Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет інформаційних радіотехнологій та технічного захисту інформації
(повна назва)

Кафедра медіаінженерії та інформаційних радіоелектронних систем
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Розробка цифрового 2D додатку для Android у середовищі Unity
(тема)

Виконав:
студентка 2 курсу, групи СТМм-20-1
Климова А.М.
(прізвище, ініціали)

Спеціальність 171 Електроніка
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи, технології і комп’ютерні засоби мультимедіа
(повна назва освітньої програми)

Керівник проф. Карташов В.М.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ____________ Карташов В.М.
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Факультет інформаційних радіотехнологій та технічного захисту інформації
Кафедра медіаінженерії та інформаційних радіоелектронних систем

Рівень вищої освіти другий (магістерський)
Спеціальність 171 Електроніка
(код і повна назва)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма Системи, технології і комп'ютерні засоби мультимедіа
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«____»____________ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентці Климовій Анастасії Михайлівні
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка цифрового 2D додатку для Android у середовищі Unity.

затверджена наказом по університету від "08" 11 2021 р. № 1675 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 08.12.2021 р.

3. Вихідні дані до роботи Розглянути програмні середовища і засоби формування цифрового 2D додатку для Android. Здійснити аргументований вибір технічних засобів, які мають необхідні якості для вирішення завдання. Розробити 2D платформер для Android смартфона в середовищі розробки Unity.

4. Зміст пояснювальної записки (перелік питань, що потрібно розробити)
Вступ.
1. Жанр комп'ютерних ігор – платформер
2. Алгоритм створення двовимірного платформеру
3. Розробка гри в Unity
Висновки.
Перелік посилань
Додатки

5. Перелік графічного матеріалу із зазначенням обов'язкових креслеників, схем, плакатів, комп'ютерних ілюстрацій
1. Спрайти.
2. Налаштування героя та об'єктів з якими він контактує.
3. Налаштування поверхі та інтерфейсу.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналітичний огляд літератури	08.11.21–09.11.21	
2	Огляд аналогічних проектів	10.11.21–11.11.21	
3	Теоретичне обґрунтування вибору теми	12.11.21–13.11.21	
4	Теоретичне обґрунтування вибору інструментів	14.11.21–15.11.21	
5	Розробка програмної частини	16.11.21-19.11.21	
6	Розробка проекту	20.11.21–24.11.21	
7	Робота з графікою	25.11.21–02.12.21	
8	Перевірка керівником	03.12.21-04.12.21	
9	Перевірка на академічний плагіат	05.12.21	
10	Перевірка завідувачем кафедри, рецензування	06.12.21–07.12.21	

Дата видачі завдання ______________08.11.2021 р.______________

Студентка ____
(підпис)

Керівник роботи (проекту) ______________________ ______проф. Карташов В.М.______
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи: 86 сторінки, 19 рисунків, 30 джерел.

РОЗРОБКА, ПЛАТФОРМЕР, ДВОВИМІРНИЙ ДОДАТОК, ПРОГРАМУВАННЯ, UNITY, C#

Об’єкт дослідження – розважальний додаток для мобільного пристрою з двовимірною графікою.

Мета дослідження – створити двовимірну гру для смартфонів с операційною системою Android, удосконалити навики роботи з мовою програмування C# та з середовищем розробки Unity.

Методи дослідження – теоретичний аналіз аналогічних проектів, що дозволить розробити концепцію оптимального платформеру, збір інформації про створення подібних ігор та використання отриманих знань при розробці.

В кваліфікаційній роботі розглянута історія розвитку відеоігор в цілому та платформерів в часності, проведено аналіз сучасних платформерів, проведено огляд середовищ розробки та платформ для вибору оптимальних інструментів, детально описано алгоритм розробки двовимірного додатку та з використанням обраних інструментів розроблено гру для мобільної платформи з операційною системою Android, яка розважає користувачів, допомагає рослабитись після важкої праці та розвиває спритість і логіку.

РЕФЕРАТ

Объяснительная записка к квалификационной работе: 86 страниц, 19 рисунков, 30 источников.

РАЗРАБОТКА, ПЛАТФОРМЕР, ДВОМЕРНОЕ ПРИЛОЖЕНИЕ, ПРОГРАММИРОВАНИЕ, UNITY, C#

Объект исследования – развлекательное приложение для мобильного устройства с двумерной графикой.

Цель исследования – создать двумерную игру для смартфонов с операционной системой Android, получить навыки работы с языком программирования C# и ознакомиться со средой разработки Unity.

Методы исследования – теоретический анализ аналогичных проектов, который позволит разработать концепцию оптимального платформера, сбор информации о создании подобных игр и использовании полученных знаний при разработке.

В квалификационной работе рассмотрена история развития видеоигр в целом и платформеров в частности, проведен анализ современных платформеров, проведен обзор сред разработки и платформ для выбора лучших инструментов, подробно описан алгоритм разработки двумерного приложения и следуя данной инструкции разработана игра для мобильной платформы с операционной системой Android, в результате получены знания программирования на языке C# и навыки работы со средой разработки Unity, которая развлекает пользователей, помогает раслабиться после тяжелолой работы и развивает ловкость и логику.

ABSTRACT

Explanatory note to the qualification work: 86 pages, 19 figures, 30 sources.

DEVELOPMENT, PLATFORMER, DUAL APPLICATION, PROGRAMMING, UNITY, C #

The object of the research is an entertainment application for a mobile device with two-dimensional graphics.

The aim of the research is to create a two-dimensional game for smartphones with the Android operating system, gain skills in working with the C # programming language and get acquainted with the Unity development environment.

Research methods - theoretical analysis of similar projects, which will allow developing the concept of an optimal platforming game, collecting information about creating such games and using the knowledge gained during development.

In the qualification work, the history of the development of video games in general and platformers in particular is considered, an analysis of modern platformers is carried out, a review of development environments and platforms for choosing the best tools is carried out, an algorithm for developing a two-dimensional application is described in detail and, following this instruction, a game is developed for a mobile platform with the Android operating system. As a result, knowledge of programming in the C # language and skills of working with the Unity development environment were obtained, which entertains users, helps to relax after hard work and develops agility and logic.

ЗМІСТ

Перелік умовних позначень, символів, скорочень і термінів		8
Вступ		9
1	Жанр комп'ютерних ігор – платформер	11
1.1	Різновид платформ	11
1.2	Історія створення жанру та його розвиток	14
1.2.1	Платформери в рамках одного екрану	14
1.2.2	Механіка фліп-скрін	15
1.2.3	Скролл платформери	16
1.2.4	Платформери з відкритим світом	16
1.3	Середовища розробки	17
1.4	Аналіз сучасних платформерів	19
1.5	Висновок по розділу 1	21
2	Алгоритм створення двовимірного платформеру	25
2.1	Категорія гри та цільова аудиторія	25
2.2	Вибір операційної системи	28
2.3	Візуальна складова	30
2.4	Висновок по розділу 2	32
3	Розробка гри в Unity	34
3.1	Управління героєм та його взаємодія з оточенням	34
3.2	Створення ворогів різного типу	59
3.3	Дизайн рівнів	76
3.4	Головне меню	77
3.5	Висновок по розділу 3	81
Висновки		83
Перелік посилань		84
Додаток А		87
Додаток Б		97

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

Бібліотека – набір об’єктів чи підпрограм, які використовуються для розробки програмного забезпечення.

Колайдер – це допоміжний компонент, що окружає модель чи її частину та взаємодіє з іншими моделями, що мають колайдер.

Корутина – це потоки виконання коду, що організуються над системними потоками.

Префаб – це заздалегідь підготовлений ігровий об’єкт, що використовується декілька разів при створенні гри

Поток виконання коду – це послідовність операцій, що виконуються один за одним.

Скрипт – це невелика програма, що складається із послідовності команд та методів, створених для автоматичного виконання команд.

Спрайт – графічний об’єкт в комп’ютерній графіці.

Тригер – компонент, що дозволяє визначити, зіткнення об’єктів і при цьому не зупиняти їх.

Tilemap (карта тайлов) – це сітка, що використовується для створення макету гри.

ВСТУП

З кожним роком комп’ютерні ігри набирають все більшу популярність та розвиваються. Перша комп’ютерна гра була представлена публіці у 1940 році у Нью-Йорку [1]. То був великий комп’ютер в якого замість екрану були лампи. Вони прикріпленні по сім штук у чотири ряди. Суть гри була в тому, щоб по черзі людина та комп’ютер гасили лампи (користувач для цього натискав на важіль). Гасити за раз можна скільки завгодно, але у одному стовбцю. Той хто гасив останню лампу перемагав.

Люди були в захваті від гри, але з часом той факт, що комп’ютер робить хід занадто швидко, почав їх лякати. Тому у програму ввели затримку, щоб після ходу людини, комп’ютер трохи чекав. Так з’явилась ілюзія немов комп’ютеру потрібен час подумати. Але він все одно був розумніший за людей та весь час перемагав.

Ігри мають велику роль у сучасному суспільстві. Деякі з них роповідають цілі історії, що робить їх в деякому сенсі витвором мистецтва. Тепер ігри не тільки розважають, відволікають від проблем, а й знайомлять та об’єднують людей, допомогають розвивати логіку та спритність, дають змогу розбагатіти та набути популярність. Можна записувати на відео процес гри та викладати до мережі, що збиратиме просмотри та за рахунок реклами приходитимать грощі, чи можно брати участь у змаганнях, де можна виграти мільйони.

Основні жанри комп’ютерних ігор на цей час [2]: симулятор, головоломки, стратегії, пригодницькі, екшн, рольові.

Симулятор [3] – це імітація певного процесу як наприклад водіння (керуєш транспортом Need for Speed), лікаря (лікуєш хворих Surgeon Simulator), чи життя в цілому (де можна створити якого завгодно персонажа та розробити власний сценарій The Sims).

Головоломки [4] – ігри де необхідно розв’язувати задачі на логіку (наприклад тетріс та сапер).

Стратегії [5] – ігри що потребують продумувати свої дії наперед як в шахах (наприклад Warcraft, Heroes of Might and Magic, Stronghold).

Пригодницькі [6] – де користувач, управляючи героєм гри приймає рішення, що впливають на хід гри, досліджує світ та виконує завдання (наприклад Horizon Zero Dawn, Assassin's Creed, Detroit: Become Human).

Екшн[7] – жанр в якому перевіряється реакція користувача (Grand Theft Auto, Cyberpunk 2077, The Witcher).

Рольові [8] – по суті цей жанр пішов від настільних ігор. Тут теж в героя є певні характеристики, що впливають на гру. Їх можна змінювати , отримав бали за виконання завдань.

За 81 рік з'явились не тільки нові жанри відео ігор, а й пристрої на яких можна грати: мобільні пристрої, консолі, кишенькові приставки, окуляри віртуальної реальності.

Графіка також розвинулась. Все починалось з ламп, а тепер ми маємо такі великі екрани, що можемо роздивитись пори на обличчі героїв гри.

Основна задача магістерської кваліфікаційної роботи - створення гри з оптимальним рівнем складності, з використанням якої користувачі зможуть розважити себе у черзі, транспорті чи зробити цікаву паузу між годинами тяжкої праці. Водночас гра не займатиме забагато часу, щоб в гравця не було жалю за витрачений час.

1 ЖАНР КОМП’ЮТЕРНИХ ІГОР – ПЛАТФОРМЕР

1.1 Різновид платформ

Водночас робити гру на персональний комп’ютер, шолом віртуальної реальності, браузера, консолі та смартфон не можливо бо кожен пристрій має свої особливості.

У Персональних комп’ютерах [9] вся графіка транслюється на монітор, що підключено до системного блоку. Характеристики такі як роздільна здатність екрану, частота кадрів, діагональ дуже важливі при роботі з ігровим інтерфейсом. У якості офіційних переносників зазвичай онлайн магазини та у рідких випадках диски. Для керування героєм використовуються комп’ютерна миш, клавіатура та можна підключити геймпад. Чим більша оперативна пам’ять, тим скоріше будуть відкриватись програми та працювати, легше буде запускати ігри с деталізованою графікою, кількість кадрі в секунду буде більша, а отже комфортніше буде за нею спостерігати, але тут ще важлива відеокарта.ЇЇ модель та оперативна пам’ять [10].

Окуляри чи шолом віртуальної реальності мають два маленьких екрана (по одному на кожне око) які дають зображення на 360 градусів. Це дає відчуття немов дійсно знаходишся в іншому місці та можна розгледіти його повертаючи куди завгодно голову [11]. Пересуватись по віртуальному світу можна використовуючи джойстики чи за допомогою шолому зробити віртуальний кордон в межах якого гравець буде пересуватись ногами як у реальності. Замість джойстиків можуть бути спеціальні рукавички, що виконують такі ж функції. Незвичний спосіб користування. Тут в нас не плоский екран, а картинка навколо людини. Віртуальна реальність дає змогу поринути у гру більш ніж будь-який інший пристрій [12], але є й свої недоліки. Ця технологія ще занадто юна та недосконала, тож не має таких широких можливостей як конкурентні пристрої. До того ж не всі в змозі користуватись окулярами так як може швидко настигнути нудота. Проблема в тому, що наш мозок під впливом

шолому відчуває протиріччя. Він бачить наче його власник рухається, але не відчуває цього. Починається кінетоз [13] – мозок наче галюцинує, як при отруєнні, а отже необхідно позбавитись від небезпечної речовини у шлунку і людину нудить. На щастя технологію покращують та боряться з цим ефектом.

В браузерні ігри грати можна як на персональному комп’ютері, так і зі смартфону [14]. Потрібен лиш браузер. Браузерні ігри не повинні багато важити, щоб час завантаження був невеликим. Навряд чи користувачам сподобається затяжне очікування. Також мало хто забажає платити додаткові гроші за витрачені гігабайти, якщо грають, використовуючи мобільний інтернет. Тож гра рівня Відьмак 3 чи Cyberpunk 2077 в браузері неможлива. Здебільшого ігри двовимірні. Розробка тривимірних можлива, але ця тема поки не так добре розвинена. Що стосується розробки, то тут все простіше ніж в «конкурентів» та швидше. Браузерні ігри в основному прості та не дають змоги з’єднатися з товаришем, щоб грати разом через обмеження можливостей. При бажанні можна зробити мереживу гру, та доведеться прикласти багато зусиль.

В консолях для керування використовуються геймпад чи джойстик [15]. Не для всіх ігор такий спосіб є комфортним, для більшості гравців клавіатура та мишка улюблена периферія для гри. Розробляти на консоль ігри більш вигідно ніж на персональні комп’ютери та мобільні пристрої, бо їх зламати більш важко. Тобто менше піддані піратству. Отже якщо зробити дійсно якісний продукт, що багатьом сподобається – їм доведеться купляти гру. Наприклад компанія Playstation часто випускає ексклюзиви яких немає на інших платформах, що дає їм великі гроші. Розробляючи гру на консоль, треба концентруватись лише на одному пристрої. Playstation 5 для прикладу. З персональними комп’ютерами більше труднощів бо там великий вибір процесорів та відеокарт. Усе потрібно враховувати. Та операційна система не одна. Зі смартфонами подібні проблеми через їх великий вибір на ринку. До того ж доводиться добиватись сумісності з різними версіями однієї операційної системи. Це не так легко як хотілось, тож з часом розробники повідомляють про завершення підтримки старих операційних систем. Звісно це не до вподоби

користувачам із старими гаджетами. З графікою тут можна як слід розвернутись, так як можливості неймовірні. Процес ліцензування для консольних ігор доволі важкий та довгий. Є багато пунктів яким гра повинна відповідати. Якщо хоч з одним не справиться (навіть дрібним), то розробник отримає відмову. Транслюється гра, як і в випадку з персональним комп'ютером, на екран монітору.

Смартфони управляються через невеликий екран, що вміщується у руці. При дизайні меню та гри в цілому слід добре обміркувати як правильно розмістити усі іконки, щоб ті не заважали гравцю. Так як екран смартфонів та планшетів у рази менший ніж екран монітору, то графіка обмежена. Да і в цілому наприклад малювати зморшки персонажам сенсу немає бо їх не буде помітно. Так як смартфони зазвичай використовують мобільний інтернет, можливі проблеми із з'єднанням. Слід продумати цю частину дуже добре, щоб зберегти гравця від втрати процесу бою, наприклад. У девайсів не найкраща ситуація із оперативною пам'яттю, тож краще гру максимально спрощувати у рамках дозволеного [16]. Якщо щось не є необхідним, то видаляти. Відвантажувати усі текстури та ресурси що в цей час не мають значення. Звук краще стискувати. Формати слід використовувати із компресією. Найпопулярніші операційні системи Android та IOS [17]. З останньою набагато простіше. Там смартфони лише компанії Apple, тож із сумісністю не буде таких великих проблем як в Android. Бо там величезній вибір мобільних девайсів від різних фірм. Щоб користувачі змогли завантажити гру, необхідно помістити її у магазин необхідної операційної системи. Там її повинні схвалити. Розробляти додатки та ігри на IOS легше на персональному комп'ютері чи ноутбуці від фірми, що і створили цю операційну систему, а саме Apple. Є можливість робити це і на звичайному комп'ютері із операційною системою Windows, але тоді від повинен бути більш потужним, щоб витримати Хакинтош – зламана версія Mac OS, що може працювати на персональному комп'ютері, що не був для неї призначений. Є й інші способи, але вони потребують часу та нервів, так як можуть виникнути проблеми. Android найбільш розповсюджена мобільна

операційна система в світі, тож це дуже гарний варіант щоб розпочати кар'єру розробника. Розробляти додатки на Android зручно з будь-якої операційної системи [18].

