МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ

Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ГРИ ЗАСОБАМИ БІБЛІОТЕКИ SDL ДЛЯ МОВИ С++»**

Виконав: студент 4 курсу, групи 6.1210-2пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
І.В. Козловський
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, PhD Чопорова О.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

________________ Лісняк А.О.
(підпис)

“ _____ ” ___________ 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Козловському Івану Вікторовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка гри засобами бібліотеки SDL для мови C++

керівник роботи Чопорова Оксана Володимирівна, PhD
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 21 » грудня 2023 року № 2180-с

2. Строк подання студентом роботи 03.06.2024 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   1. Постановка задачі, аналіз предметної області.
   2. Проєктування системи.
   3. Реалізація програмного застосунку.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою докладу

6. Консультанти розділів роботи

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

7. Дата видачі завдання 25.12.2023 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	09.01.2024	
2.	Збір вихідних даних.	24.01.2024	
3.	Обробка методичних та теоретичних	12.02.2024	
	джерел.		
4.	Розробка першого та другого розділу.	29.03.2024	
5.	Розробка третього розділу.	20.05.2024	
6.	Оформлення та нормоконтроль	27.05.2024	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	19.06.2024	

Студент ____________ І.В. Козловський
(підпис) (ініціали та прізвище)

Керівник роботи ____________ О.В. Чопорова
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер ____________ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка гри засобами бібліотеки SDL для мови C++»: 66 с., 22 рис., 17 джерел, 5 додатків.

БІБЛІОТЕКА SDL, МОВА ПРОГРАМУВАННЯ C++, ПАТТЕРНИ ПРОЄКТУВАННЯ, ПРИГОДНИЦЬКА ГРА, РОЗРОБКА, UML ДІАГРАМИ.

Об'єкт дослідження – процес розробка ігрового застосунку засобами бібліотеки SDL.

Мета роботи: створення пригодницької гри із механіками пересування та бою.

Методи дослідження – методи збору та аналізу вимог до програмного забезпечення, методи моделювання, проєктування та конструювання програмного забезпечення.

У кваліфікаційній роботі було досліджено можливість розробки ігрового застосунку з допомогою бібліотеки SDL та мови програмування C++. Ця крос-платформна бібліотека дозволяє реалізувати різноманітні застосунки для підтримуваних платформ.

Необхідну для дослідження інформацію було зібрано та проаналізовано.

Було спроєктовано архітектуру системи за допомогою UML у вигляді наборів діаграм варіантів використання, послідовностей та діяльності. Реалізовано ігрові механіки з використанням патернів проєктування та можливостей мови програмування C++.

Результатом роботи став ігровий застосунок у жанрі «платформер».

Знайдені методи та інструменти можна використовувати для створення власного застосунку, чи для пошуку нових методів розробки ігор.

SUMMARY

Bachelor's qualifying paper "Development of a Game Using the SDL Library for the C++ Language": 66 pages, 22 figures, 17 references, 5 supplements.

ADVENTURE GAME, C++ PROGRAMMING LANGUAGE, DESIGN PATTERNS, DEVELOPMENT, SDL LIBRARY, UML DIAGRAMS.

The object of the study is the process of developing a game application using the SDL library.

The aim of the study is creation of adventure game with movement and combat mechanics.

The methods of research are methods of gathering and analyzing software requirements, methods of modeling, projecting and designing software.

Including the qualifying work, the possibility of developing a game application using the SDL Library programing language C++. This cross-platform library allows you to implement a variety of applications for supported platforms.

Necessary information for research was token and analyzed.

The system architecture was designed using UML in the form of sets of diagrams of use cases, sequences and activities. Implemented game mechanics using design patterns and capabilities of the C++ programming language.

The result of the work was a game application as a "platformer".

The found methods and tools can be used to create your own application, or to find new game development methods.

ЗМІСТ

Завдання на кваліфікаційну роботу……………………………………….2

Реферат 4

Summary 5

Вступ.. 8

1 Аналіз предметної області.. 9

1.1 Ігрова індустрія 9

1.2 «Платформери» 10

1.2.1 “Donkey Kong” 11

1.2.2 “Super Mario Bros” 12

1.2.3 “Sonic the Hedgehog” 12

1.2.4 “Hollow Knight”... 13

1.3 Мова програмування та основні поняття 14

1.4 UML – як засіб проєктування систем 15

1.5 Метод тайлінгу 16

1.6 Висновки до розділу 1 17

2 Проєктування системи.. 18

2.1 UML діаграми.. 18

2.1.1 Діаграма прецедентів... 18

2.1.2 Діаграма діяльності.. 19

2.1.3 Діаграма послідовності 20

2.2 Головні механіки та рішення. Використані патерни проєктування .. 21

2.3 Висновки до розділу 2 23

3 Реалізація системи.. 24

3.1 Мінімальні вимоги до гри та користувачів. 24

3.2 Створення базового вікна.. 24

3.3 Система класів для відображення на екрані... 26

3.4 Створення ігрових сцен 27

3.5 Створення мапи 29
3.6 Створення системи колізії 30
3.7 Створення сімейства створінь та їх поведінки 32
3.8 Правильне відображення текстур у вікні 33
3.9 Клас менеджеру даних гри та додаткові інструменти 34
3.10 Огляд результату 36
3.11 Висновки до розділу 3 37
Висновки 38
Перелік посилань 39
Додаток А Код класів сімейства Screen та класу Drawable 41
Додаток Б Код класів сімейства Entity 48
Додаток В Код класів для перевірки зіткнень 55
Додаток Г Код класів сімейства State 58
Додаток Д Код файлу менеджеру даних 64

ВСТУП

У наш час комп’ютерні ігри стали чимось більше ніж просто розвагою. Це великий бізнес, у який вкладено багато грошей. Ігрова індустрія це окрема велика частина всесвітнього ринку. Зараз неможливо представити телефон чи комп’ютер без жодної гри.

Великий попит на нові й кращі ігри спричиняє розвиток нових технологій та рішень, що стають все швидшими та доступнішими ніж декілька років тому. А розробка комп’ютерних ігор завжди привертала увагу програмістів своєю складністю і цікавістю.

Проте багато ігор створені з однакових інструментів та ресурсів. Також багато ігор копіюють інші й ринок перенасичений поганими іграми чи копіями інших [1].

SDL є потужною бібліотекою, яка дозволяє створювати ігри різної складності, забезпечуючи при цьому високий рівень контролю над процесом розробки. Вона зручна у використанні та постійно розвивається. А використання мови C++ відрізняється швидкістю роботи та доступом до пам’яті. Однак на сьогодні є дуже мало ігор створених за допомогою цих інструментів.

Саме тому розробка нового унікального застосунку має велике значення. Збільшивши рівень унікальності ігор з’явиться більше можливостей для конкуренції, що штовхне розробників шукати нові рішення. Привертання уваги до SDL дозволить більшій кількості добровольців приєднуватись до розробки цієї бібліотеки. Сам застосунок стане засобом гарного проведення часу та підвищення настрою.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Ігрова індустрія

Відеоігри існують вже більше 30 років та за цей час стали невід'ємною частиною сучасної культури та бізнесу. Величезні корпорації, не менші за найвідоміші компанії, що у минулому починались з горстки людей, зараз отримують мільйони доларів щорічно. Станом на 2023 рік приблизно 38% людства, не залежно від віку або статі, грає або грало у відеоігри, що на 1 мільярд більше, ніж 8 років тому (див. рис. 1.1).

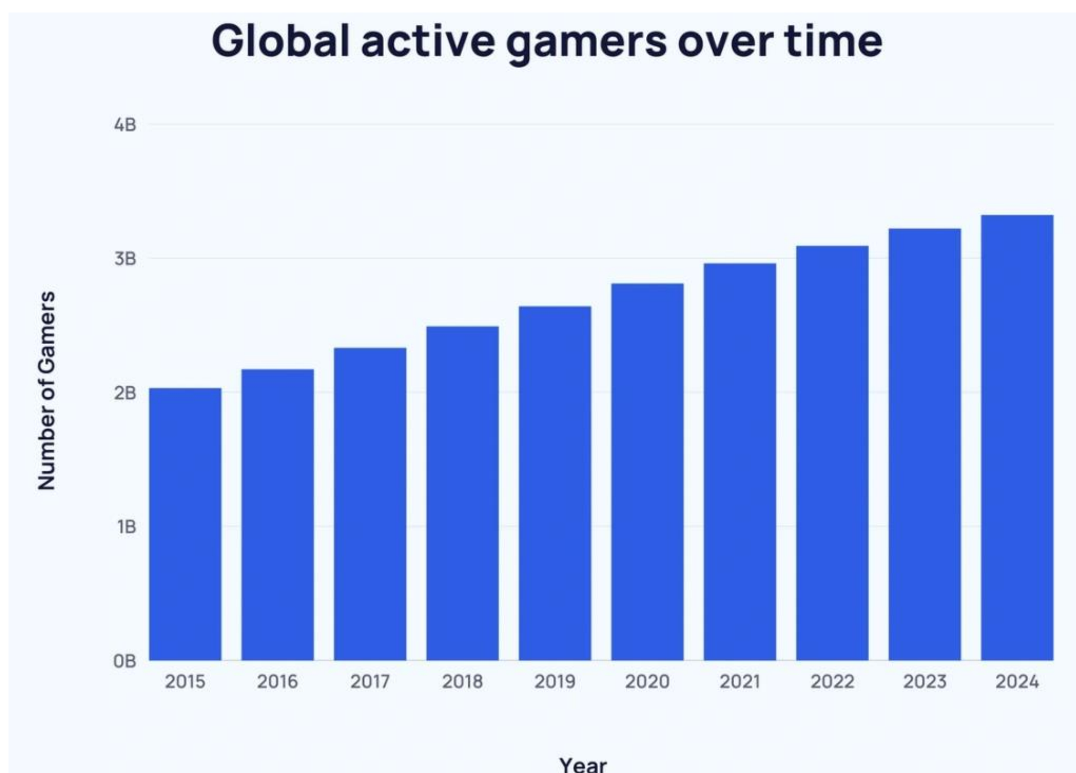

Рисунок 1.1 – Графік кількості гравців у ігри за останні 8 років [2]

Початком великої історії ігор вважається створення електронної версії гри «ОХО» – (хрестики-нолики) у 1952 році британським професором А. С. Дугласом у рамках його докторської дисертації у Кембриджському університеті [3]. Дуглас позиціонував своє творіння як демонстрацію

можливостей взаємодії людини та комп'ютера, що стало одним з найперших застосувань примітивного, проте справжнього, штучного інтелекту [4].

З тих часів виникло безліч жанрів відеоігор – від розважальних до змагальних, від головоломок та стратегічних до активних та швидкісних.

До найбільш популярних належать:

– шутер – жанр, головною механікою якого є стрільба;
– пригоди – вирізняються глибиною оповіді, дослідженням ігрового світу та розв'язанням головоломок;
– рольові ігри – дозволяють зануритись у насичені, часто фантастичні світи, граючи роль різних персонажів зі складним розвитком;
– королівська битва – один з наймолодших жанрів багатокористувацьких ігор, суть якого у закритті гравців у певній зоні, що постійно звужується, наповненої зброєю, доки не залишиться один гравець [5].

Проте для виконання завдання було обрано інший жанр – «платформер».

1.2 «Платформери»

«Платформер» – жанр відеоігор, головний ігровий процес якого зосереджується на пересуванні по «платформах», які треба долати використовуючи механіку стрибка (див. рис. 1.2).

Рисунок 1.2 – Shovel Knight: Shovel of Hope. Типовий рівень «платформеру»

На відміну від вищезгаданого жанру «Королівська битва», «платформер» є одним з найбільш старих жанрів. Це обумовлено відсутністю 3D-графіки на ранніх етапах становлення ігор. Першою грою в цьому жанрі стала гра від студії Nintendo під назвою “Donkey Kong”.

1.2.1 “Donkey Kong”

Гра мала досить просту умову для проходження. Користувачі грали за персонажа Jumpman і на кожному рівні повинні були врятувати принцесу від гігантської мавпи Донкі-Конга. Використовуючи свою спритність гравці проходили велику серію рівнів, піднімаючись вище по сходах, використовуючи бонуси молота, щоб знищувати об’єкти, і збираючи бонусні предмети по дорозі, щоб отримати додаткові бали. “Donkey Kong” незмінно хвалять як одну з найскладніших ігор своєї епохи. “Donkey Kong” ознаменувала дебют жанру «платформерів» [6]. Історія персонажа Jumpman знайшла продовження у наступних іграх від Nintendo. Отримавши брата Луїджі та нове ім’я, Маріо, розпочалася історія однієї з найвідоміших ігрових серій “Super Mario Bros.” (див. рис. 1.3).

Рисунок 1.3 – “Donkey Kong”

1.2.2 “Super Mario Bros”

Гра створена Nintendo Company у 1985 році для Nintendo Entertainment System (NES). Головними героями є Маріо та Луїджі, два італійські водопровідники, що мають врятувати принцесу від злого короля Боузера подорожуючі Грибним королівством. Це одна з найбільш продаваних серій ігор, копій яких було продано понад 40 мільйонів (див. рис. 1.4).

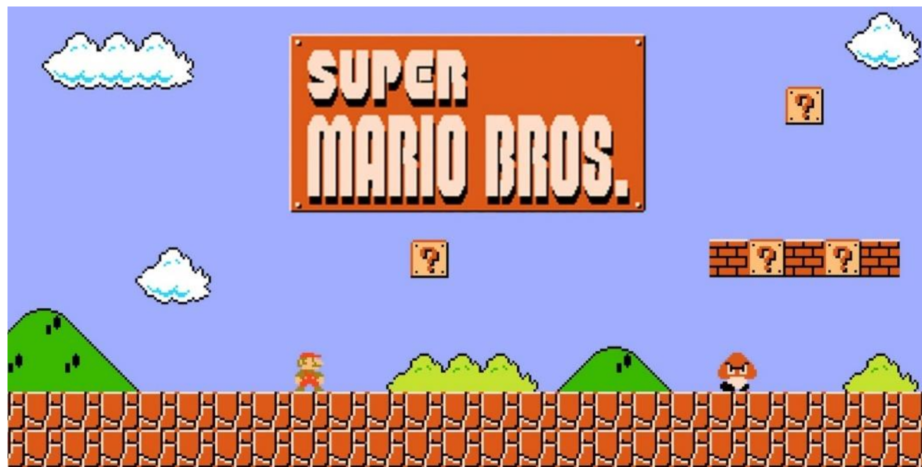

Рисунок 1.4 – “Super Mario Bros.”

