Звіт

Про реалізацію інтерактивного проекту: гри-платформера “LaserHunter”

Студента групи ІП-81

Чернишова Дениса

Зміст

[Завдання 3](#_Toc535024072)

[Вибір технології 4](#_Toc535024073)

[Архітектура програми 6](#_Toc535024074)

[Проблеми та методи їх вирішення 10](#_Toc535024075)

[Обробка зіткнень 10](#_Toc535024076)

[Слідкування лазера за персонажем 12](#_Toc535024077)

[Приклад роботи програми 13](#_Toc535024078)

[Висновки 18](#_Toc535024079)

# Завдання

Метою даної роботи є розробка інтерактивної гри-платформера для операційних систем Windows та Android. Гравець повинен дістатись до кінця карти, долаючи перешкоди та ворогів на своєму шляху.

Дана гра реалізує наступні завдання:

1. У головного героя є певний запас здоров’я та сил. Головний герой має одне життя. Гра буде завершена, якщо здоров’я гравця досягне 0.
2. Карта складається з сітки, яка заповнюється тайлами. Кожен тайл реалізує один з блоків. Є чотири види таких блоків:
   1. Блоки, з якими можна зіткнутися(земля, камінь, шипи тощо)
   2. Візуальні, з якими не можна зіткнутися(небо, підземелля, зірки, кущі, хмари)
   3. Рідина, в яких можна плавати(вода, лава)
   4. Предмети, які можна підібрати
3. Головний герой повинен долати різноманітні перешкоди, взаємодіючі з фізикою цього світу, яка реалізується за допомогою наступних блоків:
   1. Вода та лава, які уповільнюють рух персонажа.
   2. Пружний блок, якій надає вертикальної швидкості гравцю.
   3. Лід, який дозволяє гравцю ковзати.
   4. Шипи, які наносять урон гравцю.
4. Карта містить предмети, які додають запас сил або здоров’я. Запас сил з часом відновлюється.
5. По карті розташовані інші персонажі:
   1. Ворожий персонаж – собака, який атакує та переслідує в певній зоні головного героя
   2. Союзний персонаж – кіт, який відновлює запас здоров’я або сил
   3. Червоний лазер – пристрій, який автоматично наводиться та атакує головного героя в певній зоні
   4. Зелений лазер – пристрій, який автоматично наводиться та вистрілює променями, які відновлюють здоров’я героя

Можливості героя:

1. Біг (вліво, вправо)
2. Стрибок
3. Подвійний стрибок(вимагає певний запас сил)
4. Атака ворогів
5. Захищатися від атак ворогів(вимагає певний запас сил)

# Вибір технології

У якості технології я обрав мову програмування Java та крос-платформерний фреймворк libGDX, який з легкістю дозволяє створювати ігри та їх інтерфейс.

**Java** - об'єктно-орієнтована мова програмування, випущена 1995 року компанією «Sun Microsystems» як основний компонент платформи Java.

«Oracle» надає компілятор Java та віртуальну машину Java, які задовольняють специфікації Java Community Process, під ліцензією GNU General Public License.

Мова значно запозичила синтаксис із C і C++. Зокрема, взято за основу об'єктну модель С++, проте її модифіковано. Усунуто можливість появи деяких конфліктних ситуацій, що могли виникнути через помилки програміста та полегшено сам процес розробки об'єктно-орієнтованих програм. Ряд дій, які в С/C++ повинні здійснювати програмісти, доручено віртуальній машині. Передусім Java розроблялась як платформо-незалежна мова, тому вона має менше низькорівневих можливостей для роботи з апаратним забезпеченням, що в порівнянні, наприклад, з C++ зменшує швидкість роботи програм. За необхідності таких дій Java дозволяє викликати підпрограми, написані іншими мовами програмування.

**libGDX** - це Java фреймворк, який надає крос-платформне API для розробки ігор і додатків, що працюють в режимі реального часу. Це високопродуктивний, кросплатформний ігровий фреймворк, що в першу чергу використовується для написання ігрових рушіїв та ігор. Позиціонується, як фреймворк та дозволяє нам максимально зосередитися на міцному фундаменті, замість того, щоб намагатися реалізувати найновіше і найкраще з ігрових рушіїв. LibGDX надає Вам гнучкість і дозволяє уникнути суворої методології.

