МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп’ютерних наук

Кафедра комп’ютерної інженерії

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри,

д–р техн. наук, проф.

___________ І. М. Журавська

«__»__________ 2023 р.

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

МОДЕЛЮВАННЯ ТА РЕАЛІЗАЦІЯ СКІНЧЕНОГО АВТОМАТУ

Спеціальність «Комп’ютерна інженерія»
123 – КМР.1 – 605.21710922

Студент __________ В. В. Сирота

підпис

«__» __________ 2023 р.

Керівник доктор фізико – математичних наук, професор кафедри комп’ютерної інженерії __________ Г. П. Чуйко

підпис

«__» __________ 2023 р.

Миколаїв – 2023

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 4

ВСТУП 5

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ. СПЕЦИФІКАЦІЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ОБГРУНТУВАННЯ ПЛАНУ ВИКОНАННЯ ЗАВДВННЯ 7

1.1 Штучний інтелект. Види та сфери застосування 7

1.2 Штучний інтелект у відеоіграх 9

1.3 Постановка задачі 13

1.4 Вибір та обґрунтування підходів виконання завдання 14

1.4.1 Вибір мови програмування 14

1.4.2 Скінченний автомат 15

1.4.3 Вибір ігрового рушія 17

Висновки до розділу 1 18

2 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ПРОГРАМНОЇ БІБЛІОТЕКИ. ПРОЄКТУВАННЯ ПРОЄКТІВ–ПРИКЛАДІВ 19

2.1 Моделювання скінченного автомату 19

2.2 Проєктування програмної бібліотеки 23

2.2.1 Специфікація вимог 23

2.2.2 Обґрунтування інструментів реалізації 24

2.2.3 Побудування діаграми класів бібліотеки 27

2.3 Проєктування гри–прикладу на Unity 31

2.5 Проєктування апаратного прикладу 35

Висновки до розділу 2 37

3 РОЗРОБКА ПРОГРАМНОЇ БІБЛІОТЕКИ. РОЗРОБКА ПРОЄКТІВ–ДЕМОНСТРАЦІЇ 38

3.1 Розробка програмної бібліотеки 38

3.2 Розробка гри–демонстрації 46

3.3 Розробка АПЗ демонстрації 51

Висновки до розділу 3 54

4 ТЕСТУВАННЯ БІБЛІОТЕКИ. ДОСЛІДЖЕННЯ ПРОЦЕССУ РОЗРОБКИ ШІ ЗА ДОПОМОГОЮ РОЗРОБЛЕНОЇ БІБЛІОТЕКИ. АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕНЬ ТА ТЕСТУВАННЯ 55

4.1 Тестування бібліотеки за допомогою гри–демонстрації 55

4.2 Тестування бібліотеки у внутрішньому середовищі 58

4.3 Дослідження і порівняння процесу розробки ШІ за допомогою розробленої бібліотеки та аналогів 62

4.4 Аналіз результатів тестування та досліджень 65

Висновки до розділу 4 67

ВИСНОВКИ 68

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 69

ПЕРЕЛІК СКОРОЧЕНЬ

МПР – марковські процеси рішень
НІП – не ігровий персонаж
ШІ – штучний інтелект
API – application programming interface
AR – augmented reality
CLI – command line interface
DLL – dynamic link library
FSM – finite state machine
IDE – integrated development environment
NPC – not playable character
VR – virtual reality

ВСТУП

Математичне моделювання є важливим і базовим інструментом у світі ігор та штучного інтелекту (ШІ). Математичні моделі дозволяють краще зрозуміти складність реального світу, а також розробляти і створювати ігри та штучний інтелект, які є одночасно реалістичними і пізнавальне-цікавими.

В іграх математичні моделі використовуються для створення захоплюючих і реалістичних світів. Незалежно від того, чи це гра віртуальної реальності, чи одно користувацька гра, математичні моделі використовуються для відтворення фізичних законів, які керують ігровим світом. Застосовуючи математичні рівняння і формули, розробники можуть створювати фізичні взаємодії, які дозволяють реалістичні і правдоподібні дії, що відбуваються в ігровому світі.

У світі ШІ математичне моделювання уживається для створення роботів і машин, які можуть взаємодіяти з навколишнім світом і відтворювати його. Математичні моделі використовують, щоб навчити машини розпізнавати закономірності та робити прогнози. Наприклад, у галузі робототехніки математичні моделі призначені для керування рухами робота та його взаємодією з навколишнім середовищем. Математичні моделі також придатні для навчання систем штучного інтелекту розуміти природну мову і приймати рішення на основі отриманих даних.

Одним з популярних способів запровадження ШІ у відеоіграх є скінченні автомати (Finite State Machine). Такий автомат – це математична модель обчислень, яка використовується для представлення поведінки системи вводу/виводу. В іграх скінченні автомати використовуються для моделювання поведінки ШІ–супротивників, а також для імітації та моделювання реакцій гравців під час взаємодії з ігровим середовищем. Автомат використовує кінцевий (обмежений) набір станів і переходів для представлення та урахування потенційних результатів будь якої ситуації.

Загалом, математичне моделювання є важливим інструментом, який можна використовувати для створення цікавих ігор, роботів і систем штучного інтелекту. Застосовуючи математичні рівняння та формули, розробники та інженери можуть створювати реалістичні симуляції та ШІ, які можуть взаємодіяти з навколишнім світом.

Актуальність теми та її науково–практичне значення: Скінченні автомати мають багато застосувань у сферах математиці, інформатиці та інженерії. Їх використовують у багатьох галузях, від обробки природної мови та розробки відеоігор до штучного інтелекту та робототехніки. Моделювання та реалізація скінченних автоматів є важливим у багатьох з цих областей, оскільки вони можуть забезпечити ефективне вирішення таких проблем, які не можуть бути вирішені за допомогою традиційних алгоритмів. Таким чином, моделювання та реалізація скінченних автоматів актуальні і мають науково-практичне значення в різних застосунках.

Мета роботи: Спростити та покращити процес створення штучного інтелекту у відеоіграх та інших комп’ютерних застосунках, шляхом розробки програмного забезпечення, а саме створення програмної бібліотеки на основі математичної моделі скінченного автомату.

Об’єкт дослідження: Процес створення штучного інтелекту на основі правил у відеоіграх та інших системах.

Предмет дослідження: Програмна бібліотека на основі математичної моделі скінченного автомату, для створення штучного інтелекту у відеоіграх та інших системах зі скінченим набором станів поведінки.

Робота пройшла апробацію під час XXV Всеукраїнської науково-практичної конференції «Могилянські читання» (Миколаїв, 07–10 лютого 2023 р.). Публікації. Основні положення та результати магістерської роботи опубліковані у збірнику матеріалів XXV Всеукраїнської науково-практичної конференції «Могилянські читання–2022».

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ. СПЕЦИФІКАЦІЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ОБГРУНТУВАННЯ ПЛАНУ ВИКОНАННЯ ЗАВДВННЯ

1.1 Штучний інтелект. Види та сфери застосування

Штучний інтелект (ШІ) – це технологія, що стрімко розвивається і широко використовується в різних сферах нашого життя, таких як робототехніка, медична діагностика, фінанси, виробництво, транспорт і так далі. Ця технологія була розроблена для того, щоб змусити машини імітувати людську поведінку та мислення. ШІ визначається як набір методів, які дозволяють машинам діяти і мислити автономно і незалежно, так, як якщо б вони були людьми, що виконують ту ж саму задачу. Вважається, що ШІ матиме великий вплив на наше життя в майбутньому.

Штучний інтелект – це здатність машин вчитися з навколишнього середовища і приймати рішення на основі цієї інформації. Це широка галузь комп'ютерних наук, яка охоплює безліч тем, включаючи програмування, машинне навчання, інтелектуальний аналіз даних, обробку природної мови, комп'ютерний зір і робототехніку[1]. ШІ використовується для вирішення складних завдань у багатьох галузях, включаючи медицину, фінанси, інженерію та військову сферу.

Існує кілька різних підходів до ШІ, кожен з яких має свої переваги та недоліки. Найбільш популярними є такі підходи:

- **ШІ на основі правил:** цей підхід використовує набір правил для визначення поведінки машини. Правила запрограмовані в машині, і машина може вчитися з навколишнього середовища, спостерігаючи за тим, як вона реагує на різні вхідні дані. Цей підхід часто використовується в медичній діагностиці та робототехніці.

- **Машинне навчання:** Цей підхід використовує алгоритми для виявлення закономірностей у даних. Машина здатна вчитися на основі даних і робити прогнози або приймати рішення на їх основі. Цей підхід часто використовується в інтелектуальному аналізі даних, фінансах та інженерії.
- **Обробка природної мови:** Цей підхід використовує алгоритми для обробки природної мови і генерування з неї сенсу. Цей підхід часто використовується в пошукових системах і в чат–ботах.

Штучний інтелект використовується в багатьох сферах нашого життя, і його потенціал для інновацій тільки починає розкриватися. Ось деякі з найпопулярніших застосувань:

- **Робототехніка:** ШІ використовується для створення роботів, які можуть взаємодіяти з навколишнім середовищем і виконувати завдання. Ці роботи можуть використовуватися у виробництві, охороні здоров'я, безпеці та інших сферах.
- **Медична діагностика:** ШІ використовується для діагностики захворювань і рекомендацій щодо лікування. Алгоритми ШІ використовуються для аналізу медичних зображень і виявлення аномалій.
- **Фінанси:** ШІ використовується для виявлення шахрайства та інших фінансових порушень. Алгоритми ШІ використовуються для аналізу фінансових даних і прогнозування майбутніх ринкових тенденцій.
- **Виробництво:** ШІ використовується для автоматизації виробничих процесів, що дозволяє підвищити їхню ефективність і точність. Алгоритми ШІ використовуються для виявлення дефектів у продукції та оптимізації виробничого процесу.
- **Транспорт:** ШІ використовується для підвищення безпеки та ефективності транспортних систем. Алгоритми ШІ використовуються для виявлення схем руху і планування більш швидких і ефективних маршрутів.

1.2 Штучний інтелект у відеоіграх

Штучний інтелект (ШІ) набуває все більшого значення в ігровій індустрії, і очікується, що його вплив продовжить зростати в найближчі роки. У найпростішому вигляді ШІ – це використання машин для виконання завдань, які зазвичай виконує людина. Ця технологія може бути використана в іграх для створення інтерактивного, реалістичного досвіду для гравців.

Один із способів використання ШІ в іграх – це створення опонентів для гравців. ШІ–супротивників можна запрограмувати так, щоб вони думали і діяли, як реальні люди, що робить їх більш складними для перемоги. Це може створити більш реалістичний досвід для гравців, оскільки вони не просто змагаються проти статичного набору правил. ШІ–супротивників також можна запрограмувати так, щоб вони змінювали рівень складності залежно від рівня навичок гравця, роблячи гру приємнішою для всіх.

Ще одне застосування ШІ в іграх – створення більш захопливого та реалістичного середовища. Програмуючи керованих ШІ NPC (неігрових персонажів), розробники ігор можуть зробити ігровий світ більш живим та інтерактивним. Крім того, ШІ можна використовувати для того, щоб зробити ігровий світ більш динамічним, оскільки NPC, керовані ШІ, можуть реагувати на дії гравців і навіть змінювати навколишнє середовище залежно від дій гравця[2].

Найбільш найпопулярнішими підходами ШІ, що використовуються у відеоіграх, є системи на основі правил, дерева поведінки, дерева рішень та генетичні алгоритми[3].

Системи на основі правил

Системи, засновані на правилах, використовуються для прийняття рішень на основі заздалегідь визначених правил і сценаріїв. ШІ–агент запрограмований на набір правил, яким він повинен слідувати, і використовує ці правила для прийняття рішень і виконання дій. Ці правила можуть бути настільки простими або складними, наскільки це необхідно, і ШІ–агент може бути запрограмований діяти по–різному залежно від ситуації.

Наприклад, ці системи включають в себе кінцеві автомати. Кінцеві автомати – це тип моделі, який використовує набір станів, переходів і дій для представлення системи. Автомат починає роботу в початковому стані, а потім слідує за переходами станів, щоб досягти бажаної мети. Кожен перехід запускається дією, яка може бути введеною користувачем або зовнішньою подією. Кінцеві автомати корисні для моделювання та імітації складної поведінки, і їх часто використовують у розробці ігор.

Дерева поведінки

Дерева поведінки – це тип системи, заснованої на правилах. Вони використовують ієрархічну структуру взаємопов'язаних вузлів, де кожен вузол представляє певну поведінку. ШІ–агент використовує цю структуру для прийняття рішень і виконання дій. Дерева поведінки зазвичай використовуються в іграх зі складним прийняттям рішень, таких як стратегічні ігри в реальному часі[4].

Дерева рішень

Дерева рішень – це тип системи, заснованої на правилах. Вони використовують ієрархічну структуру взаємопов'язаних вузлів, де кожен вузол представляє можливе рішення. ШІ–агент використовує цю структуру для прийняття рішень і виконання дій. Дерева рішень зазвичай використовуються в іграх зі складним прийняттям рішень, таких як стратегічні ігри в реальному часі.

Генетичні алгоритми

Генетичні алгоритми – це тип підходу ШІ, який використовує еволюційні методи для пошуку рішень проблем. ШІ–агент програмується за допомогою набору правил, і він використовує ці правила для генерації потенційних рішень. Потім ШІ–агент оцінює потенційні рішення, обирає найкраще з них і відповідно коригує правила. Цей процес повторюється, поки не буде знайдено оптимальне рішення. Генетичні алгоритми зазвичай використовуються в іграх зі складним прийняттям рішень, таких як стратегічні ігри в реальному часі[5].