1.2 Історія створення жанру та його розвиток

Платформер – гра суть якої стрибати по платформам (звідси й назва), проходити через ворогів (пастки) , збираючи різноманітні бонуси (додаткові життя, монети та т.д.). Графіка звичайно далека від реалізму і більш мультиплікаційна. Почали свій шлях платформери з 1980 року [19]. Перші були двовимірними так як на більше тоді техніка була не здібна.

1.2.1 Платформери в рамках одного екрану

Першим платформером вважається Space Panic [20]. Вийшла вона на аркадних автоматах. Події відбувалися у одній локації. Не було можливості перемістити камеру. Користувачу була доступна лише та область, яку він побачив від початку. Нова гра не користувалась попитом через нові функції, що можливо робили гру занадто важкою для гравців. Зараз герої спокійно стрибають у будь-якому платформері, але тоді подібного не було. Персонаж Space Panic переміщався вгору та вниз завдяки драбинам. Серед можливостей також було копання ям для ворогів. Коли ті туди потрапляли гравець повинен нанести останній удар та викинути за кордони екрану. Все це потрібно виконати до закінчення кисню, що по суті було таймером.

Далі публіці представили гру Donkey Kong 1981 року. Гравці вперше познайомились із Маріо – популярним нині, вже не перший десяток років, італійським водопровідником. Ця гра принесла часті стрибки, що мало великий вплив на весь жанр. Суть гри була в тому, що горила викрадає кохану у італьянця, а той долає пастки, підіймаючись в гору екрану, перемагає викрадача та рятує принцесу. Гра розповідала невеличку історію за допомогою

спеціальних сцен у початку та вкінці, що стало ковтком свіжого повітря в іграх. В результаті гра мала успіх, як і її продовження. Люди й досі грають управляють водопровідником, але зараз гра називається Супер Маріо.

У 1983 році італієць отримав брата Луіджі та гру зі своїм ім’ям у назві – Mario Bros. У цей раз компанія Nintendo, що створила гру, також зробила важливе нововведення у ігрову індустрію. Тепер усі персонажі гри доходячи до лівого краю екрану виходили з правого краю та навпаки. Ця технологія називається – wraparound. Тоді як раніше герой та його вороги лиш впирались у стінку і в них не було іншого виходу окрім як повернутись та піти в іншу сторону.

1.2.2 Механіка фліп-скрін

Суть її в тому, що коли герой доходить до кінця екрану, зображення перемикалось на інше. Кількість фонів з намальованими джунглями – 255. Тобто герой не залишався на одній локації та дійсно рухався далі.

Найвідомішим серед перших хто зміг вийти за кордоні однієї сцени є Гаррі Пітфол з серії ігор «Pitfall!» 1982 року від компанії Activision. Також з ним в гру почали приходити різноманітні фони замість монотонних темних. Ціллю Гаррі було знайти 36 скарбів, перемагаючи ворогів та укластися у 20 хвилин.

У 1984 році вийшло продовження Pitfall II: Lost Caverns. Тепер герой міг рухатись не лише у сторону екрану, а й вниз та вгору. Наприклад впав у дірку в землі, а там замість смерті багатоповерхова печера з якої можна вибратись.

Гра під назвою Manic Miner, що вийшла у 1983 році ввела фонову музику в індустрію. Через недосконалість тогочасної техніки, зробити так, щоб під час гри йшла музика нікому не вдавалось. Доки Метью Сміт, що працював над грою не винайшов цікаву техніку. Він зробив так, щоб центральний обробляючий пристрій перемикався між грою та звуком. Що стосується методики самої гри то там використовувалась найперша – єдиний екран, який змінювався тільки при повному проходженні. Тобто коли всі бонуси зібрані.

У продовженні 1984 року під назвою Jet Set Willy спробували переключення екрану, коли герой доходить до краю. Гравець отримав можливість прогулюватись по великій будівлі з різноманітними кімнатами. Безумовною перевагу серед конкурентів була можливість для гравців продумувати та творити власні рівні.

Перший інвентар з'явився у грі H.E.R.O. Там герой літав на джетпаці та підривав кам'яні стінки динамітом.

1.2.3 Скролл платформери

У цій методиці фон рухається разом з героєм.

Pac-Land 1984 року покращила цю методику, а саме зробила так щоб он слудкував за героєм повільніше и це добавило глибини. Отриманий ефект називається параллакс-скроллинг

Super Mario Bros 1985 року поєднав в собі усі наробітки попередників, що зробило гру дуже популярною. Маріо окрім звичних нагород збирав бонуси, що дають перевагу у боротьбі з ворогами. Противники теж на місці на стоять та становяться небезпечними. Пошук монет стає більш цікавим завдяки таємним локаціям, що ще треба знайти. А найважливіше те, що додали рівень з навчанням. Щоб ознайомити гравця з механікою гри та облегшити проходження. Супер Маріо й досі усі знають, хтось навіть грає.

1.2.4 Платформери з відкритим світом

У таких іграх герой може спокійно рухатись по всій карті окрім закритих місць, до яких необхідно відкрити доступ. В цілому можна досліджувати світ, збирати різні бонуси з якими герою до снаги сильні противники. Цю методику також називають «Метроідванія», що є об'єднанням назв двох ігор на суті яких вона і основується «Metroid» та «Castlevania».

Brain Breaker 1985 року одна з перших подібних ігор. Гравцю необхідно досліджувати світ та знаходити пристрої, що можуть всіх врятувати.

Castlevania III: Dracula`s Curse одна з найяскравіших представниць жанру дала гравцю можливість приймати рішення, що впливає на гру.

Mega Man розроблений у 1987 році дав користувачам доволі незвичний вибір – порядок проходження рівнів.

У деяких іграх стали робити кат-сцени та через них розповідати історію персонажів.

Гра Prince of Persia розроблена 1989 року. Плавність анімації у ній була отримана завдяки ротоскопії. Джордан Мекнер, що створив гру, для малювання більш реалістичних рухів, знімав світлини молодшого брата, що виконував трюки головного героя гри.

Далі в іграх почали з’являтися помічники, а самі ігри з’явились на портативних приставках.

1.3 Середи розробки

Створюються ігри у спеціальному програмному забезпеченні. Їх вибір доволі великий. Є навіть такі, що не потребують знання жодної мови програмування. Людина усе робить через графічній редактор. Можливості подібних звісно обмежені, тож краще власноруч писати код, щоб гра повністю відповідала бажання розробника, та не була скована у рамках середовища розробки.

Наприклад є CONSTRUCT 2. Простий та зрозумілий інтерфейс. При бажанні можна й код написати. Для цього необхідне знання мови Javascript. Безкоштовна версія має певні обмеження, що знімаються з покупкою професійної. Коли розробник отримує з гри суму більшу ніж 5 000 доларів, то повинен заплатити за комерційну ліцензію. Різниці у можливостях не буде. По суті таким чином компанія бере свій процент з заробітку розробника. Вибір

платформ для створення ігор та додатків великий. Як комп’ютерні операційні системи, так і мобільні, і браузери.

DEFOLD є кросплатформним, тобто підходить майже для усіх операційних систем. Найбільше підходить для двовимірних ігор, але є можливість і тривимірні розробляти. Робота в ньому не буде коштувати нічого. Версій з більш широким функціоналом немає. Код пишеться на процедурній динамічно типізованій модульній мові Lua.

Phaser має спрайтову графіку, підтримує декілька камер, двовимірну графіку, система часток, завантаження ресурсів кодом у одну строку, система плагінів для розширення функцій середи розробки та підтримка мобільних браузерів. Розробляти можна для персональних комп’ютерів, Android та iOS.

CORONA також дає змогу обирати із широкого списку операційних систем в числі якого є Android. Мова кодування така ж як і в DEFOLD, тобто Lua. Користування безкоштовне для розробників. Раніше було дві версії цієї середи розробки. Одна була з повним функціоналом за певні кошти, а інша з обмеженнями та безкоштовна.

Lumberyard безкоштовна середа розробки та доволі нова. Має вбудовану пудтримку для деяких сервісів Amazon, а саме Twitch та AWS. Розробляти ігри можна для таких платформ як Xbox One, Windows, VR (HTC, Vive, Oculus Rift), Android, iOS.

UNREAL ENGINE більше підходить для розробки ігор призначених персональним комп’ютерам, але й для мобільних пристроїв можна творити. Можливості доволі великі завдяки з використанням мови C++. Також має свою власну систему програмування, що називається Blueprint. Можна розробляти ігри не тільки на персональні компьютери та смартфони, а ще й на консолі як стаціонарні, так й портативні наприклад Nintendo Switch. Стосовно графічної складової з Unreal engine важко конкурувати. Двигун володіє вражаючими можливостями. Тут і доволі потужний штучний інтелект, і підтримка DirectX 12, і для створення кат-сцен редактор. Стосовно оплати середа розробки безкоштовна доки прибуток з гри за квартал не більше ніж 3 000 доларів.

Надалі Epic Games - фірма, що володіє двигуном, буде забирати 5 відсотків з отриманих вами за гру грошей.

UNITY мультиплатформенна середа разробки. Тобто можна створювати ігри на Apple iOS, Windows Phone, PlayStation 4, Xbox One, PlayStation 3, Xbox 360, Nintendo wii, Nintendo Switch, OS X, Windows, Android та Linux. Модуль під назвою Unity Web Player та технологія WebGL дають змогу створювати проекти для браузерів. Використовувати Unity можна для розробки ААА-ігор (високо бюджетні ігри) та Indie ігор (independent video game — незалежна комп'ютерна гра). OpenGL та DirectX підтримуються додатками що були створені у цьому середовищі розробки. Інтерфейс Drag&Drop доволі зрозумілий та не складний в користуванні. Налагодження гри можна виконувати у самому редакторі. NVIDIA PhysX допомагає о розрахунках фізики. Стосовно оплати тут звична та справедлива форма. Доки людина отримує прибуток менше 100 000 доларів у рік, може користуватись безкоштовним Unity Personal, далі має оплатити користування Unity Plus, ті хто заробляють більше 200 000 доларів на рік повинні оплатити Unity Pro. Усе чесно. Люди що заробляють подібні суми можуть оплатити підписку на середу розробки, завдяки якій вони ці гроші і заробили.

1.4 Аналіз сучасних платформерів

Для зрівняння були убрані ігри для різних девайсів, не тільки для мобільних пристроїв.

Limbo – мультиплатформна гра у жанры пазл-платформер з елементами виживальних жахів (Survival horror), що була створена датською студією під назвою Playdead. У якості фізичного двигуна використовується Box2D. За своєю суттю гра є сайд-скроллером. Тобто за процесом гри слідкує камера, що і дає картинку гравцю. Рухатись можна вправо, вліво, якщо є ями чи платформи, що ведуть в небо, то можливий рух вниз та вгору.

Хлопець прокидається серед лісу. У пошуках сестри йому доводиться боротись із зустрічними персонажами та вирішувати головоломки. Раптово він бачить дівчинку, що сяє, але підійти не встигає, бо вона зникає. Закінчується гра тим, що він через розбите вікно вилазить з забутого заводу, Знаходить поле, а там дівчинку, що підскакує з наляканим обличчям.

Стилістично гра притримується мінімалізму з темними тонами, що дало змогу робити пастки менш помітними і в них легко потрапити. Останнє для когось може вважатись недоліком. Стосовно геймплею нічого нового та особливого. Грі нема чим виділитись.

Mark of the Ninja – жанр стелс-екшн (тобто гра розрахована на скритність). Виробник канадська студія Klei Entertainment.

Суть гри в тому, щоб бути максимально непомітним. Старатись зовсім не мати жодних контактів з ворогами та не потрапляти в пастки. У якості зброї спеціальний меч та дротики з бамбуку. Вбивати ворогів необов'язково, можна обійтись і без насилля. За виконані дії гравець отримує бали, що знадобляться для купівля спорядження та відкриття нових прийомів. Є костюми, що спеціалізовані для певного стилю проходження. Для отримання балів достатньо перейти на новий рівень, знаходити таємні місця з артефактами, вирішувати головоломки.

Грати потрібно за ніндзя з особливою татуїровкою, що робить його здібності сильнішими, але приводить до втрати глузду. Ціль звільнити членів клану, що у полоні загіну Гессен.

У своєму жанрі гра доволі гідна, але керування трохи перенавантажене.

Ori and the Blind Forest – гра з відкритим світом. Студія-розробник австрійська Moon Studios.

Керувати у грі потрібно Орі (біла істота) та лісним духом на ім'я Сейн, що є його захисником. По геймплею все доволі стандартно: переміщення по карті через різноманітні перепони та вороги, що бажають помірятись силою. Для прокачування здібностей є спеціальне дерево та енергія, що розкидана по карті.

Ділянки для проходження можна на карті, але деякі недоступні без певних навиків.

Відмінною рисою гри є доволі гарне оформлення. Розроблений сюжет у зв'язці з підходящим звуковим супроводом дають підходящу атмосферу. З недоліків лиш більш складний геймплей через неінтуїтивне керування.

Wings of Vi – від датської компанії Grynsoft.

За геймплеєм гра стандартна. Боротьба з ворогами, пастки та боси. Що стосується складності, то вона від початку доволі висока за рахунок сильних ворогів та дизайн рівнів розроблено таким чином, щоб проходження медом не здавалось. Це доволі інтересний хі, але подібне не кожному до вподоби.

Пикселизована графіка надає грі ретро-стилю.

DuckTales: Remastered – перевидання платформеру, що засновано на мультсеріалі Качині історії. Розроблено компанією Capcom сумісно з WayForward.

У новій версії обновили графику, змінили умови рівнів складності (на легкому рівні життів нескінченно, на середньому та складному по два житті), добавили «екстремальний» рівень складності та два додаткові рівні самої гри.

Суть гри як і в попередніх іграх – вбивати ворогів та не потрапляти у пастки. У кінці чекає бос. Вороги та персонажі намальовані у двовимірному середовищі, а декор у тривимірному.

Художники добре постарались та в цілому грати зручно, але так як це оновлення старої гри, слідувало придумати побільше нововведень в ігровий процес.

1.5 Висновки по розділу 1

Ігри на персональні комп'ютери набули такого рівня, що самотужки розробнику важко буде конкурувати з багатомільйонними компаніями. Та навіть неможливо. Найбільш популярні об'єкти з тривимірною реальністю. Вони більш важкі, а отже не кожен комп'ютер потягне подібну роботу. Велика

кількість полігонів на одну сцену будуть перенапружувати девайс. Так як впевненості в тому, що мій персональний комп'ютер впорається з подібним навантаженням немає, то такий варіант мені не підходить. Робити двовимірну гру великого сенсу немає, бо люди не для того збирають потужні машини, щоб у легкі ігри грати. Уся суть гри на великому моніторі саме у вражаючій мозок графіці – як можна кращій якості зображення, деталізованим тривимірним моделям, що можна роздивитись з будь-якої сторони, плавній анімації та яскравим спецефектам. Подібна робота потребує дуже багато часу та зусиль, що може бути не до снаги людині, яка лише починає свій шлях у розробці ігор. Щоб створити продукт, що зможе хоч трохи заінтересувати людей, необхідна ціла команда. Спеціаліст зі створення тривимірних моделей, аніматор – його робота полягає в тому, щоб рухати точки з яких складається скелет моделі, та робити у потрібних положеннях так звані ключі (фіксувати положення на шкалі часу), та в результаті змушувати об'єкти рухатись необхідним чином, якщо мова йде про істот, у створення анімації вогню та води вже інші технології, дизайнер рівнів, що все разом з'єднає, програміст, що напише скрипти, людина, що тестуватиме проект, по завершенню та концепт-художник, що розробить дизайн майбутніх моделей [21]. Це мінімум який необхідний для конкурентоспроможного тривимірного об'єкту для персональних комп'ютерів.

