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

Гушка Олексія Сергійовича
(прізвище, ім'я, по батькові здобувача)

на тему Розробка гри «Морський бій»
(повна назва теми)

за матеріалами праць провідних спеціалістів з розробки ПЗ та проектування БД
(повна назва бази дослідження)

науковий керівник к.т.н. ________ Медведєв Д.Г.
(наук. ступінь, вчене звання) *(підпис)* *(прізвище, ініціали)*

Робота допущена до захисту в ЕК
Протокол засідання кафедри
від 11 червня 2025 року № 12
Завідувач кафедри ________________
(підпис)

д.т.н., професор Зеленський О.С.
Наук. ступень, вчене звання *Ініціали, прізвище*

Кривий Ріг – 2025

ЗГОДА здобувача вищої освіти

Державного університету економіки і технологій про перевірку кваліфікаційної роботи на прояви академічного плагіату та розміщення в Репозитарії Університету

Я, Гушко Олексій Сергійович, підтримую політику Державного університету економіки і технологій з академічної доброчесності і відкритого доступу.

Засвідчую, що кваліфікаційна бакалаврська робота Розробка гри «Морський бій» виконана самостійно та не містить академічного плагіату. Я не надавав і не одержував недозволену допомогу під час підготовки цієї роботи. Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Із чинним Положенням про запобігання та виявлення академічного плагіату в роботах здобувачів вищої освіти Державного університету економіки і технологій ознайомлений. Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі порушення норм академічної доброчесності робота не допускається до захисту або оцінюється незадовільно.

Також я поінформований, що відповідно до «Положення про Репозитарій (електронну базу даних) Державного університету економіки і технологій» зазначена робота буде розміщена в Електронному архіві Університету (Репозитарії ДУЕТ). З умовами такого розміщення ознайомлений.

Дата   підпис   ініціали, прізвище (власноруч)

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 «Морський бій»

Керівник роботи к.т.н. Медведєв Д. Г.

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. Проектування задачі

Розділ 3. Проектування бази даних

Розділ 4. Розробка програмного забезпечення

Об'єкт дослідження: морський бій

Предмет дослідження: алгоритми гри

Мета кваліфікаційної роботи: *розробка програмного забезпечення гри*

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 ______________ __Медведєв Д. Г.___
(підпис) (прізвище та ініціали)

Завдання одержав ______________ __Гушко О.С._____.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ
на кваліфікаційну бакалаврську роботу

«Розробка гри “Морський бій”»

Гушка Олексія Сергійовича

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 дипломній роботі розроблене програмне забезпечення гри «Морський Бій». Програмний додаток розроблено на мові C# з використанням бібліотеки OpenGL на основі OpenTK для роботи з тривимірною графікою та технології ADO .NET для роботи з базами даних. Сторінка турнірної таблиці розроблена на основі HTML, CSS та JavaScript. Взаємодія з сервером здійснюється через REST API з використанням асинхронних викликів.

Ключові слова: МОРСЬКИЙ БІЙ, КОРАБЕЛЬ, ТУРНІРНА ТАБЛИЦЯ, АЛГОРИТМИ, СУБД, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД(база даних)	Впорядкований набір логічно взаємопов’язаних даних, що використовуються спільно та призначені для задоволення інформаційних потреб користувачів.
СУБД	Система управління базами даних.
ПЗ	Програмне забезпечення.
ADO .NET	ActiveX Data Object для .NET – технологія доступу і управління базами даних для платформи .NET

ЗМІСТ

ВСТУП 8

РОЗДІЛ І ПОСТАНОВКА ЗАДАЧІ 10

1.1. Характеристика задачі 10

1.2. Огляд існуючих аналогів гри 12

1.3. Аналіз вимог до гри 23

РОЗДІЛ ІІ ПРОЕКТУВАННЯ ЗАДАЧІ 26

РОЗДІЛ ІІІ ПРОЕКТУВАННЯ БАЗИ ДАННИХ ДЛЯ ПРОЕКТУ 33

3.1. Вибір системи управління базами даних 33

3.2. Структури таблиць бази даних 38

РОЗДІЛ IV РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 44

4.1. Інтерфейс та алгоритм гри 44

4.2 Інтерфейс та алгоритм розробки сайту 55

ВИСНОВКИ 58

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 59

ДОДАТКИ 60

ВСТУП

У сучасному інформаційному середовищі стрімко зростає попит на інтерактивні програмні продукти, в тому числі ігрові. Вони поєднують ігрову функціональність з елементами мережевої взаємодії. Ігрова індустрія популярна через свою стабільність, різноманітність, не прив'язаність до якоїсь конкретної платформи та великого асортименту візуальних інструментів для розробки графічної частини програми. Разом з тим, кожна гра потребує впровадження веб-платформи. Ігри можуть потребувати наявності сторінки для реклами, завантажувача, турнірної таблиці або форма підписки. Доповнення гри сайтом дозволяє розширити програми функціями онлайн-взаємодії, та допомагає під час просування додатку.

Об’єктом дослідження виступає процес розробки клієнт-серверної системи, яка включає гру морського бою та веб-сервіс для турнірної таблиці.

Предметом дослідження є технології реалізації гри у середовищі C# з використанням OpenGL та розробка веб-платформи для обліку результатів на основі C# і JavaScript.

Мета роботи полягає у створенні повноцінного прикладного програмного комплексу, що складається з гри «Морський бій» і веб-системи для збереження та відображення турнірних результатів.

Для досягнення мети визначено такі завдання:

- реалізувати графічний інтерфейс гри у двовимірному просторі з використанням OpenGL;
- створити логіку ігрового процесу: розміщення кораблів, обробка ходів, виявлення переможця;
- розробити серверну частину для обліку результатів матчів;
- створити клієнтську частину веб-інтерфейсу для перегляду статистики;
- забезпечити збереження та обробку даних у базі;
- реалізувати обмін даними між клієнтом і сервером.

Проблематика проекту пов’язана з інтеграцією різних технологій у межах одного комплексу: забезпечення коректної взаємодії між десктопною грою та вебкомпонентом, налаштування зберігання та оновлення статистики у режимі реального часу, налаштування зручного та доступного інтерфейсу, а також підтримка зручності користування частинами системи.

Актуальність зумовлена потребою у практичному поєднанні ігрового додатку та засобів обліку результатів у цифровому середовищі, що дозволяє моделювати реальні турнірні сценарії в інтерактивному форматі.

Для реалізації проекту використано мову програмування C#, бібліотеку OpenGL для розробки графічної частини, JavaScript для побудови клієнтської частини вебінтерфейсу, а також технології ASP.NET для створення серверної логіки та обробки запитів.

РОЗДІЛ І
ПОСТАНОВКА ЗАДАЧІ

1.1. Характеристика задачі

Поставлене завдання являє собою розробку програмного комплексу, яке поєднує настільну гру з динамічною візуалізацією та вебресурс для обліку результатів. Проект складається з двох взаємопов'язаних частин: десктопного застосунку, який відтворює геймплей «Морського бою», та вебсервісу, що збирає, зберігає і відображає статистику гравців. Важливо, що ці складові не існують окремо - вони взаємодіють між собою через загальну інфраструктуру даних, формуючи цілісну систему.

Робота починається з проектування самої гри Windows-додатку, побудованого на класичних правилах настільної версії. Завдяки знайомій структурі бою процес моделювання спрощується: двовимірне поле, послідовні ходи, об'єкти кораблів. Проте візуалізація - це не лише малюнок. Вона має реагувати на дії гравця, наочно передавати стан партії та не створювати затримок. Застосування OpenGL дозволяє ефективно реалізувати рендеринг - кожен постріл, попадання чи знищення судна миттєво відображається на екрані, реагуючі на кліки. Основу графічної сцени формує координатна сітка, яка визначає розміщення об'єктів і межі поля.

Водночас графіка - лише оболонка для внутрішньої логіки. У центрі роботи гри лежить алгоритм, який перевіряє правильність розстановки кораблів, керує черговістю ходів, обробляє події та завершує гру при досягненні умов перемоги. Ця логіка реалізується засобами C#, що дозволяє поєднати структури даних, подієвий механізм взаємодії та обчислення без зайвої складності.

Після завершення партії гра формує результат переможця, програвшого, кількість зроблених ходів за матч. Ці дані записуються у базу даних. З бази даних сервер забирає інформацію про партію, та передає її на сайт з турнірною

таблицею. На сервері працює API, створене на основі ASP.NET. Воно відправляє запити до сайту, перевіряє їхню структуру і зчитує записи в базі даних. Сервер не втручається у логіку гри — він лише забирає результат гри з бд, що дозволяє зберігати універсальність і простоту системи.

Вебінтерфейс - це клієнтська частина, написана з використанням JavaScript. На головній сторінці виводиться таблиця гравців, яка автоматично оновлюється при появі нових даних. Весь процес, від завантаження до сортування, відбувається без перезавантаження сторінки, завдяки fetch-технології взаємодії фронтенда з бекендом. Структура таблиці дозволяє швидко орієнтуватися у результатах навіть за великої кількості учасників. Повинна бути передбачена система сортування результатів та пагінація. Якщо усі результати відображати одразу на одній сторінці, буде перенасичення сторінки та сторінка може після такого довго завантажуватись.

Передбачено також обробку нестандартних ситуацій. Якщо дані з гри надійшли неповними або з помилками, система їх не зберігає, а повертає повідомлення про помилку. Всі дії мають отримувати підтвердження, яке гра зчитує у відповідь. Такий обмін забезпечує синхронізацію і виключає ситуації, коли один і той самий матч потрапляє до таблиці двічі.

Через демонстраційний характер проекту свідомо обмежено функціональність: поки що не реалізовано онлайн-гру між кількома гравцями або вхід через облікові записи. Всі дані вводяться вручну, а зберігаються тільки завершені партії. Проте навіть у такому форматі система повністю передає задум - сформувати послідовний цикл: спочатку гра, далі збереження даних у бд, передача сервером даних на сайт, малювання турнірної таблиці, перегляд статистики на сайті.

1.2. Огляд існуючих аналогів гри

Наразі не складно знайти велику кількість інтерпретацій гри Морського бою. Існують продвинуті версії, з ускладненими правилами, ігри з класичними правилами, онлайн ігри та ігри розраховані на 2 гравців.

У розрізі даного розділу пропоную розглянути два аналога гри морський бій. Мобільна гра “Sea Battle” від компанії Byril найпопулярніша мобільна гра, з хорошими оцінками та позитивними відгуками. Перше на що слід звернути увагу - інтерфейс гри. Він імітує стиль настільної гри - кораблі намальовані ручкою, поле - це клаптик паперу з зошиту в клітинку, перехід між екранами має анімацію перегортання листка.

У грі є 3 доступні режими (Рис. 1.1):

- з ботом;
- онлайн;
- з другом.

Рис. 1.1. Вибір режиму гри у Sea Battle

Після вибору режиму починається етап розстановки кораблів (Рис. 1.2). Асортимент кораблів класичний:

- один чотирьохпалубний корабель;
- два трьохпалубних корабля;
- три двухпалубних корабля;

- чотири однопалубних корабля.

