МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет	Інформаційних технологій
Кафедра	Інформатики і прикладного програмного забезпечення
Спеціальність	Інженерія програмного забезпечення
Форма навчання	Денна

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Білий Ярослав Олександрович
(прізвище, ім'я, по батькові здобувача)

на тему «Розробка гри «The binding of isaac»»
(повна назва теми)

за матеріалами праць провідних спеціалістів з розробки ПЗ та проектування БД

(повна назва бази дослідження)

науковий керівник к.т.н., доц. _____ Хоцкіна В.Б.
(наук. ступінь, вчене звання) *(підпис)* *(прізвище, ініціали)*

Робота допущена до захисту в ЕК
Протокол засідання кафедри
від 11.06.2025 р. № 12
Завідувач кафедри _____
(підпис)
д.т.н., професор Зеленський О.С.
Наук. ступень, вчене звання *Ініціали, прізвище*

Кривий Ріг – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет	Інформаційних технологій
Кафедра	Інформатики і прикладного програмного забезпечення
Спеціальність	Інженерія програмного забезпечення
Форма навчання	Денна

«ЗАТВЕРДЖУЮ»
Завідувач кафедри _____ Зеленський О.С.
(підпис) (Прізвище, ініціали)
« 11 » червня 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ

1. Тема роботи «Розробка гри «The binding of isaac»»

Керівник роботи к.т.н., доц , Хоцкіна В.Б.
затверджені наказом закладу вищої освіти від «04» квітня 2025 р. № 222-ст

2. Строк подання здобувачем роботи до «09» червня 2025 р.

3. Зміст кваліфікаційної роботи, об'єкт, предмет та мета дослідження:

Розділ 1. Постановка задачі

Розділ 2. Розробка алгоритму розв'язання задачі

Розділ 3. Організація інформаційного забезпечення

Розділ 4. Розробка програмного забезпечення

Об'єкт дослідження: відео ігри

Предмет дослідження: гра «The binding of isaac»

Мета кваліфікаційної роботи: *розробка гри «The binding of issac»*

5. Дата видачі завдання «04» квітня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів МДР	Строк виконання етапів роботи	Відмітка керівника про виконання етапів (дата, підпис)
1	Підготовка розділу 1	04.04.2025-13.04.2025	
2	Підготовка розділу 2	14.04.2025-26.04.2025	
3.	Підготовка розділу 3	27.04.2025-15.05.2025	
4	Підготовка розділу 4	16.05.2025-08.06.2025	
5	Реєстрація завершеної кваліфікаційної роботи	09.06.2025	Реєстраційний №___ «09»червня 2025 р.
6	Отримання відгуку від наукового керівника	10.06.2025	
7	Подання кваліфікаційної роботи на перегляд завідувачу кафедри	11.06.2025	
8	Отримання зовнішньої рецензії	12.06.2025	
9	Попередній захист кваліфікаційної роботи на кафедрі	13.06.2025	
10	Підготовка до захисту в ЕК	16.06.2025-21.06.2025	

Завдання підготував науковий керівник _____ Хоцкіна В.Б.
(підпис) (прізвище та ініціали)

Завдання одержав _____ Білий Я.О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

на кваліфікаційну бакалаврську роботу

«Розробка гри «The binding of isaac»»

Білого Ярослава Олександровича

Кваліфікаційна бакалаврська робота на здобуття освітньо-кваліфікаційного рівня бакалавра зі спеціальності 121 «Інженерія програмного забезпечення» – Державний університет економіки і технологій – Кривий Ріг, 2025.

У бакалаврській роботі розроблено програмне забезпечення для створення та редагування ігрових мап із використанням React для фронтенду, Node.js та MongoDB для збереження даних, а також C# для ігрового клієнта. Взаємодія між клієнтом і сервером реалізована через REST API, що забезпечує ефективне завантаження і збереження структур рівнів у форматі JSON. Ігровий клієнт на C# перетворює отримані дані у внутрішні об’єкти гри, підтримуючи логіку розміщення кімнат, ворогів і босів. Використання MongoDB дозволяє гнучко зберігати багаторівневі структури ігрових карт.

Ключові слова: ІГРОВИЙ РЕДАКТОР, MONGODB, C#, РЕДАГУВАННЯ РІВНІВ, ГРА, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД(база даних)	Впорядкований набір логічно взаємопов’язаних даних, що використовуються спільно та призначені для задоволення інформаційних потреб користувачів.
ПЗ	Програмне забезпечення.
TBOI	The binding of Isaac

ЗМІСТ

ВСТУП

У світі сучасних цифрових технологій комп’ютерні ігри займають одне з провідних місць, і це зовсім не випадково, адже вони давно перестали бути просто способом розважитися або убити час, перетворившись на справжнє явище, яке впливає на культуру, освіту, соціальні взаємини та навіть економіку. Щороку індустрія ігор демонструє стрімке зростання, випереджаючи за обсягами прибутків такі традиційні сфери розваг, як кіно та музика, що лише підтверджує їхню неймовірну популярність та значення у сучасному суспільстві.

Однак за всією цією масовістю та комерційним успіхом ховається складний і багатогранний процес створення ігор, який включає в себе не лише програмування, а й цілу низку інших важливих аспектів, таких як дизайн, анімація, написання сценаріїв, звукове оформлення та, звичайно ж, проектування ігрових рівнів, які є основою будь-якої гри, визначаючи її атмосферу, геймплей та загальне враження гравця.

Сьогодні розробка ігрових рівнів, або, як їх ще називають, карт, є окремим напрямком у створенні ігор, який вимагає не лише творчого підходу, а й глибоких технічних знань, адже сучасні ігрові простори стають дедалі складнішими, деталізованішими та інтерактивнішими, що, у свою чергу, висуває нові вимоги до інструментів, за допомогою яких ці рівні створюються.

Традиційно для створення ігрових карт використовувалися спеціалізовані програми, інтегровані в ігрові рушії, такі як Unreal Engine або Unity, проте з розвитком веб-технологій ситуація почала змінюватися, відкриваючи нові можливості для розробників. Веб-технології, які раніше асоціювалися виключно зі створенням сайтів, тепер активно використовуються в розробці ігор, дозволяючи створювати потужні та зручні інструменти для роботи з ігровим контентом, які можуть працювати прямо в браузері, без необхідності встановлювати додаткове програмне забезпечення.

Саме тому виникла ідея розробити спеціалізований веб-редактор для створення ігрових карт, який би поєднував у собі зручність сучасних веб-

інтерфейсів із потужністю професійних інструментів для розробки ігор. Такий редактор міг би стати справжнім порятунком для багатьох розробників, особливо для тих, хто тільки починає свій шлях у створенні ігор, адже він дозволив би значно спростити процес проектування рівнів, зробивши його більш доступним та інтуїтивно зрозумілим.

Для реалізації цієї ідеї було вирішено використати сучасні технології веб-розробки, такі як React.js для створення динамічного та інтерактивного інтерфейсу, що дозволить користувачам комфортно працювати з картою, додаючи, видаляючи та редагуючи різні елементи без необхідності заглиблюватися у складні технічні деталі. Використання React.js є особливо доречним у цьому випадку, оскільки ця бібліотека дозволяє створювати дуже гнучкі та швидкі інтерфейси, які можуть реагувати на дії користувача в режимі реального часу, що є критично важливим для комфортної роботи з ігровими рівнями. Крім того, React.js має велику спільноту розробників і багато готових рішень, що значно прискорює процес розробки та дозволяє легко знаходити відповіді на будь-які питання, які можуть виникнути під час роботи над проектом.

Для зберігання даних про створені карти було обрано MongoDB – сучасну нереляційну базу даних, яка ідеально підходить для роботи зі складними ієрархічними структурами, такими як ігрові рівні. На відміну від традиційних реляційних баз даних, MongoDB дозволяє зберігати дані у форматі, близькому до JSON, що дуже зручно для роботи з веб-додатками, а також забезпечує високу швидкість запитів навіть при роботі з великими обсягами інформації. Крім того, MongoDB легко масштабується, що дозволить у майбутньому розширювати функціонал редактора без необхідності повної переробки архітектури бази даних.

Окремо варто зупинитися на процесі візуалізації створених карт у самій грі. Для цього було вирішено використати мову програмування C# у поєднанні з бібліотекою OpenGL, що дозволить забезпечити високу продуктивність та якісний рендеринг ігрового світу. C# є однією з найпопулярніших мов для

розробки ігор, завдяки своїй простоті, гнучкості та потужним можливостям, а OpenGL – це перевірена часом технологія, яка дозволяє створювати високоякісну графіку з мінімальними витратами ресурсів.

Важливо зазначити, що вибір саме таких технологій не є випадковим – він ґрунтується на аналізі сучасних тенденцій у розробці ігор та веб-додатків, а також на досвіді багатьох успішних проектів. Поєднання веб-технологій для створення інтерфейсу редактора з потужними інструментами для рендерингу гри дозволяє отримати оптимальний баланс між зручністю використання та технічною могутністю, що є ключовим фактором успіху будь-якого подібного проекту.

Крім того, розробка такого редактора має важливе значення не лише для конкретної гри, а й для розвитку ігрової індустрії в цілому. Він може стати універсальним інструментом, який буде використовуватися в багатьох проектах, спрощуючи процес створення ігрових світів та відкриваючи нові можливості для розробників. У майбутньому такий підхід може стати стандартом у створенні ігор, поєднуючи в собі переваги веб-технологій та традиційних методів розробки.

Не можна не згадати і про те, що подібні інструменти мають великий потенціал у сфері освіти. Вони можуть використовуватися для навчання майбутніх розробників ігор, допомагаючи їм освоїти основи створення ігрових рівнів у доступній та зрозумілій формі. Це особливо актуально в умовах, коли інтерес до розробки ігор постійно зростає, а ринок потребує все більше кваліфікованих фахівців.

Таким чином, розробка веб-редактора ігрових карт є не просто технічним завданням, а цілим комплексом заходів, спрямованих на вдосконалення процесу створення ігор, розширення можливостей розробників та сприяння розвитку ігрової індустрії в цілому. Це інноваційний підхід, який поєднує в собі останні досягнення у сфері веб-технологій та ігрового дизайну, і його реалізація може мати далекосяжні наслідки для всієї галузі.

Тому **метою дослідження** є розробка веб-орієнтованого редактора ігрових карт, який дозволяє створювати, редагувати та зберігати структури рівнів із подальшою інтеграцією їх у гру, реалізовану на мові програмування C# з використанням бібліотеки OpenTK.

Виходячи з мети, **завданнями дослідження** є:

- Провести аналіз сучасних засобів створення ігрових карт.
- Обґрунтувати вибір архітектури та технологій для реалізації веб-редактора та гри.
- Розробити інтерфейс редактора карт за допомогою React.js.
- Реалізувати серверну частину на Node.js з використанням MongoDB для зберігання структури ігрових карт.
- Створити рушій гри на C# із використанням бібліотеки OpenTK.
- Реалізувати логіку відображення карт, переходів між кімнатами, руху персонажа та базової ігрової взаємодії.
- Налагодити обмін даними між редактором карт і рушієм гри через REST API.
- Провести тестування працездатності системи та оцінити перспективи її розширення.

Об’єктом дослідження є процес створення ігрових рівнів.

Предметом дослідження є веб-редактор карт та механізми інтеграції з рушієм гри, створеним на C# з використанням OpenGL (OpenTK).

У майбутньому можна очікувати подальшого вдосконалення подібних інструментів, зокрема за рахунок інтеграції штучного інтелекту для автоматизації окремих процесів створення рівнів, використання хмарних технологій для колаборативної роботи над проектами, а також застосування віртуальної та доповненої реальності для більш глибокого занурення в процес розробки. Все це робить тему створення веб-редакторів для ігрових карт надзвичайно актуальною та перспективною для подальших досліджень і розробок.

Сучасний етап розвитку інтерактивних технологій демонструє яскравий тренд на стирання меж між традиційними платформами для розробки ігор. Веб-редактори карт, такі як описаний вище, стають своєрідним мостом між звичними браузерними технологіями та професійними ігровими рушіями. Це відкриває цілий спектр нових можливостей для творців ігрового контенту.

Важливим аспектом, який варто розглянути детальніше, є питання інтеграції таких редакторів у повноцінний цикл розробки ігор. Наприклад, сучасні системи контролю версій, такі як Git, можуть бути адаптовані для роботи з проектами ігрових карт, що дозволить команді розробників ефективно керувати змінами та відстежувати історію редагування рівнів. Особливо цінною є можливість відкату до попередніх версій карти у разі виникнення проблем, що часто трапляється під час ітеративного процесу розробки.

