УДК 004.584

Блажко Олександр Анатолійович
Кандидат технічник наук, доцент кафедри системного програмного забезпечення, *orcid.org0000-0001-7413-275X*
Одеський національний політехнічний університет, Одеса
Антонюк Віктор Вікторович
Старший викладач кафедри інтелектуальних комп'ютерних систем і мереж, *orcid.org0000-0001-8436-5338*
Одеський національний політехнічний університет, Одеса
Трояновська Юлія Людвигівна
Старший викладач кафедри інформаційних систем, *orcid.org0000-0002-6716-9391*
Одеський національний політехнічний університет, Одеса

ІНСТРУМЕНТАЛЬНІ ОСОБЛИВОСТІ АВТОМАТНОГО ПРОГРАМУВАННЯ КОМП'ЮТЕРНИХ ІГОР

Анотація. *Підвищення інтересу до гейміфікації бізнес-процесів зобов`язує розробників формалізувати технологічні процеси з використанням Unified Modeling Language (UML) на основі UML-діаграм. Парадигма автоматного програмування для розробки ПЗ на основі кінцевих автоматів спрощує аналіз створеного програмного коду та його тестування. Розглянуто особливості автоматного програмування комп'ютерних ігор при переході від UML-моделювання до реалізації кінцевого автомата сценарію гри, враховуючи форми візуального представлення автоматів плагінами сучасних рушіїв. Результати експериментів із переведення UML-діаграми станів у модель машини станів виявили основі особливості, які можуть ускладнити процес переводу та підвищити ймовірність помилок реалізації. У подальшому на основі виявлених особливостей буде розроблено алгоритм автоматизованого синтезу покрокових рекомендацій з реалізації UML-діаграми станів для різних плагінів.*

Ключові слова: гейміфікація; автоматне програмування; UML; Unity3D; GameHub

Вступ

Впродовж багатьох років основними відмінностями життєвого циклу багатьох комп'ютерних ігор від програмного забезпечення (ПЗ) будь-яких виробничих процесів була відсутність стандартів на структуру вимог до ПЗ, які представляються у формі *Game Design Document* [1], та короткий час експлуатації гри користувачами. Але підвищення інтересу до гейміфікації бізнес-процесів за останні 10 років почало виводити розробку комп'ютерних ігор з вузькоспеціалізованого процесу розроблення ПЗ до індустрії розроблення на основі класичних інструментальних середовищ мов програмування *Java*, *C++, C#* або спеціалізованих інструментальних середовищ (ігрових рушіїв), таких як *Unity3D* та *Unreal Engine*. Це зобов`язує розробників з метою автоматизації активно формалізувати технологічні процеси з використанням *Unified Modeling Language* (*UML*) як уніфікованої універсальної мови моделювання на основі *UML*-діаграм використання ПЗ, концептуальних класів об'єктів, станів об'єктів класів та інших [2]. Однак вже понад 30 років існує парадигма автоматного програмування або «програмування від станів» для розроблення ПЗ на основі кінцевих автоматів [3]. Перевагами графічного представлення поведінки ПЗ у вигляді моделі кінцевого автомата над алгоритмічним програмуванням є спрощення аналізу створеного програмного коду та автоматизація тестування. Але в роботі [4] зазначено, що методи використання діаграм станів кінцевих автоматів або мереж Петрі показали ефективність лише у спеціалізованих сферах, однак у сфері проектування комп'ютерних ігор розробники стикаються з тим, що кількість ігрових станів зазвичай не є кінцевою і має швидко зростаючу складність при підвищенні деталізації моделі гри. Це обмежує класи ігор, які піддаються автоматному програмуванню, до рівня казуальних ігор з простими *GamePlay*-правилами та графікою [5; 6]. В цьому випадку для розширення класів ігор необхідна автоматизація процесу керування станами автоматної моделі гри за наявності відповідних інструментів в ігрових рушіях.

В ігровому рушії *Unity3D* розробникам надається можливість використовувати автоматне програмування у декількох сторонніх програмних компонентах (плагінах), найпоширенішими з яких є плагіни *PlayMaker* (*PM*-плагін) [7] та *Behaviour Machine* (*BM*-плагін) [8]. Але в більшості посібників зі створення ігор за вказаними плагінами відсутні рекомендації з використання формалізації автоматного

програмування. Хоча подібні методики є в роботі [9], яка пропонує процес формалізованого перетворення сценарію гри у програмний код, а в роботі [10] представлено процес проектування гри через поступове створення діаграми класів, діаграми станів та пов`язаних з кожним станом кадрів сценарію гри. Але їх треба адаптувати з урахуванням особливостей роботи конкретних плагінів.

Мета статті

Мета – визначити особливості автоматного програмування комп'ютерних ігор при переході від *UML*-моделювання до реалізації кінцевого автомата сценарію гри, враховуючи форми візуального представлення автоматів плагінами сучасних рушіїв.

Для досягнення вказаної мети автоматну модель необхідно розглядами як останній етап в технологічному процесі розроблення казуальної комп'ютерної гри.

Виклад основного матеріалу

Етапи технологічного процесу створення автоматної моделі

При проектуванні ігрової механіки вхідними даними є, зазвичай, сценарій гри, поданий як довільний текстовий опис. В результаті проектування необхідно отримати кінцевий автомат, який можна використовувати як шаблон при візуальному програмуванні гри в ігрових рушіях. Пропонується представити технологічний процес створення автоматної моделі ігрового рушія Unity 3D як послідовність етапів.

Eman 1. Формалізація сценарію гри: представлення вхідного текстового сценарію гри у вигляді сценарію використання та *Use-Case*-діаграми, тобто як маркованого списку, який покроково описує *GamePlay*-правила. При створенні *Use-Case*-діаграми та визначенні імен акторів враховується класифікація за типом візуалізації гравця від першої особи (*first-person shooter*, *FPS*) або від третьої особи (*third-person shooter, TPS*) [11].

Eman 2. Проведення аналізу сценарію використання гри: виділення ігрових об'єктів. Зазвичай у сценарії вони виступають як іменники, які мають характеристики (прикметники та іменники) та над ними виконують дії (дієслова). Ігрові об'єкти треба розділити на динамічні, від яких залежить *GamePlay*-правила, та статичні.