До популярних відеоігор, які використовують ШІ, належать такі ігри:

1. **Civilization** використовує форму ШІ під назвою "нейронні мережі", яка дозволяє грі визначати різноманітні стратегії та поведінку на основі вхідних даних;

2. **The Sims** використовує комбінацію "ШІ на основі правил" і "стохастичного ШІ" для створення реалістичного оточення і поведінки персонажів гри;

3. **Grand Theft Auto** використовує "кінцеві автомати", які дозволяють грі інтерпретувати дані та генерувати відповідні реакції;

4. **Fallout** використовує "експертні системи", щоб генерувати інтелектуальну поведінку для NPC гри;

5. **Spore** використовує поєднання "нечіткої логіки" та "генетичних алгоритмів" для створення середовища, що постійно розвивається.

Деякі з останніх інновацій ШІ у відеоіграх включають:

— ШІ ворогів, керований ШІ: ШІ ворогів, керований ШІ, використовується для того, щоб зробити ворогів розумнішими і швидше реагувати на рішення гравця, створюючи більш захопливий і складний ігровий досвід;

— діалоги зі штучним інтелектом: Діалоги зі штучним інтелектом можна використовувати для створення більш реалістичних персонажів з реалістичною реакцією на вибір гравця;

— середовище зі штучним інтелектом: Середовище зі штучним інтелектом може реагувати на рішення гравця, створюючи більш захопливий ігровий світ;

— підбір матчів зі штучним інтелектом: Підбір матчів зі штучним інтелектом можна використовувати для створення збалансованих ігор і підбору гравців зі схожими рівнями навичок;

— графіка зі штучним інтелектом: Графіка зі штучним інтелектом може використовуватися для створення більш реалістичних візуальних ефектів, що робить ігровий процес більш захопливим.

1.3 Постановка задачі

Дана робота має призначення спростити процес створення ШІ ігрових персонажів, шляхом створення програмної бібліотеки яка має увесь необхідний функціонал. У подальшому ця бібліотека може бути інтегрована майже у будь–який проєкт.

Для досягнення даної мети, необхідно сформулювати та вирішити наступні задачі:

1. Обрати мову програмування для бібліотеки та ігровий рушій на якому будуть проводитися тести;
2. Обрати додаткові функції бібліотеки окрім реалізації скінченного автомату;
3. Дослідити існуючі рішення і виділити потрібні риси;
4. Розробити структури даних для зберігання інформації та логіки, пов'язаної з машиною станів;
5. Написати програмний код який буде реалізовувати весь потрібний функціонал бібліотеки;
6. Написати модульні тести для створених методів, щоб переконатися, що вони працюють правильно;
7. Упакувати бібліотеку у формат, який може бути використаний іншими програмами;
8. Інтегрувати бібліотеку у обраний ігровий рушій та створити гру для демонстрації роботи бібліотеки. Додатково протестувати за необхідності;

1.4 Вибір та обґрунтування підходів виконання завдання

1.4.1 Вибір мови програмування

C# – це популярна об'єктно–орієнтована мова програмування, розроблена компанією Microsoft. Вона була розроблена Андерсом Хейлсбергом у 2000 році і має підтримку строго типізованого програмування та систему збору сміття. Вона використовується для створення різноманітних додатків, від настільних Windows–додатків до веб–додатків і сервісів, мобільних додатків тощо. Вона є мовою вибору для багатьох розробників, коли мова йде про написання багато платформних додатків.

Багатоплатформна розробка – це процес розробки додатків для декількох платформ. Це означає, що один додаток може бути розроблений і розгорнутий на декількох платформах, таких як Windows, iOS і Android. Це може заощадити час і гроші розробників, оскільки їм не потрібно розробляти кілька версій одного і того ж додатка для кожної платформи.

C# можна використовувати для написання багатоплатформних додатків двома різними способами: за допомогою фреймворків, таких як Xamarin, або за допомогою .NET Core.

Xamarin – це популярний крос–платформний фреймворк, розроблений компанією Microsoft. Він дозволяє розробникам писати код на C# та розгортати його на різних платформах, таких як Windows, Android та iOS.

.NET Core – це нова версія фреймворку .NET з відкритим вихідним кодом, розроблена компанією Microsoft. Він розроблений як крос–платформний і може використовуватися для написання додатків, які можуть працювати на Windows, Linux і macOS. Вона надає доступ до тих самих API–інтерфейсів C#, що й повна версія .NET framework, тому розробники мають доступ до тих самих функцій, що й у випадку з повною версією .NET framework.

Виходячи з усіх вище описаних переваг, для реалізації даної роботи було вирішено обрати у якості мови програмування саме C#.

1.4.2 Скінченний автомат

Кінцеві автомати (Finite State Machines, FSM) – це тип математичної моделі, що використовується для представлення поведінки системи. Вони використовуються для моделювання поведінки системи в термінах набору станів, переходів між ними, а також входів і виходів для кожного переходу. FSM широко використовуються в галузі штучного інтелекту для створення агентів, які можуть автономно сприймати і взаємодіяти з навколишнім середовищем[6].

Переваги кінцевих автоматів:

— Забезпечують просте та інтуїтивно зрозуміле представлення поведінки системи. Їх легко зрозуміти навіть непрограмістам;

— є детермінованими, що означає, що при однаковому наборі входів, вихід завжди буде передбачуваним. Це робить їх ідеальними для побудови систем, які повинні постійно видавати однакові результати;

— добре підходять для моделювання складних, реальних систем. Використовуючи ієрархічну структуру кінцевих автоматів, можна моделювати складну поведінку;

— можна використовувати для створення автономних агентів, які здатні сприймати і взаємодіяти з навколишнім середовищем.

Недоліки скінченних автоматів:

— може бути важко налагоджувати. Оскільки поведінка системи визначається її поточним станом, може бути важко визначити, що змушує систему поводитися несподівано;

— у деяких випадках FSM можуть ставати дуже великими і складними. Це може ускладнити їх обслуговування і призвести до збільшення часу та вартості розробки;

— можуть стати неефективними при роботі з великими масивами даних. Це може призвести до низької продуктивності або навіть до збоїв системи.

Одне з найпопулярніших застосувань автоматів у ШІ – це використання Марковських процесів Прийняття Рішень (МПР). Це тип алгоритму ШІ, який моделює поведінку агента в навколишньому середовищі, представляючи його як набір станів і переходів. На основі поточного стану МПР здатний приймати оптимальні рішення, які максимізують винагороду агента[7].

Інші застосування автоматів в ШІ включають використання систем нечіткої логіки, які використовують нечітку логіку для представлення невизначених знань, і використання байєсівських мереж, які використовуються для представлення невизначених взаємозв'язків між змінними.

Існує три типи скінченних автоматів:

1. **детерміновані кінцеві автомати (Deterministic Finite Automata, DFA).** DFA – це машина, яка переходить між станами на основі набору заздалегідь визначених правил. Це тип автомата, який може перебувати точно в одному зі скінченної кількості станів у будь–який момент часу. Перехід з одного стану в інший визначається вхідним символом і поточним станом;

2. **недетерміновані скінченні автомати (Non–Deterministic Finite Automata , NFA).** NFA схожий на DFA, але він може переходити з одного стану в інший на основі одного або декількох вхідних символів. Він не обмежується одним переходом на стан;

3. **машина Мура.** Це тип скінченного автомата, який виробляє вихід на основі поточного стану. Він складається зі скінченної множини станів і вихідної функції, яка ставить у відповідність кожному стану вихід. Вихід автомата визначається поточним станом, а не вхідними символами.

Отже, кінцеві автомати є потужною та універсальною обчислювальною моделлю для моделювання поведінки системи. Вони широко використовуються в штучному інтелекті для створення агентів, які можуть автономно сприймати і взаємодіяти з навколишнім середовищем. Незважаючи на свої переваги, FSM мають деякі недоліки, які слід враховувати при їх проектуванні та впровадженні[8].

Виходячи з усього описаного вище було обрано використовувати детермінований скінчений автомат, оскільки він найбільш підходить для ШІ НІП.

1.4.3 Вибір ігрового рушія

Unity – це потужний кросплатформенний ігровий рушій, що використовується для створення 3D/2D ігор та інтерактивних додатків. Його використовують мільйони розробників по всьому світу для створення високоякісних 2D та 3D ігор, симуляторів та інтерактивних додатків. Unity був спочатку створений як 3D рушій, але з тих пір розвинувся до підтримки не лише 3D ігор.

Unity був вперше розроблений у 2005 році Unity Technologies, ігровою студією, що базується в Копенгагені, Данія. Компанію заснували Девід Хельгасон, Ніколас Френсіс та Йоахім Анте.

Вперше рушій був випущений у 2005 році як рушій для 3D–ігор і використовувався для створення 3D–ігор для мобільних платформ. З того часу Unity перетворився на потужний кросплатформний рушій.

Unity відомий своєю гнучкістю, масштабованістю та простотою використання. Його легко використовувати як початківцям, так і досвідченим розробникам. Рушій підтримує різні платформи, такі як настільні, консольні, мобільні, веб–платформи та платформи VR/AR.

Він також інтегрований з популярними ігровими рушіями, такими як Unreal Engine та PhysX. Рушій добре підходить для створення високоякісних 3D і 2D ігор, симуляторів та інтерактивних додатків.

Unity – ідеальний рушій для створення симуляцій та моделей. Він швидкий, надійний і підтримує багато платформ. Він також має велику бібліотеку інструментів та функцій для створення симуляцій, включаючи фізичні рушії, системи частинок, аудіосистеми та штучний інтелект.

Unity – один з найпопулярніших ігрових рушіїв. Він добре підходить для створення симуляцій та моделей завдяки своїй гнучкості, масштабованості та простоті використання. Він також інтегрований з популярними ігровими рушіями, такими як Unreal Engine та PhysX. Порівняно з іншими рушіями, Unity є більш дружнім до початківців і краще підходить для створення симуляцій.

Висновки до розділу 1

Штучний інтелект це нова та широка галузь комп'ютерних наук. Вона охоплює безліч тем, інтелектуальний аналіз даних, машинне навчання, включаючи програмування, обробку природної мови, комп'ютерний зір і робототехніку. ШІ використовується для вирішення складних завдань у багатьох сферах нашого життя, включаючи медицину, фінанси, інженерію та військову сферу.

У відеоіграх ШІ також відіграє дуже важливу роль та все частіше з’являється у нових іграх. Він допомагає створити світ відеогри більш реалістичним, більш несиченим та цікавим. Не ігрових персонажів робить більш живими, цікавими та непередбачуваними.

Кінцеві автомати є потужною та універсальною обчислювальною моделлю для моделювання поведінки системи. Вони широко використовуються в штучному інтелекті для створення агентів, які можуть автономно сприймати і взаємодіяти з навколишнім середовищем.

Процес створення штучного інтелекту та додавання його у гру є дуже трудомістким і займає багато часу. До того ж не всі розробники мають потрібний досвід, та хочуть отримати швидке рішення без додавання над зусиль. Тому

створення бібліотеки може допомогти деяким іншим розробникам швидко і просто створити і додати ШІ до персонажів у своїй грі.

2 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ПРОГРАМНОЇ БІБЛІОТЕКИ. ПРОЄКТУВАННЯ ПРОЄКТІВ–ПРИКЛАДІВ

2.1 Моделювання скінченного автомату

Першочерговою задачею було моделювання скінченного автомату, за для повного розуміння усього необхідного функціоналу розробляємої бібліотеки та для подальшої реалізації цього автомату у відеогрі–прикладі. Для прикладу було вирішено взяти найпопулярніший варіант використання скінченних автоматів – ШІ не ігрового персонажу у грі.

Моделюємий скінченний автомат має наступні стани:

— спати (Sleep);

— сховатися від дощу (Hide);

— збирати врожай (Harvest);

— розпалити багаття (Light Campfire);

— відпочивати (Rest).

Моделюємий скінченний автомат має наступні вхідні дані:

— внутрішньо ігровий час доби;

— температура повітря;

— чи є врожай готовий до збирання;

— рівень втоми.

Моделюємий скінченний автомат має наступні вихідні дані залежно від поточного стану:

— стан Harvest – рівень втоми;

— стан Sleep – рівень втоми;

— стан Rest – рівень втоми;

— стан Harvest – чи є врожай готовий до збирання;

— стан LightCampfire – температура повітря.

Таблиця переходів – це таблиця, яка визначає перехід станів між двома різними станами. Зазвичай вона використовується в теорії автоматів, яка вивчає абстрактні автомати та їхні властивості. Це графічне представлення поведінки автомата, яке можна використовувати для визначення поведінки автомата при переході з одного стану в інший. Вона містить поточний стан, подію, яка спричиняє перехід до наступного стану, і наступний стан, що виникає в результаті. Вона потрібна для того, щоб допомогти визначити поведінку автомата, і її важливість полягає у визначенні послідовності подій і пов'язаних з ними станів, які відбуваються під час переходу з одного стану в інший[9].

У таблиці 2.1 відображаються переходи між станами машини. Де перший стовпчик відображає поточний стан, другий – умови переходу, третій – наступний стан.

Таблиця 2.1 – Таблиця переходів скінченного автомату

Current State	**Action\Events**	**Resulting State**
Sleep	Is it morning and harvest ready	Harvest
Hide	Is it sunny and harvest ready	Harvest
Hide	Is it night	Sleep
Harvest	Is it raining	Hide
Harvest	Is tired	Rest
Harvest	Is it cold	LightCampfire
Harvest	Is it night	Sleep
LightCampfire	Is it night	Sleep
LightCampfire	Is it tired	Rest
LightCampfire	Is it raining	Hide
LightCampfire	Is harvest ready	Harvest
Rest	Is it cold	LightCampfire
Rest	Is it morning and harvest ready	Harvest
Rest	Is it raining	Hide

Останнім кроком моделювання скінченного автомату є створення діаграми станів та переходів між ними. На малюнку 2.1 зображена діаграма станів розробляємої моделі детермінованого скінченного автомату.

