МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Факультет Магістерської підготовки
Кафедра Автоматизованих систем моніторингу навколишнього середовища

КОМПЛЕКСНА МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: Розробка та оптимізація сюжетної побудови мобільного додатка Dino`s Adventure на платформі Android

Склад:

Частина 1. Розробка та оптимізація сюжетної побудови мобільного додатка Dino`s Adventure

Виконав: студент групи МІС-18 Полєвой Є.В.
(п.і.б.)

Керівник: Перелигін Б.В..
(п.і.б.)

Частина 2. Розробка та оптимізація програмного комплексу мобільного додатка Dino`s Adventure

Виконав: студент групи МІС-18 Клименко М.О
(п.і.б.)

Керівник: Перелигін Б.В.
(п.і.б.)

Староста роботи: Полєвой Є.В.
(п.і.б.)

Керівник роботи: к.т.н., доцент, Перелигін Б.В.
(п.і.б.)

Рецензент: к.ф-м.н. проф., Козловська В.П.
(п.і.б.)

ОДЕСА 2019

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Факультет Магістерської підготовки

Кафедра Автоматизованих систем моніторингу навколишнього середовища

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: Розробка та оптимізація програмного комплексу мобільного додатка Dino`s Adventure

Виконав студент 2 курсу групи МІС-18
спеціальності 122 Комп'ютерні науки
Клименко Микита Олексійович

Керівник д.геог.н., проф.
Перелигін Б.В.

Консультант

Рецензент к.ф.-м.н., доцент
Козловська Валентина Петрівна

Одеса 2019

МІНІСТЕРСТВО ОСВІТИ І НАУКИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Факультет Магістерської підготовки
Кафедра Автоматизованих систем моніторингу навколишнього середовища
Рівень вищої освіти магістр
Спеціальність 122 Комп'ютерні науки
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри АСМНС
Перелигін Б.В.
" 28 " жовтня 2019 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

КЛИМЕНКО МИКИТА ОЛЕКСІЙОВИЧ
(прізвище, ім'я, по батькові)

1. Тема роботи: Розробка та оптимізація програмного комплексу мобільного додатка Dino's Adventure

керівник роботи: Перелигін Борис Вікторович, к.т.н., доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджений наказом вищого навчального закладу від „18" жовтня 2018 року № 235-с

2. Строк подання студентом роботи: 10. 12. 2018 р.
3. Вихідні дані до роботи:
на основі використання існуючих програмних засобів складання інтерактивних моделей реалізувати та оптимізувати програмний комплекс створюваного мобільного додатку

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
1. Аналіз скриптингу на платформі Unity3D
2. Обгрунтування вибору програмних засобів
3. Розробка ігрових механік

5. Перелік графічного матеріалу:
1. Презентація роботи

6. Консультанти розділів роботи:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання:____„28” жовтня 2019 року_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Термін виконання етапів роботи	Оцінка виконання етапу	
			у %	за 4-х бальною шкалою
1	Одержання завдання на виконання магістерської роботи	28.10.2018		
2	Пошук та підбір літератури та інших джерел інформації	31.10.2018		
3	Проведення аналізу предметної області і написання першого розділу пояснювальної записки до магістерської роботи	05.11.2018		
4	Проведення аналізу предметної області і написання другого розділу пояснювальної записки до магістерської роботи	08.11.2018		
5	Проведення аналізу предметної області і написання третього розділу пояснювальної записки до магістерської роботи	14.11.2018		
6	Рубіжна атестація	18-23.11.2018		
7	Проведення аналізу предметної області і написання четвертого розділу пояснювальної записки до магістерської роботи	30.11.2018		
8	Виготовлення презентації	08.12.2018		
9	Друкування пояснювальної записки	08.12.2018		
10	Одержання висновку керівника магістерської роботи	09.12.2018		
11	Проходження нормативного контролю	09.12.2018		
12	Переплетіння пояснювальної записки	09.12.2018		
13	Здача роботи ка кафедру та одержання висновку кафедри про допуск магістерської роботи до захисту	10.12.2018		
14	Перевірка на оригінальність тексту	10.12.2018		
15	Одержання рецензії	10.12.2018		
16	Здача готової магістерської роботи і документів секретарю АК	10.12.2018		
	Інтегральна оцінка виконання етапів календарного плану (як середня по етапам)			

Студент _____ Клименко М.О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Перелигін Б.В.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Кваліфікаційна магістерська робота: 82 с., 4 рис., 1 табл., 2 дод., 25 джерел.

СКРИПТ, ТИП ДАНИХ, ОБ'ЄКТ, ФУНКЦІЯ, ПРОГРАМНИЙ КОД, ЗМІННА, МАСИВ, КООРДИНАТИ, ТРИГЕР, МОВА ПРОГРАМУВАННЯ.

Мета роботи – створення програмних модулів для мобільного додатку Dino's Adventure та їх оптимізація.

Об'єкт роботи – програмний комплекс мобільного додатку Dino's Adventure.

Метод дослідження – створення скриптів у середовищі розробки програмного коду MonoDevelop.

Перший розділ присвячений аналізу скриптинга. У ньому описана роль скриптинга в розробці ігор, створення скриптів, їх основні елементи, структура, вплив на події гри, а також різні типи подій.

Другий розділ присвячений вибору програмних засобів, опису та порівнянню мов програмування, а також особливостям середовища розробки програмного коду.

Третій розділ присвячений розробці ігрових механік для додатку Dino's Adventure. У ньому описуються різні дії об'єктів гри, а також детально описуються основні скрипти, які відповідають за ці дії.

В результаті проведеної роботи було створено понад 30 скриптів, кожен з яких відповідає за різні дії у мобільному додатку Dino's Adventure. Кожен скрипт був прив'язаний до потрібного ігрового об'єкту, а також налаштований для взаємодії з іншими скриптами додатку. Була проведена оптимізація скриптів, а також були усунені програмні помилки.

Створені скрипти дозволяють зробити висновки про складність і обсяг роботи, що виконує розробник програмного коду мобільних додатків, а також про його співпрацю з іншими розробниками. Об'єкт розробки буде розвиватися шляхом додавання нових ігрових механік і покращення існуючих.

ABSTRACT

Qualification master's work: 82 p., 4 pict., 1 tabl., 2 additions, 25 sources.

SCRIPT, DATA TYPE, OBJECT, FUNCTION, PROGRAM CODE, VARIABLE, ARRAY, COORDINATES, TRIGGER, PROGRAMMING LANGUAGE.

Objective of the work – development of the program modules for mobile software Dino's Adventure and their optimization.

Work`s object – program package for mobile software Dino's Adventure.

Research method – development of the scripts in integrated development environment MonoDevelop.

First section is devoted to scripting analysis. There described the role of scripting in game development, scripts development, their main elements, structure, influence on game events and also different types of events.

Second section is devoted to the choice of software means, to description and compare of programming languages, and also it`s devoted to the features of IDE.

Third section is devoted to development of game mechanics for mobile software Dino's Adventure. It describes in details different actions of game objects, and also main scripts that respond for these actions.

As a result of the work were made more than 30 scripts, each of them responds for different actions in mobile software Dino's Adventure. Each script was connected to needed game object and also tuned for interaction with other scripts of the software. The optimization of scripts was made, program errors were eliminated.

The scripts that were made allow to make a conclusion about work`s complexity and capacity that mobile software program code developer performs, and also about his teamwork with other developers. Development object will be evolved with adding of new game mechanics and improvement of those that exist.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

Батьківський об’єкт – це звичайний об’єкт, який виступає для одного або декількох об’єктів "батьком", тобто ці об’єкти успадковуються від нього.

Графічний інтерфейс користувача – тип інтерфейсу, який дозволяє користувачам взаємодіяти з електронними пристроями через графічні зображення та візуальні вказівки.

Дочірні об’єкти – це об’єкти, які мають батьків і успадковують їх події і поведінку.

Клас – це спеціальна конструкція, яка використовується для групування пов’язаних змінних та функцій.

Простір імен – це просто набір класів, в якому для всіх імен цих класів використовується певний префікс.

Пул – це використання декількох блоків пам’яті однакового розміру, для управління пам’яттю, що забезпечує її динамічний розподіл.

Рендеринг – в комп’ютерній графіці це процес отримання зображення за моделлю за допомогою комп’ютерної програми.

Тригер – в комп’ютерних іграх механізм, що перевіряє присутність будь-яких об’єктів ігрового світу в заданому просторі або відстань від цих об’єктів до спеціальної точки.

API – набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення.

CLI – специфікація визначає архітектуру виконавчої системи .NET – CLR і сервіси, які надаються CLR програмам, що виконуються, класи, синтаксис і мнемоніку загальної проміжної мови.

CLR – це компонент пакету Microsoft .NET Framework, віртуальна машина, на якій виконуються всі мови платформи .NET Framework.

DLL – реалізовані компанією Microsoft загальні бібліотеки в ОС Windows та OS/2.

FPS – це частота (швидкість), з якою пристрій формування зображення відображає послідовні зображення, що називаються кадрами.

GTK – кросплатформовий набір інструментів для створення графічних інтерфейсів користувача.

HUD – є частиною графічного інтерфейсу користувача, яка дозволяє отримати різноманітну ігрову інформацію, не викликаючи додаткові меню.

IDE – комплексне програмне рішення для розробки програмного забезпечення.

JIT-компіляція – це технологія збільшення продуктивності програмних систем, що використовують байт-код, шляхом трансляції байт-коду в машинний код безпосередньо під час роботи програми.

Microsoft .NET – програмна технологія, запропонована фірмою Microsoft як платформа для створення звичайних програм та веб-застосунків.

XML – стандарт побудови мов розмітки ієрархічно структурованих даних для обміну між різними застосунками, зокрема, через Інтернет.

ШІ – штучний інтелект.

API – Application Programming Interface, прикладний програмний інтерфейс.

CLI – Common Language Infrastructure, специфікація загальномовної інфраструктури.

CLR – Common Language Runtime, загальномовне виконуюче середовище.

DLL – Dynamic Link Library, динамічно приєднувана бібліотека.

FPS – frames per second, кадрова частота.

GTK – GIMP ToolKit.

GUI – Graphical user interface, графічний інтерфейс користувача.

HUD – head-up display, призначений для перегляду без нахилу голови.

IDE – Integrated development environment, інтегроване середовище розробки.

JIT – Just-in-time, «на льоту».

XML – Extensible Markup Language, розширювана мова розмітки.

ВСТУП

За останні роки ринок мобільних ігор продемонстрував стрімкий розвиток, значно обігнавши попит на версії для інших платформ. Багато популярних IT-компаній зосередили свої зусилля на розробці мобільних ігор, оскільки попит на них знаходиться на високому рівні і безліч розробників перейшли на створення ігрових додатків. Навіть платформи соціальних мереж, такі як Facebook, Instagram і Twitter, приступили до розробки мобільних ігор.

Актуальність роботи полягає в тому, що скриптінг – необхідна складова усіх ігор. Навіть найпростіші ігри потребують скрипти для реакції на дії гравця і організації подій геймплею. Крім того, скрипти можуть бути використані для створення графічних ефектів, управління фізичною поведінкою об’єктів або реалізації користувальницької системи для персонажів гри.

Мета роботи – створення програмних модулів для мобільного додатку Dino’s Adventure та їх оптимізація.

Основні задачі, які потрібно виконати у ході виконання роботи:

– ознайомитись із середовищем розробки програмного коду;
– ознайомитись з інспектором плафторми Unity3D;
– навчитися створювати скрипти;
– навчитися прив’язувати скрипти до об’єктів;
– навчитися пов’язувати скрипти один з одним;
– розробити ігрові механіки для різних рівнів і типів гри;
– розробити функціональність користувальницького інтерфейсу;
– оптимізувати скрипти і виправити програмні помилки.

Скрипти, які розроблюються, тісно пов’язані з іншими частинами додатку, такими як анімація, моделі та об’єкти Unity3D. Скрипти відповідають за відтворення і зупинку анімацій в потрібний момент, частоту програвання анімацій, тривалість їх відтворення та їх швидкість, за

пересування об’єктів і моделей, їх появу і зникнення на ігровий карті, а також за їх взаємозв’язок із користувачем.

1 АНАЛІЗ СКРИПТИНГУ НА ПЛАТФОРМІ UNITY3D

Під терміном «скрипт» Unity3D має на увазі файл з вихідним кодом, в якому описується клас, успадкований від класу MonoBehaviour. Скрипти це призначені для користувача компоненти, які додають «поведінку» об'єкту сцени, так само як і стандартні компоненти. Скрипти можуть бути використані для створення графічних ефектів, управління фізичною поведінкою об’єктів або реалізації користувальницького ШІ (штучного інтелекту) системи для персонажів гри.

Поведінка ігрових об’єктів контролюється за допомогою компонентів (Components), які приєднуються до них. Unity дозволяє створювати свої компоненти, використовуючи скрипти. Вони дозволяють активувати ігрові події, змінювати параметри компонентів, і відповідати на введення користувача.

Unity спочатку підтримує дві мови програмування:

– C#;

– JavaScript.

На додаток до цих, з Unity можуть бути використані багато інших мов сімейства .NET, якщо вони можуть компілювати сумісні DLL [1], [2][1)].

1.1 Опис створення скриптів

На відміну від інших ресурсів, скрипти зазвичай створюються безпосередньо в Unity. Можна створити скрипт, використовуючи меню

[1)] [1] Unity – руководство: Скриптинг. URL: https://docs.unity3d.com/ru/530/Manual/ScriptingSection.html (дата звернення 4.11.2019).

[2] Unity – руководство: Обзор скриптинга. URL: https://docs.unity3d.com/ru/530/Manual/ScriptingConcepts.html (дата звернення 1.11.2019).

Create в лівому верхньому кутку панелі Project або обравши Assets> Create> C# Script (або JavaScript скрипт) в головному меню.

Новий скрипт (рис. 1) буде створений в папці, яку обрали в панелі Project. Ім’я нового скрипта буде виділено, пропонуючи ввести нове ім’я.

Краще ввести нове ім’я скрипта відразу після створення, ніж змінювати його потім. Введене ім’я буде використано, щоб створити початковий текст у скрипті [3][1)].

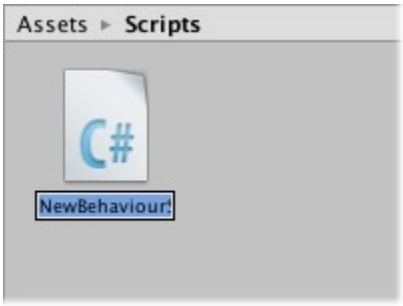


Рисунок 1 – Новий скрипт C#

1.2 Опис змінних та інспектора

При створенні сценарію створюється й свій власний новий тип компонента, який можна приєднати до ігрових об’єктів, як і будь-який інший компонент.

Так само, як і інші компоненти часто мають властивості, які можна редагувати в інспекторі, можна дозволити редагування значень у сценарії і від інспектора.

```
using UnityEngine;
using System.Collections;
public class MainPlayer : MonoBehaviour {
    public string myName;
    // Use this for initialization
    void Start () {
```

[1)] [3] Unity – руководство: Создание и Использование Скриптов. URL: https:// docs.unity3d.com/ru/530/Manual/CreatingAndUsingScripts.html (дата звернення 1.11.2019).

```
        Debug.Log("I am alive and my name is " + myName);
    }
    // Update is called once per frame
    void Update () {
    }
}
```

Цей код створює поле для редагування в інспекторі з написом "My Name" (рис. 2).

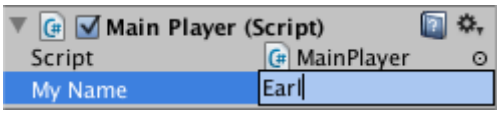


Рисунок 2 – Поле для редагування в інспекторі з написом "My Name"

Unity створює мітку у вікні Inspector шляхом додавання пробілу всюди, де в імені змінної зустрічається велика літера. Однак це виключно в цілях відображення, і треба завжди в коді використовувати ім’я змінної. Якщо відредагувати значення змінної і потім натиснути Play, можна побачити, що повідомлення містить введений текст.

У C# потрібно оголосити змінну загальнодоступною, щоб побачити її в інспекторі. У JavaScript змінні є загальнодоступними, якщо не вказано, що вони повинні бути приватними.

```
#pragmastrict
privatevarinvisibleVar: int;
functionStart () {
}
```

Unity дозволяє змінювати значення змінних скрипта у запущеній грі. Це дозволяє подивитися на всі ефекти від змін відразу ж, без зупинки і перезапуску. Коли програвання закінчується, значення змінних скидається в

той стан, який вони мали до натискання кнопки Play. Це гарантує, що можна змінювати настройки об’єктів, не лякаючись щось зіпсувати [4][1)].