Eman 3. Визначення структури взаємодії ігрових об'єктів: побудова діаграми класів, на якій показані класи динамічних ігрових об'єктів та статичних, з якими вони взаємодіють. Атрибутами класів виступають характеристики ігрових об'єктів, операціями – дії, які призводять до зміни значень їх атрибутів з дозволеної множини. Класи задають шаблон ігрового об'єкта.

Eman 4. Проведення аналізу семантики взаємозв'язків між класами з метою визначення шаблонів механіки *GamePlay*-правил [12] та епістемологічні шаблони (*Epistemic Patterns*) [11], при цьому семантичні назви зв'язків між класами замінюються на шаблонні.

Eman 5. Визначення станів ігрових об'єктів: на основі розробленої діаграми класів та формалізованого ігрового сценарію визначаються стани об'єктів ігрових класів як деякі значення їх атрибутів з дозволеної множини, тригери, які призводять до зміни станів (дія користувача чи іншого ігрового об'єкта) та виконанню необхідних операції. Як результат будується кінцевий автомат у форматі *UML*-діаграми станів.

Eman 6. Визначення шаблонів людино-машинної взаємодії (*Human Computer Interface*, *HCI*) та значень *HCI*-подій [13] для тригерів подій *UML*-діаграми станів об'єктів ігрових класів. Наприклад, для *HCI*-типу «Стандартна клавіатура» використовуються клавіші «стрілка вгору», «стрілка вниз і інші, а для *HCI*-типу «2/3-кнопковий маніпулятор миша» використовується« натискання лівої кнопки миші», «натискання правої кнопки миші» та інші.

Eman 7. Перетворення діаграми станів у автоматну модель відповідного плагіну (*PM*-плагіну, *BM*-плагіну) з урахуванням таблиці відповідності між назвами тригерів подій з *UML*-діаграми станів та назвами тригерних подій у відповідних плагінах.

Автоматне програмування на прикладі гри з простими правилами

Для визначення особливостей *UML*-проектування розглянуто приклад гри «Змійка» з такими правилами сценарію:

1) змійка рухається у пошуках яблука по поверхні, яку обмежено по периметру стіною;

2) змійка може рухатися вперед, повертати праворуч або ліворуч;

3) якщо змійка досягає яблука, вона його з'їдає;

4) якщо змійка не встигає повернути, вона вдаряється у стіну.

На рис. 1 схематично показано об'єкти гри.

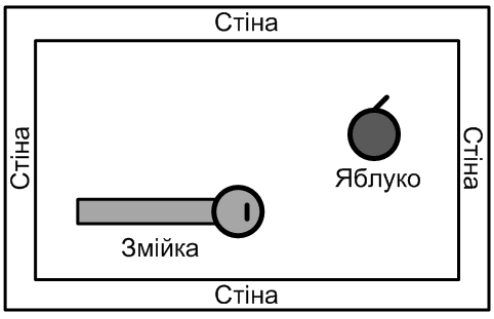


Рисунок 1 – Схематичне представлення об'єктів гри «змійка»

Вказані правила є спрощеними і алгоритм переміщення довгого тіла змійки вони не враховують, змійка має фіксований розмір і не зростає при з'їданні яблука.

Перший етап процесу проектування завершується створенням *UML*-діаграми прецедентів та маркованого списку правил сценарію. На рис. 2 показано *UML*-діаграму прецедентів з візуалізацією гравця *FPS*-типу, коли актором стає персонаж змійки.

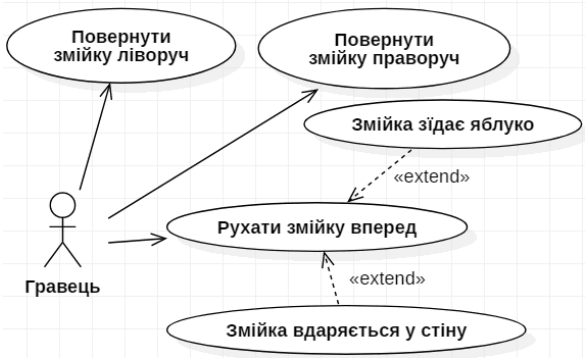


Рисунок 2 – UML-діаграми прецедентів гри «змійка»

Діаграма концептуальних класів (*Concept-class*) моделює статику системи з урахуванням деталей структур класів і взаємин між класами. *UseCase*-діаграма не враховує деталі структур та динаміку. Така мова моделювання найбільш придатна для створення ігрового світу: ієрархії сил та героїв, взаємовідносини між елементами гри (транзитивність та нетранзитивність гри), створення ігрових кланів (родин, груп, ігрових ролей) тощо. Такі діаграми дають можливість скоротити час на пошук подібних та споріднених об'єктів або об'єктів за певними ознаками (атрибутами) [14]. На рис. 3 наведено приклад діаграми концептуальних класів гри «Змійка».

Тож діаграма концептуальних класів дає відповідь на питання самовизначення ігрових персонажів, самих гравців та окремо кожного ігрового об'єкта в ігровому світі, в той час як діаграма варіантів використання моделює динаміку системи на глобальному рівні її взаємин з користувачем.

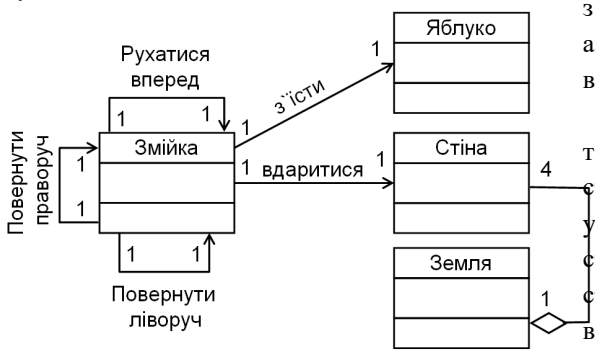


Рисунок 3 – Приклад UML-діаграми концептуальних класів гри «змійка»

Для більш детального моделювання поведінки активних ігрових об'єктів, тобто таких, від яких залежать *GamePlay*-правила, можливе використання *UML*-діаграм станів. Вони будуються на основі теорії кінцевих автоматів і показують динаміку об'єкта як зміну його станів у процесі життєвого циклу.