У грі можна грати вдвох, один гравець за Маріо, а другий – Луїджі. Гра проходить на серії рівнів із боковим прокручуванням, кожен з яких наповнений ворогами. Рівні відбуваються в різних локаціях: в підземеллях, над землею, кінець кожного у замку. Після того, як черговий самозванець головного злодія переможений, житель Грибного королівства повідомляє Маріо або Луїджі, що принцеса знаходиться в іншому замку. Гра завершується поразкою справжнього Боузера і порятунком принцеси [7].

1.2.3 “Sonic the Hedgehog”

Гра створена компанією Sega у 1991 році як прямий конкурент найпопулярнішому на той час Маріо. Головні розробники, Юдзі Нака та Наото

Осіма, взялися за справу створити нового талісмана компанії Sega та її однойменної консольної ігрової приставки. Щоб показати її потужність було обрано напрямок гри у швидкість. Наото Осіма створив образ синього їжака на ім’я Сонік, який до сьогодні є синонімом швидкості та бунтарства. Ця концепція вимагала великих зусиль від команди, адже доводилось в короткі терміни оптимізувати продуктивність гри, щоб відповідати швидкості їжака, змагаючись з тиском від вже усталеної франшизи італійського водопровідника Маріо від Nintendo [8].

Інновацією у жанрі стали «криві» платформи. Рівні окрім класичних прямокутних платформ рівні мали плавні трампліни, підйоми та спуски, та навіть загорнуті у петлю доріжки, створені для трюку «Мертва петля». Розробники створили спеціальний алгоритм для пересування спрайту по кривій, визначаючи місцеположення за допомогою матриці (див. рис. 1.5).

Рисунок 1.5 – “Sonic the Hedgehog”

1.2.4 “Hollow Knight”

“Hollow Knight” – це метроїдванія-платформер, розроблений командою Team Cherry. Гра отримала величезну популярність завдяки своєму чудовому дизайну персонажів, музиці, атмосфері та глибокому геймплею.

У грі ви виступаєте у ролі маленького комахоподібного героя, який досліджує великий і загадковий підземний світ під назвою Hallownest. Головна мета – виживати в небезпечному середовищі, боротися зі зловісними монстрами та босами, збирати різноманітні предмети та вдосконалювати навички.

Однією з ключових особливостей гри є її величезний відкритий світ з численними схованими зонами, секретами та альтернативними шляхами. Гра також відзначається високим рівнем виклику, особливо у боях з босами, які вимагають від гравця відмінного рефлексу та стратегічного мислення [9].

Загалом, “Hollow Knight” відомий своєю красою, глибоким сюжетом та атмосферою, яка захоплює з перших хвилин гри (див. рис. 1.6).

Рисунок 1.6 – “Hollow Knight”

1.3 Мова програмування та основні поняття

Існує багато мов програмування, що дозволяють створювати ігри. Такі відомі ігрові рушії як Unity Unreal та Engine використовують C# та C++ відповідно. Менш відомий рушій Godot також використовує C#. Усі ігри на мобільні пристрої створюються або мовою Java для андроїд, або Objective-C чи Swift для системи IOS. Завдяки набору модулів PyGame є можливість створювати ігри на мові Python. Проте найкращим вибором є мова C++.

Особливістю C++ є вільний доступ до пам’яті. Програміст має можливість вільно розпоряджатися нею – займати за потреби чи вивільняти у будь-який момент часу. Ігри можуть нести у собі дуже «важкі» дані, зазвичай це текстури. Тому дуже важливо оптимізувати їх роботу, адже надмірне споживання пам’яті призведе до повільної роботи усієї системи. Керування пам’яттю у C++ дозволяє максимально точно оптимізувати усі випадки використання пам’яті. Проте C++ у чистому вигляді недостатньо для створення повноцінної гри одразу. До цього незмінно буде створено інструмент для використання комп’ютерної графіки та іншій важливих елементів потрібних для ігор [10]. Саме таким інструментом є кросплатформна бібліотека SDL.

Simple DirectMedia Layer, вона ж SDL – це бібліотека для розробки крос-платформного програмного забезпечення, що виникла в результаті дослідження Міжнародної спілки електрозв’язку, розпочатого в 1968 році щодо способу обробки систем комутації керування збереженою програмою. Вона поширюється за ліцензією zlib, що дозволяє вільно використовувати цю бібліотеку у будь-якому проєкту. SDL надає низькорівневий доступ до аудіо та графічного обладнання, та комп’ютерної периферії через такі бібліотеки як OpenGL, Direct3D, Metal та Vulkan, та використовується у багатьох категоріях програмного забезпечення, таких як відео плеєри, емулятори та відео-ігри.

Офіційно існують релізи для таких платформ як Windows, macOS, Linux, iOS, та Android та оброблює код таким чином, що можна скомпілювати на будь-яку іншу платформу. Також SDL поширюється як вихідний код, написаний на мові C. Повністю підтримує роботу з C++ та частково працює з кількома іншими мовами, включаючи C# і Python [11].

1.4 UML – як засіб проєктування систем

Unified Modeling Language – це стандартизована мова моделювання, що складається з набору діаграм, що розроблені для візуалізації та побудови

програмних систем, бізнес-моделювання та інших непрограмних систем. UML являє собою збірку найкращих рішень, що неодноразово доводили це в моделюванні систем різного складу та складності. Графічні діаграми допомагають командам спілкуватися на зрозумілій одне одному мові, досліджувати потенційні проєкти та перевіряти архітектурний дизайн програмного забезпечення. Повний перелік діаграм дозволяє у повній мірі описати систему на різних рівнях деталізації.

UML є стандартом, тому він має чітко визначений синтаксис і семантичну модель, яка дозволяє розробникам розуміти діаграми, створені іншими. Ці діаграми можна використовувати для моделювання структури, поведінки, інтерфейсів та інших аспектів системи на різних типах і етапах проєктів програмного забезпечення. Використання UML як єдиної мови розробки дозволяє всім учасникам проєкту розуміти один одного та ефективно обговорювати, аналізувати та документувати вимоги до програмного забезпечення, тим самим знижуючи ризик непорозумінь.

UML має велику кількість діаграм, які дозволяють створювати великі та складні системи, а потім розкладати їх на незалежні, менш складні та легкокеровані системи. UML діаграми є гарним засобом документування проєкту, що забезпечують детальний опис архітектури та функціональності програмного забезпечення, дозволяючи зберігати інформацію про проєкт у структурованому та зрозумілому форматі для майбутніх розробників та інших зацікавлених сторін [12].

1.5 Метод тайлінгу

Для створення мапи було обрано метод тайлінгу. Суть цього методу у створенні набору «плиток» текстур, які можна поєднувати один з одним у багатьох комбінаціях, адже їх краї зроблені таким чином, що текстура плавно переходить на наступний тайл. Розміри тайлів можуть бути різного розміру,

проте зазвичай використовуються розміри 16x16 або 32x32. Кожен тайл унікальний, тобто існує одним екземпляром. Для побудови мапи кожен тайл нумерується, і потім з них вибудовується уся мапа записуючи номер тайлу у матрицю мапи на відповідне місце.

Такий метод дозволяє економити пам’ять, адже не треба для кожної мапи зберігати додаткове зображення великого розміру (див. рис. 1.7).

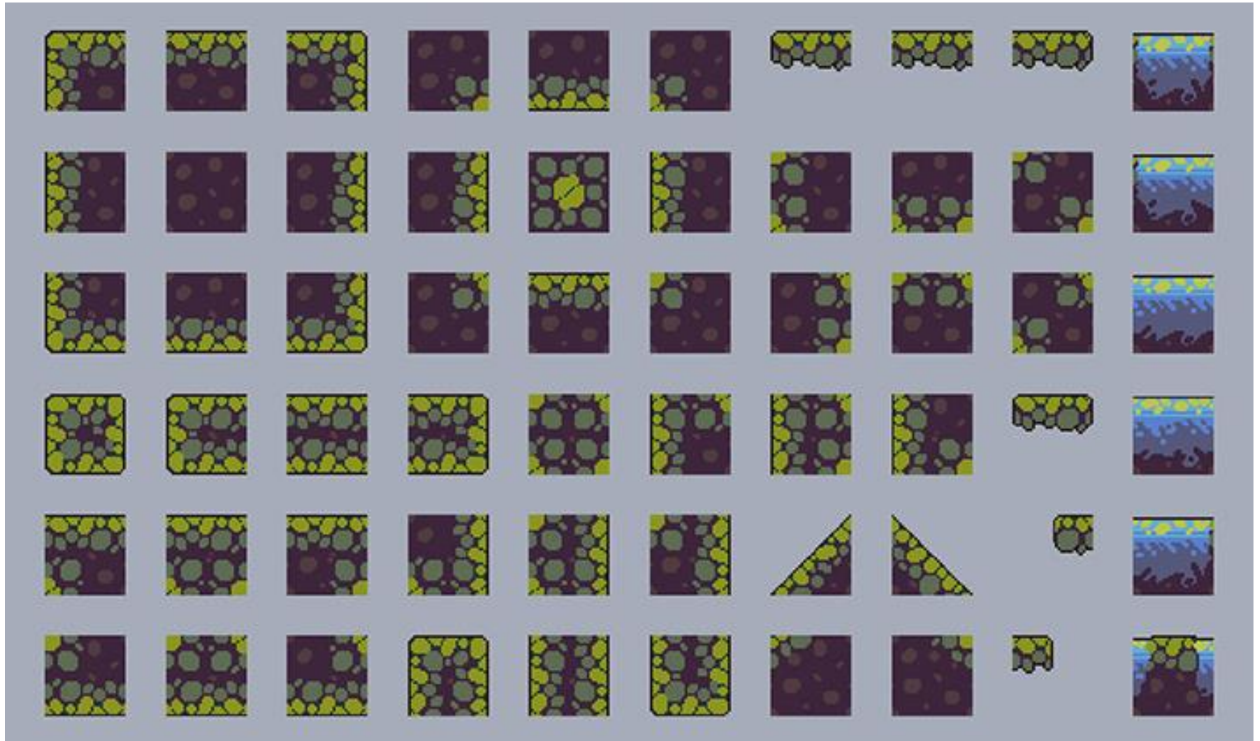

Рисунок 1.7 – Приклад набору тайлів

1.6 Висновки до розділу 1

У першому розділі було ознайомлено з історією розвитку ігор та обрано цільовий жанр для дослідження та розробки. Було проаналізовано обрану предметну область та досліджено представників обраного жанру. Розглянуто інструменти для розробки цільової системи, а саме мову програмування C++ та графічну бібліотеку SDL. Було досліджено метод тайлінгу для оптимізації системи. Досліджено та обрано діаграми UML для проєктування системи.

2 ПРОЄКТУВАННЯ СИСТЕМИ

До розроблювальної системи було обрано наступні вимоги: гра у жанрі «платформер» з рівнями та ворогами, механікою бою та пересування по рівням.

2.1 UML діаграми

Для створення гри було використано діаграми прецедентів, діяльності та послідовності опису її головних аспектів, порядку подій та загальної структури.

2.1.1 Діаграма прецедентів

Діаграма прецедентів, або Use Case Diagram, показує основних дійових осіб – акторів, та основний функціонал системи, описує функціональні вимоги системи з точки зору варіантів використання. Варіанти використання дозволяють зв’язати вимоги до системи зі способом задоволення цих вимог [13].

Для системи було обрано наступний перелік акторів та прецедентів:

- Player (Гравець) – актор, що уособлює користувача системи;
- Enter to Menu (Повернення до Меню) – прецедент потрапляння до головного меню при запуску гри чи закінченні;
- Play game (Грати гру) – прецедент початку гри;
- Next Level (Наступний рівень) – перехід на наступний рівень при вдалому проходженні попереднього;
- Game over (Кінець гри) – поразка на певному рівні, або успішне

проходження усіх рівнів;

– Back to Menu (Повернення до Меню) – головне меню після закінчення гри;

– Enemy Control (Управління ворогами) – штучний інтелект ворогів на рівні (див. рис. 2.1).

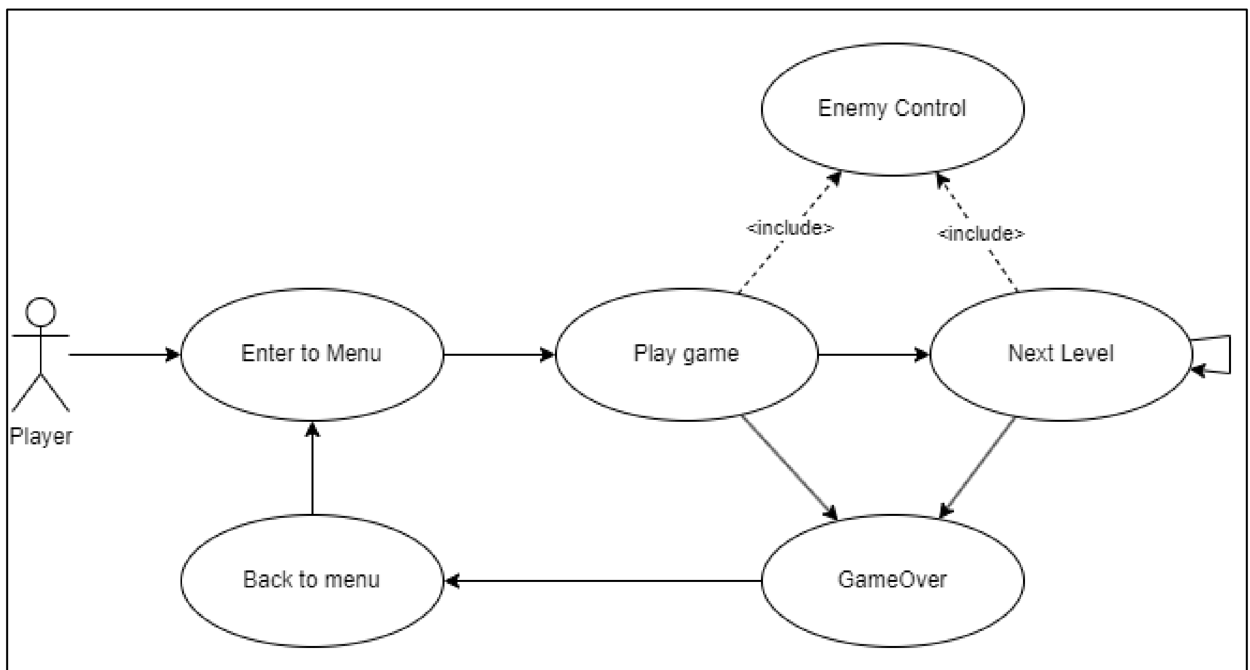

Рисунок 2.1 – Use Case diagram

2.1.2 Діаграма діяльності

Діаграма діяльності візуально представляє ряд дій або потік керування в системі, подібний до блок-схеми або діаграми потоку даних. Діаграми діяльності часто використовуються при моделюванні бізнес-процесів. Вони також можуть описувати кроки на діаграмі варіантів використання.