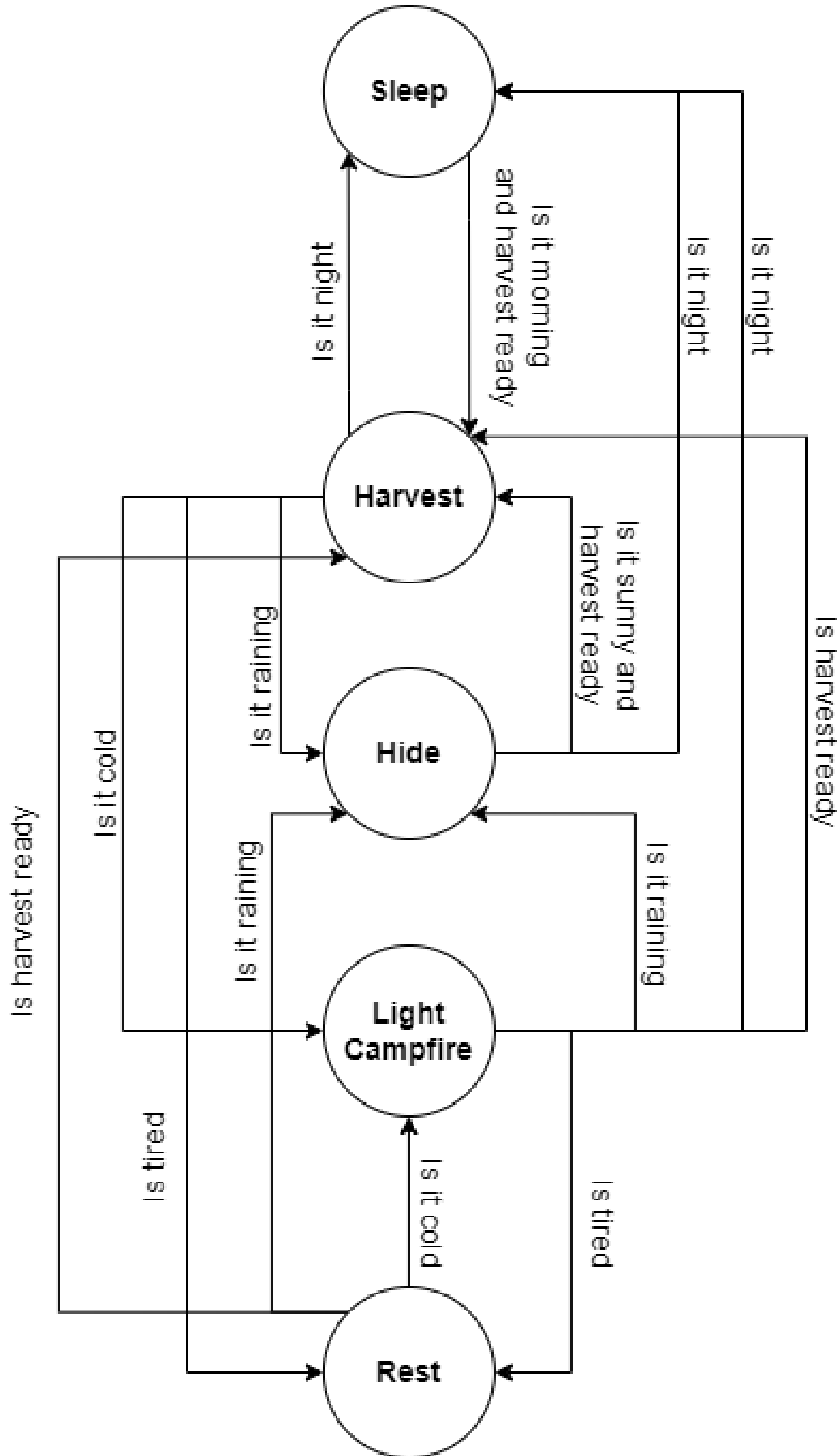

Рисунок 2.1 – Діаграма станів скінченного автомату

2.2 Проєктування програмної бібліотеки

2.2.1 Специфікація вимог

Призначення системи (застосунку), для якої розробляється програмне забезпечення: спрощення створення ШІ НІП персонажів, на будь–якому ігровому рушії або проєкті.

Межі проєкту ПЗ: програмна бібліотека буде використовуватися у середовищах які підтримують .NET Core.

Сфера застосування: використання у сфері ігрової індустрії.

Характеристики користувачів: бібліотека створена для розробників відеоігор які хочу використовувати ШІ для НІП у своїх іграх або проєктах.

Загальна структура і склад системи: система розробляється на основі детермінованого скінченного автомату.

Склад системи: система машини станів, система обміну даних між станами, правила переходів між станами, система відслідковування станів та відправки повідомлень подій.

Загальні обмеження: Платформи, ігрові рушії або середовища які не підтримує .NET Core або нативні .dll бібліотеки.

Основні функції системи:

1. створення власних станів;
2. створення правил переходів між станами;
3. додавання власних даних до машини та обмін ними між станами;
4. керування даними між станами та ззовні;
5. управління станами та переходами;
6. створення власних подій та додавання їх станів;
7. відслідковування події у станах машини;
8. можливість розширювати функціонал машини та станів.

Джерела і зміст вхідної інформації (даних): всі дані які находять у машині станів під час виконання зберігаються у оперативній пам’яті, та видаляються після закінчення процесу. Але користувач має можливість серіалізувати поточний стан машини і зберегти його, а потім загрузити дані під час наступного запуску програми.

Вимоги до способів організації, збереження та ведення інформації: інформація про поточний стан автомату повинна повністю серіалізуватися та десеріалізуватися у JSON форматі.

Доступність: бібліотека має бути безкоштовною і з відкритим кодом.

Супроводжуваність: бібліотека має мати можливість розширення функціоналу шляхом наслідування її класів або композиції.

Переносимість: бібліотека має працювати однаково на усіх платформах, ігрових рушіях, та середовищах які підтримують .NET Core та .dll бібліотеки.

Продуктивність: внутрішня структура машини повинна не використовувати виділення пам’яті під час виконання, має реалізовувати хєш таблиці для звірки ключів даних або найменування станів, використовувати найшвидші алгоритми пошуку та сортування даних.

2.2.2 Обґрунтування інструментів реалізації

У якості мови програмування було обрано C#, оскільки він має можливість багатоплатформної розробки. C# підходить для багатоплатформної розробки, оскільки це потужна та універсальна мова програмування, що дозволяє легко писати додатки, які можна використовувати на різних платформах, таких як Windows, Mac, Linux, iOS та Android. Вона підтримує об'єктно–орієнтоване програмування, що дозволяє розробникам створювати багаторазовий код, а безпека типів допомагає забезпечити надійність коду.

Також C# можна використовувати для програмування мікроконтролерів. Мікроконтролери можна програмувати за допомогою .NET Micro Framework, який є версією .NET Framework, адаптованою для вбудованих пристроїв. .NET Micro Framework включає версію C# та Visual Studio, що дозволяє розробникам писати, компілювати та розгортати код на мікроконтролерах[10].

До переваг використання мови C# включають наступне:

1. надійна типізація та безпека типів;
2. можливості об'єктно–орієнтованого програмування;
3. багатоплатформна підтримка, включаючи Windows, Linux, macOS, Android та iOS;
4. легко читати та підтримувати код;
5. велика бібліотека корисного готового коду ;
6. оптимізація компілятора для підвищення продуктивності;
7. комплексний набір інструментів для налагодження.

На малюнку 2.2 зображено логотип мови програмування C#.

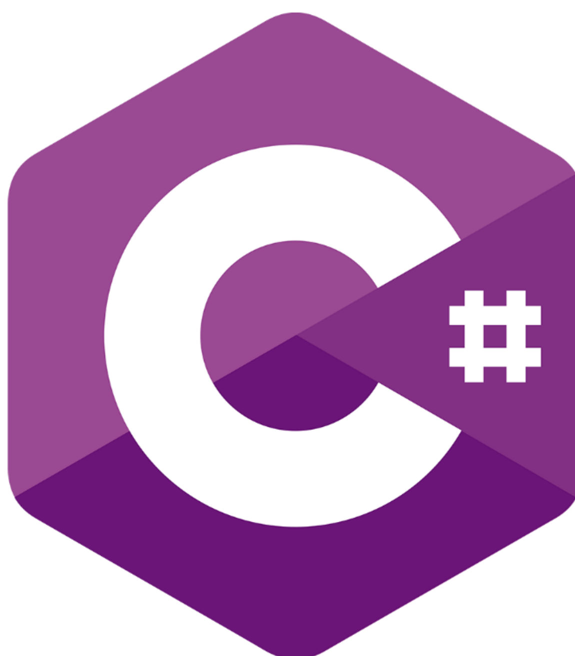

Рисунок 2.2 –Логотип мови програмування C#

У якості середовища розробки було обрано – Rider. Середовище розробки Rider це інтегроване середовище розробки (IDE), створене компанією JetBrains, яке спеціально розроблене для розробки на C# та .NET. Воно має широкий спектр функцій, що робить його чудовим вибором для .NET розробки та дозволяє розробникам швидко і легко створювати високоякісні додатки. На малюнку 2.3 зображено логотип середовища Rider.

Рисунок 2.3 – Логотип IDE Rider

Однією з найбільших переваг Rider є його швидкість та чуйність. Він швидко запускається і дозволяє розробникам відразу приступити до кодування, не турбуючись про повільне завантаження або очікування компіляції програми. Це робить його особливо придатним для роботи з великими проектами, які часто потребують тривалого завантаження та компіляції.

Rider також має ряд корисних функцій, які полегшують розробникам написання коду. Він має вбудований відладчик, який дозволяє розробникам легко налагоджувати свій код і знаходити джерело будь–яких помилок. Він також включає в себе інструмент рефакторингу, який допомагає розробникам реструктурувати свій код і зробити його більш читабельним і зручним для підтримки.

Ще однією чудовою особливістю Rider є його підтримка декількох фреймворків. Він підтримує декілька версій .NET, а також ASP.NET Core, Mono, Xamarin і Blazor. Це робить Rider чудовим вибором для розробників, які прагнуть створювати додатки, що можуть працювати на різних платформах.

Загалом, Rider є чудовим вибором для розробників, яким потрібне швидке та надійне середовище розробки для створення своїх .NET додатків. Воно розроблене таким чином, щоб бути простим у використанні і пропонує широкий спектр функцій, які роблять його добре придатним для розробки .NET. Він також дуже швидкий і чуйний, що робить його ідеальним для великих проектів, і має підтримку декількох фреймворків, що робить його чудовим вибором для розробників, яким потрібно створювати додатки для декількох платформ.

2.2.3 Побудування діаграми класів бібліотеки

Діаграми класів – це тип візуальної діаграми, яка допомагає документувати структуру та взаємозв'язки системи. Вони використовуються для передачі архітектури системи, а також для виявлення потенційних проблем. Діаграми класів є невід'ємною частиною програмної інженерії та системного дизайну і часто використовуються на початкових етапах розробки програмного забезпечення.

Діаграми класів складаються з класів та об'єктів і описують, як вони взаємодіють. Клас – це абстрактне поняття, яке пояснює ролі та обов'язки об'єктів у системі. Кожен клас зазвичай має набір атрибутів, тобто характеристик, які описують клас, і операцій, тобто функцій, які клас може виконувати.

Діаграми класів часто використовуються для моделювання бізнес–процесів, показують взаємозв'язки між об'єктами та ілюструють, як організовані дані в системі.

Вони також можуть бути використані для виявлення потенційних проблем у системі. Наочно показуючи поведінку об'єктів та їх взаємозв'язки, діаграми класів можуть допомогти забезпечити успіх системи.

Діаграми класів можна побудувати кількома способами. Команда розробників програмного забезпечення може використовувати такий інструмент, як UML (уніфікована мова моделювання) для візуальної побудови діаграми.

Крім того, діаграми класів можна створювати в електронних таблицях, таких як Microsoft Excel. Незалежно від використовуваного методу, діаграми класів слід ретельно продумати, щоб переконатися, що вони точно відображають потреби системи.

На закінчення, діаграми класів є безцінним інструментом для інженерів–програмістів і системних дизайнерів. Їх можна використовувати для документування структури та взаємозв'язків системи, а також для виявлення потенційних проблем.

Діаграми класів складаються з класів та об'єктів і можуть бути побудовані за допомогою різноманітних інструментів. Розуміючи призначення та функції діаграм класів, інженери–програмісти можуть створити потужний інструмент, який допоможе їм розробляти успішні системи.

Перед програмуванням буд ’якої програмної бібліотеки спочатку створюються діаграми класів. У них наведено які класи має бібліотека, опис елементів класів, таких як методи і члени класів, та зв’язки між класами.

У розробленій бібліотеці існує чотири класи:

1. **FiniteStateMachine** – Це програмний, не насілдуємий клас, який відповідає за відпрацювання логіки машини станів, переходи між станами, оновлення станів та інше;

2. **Transition** – Це програмний, не насілдуємий клас, який відповідає за перехід між двома станами, має у собі посилання на стани та функцію умов переходу;

3. **State** – Це програмний клас, який відповідає за базову логіку стану, яку користувач буде мати можливість наслідувати, реалізовувати базові методи та додавати свою персональну логіку поведінки;

4. **DataSystem** – Це програмний, не насілдуємий клас, який відповідає за обмін та збереження базових типів даних між усіма станами, переходами, та машиною станів.

На рисунку 2.4 зображено діаграму класів розробленої бібліотеки.