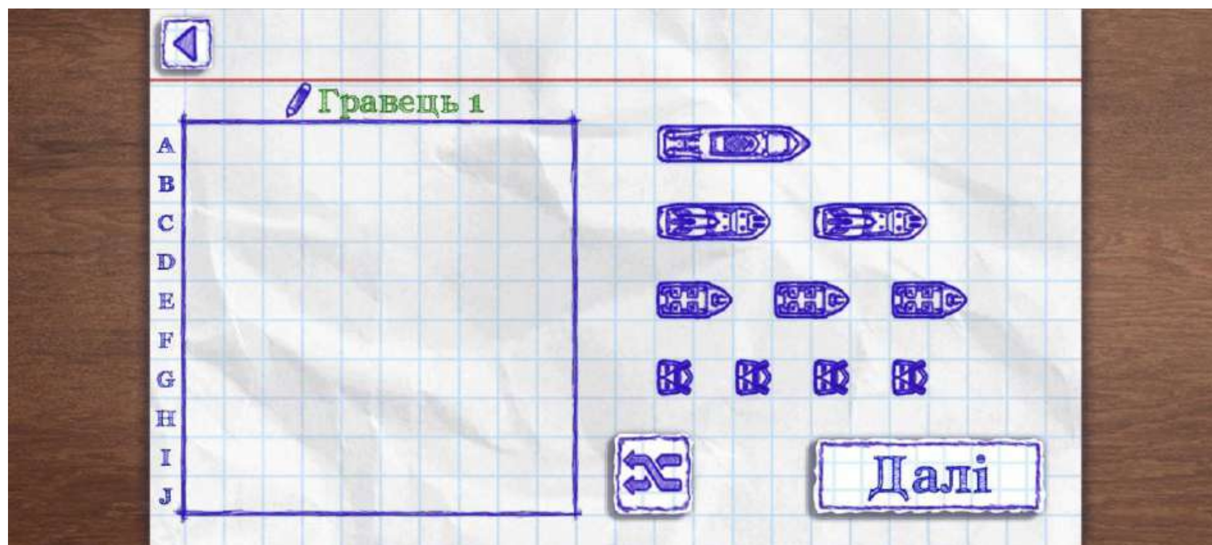

Рис. 1.2. Кораблі доступні до розстановки у Sea Battle

Кораблі розставляються звичайним перетягуванням корабля на поле. При спробі перейти до наступного етапу, не поставивши усі кораблі на поле, алгоритм не пропускає гравця. Кораблі мигають, з натяком на те, що гравець спершу вимушений їх розставити по полю. Якщо гравець намагається поставити корабель в заборонене місце, клітинку навколо вже поставленого корабля або на сам корабель, клітинки підсвічується червоним кольором, а сам корабль поставити в клітинку не можливо - він повертається в початкову позицію (Рис. 1.3).

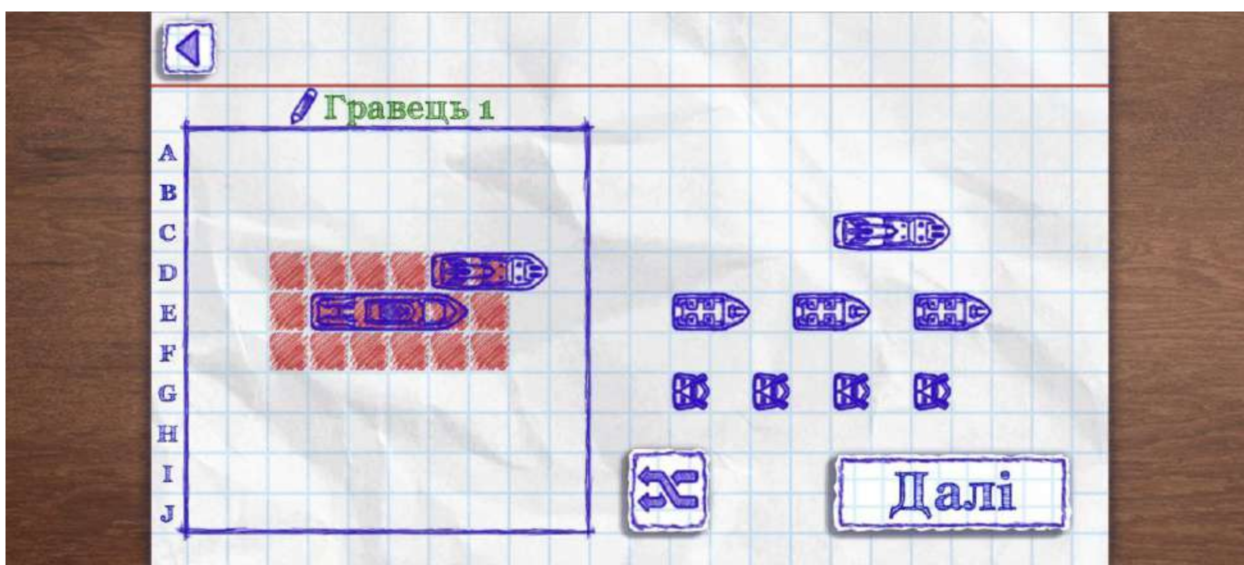

Рис. 1.3. Реакція системи на не вірну розстановку у Sea Battle

Також на цьому етапі передбачена випадкова розстановка кораблів. В такому випадку користувач не сам розставляє кораблі, а цю задачу виконує алгоритм. При цьому враховуються усі правила розстановки кораблів у грі морський бій (Рис. 1.4).

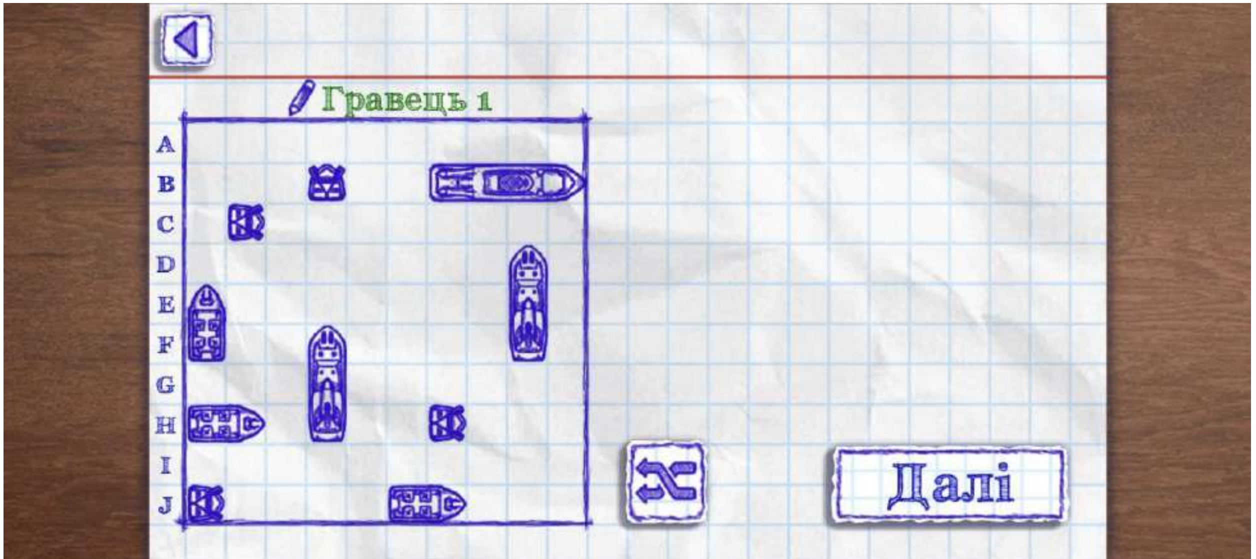

Рис. 1.4. Автоматична розстановка кораблів у Sea Battle

Ця гра відійшла від класичної версії гри, тому наступний етап розстановки стратегічний. Гравець може поставити міні, закупити літаки або радари перед початком гри (Рис. 1.5).

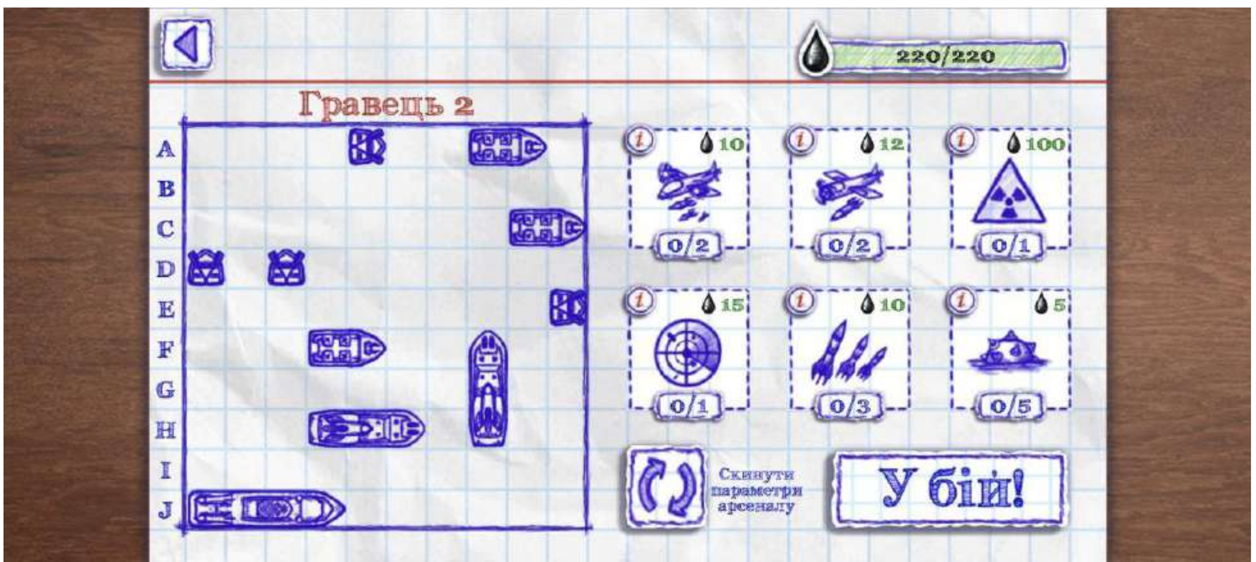

Рис. 1.5. Етап додаткової закупки у Sea Battle

Після того як усі гравці прошли етап планування гри, починається безпосередньо саима гра. Відображаються 2 поля Гравця 1 та гравця 2, між

ними малюється стрілка, яка вказує чий зараз хід. Також є вкладка арсеналу, де відображається закуплені перед грою додаткові предмети (Рис. 1.6).