1.3 Структура файлу скрипта

Після подвійного клацання на скрипті в Unity, він буде відкритий в текстовому редакторі. За замовчуванням Unity буде використовувати Mono Develop, але можна вибрати будь-який редактор з панелі External Tools в налаштуваннях Unity.

Вміст файлу буде виглядати приблизно так:

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

Скрипт взаємодіє із внутрішніми механізмами Unity за рахунок створення класу, що успадкований від вбудованого класу, званого MonoBehaviour. Можна думати про клас, як про свого роду план для створення нового типу компонента, який може бути прикріплений до ігрового об’єкту. Кожного разу, коли скриптовий компонент приєднується до ігрового об’єкту, створюється новий екземпляр об’єкту, визначений планом. Ім’я класу береться з імені, яке ви вказали при створенні файлу. Ім’я класу та ім’я файлу повинні бути однаковими, для того, щоб скриптовий компонент міг бути приєднаний до ігрового об’єкту.

[1)] [4] Unity – руководство: Variables and the Inspector. URL: https://docs.unity3d.com/ru/530/Manual/VariablesAndTheInspector.html (дата звернення 3.11.2019).

Основними є дві функції, визначені усередині класу. Функція Update – це місце для розміщення коду, який буде обробляти оновлення кадру для ігрового об’єкту. Це може бути рух, спрацьовування дій і відповідна реакція на введення користувача, в основному все, що повинно бути оброблено з плином часу в ігровому процесі. Щоб дозволити функції Update виконувати свою роботу, часто буває корисно форматувати змінні, вважати властивості та здійснити зв’язок з іншими ігровими об’єктами до того, як будуть здійснені будь-які дії. Функція Start буде викликана Unity до початку ігрового процесу (тобто до першого виклику функції Update), і це ідеальне місце для виконання ініціалізації.

Ініціалізація об’єкта виконується не у функції-конструкторі. Це тому, що створення об’єктів обробляється редактором і відбувається не на початку ігрового процесу, як можна було очікувати. Якщо спробувати визначити конструктор для скриптового компонента, він буде заважати нормальній роботі Unity і може викликати серйозні проблеми з проектом.

Скрипт Unity трохи відрізняється від сценарію C#:

```
#pragma strict
function Start () {
}
function Update () {
}
```

Тут функції Start і Update мають таке ж значення, але клас не оголошений явно. Передбачається, що скрипт сам по собі визначає клас; він буде неявно похідним від MonoBehaviour і отримає своє ім’я від імені файлу скриптового ресурсу [3], [5], [6][1)].

[1)] [3] Unity – руководство: Создание и Использование Скриптов. URL: https:// docs.unity3d.com/ru/530/Manual/CreatingAndUsingScripts.html (дата звернення 1.11.2019).

[5] Unity – руководство: Scripting Restrictions. URL: https://docs.unity3d.com/ ru/530/ Manual/ScriptingRestrictions.html (дата звернення 6.11.2019).

[6] Unity – руководство: Script Serialization. URL: https://docs.unity3d.com/ ru/530/ Manual/script-Serialization.html (дата звернення 6.11.2019).

1.4 Аналіз управління ігровим об’єктом

У редакторі Unity можна змінювати властивості компонента, використовуючи вікно Inspector. Так, наприклад, зміна позиції компонента Transform призведе до зміни позиції ігрового об’єкту. Аналогічно, можна змінити колір матеріалу компонента Renderer або масу твердого тіла (RigidBody) з відповідним впливом на відображення або поведінку ігрового об’єкту. Здебільшого скрипти також змінюють властивості компонентів для управління ігровими об’єктами. Різниця, однак, в тому, що скрипт може змінювати значення властивості поступово з часом або по отриманню введення від користувача. За рахунок зміни, створення і знищення об’єктів в заданий час може бути реалізований будь-який ігровий процес.

Скрипт визначає тільки план компонента і, таким чином, ніякий його код не буде активований до тих пір, поки екземпляр скрипта не буде приєднаний до ігрового об’єкту. Можна прикріпити скрипт перетягненням ресурсу скрипта на ігровий об’єкт в панелі Hierarchy, або через вікно Inspector вибраного ігрового об’єкту. Є також підменю Scripts в меню Component, яке містить усі скрипти, доступні в проекті, включаючи створені програмістом. Екземпляр скрипта (рис. 3) виглядає так само як і інші компоненти у вікні Inspector.

Рисунок 3 – Екземпляр скрипта

Після приєднання скрипт почне працювати, коли буде натиснута кнопка Play і гра буде запущена. Можна перевірити це, додавши наступний код у функцію Start:

```
// Usethisforinitialization
void Start () {
   Debug.Log("I am alive!");
}
```

Debug.Log – це команда, яка просто виводить повідомлення на консольне виведення Unity. Якщо натиснути зараз Play, можна побачити повідомлення понизу основного вікна редактора Unity у вікні Console (меню: Window > Console).

Найбільш простим і поширеним є випадок, коли скрипту необхідно звернутися до інших компонентів, приєднаних до того ж GameObject. Компонент насправді є екземпляром класу, таким чином першим кроком буде отримання посилання на екземпляр компонента, з яким треба працювати. Це робиться за допомогою функції GetComponent. Характерно, що об’єкт компонента зберігається в змінну, це робиться в C# за допомогою наступного синтаксису:

```
void Start () {
   Rigidbody rb = GetComponent<Rigidbody>();
}
В JavaScript синтаксис трохи відрізняється:
function Start () {
   var rb = GetComponent.<Rigidbody>();
}
```

Як тільки отримується посилання на екземпляр компонента, можна встановлювати значення його властивостей, які можна змінити в вікні Inspector:

```
void Start () {
   Rigidbody rb = GetComponent<Rigidbody>();
   // Change the mass of the object's Rigidbody.
rb.mass = 10f;
}
```

Можно мати безліч користувальницьких скриптів, приєднаних до одного і того ж об’єкту. Якщо потрібно звернутися до одного скрипту з

іншого, можна використовувати GetComponent, використовуючи при цьому ім’я класу скрипта (або ім’я файлу), щоб вказати який тип компонента потрібен.

Якщо спробувати отримати компонент, який не був доданий до ігрового об’єкту, тоді GetComponent поверне null, виникне помилка порожнього посилання при виконанні (null reference error at run time), якщо спробувати змінити будь-які значення у порожнього об’єкта.

Іноді ігрова сцена може використовувати кілька об’єктів одного типу, таких як вороги, шляхові точки і перешкоди. Може виникнути необхідність відстеження їх в певному скрипті, який управляє або реагує на них (наприклад, всі шляхові точки можуть знадобитися для скрипта пошуку шляху). Можна використовувати змінні для зв’язування цих об’єктів, але це зробить процес проектування втомливим, якщо кожну нову точку маршруту потрібно буде перетягнути в змінну в скрипті. Аналогічно, при видаленні точки маршруту доведеться видаляти посилання на відсутній об’єкт. У випадках, на зразок цього, найчастіше зручно управляти набором об’єктів, зробивши їх дочірніми до одного батьківського об’єкта. Дочірні об’єкти можуть бути отримані, використовуючи компонент Transform батька (так як всі ігрові об’єкти неявно містять Transform).

Знаходження ігрових об’єктів у будь-якому місці ієрархії є завжди, коли є деяка інформація, за якою їх можна ідентифікувати. Окремі об’єкти можуть бути отримані на ім’я, використовуючи функцію GameObject.Find:

```
GameObject player;
void Start() {
   player = GameObject.Find("MainHeroCharacter");}
```

Об’єкт або колекція об’єктів можуть бути також знайдені по їх тегу, використовуючи функції GameObject FindWithTag та GameObject FindGameObjectsWithTag [7], [8], [9][1)]:

[1)] [7] Unity – руководство: Управление игровыми объектами (GameObjects) с помощью компонентов. URL: https://docs.unity3d.com/ru/530/Manual/ControllingGameOb-

```
GameObject player;
GameObject[] enemies;
void Start() {
   player = GameObject.FindWithTag("Player");
   enemies = GameObject.FindGameObjectsWithTag("Enemy");
}
```

1.5 Опис Update подій

Гра – це щось подібне до анімації, в якій кадри генеруються на ходу. Ключовий концепт в програмуванні ігор полягає в зміні позиції, стану і поведінки об'єктів у грі прямо перед відрисовкою кадру. Такий код в Unity зазвичай розміщують у функції Update. Update викликається перед відрисовкою кадру і перед розрахунком анімацій:

```
voidUpdate() {
float distance = speed * Time.deltaTime * Input.GetAxis("Horizontal");
transform.Translate(Vector3.right * distance);
}
```

Фізичний движок також оновлюється фіксованими за часом кроками, аналогічно тому як працює відрисовка кадру. Окрема функція події FixedUpdate викликається прямо перед кожним оновленням фізичних даних. Так як оновлення фізики і кадру відбувається не з однаковою частотою, то можна отримати точніші результати від коду фізики, якщо помістити його у функцію FixedUpdate, а не в Update:

```
void FixedUpdate() {
   Vector3 force = transform.forward * driveForce * Input.GetAxis("Vertical");
rigidbody.AddForce(force);
}
```

---

jects Components.html (дата звернення 7.11.2019).

[8] Unity – руководство: What is a Null Reference Exception?. URL: https:// docs.unity3d.com/ru/530/Manual/NullReferenceException.html (дата звернення 7.11.2019).

[9] Unity – руководство: Important Classes. URL: https://docs.unity3d.com/ru/ 530/ Manual/ScriptingImportantClasses.html (дата звернення 20.11.2019).

Також іноді корисно мати можливість внести додаткові зміни у мить, коли в усіх об’єктів в сцені відпрацювали функції Update і FixedUpdate і розрахувалися усі анімації. Як приклад, камера повинна залишатися прив’язаною до цільового об’єкту; підстроювання орієнтації камери повинне робитися після того, як цільовий об’єкт змістився. Іншим прикладом є ситуація, коли код скрипта повинен перевизначити ефект анімації (припустимо, змусити голову персонажа обернутися до цільового об’єкту в сцені). У ситуаціях такого роду можна використати функцію LateUpdate [10], [11][1)]:

```
voidLateUpdate() {
Camera.main.transform.LookAt(target.transform);
}
```

1.6 Опис подій ініціалізації

Іноді корисно мати можливість викликати код ініціалізації перед будь-якими оновленнями, що відбуваються під час гри. Функція Start викликається до оновлення першого кадру або фізики об’єкту. Функція Awake викликається для кожного об’єкту в сцені у момент завантаження сцени. Хоча для різних об’єктів функції Start і Awake і викликаються в різному порядку, усі Awake будуть виконані до виклику першого Start. Це означає, що код у функції Start може використати все, що було зроблено у фазі Awake [12][2)].

1.7 Опис подій GUI

[1)] [10] Unity – руководство: Функции событий. URL: https://docs.unity3d.com/ru/530/Manual/EventFunctions.html (дата звернення 20.11.2019).

[11] Unity – руководство: Unity События (UnityEvents). URL: https://docs.unity3d.com/ru/530/Manual/UnityEvents.html (дата звернення 20.11.2019).

[2)] [12] Unity – руководство: Coroutines. URL: https://docs.unity3d.com/ru/530/Manu-al/Coroutines.html (дата звернення 1.12.2019).

Графічний інтерфейс користувача (ГІК, англ. GUI, Graphical user interface) – тип інтерфейсу, який дозволяє користувачам взаємодіяти із електронними пристроями через графічні зображення та візуальні вказівки, на відміну від текстових інтерфейсів, заснованих на використанні тексту, текстовому наборі команд та текстовій навігації.

Виконання дій в GUI – це безпосередня маніпуляція з графічними елементами. Окрім комп'ютерів, GUI використовується в мобільних пристроях, таких, як мобільні телефони, планшети, електронні книги, портативні медіаплеєри тощо. Термін ГІК зазвичай не вживають стосовно інтерфейсів з низькою роздільною здатністю. Наприклад, в відеоіграх вико-ристовують інтерфейс HUD.

У Unity є система для відрисовки елементів управління GUI поверх усього, що відбувається у сцені і реагування на кліки по цих елементах. Цей код обробляється трохи інакше, ніж звичайне оновлення кадру, так що він має бути поміщений у функцію OnGUI, яка періодично викликатиметься:

```
voidOnGUI() {
GUI.Label(labelRect, "GameOver");
}
```

Також можна визначати події миші, які спрацьовують у GameObject, що знаходиться в сцені. Це можна використовувати для наведення знарядь або для відображення інформації про персонажа під покажчиком курсору миші. Існує набір функцій подій OnMouseXXX (наприклад, OnMouseOver, OnMouseDown), який дозволяє скрипту реагувати на дії з мишею користувача. Наприклад, якщо кнопка миші натиснута в той час, коли курсор миші знаходиться над певним об'єктом, то, якщо у скрипті цього об'єкту є функція OnMouseDown, вона буде викликана [11][1)].

[1)] [11] Unity – руководство: Unity События (UnityEvents). URL: https:// docs.unity3d.-com/ru/530/Manual/UnityEvents.html (дата звернення 20.11.2019).

1.8 Опис подій фізики

Фізичний движок повідомить про зіткнення з об’єктом за допомогою виклику функцій подій в скрипті цього об’єкту. Функції OnCollisionEnter, OnCollisionStay і OnCollisionExit будуть викликані на початку, у продовженні і завершенні контакту. Відповідні функції OnTriggerEnter, OnTriggerStay і OnTriggerExit будуть викликані, коли коллайдер об’єкту налаштований як Trigger (тобто цей коллайдер просто визначає, що його щось перетинає і не реагує фізично). Ці функції можуть бути викликані кілька разів поспіль, якщо виявлено більше ніж один контакт під час поновлення фізики, тому в функцію передається параметр, що надає додаткову інформацію про зіткнення (координати, "особистість" об’єкту, що входить і т.п.) [10], [13][1)]:

```
void OnCollisionEnter(otherObj: Collision) {
   if (otherObj.tag == "Arrow") {
ApplyDamage(10);
   }
}
```

1.9 Аналіз створення і знищення ігрових об’єктів

Деякі ігри мають постійну кількість об’єктів на сцені, проте зазвичай персонажі та інші об’єкти створюються і видаляються під час гри. У Unity, ігровий об’єкт (GameObject) може бути створений, використовуючи функцію Instantiate, яка робить копію існуючого об’єкта:

```
public GameObject enemy;
void Start() {
   for (int i = 0; i < 5; i++) {
```

[1)] [10] Unity – руководство: Функции событий. URL: https://docs.unity3d.com/ ru/530/ Manual/EventFunctions.html (дата звернення 20.11.2019).

[13] Unity – руководство: Time and Framerate Management. URL: https:// docs.unity3d.com/ru/530/Manual/TimeFrameManagement.html (дата звернення 1.12.2019).

```
        Instantiate(enemy);
    }
}
```

Об’єкт, з якого береться копія, не зобов’язаний бути присутнім на сцені. Набагато частіше використовується об’єкт, який був перетягнутий на відкриту змінну (public variable) з файлів проекту в панелі Project. Також, копіюючи ігровий об’єкт (GameObject), копіюються всі компоненти оригінального об’єкта.

Також є функція Destroy, яка знищить об’єкт після того, як завантаження кадру буде завершено або опціонально після короткої паузи:

```
void OnCollisionEnter(Collision otherObj) {
    if (otherObj.gameObject.tag == "Missile") {
        Destroy(gameObject,.5f);
    }
}
```

Функція Destroy може знищувати окремі компоненти без впливу на сам об’єкт. Часта помилка – писати функцію Destroy (this), яка насправді знищить тільки скриптовий компонент, що викликає, замість того, щоб знищити ігровий об’єкт, до якого приєднаний цей скрипт [14][1)].

1.10 Опис спеціальних папок і порядку компіляції скриптів

Можна вибирати будь-які імена для папок проекту. Однак, в Unity зарезервовані деякі імена, які вказують на те, що вміст папок має спеціальне призначення. Деякі з них впливають на порядок компіляції скриптів. У компіляції скриптів є чотири окремі фази і порядок їх компілювання визначається батьківською папкою.

[1)] [14] Unity – руководство: Создание и уничтожение игровых объектов (GameObjects). URL: https://docs.unity3d.com/ru/530/Manual/CreateDestroyObjects.html (дата звернення 7.12.2019).

Це має велике значення в разі, якщо скрипт повинен звернутися до класів, визначених у інших скриптах. Основне правило полягає в тому, щоб не було посилань на скрипти, які компілюються після фази. Все, що компілюється в поточній або раніше виконаній фазі, має бути повністю доступно.