За допомогою даної бібліотеки, можна використовувати один і той же код як для систем настільних комп'ютерів так і мобільних систем. Бібліотека є кросплатформенною і підтримує Windows, Linux, Mac OS X, Android, iOS, та браузери з підтримкою WebGL.

**libGDX** дозволяє розробнику писати, тестувати і налагоджувати додатки на власному ПК і використовуючи той же код на Android. Це абстрагує відмінності між додатками Windows/Linux та Android. Основна мета полягає в забезпеченні повної сумісності між настільними і мобільними пристроями. Код написаний для однієї платформи працює так само добре і на інших.

Список можливостей

1. Крос-платформна розробка для Windows, Linux, Mac OS X, Android, Браузери з підтримкою WebGL та iOS. Підтримуються як 32 так і 64-розрядні версії;
2. Комбіновані бекенд можливості, засновані на JoGL, LWJGL, RoboVM і Android API;
3. Розробка і тестування додатків на комп'ютері і легке наступне розгортання під Android, iOS, аплети, webstart (JAWS) і під звичайний ПК.
4. Гнучка модульна структура. Можливості, яких немає в базовій версії, компенсуються розширеннями. Наприклад, є розширення для інтеграції фізичного рушія Box2D, розширення для використання TrueType шрифтів.
5. Набір вбудованих класів для реалізації графічного інтерфейсу користувача. Сюди входять як примітивні елементи - такі як кнопки, текстові поля - так і складні елементи - випадаючі списки, панелі з можливістю прокрутки (скролл) та інше.

У якості середовища програмування, я використовував Android Studio, за допомогою якої я міг не тільки створювати доданки для платформи Android, а і перевіряти їх працездатність в емуляторі.

**Android Studio** — інтегроване середовище розробки (IDE) для платформи Android, представлене 16 травня 2013 року на конференції Google I/O менеджером по продукції корпорації Google — Еллі Паверс (англ. Ellie Powers). 8 грудня 2014 року компанія Google випустила перший стабільний реліз Android Studio 1.0

Наразі, передбачені такі функції

1. Живі макети (layout): редагувальник WYSIWYG — живе кодування — подання (rendering) програми в реальному часі.
2. Консоль розробника: підказки по оптимізації, допомога по перекладу, стеження за напрямком, агітації та акції — метрики Google аналітики.
3. Базування на Gradle.
4. Android-орієнтований рефакторинг та швидкі виправлення.
5. Шаблони для створення поширених Android дизайнів та компонентів.
6. Багатий редактор макетів (layouts) що дозволяє користувачам перетягнути і покласти (drag-and-drop) компоненти користувацького інтерфейсу, як варіант, переглянути одночасно макети (layouts) на різних конфігураціях екранів.

# Архітектура програми

Архітектура усієї гри складається з механіки, інтерфейсу, та логіки. Усю реалізацію програми можна поділити на три головні задачі.

1. Реалізація відображення тексту, рендерінгу карти, та усіх складових зовнішнього вигляду інтерфейсу програми.
2. Реалізація персонажів, які визначають їх , фізику, логіку та поведінку.
3. Реалізація інтерфейсу, екранів, їх взаємодії та структури програми.

Головним класом гри є **LaserHunterGame**, який визначає який саме екран буде відображатися, та відповідно оновлює його.

**1.До першої категорії відносяться наступні класи:**

1. **GameMap** – абстрактний клас, який використовується для створення рівнів, відображення сітки карти, її тайлов, та обробки зіткнень персонажів з картою. Також обробляється та відстежується позиція персонажа та відбувається оновлення та фіксування камери згідно з головним героєм.
2. **StartMap** – (успадкований від GameMap) використовується для відображення та оновлення усіх персонажів та відображення елементів інтерфейсу таких, як текстові повідомлення, журналу дій, та полоси здоров’я для гравця та інших персонажів. Також містить декілька допоміжних функцій для коректного відображення карти.
3. **TileType** – перелік тайлов. Кожен тайл характеризується деякими полями, які визначають його функцію та поведінку гравця при взаємодії з ним. Визначається декілька головних параметрів : чи може гравець бути зімкнений з тайлом, урон, який він наносить, чи є цей тайл рідиною або візуальним тайлом, тощо.
4. **Button** – використовується для створення кнопок, обробки натиснення та відповідних дій. Також контролює коректне відображення анімації натиснення, розмір та відображення на екрані.
5. **Bar –** клас, який реалізує відображення полоси здоров’я для будь-якого об’єкту класу Entity. Відповідно до поточного стану здоров’я персонажу, заповненість цієї полоси також змінюється.
6. **Log –** використовується для обробки та відображення інформації на екрані у формі журналу останніх дій. Використовується клас TextRegion для відображення тексту.
7. **TextRegion** – клас, який реалізує відображення тексту на екрані, його розташування, розмір, шрифт тощо.
8. **AnimationHandler** – клас для обробки и коректного відображення анімації для певного персонажа у певному стані.