З віртуальною реальністю все ще гірше. Технологія не дуже популярна, а отже кількість потенційних гравців невелика. Все через доволі значні недоліки, з якими ще тільки борються. А саме світло у грі може різко погаснути, доволі малі та часті текстури, занадто схожі або навіть однакові пейзажі, відсутність на об'єкті, що рухається, текстур часом у людей викликає нудоту. При русі платформи, якщо використовується технологія згладжування, різкий поворот голови привиде до розмиття зображення, що викличе дискомфорт у користувача. У плані технічного оснащення ігри віртуальної реальності доволі вимогливі, тому у візуальному плані їх роблять як можна простіше, щоб не перенавантажувати техніку. Занадто реалістична графіка з недостатньо гарною оптимізацією може привести до падіння частоти кадрів у секунду та грати буде

дуже важко. Рівні в іграх віртуальної реальності повинні бути короткими. Краще не більше сорока хвилин, щоб гравець зміг його пройти без зупинок, а потім відпочити. Це потрібно мозку та тілу в цілому, авжеж подібні ігри потребують немалої активності від людини. Простір, що доступний більшості людей доволі обмежений у зрівнянні з простором у самій грі. Тож не можливо у повній мірі насолодитись усім, що вона пропонує, бо можна у щось врізатись та зламати. Коли людина прийде від одного кордону, що виділено для ігрової області, до іншого, їй доведеться повертатись назад, а у грі буде довга дорога до кінця якої ще йти та йти. Це необхідно враховувати при розробці гри. Дати змогу переміщатись по всій карті з урахуванням дійсних обмежень користувача. Та слід пам'ятати, що різкі телепортації, рухи камери можуть дезорієнтувати та знову причинити дискомфорт. Ще у багатьох девайсів віртуальної реальності є дроти. Під час гри вони постійно заважають. Звісно розробник ігор з чим нічого зробити не в змозі, але слід розуміти що це одна з причин чому користувачів шоломів та окуляр віртуальної реальності небагато, а отже ця сфера не дуже прибуткова. Здавалось би, що тут можуть допомогти бездротові окуляри, але ні. Вони теж мають певні недоліки. Пакети даних гаджету заважкі для комфортної дистанційної передачі. Бездротове з'єднання не таке якісне та надійне, як провідне. Тож це погіршує зображення. Наразі лише Facebook із окулярами віртуальної реальності під назвою Oculus Quest 2 змогли усунути цей недолік, але добавили новий. А саме слідкування за гравцями. Щоб використовувати пристрій, необхідно зареєструвати акаунт у цій соціальній мережі і тим самим дозволити їй володіти усією інформацією, що отримує шолом. У результаті стає зрозуміло, що розробляти гру для віртуальної реальності не найкраща думка. До того ж я навіть не зможу її протестувати, адже спеціального гаджету не маю.

Браузерні ігри втратили свою популярність дуже давно. Раніше, коли можливості комп'ютерів були у рази нижчі, подібні ігри біли гарною розвагою. Багатий вибір різноманітних жанрів. Біли навіть ігри, що колись вмикались з

картриджів на старих приставках. На сьогоднішній день ігри в браузері більш не конкурентоспроможні.

Консольні ігри приносять чималі проблеми при релізі. Керування геймпадом повинно підтримувати у всіх механіках та GUI. У кожної фірми, що випускає консолі свої правила що до випуску ігор. В цілому пробитись у цій сфері доволі важко. Дуже багато бюрократичних проблем. Тож цей варіант теж не підходить. До того ж протестувати гру на консолі також не мою змоги.

Смартфони з усіх найкращий варіант, адже вони дуже розповсюджені, грати на них можна де завгодно, характеристики девайсів хоч і обмежені, але все одно добре прокачались за останні роки. З кожним роком прибуток з мобільних ігор по світу зростає. На відміну від комп’ютерних ігор, мобільні можна робити безкоштовними і все одно заробляти. Потрібно лише інтегрувати рекламу. Для першої гри мобільна платформа відмінно підходить, а надалі можна буде покращувати навики та переходити на більш складні проекти. Для початку варто зайнятись в міру легким двовимірним додатком. Так як зараз випускають римейки старих ігор можна зробити висновок, що в людей знову повернулась жага до ностальгії, а одним з найпопулярніших жанрів ігор мого дитинства були платформери. Тож можливо для них зараз самій час. Так як Android найпопулярніша мобільна система, то саме нею варто зайнятись. До того ж поряд чимало людей, що можуть протестувати гру та вказати на недоліки.

З урахування сучасних технологій серед перерахованих механік найкращим вибором буде скролл платформер с відкритим світом. Так фон та камера повільно слідкуватимуть за героем, що найбільш комфортно для очей. Завдяки відкритому світу гравець зможе самотужки обирати яким шляхом піти чи пройти всю карту та зібрати усі бонуси.

Для розробки гри на Android найкращим середовищем розробки буде Unity, бо вона легка в використанні, безкоштовна доки заробіток не перевищу сто тисяч доларів, мульиплатформа, що надасть змогу портувати гру на інші гаджети.

Щоб створити продукт, здатний зацікавити, слід притримуватись правил перших платформерів, але добавити ще нових функцій. Змішати перевірене часом із іноваціями, бо без своєї особливості гра не виділятиметься з існуючих.

2 АЛГОРИТМ СТВОРЕННЯ ДВОВИМІРНОГО ПЛАТФОРМЕРУ

2.1 Категорія гри та цільова аудиторія

Перед розробкою гри слід визначитись якою вона буде та для кого?

Серед категорій э Casual – це ігри без сюжету із простим управлінням. За допомогою подібних ігор люди зазвичай вбивають час у чергах та транспорті. Отже управління повинне бути для однієї руки, щоб іншою спокійно триматись за поручень. У такому разі краще робити відображення на екрані в портретному режимі, щоб смартфон зручно лежав у руці, а великим пальцем здійснювалось керування. Механіка гри повинна бути зрозумілою користувачу на інтуїтивному рівні. Навчання повинне тривати близько п’яти секунд. Часто ігри роблять циклічними. Гравець виконує однакові дії знову та знову, доки герой не помре, як наприклад в Subway surfers. Складність з часом зростає.

Соціальні мережі із нескінченими стрічками новин показали, якщо людина не бачать кордонів, то може втонути. Замість кінця людина отримує все більше новин, зображень та відео. У результаті не встигає оком мигнути як вже декілька часів минуло. З іграми це теж працює. Важко відірватись від ігри, що не має кінця та цікаво як далеко вдасться дійти цього разу. Серед недоліків – однотипність. На прикладі того ж Subway surfers можна побачити, що однакові дії та декорації з часом набридають. Розробники боряться з цим за допомогою зміни атмосфери. Час від часу малюють нові декорації та роблять сезонні завдання. Ще є рейтинг гравців. За призові місця нагороджують, що заохочує грати більше.

Гра під назвою Flappy Bird, що вийшла 2013 року зіграла велику роль в історії мобільних ігор. Нарешті розробники ігор для смартфонів змогли успішно отримувати прибуток зі своїх творінь. Вдалось це завдяки рекламі.

Казуальні ігри набули немалої популярності завдяки продуманому проектуванні. За зовнішньою простотою стоять цілі команди розробників, що працюють не один місяць.

Подібні ігри для одного користувача, але компанія Voodoo змогла створити ілюзію гри у мережі з іншими користувачами. Насправді там використовується штучний інтелект. У цьому можна впевнитись покинув гру на деякий час. Повернувшись можна побачити, що усі гравці на місці та нічого не змінилось.

Ще одна з причин зросту популярності даної категорії у використанні поп-культури та соціальних мереж. Як наприклад гра Pineapple Pen, що натхненна вірусним відео з YouTube. Люди шукали його, а разом знаходили посилання на гру. Run Sausage Run! Побачила світ через декілька днів після появи фільтру з сосискою у Snapchat. Багато хто знає про Flappy Bird через чималу кількість відео із розлюченими користувачами, що розбивали свої смартфони через програш.

Реклама не єдиний спосіб монетизації. Часто роблять магазини де можна купити додаткові бонуси, що допоможуть в грі та нові дизайни головного герою чи ігрового фону.

Останнім часом люди втомлюються від безкінечних ігор, де не видно прогресу, тому деякі розробники розділяють їх на рівні, що створює ілюзію просування вперед, а не біг на одному місці. Або поєднують обидві механіки, даючи користувачам вибір.

Конкуренція в цій категорії чимала. Великі компанії захоплюють ринок. Одна з них Voodoo.

Midcore – більш складні, ніж казуальні, але не занадто. Гравець може розслабитись та насолоджуватись процесом. Тут вже можна з іншими гравцями взаємодіяти та є ресурси якими потрібно керувати з розумом. Дана категорія потребує більше інтелекту, ніж реакції, як наприклад в Angry Birds. Кожен рівень повинен приносити щось нове, інакше він не матиме сенсу і користувач може пожалкувати втраченого часу, та навіть видалить гру. На

відміну від попередньої категорії, у цій необхідна система збережень. Щоб була можливість продовжити гру на тому місці, де закінчив.

Складність зростає по ходу гри, але починається все з простого, щоб не важко було розібратись. Гравцю потрібен час, щоб розібратись з механікою. Взагалі не варто доводити до того, щоб на проходження рівню витрачалось багато спроб. Подібне дратуватиме багатьох та зменшить можливу аудиторію. Якщо герой помер, то варто дати можливість пройти рівень, на якому це відбулось, заново, а не збивати прогрес.

Що стосується системних вимог, то вони не повинні бути завеликими. Це збільшить цільову аудиторію, адже не кожен має потужний комп'ютер чи смартфон. Розмір гри повинен бути невеликим, щоб при повільному інтернеті можна було завантажити не очікуючи цілі години. Установка чим простіше, тим краще. Без додаткових програм та реєстрацій.

Hardcore – категорія для любителів складностей, у яких багато вільного часу. Навчання триває довше, адже ігри мають високий рівень складності. Рівень залучення найвищий завдяки виклику, що кидається гравцям, системі рівнів, яка дає змогу бачити прогрес та повноцінним історіям, що допомагають глибше поринути в гру.

Для категорії характерні довгі сеанси та періодична відсутність можливості зберегти прогрес (наприклад підчас бою). В деякий і зовсім немає збережень, як в бойових аренах MOBA (Multiplayer Online Battle Arena - Багатокористувацька онлайн бойова арена) чи онлайн шутерах. Вибір жанрів дуже великий. Грати можна як одному, так і з іншими гравцями через інтернет. Є можливість воювати один проти одного та об'єднуватись у команди. На разі подібні ігри навіть мають власні чемпіонати, як The International по Dota 2 з величезним призовим фондом та кількістю вболівальників.

Зацікавленість в подібних іграх зазвичай не обмежується лише процесом гри. Сюжет та персонажі так подобаються користувачам, що ті організовують цілі фан бази як це роблять з фільмами, серіалами та книгами. Фанатська творчість переповнює інтернет, а іноді переростає у самостійні великі проекти.

Інші категорії не можуть похвалитись такою відданістю фанів. Харкорні ігри найбільш довгограючі проекти, що не дивно, адже там задіяні величезні команди спеціалістів.

У комп'ютерні ігри починають грати ще з малого віку – приблизно 6 років. Згідно статистиці кількість гравців у вікових категоріях від неповнолітніх до сорока п'яти років приблизно схожа [22]. Після сорока п'яти вже не так багато користувачів. Тож цільова аудиторія від шести до сорока п'яти років.

2.2 Вибір операційної системи

Найпопулярнішими серед мобільних є Android від Google та iOS від Apple. Перша встановлена на 71 відсотку усіх мобільних девайсів, а друга на 28 відсотках.

Одним з плюсів розробки на операційну систему Android є її гнучкість. Вона менш обмежена ніж яблучна операційна система. Розроблені додатки працюватимуть майже на кожному Android смартфоні. Проблеми з сумісністю пристрою мало ймовірні. Сама розробка пристрою стала більш комфортною та гнучкою. Розробка на Java можлива на різних операційних системах персонального комп'ютеру, таких як Linux, Mac OS та звісно Windows [23]. Якщо програмувати на C#, то потрібен Windows.

Другий плюс у тому, що створюючи додаток для Android, його можна задіяти у чималій екосистемі пристроїв. Наприклад гру можна буде запустити не тільки на смартфоні, а й на телевізійних приставках. У результаті додаток можна оптимізувати для розумних часів, авто, телевізори, тощо. Можливі проблеми з дизайном, та їх можна виправити.

Третій плюс у доступності навчальної документації, як офіційної від самої компанії, так і багато відео уроків, статей та книг від досвідних розробників. Ті, кому важко навчатись власноруч, можуть піти на спеціальні курси. Книги є для різних рівнів знань, тож навіть люди, що давно займаються розробкою, зможуть найти щось для себе.

Серед недоліків те, що на розробку під Android витрачається більше часу, ніж під яблучну операційну систему.

Додатки для Android більш підвернені атакам зі сторони хакерів, через відкритий сирцевий код операційної системи. Програми для iOS знаходяться у більшій безпеці завдяки закритості.

Люди, що використовують Android доволі рідко купляють додатки. Вони більш полюбляють використовувати безкоштовні, тож мало витрачають грошей у Google Play. Краща стратегія в такому випадку – робити додатки з безкоштовною версією та більш повною платною, де буде розширений функціонал. Так якщо людям дійсно сподобається програма, вони оплатять підписку на Pro версію.

Через те що деякі люди використовують старі версії операційної системи, доводиться думати й про них, а отже робити додатки більш гнучкими. Часто на періоді тестування з’являться чимало помилок через проблеми з сумісністю.

Серед плюсів розробки на iOS – надійність системи. Яблучна продукція добре працює за що й має великий цінник. Операційна система надійна, швидка та проста у використанні. При останній компіляції кількість можливих помилок мінімальна.

Компанія Apple підготувала доволі детальні інструкції зі створення інтерфейсу для майбутніх додатків, щоб розробникам було простіше.

Екосистема компанії дуже комфортна, якщо людина має принаймні два яблучних гаджети, то з легкістю зможе передавати файли, синхронізувати інформацію та навіть управляти з одного гаджету іншим. Зрозуміло, що це робіть більш простою розробку додатків для всіх гаджетів від компанії Apple.

Серед недоліків, якщо добавляти функції, для яких необхідна взаємодія із стороннім програмним забезпеченням, то виникнуть певні проблеми й виконати подібне буде доволі важко. Суть в обмеженнях, що накладає яблучна операційна система при налаштуванні інтерфейсу майбутнього додатку.

App Store більш вимогливий магазин, ніж аналог для Android. Тут можуть відмовити у випуску не тільки через невідповідність правилам, а й за відсутність користі та неактуальність.

Для розробки під iOS краще мати ноутбук чи моноблок від компанії Apple, бо саме на них розрахована середа розробки під назвою Xcode. Можна встановити віртуальну машину на будь-яку іншу операційну систему, що дасть змогу користуватись програмами для iOS, але не кожен комп'ютер зможе добре працювати в подібних умовах.

2.3 Візуальна складова

Графіка - це дуже важлива частина гри, адже зовнішність перше з чим має справу людина. Неприваблива замальовка може бути причиною, через яку потенційні користувачі обходитимуть гру стороною.

При відсутності навиків малювання та грошей на замовлення спрайтів можна взяти безкоштовні в інтернеті. Деякі автори дають людям змогу користуватись ними безкоштовно. Зрозуміло, що для заробітку на грі знадобиться власноруч малювати графіку чи купляти, але, щоб спробувати свої сили у програмуванні достатньо й цих об'єктів. Тут є вибір головного героя у декількох варіантах: більш зеленуватий з великими очима, блакитний з трьома очима, розовий з одним оком, жовтий з прикритими очима, та блідний з оком що йде в ширину (Рисунок 2.1).

Рисунок 2.1 – Спрайти герою

Різноманітні вороги: желе образні рожевий, блакитний та зелений, дві риби зелена та рожева, муха, піранья, рожеве чудовисько, жаба, кругла пила, зелений квадрат, черв’яки рожевий та зелений, бджола, миша, равлик, жук у краплинку та половина круглої пили (Рисунок 2.2).

Рисунок 2.2 – Вороги

Шість видів фону та землі (трава, грязь, планета, сніг, пісок та камінь). Є квадратні, кутові, округлі та плоскі.

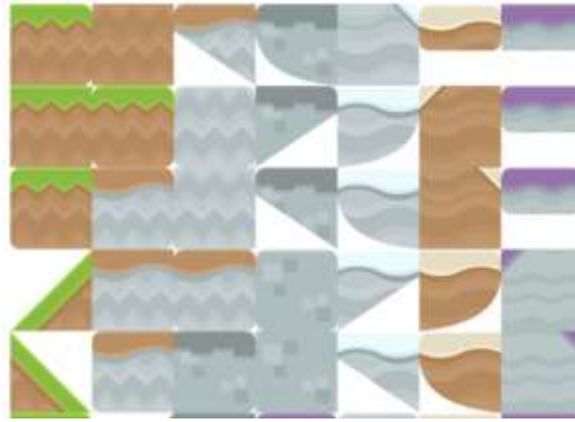

Рисунок 2.3 – Спрайти землі

Також є допоміжні об’єкти та частини декору (Рисунок 2.4).

