ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до магістерської роботи

на ступінь вищої освіти магістр

на тему: «Розробка методики створення бота у відеоіграх в жанрі шутер на основі штучного інтелекту»

Виконав: студент 6 курсу, групи ПДМ–61

спеціальності

121 Інженерія програмного забезпечення

(шифр і назва спеціальності/спеціалізації)

Сабадах В.С

(прізвище та ініціали)

Керівник Золотухіна О.А

(прізвище та ініціали)

Рецензент _____

(прізвище та ініціали)

Київ – 2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти - «Магістр»

Спеціальність підготовки – 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ Негоденко О.В.

“____”_____2021 року

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТА

Сабадаху Владиславу Сергійовичу

(прізвище, ім’я, по батькові)

1. Тема роботи: «Розробка методики створення бота у відеоіграх в жанрі шутер на основі штучного інтелекту»_____

_____

Керівник роботи:____Золотухіна О.А, доцент кафедри, кандидат технічних наук__

(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом вищого навчального закладу від «13» жовтня 2021 року №230.

2. Строк подання студентом роботи___«24» грудня 2021 року

3. Вхідні дані до роботи

Методи обробки ШІ за допомогою алгоритмів;

Науково-технічна література з питань, пов’язаних зі штучним інтелектом у іграх та у програмних забезпеченнях.

4. Зміст розрахунково-пояснювальної записки(перелік питань, які потрібно розробити).

4.1 Що таке детерміновані алгоритми штучного інтелекту.

4.2 Які методи кінцевих автоматів.

4.3 Опис власної моделі штучного інтелекту

4.4 Створення власної моделі штучного інтелекту

4.5 Створення автономно функціонуючих елементів

5. Перелік демонстраційного матеріалу (назва основних слайдів)

1. Мета, об’єкт та предмет дослідження
2. Існуючі алгоритми вирішення задач
3. Отримані результати роботи
4. Результати науково-дослідної практики

6. Дата видачі завдання «01» листопада 2021

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	02.08-07.08	Виконано
2	Вимоги до системи	22.08-24.08	Виконано
3	Створення простої моделі поведінки ШІ	02.08-05.08	Виконано
4	Створення рівня, для тестування ШІ	17.08-26.08	Виконано
5	Будування детального ШІ для тестування на комплексному рівні	04.09-30.09	Виконано
6	Вступ, висновки та основна частина	05.10-12.10	Виконано
7	Розробка демонстраційного матеріалу магістерської роботи	17.11-25.11	Виконано
8	Попередній захист роботи	16.12	
9	Здача роботи	24.12	

Студент _____

(підпис) (прізвище та ініціали)

Керівник роботи _____

(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина магістерської роботи 94 с., 57 рис., 63 джерел.

Об'єкт дослідження – поведінка ігрового бота у відеогрі в жанрі шутер.

Предмет дослідження – методи штучного інтелекту для відеоігор у жанрі шутер.

Мета роботи *– підвищення якості ігрових ботів у відеоіграх у жанрі шутер за рахунок використання методів штучного інтелекту.*

Існують різні види алгоритмів, за допомогою яких можно будувати штучний інтелект. Серед них можно виділити наступні:

Детерміновані алгоритми штучного інтелекту. Детерміновані алгоритми означають зумовлене і заздалегідь запрограмовану поведінку об'єктів.

Метод кінцевих автоматів. Вимога розумної кількості станів досить просто і зрозуміло. У нас, людей, є сотні, якщо не тисячі, емоційних станів, і в кожному з них - безліч підстанів.

А персонаж гри повинен всього лише виглядати розумним, так що і станів у нього не повинно бути надто багато. Наприклад, простий персонаж гри може мати декілька станів.

Запам'ятовування і навчання. Всі розглянуті технології ШІ працюють тільки з поточною інформацією і ніколи не враховують події, що вже відбулися. Як приклад використання пам'яті штучним інтелектом. Якщо у персонажу закінчуються боєприпаси, то набагато розумніше не починати пошук наосліп, тобто випадковим чином, а просканувати всі записи і "згадати", в якому приміщенні залишилися незаймані боєприпаси.

Алгоритм проходження за об'єктом. Штучний інтелект на основі інформації про стан деякого об'єкту, що відслідковується, змінює траєкторію нашого об'єкта таким чином, щоб він рухався у напрямку до відстежуваного об'єкту. Рух може бути спрямовано як безпосередньо на відстежуваний об'єкт, так і за деякою кривою, спрямованої до об'єкту (на зразок траєкторії самонавідної ракети).

Галузь використання – розробка ігор.

ЗМІСТ

ВСТУП

На сьогоднішній день розробка ігор є трудомістким і складним процесом, який включає до себе безліч галузей від графічного оформлення, сценарної майстерності до програмування. Широке застосування в ігровій індустрії відведено розробці штучного інтелекту (ШІ). До ШІ належать такі аспекти відеоігри: управління анімацією, рульове управління, процес пошуку на місцевості, планування, тактичне і стратегічне мислення і навчання [1, 2]. Розвиток нових методів і технологій ШІ є основною проблемою в розробці шутерів від першої особи, оскільки ШІ це складно-програмований комплекс команд, який заснований на системі прийняття рішень в певній ситуації.

Але ігровий ШІ є лише однією з гілок більш широкого застосування штучного інтелекту. Згідно Алану Тьюрингу, якого вважають батьком штучного інтелекту, агент є інтелектуальним, якщо його поведінка неможливо відрізнити від людини [3]. Ігровий ШІ додає ще одну частину визначення ШІ. Для того, щоб гра була успішною, не потрібні високорозумні, людиноподібні, непереможні експертні опоненти, потрібно впровадити переконливих супротивників, агентів ШІ, з якими не нудно грати [4].

Тому більшість ігор використовують дуже прості методи ШІ такі як кінцеві автомати (FSM) і дерева прийняття рішень [1, 2].

Відеоігра може розглядатися як така, що має два основні аспекти - контекст та гру. Гра включає елементи, що визначають фактичні виклики, з якими стикаються гравці, та проблеми, які вони мають вирішити, такі як правила та цілі. З іншого боку, контекст охоплює всі елементи, що складають обстановку, в якій виникають ці проблеми, такі як персонажі та сюжет. Дана дипломна робота зосереджена на ігровому ШІ, тобто ШІ, який займається вирішенням проблем у грі, таких як перемога суперника в бою або навігація в лабіринті. І навпаки, контекстний ШІ мав би справу з конкретними контекстними завданнями, такими як змусити персонажа виконати низку дій, спрямованих на просування платформи, що реагує на вибір гравця.

Оскільки відеоігри створені для людей, цілком природно, що вони зосереджуються на своїх пізнавальних навичках та фізичних здібностях. Чим багатша і складніша гра, тим більше навичок і вмінь вона вимагає. Таким чином, створення справді розумного та повністю автономного агента для складної відеоігри може бути вражаючим, що відтворює великі частини повноцінного людського інтелекту. З іншого боку, ШІ зазвичай розробляється незалежно для кожної гри. Це ускладнює створення ретельно надійного ШІ, оскільки його розвиток обмежений рамками окремого ігрового проекту. Хоча кожна відеоігра є унікальною, вони можуть поділяти низку концепцій залежно від свого жанру. Жанри використовуються для класифікації відеоігор відповідно до способу взаємодії гравців з ними, а також їх правил. На концептуальному рівні відеоігри одного жанру, як правило, мають подібні виклики, засновані на одних і тих же концепціях. Потім подібні виклики включають загальні проблеми, для яких можна визначити та застосувати базову поведінку незалежно від екземпляру проблеми. Наприклад, у поєдинку від першої особи «один на один» гравці стикаються з такими проблемами, як вибір зброї, прогнозування позиції суперника та навігація. Кожного моменту гравцеві потрібно оцінити ситуацію і перейти на найбільш підходящу зброю, передбачити, куди, швидше за все, прямує або рухається супротивник, і знайти найкращий шлях, щоб туди дістатися. Усі ці проблеми можуть бути сприйняті як такі, що стосуються швидкості стрільби зі зброї, поточного стану здоров'я супротивника та розташування медичних наборів. Ці концепції є загальними для багатьох ігор шутерів від першої особи і їх достатньо визначити ефективну поведінку незалежно від деталей їх інтерпретації. Такі рішення вже існують для певних навігаційних проблем, наприклад, і використовуються у багатьох відеоіграх. Більше того, люди-гравці часто можуть без особливих зусиль використовувати досвід, отриманий від однієї відеоігри, в іншому того ж жанру. Гравець, який має досвід роботи в шутерах від першої особи, в більшості випадків буде виграти краще в новій грі шутер від першої особи, ніж той, хто не має досвіду, і може навіть виграти краще, ніж гравець з певним досвідом

у новій грі, вказуючи на те, що застосовувати вивчену поведінку в одній грі, а також в інших іграх, особливості подібних концепцій, щоб добре працювати, не знаючи деталей останньої. Очевидно, що коли деталі виявляються, вони можуть бути використані для подальшого вдосконалення базової концептуальної поведінки або навіть перевизначення її. Тому може бути можливим створити міжігровий ШІ шляхом виявлення та націлювання на концептуальні проблеми, а не на їх конкретні ігрові випадки. Від'єднання ШІ або його частини від розробки відеоігор призведе до усунення обмежень проекту, які змушують розробників обмежувати його та дозволять йому мати безперервний і ретельний процес проектування.

Об’єкт дослідження – відеогра в жанрі шутер.

Предмет дослідження – використання штучного інтелекту в відеоіграх у жанрі шутер.

Мета роботи – розробка методики створення боту в відеоіграх у жанрі шутер на основі штучного інтелекту.

Мета роботи визначає наступні завдання дипломної роботи:

1. Зробити аналітичний огляд типів комп’ютерних ігор, де використовується штучний інтелект;

2. Зробити аналітичний огляд сукупності існуючих математичних алгоритмів для розробки штучного інтелекту з детальним розкриттям їх механізмів;

3. Розробити відеогру в жанрі шутер від першої особи з створенням власної моделі штучного інтелекту як частину загальної методики створення боту в таких іграх.

1.АНАЛІТИЧНИЙ ОГЛЯД ТИПІВ КОМП'ЮТЕРНИХ ІГОР, ДЕ ВИКОРИСТОВУЄТЬСЯ ШТУЧНИЙ ІНТЕЛЕКТ

На сьогодні все різноманіття відеоігор можна класифікувати за жанрами згідно рис. 1.1.

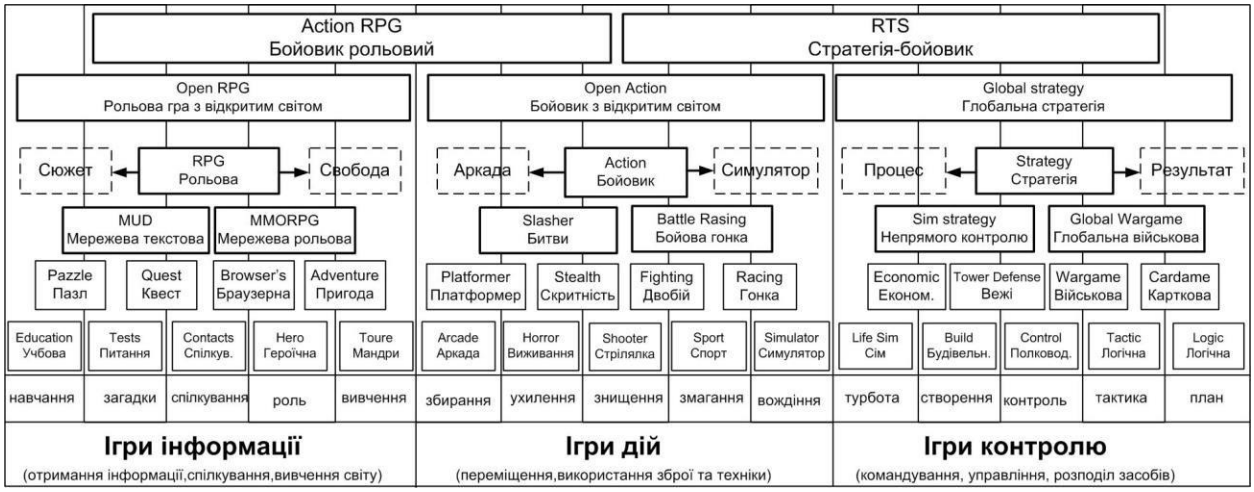


Рис. 1.1. Сучасна класифікація відеоігор за жанрами [5]

Концептуалізація відеоігор - це процес, який включає абстракцію і подібний до багатьох інших підходів, які поділяють одну ціль, а саме до розробки штучного інтелекту (ШІ) в відеоігор. У більш загальному плані абстракція дає можливість створювати рішення для цілих сімей проблем, які по суті однакові, коли певний рівень деталізації пропущений. Наприклад, проблема сортування масиву може мати різну форму залежно від типу елементів у масиві, але врахуванняабстрактного типу даних та функції порівняння дозволяє програмісту написати рішення, яке може сортувати масив будь-якого типу . Це запобігає непотрібному копіюванню коду та допомагає програмістам максимально використовувати існуючі рішення, щоб мінімізувати зусилля з розробки. Іншим прикладом широко використовуваного додатка для абстракції є апаратне абстрагування. Фізичні компоненти комп'ютера можна розглядати як абстрактні пристрої для спрощення розробки програмного забезпечення. Різні фізичні компоненти, що служать тій

самій меті, наприклад, сховище, можуть бути абстраговані в єдиний абстрактний тип пристрою зберігання даних, що дозволяє розробникам програмного забезпечення писати програми зберігання, які працюють з будь-якими компонентами сховища. Такий механізм використовується в операційних системах, таких як NetBSD [5] та сімействі операційних систем Windows NT [6].

Ідея створення уніфікованого проміжного програмного забезпечення ШІ для відеоігор не нова. Міжнародна асоціація розробників ігор (IGDA) заснувала в 2002 році Комітет зі стандартизації інтерфейсу штучного інтелекту (AIISC), метою якого було створення стандартного інтерфейсу AIinterface, який можна реалізувати, а навіть передавати код AI на аутсорсинг [7]. Комітет складався з декількох груп, кожна група зосереджувалась на певній проблемі. Була група, яка працювала над взаємодією в світі, одна з питань керування, одна з пошуку шляхів, одна з кінцевими автоматами, одна з систем, що базуються на правилах, і одна, що займається цільовим плануванням дій, хоча група, яка працює над системами, що базуються на правилах розчинені [7-9]. Таким чином, комітет був стурбований не лише створенням стандартного комунікаційного інтерфейсу між відеоіграми та ШІ, а й створенням стандартного ШІ [10]. Було припущено, що встановлення стандартів ШІ може призвести до створення спеціалізованого обладнання для ШІ.

Ідея створення проміжного програмного забезпечення ШІ для відеоігор також обговорюється в [11], де досліджуються технічні проблеми та підходи до створення такого проміжного програмного забезпечення. Серед іншого, стверджується, що при розгляді державних систем розробники відеоігор вимагають рішення між простими кінцевими автоматами та складними когнітивними моделями. Іншим цікавим аргументом є те, що бібліотеки функціональних можливостей були б більш доречними, ніж комплексні агентські рішення, оскільки вони забезпечують більшу гнучкість, дозволяючи при цьому створювати рішення на основі агентів. Тут також була спроектована можливість створення творів, спеціально обладнаних AI, і проведена паралель із впливом основних графічних прискорень на розвиток комп'ютерної графіки.

Відкрита специфікація стандартного інтерфейсу ШІ (OASIS) пропонується у роботі [12], спрямована на спрощення інтеграції ШІ до відеоігри. Структура OASIS розроблена для підтримки представлення знань, а також міркувань та навчання і складається з п’яти шарів, кожен з яких має справу з різними рівнями абстракції, такими як рівень об’єкта або рівень домену, або надає різні послуги, такі як послуги доступу, перекладу чи арбітражу цілей. . Нижні шари займаються взаємодією з грою, тоді як верхні шари представляють знання та міркування.

Очевидно, що відеоігри AImiddleware можна знайти і в движках відеоігор. Двигуни для відеоігор, такі як Unity [13], Unreal Engine [14], CryEngine [15] та Havok [16], хоча це і не може бути їх основною метою, все частіше спрямовані на створення не тільки будівельних блоків для створення реалістичних віртуальних середовищ, але реалістичні агенти.

Інший підхід, який, хоча і не стосується ШІ зокрема, також поділяє подібну мету, яка полягає у факторі зусиль, спрямованих на розвиток у галузі відеоігор, - це ігрові моделі. Шаблони ігрового дизайну дозволяють розробникам ігор документувати повторювані дизайнерські проблеми та рішення таким чином, щоб їх можна було використовувати для різних ігор, допомагаючи їм зрозуміти вибір дизайну, який бере участь у розробці гра конкретного жанру. У роботі [17] пропонує формалізм візерунків, щоб допомогти розширити знання про дизайн гри. Формалізм описує ігрові схеми з використанням чотирьох елементів. Це назва, проблема, рішення та наслідки. Проблема описує мету та перешкоди, з якими можна зіткнутися, а також контекст, у якому вона виникає. Рішення описує абстрактні механізми, особливості, що використовуються для вирішення проблеми, і як наслідок, описує вплив вибору дизайну на інші частини розробки та його витрати та переваги.

Бьорк [18] розмежовує структурну структуру, яка описує ігрові компоненти, та схеми ігрового дизайну, що описують взаємодію гравців під час гри. Структурна структура включає три категорії компонентів. Це обмежуюча категорія, яка включає компоненти, які використовуються для опису дозволених дій у грі, таких

як правила та режими гри, тимчасова категорія, яка включає компоненти, які беруть участь у часовому виконанні гри, такі як дії та події та цільова категорія, яка включає конкретні ігрові елементи, такі як гравці або персонажі. Детальніше про цю структуру можна дізнатись у [19]. Що стосується зразків ігрового дизайну, вони не включають проблемних та вирішувальних елементів, як це робиться у [17]. Вони описуються за допомогою п’яти елементів, які є ім’ям, описом, наслідками, використовуючи шаблон та відношення. Елемент наслідків тут зосереджується більше на характеристиках візерунка, а не на його впливі на розробку та інші варіанти дизайну, які слід врахувати, що є роллю використання елемента візерунка. Елемент відносин використовується для опису відносин між зразками, наприклад, підшаблони у зразках та суперечливі зразки.

В [20], шаблони дизайну інтегровані до концептуальної моделі взаємозв'язку, яка використовується для уточнення поділу проблем між ігровими моделями та механікою гри. У цій моделі ігрова механіка походить від ігрових зразків через шар контекстуалізації, роль якого полягає в конкретизації цих зразків. І навпаки, нові шаблони можна виділити з конкретної реалізації цієї ігрової механіки, яка в моделі представлена як код.

Також можна порівняти підходи, які зосереджені на вирішенні конкретних питань ШІ. Легко зрозуміти, чому, оскільки ці підходи, як правило, спрямовані на надання стандартних рішень загальних проблем ШІ у відеоіграх, тим самим впливаючи на розвиток ШІ. Наприклад, створення моделей для інтелектуальних персонажів відеоігор є широко дослідженою проблемою, для якої пропонується багато підходів. Мови поведінки мають на меті надати модель дизайну агенту, яка дозволяє інтуїтивно визначати поведінку та враховувати загальні процеси. Лоялл та Бейтс [21] представляють цільову архітектуру реактивного агенту, яка дозволяє визнавати та реагувати на події, що змінюють відповідність поточної поведінки. ABL, реактивна мова планування, призначена для створення правдоподібних агентів, що підтримує координацію багато характерів, описана в Матеасі та Штерні [22] та Матеасі та Штерна [23].

Калькуляція ситуації була запропонована, якби уможливити міркування та контроль на високому рівні у Фанге [24]. Це дозволяє персонажу бачити світ як послідовність ситуацій і розуміти, як він може змінюватися від однієї ситуації до іншої під впливом різних дій, щоб мати можливість приймати рішення та досягати цілей. Мова когнітивного моделювання (CML), що використовується для вказівки контурів поведінки для автономних персонажів і яка використовує обчислення ситуації та використовує інтервальні методи, щоб дозволити персонажам створювати плани дій у дуже складних світах, також запропонована у Фунге [25, 26].

В Оркіні [27,28] стверджувалося, що планування в режимі реального часу є більш підходящим підходом, ніж сценарії або кінцеві автомати для визначення поведінки агента, оскільки це дозволяє обробляти несподівані ситуації більш природно. Представлена модульна цільово орієнтована архітектура планування дій для ігрових агентів, подібна до тієї, що використовується у Матеаса та Стерна [26,27]. Основна відмінність від мови ABL полягає в тому, що між реалізацією та даними проводиться поділ. За допомогою ABL дизайнери реалізують поведінку безпосередньо. Тут реалізація здійснюється програмістами та дизайнерами, які визначають поведінку за допомогою даних.

Андерсон [29] пропонує іншу мову для дизайну розумних персонажів. Мова визначення аватари (AvDL) дозволяє визначити як детерміновану, так і цільову поведінку f або взагалі віртуальні сутності. Вона була розширена за допомогою TheSimpleEntityAnnotationLanguage (SEAL), що дозволяє безпосередньо вбудовувати визначення поведінки в об'єкти у віртуальному світі, анотуючи та дозволяючи символам обмінюватися з ними інформацією [30, 31].

Нарешті, навчання являє собою інший підхід, який, знову ж таки, веде до тієї ж мети. Створюючи агентів, здатних навчатись у своєму середовищі та пристосовуватися до них, питання створення інтелектуальних персонажів відеоігор вирішується більш загальним та багаторазовим способом. Відеоігри викликали великий інтерес у спільноти машинного навчання протягом останнього

десятиліття, і було зроблено кілька спроб інтегрувати навчання у відеоіграх з різним ступенем успіху. Деякі використовувані методи подібні до попередньо згаданих підходів тим, що вони використовують абстракцію або концепції для боротьби з великим розмаїттям відеоігор. Методи міркувань, що базуються на конкретних випадках, узагальнюють інформацію про стан гри, щоб змусити ШІ поводитися більш послідовно в різних, але схожих конфігураціях. Можливість використання розпізнавання планів на основі конкретних випадків для зменшення передбачуваності комп'ютерних програвачів стратегій у реальному часі обговорюється в Чанг [32]. Ага [33] представляє підхід до вивчення справи та підбору плану, який використовується у агента, який вчиться перемагати проти ряду різних опонентів ШІ в Уоргусі. В [34], розроблена та успішно апробована в Wargus також структура планування стратегічних ігор у реальному часі, яка дозволяє агентам автоматично витягувати поведінкові знання з анотованих експертних повторів. Більше роботи, використовуючи Wargusasatestplatformin, включає Матеас [35, 36] які демонструють, як концептуальні сусідські райони можуть бути використані для пошуку підходів, що базуються на конкретних випадках.

Підходи до трансферного навчання намагаються використати досвід, засвоєний з якогось завдання, для поліпшення поведінки в інших завданнях. У Шарма [37], трансферне навчання досягається поєднанням міркувань на основі конкретних випадків та навчання з підкріпленням та використовується для вдосконалення ефективності над наступними іграми проти ШІ в MadRTS.Lee-Urban та ін.

MadRTS застосовує трансфертне навчання за допомогою модульної архітектури, яка інтегрує планування ієрархічної мережі завдань (HTN) та навчання концепціям. Передача структурних навичок та понять між різними завданнями за допомогою когнітивної архітектури досягається у роботі Шапіро [39].

Хоча технологія машинного навчання може призвести до створення уніфікованого ШІ, який можна використовувати в декількох іграх, в даний час вона

страждає від недостатньої зрілості. Навіть якщо деякі методи успішно застосовані до кількох комерційних ігор, може пройти багато часу, поки вони стануть достатньо надійними, щоб стати основними. З іншого боку, двигуни для відеоігор широко використовуються і становлять більш практичний підхід до факторингу процесів розробки ігор для підвищення якості відеоігор. Однак вони є всеосяжними інструментами, які розробники повинні застосувати для всього ігрового дизайну, а не лише для свого ШІ. Крім того, вони не допускають свободи у фундаментальній архітектурі агентів, якими керують.

2. АНАЛІЗ І ДЕТАЛІЗАЦІЯ МАТЕМАТИЧНИХ АЛГОРИТМІВ ДЛЯ РОЗРОБКИ ШТУЧНОГО ІНТЕЛЕКТУ

2.1. Детерміновані алгоритми штучного інтелекту

Детерміновані алгоритми означають зумовлене і заздалегідь запрограмовану поведінку об'єктів. Наприклад, якщо розглянете систему ШІ в демонстраційній грі Asteroids (рис. 2.1), то побачите, що він дуже простий.

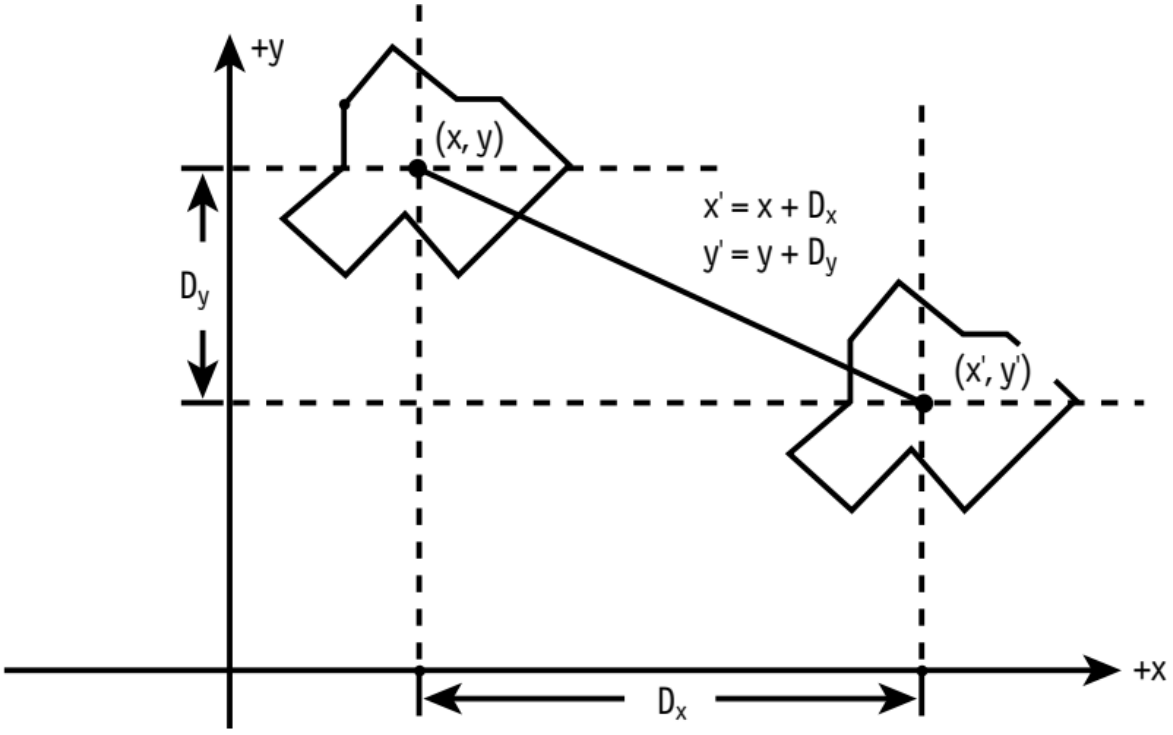


Рис. 2.1. Штучний інтелект гри Asteroids [40].

Штучний інтелект створює астероїд, який потім відправляє в політ у випадковому напрямку з випадковою швидкістю. Вся математика ШІ зводиться до двох рядків програми:

asteroid_x + = asteroid_x_veLocity;

asteroid_y + = asteroid_y_velocity;

У астероїда одна мета: рухатися за своїм курсом. Усе. Тому штучний інтелект астероїда гранично простий - йому не потрібно обробляти певне зовнішнє введення інформації, змінювати курс і т.п. Поведінка астероїдів повністю детермінована і передбачувана.

Саме такі, прості та передбачувані алгоритми ШІ і складають перший клас алгоритмів. У цьому класі є ряд технологій, які були розроблені ще за часів ігор типу PacMan.

2.1.1. Випадковий рух

В описаному вище алгоритмі досить ввести мінімальні зміни, щоб отримати хаотичне переміщення об'єкта (або зміна його властивостей), як показано на рис. 2.2.

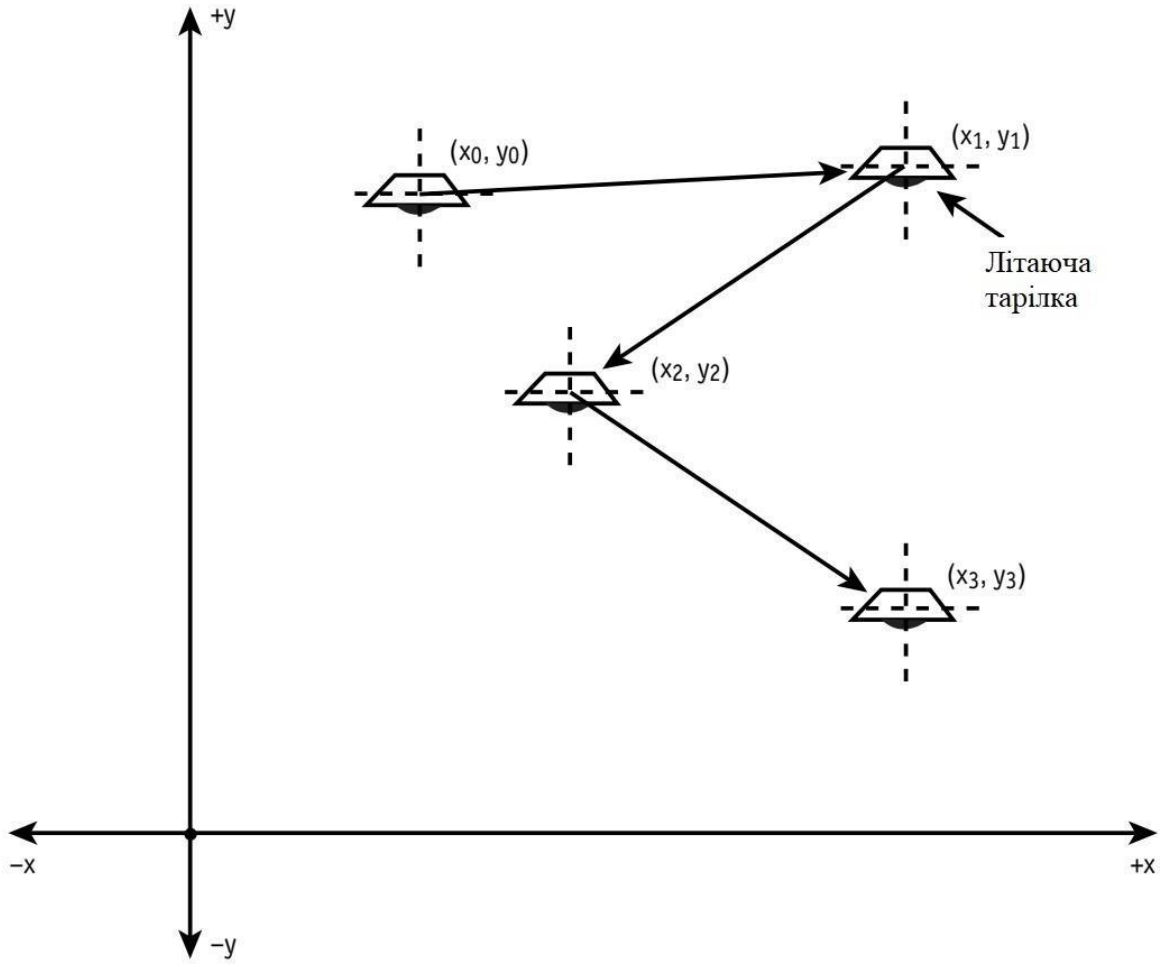


Рис. 2.2. Випадковий рух об'єкту [41].

Наприклад, якщо треба змоделювати броунівський рух атома або якийсь інший об'єкт з хаотичним рухом, можна вчинити так. Спочатку випадковим чином змінюється швидкість об'єкта:

```
fLy_x_veLocity = -8 + rand ()% 16;
fly_y_velocity = -8 + rand ()% 16;
```

Потім об'єкт деякий час рухається з обчисленою швидкістю:

```
// Рух із заданою швидкістю протягом 10 циклів
for (int fly_count = 0; fly_count <10; ++ fly_count)
{
fly_x + = fly _x_veLocity;
fly_y + = fly_y_veLocity;
// ... Обчислення нової швидкості об'єкту ...
```

В даному прикладі спочатку вибирається випадковий напрям руху і швидкість, після чого протягом 10 циклів об'єкт переміщається з використанням обраного вектору швидкості. Потім обирається новий напрямок і швидкість, і цикл повторюється. Очевидно, що до даного алгоритму можна ввести додатковий рівень хаотичності, наприклад вибираючи деяке випадкове число замість 10 циклів переміщення. Крім того, можна віддати перевагу вибору одних напрямків перед іншими (тим самим, наприклад, можна зімітувати наявність вітру).

Випадковий рух являє собою важливу частину поведінкової моделі інтелектуальних об'єктів гри. Невелика примітка про інтелектуальності: я живу в Кремнієвій Долині, і можу засвідчити, що водії тут часто поводяться відповідно до описаного алгоритмом: хаотично змінюють смуги руху, а іноді і просто рухаються по зустрічній смузі [42].