**2.Класи другої категорії** – це головний та інші персонажі, які якось взаємодіють з картою та іншими персонажами. Для кожного за такого персонажа визначено його кількість здоров’я, сил, швидкість руху, масу, та густину. Також кожний з персонажів має свою визначену кількість урону, та захисту, яке він може застосовувати проти других персонажів.

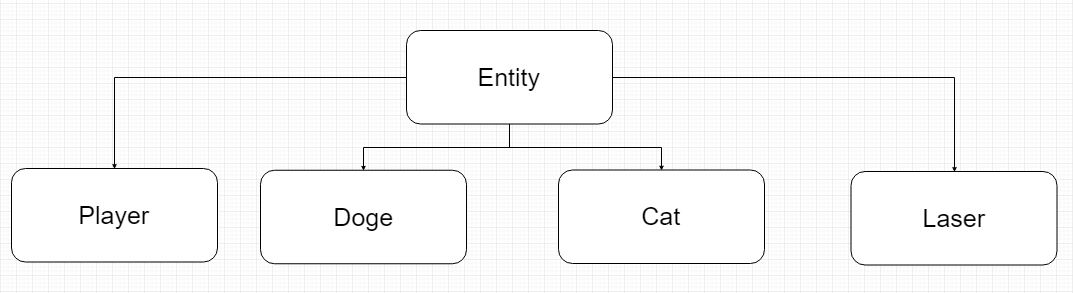
Також для кожного усіх персонажів (у абстрактному класі **Entity)** визначені фізичні сили, які постійно діють на них. Наприклад, постійно на персонажа діє сила тяжіння або при відповідному руху, на нього діє рухома сила та сила тертя(яка визначається поверхнею, на якій він стоїть).

Для персонажів окрім головного героя, визначений також штучний інтелект, та зона дії, відповідно до яких персонаж рухається та атакує. Наприклад для об’єктів класу Doge, вони будуть перевіряти, чи знаходиться персонаж у їх полі зору, та рухатись відповідно, переслідуючи.

Цю категорію представляють наступні класи:

1. **Entity –** абстрактний клас, у якому визначені усі головні моделі поведінок для кожного з персонажів, визначається та розраховуються усі сили, які діють на персонажа, його стан, здоров’я, сили, у якому напрямку рухається, та відповідні швидкості руху на основі поточних сил прикладених до нього. Також здійснюється головна обробка зіткнень з картою, та визначається його модель поведінки. Наприклад, якщо персонаж стає на блок з льодом, то у функції обробки фізики його коефіцієнт тертя зменшується, що призводить до ковзання.
2. **Player –** головний герой. У класі реалізовується обробка натискання на клавіши та відповідний рух. Та усі функції, які використовуються для обробки вводу або з клавіатури, або за допомогою кнопок(у випадку Android).
3. **Doge -** ворожий персонаж, який може бути у двох станах. У стані охорони, тобто він не буде атакувати гравця та буде стояти на місці, доки той не перетне відповідну межу, а потім буде його доганяти та атакувати. У звичайному стані він ходить у деякому діапазоні, доки гравець не буде у полі зору.
4. **Cat –** союзний персонаж, який також ходить або стоїть в очікуванні головного героя, та також може захищатись від інших ворожих персонажів. При приближенні героя досить близько, він відновить його здоров’я або сили.
5. **Laser –** єдиний статичний персонаж який постійно вистрелює променями в один відповідний напрям, або слідкує за головним героєм. Є два види промені : які наносять шкоду та які відновлюють здоров’я. Лазери характеризуються їх станом, темпом стрільби, швидкістю променю, та уроном, який наносять.

Ієрархію класів можна побачити на наступному слайді.