Не менш цікавим є аспект автоматизованого тестування створених рівнів. Сучасні підходи дозволяють реалізувати систему автоматичних перевірок, яка може виявляти потенційні проблеми на ранніх етапах - наприклад, недосяжні для гравця ділянки карти, конфлікти текстур або проблеми з ігровою логікою. Такий підхід значно підвищує якість кінцевого продукту та зменшує навантаження на команду тестувальників.

Особливу увагу привертає питання оптимізації робочого процесу. Сучасні веб-редактори можуть включати інтелектуальні інструменти підказок, які аналізують дії розробника та пропонують оптимальні рішення для розміщення об'єктів, налаштування освітлення чи створення ігрових механік. Це не лише прискорює процес створення контенту, але й допомагає новачкам швидше освоїти професійні методики роботи.

Перспективним напрямком розвитку подібних систем є реалізація механізмів процедурної генерації, коли редактор може автоматично створювати окремі елементи карти на основі заданих параметрів. Такі функції особливо корисні при створенні масштабних ігрових світів, де ручне проектування кожного елемента займає надто багато часу. При цьому розробник зберігає

повний контроль над процесом, маючи можливість корегувати згенерований контент.

Соціальний аспект таких розробок також заслуговує на увагу. Веб-орієнтовані редактори природним чином сприяють розвитку спільнот творців контенту, де користувачі можуть ділитися своїми роботами, спільно працювати над проектами або навіть створювати бібліотеки готових елементів для повторного використання. Це формує цілу екосистему навколо інструменту, що значно підвищує його цінність і життєздатність.

Технічний прогрес у галузі веб-технологій, особливо з появою WebAssembly і нових можливостей WebGL, відкриває ще більш широкі перспективи для подібних рішень. Вже зараз ми спостерігаємо, як веб-додатки досягають продуктивності, яка раніше була доступна лише для нативних програм, що дозволяє створювати все більш складні і потужні інструменти без необхідності встановлення додаткового програмного забезпечення.

Економічний ефект від впровадження таких систем також важко переоцінити. Зменшення часу на створення контенту, можливість паралельної роботи над різними частинами проекту, зниження вимог до апаратного забезпечення розробників - все це призводить до значного здешевлення процесу створення ігор без втрати якості кінцевого продукту.

Враховуючи всі ці аспекти, можна з упевненістю стверджувати, що розвиток веб-орієнтованих інструментів для створення ігрового контенту є не просто модним трендом, а серйозною технологічною інновацією, яка здатна кардинально змінити ландшафт ігрової індустрії в найближчі роки.

РОЗДІЛ 1
ПОСТАНОВКА ЗАДАЧІ

1.1. Стан сучасної розробки інструментів для створення ігрових рівнів

Сучасна ігрова індустрія постійно зростає як у технічному, так і в творчому плані, а разом із цим - зростає і потреба у гнучких, зручних, багатофункціональних інструментах для створення ігрових рівнів [1, с.45]. Карта гри - це не лише візуальна складова, а й структурна основа геймплею. Вона задає просторову логіку, визначає маршрути гравця, взаємодії з об’єктами, точки спавну ворогів та багато іншого. Саме карта, або ігрове поле, виступає центральною частиною дизайну, яка безпосередньо впливає на динаміку та атмосферу гри [2, с.72].

У наш час розробники мають доступ до великого спектра інструментів, які дозволяють швидко й ефективно створювати рівні: від класичних рівнебудівників до складних редакторів з інтеграцією в ігровий рушій. Більшість комерційних рушіїв, таких як Unity, Unreal Engine чи Godot, мають вбудовані або додаткові інструменти для побудови ігрових карт, зокрема tilemap-редактори, візуальні скриптові системи та системи зонування. Ці інструменти надають широкі можливості, однак часто бувають перевантажені функціоналом або надто загальними, що ускладнює їх застосування для вузькоспеціалізованих задач.

Однак у багатьох інді-розробників або аматорських студій виникає потреба у створенні власних редакторів, які більш точно відповідають конкретним задачам гри. Вони дозволяють не лише зекономити ресурси на ліцензії, а й краще контролювати логіку побудови рівнів, їхню структуру та формат обміну даними з рушієм гри. Це особливо важливо у проєктах, де деталі рівня мають вирішальне значення для геймплею, а редактор є інтегрованою частиною конвеєру розробки.

Особливо популярними стають веб-редактори, оскільки вони кросплатформенні, не вимагають встановлення додаткового ПЗ і можуть бути розгорнуті як окремий внутрішній інструмент студії. Такі рішення дозволяють

не лише створювати рівні, але й одразу візуалізувати зміни в реальному часі, інтегрувати API для збереження та обміну даними з ігровим рушієм. Це відкриває нові можливості для командної роботи, спрощує тестування та дозволяє швидко адаптувати карту до змін у дизайні гри.

У сучасних розробницьких середовищах також активно використовується концепція повторного використання компонентів (reuse), коли певні блоки або шаблони карт можуть бути збережені та використані повторно в інших частинах гри. Це не лише пришвидшує процес розробки, а й забезпечує єдність стилю та логіки рівнів. Таким чином, редактори карт стають не просто інструментом, а частиною ігрової екосистеми, що охоплює всі етапи - від задуму до реалізації.

Попри значні досягнення у сфері процедурної генерації карт, у багатьох жанрах (наприклад, dungeon crawler або платформери з чіткою логікою) переважає ручне створення рівнів. Це дає змогу досягти кращого балансу, передбачуваності ігрового процесу та художньої довершеності. Ретельно продумані карти сприяють більш захоплюючому і глибшому досвіду для гравців, особливо коли розробник може передбачити емоційну реакцію користувача на певні ділянки рівня або сценарії проходження.

Таким чином, наявність редактора, який дозволяє створювати, редагувати і передавати ігрову карту з мінімальними зусиллями, є ключовою вимогою до сучасного середовища розробки. Це рішення повинно бути не лише функціональним, а й адаптивним, підтримувати різні стилі гри, типи об’єктів та сценарії. Важливо не лише мати інструмент, а й організувати його так, щоб він став невід’ємною частиною загального процесу створення гри, забезпечуючи гнучкість, стабільність та ефективність роботи на всіх етапах проєктування та реалізації рівнів.

1.2. Особливості проєктування редактора карт у веб-середовищі

У сучасному світі веб-технології набувають дедалі більшого значення в різноманітних сферах діяльності, і розробка ігрових інструментів не є винятком.

Проєктування редактора карт у веб-середовищі відкриває перед розробниками унікальні можливості для створення інтуїтивно зрозумілих, доступних і при цьому функціональних рішень. Веб-додатки, що працюють безпосередньо у браузері, позбавляють користувачів необхідності встановлювати додаткове ПЗ, що суттєво розширює аудиторію потенційних користувачів і полегшує процес оновлення та підтримки продукту.

Одним із ключових аспектів проєктування такого редактора є забезпечення максимальної зручності і простоти у роботі з інструментами, адже дизайнер рівнів повинен мати змогу зосередитись саме на творчому процесі, не витрачаючи час на освоєння складних технічних функцій. Веб-середовище дозволяє реалізувати гнучкий інтерфейс з використанням сучасних бібліотек і фреймворків, таких як React.js, які підтримують компонентний підхід та реактивність. Це означає, що зміни, внесені користувачем, одразу ж відображаються на екрані, створюючи ефект миттєвого зворотного зв’язку, що є надзвичайно важливим для комфортної роботи.

Ще одна особливість - це можливість централізованого зберігання і обробки даних за допомогою серверних технологій і баз даних. Використання таких систем, як MongoDB, дозволяє організувати зручний і надійний механізм збереження створених карт [3, с.55], їх версій та різних варіацій, що забезпечує збереження історії змін і спрощує співпрацю між різними членами команди. Таким чином, дані не просто локально зберігаються на комп’ютері розробника, а стають доступними для швидкого доступу, редагування і аналізу в будь-який момент часу.

Проєктування редактора вимагає також продумування логіки взаємодії користувача з елементами карти - кімнатами, об’єктами, зв’язками між ними. Важливо, щоб інтерфейс був максимально зрозумілим, а можливості редагування - гнучкими та розширюваними. Кожен об’єкт повинен мати налаштовувані параметри, які можна легко змінити без потреби вручну редагувати складні структури даних. Для цього застосовують інтуїтивні

інструменти, наприклад drag-and-drop, контекстні меню, візуальні підказки і шаблони.

Веб-середовище також відкриває перспективи для інтеграції редактора з іншими сервісами та інструментами, такими як системи контролю версій, платформи для спільної роботи чи безпосередньо ігровий рушій. Це створює єдину екосистему, де дані можуть плавно передаватися між різними етапами розробки без зайвих проміжних конвертацій, що значно підвищує ефективність робочого процесу.

Окрім технічних переваг, веб-редактор має і соціальні переваги - він дає змогу кільком учасникам команди одночасно працювати над картиною, обмінюватися ідеями, вносити правки у режимі реального часу, що особливо важливо для команд, які працюють віддалено. Ця синергія творчості і технологій робить процес розробки більш організованим і продуктивним.

Загалом, проєктування редактора карт у веб-середовищі - це складний, багатогранний процес, який вимагає врахування як технічних аспектів, так і потреб користувача, прагнення до простоти і зручності, а також відкритості для майбутніх змін і доповнень. У результаті створюється інструмент, який не просто полегшує роботу над рівнями, а й сприяє виникненню нових творчих ідей, допомагає краще організувати процес розробки і надати грі унікальності та глибини.

1.3. Визначення задачі та вимог до системи

Процес розробки будь-якої сучасної програмної системи, особливо такої, яка покликана взаємодіяти з широким колом користувачів та вирішувати комплексні задачі, розпочинається із глибокого, всебічного аналізу. На цьому етапі важливо не лише окреслити загальну концепцію майбутнього продукту, а й детально сформулювати його ключові завдання та вимоги. Це дає змогу не лише краще зрозуміти кінцеву мету, а й уникнути непорозумінь між учасниками команди на етапах проєктування, розробки та тестування. В контексті створення

редактора ігрових карт, цей етап має вирішальне значення, оскільки система повинна гармонійно поєднувати функціональність, гнучкість, стабільність і простоту використання.

Слід зазначити, що розробка редактора ігрових рівнів -це не лише технічна задача. Це багатогранний проєкт, який об'єднує елементи UI/UX-дизайну, архітектурного планування, інтеграції даних та врахування специфіки ігрової логіки. Основна мета полягає у створенні зручного інструменту, який дозволить розробникам незалежно від досвіду легко та ефективно проєктувати, редагувати, зберігати та експортувати карти ігрового світу. Важливо, щоб інтерфейс не викликав відчуття складності, а навпаки -мотивував на творчий експеримент, зменшуючи час на технічні маніпуляції.

Визначення функціональних задач передбачає деталізацію всіх можливих сценаріїв використання редактора. Наприклад, користувач повинен мати змогу обирати та розміщувати об’єкти, змінювати їхні властивості, налаштовувати логіку взаємодії, змінювати освітлення, налаштовувати камеру тощо. Все це вимагає не тільки гнучкого механізму візуального редагування, а й ефективної системи збереження та відновлення даних, що особливо актуально в умовах сучасної розподіленої розробки.

Інтерфейс користувача має бути логічним, послідовним і передбачуваним. Це важливо як для професійних розробників, так і для новачків, які лише починають знайомство зі світом створення ігор. Зрозумілий і інтуїтивно доступний інтерфейс значно полегшує вхід у систему, зменшує бар'єр входу та сприяє швидшому досягненню результату. Однією з особливостей редактора є контекстна чутливість елементів керування - це дозволяє динамічно адаптувати інструменти відповідно до дій користувача, приховуючи зайве та підкреслюючи актуальне.

Важливою вимогою до редактора є підтримка широкого спектра об’єктів, з якими може взаємодіяти користувач. Йдеться не тільки про прості елементи, як-от стіни чи двері, але й складніші сутності: персонажі, пастки, тригери подій, динамічні світлові джерела тощо. Усі ці об'єкти мають набір характеристик, які

повинні бути редагованими у зручній формі, що дозволяє тонко налаштовувати ігрову логіку та візуальну атмосферу рівня. Це, в свою чергу, формує велику кількість вимог до архітектури зберігання даних.