Діаграма станів показує життєвий цикл об'єкта як автомат – послідовність станів, через які він проходить, реагуючи на події. Стан – це частина життя об'єкта, в якому він виконує якусь діяльність, задовольняючи деяким умовам та очікуючи деякі події. У нотації діаграм станів передбачено два особливих стани – початковий та кінцевий.

В загальному випадку діяльність може бути як складною, тобто складатися з деякого набору операцій, що виконує об'єкт, так і атомарною – з однієї операції. Така діяльність називається дія. В діаграмах станів виокремлюють декілька типів діяльності. Найбільш використовувані це вхідна діяльність (*entry*) – виконується при вході в стан, вихідна (*exit*) – виконується при виході зі стану, та внутрішня (*do*) – виконується, коли об'єкт знаходиться в стані.

На рис. 4 показано *UML*-діаграму станів об'єкта «Змійка».

Зміна одного стану об'єкта на інший виконується за допомогою переходу, який показує, що коли відбудеться відповідна подія та будуть виконані відповідні умови, тоді об'єкт перейде з поточного стану в інший.

Подія це те, що активує перехід, а граничні умови визначають, чи може відбутися перехід, чи ні. Також при переході можуть виконуватися певні дії, які вказуються після умов.

Як видно з рис. 4, при старті гри об'єкт «змійка» потрапляє в стан «Очікування», де чекає на події управління від гравця. При активації гравцем подій повороту вправо, вліво та руху, змійка переходить в стани «Ліворуч», «Праворуч» та «Рух вперед» відповідно.

В кожному з цих станів виконується діяльність з виконання відповідного переміщення. Коли гравець активує подію зупинити переміщення, змійка вертається в стан «Очікування».

Знаходячись в стані «Рух вперед», змійка може також зіткнутися з іншими об'єктами – яблуком та стіною. Це показано переходами з граничними умовами [Зіткнення з яблуком], [Зіткнення зі стіною]. При першому переході змійка переходить в стан «Зіткнення з яблуком», де відбувається відповідна діяльність. При другому переході життєвий цикл об'єкта закінчується та відбувається вихід з гри.

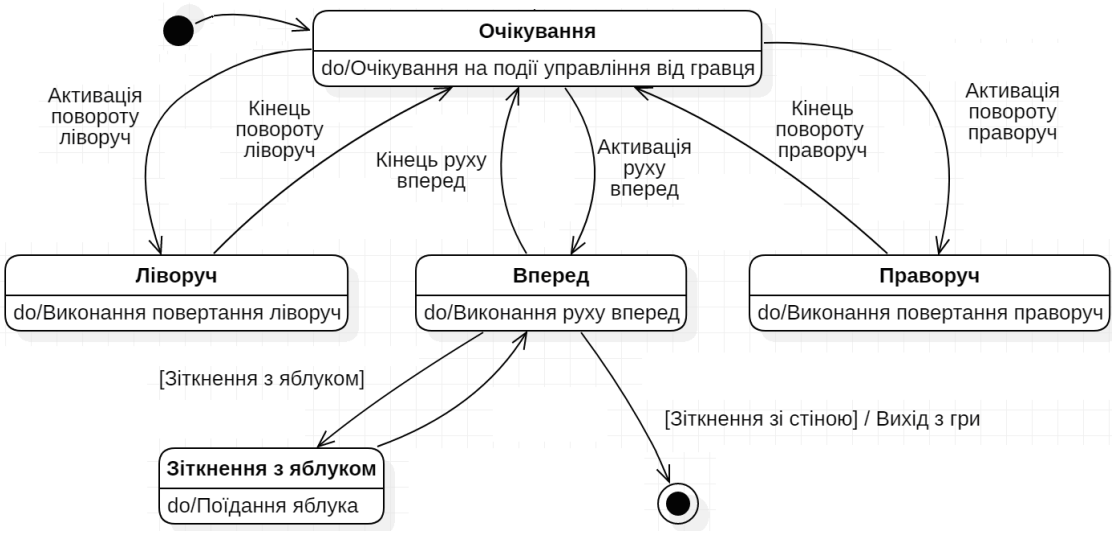


Рисунок 4 – UML-діаграми станів об'єкта «змійка»

Наприклад, в наведеній на рис. 4 діаграмі станів при находженні змійки в стан «Рух вперед» відбуваються переходи, які активуються граничними умовами зіткнення чи з яблуком, чи зі стіною.

Особливості автоматного програмування з *BM*-плагіном

В *Unity 3D* для відслідковування зіткнень поміж ігровими об'єктами використовується подія *OnCollisionEnter*, яка відбувається при зіткненні вибраного об'єкта з будь-яким іншим. Тобто реалізувати діаграму станів на рис.4 в такому вигляді за допомогою *BM*-компоненти неможливо, адже неможливо в момент зіткнення виявити, з яким саме об'єктом зіткнулася змійка.

Тому для *BM*-плагіна необхідно модифікувати *UML*-діаграму станів на рис. 5, додавши замість стану «Зіткнення з яблуком» стан «Обробка зіткнення», в який змійка переходить при події «Зіткнення з ігровим об'єктом». В цьому стані відбувається діяльність з визначення типу об'єкта, з яким зіткнулася змійка, та виконуються переходи за відповідними граничними умовами. Модифікована *UML*-діаграма станів у вигляді машини станів *BM*-плагіну наведена на рис. 6.