Змодельовані дії можуть бути послідовними та одночасними. В обох випадках діаграма діяльності матиме початок (початковий стан) і кінець (кінцевий стан) [14]. Для опису потоку подій у системі було розроблено наступну діаграму діяльності (див. рис. 2.2).

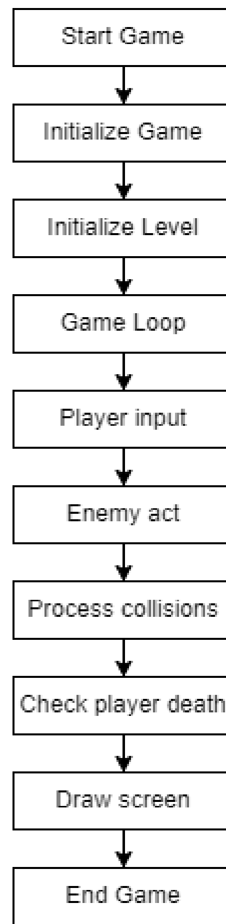

Рисунок 2.2 – Activity diagram

2.1.3 Діаграма послідовності

Діаграма послідовності UML – це діаграма взаємодії, яка детально описує виконувану операцію. Вона зображує взаємодію між об’єктами в контексті їх співпраці. Ця діаграма зосереджена на часі і візуально показують порядок взаємодії за допомогою вертикальної осі діаграми для відображення часу, які повідомлення надсилаються [15].

У наступній діаграмі показано взаємодію між об’єктами системи у головному ігровому процесі. Після початку гри створюється мапа рівня та завантажується. Далі завантажуються потрібні ресурси та створюються об’єкти супротивників. Після цього починається ігровий цикл у якому за один кадр виконуються дії у користувача, ворогів та на мапі. Перевіряються зіткнення та умова для кінця гри, після чого гра завершується (див. рис. 2.3).

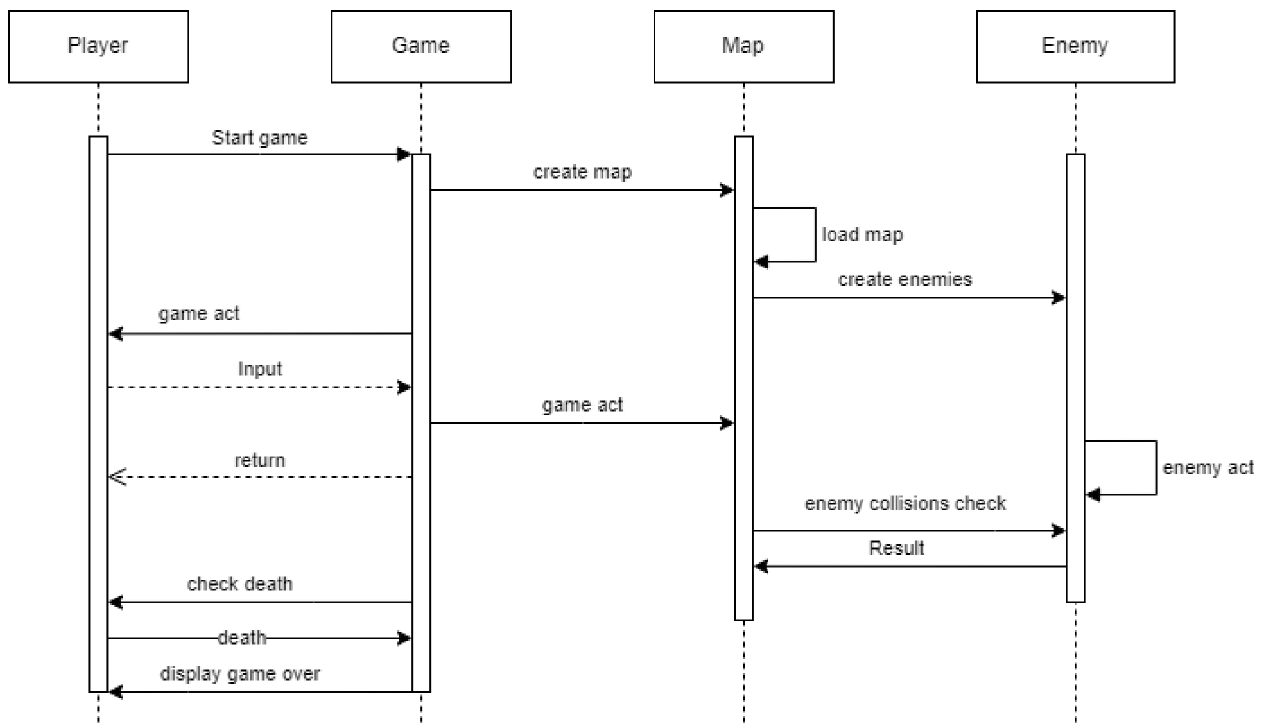

Рисунок 2.3 – Sequence diagram

2.2 Головні механіки та рішення. Використані патерни проєктування

Для правильного відображення на екрані було розроблено систему з класу Camera та сімейства класів Drawable. Карти рівнів можуть бути різних розмірів, тому для статичного розміру екрану використовується клас камери, що переміщається по карті слідуючи за гравцем. Сам клас має опис розмірів відображеного вікна гри, що можуть не співпадати з розмірами вікна програми та екрану. Також клас Camera має методи для пересування в межах карти.

Сімейство класів Drawable розроблене для опису елементів, що відображаються на екрані та має наступну структуру (див. рис. 2.4).

Для правильного керування рухомими об’єктами було реалізовано Машину станів.

Машина станів – це математична абстрактна модель, що складається з кінцевого набору станів, кожен з яких має перехід до іншого стану відповідно

до вхідних даних, та який по своєму керує системою [16]. Цей принцип покладено в основу поведінкового патерну проєктування Стан, який було використано для реалізації різних варіантів поведінки істот. Згідно патерну, існує один інтерфейс з одним або декількома методами, які буде викликати цільовий об’єкт, та декілька класів-нащадків, що по своєму реалізують ці методи, чим змінюють поведінку об’єкту, ніби змінився весь об’єкт. У самого цільового об’єкту також має бути відкритий метод зміни стану, щоб переходи між станами могли існувати [17].

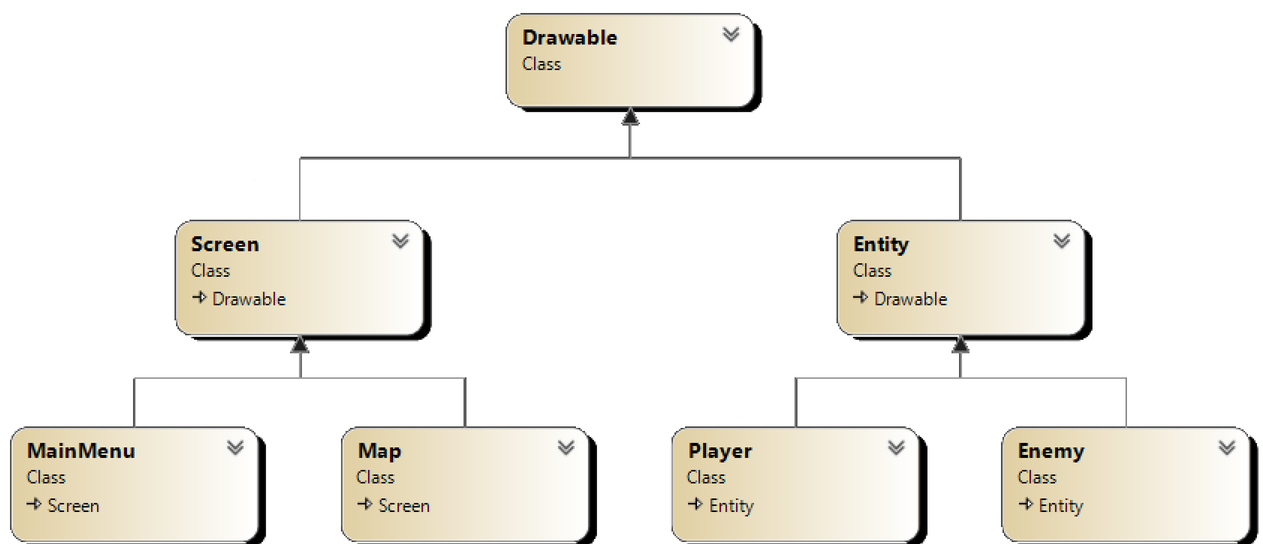

Рисунок 2.4 – Діаграма класів сімейства Drawable

Таким чином було реалізовано наступні стани (див. рис. 2.5):

- Idle – стан спокою, викликається коли істота нічого не робить;
- Run – стан пересування істоти (викликається через методи move);
- Jump – стан стрибку та падіння (викликається клавішою стрибку у головного героя);
- Attack – стан атаки (саме у цьому стані викликається метод attack);
- Hurt – стан отримання шкоди (викликається при успішній атаці суперника);
- Death – стан смерті істоти (викликається лише зі стану отримання шкоди, блокує будь яке управління).

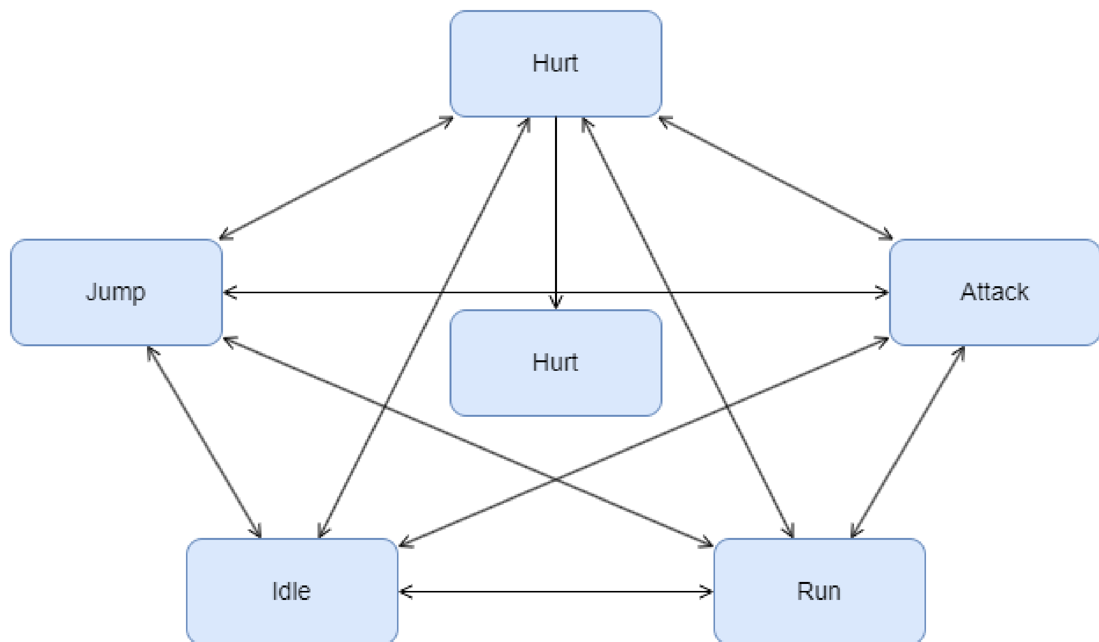

Рисунок 2.5 – Діаграма Машини Станів

2.3 Висновки до розділу 2

У другому розділі було розроблено три UML діаграми, а саме діаграма прецедентів, діяльності та послідовності. Було описано структуру класів системи для відображення на екрані та створено діаграму класів для її демонстрації. Використано патерн проєктування «Стан» для опису поведінки рухомих сутностей у залежності від навколишніх подій, чи від введення команд користувачем.

3 РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Мінімальні вимоги до гри та користувачів

Створена система має певні обмеження для використання, без виконання яких правильна робота не гарантується.

Гарантована правильна робота на таких вимогах:

- у операційній системі Windows версії не нижче 10;
- процесор має мати два ядра та тактову частоту не нижче 1 гігагерц (рекомендовано не менше 4 ядер та 2 гігагерц);
- оперативна пам’ять місткістю не менше 2 гігабайт;
- 100 мегабайт пам’яті на жорсткому диску;
- відеоадаптер, що сумісний з “DirectX 9”.

Також обов’язкова наявність клавіатури та миші для можливості керування грою, та монітор з мінімальним розширенням «800 x 600».

3.2 Створення базового вікна

Вхідною точкою в гру є файл main.cpp с однойменним методом всередині. Цей метод створює об’єкт класу Game, що являє собою Фасад для усіх компонентів гри.