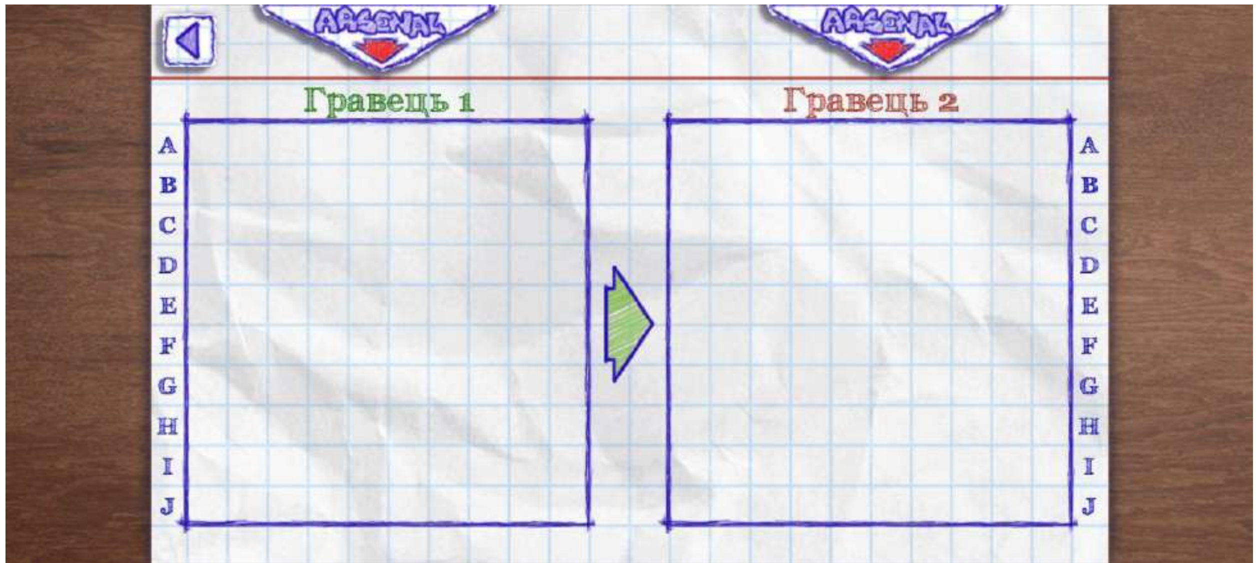

Рис. 1.6. Початок гри у Sea Battle

Щоб зробити хід необхідно, щоб гравець на протилежному полі обрав клітинку. Якщо хід являється промахом, наступає хід протилежного гравця (Рис. 1.7).

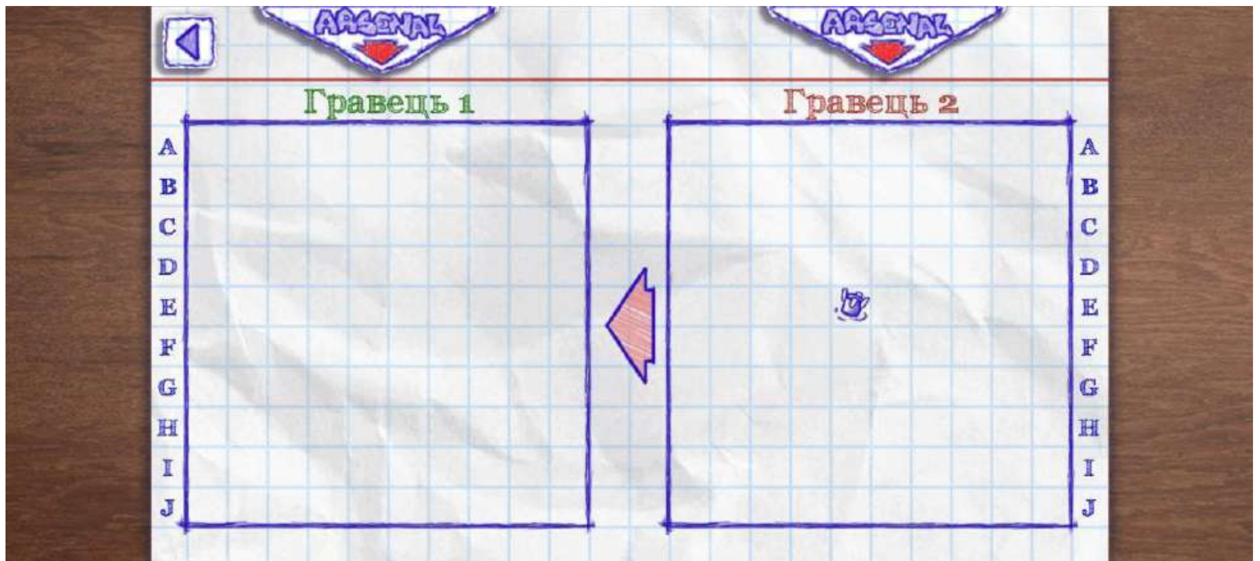

Рис. 1.7. Промах у Sea Battle

Попадання відображається хрестиком, і гравець який влучив по частині корабля має право зробити додатковий хід (Рис. 1.8).

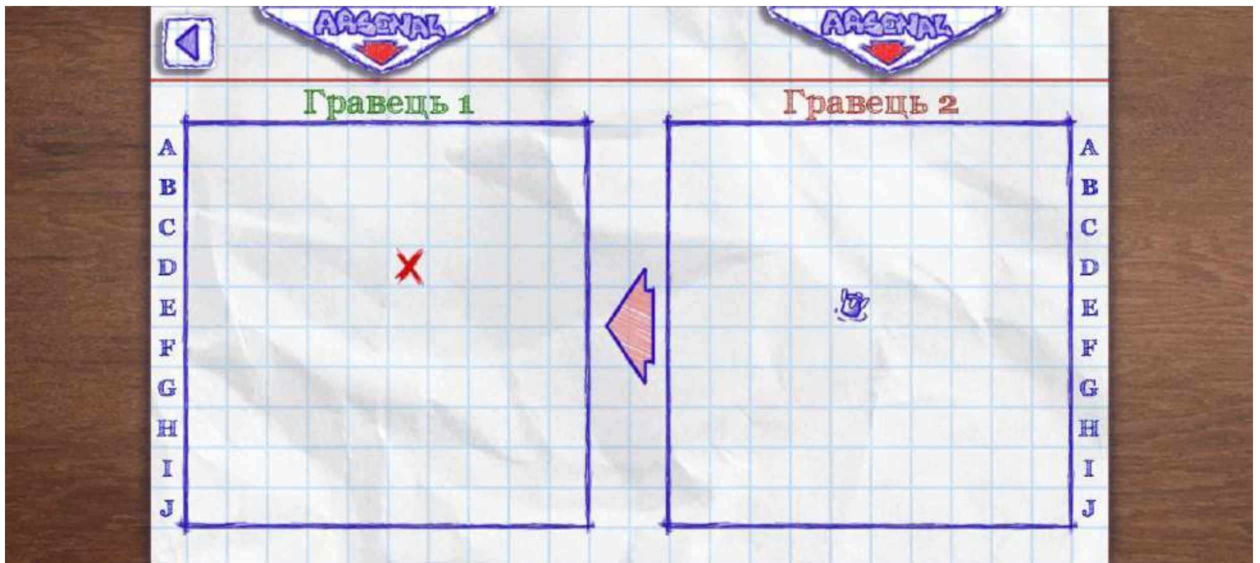

Рис. 1.8. Попадання у Sea Battle

Коли увесь корабель потоплено, клітинки навколо нього теж блокуються, і хід зробити в них неможливо (Рис. 1.9).

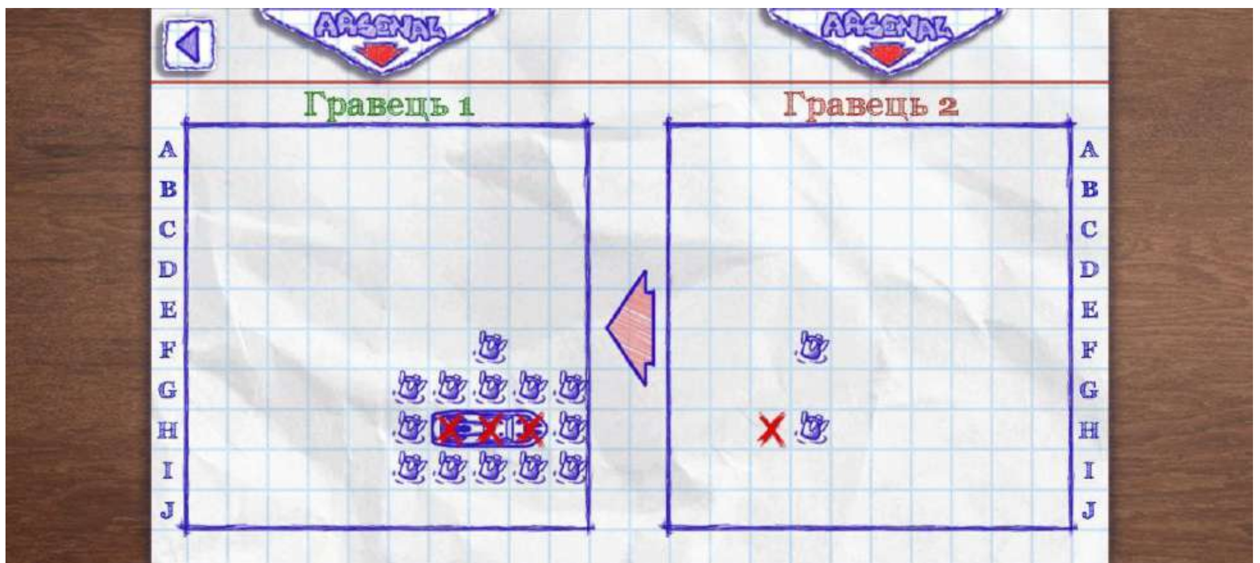

Рис. 1.9. Потопленний корабель Sea Battle

Коли один гравець топить усі кораблі противника, гра закінчується. Не затоплені кораблі противника показуються, а користувач який виграв отримує очки рівня (Рис. 1.10).

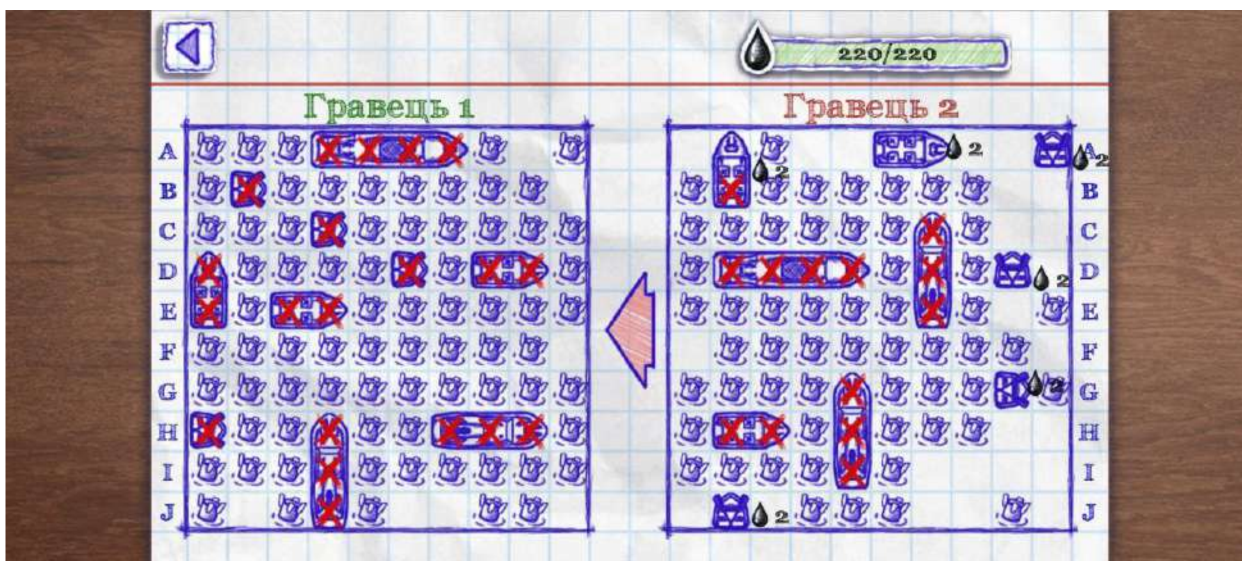

Рис. 1.10. Кінець гри у Sea Battle

Після гри відображається інформаційне вікно, де вказується рахунок між гравцями, вказується хто програв, хто виграв, а також пропонується повторити гру або ж завершити (Рис. 1.11).