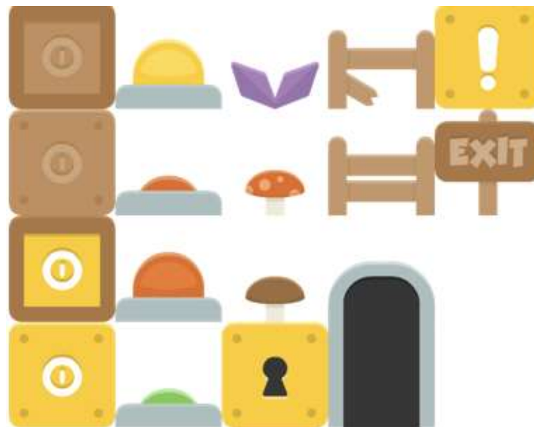

Рисунок 2.4 – Спрайти декору

Двері можна зробити порталом, якщо хід перекрито, у якості пастки може буть вода чи лава які неможливо переплести, а можна лише перейти по мосту який провалюється варто лише наступити. Спрайти показників таких як життя, зібрані ключі та ефекти (Рисунок 2.5).

Рисунок 2.5 – Спрайти показників

Є спрайти для створення анімації горіння факелу, руху ворогів та героя, його падіння, плавання, підіймання вгору.

2.4 Висновок по розділу 2

Наразі конкурувати з Hardcore категорією неможливо без достатнього фінансування та команди. В казуальних іграх звичайно використовуються нескладі та однотипні механіки, тому подібний проект може бути занадто простим. Midcore є оптимальним варіантом, де можна продемонструвати певний набір знань у програмуванні та подолати роботу самотужки. До того ж ця категорія підходить як досвідним геймерам, так і людям, що рідко грають.

Обидві платформи дуже перспективні [24]. Android найбільш популярна, а користувачі iOS більше витрачають на додатки. Що зрозуміло, адже яблучна продукція доволі дорога, а якщо людина в змозі витратити такі гроші на смартфон, то й на ігри вистачить. Причому не тільки на підписку, що виключає рекламу, а й на різноманітні бонуси всередині самої гри.

Публікувати додатки простіше в Google Play. Процес тепер займає більше часу, але все одно правила не такі жорсткі як в яблучному магазині. Там ймовірність відмови набагато більше. Щоб розмістити свої проекти в Google Play необхідно заплатити 25 доларів, а App Store потрібно платити кожного року по 99 доларів.

Українці здебільшого використовують Android, тому ця операційна система є оптимальним вибором. Особливо при відсутності яких-небудь девайсів від Apple. Гру для Android можна спокійно розробляти на компьютері з Windows та протестувати на власному смартфоні.

3 РОЗРОБКА ГРИ В UNITY

Щоб зробити розробку гри простішою слід підготувати префаби. Тобто кожній спрайт, що буде використовуватись у грі наділити необхідними властивостями та зберегти у віддільній папці. Таким чином не доведеться кожен раз заново робити ворога чи кожну клітинку землі по якій ходитиме герой.

3.1 Управління героєм та його взаємодія з оточенням

Спочатку необхідно відредагувати фізику проекту. Натискаємо кнопку Edit, відкриваємо Project Settings, у розділі Physics 2D змінюємо гравітацію по Y на -30. Це найкраще значення для платформеру.

Для фону використаємо ефект параллакс-скроллинг. Зробимо так, щоб фон слідкував за героєм, але уступав йому в швидкості. Це створить ефект глибини.

Для початку необхідно обрати спрайт із фоном і відповідно визначитись із тематикою рівня. Зробимо пустелю. Отже беремо картинку, що відповідає темі.

У правому вікні Inspector у розділі Transform натискаємо на вертикально розташовані три точки справа та обираємо Reset Position. Таким чином ми збиваємо позицію у нульові значення. Розмір (Scale) змінюємо на Y з одиниці на 1.5, щоб фон діставав до верхнього і нижнього країв екрану.

Робимо копію за допомогою клавіш Ctrl+D. Змінюємо позицію по X з нуля на -10.24 (це довжина спрайту). Копію робимо дочірнім об’єктом оригінального спрайту. Для цього затискаємо його лівою клавішою мишки та вносимо у перший спрайт. Фон буде складатись з п’яти спрайтів. Один головний і чотири дочірніх.

Зробили так, щоб через головний спрайт управляти усім фоном. Тепер ми можемо додати код програми, що наддасть фону ефект параллакс-скроллинг. Скрипт працюватиме для усіх п’яти спрайтів, хоча знаходитиметься лише у найпершому.

Створюємо у проекті папку Scripts, відкриваємо, натискаємо праву клавішу мишки, потім наводимо стрілку на Create та натискаємо C# Script. Даємо назву «BG» (від слова background). Створюємо перемінну, для інформації про довжину та стартову позицію фону:

```
float length, startpos;
```

Оголошуємо публічну перемінну, що мітить у собі об’єкт – камеру:

```
public GameObject cam;
```

Створюємо публічну перемінну типу float, що відповідатиме за те наскільки буде сильне зміщення фону. Тобто за те наскільки добрим буде паралакс. Паралакс – зміна видимого положення об’єкту відносно віддаленого фону у залежності від положення спостерігача.

```
public float parallaxEffect;
```

Метод Start виконується лише раз при запуску гри:

```
void Start()
{
```

Записуємо стартову позицію у створену для неї перемінну:

```
    startpos = transform.position.x;
```

У перемінну length записуємо довжину спрайту по осі X:

```
    length = GetComponent<SpriteRenderer>().bounds.size.x;
```

Наступний метод фізичного двигуну. Він не залежить від частот и кадрів та викликається сам по таймеру.

```
void FixedUpdate()
{
```

Створюємо тимчасову локальну перемінну. Присвоюємо значення позиції камери по осі X, помноженої на (1 - parallaxEffect). Віднімаючи від одиниці значення parallaxEffect, задаємо наскільки йде розходження позиції камери з нашим фоном.

```
    float temp = cam.transform.position.x * (1 - parallaxEffect);
```

Створюємо перемінну типу float, що буде дорівнювати позиції камери по осі X, помноженої на parallaxEffect. Dist буде відповідати за те, наскільки сильним буде зміщення фону відносно камери (ефект паралаксу).

float dist = cam.transform.position.x * parallaxEffect;

Позицію прирівнюємо до нового вектору, який буде вміщати стартову позицію плюс перемінну dist. Через кому указуємо позицію по осі Y та Z(вони залишаються незмінними).:

transform.position = new Vector3(startpos + dist, transform.position.y, transform.position.z);

Далі створюємо умову, що перевіряє, якщо розходження позиціє камери з фоном (temp) більше суми стартової позиції та довжини нашого фону:

if (temp > startpos + length)

Тоді ми повинні змінити нашу позицію:

startpos += length;

else if (temp < startpos - length)

startpos -= length;

Коли крайні фони не видно, то переміщуємо один з них у протилежну сторону. Таким чином отримуємо постійний фон, що слідкує за героєм без чорних ліній.

Чим більше parallax Effect, тим більше буде розсинхронізації між рухом фону та камери. parallaxEffect=1 – фон рухається разом з камерою (ніби замер для гравця). parallaxEffect=0 – фон рухається сам по собі (ефект паралаксу максимальний). Для коректної роботи parallaxEffect повинен бути у діапазоні від 0 до 1.

Скрипт заносимо у головний спрайт нашого фону. Він з'являється у правому вікні Inspector та має дві перемінні, що потрібно налаштувати. У строку «Cam» перетягуємо камеру, а саме Main Camera у лівому вікні «Hierarchy», а в строку Parallax Effect пишемо 0.7 для помірного ефекту.

Щоб створити поверхню по якій буде ходити герой та вороги, необхідно додати у сцену спрайт землі. Поки він є просто картинкою. Тобто, якщо

поставити на нього який-небудь об’єкт, він впаде через спрайт вниз. Щоб земля мала необхідно властивість, потрібно додати фізику. У правому вікні Inspector в самому низу натискаємо Add Component. Там обираємо Physics 2D та Box Collider 2D.

Тепер додаємо у сцену головного героя. Для цього потрібно обрати спрайт, що найбільше подобається та перенести у вікно Scene. Додаємо Collider 2D. Тільки не Box , а Capsule . Редагуємо його кордони так, щоб вони збігались з кордонами малюнку. Для цього натискаємо на кнопку поряд з надписом Edit Collider у вікні Capsule Collider 2D та переносимо точки у необхідні позиції.

Тепер герой може стояти на землі, але лише якщо рівно поставити його на неї. Щоб на героя в повній мірі впливала гравітація, необхідно додати компонент Rigidbody 2D (Рисунок 3.1).

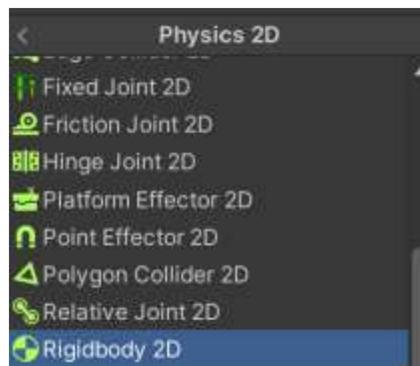

Рисунок 3.1 - Додавання компоненту Rigidbody 2D головному герою

Щоб герой не падав головою вниз, у вікні Rigidbody 2D у розділі Constraints ставимо галочку у строчці Freeze Rotation Z. Таким чином мі заморозили обертання по осі Z.

Щоб керувати героєм, необхідно написати код. Для цього створюємо новий скрипт. У ньому оголошуємо перемінну типу Rigidbody 2D, що по суті є класом вбудованим в Unity. Завдяки цій перемінній, будемо звертатись до компоненту Rigidbody 2D.

```
Rigidbody2D rb;
```

Далі створюємо перемінну дробового типу, що відповідатиме за швидкість:

```
public float speed;
```

У методі Start робимо посилання на фізику головного герою. Присвоюємо перемінній rb значення компоненту Rigidbody 2D за допомогою вбудованого методу GetComponen.

rb = GetComponent<Rigidbody2D>();

У методі Update звертаємось до компоненту Rigidbody2D, а саме до швидкості та задаємо направлення герою:

rb.velocity = new Vector2(Input.GetAxis("Horizontal") * speed, rb.velocity.y);

Клас Input відповідає за введення, тобто натискання клавіш. Horizontal Означає, що мова йде про введення кнопок горизонталі (стрілки вправо, вліво чи A, D). Якщо натиснута клавіша вправо, то значення Input.GetAxis("Horizontal") дорівнює одиниці, а якщо вліво, то мінус один. Якщо жодна кнопка горизонталі не нажата, то значення дорівнює 0.

Помножуємо отримане значення на швидкість за яку відповідає перемінна speed. Потім звертаємось до другого аргументу Vector2 по осі Y. Його залишаємо незмінним.

Герой рухається вправо та вліво. Тепер зробимо так, щоб він довився у ту сторону, куди йде. Створюємо новий метод в кінці класу:

void Flip()

{

Якщо натиснута стрілка право:

if (Input.GetAxis("Horizontal") > 0)

Персонаж знаходиться у начальному положенні – дивиться вправо:

transform.localRotation = Quaternion.Euler(0, 0, 0);

Якщо натиснута стрілка вліво:

if (Input.GetAxis("Horizontal") < 0)

Персонаж дивиться вліво (його зображення віддзеркалюється по горизонталі):

transform.localRotation = Quaternion.Euler(0, 180, 0);

}

Збільшуємо можливості герою, надав йому функцію – стрибок. Для цього оголошуємо публічну перемінну дробового типу, що буде містити данні про висоту стрибку:

public float jumpHeight;

У методі Update записуємо умову – якщо натиснута клавіша пробіл, то герой отримує імпульс вгору (тобто стрибає):

```
if (Input.GetKeyDown(KeyCode.Space)
        rb.AddForce(transform.up * jumpHeight, ForceMode2D.Impulse);
```

Силу направлену вгору помножуємо на довжину стрибка. Через кому вказується двовимірний тип сили з короткою продовжністю.

Щоб стрибок відбувався лише з землі, а не й з повітря, потрібно зробити дочірній об’єкт герою (Рисунок 3.2).

Рисунок 3.2 - Створення дочірнього пустого об’єкту

Іменуємо його groundCheck та створюємо іконку (Рисунок 3.3), щоб бачити при редагуванні (у грі його видно не буде). Переміщаємо об’єкт під героя.

Рисунок 3.3 - Створення іконки для об’єкту, що перевіряє контакт із землею

Оголошуємо дві перемінні для перевірки контакту з землею:

public Transform groundCheck;

Та логічна перемінна:

bool isGrounded;

Створюємо новий метод для перевірки чи контактує герой з землею:

void CheckGround()

{

Оголошуємо масив, якому присвоюємо усі колайдери, що потрапляють у радіус кола, який створюється за допомогою методу Physics2D.OverlapCircleAll. Центр кола знаходиться в позиції groundCheck. Радіус кола 0.2

Collider2D[] colliders = Physics2D.OverlapCircleAll(groundCheck.position, 0.2f);

Якщо довжина списку колайдерів більше одного, то герой стоїть на землі. У разі якщо значення довжини масиву дорівнює одиниці, цією одиницею є спрайт головного герою.

isGrounded = colliders.Length > 1;

if (!isGrounded && !isClimbing)

anim.SetInteger("State", 3);

У метод Update добавляємо щойно створений метод та метод, що відповідає за горизонтальне віддзеркалення герою при ходьбі в ту чи іншу сторону. Щоб методі використовувались кожен кадр:

Flip();

CheckGround();

В умову зі стрибком додаємо перевірку чи стоїть герой на землі, щоб від стрибав лише з поверхні, а в повітрі не мав змоги стрибати:

if (Input.GetKeyDown(KeyCode.Space) && isGrounded)

Тепер створюємо анімацію. Для цього обираємо героя, відкриваємо вікно Animation та натискаємо кнопку Create Створюємо нову папку для анімацій.

У нульовий кадр вносимо спрайт де герой дивиться в сторону, у третій де герой дивиться в камеру, у шостий знов дивиться в сторону.

Для анімації ходьби беремо два спрайти де тіло героя в одному положенні, а положення ніг змінюється. Один у нульовий кадр, другий у перший.

Далі створюємо анімацію стрибку. Беремо спрайт зі стрибаючим героєм та додаємо у нульовий кадр, спрайт з падаючим героєм у другий та третій кадри.

Щоб налаштувати анімації відкриваємо вікно Animator. В нас автоматично є стрілка, що від Entry йде до першої анімації для стану покою. Тобто саме з неї буде починатись гра. До двох других анімацій потрібно зробити перехід. Натискаємо на Any State правою клавішою мішки та обираємо Make Transition, проводимо стрілку до одної анімації, потім так само до другої.

У вікні Parameters натискаємо плюс та обираємо цілісний тип перемінної int. Називаємо State. Це потрібно, щоб пронумерувати анімації та в коді звертатись до них через номер. Натискаємо на стрілку першої анімації. У графі налаштувань убираємо галочку в строчці Can Transition To, щоб анімація працювала краще та більш правильно. Так робимо для всіх майбутніх анімацій. У графі Conditions додаємо State та прирівнюємо до одиниці, другу анімацію до двійки, третю до трійки.

Оголошуємо нову перемінну для анімацій:

```
Animator anim;
```

Далі у методі Start цій перемінній присвоюємо клас герою Animator:

```
anim = GetComponent<Animator>();
```

У методі Update пишимо умову – якщо не натиснута жодна клавіша горизонтал та герой стоїть на землі, то використовується перша анімація:

```
if (Input.GetAxis("Horizontal") == 0 && (isGrounded)
        {
            anim.SetInteger("State", 1);
        }
```

У іншому випадку, коли одна з клавіш натиснута та герой на землі - використовується друга анімація (ходьби):

```
else
        {
```

Перевірка, яка клавіша натиснута:

```
            Flip();
```

```
            if (isGrounded)
                anim.SetInteger("State", 2);
        }
```

У метод CheckGround додаємо умову. Якщо герой не стоїть на землі, то вмикається третя анімація, що підходить і для трибку і для падіння:

```
if (!isGrounded && !isClimbing)
        anim.SetInteger("State", 3);
```

Так як герой в нас повноцінно рухається, необхідно зробити так, щоб камера рухалась разом з ним. Потрібен новий скрипт спеціально для роботи камери. Оголошуємо в ньому перемінну швидкості та публічну перемінну у якій буде міститись перемінна герою:

```
float speed = 3f;
public Transform target;
```

У методі Start присвоюємо позиції камери позицію герою:

```
void Start()
{
    transform.position = new Vector3(target.transform.position.x, target.transform.position.y, transform.position.z);
}
```

По осі X та Y камера бере координати герою, а останній аргумент залишається незмінним, адже в нас двовимірна гра.