Для збереження й обробки карт було обрано документно-орієнтовану базу даних MongoDB. Вона надає гнучкий підхід до організації даних у вигляді JSON-подібних структур, що ідеально підходить для ієрархічного зберігання інформації про ігрові об’єкти та їх зв’язки. Такий підхід також спрощує процес експорту та імпорту, забезпечує масштабованість системи, дозволяє зберігати великі обсяги даних без втрати швидкодії та полегшує реалізацію резервного копіювання.

З технічної точки зору, система повинна бути побудована з урахуванням вимог до стабільності, продуктивності та масштабованості. Це означає, що навіть при роботі з об’ємними та складними картами редактор має демонструвати стійку поведінку, не втрачати даних, ефективно обробляти ресурсоємні дії, а також мати механізми захисту від некоректних введень та внутрішніх помилок. Надійна система логування, валідація на клієнті та сервері, збереження проміжних версій карти - все це критично важливо для безпечної роботи користувачів і зменшення ризиків.

Особливу роль відіграє синхронізація між редактором та рушієм гри. Вона реалізується за допомогою спеціального формату експорту, зокрема JSON, який дозволяє відобразити ієрархічну структуру карти та забезпечити сумісність з рушієм, що використовує C# та OpenGL для рендерингу. Передбачено, що система експорту повинна враховувати всі властивості об’єктів, їх позиціонування, взаємозв’язки та інші параметри, необхідні для точної репрезентації рівня у грі.

Узагальнюючи, можна стверджувати, що розробка редактора ігрових карт - це складна, багаторівнева задача, що включає елементи аналітики, дизайну, програмування, архітектури даних та логіки взаємодії між компонентами. Створення такого інструменту вимагає не лише технічної компетенції, а й глибокого розуміння потреб кінцевого користувача, гнучкості у виборі рішень та

здатності до адаптації під зміни технологічного ландшафту. Завдяки цьому система має потенціал стати не просто утилітарним редактором, а повноцінною платформою для створення, налаштування та керування ігровим світом.

Висновки до розділу 1

У першому розділі дослідження було здійснено поглиблений аналіз сучасного стану інструментарію для створення ігрових рівнів, із особливим акцентом на веб-орієнтовані платформи. Ретельний огляд наявних рішень дозволив виявити ключові риси, які визначають розвиток галузі: зростання вимог до гнучкості, адаптивності та масштабованості редакторів, а також суттєве збільшення інтересу до інтуїтивних та доступних інтерфейсів, що дозволяють розробникам з різним рівнем досвіду працювати ефективно.

Однією з важливих тенденцій є перехід від традиційних десктопних додатків до кросплатформених веб-інструментів, які не потребують встановлення додаткового ПЗ і доступні з будь-якого пристрою. Веб-редактори нового покоління дедалі активніше впроваджують підтримку спільної роботи в реальному часі, що значно прискорює процес створення контенту та дозволяє зручно взаємодіяти в рамках команд, розподілених по різних локаціях. Також набуває популярності інтеграція з хмарними сховищами, яка гарантує збереження результатів роботи та дозволяє відновити прогрес у разі втрат або помилок.

На основі зібраного матеріалу було сформульовано чіткі функціональні вимоги до майбутньої системи. Серед ключових технічних цілей – забезпечення стабільної та швидкої синхронізації даних між фронтендом (розробленим на React.js) та ігровим рушієм (на базі C# з використанням OpenTK). Для цього було передбачено використання JSON як основного формату передачі даних, що забезпечує гнучкість, простоту парсингу та сумісність із сучасними базами даних, зокрема MongoDB.

Особливу увагу приділено проєктуванню інтерфейсу. Завдяки використанню React.js вдалося створити легкий, динамічний і високореактивний UI. Контекстно-залежна панель інструментів, гарячі клавіші, шаблони дій - усе це спрямоване на підвищення зручності та пришвидшення роботи. Система була спроєктована так, щоб і новачки, і досвідчені користувачі могли однаково ефективно користуватись редактором, не витрачаючи зайвого часу на навчання або пошук потрібних функцій.

Також у рамках дослідження були визначені базові принципи забезпечення продуктивності та стабільності системи. Це включає реалізацію багаторівневої валідації даних (на клієнті, сервері та при імпорті в рушій), використання дельта-оновлень, системи кешування та оптимізованих алгоритмів рендерингу. Такий підхід гарантує надійність у роботі навіть зі складними ігровими рівнями, які містять велику кількість об'єктів та взаємодій.

Підсумовуючи, можна сказати, що проведена робота стала важливим підґрунтям для подальших етапів розробки. Вона дала змогу не тільки глибше зрозуміти сучасні вимоги до редакторів ігрових рівнів, а й окреслити шляхи створення інструменту, який одночасно буде технічно потужним, інтуїтивно зрозумілим і готовим до масштабування. У перспективі така система має усі передумови для подальшого вдосконалення - від інтеграції інтелектуальних підказок до повноцінної підтримки спільної VR-розробки і контекстної генерації контенту на основі машинного навчання.

РОЗДІЛ 2
РОЗРОБКА АЛГОРИТМУ РОЗВ'ЯЗАННЯ ЗАДАЧІ

2.1. Аналіз існуючих підходів до побудови ігрових карт і їх збереження

Розробка ігрових карт є одним із ключових аспектів створення будь-якої комп'ютерної гри. Саме завдяки якісно спроектованим рівням гравець занурюється у світ гри, переживає захоплюючі моменти, взаємодіє з ігровим середовищем, долає перешкоди та проходить сюжетні лінії. Власне тому сучасні розробники приділяють багато уваги не лише візуальному оформленню, але й технічним аспектам створення карт - від їхнього проектування до збереження й подальшого завантаження у гру. На сьогодні існує кілька основних підходів до побудови ігрових карт, кожен із яких має свої особливості, переваги і недоліки.

Одним із найпоширеніших способів є ручне створення карт, коли рівні проєктуються та компонуються за допомогою спеціалізованих редакторів. Такий підхід дає змогу розробникам повністю контролювати кожен елемент карти, ретельно планувати розташування кімнат, ворогів, об'єктів і перепон, що дозволяє досягти високої якості та збалансованості. Приклади таких редакторів можна знайти в популярних ігрових рушіях, як Unity та Unreal Engine, або у вигляді сторонніх програм, як Tiled. Однак цей метод має і суттєві недоліки, головним з яких є тривалість та складність процесу: створення навіть одного рівня може займати багато часу і вимагає значних зусиль від команди дизайнерів і розробників.

З іншого боку, в останні роки великої популярності набирає процедурна генерація рівнів. Цей метод передбачає автоматичне створення карт за допомогою алгоритмів - від простих випадкових генераторів до складних фрактальних або шумових методів. Процедурна генерація дозволяє швидко отримувати безліч унікальних рівнів, що значно розширює ігровий світ і підвищує реіграбельність. Але при цьому контроль над деталями дизайну

втрачається, і можуть виникати проблеми із балансом, що іноді негативно впливає на загальний ігровий досвід.

Сучасні розробники часто намагаються поєднувати обидва підходи, створюючи базову структуру карти вручну, а наповнення дрібними елементами - ворогами, предметами, перепонами - генерують за допомогою алгоритмів. Такий комбінований метод дозволяє зберегти художню цінність ігрового простору, водночас отримуючи гнучкість і різноманіття, що забезпечують процедурні технології [5, с.78].

Для полегшення створення ігрових карт розробникам пропонуються різноманітні інструменти, які мають різний рівень складності та функціоналу. Наприклад, Unity Tilemap Editor дає можливість працювати з тайловими картами - це сітка з маленьких квадратних елементів, які збираються в більшу структуру. Інструмент підтримує малювання, масове редагування, анімації, а також легко інтегрується зі скриптами, що дозволяє додавати складну поведінку. Unreal Engine пропонує ще більш масштабний редактор, що підтримує як 3D, так і 2D сцени, має інструменти для створення ландшафтів, освітлення, звуків та фізичних ефектів. Усе це дозволяє розробникам створювати не лише візуальні локації, але й впроваджувати в них багатий ігровий функціонал. Також популярністю користуються сторонні редактори, як Tiled, що спеціалізуються на створенні тайлових карт і можуть зберігати інформацію у вигляді JSON чи XML. Ці формати легко інтегруються з різними ігровими рушіями і прості у використанні.

Уже кілька років спостерігається тенденція до розробки веб-редакторів карт, які працюють безпосередньо у браузері. Завдяки прогресу веб-технологій (React, Vue, WebGL) стало можливим створювати потужні інструменти, які не потребують установки і можуть працювати на будь-якому пристрої з інтернетом. Це відкриває нові можливості для командної роботи над проєктом, спрощує оновлення і підтримку редакторів, а також знижує технічні бар’єри для користувачів.

Важливою складовою створення ігрових карт є формат, у якому зберігаються всі дані. Він повинен бути достатньо гнучким, щоб зберігати складні структури - розташування кімнат, параметри ворогів, об’єкти та інші атрибути. Найпопулярнішим форматом є JSON - він є легким для читання і написання, підтримується практично всіма мовами програмування і чудово підходить для роботи з ігровими даними. XML, хоч і більш формальний та тяжкий, використовується рідше через громіздкість, а для підвищення продуктивності іноді застосовують власні бінарні формати, які складніші для розуміння, але швидші у обробці. Важливою частиною сучасних проєктів стає збереження карт у БД. Тут популярність отримали NoSQL рішення, такі як MongoDB, що зручно зберігають документи у форматі JSON, легко масштабуються і підтримують складні структури. Реляційні бази даних також використовуються, але вони часто вимагають більш складних схем і з’єднань таблиць, що ускладнює роботу.

Збереження ігрових карт у хмарі і через веб-інтерфейси не лише спрощує процес розробки, але й покращує доступність і гнучкість. Розробники можуть швидко змінювати карти, оновлювати версії, а гравці - завантажувати нові рівні без необхідності оновлення клієнта. Однак цей підхід накладає певні виклики, зокрема щодо сумісності даних між редактором і грою, необхідності жорсткої валідації карт, щоб уникнути помилок, і забезпечення стабільної роботи при великій кількості об’єктів.

Відомі проєкти, такі як TBOI, Dead Cells та Hollow Knight, демонструють, що поєднання ручного проектування з процедурною генерацією дає змогу створювати захоплюючі та різноманітні рівні, які одночасно збалансовані та цікаві для проходження [6, с.89]. Ці ігри використовують складні алгоритми для перевірки правильності розташування кімнат, пошуку стартових позицій гравця та обробки логіки бою. Вони також показують важливість інтеграції редакторів і збереження карт у зручних форматах, що дозволяє безболісно оновлювати ігровий контент.

2.2. Розробка алгоритму розв’язання задачі

Процес розробки алгоритму розв’язання задачі в рамках нашого програмного проєкту, що поєднує функціональність редактора карт та геймплейної частини гри, є одним із найважливіших етапів всього циклу розробки. Цей процес не тільки забезпечує технічну реалізацію ключових функціональних елементів, але і формує логіку взаємодії між користувачем та системою, закладаючи основи для стабільності, гнучкості та подальшого масштабування гри. Саме тому виникає необхідність у створенні чіткого, послідовного та добре структурованого алгоритму, який зможе ефективно об’єднати редакторську ігрову частину, забезпечити повну відповідність вимогам проєкту, а також бути готовим до подальших розширень. Алгоритм має охоплювати усі аспекти – від початкового створення карти до її безпосереднього використання у геймплейному циклі гри.

Для кращого розуміння логіки реалізації алгоритму на Рис. 2.1. представлено його блок-схему у вигляді узагальненої послідовності ключових етапів.

2.2.1. Створення карти

Перший і, без перебільшення, один з найважливіших кроків у нашому алгоритмі - це створення самої карти. Тут ми маємо справу з інтерфейсом редактора, який дає змогу користувачеві у зручний та інтуїтивно зрозумілий спосіб взаємодіяти з візуальним представленням карти, створюючи унікальні рівні. Карта реалізується у вигляді сітки розміром 5×5, що передбачає наявність 25 можливих комірок для розміщення кімнат. Користувач має можливість розмістити до 6 звичайних кімнат та 1 кімнату боса, обираючи їх положення за допомогою координатної системи.

Однак варто зазначити, що ця свобода не є абсолютною - розміщення підлягає певним логічним обмеженням. Наприклад, кімната боса не може мати більше одного сусіда - це критично важливо для забезпечення коректної ігрової логіки, ускладнення доступу до фінального бою та збереження структурної

цілісності рівня. Логіка розміщення кімнат полягає в тому що першу кімнату можна розмістити в будь якій клітинці, а вже всі подальші кімнати мають бути суміжними хоча б з одною з розміщених - утворюючи зв’язний граф, що дозволяє гравцеві безперешкодно проходити рівень. Наявність «розірваних» або ізольованих кімнат є недопустимою.