У конструкторі класу Game проходить ініціалізація систем SDL та ігрових даних. А саме викликаються наступні методи:

- SDL_Init – метод ініціалізації основної системи SDL;
- IMG_Init – метод ініціалізації компонента SDL_Image для роботи з файлами зображень;
- SDL_CreateWindow – метод створення вікна програми;
- SDL_CreateRenderer – метод створення об’єкту класу SDL_Renderer, що

потрібен для відображення вікна засобом графічного процесора комп'ютера;

– TTF_OpenFont – метод для створення об'єкту шрифта.

Кожен метод перевіряється на наявність помилки, що провокує завершення програми. Окрім вищезазначених методів також у конструкторі створюється текстура завантаження, що буде з'являтись при переході між сценами, та завантажуються дані про ігрові об'єкти (див. рис. 3.1).

```
	if (SDL_Init(SDL_INIT_VIDEO) < 0){
		…
	}
	window = SDL_CreateWindow("SDL Tutorial", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, GAME_WIDTH, GAME_HEIGHT,
SDL_WINDOW_SHOWN | SDL_WINDOW_BORDERLESS);
	renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
	//Initialize SDL_ttf
	font = TTF_OpenFont(fontPath.c_str(), 28);
```

Рисунок 3.1 – Завантаження систем SDL у класі Game

Наступним кроком викликається метод завантаження головного меню. Далі починається головний цикл, у якому зчитуються натиснуті та відпущені клавіші. Після цього вираховується час від попереднього кроку циклу. Якщо цей час вище або дорівнює зазначеному часу кадру то викликаються послідовно методи act(delay) та render() для запуску кадру та малюванні нового кадру. Delay передається у метод act для вирахування позицій об'єктів згідно часу. Далі цикл повторюється.

Умовою виходу із циклу є змінна quit, яка за замовчуванням має значення false. У процесі гри клас Game може змінити це значення, що призведе до виходу з гри. Метод main повертає значення, яке повертається методом quit класу Game, що провокує звільнення пам'яті та вихід з системи (див. рис. 3.2).

```
while (!quit) {
SDL_Event e;
	while (SDL_PollEvent(&e) != 0){
		if (e.type == SDL_QUIT)
		{
			quit = true;
		}
		if (e.type == SDL_KEYDOWN && e.key.keysym.sym == SDLK_ESCAPE) {
			quit = true;
		}
		game.handleEvent(e);
	}
	Uint32 newTicks = SDL_GetTicks();
	delay += newTicks - oldTicks;
	if (delay >= 10) {
		game.act(delay);
		delay = delay % 10;
		game.render();
	}
	oldTicks = newTicks;
}
```

Рисунок 3.2 – Головний цикл гри

3.3 Система класів для відображення на екрані

Клас Drawable являє собою інтерфейс, що оголошує загальні для усіх нащадків методи, які вони повинні реалізувати по своєму:

– метод act описує зміни або дії, які мають відбутись з об’єктом за один ігровий кадр;
– метод render викликається для відображення об’єкта на екрані;
– метод getRect повертає об’єкт класу SDL_Rectangle, що описує положення та розміри прямокутника об’єкта;
– метод getFocus повертає об’єкт класу SDL_Point, що описує положення

точки, у яку «прагне» потрапити камера (див. рис. 3.3).

Абстрактний клас Screen, що наслідується від Drawable, є абстрактним класом, що описує певну ігрову сцену. Не оголошує нових методів. Оголошує посилання на об’єкти класів Game, Data та SDL_Renderer, та реалізує єдиний конструктор для їх ініціалізації.

```
class Drawable {
public:
	virtual void render(Camera&, SDL_Renderer*) = 0;
	virtual void act(Uint32) = 0;
	virtual SDL_Rect getRect() = 0;
	virtual SDL_Point getFocus() = 0;
	virtual ~Drawable() { }
};
```

Рисунок 3.3 – Код класу Drawable

Зміна сцен у грі відбувається за рахунок заміни одного об’єкту типу Screen на інший. Особливістю мови програмування C++ є тип змінної «покажчик», що являє собою адресу на ділянку у пам’яті. Покажчик типу Screen можна ініціалізувати об’єктом типу класів-нащадків класу Screen. Викликати можна лише методи класу Screen, проте буде виконуватись реалізація класу, яким було ініціалізовано покажчик. Таким чином у класі Game оголошений покажчик типу Screen, що є відображенням поточної сцени, та методами loadMainMenu та setNewScreen, що реалізують завантаження сцени головного меню, що реалізована класом MainMenu, та зміну сцени на визначену у заданому параметрі.

3.4 Створення ігрових сцен

Клас MainMenu реалізує інтерфейс головного меню, яке бачить користувач після запуску гри. Метод render виводить на екран дві заготовлені

текстури кнопок за координатами. У методі act оброблюються натискання мишко користувачем для виходу з гри, або запуску першого рівня.

Віртуальний деструктор явно вивільняє динамічну пам'ять та знищує завантажені текстури (див. рис. 3.4).

```
class Map : public Screen {
	int curLvl;
	int width, height;
	int** tileNums;
	int enemyCount;
	vector<Collide*> walls;
	vector<SDL_Texture*> tiles;
	vector<Entity*> enemies;
	Player* player;
	Rectangle winZone;
public:
	string error;
	bool isError;
	Map(Game*, Data*, SDL_Renderer*);

	virtual ~Map();
	virtual void render(Camera&, SDL_Renderer*) override;
	virtual void act(Uint32) override;
	virtual SDL_Rect getRect() override;
	virtual SDL_Point getFocus() override;
private:
	bool loadTiles(XMLElement*);
	bool loadObjects(XMLElement*, Data*, SDL_Renderer*);
	bool loadTextures(SDL_Renderer*, Data*);
	bool loadMapData();
	bool nextLevel();
	void restart();
};
```

Рисунок 3.4 – Код оголошення класу Map

Клас Map реалізує малювання рівня та ігрових істот із заготовлених текстур, та контролює усі зіткнення між ними. За допомогою методу nextLevel є можливість переходу на наступний рівень. Клас сам завантажує дані нового рівня методом loadMapData не змінюючи сцену. Цей метод відкриває файл мапи та отримує з нього розміри карти, на основі яких виділяється пам’ять у динамічному подвійному масиві tileNums.

Після цього викликаються методи loadTiles та loadObjects, що завантажують з файлу мапи матрицю номерів тайлів, яку записує у tileNums, та дані про зони зіткнень й точки відродження істот відповідно. Також у методі loadObjects створюються об’єкти істот на основі даних позицій їх відродження.

Метод loadTextures викликається один раз у конструкторі для завантаження текстур тайлів, які мають бути спільні для усіх мап за принципом методу тайлінгу.

Кожен кадр гри перевіряється чи живий головний персонаж, оновлюється кількість живих ворогів та умови закінчення рівня, та перевіряється натискання клавіші ESC, що викликає вихід на головне меню.

3.5 Створення мапи

Для використання методу тайлінгу був використаний редактор мап Tiled. Він дозволяє у режимі реального часу будувати мапу у графічному інтерфейсі та експортувати її у вигляді файлу з розширенням .tmx. Також, для зручності, ця програма дозволяє створити файл .tsx, що у собі пов’язує шлях до тайлу з його порядковим номером, що дозволяє не витрачати час на пошук потрібного тайлу, а дозволяє звернутись до нього одразу за індексом. Обидва вихідні файли насправді є файлами типу XML, але перейменовані для зручності (див. рис. 3.5).

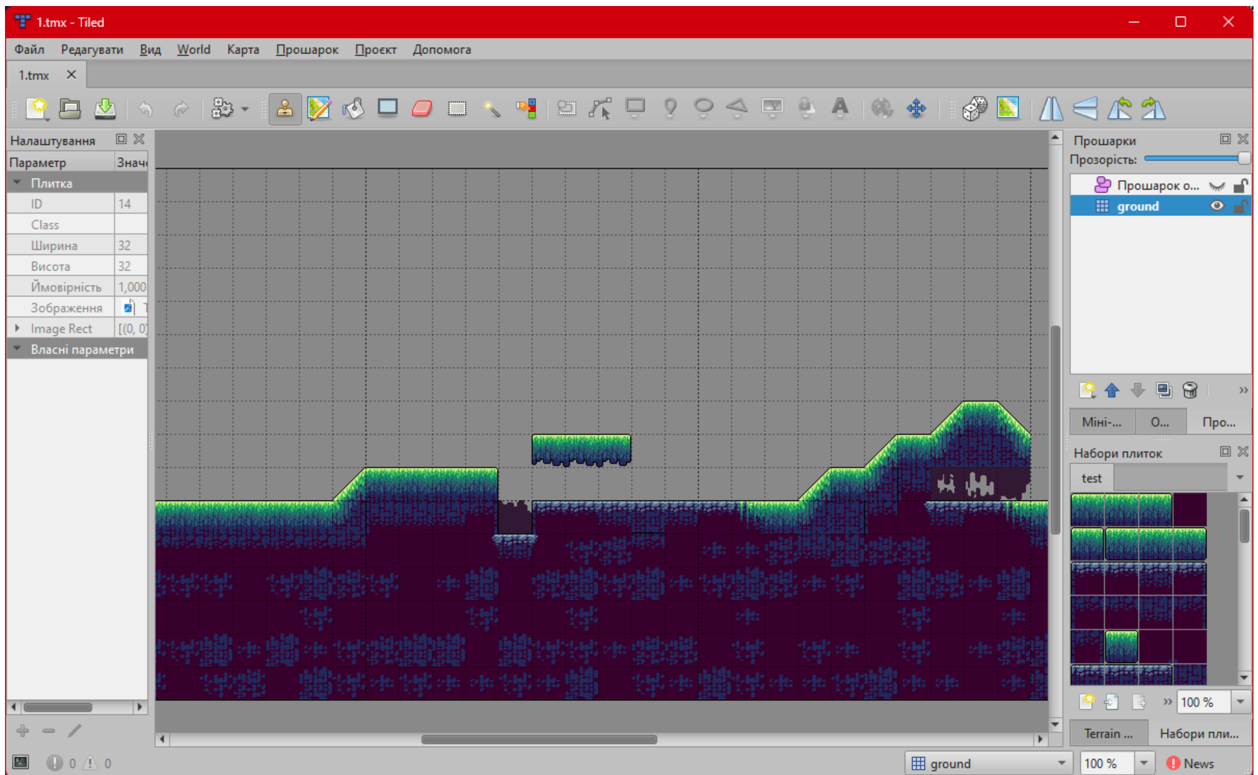

Рисунок 3.5 – Інтерфейс програми Tiled

Також Tiled дозволяє будувати невидимі об’єкти на мапі, які можна використовувати для позначення типу та положення зони зіткнень та місць відродження істот.

Для зчитування цих файлів використано безкоштовну бібліотеку TinyXML2, що дозволяє легко знаходити потрібні дані у файлі. Бібліотека розповсюджується відритим вихідним кодом та компілюється з проєктом.

3.6 Створення системи колізії

Кінцевий варіант гри двовимірний та має поняття сили тяжіння. Для того, що б істоти не падали нижче землі та не заходили за межі мапи та стін було розроблено власну систему перевірки зіткнень.

У бібліотеці SDL існують методи для перевірки перетину прямокутника за іншим прямокутником або точкою. Проте у грі присутні не тільки прямокутні тайли, а й трикутні, які повинні мати власну логіку зіткнень. Також

у результаті зіткнень істоти повинні «розуміти» яким чином реагувати на них. Тому було власна система зіткнень була необхідним рішенням.

У корені невеликого сімейства покладено абстрактний клас Collide, що оголошує методи для перевірки зіткнень. Ці методи реалізують класи-нащадки Square, LeftTriangle та RightTriangle, що уособлюють прямокутну зону та ліво- й правосторонні трикутні зони відповідно. Кожен з них по своєму реалізує методи зіткнень з точкою чи прямокутником.

Результатом є об’єкт структури CollisionInfo. Ця структура оголошує наступні змінні:

– x1 – відстань, яку має пройти об’єкт A, що б його права сторона доторкнулась до лівої сторони об’єкта B, але не перетнути її;
– x2 – відстань, яку має пройти об’єкт A, що б його ліва сторона доторкнулась до правої сторони об’єкта B, але не перетнути її;
– y1 – відстань, яку має пройти об’єкт A, що б його нижня сторона доторкнулась до верхньої сторони об’єкта B, але не перетнути її;
– y2 – відстань, яку має пройти об’єкт A, що б його верхня сторона доторкнулась до нижньої сторони об’єкта B, але не перетнути її.

Іншими словами, якщо перемістити об’єкт A на відстань вказану у будь-якій змінній CollisionInfo у відповідний напрямок то цей об’єкт не буде перетинати об’єкт B, але буде максимально близько до цього. Напрямок має обов’язкове значення.

Не дивлячись на те, що у трикутних зонах всього три сторони, логіка результату не зміниться, а так як трикутники прямокутні та рівносторонні то відстань y1 та одна з X-відстаней, що відрізняються залежно від направлення трикутника, будуть рівні (див. рис. 3.6).

Така система дозволяє контролювати напрямки зіткнень та корегувати положення між об’єктами максимально близько, ніби вони дотикаються.

Після того як створено систему зіткнень, першу мапу та відображення на екрані для повноцінного ігрового рушія не вистачає лише рухомих об’єктів.

```
CollisionInfo Rectangle::checkCollision(const Rectangle& box) {
	int x1 = box.x + box.width - x;
	int x2 = box.x - x - width;
	if (x1 < 0 || x2 > 0) {
		return { false };
	}
	int y1 = box.y + box.height - y;
	int y2 = box.y - y - height;
	if (y1 < 0 || y2 > 0) {
		return { false };
	}
	if (y1 != 0) y1 = y1;
	return { true, -x1,-x2,-y1,-y2 };
}
```

Рисунок 3.6 – Приклад розрахунку відстаней зіткнення з Rectangle

3.7 Створення сімейства створінь та їх поведінки

Для реалізації рухомих об’єктів було створено сімейство класів Entity, що є нащадками Drawable. Кореневий абстрактний клас Entity оголошує та реалізує основні атрибути та методи для усього сімейства, залишаючи не перевантаженим лише метод act, який нащадки мають перевантажити та віртуальний метод attack.