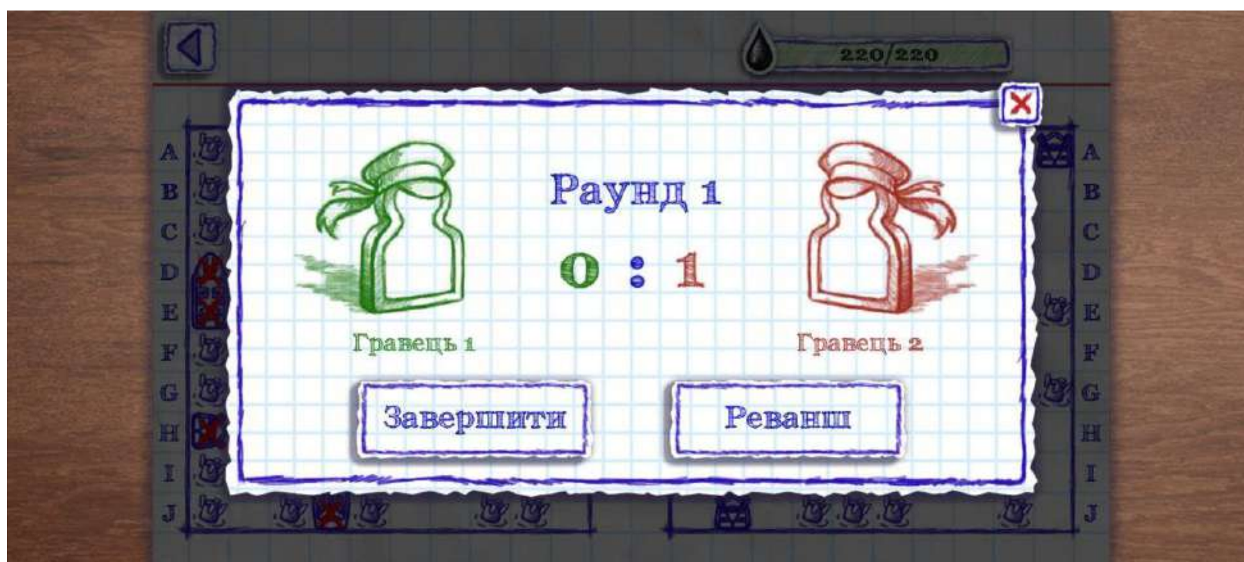

Рис. 1.11. Інформаційне вікно з результатами гри у Sea Battle

Також у додатку доступна до перегляду статистика гравця. В ній розраховується кількість матчів, програшів, виграшів, скільки потоплено кораблів. Доступна турнірна таблиця, яка також надає статистику ігор серед різних режимів гри (Рис. 1.12).

Рис. 1.12. Статистика гравця у Sea Battle

Застосунок працює без багів, має приємний дизайн, передбачено декілька режимів гри, розроблена система рангів, є профіль гравця, доступна різна локалізація, а також розширений геймплей, в якому пропонується відійти від класичної версії гри. Також особливим плюсом виступає наявність статистики профілю, яку можна взяти за основу для розробки турнірної таблиці на вебсайті. Застосунок є хорошим прикладом для інтерпретації гри Морського бою.

Онлайн веб гра Sea Battle, яка доступна по посиланню https://cardgames.io/seabattle/, набагато спрощена, від попереднього прикладу. На старті у гравця вже у випадковому порядку розставлені кораблі. За бажанням гравець може сам змінити їх позиції, або натиснути кнопку рестарту для того щоб отримати новий варіант розстановки. Цей підхід не такий зручний, оскільки, не всі клітинки доступні для розстановки. Тому користувач вимушений або довше переставляти кораблі за власним бажанням, або залишити випадкову розстановку, внісши мінімальні зміни (Рис. 1.13).

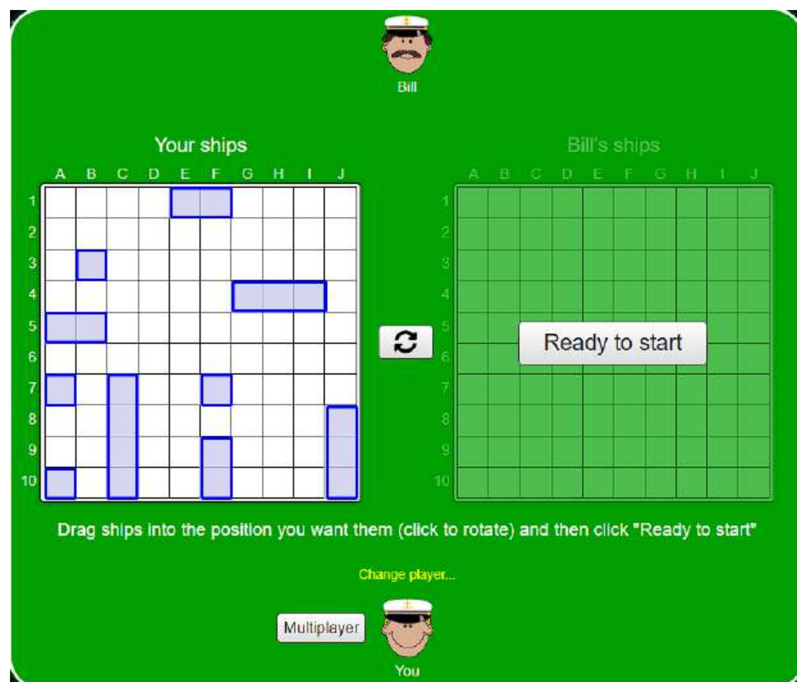

Рис. 1.13. Розстановка кораблів у веб грі Морський бій

Як можемо помітити, графіка гри набагато простіша. Кораблі малюються у вигляді звичайних прямокутників синього кольору. Поле гри - звичайна сітка, намальована на білому квадраті. Якщо користувач намагається встановити корабель в забороненому місці - увесь корабель змінює колір з синього на червоний (Рис. 1.14).

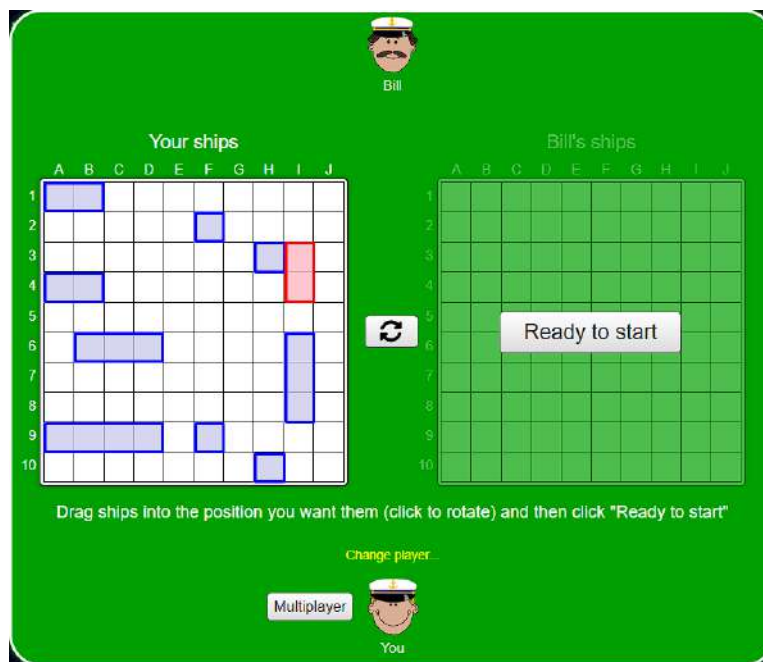

Рис. 1.14. Реакція системи на не вірну розстановку корабля у веб грі Морський бій

Промах позначається звичайною синьою точкою. Цей підхід також виявився мінусом, оскільки з часом, коли полів для вибору стає менше, складніше виявити клітинку, в яку гравець ще не стріляв (Рис. 1.15).