Рис. 2.1. Блок-схема алгоритму

На додаток до зовнішньої структури, користувач має змогу детально наповнювати кожну окрему кімнату. Це відбувається у вигляді сітки 13×7, де він може вручну розставляти ворожі юніти, бар’єри, перепони, скрині та інші ігрові

об’єкти. Усі ці елементи впливають на складність рівня, геймдизайн та гравітаційні особливості кімнати, тому їх налаштування повинне відбуватись у межах чітко регламентованих параметрів.

2.2.2. Серіалізація карти у форматі JSON та збереження в БД

Після проходження етапу створення наступним логічним кроком є серіалізація карти - тобто її перетворення у цифровий формат, придатний для збереження, передачі та використання в ігровій частині. У нашому випадку використовується формат JSON (JavaScript Object Notation) - сучасний, зручний і легковаговий формат, який дозволяє зберігати структуровані дані у вигляді пар "ключ-значення".

Кожна карта, яку створює користувач, перетворюється на seed - структурований опис рівня, що містить:

- координати розташування кожної кімнати;
- інформацію про зв’язки між кімнатами;
- внутрішній вміст кімнат (об’єкти, вороги);
- технічну метаінформацію (автор та назва карти).

Для збереження карт було обрано базу даних MongoDB, що є ідеальним інструментом для зберігання документів типу JSON. Такий підхід дозволяє уникнути складних структур реляційних баз даних, забезпечити гнучкість і масштабованість, а також легко організувати доступ до карт.

Дані передаються через REST API між клієнтом (редактором) і сервером, що відкриває широкі можливості для побудови кросплатформної архітектури, а також забезпечує зручність при розширенні проєкту. Завдяки цьому зберігання карт стає максимально ефективним та адаптивним до змін.

2.2.3. Завантаження карти у грі

Коли гравець запускає гру, в меню він може вибрати запуск останньої створеної карти або список всіх карт. Залежно від конкретного сценарію, гра може:

- завантажити останню збережену карту;
- завантажити повний список карт;

Сервер у відповідь надсилає JSON-структуру, яку ігровий клієнт парсить у реальному часі, формуючи повноцінну внутрішню репрезентацію карти. Це дозволяє уникнути дублювання даних у грі, зменшити обсяг клієнтського коду і забезпечити максимальну адаптивність.

Таким чином, завантаження карти є критично важливим проміжним етапом, який поєднує редакторську та ігрову частину і забезпечує безперебійну передачу ігрового контенту між ними.

2.2.4. Відтворення карти та реалізація геймплею

Після успішного завантаження карти гра переходить до режиму її відтворення - іншими словами, інтерпретації структури seed-файлу у вигляді реальних ігрових об’єктів, ворожих юнітів, геометрії кімнат, гравця та інших елементів.

Один з перших кроків - вибір стартової кімнати гравця. Для цього використовується алгоритм, який або застосовує пошук у ширину, або обчислює Манхеттенську відстань до кімнати боса, знаходячи найбільш віддалену початкову позицію. Такий підхід ускладнює ігровий процес, змушуючи гравця обдумано взаємодіяти з навколишнім середовищем.

Рух гравця між кімнатами реалізований на основі координатної логіки: гравець може переміщуватись лише до суміжних кімнат, перевіряючи дозволені переходи на кожному кроці. У разі недопустимої дії система або блокує переміщення, або інформує гравця через HUD.

Усередині кімнат діє бойова система, яка складається з базового штучного інтелекту ворогів. Вони:

- пересуваються в межах кімнати;
- атакують при наближенні гравця;
- реагують на отримані пошкодження;

Гравець керується через клавіатуру, може стріляти, ухилятись, взаємодіяти з предметами. Його стан відображається через HUD (панель стану), що показує поточне здоров’я, позицію на карті.

Висновки до розділу 2

У межах другого розділу було здійснено розробку алгоритму, який забезпечує повноцінну взаємодію між користувачем редактора карт і гравцем у реальному ігровому процесі. Основний акцент було зроблено на створенні цілісного, підходу до обробки ігрових рівнів, що включає всі ключові етапи: проєктування структури карти, збереження у форматі JSON, передавання через REST API та подальше відображення у грі.

Зокрема, було розроблено алгоритм перевірки коректності карти. Такий підхід не лише запобігає створенню некоректних конфігурацій, а й покращує загальний геймплей. Важливо, що внутрішня структура кожної кімнати також проєктується за допомогою окремого інтерфейсу, що дозволяє розміщувати об’єкти у межах сітки й зберігати їх у форматі, придатному для подальшої обробки.

У процесі реалізації особлива увага приділялась зручності серіалізації та збереження карт. Завдяки використанню формату JSON і бази даних MongoDB було досягнуто високої гнучкості в обробці даних та простоти обміну інформацією між клієнтом і сервером. Це створює технічне підґрунтя для масштабованості системи.

Ігрова частина системи також інтегрується з сервером: карта динамічно завантажується в гру, де визначається стартова позиція гравця за допомогою алгоритмів пошуку, таких як BFS або Манхеттенська відстань. Це дозволяє створити більш збалансований ігровий досвід. Переміщення між кімнатами відбувається на основі координат, а взаємодія з ворогами реалізована через базову логіку штучного інтелекту.

Таким чином, розділ демонструє не лише реалізацію ключових алгоритмічних рішень, але й формує технічну та логічну основу всього проєкту. Побудована система є гнучкою, масштабованою та готовою до подальшого розвитку, що робить її актуальною як для майбутнього розширення функціоналу, так і для практичного використання в умовах реального геймдеву.

РОЗДІЛ 3
ОРГАНІЗАЦІЯ ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

У процесі створення інтерактивного веб-редактора мап та відповідної ігрової частини особливе значення має правильно організоване інформаційне забезпечення системи. Інформація в такому проєкті не лише є джерелом, з якого користувач отримує контент, а й виступає центральним елементом, навколо якого будуються усі процеси - від візуалізації й редагування, до збереження, передачі й використання в ігровому рушії.

У даному випадку система базується на використанні трьох головних типів інформації:

- вхідної;
- системно-генерованої;
- збереженої.

Вхідна інформація - це та, яку безпосередньо створює або редагує користувач під час роботи з редактором. Саме вона є відправною точкою для формування структури карти. Користувач працює з інтерактивною сіткою розміром 5x5, у якій можна розмістити до шести звичайних кімнат та одну кімнату боса. При цьому враховуються певні обмеження - наприклад, кімната боса має бути віддаленою та мати лише одного сусіда, а всі інші кімнати повинні бути взаємопов’язаними.

Окрім базового розміщення, користувач також задає внутрішню конфігурацію кожної кімнати. Для цього передбачений редактор на основі сітки 13x7, де можна розташовувати ворогів та перепони. Уся ця інформація формується у вигляді вкладених структур, що миттєво зберігаються у внутрішньому стані програми - наприклад, у глобальному сховищі стану (типу Redux або Context API). Таким чином, кожен елемент карти має свою позицію, тип, вміст та логічні зв’язки з іншими кімнатами.

На основі введених даних система додатково генерує інформацію, яку користувач безпосередньо не задає. Це стосується, зокрема, координат кімнат у

просторі, автоматично згенерованого seed-а карти, а також таких параметрів, як стартова позиція гравця, яка визначається з використанням одного з алгоритмів обходу графів - наприклад, пошуку в ширину або варіації алгоритму Манхеттена. Валідація мапи, перевірка її коректності (наявність зв'язності, відсутність конфліктів, правильність розташування кімнати боса) також здійснюється автоматично. У результаті формується вже повноцінна інформаційна модель карти, готова до подальшого використання.

Наступним важливим етапом у процесі створення карти є збереження всієї сформованої інформації у базі даних. Після завершення редагування мапи на стороні клієнта, зібрана структура - яка представлена у форматі JSON - автоматично надсилається на сервер за допомогою HTTP-запиту через REST API. Цей запит включає повний набір даних, що описують карту: її загальні параметри, конфігурацію кімнат, їхній вміст, логічні зв'язки та супровідну інформацію, необхідну для правильного відтворення на боці ігрового рушія.

На сервері отримана інформація проходить початкову валідацію та обробку, після чого зберігається в базі даних MongoDB у вигляді окремого документа. Структура цього документа детально описана у Таб. 3.1. У процесі збереження для кожної мапи автоматично створюється унікальний ідентифікатор (_id), що дозволяє однозначно ідентифікувати її серед інших записів. Також фіксується userId - ідентифікатор користувача, який створив карту, що забезпечує коректну прив'язку мапи до власника. Окрім того, зберігається name - назва карти, яку ввів користувач, і mapSeed - спеціальна структура, що фактично є серцем мапи, тобто включає в себе всю логіку та розміщення кімнат.

Цей mapSeed, як вказано в Таб. 3.1, є вкладеним об'єктом, який містить дані про кожну окрему кімнату. Кожна кімната має свої координати (x, y), тип - наприклад, звичайна або кімната з босом, а також об'єкт content, який деталізує її наповнення. Повна структура однієї кімнати описана у Таб. 3.2.

Таблиця 3.1

Таблиця збереження карти

Поле	Тип	Опис
_id	ObjectId	Унікальний ідентифікатор карти
userId	ObjectId	Ідентифікатор користувача (зв’язок з колекцією `User`).
name	String	Назва карти, введена користувачем. Має бути унікальною для кожного користувача.
mapSeed	Object	Основна структура, яка містить усю інформацію про мапу (кімнати, їхній тип, вміст тощо).

Таблиця 3.2

Структура однієї кімнати

Поле	Тип	Опис
x, y	Number	Координати кімнати на мапі (зазвичай у межах сітки 5x5).
type	String	Тип кімнати: `normal` - звичайна, або `boss` - кімната з босом.
content	Object	Об’єкт, який містить вміст кімнати: ворогів, перешкоди, боса тощо.

У свою чергу, вміст кожної кімнати, що знаходиться в полі content, складається з масиву ворогів (enemies), масиву перепон (obstacles) і, при потребі, поля boss, яке вказує на наявність фінального противника. Ці атрибути дають змогу максимально гнучко та точно відтворити внутрішній простір кожної кімнати під час ігрового процесу. Детальна структура вмісту кімнати подана у Таб. 3.3.

Таблиця 3.3

Вміст кімнати

Поле	Тип	Опис
enemies	Array	Список ворогів, які розміщені в кімнаті.
obstacles	Array	Список ворогів, які розміщені в кімнаті.
boss	String	Наявність боса

Коли користувач обирає мапу для запуску, її дані завантажуються з бази через API і передаються до ігрового клієнта, який реалізований на C# із використанням OpenTK - обгортки для OpenGL. Тут JSON-структура парситься у внутрішні ігрові об’єкти, які вже безпосередньо використовуються рушієм для відображення карти, генерації ворогів та реалізації логіки боїв. Розміщення кімнат визначає просторову конфігурацію рівня, об'єкти в кожній кімнаті ініціалізуються відповідно до збережених параметрів, а поведінка ворогів та боса - згідно з заздалегідь заданими шаблонами. Усі ці процеси відбуваються динамічно, що забезпечує взаємодію між рівнем та гравцем у режимі реального часу.

Загалом, можна сказати, що інформаційне забезпечення системи - це не набір зовнішніх джерел або посилань, а комплекс логічно пов’язаних структур, що беруть свій початок у діях користувача, проходять через систему валідації та обробки, зберігаються в базі даних, а далі використовуються рушієм гри. Такий підхід дозволяє реалізувати гнучке, масштабоване і функціонально повноцінне середовище, у якому інформація не лише зберігається, а й постійно трансформується та використовується на всіх етапах роботи системи - від створення до безпосередньої взаємодії в грі.

Висновки до розділу 3

У цьому розділі було детально розглянуто інформаційне забезпечення системи інтерактивного веб-редактора ігрових мап, що є ключовим елементом для її ефективної роботи. Інформація в системі виступає не просто даними, а центральною складовою, навколо якої побудовані всі процеси: від створення і редагування карти користувачем, до її збереження і подальшого відтворення в ігровому рушії.