Інша ситуація буває, коли скрипт, написаний на одній мові, повинен посилатися на клас, визначений іншою мовою (наприклад, скрипт на мові JavaScript оголошує змінні класу, визначені в C# скрипті). В цьому випадку клас, на який посилаються, повинен бути скомпільований в більш ранній фазі.

Є такі фази компіляції:

- фаза 1: виконуються скрипти із папок з іменами Standard Assets, Pro Standard Assets і Plugins;
- фаза 2: виконуються скрипти із папок з іменами Standard Assets, Pro Standard Assets і Plugins;
- фаза 3: всі інші скрипти, які не перебувають в папці Editor;
- фаза 4: всі скрипти, що залишилися (тобто знаходяться в папці Editor).

Додатково, будь-який скрипт, що знаходиться в папці WebPlayerTemplates, на самому верху папки Assets, взагалі не буде скомпільований. Ця поведінка трохи відрізняється для вкладених папок з іншими іменами (наприклад, Scripts / Editor працює як папка для скриптів редактора і Scripts / WebPlayerTemplates не завадить компіляції).

У загальному випадку, скрипт UnityScript посилається на клас, визначений на мові C#. Потрібно розмістити C# файл в папку Plugins, а UnityScript – у будь-яку не спеціальну папку. Якщо цього не зробити, можна отримати помилку, яка говорить про те, що C# клас не знайдений [15][1)].

[1)] [15] Unity – руководство: Специальные папки и порядок компиляции скриптов. URL: https://docs.unity3d.com/ru/530/Manual/ScriptCompileOrderFolders.html (дата звернення 7.12.2019).

1.11 Опис просторів назв

Коли проект стає більшим, а кількість класів збільшується, ймовірність збігу імен класів також зростає. Таке може статися, якщо кілька програмістів працюють окремо над різними аспектами гри. Наприклад, один програміст пише код для управління персонажем гравця, в той час як інший пише еквівалентний код для управління супротивниками. Обидва програміста вирішили назвати головний клас Controller, однак при об'єднанні їх проектів це призведе до збігу імен класів, в результаті чого використання класу з таким ім'ям виявилося б двозначним.

В деякій мірі, ця проблема може бути вирішена шляхом перейменування класів, коли виявлено збіг імен (наприклад, класи вище можуть мати імена PlayerController і EnemyController). Однак, це дуже проблематично, коли є кілька таких класів, або коли оголошуються змінні з їх використанням – кожна згадка старої назви класу має бути замінена, щоб код міг скомпілюватися.

Мова C# пропонує простори імен в якості вирішення даної проблеми. Простір імен – це просто набір класів, в якому для всіх імен цих класів використовується певний префікс. У наведеному нижче прикладі класи Controller1 і Controller2 відносяться до простору імен Enemy:

```
namespace Enemy {
    public class Controller1 : MonoBehaviour {
        ...
    }
    public class Controller2 : MonoBehaviour {
        ...
    }
}
```

У коді ці класи викликаються відповідно Enemy.Controller1 і Enemy.Controller2. Це краще, ніж перейменування класів, оскільки оголошення простору імен може бути виконано для існуючих оголошень

класу (тобто, немає необхідності міняти імена усіх класів індивідуально). Крім того, можна використовувати однакові простори імен для класів, які зберігаються в різних вихідних файлах.

Можна уникнути багаторазової вказівки префікса простору імен, додавши директиву using у верхній частині файлу:

```
using Enemy;
```

Цей рядок говорить про те, що використання назви класів Controller1 і Controller2 насправді означатиме класи Enemy.Controller1 і Enemy.Controller2 відповідно. Якщо в скрипті передбачається використання класів із таким же ім’ям з іншого простору імен (припустимо з простору Player), то для їх виклику потрібно буде вказати префікс. Якщо імена використовуваних класів з двох різних просторів імен, зазначених у директиві, співпадуть – компілятор повідомить про помилку [16][1)].

1.12 Опис атрибутів

Атрибути (Attributes) – це маркери, які можуть бути розміщені перед класом, властивостями або функціями в скрипті, щоб вказати особливу поведінку. Наприклад, атрибут HideInInspector може бути доданий перед оголошенням властивості для запобігання відображення цієї властивості в інспекторі, навіть якщо воно публічне. В JavaScript ім’я атрибута починається зі знака "@", а в C # і Boo, він розміщується між квадратними дужками:

```
[HideInInspector]
public float strength;
```

[1)] [16] Unity – руководство: Пространства имён. URL: https://docs.unity3d.com/ ru/ 530/Manual/Namespaces.html (дата звернення 7.12.2019).

Unity надає ряд атрибутів, які перераховані в довіднику по скриптам. Є також атрибути, визначені в .NET бібліотеці, які іноді можуть бути корисними у коді [17][2)].

1.13 Порядок виконання функцій подій

У скриптингу Unity є деяка кількість функцій подій, які виконуються в заздалегідь заданому порядку в міру виконання скрипта.

Спочатку виконується функція редактора.

Функція Reset (скидання) викликається для ініціалізації властивостей скрипта, коли він тільки приєднується до об'єкту і тоді, коли використовується команда Reset.

Потім, функції, що викликаються при запуску сцени (один раз для кожного об'єкта на сцені).

Функція Awake завжди викликається до будь-яких функцій Start і також після того, як об'єкт був викликаний в сцену (якщо GameObject неактивний на момент старту, Awake не буде викликаний, поки GameObject не буде діяти, або функція в якомусь прикріпленому скрипті не викличе Awake).

Функція OnEnable (викликається тільки якщо об'єкт активний) викликається відразу після включення об'єкту. Це відбувається при створенні зразку MonoBehaviour, наприклад, при завантаженні рівня або при виклику GameObject з компонентом скрипта.

Функція OnLevelWasLoaded інформує гру про те, що завантажений новий рівень.

[2)] [17] Unity – руководство: Атрибуты. URL: https://docs.unity3d.com/ru/530/Manu-al/Attributes.html (дата звернення 8.12.2019).

Для об’єктів, доданих в сцену відразу, функції Awake і OnEnable для всіх скриптів будуть викликані до виклику Start, Update і т.д. Природно, для об’єктів, викликаних під час ігрового процесу, такого не буде.

Далі йдуть функції, що викликаються перед першим оновленням кадру.

Функція Start викликається до поновлення першого кадру (first frame), тільки якщо скрипт включений.

Для об’єктів, доданих на сцену, функція Start буде викликатися у всіх скриптах до функції Update. Це не може бути забезпечено при створенні об’єкта безпосередньо під час гри.

Далі виконується функція, що викликається між кадрами.

Функція OnApplicationPause викликається в кінці кадру, під час якого викликається пауза, що ефективно між звичайними оновленнями кадрів. Один додатковий кадр буде виданий після виклику OnApplicationPause, щоб дозволити грі відобразити графіку, яка вказує на стан паузи.

Коли відстежується ігрова логіка і взаємодії, анімації, позиції камери і т.д. є кілька різних подій, які можна використовувати. За загальним шаблоном, велика частина завдань виконується всередині функції Update, але є також ще інші функції, які можна використовувати.

Функція FixedUpdate – часто трапляється, що FixedUpdate викликається частіше ніж Update. FU може бути викликаний кілька разів за кадр, якщо FPS низький, і функція може бути і зовсім не викликана між кадрами, якщо FPS високий. Всі фізичні обчислення і оновлення відбуваються відразу після FixedUpdate. При застосуванні розрахунків пересування всередині FixedUpdate, не потрібно множити значення на Time.deltaTime. Тому що FixedUpdate викликається відповідно до надійного таймеру, незалежного від частоти кадрів.

Функція Update викликається раз за кадр. Це головна функція для оновлень кадрів.

Функція LateUpdate викликається раз в кадр, після завершення Update. Будь-які обчислення, зроблені в Update, будуть вже виконані на момент

початку LateUpdate. Часто LateUpdate використовують для переслідуючої камери від третьої особи. Якщо переміщати і повертати персонажа в Update, то можна виконати всі обчислення переміщення і обертання камери в LateUpdate. Це забезпечить те, що персонаж буде рухатися до того, як камера відстежить його позицію.

Потім виконуються функції рендеринга.

Функція OnPreCull викликається до того, як камера відсіче сцену. Відсікання визначає, які об’єкти будуть видні у камері. OnPreCull викликається прямо перед тим, як починається відсікання.

Функції OnBecameVisible/OnBecameInvisible викликаються тоді, коли об’єкт стає видимим / невидимим будь-якій камері.

Функція OnWillRenderObject викликається один раз для кожної камери, якщо об’єкт в полі зору.

Функція OnPreRender викликається перед тим, як камера почне прорисовувати сцену.

Функція OnRenderObject викликається після того, як всі звичайні прорисовування сцени завершаться. Можна використовувати клас GL або Graphics.DrawMeshNow, щоб рисовати призначену для користувача геометрію в даній точці.

Функція OnPostRender викликається після того, як камера завершить рендер сцени.

Функція OnRenderImage викликається після завершення прорисовування сцени, для можливості пост-обробки зображення екрану.

Функція OnGUI викликається кілька разів за кадр і відповідає за елементи інтерфейсу (GUI). Спочатку обробляються події макету і розрисовування, після чого йдуть події клавіатури / мишки для кожної події.

Функція OnDrawGizmos використовується для відтворення Gizmo у вікні Scene View з метою візуалізації.

Нормальні поновлення співпрограм запускаються після завершення із функції Update. Співпрограма – це функція, яка призупиняє своє виконання

(yield), поки дані YieldInstruction не завершаться. Різні способи використання співпрограми:

– yield співпрограма продовжить виконання після того, як все Update функції були викликані в наступному кадрі;
– yield WaitForSeconds продовжує виконання після заданої тимчасової затримки, і після всіх Update функцій, викликаних в підсумковому кадрі;
– yield WaitForFixedUpdate продовжує виконання після того, як всі функції FixedUpdate були викликані в усіх скриптах;
– yield WWW продовжує виконання після завершення WWW-завантаження;
– yield StartCoroutine зв'язує співпрограму, і буде чекати, поки не завершиться співпрограма MyFunc.

Далі виконується функція, що викликається, коли об'єкт руйнується.

Функція OnDestroy викликається після всіх оновлень кадру в останньому кадрі об'єкта, поки він ще існує (об'єкт може бути знищений за допомогою Object.Destroy або при закритті сцени).

Насамкінець виконуються функції, що викликаються при виході.

Функція OnApplicationQuit викликається для всіх ігрових об'єктів перед тим, як програма закривається. У редакторі викликається тоді, коли гравець зупиняє ігровий режим. У веб-плеєрі викликається по закритті веб вікна.

Функція OnDisable викликається, коли об'єкт відключається або стає неактивним [18][1)].

1.14 Опис загальних функцій

[1)] [18] Unity – руководство: Порядок выполнения функций событий. URL: https://docs.unity3d.com/ru/530/Manual/ExecutionOrder.html (дата звернення 8.12.2019).

Деякі функції в довідці по скриптам (наприклад, різні функції GetComponent) перераховані в варіанті, який містить букву T або ім’я типу в кутових дужках після імені функції:

```
//C#
void FuncName<T>();
//JS
function FuncName.<T>(): T;
```

Вони відомі як загальні функції. Їх значимість для програмування полягає в тому, щоб вказати типи параметрів або повертаємого типу при виконанні функції. В JavaScript, це може бути використано, щоб обійти обмеження динамічної типізації:

```
// The type is correctly inferred since it is defined in the function call.
//In C#
var obj = GetComponent<Rigidbody>();
//In JS
var obj = GetComponent.<Rigidbody>();
```

У C # це може скоротити кількість коду:

```
Rigidbody rb = go.GetComponent<Rigidbody>();
// ...as compared with:
Rigidbody rb = (Rigidbody) go.GetComponent(typeof(Rigidbody));
```

Будь-яка функція, що має загальний варіант, який вказаний на своїй сторінці довідки, дозволяє використовувати спеціальний синтаксис виклику [19][1)].

1.15 Аналіз автоматичного управління пам’яттю

При створенні об'єкта, рядка або масиву, пам'ять для його зберігання виділяється з центрального пулу, який називається купа (heap). Коли використання елемента припиняється, пам'ять, яку він займав, можна буде звільнити і використовувати для чого-небудь ще. У минулому, виділення і звільнення

[1)] [19] Unity – руководство: Общие функции. URL: https://docs.unity3d.com/ru/ 530/ Manual/GenericFunctions.html (дата звернення 8.12.2019).

цих блоків пам'яті за допомогою викликів відповідних методів в основному лежало на плечах програмістів. Тепер пам'яттю автоматично керують середовища виконання, наприклад, движок Mono у Unity. Автоматичне управління пам'яттю вимагає менше зусиль при написанні коду, ніж пряме виділення чи звільнення і значно зменшує потенціал для витоку пам'яті (ситуації, коли пам'ять була виділена, але згодом так і не була звільнена).

При виконанні функції, значення її параметрів копіюються в зону пам'яті, зарезервовану спеціально для цього виклику. Типи даних, які займають всього лише кілька байт, можуть бути скопійовані легко і швидко. Однак, зазвичай об'єкти, рядки і масиви набагато більші, і було б дуже неефективно копіювати ці дані як зазвичай. На щастя, це не обов'язково; реальне місце зберігання для великих елементів виділяється з купи і для запам'ятовування його розташування використовується невелике "вказівне" значення. Після цього, під час передачі параметра, потрібно буде скопіювати тільки покажчик. Поки система середовища виконання знаходить елемент, який визначається покажчиком, можна використовувати одиночну копію даних так часто, як це потрібно.

Типи, які зберігаються безпосередньо і копіюються при передачі параметра, називаються значущими типами (value types). У них включені integer, float, boolean і структурні типи Unity (наприклад, Color і Vector3). Типи, які виділяються з купи, після чого доступ до них виходить за допомогою покажчика, називаються посилальними типами, тому що значення, які зберігаються у змінній, тільки "посилаються" на реальні дані. Приклади посилальних типів включають об'єкти, рядки і змінні.

Менеджер пам'яті відстежує зони в купі, які визначені як невикористовувані. При запиті нового блоку пам'яті (припустимо, при створенні екземпляра об'єкту), менеджер вибирає невикористану зону, з якої слід виділити блок, і потім видаляє виділену пам'ять із зони відомого незайнятого простору. Наступні запити обробляються тим же способом, поки у невикористаній зоні буде достатньо місця для виділення блоку необхідного

розміру. Доступ до посилального елементу в купі може бути отриманий тільки поки є посилальні змінні, які можуть його знайти. Якщо усі посилання до блоку пам'яті пропадуть (тобто посилальні змінні були змінені, або вони є локальними змінними, які тепер поза контекстом), то пам'ять, яку він займає, може бути ще раз безпечно виділена.

Щоб визначити, які блоки купи більше не використовуються, менеджер пам'яті переглядає усі активні посилальні змінні і зазначає блоки, до яких вони посилаються як "live" (використовувані). В кінці пошуку, будь-який простір між використовуваними блоками менеджером вважається порожнім, і в майбутньому може бути використаний для виділення. З очевидних причин, процес виявлення і звільнення невикористовуваної пам'яті відомий як приби-рання сміття (Garbage Collection, або GC для скорочення).

Прибирання сміття – це автоматичний і невидимий програмісту процес, але процес складання на ділі вимагає деякого часу "закулісної" роботи процесора. При правильному використанні, автоматичне керування пам'яттю зазвичай не поступається по продуктивності ручному виділенню. Проте, для програміста важливо уникати помилок, які будуть викликати збірку частіше ніж треба і виражатися в затримках роботи.

Ключовою деталлю є те, що нові частини не повинні додаватися до рядка одна за одною. Насправді, в кожній ітерації циклу попереднє утримання змінної "зникає" – виділяється цілий новий рядок для розміщення у ньому оригінальної частини і нової частини в кінці. Оскільки рядок стає довшим, то зі зростаючим значенням змінної циклу, значення споживаного простору купи також підвищується і з легкістю досягає сотні байтів вільного простору купи при кожному виклику цієї функції. Якщо потрібно об'єднати багато рядків разом, то найбільш відповідним варіантом буде клас Mono бібліотеки System.Text.StringBuilder [20], [21][1)].

[1)] [20] Unity – руководство: Понимание автоматического управления памятью. URL: https://docs.unity3d.com/ru/530/Manual/UnderstandingAutomaticMemoryManagement. html (дата звернення 8.12.2019).

[21] Unity – руководство: Платформенно зависимая компиляция. URL: https://docs. unity3d.com/ru/530/Manual/PlatformDependentCompilation.html (дата звернення 8.12.2019).