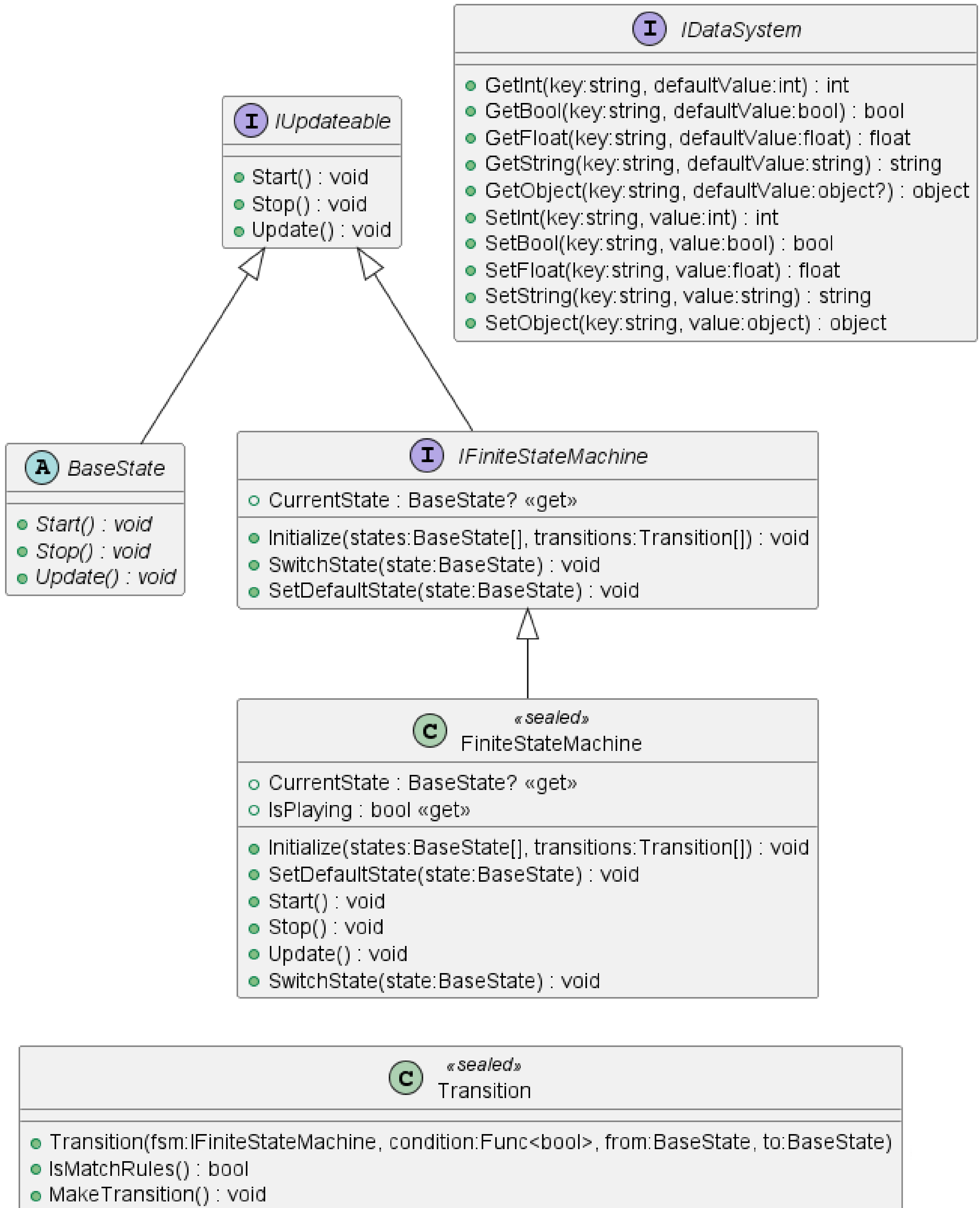

Рисунок 2.4 – Діаграма класів бібліотеки ШІ

2.3 Проєктування гри–прикладу на Unity

Unity – це потужний ігровий рушій, розроблений компанією Unity Technologies. Це один з найпопулярніших ігрових рушіїв, що використовується розробниками для створення ігор та симуляторів. Unity підтримує широкий спектр платформ, включаючи Windows, macOS, Linux, Android, iOS та інші. Однією з ключових особливостей Unity є підтримка бібліотек динамічних посилань (Dynamic Link Libraries, DLL), які дозволяють розробникам отримати доступ до широкого спектру бібліотек та функцій[11].

Використання DLL в Unity є досить простим. Для початку потрібно додати DLL до проекту. Це можна зробити або імпортувавши DLL зі сховища ресурсів, або помістивши їх до теки */Assets/Plugins*. Потім потрібно завантажити DLL у скрипти через простір імен System.Reflection. Це дозволить отримати доступ до функцій та об'єктів, що містяться всередині DLL.

Після завантаження DLL можна використовувати їхні функції для створення потужного та унікального ігрового досвіду. Наприклад, можна використати бібліотеку фізики для створення реалістичної фізики у своїй грі, або бібліотеку штучного інтелекту для створення розумних персонажів зі штучним інтелектом.

Використання DLL в Unity дозволяє розробникам скористатися перевагами існуючих бібліотек та функцій для створення дивовижних ігрових можливостей. За допомогою правильних бібліотек можна створювати більш потужні, захопливі та цікавіші ігри[12]. На рисунку 2.5 наведено логотип рушія Unity.

Рисунок 2.5 – Логотип рушія Unity

Гра–приклад повинна мати наступний функціонал і вимоги:

— відображення невеликої анімованої сцени;

— реалізувати ШІ персонажа за допомогою розробленої бібліотеки;

— відображення панелі з інформацією про роботу машини станів;

— відкриватися на Windows платформі та Android.

Для відображення даних машини станів використовується внутрішньо ігровий графічний інтерфейс. Тому було вирішено створити макет цього інтерфейсу для подальшої реалізації у грі.

Макет інтерфейсу це візуальне представлення користувацького інтерфейсу додатку або веб–сайту і того, як користувач взаємодіє з ним. Він використовується дизайнерами, щоб проілюструвати взаємодію користувача з продуктом до початку розробки.

На рисунку 2.6 зображено макет інтерфейсу розробляємої гри–прикладу.

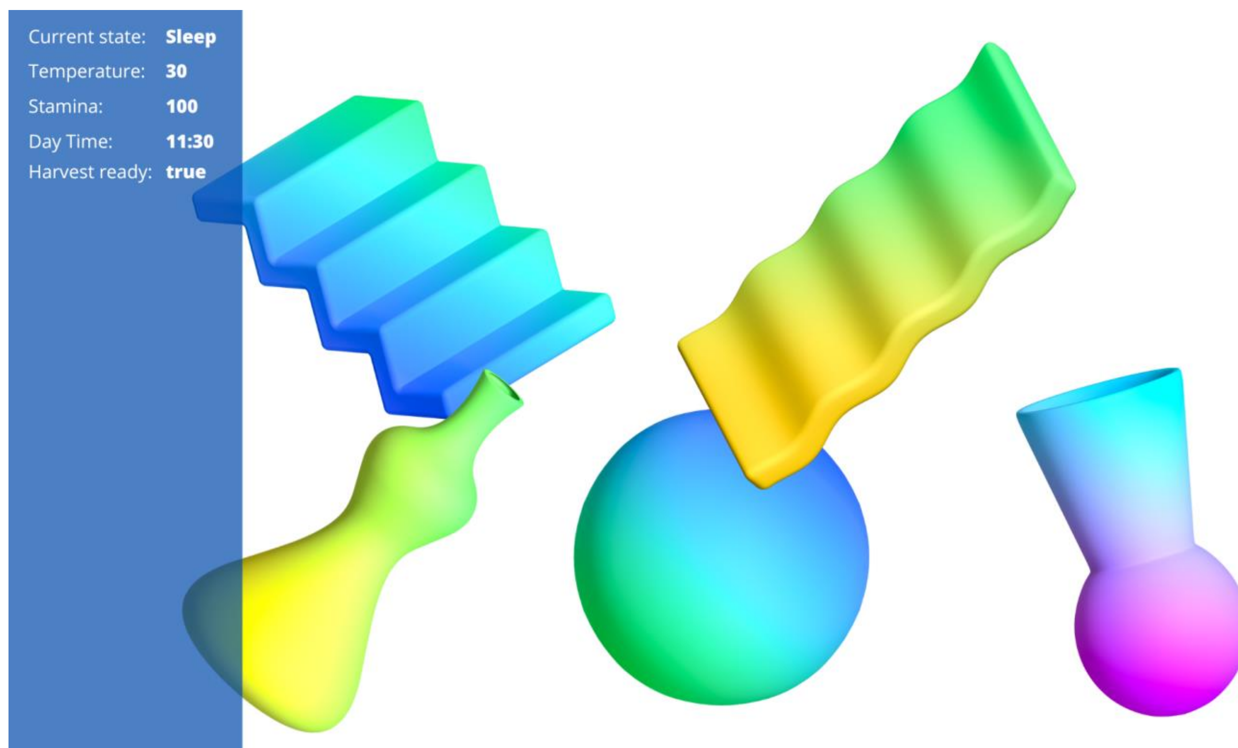

Рисунок 2.6 – Макет інтерфейсу гри прикладу

Одним із останніх етапів проєктування гри прикладу, є ескіз ігрової сцени або декількох сцен. **Ескіз ігрової сцени** – це грубий рисунок, який використовується для планування елементів художнього оформлення гри, таких як розміщення персонажів, макет рівнів і композиція сцени. Зазвичай його виконують у швидкому, вільному стилі, щоб швидко візуалізувати ідеї. На рисунку 2.7 зображено ескіз головної сцени, де будуть розміщенні усі необхідні елементи для демонстрації роботи машини станів.

Рисунок 2.7 – Ескіз ігрової сцени

Також для створення гри–прикладу використовувалось наступне програмне забезпечення:

— **Blender** – це безкоштовний пакет для створення 3D–зображень з відкритим вихідним кодом. Він підтримує весь конвеєр 3D–моделювання, ригінг, анімацію, симуляцію, рендеринг, композитинг, відстеження руху та відеомонтаж;

— **Rider –** це інтегроване середовище розробки (IDE) для платформи .NET. Це сучасний редактор з такими функціями, як IntelliSense, рефакторинг та аналіз коду. Він також постачається з потужними можливостями налагодження, модульного тестування та покриття коду;

— **Photoshop** – це професійне програмне забезпечення для редагування фотографій, створене компанією Adobe. Його використовують професійні фотографи, графічні дизайнери та веб–дизайнери для створення та покращення цифрових зображень. Він має широкий спектр функцій, які дозволяють користувачам маніпулювати своїми цифровими зображеннями у різний спосіб.

2.5 Проєктування апаратного прикладу

Оскільки бібліотека повинна мати можливість запускатися на багатьох платформах, вона також має запускатися на різноманітних мікроконтролерах та платах. Для демонстрування роботи бібліотеки на апаратній частині, потрібно розробити програмне забезпечення для певної плати, використовуючи розроблену бібліотеку. Оскільки бібліотека це DLL файл, похідний код якого написаний на C#, писати код для плати потрібно також на C#. Для цього потрібно використовувати спеціальне програмне забезпечення яке дозволяє писати та компілювати код для плат та мікроконтролерів. Одним із варіантів такого ПЗ є Visual Studio 2017 та .NET Nano Framework.

Visual Studio IDE – це інтегроване середовище розробки середовище розробки (IDE) від Microsoft, яке використовується для створення різноманітних додатків на C#. Це потужний інструмент для розробників, оскільки він має широкий спектр можливостей, таких як налагодження, IntelliSense та рефакторинг. Крім того, вона підтримує декілька мов програмування, включаючи JavaScript, Python та HTML. Він також надає велику бібліотеку шаблонів і компонентів, що робить розробку додатків швидшою і простішою. Це також дозволяє розробникам скористатися перевагами хмарних сервісів та інтегруватися з системою контролю вихідних кодів для спільної роботи над проектами з іншими розробниками. Visual Studio IDE надає повний і всеосяжний набір інструментів, які допомагають розробникам створювати додатки, які є міцними, надійними і, найголовніше, безпечними.

NanoFramework – це легка та модульна платформа з відкритим вихідним кодом та модульна платформа, яка дозволяє писати та запускати керований код (C# або Visual Basic) на вбудованих пристроях з обмеженими можливостями. Це сучасне, безпечне та готове до використання рішення для вбудованих пристроїв, яке є безкоштовним і не вимагає відрахувань. NanoFramework займає мало місця і оптимізований для роботи на недорогих мікроконтролерах з обмеженою пам'яттю та обчислювальною потужністю. Він забезпечує уніфікований досвід розробки, поєднуючи простоту використання .NET з гнучкістю вбудованої розробки.

NanoFramework складається з середовища виконання, яке забезпечує середовище виконання для додатків, написаних на C# або Visual Basic, а також набору основних бібліотек, які забезпечують доступ до апаратного забезпечення пристрою.

Додатки, написані за допомогою NanoFramework API, мають доступ до широкого спектру можливостей, включаючи, але не обмежуючись ними:

— багатопотокове виконання: NanoFramework дозволяє розробникам писати багатопотокові додатки, які можуть працювати паралельно;

— функції реального часу: NanoFramework підтримує переривання в реальному часі та надає доступ до низькорівневих апаратних функцій;

— мережа та підключення: NanoFramework дозволяє розробникам створювати додатки, які можуть взаємодіяти через мережу або підключатися до хмарних сервісів;

— міжпроцесний зв'язок: NanoFramework дозволяє додаткам взаємодіяти один з одним за допомогою механізму міжпроцесної комунікації;

— безпека та аутентифікації: NanoFramework забезпечує безпечну автентифікацію та шифрування для додатків.

Це робить NanoFramework ідеальною платформою для розробки вбудованих пристроїв для Інтернету речей (IoT). Завдяки невеликій площі, низькій вартості та широкому спектру функцій NanoFramework є ідеальною платформою для розробки додатків для вбудованих пристроїв.

Висновки до розділу 2

Для розробки було обрано мову програмування C# та інтегроване середовище розробки Rider. У якості формату бібліотеки було обрано DLL через те, що він уніфікований та може використовуватись на будь–якому ПК з операційною системою Windows.

Для тестування бібліотеки та демонстрації функціоналу, було спроєктовано два проєкти. Перший проєкт – це візуалізація та приклад використання на ігровому рушії Unity. Другий проєкт – апаратно–програмна демонстрація, на основі мікросхеми, програмування якої буде відбуватися за допомогою Visual Studio 2017 та .NET Nano Framework.