Система працює з трьома основними типами інформації: вхідною, системно-генерованою та збереженою. Вхідна інформація - це дані, які користувач безпосередньо створює та змінює в редакторі, формуючи структуру

карти на сітці 5x5 з розміщенням кімнат, враховуючи обмеження на логіку їх розташування. Для детального наповнення кімнат використовується внутрішня сітка 13x7, де можна розташовувати ворогів і перепони. Вся ця інформація зберігається у внутрішньому стані програми, що забезпечує швидкий доступ і оновлення.

Системно-генерована інформація включає автоматично обчислені параметри - координати кімнат, унікальний seed карти, стартову позицію гравця, а також валідацію правильності карти. Це дозволяє гарантувати коректність структури та логічну цілісність карти перед її збереженням.

Збереження відбувається у форматі JSON, який через REST API передається на сервер і записується у базу даних MongoDB. Кожна карта отримує унікальний ідентифікатор, прив’язку до користувача, назву і повну структуру mapSeed - об’єкта, що містить детальний опис кімнат та їх вмісту. Такий підхід забезпечує надійне зберігання і швидкий доступ до даних, що необхідні для відтворення карти у грі.

Завантажена карта парситься у ігровому клієнті, реалізованому на C# з OpenTK, де дані трансформуються у внутрішні ігрові об’єкти. Це дає змогу динамічно відтворювати рівень, розміщувати ворогів та перепони, а також реалізовувати ігрову логіку у реальному часі.

Отже, інформаційне забезпечення системи - це не просто набір статичних даних, а комплексна, динамічна структура, яка підтримує всі етапи роботи з картою - від створення до інтерактивної взаємодії в ігровому процесі. Такий підхід гарантує гнучкість, масштабованість і надійність системи, що є важливою основою для подальшого розвитку редактора та впровадження нових функціональних можливостей.

РОЗДІЛ 4
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1. Розробка веб-застосунку на React

Розробка редактора рівнів у веб-застосунку базується на використанні сучасних технологій та підходів, які дозволяють створити інтуїтивно зрозумілий, гнучкий та масштабований інструмент для побудови ігрових мап у стилі TBOI. Важливою складовою є застосування React - бібліотеки, що широко використовується для створення інтерфейсів із реактивним оновленням стану та можливістю легкої інтеграції з бекендом. Завдяки цьому забезпечується висока продуктивність, зручність взаємодії з користувачем і простота в розширенні функціоналу.

4.1.1. Архітектурні рішення

Компонент редактора карти побудований з урахуванням принципів компонентного дизайну. Інтерфейс умовно поділений на логічні частини:

- Сітка 5x5 для розміщення кімнат;
- Панель вибору типу кімнати (звичайна/бос);
- Форма введення назви карти;
- Кнопка збереження.

Кожен з цих блоків реалізовано як окремий компонент, що відповідає за свою функціональність. Такий підхід робить код прозорим, легко масштабованим та підтримуваним. Інформація про розташування кімнат, їхній тип і вміст зберігається у стані через React-хук useState, що забезпечує моментальну реакцію інтерфейсу на зміну даних без перезавантаження сторінки.

4.1.2. Логіка розміщення кімнат

Редактор передбачає чіткі правила: користувач може розмістити до 6 звичайних кімнат та одну кімнату боса. Всі дії виконуються на сітці 5x5. Алгоритм перевіряє, чи має вибрана клітинка сусідні кімнати, чи не порушено

унікальність кімнати боса, та інші обмеження, спрямовані на баланс ігрового процесу.

Ці правила відіграють надзвичайно важливу роль у процесі створення ігрових рівнів, оскільки вони необхідні не лише для того, щоб контролювати загальну структуру карти, але й для того, щоб стимулювати користувача до більш усвідомленого і стратегічного підходу при розміщенні кімнат (рис. 4.1). Без цих правил процес розташування кімнат міг би стати хаотичним і неорганізованим, що суттєво вплинуло б на якість ігрового досвіду. Натомість, чітко визначені обмеження та рекомендації дозволяють уникнути випадковості і сприяють формуванню рівнів, які є логічно послідовними, різноманітними за структурою та приємними для сприйняття з візуальної точки зору (Додаток А).

Варто підкреслити, що всі ці правила та обмеження, що регламентують можливість розміщення кімнат у просторі мапи, реалізовані в спеціальній функції під назвою canPlaceRoom. Основне завдання цієї функції полягає у ретельній перевірці, чи є допустимим розміщення нової кімнати саме на визначених координатах (x, y). При цьому враховується не лише поточний стан карти, а й специфіка обраного типу кімнати, що має бути додана. Завдяки такій реалізації забезпечується цілісність ігрової структури, виключається можливість некоректного або нелогічного розташування кімнат, що в свою чергу сприяє більш гармонійному ігровому процесу.

```
const canPlaceRoom = (x: number, y: number): boolean => {
  if (rooms.find((r) => r.x === x && r.y === y)) return false;
  const neighbors = getNeighbors(x, y);
  if (rooms.length === 0) return true;
  const bossRoom = rooms.find((r) => r.type === "boss");
  if (roomPlacementType === "boss") return neighbors.length === 1;
  if (bossRoom) {
    const bossNeighbors = getNeighbors(bossRoom.x, bossRoom.y);
    const isNextToBoss =
      (Math.abs(bossRoom.x - x) === 1 && bossRoom.y === y) ||
```

```
    (Math.abs(bossRoom.y - y) === 1 && bossRoom.x === x);
  if (isNextToBoss && bossNeighbors.length >= 1) return false;
}
return neighbors.length >= 1;
```


```
};
```

Рис. 4.1. Загальний вигляд редактора мапи з розміщеними кімнатами

4.1.3. Взаємодія користувача з інтерфейсом

Редактор побудований з акцентом на простоту взаємодії:

- Одинарний клік по клітинці додає кімнату обраного типу.
- Подвійний клік відкриває модальне вікно для детального редагування.
- Ctrl+клік - видаляє існуючу кімнату.

Кожна дія супроводжується візуальними підказками - зміною курсору, підсвіткою, спливаючими підказками - що значно покращує користувацький досвід.

4.1.4. Вибір типу кімнати

Перемикання між режимами розміщення звичайної кімнати та кімнати боса відбувається за допомогою кнопок-перемикачів. Якщо кількість певного типу кімнат перевищує допустиму, система блокує дію та показує попередження. Це убезпечує від помилок і дотримується геймдизайнерської логіки.

4.1.5. Редактор кімнат

Після створення кімнати користувач може перейти до її редагування (Рис. 4.2.). У звичайних кімнатах доступна сітка 13x7, де можна додавати ворогів, перешкоди (каміння, прірви), а також об'єкти декору. Усі елементи на правій панелі, що інтуїтивно знайомо навіть недосвідченим користувачам.

Редактор кімнати відкривається як модальне вікно, що накладається поверх мапи. Кожна зміна миттєво відображається у стані React-компонента, а при закритті - синхронізується з основною мапою.

Компонент RoomEditor відповідає за відображення редактора окремої кімнати у веб-застосунку. Його основна мета - надати користувачу зручний інтерфейс для створення та редагування вмісту кімнати, включаючи розміщення ворогів, перешкод та боса. У рамках цього інтерфейсу реалізовано повноцінну систему керування об’єктами за допомогою інтерактивної сітки та панелі інструментів.

На початку компонента оголошено кілька локальних станів за допомогою хука useState. Вони відповідають за активний інструмент (selectedTool) та вибрані типи елементів: selectedObstacle, selectedEnemy і selectedBoss. Таким чином, редагування відбувається в залежності від того, який тип об'єкта обрано користувачем на панелі.

Окремо створюється стан content, який містить локальну копію вмісту кімнати, ініціалізовану через глибоке копіювання об’єкта room.content.

Цей підхід дозволяє уникнути прямої мутації пропсів, що є важливою вимогою в React. Зміни, які вносить користувач під час редагування, відображаються лише в локальному стані. Таким чином, оригінальні дані

кімнати залишаються незмінними до моменту, поки користувач явно не натисне кнопку «Зберегти».

```
const RoomEditor: React.FC<RoomEditorProps> = ({ room, onClose, onSave }) => {
  const [selectedTool, setSelectedTool] = useState<TileType>("empty");
  const [selectedObstacle, setSelectedObstacle] = useState<Obstacle | null>(null);
  const [selectedEnemy, setSelectedEnemy] = useState<Enemy | null>(null);
  const [selectedBoss, setSelectedBoss] = useState<Boss | null>(null);

  const [content, setContent] = useState<RoomContent>(() => {
    return JSON.parse(JSON.stringify(room.content));
  });
```

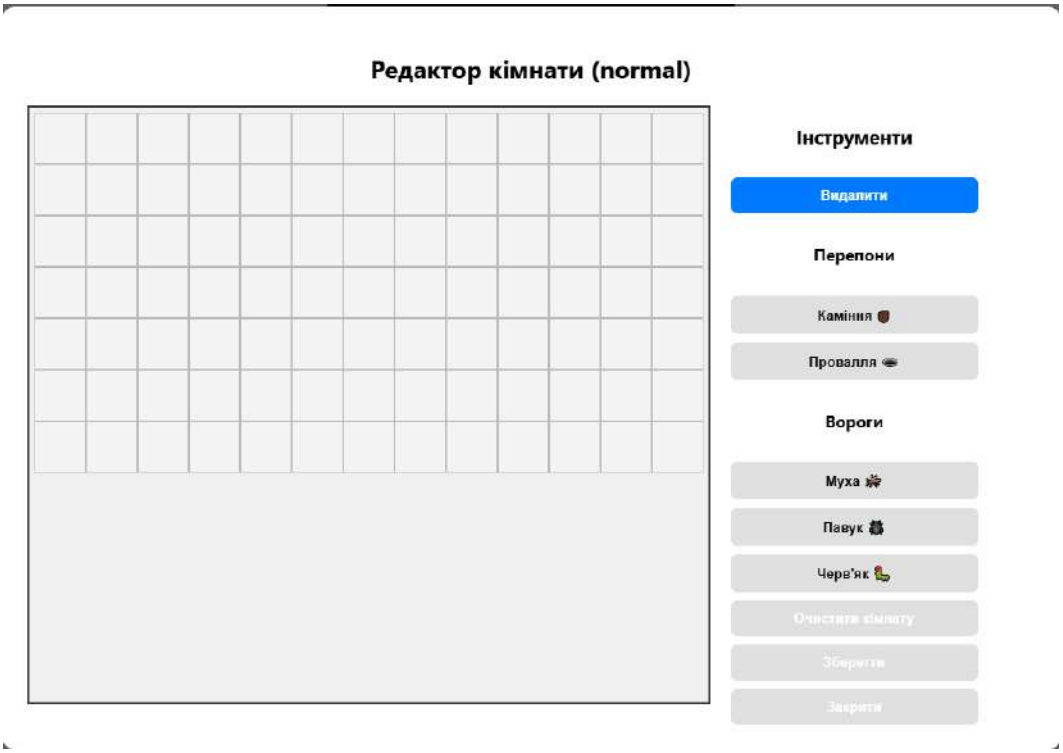

Рис. 4.2. Інтерфейс редактора звичайної кімнати

4.1.6. Особливості кімнати боса

Кімната боса (Рис. 4.3.) - унікальна за логікою та змістом. Вона розміщується лише одна і призначена для фінального протистояння. Редактор такої кімнати має доповнений набір елементів окрім звичайних дрібних ворогів можна вибрати боса зі списку. Це дозволяє створювати арену для головного бою на рівні.

У кімнаті боса також діє сітка 13x7, розміщення елементів тут не обмежене, що дозволяє ускладнити бій з босом.