2.1.2. Алгоритм проходження за об'єктом

Хоча випадкове переміщення і є абсолютно непередбачуваним, зазвичай від нього мало толку. Наступним кроком у розвитку ШІ є алгоритми, які враховують стан зовнішнього середовища і реагують на нього. Як приклад, обирається

алгоритм проходження за об'єктом (tracking algorithm). Штучний інтелект на основі інформації про стан деякого об'єкту, що відслідковується, змінює траєкторію нашого об'єкта таким чином, щоб він рухався у напрямку до відстежуваногооб'єкту.

Рух може бути спрямовано як безпосередньо на відстежуваний об'єкт, так і (в більш реалістичної моделі) за деякою кривою, спрямованої до об'єкту (на зразок траєкторії самонавідної ракети), що показано на рис. 2.3.

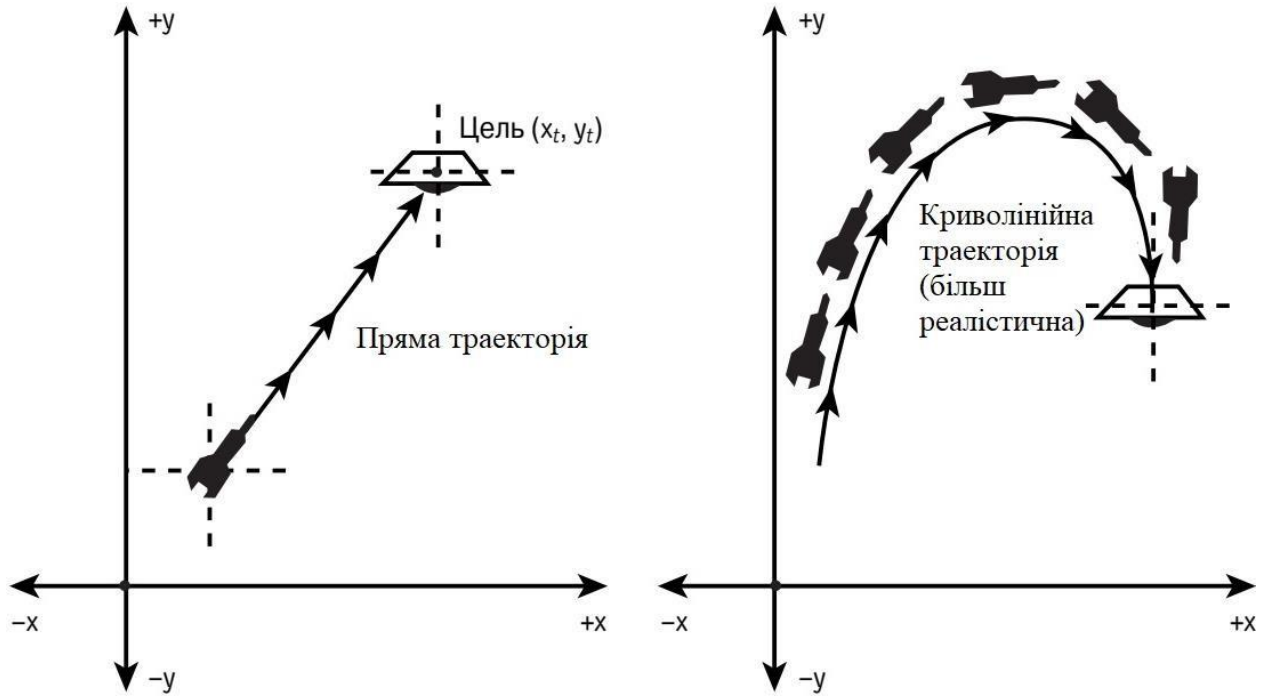


Рис. 2.3. Методи відстеження переміщення [43].

В якості грубого прикладу проходження за об'єктом можна навести такий алгоритм:

// Координати гравця - pLayer_x, player_y

// Координати персонажа - monster_x, monster_y

// Перевірка відносного розміщення по вісі X

if (pLayer_x> monster_x) ++ monster_x;

if (pLayer_x <monster_x) --monster_x;

// Перевірка відносного розміщення по вісі Y

if (player_y> monster_y) ++ monster_y;

if (player_y <monster_y) --monster_y;

Якщо використовувати цей алгоритм ШІ в найпростішій демонстраційній програмі, то отримується такий собі Термінатор, який постійно і неухильно прямує до мети. Наведений код дуже простий, але дуже ефективний.

Цей алгоритм проходження непоганий, але видається не дуже інтелектуальним; більш природним шляхом було б рух з урахуванням напрямку вектору від центру нашого об'єкта до центру об'єкта, за яким слідують (рис. 2.4).

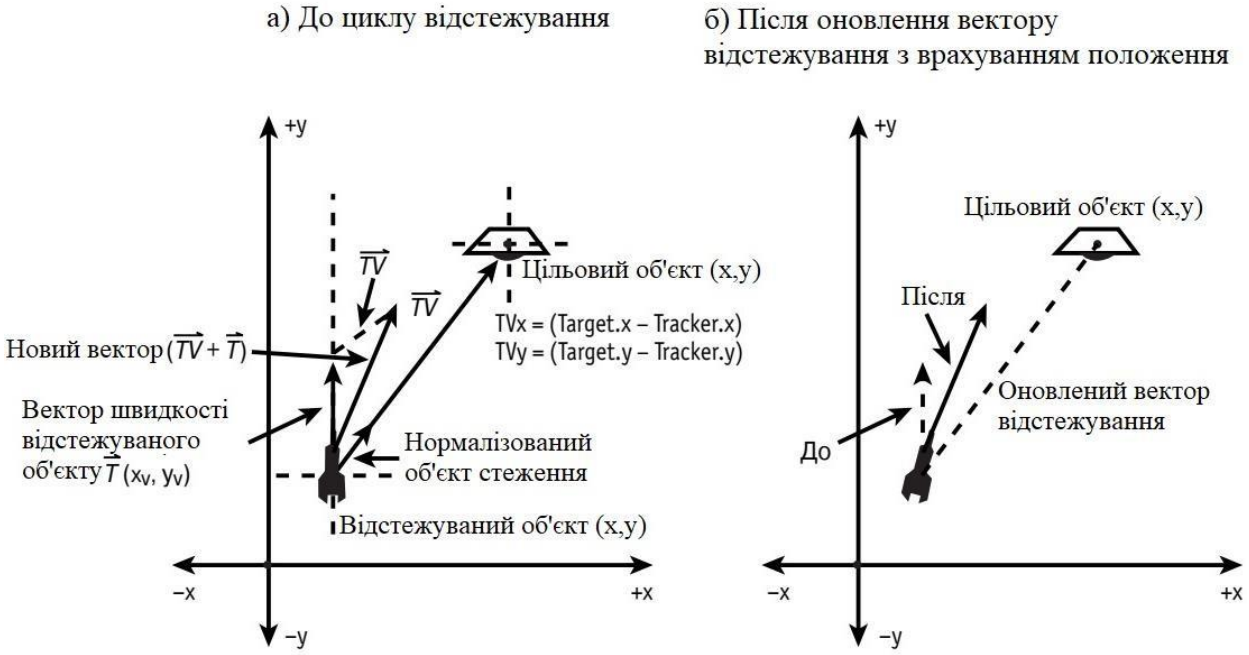


Рис. 2.4. Проходження за об'єктом з використанням вектора спостереження [44].

Даний алгоритм працює наступним чином. Нехай керований штучним інтелектом об'єкт називається tracker і має властивості:

Координати: (tracker.x, tracker.y)

Швидкість: (tracker.xv, tracker.yv)

і нехай відслідковується об'єктом буде target з властивостями:

Координати: (target.x, target.y)

Швидкість: (target.xv, target.yv)

Використовуючи дані визначення, алгоритм можна описати таким чином.

1. Обчислюється вектор TV від об'єкта до цілі:

TV = (target.x-tracker.x, target.y-tracker.y) = (tvx, tvy).

Потім здійснюється його нормалізація, тобто поділ на довжину. Позначити нормалізований вектор як TV *:

*TV * = (tvx, tvy) / sqrt (tvx * tvx + tvy * tvy).*

2. Змінюємо поточний вектор швидкості відслідковує об'єкта, додаючи до нього нормалізований вектор TV *, помножений на константу rate:

*tracker.xv + = rate * tvx;*

*tracker.yv + = rate * tvy;*

Якщо взяти значення rate, що перевищує 1.0, то зміна траєкторії відстежуваного об'єкта буде більш крутою, в результаті ціль буде наздогнана швидше [40].

3. Після зміни швидкості об'єкта слід перевірити, чи не перевищує вона максимального допустимого значення. Якщо це так, значення швидкості слід зменшити до прийнятного.

// Отримуємо значення швидкості об'єкта tspeed =

*sqrt (tracker.xxv * tracker.xv + tracker.yv * tracker.yv);*

// Швидкість занадто велика?

if (tspeed> MAX_SPEED) {

// Зменшуємо вектор швидкості

*tracker.xv * = 0.75;*

*tracker.yv * = 0.75;*

} // if

Можна використовувати і інші значення множника, наприклад 0.5 або 0.9, це не так важливо. Можна навіть вчинити зовсім чесно і просто обчислити необхідне

значення коефіцієнту, яке забезпечить необхідне максимальне значення швидкості після множення.

У разі переміщення в середовищі, яка взаємодіє з об'єктом (повітря, вода, гравітаційне поле і т.п.), слід змінювати використовувану фізичну модель [44].

2.1.3. Алгоритм ухилення від об'єкту

Тепер розглянемо алгоритм ШІ, який дозволяє об'єкту тікати від спостерігача. Цей алгоритм нітрохи не складніше алгоритму проходження за об'єктом; по суті, просто потрібно змінити знак швидкості на протилежний.

```
// Координати гравця - player_x, player_y
// Координати персонажа - monster_x, monster_y
// Перевірка відносного розміщення по вісі X
if (player_x> monster_x) --monster_x;
if (player_x <monster_x) ++ monster_x;
// Перевірка відносного розміщення по вісі Y
if (player_y> monster_y) --monster_y;
if (player_y <monster_y) ++ monster_y;
```

2.1.4. Шаблони і сценарії

Детерміновані алгоритми цілком достатні для вирішення множини завдань, проте досить часто потрібно створити об'єкт, який діє за певним сценарію. Наприклад, ось приблизний сценарій дій користувача при поїздці в автомобілі [45].

1. Вийняти ключі з кишені.
2. Відкрити дверцята машини за допомогою ключа.
3. Сісти в автомобіль.
4. Закрийте дверцята.
5. Вставити ключ в замок запалювання.
6. Повернути ключ.
7. Завести двигун.

Словом, слід виконати цілий ряд послідовних дій, що повторюються кожен раз при необхідності кудись їхати. Звичайно, якщо щось піде не так, послідовність може бути змінена: наприклад, якщо залишили автомобіль в гаражі не замкненим або якщо сів акумулятор. Шаблони являють собою важливу частину інтелектуальної поведінки, включаючи поведінку, властиве вінцям творіння на нашій планеті.

Створення шаблону для об'єкта гри може виявитися як складним, так і дуже простим, в залежності від того, для якого об'єкта створюється шаблон. Наприклад, шаблон для управління рухом реалізується дуже просто. Наприклад, розробляється гра типу Phoenix або Galaxian. Чужі рухаються по певній траєкторії і в деякий момент атакують, слідуючи деякому шаблону атаки. Штучний інтелект такого роду може використовувати множину різних технологій, але особисто одним з найпростіших є метод, заснований на трансляції інструкцій руху, як показано на рис. 2.5.

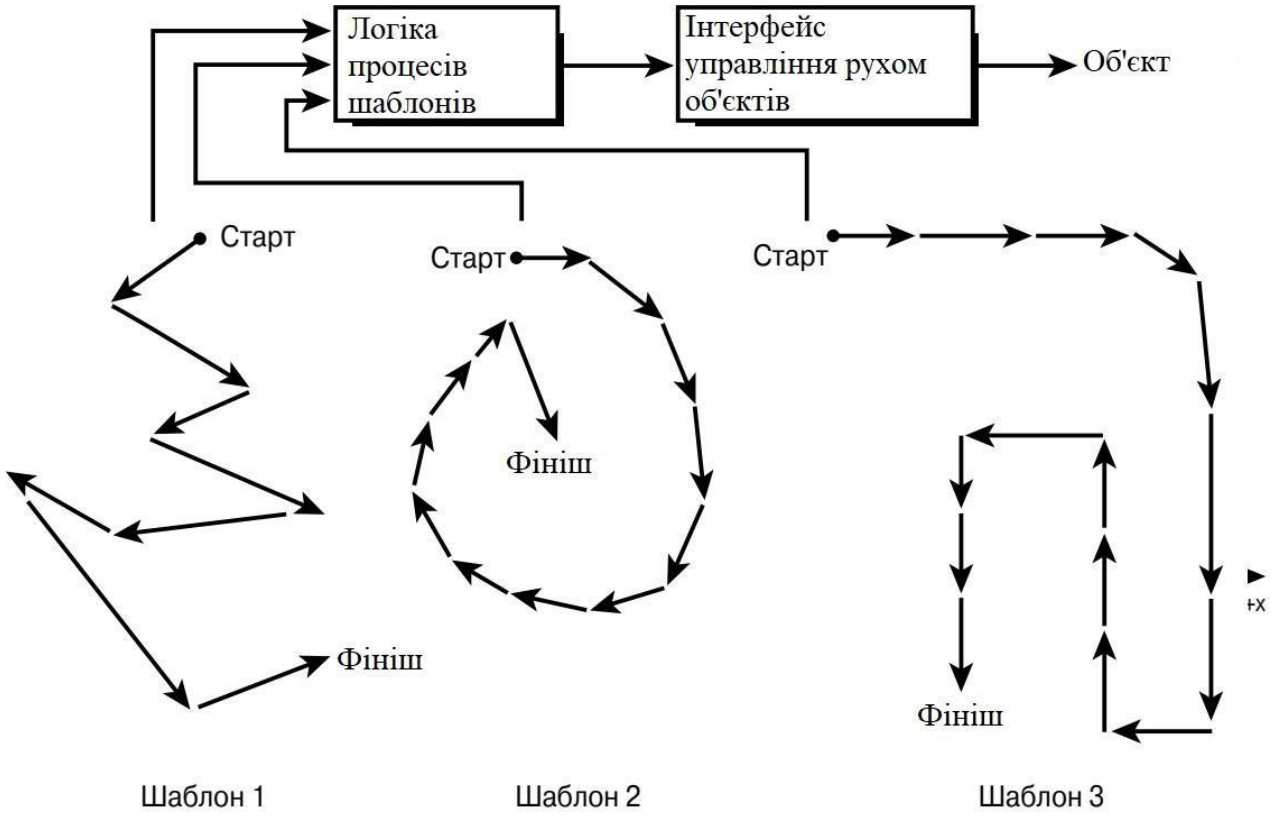


Рис. 2.5. Робота з шаблонами [40].

Кожен шаблон руху зберігається у вигляді послідовності інструкцій, наприклад, наведених в табл. 2.1.

Разом з кожної інструкцією можуть використовуватися додаткові дані, що уточнюють її, наприклад вказують, на яку відстань слід переміститися. У результате формат інструкції мови шаблону можна записати в такий спосіб:

Інструкція Операнд

Інструкція є один з кодів з наведеного вище списку, а Операнд - числове значення, яке дозволяє уточнити викликається інструкцією поведінки об'єкту. Використовуючи цей найпростіший формат інструкції, можна створити програму (послідовність інструкцій), що визначає шаблон. Потім створюється транслятор, який в процесі гри виконує розбір шаблону і відповідно управляє діями персонажа гри.

Нехай, наприклад, в даній мові шаблону інструкція являє собою пару чисел, перше з яких визначає дію, а друге - його тривалість у циклах. Створимо тривіальний шаблон, який відповідає руху по квадрату, як показано на рис. 2.6.

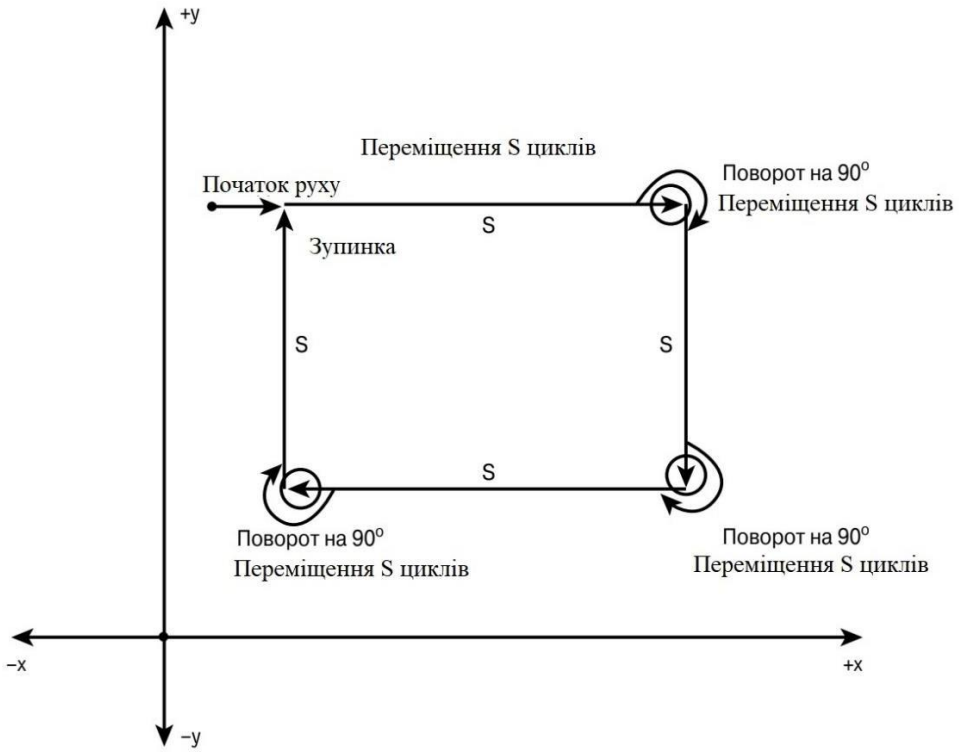


Рис. 2.6. Шаблон руху по квадрату [46].

int num_instructions = 9; // Кількість інструкцій в шаблоні

// Програма шаблону int square_stop_spin [] = {

1, 30, 3, 1, // Рух вперед і поворот вправо

1, 30, 3, 1, // Рух вперед і поворот вправо

1, 30, 3, 1, // Рух вперед і поворот вправо

1, 30, // Рух вперед; квадрат замкнутий

6, 60, // Зупинка на 60 циклів

4, 8}; // Поворот протягом 8 циклів

Таблиця 2.1. Гіпотетичний набір інструкцій шаблону [40].

Інструкція	Значення
GO_FORWARD	1
GO_BACKWARD	2
TURN_RIGHT_90	3
TURN_LEFN_90	4
SELECT_RANDOM_DIRECTION	5
STOP	6

Все, що потрібно для трансляції цієї програми, - великий оператор switch (), інтерпретуючий кожну інструкцію, і виводить відповідні вказівки об'єкту. Ось приклад такого транслятора.

// Вказує на першу інструкцію.

// Кожна інструкція складається з двох цілих чисел.

int instruction_ptr = 0;

// Отримуємо число циклів

int cycles = square_stop_spin [instruction_ptr + 1];

// Обробляємо інструкцію

switch (square_stop_spin [instruction_ptr]) {

```
case GO_FORWARD: // Переміщаємо об'єкт вперед
break;
case GO_BACKWARD: // Переміщаємо об'єкт назад
break;
case TURN_RIGHT_90: // Поворот об'єкта вправо
break;
case TURN_LEFT_90: // Поворот об'єкта вліво
break;
case SELECT_RANDOM_DIRECTION:
// Вибір випадкового напрямку
break;
case STOP: // Зупинити об'єкт
break;
} // switch
// Переміщаємо покажчик інструкцій (на 2 числа) instruction_ptr + = 2;
// Перевіряємо, не дісталися чи до кінця програми
if (instruct'on_ptr> num_instructions * 2)
{// Програма завершена}
```

У всіх шаблонів є одна тонкість - розумність руху. Об'єкт в грі твердо слідує заданим шаблоном, що в умовах гри, яка змінюються, не завжди розсудливо. Тому у проектованій системі ШІ повинний бути зворотний зв'язок, що з'ясовує, чи не відбувається щось некоректне, нерозумне або зовсім неможливе, і дозволяє вчасно перейти до іншого шаблону або стратегії в цілому, як показано на рис. 2.7.

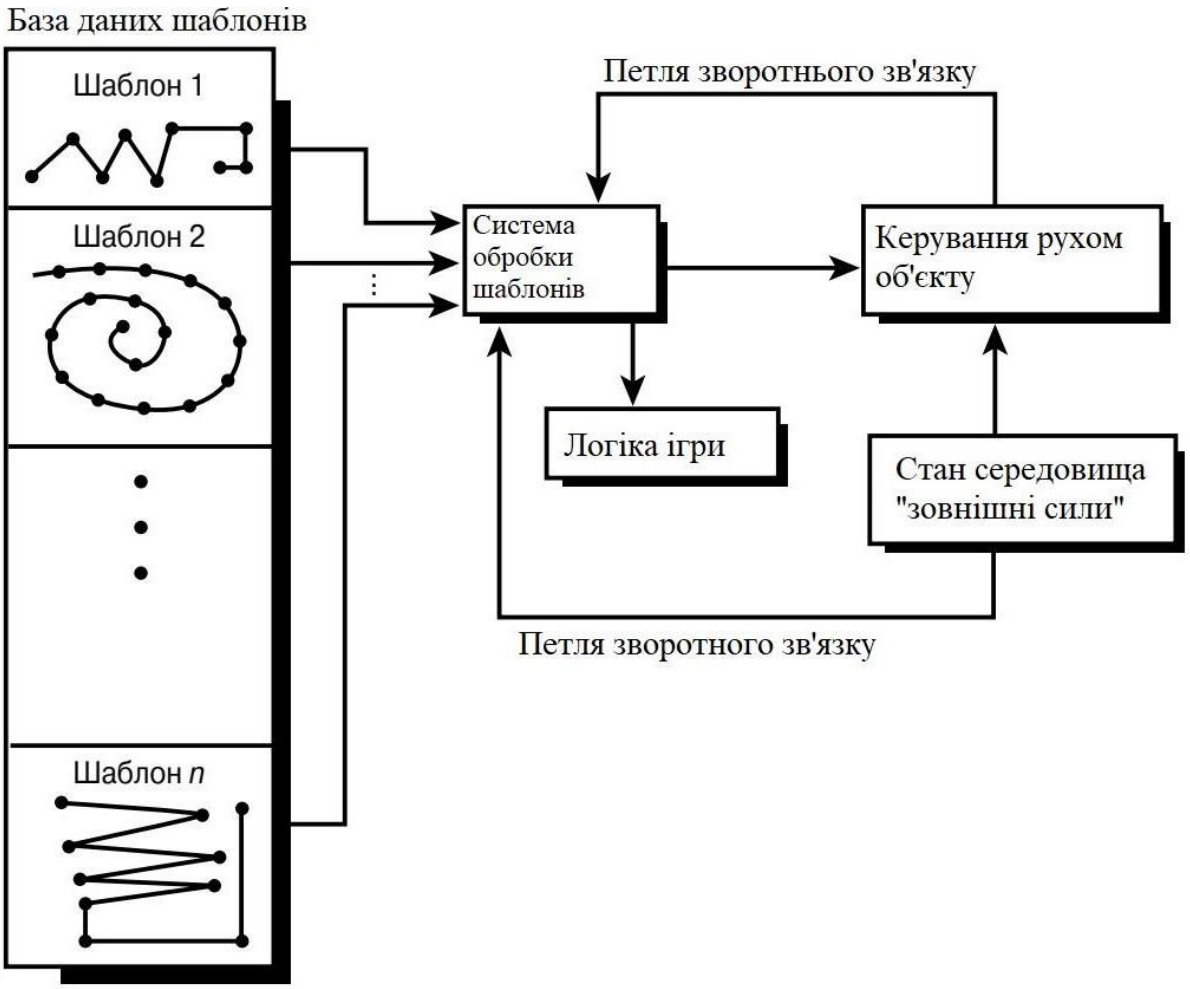


Рис. 2.7. Штучний інтелект зі зворотним зв'язком [40].

Зрозуміло, можна використовувати не масив, а ті структури даних, які в більшій мірі підходять для завдання. Наприклад, можна скористатися класом або структурою, які містять список записів в форматі [ІНСТРУКЦІЯ, Операнд] разом з кількістю інструкцій. При такому способі зберігання шаблонів можна легко створити їх масив, а потім вибирати з нього необхідний шаблон для трансляції та виконання.

Використовуючи шаблони, можна записати сотні траєкторій. Складні переміщення об'єктів, які практично неможливо здійснити за допомогою ШІ іншого типу за прийнятний час, створюються за допомогою найпростішого інструментарію (який нічого не варто написати для кожного конкретного випадку) всього за кілька хвилин, після чого записуються до файлу і відтворюються в грі.

Застосування шаблонів здатне зробити так, що персонажі проектованої гри будуть виглядати і надходити виключно розумно - нітрохи не гірше самого гравця. Саме цей метод був використаний в таких іграх, як Dead of Alive, Tekken, Soul Blade, Mortal Kombat і ін.

Крім того, застосування шаблонів не обмежується тільки шаблонами руху. Можна використовувати шаблони і для інших цілей, наприклад вибору озброєння, управління анімацією і т.п. - список можливих застосувань обмежується тільки фантазією розробника.

2.2. Метод кінцевих автоматів

Далі розглядаються кінцеві автомати, які називають також машинами станів, англійською – finite state machine (FSM). Мета - формалізувати застосування кінцевих автоматів при розробці систем ШІ в ігрових програмах.

Для створення стійкого кінцевого автомата важливо [47]:

• розумна кількість станів, кожний з яких представляє різні цілі або мотиви;

• вхідна інформація для кінцевого автомата, така, як стан середовища та інших об'єктів у ньому.

Вимога розумної кількості станів досить просто і зрозуміло. У нас, людей, є сотні, якщо не тисячі, емоційних станів, і в кожному з них - безліч підстанів.

А персонаж гри повинен всього лише виглядати розумним, так що і станів у нього не повинно бути надто багато. Наприклад, простий персонаж гри може мати декілька станів.

• Стан 1: Рухатися вперед.

• Стан 2: Рухатися назад.

• Стан 3: Поворот.

• Стан 4: Зупинка.

• Стан 5: Стрільба.

• Стан 6: Погоня за гравцем.

• Стан 7: Втеча від гравця.

Стану 1-4 прості і очевидні, однак для коректного моделювання поведінки персонажа в станах 5-7 можуть знадобитися додаткові підстани. Наприклад, гонитва за гравцем може включати в себе рух вперед і повороти. На рис. 2.9 схематично проілюстрована концепція підстанів. Однак не слід вважати, що підстани можуть бути засновані тільки на станах, що вже є; підстани можуть бути штучно введені для кожного з розглядуваних станів.

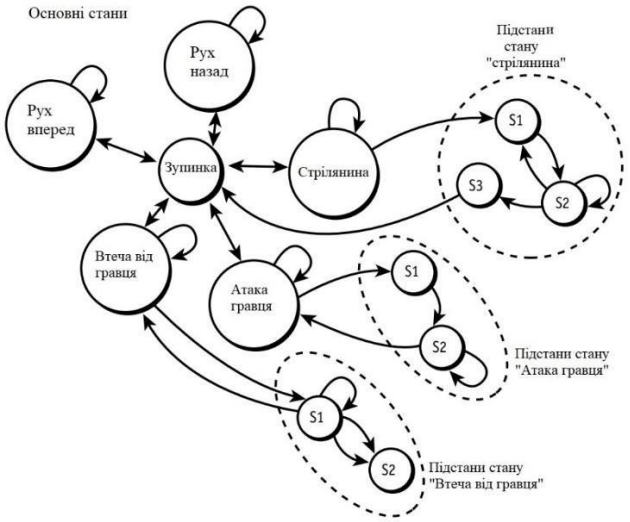


Рис. 2.9. Кінцевий автомат з підстанів [48].

Висновок з цього обговорення такий: станів не повинно бути занадто багато, інакше буде просто неможливо написати відповідну програму, і в той же час не повинно бути занадто мало, інакше об'єкт не буде здаватися досить розумним. Якщо в об'єкта тільки два стани, наприклад стояти і рухатися вперед, то важко домогтися враження розумності такого об'єкта. Станів повинно бути рівно стільки, скільки потрібно для отримання бажаного ефекту розумності, і жодним більше.

Що стосується другої вимоги, то вам необхідно забезпечити отримання інформації від інших об'єктів в грі, від гравця і ігрового середовища. Якщо об'єкт просто входить до деякого стану і знаходиться в ньому до кінця гри, це не штучний інтелект, а штучна дурість. Стан могло бути вибрано цілком розумно, але це було

100 мілісекунд тому. З того часу ігрова обстановка істотно змінилась, а гравець виконав ряд дій, на які потрібно дати розумну відповідь. Кінцевий автомат повинен відслідковувати стан гри і мати можливість у разі потреби вийти з поточного стану і увійти в інше.

Взявши до уваги все викладене, можна розробити кінцевий автомат, який добре моделює розумну поведінку об'єкту. Далі розглядаються деякі конкретні приклади, починаючи з найпростішого кінцевого автомата [48].

2.2.1. Елементарні кінцеві автомати

Важливо відмітити, що шаблони являють собою не що інше, як найпростіші кінцеві автомати. Для кращого розуміння розглядається об'єкт з простою поведінкою, яка і змоделюється за допомогою кінцевого автомату.

Більшість ігор засновані на деякому конфлікті. Чи є конфлікт базовою ідеєю гри або всього лише її фоном - в будь-якому випадку велику частину часу гравець буде шукати і знищувати ворогів або збирати різні речі. Потрібно розробити найпростішу модель поведінки персонажа гри, яка дозволить йому вижити під постійними атаками гравця. На рис. 2.10 показані взаємозв’язки між кількома станами.

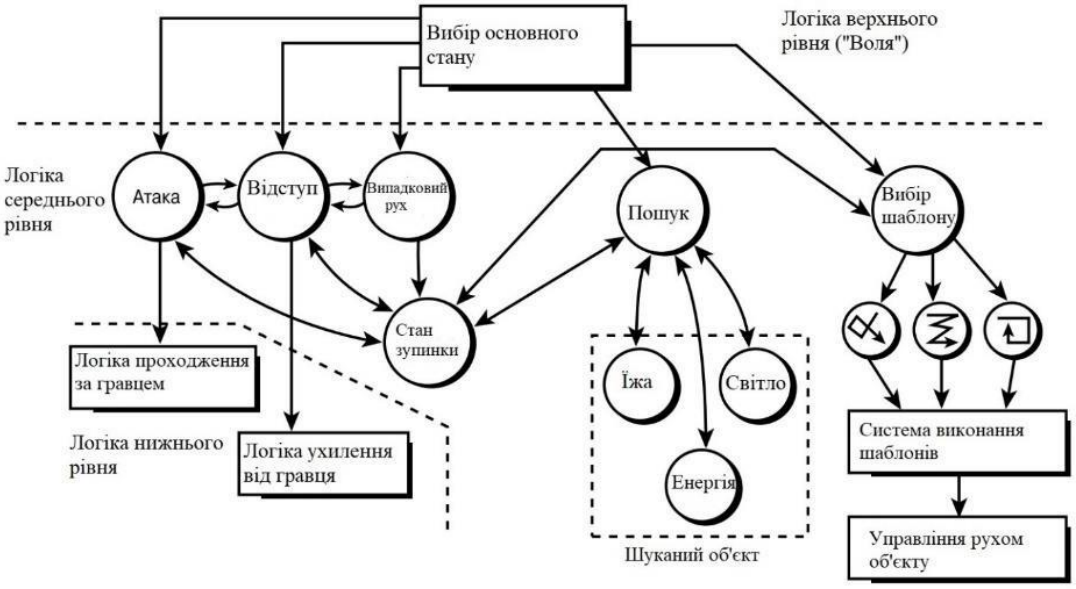


Рис. 2.10. Побудова поведінкової моделі об'єкту [48].

Основний стан 1: Атака. Основний стан 2: Відступ. Основний стан 3: Випадкове рух. Основний стан 4: Зупинка або тимчасова пауза.

Основний стан 5: Пошук спорядження (їжі, енергії, інших персонажів і т.п.). Основний стан 6: Вибір шаблону і дотримання його.

Є очевидною різниця між цими станами і попередніми прикладами. Перераховані стани функціонують на самому верхньому рівні і повинні містити підстани або логіку для їх створення "на льоту". Наприклад, стани 1 і 2 можуть бути реалізовані з використанням деяких детермінованих алгоритмів, в той час як стани 3 і 4 являють собою всього лише кілька рядків коду. Стан 6 виявляється дуже складним, оскільки, в свою чергу, передбачає виконання складних шаблонів.

Стан 5 також може являти собою складний детермінований алгоритм (або навіть набір таких алгоритмів і заздалегідь створених шаблонів пошуку). Головна думка в тому, що створюється модель персонажа "зверху вниз", тобто спочатку розробляється штучний інтелект на верхньому рівні, а потім поступово спускаючись вниз, реалізуючи все більш низькі рівні.

Повертаючись до рис. 2.10, можна бачити, що вибір основних станів персонажа відбувається логікою верхнього рівня, яку можна назвати "волею" або "Програмою" персонажа. Є безліч способів реалізації цієї логіки: випадковий вибір, вибір з використанням умовних операторів і ін. Поки досить знати, що стани верхнього рівня повинні вибиратися розумним способом, що враховують поточний стан гри [40].