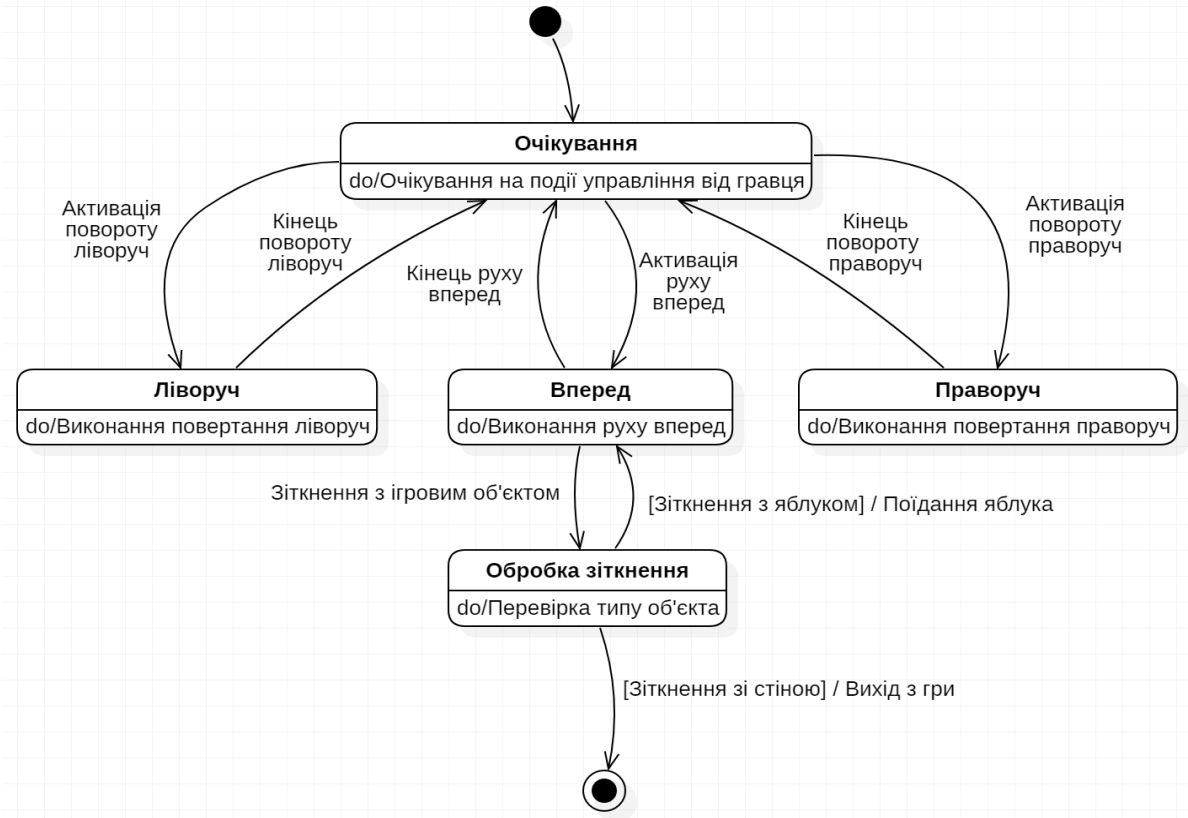


Рисунок 5 – Модифікована UML-діаграми станів об'єкта «змійка»

В цій машині станів стан *WaitingInput* відповідає стану «Очікування» модифікованої діаграми станів, *TurnLeft*, *TurnRight*, *MovingForward* та *CollisionProcessing* відповідно «Ліворуч», «Праворуч», «Рух вперед» та «Обробка зіткнення».

Окремо початкового та кінцевого станів у *BM*-плагіні не вирізняють, як початковий вважається перший у послідовності станів *WaitingInput*, що позначається помаранчевим кольором. Назви події в *BM*-плагіні підписуються у початковому стані переходу. При використанні подій можна створювати події як для кожного ігрового об'єкта, так і використовувати системні події рушія.

На рис. 6 події *Forward*, *Left*, *Right*, *StopForward*, *StopLeft*, *StopRight* відповідають подіям відповідних переходів між станами діаграми на рис. 5.

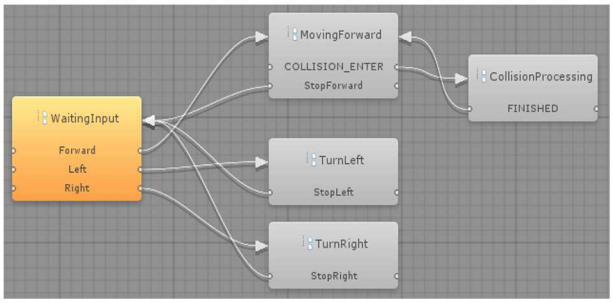


Рисунок 6 – Приклад машини станів об'єкта «змійка» в BM-плагіні

Подія *COLLISION_ENTER* є системною та відповідає події «Зіткнення з ігровим об'єктом» *UML*-діаграми на рис. 5.

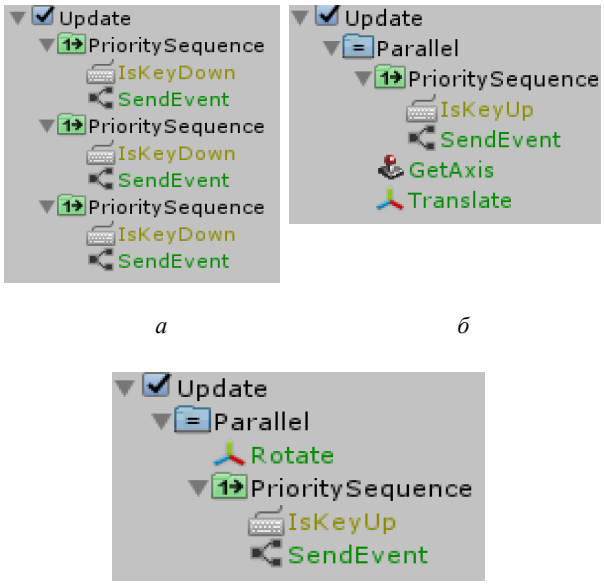
Аналогів граничних умов у *BM*-плагіні немає, тому всі перевірки умов переходів виконуються в самому стані і відповідно до них формуються події переходів.

Так, перевірка умов [Зіткнення з яблуком] та [Зіткнення зі стіною] відбувається в стані *CollisionProcessing*, причому при завершенні діяльності в стані формується системна подія *FINISHED*, по якій і виконується перехід.

Задля завдання складної діяльності в *BM*-плагіні використовується спеціальний тип станів – поведінкові дерева. Вони складаються з ієрархії вузлів, що розділяються на гілки, які організують порядок діяльності, та листя-вузли, що виконують елементарні операції.

На рис. 7 показано поведінкові дерева для станів *WaitingInput*, *MovingForward* та *TurnLeft*.