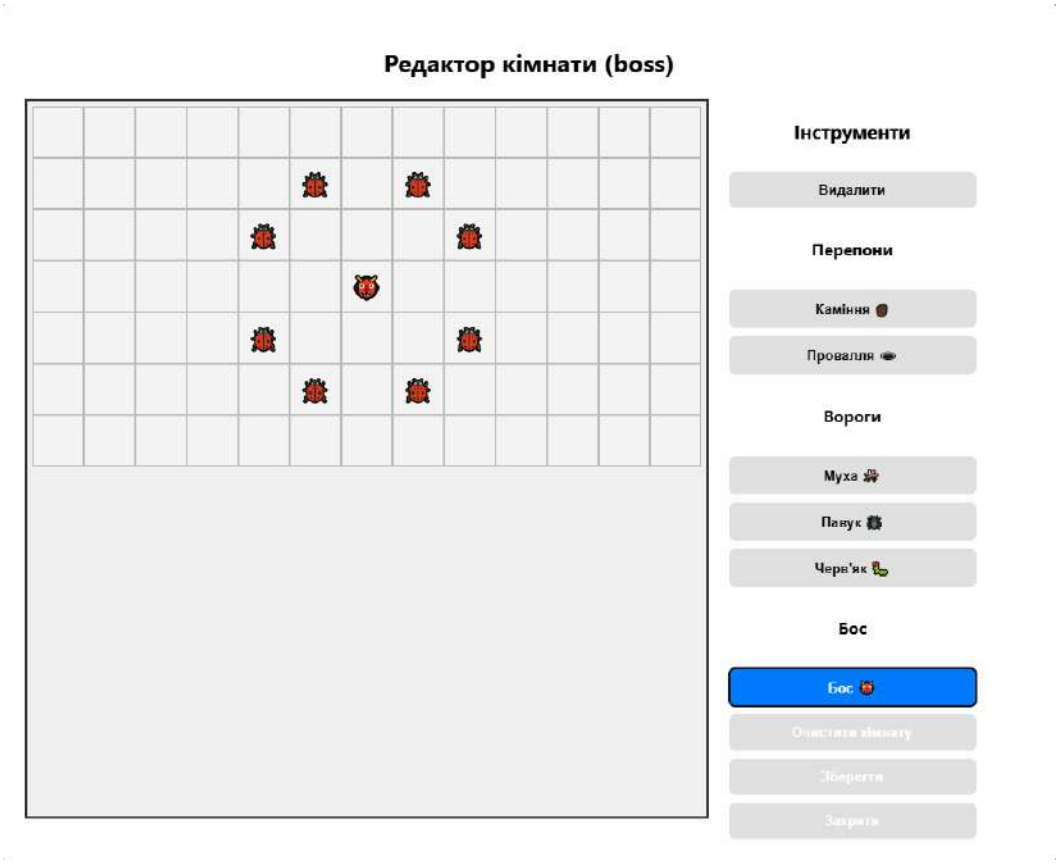

Рис. 4.3. Інтерфейс редактора кімнати боса

4.1.7. Збереження створеної мапи

Збереження мапи реалізовано через асинхронний запит до серверногоAPI. Перед відправкою система перевіряє:

- наявність унікальної назви мапи;
- авторизаційний токен;
- коректність структури.

Мапа серіалізується у формат JSON і надсилається на сервер, а у відповідь користувач отримує повідомлення про успішне збереження або помилку. Це дозволяє зберігати створені рівні, ділитись ними з іншими або повертатись до редагування пізніше.

4.2. Структура бази даних

Побудова надійної та зручної системи зберігання даних - один із ключових аспектів у розробці будь-якого складного веб-застосунку. Особливо це актуально у випадку редактора рівнів, де користувач повинен мати змогу зберігати свої напрацювання, повертатися до них пізніше, а також ділитися результатами з іншими. У даному проєкті реалізація бази даних здійснюється за допомогою MongoDB - документо-орієнтованої бази, яка дозволяє зберігати складні вкладені структури у вигляді документів JSON-подібного формату.

MongoDB ідеально підходить для застосунків з динамічною структурою даних, де кожна карта може містити різну кількість кімнат, різне наповнення та унікальні властивості. У поєднанні з ORM-бібліотекою Mongoose, яка надає можливість визначати схеми документів, забезпечувати валідацію та описувати зв’язки між сутностями, ми отримуємо потужний і зручний інструментарій для розробки.

У цьому розділі розглянемо, як виглядає структура зберігання даних у базі, які типи документів використовуються, як вони пов’язані між собою, та як організована інформація, що відображає ігрову карту, її внутрішній вміст і власника.

Для реалізації функціоналу редактора достатньо двох ключових сутностей - карти та користувача. Кожна створена мапа належить певному користувачу, що дозволяє не лише зберігати дані персоналізовано, але й реалізовувати авторизацію, обмеження доступу до редагування, та впорядковування карт за авторами.

Кожне поле в таблиці Map (Таб. 4.1) виконує важливу роль. Наприклад, _id унікальний ідентифікатор карти, а userId дозволяє пов’язати мапу з автором, що критично для фільтрації карт, виведення списку своїх рівнів, а також для реалізації авторизації. Поле name виконує функцію ідентифікатора мапи на рівні користувача і дозволяє швидко знаходити потрібний рівень. Нарешті, mapSeed - це справжнє ядро мапи: у ньому зберігається структура кімнат, тобто все, що бачить гравець у редакторі.

Таблиця 4.1

Таблиця збереження карти

Поле	Тип	Опис
_id	ObjectId	Унікальний ідентифікатор карти.
userId	ObjectId	Ідентифікатор користувача (зв’язок з колекцією `User`).
name	String	Назва карти, введена користувачем. Має бути унікальною для кожного користувача.
mapSeed	Object	Основна структура, яка містить усю інформацію про мапу (кімнати, їхній тип, вміст тощо).

Поле mapSeed (таб. 4.2) - це складний вкладений об’єкт, який містить в собі дані про всі кімнати, розташовані на мапі. Ці дані включають координати кожної кімнати (відповідно до сітки 5x5), її тип, а також деталі вмісту: ворогів, перешкоди, та в особливих випадках - боса.

Ця гнучка структура дозволяє без особливих труднощів додавати нові типи кімнат, додаткові параметри або механіки, як-от пастки, тригери, тощо.

Таблиця 4.2

Структура однієї кімнати

Поле	Тип	Опис
x, y	Number	Координати кімнати на мапі (зазвичай у межах сітки 5x5).
type	String	Тип кімнати: `normal` - звичайна, або `boss` - кімната з босом.
content	Object	Об’єкт, який містить вміст кімнати: ворогів, перешкоди, боса тощо.

Координати x та y дозволяють візуально зобразити кімнату у правильному місці на карті. type важливий для відображення різних стилів кімнат та для застосування ігрових правил. A content - це вже безпосередньо вміст, що визначає геймплей.

Для кращого розуміння того, як зберігається наповнення кімнати, розглянемо його окремо для двох типів: звичайних (таб. 4.3) та босових (таб. 4.4).

Таблиця 4.3

Вміст звичайної кімнати

Поле	Тип	Опис
enemies	Array	Список ворогів, які розміщені в кімнаті.
obstacles	Array	Список ворогів, які розміщені в кімнаті.

Таблиця 4.4

Вміст кімнати боса

Поле	Тип	Опис
enemies	Array	Список ворогів, які розміщені в кімнаті.
obstacles	Array	Список ворогів, які розміщені в кімнаті.
boss	String	Назва або ідентифікатор боса, що з’являється у кімнаті.

Користувач (таб. 4.5) - це центральна частина системи авторизації. Завдяки наявності таблиці користувачів ми можемо зберігати карти окремо для кожного, реалізувати функцію «мої мапи», дозволити приватність, і навіть зробити спільне редагування або публікацію.

Таблиця 4.5

Таблиця збереження користувача

Поле	Тип	Опис
_id	ObjectId	Унікальний ідентифікатор користувача у базі даних.
username	String	Ім’я користувача - використовується для логіну та авторства.
password	String	Хешований пароль для захищеної авторизації.

Модель User є фундаментальною складовою системи безпеки та захисту приватності застосунку. Всіінші сутності та дані безпосередньо пов’язані з користувачем через його унікальний ідентифікатор (_id). Такий підхід забезпечує чітку прив’язку даних до конкретного користувача, що є необхідною умовою для контролю доступу та персоналізації контенту.

Зв’язки між таблицями у базі даних реалізовані максимально просто, але при цьому є дуже ефективними. Зокрема, кожна карта містить поле userId, яке посилається на відповідний запис у таблиці User. Це дозволяє в будь-який момент отримати повний список карт, створених конкретним користувачем, а також забезпечує можливість перевірки авторства карти перед наданням прав на редагування або видалення. Таким чином, модель даних гарантує цілісність інформації та підтримує належний рівень безпеки при роботі із користувацьким контентом.

4.3. Розробка гри на C#

4.3.1. Підключення до бази даних і отримання сіда

Розробка гри, особливо такої, що базується на динамічно створених рівнях, неможлива без ефективної взаємодії з базою даних. У нашому випадку дані про рівні - це не просто статичні файли, а складна структура, що зберігається на сервері у базі даних MongoDB. Для доступу до цих даних із гри, написаної на C#, ми використовуємо спеціальний проміжний шар - HTTP API.

Уявімо, що наша БД знаходиться десь на сервері, який приймає запити і повертає дані у форматі JSON. В C# ми реалізуємо клієнт, який звертається до цього API, отримує список доступних карт, а потім, вибравши потрібну, завантажує повний опис рівня - так званий mapSeed.

Цей процес складається з кількох етапів:

1. Ініціалізація HTTP-з’єднання.

Спершу ми створюємо клієнт, який знає адресу сервера, де розміщене API. Завдяки цьому клієнт може робити HTTP-запити (GET, POST і т.д.) до потрібних маршрутів.

2. Отримання списку доступних карт.

Клієнт посилає запит до маршруту API, який повертає масив об’єктів - кожен з яких містить основну інформацію про карту: унікальний ідентифікатор та назву. Це дає змогу показати користувачу перелік доступних рівнів для вибору.

3. Завантаження конкретної карти за її ідентифікатором.

Після вибору потрібної карти відбувається повторний запит до сервера, який повертає повний JSON із детальною структурою mapSeed. Цей об’єкт містить усю інформацію про кімнати, їх координати, типи, ворогів і інший вміст.

4. Обробка отриманих даних.

Після того, як JSON отримано, його треба "розпакувати" - перетворити у внутрішні об’єкти гри. Для цього використовується механізм десеріалізації, який переводить JSON у відповідні класи чи структури, зручні для подальшої роботи у грі.

5. Побудова внутрішнього представлення карти.

Отримані дані використовуються для створення об'єктів кімнат, розташування ворогів, перешкод і так далі. На основі цих об'єктів формується внутрішня модель карти, яка потім використовується в ігровому циклі для візуалізації, логіки та взаємодії.

Прикладом підключення до бази даних є метод GetAllMapsAsync, який реалізує завантаження списку всіх доступних карт із серверного API. Цей метод є асинхронною функцією, що виконує HTTP-запит до кінцевої точки "public" на бекенді.

Його основне призначення - отримати дані про карти у форматі JSON, розпарсити їх і перетворити у зручну для подальшої роботи структуру - список кортежів виду (id, name), де id - унікальний ідентифікатор карти з бази даних, а name - її назва.

Отриманий список використовується на стороні клієнта, зокрема, для відображення у вигляді переліку карт в графічному інтерфейсі користувача. Завдяки цьому користувач може зручно обрати карту для редагування або запуску гри.

На рисунку 4.4 наведено приклад інтерфейсу, де реалізовано вивід отриманого списку карт і можливість вибору однієї з них.

```
public async Task<List<(string id, string name)>> GetAllMapsAsync()
{
    HttpResponseMessage response = await _client.GetAsync("public");
    response.EnsureSuccessStatusCode();

    string json = await response.Content.ReadAsStringAsync();
    var maps = JArray.Parse(json);

    var result = new List<(string id, string name)>();

    foreach (var m in maps)
    {
        string id = m["_id"]?.ToString() ?? "";
        string name = m["name"]?.ToString() ?? "Без назви";
        result.Add((id, name));
    }

        return result;
}
```

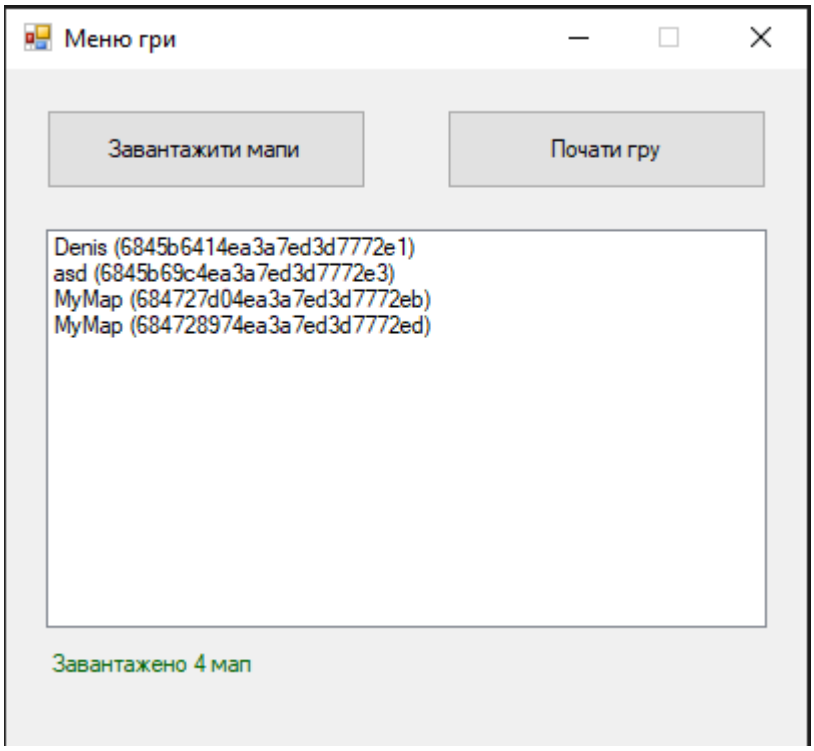

Рис. 4.4. Меню зі списком всіх карт

Важливо підкреслити, що застосований у проєкті підхід до взаємодії між клієнтською та серверною частинами програмного забезпечення дозволяє досягти високого рівня гнучкості та масштабованості. Завдяки чіткому розмежуванню обов'язків — коли логіка зберігання, обробки та передачі даних перебуває виключно на стороні сервера, а внутрішня ігрова логіка — на боці клієнта — досягається модульність системи, що є одним із найважливіших принципів сучасного програмування.

