DOI: https://doi.org/10.26642/ten-2023-2(92)-166-173
УДК 004.42

С.О. Терещук, асистент
І.В. Панаріна, к.т.н., доц.
Р.А. Вольський, асистент
Державний університет «Житомирська політехніка»

Побудова ігрового інтелекту за допомогою патерну State в ігровому рушії Unity

Комп'ютерні ігри на сьогодні увійшли в життя кожної людини. Вони займають велику частину в індустрії розваг. Індустрія ігор зростає та еволюціонує кожного дня завдяки новим технологіям, платформам та трендам. Наприклад кіберспорт став популярним способом проведення дозвілля. Трансляції ігор на платформах, таких як Twitch, YouTube набули великої популярності. Велика кількість кіберспортивних змагань включає участь комп'ютерно контрольованих гравців (ботів). Ігровий інтелект ботів впливає на рівень складності та стратегії проходження гри. Він може бути спроектований для ботів різних рівнів навичок, від початківців до дуже висококваліфікованих опонентів. Якщо ігровий інтелект забезпечує розумних противників, то гра стає більш викликаючою і цікавою. Гравці можуть відчувати вплив своїх рішень, оскільки бачать як ігровий інтелект ботів відповідає на їх дії. Прикладом може бути гра «Dark Souls», де кмітливі та нещадні вороги створюють надзвичайно викликаючий досвід гри. Також ігровий інтелект може контролювати творення сюжету та діалогів у грі. Розумні боти можуть створювати гравцям інтригуючі сцени та рішення, які впливають на розгортання сюжету. Приклад таких ботів можна побачити у грі «The Witcher 3: Wild Hunt», де ігровий інтелект впливає на розвиток сюжету. Метою роботи є дослідження використання ігрового інтелекту та його побудова, як бази для подальшої реалізації логіки ігрових об'єктів. Ігровий інтелект реалізовано на мові програмування C# з урахуванням особливостей ігрового рушія Unity та методів об'єктно-орієнтованого проектування. Він може використовуватись програмістами різних рівнів, та на різних стадіях розробки продукту залишатиметься актуальним та зручним у використанні.

Ключові слова: *Unity; Pattern State; C#; ігровий інтелект; комп'ютерні ігри.*

Актуальність теми. Ринок відеоігор завоював індустрію розваг і за останні роки обігнав традиційні ринки дозвілля. Приклад: The Last of Us, Warcraft, Pokemons, Uncharted, Lara Croft:Tomb Raider, отримали екранізацію в серіалах та фільмах. Великого успіху в індустрії розваг досягли мобільні ігри, що можна пояснити їх доступністю під час денної рутини та невеликими системними вимогами до гаджетів. Для їх створення не потрібні великі команди та колосальні бюджети, адже складні графічні зображення на екрані мобільного телефону неможливо оцінити. Запорука успіху – хороша механіка і цікавий сюжет. Саме використання патерну State дозволяє розділити складні механіки на окремі заміняємі стани, що в свою чергу знижує ризик появи неочікуваної поведінки та дозволяє легко міняти або додавати нові механіки. Застосування патерну може бути корисним як для розробки логіки гравця та противників, так і для різноманітних складових гри де відстежується зміна стану об'єкта чи системи.