Розглядається для кращого розуміння фрагмент коду, який реалізує кінцевий автомат у першому наближенні. Даний код нефункціональний - це всього лише начерк, що містить всі важливі структурні елементи. Повністю система ШІ такого рівня містить багато сторінок коду. Ось код, при створенні якого передбачалося, що в грі беруть участь тільки гравець і персонаж, керований системою ШІ.

```
// Основні стану
#define STATE_ATTACK 0 // Атака гравця
#define STATE_RETREAT 1 // Відступ
```

```
#define STATE_RANDOM 2 // Випадкове рух
#define STATE_STOP 3 // Зупинка на час
#define STATE_SEARCH 4 // Пошук енергії
#define STATE_PATTERN 5 // Вибір і виконання шаблону
// Змінні для опису персонажа
int creature_state = STATE_STOP,
// Стан персонажа
creature_counter = 0,
// Відстеження часу
creature_x = 320,
// Положення персонаже
creature_y = 200,
creature_dx = 0,
// Поточна траєкторія
creature_dy = 0;
// Змінні для опису положення гравця
int pLayer_x = 10, pLayer_y = 20;
// Реалізація логіки персонажа
// Обробка поточного стану
switch (creature_state)
{
case STATE_ATTACK:
{
// Перший крок: рух в напрямку гравця
if (pLayer_x> creature_x) creature_x ++;
if (pLayer_x <creature_x) creature_x--;
if (player_y> creature_y) creature_y ++;
if (player_y <creature_y) creature_y--;
// Другий крок: стрілянина з ймовірністю 20%
```

```
if ((rand ()% 5) == 1) Fire_Cannon ();
} Break;
case STATE_RETREAT:
{
// Рух від гравця
if (pLayer_x> creature_x) creature_x--;
if (pLayer_x <creature_x) creature_x ++;
if (player_y> creature_y) creature_y--;
if (player_y <creature_y) creature_y ++; }
Break;
case STATE_RANDOM:
{
// Переміщення персонажа у випадковому
// напрямку, яке було визначено
// при вході до цього стану
creature_x + = creature_dx; creature_y + = creature_dy; } Break;
case STATE_STOP:
{
// Нічого не робимо
} Break;
case STATE_SEARCH:
{
// Вибираємо об'єкт для пошуку і рухаємося по
// напрямку до нього
if (energy_x> creature_x) creature_x ++;
if (energy_x <creature_x) creature_x--;
if (energy_y> creature_y) creature_y ++;
if (energy_y <creature_y) creature_y--;
} Break;
```

```
case STATE_PATTERN:
{
// Виконуємо обраний шаблон Process_Pattern (); } Break;
default: break;
} // switch
// Оновлюємо вміст лічильника і перевіряємо, чи не варто
// здійснити вибір нового стану
if (--creature_counter <= 0)
{
// Вибираємо новий стан, застосовуючи ту чи іншу
// логіку. В даному випадку використовується випадковий вибір
creature_state = rand ()% 6;
// В залежності від обраного стану слід
// виконати деякі налаштування
if (creature_state == STATE_RANDOM)
{
// Вибір випадкового напрямки
creature_dx = -4 + rand ()% 8;
creature_dy = -4 + rand ()% 8; }
// if
// У разі необхідності - інші налагоджувальні дії
// Вибір часу для роботи в даному стані. Виходячи // з 30 кадрів в секунду, певні години - від 1 до 5
// секунд
creature_counter = 30 * (1 + rand ()% 5); }
// if
```

Наведений код починається з обробки поточного стану і включає до себе локальну логіку, алгоритми і навіть виклики функцій інших підсистем ШІ, наприклад для виконання шаблону. Після обробки поточного стану оновлюється

вміст лічильника і, якщо виконання поточного стану завершено, вибирається новий стан. Якщо для його роботи потрібні попередні налаштування, вони тут же виконуються. І нарешті, вибирається тривалість перебування персонажа в новому стані [41].

Цей нарис коду не просто можна, а потрібно значно поліпшити. В першу чергу це стосується переходів між станами, які можуть здійснюватися разом з обробкою поточного стану, і вибору нового стану, яке можна і потрібно здійснювати на основі інформації про поточний стан гри, а не наосліп, випадковим чином.

2.2.2. Додавання індивідуальності

Індивідуальність, по суті, не що інше, як передбачуваність поведінки того чи іншого персонажа. Наприклад, у людини є різні знайомі. Серед них є один малий чималого розміру і з гарячим характером, і якщо хтось (особливо в барі після кількох келихів пива) скаже йому щось, що йому не сподобається, то ризикує отримати чималий рахунок від стоматолога, який буде чинити щелепу. Інший знайомий в такій же ситуації просто промовчить і зробить вигляд, що нічого не розчув.

Звичайно, люди набагато більш складні і менш передбачувані, ніж в даному конкретному прикладі, але для персонажів гри можна обійтися простою моделлю розподілу ймовірностей. Це означає, що у кожного персонажа свої ймовірності вибору тієї чи іншої поведінки в різних ситуаціях; набір цих ймовірностей, по суті, і визначає їх індивідуальність, беручи участь у виборі переходу між станами (рис. 2.11).

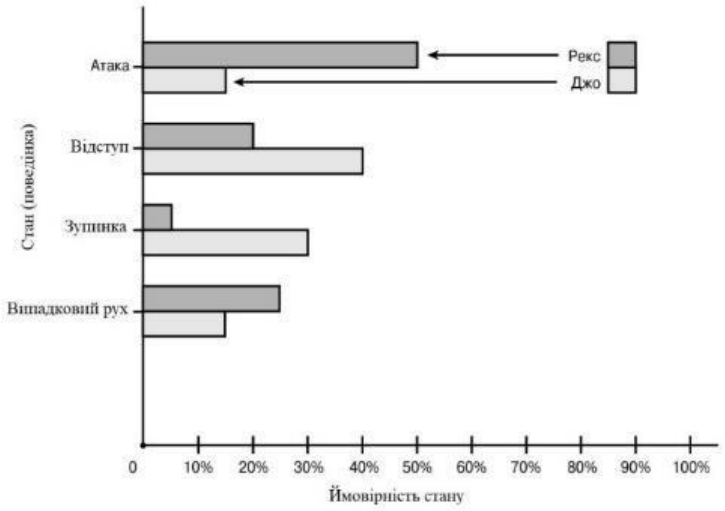


Рис. 2.11. Розподіл ймовірностей основних станів різних персонажів гри [44].

Нехай в обраній моделі є чотири основні стани.

Стан 1: Атака. Стан 2: Відступ. Стан 3: Зупинка. Стан 4: Випадкове рух.

Замість рівноймовірного вибору нового стану, як було раніше, створюється та використовується для кожного персонажа розподіл ймовірностей, що визначає його індивідуальність. Наприклад, в табл. 2.2 наведені розподілу ймовірностей для персонажів Рекса і Джо, що імітують друзів, описаних раніше.

Як бачите, така модель не позбавлена сенсу. Якщо застосувати наведене розподіл ймовірностей, Рекс буде бездумно кидатися до атаки, в той час як Джо надає перевагу ухилянню від зустрічі з супротивником; крім того, випадкового в його поведінці буде менше, ніж у Рекса, від якого в реальному житті можна очікувати чого завгодно.

Зрозуміло, цей приклад повністю придуманий, але якщо уявити собі персонажа, чия поведінка визначається наведеними в табл. 2.2 і на рис. 2.11 розподілами ймовірності, то картина буде приблизно такою, як вона описана.

Таблиця 2.2. Індивідуальні розподілу ймовірностей [40].

Стан	Імовірність для Рекса,%	Імовірність для Джо,%
Атака	50	15
Відступ	20	40
Зупинка	5	30
Випадковий рух	25	15

Для реалізації описаного методу можна, наприклад, скористатися таблицею з 20-50 записами (де кожен запис відповідає тому або іншому стану) і заповнити її відповідно до необхідного розподілу ймовірностей. Наприклад, ось як будуть виглядати 20-елементні таблиці станів для Рекса і Джо (кожен елемент в такій таблиці має вагу 5%):

int rex_pers [20] = {1,1,1,1,1,1,1,1,1,1,2,2,2,2,3,4,4,4,4,4};

int joe_pers [20] = {1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,4,4,4};

При розробці систем ШІ до використання індивідуальності персонажів можна додати радіус впливу. Це означає, що розподіл ймовірностей персонажа залежить від деякого фактору, наприклад відстані до гравця або якогось іншого об'єкта (рис. 2.12). Як бачите, коли персонаж знаходиться досить далеко від гравця, він переключається на неагресивний пошук; якщо гравець наближається до гравця, той постарається уникнути зустрічі, а якщо гравець підійде занадто близько, спробує атакувати його. Як бачите, в кожному діапазоні відстаней використовується своя таблиця розподілу ймовірностей.

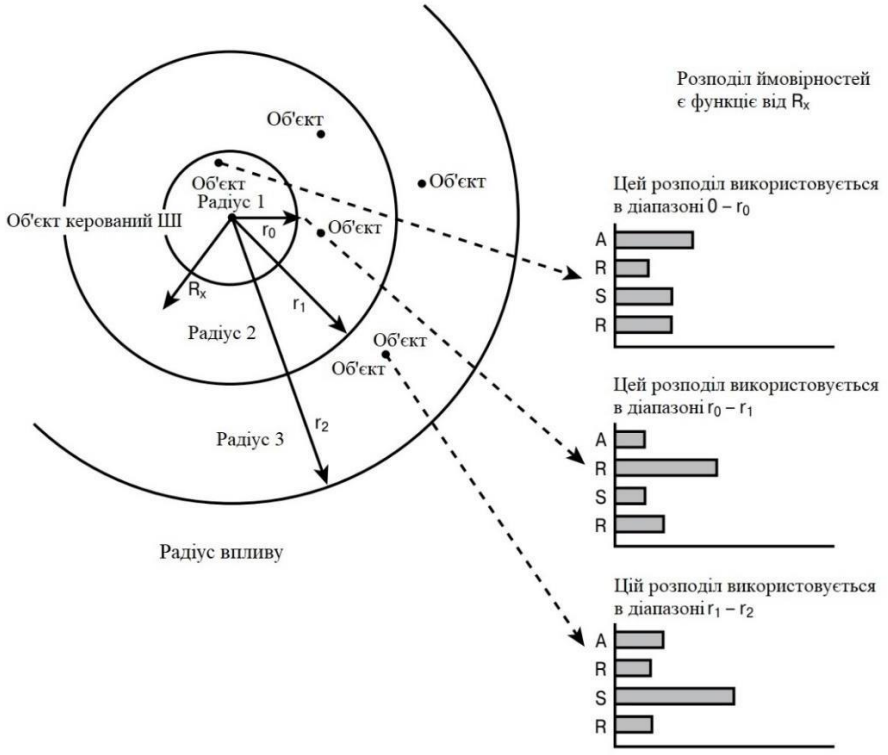


Рис. 2.12. Зміна розподілу ймовірностей в залежності від відстані [48].

2.2.3. Запам'ятовування і навчання

Ще одним важливим елементом гарної системи ШІ є запам'ятовування і навчання. Всі розглянуті технології ШІ працюють тільки з поточною інформацією і ніколи не враховують події, що вже відбулися.

Наприклад, якщо при атаці з боку персонажа гравець постійно йде від атаки вправо, ні одна з розглянутих технологій ШІ не в змозі відстежити це і вжити попереджувальну атаку з урахуванням даної особливості гравця.

Як інший приклад можна уявити, що персонажі гри повинні шукати боєприпаси точно так же, як і гравець. Однак персонажі вели б себе набагато розумніше, якби запам’ятовували місце, де боєприпаси були знайдені в останній раз, і починали пошук саме звідти, а не шукали б їх випадковим чином (можливо, з використанням шаблонів).

Це тільки кілька прикладів, які демонструють, наскільки розумніше виглядали б персонажі, якби мали можливість запам'ятовувати і навчатися. На щастя, реалізувати запам’ятовування нескладно, хоча тільки дуже мало хто з

програмістів роблять це. Зазвичай їм просто не вистачає часу (або вони вважають, що гра і без цього вийшла непогано). А дарма! Запам'ятовування і навчання - дуже красиві речі, і гравці відразу помітять різницю між грою з використанням цих можливостей і без них. Спробуйте знайти, де у певній грі можна реалізувати найпростіше запам'ятовування і навчання штучного інтелекту без особливих складнощів, і подивіться, наскільки при цьому зміниться гра.

Як саме можна реалізувати описану методику? Це залежить від конкретної гри. Розглянемо, наприклад, рис. 2.13.

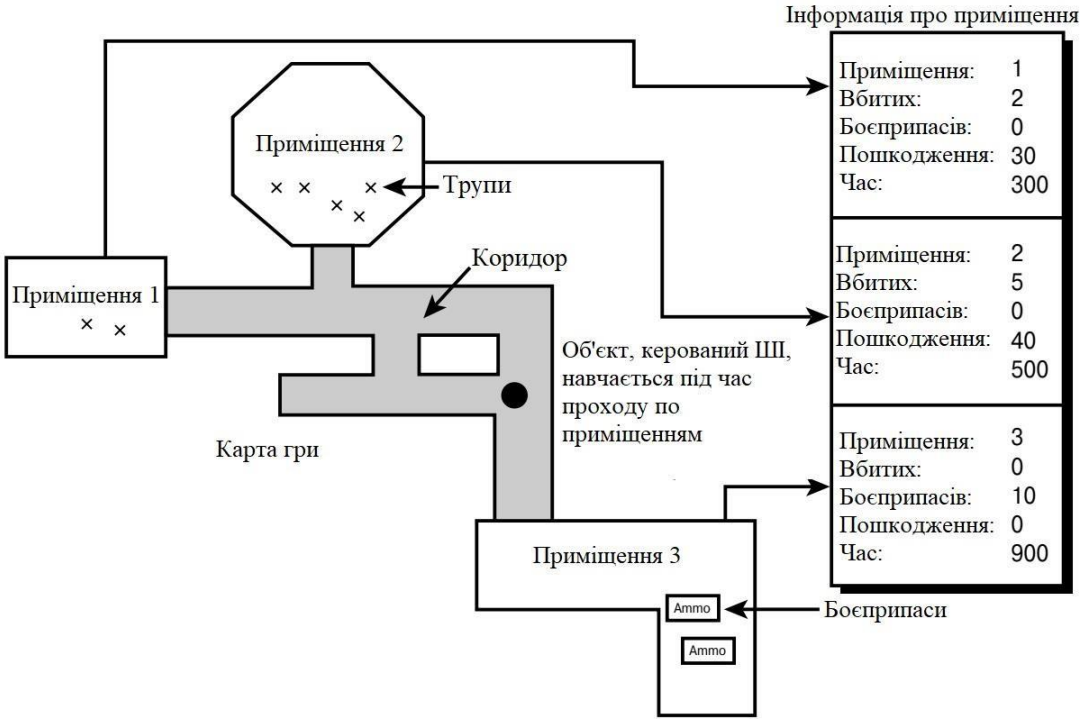


Рис. 2.13. Використання пам'яті в системі штучного інтелекту [44].

На ньому можна бачите карту ігрового світу, де кожному приміщенню відповідає запис, що зберігає наступну інформацію [48]:

Убиті персонажі

Пошкодження, завдані гравцем

Виявлені боєприпаси

Час перебування в даному приміщенні

Всякий раз, коли настає час дії персонажа, в процесі вибору чергового стану можна звернутися до інформації, що зберігається в запису і являє собою не що інше, як пам'ять персонажу. Наприклад, виявивши, що в деякому приміщенні абсолютно немає боєприпасів, зате в ньому гравець завдав персонажу великої шкоди, персонаж може тут же спробувати перейти до іншого приміщення [40].

Інший приклад використання пам'яті штучним інтелектом. Якщо у персонажу закінчуються боєприпаси, то набагато розумніше не починати пошук наосліп, тобто випадковим чином, а просканувати всі записи і "згадати", в якому приміщенні залишилися незаймані боєприпаси.

Крім того, персонажі можуть обмінюватися інформацією! Наприклад, якщо один персонаж зустрічається з іншим, вони можуть об'єднати свої записи про приміщення, в яких побували. Можна реалізувати обмін не лише інформацією, а й, наприклад, зброєю, боєприпасами, іншим спорядженням. Ще один варіант обміну інформацією, коли поранений в бою з гравцем персонаж розповідає іншому, свіжому і озброєному до зубів, де саме тільки що знаходився гравець, а заодно передає йому частину свого озброєння; сам же він після цього йде відновлювати сили і боєзапас (можливо, використовуючи для цього тільки що отриману інформацію).

Немає межі способам застосування запам'ятовування і навчання. Єдине, про що слід сказати окремо: штучний інтелект повинен грати чесно. Наприклад, було б нечесною грою забезпечити ШІ інформацією про всі мапі гри. Штучний інтелект повинен бути поставлений у ті ж умови, і грати за тими ж правилами, що і звичайний гравець.

2.2.4. Дерева планування і прийняття рішень

До сих пір всі технології ШІ були занадто реакційними і безпосередніми, вони не використовували ні планування, ні високорівневої логіки. Після огляду реалізації низькорівньового ШІ, можна поговорити про високорівневий штучний інтелект, про який зазвичай кажуть, як про планування.

План являє собою просто високорівневу множину дій, які виконуються в певному порядку для досягнення поставленої мети. Крім дій, в план входить ряд умов, які повинні задовольнятися перед тим, як буде виконано те чи інше конкретне дію. Наприклад, ось як може виглядати план походу до театру.

1. Вибрати п'єсу, яку б вам хотілося подивитися.

2. Відправитися в театр, як мінімум, за годину до початку вистави.

3. Опинившись в театрі, придбати квиток.

4. Переглянути п'єсу і відправитися додому.

Виглядає більш-менш розумно. Однак є багато суттєвих деталей. Наприклад, до якого саме театру потрібно їхати? Чи вистачить однієї години, щоб дістатися до театру? Що робити, якщо не вистачає грошей на квиток? Ну і так далі ... Ці деталі можуть бути важливі (або не важливі), в залежності від того, наскільки складний план будується. Однак взагалі кажучи, має бути стільки умов і підпланів, щоб при виконанні ШІ основного плану не виникало жодного питання [46].

Реалізація алгоритмів планування для штучного інтелекту ігрової програми грунтується на тій же концепції, що і раніше. Є об'єкт, керований системою ШІ, і хочеться, щоб він дотримувався деякого плану для досягнення певної мети. Отже, необхідно змоделювати цей план за допомогою деякого мови програмування; зазвичай це C / C ++, але можна використовувати і деякий спеціалізовану високорівневу мову сценаріїв. У будь-якому випадку, крім моделювання плану, треба змоделювати всі об'єкти, які є частиною плану: дії, мети і умови. Кожен з перерахованих елементів може бути представлений простою структурою або класом C / C ++. Наприклад, мета може виглядати наступним чином:

typedef struct GOAL_TYP

```
{
int class;
// Клас мети
char * name;
// Назва цілі
int time;
// Час, відпущений на досягнення мети
int * subgoals;
// Покажчик на список підцілей,
// які повинні бути досягнуті int (* eval) (void);
// Покажчик на функцію, що визначає,
// досягнута мета
// Інші дані}
GOAL, * GOAL_PTR;
```

Звичайно, це визначення - всього лише приклад, і в конкретному випадку полів може виявитися набагато більше; головне, щоб було зрозуміло ідею. Створюється структуру, яка може представляти будь-яку ціль у певній грі - від "підірвати міст" до "знайти їжу".

Наступна структура, що вимагається, це структура дії, яка являє щось, що об'єкт повинен зробити як частину плану по досягненню мети. Вибір конкретної структури - за розробником; знову ж, дана структура повинна дозволяти описувати все, що повинен вміти робити об'єкт. Ось приклад подібної структури [40]:

```
typedef struct ACTION_TYP
{
int class;
 // Клас дії
int * name;
// Назва дії
int time;
```

// Час, виділений для

*// виконання дії RESOURCE * resource;*

// Покажчик на запис,

// описує ресурси,

// які можуть знадобитися

*// при виконанні дії CONDITIONS * cond;*

// Покажчик на запис,

// описує всі умови,

// які повинні виконуватися до

// того, як дана дія

*// буде виконано UPDATES * update;*

// Покажчик на запис,

// описує всі оновлення і

// зміни, які повинні бути

// виконані по завершенні дії int (action_functions) (void);*

// Покажчик на функцію (і), яка

// виконує дану дію

*} ACTION, * ACTION_PTR;*

Зрозуміло, що це дуже абстрактний приклад. Очевидно, що у певній реалізації ця структура може бути зовсім іншою.

2.2.5. Кодування планів

Існує ряд способів кодування планів. Ваш код, який реалізує дії, цілі та план, може бути безпосередньо розроблений на мові C / C ++ і бути невід'ємною частиною програми. Така технологія "прошитого", або жорстко закодованого (hard coded) в програмі плану була широко поширена в минулому [43].

Більш елегантний метод кодування плану полягає в використанні правил продукції, або просто продукції, і дерев прийняття рішень. Продукція являє собою логічне твердження з рядом посилок і наслідком:

IF X OP Y THEN Z

Тут X і Y - посилки, Z - наслідок, а OP може бути будь-який логічною операцією, наприклад AND, OR і т.д. Крім того, X і Y можуть бути складені з інших продукцій, тобто продукції можуть бути вкладеними. Розглянемо, наприклад, наступний оператор [45]:

if (P> 20) AND (damage <100) THEN consequence

Таким чином, фактично продукція являє собою умовний вираз, а прошитий план є не чим іншим, як набором умовних операторів разом з діями і цілями. Мета розробки "планувальника" полягає в тому, щоб змоделювати поведінку об'єктів трохи абстрактніше. Хоча ніхто не заважає розробнику закодувати продукції безпосередньо, краще все ж створити структуру, яка може читати продукцію, містити дії і цілі і представляти план.

Однією зі структур, яка допомогла б в реалізації цієї системи, є дерево прийняття рішень. Як показано на рис. 2.14, це просте бінарне дерево, кожен вузол якого представляє продукцію і / або дію.

Однак замість використання жорсткого кодування для реалізації дерева завантажується його з файлу або будується з даних, наданих розробником. Цей спосіб загального призначення, на відміну від жорсткого кодування, не вимагає перекомпіляції програми при зміні дерева прийняття рішення. Як приклад розглядається невелика мову планування для управління бойовим роботом з використанням деяких вхідних змінних і набору дій.

Наведемо параметри, які можуть бути протестовані системою ШІ.

DISPLY Відстань до гравця (0 - 100)

FUEL Залишок пального (0 - 100)

AMO Залишок боєприпасів (0 - 100)

DAM Поточні пошкодження (0 - 100)

TMR Поточний час гри в віртуальних хвилинах

PLAYST Стан гравця (атакуючий, що не атакуючий)

Дії, які може виконувати робот.

FW Вогонь по гравцеві

SD Самознищення

SEARCH Пошук гравця

EVADE Ухилення від гравця

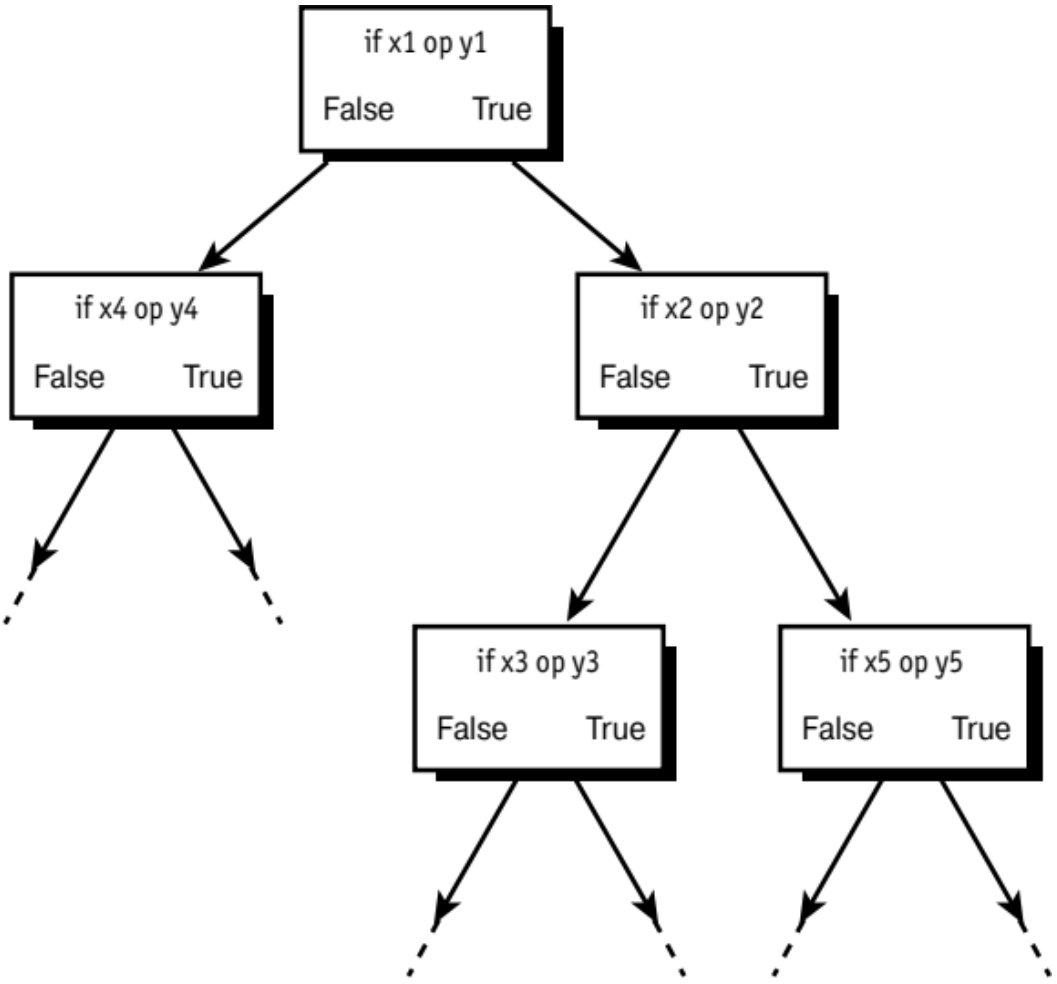


Рис. 2.14. Дерево прийняття рішень [48].

Тепер розробляється структуру дерева прийняття рішень. Можна вважати, що в кожному вузлі може бути перевірена одна посилка (або дві, пов'язані логічним оператором AND або OR), до якої може бути застосований оператор логічного заперечення NOT. Посилкі самі по собі являють порівняння вхідних змінних і констант за допомогою операторів »,«, == і! =. Крім того, кожен вузол має дві гілки

- TRUE і FALSE, а також список можливих дій (до восьми пунктів) (рис. 2.15). Нижче наведена структура, яка може бути використана для реалізації вузла.

```
typedef struct DECNODE_TYP {
int operand1, operand2;
int comp1;
int operator;
// Перша пара операторів
// Оператор порівняння
// Оператор об'єднання
int operand3, operand4;
// Друга пара операторів
int comp2;
// Оператор порівняння
ACTION * act_true; ACTION * act_false;
// Список дій при виконан –
// нні і невиконанні умови
DECNODE_PTR * dec_true;
// Гілки при виконанні і DECNODE_PTR * dec_false;
// невиконанні умови
} DECNODE, * DECNODE_PTR;
```

Як бачите, ця структура може реалізувати різні типи посилок, що складаються з одного або двох порівнянь, причому порівнюватися можуть як дві змінні, так і змінні з константами.

Програмна реалізація даної технології не становить особливих труднощів, так що я не буду детально на ній зупинятися. Просто скажу, що ви повинні слідувати по вузлам, обчислювати умовні вирази в них і, в залежності від отриманого результату, виконувати ті чи інші дії і переходи до інших вузлів дерева прийняття рішень [47].

Після цього потрібно лише розробити саме дерево прийняття рішень, яке і буде визначати поведінку робота в різних ситуаціях. Розглянемо, наприклад, систему управління вогнем. Зрозуміло, що наведений нижче план далеко не повний, але як приклад плану, що визначає тактику ведення вогню, його можна прийняти. Грубо його можна сформулювати російською мовою наступним чином.

• Якщо гравець близько і пошкодження малі, то атакувати гравця.

• Якщо гравець далеко і пального багато, то шукати гравця.

• Якщо пошкодження великі і гравець близько, то тікати від гравця.

• Якщо пошкодження великі, і боєприпасів немає, і гравець близько, то самознищитися.

Ось такий малий псевдоплан дій робота. Зрозуміло, що повний план може містити десятки або навіть сотні таких пропозицій. Але найприємніше в ньому, що дизайнер гри не повинен його кодувати, а може використовувати замість цього графічний інструментарій для побудови схеми на зразок показаної на рис. 2.15, яку потім легко перетворити в програму з використанням розробленого вами мови планування.

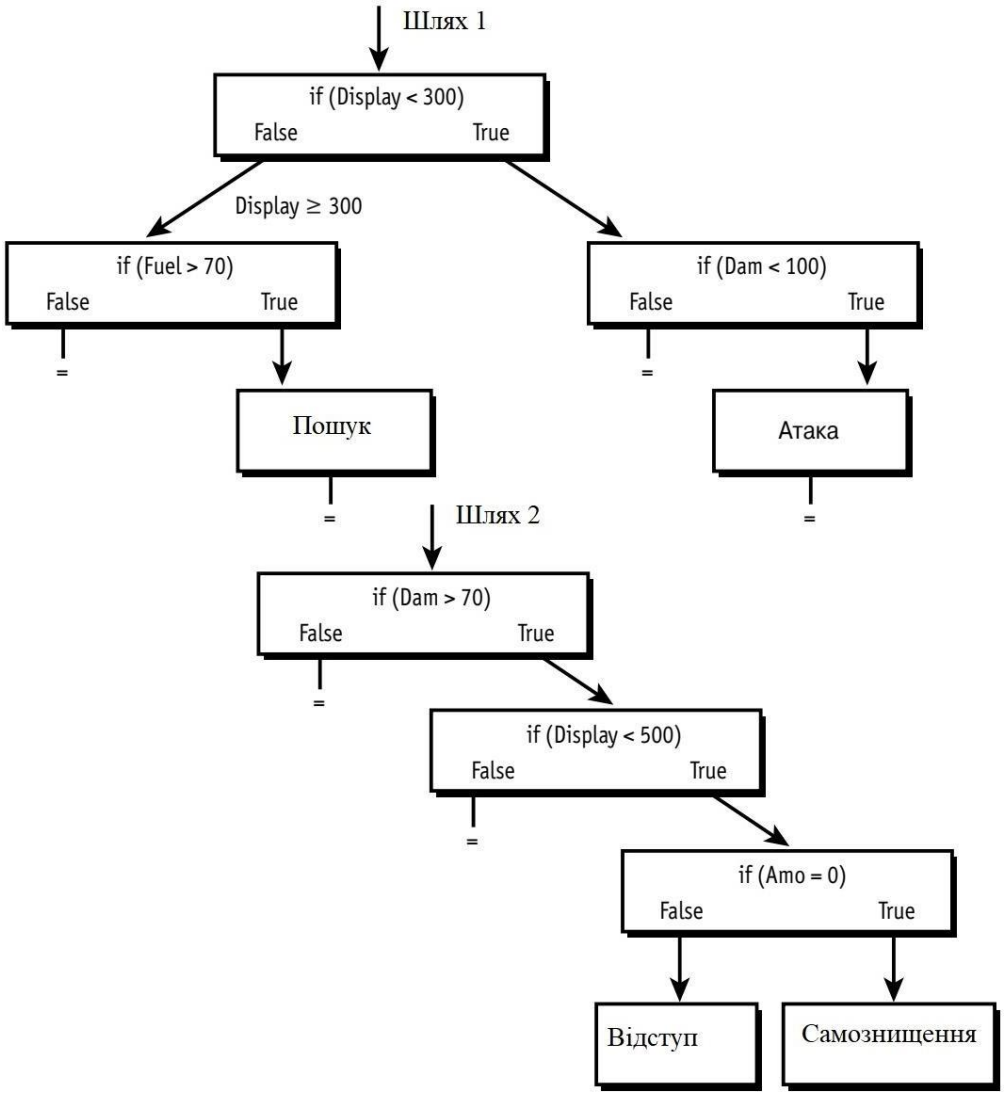


Рис. 2.15. Дерево прийняття рішень [48].

Отже, головне в використанні описаного методу - це поділ представлення плану і його обробки. Ви створюєте обробник плану, який рухається по гілках дерева прийняття рішень, обчислюючи умовні вирази і виконуючи необхідні дії. Завершимо розгляд формальним алгоритмом планування, який не тільки досягає поставленої мети, а й виконує аналіз плану.

2.2.6. Реалізація реальних планувальників

Вже познайомилися з тим, як реалізувати частину планування, що відповідає за обрахування умовних виразів, і частина, що виконує дії. Мета плану являє собою

просто формалізацію того факту, що кожен план повинен мати мету, яка досягається в процесі його виконання. Відповідно, коли виконання плану завершується, в обов'язковому порядку потрібно перевірка того, чи досягнута поставлена мета. Можлива також ситуація, коли поряд з основним планом є паралельно виконуються підплани, і тоді для досягнення мети основним планом потрібно досягнення цілей всіма підпланами [42].

Розглянемо глобальний план "Всі роботи зустрічаються в точці (x, y, z)". Ця мета не може бути досягнута, поки не буде досягнута мета "Переміститися до точки (x, y, z)" для кожного робота. Крім того, якщо один з роботів не в змозі досягти заданої точки, планувальник повинен виявити цю ситуацію і відреагувати на неї. В цьому і полягає ідея контролю і аналізу плану.