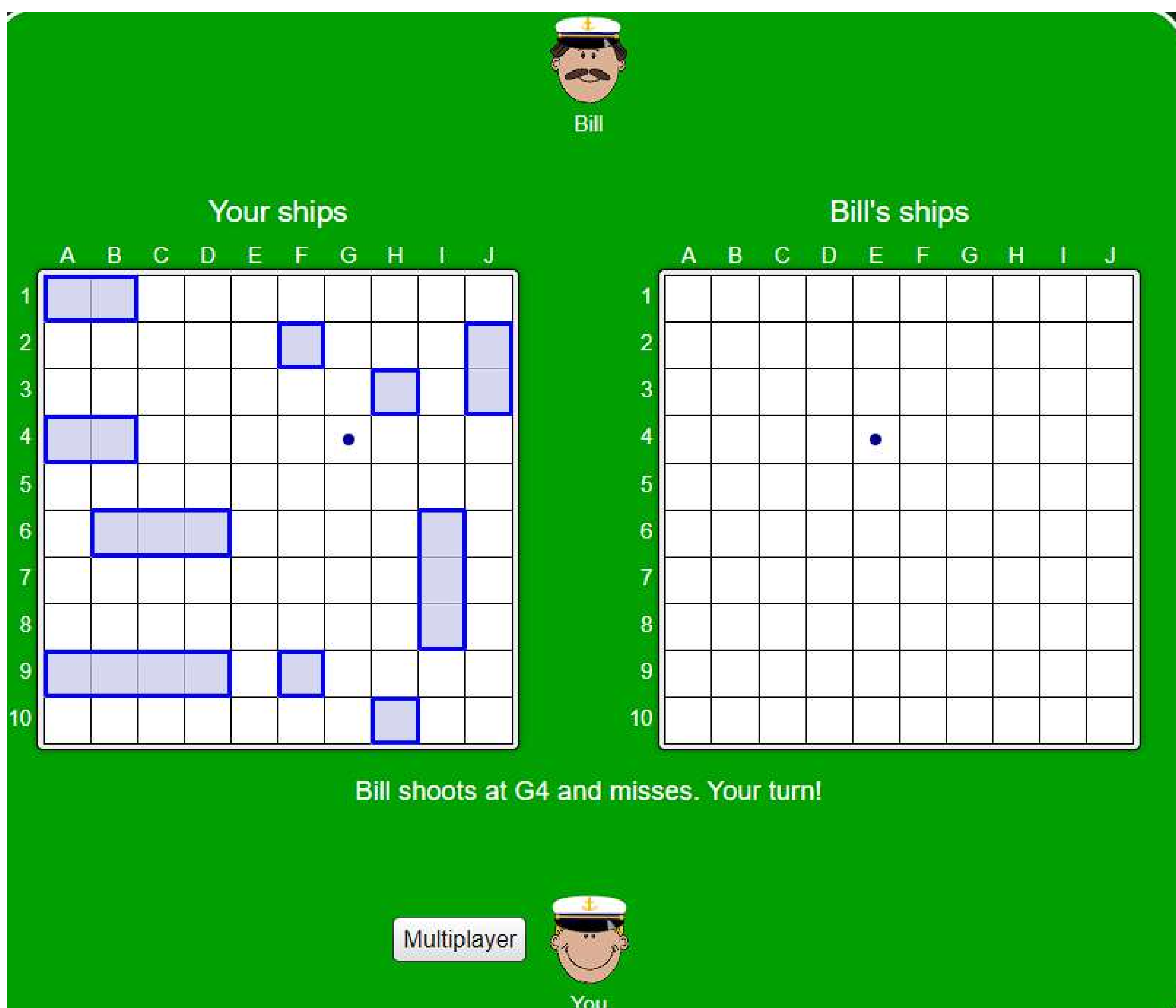

Рис. 1.15. Графічне зображення промаху у веб грі Морський бій

Попадання зображується червоним хрестом (Рис. 1.16). На відміну від промаху, хрест великий, чіткий, що зручно. Увага гравця не розфокусується, при цьому легко порахувати, скільки попадань вже є. Під полями для гри також є текст який пояснює, що наразі відбувається - чий хід, промахнувся гравець, чи потрапив, або можливо невірно обрав клітинку для ходу. Основні правила гри дотримані - після попадання гравець має додатковий хід.

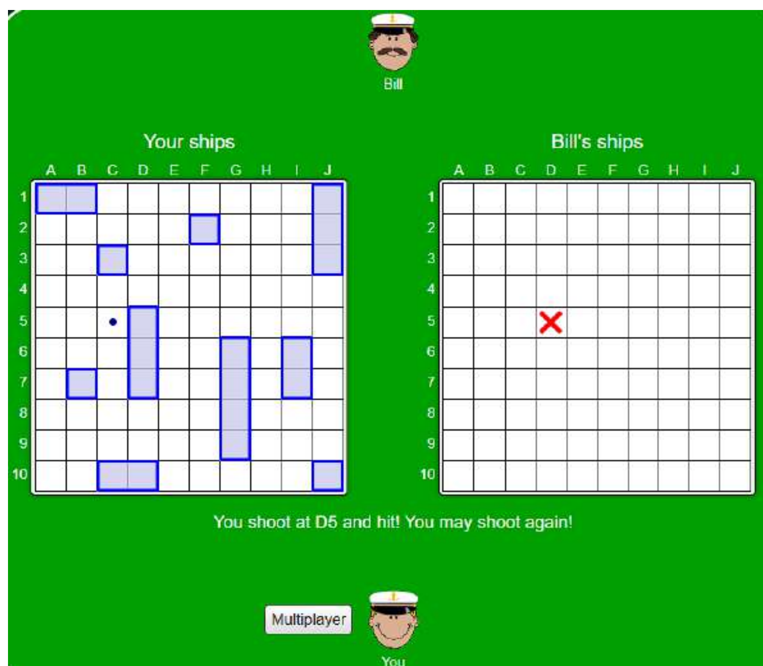

Рис. 1.16. Графічне зображення попадання по частині корабля у веб грі Морський бій

Потоплений корабль помічується червоним кольором. Клітинки які були заблоковані внаслідок попадання по усім частинам, відмічаються сірим кольором. Це зручно, оскільки можна легко підрахувати скільки ходів витримав користувач за гру (Рис. 1.17).

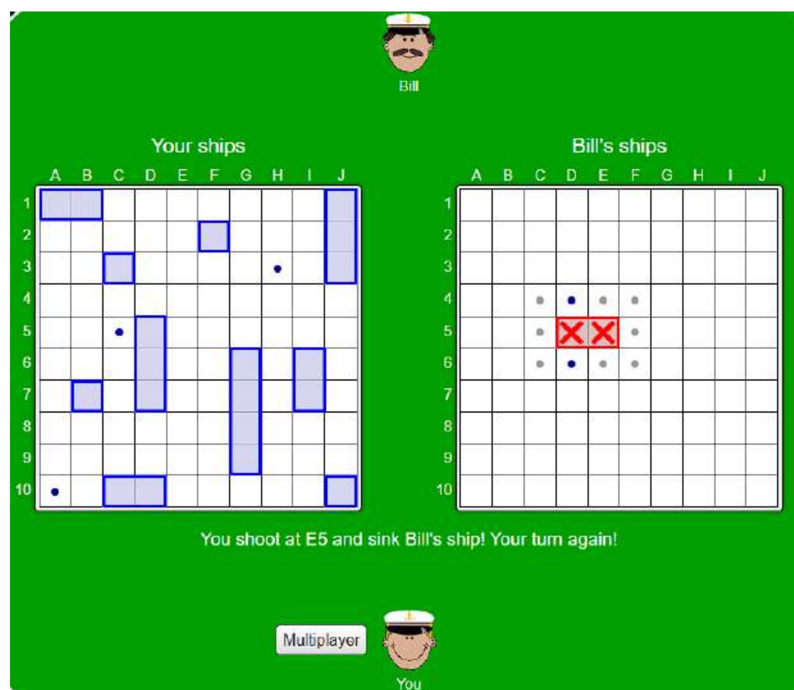

Рис. 1.17. Графічне зображення попадання по всьому кораблю у веб грі Морський бій

Після того, як один з гравців потопив усі кораблі, не затоплені кораблі переможця стають видимі. Але в той же час відображається діалогове вікно, де вказується переможець і пропонується рестарт гри. Діалогове вікно перекриває частину полів, через що не можна побачити всі затоплені кораблі переможця (Рис. 1.18).

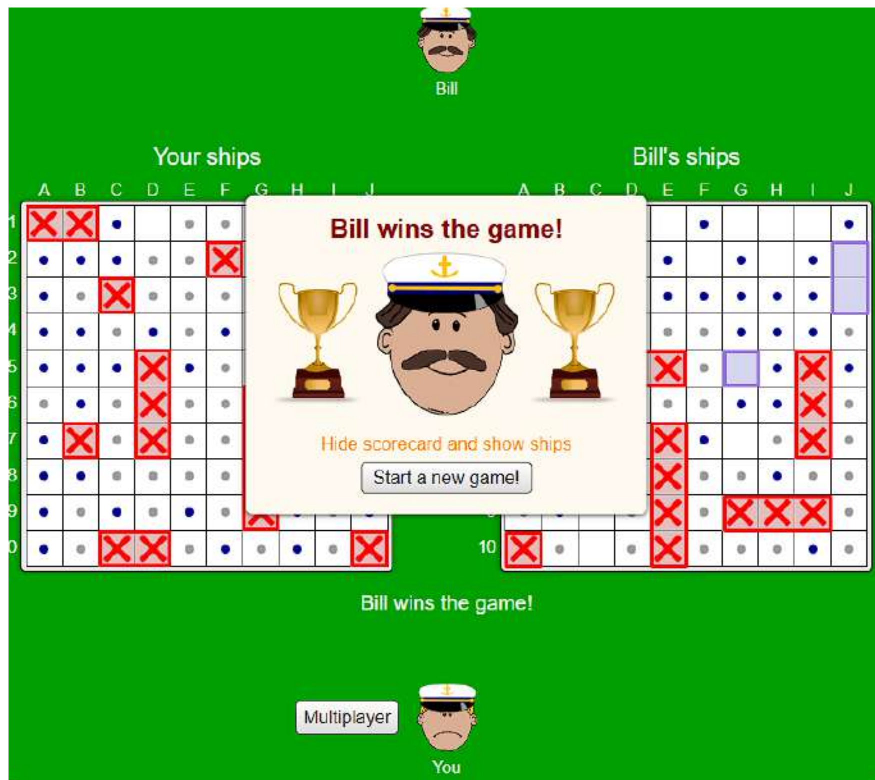

Рис. 1.18. Кінець гри у веб грі Морський бій

Гра має кілька цікавих рішень, які можна перейняти у власну інтрепретацію:

- проста графіка, яка не навантажує систему;
- рядок повідомлень, який описує дії гравців;
- класичний алгоритм гри “Морський бій”.

Недивлячись на це, гра має низку недоліків, які варто врахувати та не повторювати при розробці власної гри:

- елементи які перекривають один одного;
- маленький вибір мов;
- відсутність адаптації до ручної розстановки кораблів.

1.3. Аналіз вимог до гри

Проект гри «Морський бій» базується на класичних принципах гри, які знайомі користувачам ще з паперових варіантів. Це стало основою для формування функціональних і нефункціональних вимог, що мають забезпечити зрозумілу, стабільну та приємну взаємодію користувача із системою.

Основна увага зосереджувалась на реалізації типової ігрової ситуації: перед початком користувач розміщує флот на сітці 10×10, з класичним набором кораблів, після чого розпочинається черговий обмін ударами з опонентом. Влучання і промахи повинні супроводжуватись наочним візуальним сигналом, а перемога - фіксуватись у базі даних, і автоматично додаватися у турнірну таблицю веб сайту. Весь процес гри повинен проходити без затримок або зайвих відволікаючих елементів.

Функціональні вимоги до програмного забезпечення:

- відображення двох ігрових полів (для гравця і супротивника);
- ручне або автоматичне розміщення кораблів перед початком гри;
- система перевірки на коректність розташування кораблів;
- покроковий режим гри з чергуванням ходів;
- індикація влучань, промахів і знищення кораблів;
- автоматичне завершення гри при знищенні всього флоту одного з учасників;
- передача результату до вебінтерфейсу (ім’я, переможець, дата гри, кількість ходів).

Нефункціональні вимоги до програмного забезпечення:

- гравець повинен інтуїтивно розуміти керування без вивчення інструкцій;
- кожна дія супроводжується реакцією системи;
- гра має стабільно працювати на більшості сучасних комп’ютерів;
- відсутність потреби в мережевому з'єднанні під час самої гри;

- мінімалістичний інтерфейс без перевантаження графікою.

Технічні вимоги до реалізації:

- реалізація десктопної частини мовою C#;
- реалізація графічної частини з використанням OpenGL;
- підтримка ОС Windows (версії не нижче Windows 7);
- інтеграція з вебсервісом через HTTPS-запит (REST API);
- реалізація сторінки турнірної таблиці мовою JavaScript
- зберігання даних про результати гри в базі даних SQL;
- реалізація серверної за технологією ASP.NET;

Окрему увагу було приділено перевагам, які може запропонувати розроблювана система порівняно з наявними аналогами. Головною з них є поєднання десктопного інтерфейсу із сучасним веб механізмом збереження та перегляду результатів у турнірній таблиці. Завдяки введенню такої функції гра буде не лише засобом розваги, а й частиною повноцінної турнірної системи. На відміну від більшості існуючих реалізацій, де статистика обмежується лише локальним збереженням, даний проект дозволяє виводити динамічну таблицю результатів онлайн з будь-якого пристрою.