Аналіз останніх досліджень та публікації, на які спирається автор. При побудові комп'ютерних ігор все частіше використовують методи як ігрового так і штучного інтелекту. Авторами статті [1] були розглянуті основні архітектурні підходи, які використовуються для створення ігрового штучного інтелекту. В роботі проаналізовано переваги, недоліки та обмеження таких архітектур, як Finite State Machine, Behavior Tree, Utility AI. Шестопалов С. В., Григорюк Д. К.[2] розглянули основні методи реалізації ігрового штучного інтелекту в іграх жанру RPG. Авторами зроблено висновок, що гарно спроектований ігровий інтелект дозволяє не ігровим персонажам вести себе більш реалістично, що робить більш цікавим геймплей та занурює гравця в гру. Штучний інтелект на сьогодні використовується в різних сферах. Наприклад авторами роботи [3] запропоновано програмну реалізацію з використанням методів штучного інтелекту для пошуку вільного місця на парковці. Розроблена система надає змогу користувачам авто здійснювати пошук вільних місць, витрачаючи при цьому мінімум часу. Марчук Д.К. та інші [4] описують процес застосування нейронних мереж для розпізнавання дактильної мови української абетки. В роботі використано особливий різновид архітектури рекурентних нейронних мереж, здатний до навчання, а саме модель довготривалої короткочасної пам'яті LSTM (Long short-term memory), який довів свою ефективність. У статті [5] описано основні алгоритми аналізу потоку кадрів відеоданих, шо надходять з камер міста. Основною метою дослідження є мінімізація часу на пошук вільного місця для паркування автомобіля. У статті [6] досліджуються алгоритми інтелектуального аналізу даних, які на основі правил і обчислень дозволяють створити модель, що аналізує дані, здійснюючи пошук певних закономірностей і тенденцій. Шляхом дослідження алгоритмів інтелектуального аналізу даних було розроблено моделі та методи для встановлення впливу одних хронічних захворювань на інші. Проведені дослідження свідчать про перспективність використання методів інтелектуального аналізу даних для підвищення якості медичної допомоги пацієнтам.

Finite-state machine (FSM) – це математична модель. Дані, які поступають на вхід, в деякий момент часу можуть знаходитися тільки в одному з кінцевого числа станів. Задачі, які можна вирішувати за допомогою кінцевих автоматів (КА) різні, наприклад в роботі [7] проведено дослідження присвячене розробці алгоритму аналізатора коду для системи дистанційного навчання мов програмування. Отриманий результат полягає у модифікації методу синтаксичного аналізу коду на основі ітеративного алгоритму. Також у роботі побудовано модель синтаксичного аналізатора з розподілом лексем. Дослідники та практики все ще намагаються знайти ефективні способи моделювання та тестування веб-додатків. У документі [8] пропонується техніка тестування на системному рівні, яка поєднує генерацію тестів на основі кінцевих автоматів з обмеженнями. Розробка тестових випадків є важливою проблемою для тестування програмного забезпечення, протоколів зв'язку та інших реактивних систем. Відомо ряд методів для розробки набору тестів на основі формальної специфікації, заданої у формі кінцевого автомата. У [9] розглядається та проводиться експеримент з цими методами, щоб оцінити їх складність, повноту, здатність до виявлення помилок, тривалість і час створення наборів тестів.

Метою дослідження є аналіз можливостей використання ігрового інтелекту при створенні ігор використовуючи рушій Unity. Для досягнення поставленої мети, необхідно побудувати ігровий інтелект на базі патерну State, враховуючи особливості ігрового рушія Unity. Ігровий інтелект (ІІ) має бути спроектований таким чином, щоб можна було легко та швидко додавати унікальні механіки та логіку, ігровим об'єктам які відрізняються один від одного. Побудований ІІ має стати міцним підґрунтям для починаючих розробників ігор, як на етапі проектування проекту, так і на етапі створення робочої версії гри.

Викладення основного матеріалу. Зазвичай розробники, які не знайомі з концепцією патерну State, вже в якійсь мірі його реалізовували за допомогою величезної кількості умовних операторів. Розглянувши псевдокод (рис. 1) можна побачити, що така реалізація має свої мінуси. По-перше всі умови перевіряються кожний кадр у методі Update, що призводить до втрати продуктивності.

```
public class Bot {

  void Update()
  {
    if(hp == 0)
    {
      animator.SetBool("Dead",true);
    }
    if (hp <= criticalHp && target != null && !undercover && !inWater)
    {
      TakeCover();
      undercover = true;
    }
    if(inWater)
    {
      Swim();
    }
    if(hp <= criticalHp && target != null && undercover && !inWater || hp > criticalHp && target != null && !inWater)
    {
     if(target !=null && distanceToTarget < distanceToAttack && ammo !=0 )
     {
       weapon.Fire();
       animator.SetBool("Fire",true);
     }
     if (target !=null && distanceToTarget < distanceToAttack && ammo == 0)
     {
        weapon.Reload();
        animator.SetBool("Reload",true);
     }

    }
    else if(!target && !inWater)
    {
      if(transform.position.Equals(patroolPoints[currentPatrolPoint]))
        ChangePatrolPoint();
      animator.SetBool("Walk",true);
      Move();
    }
  }
}
```