Клас-нащадок Player додає лише один атрибут – посилання на список ворогів, з якими він має взаємодіяти у механіці бою.

Другий нащадок Enemy за аналогією має посилання на об’єкт Player та має додаткову змінну-флаг, що показує чи знайдена ціль для атаки.

Обидва нащадки реалізують метод attack схожим чином – перевіряється чи є ворог/и у зоні враження та, якщо це так, викликають у цілі метод takeDamage з параметрами шкоди та напрямку удару.

Кожен стан перевантажує метод act, який викликається з об’єкту Player

або Enemy. Стани для Enemy відрізняються відсутністю керування ззовні. Для них логіка керування закріплена у класі Enemy.

Другою важливою функцією станів у системі стала керування анімаціями текстур. Кожен стан завантажує свій набір текстур, який по черзі виводить на екран. Окремо слід виділити стани Attack та Death. Перший чекає на певний кадр для виклику методи attack, а другий виконує анімацію один раз та зберігає останню текстуру.

Усі стани зберігаються у колекції типу map, що зберігає пари «Тип стану – Стан» у класі Entity. Також цей клас зберігає посилання на список стін, що передається з класу Map через конструктор, змінні параметрів істоти, такі як кількість здоров'я, швидкість, шкода та положення у просторі. Для перевірки атаки є дві прямокутні зони hitbox та attackbox. Метод collissionUpdate перевіряє зіткнення з усіма зонами зіткнень та виправляє положення згідно результату.

3.8 Правильне відображення текстур у вікні

Так як розміри мап, камери та вікна гри відрізняються усі координати потрібно перетворити, а загальну картинку гри масштабувати.

Для першої цілі у кожен метод render передається камера, координати якої віднімаються від координат об'єкту, що малюється.

Друга проблема вирішується шляхом можливостей бібліотеки SDL. Завдяки методу SDL_SetRenderTarget класу SDL_Renderer можна перенаправити вивід картинки з екрану на окремий об'єкт SDL_Texture.

Усі об'єкти малюються на цій текстурі у повному розмірі. Потім ця текстура масштабується під потрібний розмір у методі SDL_RenderCopy, у параметрах якого можна передати прямокутник цільового екрану (див. рис. 3.7).

```
void Game::render() {
	SDL_Texture* rawView = SDL_CreateTexture(renderer, SDL_PIXELFORMAT_RGBA8888, SDL_TEXTUREACCESS_TARGET, camera.rect.w, camera.rect.h);
	}
	SDL_SetRenderTarget(renderer, rawView);
	SDL_SetRenderDrawColor(renderer, 0xFF, 0xFF, 0xFF, 0xFF);
	SDL_RenderClear(renderer);
	SDL_Rect back = { 0,0,camera.rect.w, camera.rect.h };
	SDL_RenderCopy(renderer, background, NULL, &back);
	screen->render(camera, renderer);
	SDL_Rect pos{0,0,GAME_WIDTH,
		camera.rect.h * (float)GAME_WIDTH / (float)camera.rect.w
	};
	//Clear screen
	SDL_SetRenderTarget(renderer, NULL);
	SDL_SetRenderDrawColor(renderer, 0xFF, 0xFF, 0xFF, 0xFF);
	SDL_RenderClear(renderer);
	SDL_RenderCopy(renderer, rawView, NULL, &pos);
	//Update screen
	SDL_RenderPresent(renderer);
	SDL_DestroyTexture(rawView);
}
```

Рисунок 3.7 – Код масштабування картинки

3.9 Клас менеджеру даних гри та додаткові інструменти

Багато компонентів системи мають містити посилання на файли та дані про ці файли. Розміщувати у кожному класі свої дані не найкраще рішення, адже при будь-якій зміні файлової системи доведеться знову шукати місця старих даних. Тому було вирішено створити структуру, що централізовано міститиме усі потрібні дані.

Для збереження даних було обрано мову розмітки XML. У каталозі гри створено файл data.xml, який буде зберігати усю потрібну інформацію. Відповідно до структури файлу створено набір класів, що уособлюють елементи файлу.

Клас Data зберігає об'єкти цих класів та у конструкторі приймає відкритий xml файл, у якому шукає потрібний елемент та передає його відповідному об'єкту, який оброблює його та записує дані. Об'єкт класу Data створюється у класі Game та передається у конструктори усім, хто його потребує (див. рис. 3.8).

```
struct Data {
	TilesData tiles;
	EntityData player;
	EntityData bandit[2];
	MenuData menu;
	MapsData maps;
	string backgroundPath;
	bool set(XMLDocument& d) {
		XMLElement* data = d.FirstChildElement("data");
		XMLElement* e = data->FirstChildElement("tiles");
		tiles.set(e);
		…
	}};
```

Рисунок 3.8 – Клас Data

Також було створено файл Tools.h, у якому записано додаткові глобальні функції, які можуть бути використані у будь-якому місці системи.

Вони містять такі функції:

– trim – функція для видалення «невидимих» символів з обох кінців рядка;
– split – «розрізає» рядок на підрядки за вказаним розділителем;
– loadTexture – приймає посилання на SDL_Renderer та рядок шляху до файлу (завантажує текстуру у пам'ять та повертає посилання на неї).

3.10 Огляд результату

Після запуску гри на екрані з’являється головне меню (див. рис. 3.9). Для початку гри треба натиснути кнопку Play. Для виходу з неї кнопка Exit.

Рисунок 3.9 – Головне меню

Після переходу у гру починається управління персонажем. На екрані з’являється обмежена частина карти, на якій є персонаж.

Клавіші A та D відповідають за пересування вліво та вправо відповідно. Клавіша SPACE відповідає за стрибок, а атака виконується лівою клавішою миші.

На карті з’являються вороги, яких треба здолати для переходу на наступний рівень. Щоб подолати ворогів треба використовувати «Удар», що викликається лівою клавішою миші. Використовувати «Удар» можливо лише знаходячись на землі.

Подолані вороги залишаються на землі до кінця рівня. Після подолання усіх ворогів відкривається доступ до наступного рівня (див. рис. 3.10).

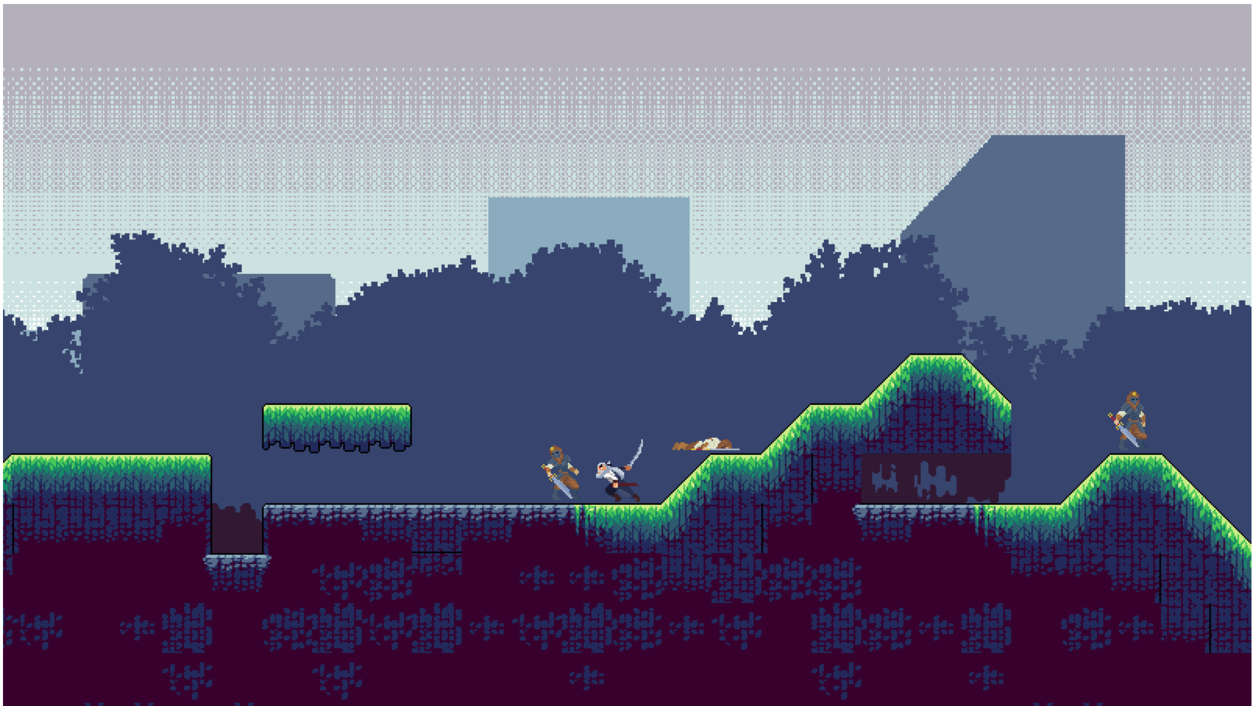

Рисунок 3.10 – Ігровий процес

3.11 Висновки до розділу 3

У третьому розділі було описано мінімальні вимоги до гри. Було реалізовано спроєктовану систему. Створено усі головні механіки та поєднано усі модулі системи. Реалізовано сімейства класів істот, сцен та станів. Створено перший рівень з допомогою програми Tiled та головне меню. Готову збірку перевірено на працездатність.

ВИСНОВКИ

В результаті кваліфікаційної роботи розроблено робочий ігровий застосунок у жанрі «платформер». Для створення додатку була використана бібліотека SDL та мова програмування C++. Було вивчено технології створення ігор та досліджено особливості використання бібліотеки SDL.

У першому розділі була проаналізована предметна область. Були переглянуті аналоги цільового продукту, основні інструменти для реалізації застосунку та метод тайлінгу.

У другому розділі була спроєктована та побудована структура гри, а саме було створено UML діаграми: діаграма прецедентів, діаграма діяльності та діаграма послідовності. Також було спроєктовано систему керування поведінкою об’єктів.

У третьому розділі було реалізовано усі необхідні компоненти гри. Створено базове вікно програми, ігрові сцени, система зіткнень, сімейство створінь, їх поведінка та додаткові інструменти. Об’єднано усі модулі в одну систему та протестовано на працездатність.

У результаті було отримано ігровий застосунок, який є коштовним продуктом, який має свою ціну та вплив на ігрову індустрію. Він дозволяє залучити нових людей до товариства розробників ігор, товариства гравців або товариства бізнесменів, які створюють свої мільйонні ігрові корпорації.

Отже, розробка ігор є перспективною галуззю розробки програмного забезпечення.

ПЕРЕЛІК ПОСИЛАНЬ

1. Pandey S. Problems With Monotonous Games And What Are Adaptive Gaming Solutions, Stackademic. URL: https://blog.stackademic.com/problem-with-monotonous-games-and-what-are-adaptive-gaming-solution-4012de5d8e9b (дата звернення: 28.01.2024).
2. Howarth J. How Many Gamers Are There? (New 2024 Statistics). Exploding Topics. URL: https://explodingtopics.com/blog/number-of-gamers#gamer-demographics (дата звернення: 05.02.2024).
3. Video Game History. History.com URL: https://www.history.com/topics/inventions/history-of-video-games (дата звернення: 07.02.2024).
4. Cohen D. S. OXO aka Noughts and Crosses – The First Video Game. Lifewire: Tech For Humans. URL: https://www.lifewire.com/oxo-aka-noughts-and-crosses-729624 (дата звернення: 09.02.2024).
5. Most Popular Video Game Genres In 2024: Revenue, Statistics, Genres Overview. RocketBrush Studio URL: https://rocketbrush.com/blog/most-popular-video-game-genres-in-2024-revenue-statistics-genres-overview (дата звернення: 11.02.2024).
6. The Editors of Encyclopedia Britannica. Donkey Kong. Britannica. URL: https://www.britannica.com/topic/Donkey-Kong (дата звернення: 13.02.2024).
7. History Of Mario. Tech-n-Gamer. URL: https://www.techngamer.net/history-of-mario (дата звернення: 15.02.2024).
8. Vasconcelos T. #44 Sonic the Hedgehog: A Sprint Through History – The Creation of a Gaming Icon. Linkedin. URL: https://www.linkedin.com/pulse/44-sonic-hedgehog-sprint-through-history-creation-icon-vasconcelos-ebime (дата звернення: 17.02.2024).
9. Team cherry. Hollow Knight. URL: https://www.hollowknight.com (дата звернення: 19.02.2024).

10. Johns R., Semah B. 7 Best Programming Languages for Game Development in 2024. Hackr.io URL: https://hackr.io/blog/best-programming-language-for-games (дата звернення: 20.02.2024).
11. SDL team. SDL. Simple DirectMedia Layer. URL: https://www.libsdl.org (дата звернення: 21.02.2024).
12. OMG® SDO. Unified Modeling Language. URL: https://www.uml.org (дата звернення: 23.02.2024).
13. Lucidchart team. UML Use Case Diagram Tutorial. Lucidchart URL: https://www.lucidchart.com/pages/uml-use-case-diagram (дата звернення: 24.02.2024).
14. What is Activity Diagram? URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/ (дата звернення: 25.02.2024).
15. Lucidchart team. UML Sequence Diagram Tutorial. Lucidchart URL: https://www.lucidchart.com/pages/uml-sequence-diagram (дата звернення: 26.02.2024).
16. What is a state machine? URL: https://www.itemis.com/en/products/itemis-create/documentation/user-guide/overview_what_are_state_machines (дата звернення: 27.02.2024).
17. Shvets A. State. URL: https://refactoring.guru/design-patterns/state (дата звернення: 29.02.2024).