У стані *WaitingInput* коренем дерева є системна функція *Update*, що виконується кожний фрейм. В ній розміщені три гілки *PrioritySequence*, що виконують свої листя за порядком слідування, але при кожному фреймі обов'язково перевіряючи перший лист, яким у всіх трьох гілках є умовний вузел *IsKeyDown*, що очікує на натискання заданої клавіші. Якщо вона натиснута, вузол повертає істину і виконується наступний вузол *SendEvent*, що формує користувацьку подію. Отже, кожна з трьох гілок *PrioritySequence* чекає на натискання гравцем клавіш руху вперед або повороту наліво чи право, тоді формуючи відповідну подію *Forward*, *Left* чи *Right*. У стані *MovingForward* коренем також є функція *Update*, в якій розміщена гілка *Parallel*, що кожен фрейм одночасно виконує всі свої вузли.



а *б*

в

Рисунок 7 – Приклад поведінкових дерев в BM-плагіні для станів WaitingInput (а), MovingForward (б), TurnLeft (в)

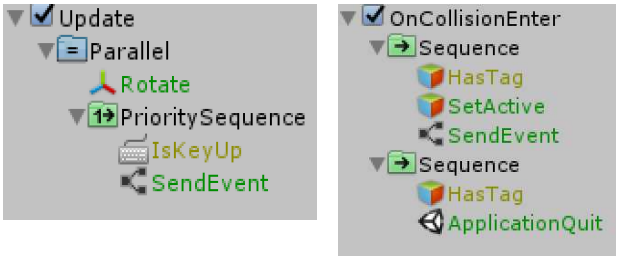
Першим з них є гілка *PrioritySequence*, в які умовний вузол *IsKeyUp* чекає на відпускання клавіші руху вперед гравцем, тоді формуючи подію *StopForward* у вузлі *SendEvent*. Другим та третім листями є *GetAxis*, що отримує координати переміщення по заданих осях координат, та *Translate*, що переміщує ігровий об'єкт.

У стані *TurnLeft* корень – функція *Update*, що має гілку *Parallel*. В ній є лист *Rotate*, який виконує поворот ігрового об`єкта ліворуч, та гілка *PrioritySequence,* що виконує такі ж дії, що і аналогична гілка в стані *MovingForward*, але для відслідковування відпускання клавіши поворота наліво та формування події *StopLeft*.

На рис. 8 наведено поведінкові дерева для станів *TurnRight* та *CollisionProcessing*.

Набір вузлів стану *TurnRight* аналогічний до стану *TurnLeft*, тільки вузол *Rotate* виконує поворот праворуч, *IsKeyUp* відслідковує відпускання клавіши повороту праворуч, а *SendEvent* формує подію *StopRight*.

Стан *CollisionProcessing* за корінь має функцію *OnCollisionEnter*, що складається з двох гілок *Sequence*, що виконують по черзі свої вузли кожен фрейм.



а *б*

Рисунок 8 – Приклад поведінкових дерев у BM-плагіні для станів TurnRight (а), CollisionProcessing (б)

Перша гілка *Sequence* виконує перевирку чи є об`єкт, з яким зіткнулася змійка, яблуком. Це перевіряє умовний вузол *HasTag*. Якщо це яблуко, він вертає істину й виконуються наступні вузли гілки *SetActive*, який робить об`єкт яблуко неактивним на ігровій сцені, та вузол *SendEvent*, що формує системну подію *FINISHED*.

Друга гілка *Sequence* виконує перевірку зіткнення зі стіною. Це також перевіряє вузол *HasTag*. Якщо це дійсно стіна, то наступний нод *ApplicationQuit* виконує вихід з гри.

Особливості автоматного програмування з *PM*-плагіном

На відміну від *BM*-плагіна використання *PM*-плагіна дало змогу перенести схему кінцевих автоматів, одержану при *UML*-проектуванні на рис. 4, до інтегрованого середовища розробки майже без модифікування, як це видно на рис. 9.

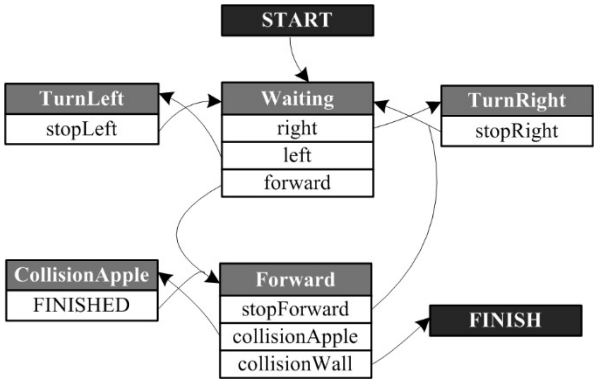


Рисунок 9 – Приклад машини станів об'єкта «змійка» в PM-плагіні

В цій машині станів стан *WaitingInput* відповідає стану «Очікування» *UML*-діаграми станів, *TurnLeft*, *TurnRight*, *MovingForward* та *CollisionApple* відповідно «Ліворуч», «Праворуч», «Рух вперед» та «Зіткнення з яблуком». Також машина станів має початковий та кінцевий стани *START* та *FINISH*.

Переходи між станами в *PM*-плагіні визначені у самому стані. При використанні подій можна створювати події як для кожного ігрового об'єкта, так і використовувати системні події рушія. На рис. 9 події *forward*, *left*, *right*, *stopForward*, *stopLeft*, *stopRight*, *collisionApple*, *collisionWall* відповідають подіям відповідних переходів між станами діаграми на рис. 4. *START* є обов'язковим початковим системним станом. *FINISH* – кінцевий стан, створений користувачем для обробки зіткнення зі стіною. Подія *FINISHED* є системною і визначає безумовний перехід між станами «Зіткнення з яблуком» та «Очікування» *UML*-діаграми станів на рис. 4, який виникає, коли всі дії всередині стану закінчено. Аналогів граничних умов у *PM*-плагіні немає, тому всі перевірки умов переходів виконуються в самому стані, і відповідно до них формуються події переходів.

У стані *WaitingInput* очікується натискання заданої клавіші. Додамо аспект *HCI*, який налаштовується в розділі *InputManager Unity3D*. В цьому прикладі використовується стандартний набір клавіш клавіатури для переміщення гравця *WASD* та стрілки курсора, як показано на рис. 10:

'*W*' або '↑' – реалізує перехід *forward* до стану *MovingForward*;

'*A*' або '←' – реалізує переход *left* до стану *TurnLeft;*

'*D*' або '→' – реалізує переход *right* до стану *TurnRight*;

'*S*' або '↓' – в цій машині станів не використовуються.