У методі Update створюємо перемінну, що вмістить позицію герою:

```
void Update()
{
    Vector3 position = target.position;
```

По осі Z перемінній присвоюємо позицію камери по тій же осі (це знадобиться при роботі зі шарами, коли один об’єкт буде перекривати інший і тоді будемо змінювати значення Z):

```
    position.z = transform.position.z;
```

Позиції камери, присвоюємо метод, що плавно зменшує відстань між обіектами:

transform.position = Vector3.Lerp(transform.position, position, speed * Time.deltaTime);

}

Перший аргумент позиція звідки починається зближення, тобто об’єкт, що рухається (камера). Другий, куди рухається камера – це перемінна яку ми щойно створили. Третій швидкість приближення, яку помножуємо на Time.deltaTime для більш плавної картинки. Скрипт додаємо до камери.

У перемінну Target вікні скрипту переносимо героя, щоб камера слідкувала за ним.

Зробимо різноманітні платформи. Почнемо з платформи, що рухається від важкості герою. Додаємо на сцену спрайт платформи, робимо три копії та розташовуємо таким чином, щоб вони зливались у одну велику.

Ті що по бокам робимо дочірніми об’єктами середньої та в неї додаємо компоненти колайдер (редагуємо його під спільний розмір), Rididbody 2D (Dynamic, щоб реагували на імпульси, наприклад стрибки), Spring Joint 2D (щоб платформа рухалась немов на пружині по синусоїдальному закону). Координати у строчці Connected Anchor редагуємо так, щоб вони були над платформою. Це немов початок гумки на якій підскакує платформа.

Дублюємо створену платформу, але видаляємо Spring Joint 2D, замість нього додаємо Hinge Joint 2D. Ставимо галочку навпроти Use Motor, у строчці Motor Speed встановлюємо 100 та отримуємо платформу, що кружляє навколо своєї осі з помірною швидкістю.

Тепер беремо один спрайт платформи, додаємо Platform Effector 2D та колайдер, де ставимо галочку напроти Used By Effector. Тепер маємо змогу пригнути на платформу не тільки зі сторони, але й стоячі під нею.

Створимо платформу, що рухається від точки до точки. Беремо платформу, додаємо колайдер та новий скрипт. Оголошуємо масив точок по яким буде рухатись платформа:

```
public Transform[] points;
```

Швидкість платформи:

```
public float speed = 1f;
```

Лічильник для масиву:

```
int i = 1;
void Start()
{
```

Звертаємось до позиції платформи, щоб та мала позицію нульової точки:

```
transform.position = new Vector3(points[0].position.x, points[0].position.y, transform.position.z);
}
```

Створюємо метод, що працює поки герой стоїть в області колайдеру платформи:

```
void OnCollisionStay2D(Collision2D collision)
{
```

Якщо герой встав на платформу, то вона рухається:

```
if (collision.gameObject.tag == "Player")
{
```

Щоб герой рухався разом з платформою оголошуємо дві локальні перемінні яким присвоюємо теперішні позиції платформи по X та Y:

```
float posX = transform.position.x;
float posY = transform.position.y;
```

Перемінні повинно бути дві, на випадок, якщо платформа рухається по діагоналі.

Позиції платформи присуджуємо MoveTowards, що змушує об’єкт рухатись зі своєї позиції в іншу:

```
transform.position = Vector3.MoveTowards(transform.position, points[i].position, speed * Time.deltaTime);
```

У якості аргументів точка з якої відбувається рух, точка до якої платформа рухається та швидкість. До позиції героя додається різниця між теперішньою позицією платформи і старою позицією героя:

```
collision.gameObject.transform.position = new Vector3(collision.gameObject.transform.position.x + transform.position.x - posX, collision.gameObject. transform.position.y + transform.position.y - posY, collision.gameObject. transform.position.z);
```

Якщо платформа біля і-тої точки:

```
if (transform.position == points[i].position)
{
```

Якщо і менше довжини масиву мінус один:

```
if (i < points.Length - 1)
```

То платформа ще не дійшла до останньої точки, а отже продовжує рух:

```
i++;
```

В фншому випадку повертається до нульової точки:

```
else
   i = 0;
```

Скрипт додаємо до платформи, створюємо дві точки на відстані один від одного через вікно іконок.

У налаштуванні скрипту в вікні Inspector встановлюємо кількість точок дві та додаємо їх у перемінні (Рисунок 3.4).

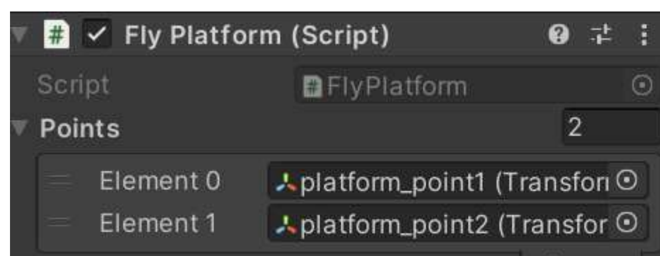

Рисунок 3.4 - Вікно налаштування скрипту платформи

Створюємо платформу, рухом якої управляє гравець. Для цього потрібен новий скрипт. Оголошуємо публічні перемінні для двох точок, що гратимуть роль детектору перешкод:

```
public Transform left, right;
```

Метод, що працює доки колайдери контактують:

```
void OnTriggerStay2D(Collider2D collision)
{
```

Якщо з платформою контактує об’єкт з тегом «Player»:

```
if (collision.gameObject.tag == "Player")
{
```

Створюється лівий луч, який не дозволятиме платформі заїжджати на землю:

```
RaycastHit2D leftWall = Physics2D.Raycast(left.position, Vector2.left, 0.5f);
```

Перший аргумент це ліва точка, другий направлення промінню, а третій його довжина.

Робимо те ж саме для правої сторони:

```
RaycastHit2D rightWall = Physics2D.Raycast(right.position, Vector2.right, 0.5f);
```

Умова з двома зіставними для перевірки чи може платформа рухатись вправо чи вліво:

```
if(((Input.GetAxis("Horizontal")>0)&& !rightWall.collider &&(collision.transform.position.x>transform.position.x))||((Input.GetAxis("Horizontal") < 0) && !leftWall.collider && (collision.transform.position.x < transform.position.x)))
```

У першій частині перевіряється чи може платформа рухатись вправо і чи не зіштовхнулась з об’єктом в якого є колайдер та чи знаходиться герой у центрі платформи чи правіше, друга те ж саме тільки навпаки з лівою стороною.

```
transform.position = new Vector3(collision.transform.position.x, transform.position.y, transform.position.z);
```

Коли герой на платформі нею можна керувати точно так як героєм по горизонталі.

Створимо телепорт. Для цього беремо спрайти двері. Верхню частину робимо дочірнім об’єктом.

До головного спрайта додаємо колайдер, розтягуємо на довжину повної двері, ставимо галочку у графі Is Trigger, щоб герой міг проходити через двері, та створюємо новий скрипт. Оголошуємо перемінну, що вказує відкриті двері чи ні:

```
public bool isOpen = false;
```

Перемінну, що вказуватиме на другі двері, через які виходитиме герой:

```
public Transform door;
```

Перемінні у які внесемо спрайти відчинених дверей:

```
public Sprite mid, top;
}
```

Створюємо метод Unlock, де робимо перемінну isOpen правдивою та міняємо спрайти зачинених дверей, на відчинені:

```
public void Unlock()
{
    isOpen = true;
```

Тут змінюється головний спрайт:

```
    GetComponent<SpriteRenderer>().sprite = mid;
```

Тут дочірній:

```
    transform.GetChild(0).GetComponent<SpriteRenderer>().sprite = top;
}
```

Створюємо метод телепортації з власною перемінною, що є об’єктом player:

```
public void Teleport(GameObject player)
{
```

Присвоюємо позиції героя, позицію другої двері:

```
    player.transform.position = new Vector3(door.position.x, door.position.y,
player.transform.position.z);
```

Тобто заходячи в одні двері, герой виходить з інших.

Тепер потрібно створити умову при якій двері відчинятимуться. Додаємо спрайт ключа, йому колайдер по розміру з галочкою Is Trigger та створюємо йому тег «Key». Повертаємось до скрипту герою, де оголошуємо перемінні наявності ключа (Від початку гри його немає) та чи може герой телепортуватись (так):

```
public bool key = false;
bool canTP = true;
```

Відкриваємо новий метод, що перевіряє зіткнення колайдерів:

```
void OnTriggerEnter2D(Collider2D collision)
{
```

Якщо герой зіткнувся з об’єктом, у якого тег «Key», то цей об’єкт знищується, а перемінна наявності ключа стає правдивою:

```
if (collision.gameObject.tag == «Key»)
{
    Destroy(collision.gameObject);
    key = true;
}
```

Якщо герой зіткнувся з об’єктом в якого тег «Door» (додаємо до двері тег):

```
if (collision.gameObject.tag == «Door»)
{
```

Та якщо двері відкриті та герой може телепортуватись:

```
if (collision.gameObject.GetComponent<Door>().isOpen && canTP)
{
```

Відбувається телепортація, яку мі створили у скрипті дверей:

```
collision.gameObject.GetComponent<Door>().Teleport(gameObject);
```

Одразу вимикаємо функцію телепортації та запускаємо очікування в секунду, щоб герой не застряв між порталами та міг вийти:

```
canTP = false;
StartCoroutine(Tpwait());
```

Якщо герой має ключ, то двері відчиняються:

```
            else if (key)
            {
                collision.gameObject.GetComponent<Door>().Unlock();
            }
        }
```

Створюємо корутину очікування між телепортаціями:

```
Ienumerator Tpwait()
    {
        yield return new WaitForSeconds(1f);
        canTP = true;
    }
```

Зберігаємо скрипти, додаємо новий скрипт дверям та копіюємо об’єкт. Оригінал вносимо у строку Door в скрипті копії, а копію у таку ж строку в оригіналі. До строк Mad та Top додаємо спрайти нижньої та верхньої частини відкритої двері (Рисунок 3.5).

Рисунок 3.5 - Налаштування скрипту дверей

Створюємо воду. Для початку потрібно Додати відповідний скрипт та продублювати дев’ять разів, потім розмітити у два ряди один до одного, згори додаємо п’ять спрайтів хвилей. Зробимо центральний спрайт головним, помістивши в нього навколишні. У головний додаємо колайдер та збільшуємо на всю площу води. Ставимо галочки навпроти Is Trigger та Used By Effector, додаємо Buoyancy Effector 2D, що відповідає за фізику води. У строчці Density пишемо нуль, щоб герой перебував на дні та йому було більш важко вибратись з води. Якщо збільшити значення, то герой буде плавати на більш високому рівні. Створюємо скрипт для води. Створюємо дві перемінні зі значенням нуль. Перша таймер для анімації хвиль, друга таймер втрати життя у воді:

```
float timer = 0f;
float timerHit = 0f;
void Update()
{
```

Додаємо до таймеру час, що пройшов між теперішнім кадром та останнім:

```
    timer += Time.deltaTime;
```

Time – системний клас, а deltaTime системна перемінна.

Якщо пройшло 2 секунди чи більше, анулюємо таймер:

```
    if (timer >= 2f)
    {
        timer = 0;
```

Звертаємося до розміру води та присвоюємо новий вектор зі значенням -1 по X:

```
        transform.localScale = new Vector3(-1f, 1f, 1f);
```

В іншому випадку, якщо в таймері значення дорівнює одиниці чи більше, усі аргументи вектору рівні одиницям:

```
    else if(timer>=1f)
        transform.localScale = new Vector3(1f, 1f, 1f);
}
void OnTriggerStay2D(Collider2D collision)
{
```

Якщо до води потрапляє герой, то значення,що вказує на знаходження у воді (значення в скрипті герою) стає правдивим:

```
    if (collision.gameObject.tag == «Player»)
    {
        collision.GetComponent<Player>().inWater = true;
```

До таймеру втоплення додаємо час між кадрами. Якщо пройшло дві чи більше секунди, то герой втрачає життя, а таймер анулюється:

```
        timerHit += Time.deltaTime;
        if (timerHit >= 2f)
```

```
        {
            collision.gameObject.GetComponent<Player>().RecountHp(-1);
            timerHit = 0;
```

Відкриваємо метод, що спрацьовує, коли об’єкт виходить із тригеру, тобто коли герой покине воду, значення inWater буде невірним, а таймер втоплення повернеться до нуля:

```
void OnTriggerExit2D(Collider2D collision)
{
    if (collision.gameObject.tag == «Player»)
    {
        collision.GetComponent<Player>().inWater = false;
        timerHit = 0;
```

У коді герою створюємо перемінну, що викликали в новому скрипті:

```
public bool inWater = false;
```

Від початку вона не вірна, бо герой не в воді. У методі Update створюємо умову за якої, якщо герой у воді, то герой змінює анімацію на четверту:

```
if (inWater)
    {
        anim.SetInteger(«State», 4);
```

Дозволяємо стрибати у воді:

```
        isGrounded = true;
```

Якщо натиснута одна з клавіш горизонталі, то спрацьовує метод, що буде змінювати напрям погляду героя у той бік, куди він направляється:

```
        if (Input.GetAxis(«Horizontal») != 0)
            Flip();
```

Створимо драбину для місць, куди герою не достривнути. Поєднуємо між собою чотири спрайти драбини як робили це з платформами, але не по горизонталі, а по вертикалі.

Додаємо колайдер (з галочкою біля Is Trigger) по розміру драбини та тег, щоб використати його у скрипті герою, до якого переходимо. Створюємо перемінну, що вказує чи користується герой драбиною (від початку гри ні):

```
bool isClimbing = false;
```

Створюємо метод, що працює лише, коли герой знаходиться в області тригеру:

```
void OnTriggerStay2D(Collider2D collision)
  {
```

Якщо герой зайшов в область колайдеру об’єкту з тегом «Ladder», то перемінна, що вказує на це стає вірною:

```
if (collision.gameObject.tag == «Ladder»)
{
  isClimbing = true;
```

Робимо компонент Rigidbody 2D Кінематичною (була динамічною), щоб на героя не діяла сила тяжіння і він міг піднятися вгору:

```
rb.bodyType = RigidbodyType2D.Kinematic;
```

Якщо герой не лізе вгору чи вниз, то спрацьовує п’ята анімація (герой стоїть тримаючись за драбину):

```
if (Input.GetAxis(«Vertical») == 0)
```

Input.GetAxis(«Vertical») – зчитує використання клавіш пертикалі.

```
{
  anim.SetInteger(«State», 5);
}
```

Коли одна з клавіш вертикалі натиснута, спрацьовує шоста анімація (підійманн, спускання по драбині):

```
else
{
  anim.SetInteger(«State», 6);
```

Звертаємося до методу, що транслює героя (знаходиться в об’єкті transform):

transform.Translate(Vector3.up * Input.GetAxis(«Vertical»)*speed*0.5f* Time.deltaTime);

Змінюємо позицію героя. У якості аргументу вектор напряму, помножений на швидкість, 0.5 (щоб на драбині був в два рази повільніший) та Time.deltaTime .

Якщо натиснута клавіша вгору, то вектор дорівнює одиниці, а якщо вниз дорівнює мінус одиниці.

Створюємо герою три анімації (плавання, тримання за драбину та користування нею) та нумеруємо їх як робили це раніше.

Тепер створюємо батут для більш швидкого підйому. Обираємо потрібний спрайт, додаємо йому два колайдери. Один звичайний (головний), щоб батут був матеріальний, другий для стрибка (дочірній). Останньому додаємо тег «Trampoline».