Рис. 1. Псевдокод

Навіть після того, коли бот сховався за укриттям, програма продовжує перевіряти цю умову. По-друге деякі з умов залежать від інших. Наприклад, бот буде прямувати до укриття тоді, коли в нього критичний запас здоров'я та він має супротивника. Або ж, як би до бота додали логіку плавання, то потрібно було би модифікувати попередні умови. Наприклад бот не може стріляти і перезаряджатись, коли він знаходиться у воді. В майбутньому такий проєкт буде обростати шарами непотрібної складності, що призведе до погіршення читаності коду, буде збільшуватись ймовірність допущення помилок, що в свою чергу призведе до появи неочікуваної логіки. Це все заважатиме розвитку та підтримці вашого проєкту.

Перед тим як розглядати патерн State, потрібно познайомитись з концепцією машини станів, також відомою як стейт-машина, яка лежить в основі реалізації ігрового інтелекту (рис. 2). Ідея цієї концепції полягає в тому, що ваша програма може знаходитись в момент часу тільки в одному зі станів, та їх кількість є скінченною. Реалізація станів, їх кількість та переходи між ними визначені заздалегідь розробником. Що ж собою представляє стан? Це може бути абсолютно будь що, що нам потрібно. Наприклад, це може бути фізичний або ментальний стан персонажу, пора року або час дня. Стан визначає поведінку об'єкта або системи. Як би у вашій стейт-машині був стан поранення, то він міг би впливати на характеристики вашого персонажу, що б зменшити швидкість бігу або силу удару. Для того щоб створити стейт-машину вам потрібно визначитись зі станами, які матиме система або об'єкт, та переходами між ними. Також потрібно обрати початковий стан, який буде запускатись при старті машини.

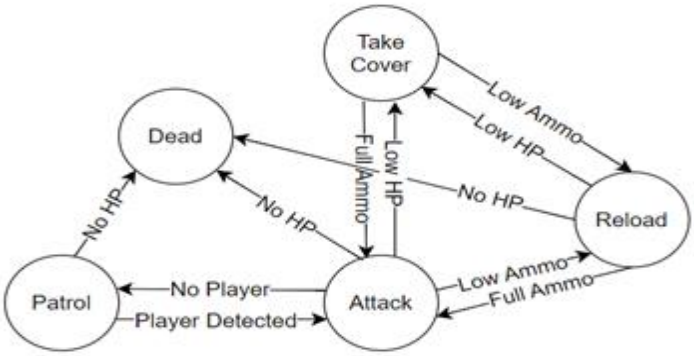

Рис. 2. Стейт-машина

Патерн State – це поведінковий патерн проектування, який використовується разом з об'єктно-орієнтованими мовами програмування. Основна ідея патерну полягає у тому, що кожен стан реалізований як окремий клас, і коли стан об'єкта змінюється то викликається відповідна реалізація класу. В класах станів знаходиться притаманна їм логіка, яка відповідає за специфічну поведінку об'єкта у стані. Всі умови для переходу між станами об'єкта, описуються у відповідних класах станів, що покращує читабельність та підтримуваність коду. Також в структурі патерну наявний керуючий клас, який зберігає посилання на поточний стан та всі інші притаманні об'єкту стани. Він буде делегувати роботу, яка відповідає за поведінку об'єкта у якомусь стані, відповідним класам станів. Цей патерн робить процеси керування станами простими у розумінні та реалізації. Окрім цього, він дозволяє легко та швидко додавати нові стани об'єкту, без потреби у модифікуванні існуючих станів.

