - next . Position .x) ;
4 double dz = Math .Abs( goal . Position .z
5 - next . Position .z) ;
6 return D * Math . Sqrt (dx * dx + dz * dz) ;
```

Лістинг 2.8

```
1 return clusterCosts [ next . Cluster . Index ][ goal .Cluster . Index ] *
clusteringMultiplier ;
```

Лістинг 2.9

```
1 Stack < Vector3 > path = new Stack < Vector3 >() ;
2 Node current = goalNode ;
3 do {
4 path . Push ( current . Position ) ;
5 if (! aStar . CameFrom . TryGetValue ( current , out
current ) ) {
6 path . Clear () ;
7 lastPathLength = 0;
8 lastPathCost = 0;
9 return ;
10 }
11 }
12 while ( current != startNode ) ;
```