Створюємо фізичний матеріал. Для цього правою кнопкою миші натискаємо у вікні Project, потім Create, 2D та Physics Materials. Строка Friction відповідає за силу тертя, а строка Bounciness за прижки. В другій строчці ставимо одиницю. Вставляємо у строку Material в дочірньому колайдері. Тепер додаймо дві анімації батуту. Одна для стану покою, друга для стрибку. У вікні Animator створюємо перемінну логічного типу під назвою «isJump». Для анімації покою встановлюємо значення false, для анімації стрибку true. В скрипті герою відкриваємо новий метод, що працює тільки при вході в область колайдеру:

void OnCollisionEnter2D(Collision2D collision)

{

Якщо герой увійшов в область колайдеру з тегом «Trampoline», то почати корутину, яка запустить анімацію батьківського об’єкту:

if (collision.gameObject.tag == «Trampoline»)

StartCoroutine(TrampolineAnim(collision.gameObject.GetComponentInParent<Animator>()));

Створюємо корутину з перемінною в яку записуємо компонент Animator батьківського об’єкту:

```
Ienumerator TrampolineAnim(Animator an)
{
```

Робимо значення isJump правдивим і відповідно починається анімація стрибку:

```
an.SetBool(«isJump», true);
```

Проходить пів секунди, значення стає брехливим і починається анімація покою:

```
yield return new WaitForSeconds(0.5f);
an.SetBool(«isJump», false);
```

Тепер додамо об’єкт, що мають вплив на інший об’єкт. Наприклад гиря падає на коробку, що висіла у повітрі і та падає під вагою. Обираємо спрайт гирі, додаємо колайдер та компонент Rigidbody 2D. У строчці Mass (Вага) пишемо 20. Беремо спрайт з коробкою, додаємо колайдер, Rigidbody 2D з динамікою (щоб імпульси впливали) та Fixed Joint 2D. У строчці Break Force ставимо 2000 – це сила, яку потрібно прикласти, щоб на об’єкт знов діяла сила тяжіння. Тобто герой скидає гирю на коробку і вона падає.

Створимо кнопку, при натисканні на яку відкривається прохід. Беремо новий спрайт, додаємо колайдер, Rigidbody 2D динамічний та тег «MarkBox». Цей об’єкт герой буде скидувати на кнопку, щоб натиснути її. До спрайту кнопки додаємо колайдер та створюємо для неї крипт. Оголошуємо масив з об’єктів:

```
public GameObject[] block;
```

Перемінна зі спрайтом, який з’являтиметься замість оригінального після натискання кнопки:

```
public Sprite btnDown;
void OnCollisionEnter2D(Collision2D collision)
{
```

Якщо у колайдер кнопки, увійшов об’єкт з тегом «MarkBox», то звертаємось до компоненту «SpriteRenderer», а саме до перемінної «sprite» та присвоюємо створену раніше переміну (у яку пізніше занесемо спрайт натиснутої кнопки):

```
if (collision.gameObject.tag == «MarkBox»)
```

```
{
  GetComponent<SpriteRenderer>().sprite = btnDown;
```

Робимо неактивним круглий колайдер кнопки:

```
  GetComponent<CircleCollider2D>().enabled = false;
```

Звертаємося до об'єктів масиву:

```
  foreach (GameObject obj in block)
```

У дужках тип перемінних, їх ім'я, а після «in» назва масиву.

```
  {
```

Знищуємо кожен об'єкт у масиві:

```
    Destroy(obj);
```

Вносимо спрайт натиснутої кнопки у строку Btn Down та об'єкти землі у масив Block. Перед цим вказуємо кількість елементів.

Робимо монети. У методі OnTriggerEnter2D скрипту герою створюємо нову умову. Якщо герой зіткнувся с об'єктом в якого тег «Coin», то монета знищується, їх кількість збільшується на один, а кількість виводиться у консоль:

```
if (collision.gameObject.tag == «Coin»)
{
  Destroy(collision.gameObject);
  coins++;
  print(«Количество монет равно « + coins);
}
```

Спрайту монети додаємо колайдер з Is Trigger та тег.

Додаткові життя. Зі спрайтом серця робимо те ж саме, тільки тег «Heart». У тому ж методі пишемо умову. Якщо герой зіткнувся з сердечком, то об'єкт знищується та додається життя:

```
if (collision.gameObject.tag == «Heart»)
{
  Destroy(collision.gameObject);
  RecountHp(1);
```

```
}
```

У метод RecountHp додаємо умову. Якщо кількість життів максимальна, то життя не додається:

```
else if (curHp > maxHp)
{
    curHp = maxHp;
}
```

По такому ж принципу створюємо мухомор, а в методі OnTriggerEnter2D пишемо умову. Якщо герой зіткнувся з об'єктом в якого тег «MushroomRed», то гриб знищується, а герой втрачає життя:

```
if (collision.gameObject.tag == «MushroomRed»)
    {
        Destroy(collision.gameObject);
        RecountHp(-1);
    }
```

Бонуси. Їх буде два, додаємо два алмази на екран та додаємо їм колайдери, теги «BlueGem» та «GreenGem».

Створюємо умову, якщо зіткнувся з об'єктом в якого тег «BlueGem», об'єкт знищується та починається корутина невразливості:

```
if (collision.gameObject.tag == «BlueGem»)
{
    Destroy(collision.gameObject);
    StartCoroutine(NoHit());
}
```

Оголошуємо три перемінні. Одна вказує, що герой вразлив до ударів, у другу додаватимемо бонуси, третя підраховує кількість бонусів (від початку їх нуль):

```
bool canHit = true;
public GameObject blueGem, greenGem;
int gemCount = 0;
```

Одразу попіклуємось про їх відображення над героєм, щоб гравець бачив діє бонус чи вже ні. Створюємо новий метод:

void CheckGems(GameObject obj)

{

Якщо в героя один бонус, то він відображатиметься на 0.6 квадрату вище точки центру героя:

if (gemCount == 1)

obj.transform.localPosition = new Vector3(0f, 0.6f, obj.transform. localPosition.z);

Якщо бонуси два, то голубий на пів клітинки лівіше та пів клітинки вище від точки центру героя, а зелений на пів клітинки правіше:

else if (gemCount == 2)

{

blueGem.transform.localPosition = new Vector3(-0.5f, 0.5f, blueGem. Transform.localPosition.z);

greenGem.transform.localPosition = new Vector3(0.5f, 0.5f, greenGem. Transform.localPosition.z);

Створюємо корутину зникнення бонусів над героєм. В якості перемінних SpriteRenderer, що маніпулювати видимістю об’єктів та час між сусідніми кадрами:

Ienumerator Invis(SpriteRenderer spr,float time)

{

Першій перемінній присвоюємо кольор в якого є чотири аргументи три з яких кольори, а останній видимість мінус час помножений на 2:

spr.color = new Color(1f, 1f, 1f, spr.color.a – time * 2);

Чекаємо до нового кадру:

yield return new WaitForSeconds(time);

Корутина триває доки альфа-канал не дорівнюватиме нулю (тобто доки бонус не стане невидимим):

if (spr.color.a > 0)

StartCoroutine(Invis(spr, time));

Створюємо корутину для бонусу невразливості:

Ienumerator NoHit()

{

Кількість бонусів збільшується на один:

gemCount++;

Активуємо іконку голубого алмазу над героєм:

blueGem.SetActive(true);

Метод присуджуємо перемінній методу CheckGems значення blueGem:

CheckGems(blueGem);

Робимо героя невразливим до ударів:

canHit = false;

Гем над головою видно напротязі п’яти секунд і потім запускається корутина невидимості:

```
blueGem.GetComponent<SpriteRenderer>().color = new Color(1f, 1f, 1f, 1f);
    yield return new WaitForSeconds(5f);
  StartCoroutine(Invis(blueGem.GetComponent<SpriteRenderer>(), 0.02f));
```

Чекаємо секунду і героя знов можна бити:

yield return new WaitForSeconds(1f);

canHit = true;

Кількість алмазів зменшується на один:

gemCount--;

Голубий алмаз не видно:

blueGem.SetActive(false);

Вносимо в якості аргументу методу зелений алмаз.:

CheckGems(greenGem);

У разі, якщо його немає, нічого не відбуватиметься, а якщо є, то він буде над головою героя.

Бонус подвійної швидкості. Створюємо корутину:

Ienumerator DoubleSpeed()

```
{
```

Збільшуємо кількість алмазів на один:

```
gemCount++;
```

Активуємо зелений алмаз над головою:

```
greenGem.SetActive(true);
```

Додаємо його у метод:

```
CheckGems(greenGem);
```

Збільшуємо швидкість героя вдвічі:

```
speed = speed * 2;
```

Зелений алмаз видно на сто відсотків:

```
greenGem.GetComponent<SpriteRenderer>().color = new Color(1f, 1f, 1f, 1f);
```

Очікування в дев'ять секунд та починається корутина зникнення:

```
yield return new WaitForSeconds(9f);
StartCoroutine(Invis(greenGem.GetComponent<SpriteRenderer>(), 0.02f));
```

Через секунду швідкість повертається до норми:

```
yield return new WaitForSeconds(1f);
speed = speed / 2;
```

Кількість алмазів зменшується на один:

```
gemCount--;
```

Зелений алмаз не активний:

```
greenGem.SetActive(false);
```

Перевірка на випадок, якщо є голубий алмаз:

```
CheckGems(blueGem);
```

У налаштуванні скрипту у вікні Inspector додаємо алмази. Тепер в нас є два бонуси наявність яких відображається над героєм. Якщо бонус один, то він по центру, якщо два то вони по бокам.

Зробимо ворога з якого випадають бонуси при його вбивстві. Робимо звичайного ворога, що переміщається по землі. Відкриваємо скрипт для ворогів та оголошуємо публічну перемінну в яку будемо вносити бонус, що випадатиме:

```
public GameObject drop;
```

У корутині смерті додаємо умову. Якщо в перемінній drop є об’єкт, то викликаємо метод, що створить цей об’єкт на сцені:

```
if (drop != null)
{
    Instantiate(drop, transform.position, Quaternion.identity);
```

Перший аргумент це об’єкт, що з’явиться, другий позиція де він створиться (відбудеться це там, де помре ворог), третій це кути повороту об’єкта.

В налаштуванні скрипту додаємо до перемінної drop будь-який бонус, монету чи життя і при вбивстві отримаємо його.

3.2 Створення ворогів різного типу

Створюємо новий скрипт, що згодиться для усіх ворогів. Головна задача цього коду при контакті з героєм віднімати в нього життя.

Спочатку герою додаємо тег під назвою Player, щоб через нього звертатись (Рисунок 3.6).

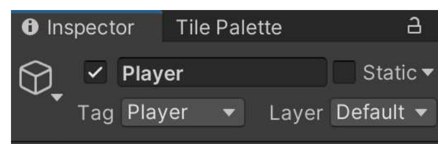

Рисунок 3.6 - Наділення героя тегом

Створюємо метод, що фіксує зіткнення з об’єктом:

```
void OnCollisionEnter2D(Collision2D collision)
{
```

Якщо відбулось зіткнення с об’єктом, що має тег «Player», то у консолі виводиться надпис про це:

```
if (collision.gameObject.tag == «Player»)
{
```

```
print(«Минус одна жизнь.»);
```

Щоб ворог дійсно наносив шкоду герою, потрібно створити систему життів. Для сього у скрипті герою оголошуємо перемінну, яка вказуватиме на дійсну кількість життів та перемінну, що вказуватиме на максимальну кількість життів (щоб при зборі додаткових життів в грі гравець не зміг зібрати їх більше трьох):

```
int curHp;
int maxHp = 3;
```

Додатково оголосимо перемінну логічного типу, що вказуватиме вразливість герою:

```
bool canHit = true;
```

Вона знадобиться нам, коли будем створювати бонус невразливості. З самого початку значення вірне, щоб герой втрачав житті від контактів з ворогом.

У методі, що запускається лише напочатку гри прирівнюємо першу перемінну до другої:

```
curHp = maxHp;
```

Таким чином спочатку гри в нас буде максимальна кількість життів.

Створюємо новий метод для перерахунку кількості життів, що прийматиме власний аргумент типу int:

```
public void RecountHp(int deltaHp)
{
```

Якщо вразливість увімкнена (тобто в героя немає бонусу невразливості):

```
if (canHit == true)
{
```

До поточної кількості життів додаємо нову перемінну:

```
curHp = curHp + deltaHp;
```

Далі переходимо до скрипту для ворогів та звертаємось до герою, що є об'єктом зіткнення та визиваємо щойно створений метод зі значенням в аргументі «-1»:

```
collision.gameObject.GetComponent<Player>().RecountHp(-1);
```

Таким чином при зіткненні відніматиметься життя. Через крипт герою виведемо інформацію про кількість життів у консоль:

```
print(curHp);
```

Кількість життів зменшується на один при зіткненні, але герой не помирає. Для цього створюємо умову, якщо кількість життів менше чи дорівнює нулю, то відбувається смерть:

```
if (curHp <= 0)
{
```

.enabled означає активацію об’єкту. Прирівнюємо її брехні, щоб дезактивувати колайдер герою і він впав під землю:

```
GetComponent<CapsuleCollider2D>().enabled = false;
```

Зробимо так, щоб при зіткненні з ворогом, герой відскакував. Таким чином гравець бачитиме, що отримав удар. Повертаємось до крипту ворога та звертаємось до компоненту Rigidbody2D:

```
collision.gameObject.GetComponent<Rigidbody2D>().AddForce(transform.up * 8f, ForceMode2D.Impulse);
```

Створюємо імпульс по тому ж принципу, що наділяли гравця здатністю стрибати, тільки довжину стрибку вводимо цифрою, а не перемінною.

Та зробимо, щоб герой червонів при ударі. Переходимо до скрипту герою та оголошуємо нову перемінну, що вказуватиме чи підвергається у цей час герой удару:

```
bool isHit = false;
```

Від початку значення не вірне, бо героя ніхто не б’є.

Створюємо корутину підрахунку ударів (на відміну від методу Update вона викликатиметься через певний час, який ми задамо) :

```
Ienumerator OnHit()
{
```

Якщо є зіткнення, то з кольорів RGB залишається лише червоний:

```
if (isHit)
```

```
GetComponent<SpriteRenderer>().color = new Color(1f, GetComponent<SpriteRenderer>().color.g – 0.04f, GetComponent<SpriteRenderer>().color.b – 0.04f);
```

В інший час повертається оригінальні кольори героя:

```
else
GetComponent<SpriteRenderer>().color = new Color(1f, GetComponent<SpriteRenderer>().color.g + 0.04f, GetComponent<SpriteRenderer>().color.b + 0.04f);
```

Я кощо герой нормального кольору, то зупиняється корутина:

```
if (GetComponent<SpriteRenderer>().color.g == 1f)
StopCoroutine(OnHit());
```

Якщо зеленого немає (а значить синього також), то тільки що героя вдарили, а значить перемінну, що вказує на підтвердження героя удару робимо невірною (такою вона залишиться до наступного удару):

```
if (GetComponent<SpriteRenderer>().color.g <= 0)
isHit = false;
```

Скрипт чекає 0.02 секунди:

```
yield return new WaitForSeconds(0.02f);
```

Та знов запускає корутину:

```
StartCoroutine(OnHit());
}
```

Теперь у методі RecountHp після строчки «curHp = curHp + deltaHp;» додаємо умову, що запустить корутину. Якщо перемінна методу негативна (якщо відбувається віднімання життя), то героя б’ють та запускається корутина (герой червоніє):

```
if (deltaHp < 0)
        {
           isHit = true;
           StartCoroutine(OnHit());
        }
```

Помираючи герой падає вниз. Так як дно в програмі відсутнє, він падатиме вічно. Що пере запускати гру після смерті створюємо новий скрипт. У

перечисленні бібліотек в самому початку додаємо нову для роботи зі сценами (рівнями):

```
using UnityEngine.SceneManagement;
```

У класі створюємо публічний метод де звертаємось до компоненту, що відповідає за роботу зі сценами та має метод, який запускає сцену:

```
public void Lose()
{
SceneManager.LoadScene(SceneManager.GetActiveScene().name);
 }
```

Далі повертаємось до скрипту герою, оголошуємо публічну перемінну, щоб мати змогу налаштовувати її у вікні Inspector:

```
public Main main;
```

Створюємо новий метод, звертаємось до компоненту нової перемінної та викликаємо метод програшу з іншого класу:

```
void Lose()
    {
        main.GetComponent<Main>().Lose();
    }
```

Після строки, що вимикає колайдер герою (у зв’язку зі смертю), запускаємо створений метод, після кому записуємо час, через який від запуститься:

```
Invoke(«Lose», 1.5f);
```

Скрипт Main додаємо до камери, а камеру заносимо у строку Main у вікні налаштування скрипту герою. Тепер після смерті проходить півтори секунди та гра перезапускається.

Скрип Enemy наразі можна додати статичним ворогам, наприклад шипам.

Тепер створюємо рухливих ворогів. Анімація робиться як і для героя. Різниця лише в їх кількості. Будуть вороги яким герой нічого не може зробити. В таких буде лише анімація ходьби чи польоту. Тобто вікно Animator не

знадобиться. В іншому випадку знадобиться друга анімація – смерті. Це для ворогів, яких герой може вбити, стрибнув на них.

Тих, що невразливі залишаємо, іншим створюємо перемінну dead у вікні Animator, як робили це с героєм, але там був тип цілих чисел. У графі Conditions додаємо створену перемінну. Для анімації руху значення перемінної – брехня, а для анімації смерті – істина.

Ворогу додаємо дочірній об’єкт з квадратним колайдером, що переміщуємо над ворогом.