2 ОБГРУНТУВАННЯ ВИБОРУ ПРОГРАМНИХ ЗАСОБІВ

2.1 Аналіз вибору мови C# для реалізації програмного коду

C# – об’єктно-орієнтована мова програмування із безпечною системою типізації для платформи .NET. Розроблена Андерсом Гейлсбергом, Скотом Вілтамутом та Пітером Гольде під егідою Microsoft Research (при фірмі Microsoft).

Синтаксис C# близький до C++ і Java. Мова має строгу статичну типізацію, підтримує поліморфізм, перевантаження операторів, вказівники на функції-члени класів, атрибути, події, властивості, винятки, коментарі у форматі XML. Перейнявши багато що від своїх попередників –мов C++, Delphi, Модула і Smalltalk– C#, спираючись на практику їхнього використання, виключає деякі моделі, що зарекомендували себе як проблематичні при розробці програмних систем, наприклад множинне спадкування класів (на відміну від C++).

C# розроблялась як мова програмування прикладного рівня для CLR і тому вона залежить, перш за все, від можливостей самої CLR. Це стосується,

перш за все, системи типів C#. Присутність або відсутність тих або інших виразних особливостей мови диктується тим, чи може конкретна мовна особливість бути трансльована у відповідні конструкції CLR. Так, з розвитком CLR від версії 1.1 до 2.0 значно збагатився і сам C#; подібної взаємодії слід чекати і надалі. (Проте ця закономірність буде порушена з виходом C# 3.0, що є розширеннями мови, які не спираються на розширення платформи .NET.) CLR надає C#, як і всім іншим .NET-орієнтованим мовам, багато можливостей, яких позбавлені «класичні» мови програмування. Наприклад, збірка сміття не реалізована в самому C#, а проводиться CLR для програм, написаних на C# точно так, як і це робиться для програм на VB.NET, J# тощо.

Специфікація C# визначає мінімальний набір бібліотек типів і класів, на який має розраховувати компілятор. На практиці, C# найчастіше використовується з якоюсь реалізацією Common Language Infrastructure (CLI), яка стандартизована як ECMA-335 Common Language Infrastructure (CLI).

Титульним компілятором C# є Microsoft Visual C#.

Існують інші компілятори C#, часто вони включають реалізації Common Language Infrastructure і бібліотеки класів .NET.

Проект Microsoft Rotor (який тепер зветься Shared Source Common Language Infrastructure, ліцензований тільки для навчального і дослідницького використання) забезпечує реалізації CLR runtime і компілятор C#, і підмножину бібліотек фреймворка Common Language Infrastructure, відповідно до специфікації ECMA (до C# 2.0, і з підтримкою тільки Windows XP).

Проект SharpDevelop від компанії icsharpcode, який використовується як альтернатива Visual Studio. Забезпечує повну реалізацію Common Language Infrastructure. Остання стабільна версія IDE 4.4 (28 серпня 2013), тестова версія 5.0 (13 лютого 2014). Зовнішній вигляд IDE дуже нагадує

Microsoft Visual C#, що робить комфортним перехід від одного середовища до іншого.

Проект Mono, початий компанією Xamarin і продовжений її покупцем і наступником Novell, забезпечує відкритий компілятор C#, повну відкриту реалізацію Common Language Infrastructure, включаючи потрібні бібліотеки фреймворка відповідно до специфікації ECMA, і близьку до повної реалізацію власницьких бібліотек класів Microsoft .NET до .NET 2.0, але не специфічних бібліотек .NET 3.0 і .NET 3.5, як для Mono 2.0.

Проект DotGNU також надає відкритий компілятор C#, близьку до повної реалізацію Common Language Infrastructure, включаючи потрібні бібліотеки фреймворка відповідно до специфікації ECMA, і підмножину деяких залишених власницьких бібліотек класів Microsoft .NET до .NET 2.0 (які не документовані або не включені у специфікації ECMA, але включені у стандартне визначення Microsoft .NETFramework).

DotNetAnywhere MicroFrameworkCommonLanguageRuntime націлений на вбудовані системи і підтримує майже всі специфікації C# 2.0 [22][1)].

Останнім часом C і C++ є найбільш використовуваними мовами для розробки комерційних і бізнес-програм. Ці мови влаштовують багатьох розробників, але в дійсності не забезпечують належної продуктивності розробки. Наприклад, процес написання програми на C++ часто займає значно більше часу, ніж розробка еквівалентноїпрограми на Visual Basic. Зараз існують мови, що збільшують продуктивність розробки за рахунок втрати гнучкості, яка так звична і необхідна програмістам на C / C++. Подібні рішення є дуже незручними для розробників і часто пропонують значно менші можливості. Ці мови також не орієнтовані на взаємодію із системами, що з'являються сьогодні, і дуже часто вони не відповідають існуючій практиці програмування для Web. Багато розробників хотіли б використовувати сучасну мову, яка дозволяла б писати, читати і супроводжувати програми з простотою Visual Basic і в той же час надавала

[1)] [22] Обзор языка C# – руководство по C# | Microsoft Docs. URL: https:// docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp (дата звернення 9.12.2019).

гнучкість C++, забезпечувала доступ до усіх функціональних можливостей системи, взаємодіялаіз існуючими програмами і легко працювала з виникаючими Web стандартами.

З огляду на всі подібні побажання, Microsoft розробила нову мову – C#. До неї входить багато корисних особливостей –простота, об'єктна орієнтованість, типова захищеність, "збірка сміття", підтримка сумісності версій і багато іншого. Дані можливості дозволяють швидко і легко розробляти програми, особливо COM + програми і Web сервіси. При створенні C#, його автори враховували досягнення багатьох інших мов програмування: C++, C, Java, SmallTalk, Delphi, Visual Basic і т.д. Треба зауважити, що у зв'язку з тим, що C # розроблявся з чистого аркуша, у його авторів була можливість (якою вони явно скористалися) залишити в минулому всі незручні і неприємні особливості (існуючі, як правило, для забезпечення сумісності), будь-якої з попередніх їй мов. В результаті вийшла дійсно проста, зручна і сучасна мова, яка не поступається C++, але істотно підвищує продуктивність розробок.

Дуже часто можна простежити такий зв'язок –чим більше мова захищена і стійка до помилок, тим менше продуктивність програм, написаних на ній. Наприклад, розглянемо дві крайності – Assembler і Java. У першому випадку можна домогтися фантастичної швидкості своєї програми, але доведеться дуже довго змушувати її працювати правильно не на вашому комп'ютері. У випадку ж з Java –виходить захищеність, незалежність від платформи, але, на жаль, швидкість програми навряд чи сумісна з існуючим уявленням про швидкість, наприклад, будь-якої окремої клієнтської програми (звичайно, існують застереження – JIT компіляція та інше). C++ – це співвідношення в швидкості і захищеності близько до бажаного результату, але практично завжди краще понести незначну втрату в продуктивності програми і придбати таку зручну особливість, як "прибирання сміття", яка не тільки звільняє від стомлюючого обов'язку керувати пам'яттю вручну, але і допомагає уникнути багатьох потенційних

помилок в програмі. Насправді, скоро "збірка сміття", та й будь-які інші кроки до усунення потенційних помилок стануть відмінними рисами сучасної мови.

У C#, як і в, поза сумнівом, сучасній мові, також існують характерні особливості для обходу можливих помилок. Наприклад, крім згаданої "збірки сміття", там усі змінні автоматично ініціалізовані середовищем і мають типову захищеність, що дозволяє уникнути невизначених ситуацій у разі, якщо програміст забуде ініціалізувати змінну в об’єкті, або спробує провести неприпустиме перетворення типів. Також в C# були вжиті заходи щодо виключення помилок при оновленні програмного забезпечення. Зміна коду в такій ситуації може непередбачувано змінити суть самої програми. Щоб допомогти розробникам боротися з цією проблемою, C# включає до себе підтримку сумісності версій (vesioning). Зокрема, на відміну від C++ і Java, якщо метод класу був змінений, це повинно бути спеціально обумовлено. Це дозволяє обійти помилки в коді і забезпечити гнучку сумісність версій. Також новою особливістю є native підтримка інтерфейсів і успадкування інтерфейсів. Дані можливості дозволяють розробляти складні системи і розвивати їх з часом.

У C# була уніфікована система типів, тепер можна розглядати кожен тип як об’єкт. Незважаючи на те, як використовувати клас, структуру, масив або вбудований тип, можна звертатися до нього, як до об’єкту. Об’єкти зібрані в просторі імен (namespaces), які дозволяють програмно звертатися до будь-чого. Це означає, що замість списку файлів заголовків, що включаються, в програмі потрібно написати, які простори імен потрібно використовувати, щоб отримати доступ до об’єктів і класів всередині них. У C# вираз using дозволяє не писати кожен раз назву простору імен, коли використовується клас з нього. Наприклад, простір імен System містить декілька класів, у тому числі і Console. І можна писати або назву простору імен перед кожним зверненням до класу, або використовувати using.

Важливою і відмінною від C++ особливістю C# є його простота. Наприклад, програміст може забути, коли пише на C++, де потрібно використовувати "->", де "::", а де ".". Навіть якщо ні, то компілятор завжди поправляє його в разі помилки. Це говорить лише про те, що в дійсності можна обійтися тільки одним оператором, а компілятор сам буде розпізнавати його значення. Так в C#, оператор "->" використовується дуже обмежено (в unsafe блоках), оператор "::" взагалі не існує. Практично завжди використовується тільки оператор ".".

Ще один приклад. При написанні програм на C / C++ доводиться думати не тільки про типи даних, але й про їх розмір в конкретній реалізації. У C# усе спрощено –символ Unicode називається просто char (а не wchar_t, як в C++) і 64-бітове ціле – long (а не __int64). Також в C# немає знакових і беззнакових символьних типів.

У C#, також як і в Visual Basic, після кожного виразу case в блоці switch мається на увазі break. І більш не буде відбуватися помилок, якщо забути поставити break. Однак, якщо потрібно, щоб після одного виразу case програма перейшла до наступного, можна переписати програму із використанням, наприклад, оператора goto.

Багатьом програмістам було не так легко під час вивчення C++ повністю освоїтися із механізмом посилань і покажчиків. У C# немає покажчиків. Насправді, нетривіальність покажчиків відповідала їх корисності. Наприклад, важко собі уявити програмування без покажчиків на функції. Відповідно до цього, в C# присутні Delegates – як прямий аналог покажчика на функцію, але їх відрізняє типова захищеність, безпека і повна відповідність концепціям об’єктно-орієнтованого програмування.

Хотілося б підкреслити сучасну зручність C#. Працюючи з C#, можна побачити, що досить велике значення у ньому мають простори імен. Усі файли заголовків замінені саме простором імен.

У наш час, коли посилюється зв’язок між світом комерції та світом розробки програмного забезпечення, і корпорації витрачають багато зусиль

на планування бізнесу, відчувається необхідність у відповідності абстрактних бізнес-процесів їх програмним реалізаціям. На жаль, більшість мов не мають прямого шляху для зв'язку бізнес-логіки і коду. Наприклад, сьогодні багато програмістів коментують свої програми для пояснення того, які класи реалізують якийсь абстрактний бізнес-об'єкт. C# дозволяє використовувати типізовані метадані, що розгортаються, які можуть бути прикріплені до об'єкту. Архітектурою проекту можуть визначатися локальні атрибути, які будуть пов'язані з будь-якими елементами мови –класами, інтерфейсами і т.п. Розробник може програмно перевірити атрибути будь-якого елементу. Це істотно спрощує роботу. Наприклад, замість того, щоб писати автоматизований інструмент, який буде перевіряти кожен клас або інтерфейс, на те, чи є він дійсно частиною абстрактного бізнес-об'єкту, можна просто скористатися повідомленнями, що засновані на визначених у об'єкті локальних атрибутах.

Свій синтаксис C# багато у чому успадкувала від C++ і Java. Розробники, які мають досвід створення програмного забезпечення на цих мовах, знайдуть в C# багато знайомих рис. Але разом з тим вона є багато у чому й новаторською –атрибути, делегати та події, прекрасно вписані у загальну ідеологію мови, міцно зайняли місце у .NET-розробників. Їх введення дозволило застосовувати принципово нові прийоми програмування.

Об'єктом для порівняння із C# є Java. Вона також розроблена для роботи в віртуальному середовищі виконання, що має об'єктно-орієнтовану архітектуру і «збирач сміття», заснований на механізмі посилань. При порівнянні з цією мовою відразу виділяються такі особливості, як можливість оголошувати кілька класів у одному файлі, з чого слід синтаксична підтримка ієрархічної системи просторів імен. З реалізації ООП-концепцій схожість в механізмі наслідування і реалізації (і в Java, і в C# можливе одиничне успадкування, але множинна реалізація інтерфейсів, на відміну від C++). Але в Java відсутні властивості і індексатори (а також делегати і події). Також є можливість перерахування контейнерів.

Із речей, що включенідо специфікації мови, необхідно відзначити можливість використання коментарів у форматі XML.

Але C# вніс і свої унікальні риси, які вже були згадані – це події, індексатори, атрибути і делагати. Вони являють собою дуже корисні можливості, які не залишаться незатребуваними.

C # і Java – дві мови програмування, які розвивають мову програмування C ++, з синтаксисом, багато в чому схожим на синтаксис C + +, і створених багато в чому в умовах конкуренції, і, внаслідок цього, вони володіють певною схожістю, а також мають ряд відмінностей.

Обидві мови використовують як синтаксичну основу мову програмування C. Зокрема, від неї успадковані без змін:

– позначення початку / кінця блоку коду фігурними дужками;
– позначення, асоціативність і пріоритет більшості вбудованих операцій (привласнення, арифметичні, логічні, побітові операції, операції інкремента / декремента, тернарна умовна операція «?:»);
– синтаксис опису і використання змінних і функцій (порядок «тип ім’я», використання модифікаторів, обов’язковість дужок для функцій, опис формальних параметрів);
– синтаксис всіх основних конструкцій: умовного оператора, циклів, оператора множинного вибору.

Синтаксичні відмінності представлені у табл. 1 [23][1)].

Таблиця 1 – Синтаксичні відмінності мов програмування C# та Java

Синтаксис	Java	C#
1	2	3

[1)] [23] C Sharp | Microsoft вики | Fandom. URL: https://mi-crosoft.fandom.com/ru/ wiki/ C_Sharp (дата звернення 9.12.2019).

Імпорт статичних імен (import static)	Дозволяє окремо імпортувати деякі або всі статичні методи і змінні класу і використовувати їх імена без кваліфікації у імпортуючому модулі	Починаючи з C# 6.0 це було введено (наприклад (using static System.Math))
Оператор switch	Аргумент оператора switch повинен ставитися або до целочисленного, або до перелічуваного типу. Починаючи з версії Java 7 в операторі switch стало можливо використовувати рядкові літерали	Підтримуються як константні типи, так і строкові. Також можливо задавати додаткові умови для блоку case за допомогою ключового слова when. Прямого переходу до наступного блоку case немає. Для переходу до наступного блоку case, потрібно використовувати оператор goto

Продовження таблиці 1

1	2	3
Оператор переходу goto	Від використання goto свідомо відмовилися, однак існує механізм, що дозволяє вийти на зовнішній цикл	goto зберігся, його звичайне використання – це лише передача управління на різні мітки case в операторі switch і вихід з вкладеного циклу
Константи	Констант як таких немає, замість них використовуються статичні змінні класу з модифікатором final	Окреме поняття іменованої типизованої константи і ключове слово const
Точність обчислень з	Java містить конструкцію strictfp, що	C# покладається на реалізацію, гарантії строго однакових

плаваючою точкою	гарантує однакові результати операцій з плаваючою точкою на всіх платформах	результатів обчислень немає
Відключення перевірок	У Java всі динамічні перевірки включаються / вимикаються тільки на рівні пакета	C# містить конструкції checked і unchecked, що дозволяють локально включати і вимикати динамічну перевірку арифметичного переповнення.

2.2 Аналіз вибору MonoDevelop як середовища розробки програмного коду

Програмування здійснюється не всередині Unity 3D, код існує у вигляді окремих файлів, місце розташування яких повідомляється Unity 3D. Файли сценаріїв можуть створюватися в програмі Unity 3D, але в будь-якому випадку буде потрібний текстовий редактор або IDE, де буде писатися код для цих, спочатку порожніх, файлів. У комплекті з Unity 3D постачається програма MonoDevelop, як міжплатформове інтегроване середовище розробки (IDE) для мови C # з відкритим вихідним кодом.