Реалізація власної стейт-машини, розпочинається зі створення керуючого класу Context та абстрактного класу AbstractState, які є основою для патерну State (рис. 3). Context буде займатись створенням і керуванням екземплярами класів конкретних станів під час ігрового процесу. Також керуючий клас буде передавати актуальні, необхідні дані активному стану. Абстрактний клас буде слугувати шаблоном для реалізації конкретних станів.

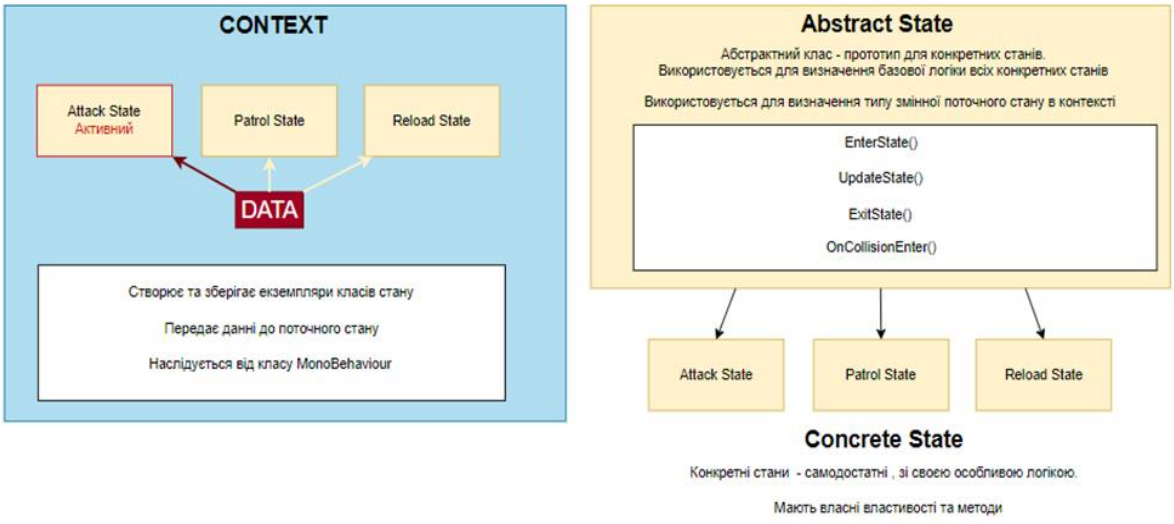

Рис. 3. Складові патерну State

Для того щоб, не втрачати у продуктивності, при проектуванні стейт-машини, наслідником MonoBehaviour є тільки один клас Context. Скрипти, які наслідуються від MonoBehaviour, можна використовувати як компоненти ігрового об'єкта. Вони мають методи Update, FixedUpdate, OnCollisionEnter, OnTriggerEnter та інші, які є частиною життєвого циклу Unity [10]. Саме завдяки цим методам, ви можете дізнатись як ваш об'єкт взаємодіє з навколишнім ігровим світом або оновлювати його поведінку. Усі скрипти MonoBehaviour знаходяться у списках оновлень Unity. Це значить що методи життєвого циклу викликатимуться в кожному кадрі, навіть якщо вони взагалі нічого не роблять. Коли кількість таких скриптів стає дуже великою, накладні витрати на виклик тисяч методів оновлення починають бути помітними. В такий момент може бути вже пізно змінювати архітектуру гри.