Створюємо новий скрипт для вбивства. Відкриваємо метод, що зчитує контакт колайдерів (об’єктв)

```
void OnCollisionEnter2D(Collision2D collision)
    {
```

Якщо зіткнення відбулось з об’єктом у якого тег «Player», то він трохи підскакує, а починається корутина смерті, що знаходиться у скрипті Enemy:

```
        if (collision.gameObject.tag == «Player»)
        {
collision.gameObject.GetComponent<Rigidbody2D>().AddForce(transform.up * 6f, ForceMode2D.Impulse);
            gameObject.GetComponentInParent<Enemy>().startDeath();
        }
    }
```

Повертаємось до скрипту Enemy. Оголошуємо нову перемінну, що вказуватиме чи наніс удар герой:

```
bool isHit = false;
```

У методі OnCollisionEnter2D розширюємо вже існуючу умову. Якщо ворог зіткнувся з героєм та це був удар не з його сторони, то відіймається життя:

```
if (collision.gameObject.tag == «Player» && !isHit)
```

Створюємо корутину:

```
public Ienumerator Death()
    {
```

Робимо перемінну істинною:

```
isHit = true;
```

Перемінну, що створили у вікні Animator робимо істинною:

```
GetComponent<Animator>().SetBool(«dead», true);
```

Звертаємось до компоненту Rigidbody2D та робимо об’єкт динамічним, щоб на нього спрацювала сила тяжіння та він впав:

```
GetComponent<Rigidbody2D>().bodyType = RigidbodyType2D.Dynamic;
```

Робимо колайдер не активним:

```
GetComponent<Collider2D>().enabled = false;
```

Малий колайдер (дочірній) також робимо не активним:

```
transform.GetChild(0).GetComponent<Collider2D>().enabled = false;
```

Через 2 секунди ворог зникає з екрану:

```
yield return new WaitForSeconds(2f);
Destroy(gameObject);
    }
```

Так як викликати корутину можна лише всередині її класу, то викликаємо її в новому методі, а вже метод визиватимемо в іншому скрипті:

```
public void startDeath()
{
    StartCoroutine(Death());
```

Коли анімації готові, беремо ворога, що переміщається по землі та додаємо до нього дочірній об’єкт. Робимо його видимим як groundCheck для героя, тільки обираємо будь-яку іншу іконку (Рисунок 3.7).

Рисунок 3.7 - Ворог із об’єктом, що перевірює наявність землі під ним

Створюємо новий скрипт, що буде перевіряти наявність землі під ворогом. Якщо земля закінчується, то ворог повертається та йде туди звідки прийшов, далі така ж схема. Оголошуємо три публічні перемінні – одна відповідає за швидкість (це значення можна буде змінити вікні Inspector), друга логічна допоможе змінювати напрям ворога, третя перевіряє наявність землі внизу:

```
public float speed = 1.5f;
public bool moveLeft = true;
public Transform groundDetect;
```

Ми бажаємо, щоб об'єкт переміщався з одної точки в іншу, для цього потрібен метод Translate, який приймає вектор, що відповідає за напрям та швидкість:

```
transform.Translate(Vector2.left * speed * Time.deltaTime);
```

Time.deltaTime для плавності руху.

Пишемо команду, що створює промінь із точки (в нашому випадку іконка, що створили раніше) у напрямі що ми задамо. Далі нова перемінна, якій присвоюється луч, що міститься у двовимірній фізиці. У дужках вписуємо початок промінню, напрям та довжину:

```
RaycastHit2D groundInfo = Physics2D.Raycast(groundDetect.position, Vector2.down, 1f);
```

Таким чином враховуватись буде лише та платформа на якій знаходиться ворог, а ті, що нижче ні.

Якщо промінь не торкається землі, а ворог рухається вліво, то змінює напрям вправо:

```
if(groundInfo.collider == false)
{
   if (moveLeft == true)
   {
      transform.eulerAngles = new Vector3(0, 180, 0);
      moveLeft = false;
```

eulerAngles дає змогу вказати кут в градусах. Вказав другий аргумент 180 градусів, повертаємо по горизонталі об’єкт.

В іншому випадку нічого не змінюється та ворог рухається вліво:

```
else
{
   transform.eulerAngles = new Vector3(0, 0, 0);
   moveLeft = true;
```

Цей скрипт додаємо до усіх об’єктів, що повинні ходити по землі. На черзі ворог, що літає. Необхідно створити дві точки.

Об’єкт буде рухатись від одної до другої, тому оголошуємо дві публічні перемінні:

```
public Transform point1;
public Transform point2;
```

Далі швидкість, та час очікування перед тим як полетіти до іншої точки:

```
public float speed = 2f;
```

Швидкість можна змінити у будь-який час. Додати скрипт до різних ворогів та кожному своє значення ввести.

```
Void Start()
{
```

Щоб ворог починав рух від першої точки, присвоюємо йому її значення:

```
gameObject.transform.position = new Vector3(point1.position.x, point1.position.y, transform.position.z);
void Update()
{
```

Якщо об’єкт може рухатись, то нехай рухається до першої точки. Серед аргументів теперішня позиція, позиція першої точки та швидкість:

```
transform.position = Vector3.MoveTowards(transform.position, point1.position, speed * Time.deltaTime);
```

Якщо об’єкт і так вже на першій точці, то міняємо дві точки місцями, а точніше їх назви. Таким чином ворог завжди рухатиметься від другої точки до

першої. Просто спочатку нижня буде першою і об’єкт полетить вниз, опинившись внизу, полетить вгору, бо вже там буде перша точка.

```
If (transform.position == point1.position)
{
```

Привласнюємо перемінну першої точки новій перемінній того ж типу:

```
Transform t = point1;
```

Перемінній першої точки привласнюємо перемінну другої:

```
point1 = point2;
```

Перемінній другої точки привласнюємо нову перемінну.

```
Point2 = t;
```

Тобто тепер нова перемінна вільна і на наступному циклі їй знов можна буде привласнити перемінну першої точки.

У строчки двох перемінних переносимо створені раніше точки, а самі точки розміщуємо на відстані один від одного.

Є ще один варіант літаючого ворога. Суть така ж, але йому можна поставили будь-яку кількість точок для переміщення.

Оголошуємо масив точок, по яким буде рухатись ворог, перемінні швидкості, очікування біля нової точки, логічну, що вказує може чи ні рухатись об’єкт та цілочисельну, що задаватиме кількість точок:

```
public Transform[] points;
public float speed = 2f;
public float waitTime=1f;
bool CanGo = true;
int I = 1;
void Start()
{
```

У самому початку гри присвоюємо позицію нульової точки

```
gameObject.transform.position = new Vector3(points[0].position.x, points[0].position.y, transform.position.z);
}
```

void Update()

{

Якщо ворог може рухатись (від початку встановлено, що може), то рухається від теперішньої точки, то і-тої:

if (CanGo)

transform.position = Vector3.MoveTowards(transform.position, points[i].position, speed * Time.deltaTime);

Якщо позиція збігається з позицією і-тої точки, та «і» на один менше довжини масиву, то значення перемінної «і» збільшується на один:

```
if (transform.position == points[i].position)
{
   if (I < points.Length – 1)
      i++;
   else
      i = 0;
```

Об’єкт зупиняється та починається корутина очікування:

```
CanGo = false;
StartCoroutine(Waiting());
```

Проходить одна секунда та ворог знов може рухатись:

```
Ienumerator Waiting()
{
   yield return new WaitForSeconds(waitTime);
   CanGo = true;
```

Створюємо ворога, що стріляє. Він буде літати, тому додаємо йому скрипт польоту між двома точками та створюємо новий, що відповідатиме за постріли. Оголошуємо три перемінні. Перша сам снаряд:

```
public GameObject bullet;
```

Друга вказує з якої точки відбуваються постріли:

```
public Transform shoot;
```

Третя періодичність пострілів:

```
public float timeShoot = 4f;
void Start()
{
```

Початкову позицію снаряду прирівнюємо позиції ворога, але на клітинку нижче:

```
shoot.transform.position = new Vector3(transform.position.x, transform.position.y – 1f, transform.position.z);
```

Оголошення корутини, що відповідає за періодичність пострілів:

```
StartCoroutine(Shooting());
Ienumerator Shooting()
{
```

Очікування перед новим пострілом (час можна міняти у вікні налаштування скрипту):

```
yield return new WaitForSeconds(timeShoot);
```

Створюємо снаряд за допомогою нового методу:

```
Instantiate(bullet, shoot.transform.position, transform.rotation);
```

Перший аргумент це наш снаряд, другий – його початкове положення, третій – кут нахилу. В даному випадку він збігається з положенням ворога. Початок корутини:

```
StartCoroutine(Shooting());
}
```

Додаємо ворогу скрипт та створюємо снаряд. Для цього беремо невеликий спрайт, який згодиться для нашої задачі, додаємо йому круглий колайдер та скрипт Enemy, щоб той наносив шкоду. Тепер змусимо їх падати – створюємо новий скрипт та оголошуємо перемінні швидкості руху вниз та час зникнення снаряду:

```
float speed=3f;
float TimeToDisable = 10f;
void Start()
{
```

Починаємо корутину зникнення снаряду:

```
        StartCoroutine(SetDisabled());
    }
    void Update()
    {
```

Направляємо снаряди вниз:

```
        transform.Translate(Vector2.down * speed * Time.deltaTime);
    }
    Ienumerator SetDisabled()
    {
```

Очікування напротязі 10 секунд:

```
        yield return new WaitForSeconds(TimeToDisable);
```

Робимо снаряд неактивним:

```
        gameObject.SetActive(false);
    }
```

Створюємо системний метод, що оброблятиме зіткнення:

```
    void OnCollisionEnter2D(Collision2D collision)
    {
```

Зупинка корутини зникнення об’єктів

```
        StopCoroutine(SetDisabled());
```

Робимо снаряди, що контактували з героєм неактивними:

```
        gameObject.SetActive(false);
```

Цей скрипт додаємо до снаряду, а сам снаряд вносимо до нової папки Prefabs. Потім вносимо об’єкт до строки Bullet. А до перемінної shoot додаємо пустий дочірній (для ворога) об’єкт. Тепер ворог стріляє снарядами.

Створюємо порога, що підіймається із-під землі. У новому скрипті оголошуємо перемінну швидкості руху монстру:

```
        public float speed = 4f;
```

Перемінна, що вказує чикає чи ні монстр (від початку не чекає):

```
        bool isWait = false;
```

Перемінна, що вказує чи схован монстр від землею (від початку ні):

```
bool isHidden = false;
```

Час очікування монстра:

```
public float waitTime = 4f;
```

Місце схову:

```
public Transform point;
void Start()
{
```

Від початку гри монстр на одну клітинку вище точки, що допомагає йому в орієнтуванні:

```
point.transform.position = new Vector3(transform.position.x, transform.position.y + 1f, transform.position.z);
}
void Update()
{
```

Якщо монстр не очікує, то ховається під землю:

```
if (isWait == false)
    transform.position = Vector3.MoveTowards(transform.position, point.position, speed * Time.deltaTime);
```

Якщо він під землею, та якщо схован (подвійна перевірка), то вилазить на світ:

```
if (transform.position == point.position)
{
    if (isHidden)
    {
        point.transform.position = new Vector3(transform.position.x, transform.position.y + 1f, transform.position.z);
        isHidden = false;
```

В іншому випадку спускається на клітинку нижче (тобто ховається):

```
    else
```

```
{
    point.transform.position = new Vector3(transform.position.x, transform.position.y – 1f, transform.position.z);
    isHidden = true;
```

Монстр переходить у стан очікування та починається корутина:

```
isWait = true;
StartCoroutine(Waiting());
}
Ienumerator Waiting()
{
```

Через чотири секунди очікування завершається та монстр знов рухається:

```
yield return new WaitForSeconds(waitTime);
isWait = false;
```

Скрипт вносимо до монстра, створюємо точку, яку додаємо до перемінної Point та розміщуємо на клітинку нижче монстра.

Створимо пилу, що рухається на ціпку. Створюємо ворога у вигляді круглої пили та додаємо у якості дочірнього об’єкту два спрайти ланцюгів. До пили додаємо компонент Hinge Joint 2D, переносимо його на початок ланцюга (щоб саме навколо цієї точки відбувалось обертання) та встановлюємо швидкість (Рисунок 3.8).

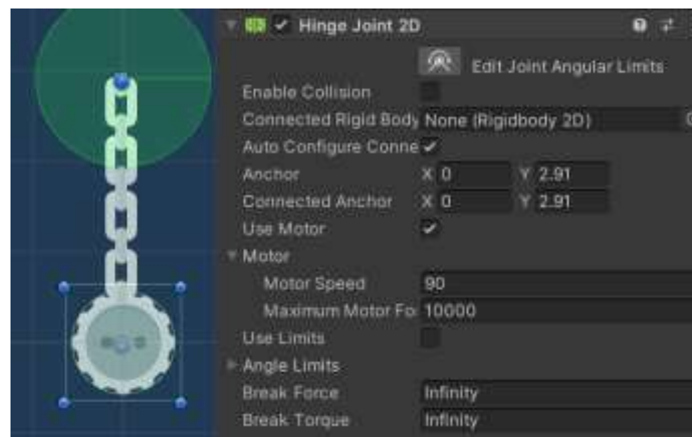

Рисунок 3.8 - Налаштування ворога на ланцюгу

Щоб вороги правильно працювали, кожному потрібен скрипт Enemy, колайдер та компонент Rigidbody2D. Body Type завжди ставимо Kinematic, щоб на нього не впливали імпульси та зіткнення.

Останній ворог це шипи які падають, коли герой під ними. Для спрайту шипів створюємо батьківський пустій об'єкт, шипам додаємо колайдер та створюємо дочірній об'єкт батьківського в якому теж робимо колайдер, але на над землею. Виходить, що спрайт з об'єктом в якого є колайдер знаходяться у пустому об'єкті. Об'єкт з колайдером потрібен для перевірки. Якщо на нього наступили, то шипи падають.

Потрібен новий скрипт для головного. Оголошуємо перемінну перевірки:

```
bool check;
```

Метод, що потім викликатимемо в Update:

```
void CheckMarker()
{
```

Якщо перемінна істинна, то активуємо скрипт дочірнього об'єкту та шипи падають вниз:

```
    if (check == true)
    {
        gameObject.GetComponentInChildren<ShotThorn>().Fly();
    }
}
private void Update()
{
```

Постійно дістаємо значення перемінної з області для перевірки (об'єкт з колайдером на землі):

```
    check = GetComponentInChildren<Marker>().ColEnter
```

Постійний виклик методу, що перевіряє логічну перемінну:

```
    CheckMarker();
}
```

Пишимо новий скрипт для області перевірки. Оголошуємо логічну перемінну, що перевіряє контакт з героєм:

```
public bool ColEnter = false;
private void OnTriggerEnter2D(Collider2D collision)
{
```

Якщо герой увійшов у триггер:

```
if (collision.gameObject.tag == "Player")
{
```

То значення перемінної вірне:

```
ColEnter = true;
```

У шипи додаємо скрипт для ворогів та новий. Направлення шипів:

```
Vector2 shot = new Vector2();
```

Швидкість падіння:

```
public float speedshot;
```

Метод для падіння:

```
public void Fly()
{
```

Переміщаємо шип:

```
    transform.Translate(shot * Time.deltaTime);
}
void Start()
{
```

Вектору Y привласнили значення швидкості:

```
shot.y = speedshot;
```

Значення по осі X не змінюється, до шипи падають рівно вниз:

```
shot.x = 0;
```

3.3 Дизайн рівнів

Для створених об’єктів створюємо папку Prefabs та поміщаємо все у неї, але перед цим скидуємо позиції кожного. Якщо, наприклад, у ворога є точки по

яким він рухається, тоді кожен об'єкт анулюється окремо, потім одна точка відводиться в сторону, усе групується за допомогою пустого батьківського об'єкту та додається у папку.

Тепер із підготовлених прифабів можна створювати саму гру, але спочатку, щоб поверхня по якій ступає герой була рівною, необхідно створити сітку. Вона допоможе правильно розставляти об'єкти. А щоб не копіювати кожен раз спрайти землі (що буде займати забагато часу та дуже незручно), необхідно створити Tilemap (Рисунок 3.9).

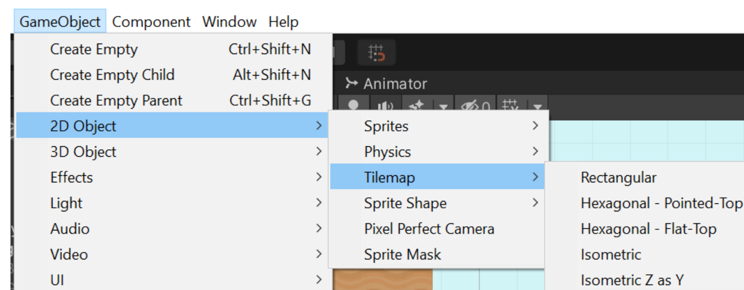

Рисунок 3.9 - Створення Tilemap

У об'єкті Grid в строчці Cell Size пишемо 1.25 по X та Y. Це розмір сітки.