Unity 3D використовує Mono як основу для програмування, і саме середовище MonoDevelop забезпечує міжплатформові можливості усього процесу розробки.

MonoDevelop–відкрите інтегроване середовище розробки для платформ Linux, MacOS та MicrosoftWindows, передусім націлене на розробку програм, які використовують і Mono, і Microsoft .NETframework. На даний момент підтримуються мови C#, Java, Boo, VisualBasic.NET, CIL, Python, Vala, C та

C++. Також MonoDevelop підтримує такі технології, як Gtk#, ASP.NETMVC, Silverlight, MonoMac и MonoTouch.

MonoDevelop включає можливості подібні до NetBeans та MicrosoftVisualStudio, такі як автоматичне доповнення, інтеграція контролю коду, графічний користувацький інтерфейс і веб-дизайнер. В MonoDevelop інтегрований Gtk# GUI дизайнер під назвою Stetic.

MonoDevelop також може використовуватися на платформах Windows та MacOS. Щоправда, до версії 2.2 жодна з даних платформ не мала офіційної підтримки. Тим не менше, жодна з них не підтримується настільки добре, як Linux-версія. MonoDevelop постачається разом з інсталятором Mono для MacOS, але ця версія не включає до себе дизайнер графічного інтерфейсу користувача SteticGUIdesigner через проблеми Drag&Drop у нативній для MacOS версії GTK. MonoDevelop на FreeBSD найвірогідніше підтримується лише зусиллями FreeBSD спільноти.

Потрібно пам’ятати, що у програмі MonoDevelop код тільки пишеться, але не запускається. Це середовище розробки –добре оснащений текстовий редактор, а для того щоб відтворити код слід натсинути кнопку Play у програмі Unity 3D [24][1)].

У MonoDevelop багато можливостей, таких як підсвітка синтаксису, згортання та автодоповнення коду, вбудований відладчик та візуальний конструктор форм

Підсвітка синтаксису –виділення певним чином (зазвичай кольором), певних елементів тексту (лексем), для покращення сприйняття його вмісту. Зазвичай застосовується до кодів програм та розмітки документів. Розмітка синтаксису підтримується багатьма текстовими редакторами та деякими сайтами, що публікують код.

При підсвічуванні синтаксису в початковому тексті мов програмування виділяються:

– конструкції мови;

[1)] [24] Unity – руководство: MonoDevelop. URL: https://docs.unity3d.com/ru/530/ Manual/MonoDevelop.html (дата звернення 9.12.2019).

– коментарі;

– числові і рядкові дані.

Більш просунуті системи підсвічування синтаксису також виділяють:

– змінні;

– дужки;

– стандартні функції мови.

Багато текстових редакторів і середовища розробки мають також функцію підсвічування парних дужок під курсором: при наближенні текстового курсора до скобці виділяється як дужка, біля якої знаходиться курсор, так і парна їй.

Підсвічування синтаксису, крім зручності читання, дозволяє уникати також синтаксичних помилок: неправильного написання конструкцій мови, незакритих лапок і т.д. Може бути присутнім також підсвічування свідомо некоректного коду: наприклад, непарних дужок, або неприпустимих символів поза строкових даних в лапках.

Згортання, або фолдінг (англ. Folding) – одна з функцій текстового редактора, що дозволяє приховувати певний фрагмент редагованого коду або тексту, залишаючи лише один рядок. В якості таких фрагментів зазвичай виступають логічно цілісні фрагменти коду програми, наприклад, функція, клас, цикл і т. п., або фрагменти тексту, наприклад, абзац, глава, секція.

Наприклад, фолдінг функції призводить до згортання всього коду функції в один рядок таким чином, що буде видно тільки назву функції.

Зазвичай, щоб згорнути фрагмент, потрібно натиснути на символ «-» зліва від нього. Щоб побачити весь фрагмент, тобто розгорнути його, потрібно натиснути на символ «+», що з’являється у згорнутих фрагментів.

Практично всі IDE і переважна більшість текстових редакторів, призначених для редагування вихідного коду, підтримують згортання.

Автодоповнення (англ. Autocomplete) –функція в програмах, які передбачають інтерактивне введення тексту (редактори, оболонкикомандної строки, браузери и т. п.) по доповненню тексту по введеній його частині.

Командний інтерпретатор cmd.exe підтримує автодоповнення після натискання клавіші Tab шляхом перебору імен файлів і піддиректорій поточної директорії. Автодоповнення команд та імен виконуваних файлів, що знаходяться за межами поточної директорії, не підтримується. Імена, що підставляються, замінюють вже введені або підставлені. Якщо ім’я файлу або директорії містить пробіл, то підстановка цього варіанту відбувається з лапками.

Вбудований відладчик, який проводить налагодження –етап розробки комп’ютерної програми, на якому виявляють, локалізують і усувають помилки. Щоб зрозуміти, де виникла помилка, доводиться:

– дізнаватися поточні значення змінних;

– з’ясовувати, яким шляхом виконувалася програма.

Існують дві взаємодоповнюючі технології налагодження.

По-перше використання відладчиків – програм, які включають до себе користувальницький інтерфейс для покрокового виконання програми: оператор за оператором, функція за функцією, із зупинками на деяких рядках вихідного коду або при досягненні певної умови;

По-друге висновок поточного стану програми за допомогою розташованих в критичних точках програми операторів виведення – на екран, принтер, гучномовець або в файл. Висновок налагоджувальних відомостей в файл називається журналюванням.

Відладчик це програмний інструмент, що дозволяє програмісту спостерігати за виконанням досліджуваної програми, зупиняти і перезапускати її, проганяти в уповільненому темпі, змінювати значення в пам’яті і навіть, в деяких випадках, повертати назад по часу.

Також корисними інструментами в руках програміста можуть виявитися:

– профілювальники – дозволять визначити, скільки часу виконується та чи інша ділянка коду. Аналіз покриття дозволяє виявити ділянки коду, що є невиконуваними;

– API логгери дозволяють відстежити взаємодію програми і Windows API за допомогою запису повідомлень Windows у лог;
– дизасемблери дозволяють подивитися асемблерний код виконуваного файлу;
– сніффери допоможуть відстежити мережевий трафік, що генерується програмою;
– сніффери апаратних інтерфейсів дозволяють побачити дані, якими обмінюються система і пристрій;
– список системи.

Використання мов програмування високого рівня зазвичай спрощує налагодження, якщо такі мови містять, наприклад, засоби обробки винятків, сильно полегшують пошук джерела проблеми.

У програмному коді може бути так звана недокументована поведінка – серйозні помилки, які не виявляються при нормальному ході виконання програми, проте вельми небезпечні для безпеки всієї системи в разі цілеспрямованої атаки. В даному випадку завдання налагодження це:

– виявлення недокументованої поведінки системи;
– усунення небезпечного коду.

Виділяють такі методи відладки:

– статичний аналіз коду – на цій фазі програма-сканер шукає послідовності у початковому тексті, відповідні небезпечним викликам функцій і т. п. Фактично йде сканування вихідного тексту програми на основі спеціальної бази правил, яка містить опис небезпечних зразків коду.
– фаззінг – це процес подачі на вхід програми випадкових або некоректних даних і аналіз реакції програми.
– reverse engineering (Зворотна інженерія) – цей випадок виникає, коли незалежні дослідники шукають уразливості і недокументовані можливості програми.

Візуальний конструктор форм (GTK#) GTK+ (від The GIMP ToolKit) – кросплатформовий набір інструментів для створення графічних інтерфейсів користувача. Разом із Qt є одним із найпопулярніших інструментів для X Window System.

До складу тулкіта входить повний набір віджетів, що дозволяють використовувати GTK+ для проектів різного рівня і розміру. GTK+ спеціально спроектований для підтримки не тільки C/C++, але й інших мов програмування, таких як Perl і Python, що в поєднанні з використанням візуального будівника інтерфейсу Glade дозволяє істотно спростити розробку і скоротити час написання графічних інтерфейсів. Організація виводу в GTK+ абстрагована від типу віконних систем, наприклад, поставляється бекенд, що забезпечує можливість роботи поверх дисплейного сервера Wayland, а також бекенд, котрий дозволяє здійснювати виведення бібліотеки GTK+ у вікні веб-браузера (запустивши GTK-додаток на одній машині, можна відкрити браузер на іншій машині і отримати доступ до інтерфейсу цієї програми).

Переваги середовища розробки MonoDevelop у збільшенні продуктивності, нової віртуальної папці ресурсів для приведення проектів у порядок і підтримці архівації MonoTouch і MonoMac проектів. Серед інших переваг варто відзначити поліпшену підтримку систем контролю версій, фреймворків ASP.NETMVC2 і MVC3, а також Retina-дисплеїв. Крім того середовище забезпечене більш надійною системою розгортання Android-додатків [25][1)].

[1)] [25] MonoDevelop – MonoDevelop – qwertyu.wiki. URL: https://ru.qwertyu.wiki/wiki/MonoDevelop (дата звернення 9.12.2019).

3 РОЗРОБКА ІГРОВИХ МЕХАНІК

У грі Dino’s Adventures розроблено безліч різних ігрових механік, що полягають у поведінці персонажів та ігрових об’єктів. За будь-яку поведінку об’єктів відповідають ділянки програмного коду у скриптах. Поведінкою об’єктів може бути їх рух, взаємодія з іншими об’єктами, взаємодія з гравцем і т. ін.

3.1 Опис пересування

Функція Physics2D.OverlapCircle перевіряє місце розташування головного персонажа гри. Функція змінює значення, якщо персонаж знаходиться на землі або в повітрі:

```
grounded=Physics2D.OverlapCircle (groundCheck.position, groundRadius, whatIsGround);
```

Функція Rigidbody2D.AddForce використовується для завдання імпульсу руху об’єкту, а саме додає крутний момент до фізичного тіла. Сила при-кладається безперервно вздовж напрямку вектора сили. Завдання режиму ForceMode дозволяє змінити тип сили на прискорення, імпульс або

зміну швидкості. Сила може бути застосована тільки до активного твердого тіла. Якщо GameObject неактивний, AddForce не діє. Приклад:

```
GetComponent <Rigidbody2D> (). AddForce (new Vector2 (0f, jumpForce));
```

Функція GetComponent знаходить елемент під ім'ям Rigidbody2D у об'єкта, до якого прив'язаний скрипт і змінює значення функції AddForce.

За допомогою функції Rigidbody2D. velocity можна задавати вектор швидкості твердого тіла:

```
GetComponent<Rigidbody2D>().velocity = new Vector2 (move * maxSpeed * 2f, GetComponent<Rigidbody2D>().velocity.y);
```

Компонент Transform визначає Position (положення), Rotation (обертання), і Scale (масштаб) кожного об'єкта в сцені. У кожного GameObject'а є Transform (рис. 4).

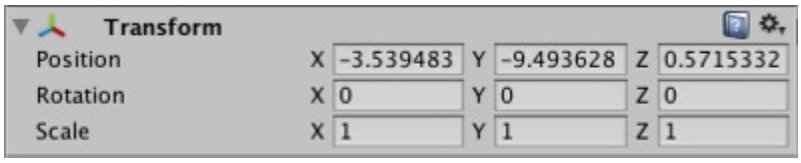


Рисунок 4 – Компонент Transform

Властивості компонента Transform:

- Position: Положення Transform в координатах X, Y, і Z.
- Rotation: Обертання Transform навколо осей X, Y, і Z, вимірюється в градусах.
- Scale: Масштаб Transform'а уздовж осей X, Y, і Z. Значення "1" означає оригінальний розмір (розмір, з яким був імпортований об'єкт).

Значення положення, обертання і масштабу Transform'а вимірюються щодо його батька. Якщо у Transform'а немає батька, тоді всі ці властивості вимірюються в світовому просторі.

Приклад:

```
Vector3 theScale = transform.localScale;
```

Частина коду, що відповідає за перевірку знаходження головного персонажа гри на землі і задає силу стрибка:

```
if (col.gameObject.name == "best" || col.gameObject.name == "ground" ||
col.gameObject.name == "ground2"  || col.gameObject.name == "kol" ||
col.gameObject.name == "tel" || col.gameObject.name == "GameObject") {
                jumpForce = 930f;}
```

Функція, що відповідає за стрибок:

```
public void Jump(bool isJump)
   {
            isJump = groundCheck;
            if (groundCheck)
            {
                    GetComponent<Rigidbody2D>  ().AddForce (new Vector2
(0f, jumpForce));
                    jumpForce=0;
            }
   }
```

Функція, що відповідає за пересування по землі:

```
public void Move(int move1)
   {
            move = move1;
            GetComponent<Rigidbody2D>().velocity = new Vector2 (move1 *
maxSpeed, GetComponent<Rigidbody2D>().velocity.y);
   }
```

3.2 Опис взаємодії об’єктів між собою

Функція OnCollisionEnter2D відповідає за зіткнення об’єктів між собою. Ця функція спрацьовує у разі, якщо коллайдер одного об’єкту стикається з колайдером іншого об’єкту або перетинає його.

Коллайдер – це компонент об’єкту, що не відображається, який складає фізичну присутність по відношенню до інших коллайдерів, тобто, тіла, що мають коллайдери, взаємодіють між собою фізично. Наприклад, персонаж не може пройти крізь стіну. Якщо на стінах не буде коллайдера, то можна проникати крізь модель. Коллайдери бувають різні за формою. Базові це: куб, сфера, капсула.

Частина коду, що відповідає за взаємодію коллайдерів:

```
	void OnCollisionEnter2D(Collision2D col){
	if (col.gameObject.name == "out")
	out1 = true;
	if (col.gameObject.name == "best" || col.gameObject.name == "ground" ||
col.gameObject.name == "ground2"  || col.gameObject.name == "kol" ||
col.gameObject.name == "tel" || col.gameObject.name == "GameObject") {
	jumpForce = 930f;
			}
	}
```

Також коллайдери можуть перетинатися із тригерами. Функція OnTriggerEnter викликається, коли коллайдер входить у тригер. Функція OnTriggerEnter викликається, коли Collider other виходить із тригера. Код, який буде розміщений в цих функціях, буде виконаний один раз.

Приклад коду, який відповідає за перетин коллайдера з тригером:

```
	void OnTriggerEnter2D (Collider2D col){
			if (col.gameObject.name == "star") {
				score++;
				for_star_trig = true;
			}
```

```
        if (col.gameObject.name == "dead") {
                for_hp.dino_hp --;
                for_dead_trig = true;
        }
    }
```

Для взаємодії із гравцем існують функції, що зчитують натискання клавіш клавіатури і миші. Функція Input.GetKey буде повторно повертати true поки користувач утримує зазначену клавішу. Це можна використовувати для багаторазового запуску певних дій і функцій. Функція Input.GetKeyDown буде спрацьовувати тільки один раз при натисканні зазначеної клавіші. Це ключова відмінність між Input.GetKey і Input.GetKeyDown. Одним із прикладів її використання є включення призначеного для користувача інтерфейсу. Функція Input.GetKeyUp – це повна протилежність Input.GetKeyDown. Вона використовується при відпуску зазначеної клавіші. Як і Input.GetKeyDown, вона повертає true тільки один раз.

Приклад використання функції Input.GetKey:

```
if (Input.GetKey(KeyCode.D))
        {
                move=1;
                animator.SetFloat ("Speed", 0);
        }
```

3.3 Розробка основних скриптів

3.3.1 Розробка скриптів для типу гри Catch

Блок схема поведінки об’єктів для типу гри Catch представлена у додатку А.1. Виходячи з блок-схеми, в першу чергу повинні випадковим чином згенеруватися позиція та тип об’єкту, який буде падати зверху, потім відразу активується таймер зворотнього відліку, після закінчення якого буде згенеровано наступний об’єкт. Після того, як об’єкт згенерований, він

з’являється зверху сцени, за межами камери гравця, потім об’єкт починає програвати анімацію падіння і гравець може з ним взаємодіяти. Якщо гравець не «зловив» об’єкт, то об’єкт, зіткнувшися із колайдером землі, знищується зі сцени. Якщо ж гравець «спіймав» об’єкт, а саме коллайдер об’єкту зіткнувся із колайдером головного персонажу гри, то, в залежності від типу об’єкта, відбуваються такі дії:

– якщо об’єкт типу Star, то до кількості зароблених очок буде додана одиниця;
– якщо об’єкт типу Saw або Dead, то з кількості життів головного персонажа відніметься одиниця;
– якщо об’єкт типу HP, то до кількості життів головного персонажа буде додана одиниця.

Далі будуть розглянуті скрипти, призначені для реалізації механіки типу гри Catch.