Далі створюємо абстрактний клас AbstractState, який буде використовуватись при реалізації конкретних станів. В цьому класі було створено чотири абстрактні методи: EnterState, UpdateState, ExitState та OnCollisionEnter. Ключове слово abstract біля модифікаторів доступу методів, дозволить перевизначити логіку для класів наслідників. Також вони мають бути публічними, щоб можна було їх викликати в керуючому класі. Окрім цього був реалізований конструктор, який приймає в параметрах екземпляр класу Context, що дозволить отримувати всі необхідні компоненти для роботи стану, за допомогою методу GetComponent().

```
using UnityEngine;
Ссылок: 12
public abstract class AbstractState
{
    protected Context context;

    Ссылок: 4
    public AbstractState(Context context)
    {
        this.context = context;
    }

    Ссылок: 6
    public abstract void EnterState();
    Ссылок: 5
    public abstract void UpdateState();
    Ссылок: 5
    public abstract void ExitState();
    Ссылок: 5
    public abstract void OnCollisionEnter(Collision collision);
}
```

Рис. 4. Реалізація шаблона стану

Для подальшого проектування системи були обрані стани, необхідні боту з псевдокоду. Створюємо класи AttackState, ReloadState, TakeCoverState, PatrolState, які наслідуються від класу AbstractState, та керуючий клас Context. Реалізовувати стани починаємо зі стану патрулювання, який запускатиметься одразу при старті нашої машини (рис. 5). В конструкторі класу ініціалізуємо NavMeshAgent та створюємо координати для патрулювання. За допомогою ключового слова base, отримуємо доступ до конструктора класу AbstractState, де відбувається ініціалізація захищеного поля в якому зберігається Context.

```
using UnityEngine;
using UnityEngine.AI;

Ссылок: 2
public class PatrolState : AbstractState
{
    private NavMeshAgent _agent;
    private Vector3[] _patrolPoints;
    private int _patrolPointIndex = 0;
    Ссылок: 1
    public PatrolState(Context context) : base(context)
    {
        _agent = context.GetComponent<NavMeshAgent>();
        _patrolPoints = new Vector3[2];
        Vector3 contextPosition = context.transform.position;
        _patrolPoints[0] = new Vector3(contextPosition.x, contextPosition.y, contextPosition.z + 10f);
        _patrolPoints[1] = new Vector3(contextPosition.x, contextPosition.y, contextPosition.z - 10f);
    }

    Ссылок: 3
    public override void EnterState()
    {
        _agent.SetDestination(_patrolPoints[_patrolPointIndex]);
    }

    Ссылок: 2
    public override void UpdateState()
    {
        if (_agent.remainingDistance == 0)
        {
            if (_patrolPointIndex == _patrolPoints.Length - 1)
                _patrolPointIndex = 0;
            else
                _patrolPointIndex++;
            _agent.SetDestination(_patrolPoints[_patrolPointIndex]);
        }
    }

    Ссылок: 2
    public override void ExitState()
    {
        _agent.isStopped = true;
    }

    Ссылок: 2
    public override void OnCollisionEnter(Collision collision)
    {
        if (collision.collider.tag.Equals("Target"))
            context.ChangeState(StatesNames.Attack);
    }
}
```

Рис. 5. Спроєктований стан патрулювання

Далі перевизначаємо методи класу AbstractState за допомогою ключового слова override, та наповнюємо їх кодом, який відповідає за патрулювання по координатам. При вході в цей стан, NavMeshAgent буде отримувати координату прибуття з масиву _patrolPoints. Далі в методі UpdateState буде перевірятись відстань до цієї точки, та у випадку завершення пересування вона буде мінятись на наступний елемент масиву _patrolPoints. У випадку виходу зі стану, пересування бота зупинятиметься. В методі OnCollisionEnter, буде перевірятись чи зіткнення відбулось з ціллю для атаки і в такому випадку відбудеться перехід до стану атаки.

За аналогією потрібно наповнити інші стани притаманною для них логікою. Наступним кроком буде реалізація класу Context та його методу ChangeState(), який використовується для переходу з одного стану в інший (рис. 6). Для того, щоб в цьому класі зберігати стани, використовується популярний тип колекції Dictionary. Ця колекція зберігає об'єкти, які представляють пару ключів-значення, де ключем буде перелічуваний тип даних enum, а значеннями будуть наслідки абстрактного класу AbstractState. При виклику метода Start(), створюються екземпляри класів, які є станами об’єкта, і додаються у словник _states.