У папці Spritesheets натискаємо на набір спрайтів із землею. У вікні Inspector в строчці Sprite Mode обираємо Multiple та натискаємо Apply. До компоненту Tilemap додаємо Tilemap Collider 2D. Завдяки цьому не потрібно кожен раз створювати нову землю та додавати колайдер.

Тепер у вікні Tile Palette можемо обрати необхідний спрайт, обрав пензлик, чи заливку та «малювати» рівень.

Щоб створити другий рівень можна скопіювати вже існуючий натиснувши кнопку Save as, видалити його вміст та робити по-новому.

3.4 Головне меню

Для створення користувацького інтерфейсу створюємо нову сцену (Рисунок 3.10).

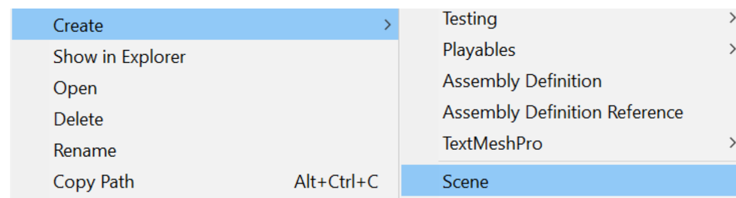

Рисунок 3.10 - Створення нової сцени

Створюємо полотно на якому буде знаходитись меню (Рисунок 3.11). Уся об'єкти мають залежити від нього.

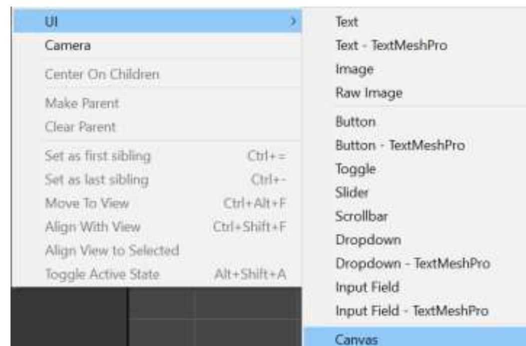

Рисунок 3.11 - Створення Canvas

У строчці Render Mode обираємо Screen Space – Camera (найкращий режем для роботи з інтерфейсом), а в строчку Render Camera вносимо нашу камеру. У Canvas створюємо малюнок. У строчку Source Image додаємо спрайт кнопки. Всередині малюнку створюємо текст та пишемо «Menu». Розміщуємо табличку вгорі та фіксуємо положення через Rect Transform (Рисунок 3.12).

Рисунок 3.12 - Фіксування надпису вгорі по центру екрану

Потім створюємо три кнопки так само, як і минулий об'єкт, тільки замість Image обираємо Button. У тексті пишемо назву кожної кнопки: Play, Settings, Exit. Пишемо на англійській, бо для шрифт, що ми використовуємо не працює

на українській. Щоб зробити кнопку Play робочою, створюємо скрипт. До існуючих бібліотек приписуємо ще одну:

using UnityEngine.SceneManagement;

Створюємо новий метод з перемінною індекс:

public void OpenScene (int index)

{

Звертаємось до нової бібліотеки та запускаємо метод LoadScene:

SceneManager.LoadScene(index);

Додаємо скрипт камері, а камеру вносимо у вікно налаштування кнопки, потім обираємо перемінну індексу у скрипті меню (Рисунок 3.13).

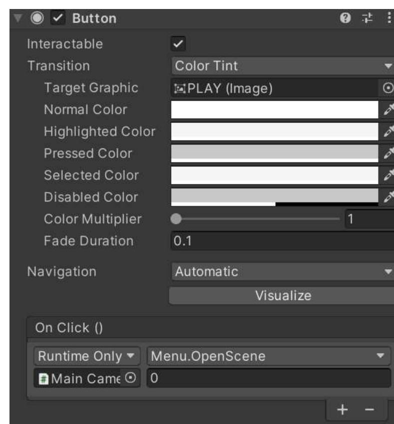

Рисунок 3.13 - Вікно налаштування кнопки

Тепер при натисканні кнопки Play відкривається перший рівень.

Для виводу на екран кількості монет у скрипті герою додаємо новий метод виклик якого повертає кількість монет:

public int GetCoins()

{

return coins;

У скрипті Main додаємо нову бібліотеку для роботи з текстом і картинками в інтерфейсі:

using UnityEngine.UI;

Оголошуємо нові перемінні. Герой:

public Player player;

Текст, що інформуватиме про кількість монет:

```
public Text coinText;
```

Іконки життів:

```
public Image[] hearts;
```

Спрайти з наявним життям та відсутнім:

```
public Sprite isLife, nonLife;
public void Update()
```

Звертаємось до героя, викликаємо метод GetCoins та преобразуємо число у текст:

```
coinText.text = player.GetCoins().ToString();
```

У сцені першого рівня створюємо текст з іконкою монети та розміщуємо у лівому верхньому куту, фіксуємо.

Вставляємо в один пустий об’єкт три іконки серця та фіксуємо у правому верхньому куті. У скрипті герою створюємо метод, виклик якого повертає інформацію про кількість життів:

```
public int GetHP()
{
    return curHp;
```

У скрипті Main запускаємо цикл, що відбуватиметься доки і менше трьох. Починається розрахунок з нуля і кожне коло додається одиниця:

```
for (int i=0; i < 3; i++)
```

Якщо кількість життів більше ніж і, то залишається спрайт з повним серцем, якщо менше то спрайт замінюється на пусте серце:

```
if (player.GetHP() > i)
    hearts[i].sprite = isLife;
else
    hearts[i].sprite = nonLife;
```

Заповнюємо перемінні у вікні налаштування скрипту. Додаємо героя в першу перемінну, в другу текст у якому буде відображатись кількість монет, третя кількість життів та серця які закріпили на екрані, останні місця для спрайтів повного серця та пустого (Рисунок 3.14).

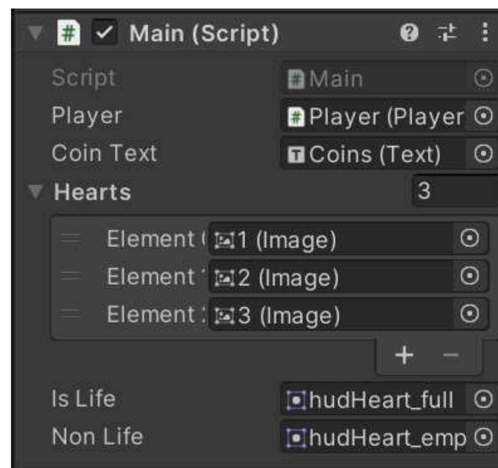

Рисунок 3.14 - Вікно налаштування скрипту Main

У верхньому правому куті робимо кнопку паузи, а по центру меню, що буде викликатись цією кнопкою.

Повертаємось до скрипту Main та оголошуємо перемінну об’єкту PauseScreen:

```
public GameObject PauseScreen;
```

Створюємо метод в якому все зупиняються, герой стає неактивним та вмикається екран паузи:

```
public void PauseOn()
{
    Time.timeScale = 0f;
    player.enabled = false;
    PauseScreen.SetActive(true);
```

У строку Pause Screen (що в налаштуванні скрипту) додаємо екран паузи. З іншими екранами така ж схема. Тільки при вимиканні паузи, запуску меню після програшу та переході на інший рівень час йде та герой активний:

```
Time.timeScale = 1f;
player.enabled = true;
```

При переході на наступний рівень ми не загружаємо екран, а загружаємо сцену, що має індекс на один більше теперішньої сцени:

```
SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex+1);
```

3.5 Висновок по розділу 3

Щоб створити платформер необхідно підготувати спрайти, з яких складатиметься гра та наділити їх певними властивостями. Додати колайдери по формі, щоб не проходили скрізь один одного, компоненти, що відповідають за фізику та головне – скрипти. В одному об’єкті може міститись декілька чи жодного скрипту. Все залежить від його задачі. Код герою найбільший, бо він взаємодіє з усіми спрайтами та на кожен по різному реагує. Щоб анімації спрацьовували у влучний час, вони були прономеровані та вписані в скрипт. Таким чином герой має анімацію ходьби, коли прересувається по землі, плавання, коли у воді, прижку, коли у повітрі, лазання, коли в області драбини. Вороги теж мають анімації. Кожен має анімацію руху (ходьби чи польоту), а деякі ще й смерті.

Також є декоративні об’єкті. Це ті що ніяк не впливають не героя, а лише стоять на певному місці за виду. Гравець взмозі обрати свій шлях до фінішу, чи пройти кожен куток рівня. При бажанні може повернутись назад в рамках одного рівня. Навіть якщо герой не зможе прокласти собі шлях в одному місці, то він зможе обійти та все одно зібрати всі монети.

Інтерфейс доволі простий та повністю зрозумілий на інтуітивному рівні. Навіть в людей, що не знають англійську мову не буде проблем. Наразі можна запустити рівень, дійти до фінішу та перейти на другий. Якщо гравець програє, то має варіант вийти в меню чи попробувати знову. З меню зможе знову запустити перший рівень.Обидва рівні розроблені таким чином, щоб людина, що рідко грає змогла пройти їх без особливих проблем. Навіть падіння з платформи не смертельні для героя, він лише впаде на нижчий рівень та відкриє новий шлях. У другому рівні їх чотири, а в першому два. Зіткнення з ворогом звісно нанесе шкоди, але пастки розставлені з оптимальною періодичністю. До того ж є бонуси невразливості та життя. А бонус швидкості допоможе перепригнути велику кількість ворогів, що стоять поспіль.

ВИСНОВКИ

Відеоігри дуже розвинулись за 80 років. Пристрої для ігор також не стоять на місці. Грати можна не тільки вдома та спеціалізованих закладах, а й будь-де завдяки портативним гаджетам та високошвидкісному інтернету, що наразі доступний навіть в метро та малих селах.

Дана кваліфікаційна робота присвячена розробці власної гри. В результаті проведеного аналізу було вирішено розробляти гру для мобільної платформи, бо цей ринок активно розвивається, щоб заробляти достатньо інтегрувати рекламу (а не чекати, поки гру нарешті помітять та почнуть купляти) та випустити додаток найпростіше саме у мобільному магазині. Графіка обрана двовимірна, бо на малому екрані важко оцінити всю красу тривимірної. Тож вона дарма напружувала гаджет. Завдяки гнучкості та популярності у якості операційної системи обрано Android. Більшість людей на планеті використовують саме його, тож хтось обов'язково завантажить гру собі та протестує. Розробка проводилась у середовищі розробки Unity, через зручний та зрозумілий інтерфейс, в якому легко розібратись, велику кількість бібліотек ассетів та плагінів, що значно прискорюють роботу, мультиплатформеність, що дає змогу портувати гру не тільки на мобільні пристрої, а й на персональні комп'ютери та консолі, фізику та здатність створювати важкі анімації.

Під час роботи над написанням скриптів використовувались різноманітні методи, щоб повною мірою використовувати можливості мови програмування C#.

У результаті було створено повноцінну гру з інтерфейсом та двома рівнями оптимальної важкості, різноманітними пастками, бонусами та анімаціями.

ПЕРЕЛІК ПОСИЛАНЬ

1. Donovan, Tristan. Replay, The History of Video Games — Yellow Ant; Illustrated edition 2010 – 516 с.
2. Характеры и жанры видеоэкранных игр - Грановская О.В. Дуков Е. В., Иоскевич Я. Б. и др. – М. Едиториал УРСС 2005 — 488 с.
3. McLeod J. Simulation: the Dynamic Modeling of Ideas And Systems with Computers – McGraw-Hill First Edition 1968 – 351 с.
4. Skyler Miller. The History of Puzzle Games Tetris. - PublicAffairs 1st edition 2016 – 272 с.
5. Geryk, Bruce. A History of Real Time Strategy Games – Crown 1st edition, 2004 – 624 с.
6. Luban, Pascal. Designing and Integrating Puzzles in Action-Adventure Games – Oxford University Press 1st edition 2002 – 320 с.
7. Тимур Хорев. История жанра боевики от первого лица – Мальопус 2007, - 182 с.
8. Thiboust, Jordane. Focusing Creativity: RPG Genres – Routledge, 2013 – 464 с.
9. Andre LaMothe. Tricks of the Windows Game Programming Gurus – Sams Bk CD Rom edition 2003 – 1040 с.
10. C.S. Barnett. Build your own gaming computer – Independently published 2021 – 85 с.
11. Носов Н. А. Виртуальная психология – Ridero 2000 – 262 с.
12. А. Россохин, В. Измагурова. Виртуальное счастье или виртуальная зависимость – Энциклопедия. Минск 2004 – 186 с.
13. База даних Disease ontology – F.A. Davis Company 2016 – 516 с.
14. Reeves, Ben. How Flash Games Changed Video Game History – No Starch Press; 2nd edition 2018 – 312 с.
15. Forster, Winnie. The Encyclopedia of Game Machines – Variant Press 2005 – 224 с.

16. Dal, Yong Jin. Mobile Gaming in Asia: Politics, Culture and Emerging Technologies – Springer; Softcover reprint of the original 1st ed., 2016 – 253 с.
17. Нургалиев, Есбол. Мобильные игры - основные участники и перспективы отрасли – Манн, Іванов і Фербер, 2009 – 240 с.
18. Jon Peddie. The History of Visual Magic in Computers: How Beautiful Images are Made in CAD, 3D, VR and AR - Springer 2013th edition 2013 – 743 с.
19. Pearl, Rick. Closet Classics, Electronic Games – Apress 1st ed. Edition 1983 – 291 с.
20. Рузмайкина И., Хокинг Дж. Unity в действии, мультиплатформенная разработка на C# - Питер Пресс 2016 – 352с.
21. Evolution of Platformers. Retro Gamer: journal. — Live Publishing 2013 – 453 с.
22. Джессі Шелл. Геймдизайн. Як створити гру, у яку будуть грати всі - Альпіна Паблішер 2019 – 640 с.
23. Голощапов А. Google Android: программирование для мобильных устройств – спеціалізації "Системи, технології і комп'ютерні засоби мультимедіа" ХНУРЕ, 2010, 152 с.
24. Комплекс навчально-методичного забезпечення навчальної дисципліни "Технології сучасних мобільних додатків" підготовки магістра, спеціальності 171 - Електроніка Електронний ресурс спеціалізації "Системи, технології і комп'ютерні засоби мультимедіа" ХНУРЕ розроб. Р. І. Цехмістро. – Харків 2018. – 143 с.
25. Комплекс навчально-методичного забезпечення навчальної дисципліни "Цифрові технології мультимедіа" підготовки бакалавра, спеціальності 171 - Електроніка, спеціальності 172 - Телекомунікації та радіотехніка Електронний ресурс ХНУРЕ розроб. І. В. Коритцев. – Харків 2017. – 317 с.

26. Комплекс навчально-методичного забезпечення навчальної дисципліни "Соціальні аспекти мультимедіа" підготовки бакалавра, спеціальність 171 - Електроніка Електронний ресурс спеціалізація "Системи, технології і комп'ютерні засоби мультимедіа" ХНУРЕ розроб. М. М. Колендовська. – Харків, 2017. – 160 с.
27. Комплекс навчально-методичного забезпечення навчальної дисципліни "Сучасні технології анімації" підготовки магістра, спеціальності 171 - Електроніка Електронний ресурс спеціалізації "Системи, технології та комп'ютерні засоби мультимедіа" ХНУРЕ розроб. В. М. Карташов. – Харків 2017. – 119 с.
28. Комплекс навчально-методичного забезпечення навчальної дисципліни "Дизайн інтерфейса" підготовки бакалавра спеціальності 171 - Електроніка Електронний ресурс : спеціалізації "Системи, технології та комп'ютерні засоби мультимедіа" ХНУРЕ розроб. О. В. Бєляєв. – Харків 2018. – 175 с.
29. Методичні вказівки з виконання атестаційної роботи магістра для студентів усіх форм навчання спеціальності 171 «Електроніка» освітньопрофесійної програми «Системи, технології і комп'ютерні засоби мультимедіа» (СТМ). Освітній ступінь – магістр Упоряд. В.М. Карташов, І.В. Савченко – Харків ХНУРЕ 2019. – 42 с.
30. Климова А.М. Аналіз жанру мобільних ігр «графічні інтерактивні новели» // 25-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті». Зб. матеріалів форуму. Т. 3. – Харків: ХНУРЕ. 2021. – с. 28 – 29