3 РОЗРОБКА ПРОГРАМНОЇ БІБЛІОТЕКИ. РОЗРОБКА ПРОЄКТІВ–ДЕМОНСТРАЦІЇ

3.1 Розробка програмної бібліотеки

Для розробки програмної бібліотеки використовувалась мова програмування C# та Jet Brains Rider IDE. Реалізується бібліотека за визначеною діаграмою класів.

Спочатку описуються всі інтерфейси бібліотеки.

Інтерфейс *IUpdateable* (див. рис. 3.1) відповідає за логіку оновлювання, він має методу початку, зупинки та оновлення.

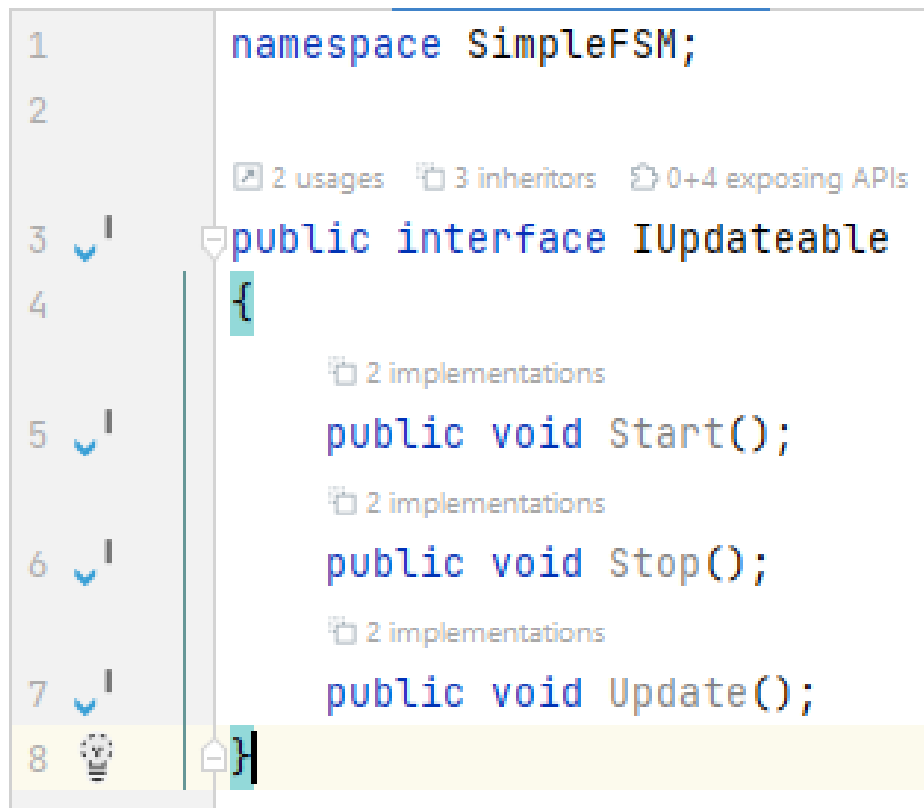

Рисунок 3.1 – Код інтерфейсу IUpdateable

Інтерфейс *IFiniteStateMachine* (див. рис. 3.2) унаслідується від інтерфейсу *IUpdateable*, та має додаткові методи для ініціалізації і зміни станів, встановлення стану за замовченням, та отримання поточного стану.

```
namespace SimpleFSM;

3 usages  1 inheritor
public interface IFiniteStateMachine : IUpdateable
{
    2 usages  1 implementation
    public BaseState? CurrentState { get; }
    1 implementation
    public void Initialize(BaseState[] states, Transition[] transitions);
    1 usage  1 implementation
    public void SwitchState(BaseState state);
    1 implementation
    public void SetDefaultState(BaseState state);
}
```

Рисунок 3.2 – Код інтерфейсу IFiniteStateMachine

Інтерфейс *IDataSystem* (див. рис. 3.3) визначає методи для отримання та встановлення базових типів даних за ключем який являється строчкою. Цих типів даних цілком достатньо щоб реалізувати будь яку логіку. Також він визначає метод для перевірки чи існує запис за певним ключем.

```
1   namespace SimpleFSM;
2
    14 usages   1 inheritor   1 exposing API
3   public interface IDataSystem
4   {
        1 implementation
5       public int GetInt(string key, int defaultValue = 0);
        1 implementation
6       public bool GetBool(string key, bool defaultValue = false);
        1 implementation
7       public float GetFloat(string key, float defaultValue = 0f);
        1 implementation
8       public string GetString(string key, string defaultValue = "");
        1 implementation
9       public object? GetObject(string key, object? defaultValue = null);
        1 implementation
10      public void SetInt(string key, int value);
        1 implementation
11      public void SetBool(string key, bool value);
        1 implementation
12      public void SetFloat(string key, float value);
        1 implementation
13      public void SetString(string key, string value);
        1 implementation
14      public void SetObject(string key, object value);
        1 implementation
15      bool ContainsKey(string key);
16  }
```

Рисунок 3.3 – Код інтерфейсу IDataSystem

Наступним етапом було розроблення базової логіки станів та переходів. Для цього було створено два класи *BaseState* та *Transition*.

Клас *BaseState* (див. рис. 3.4) зроблено абстрактним для того щоб його можна було наслідувати та додавати власну логіку. Він має посилання на об’єкт інтерфейсу *IDataSystem* яке встановлюється у конструкторі, він потрібен для обміну даними між станами, переходами, та машиною станів. Також він має абстрактні методи які є наслідувальними від *IUpdateable*.

```
1   namespace SimpleFSM;
2
    13 usages   4 exposing APIs
3   public abstract class BaseState : IUpdateable
4   {
5       protected readonly IDataSystem DataSystem;
6
7       protected BaseState(IDataSystem dataSystem)
8       {
9           DataSystem = dataSystem;
10      }
11
        2 usages
12      public abstract void Start();
        2 usages
13      public abstract void Stop();
        1 usage
14      public abstract void Update();
15  }
```

Рисунок 3.4 – Код класу BaseState

Клас *Transition* (див. рис. 3.5) має посилання на машину станів, метод перевірки умов переходу, стан звідки відбувається перехід та куди, вони встановлюються у конструкторі. Також він має метод для перевірки чи є потрібні умови для переходу у поточний момент часу, та метод для виконання переходу. Клас є закритим для наслідування оскільки має усю необхідну логіку і не передбачає наслідування.

```
namespace SimpleFSM;

public sealed class Transition
{
    private readonly Func<bool> _condition;
    private readonly IFiniteStateMachine _fsm;
    private readonly BaseState _from;
    private readonly BaseState _to;

    public Transition(IFiniteStateMachine fsm, Func<bool> condition,
        BaseState from, BaseState to)
    {
        _to = to;
        _from = from;
        _fsm = fsm;
        _condition = condition;
    }

    public bool IsMatchRules() => _fsm.CurrentState != null &&
                                  _fsm.CurrentState.Equals(_from)
                                  && _condition.Invoke();

    public void MakeTransition() => _fsm.SwitchState(_to);
}
```

Рисунок 3.5 – Код класу Transition

Останнім етапом розроблення програмної бібліотеки було додавання базової реалізації машини станів та системи управління даними. Оскільки бібліотека має інтерфейси для цих систем, користувач залишає за собою право використовувати базову реалізацію цих систем або зробити власну реалізацію.

Клас *DataSystem* (див. рис. 3.6) реалізує вище описаний інтерфейс *IDataSystem*. У якості структури сховища даних використовується словник, де ключ це строчка, а значення базовий тип – *object*. Далі всі значення перетворюються у потрібний тип даних залежно від того який метод реалізується.

```
namespace SimpleFSM;

public sealed class DataSystem : IDataSystem
{
    private readonly Dictionary<string, object> data = new Dictionary<string, object>();

    bool IDataSystem.ContainsKey(string key) => data.ContainsKey(key);

    int IDataSystem.GetInt(string key, int defaultValue) => GetValue(key, defaultValue);
    bool IDataSystem.GetBool(string key, bool defaultValue) => GetValue(key, defaultValue);
    float IDataSystem.GetFloat(string key, float defaultValue) => GetValue(key, defaultValue);
    string IDataSystem.GetString(string key, string defaultValue) => GetValue(key, defaultValue);
    object? IDataSystem.GetObject(string key, object? defaultValue) => GetValue(key, defaultValue);

    void IDataSystem.SetInt(string key, int value) => SetObject(key, value);
    void IDataSystem.SetBool(string key, bool value) => SetObject(key, value);
    void IDataSystem.SetFloat(string key, float value) => SetObject(key, value);
    void IDataSystem.SetString(string key, string value) => SetObject(key, value);
    void IDataSystem.SetObject(string key, object value) => SetObject(key, value);

    5 usages
    private void SetObject(string key, object value)
    {
        if (data.ContainsKey(key))
            throw new Exception( message: "Value with same key already exist!");

        data.Add(key, value);
    }

    1 usage
    private object? GetObject(string key, object? defaultValue = null)
    {
        return !data.ContainsKey(key) ? defaultValue : data[key];
    }

    5 usages
    private T GetValue<T>(string key, T defaultValue)
    {
        object? value = GetObject(key);

        return value switch
        {
            null => defaultValue,
            T castedValue => castedValue,
            _ => defaultValue
        };
    }
}
```

Рисунок 3.6 – Код класу DataSystem

Клас *FiniteStateMachine* (див. рис. 3.7–3.8) реалізує вище описаний інтерфейс *IFiniteStateMachine*. Ініціалізація класу відбувається у спеціальному методі, а не конструкторі оскільки об'єкт переходу може бути створений раніше і потребує посилання на об'єкт машини станів. Цей клас повинен бути обов'язково ініціалізованним щоб отримати посилання на масиви станів та переходів.

```
namespace SimpleFSM;

public sealed class FiniteStateMachine : IFiniteStateMachine
{
    8+2 usages
    public BaseState? CurrentState { get; private set; }
    public bool IsPlaying => !_stopped;

    private bool _stopped;
    private bool _initialized;
    private BaseState[]? _states;
    private Transition[]? _transitions;

    public void Initialize(BaseState[] states, Transition[] transitions)
    {
        _initialized = true;
        _transitions = transitions;
        _states = states;
        CurrentState = _states[0];
    }

    public void SetDefaultState(BaseState state)
    {
        if (!_initialized)
            throw new Exception( message: "FSM not initialized!");

        if(_stopped)
            return;

        if (_states == null || !_states.Contains(state))
            throw new Exception( message: $"Can`t switch state! State \"{state}\" not found!");

        CurrentState = state;
    }
```

Рисунок 3.7 – Код класу FiniteStateMachine, перша частина

```
    public void Start()
    {
        if (!_initialized)
            throw new Exception( message: "FSM not initialized!");

        _stopped = false;
        CurrentState?.Start();
    }

    public void Stop()
    {
        if (!_initialized)
            throw new Exception( message: "FSM not initialized!");

        _stopped = true;
        CurrentState?.Stop();
    }

    public void Update()
    {
        if (!_initialized)
            throw new Exception( message: "FSM not initialized!");

        if(_stopped || _transitions == null)
            return;

        for (int i = 0; i < _transitions.Length; i++)
            if(_transitions[i].IsMatchRules())
                _transitions[i].MakeTransition();

        CurrentState?.Update();
    }

    0+1 usages
    public void SwitchState(BaseState state)
    {
        if (!_initialized)
            throw new Exception( message: "FSM not initialized!");

        if (_states == null || !_states.Contains(state))
            throw new Exception( message: $"Can`t switch state! State \"{state}\" not found!");

        CurrentState?.Stop();
        CurrentState = state;
        CurrentState.Start();
    }
}
```

Рисунок 3.8 – Код класу FiniteStateMachine, друга частина

3.2 Розробка гри–демонстрації

Для розробки гри–прикладу використовувалось наступне програмне забезпечення: **Unity, Rider IDE, Photoshop, Blender**.

Першим кроком створення гри–прикладу було створення візуальної складової ігрової сцени. Для цього використовувалися різноманітні моделі, шейдери та освітлення. Більшість цих елементів було взято у відкритому доступі або куплено. Візуальна частина створювалась згідно ескізу ігрової сцени описаному вище. На рисунку 3.9 зображено створена ігрова сцена з усім необхідним візуалом.