StatesNames також буде параметром методу ChangeState(), для того щоб в ньому здійснити пошук по словнику за ключем. Перед тим як змінити стан, викликається його метод ExitState(). Далі поле _currentState, яке зберігає посилання на поточний стан, ініціалізується знайденим значенням зі словника, і викликається EnterState() для нового стану.

Щоб логіка стану оновлювалась кожен кадр і скрипти станів могли реагувати на колізію об’єктів, ми викликаємо методи станів UpdateState() та OnCollisionEnter() у відповідних методах життєвого циклу Unity.

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

Ссылок: 9
public enum StatesNames
{
    Attack,
    Reload,
    Patrol,
    TakeCover,
}

Скрипт Unity (1 ссылка на ресурсы) | Ссылок: 6
public class Context : MonoBehaviour
{
    private AbstractState _currentState;
    private Dictionary<StatesNames, AbstractState> _states;

    Сообщение Unity | Ссылок: 0
    private void Start()
    {
        _states = new Dictionary<StatesNames, AbstractState>();

        _states.Add(StatesNames.Attack, new AttackState(this));
        _states.Add(StatesNames.Reload, new ReloadState(this));
        _states.Add(StatesNames.Patrol, new PatrolState(this));
        _states.Add(StatesNames.TakeCover, new TakeCoverState(this));

        _currentState = _states.GetValueOrDefault(StatesNames.Patrol);
        _currentState.EnterState();
    }

    Сообщение Unity | Ссылок: 0
    private void Update()
    {
        _currentState.UpdateState();
    }

    Ссылок: 1
    public void ChangeState(StatesNames stateName)
    {
        _currentState.ExitState();
        _currentState = _states.GetValueOrDefault(stateName);
        _currentState.EnterState();
    }