Реалізація та побудова архітектури коду повинні створювати передумови для розширення функціоналу у майбутньому, зберігаючи стабільність основної і вже існуючої механіки гри.

Висновки до розділу 1

У результаті проведеного аналізу вимог було визначено ключові функціональні та нефункціональні характеристики, які формують основу проектування гри. Визначені параметри забезпечують відповідність класичним правилам «Морського бою». Чітке розмежування обов’язкової логіки, графічного відображення та інтеграції з веб сайтом дозволило сформувати комплексне бачення майбутньої реалізації.

Було розділено логіку, графічного відображення та інтеграції з

вебсистемою, сформовано комплексне бачення майбутньої реалізації. Акцент на простоті керування, швидкому відгуку системи та мінімальних апаратних вимогах забезпечує широке охоплення користувачів і можливість використання гри як у локальному, так і в турнірному режимі.

Сформульовані технічні вимоги лягли в основу вибору мови програмування, бібліотек і засобів розробки. Вони оптимізують процес реалізації та забезпечити відповідність цільовим завданням проєкту.

РОЗДІЛ II
ПРОЕКТУВАННЯ ЗАДАЧІ

Розробка програмного комплексу потребує ретельного підбору інструментів, які не лише відповідають поставленим функціональним вимогам, а й забезпечить зручність підтримки та візуалізації. У межах даного проекту основну увагу було зосереджено на проектування десктопної і веб частини додатку. Враховуючи особливості задачі, необхідно зробити вибір окремих мов програмування, бібліотек та фреймворків, які підійдуть під вимоги програмного забезпечення.

Ігрова частина, що реалізує «Морський бій», буде створена за допомогою мови C#, а конкретно технології Windows Forms. C# надає можливості для побудови графічного інтерфейсу, має структуру класів та зручну обробку подій, що є вимогою для побудови динамічного ігрового процесу та чистої архітектури коду. Для виведення візуального середовища було обрано бібліотеку OpenGL, яка забезпечує побудову графіки на низькому рівні й дозволяє контролювати кожен етап рендерингу. Через OpenGL буде реалізовано виведення ігрового поля, розміщення кораблів, позначення влучень і промахів.

В ході роботи було побудовано схему архітектури ігрового застосунку, де показано зв’язок між ігровою логікою, графічним рендером та обробкою подій користувача (Рис. 2.1).

Рис. 2.1. Архітектура ігрового застосунку

Гра після завершення партії створює локальний запис у базі даних формату SQLite. Для цього використовується вбудований модуль, що формує SQL-запит і додає новий рядок до таблиці з фіксацією результату. У таблиці зберігаються поля з іменами гравців, кількістю ходів, результатом і датою. База розташована у папці поряд із виконуваним файлом програм. Додаткових служб для роботи з нею не потрібно використовувати.

Вебінтерфейс зчитує дані з тієї ж бази без права змін. Сервер працює за стандартною схемою API-запиту: отримує звернення, формує SQL-вибірку, переводить дані у формат JSON і передає на фронт. Фронтенд, написаний на JavaScript, приймає відповідь і будує турнірну таблицю. Оновлення інформації відбувається за запитом, без перезавантаження сторінки.

Гра й сайт не обмінюються даними між собою напряму. Вони працюють з однією базою, але по окремим алгоритмам: гра записує дані, сайт їх читає.

Було побудовано блок-схему клієнт-серверної архітектури комплексу, де показано, сервер взаємодіє із базою даних, та передає дані до клієнтської частини веб інтерфейсу (Рис. 2.2).

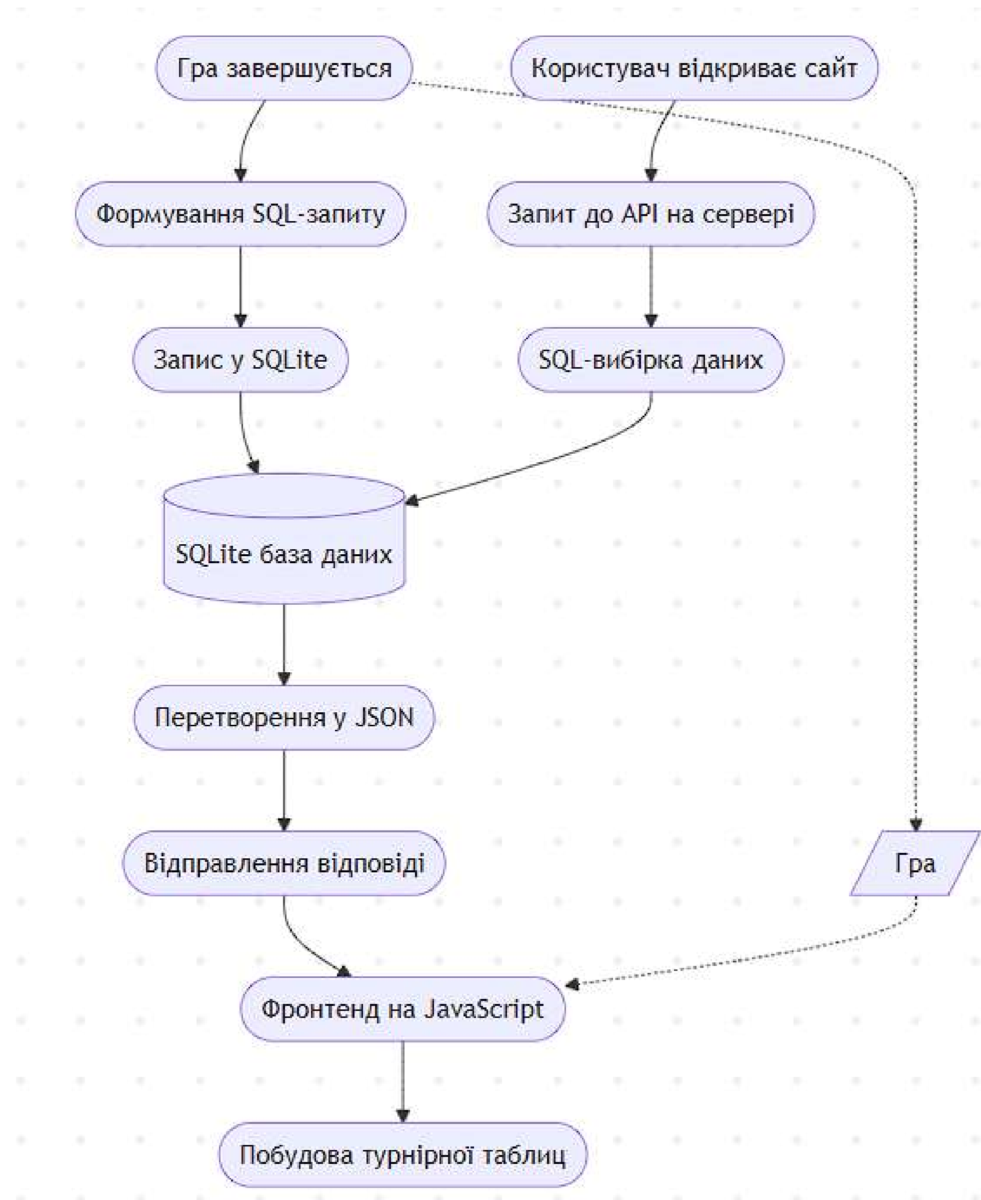

Рис. 2.2. Архітектура клієнт-серверної частини

Клієнтська частина сайту побудована на JavaScript і HTML. Інтерфейс формує таблицю зі статистикою гравців і підтримує періодичне оновлення даних за допомогою AJAX-запитів. Це дає змогу отримувати оновлення з сервера у фоновому режимі, не перезавантажуючи сторінку. Дані, які

надходять у відповідь, одразу перетворюються на елементи таблиці з іменами гравців, кількістю перемог та іншими показниками.

Було побудовано схему структури веб інтерфейсу. У ньому показано приблизне розташування таблиці, кнопки ручного оновлення та блоку з інформацією про кількість проведених ігор (Рис. 2.3).

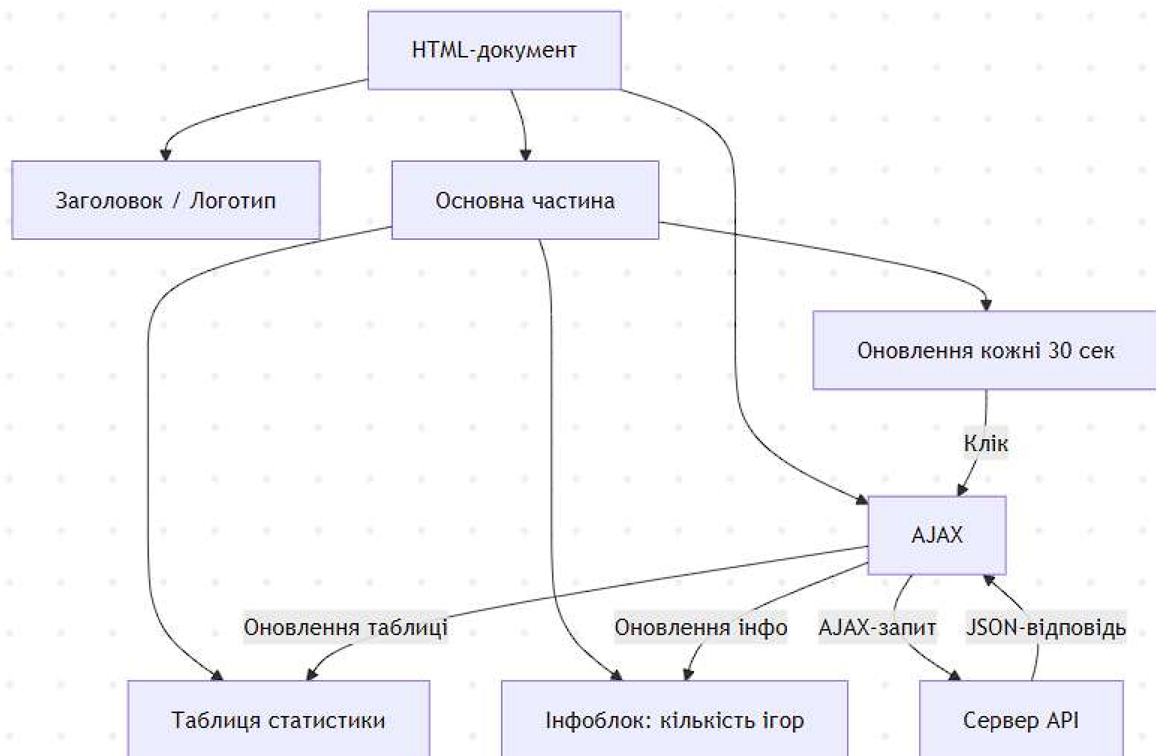

Рис. 2.3. Структура інтерфейсу сайту

Обмін даними з сервером побудований на REST-архітектурі. Кожен запит обробляється окремим ендпоінтом, що спрощує підтримку й дає змогу додавати нові функції без зміни основної логіки. За потреби до цього ж API можна підключити сторонні клієнти.