Сам по собі план може бути неявно представлений в дереві рішень або являти собою список рішень і дій, кожне з яких, в свою чергу, є деревом або послідовністю. Який спосіб представлення вибрати - вирішувати розробнику. Важливо лише, що повинна бути здатність сформулювати план, послідовність дій щодо його виконання і мета. Дії зазвичай включають до себе обчислення умовних виразів і виконання піддією на низькому рівні, наприклад таких, як переміщення з однієї точки до іншої або стрілянина. Термін дії на вищих рівнях логіки штучного інтелекту означає виконання завдання типу "Знищити гравця" або "Захопити зміцнення"; дії ж на нижньому рівні виконуються безпосередньо процесором гри.

Отже, в припущенні, що план являє собою якийсь масив або пов'язаний список, який можна обійти, планувальник може мати наступний вигляд:

whiLe (План не порожній, і мета не досягнута) {

Отримати чергова дія плану Виконати дію}

// while

Звичайно, треба розуміти, що це всього лише абстрактна реалізація планувальника. У реальній програмі вся ця робота повинна виконуватися паралельно з рішенням інших завдань. Зрозуміло, що не можна постійно перебувати в циклі while в очікуванні, поки не буде досягнута мета плану.

Планувальник слід реалізувати в вигляді кінцевого автомата або подібної структури і відстежувати стан плану під час роботи гри [46].

Даний алгоритм планування кілька "тупуватий". Він не бере до уваги, що в майбутньому дії можуть виявитися неможливі, а план - марний. В результаті планувальник повинен проводити додатковий аналіз плану і переконуватися, що він має сенс. Наприклад, якщо план складається у вибуху моста, але в ході його виконання міст виявляється підірваний кимось іншим, планувальник повинен зуміти зробити висновок про неможливості виконання плану і припинити його виконання. Це можна зробити, розглядаючи мети плану і перевіряючи, чи не досягнуті ці цілі іншими персонажами, і якщо так, то виконання такого плану слід припинити.

Планувальник повинен також розглядати події або стану, які можуть робити план неможливим. Наприклад, в певний момент при виконанні плану може знадобитися синій ключ, який був втрачений. Така ситуація також має відстежуватися планувальником, який переглядає план у контексті того, що може знадобитися в майбутньому, і який повинен вміти коректувати план з урахуванням зроблених висновків. Погодьтеся, що при виконанні плану "Пройти 1000кілометрів і підірвати форт" нерозумно буде пройти цю 1000 кілометрів і з'ясувати, що вибухівки-то у тебе немає! Це вже не штучний інтелект, а штучна дурість. Планувальник повинен розглядати мету, відстежити все необхідне для її виконання і переконатися, що персонаж,

І нарешті, коли план знаходиться в стадії виконання, не обов'язково повністю припинити його виконання. Можна змінити його або вибрати новий план на основі вже зробленого. Наприклад, можна спочатку мати, скажімо, три плани дій - один основний і два резервних на випадок, якщо виконання основного плану виявиться неможливим.

Планування - дуже потужний інструмент штучного інтелекту, використовуваний, по суті, в будь-якій грі. Найкращий спосіб використання планування в реальній розробці ігор - це розробити власну мову планування і дати

його дизайнеру для складання плану разом з набором змінних і об'єктів, які можуть бути його частиною. Це дозволить дизайнеру розробити такі далекосяжні плани, які ви не змогли б не тільки жорстко закодувати, але навіть і просто придумати!

2.2.7. Пошук шляху

Говорячи просто, пошук шляху (pathfinding) являє собою обчислення і виконання руху по дорозі з точки p_1 до цільової точці p_2 (рис. 2.16). При відсутності перешкод найпростіший спосіб потрапити до цільової точки - рухатися по прямій в її напрямі, поки цільова точка не буде досягнута. Однак завдання істотно ускладнюється, якщо на шляху зустрічаються перешкоди, які доведеться огинати.

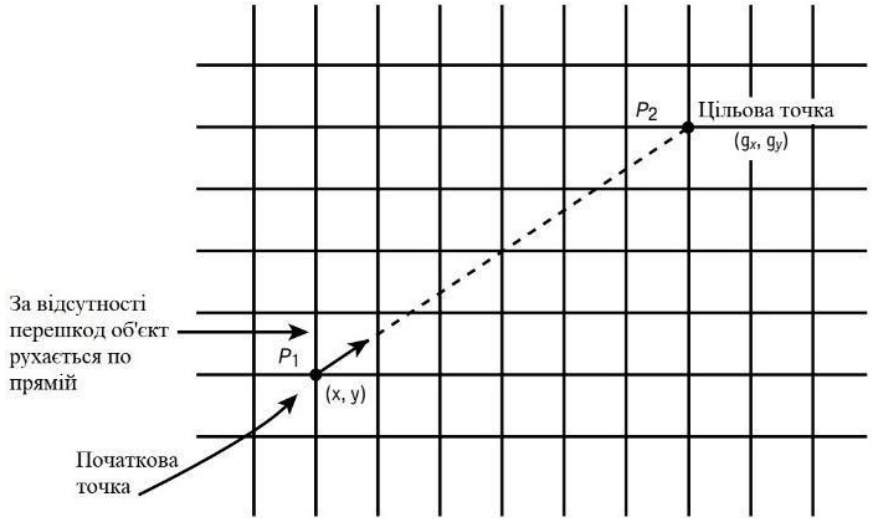


Рис. 2.16. Пошук шляху від точки до точки [40].

2.2.8. Метод спроб та помилок

Коли на шляху з'являються невеликі опуклі перешкоди, зазвичай використовується алгоритм, що складається в тому, щоб при зіткненні з перешкодою відійти назад, виконати поворот вправо або вліво на 45-90 ° та переміститися на деяку наперед визначену відстань (AVOIDANCE_DISTANCE). Після цього штучний інтелект знову визначає напрямок на цільову точку і

повторює спробу дістатися до неї по прямій. Приклад роботи цього алгоритму показаний на рис. 2.17.

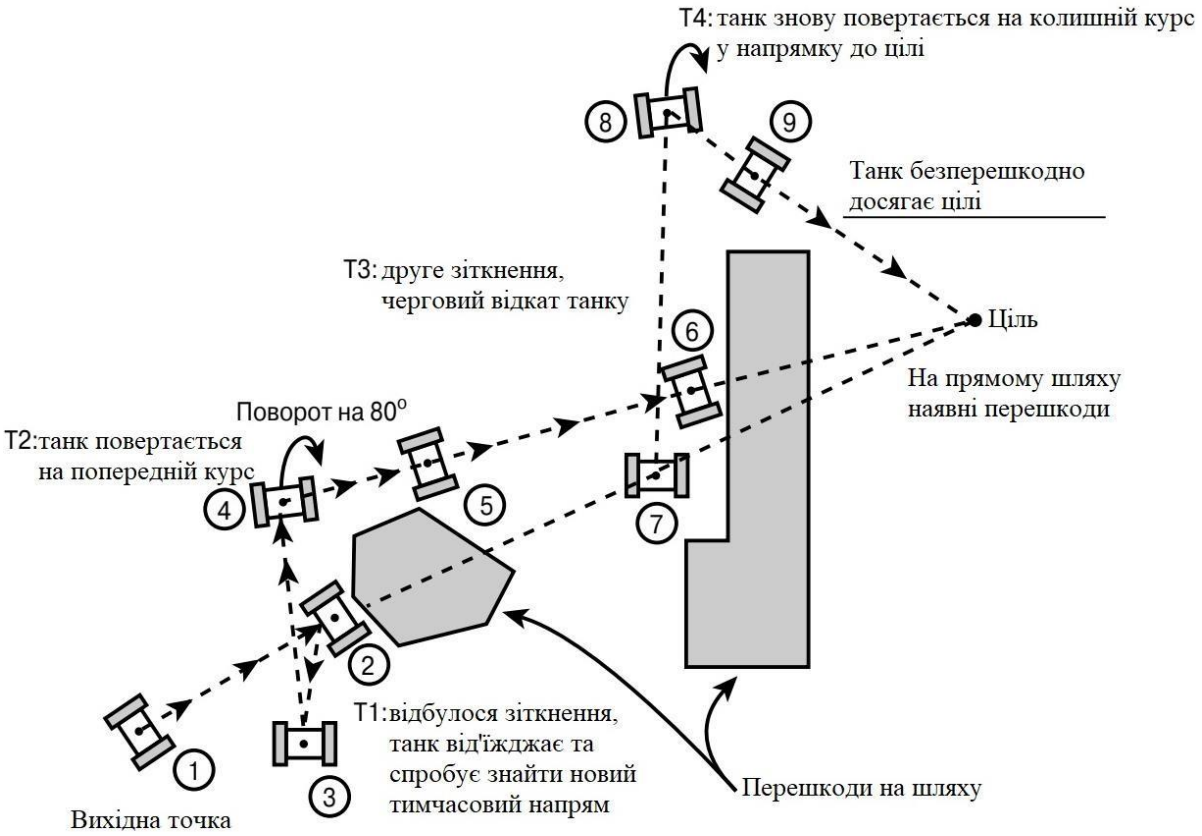


Рис. 2.17. Метод спроб та помилок [44].

Хоча даний алгоритм не так надійний і "розумний", як хотілося б, проте він працює завдяки використанню принципу випадковості. Кожен раз при новій спробі об'єкт повертається у випадковому напрямку, так що рано чи пізно шлях навколо перешкоди буде знайдений.

2.2.9. Обхід по контуру

Інший метод обходу перешкод полягає в обході по контуру. Даний алгоритм відстежує контур перешкоди на дорозі об'єкта. Реалізація даного алгоритму полягає в переміщенні об'єкта навколо перешкоди і періодичної перевірки, чи продовжує відрізок між об'єктом і метою перетинати перешкоду. Якщо немає - об'єкт може рухатися у напрямку до своєї мети; в іншому випадку обхід перешкоди продовжується. Приклад застосування такого алгоритму показаний на рис. 2.18.

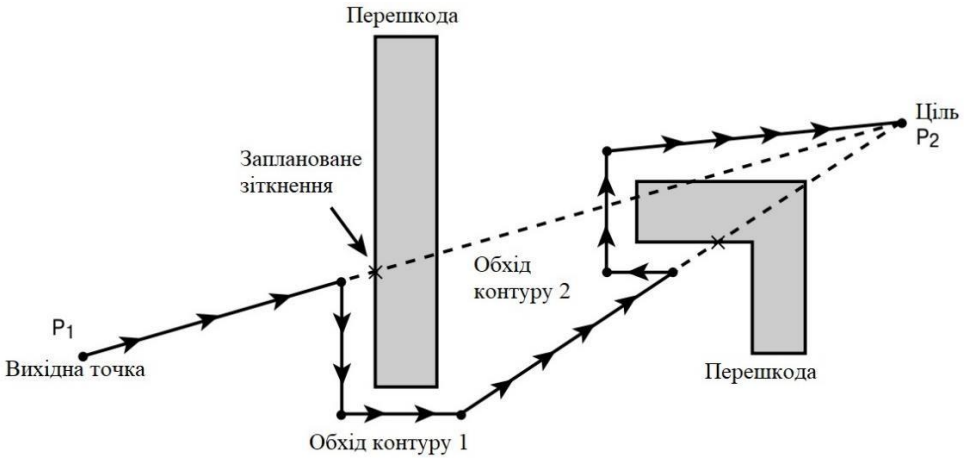


Рис. 2.18. Обхід по контуру [48].

Цей алгоритм цілком працездатний, хоча і виглядає не найрозумнішим, оскільки, як правило, об'єкт рухається не по очевидному з першого погляду найкоротшому шляху. Ви можете скомбінувати два описаних методу пошуку шляху в один: спочатку об'єкт рухається методом проб і помилок, і якщо не досягає мети за заздалегідь заданий час, то в дію вступає алгоритм обходу по контуру.

Звичайно, в іграх, де погляду гравця недоступно все поле відразу, помітити примітивність методу можна лише по тому, що досягнення мети займає трохи більше часу, ніж мінімально необхідне. Однак в стратегічних іграх, де погляду гравця доступне поле цілком, переміщення по контуру виглядає жалюгідно. Чи не можна скористатися якимось більш "розумним" алгоритмом?

2.2.10. Уникнення зіткнень

При використанні даного методу створюються віртуальні шляху навколо об'єктів, що складається з серії точок або векторів, які дозволяють обійти об'єкт по цілком розумному шляху. Цей шлях може бути обчислений з використанням алгоритму найкоротшого шляху або створений вручну вами або дизайнером гри за допомогою відповідного інструментарію.

Навколо кожної великої перешкоди створюється невидимий для всіх, крім керованого штучним інтелектом персонажу, шлях. Коли об'єкту потрібно обійти перешкоду, він запитує найкоротший шлях обходу для даної перешкоди і отримує

його. Тим самим гарантується, що персонаж завжди знає, як обійти ту чи іншу перешкоду. Звичайно, для того щоб ввести до гру різноманітність, можна розробити для перешкоди кілька різних шляхів обходу або додати невеликий"шум" до шляху з тим, щоб об'єкт не дотримувався завжди в точності по одному і тому ж шляху. Проілюстрований описаний метод на рис. 2.19.

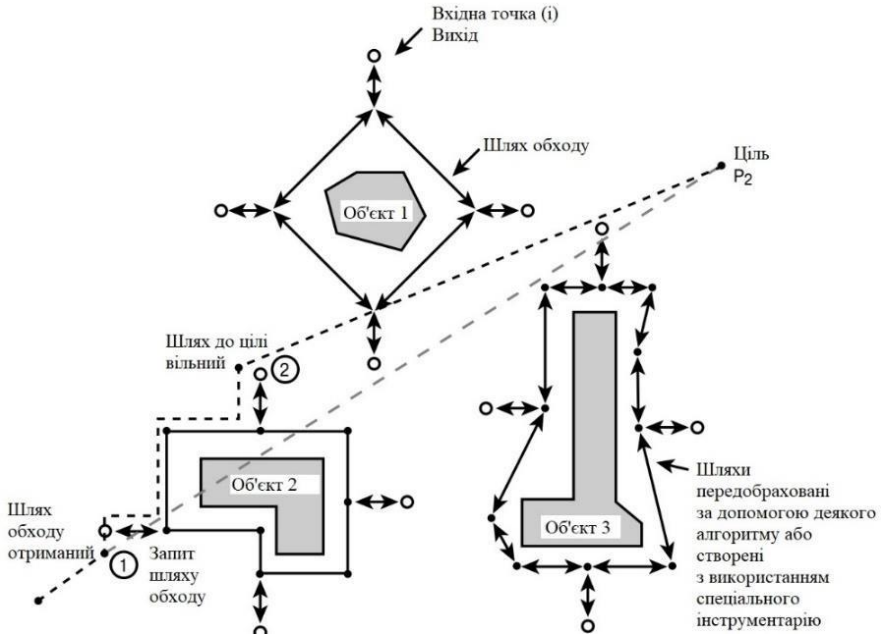


Рис. 2.19. Використання попередньо створених шляхів обходу перешкод [44].

Описаний метод підводить до іншої ідеї: чому б не мати переобраховані шляхи для всіх важливих точок гри? Тоді, якщо об'єкт повинен дістатися від точки p_i до точки p_j, замість визначення напрямку і обходу перешкод можна скористатися наперед обчисленим шляхом.

2.2.11. Пошук шляхів з використанням проміжних пунктів

Нехай в певній грі створений дійсно складний світ з перешкодами різноманітних типів. Звичайно, можна створити персонаж, досить розумний для того, щоби переміщатися по такому світу, спираючись тільки на свої сили; але так чи так уже це необхідно? Можна просто створити мережу шляхів, яка з'єднує всі

важливі точки гри. Кожен вузол такої мережі є проміжним пунктом (waypoint), а дуги такої мережі являють собою напрямок вектору руху і відстань від одного вузла мережі до наступного [42].

Наприклад, нехай є план руху і потрібно перемістити персонаж з його поточного положення по мосту і привести до міста. Взагалі-то це складне завдання, але при наявності мережі шляхів вона зводиться до пошуку такого шляху в місто, який проходить по мосту, після чого потрібно просто слідувати цим шляхом. В такому випадку персонаж гарантовано прийде куди потрібно, благополучно уникнувши при цьому зіткнень з перешкодами. На рис. 2.20 показаний приклад виду такого світу зверху, разом з мережею шляхів. Зазначений шлях і є той, який був шуканий. Запам'ятайте, що така мережа не тільки обходить всі перешкоди, але і повинна містити шляху до всіх скільки-небудь важливих точок гри (наприклад, до підпросторових переходів у вашій всесвіту).

У цій великій схемі є два складних моменти: по-перше, слідувати по шляху і, по-друге, створити структуру даних, яка могла б представляти цю мережу. Почнемо з проходження по шляху.

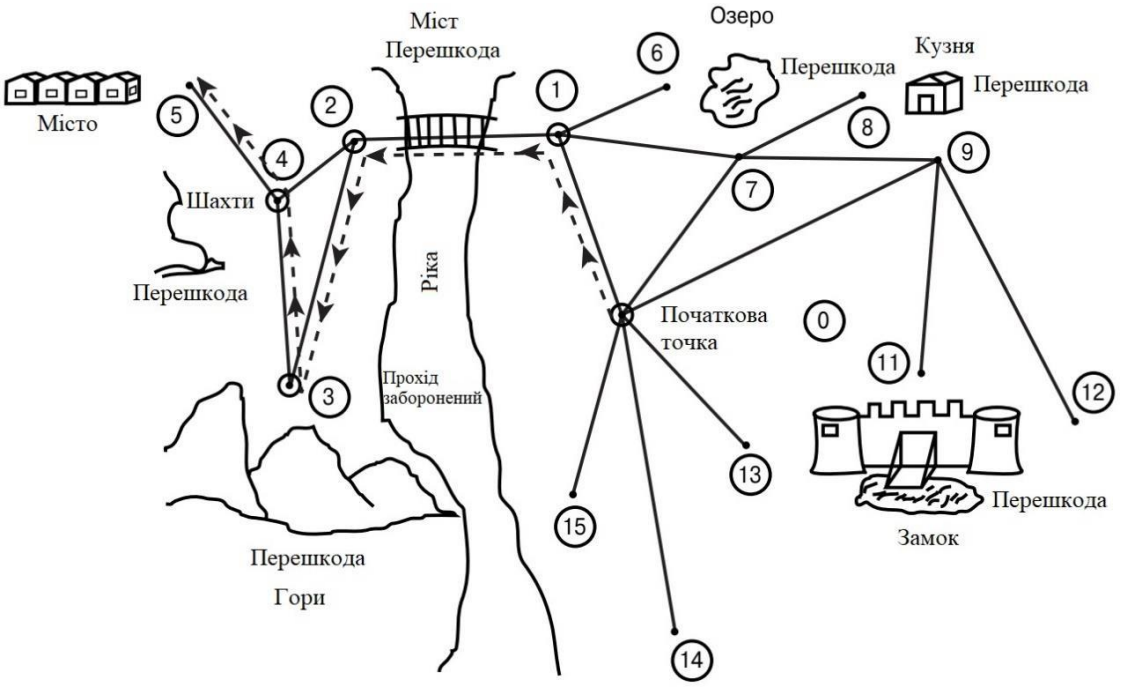


Рис. 2.20. Мережа шляхів на карті світу [44].

Припустимо, у нас є шлях від точки p_1 до точки p_2, що складається з n вузлів, представлених наступною структурою:

typedef struct WAYPOINT_TYP {

int id;

// Ідентифікатор проміжної точки

*char * name;*

// Ім'я проміжної точки

int x, y;

// Положення проміжної точки

int distance;

// Відстань до наступної точки

*WAYPOUNT_TYP **

next;

// наступна точка в списку

} WAYPOINT;

Тепер припустимо, що є шість проміжних точок, включаючи початкову p_1 і кінцеву p_2, як показано на рис. 2.21.

WAYPOINT path [6] = {{0, "START", x0, y0, d0, & path [1]},

{1, "ONPATH", x1, y1, d1, & path [2]},

{2, "ONPATH", x2, y2, d2, & path [3]},

{3, "ONPATH", x3, y3, d3, & path [4]},

{4, "ONPATH", x4, y4, d4, & path [5]},

{5, "ONPATH", x5, y5, d5, NULL}};

Перш за все слід звернути увагу на те, що, незважаючи на статичне виділення масиву для зберігання шляху, його точки пов'язані одна з одною. Останній покажчик в цьому пов'язаному списку дорівнює NULL, оскільки це поле належить кінцевій точці нашого шляху.

Для того щоб слідувати за вказаним шляхом, необхідно взяти до уваги декілька речей. Наприклад, треба дістатися до першого вузла шляху або вузла, наступного за ним, що може створити певну проблему. Вирішити її можна, знайшовши найближчу до сучасного стану об'єкту вхідні точку або точку на шляху, що є цікавою і рухаючись до неї з використанням одного з описаних раніше алгоритмів. Після того, як досягається точка в мережі, подальше переміщення по шляху являє собою просту задачу [40].

2.2.12. Проходження по шляху

Шлях являє собою серію точок, між якими гарантовано немає ніяких перешкод. Чому? Та тому, що розробники самі створювали ці шляхи і точно знають це! В результаті можна просто переміщувати об'єкт з однієї точки, описуваною структурою WAYPOINT, до наступної, і продовжувати діяти таким чином до тих пір, поки не переміститься до останньої, цільової точки.

Знайти найближчу точку WAYPOINT на шляху, який нас цікавить.

whiLe (мета не досягнута) {

Обчислити траєкторію від даної точки до наступної і переміститися по ній

Після досягнення наступної точки траєкторії оновити поточну і чергову точки

} // while

Таким чином, треба просто прямувати по ряду точок до тих пір, поки не досягається цільова точка. Вектор переміщення від однієї точки WAYPOINT до наступної знаходиться елементарно:

```
// Починаємо зі стартової точки
WAYPOUINT_PTR current = & path [0];
// Знаходимо вектор в напрямку наступної точки:
trajectory_x = current-> next.x - current-> x;
trajectory_y = current-> next.y - current-> y;
// Нормалізуємо отриманий вектор
```

normaLize (& trajectory_x, & trajectory_y);

Процес нормалізації робить довжину вектору дорівнює 1.0, залишаючи його напрямок незмінним. Це досягається шляхом ділення кожної компоненти вектору на його довжину. Як тільки об'єкт досягає черговий точки на шляху, вона стає поточної і відбувається обчислення напрямку руху до точки, наступної за нею.

Існують проблеми і з пошуком шляху. Знайти шлях - завдання не менш складна, ніж дістатися від точки до точки при наявності перешкод. Тут все вирішується в основному на рівні використовуваних структур даних. Справа в тому, що одні й ті ж відрізки між точками можуть використовуватися в різних шляхах (відповідний приклад показаний на рис. 2.21) і, чи будуть одні і ті ж проміжні точки повторно використовуватися в різних шляхах, залежить від конкретної реалізації алгоритму.

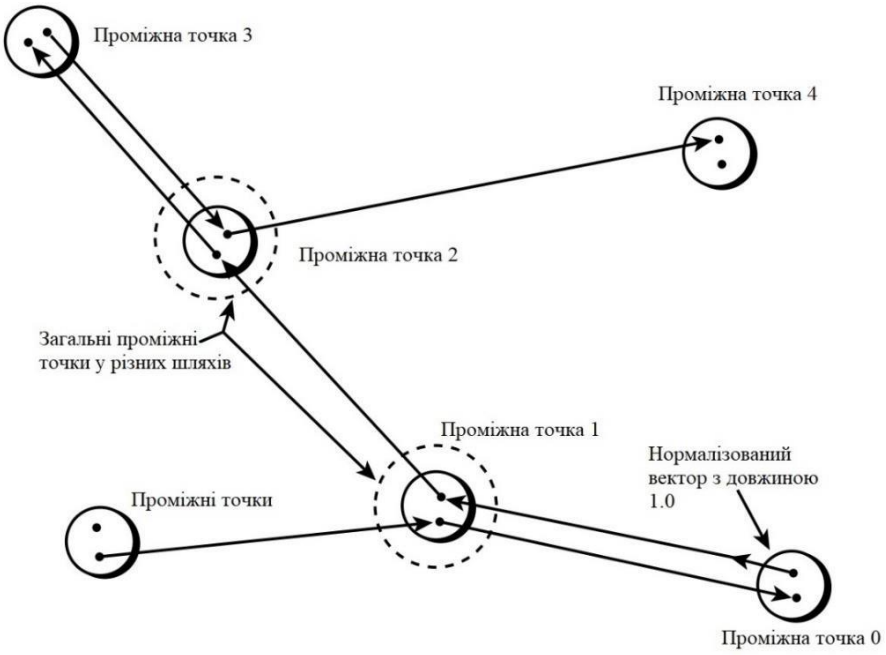


Рис. 2.21. Шляхи з загальними проміжними точками [44].

Хорошим прикладом використання шляхів є ігри з гонками. Уявіть, що на треку повинно знаходитися кілька автомобілів, які повинні рухатися по треку, при цьому уникаючи зіткнень з гравцем і поводячись досить розумно, з точки зору

стороннього спостерігача. Для таких ігор технологія шляхів підходить особливо добре.

Можна створити, наприклад, півтора десятка різних шляхів, у кожного з яких є свої особливості. Кожен автомобіль, керований системою ШІ, рухається в процесі гри по своєму шляху.

У разі аварії, поломки та іншого автомобіль вибирає найближчий шлях і слід за ним. Можливий також перехід від одного шляху до іншого в процесі руху. Все це робить спостережувану картину досить реалістичною.

Тепер про застосування обчислювальних алгоритмів для пошуку шляху між точками p_1 і p_2. Для вирішення даної задачі є ряд алгоритмів, проте всі вони не застосовні для роботи в реальному часі і тому не можуть використовуватися в іграх. Однак вони цілком придатні для побудови шляхів у відповідних інструментах, а також в іграх, але з певними спрощеннями.

Всі алгоритми такого типу працюють з графоподібними структурами, що представляють ігровий простір у вигляді множини вузлів з дугами, що вказують, які саме вузли можуть бути досягнуті з даного. Зазвичай з кожної дугою пов'язана її ціна. Типовий граф такого виду показаний на рис. 2.22.

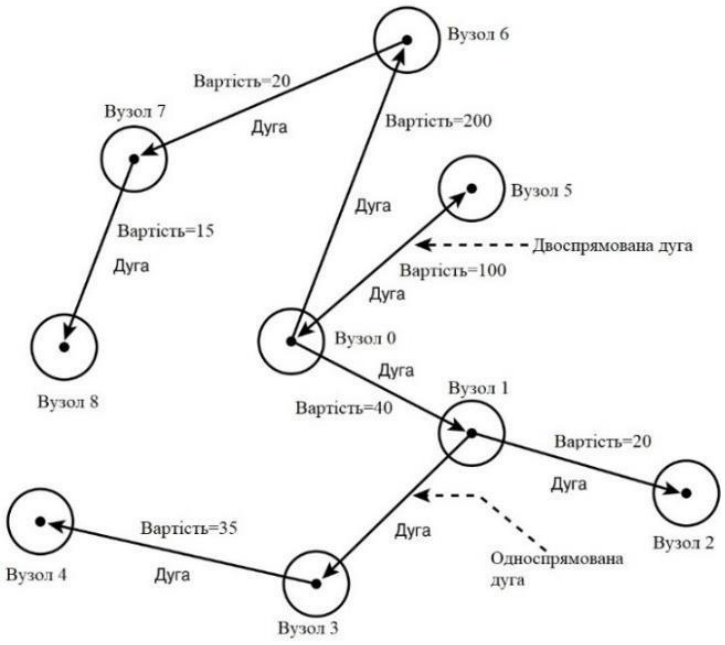


Рис. 2.22. Граф мережі [45].

Оскільки ведеться про справу з дво- і тривимірними іграми, можна вирішити, що граф є накладеним на ігрове поле решіткою, в якій кожна клітинка неявно пов'язана з усіма вісьмома сусідніми осередками; при цьому вартість являє собою не що інше, як відстань між осередками (рис. 2.23).

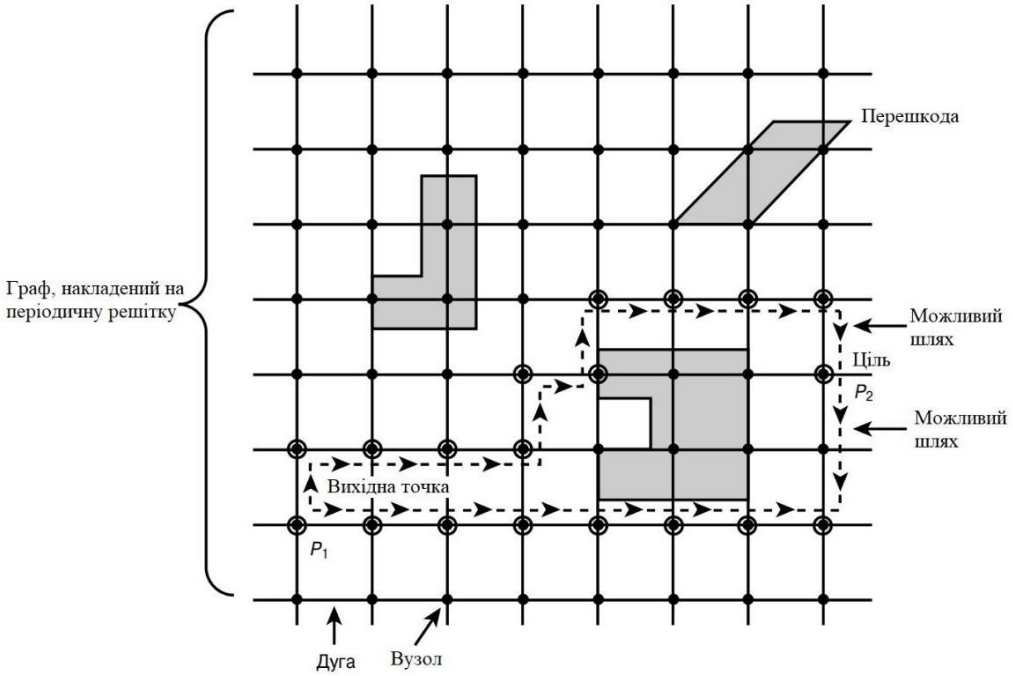


Рис. 2.23. Створення графа з використанням решітки на ігровому полі [48].

У будь-якому випадку, яким би не було представлення ігрового поля увигляді графа, головне завдання - знайти найкоротший шлях між точками p_1 і p_2, наякому не зустрінеться жодної перешкоди (оскільки поняття перешкоди в графі відсутнє, їх просто не може виявитися на знайденому шляху.

2.3. Штучні нейронні мережі

Нейронні мережі - це одна з тих речей, про які всі чули, але ніхто не бачив. Проте в цій області в останні роки відбулися істотні зрушення. Тут я розповім про нейронних мережах дуже коротко (якщо це взагалі можна назвати розповіддю) і не

в контексті практичного застосування в іграх, а просто для того, щоб ви мали уявлення про те, що це таке.

Нейронна мережа являє собою модель людського мозку. Мозок містить від 10 до 100 мільярдів клітин, кожна з яких може обробляти і пересилати інформацію. На рис. 2.24 показана модель клітини людського мозку (нейрона).

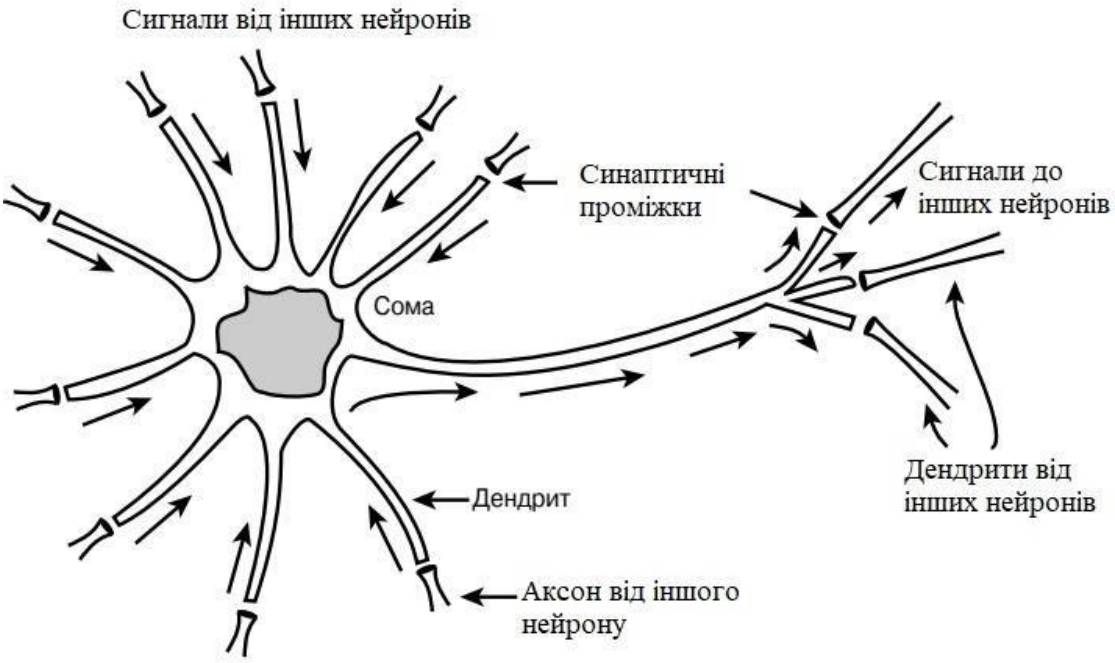


Рис. 2.24. Нейрон [48].

Основними частинами нейрона є сома, аксон і дендрити. Сома є основне тіло клітини і виконує обробку сигналів, в той час як аксон пере¬дает сигнали дендритам, що приносять їх іншим нейронам.