Скрипт Anim_dead_saw призначений для запуску анімації обертання пилки, а також для тимчасової зупинки віднімання життів у головного персонажу:

```
public float not_dead_time = 0;
    public GameObject saw;
    public AnimationClip dead;
    // Use this for initialization
    void Start () {
    }
    // Update is called once per frame
    void Update () {
                GetComponent<Animation> ().Play (dead.name);
                if (not_dead_time > 0) {
                        not_dead_time -= Time.deltaTime;
                } else {
                        not_dead_time = 0;
                        saw.tag = "Dead";
                                }
    }
    void OnTriggerEnter2D (Collider2D col){
                if (col.gameObject.tag == "Dino") {
                        if(not_dead_time == 0){
                                not_dead_time = 2;
                                saw.tag = "Untagged";
```

```
                }
            }
        }
```

Спочатку оголошуються змінні not_dead_time, saw і dead. Змінна not_dead_time відповідає за кількість часу, протягом якого у головного персонажу не буде відніматися життя. Змінна saw буде містити в собі об’єкт типу Saw. Змінна dead буде містити в собі анімацію обертання пилки.

В першу чергу при запуску цього скрипта буде запускатися анімація обертання пилки. За допомогою функції GetComponent <Animation> (), скрипт отримує доступ до компоненту Animation об’єкта, до якого він прив’язаний. Функція Play (dead.name) запускає анімацію, яка знаходиться у змінній dead.

Потім відбувається перевірка – якщо час not_dead_time більше ніж 0, то він починає зменшуватися за допомогою функції Time.deltaTime, яка відповідає за час у грі. Якщо ж час менше ніж 0, то змінна not_dead_time стає рівною 0, а також об’єкту, який знаходиться у змінній saw, задається тег "Dead". Цей тег відповідає за те, що при зіткненні головного персонажу з об’єктом, у якого є цей тег, у головного персонажу віднімається одне життя.

Функція void OnTriggerEnter2D (Collider2D col) відповідає за перевірку зіткнення коллайдера поточного об’єкту із колайдерами інших об’єктів. У змінну col типу Collider2D передається об’єкт, з яким було зіткнення. У цій функції відбувається перевірка – якщо об’єкт, з яким було зіткнення має тег "Dino", то запускається час, протягом якого у головного персонажу не буде відніматися життя, а також об’єкту у змінній saw присвоюється тег "Untagged".

Скрипт Catch_win відповідає за перевірку кількості набраних очок, якщо кількість набраних очок дорівнюватиме певному значенню, рівень буде пройдений:

```
public int score_win = 20;
    public For_catch_game game;
```

```
// Use this for initialization
void Start () {
}
// Update is called once per frame
void Update () {
        if (game.score >= score_win) {
                Application.LoadLevel("DinoEXT");
        }
}
```

Оголошуються дві змінні – змінна score_win, що відповідає за кількість очок, потрібних для проходження рівню, та змінна game типу скрипта For_catch_game, за допомогою якої можна отримати інформацію про скрипт For_catch_game, а також змінювати значення змінних цього скрипта.

При запуску скрипта відбувається перевірка, в якій, за допомогою змінної game, отримується значення змінної score з скрипта For_catch_game, після чого це значення порівнюється зі значенням змінної score_win. Якщо ці значення рівні або значення змінної score більше, то за допомогою функції Application.LoadLevel здійснюється перехід на сцену перемоги в рівні та гра закінчується.

Скрипт Corroz_ground відповідає за руйнування землі під головним персонажем на одному з рівнів гри Catch.

Руйнування землі відбувається в 3 етапи. На першому етапі об’єкт землі замінюється об’єктом потрісканої землі, що попереджає гравця про те, що цю ділянку землі необхідно покинути. На другому етапі об’єкт потрісканої землі зникає з ігрової карти і, якщо гравець в цей час стояв на цьому об’єкті, то він провалиться, і гра закінчиться, рівень буде провалений. На третьому етапі об’єкт землі знову з’являється на карті.

Ділянка коду, яка відповідає за перший етап:

```
if(go_and_ruin == 1){
                        time+= Time.deltaTime;
                        if(time >= corroz){
                                rand_ter = (int)Random.Range (1, 4);
        if(rand_ter == 1)corroz_ter1.SetActive(true);
        if(rand_ter == 2)corroz_ter2.SetActive(true);
```

```
                    if(rand_ter == 3){corroz_ter1.SetActive(true);
corroz_ter2.SetActive(true);}
                                        time = 0;
                                        go_and_ruin = 2;
                                }
                    }
```

Спочатку відбувається перевірка етапу. Змінна go_and_ruin відповідає за номер етапу. Після перевірки запускається зворотний відлік до переходу на наступний етап за допомогою функції Time.deltaTime. Потім відбувається наступна перевірка – якщо час вийшов, то випадково вибирається одна з трьох дій. Перша дія – перший об'єкт землі стає потрісканим, друга дія – другий об'єкт стає потрісканим, третя дія – обидва об'єкта землі стають потрісканими. Функція corroz_ter2.SetActive(true) вмикає об'єкт потрісканої землі, де corroz_ter2 – змінна, яка зберігає в собі об'єкт землі. Після цього змінна time, призначена для зберігання часу, скидається в 0, а змінна go_and_ruin стає рівною 2, що означає перехід на наступний етап.

Ділянка коду, яка відповідає за другий етап:

```
    if(go_and_ruin == 2){
                        time+= Time.deltaTime;
                        if(time >= ruin){
                        if(rand_ter == 1)ter1.SetActive(false);
                        if(rand_ter == 2)ter2.SetActive(false);
                        if(rand_ter == 3){ter1.SetActive(false);
ter2.SetActive(false);}
                                        time = 0;
                                        go_and_ruin = 3;
                                }
                    }
```

На цьому етапі також починається відлік часу до наступного етапу. Коли змінна time стає більше або дорівнює змінній ruin, відбувається перевірка – який з об'єктів землі відключати, або відключити обидва. Потім відбувається перехід на третій, завершальний етап.

Ділянка коду, який відповідає за третій етап:

```
if(go_and_ruin == 3){
                time+= Time.deltaTime;
                if(time >= empt){
                        ter1.SetActive(true);
                        ter2.SetActive(true);
                        corroz_ter1.SetActive(false);
                        corroz_ter2.SetActive(false);
                        time = 0;
                        go_and_ruin = 1;
                }
        }
```

На третьому етапі після відліку часу відбувається включення всіх об’єктів землі, а також виключення об’єктів потрісканої землі (SetActive(true) – відповідає за включення ігрового об’єкту, SetActive(false) – відповідає за вимкнення ігрового об’єкту). Після чого відбувається перехід назад на перший етап.

Скрипт For_nice_catch відповідає за зміну швидкості падіння об’єктів:

```
public float an_sp = 10;
    public GameObject dino;
    public AnimationClip obj_anm;
    public Animation anm;
    void Start () {
            dino = GameObject.FindGameObjectWithTag("Dino");
    }

    void Update () {
            anm[obj_anm.name].speed = dino.GetComponent <Speed>().sp;
            GetComponent<Animation> ().Play (obj_anm.name);
    }
    void OnTriggerEnter2D (Collider2D cold){
            if (cold.gameObject.name == "Dino") {
                    Destroy(gameObject);
            }
            if (cold.gameObject.name == "trigg") {
                    Destroy(gameObject);
            }
    }
```

Створюються такі змінні, як dino, в якій буде зберігатися об’єкт головного персонажу, obj_anm типу AnimationClip, в якому буде знаходитися сама анімація, а також anm, в якому знаходиться компонент Animation. При

запуску гри в функції Start відбувається прив’язка об’єкту головного персонажу до змінної dino за допомогою функції GameObject.FindGameObjectWithTag("Dino"). Ця функція дозволяє здійснити пошук серед усіх ігрових об’єктів на поточній сцені і знайти об’єкт, у якого тег буде дорівнювати "Dino". На відміну від функції GameObject.FindGameObjectsWithTag, яка знаходить усі ігрові об’єкти з певним тегом і може помістити їх у масив, функція GameObject.FindGameObjectWithTag знаходить перший ігровий об’єкт у списку і поміщає його в змінну типу GameObject. Потім в функції Update вибирається компонент Animation з анімацією, яка знаходиться у змінній obj_anm, після чого за допомогою функції dino.GetComponent<Speed>().sp виходить змінна sp, яка знаходиться в скрипті Speed, який приєднаний до головного персонажу, і швидкість анімації стає рівною значенню змінної sp.

Рядок коду GetComponent<Animation>().Play(obj_anm.name) відповідає за запуск анімації падіння.

У функції voidOnTriggerEnter2D(Collider2Dcold) відбуваються дві перевірки – якщо падаючий об’єкт стикається з головним персонажем або з землею. В обох випадках об’єкт знищується. За знищення об’єкту відповідає функція Destroy (gameObject), яка знищує об’єкт, до якого прив’язаний даний скрипт.

Скрипт For_nice_dead відповідає за поворот об’єкту dead:

```
void Start () {
        gameObject.transform.rotation = Quaternion.Euler (0, 0, 180);
  }
```

При старті скрипта визначається поточний поворот елемента за допомогою функції gameObject.transform.rotation. За допомогою функції Quaternion.Euler(0, 0, 180) елемент повертається на 180 градусів.

Скрипт Ghost відповідає за поведінку об’єктів Ghost на деяких рівнях гри Catch:

```
public float not_dead_time = 0;
    public HP hp;
    // Use this for initialization
    void Start () {
    }
    // Update is called once per frame
    void Update () {
            if (not_dead_time > 0) {
                    not_dead_time -= Time.deltaTime;
            } else {
                    not_dead_time = 0;
            }
    }
    void OnTriggerEnter2D (Collider2D col){
            if (col.gameObject.tag == "Dino") {
                    if(not_dead_time == 0){
                            hp.dino_hp --;
                            not_dead_time = 2;
                    }
            }
    }
```

Змінна not_dead_time відповідає за кількість часу, протягом якого головний персонаж не буде втрачати життя, а змінна hp типу HP містить в собі дані скрипта HP. Якщо об’єкт Ghost стикається з об’єктом головного персонажу, то за допомогою змінної hp змінюється змінна dino_hp в скрипті HP, яка відповідає за кількість життів головного персонажу. Після отримання шкоди головним персонажем, об’єкт Ghost не може завдавати шкоди протягом двох секунд ігрового часу.

Скрипт ghost_on призначений для активування об’єктів типу Ghost після певного часу:

```
public float spawn = 0;
public GameObject ghost;
void Start () {
            ghost.SetActive(false);
    }
    // Update is called once per frame
    void Update () {
            if (spawn >= 2) {
                    ghost.SetActive (true);
            } else {
```

```
                spawn += Time.deltaTime;
            }
        }
```

Змінна spawn відповідає за час, після якого об’єкт Ghost стане активним. Змінна ghost буде містити в собі сам об’єкт Ghost.

На початку скрипта об’єкт Ghost відключений. Включається таймер, по завершенню двох секунд ігрового часу об’єкт Ghost включається.

Об’єкти Ghost протягом усієї гри зникають і з’являються. Даний скрипт був створений для того, щоб різні об’єкти Ghost зникали і з’являлися в різний час.

Скрипт Go_catch один з найважливіших скриптів гри Catch. Він відповідає за появу падаючих ігрових об’єктів. Вибір типу і позиції об’єкту обирається випадково:

```
    void Update () {
            if (numb_obj == 1)
            {
                child = Instantiate(str, new Vector3 (pos.transform.posi-
tion.x, pos.transform.position.y, pos.transform.position.z), Quaternion.identity) as
GameObject;
            }
            if (numb_obj == 2)
            {
                child = Instantiate(ship, new Vector3 (pos.transform.posi-
tion.x, pos.transform.position.y, pos.transform.position.z), Quaternion.identity) as
GameObject;
            }
            if (numb_obj == 3)
            {
                child = Instantiate(saw, new Vector3 (pos.transform.posi-
tion.x, pos.transform.position.y, pos.transform.position.z), Quaternion.identity) as
GameObject;
            }
            if (numb_obj == 4)
            {
                child = Instantiate(hp, new Vector3 (pos.transform.posi-
tion.x, pos.transform.position.y, pos.transform.position.z), Quaternion.identity) as
GameObject;
            }
            if (numb_obj != 0) {
                child.transform.SetParent (pos.transform);
```

```
                    child.GetComponent<For_nice_catch> ().an_sp =
anim_speed;}
            numb_obj = 0;}
```

Спочатку визначається тип об’єкту, який визначає інший скрипт під назвою Obj_random. Потім створюється сам об’єкт за допомогою функції Instantiate. Функція Instantiate включає до себе три параметри. Перший параметр – це сам об’єкт, який потрібно додати на ігрову карту. Другий параметр – це позиція об’єкту, яка задається за допомогою функції newVector3 із координатами x, y, z. В даному скрипті беруться координати поточного об’єкту, до якого прив’язаний скрипт за допомогою функції pos.transform.position. Третій параметр відповідає за поворот об’єкту, функція Quaternion.identity дозволяє встановити поворот об’єкту таким, який він є в самому об’єкті. Після створення об’єкту йому присвоюється батьківський об’єкт за допомогою функції child.transform.SetParent(pos.transform). Це зроблено для того, щоб кожен об’єкт програвав анімацію з тієї позиції, на якій він з’явився. Child.GetComponent <For_nice_catch> () .an_sp = anim_speed задає кожному об’єкту, який з’явився, швидкість анімації, яка визначена змінною anim_speed.

Скрипт HP відповідає за кількість життів головного пресонажу:

```
void Update () {
            if (dino_hp > 3) dino_hp = 3;
            if (dino_hp < 3) {
                    hp3.SetActive(false);
            } else hp3.SetActive(true);
            if (dino_hp < 2) {
                    hp2.SetActive(false);
            } else hp2.SetActive(true);
            if (dino_hp < 1) {
                    hp1.SetActive(false);
            } else hp1.SetActive(true);
            if (dino_hp < 0) {
                    Application.LoadLevel("DinoDead");
            }
    }
```

Змінна dino_hp зберігає у собі кількість життів головного персонажу. Перша перевірка в скрипті потрібна для того, щоб кількість життів не була більшою, ніж три. Якщо кількість життів дорівнює двом, то за допомогою функції SetActive (false) відключається об’єкт, який відображає перше очко життя. Якщо кількість життів дорівнює одному, відключається друге очко, якщо кількість життів дорівнює нулю – відключається третє. Після того як змінна dino_hp стає менше нуля, відбувається перехід на сцену програшу і гра завершується.

Скрипт Obj_random другий за важливістю скрипт для гри Catch, він відповідає за випадкове визначення типу падаючого об’єкту, а також за його випадкову позицію на ігровій карті.

Перша частина скрипта:

```
rand_wall = (int)Random.Range (1, 101);
                        if(rand_wall <= 20 && wall_stop == 0){
                                rand_obj = (int)Random.Range (1, 8);
                        if(rand_obj == 1){str1.numb_obj = 1;} else{str1.numb_obj =
3;}
                        if(rand_obj == 2){str2.numb_obj = 1;} else{str2.numb_obj =
3;}
                        if(rand_obj == 3){str3.numb_obj = 1;} else{str3.numb_obj =
3;}
                        if(rand_obj == 4){str4.numb_obj = 1;} else{str4.numb_obj =
3;}
                        if(rand_obj == 5){str5.numb_obj = 1;} else{str5.numb_obj =
3;}
                        if(rand_obj == 6){str6.numb_obj = 1;} else{str6.numb_obj =
3;}
                        if(rand_obj == 7){str7.numb_obj = 1;} else{str7.numb_obj =
3;}
                        wall_stop = 3;}
```

Змінна rand_wall відповідає за шанс появи «стіни» з падаючих об’єктів. Під «стіною» мається на увазі, що з усіх позицій з’являться об’єкти Saw, і лише з однієї позиції з’явиться об’єкт Star.

У змінну rand_wall за допомогою функції (int)Random.Range(1, 101) записується випадкове цілочисельне значення від 1 до 100. Далі відбувається перевірка – якщо у змінній rand_wall буде число від 1 до 20 (з ймовірністю

20%), і якщо змінна wall_stop буде дорівнювати нулю (змінна wall_stop відповідає за час, протягом якого падіння наступних об’єктів буде призупинено), то «стіна» з’явиться на ігровій карті. Після перевірки в змінну rand_obj записується випадкове число від 1 до 8. Ця змінна відповідає за номер позиції, на якій з’явиться об’єкт Star. Наступні умови відповідають за перевірку кожної позиції і визначають – з’явиться на цій позиції об’єкт Star чи об’єкт Saw. Змінні str1 – str7 зберігають у собі координати можливих позицій.