    Сообщение Unity | Ссылок: 0
    private void OnCollisionEnter(Collision collision)
    {
        _currentState.OnCollisionEnter(collision);
    }
}
```

Рис. 6. Реалізація класу Context

Висновки та перспективи подальших досліджень. У ході роботи було проведено аналіз задачі, засобів та методів її вирішення, в результаті якого визначено необхідність використання патерну State при розробці ігрового інтелекту. Було розглянуто концепцію машини станів, проблематику та способи побудови ІІ в ігровому рушії Unity. Код було написано мовою програмування C#, з урахуванням особливостей ООП.

Результатом проведеного дослідження стало створення ігрового інтелекту в рушії Unity. ІІ спроектований таким чином, щоб можна було легко та швидко додавати унікальні механіки та логіку, ігровим об’єктам які відрізняються один від одного. Побудований ігровий інтелект є міцним підґрунтям для починаючих розробників ігор, як на етапі проектування проекту, так і на етапі створення робочої версії гри. Побудований ІІ дозволяє працювати над станами об’єкта, одночасно багатьом програмістам, незалежно один від одного, що значно прискорює процес розробки.

Список використаної літератури:

1. *Шестопалов С.В.* Ігровий штучний інтелект в іграх жанру RPG. Інформаційні технології і автоматизація – 2020 : зб. доп. XIII Міжнар. наук.-практ. конф. / *С.В. Шестопалов, Д.К. Григорюк* // Одес. нац. акад. харч. технологій, Інститут комп’ютерних систем і технологій «Індустрія 4.0» ім. П.М. Платонова. – Одеса, 2020. – С. 300–303.
2. *Кириченко. І.* Підходи розробки ігрового штучного інтелекту / *І.Кириченко, В.Рошка* // InterConf. – Вип. 96. – 2022 [Електронний ресурс]. – Режим доступу : https://ojs.ukrlogos.in.ua/index.php/interconf/article/view/18309.
3. *Левківський В.Л.* Комп’ютерна програма «Алгоритмічно-програмне забезпечення обробки та аналізу потоку кадрів відеоданих, що надходять з камер міста» / *В.Л. Левківський, Г.В. Марчук, В.В. Ципоренко, Д.К. Марчук*. – 2021 [Електронний ресурс]. – Режим доступу : http://eztuir.ztu.edu.ua/bitstream/handle/123456789/8019/109822.pdf?sequence=1&isAllowed=y
4. *Марчук Д.К.* Система розпізнавання дактильної мови української абетки / *Д.К. Марчук, В.Л. Левківський, Г.В. Марчук, М.Ю. Голенко* // Вчені записки Таврійського національного університету імені

В.І. Вернадського. Серія: Технічні науки. – 2022. – Т. 33 (72), № 6. – С. 109–114 [Електронний ресурс]. – Режим доступу : https://doi.org/10.32782/2663-5941/2022.6/19
5. Available parking places recognition system / *V.Levkivskyi, D.Marchuk, N.Lobanchykova, I.Pilkevych, D.Salamatov* // CEUR Workshop Proceedings 4th Workshop for Young Scientists in Computer Science & Software Engineering. – Volume 3077 (2022). – pp.123–134 [Електронний ресурс]. – Режим доступу : http://ceur-ws.org/Vol-3077/paper07.pdf
6. *Levkivskyi V.* Research of algorithms of Data Mining / *V.Levkivskyi, N.Lobanchykova, D.Marchuk* // The International Conference on Sustainable Futures: Environmental, Technological, Social and Economic Matters. – Volume 166, 05007. – 2020. – pp.1–6 [Електронний ресурс]. – Режим доступу : https://doi.org/10.1051/e3sconf/202016605007.
7. *Сугоняк І.І.* Синтаксичний аналіз коду для системи дистанційного навчання Програмування на мові C#. / *І.І. Сугоняк, Г.В. Марчук, С.О. Бобровнік* // Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки. – Т. 29 (68). – № 5. – 2018. – С. 65–72.
8. *Andrews A.A.*Testing web applications by modeling with FSMs / *A.A. Andrews, J.Offutt, R.T. Alexander* // Software & Systems Modeling. – 2005. – № 4. – pp. 326–345.
9. FSM-based conformance testing methods: a survey annotated with experimental evaluation / *R.Dorofeeva, K. El-Fakih, S.Maag, A.R. Cavalli, N.Yevtushenk*o // Information and Software Technology. – 2010. – № 52(12). – pp. 1286–1297.
10. Order of execution for event functions. – 2023 [Електронний ресурс]. – Режим доступу : https://docs.unity3d.com/Manual/ExecutionOrder.html

References:

1. Shestopalov, S.V. & Hryhoriuk, D.K. (2020), «Ihrovyi shtuchnyi intelekt v ihrakh zhanru RPG», *Informatsiini tekhnolohii i avtomatyzatsiia – 2020: zb. dop. XIII Mizhnar. nauk.-prakt. konf.*, Odesa, Instytut kompiuternykh system i tekhnolohii «Industriia 4.0» im. P.M. Platonova, 2020, pp. 300–303.
2. Kyrychenko, I. & Roshka, V. (2022), «Pidkhody rozrobky ihrovoho shtuchnoho intelektu», *InterConf*, Vyp. 96, [Online], available at: https://ojs.ukrlogos.in.ua/index.php/interconf/article/view/18309
3. Levkivskyi, V.L., Marchuk, G.V., Cyporenko, V.V. & Marchuk, D.K. (2021), «Kompiuterna programa Algorytmichno-programne zabezpechennja obrobky ta analizu potoku kadriv videodanyh, shho nadkhodiat z kamer mista», [Online], available at: http://eztuir.ztu.edu.ua/bitstream/handle/123456789/8019/109822.pdf?sequence=1&isAllowed=y