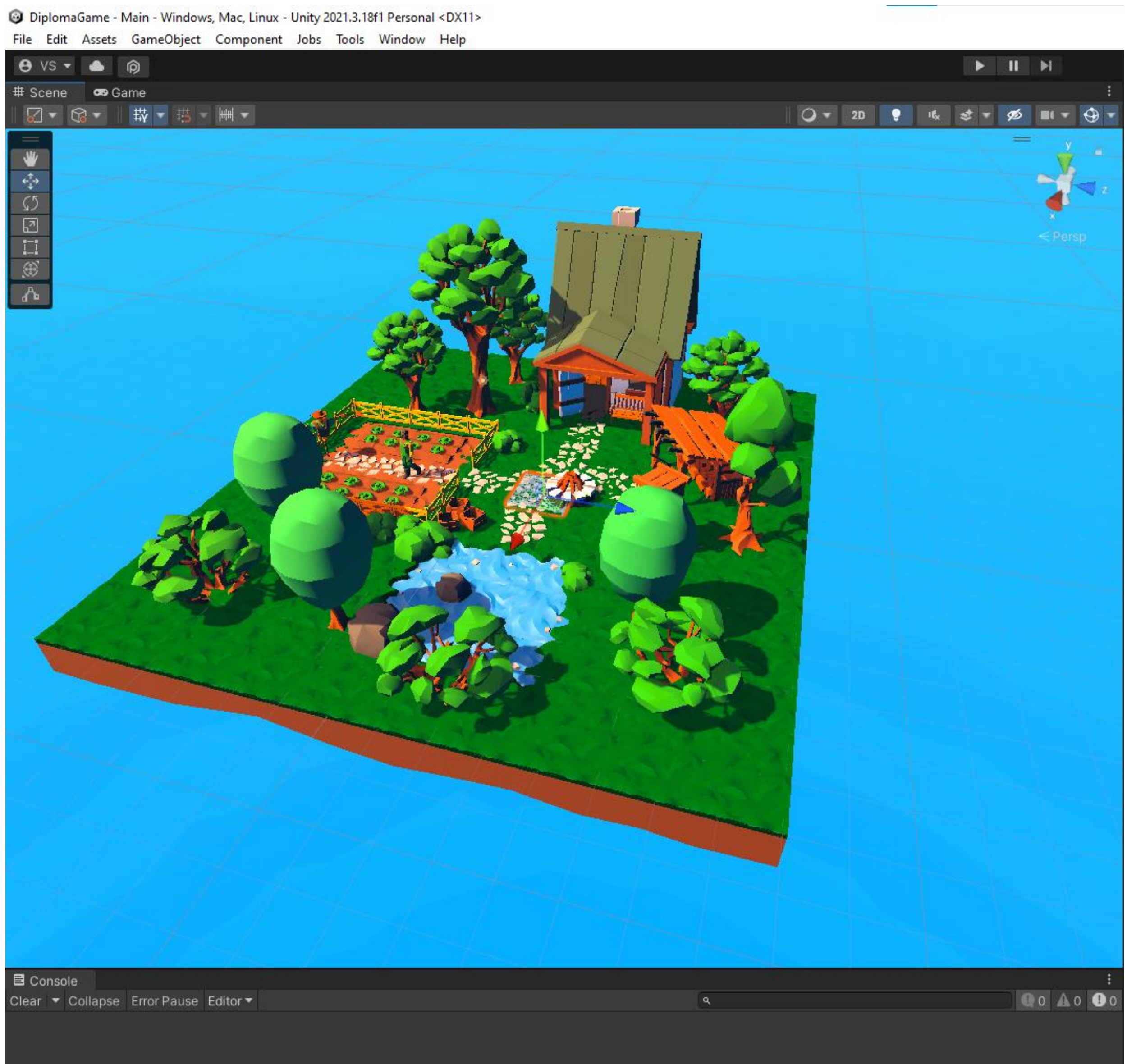

Рисунок 3.9 – Візуальна частина ігрової сцени

Наступним етапом було створення та налаштування анімації для персонажу. Обраний персонаж згідно з модельованої раніше системи повинен мати наступні анімації:

— анімація руху – ходьби;
— анімація збору врожаю – наклони та випади рукою;
— анімація сидіння – коли персонаж відпочиває;
— анімація відпочинку стоячи;
— анімація підпалювання багаття;
— анімація сну.

Після збору відповідних анімації, було налаштовано їх у спеціальному компоненті – *AnimationController*.

Компонент Animator Controller у Unity – це компонент, який дозволяє керувати поведінкою компонентів Аніматора. Він дозволяє контролювати хронометраж, переходи та інші аспекти анімованого персонажа. Його також можна використовувати для створення власних анімацій за допомогою об'єктів AnimationClip.

На рисунку 3.10 зображено процес налаштування переходів анімації.

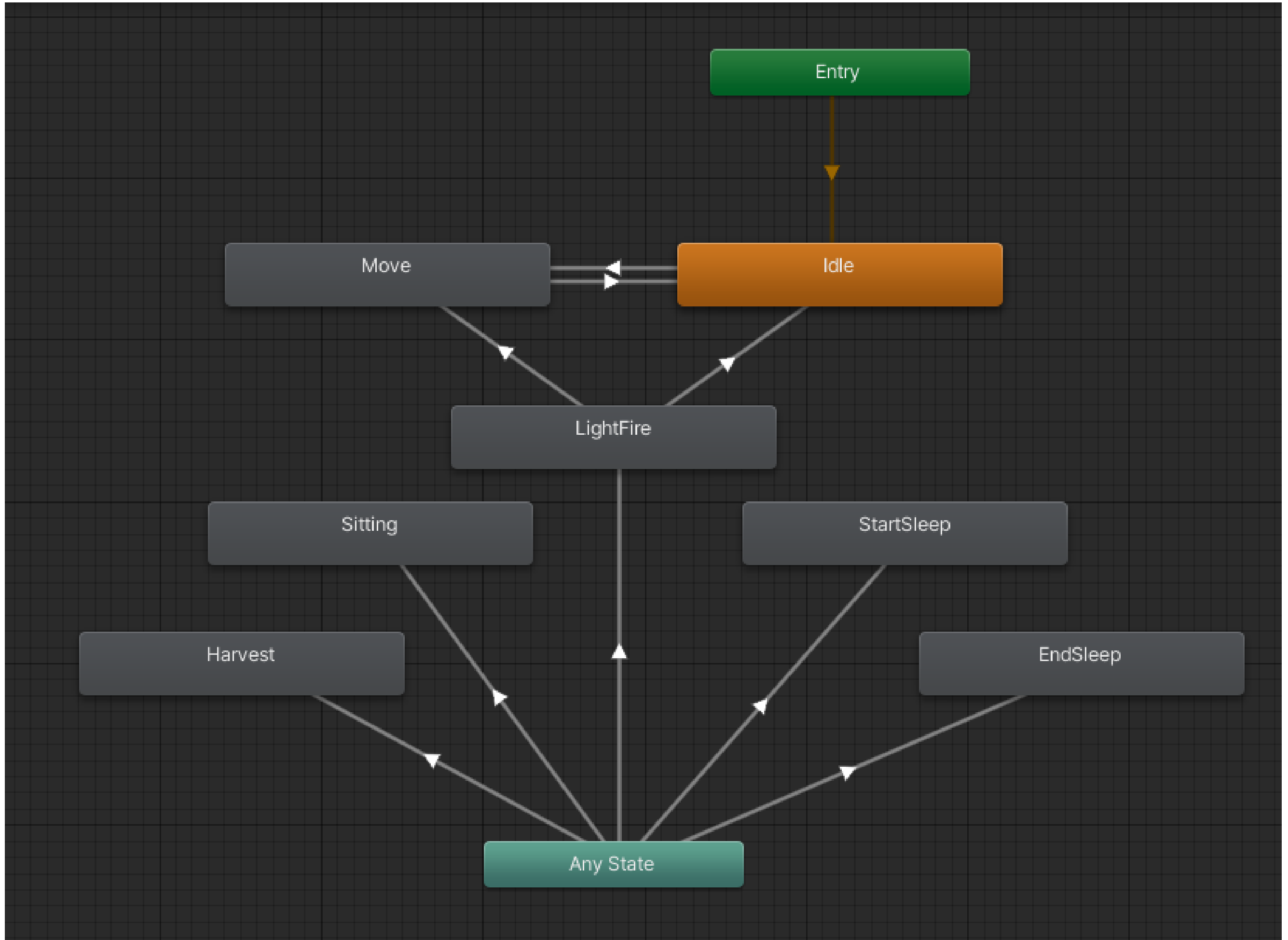

Рисунок 3.10 – Налаштування анімацій персонажу

Наступним етапом після створення візуалу ігрової сцени було створення і налаштування користувацького інтерфейсу по макету описаному раніше. Було створено спеціальний слайдер для приближення камери та інформаційну панель яку можна скривати та відкривати по бажанню. Слайдер у Unity – це елемент інтерфейсу, який дозволяє користувачам вибирати з діапазону значень, перетягуючи ручку. Повзунки можна використовувати для регулювання таких параметрів, як гучність, яскравість або застосування фільтрів зображення.

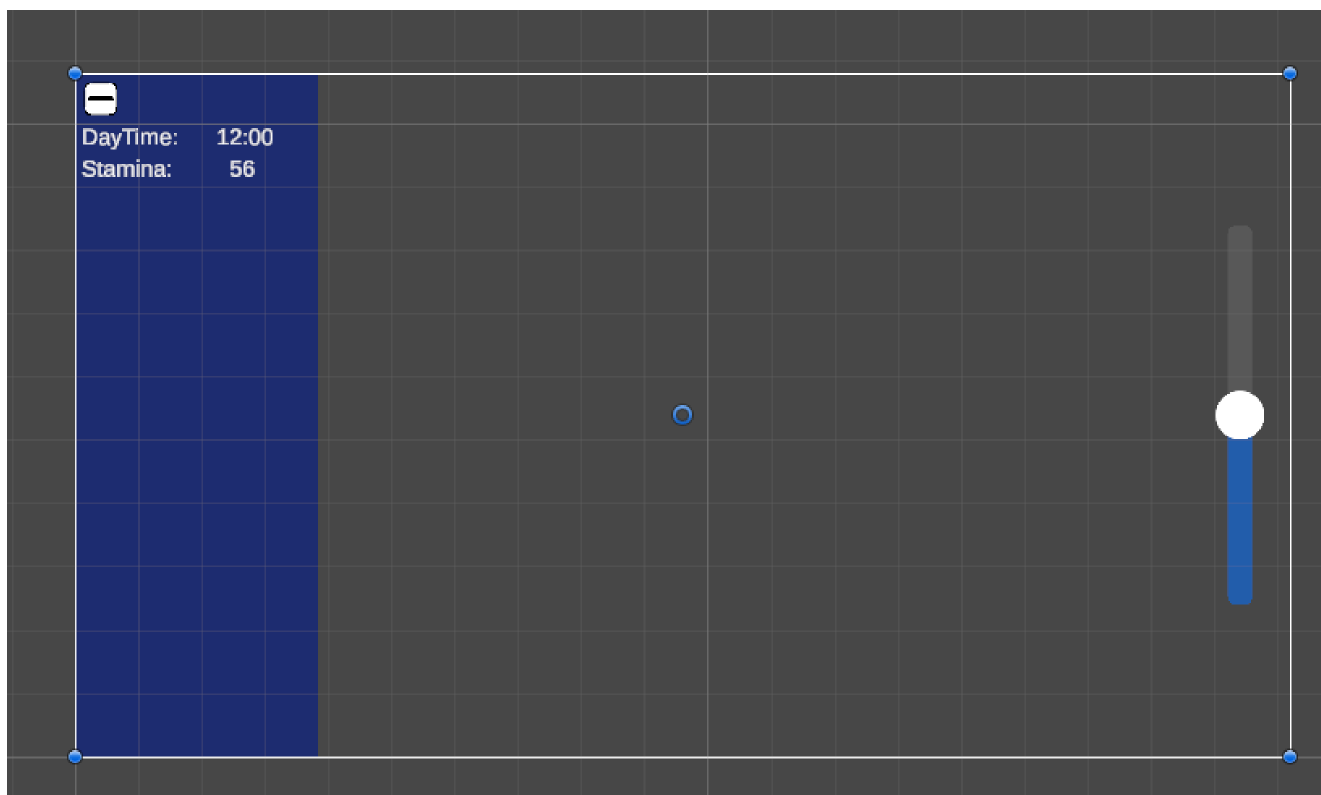

Рисунок 3.11 – Процес розробки користувацького інтерфейсу

Останнім етапом розробки гри–прикладу було написання коду ШІ персонажа, за допомогою розробленої бібліотеки. Було розроблено скрипти для управління анімаціями НІП, руху та взаємодії з предметами оточення. Також був написаний код для інтерактивних елементів гри, та керування камерою. На рисунку 3.12 наведено код штучного інтелекту НІП.

```
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;

namespace DemoGameFSM
{
    [RequireComponent(typeof(RectTransform))]
    public class PlayerInputUI : MonoBehaviour, IDragHandler, IBeginDragHandler, IEndDragHandler
    {
        [SerializeField] private Slider _slider;

        public bool IsDrag { get; private set; }
        public Vector2 DragDelta { get; private set; }

        public Slider Slider => _slider;

        private void OnDisable()
        {
            IsDrag = false;
            DragDelta = Vector2.zero;
        }

        private void LateUpdate()
        {
            DragDelta = Vector2.zero;
        }

        public void OnDrag(PointerEventData eventData)
        {
            Vector2 delta = eventData.delta;
            delta.x /= Screen.width;
            delta.y /= Screen.height;
            DragDelta = delta;
        }

        public void OnBeginDrag(PointerEventData eventData)
        {
            IsDrag = true;
        }

        public void OnEndDrag(PointerEventData eventData)
        {
            IsDrag = false;
            DragDelta = Vector2.zero;
        }
    }
}
```

Рисунок 3.12 – Код ШІ не ігрового персонажу

3.3 Розробка АПЗ демонстрації

Для розробки апаратно програмного проекту–демонстрації, було використано Visual Studio 2017 та .NET Nano Framework.

Visual Studio – це інтегроване середовище розробки (IDE) від Microsoft. Використовується для створення додатків на різних платформах, включаючи веб–, мобільні та настільні комп'ютери. Також використовується для створення ігор та веб–додатків. Пропонує широкий спектр функцій, включаючи редактор, налагоджувач, компілятор, систему збірки та багато іншого. Він також підтримує кілька мов програмування.

.NET NanoFramework – це потужна платформа для розробки додатків Інтернету речей (IoT). Це легка, модульна платформа, яка дозволяє розробникам створювати додатки, адаптовані до пристроїв, які вони використовують. Вона призначена для використання з Visual Studio і підтримує такі мови, як C# і Visual Basic .NET. Вона надає бібліотеку API, які можна використовувати для створення додатків, що працюють на таких пристроях, як Raspberry Pi та Arduino. Він також має вбудовану підтримку різноманітних датчиків та пристроїв.

Першим кроком перед початком програмування, потрібно завантажити образ .NET nanoFramework у флеш–пам'ять плати. Найкраще використовувати інструмент nano Firmware Flasher (nanoff). Це командний інструмент .NET Core CLI. Для цього потрібно знати COM–порт, підключений до пристрою.

На рисунку 3.13 зображено де можна подивитися інформацію про COM–порт який використовує плата.