Підбір програмних і технічних засобів відбувався з урахуванням вимог до стабільної роботи, доступності середовища розробки та можливості подальшого розширення функціональності. Завдяки цьому структура системи залишилася простою для реалізації, водночас зберігає необхідний функціонал.

Головним елементом гри є ігрове поле, реалізоване у вигляді двовимірного масиву. Кожна клітинка цього масиву містить інформацію про

стан: порожня, зайнята, влучена, недоступна. У грі використовується два поля — по одному на кожного гравця. Всі дії на полі виконуються по черзі, згідно з визначеними правилами.

Кораблі описуються за допомогою довжини, початкових координат і напрямку. Крім базових параметрів, кожен корабель має лічильник пошкоджених сегментів, щоб потім можна було автоматично визначати момент його повного знищення. Усі перевірки правильності розміщення виконуються до початку гри, включаючи перевірку відсутності перетинань і дотримання відстані між об'єктами.

Частина інтерфейсу забезпечує обробку дій користувача. Натискання на клітинку інтерпретується як постріл. Далі система змінює стан відповідної клітинки, відображає результат ходу, перевіряє чи було це влучання, промах, знищення, і при потребі блокує клітинки довкола знищеного корабля.

Результати гри обробляються окремим кодом. У разі завершення формується структура з параметрами: ім’я гравця, результат гри, кількість ходів, дата. Ці дані передаються зберігаються у базі дані, які далі буде зчитувати серверна частина та передавати їх на сайт.

Загальна структура системи поділена на три функціональні рівні:

- рівень графічного інтерфейсу;
- рівень логіки;
- рівень взаємодії з базою даних.

Таким методом можна мінімізувати зв’язки між модулями, що спрощує їхню підтримку. Кожен рівень функціонує як окрема частина з чітко визначеним функціоналом, не залежачи від внутрішньої реалізації інших блоків.

Паралельно з реалізацією клієнтської частини гри було спроектовано супровідну веб систему, призначену для збереження та відображення статистичних даних. Основною функцією цього модуля є формування турнірної таблиці, що фіксує результати кожного сеансу гри.

Архітектура веб модуля передбачає наявність двох рівнів: клієнтського та серверного. Серверна частина реалізована на основі ASP.NET із використанням мови програмування C#. Передача даних відбувається у форматі JSON через REST API. Сервер зчитує дані з бази даних, куди попередньо були записані дані з ігрової частини реалізації, здійснює базову перевірку отриманої інформації та передає дані на сайт, де вже малюється турнірна таблиця, та заповнюється даними з бази даних.

Кожен запис у базі містить фіксовану структуру: ім’я користувача, дату гри, результат, а також кількість ходів. Така уніфікація спрощує обробку даних на клієнтському рівні та забезпечує можливість сортування і фільтрації за визначеними параметрами.

Клієнтська частина реалізована з використанням HTML, CSS та JavaScript. Головна сторінка містить таблицю, що виводить інформацію з бази даних. Доступ до актуальних даних забезпечується через асинхронні AJAX-запити до API. Усі дії на стороні користувача — зчитування, оновлення, сортування - виконуються без перезавантаження сторінки.

У процесі проектування сформовано загальну схему реалізації, яка охоплює всі компоненти розробки: логіку ігрового процесу, візуалізацію дій користувача, а також взаємодію з веб частиною. Завдяки проектуванню вдалось оптимізувати функціональні зв’язки між модулями, зменшити складність коду та підготувати проект до подальших змін без порушення структури.

Загальна модель взаємодії між ігровою програмою та вебкомпонентом передбачає чіткий розподіл функціональних обов’язків. Клієнт відповідає виключно за побудову та відображення таблиці, а сервер - за обробку та передачу даних.

Висновки до розділу 2

У результаті проектування основних компонентів системи вдалося сформувати цілісну архітектуру, яка охоплює як десктопну частину гри, так і веб систему для ведення турнірної таблиці. Кожен елемент структури отримав вимоги до функціональності. Вдалося уникнути дублювання задач і забезпечити логічну цілісність взаємодії між кодом різних частин програмного забезпечення.

Детальне опрацювання моделі ігрового поля, об’єктів кораблів, логіки пострілів і перевірки стану гри дало змогу закласти стабільний фундамент для реалізації механіки, наближеної до класичного варіанту «Морського бою». Водночас розробка веб частини та серверного API забезпечила відображення результатів і можливість віддаленого перегляду статистики для досягнення мети розширення призначення застосунку.

Спроектована система базується на модульному підході, що полегшує подальший розвиток - як у бік розширення функціональності гри, так і у вдосконаленні сервісної частини.

РОЗДІЛ III
ПРОЕКТУВАННЯ БАЗИ ДАННИХ ДЛЯ ПРОЕКТУ

3.1. Вибір системи управління базами даних

Вибір системи управління базами даних (СУБД) є важливою складовою проектування програмного забезпечення. СУБД визначає, як саме відбувається зберігання, обробка та доступ до даних, що, в свою чергу, впливає на продуктивність системи в цілому. Вибір СУБД для даного проекту був здійснений з огляду на вимоги до обсягу та частоти обробки даних, специфіки взаємодії з додатком та можливостей розширення можливостей в майбутньому.

Аналізуючи характер даних, що зберігатимуться в системі, можна виокремити кілька ключових типів інформації. Перший тип - це сутність записи гравців - структуровані, низько динамічні дані, які рідко змінюються після створення. Другий тип даних є результати ігрових сесій - високочастотні записи, що вносяться в базу даних гравцями після кожного завершеного матчу, мають чітку структуру та однозначні зовнішні зв’язки (ідентифікатори гравців-переможця і переможеного, кількість ходів, час гри).

Особливістю цієї моделі є її простота з точки зору реляційної структури: невелика кількість таблиць із чітко визначеними зв’язками «один-до-багатьох», мінімальне використання вкладених запитів, відсутність необхідності в зберіганні неструктурованих або мультимедійних даних [9]. Навантаження на систему переважно зчитувального типу: очікується велика кількість операцій вибірки з бази для побудови турнірної таблиці, при цьому операції запису виконуються рідше, у момент завершення гри.

Крім того, відсутність вимог до одночасної багатокористувацької роботи з базою на рівні транзакцій дозволяє не враховувати такі фактори, як блокування записів, паралельна обробка, оптимізація багатопоточності тощо. Більшість запитів є типовими SELECT-операціями або INSERT-запитами із

простими умовами фільтрації та сортування. Очікувана кількість записів обмежується кількома тисячами елементів. Такий підхід також дозволяє уникнути перевищення критичних обсягів для обробки.

Враховуючи вищезазначене, можна зробити висновок про доцільність використання СУБД, орієнтованої на легковагову, локальну обробку структурованих даних з низьким ступенем складності запитів і невеликим обсягом транзакційної активності. Саме такими характеристиками володіє SQLite, вбудована реляційна СУБД, яка функціонує у вигляді єдиного файлу на файловій системі. Вона не потребує окремого сервера або системного процесу, забезпечує повноцінну підтримку SQL-операторів та є придатною для швидкого розгортання в рамках десктопного або малонавантаженого вебзастосунку.

SQLite була обрана як основна СУБД для програмного забезпечення також з додаткових причин. Це вбудована реляційна база даних, яка не вимагає окремого серверного програмного забезпечення. Це робить її ідеальним вибором для додатків, працюючих в локальному середовищі або маючих невеликий обсяг даних для обробки.

SQLite підтримує стандарт SQL. Тому можна використовувати звичні механізми для роботи з таблицями, індексами, запитами, а також здійснювати трансакційні операції, щоб забезпечити збереження цілісності даних [12]. Також важливою перевагою є невеликі вимоги до ресурсів, які варто врахувати при розробки програми, в якій обсяг даних не перевищує кількох гігабайт, а складність запитів є середньою.

Технічні вимоги до обраної СУБД включають:

- простота інтеграції та налаштування;
- продуктивність;
- забезпечення безпеки даних.

SQLite легко інтегрується в проекти без необхідності в складних налаштуваннях серверного програмного забезпечення. Вона не потребує окремого сервера для роботи, оскільки дані зберігаються безпосередньо в

файлі, що зменшує витрати на адміністрування та конфігурацію. Дана СУБД забезпечує достатню швидкість обробки даних при помірному навантаженні. Її продуктивності вистачає для додатків, де обсяг даних обмеженийількома мільйонами записів. SQLite має механізми транзакцій, підтримку цілісності даних і легко відновлюється після збоїв.

Для доцільної оцінки СУБД був проведений порівняльний аналіз з іншими популярними системами (табл. 3.1).

Таблиця 3.1

Порівняння СУБД

Характеристика	SQLite	MySQL	PostgreSQL
Тип	Вбудована, файлова	Клієнт-серверна	Клієнт-серверна
Продуктивність	Висока для невеликих обсягів даних	Висока при великому навантаженні	Висока при складних запитах
Підтримка транзакцій	Так	Так	Так
Легкість налаштування	Просте налаштування	Потрібна складна настройка	Складне налаштування
Вартість використання	Безкоштовна	Безкоштовна (за умови використання open-source версії)	Безкоштовна

Такий вибір об'єктів для порівняння не є випадковим: обидві ці системи є відкритими, безкоштовними та широко використовуються в проектуванні сучасних програмних продуктів. Як MySQL, так і PostgreSQL - це повнофункціональні клієнт-серверні рішення, що забезпечують високу продуктивність, підтримку складних транзакцій, багатокористувацький доступ, реплікацію, резервування та інші інструменти для обслуговування розподілених систем.

Однак, незважаючи на багатий функціонал цих систем, їх використання для даного типу задачі не є виправданим. Основними критеріями, що не дозволяють MySQL та PostgreSQL вважати оптимальними в даному випадку, є:

- надлишкова складність;
- зайві ресурси;
- зайве навантаження на систему;
- залежності від інфраструктури.

Обидві згадані СУБД вимагають встановлення, налаштування та підтримки серверного ПЗ, окремого процесу авторизації, обробки з’єднань та керування ресурсами, що значно ускладнює розгортання локального десктопного застосунку. Для невеликих обсягів інформації, обмеженої кількості користувачів та низької частоти транзакцій, використання повноцінного серверного ПЗ виглядає як надлишкове рішення, яке споживає більше оперативної пам’яті та процесорного часу, ніж це дійсно необхідно. У випадку з MySQL чи PostgreSQL розгортання додатку на іншому пристрої потребує попереднього встановлення СУБД, налаштування мережевих з'єднань, імпорту структури бази та її наповнення. Натомість SQLite не потребує нічого окрім самого файла БД, що дозволяє переносити застосунок разом із усіма даними навіть між різними операційними системами.