Кожен нейрон виконує дуже просту функцію: обробити вхідний сигнал і послати або чи не послати вихідний сигнал. Нейрони мають безліч входів, єдиний вихід (який може бути розділений) і деякий правило, відповідно до якого обробляються вхідні і генеруються вихідні сигнали. Правила обробки надзвичайно складні; досить лише сказати, що сигнали деяким чином підсумуються і отриманий результат призводить до передачі нейроном вихідного сигналу.

Штучні нейронні мережі являють собою прості моделі, які, як і мозок, в змозі обробляти інформацію паралельно. Розглянемо основні типи штучних нейронів.

Штучна нейронна мережа показана на рис. 2.25.

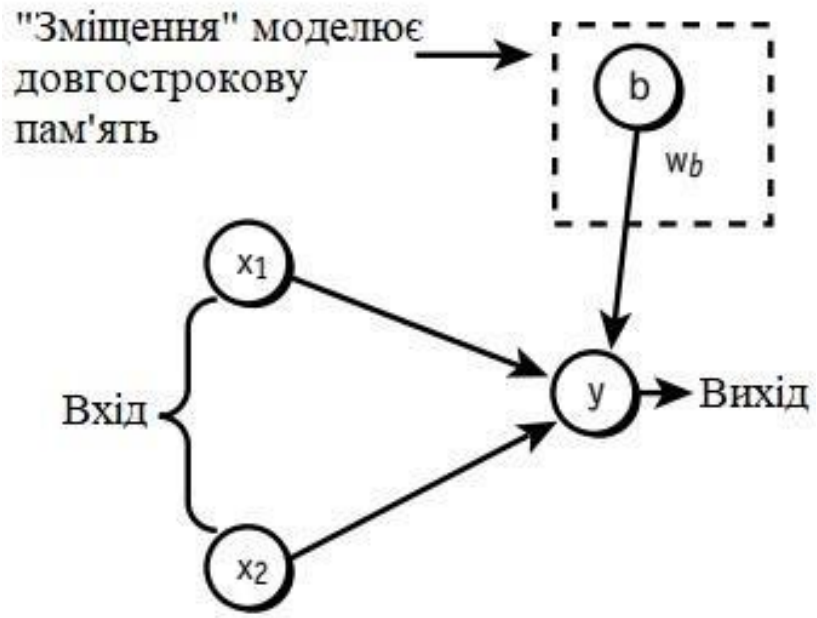


Рис. 2.25. Штучний нейрон [44].

Нейрон складається з ряду входів X_t, які підсумовуються з урахуванням ваг w_t, а потім обробляються за допомогою функції активізації. Ця функція може бути як простою пороговою функцією, так і більш складною, наприклад лінійної або експоненціальної. У порогової моделі підсумкове значення сигналів порівнюється з граничним значенням і нейрон відправляє свій сигнал в тому випадку, коли сумарне значення перевищує порогове. Математично сумарне значення вхідних сигналів можна виразити таким чином:

$$Y = \sum_{i=1}^{n} X_i \, w_i$$

Відповідно вихідний сигнал в порогової моделі буде надсилатися тільки в тому випадку, коли значення Y перевищить граничне. У моделі зі зміщенням підсумкове значення визначається формулою:

$$Y = bw_b + \sum_{i=1}^{n} X_i w_i$$

Для того щоб на власні очі побачити роботу нейрону, припустимо, що є нейрон з двома вхідними сигналами X_1 і X_2, які можуть приймати тільки одне з двох значень - 0 або 1. Встановимо поріг рівним 2, а ваги сигналів $w_1 = w_2 = 1$. Вихідний сигнал при різних вхідних сигналах для цієї схеми нейрону показаний в табл. 2.2. Як бачите, цей нейрон просто реалізує логічний оператор AND. За допомогою відповідної функцій активізації і ваг сигналів нейрони можуть легко реалізувати обчислення будь-якої логічної функції. Наприклад, на рис. 2.26 показані моделі для обчислення операторів AND, OR і XOR.

Зрозуміло, реальні нейронні мережі набагато складніше, мають безліч шарів, складні функції активізації, а кількість нейронів в них може обчислюватися тисячами.

Таблиця 2.2. Таблиця істинності простого нейрону [44].

X1	X2	Сумарний сигнал	Вихідний сигнал
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

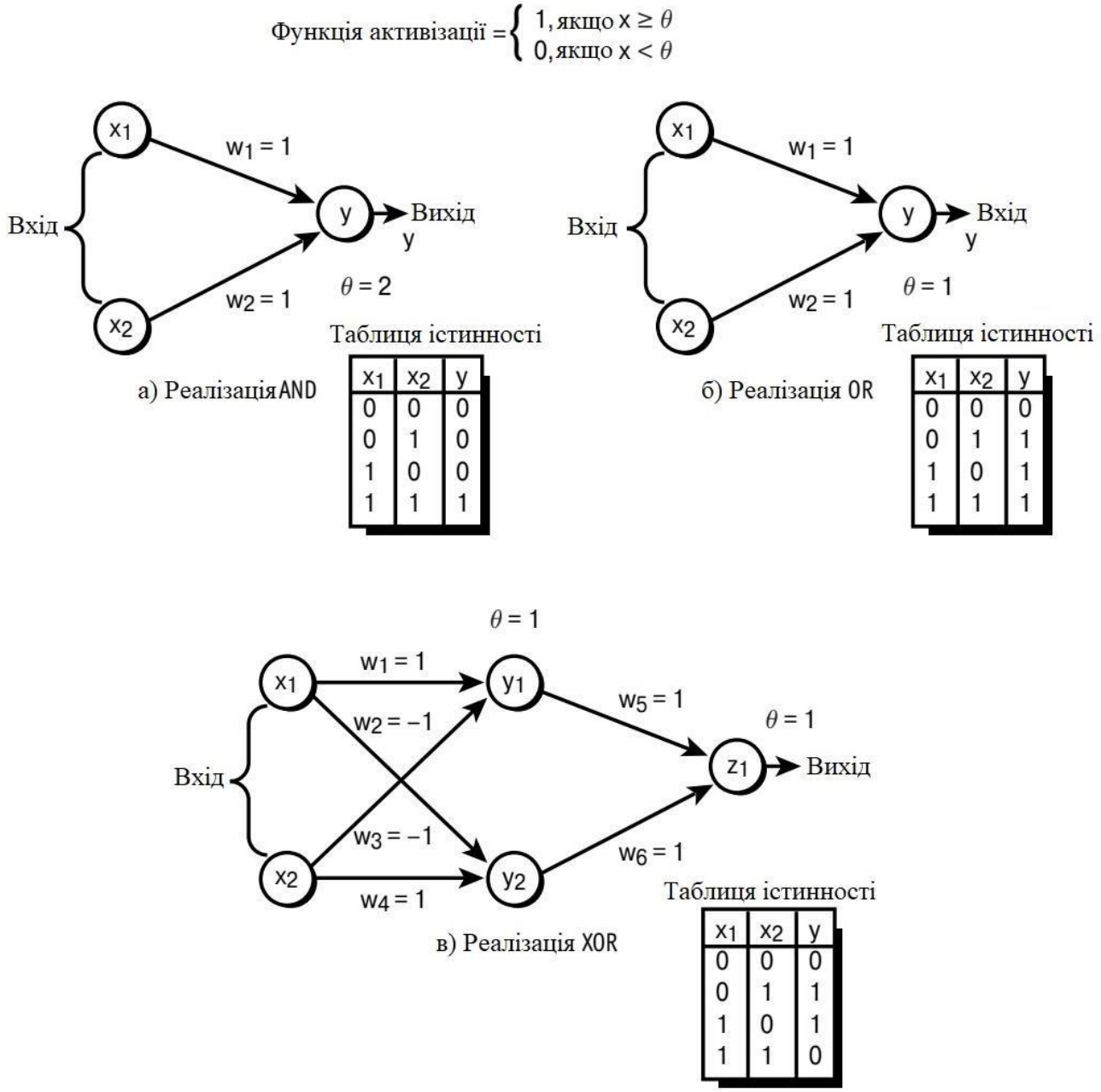


Рис. 2.26. Реалізація логічних операторів [40].

2.4. Генетичні алгоритми

Генетичні алгоритми являють собою метод обчислень, заснований на біологічних моделях еволюції. Вони намагаються застосувати такі концепції, як природній (і штучний) відбір і генетичні мутації в комп'ютерних моделях, покликані допомогти вирішити завдання, які не можуть бути дозволені стандартними обчислювальними засобами.

Принцип роботи генетичних алгоритмів приблизно наступний. Можна представити отримувану інформацію у вигляді бітового вектору, що нагадує ДНК (рис. 2.27).

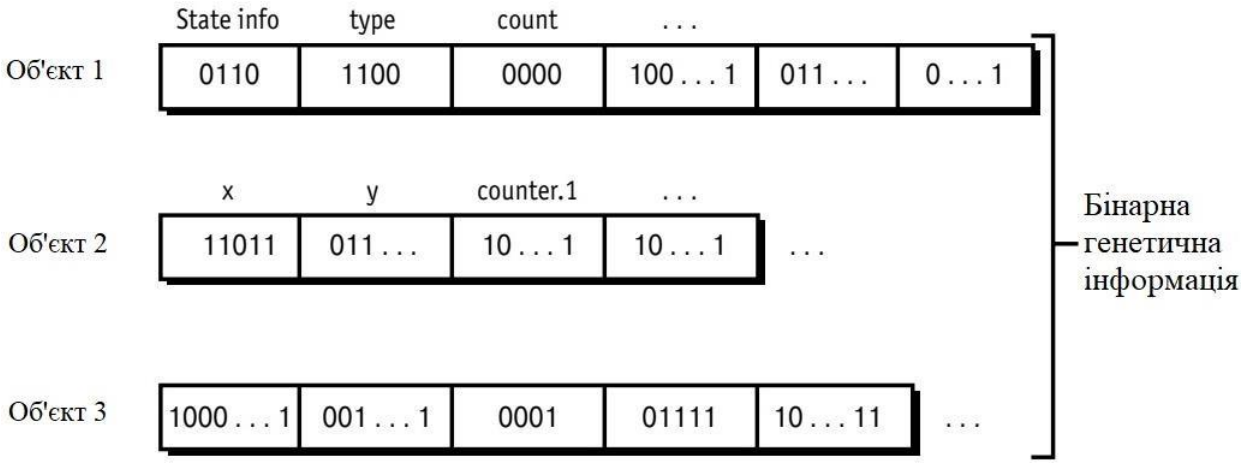


Рис. 2.27. Бінарне представлення генетичної інформації [40].

Цей бітовий вектор являє собою стратегію або код алгоритму розв'язання задачі. Для початку потрібна деяка кількість таких бітових векторів. Потім оброблюється бітовий рядок і отримується за допомогою деякої цільової функції ступінь відповідності цього рядка поставленої мети. Початкові бітові вектори можуть бути отримані на підставі деяких апріорних даних або навіть інтуїтивних уявлень.

Після обробки ми порівнюємо ступеня відповідності вихідних векторів поставленої задачі, а далі в дію вступає генетичний алгоритм. Відбираються ті

вектори, які дають найкраще наближення до вирішення даного завдання, і моделюємо їх схрещування, наприклад так, як показано на рис. 2.28.

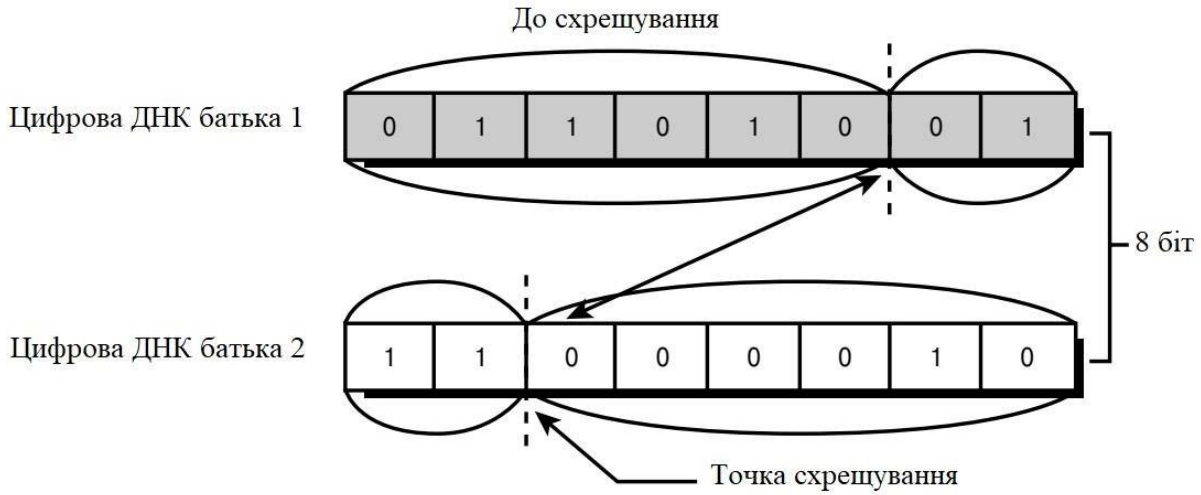


Рис. 2.28. Цифрове схрещування [44].

Для додавання невеликої непередбачуваності в процесі схрещування деякі біти можуть бути змінені (моделювання мутацій). Отримавши нове покоління рішень, можна відчути їх придатність поряд з попереднім поколінням і відбираються для подальшого розмноження тільки кращі екземпляри. Це і є процес генетичної еволюції. Найкраще рішення еволюціонує слабо, і, що цікаво, при такому методі часто виходять результати, які часом навіть неможливо собі уявити.

Ключовим в ідеї генетичних алгоритмів є те, що область пошуку рішення виявляється дуже широкою. Це зазвичай ніколи не досягається при пошуку рішення крок за кроком, а досягається, крім іншого, моделюванням мутацій, що являє собою абсолютно випадкові події, які можуть привести (або не привести) на краще відповідності бітового вектору вирішення поставленого завдання.

В якості цифрової ДНК можна використовувати імовірнісні установки штучного інтелекту, а потім розглядати в якості цільової функції виживання персонажів з тих чи інших імовірнісних поведінкою, поступово "вирощуючи" покоління персонажів, все більш пристосованих до даної гри. Зрозуміло, що вносити зміни в ДНК можливо тільки при створенні нового персонажу.

2.5. Нечітка логіка

Серед продвинутих методів нечітка логіка є одним з найважливіших інструментів, які повинен використовувати розробник ігрового ШІ через простоту її формулювання в поєднанні з її виразності.

Нечітка логіка являє собою надмножину традиційної логіки, яке було розширено для обробки поняття значень часткової правди між булевою функцією істини та хибності. Нечітка логіка зазвичай приймає форму нечіткої системи міркувань, а її компоненти - це нечіткі змінні, нечіткі правила і механізм нечіткого виведення.

Теорія нечітких множин була введена в 1965 році Лотті А. Заде в області академічного штучного інтелекту [50], але реалізувати його ідеї вдалося японським дослідникам, які продемонстрували практичне використання нечітких систем управління. Нечітка логіка також використовується в поєднанні з іншими методами ШІ, такими як еволюційні алгоритми або нейронні мережі, в навчанні та класифікації [51].

Як і багато інших академічних методів ШІ, нечітка логіка була протестована в відеоіграх [51]. Нечітка логіка була офіційно представлена в розробці ігор в 1996 Ларрі О'Брайеном [52] і з тих часів вивчена і вдосконалена іншими авторами [1,2].

Нечітка логіка може бути корисна для ігрового ШІ в декількох аспектах. Вона може використовуватися для прийняття рішень NPC, таких як вибір предметів або зброї, для управління рухом юнітів.

Через лінгвістичний характер нечіткої логіки формулювання правил може бути виконано експертами в галузі ШІ, а потім нечітка система може використовуватися для наслідування міркувань експерта [53]. Це велика перевага по відношенню з іншими методами, які вимагають знання як самого методу, так і правил його налаштування.

Нечіткі методи дозволяють будувати відносини введення-виведення на основі знань експерта без необхідності створення складної математичної моделі, яка може виявитися стомлюючої або виснажливою для отримання [51].

Нечітка логіка може використовуватися для моделювання складної поведінки з низькою обчислювальною вартістю [1].

Нечітка логіка вимагає правильного визначення вхідних та вихідних змінних, а також їх відносин. Якщо немає експерта в галузі ШІ, складно буде розробити правила і, можливо, буде потрібно багато налаштувань, що призведе до збільшення часу розробки.

Конструювання нечіткої системи для гри з багатьма агентами, в разі якщо вони не будуть ретельно розроблені, може призвести до перевірки сотень правил на кожному часовому кроці, повністю виключаючи переваги низької обчислювальної вартості одиничних перевірок. Хоча найпростіший варіант нечіткої системи з'являється в більшості ігор, через цього недоліку були запропоновані поліпшення, пов'язані з природою ігрового ШІ. [2].

Один конкретний приклад нестачі багатьох правил в нечіткій логіці називається комбінаторним вибухом [46]. Розробники ігрового ШІ пропонують прийняти метод Комбса [47], який дозволяє лінійне зростання правил щодо числа змінних і множин.

Переглядаючи літературу, можна знайти кілька згадок про використання нечіткої логіки в відеоіграх.

BattleCruiser: 3000AD6 - це космічна стратегічна гра з суперечливою історією розвитку, яка використовує нечітку логіку поряд з нейронними мережами для управління неігровими персонажами в грі [56].

S.W.A.T. 27, являє собою тактичну гру в реальному часі, яка була присвячена широкому використанню нечіткої логіки, що дозволяє неігрових персонажів вести себе спонтанно на основі їх певних особистостей і здібностей [3].

Civilization: Call to Power, покрокова стратегічна гра, яка є побічним ефектом дуже відомої франшизи, використовує FuSMs для визначення пріоритетів для ШІ

стратегічного рівня, дозволяючи визначити індивідуальні риси для різних лідерів цивілізації [3].

З наведених прикладів можна бачити, що нечітка логіка успішно використовується в іграх, хоча розробники, як правило, не йдуть далі, ніж прості логічні системи виведення.

Мета і завдання дослідження. Основною метою роботи представлена розробка інтелектуального агенту, який реалізує поведінкові реакції на основі правил з використанням нечітких множин. В якості практичної задачі будемо розглядати конкретну ситуацію в якій беруть участь об'єкти інтерактивного оточення, такі як двері і сходи. Завдання полягає в тому, щоб реалізувати поведінкову реакцію на відкриття дверей і підняття по сходах для досягнення певної мети в ігровій сцені [57].

Розробка правил з використанням нечітких множин. При загальному підході NPC буде приймати ці об'єкти як перешкоди і не стане взаємодіяти з ними. З метою усунення цієї проблеми можна використовувати нечітку логіку для вироблення форм поведінки, необхідних для взаємодії з цими об'єктами. Щоб уникнути багатозначності трактування семантичних значень одного і того ж параметра в різних ситуаціях, будуються повні ортогональні семантичні (постортогональние) простору [58], які служать областями нечітких значень кожного з параметрів незалежно від даної системи.

Для побудови повного ортогонального семантичного простору (ПОСП) деякого нечіткого параметра $\tilde{p}$ визначається множина нечітких значень $\tilde{D}=\{\tilde{p}_i^k\}k = 1 \dots K_i$, де K_i кількість нечітких значень, прийнятих і-м параметром, у вигляді нечітких чисел з функцією приналежності μ_i^k, яка визначена на інтервалі (p_{ib}^k, p_{ie}^k), де $(p_{ib}^k, p_{ie}^k) \in D_i$ - значення початку і кінця інтервалу відповідно, а D_i – базова множина нечітких значень параметра $\tilde{p}$.

Для того, щоб побудовані D_i множини були ПОСП, необхідно, щоб вони задовольнили наступним аксіомам:

Аксіома 1 - нормальність: кожна функція приналежності μ_i^k нечітких значень p_{ib}^k і досягає одиниці на деякому ненульовому відрізку значень $[p_{ib_l}^k, p_{ie_l}^k]$ базової множини D_i:

$$\forall k \in [1; K_i] \quad \exists \tilde{p}_i \in \tilde{D}_i \mu^k(p) = 1 \qquad p \in [p_{ib_l}^k, p_{ie_l}^k] \tag{2.1}$$

Аксіома 2 - функція μ_i^k не убуває зліва від $p_{ib_l}^k$ і не зростає праворуч від $p_{ie_l}^k$:

$$\mu_i^k(p) \geq \mu_i^k(p_{ib}^k), \quad p < p_{ib}^k$$

$$\mu_i^k(p) \leq \mu_i^k(p_{ib}^k), \quad p > p_{ib}^k \tag{2.2}$$

Аксіома 3 - функції μ_i^k не можуть мати більше двох точок розриву першого роду.

Аксіома 4 - повнота: для будь-якого значення p з множини D_i знайдеться нечітке значення $\tilde{p}_i \in \tilde{D}_i$ з ненульовим значенням функції приналежності $\mu^k(p_i)$ в даній точці:

$$\forall p \in \tilde{D}_i \quad \forall k \in [1; K_i]: \mu_i^k(p) \neq 0 \tag{2.3}$$

Аксіома 5 - ортогональність: сума всіх значень функцій належностей $\mu_i^k(p)$ в деякій точці p базового множин D_i повинна рівнятися одиниці [52]:

$$\sum_{k=1}^{K_i} \mu^k(p) = 1, \ p \in D_i \tag{2.4}$$

Кожне нечітке число $\tilde{p}_i \in \tilde{D}_i$ визначається через функцію належності такого вигляду:

$$p_i^k \Rightarrow \mu_i^k(p') = \begin{cases} 0, p' \leq p_{kb}^i, p'_i \geq p_{ke}^i \\ \frac{p_i - p_{kb}^i}{p_{kb_l}^i - p_{kb}^i}, p_{kb}^i < p' < p_{ke}^i \\ 1, p_{kb}^i < p' < p_{ke_l}^i \\ \frac{p_i - p_{ke}^i}{p_{ke_l}^i - p_{ke}^i}, p_{kb_l}^i < p' < p_{ke}^i \end{cases} \quad i = 1 \ldots N_p, k = 1 \ldots K_i \tag{2.5}$$

З урахуванням $\mu_1^k \prec D_i, \mu_2^k \prec D_i, \mu_n^k \prec D_i$ відношення строгого порядку на множині нечітких значень і -го параметра, повинні виконуватися наступні умови

[52]:

$$\begin{cases} p_{kb}^{i} = p_{kb_l}^{i} = \frac{min(p_i)}{D_i} \quad i = 1 \ldots \tilde{N} \\ p_{ke}^{i} = p_{ke_l}^{i} = \frac{max(p_i)}{D_i} \end{cases} \qquad (2.6)$$

де $p_i^{'}$ - деяке чітке значення і-го нечіткого параметру; p_{kb}^{i}, p_{ke}^{i} відповідно початкове і кінцеве значення інтервалу значень базового безлічі D_i, на якому функція приналежності k-го нечіткого значення і параметру позитивно визначена; $p_{kb_l}^{i}, p_{ke_l}^{i}$ - початкове і кінцеве значення відповідно інтервалу значень базової множини D_i, на якому функція приналежності k-го нечіткого значення і-го параметра дорівнює одиниці.

Для визначення стану об'єкта необхідно порівнювати деяку вхідну нечітку ситуацію S_i з кожною нечіткою ситуацією S_j. В якості заходів для визначення ступеня близькості нечіткої ситуації S_i нечіткої ситуації S_j використовуються: ступінь нечіткого включення нечіткої ситуації S_i в нечітку ситуацію S_j, ступінь нечіткої рівності нечіткої ситуації S_i і нечіткої ситуації S_j. Вибір міри близькості визначається особливостями нечіткої ситуації і організацією блоку прийняття рішень.

Так, для наступних нечітких ситуацій:

$$S_i = \{\underset{p_m}{\mu_{s_i}(p_m)}\}_{m=1}^{M}, \quad S_j = \{\underset{p_m}{\mu_{s_j}(p_m)}\}_{m=1}^{M} \qquad (2.7)$$

Де $\{\underset{p_m}{(p_m)}\}_{m=1}^{M}$ - набір ознак, за якими визначаються нечіткі ситуації, а

$$\mu_{s_j}(p_m) = \{\underset{T_{mk}}{\mu^t(T_{k\,mk})}\}_{k=1}^{P(m)} \qquad (2.8)$$

де $\mu_{s_j}(p_m)$ - ступінь приналежності ознаки p_m до терму T_{mk}, а $P(m)$ - кількість термів в терм-множині ознаки p_m.

Перш за все необхідно забезпечити управління рухом. Для цього використовується лінгвістична змінна move (рух), яка визначається значенням швидкості руху. Вона представлена трьома різними термами forwards (вперед), stop

(зупинки), backwards (назад).

Грунтуючись на попередніх формулах (1) необхідно побудувати пост ортогональні простори. Терм forwards (вперед) визначається як нечітке число за допомогою функції приналежності $\mu_{forwards}(v)$, яка задається співвідношенням [52]:

$$\mu_{forwards}(v) = \{ \begin{matrix} \frac{v}{V1}, 0 \le v \le V1 \\ 1, V0 \le v \le V_{max} \end{matrix} \quad (2.9)$$

Терм backwards (назад) визначається на нечітке число за допомогою функції приналежності $\mu_{\text{backwards}}(v)$, яка задається співвідношенням:

$$\mu_{\text{backwards}}(v) = \{ \begin{matrix} \frac{v}{V1}, 0 \le v \le -V1 \\ 1, -V1 \le v \le -V_{max} \end{matrix} \quad (2.10)$$

Терм stop (стоп) визначається на нечітке число за допомогою функції приналежності "яка задається співвідношенням (4).

$$\mu_{\text{stop}}(v) = \{ \begin{matrix} \frac{v+V1}{V1}, -V1 \le v \le 0 \\ 1, V = 0 \\ \frac{V1-v}{V1}, 0 \le v \le V1 \end{matrix} \quad (2.11)$$

Постортогональний простір змінної move представлено на рис. 2.27.

До процесу пересування необхідно застосувати ще одну лінгвістичну змінну turn, яка відповідає за поворот тіла. Описана базова змінна визначається відносним кутом повороту в градусах. Мінлива turn вказується 4 нечіткими ситуаціями для змінних, кожна з яких вказує на об'єкт, до якого повинен повернутися NPC: button (кнопка), door (двері), ladder (сходи), exit (вихід). Функції цих змінних змінюються динамічно, оскільки позиція цих об'єктів змінюється з часом.

Також передбачена лінгвістична змінна look, яка дозволяє управляти кутом нахилу голови. До змінної look входять 2 терм-множини up(вгору), down (вниз), non(не зраджувати).

Перераховані вище лінгвістичні змінні door і ladder є змінними, що описують сприйняття об'єкта.

Змінна door представлена термами open (відкрита) close (закрита) та fully open (повністю відкрита).

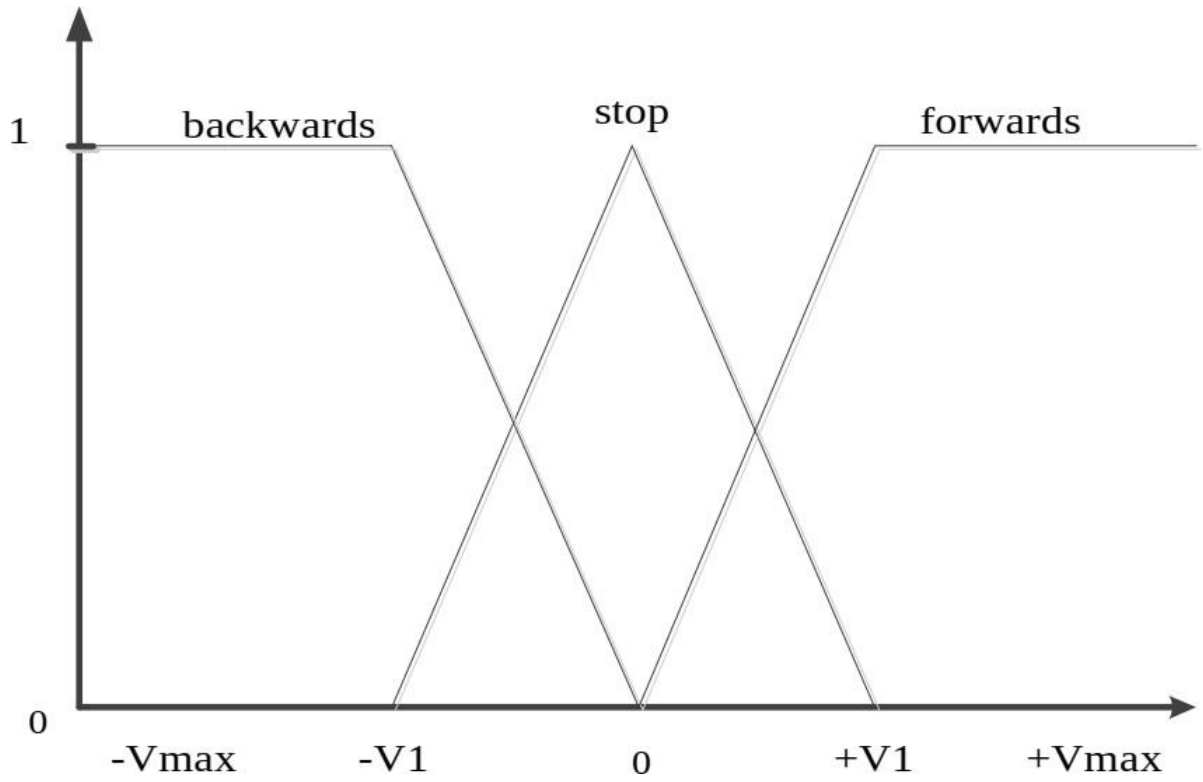


Рис. 2.27. Постортогональний простір множині змінної move [59].

Змінна button являє собою блок керуючих рішень, значення якого має кілька станів B_1 (закрити повністю), B_2 (закрити наполовину), B_3 (доставити), B_4 (відкрити наполовину), B_5 (відкрити повністю). Для кожного певного стану будується нечітка ситуаційна мережа (НСМ). Нечітка ситуаційна мережа [15] являє собою нечіткий орієнтований зважений граф переходів за нечіткими еталонними ситуаціями.

Лінгвістична змінна ladder, визначена за допомогою двох термів held (зайнятий) і top(вгору). Терм held визначено в часі і має максимальне значення 1, через секунду як NPC торкнеться сходи і повертається в 0 з той момент, коли через секунду NPC зійде зі сходів. Терм top представлений змінною height (висота). Змінна приймає значення 1 в момент, коли NPC готовий зійти зі сходів і значення 0, після того, як буде зроблений крок від верхньої сходинки.

Далі, для прикладу, необхідно розробити набір правил для послідовного виконання дій. Для того, щоб NPC пройшов через дверний проріз спочатку

необхідно натиснути на кнопку, а потім пройти через дверний проріз. Правила такого завдання описуються в наступній таблиці.

При організації цієї форми поведінки необхідно забезпечити пріоритет зі зверненням до натискання кнопки, а після проходити через дверний проріз.

Для того, щоб NPC виліз по драбині необхідно спочатку підійти до сходів, схопитися за неї, потім підняти погляд вгору і здійснювати рух в прямому напрямку до тих пір, поки не буде досягнута вершина сходів, після NPC необхідно направити погляд вперед і зійти зі сходів на підлога. Правила такого завдання описуються наступним чином.

Таблиця 2.3. Правила завдання взаємодія з дверима [59]

Умови	Дія
If not button_pressed	Повернутися до сторони кнопки, рушити вперед
If button_pressed	Повернутися до дверей
If button_pressed and door not fully open	Зупинитися
If door fully open	Рушити вперед, повернутися до дверей

Таблиця 2.4. Правила завдання взаємодія зі сходами [59]

Умови	Дія
True	Рушити вперед
If ladder is not held	Повернутися до сходів
If ladder is held	Подивитися вгору
If ladder is top	Подивитися вгору, рухатися, повернутися до виходу

На основі представлених правил поводжень можна описати ряд подібних ситуацій взаємодії NPC з об'єктами навколишнього середовища. Основною перевагою даної моделі є завдання нечітких ситуацій за допомогою змінних.

3. СТВОРЕННЯ ВЛАСНОЇ МОДЕЛІ ШТУЧНОГО ІНТЕЛЕКТУ

3.1. Огляд найбільшш популярних відеоігор в жанрі шутер

Шутери цілком можна назвати найпопулярнішим жанром ігор: якщо RTS і класичні квести знаходяться на волосині від смерті, то гри про відстріл ворогів виходять щороку і стабільно користуються попитом. Не дивно, все ж механіка «прицілився - вистрілив» проста для освоєння, та й у багатьох геймерів знайомство з іграми як такими почалося саме з шутерів - Call of Duty, Battlefield, Counter-Strike, Doom.

Далі наводиться огляд найважливіших шутерів від першої особи, які сформували жанр таким, яким він є. Проекти, з якими прийнято порівнювати і на які прийнято рівнятися. У цьому матеріалі буде обмежено поодинокими іграми - орієнтовані на кооперативні і розраховані на багато користувачів режими шутери типу Battlefield залишилися за бортом [61].

Doom. Якщо перераховувати найважливіші гри жанру, в першу чергу на думку спадає Doom - родоначальник всіх шутерів. Тут, звичайно, можна заперечити і згадати Wolfenstein 3D, а хтось навіть назве і більш ранні приклади. Однак саме Doom дала потужний старт жанру швидких, «м'ясних» екшенів від першої особи.

Doom побудований на дуже простих механіків: бігай і стріляй у все, що рухається, а заодно намагайся не потрапляти під ворожі постріли. Але диявол криється в дрібницях: багатий арсенал зброї, дизайн ворогів і загальне побудова локацій перетворюють перестрілки в справжню тактичну гру.