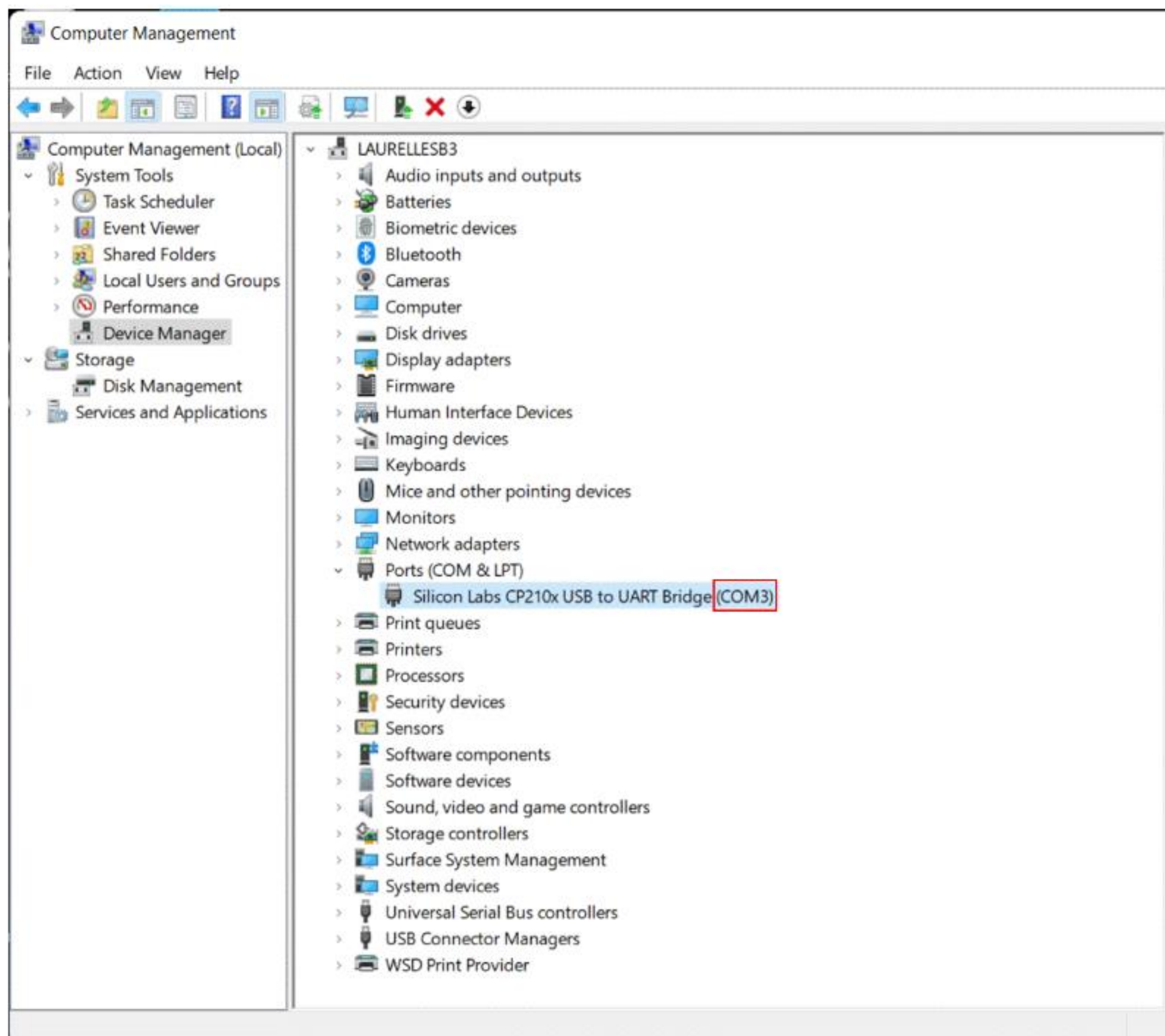

Рисунок 3.13 – Інформація про COM–порт

Далі потрібно було завантажити образ фреймворку на плату, через COM–порт який використовується. На рисунку 3.14 зображено, команді CLI для завантаження образу.

```
Windows PowerShell
Tool 'nanoff' (version '2.0.9') was successfully installed.
PS C:\repos\nanoFramework> nanoff --platform esp32 --serialport COM3 --update --preview
.NET nanoFramework Firmware Flasher v2.0.9+947a088d7c
Copyright (C) 2019 .NET Foundation and nanoFramework project contributors


Using COM3 @ 1500000 baud to connect to ESP32.
Reading details from chip...
OK

Connected to:
ESP32 (ESP32-D0WDQ6 (revision 1))
Features WiFi, BT, Dual Core, 240MHz, VRef calibration in efuse, Coding Scheme None
Flash size 4MB unknown from  (manufacturer 0x216 device 0x16406)
PSRAM: not available
Crystal 40MHz
MAC 7C:9E:BD:F6:05:8C


No target name was provided! Using 'ESP32_REV0' based on the device characteristics.

Trying to find ESP32_REV0 in development repository...OK
Extracting ESP32_REV0-1.7.3-preview.243.zip...OK

Updating to 1.7.3-preview.243

Erasing flash...
OK
Flashing firmware...
Writing at 0x000f9c74... (90 %)
```

Рисунок 3.14 – Команди завантаження образу nano Framework

Останнім кроком було створення проекту nano Framework, підключення розробленої бібліотеки ШІ, та написання простого коду для відображення станів на дисплеї. Через спеціальне API яке надає фреймворк, було отримано доступ то дисплею пристрою та виведено на нього номер поточного стану та цифру залежно від котрої змінюються стани.

Висновки до розділу 3

Було розроблено програмну бібліотеку для написання ШІ на основі скінченного автомату за допомогою мови C# та скомпільовано у DLL файл. Бібліотека має вже реалізовані класи машини станів та системи обміну даними а також надає можливість користувачу реалізовувати стани там машину власним чином. Бібліотека має дуже простий та зрозумілий функціонал та при компіляції займає 112 кілобайт, що важливо при використанні на пристроях з обмеженими ресурсами.

Для тестування та демонстрування використання бібліотеки було розроблено ігровий додаток–демонстрацію, у якому реалізовано штучний інтелект не ігрового персонажу за допомогою розробленої бібліотеки. Додаток може запускатися на операційній системі Android та Windows, та має можливість керування камерою і виводить інформацію про машину станів на користувацький інтерфейс.

Також для тестування та демонстрування бібліотеки, було розроблено інший проєкт–приклад. Це програмне забезпечення для плати Arduino Uno 3 та LCD дисплею, яке демонструє роботу машини станів та виводить на дисплей поточний стан та значення яке використовується для переходу між станами.

Розробивши два проєкти приклади було доведено що розроблена бібліотека може використовуватися як і ігрової індустрії на ігрових рушіях, так і для різноманітних вбудованих систем.

4 ТЕСТУВАННЯ БІБЛІОТЕКИ. ДОСЛІДЖЕННЯ ПРОЦЕССУ РОЗРОБКИ ШІ ЗА ДОПОМОГОЮ РОЗРОБЛЕНОЇ БІБЛІОТЕКИ. АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕНЬ ТА ТЕСТУВАННЯ

4.1 Тестування бібліотеки за допомогою гри–демонстрації

В першу чергу було вирішено протестувати бібліотеку за допомогою ручного тестування. **Ручне тестування** – це процес ручної перевірки програмного забезпечення на наявність дефектів. Він вимагає від тестувальника ручного виконання тестових кейсів без використання будь–яких засобів автоматизації. Ручне тестування підходить для тестування користувацьких інтерфейсів, продуктивності, юзабіліті та сумісності.

Переваги ручного тестування:

1. ручне тестування є недорогим та ефективним рішенням у порівнянні з автоматизованим тестуванням;
2. ручне тестування можна використовувати для перевірки функціональності;
3. ручне тестування можна використовувати для перевірки користувацького досвіду;
4. Ручне тестування забезпечує краще покриття програми.

Недоліки ручного тестування:

1. ручне тестування займає багато часу і є трудомістким;
2. ручне тестування схильне до людських помилок;
3. ручне тестування може вимагати більше ресурсів;
4. ручне тестування важко підтримувати та оновлювати.

Ручне тестування ігор передбачає проходження гри як гравець та оцінку її продуктивності, якості та стабільності. Це можна зробити вручну, керуючи грою за допомогою геймпада або клавіатури, або написавши код для управління грою. Тестувальники повинні шукати будь–які помилки або збої, які можуть порушити користувацький досвід. Вони також повинні перевірити візуальну складову, звук і зручність гри, щоб переконатися, що вона відповідає стандартам розробника.

Задля тестування бібліотеки було розроблено спеціальну гру–приклад яка імітує поведінку штучного інтелекту не ігрового персонажа – фермера. Цей персонаж має поведінку яка змінюється залежно від внутрішніх та зовнішніх факторів.

Логіка поведінки НІП наступна: персонаж прокидається зранку, та йде працювати, збирати врожай. Фермер має запас сил який поступово зменшується коли він працює, коли запас повністю закінчується персонаж починає відпочивати. Випадковим чином у день може піти дощ, у такому випадку персонаж не залежно чим він займався піде ховатися від дощу. Також у грі імітується зміна доби, тому коли стає достатньо темно і температура знижується то певного значення, персонаж іде розпалювати багаття. Коли наступає ніч, персонаж іде спати і встає наступним ранком. Таким чином цикл повторюється.

Переглядаючи гру–приклад, та імітуючи різноманітні події зовнішніх та внутрішніх факторів, було визнано що бібліотека працює правильно відповідно до запланованої поведінки. На рисунку 4.1 зображено процес дослідження та тестування роботи бібліотеки на основі гри–прикладу.

Рисунок 4.1 – Тестування роботи ШІ

4.2 Тестування бібліотеки у внутрішньому середовищі

Наступним кроком для того щоб остаточно переконатися у правильності роботи бібліотеки було проведено автоматизоване юніт тестування.

Автоматизоване тестування – це процес запуску тестів на програмних додатках, щоб переконатися, що програма працює так, як очікується. Автоматизовані тести можуть бути чудовим способом виявлення помилок та покращення якості програмного продукту.

Юніт–тести – це особливий вид автоматизованих тестів, які фокусуються на одному компоненті коду. Юніт–тести призначені для перевірки конкретних результатів для заданих вхідних даних. Щоб забезпечити точність юніт–тесту, код повинен бути написаний таким чином, щоб його можна було легко протестувати і щоб його було легко зрозуміти. Юніт–тести також повинні бути написані таким чином, щоб можна було легко ізолювати компонент, який тестується.

При написанні автоматизованих тестів і модульних тестів важливо враховувати типи тестів і те, як їх можна найкраще застосувати до коду. Автоматизовані тести можна використовувати для перевірки на наявність помилок і переконатися, що код працює належним чином. Вони також можуть бути використані для перевірки продуктивності шляхом запуску серії тестів з різними вхідними даними або сценаріями.

Юніт–тести, з іншого боку, використовуються для перевірки того, що конкретний компонент коду працює так, як очікується. Автори також повинні враховувати дані, які використовуються для тестів, і те, як вони повинні бути записані в чіткій і стислій формі.

Автоматизовані тести та модульні тести повинні бути частиною процесу розробки і повинні виконуватися регулярно. Це дозволяє зробити процес розробки більш ефективним і надійним. Автоматизовані та модульні тести допомагають переконатися, що програмний додаток працює так, як очікується, і допомагають запобігти помилкам, які можуть призвести до значних витрат на розробку та підтримку програмного забезпечення.

Під час написання модульних тестів слід пам'ятати, що тести не залежать один від одного, що тести охоплюють всі гілки коду, і що вони повинні бути написані якомога меншими, щоб гарантувати їхню швидкість та ефективність. Ви також повинні переконатися, що тести є повторюваними, тобто вони будуть поводитися однаково при кожному виконанні.

Для розробленої бібліотеки було створено юніт тести для кожного класу та методу а також симуляція простої поведінки ШІ зі зміною станів відносно випадкових чисел. На рисунках 4.2–4.3 наведено код тестів.

```
using System.Reflection.Emit;
using SimpleFSM;
using Xunit;
using Xunit.Abstractions;
using Xunit.Sdk;

namespace DefaultNamespace;

[TestCaseOrderer( ordererTypeName: "Test",  ordererAssemblyName: "TestOrder")]
public class FSMTest : ITestClass
{
    public ITypeInfo Class { get; }
    public ITestCollection TestCollection { get; }

    public void StartTest()
    {
        var fsm = new FiniteStateMachine();
        fsm.Start();
        fsm.Stop();
        fsm.Update();
        fsm.Initialize( states: null,  transitions: null);
        fsm.SwitchState(null);
        fsm.SetDefaultState(null);
```

Рисунок 4.2 – Код юніт тесту частина перша

```
25          TestMethod testMethodStart =
26              new TestMethod( class: this,
27              new ReflectionMethodInfo(new DynamicMethod( name: "Start",
28                  typeof(void),  parameterTypes: null,  owner: fsm.GetType())));
29          TestMethod testMethodStop =
30              new TestMethod( class: this,
31              new ReflectionMethodInfo(new DynamicMethod( name: "Stop",
32                  typeof(void),  parameterTypes: null,  owner: fsm.GetType())));
33          TestMethod testMethodUpdate =
34              new TestMethod( class: this,
35              new ReflectionMethodInfo(new DynamicMethod( name: "Update",
36                  typeof(void),  parameterTypes: null,  owner: fsm.GetType())));
37          TestMethod testMethodInitialize =
38              new TestMethod( class: this,
39              new ReflectionMethodInfo(new DynamicMethod( name: "Initialize",
40                  typeof(void),
41                  parameterTypes: new []{typeof(BaseState[]),
42                      typeof(Transition[])},  owner: fsm.GetType())));
43          TestMethod testMethodSwitchState =
44              new TestMethod( class: this,
45              new ReflectionMethodInfo(new DynamicMethod( name: "SwitchState",
46                  typeof(void),
47                  parameterTypes: new []{typeof(BaseState)},  owner: fsm.GetType())));
48          TestMethod testMethodSetDefaultState =
49              new TestMethod( class: this,
50              new ReflectionMethodInfo(new DynamicMethod( name: "SetDefaultState",
51                  typeof(void),  parameterTypes: new []{typeof(BaseState)},  owner: fsm.GetType())));
52      }
53
54      public void Deserialize(IXunitSerializationInfo info)
55      {
56          return;
57      }
58
59      public void Serialize(IXunitSerializationInfo info)
60      {
61          return;
62      }
63  }
```

Рисунок 4.3 – Код юніт тесту частина перша

4.3 Дослідження і порівняння процесу розробки ШІ за допомогою розробленої бібліотеки та аналогів

Перед останнім етапом було проведено дослідження та порівняння власної бібліотеки з аналогом, у якості аналогу було вирішено використовувати бібліотеку BlazeAI для ігрового рушія Unity.