ДОДАТОК А

Код класів сімейства Screen та класу Drawable

А.1 Файл Screens.h

```
#pragma once
#include "Drawable.h"
#include "Entity.h"
#include "Tools.h"
#include <filesystem>
#include <iostream>
#include <SDL_Image.h>
#include <string>
#include <vector>
namespace fs = std::filesystem;
using std::vector, std::string;
namespace myGame {
	class Game;
	class Screen : public Drawable {
	protected:
		Game *game;
		Data* data;
		SDL_Renderer* renderer;
		Screen(Game* _game, Data* data, SDL_Renderer* r) : game(_game), data(data), renderer(r) {}
	};
	class MainMenu : public Screen {
		SDL_Texture* playButtonTexture;
		SDL_Texture* exitButtonTexture;
		SDL_Rect play;
		SDL_Rect exit;
		int mX, mY;
		bool mPressed;

	public:
		MainMenu(Game* game, Data* data, SDL_Renderer* r);
		virtual ~MainMenu() {}
		virtual void render(Camera&, SDL_Renderer*) override;
		virtual void act(Uint32) override;
		virtual SDL_Rect getRect() override;
		virtual SDL_Point getFocus() override;
	};

	class Map : public Screen {
		int curLvl;
		int width, height;
		int** tileNums;
		int enemyCount;
		vector<Collide*> walls;
		vector<SDL_Texture*> tiles;
		vector<Entity*> enemies;
		Player* player;
		Rectangle winZone;
	public:
		string error;
		bool isError;
```

```
		Map(Game*, Data*, SDL_Renderer*);

		virtual ~Map();
		virtual void render(Camera&, SDL_Renderer*) override;
		virtual void act(Uint32) override;
		virtual SDL_Rect getRect() override;
		virtual SDL_Point getFocus() override;
	private:
		bool loadTiles(XMLElement*);
		bool loadObjects(XMLElement*, Data*, SDL_Renderer*);
		bool loadTextures(SDL_Renderer*, Data*);
		bool loadMapData();
		bool nextLevel();
		void restart();
	};
}
```

А.2 Файл Screens.cpp

```
#include "Screens.h"
#include "Game.h"
using tinyxml2::XML_NO_ATTRIBUTE;
namespace myGame {
	MainMenu::MainMenu(Game* game, Data* data, SDL_Renderer* r)
		: Screen(game, data, r), play{ 0,0,0,0 }, exit{ 0,0,0,0 } {
		playButtonTexture = loadTexture(renderer, data->menu.playButPath);
		exitButtonTexture = loadTexture(renderer, data->menu.exitButPath);

		play.w = data->menu.width * 10;
		exit.w = data->menu.width * 10;
		play.h = data->menu.height * 10;
		exit.h = data->menu.height * 10;

		play.x = game->CAMERA_WIDTH / 2 - play.w / 2;
		play.y = game->CAMERA_HEIGHT / 2 - play.h / 2 - 100;
		exit.x = game->CAMERA_WIDTH / 2 - exit.w / 2;
		exit.y = game->CAMERA_HEIGHT / 2 - exit.h / 2 + 100;
	}
	void MainMenu::render(Camera& camera, SDL_Renderer* renderer) {

		SDL_RenderCopy(renderer, playButtonTexture, NULL, &play);
		SDL_RenderCopy(renderer, exitButtonTexture, NULL, &exit);
	}
	void MainMenu::act(Uint32 delay)
	{
		const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);
		int mouseX = -1, mouseY = -1;

	if (SDL_GetMouseState(&mouseX, &mouseY) & 1) {
			mX = mouseX * game->CAMERA_WIDTH / game->GAME_WIDTH;
			mY = mouseY * game->CAMERA_HEIGHT / game->GAME_HEIGHT;
			mPressed = true;

		}
		else if (mPressed) {
			SDL_Point point(mX, mY);
			if (SDL_PointInRect(&point, &play)) {
```

```
                        mPressed = false;
                        game->setNewScreen(new Map(game, data, renderer));
                        return;
                    }
                    if (SDL_PointInRect(&point, &exit)) {
                        mPressed = false;
                        SDL_Event event{};
                        event.type = SDL_QUIT;
                        event.quit.timestamp = SDL_GetTicks();
                        SDL_PushEvent(&event);
                        return;
                    }
                }
            }
            SDL_Rect MainMenu::getRect()
            {
                return { 0,0,0,0 };
            }

            SDL_Point MainMenu::getFocus()
            {
                return SDL_Point();
            }

            Map::Map(Game *game, Data* data, SDL_Renderer* renderer) : Screen(game, data, renderer),
player(0), tileNums{ nullptr }, tiles{}, winZone(0,0,0,0), curLvl(1), enemyCount(0) {
                loadTextures(renderer, data);
                loadMapData();
            }
            Map::~Map() {
                for (size_t i = 0; i < height; i++)
                {
                    delete[] tileNums[i];
                }
                delete[] tileNums;

                for (size_t i = 0; i < walls.size(); i++)
                {
                    delete walls[i];
                }
                for (size_t i = 0; i < tiles.size(); i++)
                {
                    SDL_DestroyTexture(tiles[i]);
                }
                for (size_t i = 0; i < enemies.size(); i++)
                {
                    delete enemies[i];
                }

                delete player;
            }
            void Map::render(Camera& camera, SDL_Renderer* renderer) {
                SDL_Rect pos = { 0,0,32,32 };
                for (size_t i = 0; i < height; i++)
                {
                    for (size_t j = 0; j < width; j++)
                    {
                        if (tileNums[i][j] < 0) {
                            continue;
                        }
                        pos.x = j * pos.w;
                        pos.y = i * pos.h;
                        if (SDL_HasIntersection(&pos, &camera.rect) == SDL_TRUE) {
```

```
                                        pos.x -= camera.rect.x;
                                        pos.y -= camera.rect.y;
                                                SDL_RenderCopy(renderer, tiles[tileNums[i][j]], NULL, &pos);
                                        }
                                }
                        }

#ifdef DEBUG
                        for (Collide* hitBox : walls)
                        {
                        hitBox->drawBorder(renderer, camera);
                        }
#endif // DEBUG

                        player->render(camera, renderer);

                        for (Entity* enemy : enemies) {
                                enemy->render(camera, renderer);
                        }
                }

                void Map::act(Uint32 delay)
                {
                        const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);

                        if (currentKeyStates[SDL_SCANCODE_ESCAPE])
                        {
                                game->setNewScreen(new MainMenu(game, data, renderer));
                                return;
                        }
                        player->act(delay);
                        for (Entity* enemy : enemies) {
                                enemy->act(delay);
                                if (enemy->isDead()) enemyCount--;
                        }
                        if (player->isDead()) {
                                //DEAD
                        }
                        if (player->getHitBox().checkCollision(winZone) && enemyCount == 0) {
                                //WIN
                        }
                }

                SDL_Rect Map::getRect()
                {
                        return SDL_Rect{ 0,0,width * 32,height * 32 };
                }
                SDL_Point Map::getFocus()
                {
                        return player->getFocus();
                }
                bool Map::loadTextures(SDL_Renderer* renderer, Data* data) {
                        string folder = data->tiles.place;
                        for (int i = 0; i < data->tiles.count; i++) {
                                string nameFile = "";
                                if (i < 10) nameFile = "0";
                                nameFile += std::to_string(i) + "." + data->tiles.type;

                                SDL_Texture* newTexture = loadTexture(renderer, folder + nameFile);
                        if (newTexture == NULL) return false;
                                tiles.push_back(newTexture);
                        }
```

```
#ifdef DEBUG
		for (const auto& entry : fs::directory_iterator(folder)) {
		std::cout << entry.path() << std::endl;
		}
#endif // DEBUG
	}
	bool Map::loadMapData()
	{
		string path = data->maps.place + std::to_string(curLvl) + ".tmx";
		XMLDocument doc;
		doc.LoadFile(path.c_str());
		XMLElement* map = doc.FirstChildElement("map");
		map->QueryIntAttribute("width", &width);
		map->QueryIntAttribute("height", &height);
		if (width == 0 || height == 0) {
			isError = true;
		error = "Failed read width/height data";
		}
		else {
			tileNums = new int* [height];
			for (size_t i = 0; i < height; i++)
			{
			tileNums[i] = new int[width];
			}
			loadTiles(map->FirstChildElement("layer"));
			loadObjects(map->FirstChildElement("objectgroup"), data, renderer);
		}
		return false;
	}
	bool Map::nextLevel()
	{
		curLvl++;
		if (curLvl == data->maps.count) {
			game->setNewScreen(new MainMenu(game, data, renderer));
			return true;
		}

		if (tileNums != nullptr) {
			for (size_t i = 0; i < height; i++)
			{
				if (tileNums != nullptr)
				delete[] tileNums[i];
			}
			delete[] tileNums;
		}

		loadMapData();
	}
	void Map::restart()
	{
		curLvl = 1;
		loadMapData();
	}
	bool Map::loadTiles(XMLElement* layer) {
		string data = layer->FirstChildElement("data")->GetText();
		if (data.empty()) return false;
		data = trim(data);
		vector<string> lines;
		split(data, lines, 10);
		if (lines.size() != height) return false;
		vector<string> buf;
		for (size_t i = 0; i < height; i++) {
			split(lines[i], buf, ',');
```

```
                if (buf.size() < width) return false;
                for (int j = 0; j < width; j++) {
                        tileNums[i][j] = stoi(buf[j]) - 1;
                }
                bool a = 0;
        }
#ifdef DEBUG
        for (size_t i = 0; i < height; i++)
        {
                for (size_t j = 0; j < width; j++)
                {
                printf("%2d, ", tileNums[i][j]);
                }
                printf("\n");
        }
#endif // DEBUG
        return true;
    }
    bool Map::loadObjects(XMLElement* objectgroup, Data* data, SDL_Renderer* r)
    {
        XMLElement* element = objectgroup->FirstChildElement("object");
        if (element == NULL) return false;
        while (element != NULL) {
                int x = 0, y = 0;
        string type = element->Attribute("type");
                if (type.empty()) return false;

                if (element->QueryIntAttribute("x", &x) == XML_NO_ATTRIBUTE) return false;
                if (element->QueryIntAttribute("y", &y) == XML_NO_ATTRIBUTE) return false;

                if (type == "rect") {
                        int w = 0, h = 0;
                        if (element->QueryIntAttribute("width", &w) == XML_NO_ATTRIBUTE)
return false;
                        if (element->QueryIntAttribute("height", &h) == XML_NO_ATTRIBUTE)
return false;

        walls.push_back(new Rectangle(x, y, w, h));
                }
                else if (type == "triangle")
                {
        string name = element->Attribute("name");
                        if (name == "right")
                                walls.push_back(new RightTriangle(x, y - RightTriangle::SIDE));
                        else                            walls.push_back(new LeftTriangle(x, y));
                }
                else if (type == "playerSpawn")
                {
                        player = new Player(x, y, &data->player, walls, enemies, r);
                }
                else if (type == "enemy")  {
                        enemies.push_back(new Enemy(x, y, &data->bandit[rand() % 2], walls,
player, r));
                        enemyCount++;
                }
                else if(type == "win") {
                        int w = 0, h = 0;
                if (element->QueryIntAttribute("width", &w) == XML_NO_ATTRIBUTE) return
false;
                if (element->QueryIntAttribute("height", &h) == XML_NO_ATTRIBUTE) return
false;
                        winZone = Rectangle(x, y, w, h);
                }
```

```
                element = element->NextSiblingElement("object");
            }
            return true;
        }
}
```

ДОДАТОК Б

Код класів сімейства Entity

Б.1 Файл Entity.h

```
#pragma once
#include "Drawable.h"
#include "Enums.h"

namespace myGame {
	class Entity : public Drawable {
	protected:
		float x, y;
		float speedY;
		int health, damage;
		Direction lookDir;
		Direction hurtDir;
		//states
		map<States, State*> states;
		States currentState;
		//collides
		vector<Collide*>& walls;

		Rectangle hitBox;
		Rectangle attackBox;

	public:
		static const int GRAVITY = 600;
		static const int MAXSPEEDY = 200;
		static const int MAXSPEEDX = 200;
		Entity(int, int, int, int, EntityData*, vector<Collide*>&, SDL_Renderer*);
		virtual ~Entity();

		float getX() const;
		float getY() const;
		float getSpeedY() const;
		void setSpeedY(float);
		Direction getLookDir();
		Direction getHurtDir();
		bool isDead();
		void setState(States);
		virtual void move(Direction, Uint32);
		void jump();
		void takeDamage(int, Direction);
		virtual void attack() = 0;
		Rectangle& getHitBox();
		bool onGround();
		void collissionUpdate();

		virtual SDL_Rect getRect() override;
		virtual SDL_Point getFocus() override;
		virtual void render(Camera& camera, SDL_Renderer* renderer) override;
	};

	class Player : public Entity {
```

```
		vector<Entity*>& enemies;
	public:
		Player(int x, int y, EntityData* data, vector<Collide*>& ground, vector<Entity*>& enemies,
SDL_Renderer* r);
		virtual ~Player();

		virtual void attack() override;
		virtual void act(Uint32 delay) override;
	};

	class Enemy : public Entity {
		Player* player;
		bool targetSpotted;
	public:
		Enemy(int x, int y, EntityData* data, vector<Collide*>& ground, Player* player,
SDL_Renderer* r);
		virtual ~Enemy();


		virtual void move(Direction dir, Uint32 delay) override;
		virtual void attack() override;
		virtual void act(Uint32 delay) override;
	};
}
```