У кожного з ворогів свій впізнаваний дизайн, звуковий супровід і унікальна тактика поведінки, завдяки чому вони моментально зчитуються візуально - досить секунди, щоб зрозуміти, з ким доведеться зіткнутися і як йому протистояти.

Рис. 3.1. Doom

Зомбі слабкі, але завдають шкоди миттєво за рахунок вогнепальної зброї. Імпи більш живучі і б'ють болючіше, але їх вогняні кулі повільніше кулі, що дозволяє ухилитися. Демони «Пінкі» не стріляють взагалі і намагаються нав'язати ближній бій, проте по здоров'ю і швидкості вони перевершують своїх колег [42].

Ігри в дусі Call of Duty і пізніх Far Cry лише йдуть по шляху збільшення складності і просто споряджають одного і того ж ворога-людини товщою бронею і важкою зброєю, тоді як в Doom кожен противник особливий і вимагає певного підходу.

Все це змушує гравця постійно думати: де зайняти позицію для кращого вогню? Яка зброя застосувати зараз - дробовик для ближнього бою або ракетомёт для контролю натовпу? Які вороги стоять перед вами і на кому з них зосередитися в першу чергу? Проста на перший погляд «стрілялка» на ділі виявляється вивіреної грою, на правилах геймдізайн якої в майбутньому буде зростати весь жанр.

Hexen. На правилах геймдізайн Doom побудована гра Hexen, відмінна риса якої - фентезійний сеттінг. Дробовики і кулемети замінені мечами, сокирами і магією, що змінює звичну формулу перестрілок Doom, але при цьому не

уповільнює ігровий процес. Варто уваги, якщо ви не боїтеся своєрідних рівнів - вони в Hexen вельми заплутані.

Рис. 3.2. Duke Nukem 3D

Крім движка Doom, в 90-е була ще одна популярна основа для шутерів від першої особи - Build Engine. На ньому вийшло багато знакових ігор, але, мабуть, найвідомішою і інноваційної на той час можна назвати Duke Nukem 3D - пафосний, маскулінний екшен про самозакоханого качка-блондина Дюка. Ігри про нього виходили і раніше, але в основному вони були в 2D і не здобули настільки своєї слави, якої добилася тривимірна частина.

Причина популярності - величезна кількість нововведень і знахідок, якими Duke Nukem 3D виділялася на тлі Doom, Wolfenstein 3D і їх наслідувачів. Шутер від 3D Realms вніс істотні зміни в динаміку боїв: ворогів в Duke Nukem 3D куди менше, ніж в тому ж Doom, однак боротися з ними складніше. Дюк не такий живучий, як Думгай, та й противники знають пару хитрих трюків на кшталт стрільби з положення лежачи або телепортації за спину.

Duke Nukem 3D: Megaton Edition додала нормальну підтримку клавіатури / миші і текстури високого дозволу - мабуть, зараз це найприємніший спосіб оцінити гру [42].

Але і Дюк не такий простий: в його арсеналі як важкі і «круті» гармати, так і всіляке корисне спорядження. Бомби з детонатором, настінні міни з лазерним датчиком, що зменшує промінь, заморожуюча гармата, голографічна копія для обману ворогів. Арсенал Duke Nukem 3D виглядає по-справжньому свіжим і цікавим в порівнянні зі стандартним набором «пістолет-дробовик-кулемет-ракетниця», який популяризував Doom. Втім, вони в грі теж є.

Рівні в Duke Nukem 3D зробили великий крок вперед у порівнянні з конкурентами. Вони куди цікавіше на вигляд: на зміну одноманітним лабіринтах ігор id Software прийшли яскраві вулиці американських міст, стрипклуби, космічні станції, ресторани, магазини коміксів та інші локації з реального світу, що задають відмінний контраст війні з гротескними прибульцями, чия головна мета - красти земних дівчат .

Quake. Прихід епохи 3D ознаменувався релізом Quake від id Software. Ідея гри про відстріл нечисті в готичних замках і похмурих світах, натхнених Гігера і Лавкрафт, прийшла з D & D партій розробників id Software. Спочатку Quake замислювалася як пригодницька гра про хлопця з електричним молотом, який бореться з монстрами, проте в ході розробки той перетворився на безіменного солдата, який подорожує між світами.

Quake стала першою грою з повністю тривимірним світом, що дозволило додати в шутер багато нових фіч, головна з яких - вертикальний геймплей.

Чи жарт, але у всіх знаменитих шутерах від першої особи до Quake не було повноцінної багаторівневості: додаткові поверхи і височини в Doom і Duke Nukem 3D створювалися за рахунок трюків та візуальних ілюзій, коли насправді кожен рівень складався всього з однієї площини або поверху. І тільки у Quake з'явився движок, здатний реалізувати повноцінне об'ємне простір - Quake Engine [43].

Рис. 3.3. Quake

Це дозволило створити куди більше унікальних ігрових ситуацій і зробити рівні значно цікавіше: секретні поверхи з бонусами під дахом, багатоповерхові вежі, бездонні прірви і затоплені тунелі, чиї таємниці і проходи приховані каламутною водою.

Архітектура локацій в Quake була абстрактною і сюрреалістичною - таких храмів і підземель в реальному житті бути не могло, однак вони бездоганно працювали в рамках гри. До слова, завдяки 3D і грамотної реалізації переміщення в просторі, Quake стала однією з перших ігор, де підводні етапи перестали викликати фізичну біль через жахливий управління.

Завдяки технічному стрибка Quake ввела або популяризувала практично всі звичні для наступних шутерів елементи: отскакивающие від стін снаряди, Rocket Jump (стрибок великої висоти за допомогою пострілу ракетою під ноги), расприжка (збільшення швидкості за рахунок постійних стрибків в сторони) і багато іншого [43].

Багатий на можливості движок став основою практично для всіх наступних за грою id Software екшенів: знаменитий Source і движок ігор серії Call of Duty засновані саме на Quake Engine. «Серце» Quake дозволило створити не тільки якісні

ігри з купою технічногоких знахідок, але і самі різні фанатські модифікації: наприклад, саме з подачі моддеров з'явився Team Fortress, який пізніше став однойменної мультіплєєрной грою від Valve.

Однак движок був не єдиною перевагою Quake. У ній чітко дізнавався шутер id Software: значний і запам'ятовується арсенал, унікальний сеттинг, відмінний саундтрек і різноманітні, нетривіальні противники. Лицарі, що без страху біжать на гравця з одним лише мечем; практично безсмертні зомбі, що закидають ворога частинами власного тіла; величезні шамблери, метан заряди блискавки і, звичайно ж, огри - здоровані з гранатометами, які напрочуд влучно закидають героя гранатами, користуючись рикошетом і вогнем на випередження.

Return to Castle Wolfenstein. Один з найпопулярніших сеттінгом для ігор - ця Друга світова війна. Стратегії (як віртуальні, так і настільні), тактичні ігри, сюжетні пригоди і шутери регулярно звертаються до цієї теми - аж надто великий слід вона залишила в історії людства.

Однак якщо багато творів відносяться до нацизму досить серйозно, то серія Wolfenstein використовує антураж Другої світової і образ «злих фріців» в незвичайному ключі. Пригоди бравого американського солдата Бі Джея Йоганес Бласковіц роблять ставку на фарс і абсурд: агент США раз у раз бореться з нечистю, мутантами і іншими експериментами особливих окультно-паранормальних підрозділів Третього Рейху [61].

Wolfenstein 3D. Грунтовно за цю тему франшиза зачепилася з виходом Wolfenstein 3D. Вона не була першою грою про Бі Джея, але саме цей шутер зробив серію по-справжньому відомою. Пізніше нехитра гра про бої з нацистами, собаками і кібернетичним Гітлером була швидко потіснила Doom - просто тому що як шутер він був краще у всьому.

Серія довго залишалася в тіні, поки в 2001 році, вже під видавництвом Activision, в світло не вийшов перезапуск - Return to Castle Wolfenstein. Нова гра не тільки повернула франшизу в лад, але і знову зробила революцію. Геймери знову почали говорити про замок Вольфенштейн.

Головна особливість Return to Castle Wolfenstein, яка зробила її по-справжньому унікальною - це симбіоз жанрів. У грі було потроху від усього відразу: шутер старої школи на манер Doom, стелс-гра, околореалістічний екшен про Другу світову і пригода з пошуком секретів і артефактів.

Від «старої школи» грі дісталися аптечки, колекціонування зброї, продуманий дизайн локацій, важливість правильного позиціонування та різноманітні вороги з впізнаваним дизайном і унікальним патерном поведінки.

Вони, до речі, стали набагато розумнішими і різноманітніше в порівнянні з тим же Quake: противники можуть кооперуватися, патрулювати місцевість, кидати гранати, піднімати тривогу і навіть битися один з одним по «ідеологічним» причин (нежить з нацистами уживається не дуже добре). Після пхнущих напролом демонів Doom і монстрів з Quake все це відчувалося величезним кроком вперед.

А ще в Return to Castle Wolfenstein є повноцінний стелс. Зброя з глушником і ніж не привертають уваги і дозволяють пройти за рівнем, не піднімаючи тривогу. Німецькі штурмовики наполегливо патрулюють місцевість і спілкуються на абстрактні теми, наївно вважаючи, що поруч немає ворогів. А почувши постріли або звуки бою, вони тут же побіжать досліджувати кімнати одну за однією, поки не знайдуть вас.

Незважаючи на те, що в своїй основі RtCW - олдскульний шутер, гра вміло використовує елементи реалізму: герой так само смертна, як і противники, а на точність зброї впливає розкид - гравець, затискає кнопку пострілу до переможного, ризикуючит пустити весь боєзапас в молоко. А ось стрілянина акуратними, вивіреними чергами дозволяє вбивати ворогів без втрати влучності навіть на великих відстанях.

На точність так само впливає те, стоїте ви або переміщується - на бігу стріляти важко. Зараз ці речі здаються буденністю, але тоді, на зорі епохи 3D-шутерів, подібні ідеї були революційними [44].

Return to Castle Wolfenstein затягує не тільки через геймплея, але і завдяки дизайну локацій. Кожен рівень тут - нова незвичайна ситуація: замок нацистів,

окуповані міста, стародавні підземелля, секретні лабораторії, темні ліси, покинута церква. Скрізь свої особливості, впізнаваний зовнішній вигляд і безліч секретних кімнат.

Настрій і темп гри постійно змінюються, тримаючи в тонусі: зараз ви перемагаєте нацистів в напруженій перестрілці посеред замку Вольфенштейн, а вже через півгодини боязко краде по підземному склепу, ухиляючись від пасток і відстрілюючи ожилих мерців.

Unreal. Return to Castle Wolfenstein - не єдиний реформатор початку нульових. Набагато раніше новим словом в жанрі стала Unreal - гра від Epic MegaGames (яка зараз зветься просто Epic Games). Таку назву шутер отримав неспроста: для кінця дев'яностих він і правда став чимось нереальним.

Краси планети На-Пали (Na-Pali) вражали своїм стилем: суміш фантастичних храмів, літаючих островів і гігантських гір з низенькими селами місцевих жителів і футуристичними базами злісної раси Скааржей (Skaarj). Можливо, зараз квадратні будинки і трикутні гори виглядають не так вражаюче, але свого часу потрібен був по-справжньому потужний комп'ютер, щоб розглянути простори На-Пали у всій красі.

Одна з переваг візуальної частини Unreal - відмінна робота з водою і освітленням. Останнє було повністю динамічним: тьму підземель можна розсіювати ліхтарем і освітлювальними ракетами

Unreal пам'ятають в основному за чарівний сеттинг далекої планети і невимовну красу оточення, однак вони - не єдині переваги гри. Шутер від Epic Games запам'ятався новим підходом до оповідання через оточення. Без всяких діалогів і мінлива гравець може познайомитися з На-Пали, просто роздивляючись архітектуру і пам'ятки. Єдиний прямий джерело інформації про те, що відбувається - аудіологи і записки інших людей, що потрапили на планету, які можна знайти на шляху [61].

Крім цього, в Unreal незвичайний штучний інтелект. Мирні істоти - тварини, птахи, риби - займаються своїми справами, не чіпаючи протагоніста. Хоча птахи можуть і напасть - або самі, або якщо їх спровокувати.

Ще з мирних істот в Unreal можна зустріти Налі - релігійний народ, який знаходиться під гнітом Скааржей, які захопили планету. Головного героя місцеві вважають месією, що спустився з небес, а тому допомагають йому припасами і показують секретні кімнати з бонусами - якщо, звичайно, переживуть перестрілку.

Противники - окрема розмова. Крім величезного розмаїття, вони мають вражаючим розумом: самотній піхотинець НЕ буде несамовито намагатися вбити гравця, а побіжить за підмогою, якщо поблизу є союзники.

Вороги постійно патрулюють рівні, намагаються обійти вас, а деякі жваво ухиляються від пострілів і використовують захисні пристосування. Про бездумному Раше в дусі Doom тут немає й мови.

Unreal вивела внутрішньоігрових супротивників на новий рівень для всієї ігрової індустрії. Навіть перша поява більшості інопланетян тут зроблено по-особливому: зустріч з новим типом ворогів зазвичай поставлена так, щоб гравець встиг з ним ознайомитися і зрозуміти, як йому протистояти.

Але якими б крутими були вороги, їх треба чимось вбивати. І арсенал в Unreal відповідає: з десяток зовсім різних гармат, у кожноїз яких є альтернативний режим вогню. Стандартний шоковий бластер може накопичувати заряд, кулемет - випускати пучок снарядів, а шестиствольний ракетомёт - посилати кілька ракет відразу.

Причому кожен стовбур залишається корисним протягом всієї гри. Крім, хіба що, піломёта, який приносить більше проблем, ніж користі через дикого рикошету снарядів. У запалі бою вони можуть не тільки обезголовити ворога, але і серйозно поранити самого героя [44].

Unreal - це гра, в якій неймовірна атмосфера сусідить з відточеною шутерной частиною. Бої відмінно відчуваються за рахунок тямущих супротивників, різноманітності локацій і ворогів, а також опрацьованого арсеналу.

Наступні частини робили упор на мультиплеер, і одиночна кампанія вражала там вже не так сильно - чи нові декорації могли зрівнятися з магією На-Пали. Та й сама серія подзаглохла і зараз не в пріоритеті у Epic Games. Так що єдине, що залишається - це грати в Unreal Gold, повністю робочий збірник з оригінальної гри і доповнення, мультиплеєрний сервера якого працюють до цих пір.

Half-Life. Через півроку після Unreal вийшов ще один знаковий для жанру шутер - Half-Life. Гра від тоді ще не дуже відомої студії Valve заробила світову славу за рахунок новаторського підходу до оповідання і геймдізайн. Багато прийоми в ній дуже схожі на ідеї Unreal, проте зроблені вони зовсім інакше.

В першу чергу, тут особливий спосіб подачі сюжету. В Unreal вся історія і лор розповідалися через декорації і текстові повідомлення, розкидані по локаціях. Half-Life теж активно використовує оповідання через оточення, проте гра Valve робить ставку не на естетику, а на функціональність.

Будь епізод з пригод фізика-ядерника Гордона Фрімана - це справжня завдання, яку потрібно вирішити в перервах від стрільби (ну або прямо під час перестрілки).

Наприклад, рівень з підводним чудовиськом. Здавалося б, це просто філлерний епізод для розтягування сюжету. Але в підсумку через цю подію Геймдизайнер одночасно знайомлять гравця і з новим противником (риба-мутант), і з новою зброєю - арбалет, що працює як снайперська гвинтівка.

З таких ось моментів складається вся Half-Life: гравець рухається по сюжету, не відволікаючись ні на що, крім геймплея. Всі події і важливі діалоги відбуваються прямо в грі, без жодних зрежисованих катсцен [61].

Сам Гордон Фрімен, головний герой гри, не промовляє жодного слова, навіть коли його запитують безпосередньо. Це зроблено для того, щоб гравцеві було легше себе асоціювати з головним героєм - нібито персонажі звертаються саме до нього, а не до Гордону.

Саме Half-Life ми повинні дякувати за популяризацію сюжетно-орієнтованих шутерів: крім неї і Unreal над постановкою і кінематографічністю працювала хіба

що серія Call of Duty, та й та робила ставку більше на видовищність, ніж на якийсь виразне оповідання.

Крім цього, у гри вистачає і інших достоїнств: відмінна стрільба, фізика об'єктів (скажемо спасибі переробленому движку Quake Engine), що зачаровує фантастичний сеттінг і чудовий дизайн - як оточення, так і численних чудовиськ з іншого виміру.

Не варто забувати і про сиквел: Half-Life 2 повторила успіх оригіналу і знову зробила революцію в жанрі. Подача сюжету стала ще краще, рівно як і стрілянина, дизайн і графіка. Але і без нововведень не обійшлося: друга частина зроблена на движку Source, що дозволило Valve реалізувати неймовірну для тих років фізику об'єктів. Добра половина Half-Life 2 побудована на фізичних завданнях. Зламати дошку, щоб спустити платформу; накидати ящиків на міст, щоб опустити його або притягнути машину величезним магнітом.

Окремо варто відзначити місто Рейвенхольм - заповнений зомбі селище шахтарів, в який головний герой потрапляє практично беззбройним. Тому відбиватися від тих, що ходять мерців доводиться підручними засобами: пастками, вибуховими бочками і дисками від циркулярної пилки. Предмети, розкидані по карті, можна кидати на ворогів за допомогою спеціальної гравію-гармати, яку Гордон отримує в першій половині [61].

Серія Half-Life, так само як і її движок Source, справили величезний вплив на всі наступні ігри: до сих пір багато сюжетно-орієнтовані шутери порівнюють з іграми Valve. Не дивно, що багато хто і до цього дня чекають третю частину, якої навряд чи судилося коли-небудь вийти: повторити революцію втретє поспіль нелегко, а в іншому вигляді Half-Life 3 фанатам просто не потрібна.

Halo. Оригінальна трилогія Halo змінила шутери: без неї Call of Duty, Destiny, DOOM і інші представники жанру були б зовсім іншими або пройшли б той же самий шлях інакше. Можна ненавидіти консольний метод управління та автонаводку, але навіть непомітна допомога в прицілюванні, яка робить стрілянину

чесніше і комфортніше, з'явилася завдяки цій грі. І це не кажучи про тактичний бій і мультиплеер.

Революція почалася з Marathon, першої серії FPS від Bungie. Саме з неї в Halo перекочує основна тема - боротьба з ворожої інопланетної расою, - а також ідея з ІІ-напарником героя і загальний стиль. А ще це був перший шутер з можливістю дивитися вгору і вниз за допомогою миші і перший шутер з повноцінним ракетним стрибком - ще до Quake.

Кольорові прибульці і незвичайну зброю були вже в Marathon

Головне звершення Bungie в тому, що вони зробили шутери зручними. Ніколи раніше герой не керувався так інтуїтивно зрозуміло з двох стіків на геймпаді - правим ходиш, лівим прицілюватися.

Додатково до цього під час стрілянини гравцеві допомагали в прицілюванні. Досягалося це кількома способами: приціл сповільнювався, коли приходив через ворога, погляд героя злегка притягався до ворога, але не «залипає» на ньому. А в разі незначного промаху кулі все одно потрапляли в ціль, компенсуючи брак швидкості стіки геймпада. Вся допомога була настільки непомітною, що здавалося справедливою.

Перестрілки в грі - справжнісінькі тактичні завдання. Halo спочатку планувалася як стратегія, але навіть зі зміною жанру нібито їй і залишилася - тільки в іншій перспективі. Битви тут проходять на відкритих локаціях, а головному герою протистоять різні види юнітів і техніки. Розібратися з кожним по-окремо - не проблема [47].

Але кілька ворогів разом представляють серйозну загрозу. Таку, що поки герой буде відстрілювати солдат зі щитами, що захищають стрілка, з флангу нападе підкріплення. І це тільки один із сценаріїв, тому що противники в грі діють по ситуації, роблячи кожну сутичку унікальною.

Halo зробила дуже багато для об'єднання гравців: локальний мультиплеєр першої Halo, мультиплеер Xbox Live другий, а також модифікація карт, запис мережевих баталій і можливість ділитися своїми твореннями з іншими в третій

частині. І все ж особлива заслуга у Halo 2 - вона не тільки популяризувала онлайн на консолях, але і показала, як виділені сервери можуть зробити очікування матчу швидким і зручним.

Словом, якщо Doom і Wolfenstein прийнято розглядати як основоположників жанру, то трилогія Halo визначила його подальший розвиток.

Call of Duty. Якщо запитати випадкового перехожого, які «стрілялки» він знає, то з великою ймовірністю ви почуєте назву саме цієї серії. Популярність, яка супроводжує Call of Duty досі, обумовлена не якоюсь неймовірною магією, а цілком звичайними речами: будь-яка з ігор серії проста в освоєнні, чудово поставлена і відмінно зроблена з точки зору околореалістічной стрільби (реальне зброю замість футуристичних гармат і симуляція такий-сякий віддачі).

Перша частина вийшла через пару років після тріумфу Return to Castle Wolfenstein. Так само, як і гра про зомбі-нацистів, Call of Duty експлуатує тему Другої світової і копіює деякі геймдізайнерскіе рішення.

Однак вона намагається показати обрану епоху натурально - без всяких таємних досліджень, окультних співтовариств і інших інопланетян. Тільки сувора дійсність і важливі історичні події. За цією формулою серія буде жити до самої четвертої частини [47].

Здавалося б - в Call of Duty немає різноманітних супротивників, цікавих рівнів-завдань і якихось своїх фішок. Це просто коридорний тир про Другу світову - в кінці-кінців, до неї був Medal of Honor, гра в тому ж сеттинге.

Але саме CoD сильніше запала в душу гравцям. Вона була дуже зрозуміла і проста в освоєнні: ніяких важких механік, расприжек, складної тактики для супротивників. Тільки чистий екшен з формулою «прицілився-вистрілив», «траншейний» стиль бою і різні тактичні завдання. Навіть лікування аптечками зникло після першої ж частини, поступившись місцем регенерації за типом Halo. Тож не дивно, що багато геймери, які виросли на Quake, прозвали серію «тиром».

Bulletstorm. Якісь шутери вивели на новий рівень ігровий процес. Якісь - загальну естетику оточення. Але що щодо видовищності стрільби? Звичайно, сотні

дизайнерів за десятиліття доклали чимало зусиль, щоб зброя звучало і виглядало максимально круто або реалістично.

Розробники думали про красу, про зручність і про те, щоб перестрілки були цікавими. Але мало хто підштовхував гравців вбивати ворогів майстерно або винахідливо - зазвичай таку мету переслідували самі геймери, яким хотіло покрасуватися своїм умінням стріляти в іграх.

Пару прикладів ігор, в яких за стиль нагороджували, зрозуміло, можна згадати: та ж серія Devil May Cry, де красива перемога над ворогом - головна мета. Але серед безпосередньо шутерів ніхто не вибудовував весь геймдізайн навколо ідеї очок стилю. Ще не з'явилася Bulletstorm, яка, ймовірно, не першої придумала використовувати ідею винахідливих убивств, але точно першої її популяризувала.

Гра від студії People Can Fly перетворює вбивство бандитів в справжнє мистецтво. Щоб купувати поліпшення для зброї та отримувати кошти на закупівлю патронів, потрібно не просто пристрелити ворога - зробити це треба винахідливо. Потрапити в голову, прострелити через стіну, штовхнути на шипи, штовхнути на оголений провід, застрелити в повітрі, підірвати ракетою, запущеною в дупу - способи креативної розправи в Bulletstorm перевалюють за сотню [61].

Цим весело займатися, зроблений процес легко і зрозуміло. До того ж, здорово дізнатися якийсь новий, абсолютно хворий спосіб вбивства ворогів: наприклад, штовхнути одного ворога і пристрелити іншого, поки перший ще не долетів до землі. Ну або пальнути в пах - працює безвідмовно з часів Fallout.

При цьому Bulletstorm не спонукати гравців бути якимись хворими на голову маніяками: гра всіма силами підтримує атмосферу пустотливого божевілля, такого собі «типового літнього блокбастера».

Дія гри відбувається на планеті-курорті, який захопили кримінальники. Там можна знайти: нескінченні бари і зони відпочинку, перероблені під бандитські бази; величезного керованого динозавра-аніматроніка, здатного стріляти лазером з очей; електричні бурі, що можуть за секунду вбити будь-кого, і знамените гігантське колесо, що женеться за головними героями на початку гри.

Чого вже говорити про серйозність або реалізмі, коли головний герой може посеред смертельно небезпечної перестрілки глушити пиво прямо з горла - і це лише приносить більше очок стилю, за які той потім докупить зайвих патронів.

Додайте до цього відмінну роботу дизайнерів, добре прописаний сюжет і персонажів і, що найголовніше, абсолютно хворе (в хорошому сенсі) зброя: пускова установка зі свердлами, револьвер з підствольного сигнальної ракетницею, рушницю з цілими чотирма горизонтально розташованими стовбурами і гвинтівка з кулями, якими можна управляти в слоу-мо. Що не кажи, а в Bulletstorm дуже весело грати [61].

Far Cry. Спочатку Far Cry була грою Crytek і створювалася скоріше як демонстрація їх графічного движка. Простецький сюжет про секретну лабораторію з мутантами на тропічному острові, непогана стрілянина і дуже великі рівні з керованим транспортом - єдине, чим могла похвалитися гра.

Far Cry не була чимось видатним, але все ж запам'яталася. Чимось особливим серія стала тільки після того, як перейшла до Ubisoft, яка використовувала бренд для експериментів з шутерами у відкритому світі.

У плані самих перестрілок серія Far Cry не робить нічого нового: хіба тільки друга частина намагалася реалізувати ідею повного занурення, практично позбувшись інтерфейсу. Головна перевага серії - це відкритий світ. Саме Far Cry популяризувала відмова від коридорних рівнів в шутерах, нехай це і коштувало геймерам засилля вишок і крафтінга, як в інших іграх, так і в наступних частинах серії.

Кожна Far Cry - це пісочниця на манер Crysis, де все перестрілки протікають по-різному. Ось тільки якщо в Crysis (та й першої Far Cry, чого вже там) «відкриті» локації були великими, але замкнутими зонами, Far Cry 2 і наступні частини випускали гравців в повноцінні відкриті світи: простори охопленої громадянською війною Африки, тропічні острови під владою піратів-работоргівців і індо-тибетська країна киратів, де править жорстокий диктатор Паган мін. Остання на поточний

момент частина серії і зовсім пропонує відвідати звичайну американську глибинку, де до влади прийшов релігійний культ.

3.2. Створення власної моделі штучного інтелекту

Ігровий додаток розрахований на його використання одним гравцем. Гра починається з ігрового поля, в центрі якого розміщується керований об'єкт (пістолет Glok з рукою гравця). Гравець управляє пістолетом, який може змінювати кут огляду та стріляти в обраному гравцем напрямку.

Розробка використання штучного інтелекту відбувається за допомогою методу кінцевих автоматів (FPS).

Безпосередньо переможні умови відсутні. Мета гравця полягає в тому, щоб якомога більше набрати очок за знищення кубиків-мішеней, що знаходяться на спеціальних стовпчиках посеред приміщення. Гра завершується, якщо гравець знищив всі мішені. Нові мішені з’являються лише в зв’язку з новим входом гравця до гри.

Ігрова камера має фіксоване положення і охоплює ігрове поле цілком.

4. СТВОРЕННЯ АВТОНОМНО ФУНКЦІОНУЮЧИХ ЕЛЕМЕНТІВ (ПЕРСОНАЖІВ) ГРИ

4.1. Огляд обраних засобів реалізації

Unity - це мультиплатформовий інструмент для розробки двох- та тривимірних додатків та ігор, що працює під операційними системами Windows і OS X. Ігри, створені на цьому інструменті можна перенести на: Windows, OS X, Android, Apple iOS, Linux, а також на ігрові приставки Wii, PlayStation 3 і XBox 360. Так само можна створювати ігри, що працюють в браузері, для цього треба встановити спеціальний модуль Unity Web Player. Також додатки створені, за допомогою Unity3D підтримують обидві специфікації 3D графіки DirectX і OpenGL. [63]

На рис. 4.1. представлена графічна оболонка Unity3D.

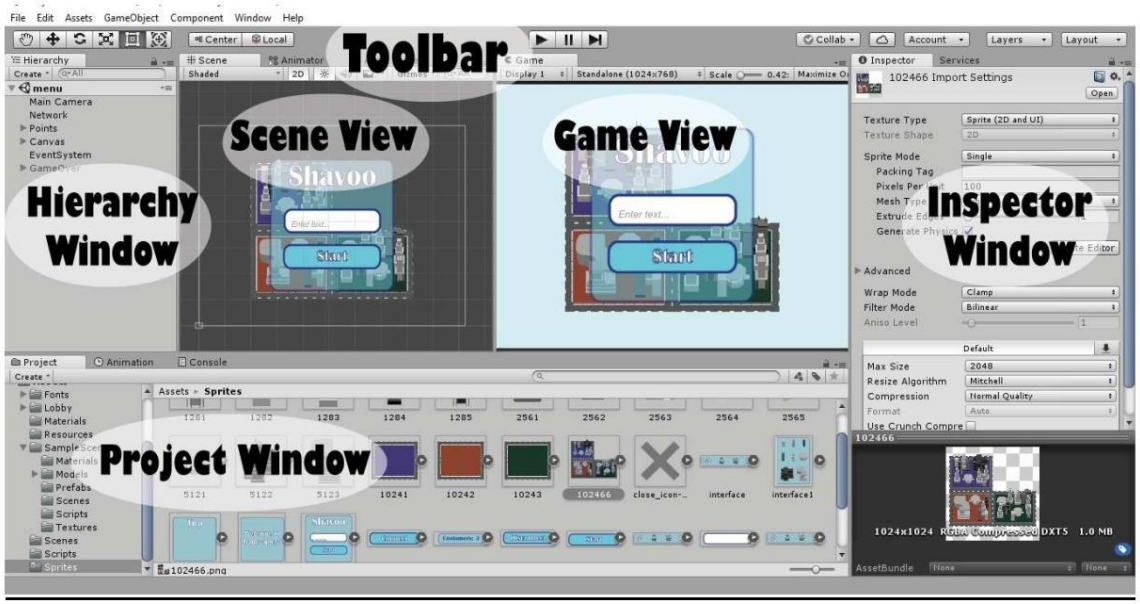


Рис. 4.1. Інтерфейс Unity

Project Window (Оглядач проекту). Ця частина інтерфейсу, щоб виконувати завдання ресурсами, які знаходяться в ігровому проекті.

У лівій частині оглядача міститься ієрархічний список, який відображає структуру папок проекту. Окремі ресурси представлені у вигляді іконок, що вказують їх тип (спрайт, скрипт і т.д.).

Hierarchy Window (Ієрархія). Вікно містить всі об'єкти, що знаходяться в сцені. Це можуть бути будь-які UI елементи (кнопки, картинки і т.д.), 3D моделі, екземпляри префабов (об'єктів) та інше. Кожен об'єкт (GameObject) може містити дочірні об'єкти або входити в об'єкти більш високого рангу. Можна вибрати об'єкт в ієрархії і перемістити його в інший об'єкт, таким чином сформується батьківська зв'язок (parenting). Дочірній об'єкт успадковує зміни свого батька, пов'язані з переміщенням, обертанням і масштабуванням.

Toolbar (Панель інструментів). У цій частині розташовуються елементи, потрібні для трансформації, кнопки запуску і зупинки гри, меню, що випадає, шари [54].

Scene View (Сцена). У цьому вікні встановлюється позиціонування елементів гри. Game View (Гра). В даному вікні відображається все, що побачить користувач.

Inspector Window (Інспектор). Інспектор відображає інформацію про конкретний вибраному об'єкті і його властивості. Тут можна змінювати функціонал об'єктів в сцені.

MonoDevelop - це інтегроване середовище розробки (IDE), що поставляється разом з Unity. IDE поєднує в собі функції текстового редактора з додатковими можливостями для налагодження і виконання інших завдань з управління проектами. [42]

MonoDevelop володіє всіма основними можливостями, необхідними для сучасної інтегрованої середовища розробки:

- настроюється підсвічування синтаксису;
- автоматичне доповнення коду;
- виділення блоків коду з можливістю їх згортання / розгортання;
- інтелектуальна робота з відступами в коді;

- можливості рефакторингу (перейменування класів і методів, автоматична реалізація інтерфейсів в похідних класах);

- зручна навігація по коду (навігація по класах, методам, властивостям);

- візуальний редактор форм для проектів на Gtk #;

- створення кількох розкладок інтерфейсу і перемикання між ними;

- безліч стандартних шаблонів проектів;

- можливість автоматичного створення бінарних пакетів і архівів після компіляції вихідного коду;

- робота з базами даних;

- створення додатків з GUI, що підтримує кілька мов;

- інтеграція з Subversion для управління вихідним кодом;

- підтримка NUnit для створення Unit-тестів;

- автоматичне створення документації;

- розширення можливостей за рахунок доповнень і зовнішніх інструментів;

- можливість інтеграції з Microsoft Visual Studio і .NET Framework (в середовищі Microsoft Windows). [44]

На рис. 4.2 представлений інтерфейс роботи програми.