**3.Третя категорія** реалізує інтерфейс програми, та як взаємодіють компоненти гри.

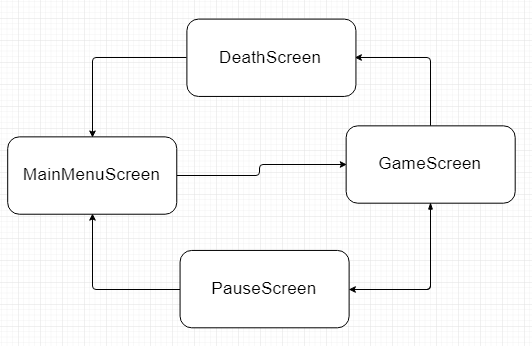
За це відповідають класи успадковані від класу бібліотеки LibGDX Screen, який використовується для створення інтерфейсу, екранів та меню в грі.

1. **MainMenuScreen –** екран з головним меню, в якому користувач може почати грати або вийти з гри.
2. **GameScreen –** екран з самою грою, на якому відображається відповідна карта StartMap, та усі персонажі.
3. **PauseScreen –** екранпаузи, який можна визвати, натиснувши на Esc у грі. З цього екрану можна також перейти у головне меню або повернутись у гру, або закінчити гру.
4. **DeathScreen –** екран закінчення гри. У якому є можливість тільки вернутися у меню та почати заново.

Кожен клас екрану має відповідні методи для коректної роботи, відображення та обробки дій.

1. **show()** – метод, який викликається кожного разу, коли створюється вікно. Наприклад у класі GameScreen ініціалізується карта та її компоненти.
2. **render() –** метод, який викликається для відображення цього екрану та його складових.
3. **update() –** метод, який оновлює стан цього екрану, та який використовується для оновлення елементів, або карти, яку він відображає.
4. **resize()** – метод, який викликається кожного разу при зміні розміру екрана.
5. **pause() –** метод, який викликається при зупинці програми. Ресурси не звільнюються при цьому методі.
6. **resume() –** метод, який виконується одразу при відновлені роботи програми, тобто після pause()
7. **hide()** – метод, який викликається при згорненні програми у фоновий режим
8. **dispose()** – метод, який звільнює пам’ять та усі ресурси, які були використані цим екраном протягом роботи, тобто викликається при його закритті.

Взаємодію класів можна побачити на діаграмі.



# Проблеми та методи їх вирішення

## Обробка зіткнень

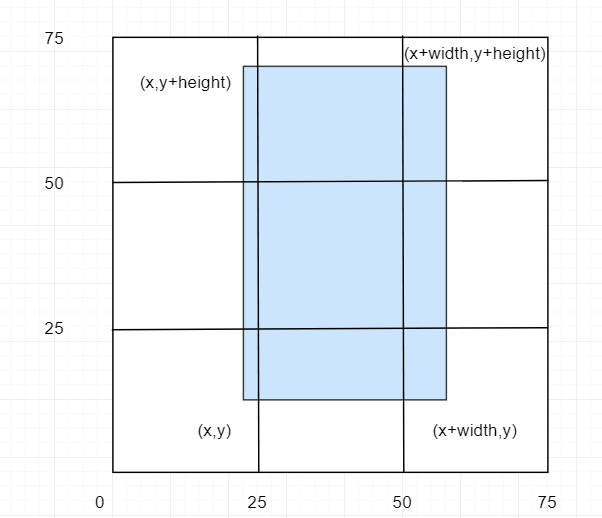
Перша важлива проблема, яка встала у мене на шляху при написання платформера це безсумнівно проблема обробки зіткнень персонажів з картою, оскільки ця взаємодія персонажа з картою є ключовою у такий грі.

При написанні таких ігор є багата кількість алгоритмів, які можна використовувати для коректного відстеження зіткнень, але саме для tiled-based карти з врахуванням фізики карти та взаємодії персонажа з фізичними блоками, алгоритм повинен був обробляти усі зіткнення відповідно до поточної позиції персонажа.

Алгоритм базується на тому, щоб перевірити усі тайли, які знаходяться в околі персонажа, чи є вони тайлами, з якими можна зіткнутися (collidable == true).

Около персонажа визначається наступним чином:

1. Відповідно до позиції персонажа, його положення у сітці карти (ширина клітини – TILE\_SIZE) вираховується відповідно до його поточного положення, тобто х / TILE\_SIZE.(Наприклад ширина тайла – 25, координата персонажа – 424, то береться координата 16)
2. Відповідно визначається найбільша координата згідно до ширини персонажа тобто (х + width) / TILE\_SIZE
3. Відповідно визначається координати для осі ординат.