Б.2 Файл Entity.cpp

```
#include "Entity.h"

namespace myGame {
	Entity::Entity(int x, int y, int health, int damage, EntityData* data, vector<Collide*>& walls,
SDL_Renderer* r) :
		x{ (float)x }, y{ (float)y }, health{ health }, damage{ damage }, walls{ walls }, currentState{
States::Idle },
		hitBox{ x - data->hitW / 2, y - data->hitH,data->hitW,data->hitH }, attackBox{ x, y -
hitBox.height / 2 - attackBox.height / 2,data->attackW,data->attackH },
		lookDir{ Direction::Right }, hurtDir{ Direction::Left }, speedY{ 0 }
	{
		int w = data->width;
		int h = data->height;
	}
	Entity::~Entity()
	{

	}

	float Entity::getX() const {
		return x;
	}
	float Entity::getY() const {
		return y;
	}
	float Entity::getSpeedY() const {
		return speedY;
	}
	void Entity::setSpeedY(float speed)
	{
```

```
		speedY = speed;
	}
	Direction Entity::getHurtDir()
	{
		return hurtDir;
	}
	bool Entity::isDead()
	{
		return health <= 0;
	}
	Direction Entity::getLookDir() {
		return lookDir;
	}

	void Entity::setState(States state) {
		if (currentState == state) return;
		currentState = state;
		auto result = states.find(state);
		if (result != states.end()) {
			result->second->reset();
		}
	}
	void Entity::move(Direction dir, Uint32 delay)
	{
		int dop = MAXSPEEDX * delay / 1000;
		switch (dir)
		{
		case myGame::Direction::Right:
			x += dop;
			break;
		case myGame::Direction::Left:
			x -= dop;
			break;
		default:
			break;
		}
	}
	void Entity::jump()
	{
		setState(States::Jump);
		speedY = -(float)MAXSPEEDY;
	}
	void Entity::takeDamage(int damage, Direction dir)
	{
		if (health > 0) {
			health -= damage;
		}
		if (health > 0) {
			setState(States::Hurt);
			hurtDir = dir;
		}
		else {
			setState(States::Death);
		}
	}
	Rectangle& Entity::getHitBox() {
		hitBox.setPos(x - hitBox.width / 2, y - hitBox.height);
		return hitBox;
	}
	bool Entity::onGround()
	{
		for (Collide* wall : walls) {
			if (wall->checkCollision(x - hitBox.width / 2 + 1, y + 1) || wall->checkCollision(x +
```

```
hitBox.width / 2 - 1, y + 1))
					return true;
			}
			return false;
		}
		void Entity::collissionUpdate()
		{
			for (Collide* wall : walls) {
				if (CollisionInfo info = wall->checkCollision(getHitBox())) {
int _x = std::abs(info.x1) < std::abs(info.x2) ? info.x1 : info.x2;
int _y = std::abs(info.y1) < std::abs(info.y2) ? info.y1 : info.y2;

				if (std::abs(_x) < std::abs(_y)) {
						x += _x;
					}
					else {
					y += _y;
					if (_y > 0) y++;
					speedY = 0;
					}
				}
			}
		}
		SDL_Rect Entity::getRect()
		{
			return getHitBox().getRect();
		}
		SDL_Point Entity::getFocus()
		{
			return { (int)x,(int)y - hitBox.height / 2 };
		}

		Player::Player(int x, int y, EntityData* data, vector<Collide*>& ground, vector<Entity*>& enemies,
SDL_Renderer* r)
			: Entity(x, y, 100, 50, data, ground, r), enemies{ enemies } {
			int w = data->width;
			int h = data->height;
			states[States::Attack] = new AttackState(this, w, h, &data->attack, data->type, r);
			states[States::Death] = new DeathState(this, w, h, &data->death, data->type, r);
			states[States::Hurt] = new HurtState(this, w, h, &data->hurt, data->type, r);
			states[States::Idle] = new IdleState(this, w, h, &data->idle, data->type, r);
			states[States::Jump] = new JumpState(this, w, h, &data->jump, data->type, r);
			states[States::Run] = new RunState(this, w, h, &data->run, data->type, r);
		}

		Player::~Player() {
			delete states[States::Attack];
			delete states[States::Death];
			delete states[States::Hurt];
			delete states[States::Idle];
			delete states[States::Jump];
			delete states[States::Run];
		}

		void Player::attack()
		{
			if (lookDir == Direction::Left)
attackBox.setPos(x - attackBox.width, y - hitBox.height / 2 - attackBox.height / 2);
			else
attackBox.setPos(x, y - hitBox.height / 2 - attackBox.height / 2);
for (Entity* enemy : enemies) {
		if (attackBox.checkCollision(enemy->getHitBox())) {
			enemy->takeDamage(damage, lookDir);
```

```
                }
            }
        }

        void Player::act(Uint32 delay) {
            float oldX = x;
            states[currentState]->act(delay);
            if (oldX < x) {
                lookDir = Direction::Right;
            }
            else if (oldX > x) {
                lookDir = Direction::Left;
            }

            y += speedY * delay / 1000;

            collissionUpdate();

            if (!onGround()) {
        speedY += (float)(delay * GRAVITY) / 1000;

        speedY = std::min((float)MAXSPEEDY, speedY);
                if(currentState != States::Hurt)
                    setState(States::Jump);
            }
            else {
                speedY = 0;
            }
            if (isDead()) setState(States::Death);
        }
void Entity::render(Camera& camera, SDL_Renderer* renderer) {
        states[currentState]->render(camera, renderer);
#ifdef DEBUG
            SDL_Rect pos = { x,y,1,1 };
            if (SDL_HasIntersection(&pos, &camera.rect) == SDL_TRUE) {
                pos.x -= camera.rect.x;
                pos.y -= camera.rect.y;
                SDL_SetRenderDrawColor(renderer, 0x00, 0x00, 0xFF, 0xFF);
                SDL_RenderDrawLine(renderer, pos.x, pos.y, pos.x + 10, pos.y);
                SDL_SetRenderDrawColor(renderer, 0xFF, 0x00, 0x00, 0xFF);
                SDL_RenderDrawLine(renderer, pos.x, pos.y, pos.x, pos.y + 10);
            }
            hitBox.drawBorder(renderer, camera);
            attackBox.drawBorder(renderer, camera);
#endif // DEBUG
        }
        Enemy::Enemy(int x, int y, EntityData* data, vector<Collide*>& ground, Player* player,
SDL_Renderer* r)
            : Entity(x, y, 100, 25, data, ground, r), player{ player }, targetSpotted{ false }
        {
            int w = data->width;
            int h = data->height;
            states[States::Attack] = new EnemyAttackState(this, w, h, &data->attack, data->type, r);
            states[States::Death] = new EnemyDeathState(this, w, h, &data->death, data->type, r);
            states[States::Hurt] = new EnemyHurtState(this, w, h, &data->hurt, data->type, r);
            states[States::Idle] = new EnemyIdleState(this, w, h, &data->idle, data->type, r);
            states[States::Jump] = new EnemyJumpState(this, w, h, &data->jump, data->type, r);
            states[States::Run] = new EnemyRunState(this, w, h, &data->run, data->type, r);
        }
        Enemy::~Enemy() {
            delete states[States::Attack];
            delete states[States::Death];
            delete states[States::Hurt];
```

```
			delete states[States::Idle];
			delete states[States::Jump];
			delete states[States::Run];
		}

		void Enemy::move(Direction dir, Uint32 delay)
		{
float dop = (float)MAXSPEEDX * (float)delay * 0.0005;
			switch (dir)
			{
			case myGame::Direction::Right:
				x += dop;
				break;
			case myGame::Direction::Left:
				x -= dop;
				break;
			default:
				break;
			}
		}

		void Enemy::attack()
		{
			if (lookDir == Direction::Left)
				attackBox.setPos(x - attackBox.width, y - hitBox.height / 2 - attackBox.height / 2);
			else
attackBox.setPos(x, y - hitBox.height / 2 - attackBox.height / 2);
		if (attackBox.checkCollision(player->getHitBox())) {
			player->takeDamage(damage, lookDir);
			}

		}
		void Enemy::act(Uint32 delay) {
			if (isDead()) {
				setState(States::Death);
			}
			else {
		int distanceToPlayerX = player->getX() - getX();
			if (abs(distanceToPlayerX) < 100) {
					targetSpotted = true;
		if (distanceToPlayerX < 0) lookDir = Direction::Left;
else if (distanceToPlayerX > 0) lookDir = Direction::Right;
				}
				else {
					targetSpotted = false;
				}

				if (currentState == States::Hurt || currentState == States::Death || currentState ==
States::Attack) {
					targetSpotted = false;
				}
				else {
			bool onGroundTemp = onGround();
					if (onGroundTemp) {
						//to idle:
			if (!targetSpotted || player->isDead()) {
							setState(States::Idle);
						}
						else {
						//to attack:
							if (abs(distanceToPlayerX) < attackBox.getRect().w) {
								setState(States::Attack);
							}
```

```
                        //to run:
                    else setState(States::Run);
                        }
                    }
                    //to jump:
                    else {
                        setState(States::Jump);
                        if (targetSpotted) {
                            move(lookDir, delay);
                        }
                    }
                }
            }
            states[currentState]->act(delay);
            if (!onGround()) {
    speedY += (float)(delay * GRAVITY) / 1000;
    speedY = std::min((float)MAXSPEEDY, speedY);
            }
            y += speedY * delay / 1000;
            collissionUpdate();
    }
}
```

ДОДАТОК В

Код класів для перевірки зіткнень

В.1 Файл Collide.cpp

```
#include "Collide.h"
#include "Entity.h"

namespace myGame {

	CollisionInfo RightTriangle::checkCollision(const Rectangle& box) {
		if (box.y + box.height < y) return { false };
		if (box.x > x + SIDE) return { false };

		int x1 = x - box.x - box.width;
		if (x1 > 0) return { false };

		int y2 = y + SIDE - box.y;
		if (y2 < 0) return{ false };

		int Xdif = std::max(x1+SIDE, 0);
		int topY = y + Xdif;
		int y1 = topY - box.y - box.height;
		if (y1 > 0) return { false };
		int x2 = box.x > x ? box.x + box.width : y1;

		return{ true, x1,x2,y1,y2 };
	}
	CollisionInfo RightTriangle::checkCollision(const int _x, const int _y)
	{
		if (_y < y) return { false };
		if (_x > x + SIDE) return { false };

		int x1 = x - _x;
		if (x1 > 0) return { false };

		int y2 = y + SIDE - _y;
		if (y2 < 0) return{ false };

		int Xdif = std::max(x1 + SIDE, 0);
		int topY = y + Xdif;
		int y1 = topY - _y;
		if (y1 > 0) return { false };
		int x2 = _x > x ? _x : y1;

		return{ true, x1,x2,y1,y2 };
	}
	CollisionInfo LeftTriangle::checkCollision(const Rectangle& box) {
		if (box.y + box.height < y) return { false };
		if (box.x > x + SIDE) return { false };

		int x1 = x - box.x - box.width;
		if (x1 > 0) return { false };

		int y2 = y + SIDE - box.y;
```

```
		if (y2 < 0) return{ false };
		int Xdif = std::max(box.x - x, 0);
		int topY = y + Xdif;
		int y1 = topY - box.y - box.height;
		if (y1 > 0) return { false };
		int x2 = box.x < x ? x + SIDE - box.x : -y1;

		return{ true, x1,x2,y1,y2 };
	}
	CollisionInfo LeftTriangle::checkCollision(const int _x, const int _y)
	{
		if (_y < y) return { false };
		if (_x > x + SIDE) return { false };

		int x1 = x - _x;
		if (x1 > 0) return { false };

		int y2 = y + SIDE - _y;
		if (y2 < 0) return{ false };
		int Xdif = std::max(_x - x, 0);
		int topY = y + Xdif;
		int y1 = topY - _y;
		if (y1 > 0) return { false };
		int x2 = _x < x ? x + SIDE - _x : -y1;

		return{ true, x1,x2,y1,y2 };
	}
	CollisionInfo Rectangle::checkCollision(const Rectangle& box) {
		int x1 = box.x + box.width - x;
		int x2 = box.x - x - width;
		if (x1 < 0 || x2 > 0) {
			return { false };
		}

		int y1 = box.y + box.height - y;
		int y2 = box.y - y - height;
		if (y1 < 0 || y2 > 0) {
			return { false };
		}

		if (y1 != 0) {
			y1 = y1;
		}

		return { true, -x1,-x2,-y1,-y2 };
	}

	CollisionInfo Rectangle::checkCollision(const int _x, const int _y)
	{
		int x1 = _x - x;
		int x2 = _x - x - width;
		if (x1 < 0 || x2 > 0) {
			return { false };
		}

		int y1 = _y - y;
		int y2 = _y - y - height;
		if (y1 < 0 || y2 > 0) {
			return { false };
		}

		return { true, -x1,-x2,-y1,-y2 };
	}
```

```
	void RightTriangle::drawBorder(SDL_Renderer* renderer, Camera& camera)
	{
		SDL_Rect pos{ x,y,SIDE,SIDE };
		if (SDL_HasIntersection(&pos, &camera.rect) == SDL_TRUE) {
			pos.x -= camera.rect.x;
			pos.y -= camera.rect.y;
			SDL_SetRenderDrawColor(renderer, 0x00, 0x00, 0xFF, 0xFF);
			SDL_RenderDrawRect(renderer, &pos);
			SDL_SetRenderDrawColor(renderer, 0xFF, 0x00, 0x00, 0xFF);
			SDL_RenderDrawLine(renderer, pos.x, pos.y + SIDE, pos.x + SIDE, pos.y + SIDE);
			SDL_RenderDrawLine(renderer, pos.x + SIDE, pos.y, pos.x + SIDE, pos.y + SIDE);
			SDL_RenderDrawLine(renderer, pos.x + SIDE, pos.y, pos.x, pos.y + SIDE);
		}
	}
	void LeftTriangle::drawBorder(SDL_Renderer* renderer, Camera& camera)
	{
		SDL_Rect pos{ x,y,SIDE,SIDE };
		if (SDL_HasIntersection(&pos, &camera.rect) == SDL_TRUE) {
			pos.x -= camera.rect.x;
			pos.y -= camera.rect.y;
			SDL_SetRenderDrawColor(renderer, 0x00, 0x00, 0xFF, 0xFF);
			SDL_RenderDrawRect(renderer, &pos);
			SDL_SetRenderDrawColor(renderer, 0xFF, 0x00, 0x00, 0xFF);
			SDL_RenderDrawLine(renderer, pos.x, pos.y + SIDE, pos.x + SIDE, pos.y + SIDE);
			SDL_RenderDrawLine(renderer, pos.x, pos.y, pos.x, pos.y + SIDE);
			SDL_RenderDrawLine(renderer, pos.x, pos.y, pos.x + SIDE, pos.y + SIDE);
		}
	}
	void Rectangle::drawBorder(SDL_Renderer* renderer, Camera& camera)
	{
		SDL_Rect pos{ x,y,width,height };
		if (SDL_HasIntersection(&pos, &camera.rect) == SDL_TRUE) {
			pos.x -= camera.rect.x;
			pos.y -= camera.rect.y;
			SDL_SetRenderDrawColor(renderer, 0xFF, 0x00, 0x00, 0xFF);
			SDL_RenderDrawRect(renderer, &pos);
		}
	}

	CollideType RightTriangle::getType()
	{
		return CollideType::RightTriangle;
	}
	CollideType LeftTriangle::getType()
	{
		return CollideType::LeftTriangle;
	}
	CollideType Rectangle::getType()
	{
		return CollideType::Rectangle;
	}

	void Collide::setPos(int x, int y)
	{
		this->x = x;
		this->y = y;
	}
}
```