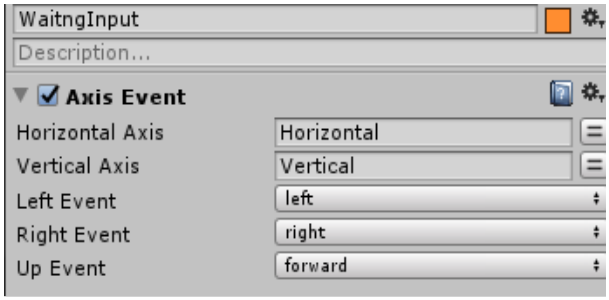


Рисунок 10 – Приклад реалізації стану WaitingInput

У стані *MovingForward* *Get Axis Vector* отримує координати переміщення за заданими вісями координат, *Translate* переміщує ігровий об`єкт. В цьому же стані події *Collision Event* формують переходи до станів *CollisionApple* та *FINISH* відповідно *collisionApple* та *collisionWall*, відслідковуючи зіткнення за тегом *apple* чи *wall*. При відпусканні клавіші руху подія *Axis Event* формує подію *StopForward*, яка повертає машину станів у стан *WaitingInput*. На прикладі цього стану можна зробити більш детальне проектування, використовуючи *UML*-діаграми, після того, як окреслена *HCI* та стратегія реалізації. Стратегія реалізації, наприклад, враховує чи потрібно у середовищі *Unity* для реалізації руху використання механізму *Character Controller*.

На рис. 11 предствлено приклад реалізації стану *MovingForward*.

Рисунок 11 – Приклад реалізації стану MovingForward

На рис. 12 наведено приклад реалізації стану *TurnLeft*. В стані *TurnLeft* подія *Rotate* виконує поворот ігрового об`єкта ліворуч за віссю Y з кутом ‘0.5’, подія *Axis Event* виконує такі ж дії, що і аналогічна подія в стані *MovingForward*, але формується перехід *StopLeft*.

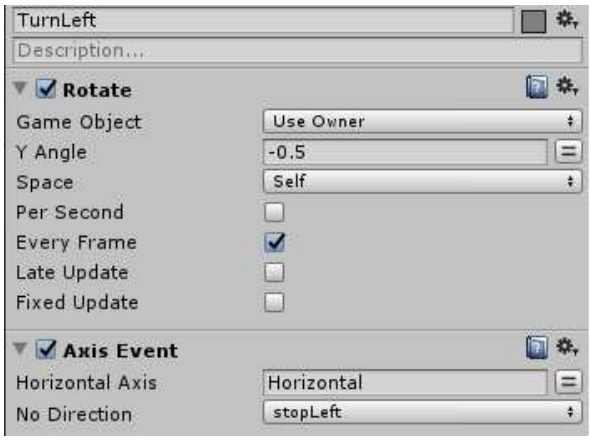


Рисунок 12 – Приклад реалізації стану TurnLeft

На рис. 13 наведено приклад реалізації стану *TurnLeft*. Стан *TurnRight* аналогічний стану *TurnLeft*, але кут повороту ‘-0.5’, при відпусканні клавіші повороту вправо формується перехід *StopRight*.

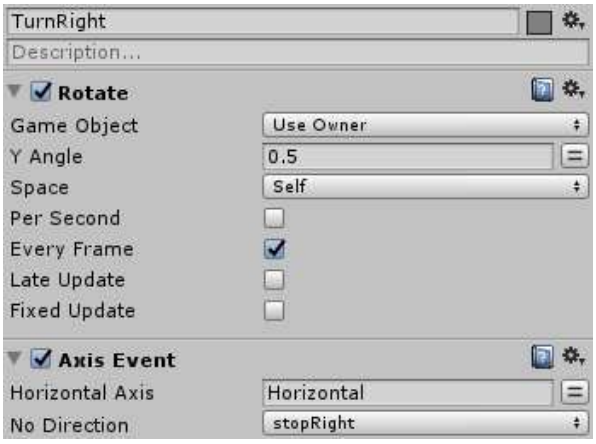


Рисунок 13 – Приклад реалізації стану TurnRight

На рис. 14 наведено приклад реалізації станів *CollisionApple* та *FINISH*.

Рисунок 14 – Приклад реалізації станів CollisionApple

На рис. 15 наведено приклад реалізації станів *CollisionApple* та *FINISH*.

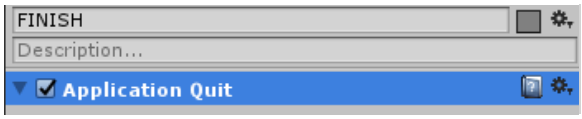


Рисунок 15 – Приклад реалізації стану FINISH

Стан *CollisionApple* виконує знищення яблука, а *FINISH* виконує вихід з гри після зіткнення з відповідними об’єктами, які ідентифікуються за тегом.

Висновки

Особливістю автоматного програмування в ігровому рушії *Unity3D* є можливість перенесення діаграм станів, отриманих при *UML*-моделюванні, але результати експериментів з переведення *UML*-діаграми станів у модель машини станів з використанням *PM*-плагіну та *BM*-плагіну виявили основі чотири особливості, які можуть ускладнити процес переводу та підвищити ймовірність помилок реалізації. При *реалізації UML*-діаграми станів виявлено такі особливості.

По-перше, у *BM*-плагіні використовується спеціальний тип станів – поведінкові дерева. Вони складаються з ієрархії вузлів, що розділяються на гілки, які організують порядок діяльності, та вузли-листя, що виконують елементарні операції.

По-друге, усередині станів *PM*-плагіну не існує ієрархії дій. Вона формується за додатковими схемами поведінки, враховуючи, що весь список дій виконується за один фрейм.

По-третє, у *BM*-плагіні не виділяють окремо початковий та кінцевий стани автомату, бо початковим станом вважається перший стан у послідовності станів *WaitingInput*, який позначається помаранчевим кольором. У *PM*-плагіні обов’язково наявний початковий стан *START*, але кінцевий стан не є обов’язковим.

По-четверте, застосування *BM*-плагіну потребує доопрацювання діаграми, бо при використанні системної події *OnCollisionEnter* неможливо в момент зіткнення виявити, з яким саме об’єктом зіткнулася змійка. *PM*-плагін використовує послідовно дві події *Collision Event*.