Такий підхід дає змогу розвивати гру незалежно від змін у структурі даних, які зберігаються у базі. Наприклад, у разі необхідності додати нові параметри до карти або змінити формат збереження mapSeed, достатньо внести зміни на рівні сервера або API, не зачіпаючи безпосередньо код самої гри. Це значно полегшує процес оновлення, знижує ризик виникнення помилок та дозволяє уникнути дублювання логіки у різних частинах проєкту.

Таким чином, подібна архітектурна організація системи дозволяє ефективно масштабувати проєкт, підтримувати його актуальність без необхідності постійного втручання в клієнтську логіку, а також забезпечує

зручну платформу для подальшого розширення функціональності як гри, так і редактора рівнів.

4.3.2. Опис гри та її концепція

Тепер, коли розглянуто механізм взаємодії з базою даних, варто перейти до опису безпосередньо гри, що була розроблена в межах цього проєкту.

Програмна реалізація гри поєднує класичні елементи пригодницького жанру з сучасними підходами до генерації, редагування та динамічного завантаження контенту. Основна ідея полягає у створенні та дослідженні унікальних карт — кожна з яких є сіткою розміром 5×5 кімнат. Ці кімнати користувач може вільно редагувати в інтерактивному редакторі: змінювати їхній тип, наповнювати ворогами, пастками або фінальними босами.

Кожна мапа представлена у вигляді координатної сітки, що формує унікальну структуру рівня. Відмінною особливістю є динамічне завантаження карт з сервера через API. Завдяки цьому користувач може в будь-який момент дослідити чужі мапи або створити власну, що робить ігровий процес практично безкінечним. Кількість доступних рівнів постійно зростає разом із активністю спільноти.

Типи кімнат умовно поділяються на:

- Звичайні кімнати — містять ворогів, пастки та інші перешкоди;
- Особливі кімнати — містять босів і є кульмінаційними точками рівня.

Ігровий процес побудовано за принципом поступового дослідження карти: гравець переміщується між кімнатами, долаючи ворогів і пастки, і врешті стикається з босом (рис. 4.5), що є фінальним випробуванням рівня.

Завдяки серверному зберіганню рівнів, а також можливості редагування та повторного використання мап, гра формує основу для нескінченного геймплею з високою реіграбельністю. Це робить її не лише цікавою з точки зору користувача, але й технічно масштабованою платформою для творчого самовираження.

Рис. 4.5. Битва з босом

Технічно гра реалізована на мові програмування C#, що забезпечує стабільну основу для ігрової логіки, фізики та графіки. Для візуалізації використовується бібліотека OpenTK — потужний інструмент на базі OpenGL, який дозволяє реалізувати якісну двовимірну графіку, анімацію персонажів та ефекти у реальному часі.

Структури даних у грі ретельно спроектовані для швидкого доступу та ефективної обробки - наприклад, кімнати представлені у вигляді об’єктів із координатами, а вороги і перешкоди - у вигляді окремих підоб’єктів, що можуть мати власні властивості та поведінку.

Ще одна перевага такої архітектури - легкість внесення змін у гру. Якщо потрібно додати новий тип ворога, нові перешкоди, або змінити розмір мапи - це можна зробити, просто змінивши структуру збереження у базі та логіку обробки у грі. Ніяких кардинальних переписувань коду не потрібно.

Також у перспективі можна додати соціальні функції: наприклад, рейтинг найкращих мап, обмін картами між гравцями, чи навіть спільне редагування рівнів у реальному часі.

Висновки до розділу 4

У четвертому розділі було проведено комплексний аналіз архітектури та реалізації системи створення ігрових рівнів, яка поєднує веб-редактор на базі React.js із ігровим рушієм на C#. Основна увага зосереджена на інтеграції компонентів, структурі даних, механізмах передачі та відтворення інформації між клієнтом і сервером.

Використання компонентного підходу в реалізації редактора продемонструвало високу ефективність: кожен модуль має чітке функціональне призначення, що дозволяє легко масштабувати та доповнювати систему новими типами ігрових об’єктів і функцій. State-менеджмент у поєднанні з JSON-структурами забезпечив високу продуктивність при роботі з великими рівнями, а також надійність у збереженні даних.

Важливим етапом стало впровадження seed-системи, яка дозволяє як точно відтворювати конкретні конфігурації рівнів, так і створювати їхні випадкові варіації. Це розширює творчі можливості платформи та підтримує процедурну генерацію.

Інтеграційний шар між редактором і рушієм реалізовано через HTTP API з використанням оптимізованої серіалізації, системи автентифікації, кешування та обробки помилок. Завдяки цьому забезпечено стабільну та швидку взаємодію обох частин системи. Ігровий рушій має власний парсер, який перетворює JSON-дані у внутрішні об’єкти, ініціалізуючи всі елементи рівня — від геометрії до логіки AI.

На технічному рівні архітектура системи враховує принцип розділення відповідальностей: редактор зосереджений на інтерфейсі й модифікації рівнів, тоді як рушій відповідає за відтворення поведінки ігрових об'єктів. Це дозволяє уникнути дублювання логіки та спрощує підтримку проєкту.

Особливу увагу приділено збереженню та версіюванню рівнів. Гнучкість JSON-формату дозволила реалізувати глибоку ієрархію об’єктів, а система відстеження змін — повертатися до попередніх версій, що важливо для

колаборативної розробки. MongoDB як сховище продемонструвала високу ефективність при роботі зі складними структурами даних.

Система підтримує крос-платформність: веб-редактор доступний із будь-якого пристрою, а рушій, написаний на .NET Core, сумісний з основними ОС. Це розширює потенційну аудиторію розробників та користувачів.

У питаннях безпеки реалізовано базові механізми автентифікації, шифрування та авторизації, що дозволяє контролювати доступ до контенту та захистити систему від типових загроз. Передбачена гнучка система прав доступу.

У перспективі система може бути розширена за рахунок:

- інструментів колаборативної розробки;
- інтеграції машинного навчання для генерації рівнів;
- реалізації сценаріїв зворотного зв'язку та аналітики поведінки користувачів.

Таким чином, отримані результати свідчать про ефективність запропонованої архітектури та обраного технічного стеку. Розроблена система є не лише завершеним продуктом, а й платформою, готовою до подальшого розвитку, яка відповідає сучасним вимогам індустрії та має потенціал для використання в комерційних і освітніх проєктах.

ВИСНОВКИ

В процесі реалізації цього проєкту було виконано комплексну роботу, що поєднала в собі різноманітні аспекти розробки сучасних програмних продуктів для ігрової індустрії. Від самого початку ідея полягала у створенні системи, яка б дозволяла не тільки генерувати унікальні ігрові світи, але й надавала змогу користувачам активно долучатися до процесу створення контенту через інтуїтивний та зручний редактор рівнів. Цей проєкт став чудовим прикладом того, як поєднати сучасні технології фронтенду, бекенду, баз даних і клієнтської логіки в одному масштабованому рішенні.

Важливо відзначити, що кожен етап розробки супроводжувався глибоким аналізом потреб користувача та технічних вимог. Особлива увага приділялася тому, щоб система була максимально гнучкою та відкритою для подальших змін і розширень. В результаті вдалося створити не просто гру, а цілісну екосистему, де гравець стає не лише споживачем контенту, але й його творцем. Це є надзвичайно важливим аспектом сучасних ігрових проектів, що орієнтовані на довготривалу взаємодію з аудиторією.

Одним із ключових факторів успіху проєкту стало правильне визначення технологічного стека, який забезпечив баланс між простотою реалізації та масштабованістю. Для створення редактора рівнів була обрана бібліотека React, яка давно зарекомендувала себе як надійний інструмент для побудови складних і динамічних інтерфейсів користувача. React дозволяє організувати код у вигляді компонентів, що є надзвичайно зручним з точки зору підтримки і розширення функціоналу. Такий підхід не лише полегшує роботу розробників, а й сприяє створенню швидкого та відзивчивого інтерфейсу, що є критично важливим для інтерактивного редактора.

Використання MongoDB для збереження даних також було цілком виправданим вибором. Ця БД, що базується на документній моделі, ідеально підходить для зберігання складних і багаторівневих структур, які притаманні ігровим мапам. Кожна мапа в нашому випадку - це фактично набір кімнат із

різними параметрами, ворогами, перешкодами та іншими елементами. Гнучкість MongoDB дала можливість без проблем додавати нові властивості до схеми без необхідності складних міграцій, що значно прискорило розробку.

Взаємодія фронтенда і бекенда через REST API стала невід'ємною частиною архітектури. Вона забезпечила чітку роздільність обов'язків між клієнтом і сервером, а також спростила масштабування системи в майбутньому. REST API надає можливість гнучко управляти даними, завантажувати списки мап, зберігати зміни, а також підтримує відкритість для потенційної інтеграції з іншими сервісами чи платформами.

Редактор рівнів є серцем проєкту, адже саме через нього користувачі можуть втілювати свої творчі ідеї в життя. Він був розроблений з орієнтацією на максимальну зручність, щоб навіть люди без глибоких технічних знань могли легко і швидко створювати цікаві ігрові світи. Використання сітки 5x5 для розміщення кімнат було обране, як баланс між складністю і простотою навігації - це дозволяє утворювати різноманітні комбінації розташувань, не перевантажуючи користувача надмірною кількістю опцій.

Логіка розміщення кімнат була детально продумана, аби запобігти виникненню некоректних структур - наприклад, кімнати повинні бути взаємопов'язаними, не утворювати "порожніх" місць, що можуть порушувати ігровий баланс. Також існує чітке розмежування типів кімнат: звичайні, з босами, кімнати з особливими властивостями. Це сприяє збереженню інтриги під час проходження гри, а також забезпечує різноманітність ігрового процесу.

Інтерфейс редактора поділений на логічні компоненти, кожен з яких відповідає за окремий аспект - управління мапою в цілому, редагування конкретної кімнати, розташування ворогів та об'єктів. Такий поділ покращує сприйняття і дозволяє гнучко масштабувати редактор у майбутньому, додаючи нові функції без порушення існуючого коду.

Наступним важливим етапом стала розробка клієнтської частини гри на C#, яка відповідала за візуалізацію, логіку ігрового процесу, а також інтеграцію із серверною базою даних. Підключення до API було організоване таким чином,

щоб забезпечити швидке і безпомилкове отримання даних про мапи і конвертацію їх у внутрішні об’єкти гри.

Це означає, що після отримання JSON-структури з бази даних, клієнт здійснює перетворення цієї інформації у кімнати з позиціями, ворогами, перешкодами тощо. Такий підхід є класичним прикладом патерну "Data Transfer Object", що дозволяє розділити структуру даних на шари і спрощує подальшу роботу з ними.

Особливістю реалізації було те, що кожна кімната в мапі не лише містить інформацію про свій тип, але й про вміст - ворогів, босів, перешкоди. Ігровий клієнт враховує це, створюючи об’єкти, які в майбутньому будуть взаємодіяти з гравцем в режимі реального часу.

Ще одним надзвичайно важливим аспектом була організація зберігання даних. Використання MongoDB з її можливістю зберігати складні документи дозволило відмовитись від традиційних таблиць з жорсткими зв’язками і дати користувачам повну свободу створення структурованих мап.