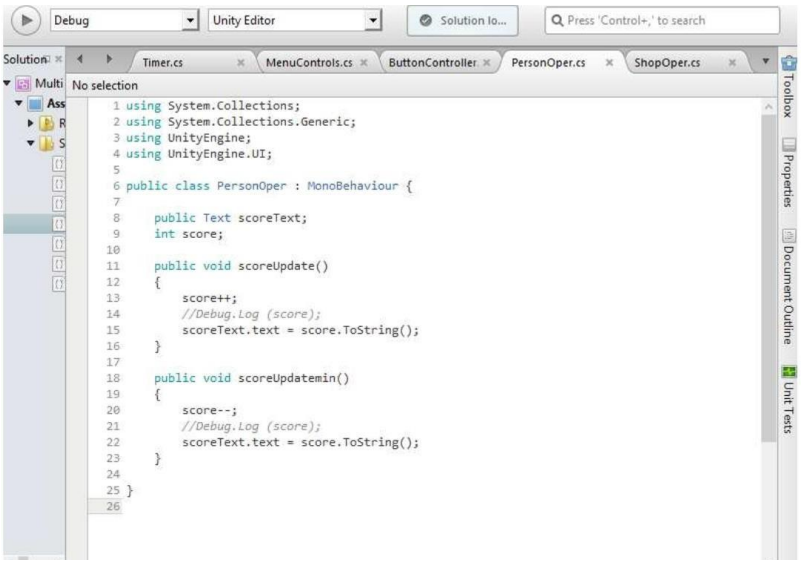


Рис. 4.2. Інтерфейс MonoDevelop

Мова програмування C #. C # (вимовляється Сі Шарп) - об'єктно-орієнтована мова програмування. Розроблено в 1998-2001 роках групою інженерів під керівництвом Андерса Хейлсберг в компанії Microsoft як мова розробки додатків для платформи Microsoft .NET Framework і згодом був стандартизований як ECMA-334 і ISO / IEC 23270.

C # відноситься до сім'ї мов з C-подібним синтаксисом, з них його синтаксис найбільш близький до C ++ і Java. Мова має статичну типізацію, підтримує поліморфізм, перевантаження операторів (в тому числі операторів явного і неявного приведення типу), делегати, атрибути, події, властивості, узагальнені типи і методи, ітератори, анонімні функції з підтримкою замикань, LINQ, виключення, коментарі у форматі XML.

Перейнявши багато від своїх попередників - мов C ++, Java, Delphi, Модула і Smalltalk - C #, спираючись на практику їх використання, виключає деякі моделі, що зарекомендували себе як проблематичні при розробці програмних систем, наприклад, C # на відміну від C ++ не підтримує множинне успадкування класів (між тим допускається множинне спадкування інтерфейсів). [62]

Дана мова був обраний в якості основного, так як має потрібні якості для реалізації, має вбудовану підтримку узагальнень, делегатів і подій, що полегшить реалізацію.

Unity використовує компонентний підхід. Компонент - це клас, успадкованих від MonoBehaviour. Один компонент повинен відповідати за одне поведе ня.

Скрипти можуть бути створені в панелі проектора. Для цього достатньо натиснути кнопку Create і вибрати мову, на якому буде створюватися скрипт. Прикріпити скрипт до об'єкта можна шляхом перетягування або натискання на кнопку Add Component.

Після того, як буде створено скрипт в Unity і відкриється за замовчуванням в MonoDevelop, можна буде побачити наступне:

using UnityEngine;

```
using System.Collections;
public class MainPlayer: MonoBehaviour {
// Use this for initialization
void Start () {
}
// Update is called once per frame void Update () {
}
}
```

Назва створеного скрипта має повністю відповідати назві створеного автоматично класу, який успадковується від вбудованого класу MonoBehaviour.

Функція Update - сама використовувана функція в Unity. Вона викликається один раз за кадр в кожному скрипті, що використовує її. Майже все, що вимагає змін або регуляції на постійній основі, прописується в цій функції. Переміщення нефізичних об'єктів, прості таймери і виявлення введення зазвичай реалізуються в цій функції. Варто зазначити, що ця функція не викликається на основному таймлайне. Якщо один кадр займає більше часу для обробки, ніж наступний, тоді час від одного виклику функції буде різним.

Функція Start - функція, яка викликається автоматично перед функцією Update, коли скрипт завантажений. Функцію можна використовувати для всього, що потрібно, тільки коли компонент скрипта включений. Це дозволяє відстрочити будь-яку частину коду ініціалізації, поки вона не знадобиться [62].

4.2. Створення автономно функціонуючих елементів (персонажів) гри

Створюється новий проект в середовищі Unity в версії 2019.4.28.

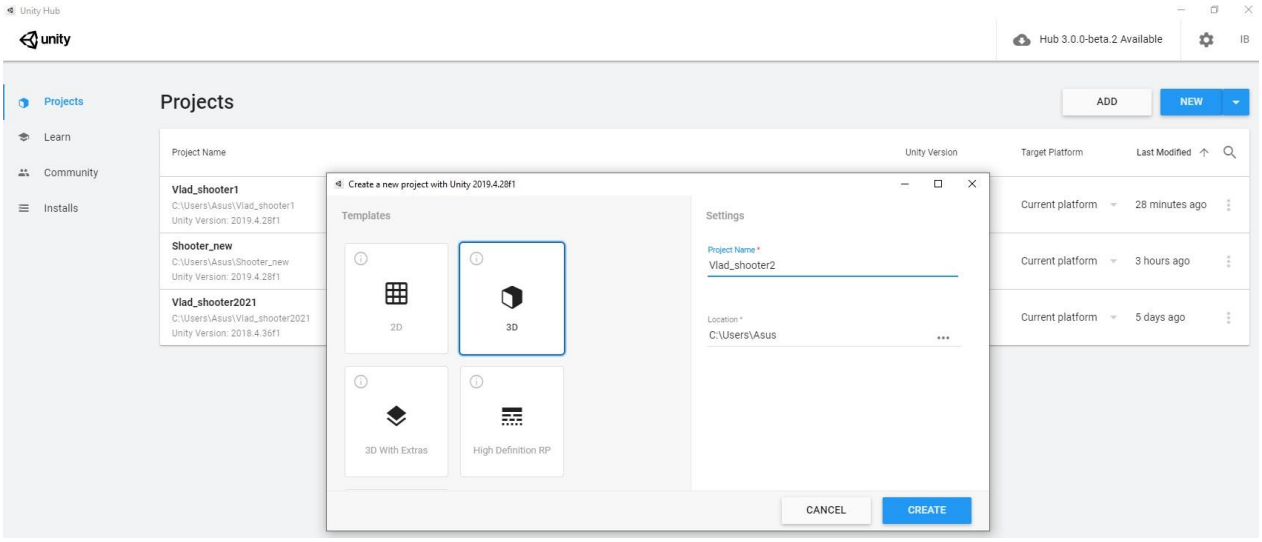


Рис. 4.4. Скріншот створення нового проекту Vlad_shooter2.

Після відкриття головного вікна проекту в програмі здійснюється перехід до Asset store, звідки до створеного проекту імпортується asset Let's Try Assets.

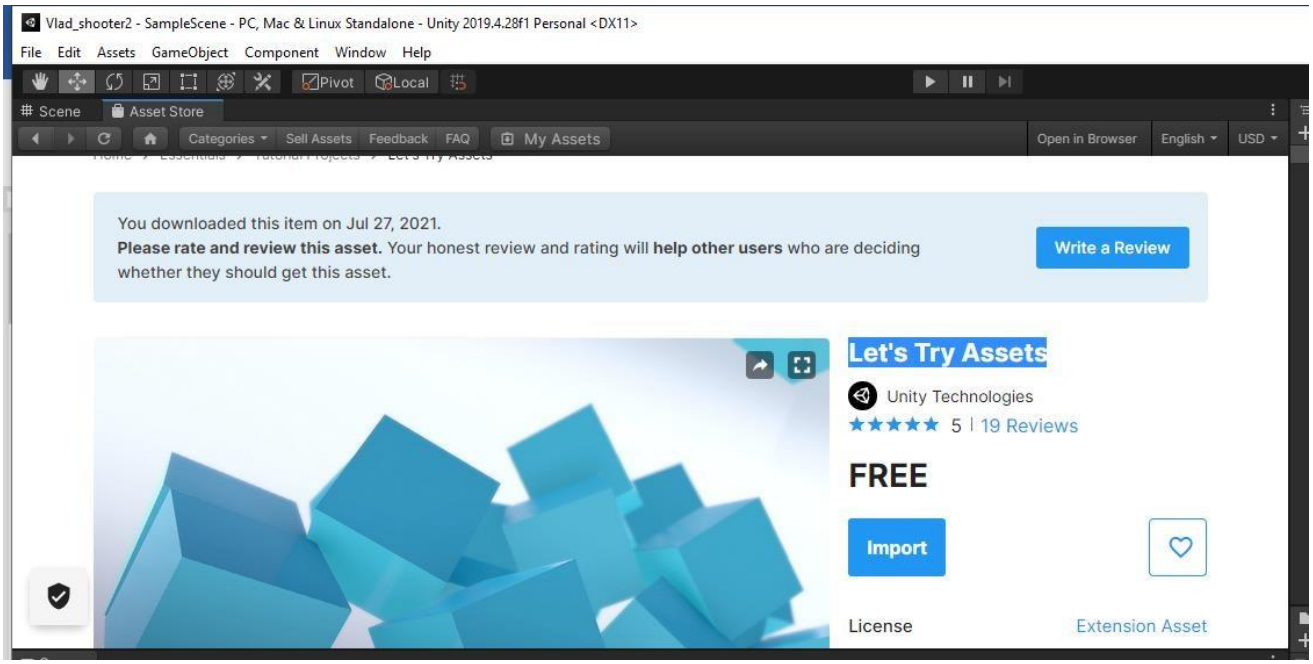


Рис. 4.5. Скріншот імпорту asset Let's Try Assets до проекту.

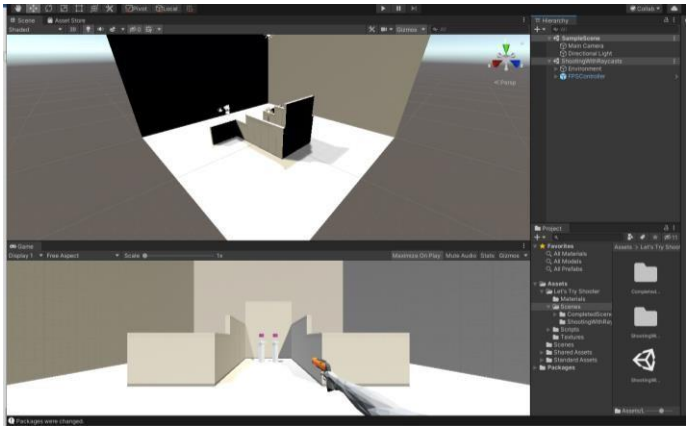


Рис. 4.6. Скріншот первинного вигляду проекту, побудованого з використанням Let’s Try Assets.

Даний asset складається з побудованої сцени Arena, налаштованого джерела світла Lighting та першої особи FPSController, яка містить в собі об’єкт Gun, що і є зброю першої особи в вигляді пістолету.

Здійснюється вхід до папки Scripts та створюється скрипт на мові програмування C# під назвою RaycastShooter та прикріплюється до складової об’єкту FPSController Gun. Саме Gun і є зброєю від першої особи.

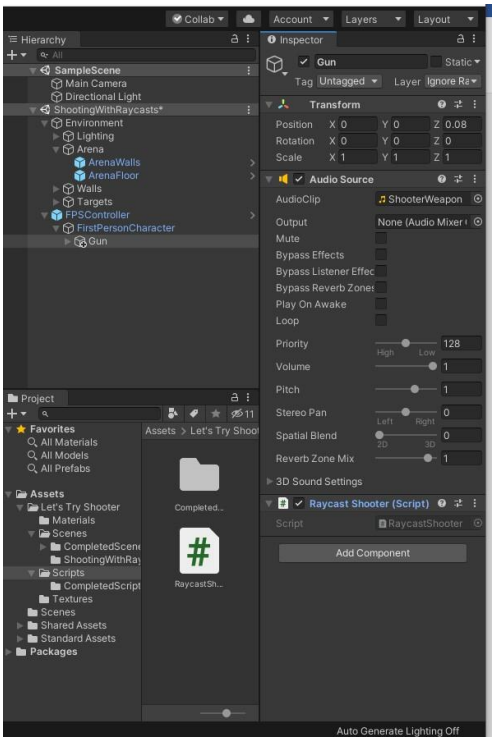


Рис. 4.7. Скріншот створення нового скрипту RaycastShooter.cs з прикріпленням його до об’єкту Gun.

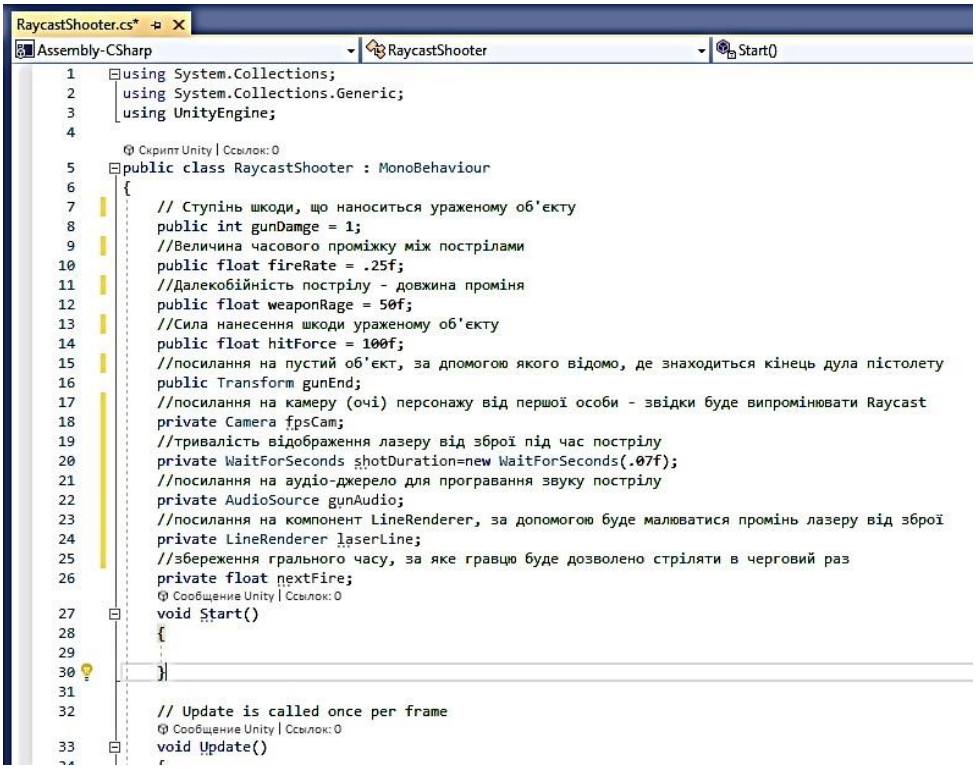


Рис. 4.8. Скріншот створення змінних зі статусом public та private в скрипті RaycastShooter

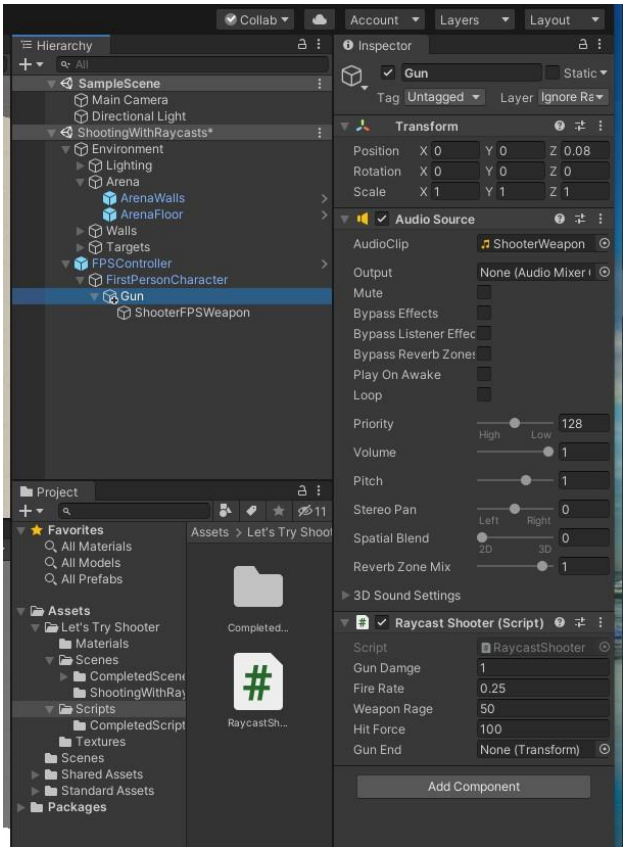


Рис. 4.9. Відображення значень змінних зі статусом public для об’єкту Gun

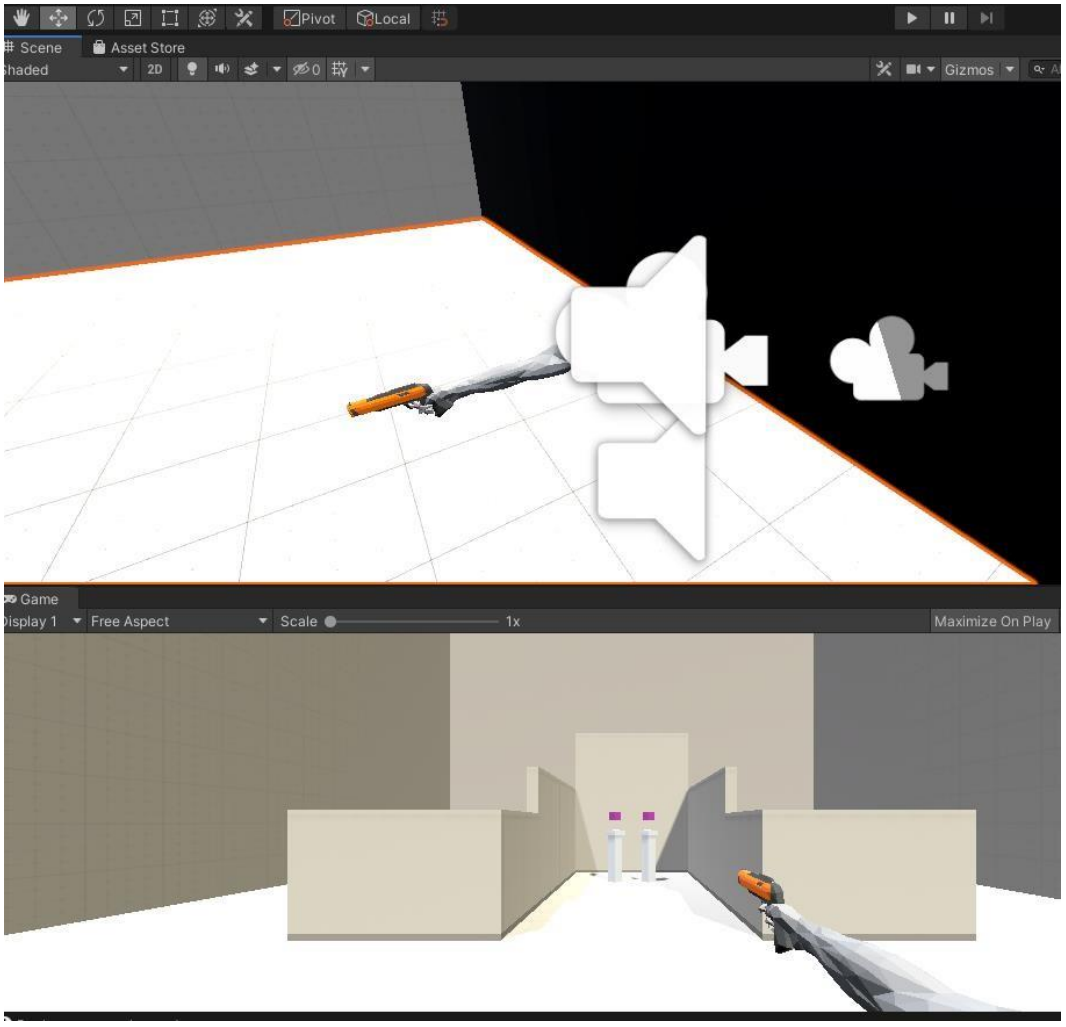


Рис. 4.10. Реальний вигляд в програмі об’єкту Gun, який є першою особою з рукою, що тримає зброю.

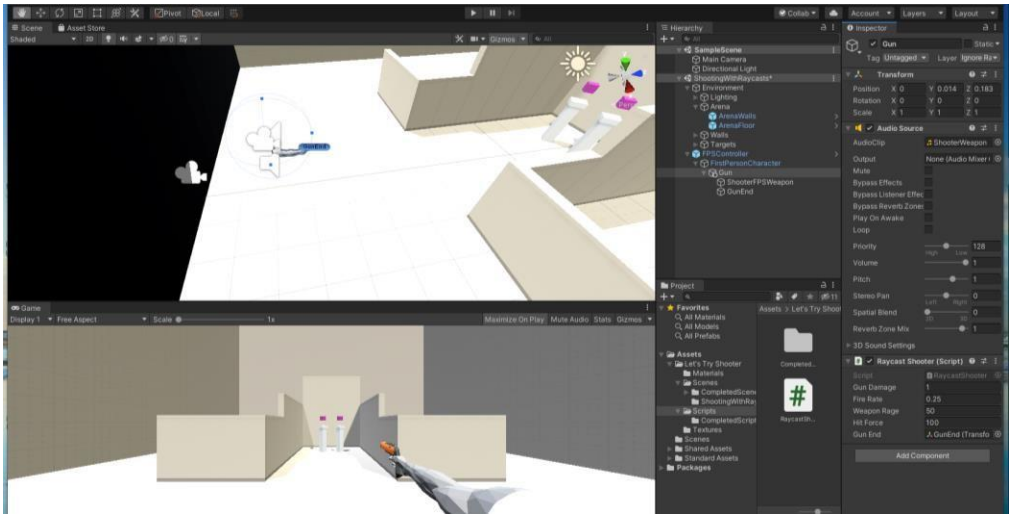


Рис. 4.11. Створення об’єкта GunEnd, що є кінцем стволу пістолета, та внесення його до скрипту.

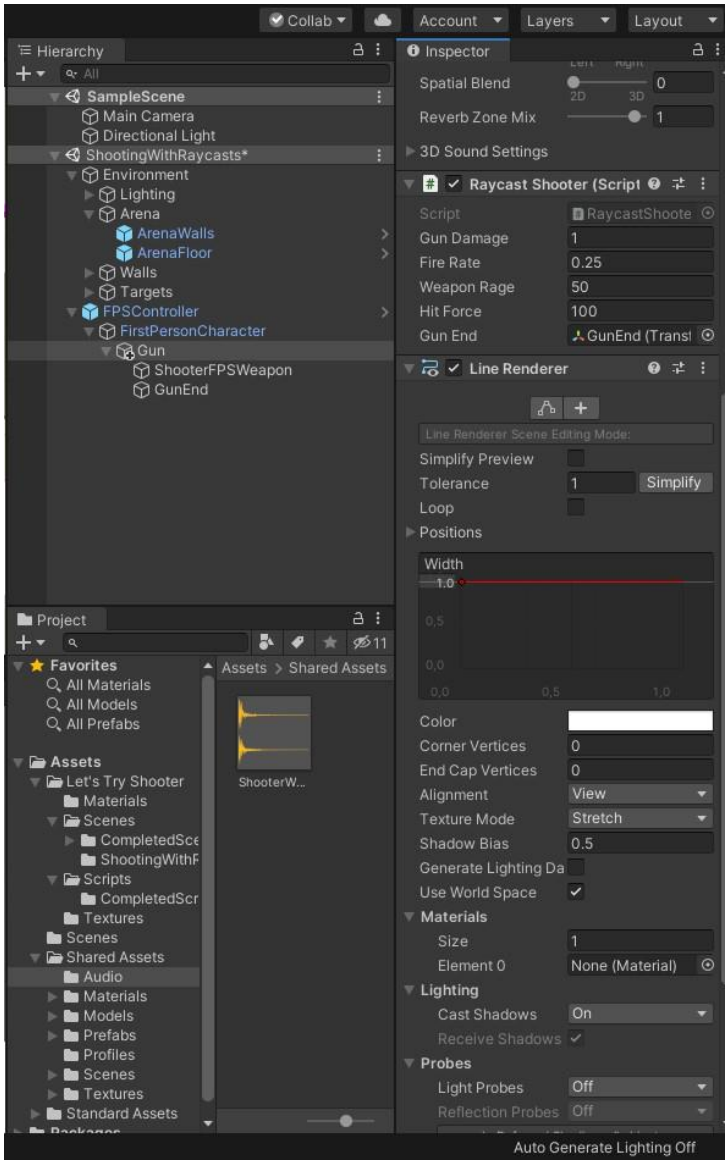


Рис. 4.12. Додавання нового компоненту LineRenderer до об’єкту Gun.

В файлі RaycastShooter.cs в методі Update прописуються налаштування кнопки Fire1 для керування пострілом:

```
void Update()
    {
        //За допомогою цього методу користувач знає, якою кнопкою користуватися для пострілу
        if (Input.GetButtonDown("Fire1") && Time.time>nextFire)
        {
            nextFire = Time.time + fireRate;
```

```
    }
}
```

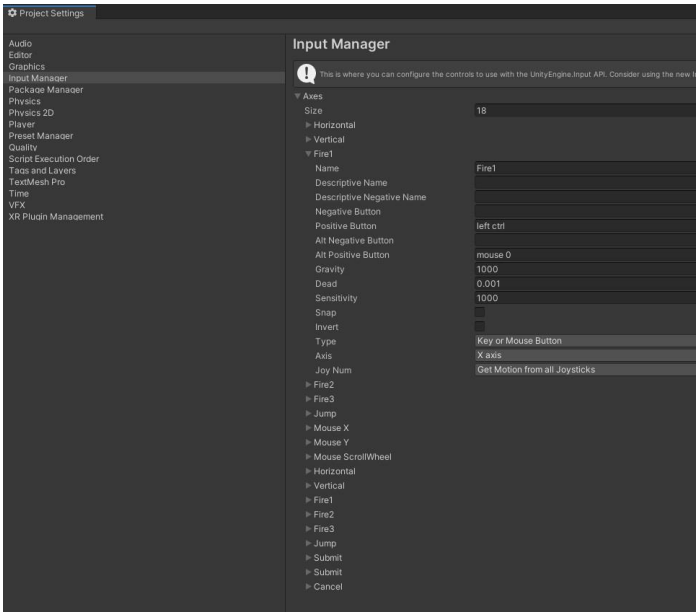


Рис. 4.13. Нові налаштування кнопки Fire1 за вкладкою edit/project settings/input manager.

Тепер постріли в грі можна виробляти за допомогою лівої кнопки комп’ютерної миші. Подальші перетворення в методі Update наступні.

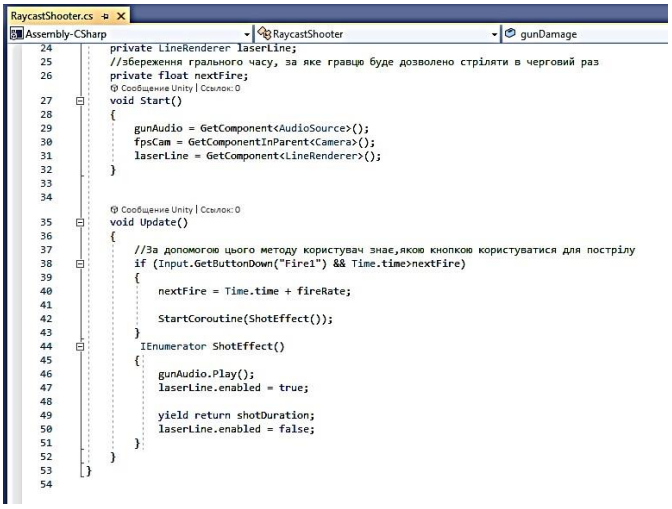


Рис. 4.14. Прописування налаштувань звуку пострілу та лазеру в файлі RaycastShooter.cs.

Додаємо зміни коду до методу Update в файлі RaycastShooter.cs:

```
//Код для відмаальовки лазерного променю
        Vector3 rayOrigin = fpsCam.ViewportToWorldPoint(new Vector3(.5f, .5f, 0));
        RaycastHit hit;
        laserLine.SetPosition(0, gunEnd.position);
        if(Physics.Raycast(rayOrigin,fpsCam.transform.forward, out hit, weaponRage))
        {
           laserLine.SetPosition(1, hit.point);
        }
```

Для реалізації процесу фізичного знищення мішеней внаслідок пострілів здійснюємо таке дописування коду до методу Update в файлі RaycastShooter.cs з впровадженням властивостей об’єкту типу ShootableBox:

```
//Впровадження методу руйнування цілей до загального методу
            ShootableBox health = hit.collider.GetComponent<ShootableBox>();
            if (health != null)
            {
               health.Damage(gunDamage);
            }
            if (hit.rigidbody != null)
            {
               hit.rigidbody.AddForce(-hit.normal * hitForce);
            }
```

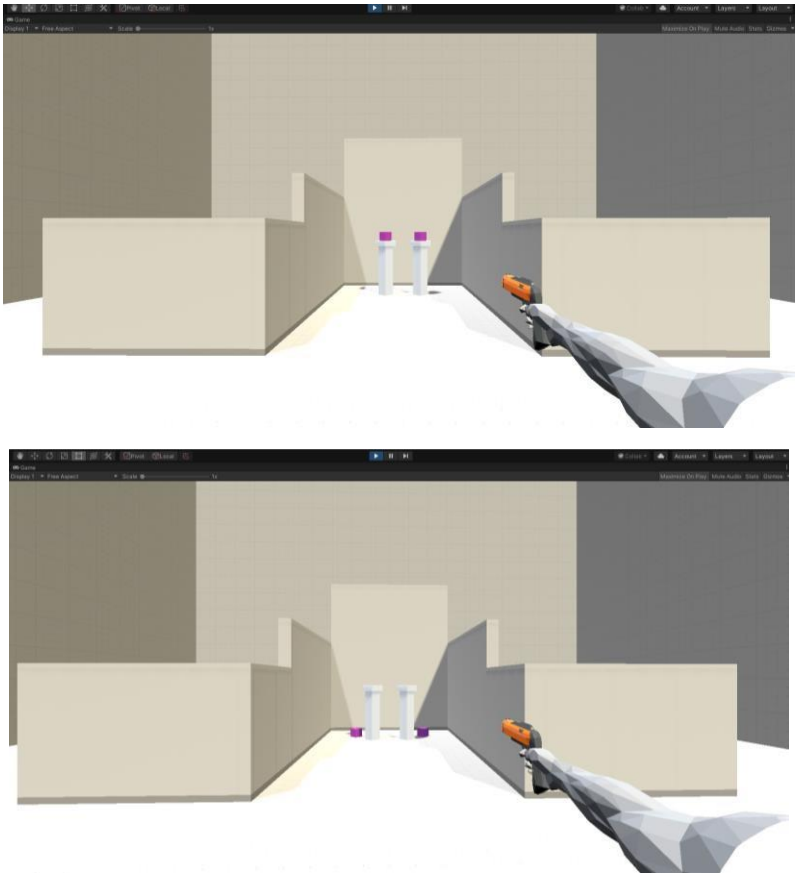


Рис. 4.15. Скріншот здійснення тестових пострілів, як результат відбувається фізичне падіння мішеней, що є об’єктами ShootableBox.

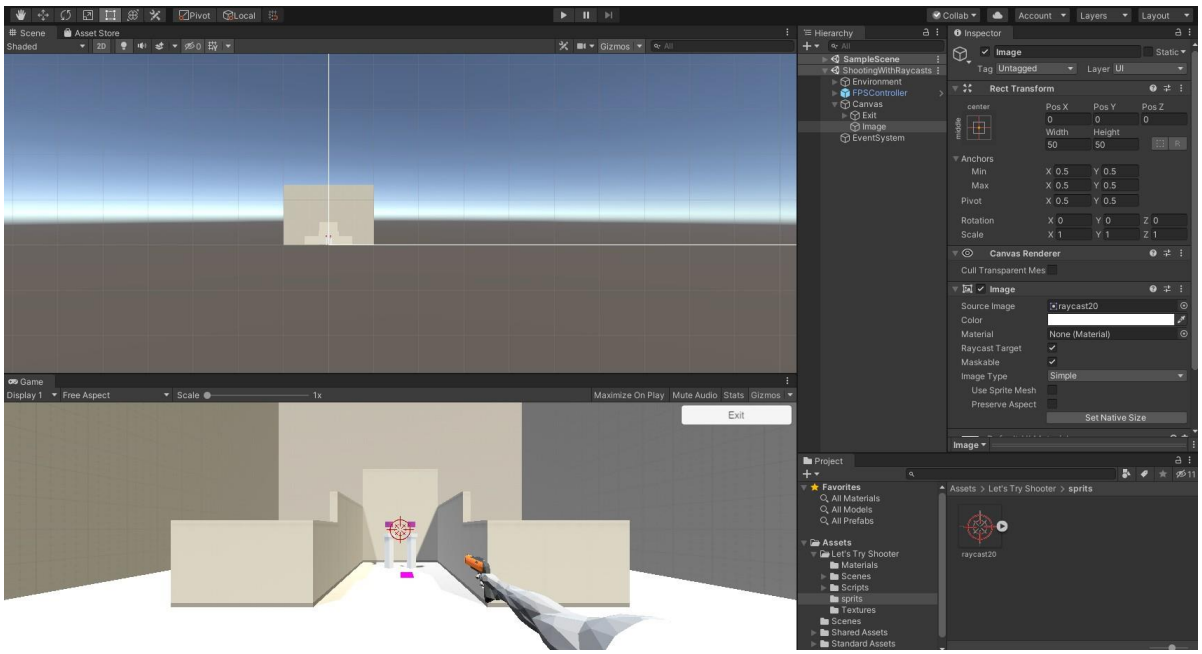


Рис. 4.16. Скріншот додавання статичного прицілу для зброї шляхом додавання підготовленої картинки прицілу та перетворення її до спрайту.