Blaze AI – пропонує дуже спрощений підхід до створення штучного інтелекту і не нав'язує певну методологію або фреймворк, він реалізує власний скінченний автомат. Однак, Blaze дає свободу робити це будь–яким способом у власних сценаріях і моноповедінці. Це означає, що Blaze можна інтегрувати з будь–якою системою або ресурсом. Незалежно від того, що це. Він також працює з візуальними сценаріями.

Кожен стан має власний сценарій поведінки, який є моноповедінкою, що може бути відредагований або навіть написаний з sctach для отримання поведінки з високим ступенем кастомізації. Можливо навіть поміняти місцями сценарії поведінки станів під час виконання, щоб змусити ворога діяти по–іншому за певних умов. Нарешті, не всі стани потрібно використовувати. Використовуйте те, що вам потрібно, а решту залиште порожніми.

Blaze пропонує численні API та доступ до властивостей для повного динамічного контролю над власними штучними інтелектами. До всіх властивостей інспектора можна отримати доступ і динамічно змінювати їх за допомогою коду, щоб змінити поведінку ШІ під час виконання. Всі API та загальнодоступні властивості перераховані в документації.

Blaze AI стверджує що він «оптимізований за кодом і створено з думкою про продуктивність». Пропонує відбір на відстані, кадри циклів зору, аудіо ШІ в об'єктах зі скриптами для меншого споживання пам'яті та шари, що налаштовуються, у всіх фізичних операціях.

Далі буде описано процес інтеграції плагіну Blaze AI і налаштування базового ШІ НІП.

1. імпортується плагін в проєкт Unity, архівований плагін важить 57.7MB, після імпортування плагіну він займає вже 285MB;

2. налаштувати стандартний інструмент Unity для пошуку шляху – NavMesh. Для цього потрібно створити межи для пресування та налаштувати параметри агента, після запустити процес побудови сітки поверхні;

3. створити об’єкт НІП та додати до нього компонент Blaze AI;

4. додати стан за замовчуванням, натиснувши на кнопку «Add Behaviour» на компоненті. Плагін дозволяє створювати та додавати власні стани;

5. налаштувати анімації та передати їх нави у створений стан;

6. далі потрібно вибрати та налаштувати точку патрулювання, які за замовчуванням вибираються випадково на навігаційній сітці;

7. у розділі «Зір» можна за бажанням налаштувати теги або шари ворогів, та кути зору агента;

8. після усіх маніпуляції потрібно зберегти налаштування, та запустити сцену, після чого можна побачити що створений агент рухається по сцені.

Після створення простого ШІ агента за допомогою Blaze AI, був створений ШІ агента з аналогічним функціоналом але за допомогою розробленої бібліотеки.

1. імпортувати плагін в проєкт. Розроблена DLL бібліотека має розмір у 9KB тому дуже швидко імпортується і не займає багато місця;

2. налаштувати інструмент для пошуку шляху. Оскільки розроблена бібліотека не має залежності від рушію, фреймворку чи інших плагинів, у якості інструмента пошуку шляху може використовуватися будь який плагін або інше. Але, для прикладу використовувався стандартний інструмент Unity – NavMesh. Тому цей етап аналогічний як і у випадку Blaze AI;

3. створити компонент який буде використовувати машину станів із розробленої бібліотеки, та додати його на будь який об’єкт у грі;

4. створити стани для руху і вибору точок патрулювання та додати їх у машину станів;

5. налаштувати анімації та викликати їх у створений станах;

6. запустити гру та перевірити що агент рухається за визначеними точкам напрямку.

Після розроблення ШІ НІП двома різними засобами, за результатами цих процесів було розроблено порівняльну таблицю для розробленої бібліотеки та аналогу Blaze AI. У таблиці 4.1 наведено порівняння двох цих засобів.

Таблиця 4.1 – Порівняння інструментів створення ШІ

Характеристика	Власне рішення	Blaze AI
Незалежність від фреймворку та рушія. Можливість використовувати на інших рушіях або вбудованих системах.	Так	Ні
Можливість створювати власні стани	Так	Так
Реальний розмір плагіну	9 кілобайт	285 мегабайт
Можливість створити власну імплементацію автомату	Так	Ні
Залежність від сторонніх плагинів/інструментів	Ні	Так
Швидкість створення простого ШІ	Приблизно 80 хв.	Приблизно 95 хв.
Наповненість додатковим функціоналом (зір, рух, анімації, аудіо і т.д.)	Ні	Так
Оптимізація/швидкодія	12ms +– 2	12ms +– 2

Підводячи підсумки можна сказати що кожен із плагинів має як переваги так і недоліки, але залежить від мети використання. Розроблена бібліотека є більш уніфікованою та універсальною. Вона може використовуватися у багатьох середовищах, рушіях, системах. Через це багато функціоналу користувачу потрібно робити самому, наприклад системи рухів, анімацій і т.д. Також перевагою розробленої системи є її маленький розмір, що може бути доцільно у використанні у системах з обмеженими ресурсами.

4.4 Аналіз результатів тестування та досліджень

Першим етапом тестування було забезпечення точності структури та вихідного коду бібліотеки. На цьому етапі було перевірено, що весь код є коректним, і що всі компоненти можуть належним чином взаємодіяти один з одним.

Другий етап тестування передбачав запуск бібліотеки в змодельованих умовах, щоб переконатися, що її результати є точними і передбачуваними. Аналізуючи отримані дані можна сказати що цей етап тестування пройдений, ШІ діє відповідно заданих інструкцій.

Третім етапом тестування був запуск програми з реальними даними, щоб імітувати реальне середовище, в якому буде працювати ШІ. Цей етап також був успішно пройдений.

Останній, четвертий етап тестування передбачав запуск бібліотеки в реальному середовищі, в якому вона буде використовуватися. Бібліотека запустилась у середовищі ігрового рушія та фреймворку для програмування плат, тому можна вважати цей етап також успішно завершеним.

Розроблена бібліотека має кілька переваг порівняно з дослідженим аналогом. По–перше, використання бібліотечної системи на основі детермінованого скінченного автомату набагато ефективніше та рентабельніше. Це пов'язано з тим, що бібліотечні системи на основі штучного інтелекту здатні швидко аналізувати дані в бібліотеці і робити кращі прогнози та рішення щодо ресурсів бібліотеки. Це дозволяє бібліотеці зосередитися на наданні більшої кількості послуг і ресурсів своїм користувачам, зменшуючи при цьому навантаження на бібліотечний функціонал.

По–друге, бібліотека може використовувати бібліотечні системи на основі штучного інтелекту для надання індивідуальних можливостей і ресурсів різним користувачам. Оскільки бібліотечний ШІ має доступ до даних у бібліотеці і може приймати рішення на основі цих даних, він може визначати використовуватися індивідуально і відповідно адаптувати бібліотечні функції та ресурси. Це може допомогти користувачу забезпечити кращий і більш персоналізований досвід для кожної задачі.

По–третє, бібліотечні системи на основі штучного інтелекту можуть надавати більш точну та актуальну інформацію про ресурси бібліотеки. Постійно відстежуючи бібліотечні дані, ця система може допомогти бібліотеці виявити тенденції та зміни в даних, які можуть бути використані для прийняття рішень про ресурси бібліотеки і про те, як їх максимально ефективно використовувати. Це може допомогти бібліотеці краще обслуговувати своїх відвідувачів і переконатися, що її ресурси використовуються з максимальною ефективністю.

Нарешті, бібліотечна система на основі штучного інтелекту може допомогти користувачу заощадити час і гроші. Оскільки бібліотечний штучний інтелект постійно відстежує дані бібліотеки, він може виявити і вжити заходів щодо будь–яких потенційних питань або проблем до того, як вони стануть надто серйозними. Це може заощадити час і гроші, які були б витрачені на вирішення проблеми після того, як вона виникла.

Загалом, бібліотечна система на основі штучного інтелекту забезпечує ефективний, економічно вигідний і персоналізований досвід, який може допомогти користувачу заощадити час і гроші, надаючи при цьому кращі послуги та менеджмент ресурсів.

Висновки до розділу 4

Переглядаючи приклад гри під час ручного тестування та моделювання різних подій зовнішніх і внутрішніх факторів, було виявлено, що бібліотека поводилася правильно відповідно до очікуваної поведінки.

Для розроблених бібліотек створено модульні тести для кожного класу та методу. Також була змодельована проста поведінка ШІ зі змінами стану відносно випадкових чисел. Результати тесту не виявили помилок.

Розроблена бібліотека має ряд переваг перед досліджуваною бібліотекою–аналогам.

По–перше, для простих передбачуваних ШІ НІП ефективніше та рентабельніше використовувати бібліотечну систему на основі детермінованих кінцевих автоматів.

По–друге, бібліотека є розширюваною та надає можливість персоналізувати певні функції для різних користувачів залежно від їх потреб.

По–третє, розроблена бібліотека може бути розгорнута на різних платформах незалежно від фреймворку.

ВИСНОВКИ

Результатом магістерської роботи стали математична модель [13] та реалізація бібліотеки штучного інтелекту на основі детермінованого скінченного автомату.

Розробка бібліотеки дозволила створити два окремих проекти, з метою тестування бібліотеки на реальних даних та демонстрації функціоналу розробленого програмного забезпечення. Перший проєкт надає приклад використання бібліотеки у ігровому рушії, а другий — на вбудованих системах.

Створена бібліотека була успішно протестована за допомогою ручного та автоматичного тестування.

Проведено дослідження процесу розробки ШІ за допомогою порівняльного аналізу розробленої бібліотеки та існуючого плагіну–аналогу. Виявлені наступні головні переваги розробленої бібліотеки:

— уніфікованість та універсальність;

— незалежність від рушія або платформи;

— незалежність від сторонніх інструментів або плагинів;

— відносно компактний розмір (лише 9 кілобайт).

Таким чином, мета кваліфікаційної роботи досягнута, оскільки розроблена бібліотека не поступається аналогам, натомість має менший розмір та більшу гнучкість.

Розроблена бібліотека штучного інтелекту має можливість подальшого розвитку шляхом додавання додаткових інструментів та функціоналу у створенні універсальних ШІ.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. I. Akharas, M. P. Hennessey, and E. J. Tornoe, ‘Simulation and visualization of dynamic systems in virtual reality using solidworks, matlab/simulink, and unity’, in *ASME International Mechanical Engineering Congress and Exposition, Proceedings (IMECE)*, 2020, vol. 7A–2020. doi: 10.1115/IMECE2020–23485.
2. A. Barczak and H. Woźniak, ‘Comparative Study on Game Engines’, *Studia Informatica*, no. 23, 2020, doi: 10.34739/si.2019.23.01.
3. T. M. Makhamatov, ‘Philosophy of Artificial Intelligence’, *Humanities and Social Sciences. Bulletin of the Financial University*, vol. 9, no. 4, 2019, doi: 10.26794/2226–7867–2019–9–4–52–56.
4. W. Westera *et al.*, ‘Artificial intelligence moving serious gaming: Presenting reusable game AI components’, *Educ Inf Technol (Dordr)*, vol. 25, no. 1, 2020, doi: 10.1007/s10639–019–09968–2.
5. B. Xia, X. Ye, and A. O. M. Abuassba, ‘Recent Research on AI in Games’, in *2020 International Wireless Communications and Mobile Computing, IWCMC 2020*, 2020. doi: 10.1109/IWCMC48107.2020.9148327.
6. D. Perez–Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, ‘General video game AI: A multitrack framework for evaluating agents, games, and content generation algorithms’, *IEEE Trans Games*, vol. 11, no. 3, 2019, doi: 10.1109/TG.2019.2901021.
7. E. Lee, Y. G. Kim, Y. D. Seo, K. Seol, and D. K. Baik, ‘RINGA: Design and verification of finite state machine for self–adaptive software at runtime’, *Inf Softw Technol*, vol. 93, 2018, doi: 10.1016/j.infsof.2017.09.008.
8. E. A. Laksono and A. Susanto, ‘Mathematics Education Game Using the Finite State Machine Method to Implement Virtual Reality in Game Platformer’, *Inform : Jurnal Ilmiah Bidang Teknologi Informasi dan Komunikasi*, vol. 5, no. 1, 2020, doi: 10.25139/inform.v5i1.1860.
9. G. Rathee, S. Garg, G. Kaddoum, D. N. K. Jayakody, J. Piran, and G. Muhammad, ‘A Trusted Social Network using Hypothetical Mathematical Model

and Decision–based Scheme', *IEEE Access*, 2020, doi: 10.1109/ACCESS.2020.3048077.

10. A. Iqbal, L. J. Gunn, M. Guo, M. Ali Babar, and D. Abbott, 'Game theoretical modelling of network/cybersecurity', *IEEE Access*, vol. 7, 2019, doi: 10.1109/ACCESS.2019.2948356.
11. Unity Technology, 'Unity 3D', *Unity Technology*. 2018.
12. J. Brookes, M. Warburton, M. Alghadier, M. Mon–Williams, and F. Mushtaq, 'Studying human behavior with virtual reality: The Unity Experiment Framework', *Behav Res Methods*, vol. 52, no. 2, 2020, doi: 10.3758/s13428–019–01242–0.
13. Г.П. Чуйко, О.В. Дворник, О.М. Яремчук. Математичне моделювання систем і процесів. Навчальний посібник. Миколаїв: Вид- во ЧДУ ім.. Петра Могили, 2015.-244 с., ISSN 078-336-340-0