У подальшому на основі виявлених особливостей буде розроблено алгоритм автоматизованого синтезу покрокових рекомендацій з реалізації *UML*-діаграми станів для різних плагінів.

Подяки

Ця робота була частково профінансована Європейським Союзом у контексті проекту "GameHub – University–Enterprises Cooperation in Game Industry in Ukraine" (Project Number: 561728-EPP-1-2015-1-ES-EPPKA2-CBHE-JP) за програмою ERASMUS+. Підтримка Європейською Комісією створення цього видання не є схваленням змісту, яка відображає лише погляди авторів, і Комісія не може нести відповідальність за будь-яке використання інформації, що міститься в ній.

Список літератури

1. Роллингз, Э., Моррис, Д. Проектирование и архитектура игр. : Пер. С англ. – М. : Издательский дом «Вильямс», 2006. – 1040 с.

2. Буч Г., Рамбо Д., Якобсон И. Язык UML. Руководство пользователя. 2-е изд.: Пер. с англ. Мухин Н. – М.: ДМК Пресс, 2006. – 496 с.

3. Поликарпова Н. И., Шалыто А. А. Автоматное программирование. 2008. – 167 с.: – Режим доступа: http://is.ifmo.ru/books/_book.pdf (дата обращения: 01.06.18). – Назва з екрану.

4. Dormans J. Engineering emergence: applied theory for game design // PhD thesis The School of Graduate and Postdoctoral Studies The University of Western Ontario, Ontario, Canada 13 January Publisher Amsterdam, 2012 2012. Pages 288 http://hdl.handle.net/11245/1.358623

5. Блажко, О. А. Формалізація проектування ігрової механіки казуальних комп'ютерних ігор / О. А. Блажко, В. В. Антонюк, Ю. Л. Трояновська // Збірник тез доповідей V українсько-німецької конференції «Інформатика. Культура. Техніка», м. Одеса, 22.05 – 26.05., 2017. – Одеса : Політехперіодика, 2017. – С. 91-93.

6. Blazhko, Oleksandr Communication Model of Open Government Data Gamification Based on Ukrainian Websites / Oleksandr Blazhko, Tetiana Luhova, Sergey Melnik, Viktoriia Ruvinska // 4th Experiment@ International Conference (exp.at'17) June 6th – 8th, 2017, University of Algarve, Faro, Portugal. – Pp. 181 – 186.

7. Jere Miles Unity 3D and PlayMaker Essentials: Game Development from Concept to Publishing (Focal Press Game Design Workshops) Paperback – July 27, 2016 Paperback: 506 pages Publisher: A K Peters/CRC Press; 1 edition (July 27, 2016)

8. {b} Behaviour Machine – The Unity Way of Visual Scripting! – Access mode: https://www.behaviourmachine.com/ (last access: 01.06.18). – Title from the screen.

9. Мазин М. А. Парфенов В. Г., Шалыто А. А. Разработка интерактивных приложений Macromedia Flash на базе автоматной технологии // Проектная документация. СПбГУ ИТМО. – Режим доступа: http://is.ifmo.ru/projects/flash/ (дата обращения: 01.06.2018). – Назва з екрану.

10. А. В. Беляев, Д. И. Суясов, А. А. Шалыто Компьютерная игра «Космонавт». Проектирование и реализация // Журнал «Компьютерные инструменты в образовании». 2004. № 4, с. 75 – 84.

11. Adams, E., & Rollings, A. (2007). Fundamentals of Game Design. Upper Saddle River, NJ: Pearson Education, Inc.

12. Damien Djaouti, Julian Alvarez,Jean-Pierre Jessel,Gilles Methel,Pierre Molinier A Gameplay Definition through Videogame Classification // International Journal of Computer Games Technology, 2008. http://dx.doi.org/10.1155/2008/470350

13. Alan Dix, Janet Finlay, Gregory D. Abowd, Russell Beale Human-Computer Interaction Hardcover – Prentice Hall; 3 edition, 2003. 832 pages

14. Лугова, Т.А. UML-моделі як основа проектування комп`ютерних ігор / Т.А. Лугова, Д.А. Лись // Матеріали Восьмої Міжнародної наукової конференції студентів та молодих вчених «Сучасні інформаційні технології 2018» «Modern Information Technology 2018» (23-25 травня 2018) / МОН України; Одес. Нац. політех. ун-т; Ін-т комп'ют. систем. – Одеса : Екологія, 2018. – С.198 – 199.

Стаття надійшла до редколегії 16.09.2018

Рецензент: д-р техн. наук, проф. С.Г. Антощук, Одеський національний політехнічний університет, Одеса.

Блажко Александр Анатольевич
Кандидат технических наук, доцент, доцент кафедры системного программного обеспечения, *orcid.org/0000-0001-7413-275X*
Одесский национальный политехнический университет, Одесса
Антонюк Виктор Викторович
Старший преподаватель кафедры интеллектуальных компьютерных систем и сетей, *orcid.org/0000-0001-8436-5338*
Одесский национальный политехнический университет, Одесса
Трояновская Юлия Людвиговна
Старший преподаватель кафедры информационных систем, *orcid.org/0000-0002-6716-9391*
Одесский национальный политехнический университет, Одесса

ИНСТРУМЕНТАЛЬНЫЕ ОСОБЕННОСТИ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ КОМПЬЮТЕРНЫХ ИГР

Аннотация. *Повышение интереса к геймификации бизнес-процессов обязывает разработчиков формализовать технологические процессы с использованием UML-диаграмм. Парадигма автоматного программирования для разработки программного обеспечения на основе конечных автоматов упрощает тестирование методами белого ящика, а также анализ программного кода при последующем его сопровождении. Но отсутствуют методики автоматизированного синтеза автоматов на основе UML-диаграмм, учитывающие современные игровые движки, например, программные компоненты (плагины) PlayMaker и Behaviour Machine для Unity3D. В работе проведен анализ процесса программирования с использованием указанных плагинов с целью выявления особенностей автоматного программирования компьютерных игр при переходе от UML-моделирования к реализации конечного автомата сценария игры, учитывая формы визуального представления автоматов. Результаты экспериментов выявили четыре основные особенности перевода UML-диаграммы состояний в модель машины состояний, которые могут усложнить процесс перевода и увеличить вероятность ошибок реализации. В дальнейшем указанные особенности будут представлены в виде шаблонов сценариев программирования автоматов, которые, в свою очередь, будут использованы в методике автоматизированного синтеза пошаговых рекомендаций по реализации UML-диаграмм состояний для разных плагинов.*