База містить кілька основних колекцій - користувачі, мапи, а також пов’язана інформація про складові мап. Така схема дозволяє ефективно виконувати запити, швидко завантажувати дані і не створює зайвих перешкод при розширенні функціоналу. Крім того, це забезпечує надійність і цілісність даних, що особливо важливо в багатокористувацьких проектах.

Розглядаючи майбутнє цього проєкту, можна впевнено стверджувати, що його потенціал є величезним. По-перше, існує безліч напрямків для розвитку редактора рівнів - додавання нових типів кімнат, ворогів, елементів декору, а також можливостей для більш глибокого налаштування ігрової логіки.

По-друге, з технічної точки зору, проєкт можна масштабувати - перейти на більш потужні сервери, впровадити кешування, оптимізувати роботу з базою даних. Все це дозволить підтримувати великий потік користувачів і забезпечити стабільну роботу системи навіть при високих навантаженнях.

Також перспектива введення багатокористувацьких функцій - наприклад, спільне редагування мап, змагання між гравцями або обмін створеним контентом - відкриває нові горизонти для залучення аудиторії і підвищення інтересу до гри.

Підсумовуючи, цей проєкт демонструє важливість комплексного підходу до розробки сучасних ігрових систем. Кожен етап, від вибору технологій, через розробку редактора до створення ігрового клієнта, був спрямований на досягнення максимальної гнучкості, зручності та якості. Результатом стала не просто гра, а платформа, яка може розвиватися і трансформуватися відповідно до потреб користувачів і ринку.

Ця робота є яскравим прикладом того, як сучасні технології можуть об'єднуватися для створення продуктів, що дарують користувачам можливість творчості та самовираження, одночасно забезпечуючи стабільність і надійність роботи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Gregory J. Game Engine Architecture. - 3rd Edition. - A K Peters/CRC Press, 2018. - 800 с.
2. Rogers S. Level Up! The Guide to Great Video Game Design. - Wiley, 2010. - 400 с.
3. MongoDB Manual [Електронний ресурс]. - Режим доступу: https://docs.mongodb.com/manual/
4. Nystrom R. Game Programming Patterns. - Genever Benning, 2014. - 350 с.
5. Smith J. Procedural Generation in Game Design. – GameDev Publishing, 2019. - 120 с.
6. Roberts K. Procedural Generation in Indie Games: Case Studies of TBOI, Dead Cells, and Hollow Knight. – IndieGame Research, 2022. – 110 с.
7. Smith J., Game Design Algorithms: Procedural Level Generation. – New York: GameDev Press, 2019. – 250 с.
8. Adams E., Fundamentals of Game Design. – Pearson, 2014. – 320 с.

ДОДАТКИ

Додаток А

Реалізація конструктора та підключення до бази React

```
import React, { useState } from "react";
import "./EditorMap.css";
import { Room, RoomType } from "../models/MapModel";
import RoomEditor from "./RoomEditor";
import Header from "./Header";
import { useAuth } from "../context/AuthContext";
const GRID_SIZE = 5;
const EditorMap: React.FC = () => {
  const { token } = useAuth();
  const [rooms, setRooms] = useState<Room[]>([]);
  const [selectedRoom, setSelectedRoom] = useState<Room | null>(null);
  const [roomPlacementType, setRoomPlacementType] =
useState<RoomType>("normal");
  const [mapName, setMapName] = useState("");
  const [saveStatus, setSaveStatus] = useState<"idle" | "saving" | "success" |
"error">("idle");
  const getNeighbors = (x: number, y: number) => {
    return rooms.filter(
      (r) =>
        (Math.abs(r.x - x) === 1 && r.y === y) ||
        (Math.abs(r.y - y) === 1 && r.x === x)
    );
  };

  const canPlaceRoom = (x: number, y: number): boolean => {
    if (rooms.find((r) => r.x === x && r.y === y)) return false;
    const neighbors = getNeighbors(x, y);
    if (rooms.length === 0) return true;
```

```
  const bossRoom = rooms.find((r) => r.type === "boss");
  if (roomPlacementType === "boss") return neighbors.length === 1;
  if (bossRoom) {
    const bossNeighbors = getNeighbors(bossRoom.x, bossRoom.y);
    const isNextToBoss =
      (Math.abs(bossRoom.x - x) === 1 && bossRoom.y === y) ||
      (Math.abs(bossRoom.y - y) === 1 && bossRoom.x === x);

    if (isNextToBoss && bossNeighbors.length >= 1) return false;
  }

  return neighbors.length >= 1;
};

const handleGridClick = (x: number, y: number, event?: React.MouseEvent) => {
  if (selectedRoom) return;

  const existing = rooms.find((r) => r.x === x && r.y === y);

  if (existing && event?.ctrlKey) {
    setRooms((prev) => prev.filter((r) => r.id !== existing.id));
    return;
  }

  if (existing) {
   setSelectedRoom(existing);
    return;
  }
  const bossRoom = rooms.find((r) => r.type === "boss");
```

```
const normalRooms = rooms.filter((r) => r.type === "normal");

if (roomPlacementType === "boss" && bossRoom) {
  alert("Можна лише одну кімнату боса");
  return;
}

if (roomPlacementType === "normal" && normalRooms.length >= 6) {
  alert("Максимум 6 звичайних кімнат");
  return;
}

if (!canPlaceRoom(x, y)) {
  alert("Недопустиме розташування кімнати");
  return;
}

const newRoom: Room = {
  id: `${x}-${y}`,
  x,
  y,
  type: roomPlacementType,
  content: {
    obstacles: [],
    enemies: [],
    boss: null,
  },
};
```

```
  setRooms([...rooms, newRoom]);
};

const saveMap = async () => {
  if (!token) {
    alert("Увійдіть, щоб зберегти мапу");
    return;
  }

  if (!mapName.trim()) {
    alert("Введіть назву мапи");
    return;
  }

  setSaveStatus("saving");
  try {
    const res = await fetch("http://localhost:5000/api/maps/save", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${token}`,
      },
      body: JSON.stringify({
        name: mapName,
        mapSeed: rooms,
      }),
    });
    if (!res.ok) throw new Error("Помилка при збереженні");
    setSaveStatus("success");
```

```
    alert("Мапа збережена!");
   } catch (err) {
    setSaveStatus("error");
    alert("Не вдалося зберегти мапу");
   }
  };

  return (
   <div className="editor-container">
    {/* Хедер тепер просто без пропсів */}
    <Header />

    <div className="grid-wrapper">
     <div className="grid">
      {Array.from({ length: GRID_SIZE }).map((_, row) => (
       <div className="row" key={row}>
        {Array.from({ length: GRID_SIZE }).map((_, col) => {
         const room = rooms.find((r) => r.x === col && r.y === row);
         const canPlace = canPlaceRoom(col, row);
         return (
          <div
           key={`${col}-${row}`}
           className={`cell
            ${room ? (room.type === "boss" ? "boss" : "normal") : ""}
            ${!room && !canPlace ? "blocked" : ""}
           `}
           onClick={(e) => handleGridClick(col, row, e)}
           onDoubleClick={() => !selectedRoom && room &&
setSelectedRoom(room)}
```

```
          title={
           room
             ? "Подвійний клік - редагувати, Ctrl+клік - видалити"
             : canPlace
             ? "Клік - створити кімнату"
             : "Сюди не можна ставити кімнату"
          }
        >
          {room && (
            <img
              className="room-icon"
              src={
                room.type === "boss"
                  ? "/icons/room-boss.png"
                  : "/icons/room-normal.png"
              }
              alt={room.type}
            />
          )}
        </div>
      );
    })}
  </div>
))}
</div>
<div className="room-type-selector">
  <button
    className={roomPlacementType === "normal" ? "active" : ""}
    onClick={() => setRoomPlacementType("normal")}
```

```
    >
      Звичайна кімната
    </button>
    <button
      className={roomPlacementType === "boss" ? "active" : ""}
      onClick={() => setRoomPlacementType("boss")}
    >
      Кімната боса
    </button>
  </div>
</div>

<div className="save-controls">
  <input
    type="text"
    placeholder="Назва мапи"
    value={mapName}
    onChange={(e) => setMapName(e.target.value)}
  />
  <button onClick={saveMap} disabled={saveStatus === "saving"}>
    {saveStatus === "saving" ? "Збереження..." : "Зберегти мапу"}
  </button>
</div>
{selectedRoom && (
  <RoomEditor
    room={selectedRoom}
    onClose={() => setSelectedRoom(null)}
    onSave={(updated) => {
      setRooms((prev) =>
```

Продовження Додатку А

```
            prev.map((r) => (r.id === updated.id ? updated : r))
          );
          setSelectedRoom(null);
        }}
      />
    )}
  </div>
 );
};

export default EditorMap;
```

Додаток Б

Підключення гри до БД

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Threading.Tasks;
using Newtonsoft.Json.Linq;
using OpenTK.Mathematics;
using System.Linq;
namespace MyGame
{
    public class MapWebLoader
    {
        private readonly HttpClient _client;
        public MapWebLoader()
        {
            _client = new HttpClient
            {
                BaseAddress = new Uri("http://localhost:5000/api/maps/")
            };
        }
        // Отримати всі мапи без авторизації
        public async Task<List<(string id, string name)>> GetAllMapsAsync()
        {
            HttpResponseMessage response = await _client.GetAsync("public");
            if (!response.IsSuccessStatusCode)
                throw new Exception("Не вдалося отримати список мап");
            string json = await response.Content.ReadAsStringAsync();
            var maps = JArray.Parse(json);
            return maps.Select(m => (
```

```
        m["_id"]?.ToString() ?? "",
        m["name"]?.ToString() ?? "Без назви"
    )).ToList();
}

// Завантажити мапу за ID (через повторний запит до public)
public async Task<Map> LoadMapByIdAsync(string id)
{
    HttpResponseMessage response = await _client.GetAsync("public");
    if (!response.IsSuccessStatusCode)
        throw new Exception("Не вдалося отримати мапи");
    string json = await response.Content.ReadAsStringAsync();
    var maps = JArray.Parse(json);
    var selected = maps.FirstOrDefault(m => m["_id"]?.ToString() == id);
    if (selected == null)
        throw new Exception("Мапа не знайдена");

    var mapSeed = selected["mapSeed"].ToObject<List<RoomDTO>>();
    return ConvertToMap(mapSeed);
}
private Map ConvertToMap(List<RoomDTO> seed)
{
    var map = new Map();

    foreach (var dto in seed)
    {
        var room = new Room(dto.type == "boss");
        room.Position = new Vector2i(dto.x, dto.y);
```

```
            foreach (var obs in dto.content.obstacles ?? new List<List<float>>())
            {
                if (obs.Count >= 2)
                    room.AddObstacle(new Vector2(obs[0], obs[1]));
            }

            foreach (var e in dto.content.enemies ?? new List<EnemyDTO>())
            {
                var type = e.type == "melee" ? EnemyType.Melee : EnemyType.Ranged;
                var pos = GetCenteredRoomPos(dto.x, dto.y);
                room.AddEnemy(new Enemy(pos, type, false, e.health, e.damage,
e.speed));
            }
            if (dto.content.boss != null)
            {
                var bossPos = GetCenteredRoomPos(dto.x, dto.y);
                room.AddEnemy(new Enemy(bossPos, EnemyType.Melee, true,
dto.content.boss.health, dto.content.boss.damage, 0.8f));
            }

            map.AddRoom(room, room.Position);
        }

        foreach (var a in map.Rooms.Keys)
        {
            foreach (var b in map.Rooms.Keys)
            {
                if ((Math.Abs(a.X - b.X) + Math.Abs(a.Y - b.Y)) == 1)
                    map.AddConnection(a, b);
```

```
            }
        }

        return map;
    }

    private Vector2 GetCenteredRoomPos(int x, int y)
    {
        return new Vector2(x * 2.6f - 3.25f, y * 1.4f - 1.75f);
    }
}

// DTO-класи (як були раніше)
public class RoomDTO
{
    public string id { get; set; }
    public int x { get; set; }
    public int y { get; set; }
    public string type { get; set; }
    public RoomContentDTO content { get; set; }
}

public class RoomContentDTO
{
    public List<List<float>> obstacles { get; set; }
    public List<EnemyDTO> enemies { get; set; }
    public BossDTO boss { get; set; }
}
```

Продовження Додатку Б

```
    public class EnemyDTO
    {
        public string type { get; set; }
        public int health { get; set; }
        public int damage { get; set; }
        public float speed { get; set; }
    }

    public class BossDTO
    {
        public int health { get; set; }
        public int damage { get; set; }
    }
}
```