Друга частина скрипта:

```
else{
rand_pos = (int)Random.Range (1, 8);
if(rand_pos == 1) {str1.numb_obj = (int)Random.Range (1, obj_count);}
if(rand_pos == 2) {str2.numb_obj = (int)Random.Range (1, obj_count);}
if(rand_pos == 3) {str3.numb_obj = (int)Random.Range (1, obj_count);}
if(rand_pos == 4) {str4.numb_obj = (int)Random.Range (1, obj_count);}
if(rand_pos == 5) {str5.numb_obj = (int)Random.Range (1, obj_count);}
if(rand_pos == 6) {str6.numb_obj = (int)Random.Range (1, obj_count);}
if(rand_pos == 7) {str7.numb_obj = (int)Random.Range (1, obj_count);}
}
if(wall_stop > 0){wall_stop--;}
rand_time = 0;
```

Ця частина скрипта виконується в тому випадку, якщо змінна rand_wall містить у собі число більше 20. Мінлива rand_pos відповідає за позицію появи одиночного об’єкту, в неї записується випадкове число від 1 до 8. Потім на обраній позиції з’являється випадковий об’єкт. Мінлива obj_count зберігає у собі кількість типів об’єктів.

Скрипт Saw_all_catch відповідає за випадкову появу об’єктів на рівні гри Catch, в якому зверху завжди падають «стіни»:

```
if (rand_time >= 1)
            {
if(rand_obj == 1){str1.numb_obj = 1;}    else{str1.numb_obj = 3;}
if(rand_obj == 2){str2.numb_obj = 1;}    else{str2.numb_obj = 3;}
if(rand_obj == 3){str3.numb_obj = 1;}    else{str3.numb_obj = 3;}
if(rand_obj == 4){str4.numb_obj = 1;}    else{str4.numb_obj = 3;}
if(rand_obj == 5){str5.numb_obj = 1;}    else{str5.numb_obj = 3;}
```

```
if(rand_obj == 6){str6.numb_obj = 1;}     else{str6.numb_obj = 3;}
if(rand_obj == 7){str7.numb_obj = 1;}     else{str7.numb_obj = 3;}
if(rand_obj == 1){
   rand_obj ++;
}else if(rand_obj == 7){
   rand_obj --;
}else{
   rand_plus_min = (int)Random.Range (1, 3);
if(rand_plus_min == 1)rand_obj ++;
if(rand_plus_min == 2)rand_obj --;
}
rand_time = 0;
}
```

У першій частині скрипта, так само як і в скрипті Obj_random, визначається одна позиція для об’єкта Star, а на всіх інших позиціях з’являться об’єкти Saw. Потім відбувається кілька перевірок, які відповідають за те, щоб у кожній наступній «стіні» позиція об’єкту Star відрізнялася на одиницю. Таким чином, якщо об’єкт Star на першій позиції, його позиція в наступній «стіні» буде на одиницю більше, якщо об’єкт Star на останній позиції, на наступній «стіні» позиція буде на одиницю менше. Це зроблено для того, щоб номер позиції об’єкта Star не виходив за межі кількості існуючих позицій. Якщо ж об’єкт Star на інших позиціях, то випадковим чином обчислюється, чи буде його позиція у наступній «стіні» більше або менше на одиницю.

Скрипт Speed відповідає за збільшення швидкості падіння об’єктів через деякий ігровий час:

```
void Update () {
		if(stop_sp){
			if(time_to_speed >= 10) sp = 1.5f;
		if(sp != 10) time_to_speed += Time.deltaTime;
		}
}
```

По закінченню 10 секунд ігрового часу, змінна sp, яка відповідає за швидкість, стане рівною 1.5.

3.3.2 Розробка скриптів для типу гри Run

Блок схема поведінки об’єктів для типу гри Run представлена у додатку А.2. Виходячи з блок-схеми, в першу чергу випадковим чином генерується позиція та тип об’єкту, який буде рухатися зліва на головного персонажа гри. Якщо об’єкт зіткнеться із головним персонажем, то відбувається або відібрання одного життя у гравця, або збільшення кількості очок. Якщо ж об’єкт виходить за межі камери гравця, об’єкт знищується. Головний персонаж може як перестрибувати об’єкти, так і ухилятися від них.

Далі будуть розглянуті скрипти, призначені для реалізації механіки типу гри Run.

Скрипт Axe_rotate призначений для зміни кута повороту об’єкту Axe, який може з’явитися випадковим чином на одному з рівнів гри Run. Кут повороту можна у будь-який час змінити за допомогою інспектора Unity3D:

```
public float rt = -180f;
   // Use this for initialization
   void Start () {
            transform.Rotate (new Vector3 (0f, 0f, rt));}
```

При запуску скрипта об’’єкт буде повернутий на кількість градусів, яка задана у змінній rt. За замовчуванням об’єкт буде повертатися на -180 градусів.

Скрипт Go_dead відповідає за пересування об’єктів справа наліво на ігровий карті:

```
void Update () {
transform.Translate(Vector3.left * Time.deltaTime * speed);
            if (transform.position.x <= -66) {
Destroy(gameObject);
            }
   }
```

При кожному запуску скрипта об’єкт, до якого прив’язаний даний скрипт, буде змінювати положення по осі X. За зміну координат відповідає

функція transform.Translate. Функція Vector3.left дозволяє змінювати значення осі X та зменшувати його. За швидкість переміщення об’єкту відповідає змінна speed. Після кожного переміщення відбувається перевірка – якщо значення по осі X об’єкта стає більше або дорівнює -66 (позиція за камерою гравця), то об’єкт знищується.

Скрипт Inv_ghost відповідає за зникнення та появу об’єктів Ghost у грі Run:

```
void Update () {
        if (rg.rnd == my_turn) {
                time += Time.deltaTime;
                if (time >= 1.5) {
                gameObject.collider2D.enabled = false;
        _color = gameObject.renderer.material.color;
                        if(_color.a > 0){
                                _color.a -= step;
                gameObject.renderer.material.color = _color;
                        }
                }
        }
  }
```

Після перевірки включається відлік часу, після завершення якого вмикається плавне зникнення об’єкту. Функція gameObject.collider2D.enabled = false дозволяє відключити колайдер об’єкту. Колайдер відключається, щоб об’єкт не міг доторкнутися до головного персонажа. Після відключення колайдера відбувається поступове зникання об’єкту за допомогою змінної _color. У ній зберігається компонент типу Color, який відповідає за колірну схему об’єкту. Спочатку у змінну _color записується поточне значення кольору об’єкту в форматі rgba. Потім починає зменшуватися змінна a у компонента, яка відповідає за прозорість колірної схеми. Шляхом зміни прозорості колірної схеми створюється вид плавного зникнення об’єкту.

У грі Run одним з типів рухомих на головного персонажа об’єктів є два об’єкти Ghost – один на землі, другий над ним. Коли вони наближаються до головного персонажу, один з об’єктів Ghost повинен зникнути і, в залежності

від того, який з об’єктів зникне, гравцеві доведеться або перестрибнути об’єкт, що залишився, або ухилитися від нього.

Скрипт Jump_dead дозволяє одиночним об’єктам Ghost «літати» по ігровій карті, поступово наближаючись до головного персонажу:

```
void Update () {
		if (transform.position.y <= -7.8) {
			trig = true;
		}
		if (transform.position.y >= 0.5) {
			trig = false;
		}
		if (trig) {
   transform.Translate(Vector3.up * Time.deltaTime * speed);
		} else{
   transform.Translate(Vector3.down * Time.deltaTime * speed);
		}
   }
```

Спочатку відбувається перевірка – де знаходиться об’єкт по осі Y. Якщо об’єкт по осі Y вище ніж 0.5, то змінна-перемикач trig стає рівною false і об’єкт починає рухатися вниз, якщо ж об’єкт по осі Y нижче ніж -7.8, то trig стає рівною true і об’єкт рухається вгору.

Об’єкт Ghost може з’явитися на ігровий карті або над землею, або на землі й рухатися, відповідно, або зверху-вниз, або знизу-вгору. Гравець повинен швидко визначити, як рухається об’єкт Ghost, та перестрибнути його, або ухилитися від нього.

Скрипт Road_move відповідає за пересування землі, на якій стоїть головний персонаж:

```
void Update () {
		transform.Translate(Vector3.left * Time.deltaTime * speed);
		if (transform.position.x <= -71f) {
		Instantiate(road, new Vector3(21f, -11.9f, 0),  Quaternion.identity);
			Destroy(gameObject);
		}
   }
```

Функція transform.Translate(Vector3.left * Time.deltaTime * speed) задає швидкість переміщення кожного об’єкту землі. Потім, якщо позиція об’єкту по осі X стає менше або дорівнює -71, за допомогою функції Instantiate (road, newVector3 (21f, -11.9f, 0), Quaternion.identity) створюється новий об’єкт землі, праворуч, поза межею камери гравця, а поточний об’єкт знищується.

Таким чином створюється відчуття, що головний персонаж біжить по землі. Але насправді пересувається кожна ділянка землі, коли ділянка землі виходить за межі камери, вона створює нову ділянку землі, яка так само починає рухатися, а сама знищується.

Скрипт Run_dead призначений для знищення об’єкту при взаємодії з головним персонажем:

```
void OnTriggerEnter2D (Collider2D cold){
        if (cold.gameObject.tag == "Dino") {
            Destroy(gameObject);
        }
   }
```

При взаємодії колайдера об’єкту із колайдером головного персонажу, об’єкт видаляється з карти.

Скрипт Run_score відповідає за нарахування очок в грі Run:

```
void Update () {
        scr += Time.deltaTime;
        fcg.score = (int)scr;
   }

   void OnTriggerEnter2D (Collider2D col){
        if (col.gameObject.tag == "Star") {
            scr += 10;
        }
}
```

У грі Run гравцеві не обов’язково ловити об’єкти Star, очки нараховуються кожну секунду ігрового часу.

Змінна scr відповідає за кількість минулих секунд. Значення цієї змінної перетворюється в цілочисельне значення і присвоюється змінної score, яка знаходиться в скрипті For_run_game, який прив’язаний до змінної fcg. Існує також перевірка на зіткнення головного персонажу з об’єктом Star. Якщо гравець зіткнеться з об’єктом Star, то до поточної кількості очок буде додано 10.

Скрипт Secret_dead відповідає за поведінку одного з об’єктів Dead:

```
void Update () {
		time += Time.deltaTime;
		if (time >= time_to_dead) {
			if(transform.position.y <= pos){
		transform.Translate(Vector3.up * Time.deltaTime * speed);
			}
		}
   }
```

Один з об’єктів Dead може з’явитися на карті за землею, щоб його не було видно. Він з’явиться в безпосередній близькості від головного персонажу, і гравець повинен вчасно зреагувати на нього.

При запуску скрипта, запускається таймер, який затримує появу об’єкту Dead. Після закінчення часу, позиція об’єкту по осі Y зміниться зі швидкістю, яка задана у змінній speed і об’єкт буде рухатися вгору, поки його позиція по Y не стане більше або не буде дорівнювати змінній pos, яка відповідає за кінцеву позицію об’єкту.

Скрипт Secret_saw відповідає за вибір позиції одного з об’єктів Saw:

```
void Start () {
		rnd = Random.Range (1, 3);
   }
   // Update is called once per frame
   void Update () {
		time += Time.deltaTime;
		if (time >= time_to_choice) {
			if(rnd == 1){
```

```
                                        if(transform.position.y <= -0.8){
    transform.Translate(Vector3.up * Time.deltaTime * speed);
                                        }
                                }
                                if (rnd == 2) {
                                        if(transform.position.y >= -8.0){
transform.Translate(Vector3.down * Time.deltaTime * speed);
                                        }
                        }
                }
        }
```

Один з об'єктів Saw в грі Run з'являється на одній позиції на карті, а потім, при наближенні до головного персонажу, змінює свою позицію, переміщаючись або вище, або нижче. Залежно від зміни позиції об'єкту, гравцеві потрібно перестрибнути об'єкт, або ухилитися від нього.

На початку скрипта в функції Start є змінна rnd, яка відповідає за вибір нової позиції об'єкту Saw. Запускається таймер, після закінчення якого об'єкт або переміщається вгору по осі Y до значення -0.8, або переміщається вниз до значення -8.0.

Скрипт Spawn є найважливішим скриптом у грі Run, його завданням є створення ігрових об'єктів на заданих координатах.

Перша частина скрипта:

```
        void Start () {
                        StartCoroutine(spawn());
                }
                IEnumerator spawn(){
                        while (true) {
                                rnd = Random.Range(1, lvl_r);
                                if(rnd == 1){
                        Instantiate(dead, new Vector3(7.4f, -7.1f, 0),  Quaternion.identity);
                                }
                                if(rnd == 2){
                        Instantiate(dead_saw, new Vector3(7.4f, -1.1f, 0), Quaternion.iden-
tity);
                                }
                                if(rnd == 3){
                        Instantiate(star, new Vector3(7.4f, -5.7f, 0),  Quaternion.identity);
                                }
```

На початку скрипта запускається функція spawn за допомогою функції StartCoroutine(spawn()). У функції spawn працює нескінченний цикл (while (true)), в якому відбувається створення об’єктів. Змінна rnd відповідає за номер типу об’єкту, який буде додано на карту, в неї записується випадкове число від 1 до значення змінної lvl_r. Змінна lvl_r задається з інспектора Unity3D окремо для кожного рівня гри Run, вона відповідає за максимальну кількість типів об’єктів, які можуть бути створені. Далі виконуються перевірки на кожен тип об’єкту. Якщо тип об’єкту дорівнює одиниці, то за допомогою функції створюється об’єкт Dead, якщо тип дорівнює двійці, то створюється об’єкт Saw, трійці – об’єкт Star.

Друга частина скрипта:

```
if(rnd == 7){
   Instantiate(axe_crazy, new Vector3(7.4f, -7.2f, 0),    Quaternion.identity);
   }
   if(rnd == 8){
   Instantiate(double_ghost, new Vector3(7.4f, -4.7f, 0), Quaternion.identity);
   }
   if(rnd == 9){
   Instantiate(secret_dead, new Vector3(7.4f, -11.6f, 0), Quaternion.identity);
   }
   if(rnd == 10){
   Instantiate(suddenly_saw, new Vector3(7.4f, -11.6f, 0), Quaternion.identity);
   }
   sp = Random.Range(min_time_to_spawn, max_time_to_spawn);
   yieldreturnWaitForSeconds(sp);
            }
```

У другій частині скрипта так само тривають перевірки на тип об’єкту. Після них у змінну sp записується час до появи наступного об’єкту. Час до появи вибирається випадково в проміжку від значення змінної min_time_to_spawn до значення змінної max_time_to_spawn. Обидві ці змінні задаються у інспекторі Unity3D, що дозволяє у будь-який час їх змінювати.

3.3.3 Розробка скриптів для переходів між сценами Unity3D

Правильний порядок сцен і переходи між ними важливі у кожній грі. У грі Dino’s Adventure існую такі сцени:

– сцена Головне меню;
– сцена Меню ігор;
– сцени уровней;
– ігрові сцени.

Скрипт mainMenu відповідає за переходи сцен у Головному меню:

```
public void Continue(bool CGame)
   {
            Application.LoadLevel (2);
   }
   public void Quit(bool Quit)
   {
            Application.Quit();
   }
```

Функція Continue, яка викликається при кліці на кнопку Games. У цій функції викликається функція Application.LoadLevel, що відповідає за перехід між сценами. В даному скрипті вона здійснює перехід на сцену номер два, а саме на сцену Меню ігор.

Функція Quit викликається при кліці на кнопку Exit. У ній викликається функція Application.Quit (), яка призводить до закриття програми.

Скрипт Second_menu відповідає за переходи сцен в Меню ігор:

```
public void Games(int game_type)
   {
            Application.LoadLevel (3);
            save = GameObject.FindGameObjectWithTag ("Saves");
            save.GetComponent<svs> ().type_scene = game_type;
   }
```

При кліці на будь-яку кнопку із типом гри викликається функція Games, яка викликає перехід до сцени номер три, до першого кадра рівнів. Також у змінну save записується об’єкт, який знаходиться по тегу за допомогою функції GameObject.FindGameObjectWithTag. Потім отримується

скрипт знайденого об’єкту і в цей скрипт за допомогою змінної type_scene задається номер типу гри.

Скрипт svs відповідає за збереження номера типу гри:

```
void Start () {
		DontDestroyOnLoad(this);
		an_sv = GameObject.FindGameObjectsWithTag("Saves");
		if (an_sv.Length> 1) {
			Destroy(gameObject);
		}
   }
```

Збереження номера типу гри потрібне для того, щоб на сцені з рівнями клік по кнопці рівня гри здійснював перехід на сцену конкретного типу гри.