ВИСНОВКИ

В першому розділі був здійснений огляд досліджень, які здійснювалися в звязку з розробкою ШІ в шутерах від першої особи.

В другому розділі розглянуті численні алгоритми, що застосовуються сьогодні для розробки ШІ в комп’ютерних іграх. Практична розробка більшості комерційних ігор продовжує базуватися на детермінованих методах, методах кінцевих автоматів та деревах прийняття рішень. Великі вимоги до наповненості простору ігор складними 3D-об’єктами, що призводить до великого завантаження пам’яті гри, обмежують використання таких дещо екзотичних методів, як еволюційні алгоритми, нейронні мережі та нечітка логіка. Далі наведений огляд найбільш актуальних на сьогодні шутерів від першої особи лише підтверджує великі комерційні вимоги до наповненості простору гри складною та насиченою відеографікою.

Для розробки шутера від першої особи обране програмне середовище 3D-Unity з мовою програмування C#. Розробка алгоритмів здійснювалася методом кінцевих автоматів. Ігровий додаток розрахований на його використання одним гравцем. Гра починається з ігрового поля, в центрі якого розміщується керований об'єкт (пістолет Glock з рукою гравця).

Безпосередньо переможні умови відсутні. Мета гравця полягає в тому, щоб якомога більше набрати очок за знищення кубиків-мішеней, що знаходяться на спеціальних стовпчиках посеред приміщення. Гра завершується, якщо гравець знищив всі мішені. Нові мішені з’являються лише в зв’язку з новим входом гравця до гри.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mat Buckland. Programming Game AI by Example. Jones & Bartlett Publishers, 1 edition, 2004. P. 36-70.

2. Ian Millington. Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology).Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. P. 371-395.

3. Alan Turing. Computing machinery and intelligence. Mind LIX, 1950. P. 2.

4. Daniel Johnson and Janet Wiles. Computer games with intelligence. In In Procs. 10th IEEE Intl Conf. on Fuzzy Systems, IEEE, 2001. P. 61–68.

5. The NetBSD Foundation, "Portability and supported hardware platforms," http://netbsd.org/about/portability.html.

6. Microsoft, Windows NT Hardware Abstraction Layer (HAL), http://support.microsoft.com/kb/99588.

7. A. Nareyek, N. Combs, B. Karlsson, S. Mesdaghi, and I. Wilson, "The 2003 report of the IGDA's artificial intelligence interface standards committee," Tech. Rep., International Game Developers Association, 2003, http://www.igda.org/ai/report-2003/report-2003.html, http://archive.org/web/.

8. A. Nareyek, N. Combs, B. Karlsson, S. Mesdaghi, and I. Wilson, "The 2004 report of the IGDA's artificial intelligence inter¬face standards committee," Tech. Rep., International Game Developers Association, 2004, http://www.igda.org/ai/report-2004/report-2004.html.

9. A. Nareyek, N. Combs, B. Karlsson, S. Mesdaghi, and I. Wilson, "The 2005 report of the IGDA's artificial intelligence inter¬face standards committee," Tech. Rep., International Game Developers Association, 2005, http://www.igda.org/ai/report-2005/report-2005.html, http://archive.org/web/.

10. B. Yue and P. de Byl, "The state of the art in game AI standard¬isation," in Proceedings of the 2006 International Conference on Game Research and Development, pp. 41-46, Murdoch Univer¬sity., 2006.

11. B. F. F. Karlsson, "Issues and approaches in artificial intelligence middleware development for digital games and entertainment products," CEP 50740:540, 2003.

12. C. Berndt, I. Watson, and H. Guesgen, "OASIS: an open AI standard interface specification to support reasoning, repre¬sentation and learning in computer games," in Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games (IJCAI '05), pp. 19-24, 2005.

13. Unity Technologies, "Unity—Game Engine," http://unity3d .com/.

14. Epic Games, Unreal Engine Technology - Home, https://www .unrealengine.com/.

15. Crytek, CRYENGINE: The complete solution for next genera¬tion game development by Crytek, http://cryengine.com/.

16. Havok, http://www.havok.com/.

17. B. Kreimeier, The case for game design patterns, 2002, http:// www.gamasutra.com/view/feature/132649/me_case Jor^ame. design_patterns.php?print=1.

18. S. Bjork, L. Sus, and H. Jussi, "Game design patterns," in Pro¬ceedings of the Level Up-lst International Digital Games Research Conference,Utrecht, TheNetherlands, November2003.

19. S. Bjork and J. Holopainen, "Describing games—an interaction-centric structural framework," in Level Up: Proceedings of Digital Games Research Conference,2003.

20. C. M. Olsson, S. Bjork, and S. Dahlskog, "The conceptual rela¬tionship model: understanding patterns and mechanics in game design," in Proceedings ofthe DiGRA International Conference (DiGRA 14),2014.

21. A B Loyall and J Bates, "Hap: a reactive, adaptive architecture for agents," Tech. Rep. CMU-CS-97-123, Carnegie Mellon University, School of Computer Science, 1991.

22. M. Mateas and A. Stern, "A behavior language for story-based believable agents," IEEE Intelligent Systems and Their Applications,vol.17,no.4,pp.39-47,2002.

23. M. Mateas and A. Stern, "A behavior language: joint action and behavioral idioms," in Life-Like Characters, Cognitive Technolo¬gies, pp. 135-161, Springer, Berlin, Germany, 2004. International Journal of Computer Games Technology

24. J. D. Funge, "Making them behave: cognitive models for com¬puter animation," 1998.

25. J. Funge, X. Tu, and D. Terzopoulos, "Cognitive modeling: knowledge, reasoning and planning for intelligent charac¬ters," in Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques,pp. 29-38, ACM Press/Addison-Wesley, 1999.

26. J. Funge, "Representing knowledge within the situation calculus using interval-valued epistemic fluents," Reliable Computing, vol. 5, no. 1, pp. 35-61,1999.

27. J. Orkin, "Symbolic representation of game world state: toward real-time planning in games," in Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence,2004.

28. J. Orkin, "Agent architecture considerations for real-time plan¬ning in games," in Proceedings of the Artificial Intelligence and Interactive Digital Entertainment (AIIDE '05), pp. 105-110,2005.

29. E. F. Anderson, "Scripting behaviour—towards a new language for making NPCs act intelligently," in Proceedings of the zfx-CON05 2nd Conference on Game Development,2005.

30. E. F. Anderson, "SEAL—a simple entity annotation language," in Proccedings of zfxCON05-2nd Conference on Game Develop-ment,pp. 70-73, Stefan Zerbst, Braunschweig, Germany, 2005.

31. E. F. Anderson, "Scripted smarts in an intelligent virtual envi¬ronment," in Proceedings of theConferenceonFuturePlay: Research, Play, Share, pp. 185-188, ACM, 2008.

32. D. C. Cheng and R. Thawonmas, "Case-based plan recognition for real-time strategy games," in Proceedings ofthe 5th Inter¬national Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE 04),pp. 36-40, 2004.

33. D. W. Aha, M. Molineaux, andM.J.V. Ponsen, "Learningto win: case-based plan selection in a real-time strategy game," in Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR 05),pp. 5-20, August 2005.

34. S. Ontanon, K. Mishra, N. Sugandh, andA. Ram, "Case-based planning and execution for real-time strategy games," in Case-Based Reasoning Research and Development: 7th International Conference on Case-Based Reasoning, ICCBR 2007 Belfast, Northern Ireland, UK, August13-16,2007Proceedings, vol. 4626, pp. 164-178, Springer, Berlin, Germany, 2007.

35. B. Weber and M. Mateas, "Conceptual neighborhoods for retrieval in case-based reasoning," in Proceedings of the 8th International Conference on Case-Based Reasoning (ICCBR '09), pp. 343-357, 2009.

36. B. G. Weber and M. Mateas, "Case-based reasoning for build order in real-time strategy games," in Proceedings of the 5th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE '09), pp. 106-111, October 2009.

37. M. Sharma,M.Holmes, J. Santamaria,A.Irani,C.Isbell, and A. Ram, "Transfer learning in real-time strategy games using hybrid CBR/RL," in Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI '07), pp. 1041-1046, January 2007.

38. S. Lee-Urban, H. Munoz-Avila, A. Parker, U. Kuter, and D. Nau, "Transfer learning of hierarchical task-network planning methods in a real-time strategy game," in Proceedings of the 17th International Conference on Automated Planning & Scheduling (ICAPS '07), Workshop on AI Planning and Learning (AIPL), 2007.

39. D. Shapiro, T. Konik, and P. O'Rorke, "Achieving far transfer in an integrated cognitive architecture," in Proceedings ofthe 23rd National Conference on Artificial Intelligence (AAAI '08), pp.1325-1330,July2008.

40. Ламот, Андре. Программирование игр для Windows. Советы профессионала: Пер. с англ. - М. : Издательский дом “Вильямс”, 2018. - 880 с.

41. Borromeo N.A. Hands-On Unity 2020 Game Development: Build, customize, and optimize professional games using Unity 2020 and C# - Packt, 2020. - 580p.

42. Geig M. Unity 2018 Game Development in 24 Hours, Sams Teach Yourself, - Pearson, 2018 – 450 p.

43. Hocking J. Unity in Action: Multiplatform Game Development in C# - Amazon, 2018 – 735 p.

44. Паласиос Х. Unity 5.x. Программирование искусственного интеллекта в играх: пер. с англ. Р. Н. Рагимова. - М.: ДМК П ресс, 2017. - 272 с.

45. Smith M., Queiroz C. Unity 5.x Cookbook: More than 100 solutions to build amazing 2D and 3D games with Unity - Amazon, 2015. – 786 p.

46. Sumner J., Gibson W. Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game - - Amazon, 2014. – 533 p.

47. Lukosek G. Learning C# by Developing Games with Unity 5.x – Amazon, 2016. – 677 p.

48. Aversa D., Kyaw A.S., Peters C. Unity Artificial Intelligence Programming: Add powerful, believable, and fun AI entities in your game with the power of Unity – Amazon, 2018. – 702 p.

49. E. Hastings, R. Guha, and K. Stanley. Automatic content generation in the galactic arms race video game. In Proceedings of the 5th international conference on Computational Intelligence and Games,CIG’09, 2009. P. 5.

50. Lofti A. Zadeh. Fuzzy sets and systems. System Theory,1965.

51. Jiljang Wang Gabriyel Wong. A fuzzy-control approach to managing scene complexity. Game Programming Gems 6, 2006. P. 5-7.

52. Larry O’Brien. Fuzzy logic in games. Game Developer Magazine, 1996. P. 6-7.

53. Mason McCuskey. Fuzzy logic for video games. Game Programming Gems 1, 2000. P. 7-8.

54. Michael Zarozinski. Imploding combinatorial explosion in a fuzzy system. Game Programming Gems 2, 2001 P 342-350.

55. William E. Combs. The Fuzzy Systems Handbook 2nd Ed, Academic. 1999. P 48-50.

56. Penelope Sweetser and Janet Wiles. Current ai in games: a review. Australian Journal of Intelligent Information Processing Systems, 2002 P. 24-42.

57. Алекс Дж. Шампандар. Искусственный интеллект в компьютерных играх. Москва. «Вильямс», 2007 стр. 450-478.

58. Копытчук И. Н., Тишин П. М. Модели, методы и информационная технология повышения точности регистрации массы движущихся грузов в тензометрических системах. Одеса - 2016 стр.35-46.

59. Борисов В.В., Круглов В.В., Федулов А.С. Нечеткие модели и сети. //Телеком – 2007г. 230 с.

60. Бутов Б.В, Тишин П.М., Шапорин В.О. Исследование использования нечеткой логики для искусственного интеллекта в играх //Автоматизація технологічних і бізнес-процесів - Volume 10 - Issue 4 /2018 – С. 66-73.

61. Elias H. First Person Shooter. The Subjective Cyberspace. - Covilhã, Portugal: LabCom Books, 2009 – 196 p.

62. Гибсон Б. Д. Unity и C#. Геймдев от идеи до реализации. 2-е изд. - СПб.: Питер, 2019. - 928 с.

63. https://docs.unity3d.com/ru/2019.4/Manual/UnityManual.html

ДОДАТОК А

Скрипти

```
using UnityEngine;
using System.Collections;


public class RayCastShootComplete : MonoBehaviour {

    public int gunDamage =
1;                                          // Set the number
of hitpoints that this gun will take away from shot objects with
a health script
    public float fireRate =
0.25f;                                      // Number in seconds
which controls how often the player can fire
    public float weaponRange =
50f;                                     // Distance in Unity
units over which the player can fire
    public float hitForce =
100f;                                       // Amount of force
which will be added to objects with a rigidbody shot by the
player
    public Transform
gunEnd;                                                 // Holds a
reference to the gun end object, marking the muzzle location of
the gun


    private Camera
fpsCam;                                                  // Holds a
reference to the first person camera
```

```
    private WaitForSeconds shotDuration = new
WaitForSeconds(0.07f);    // WaitForSeconds object used by our
ShotEffect coroutine, determines time laser line will remain
visible
    private AudioSource
gunAudio;                                        // Reference to
the audio source which will play our shooting sound effect
    private LineRenderer
laserLine;                                      // Reference to
the LineRenderer component which will display our laserline
    private float
nextFire;                                              // Float
to store the time the player will be allowed to fire again,
after firing


    void Start ()
    {
        // Get and store a reference to our LineRenderer
component
        laserLine = GetComponent<LineRenderer>();

        // Get and store a reference to our AudioSource
component
        gunAudio = GetComponent<AudioSource>();

        // Get and store a reference to our Camera by searching
this GameObject and its parents
        fpsCam = GetComponentInParent<Camera>();
```

```
    }


    void Update ()
    {
        // Check if the player has pressed the fire button and
if enough time has elapsed since they last fired
        if (Input.GetButtonDown("Fire1") && Time.time >
nextFire)
        {
            // Update the time when our player can fire next
            nextFire = Time.time + fireRate;

            // Start our ShotEffect coroutine to turn our laser
line on and off
            StartCoroutine (ShotEffect());

            // Create a vector at the center of our camera's
viewport
            Vector3 rayOrigin = fpsCam.ViewportToWorldPoint (new
Vector3(0.5f, 0.5f, 0.0f));

            // Declare a raycast hit to store information about
what our raycast has hit
            RaycastHit hit;

            // Set the start position for our visual effect for
our laser to the position of gunEnd
            laserLine.SetPosition (0, gunEnd.position);
```

```
            // Check if our raycast has hit anything
            if (Physics.Raycast (rayOrigin,
fpsCam.transform.forward, out hit, weaponRange))
            {
                // Set the end position for our laser line
                laserLine.SetPosition (1, hit.point);

                // Get a reference to a health script attached
to the collider we hit
                ShootableBox health =
hit.collider.GetComponent<ShootableBox>();

                // If there was a health script attached
                if (health != null)
                {
                    // Call the damage function of that script,
passing in our gunDamage variable
                    health.Damage (gunDamage);
                }

                // Check if the object we hit has a rigidbody
attached
                if (hit.rigidbody != null)
                {
                    // Add force to the rigidbody we hit, in the
direction from which it was hit
                    hit.rigidbody.AddForce (-hit.normal *
hitForce);
```

```
                }
            }
            else
            {
                // If we did not hit anything, set the end of the line to a position directly in front of the camera at the distance of weaponRange
                laserLine.SetPosition (1, rayOrigin + (fpsCam.transform.forward * weaponRange));
            }
        }
    }

    private IEnumerator ShotEffect()
    {
        // Play the shooting sound effect
        gunAudio.Play ();
        // Turn on our line renderer
        laserLine.enabled = true;
        //Wait for .07 seconds
        yield return shotDuration;

        // Deactivate our line renderer after waiting
        laserLine.enabled = false;
    }
}
```

RaycastShootComplete.cs

```
using UnityEngine;
using System.Collections;

public class RayViewerComplete : MonoBehaviour {

    public float weaponRange = 50f;                         //
Distance in Unity units over which the Debug.DrawRay will be
drawn

    private Camera fpsCam;                                  //
Holds a reference to the first person camera


    void Start ()
    {
        // Get and store a reference to our Camera by searching
this GameObject and its parents
        fpsCam = GetComponentInParent<Camera>();
    }

    void Update ()
    {
        // Create a vector at the center of our camera's
viewport
        Vector3 lineOrigin = fpsCam.ViewportToWorldPoint(new
Vector3(0.5f, 0.5f, 0.0f));
```

```
        // Draw a line in the Scene View from the point
lineOrigin in the direction of fpsCam.transform.forward *
weaponRange, using the color green
        Debug.DrawRay(lineOrigin, fpsCam.transform.forward *
weaponRange, Color.green);
    }
}
```

RayViewerComplete.cs

```
using UnityEngine;
using System.Collections;

public class ShootableBox : MonoBehaviour {

    //The box's current health point total
    public int currentHealth = 3;
    public static int score;
    public void Damage(int damageAmount)
    {
        //subtract damage amount when Damage function is called
        currentHealth -= damageAmount;

        //Check if health has fallen below zero
        if (currentHealth <= 0)
        {
            //if health has fallen below zero, deactivate it
            gameObject.SetActive (false);
            score++;
            Debug.Log(score);
```

```
        }
    }
}
```

ShootableBox.cs

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

public class ExitMenu : MonoBehaviour
{

   public void ExitGame()
    {
            Application.Quit();    // закрыть приложение
    }
}
```

ExitMenu.cs

```
using UnityEngine;

namespace UnityStandardAssets.Utility
{
    public class SmoothFollow : MonoBehaviour
    {
        //disable warning about unassigned variable, as it is
assigned by the serialization.
```

```
#pragma warning disable CS0649
        // The target we are following
        [SerializeField]
        private Transform target;
        // The distance in the x-z plane to the target
        [SerializeField]
        private float distance = 10.0f;
        // the height we want the camera to be above the target
        [SerializeField]
        private float height = 5.0f;

        [SerializeField]
        private float rotationDamping;
        [SerializeField]
        private float heightDamping;
#pragma warning restore CS0649

        // Use this for initialization
        void Start() { }

        // Update is called once per frame
        void LateUpdate()
        {
            // Early out if we don't have a target
            if (!target)
                return;

            // Calculate the current rotation angles
            var wantedRotationAngle = target.eulerAngles.y;
```

```
            var wantedHeight = target.position.y + height;

            var currentRotationAngle = transform.eulerAngles.y;
            var currentHeight = transform.position.y;

            // Damp the rotation around the y-axis
            currentRotationAngle =
Mathf.LerpAngle(currentRotationAngle, wantedRotationAngle,
rotationDamping * Time.deltaTime);

            // Damp the height
            currentHeight = Mathf.Lerp(currentHeight,
wantedHeight, heightDamping * Time.deltaTime);

            // Convert the angle into a rotation
            var currentRotation = Quaternion.Euler(0,
currentRotationAngle, 0);

            // Set the position of the camera on the x-z plane
to:
            // distance meters behind the target
            transform.position = target.position;
            transform.position -= currentRotation *
Vector3.forward * distance;

            // Set the height of the camera
            transform.position = new
Vector3(transform.position.x ,currentHeight ,
transform.position.z);
```

```
            // Always look at the target
            transform.LookAt(target);
        }
    }
}
```

SmoothFollow.cs

```
using System;
using UnityEngine;


namespace UnityStandardAssets.Utility
{
    public class FollowTarget : MonoBehaviour
    {
        public Transform target;
        public Vector3 offset = new Vector3(0f, 7.5f, 0f);


        private void LateUpdate()
        {
            transform.position = target.position + offset;
        }
    }
}
```

FollowTarget.cs

```
using System;
using UnityEngine;
```

```
using UnityStandardAssets.CrossPlatformInput;
using UnityStandardAssets.Utility;
using Random = UnityEngine.Random;

namespace UnityStandardAssets.Characters.FirstPerson
{
    [RequireComponent(typeof (CharacterController))]
    [RequireComponent(typeof (AudioSource))]
    public class FirstPersonController : MonoBehaviour
    {

        //disable warning about unassigned variable, as it is
assigned by the serialization.
#pragma warning disable CS0649
        [SerializeField] private bool m_IsWalking;
        [SerializeField] private float m_WalkSpeed;
        [SerializeField] private float m_RunSpeed;
        [SerializeField] [Range(0f, 1f)] private float
m_RunstepLenghten;
        [SerializeField] private float m_JumpSpeed;
        [SerializeField] private float m_StickToGroundForce;
        [SerializeField] private float m_GravityMultiplier;
        [SerializeField] private MouseLook m_MouseLook;
        [SerializeField] private bool m_UseFovKick;
        [SerializeField] private FOVKick m_FovKick = new
FOVKick();
        [SerializeField] private bool m_UseHeadBob;
        [SerializeField] private CurveControlledBob m_HeadBob =
new CurveControlledBob();
```

```
        [SerializeField] private LerpControlledBob m_JumpBob =
new LerpControlledBob();
        [SerializeField] private float m_StepInterval;
        [SerializeField] private AudioClip[]
m_FootstepSounds;    // an array of footstep sounds that will be
randomly selected from.
        [SerializeField] private AudioClip
m_JumpSound;           // the sound played when character leaves
the ground.
        [SerializeField] private AudioClip
m_LandSound;           // the sound played when character
touches back on ground.
#pragma warning restore CS0649

        private Camera m_Camera;
        private bool m_Jump;
        private float m_YRotation;
        private Vector2 m_Input;
        private Vector3 m_MoveDir = Vector3.zero;
        private CharacterController m_CharacterController;
        private CollisionFlags m_CollisionFlags;
        private bool m_PreviouslyGrounded;
        private Vector3 m_OriginalCameraPosition;
        private float m_StepCycle;
        private float m_NextStep;
        private bool m_Jumping;
        private AudioSource m_AudioSource;

        // Use this for initialization
```

```
        private void Start()
        {
            m_CharacterController =
GetComponent<CharacterController>();
            m_Camera = Camera.main;
            m_OriginalCameraPosition =
m_Camera.transform.localPosition;
            m_FovKick.Setup(m_Camera);
            m_HeadBob.Setup(m_Camera, m_StepInterval);
            m_StepCycle = 0f;
            m_NextStep = m_StepCycle/2f;
            m_Jumping = false;
            m_AudioSource = GetComponent<AudioSource>();
            m_MouseLook.Init(transform , m_Camera.transform);
        }


        // Update is called once per frame
        private void Update()
        {
            RotateView();
            // the jump state needs to read here to make sure it
is not missed
            if (!m_Jump)
            {
                m_Jump =
CrossPlatformInputManager.GetButtonDown("Jump");
            }
```

```
            if (!m_PreviouslyGrounded &&
m_CharacterController.isGrounded)
            {
                StartCoroutine(m_JumpBob.DoBobCycle());
                PlayLandingSound();
                m_MoveDir.y = 0f;
                m_Jumping = false;
            }
            if (!m_CharacterController.isGrounded && !m_Jumping
&& m_PreviouslyGrounded)
            {
                m_MoveDir.y = 0f;
            }

            m_PreviouslyGrounded =
m_CharacterController.isGrounded;
        }


        private void PlayLandingSound()
        {
            m_AudioSource.clip = m_LandSound;
            m_AudioSource.Play();
            m_NextStep = m_StepCycle + .5f;
        }


        private void FixedUpdate()
        {
            float speed;
```

```
            GetInput(out speed);
            // always move along the camera forward as it is the
direction that it being aimed at
            Vector3 desiredMove = transform.forward*m_Input.y +
transform.right*m_Input.x;

            // get a normal for the surface that is being
touched to move along it
            RaycastHit hitInfo;
            Physics.SphereCast(transform.position,
m_CharacterController.radius, Vector3.down, out hitInfo,
                               m_CharacterController.height/2f,
~0, QueryTriggerInteraction.Ignore);
            desiredMove = Vector3.ProjectOnPlane(desiredMove,
hitInfo.normal).normalized;

            m_MoveDir.x = desiredMove.x*speed;
            m_MoveDir.z = desiredMove.z*speed;


            if (m_CharacterController.isGrounded)
            {
                m_MoveDir.y = -m_StickToGroundForce;

                if (m_Jump)
                {
                    m_MoveDir.y = m_JumpSpeed;
                    PlayJumpSound();
                    m_Jump = false;
```

```
                    m_Jumping = true;
                }
            }
            else
            {
                m_MoveDir +=
Physics.gravity*m_GravityMultiplier*Time.fixedDeltaTime;
            }
            m_CollisionFlags =
m_CharacterController.Move(m_MoveDir*Time.fixedDeltaTime);

            ProgressStepCycle(speed);
            UpdateCameraPosition(speed);

            m_MouseLook.UpdateCursorLock();
        }


        private void PlayJumpSound()
        {
            m_AudioSource.clip = m_JumpSound;
            m_AudioSource.Play();
        }


        private void ProgressStepCycle(float speed)
        {
            if (m_CharacterController.velocity.sqrMagnitude > 0
&& (m_Input.x != 0 || m_Input.y != 0))
            {
```

```
                m_StepCycle +=
(m_CharacterController.velocity.magnitude + (speed*(m_IsWalking
? 1f : m_RunstepLenghten)))*
                             Time.fixedDeltaTime;
            }

            if (!(m_StepCycle > m_NextStep))
            {
                return;
            }

            m_NextStep = m_StepCycle + m_StepInterval;

            PlayFootStepAudio();
        }


        private void PlayFootStepAudio()
        {
            if (!m_CharacterController.isGrounded)
            {
                return;
            }
            // pick & play a random footstep sound from the
array,
            // excluding sound at index 0
            int n = Random.Range(1, m_FootstepSounds.Length);
            m_AudioSource.clip = m_FootstepSounds[n];
            m_AudioSource.PlayOneShot(m_AudioSource.clip);
```

```
            // move picked sound to index 0 so it's not picked
next time
            m_FootstepSounds[n] = m_FootstepSounds[0];
            m_FootstepSounds[0] = m_AudioSource.clip;
        }


        private void UpdateCameraPosition(float speed)
        {
            Vector3 newCameraPosition;
            if (!m_UseHeadBob)
            {
                return;
            }
            if (m_CharacterController.velocity.magnitude > 0 &&
m_CharacterController.isGrounded)
            {
                m_Camera.transform.localPosition =
                    m_HeadBob.DoHeadBob(m_CharacterController.ve
locity.magnitude +
                                        (speed*(m_IsWalking ? 1f :
m_RunstepLenghten)));
                newCameraPosition =
m_Camera.transform.localPosition;
                newCameraPosition.y =

m_Camera.transform.localPosition.y - m_JumpBob.Offset();
            }
            else
            {
```

```
                newCameraPosition =
m_Camera.transform.localPosition;
                newCameraPosition.y = m_OriginalCameraPosition.y
- m_JumpBob.Offset();
            }
            m_Camera.transform.localPosition =
newCameraPosition;
        }


        private void GetInput(out float speed)
        {
            // Read input
            float horizontal =
CrossPlatformInputManager.GetAxis("Horizontal");
            float vertical =
CrossPlatformInputManager.GetAxis("Vertical");

            bool waswalking = m_IsWalking;

#if !MOBILE_INPUT
            // On standalone builds, walk/run speed is modified
by a key press.
            // keep track of whether or not the character is
walking or running
            m_IsWalking = !Input.GetKey(KeyCode.LeftShift);
#endif
            // set the desired speed to be walking or running
            speed = m_IsWalking ? m_WalkSpeed : m_RunSpeed;
```

```
            m_Input = new Vector2(horizontal, vertical);

            // normalize input if it exceeds 1 in combined
length:
            if (m_Input.sqrMagnitude > 1)
            {
                m_Input.Normalize();
            }

            // handle speed change to give an fov kick
            // only if the player is going to a run, is running
and the fovkick is to be used
            if (m_IsWalking != waswalking && m_UseFovKick &&
m_CharacterController.velocity.sqrMagnitude > 0)
            {
                StopAllCoroutines();
                StartCoroutine(!m_IsWalking ?
m_FovKick.FOVKickUp() : m_FovKick.FOVKickDown());
            }
        }
        private void RotateView()
        {
            m_MouseLook.LookRotation (transform,
m_Camera.transform);
        }
        private void
OnControllerColliderHit(ControllerColliderHit hit)
        {
            Rigidbody body = hit.collider.attachedRigidbody;
```

```
            //dont move the rigidbody if the character is on top
of it
            if (m_CollisionFlags == CollisionFlags.Below)
            {
                return;
            }

            if (body == null || body.isKinematic)
            {
                return;
            }
            body.AddForceAtPosition(m_CharacterController.veloci
ty*0.1f, hit.point, ForceMode.Impulse);
        }
    }
}
```

FirstPersonController.cs

ДОДАТОК Б

ПРЕЗЕНТАЦІЯ ДО ЗВІТУ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Магістерська робота

«ТЕМА»

Методика створення бота у відеоіграх в жанрі шутер на основі штучного інтелекту

Виконав: Сабадах В.С
Керівник: Золотухіна О.А

Київ - 2021

МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення якості ігрових ботів у відеоіграх у жанрі шутер за рахунок використання методів штучного інтелекту.

Об'єкт дослідження: поведінка ігрового бота у відеогрі в жанрі шутер.

Предмет дослідження: методи штучного інтелекту для відеоігр у жанрі шутер.

ІСНУЮЧІ АЛГОРИТМИ ВИРІШЕННЯ ЗАДАЧ

3

Детерміновані алгоритми штучного інтелекту. Детерміновані алгоритми означають зумовлене і заздалегідь запрограмовану поведінку об'єктів.

Метод кінцевих автоматів. Вимога розумної кількості станів досить просто і зрозуміло. У нас, людей, є сотні, якщо не тисячі, емоційних станів, і в кожному з них - безліч підстанів.
А персонаж гри повинен всього лише виглядати розумним, так що і станів у нього не повинно бути надто багато. Наприклад, простий персонаж гри може мати декілька станів.

Запам'ятовування і навчання. Всі розглянуті технології ШІ працюють тільки з поточною інформацією і ніколи не враховують події, що вже відбулися. Як приклад використання пам'яті штучним інтелектом. Якщо у персонажу закінчуються боєприпаси, то набагато розумніше не починати пошук наосліп, тобто випадковим чином, а просканувати всі записи і "згадати", в якому приміщенні залишилися незаймані боєприпаси.

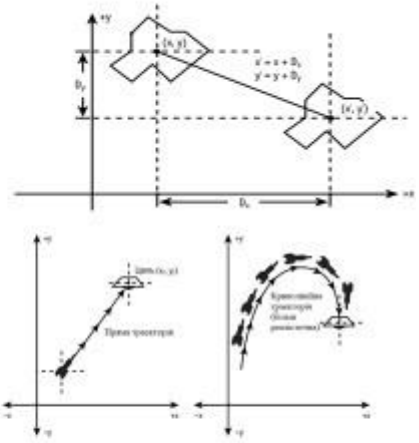
Алгоритм проходження за об'єктом. Штучний інтелект на основі інформації про стан деякого об'єкту, що відслідковується, змінює траєкторію нашого об'єкта таким чином, щоб він рухався у напрямку до відстежуваного об'єкту. Рух може бути спрямовано як безпосередньо на відстежуваний об'єкт, так і за деякою кривою, спрямованої до об'єкту (на зразок траєкторії самонавідної ракети).

ОТРИМАНІ РЕЗУЛЬТАТИ РОБОТИ

4

Науковий результат:

Модернізовано модель поведінки ігрового бота у відеогрі в жанрі шутер на основі детермінованих алгоритмів штучного інтелекту та за рахунок застосування алгоритму проходження за об'єктом.



Практичний результат:

Реалізовано автономно функціонуючі елементи (персонажі) гри з елементами штучного інтелекту, які більш гнучко адаптуються до оточуючого середовища.

СТАН НАПИСАННЯ ДИПЛОМУ 5

Назва розділу	Кількість сторінок
1. АНАЛІТИЧНИЙ ОГЛЯД ТИПІВ КОМП'ЮТЕРНИХ ІГОР, ДЕ ВИКОРИСТОВУЄТЬСЯ ШТУЧНИЙ ІНТЕЛЕКТ	7
2. АНАЛІЗ І ДЕТАЛІЗАЦІЯ МАТЕМАТИЧНИХ АЛГОРИТМІВ ДЛЯ РОЗРОБКИ ШТУЧНОГО ІНТЕЛЕКТУ	55
3. СТВОРЕННЯ ВЛАСНОЇ МОДЕЛІ ШТУЧНОГО ІНТЕЛЕКТУ	16
4. СТВОРЕННЯ АВТОНОМНО ФУНКЦІОНУЮЧИХ ЕЛЕМЕНТІВ (ПЕРСОНАЖІВ) ГРИ	13

Кількість джерел в списку літератури: 63

РЕЗУЛЬТАТИ НАУКОВО-ПЕДАГОГІЧНОЇ ПРАКТИКИ 6

Назва дисципліни, з якої було проведено заняття:
Розробка ігор

Лектор дисципліни:
Доцент кафедри, Дібрівний О.А

Тема заняття, тип (лек., пр., лаб.):
Розробка платформеру, практика та лабораторна

Практичні результати:
Презентаційний матеріал та проект.

Стан звіту з науково-педагогічної практики:
Здано на кафедру

РЕЗУЛЬТАТИ НАУКОВО-ДОСЛІДНОЇ ПРАКТИКИ

Статті:

1. Золотухіна О.А., Сабадах В.С. Особливості побудови моделі поведінки ігрових ботів у відеоіграх в жанрі шутер на основі штучного інтелекту. (подано до друку в журнал Телекомунікаційні та інформаційні технології, 2021 №2)

Тези доповідей:

1. Сабадах В.С. Перспективи систем штучного інтелекту у повсякденні.

Стан звіту з науково-дослідної практики:

На затвердженні у керівника

ДЯКУЮ ЗА УВАГУ!