4. Marchuk, D.K., Levkivskyi, V.L., Marchuk, G.V. & Holenko, M.Yu. (2022), «Systema rozpiznavannia daktylnoi movy ukrainskoi abetky», *Vcheni zapysky Tavriiskoho natsionalnoho universytetu imeni V.I. Vernadskoho. Seriia: Tekhnichni nauky*, T. 33 (72), No 6, pp. 109–114, [Online], available at: https://doi.org/10.32782/2663-5941/2022.6/19
5. Levkivskyi, V., Marchuk, D., Lobanchykova, N., Pilkevych, I. & Salamatov, D. (2022), «Available parking places recognition system», *CEUR Workshop Proceedings 4th Workshop for Young Scientists in Computer Science & Software Engineering*, Vol. 3077, pp.123–134, [Online], available at: http://ceur-ws.org/Vol-3077/paper07.pdf
6. Levkivskyi, V., Lobanchykova, N. & Marchuk, D. (2020), «Research of algorithms of Data Mining», *E3S Web of Conferences*, Vol. 166, The International Conference on Sustainable Futures: Environmental, Technological, Social and Economic Matters, pp.1–6, [Online], available at: https://doi.org/10.1051/e3sconf/202016605007
7. Suhoniak, I.I., Marchuk, G.V. & Bobrovnik, S.O. (2018), «Syntaksychnyi analiz kodu dlia systemy dystantsiinoho navchannia Prohramuvannia na movi C#», *Vcheni zapysky Tavriiskoho natsionalnoho universytetu imeni V.I. Vernadskoho, Seriia: Tekhnichni nauky*, T. 29 (68), No 5, pp. 65–72.
8. Andrews, A.A., Offutt, J. & Alexander, R.T. (2005), «Testing web applications by modeling with FSMs», *Software & Systems Modeling*, No 4, pp. 32–345.
9. Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A.R. & Yevtushenko, N. (2010), «FSM-based conformance testing methods: a survey annotated with experimental evaluation», *Information and Software Technology*, No 52(12), pp. 1286–1297.
10. Order of execution for event functions (2023), [Online], available at: https://docs.unity3d.com/Manual/ExecutionOrder.html

Терещук Степан Олександрович – асистент кафедри комп'ютерних наук, Державний університет «Житомирська політехніка».

https://orcid.org/0009-0000-5680-8445.

Наукові інтереси:

– розробка ігор за допомогою рушія Unity.

Панаріна Ірина Володимирівна – кандидат технічних наук, доцент кафедри комп'ютерних наук, Державний університет «Житомирська політехніка».

https://orcid.org/0000-0003-4783-2587.

Наукові інтереси:

– інформаційні системи та технології.

Вольський Ростислав Аркадійович – асистент кафедри комп'ютерних наук, Державний університет «Житомирська політехніка».

https://orcid.org/0009-0009-1336-9492.

Наукові інтереси:

– інформаційні системи та технології.

Tereshchuk S.O., Panarina I.V., Volskyi R.A.

Building game intelligence with the State pattern in the Unity game engine

Computer games have become a part of everyone's life today. They occupy a large part of the entertainment industry. The gaming industry is growing and evolving every day thanks to new technologies, platforms, and trends. For example, esports has become a popular way to spend leisure time. Game broadcasts on platforms such as Twitch and YouTube have become very popular. A large number of esports competitions involve computer-controlled players (bots). The game intelligence of bots affects the level of complexity and strategies of the game. It can be designed for bots of different skill levels, from beginners to very highly skilled opponents. If game intelligence provides smart opponents, the game becomes more challenging and interesting. Players can feel the impact of their decisions as they see how the game intelligence of bots responds to their actions. An example is the game «Dark Souls», where intelligent and merciless enemies create an extremely challenging gaming experience. Game intelligence can also control the creation of the game's plot and dialogues. Intelligent bots can create intriguing scenes and decisions for players that affect the plot. An example of such bots can be seen in the game The Witcher 3: Wild Hunt, where game intelligence influences the plot development. The purpose of the paper is to study the use of game intelligence and its construction as a basis for further implementation of the logic of game objects. Game intelligence is implemented in the C# programming language, taking into account the features of the Unity game engine and object-oriented design methods. It can be used by programmers of different levels and will remain relevant and easy to use at different stages of product development.

Keywords: Unity; Pattern State; C#; game intelligence; computer games.

Стаття надійшла до редакції 15.10.2023.