ДОДАТОК Г

Код класів сімейства State

Г.1 Файл State.h

```
#pragma once
#include "Data.h"
#include "Enums.h"
#include "Tools.h"
#include <map>
#include <SDL.h>
#include <vector>
using std::map, std::vector;

namespace myGame {

	class Entity;

	class State {
	protected:
		int w, h;
		Entity* entity;
		vector<SDL_Texture*> frames;
		int currentFrame;
		Uint32 timer;
		SDL_RendererFlip flip;
	public:
		State(Entity* e, int _w, int _h, StateData* data, string type, SDL_Renderer* r);
		virtual void act(Uint32 delay);
		virtual void reset();
		void render(Camera& camera, SDL_Renderer* renderer);

		virtual ~State();
	};
	//Player States
	class IdleState : public State {
	public:
		IdleState(Entity* entity, int _w, int _h, StateData* data, string type, SDL_Renderer* renderer)
			: State(entity, _w, _h, data, type, renderer) {}
		virtual void act(Uint32 delay) override;
		virtual ~IdleState() {}
	};
	…
	//Enemy States
	class EnemyIdleState : public State {
	public:
		EnemyIdleState(Entity* entity, int _w, int _h, StateData* data, string type, SDL_Renderer*
renderer)
			: State(entity, _w, _h, data, type, renderer) {}
		virtual void act(Uint32 delay) override;
		virtual ~EnemyIdleState() {}
	};
	…
}
```

Г.2 Файл State.cpp

```
#include "Drawable.h"
#include "Entity.h"
#include "Game.h"
#include "State.h"

namespace myGame {
	//State

	State::State(Entity* e, int _w, int _h, StateData* data, string type, SDL_Renderer* r) :
		entity{ e }, frames{}, currentFrame{ 0 }, timer{ 0 }, flip{ SDL_FLIP_NONE }, w{ _w }, h{ _h
}
	{
		for (int i = 0; i < data->count; i++) {
			string name = i < 10 ? "0" : "";
			name += std::to_string(i) + "." + type;
			SDL_Texture* buf = loadTexture(r, data->place + name);
			if (buf == NULL) {
				printf("Error state loading!\n");
				getError();
			}
			else frames.push_back(buf);
		}
	}
	void State::act(Uint32 delay)
	{
		timer += delay;
		if (timer >= 100) {
			currentFrame++;
			if (currentFrame == frames.size())
				currentFrame = 0;
			timer -= 100;
		}
	}
	void State::reset()
	{
		currentFrame = 0;
		timer = 0;
	}
	void State::render(Camera& camera, SDL_Renderer* renderer)
	{
		currentFrame = min(currentFrame, frames.size() - 1);
		SDL_Rect pos = { entity->getX() - w / 2, entity->getY() - h, w, h };
		if (SDL_HasIntersection(&pos, &camera.rect) == SDL_TRUE) {
			pos.x -= camera.rect.x;
			pos.y -= camera.rect.y;
			if (entity->getLookDir() == Direction::Left) flip = SDL_FLIP_HORIZONTAL;
			else if (entity->getLookDir() == Direction::Right) flip = SDL_FLIP_NONE;
			SDL_RenderCopyEx(renderer, frames[currentFrame], NULL, &pos, 0.f, NULL, flip);

#ifdef DEBUG
			SDL_SetRenderDrawColor(renderer, 0xFF, 0x00, 0x00, 0xFF);
			SDL_RenderDrawRect(renderer, &pos);
#endif // DEBUG
		}

	}
	State::~State()
	{
		for (SDL_Texture* t : frames) {
```

```
				SDL_DestroyTexture(t);
			}
	}
	//IdleState
	void IdleState::act(Uint32 delay)
	{
			State::act(delay);
			const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);

			if (currentKeyStates[SDL_SCANCODE_A])
			{
				entity->move(Direction::Left, delay);
				entity->setState(States::Run);
			}
			else if (currentKeyStates[SDL_SCANCODE_D])
			{
				entity->move(Direction::Right, delay);
				entity->setState(States::Run);
			}
			if (!entity->onGround()) {
				entity->setState(States::Jump);
			}
			else if (currentKeyStates[SDL_SCANCODE_SPACE])
			{
				entity->jump();
			}
			else if (SDL_GetMouseState(NULL, NULL) & 1) {
				entity->setState(States::Attack);
			}

	}
	//RunState
	void RunState::act(Uint32 delay)
	{
			State::act(delay);
			const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);

			if (!entity->onGround()) {
				entity->setState(States::Jump);
			}
			if (currentKeyStates[SDL_SCANCODE_SPACE])
			{
				entity->jump();
			}
			if (SDL_GetMouseState(NULL, NULL) & 1) {
				entity->setState(States::Attack);
			}
			else if (currentKeyStates[SDL_SCANCODE_A])
			{
				entity->move(Direction::Left, delay);
			}
			else if (currentKeyStates[SDL_SCANCODE_D])
			{
				entity->move(Direction::Right, delay);
			}
			else {
				entity->setState(States::Idle);
			}
	}
	//JumpState
	void JumpState::act(Uint32 delay)
	{
			const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);
```

```
		if (currentKeyStates[SDL_SCANCODE_A])
		{
			entity->move(Direction::Left, delay);
		}
		if (currentKeyStates[SDL_SCANCODE_D])
		{
			entity->move(Direction::Right, delay);
		}
		int speedY = entity->getSpeedY();
		if (speedY < 0)				currentFrame = 0;
		else if (speedY > 0)		currentFrame = 2;
		else						currentFrame = 1;
		if (entity->onGround()) {
			entity->setState(States::Idle);
		}
	}
	//AttackState
	void AttackState::reset()
	{
		State::reset();
		ended = false;
		cooldown = false;
	}
	void AttackState::act(Uint32 delay)
	{
		if (ended) {

			const Uint8* currentKeyStates = SDL_GetKeyboardState(NULL);

			if (currentKeyStates[SDL_SCANCODE_A])
			{
				entity->move(Direction::Left, delay);
				entity->setState(States::Run);
			}
			else if (currentKeyStates[SDL_SCANCODE_D])
			{
				entity->move(Direction::Right, delay);
				entity->setState(States::Run);
			}
			else if (currentKeyStates[SDL_SCANCODE_SPACE])
			{
				entity->jump();
			}
			else if (SDL_GetMouseState(NULL, NULL) & 1) {
				reset();
			}
			else
			{
				entity->setState(States::Idle);
			}
		}
		else {
			timer += delay;
			if (timer >= 100) {
				currentFrame++;
				if (currentFrame == frames.size())
				{
					currentFrame = 0;
					ended = true;
				}
				timer -= 100;
			}
			if (currentFrame == 1 && !cooldown) {
```

```
                        entity->attack();
                        cooldown = true;
                    }
                }
            }

            void HurtState::reset()
            {
                hurtTimer = 0;
                State::reset();
            }

            void HurtState::act(Uint32 delay)
            {
                if (hurtTimer == 0) {
                    entity->setSpeedY(-25);
                }
                hurtTimer += delay;
                if (hurtTimer > 500) {
                    entity->setState(States::Idle);
                }
                State::act(delay);
                entity->move(entity->getHurtDir(), delay);
            }

            void DeathState::act(Uint32 delay)
            {
                if (currentFrame != frames.size() - 1)
                    State::act(delay);
            }

            //Enemy States
            void EnemyIdleState::act(Uint32 delay)
            {
                State::act(delay);

            }
            void EnemyRunState::act(Uint32 delay)
            {
                State::act(delay);
                entity->move(entity->getLookDir(), delay);
            }
            void EnemyJumpState::act(Uint32 delay)
            {
                //if (entity->onGround()) {
                //      entity->setState(States::Idle);
                //}
            }
            void EnemyAttackState::reset()
            {
                State::reset();
                ended = false;
                cooldown = false;
            }
            void EnemyAttackState::act(Uint32 delay)
            {
                if (ended) {
                    entity->setState(States::Idle);
                }
                else {
                    timer += delay;
                    if (timer >= 100) {
                        currentFrame++;
```

```
					if (currentFrame == frames.size())
					{
						currentFrame = 0;
						ended = true;
					}
					timer -= 100;
				}
				if (currentFrame == 4 && !cooldown) {
					entity->attack();
					cooldown = true;
				}
			}
		}
		void EnemyDeathState::act(Uint32 delay)
		{
			if (currentFrame != frames.size() - 1)
				State::act(delay);
		}
		void EnemyHurtState::reset()
		{
			hurtTimer = 0;
			State::reset();
		}
		void EnemyHurtState::act(Uint32 delay)
		{
			hurtTimer += delay;
			if (hurtTimer > 1250 + rand()%250) {
				entity->setState(States::Idle);
			}
			State::act(delay);
		}
}
```

ДОДАТОК Д

Код файлу менеджеру даних

Д.1 Код файлу Data.h

```
#pragma once
#include "Enums.h"
#include "tinyxml2.h"
#include <iostream>
#include <string>

using tinyxml2::XMLElement, tinyxml2::XMLDocument, tinyxml2::XML_SUCCESS;
using std::string;

namespace myGame {
	struct TilesData {
		string type, place;
		int count, width, height;
		bool set(XMLElement* e) {
			type = e->Attribute("type");
			if (type.empty()) return false;
			place = e->Attribute("place");
			if (place.empty()) return false;
			if (e->QueryIntAttribute("count", &count) != XML_SUCCESS) return false;
			if (e->QueryIntAttribute("width", &width) != XML_SUCCESS) return false;
			if (e->QueryIntAttribute("height", &height) != XML_SUCCESS) return false;
			return true;
		}
	};
	struct StateData {
		int count;
		string place;
		bool set(XMLElement* e) {
			place = e->Attribute("place");
			if (place.empty()) return false;
			if (e->QueryIntAttribute("count", &count) != XML_SUCCESS) return false;
			return true;
		}
	};
	struct EntityData {
		string type;
		int width, height;
		int hitW, hitH;
		int attackW, attackH;
		StateData attack;
		StateData death;
		StateData hurt;
		StateData idle;
		StateData jump;
		StateData run;
		bool set(XMLElement* e) {
			type = e->Attribute("type");
			if (type.empty()) return false;
			if (e->QueryIntAttribute("width", &width) != XML_SUCCESS) return false;
			if (e->QueryIntAttribute("height", &height) != XML_SUCCESS) return false;
```

```
			if (e->QueryIntAttribute("hitW", &hitW) != XML_SUCCESS) return false;
			if (e->QueryIntAttribute("hitH", &hitH) != XML_SUCCESS) return false;
			if (e->QueryIntAttribute("attackW", &attackW) != XML_SUCCESS) return false;
			if (e->QueryIntAttribute("attackH", &attackH) != XML_SUCCESS) return false;
			XMLElement* e2;
			e2 = e->FirstChildElement("attack");
			if (e2 == nullptr) return false;
			if (!attack.set(e2)) return false;
			e2 = e->FirstChildElement("death");
			if (e2 == nullptr) return false;
			if (!death.set(e2)) return false;
			e2 = e->FirstChildElement("hurt");
			if (e2 == nullptr) return false;
			if (!hurt.set(e2)) return false;
			e2 = e->FirstChildElement("idle");
			if (e2 == nullptr) return false;
			if (!idle.set(e2)) return false;
			e2 = e->FirstChildElement("jump");
			if (e2 == nullptr) return false;
			if (!jump.set(e2)) return false;
			e2 = e->FirstChildElement("run");
			if (e2 == nullptr) return false;
			if (!run.set(e2)) return false;
			return true;

		}
	};
	struct MenuData {
		int width, height;
		string playButPath;
		string exitButPath;
		bool set(XMLElement* e) {
			if (e->QueryIntAttribute("width", &width) != XML_SUCCESS) return false;
			if (e->QueryIntAttribute("height", &height) != XML_SUCCESS) return false;
			XMLElement* e2;
			e2 = e->FirstChildElement("play");
			if (e2 == nullptr) return false;
			playButPath = e2->Attribute("path");
			if (playButPath.empty()) return false;

			e2 = e->FirstChildElement("exit");
			if (e2 == nullptr) return false;
			exitButPath = e2->Attribute("path");
			if (exitButPath.empty()) return false;
			return true;
		}

	};
	struct MapsData {
		string type, place;
		int count;
		bool set(XMLElement* e) {
			type = e->Attribute("type");
			if (type.empty()) return false;
			place = e->Attribute("place");
			if (place.empty()) return false;
			if (e->QueryIntAttribute("count", &count) != XML_SUCCESS) return false;
			return true;
		}
	};
	struct Data {
		TilesData tiles;
		EntityData player;
```

```
            EntityData bandit[2];
            MenuData menu;
            MapsData maps;
            string backgroundPath;
            bool set(XMLDocument& d) {
                XMLElement* data = d.FirstChildElement("data");
                XMLElement* e;
                e = data->FirstChildElement("tiles");
                tiles.set(e);
                e = data->FirstChildElement("player");
                player.set(e);
                e = data->FirstChildElement("bandit1");
                bandit[0].set(e);
                e = data->FirstChildElement("bandit2");
                bandit[1].set(e);
                e = data->FirstChildElement("menu");
                menu.set(e);
                e = data->FirstChildElement("maps");
                maps.set(e);
                e = data->FirstChildElement("background");
                backgroundPath = e->Attribute("path");
                return true;
            }
        };
```