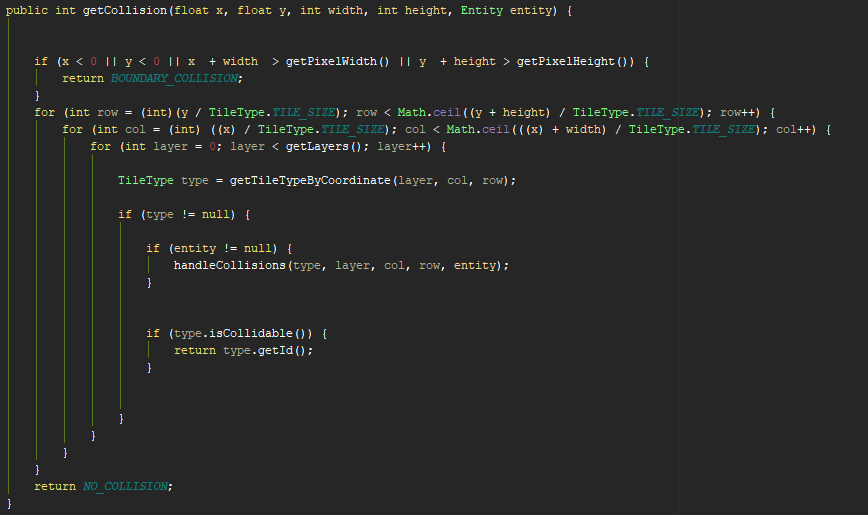


Далі за допомогою циклу, перевіряємо усі можливі варіанти, які відповідають тим тайлам, які знаходяться у діапазоні від x/TILE\_SIZE до (х + width) / TILE\_SIZE (для осі ординат відповідно) на відповідній карті, та також перебираємо кожний слой цієї карти.

Далі отримавши координати, які відповідають тайлам в околі персонажа, перевіряємо, чи є поле collidable істиним, тобто чи може персонаж зіткнутися з цим тайлом. Якщо так, то повертаємо значення, яке відповідає айді цього тайлу для подальшої обробки у класі Entity. Якщо collidable == false, то обробляємо цій тайл, на випадок, якщо персонаж контактує з рідиною, або з предметом, який можна підібрати у функції handleCollisions()

Також ця функція може повернути код зіткнення з границею карти BOUNDARY\_COLLISION, або код, який відповідає, що ніякого зіткнення в околі немає – NO\_COLLISION.

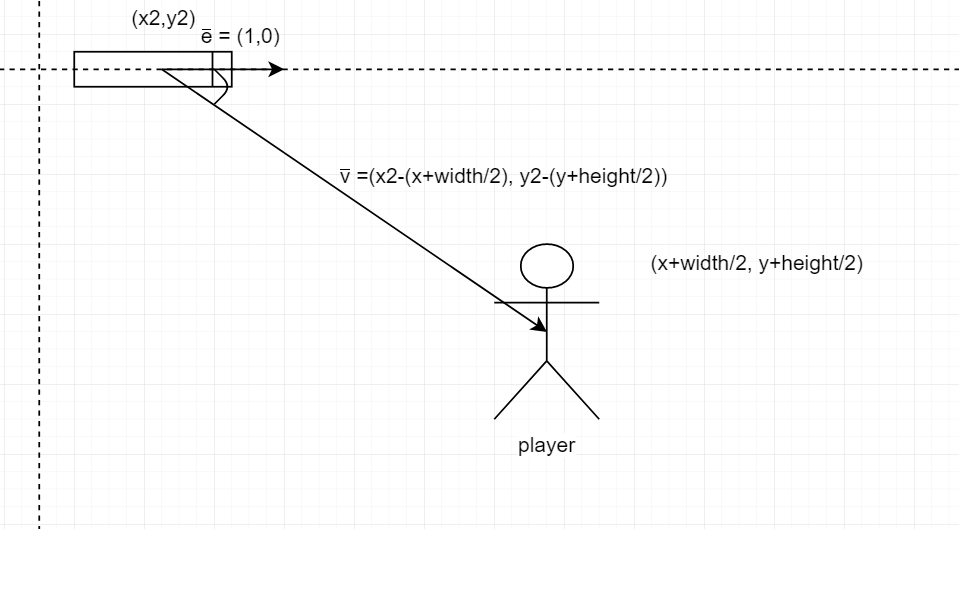
Дана функція викликається в методі update класа Entity, де відповідно до кожного коду, якого ця функція повертає, обробляється подальший стан та поведінка персонажа.