Ключевые слова: геймификация; автоматное программирование; UML-проекттрование; Unity3D; GameHub

Blazhko Oleksandr
PhD (Eng.), Associate Professor at System Software Department, *orcid.org/0000-0001-7413-275X*
Odessa National Polytechnic University, Odessa
Antoniuk Viktor
Senior Lecturer at Computer Intellectual Systems and Networking Department, *orcid.org/0000-0001-8436-5338*
Odessa National Polytechnic University, Odessa
Troianovska Yuliia
Senior Lecturer at Information System Department, *orcid.org/0000-0002-6716-9391*
Odessa National Polytechnic University, Odessa

INSTRUMENTAL FEATURES OF AUTOMATA-BASED PROGRAMMING OF COMPUTER GAMES

Abstract. *Increasing interest in the gamification of business processes obliges developers to formalize technological processes using Unified Modeling Language (UML) based on UML diagrams. The automat programming paradigm for software development on the basis of finite state machines simplifies white box testing, as well as code analysis with subsequent support. But there are no methods for automating the synthesis of automata based on UML-diagrams, taking into account modern game engines, for example, the software components (plug-ins) PlayMaker and Behavior Machine for Unity3D. Features of automatic programming of computer games in the transition from UML-modeling to the implementation of the finite state machine of the game scenario are considered, taking into account the forms of visual representation of automata by plug-ins of modern computer engines. The results of the experiments of translating the UML state diagram into the state machine model revealed the main 4 features that can complicate the translation process and increase the probability of implementation errors. In the future, these features will be presented in the form of programming scripts that will be used in the technique of automated synthesis of step-by-step recommendations for implementing UML state diagrams for different plug-ins.*

Keywords: gamification; automat programming; game scenario automation; UML design; Unity3D; GameHub

References

1. Rolings, E. & Morris, D. (2006). Design and architecture of games. Moscow, Russia. [in Russian].: Publishing house "Williams",1040.

2. Buch, G., Rambo, D. & Jacobson, I. (2006). Language of the UML. User guide. 2nd ed. Moscow, Russia. [in Russian].: DMK Press, 496.

3. Polikarpova, N.I. & Shalyto, A.A. (2008). Automaton programming., 167. Retrieved from http://is.ifmo.ru/books/_book.pdf

4. Dormans, J. (2012). Engineering emergence: applied theory for game design. PhD thesis The School of Graduate and Postdoctoral Studies The University of Western Ontario. Ontario, Canada.: 13 January Publisher Amsterdam., 288. Retrieved from http: //hdl.handle .net / 11245 / 1.358623

5. Blazhko, O.A. (2017). Formalization of the draft of the draft of the mechanics of casual computerns / O.A. Blazhko, V. V. Antonyuk, Yu. L. Troyanovska // Procc. V Ukrainian-Germany Conference "Informatics. Culture. Tehnika", Odessa, 22.05 - 26.05.2017, pp. 91-93. Odessa: Politechperiodica. [in Ukrainian].

6. Blazhko, Oleksandr. (2017). Communication Model of Open Government Data Gamification Based on Ukrainian Websites / Oleksandr Blazhko, Tetiana Luhova, Sergey Melnik, Viktoriia Ruvinska // 4th Experiment @ International Conference (exp.at'17) June 6th - 8th, 2017, University of Algarve, Faro, Portugal. (pp. 181-186).

7. Miles, Jere. (2016). Unity 3D and PlayMaker Essentials: Game Development from Concept to Publishing (Focal Press Game Design Workshops). A K Peters / CRC Press, (506).

8. {b} Behavior Machine - The Unity Way of Visual Scripting! Retrieved from https://www.behaviourmachine.com/

9. Mazin, M.A., Parfenov, V.G. & Shalyto, A.A. Development of interactive Macromedia Flash applications on the basis of automata technology // Project documentation. SPbSU ITMO. Retrieved from http://is.ifmo.ru/projects/flash/

10. Belyaev, A.V., Suyasov, D.I., Shalyto, A.A. (2004) Computer game "Astronaut". Design and implementation // Journal of Computer Tools in Education, 4, 75-84.

11. Adams, E. & Rollings, A. (2007). Fundamentals of Game Design. Upper Saddle River, NJ: Pearson Education, Inc.

12. Djaouti, Damien, Alvarez, Julian, Jessel,Jean-Pierre, Methel, Gilles & Molinier, Pierre. (2008). A Gameplay Definition through Videogame Classification // International Journal of Computer Games Technology. Retrieved from http://dx.doi.org/10.1155/2008/470350

13. Dix, Alan, Finlay, Janet, Abowd, Gregory D. & Beale, Russell. (2003). Human-Computer Interaction Hardcover. Prentice Hall; 3 edition, 832.

14. Lugova, T.A. (2018). UML models as the basis for computer game design / T.A. Lugova, D.A. Lys // Procc of the Eighth International Scientific Conference of Students and Young Scientists "Modern Information Technologies 2018"; "Modern Information Technology 2018" (May 23-25, 2018) / Ministry of Education and Science of Ukraine; Odessa. Natz. politeh. un-t; Institute of Computer Systems. – Odessa: Ecology, 2018. – P.198-199.

Посилання на публікацію

APA *Blazhko, Oleksandr, Antoniuk, Viktor & Troianovska, Yuliia. (2018). Instrumental Features Of Automata-Based Programming Of Computer Games. Management of Development of Complex Systems, 35, 83 – 92.*

ДСТУ *Блажко О.А., Інструментальні особливості автоматного програмування комп'ютерних ігор [Текст] / О.А. Блажко, В.В. Антонюк, Ю.Л. Трояновська // Управління розвитком складних систем. – 2018. – №35. – С. 83 – 92.*

Робота виконана у рамках проекту програми Еразмус+ КА2 – Розвиток потенціалу вищої освіти. №561728-EPP-1-2015-1-ES-EPPKA2-CBHE-JP-"GameHub: Співробітництво між університетами та підприємствами в сфері ігрової індустрії в Україні".