Функція DontDestroyOnLoad(this) дозволяє об’єкту, до якого прив’язаний скрипт з цією функцією, вільно переміщатися між сценами гри, при цьому зберігаючи значення параметрів своїх компонентів.

Потім в змінну an_sv, яка є масивом об’єктів, заносяться всі знайдені на сцені об’єкти з тегом Saves. Якщо в масиві буде більше одного елемента, об’єкт самознищується. Це було зроблено для того, щоб при поверненні на Головне меню об’єкт Saves не дублювався.

Скрипт lavel здійснює переходи між сценами в сценах із рівнями.

Перша частина скрипта:

```
void Start () {
		save = GameObject.FindGameObjectWithTag ("Saves");
   }
public void l1(int isl)
   {
   if (save.GetComponent<svs> ().type_scene == 1) {
		save.GetComponent<svs> ().game_lvl = 5 + isl;
			Application.LoadLevel ((5 + isl));
		}
   if (save.GetComponent<svs> ().type_scene == 2) {
		save.GetComponent<svs> ().game_lvl = 23 + isl;
			Application.LoadLevel ((23 + isl));
		}
   if (save.GetComponent<svs> ().type_scene == 3) {
		save.GetComponent<svs> ().game_lvl = 35 + isl;
   Application.LoadLevel ((35 + isl));
```

```
        }
    }
```

На початку скрипта в змінну save заноситься об’єкт з тегом Saves, в якому знаходиться інформація про обраний на сцені раніше тип гри. Функція l1 викликається при кліці на кнопку будь-якого з рівнів, в неї також передається значення isl, в якому знаходиться номер обраного рівня. Потім здійснюються перевірки на номер типу сцени і, в залежності від нього, здійснюється перехід на сцену гри потрібного типу і потрібного рівню.

Друга частина скрипта:

```
public void menuGame(bool mGame)
    {
        Application.LoadLevel (1);
    }
    public void lvl1(bool lvl1)
    {
        Application.LoadLevel (3);
    }
    public void lvl2(bool lvl2)
    {
        Application.LoadLevel (4);
    }
    public void lvl3(bool lvl3)
    {
        if ((save.GetComponent<svs> ().type_scene == 1)) {
            Application.LoadLevel (5);
        } else {
            Application.LoadLevel (3);
        }
    }
```

Функція menuGame викликається при кліці на стрілочку в лівому верхньому кутку сцени. Ця функція здійснює перехід на сцену Головного меню.

Функції lvl1 – lvl3 викликаються при кліці на стрілочки внизу сцени. Ці функції здійснюють переходи між сценами рівнів. При цьому Функція lvl3 також містить перевірку на тип гри, так як для деяких типів гри не повинно існувати третьої сцени рівнів. В результаті перевірки визначається – чи буде

здійснюватися перехід на третю сцену рівнів, або ж буде перехід назад на першу сцену.

ВИСНОВКИ

У результаті проведеної роботи був створений та оптимізований програмний комплекс мобільного додатку Dino’s Adventure, який включає до

себе більше 30 скриптів. Скрипти були розроблені за допомогою мови програмування C# в середовищі розробки програмного коду MonoDevelop. Кожен скрипт відповідає за певну механіку у грі, а також тісно пов’язаний з іншими скриптами. Усі скрипти були налаштовані за допомогою інспектора Unity3D і прив’язані до ігрових об’єктів. Усі програмні помилки в скриптах були усунені.

Значимість скриптингу в розробці будь-яких ігор, будь то мобільний додаток або комп’ютерна гра, полягає в тому, що без нього ігри являють собою одну лише картинку з повторюваними анімаціями. Будь-яка дія, рух, взаємодія гравця з ігровими об’єктами або ігрових об’єктів з гравцем створюється за допомогою програмного коду скриптів. Скрипти дозволяють взяти всі об’єкти і зв’язати їх воєдино, створивши при цьому систему, яка і називається грою.

Програмний комплекс мобільного додатку Dino’s Adventure буде в подальшому розвиватися – будуть додані нові режими ігор, для яких будуть створені унікальні ігрові механіки, а вже існуючі механіки будуть поліпшуватися. Будуть прикладені максимальні зусилля для того, щоб мобільний додаток Dino’s Adventure отримав можливість зайняти своє місце у мобільній ігровій індустрії.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Unity – руководство: Скриптинг. URL: https://docs.unity3d.com/ ru/530/ Manual/ScriptingSection.html (дата звернення 1.11.2019).

2. Unity – руководство: Обзор скриптинга. URL: https://docs.unity3d.com/ru/530/Manual/ScriptingConcepts.html (дата звернення 1.11.2019).

3. Unity – руководство: Создание и Использование Скриптов. URL: https://docs.unity3d.com/ru/530/Manual/CreatingAndUsingScripts.html (дата звернення 1.11.2019).

4. Unity – руководство: Variables and the Inspector. URL: https://docs.unity3d.com/ru/530/Manual/VariablesAndTheInspector.html (дата звернення 3.11.2019).

5. Unity – руководство: Scripting Restrictions. URL: https://do-cs.unity3d.-com/ru/530/Manual/ScriptingRestrictions.html (дата звернення 6.11.2019).

6. Unity – руководство: Script Serialization. URL: https://docs.unity3d.com/ru/530/Manual/script-Serialization.html (дата звернення 6.11.2019).

7. Unity – руководство: Управление игровыми объектами (GameObjects) с помощью компонентов. URL: https://docs.unity3d.com/ru/530/Manual/ControllingGameObjectsComponents.html (дата звернення 7.11.2019).

8. Unity – руководство: What is a Null Reference Exception?. URL: https://docs.unity3d.com/ru/530/Manual/NullReferenceException.html (дата звернення 7.11.2019).

9. Unity – руководство: Important Classes. URL: https://docs.unity3d.com/ru/530/Manual/ScriptingImportantClasses.html (дата звернення 20.11.2019).

10. Unity – руководство: Функции событий. URL: https://docs.unity3d.com/ru/530/Manual/EventFunctions.html (дата звернення 20.11.2019).

11. Unity – руководство: Unity События (UnityEvents). URL: https:// docs.unity3d.com/ru/530/Manual/UnityEvents.html (дата звернення 20.11.2019).

12. Unity – руководство: Coroutines. URL: https://docs.unity3d.com/ru/530/Manual/Coroutines.html (дата звернення 1.12.2019).

13. Unity – руководство: Time and Framerate Management. URL: https://docs.unity3d.com/ru/530/Manual/TimeFrameManagement.html (дата звернення 1.12.2019).

14. Unity – руководство: Создание и уничтожение игровых объектов

(GameObjects). URL: https://docs.unity3d.com/ru/530/Manual/CreateDestroyObjects.html (дата звернення 7.12.2019).

15. Unity – руководство: Специальные папки и порядок компиляции скриптов. URL: https://docs.unity3d.com/ru/530/Manual/ScriptCompileOrderFolders.html (дата звернення 7.12.2019).

16. Unity – руководство: Пространства имён. URL: https:// docs.unity3d.-com/ru/530/Manual/Namespaces.html (дата звернення 7.12.2019).

17. Unity – руководство: Атрибуты. URL: https://docs.unity3d.com/ru/530/Manual/Attributes.html (дата звернення 8.12.2019).

18. Unity – руководство: Порядок выполнения функций событий. URL: https://docs.unity3d.com/ru/530/Manual/ExecutionOrder.html (дата звернення 8.12.2019).

19. Unity – руководство: Общие функции. URL: https://docs.unity3d.com/ru/530/Manual/GenericFunctions.html (дата звернення 8.12.2019).

20. Unity – руководство: Понимание автоматического управления памятью. URL: https://docs.unity3d.com/ru/530/Manual/UnderstandingAutomaticMe-moryManagement.html (дата звернення 8.12.2019).

21. Unity – руководство: Платформенно зависимая компиляция. URL: https://docs.unity3d.com/ru/530/Manual/PlatformDependentCompilation.html (дата звернення 8.12.2019).

22. Обзор языка C# – руководство по C# | Microsoft Docs. URL: https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp (дата звернення 9.12.2019).

23. C Sharp | Microsoft вики | Fandom. URL: https://mi-crosoft.fandom.com/ru/ wiki/C_Sharp (дата звернення 9.12.2019).

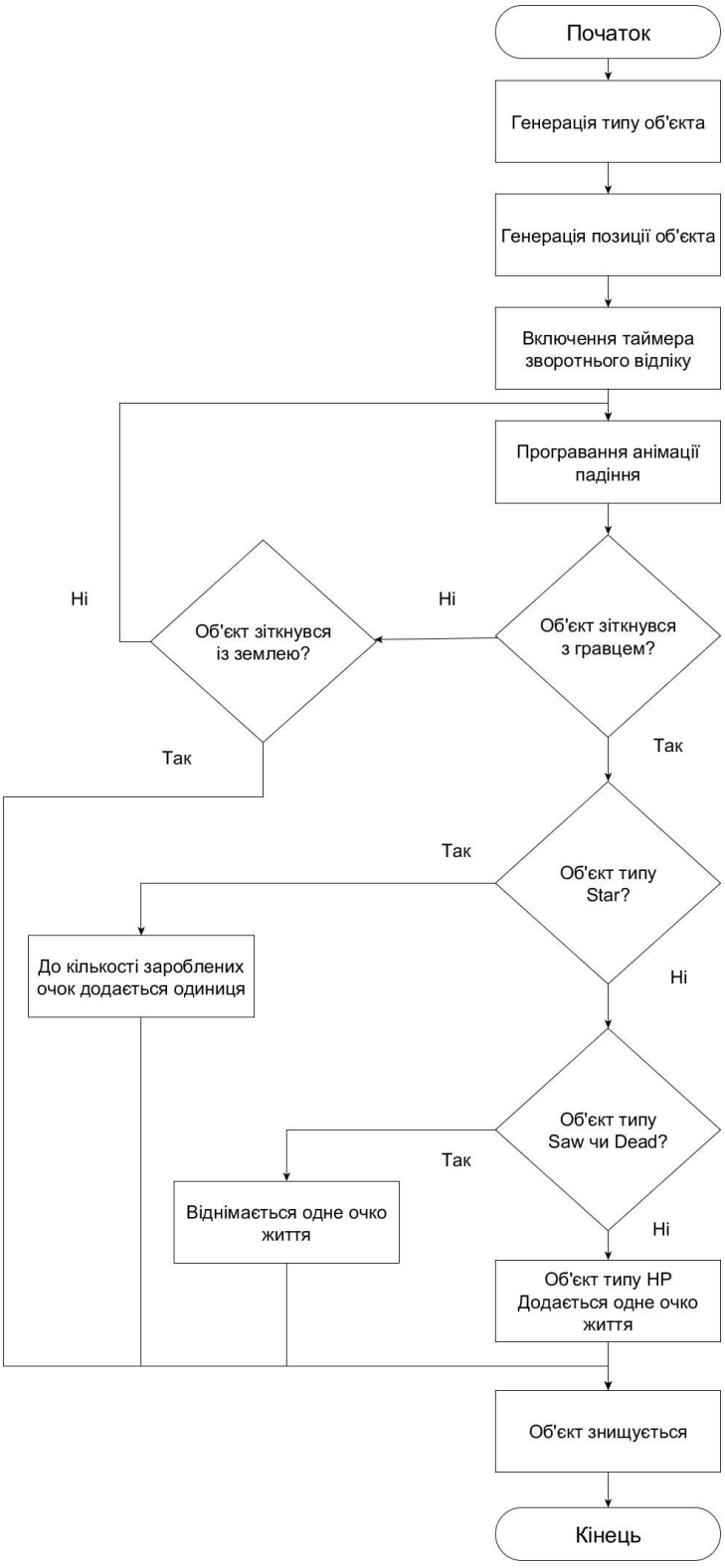
24. Unity – руководство: MonoDevelop. URL: https://docs.unity3d.com/ru/530/Manual/MonoDevelop.html (дата звернення 9.12.2019).

25. MonoDevelop – MonoDevelop – qwertyu.wiki. URL: https://ru.qwertyu.wiki/wiki/MonoDevelop (дата звернення 9.12.2019).

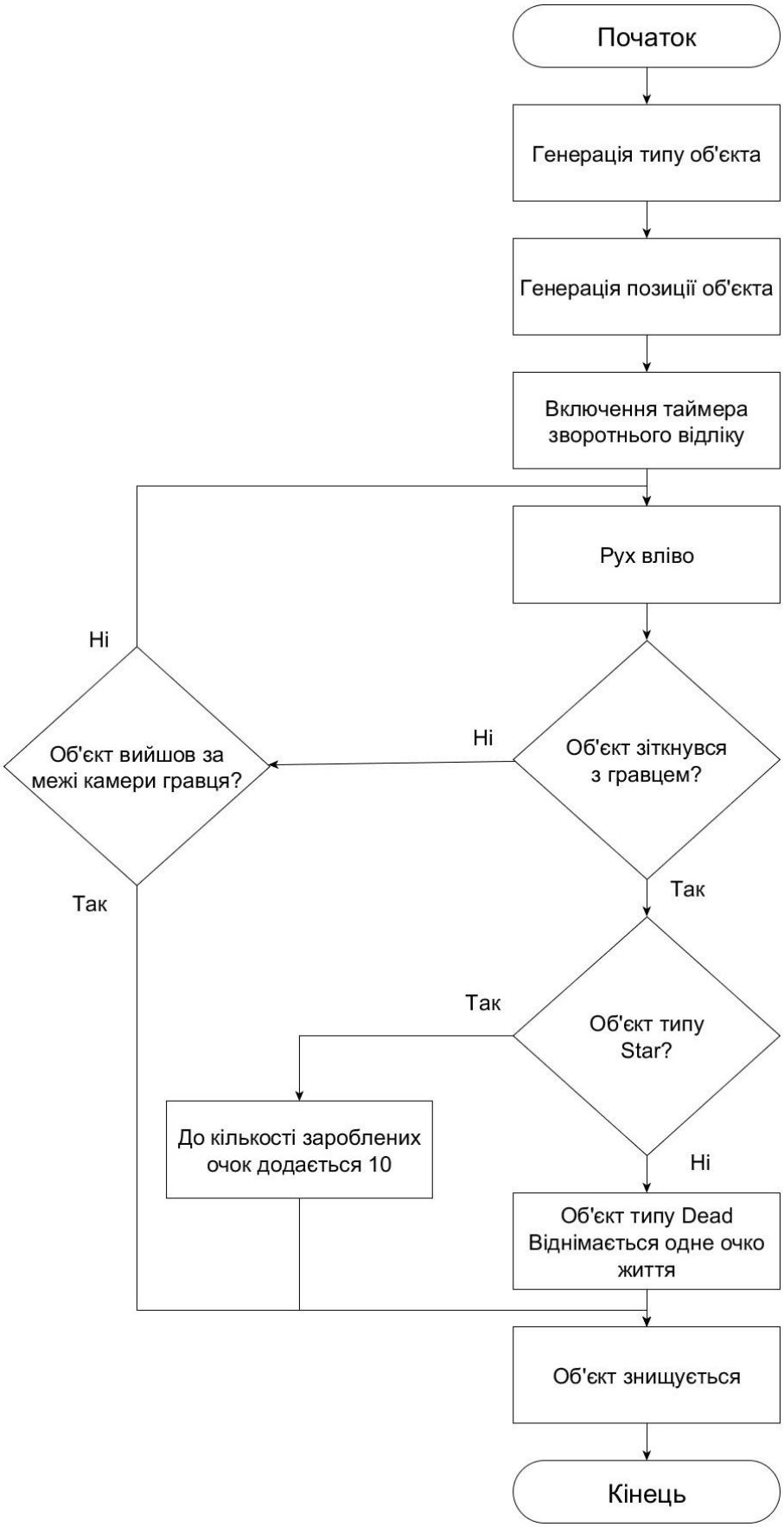
ДОДАТОК А

Блок-схеми ігрових механік мобільного додатку Dino’s Adventure

А.1 Блок-схема гри Catch



А.2 Блок-схема гри Run


Початок
Генерація типу об'єкта
Генерація позиції об'єкта
Включення таймера зворотнього відліку
Рух вліво
Об'єкт зіткнувся з гравцем?
Ні
Об'єкт вийшов за межі камери гравця?
Ні
Так
Так
Об'єкт типу Star?
Так
До кількості зароблених очок додається 10
Ні
Об'єкт типу Dead
Віднімається одне очко життя
Об'єкт знищується
Кінець