## Слідкування лазера за персонажем

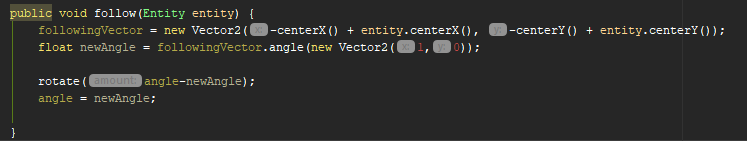
У процесі розробки класу Laser було вирішено зробити його статичним, але з можливістю слідкувати за гравцем, що і вимагало деякого рішення.

У класі Sprite, який використовувався для відображення цього лазеру існує функція, для повороту цього спрайта на деякий кут, відносно поточного положення. Тому було вирішено використовувати саме цю функцію для даної проблеми. І задача зводилася тільки к визначенню кута, на який треба повернути лазер.



Для цього була створена функція follow(), яка б слідкувала за персонажем, який буде передану у якості аргументу функції. Вона має досить простий алгоритм:

1. У конструкторі ініціалізується та створюється змінна angle, яка визначає поточний кут відносно осі ОХ, який має лазер.
2. Вираховується вектор , як напрямлений вектор від позиції лазера до центра персонажа
3. Створюється одиничний вектор – орта осі ОХ
4. Вираховується кут між цими векторами та записується у newAngle
5. Вираховується на який кут треба повернути лазер відносно свого попереднього стану тобто angle-newAngle, та повертається на цей кут.



# Приклад роботи програми

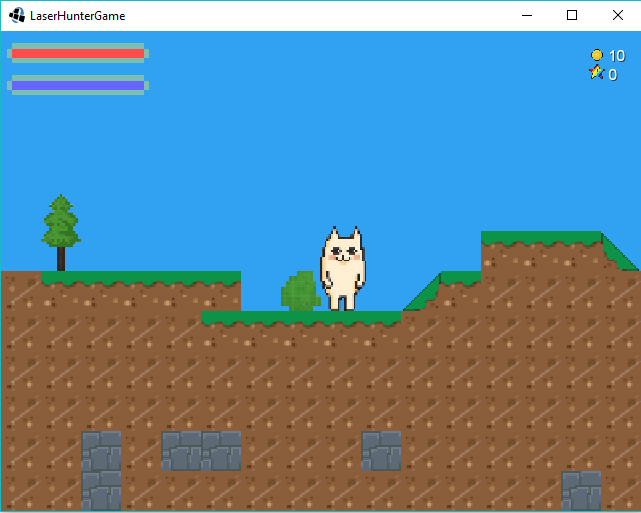
1. Одразу ж після запуску програми, гравець попадає у головне меню, яке відповідає екрану MainMenuScreen з інтерактивним інтерфейсом, де йому надається можливість або вийти з програми, або почати гру. Одразу ж у головному меню можна зустрітись з головною частиною цієї гри – лазери, які слідкують за курсором мишки, та стріляють при нажаті.





1. Далі після натиснення кнопки START, гравець одразу ж попадає на карту, на якої він може взаємодіяти з навколишнім світом а саме:
   1. Рухатись вліво-вправо (клавіши A, D, або відповідні кнопки на екрані)
   2. Стрибати (клавіша Space, або треба натиснути на праву частину екрана)
   3. Атакувати (клавіша Ctrl, або відповідна кнопка)
   4. Захищатися(клавіша Shift, або відповідна кнопка)

Приклад роботи для Windows та для Android аналогічно.

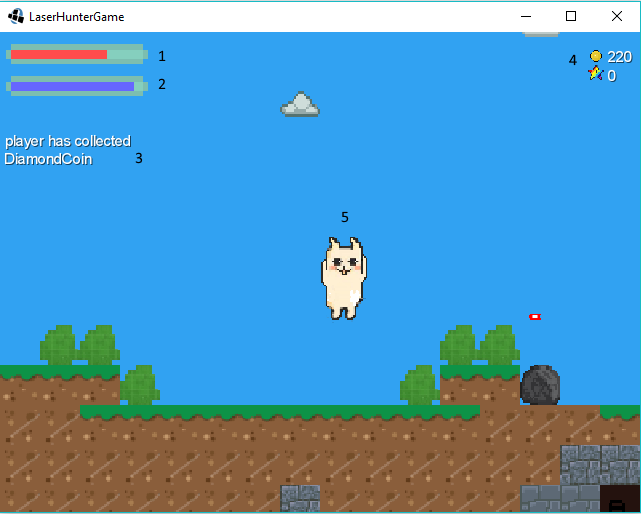




На екрані одразу користувач може побачити декілька важливих елементів інтерфейсу. А саме:

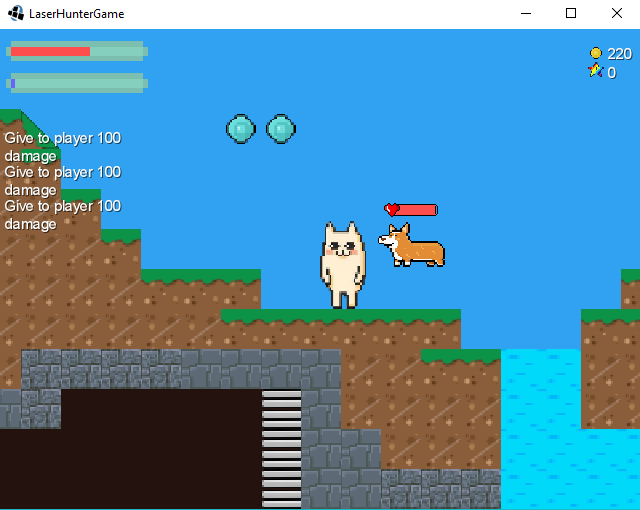
1. Полоса здоров’я.
2. Полоса сил.
3. Журнал останніх дій.
4. Кількість зібраних монет та зірок.
5. Головний герой

У версії для Android також додаються кнопки руху вправо та вліво, атаки, та захисту, які замінюються кнопками на клавіатури для Windows.

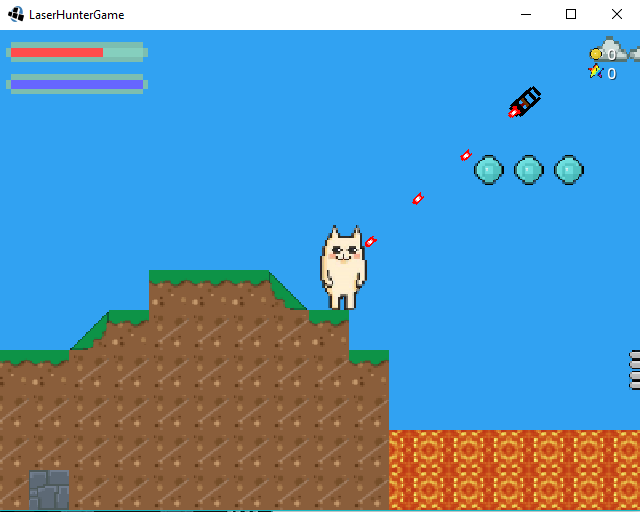
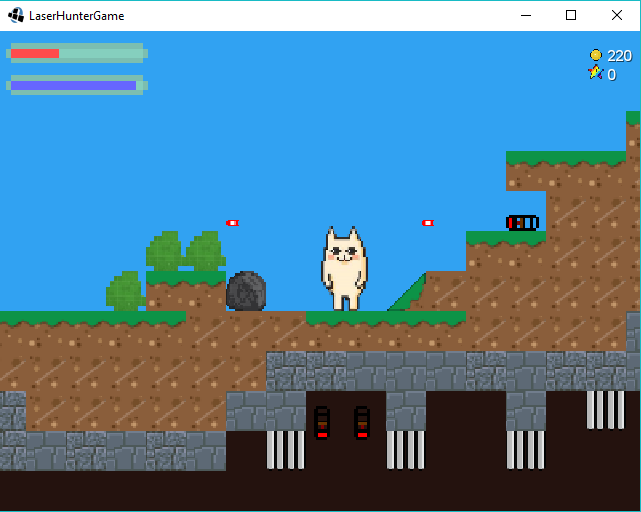


Гравець може зустрітися з різними об’єктами класу Entity протягом гри:

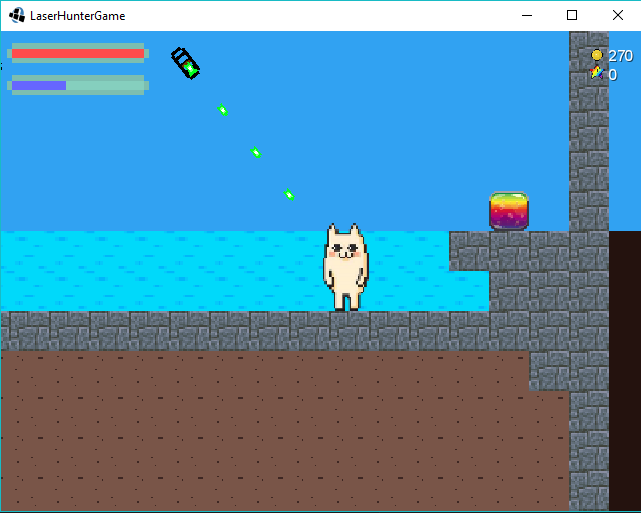
* *Doge*



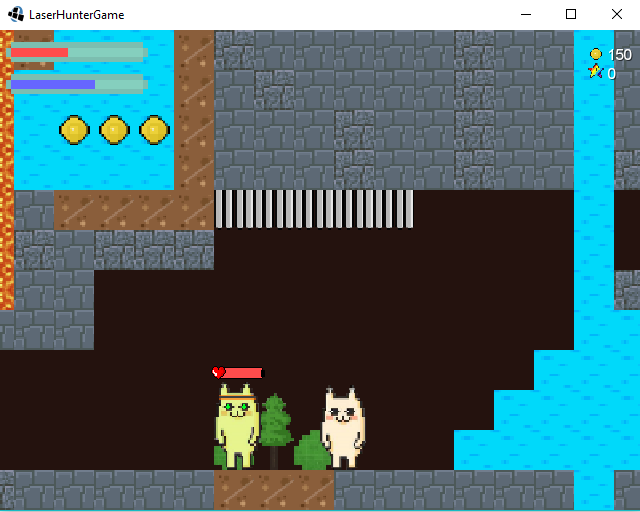
* *RedLaser*



* *GreenLaser*



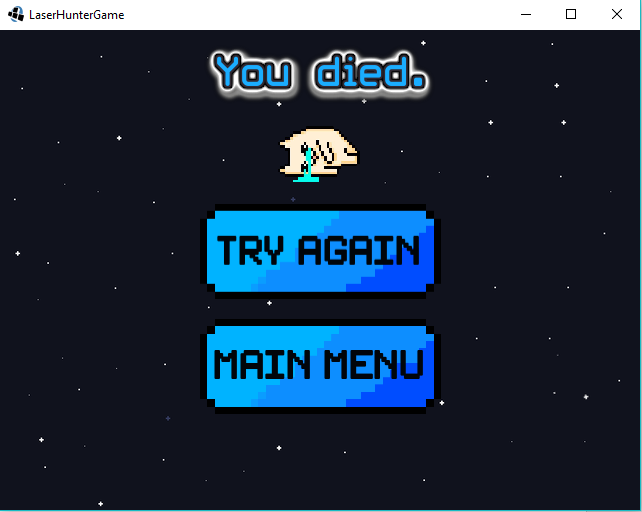
* *Cat*



3.При натисненні клавіши Esc, або на відповідну кнопку у правому верхньому куту екрана(для Android) гра буде поставлена на паузу. На екрані PauseScreen у користувача є можливість або повернутися у гру, або повернутися у головне меню.



4. Коли у гравця кількість здоров’я досягає нулю, то він вмирає, та гра закінчується. Це супроводжується переходом на екран DeathScreen та відповідним повідомленням. Далі є можливість або повернутися у меню, або почати спочатку гру.



# Висновки

При розробці програми я освоїв декілька важливих інструментів, таких як фреймворк libGDX, який дозволяє з легкістю створювати ігри для будь-яких платформ, та Android Studio – потужний засіб для розробки програм для платформи Android. Здобув досвід у створенні Android доданків, та освоїв їх архітектуру.

Також, зустрівшись з великою кількістю проблем зв’язаних з проектуванням архітектури програми, я навчився досить раціонально та ефективно використовувати засоби об’єктно-орієнтовного програмування для створення класів, які допомагають мені у реалізації великої кількості задач.

Отже, я вважаю, що задачі, які я ставив перед собою при написанні цього проекту були повністю реалізовані а навички, які я здобув протягом цього часу можна з легкістю використовувати для написання інших проектів такого плану.