Як видно з порівняння, SQLite виграє у своєму класі саме завдяки мінімалістичному підходу до реалізації. Усі базові вимоги проекту (робота з реляційними таблицями, підтримка транзакцій, забезпечення цілісності даних, швидка вибірка та вставка записів) повністю покриваються її функціоналом. При цьому відсутність зайвих серверних механізмів дозволяє зменшити обсяг коду, необхідного для інтеграції, та уникнути потреби в конфігурації сторонніх компонентів.

На відміну від MySQL або PostgreSQL, де адміністратор має справу з десятками параметрів конфігурації, у SQLite основні оптимізаційні можливості зосереджені на стороні застосунку. Це означає, що ефективність

роботи з базою залежить не лише від самої СУБД, а й від того, наскільки правильно побудована логіка взаємодії з нею: частота відкриття/закриття з’єднання (файлу), використання транзакційного кешу, структура запитів та обробка помилок запису.

Одним з аспектів, що часто недооцінюється при використанні вбудованих СУБД, є коректне налаштування параметрів середовища виконання, яке напряму впливає на стабільність і швидкодію системи в цілому. У випадку SQLite, на відміну від клієнт-серверних рішень типу MySQL чи PostgreSQL, усі транзакції виконуються у межах одного процесу, а доступ до бази реалізовано через локальні файлові операції [14]. Такий підхід зменшує накладні витрати на мережеву взаємодію та керування пулом з’єднань, однак вимагає уважного ставлення до налаштувань файлового кешу, режимів ведення журналу, а також до рівня синхронізації.

Для забезпечення належної продуктивності в умовах середнього навантаження рекомендовано використовувати параметр journal_mode = WAL, який дозволяє зчитування і запис у базу одночасно з боку різних потоків одного процесу. При цьому варто враховувати, що файл бази зберігається на локальному диску, тому тип файлової системи і носії даних SSD та HDD значно впливають на загальну продуктивність. У більшості випадків використання SSD з мінімальним часом доступу до сектора забезпечує в декілька разів кращу продуктивність при операціях вибірки та запису невеликих обсягів даних.

Рівень синхронізації повинен мати значення NORMAL або OFF - це значно покращує швидкодію, однак зменшує захист від втрати даних при аварійному завершенні роботи. У проектах із низькими ризиками втрат даних або з періодичним резервним копіюванням це може бути виправдано.

Обсяг кешу визначається параметром cache_size і напряму впливає на кількість звернень до диска при виконанні вибірок. Рекомендоване значення — від 2000 до 5000 сторінок. Це оптимально для таблиць з кількома тисячами записів.

Наявність індексації на ключових полях вибірки (WinnerPlayerId, GameDateTime) є обов'язковою умовою ефективного виконання SELECT-запитів із фільтрацією. В іншому випадку зростає час відгуку бази при збільшенні обсягу записів.

Особливістю SQLite є те, що всі операції виконуються без серверного шару, тому ефективність взаємодії з базою значною мірою залежить від правильного використання запитів: об'єднання кількох INSERT-операцій в одну транзакцію суттєво пришвидшує запис.

Збереження балансу між продуктивністю та надійністю можливе лише при ручному коригуванні ключових параметрів PRAGMA відповідно до сценарію використання. Поганим рішенням буде ігнорування цих параметрів і залишення значень за замовчуванням, бо воно може призвести до зниження швидкодії або втрати даних.

3.2. Структури таблиць бази даних

Після вибору системи управління базами даних наступним етапом є побудова структури самої бази, яка забезпечуватиме зберігання і доступ до інформації, і потім використається програмним забезпеченням. Побудова структури бази даних здійснюється на основі інфологічного аналізу предметної області, з подальшим переходом до фізичного проектування таблиць і визначення зв'язків між ними. Система, для якої проектується база даних, фіксує гравців та результати зіграних матчів, вона повинна дозволяти відстежувати ігрову статистику на сайті.

Інформаційна модель будується на основі двох сутностей: гравці та результати ігор. Сутність гравця репрезентує окрему особу, яка бере участь у грі, і описується набором атрибутів, які дозволяють точно і однозначно її ідентифікувати. У реалізованій структурі передбачено лише одне поле — ім'я користувача, яке зберігається разом із системним ідентифікатором. Оскільки ім'я не є унікальним параметром у цій реалізації, саме автоматично

згенерований ключ є основним засобом зв’язку з іншими сутностями. Розширення таблиці гравців додатковими атрибутами можливе на подальших етапах, однак у поточній версії збережено лише мінімальний необхідний набір даних.

Ігрова сесія в системі представлена як окрема сутність, яка зберігає результат взаємодії між двома гравцями. Кожен запис містить інформацію про переможця і переможеного, кількість зроблених ходів та часову мітку завершення гри. Для встановлення цілісності даних та уникнення дублювання в таблиці результатів використовуються зовнішні ключі, які посилаються на ідентифікатори гравців у відповідній таблиці. Так формується зв’язок типу «багато-до-одного» з обох боків — кожен результат гри пов'язаний з двома окремими гравцями.

Уся структура зберігається в межах двох реляційних таблиць: Players та GameResults. Таблиця Players має два поля: унікальний ідентифікатор PlayerId, який є первинним ключем і генерується автоматично, та текстове поле Name, яке містить ім’я гравця. Вибір типу даних TEXT для імені пояснюється необхідністю збереження текстових значень, що можуть містити символи різної мови. Обмеження NOT NULL запобігає додаванню анонімних записів (табл. 3.2).

Таблиця 3.2

Таблиця бази даних яка зберігає дані про гравців

Назва поля	Тип даних	Опис
PlayerId	INTEGER	Первинний ключ, унікальний ідентифікатор гравця. Автоматично інкрементується.
Name	TEXT	Ім’я гравця. Не може бути порожнім.

У таблиці GameResults передбачено п’ять полів: унікальний ідентифікатор запису ResultId, два зовнішніх ключі — WinnerPlayerId та

LoserPlayerId, поле GameDateTime, воно фіксує час завершення гри, і поле MovesCount, яке зберігає кількість зроблених ходів. Обидва зовнішні ключі мають обов’язкові значення (NOT NULL) і реалізують зв’язки з основною таблицею гравців. Тип INTEGER для ідентифікаторів відповідає формату ключів у таблиці гравців, що забезпечує сумісність при побудові зв’язків. Тип TEXT для дати збережено відповідно до синтаксису SQLite, яка не має окремого типу DATETIME, але допускає зберігання часових міток у текстовому форматі за стандартом ISO 8601. Тип поля MovesCount також є INTEGER. Воно буде використане для того, щоб фіксувати кількісне значення без втрати точності (табл. 3.3).

Таблиця 3.3

Таблиця бази даних яка зберігає дані про зіграні матчі

Назва поля	Тип даних	Опис
ResultId	INTEGER	Первинний ключ, унікальний ідентифікатор результату гри.
WinnerPlayerId	INTEGER	Зовнішній ключ на таблицю Players, гравець-переможець.
LoserPlayerId	INTEGER	Зовнішній ключ на таблицю Players, гравець, що програв.
GameDateTime	TEXT	Дата та час завершення гри.
MovesCount	INTEGER	Кількість зроблених ходів у грі.

Між таблицями реалізовано два зовнішні ключі: WinnerPlayerId та LoserPlayerId, щоб здійснювати каскадні вибірки та запити зі з’єднанням, які також називають JOIN-запитами для отримання імен гравців за їхніми ідентифікаторами. Така реалізація також допомагає уникнути появу «висячих» записів у таблиці результатів, тобто таких, які посилаються на неіснуючих гравців. Обмеження цілісності автоматично контролюються СУБД при

спробах видалення або зміни пов’язаних записів.

Запити для створення відповідних таблиць подані нижче. Вони реалізують структуру бази даних з використанням мови SQL у межах платформи SQLite:

```
CREATE TABLE IF NOT EXISTS Players (
   PlayerId INTEGER PRIMARY KEY AUTOINCREMENT,
   Name TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS GameResults (
   ResultId INTEGER PRIMARY KEY AUTOINCREMENT,
   WinnerPlayerId INTEGER NOT NULL,
   LoserPlayerId INTEGER NOT NULL,
   GameDateTime TEXT NOT NULL,
   MovesCount INTEGER NOT NULL,
   FOREIGN KEY (WinnerPlayerId) REFERENCES Players (PlayerId),
   FOREIGN KEY (LoserPlayerId) REFERENCES Players (PlayerId)
);
```

Для того, щоб запобігати дублюванню структури під час повторного запуску ініціалізації бази, обидві таблиці створюються лише у випадку їх відсутності (IF NOT EXISTS). Поля, які містять зовнішні ключі, не допускають NULL-значень, таким чином збереження частково заповнених записів результатів гри неможливе. Спосіб перевірки даних на рівні СУБД сприяє збереженню консистентності інформації та формуванню цілісної історії взаємодії користувачів.

Конструкція таблиць відповідає класичній схемі нормалізації до третьої нормальної форми. Відсутні функціональні залежності неключових атрибутів між собою, уникається надлишкове дублювання. Обидві таблиці мають чітко виражений первинний ключ, використовують атомарні поля та не зберігають масивів або структурованих даних. Завдяки цьому забезпечується прозора логіка запитів, низький рівень ентропії даних і передбачуване навантаження на систему.

Запити до цієї структури є лінійними за складністю. Найбільш типовими

є операції вибірки результатів за фільтрами часу, кількості ходів або участі конкретного гравця. При побудові турнірної таблиці виконується агрегація виграних ігор з об’єднанням по WinnerPlayerId. Для відображення історії конкретного гравця виконується фільтрація записів, де він виступає переможцем або переможеним. Такі запити не потребують вкладених підзапитів або тимчасових таблиць.

Фізична реалізація структури на основі SQLite дозволяє зберігати всі таблиці в єдиному файлі, який є доступним для читання і запису з боку застосунку. Операції з базою відбуваються без мережевої взаємодії, знижується латентність і зменшується кількість точок відмови. Відсутність сервера як окремого процесу також зменшує ресурси, необхідні для обслуговування структури.

Переваги обраної структури полягають у її простоті взаємозв’язків та відсутності зайвої надбудови. Всі залежності між таблицями є явними, однозначними і не потребують складної логіки для обробки. Дані зберігаються в атомарному вигляді, що забезпечує ефективність у пошуку, сортуванні та фільтрації. Оскільки таблиці не використовують тригери, представлення або індекси, допускається подальше вдосконалення на основі аналітики використання.

У разі необхідності розширення функціональності структура підтримує додавання нових таблиць без необхідності модифікувати наявні зв’язки. Це забезпечується завдяки тому, що основні зв’язки ґрунтуються на стабільних первинних ключах. Нові сутності можуть інтегруватись за допомогою зовнішніх ключів, не порушуючи вже існуючу логіку системи.

Розроблена структура таблиць повністю задовольняє вимоги до збереження, обробки та доступу до інформації в межах заданої функціональності програмного продукту. Вона забезпечує узгодженість і надійність даних, дозволяє реалізувати запити довільної складності в рамках заданої моделі.

Висновки до розділу 3

У результаті проведеного аналізу було сформовано та обґрунтовано підхід до вибору системи управління базами даних та спроектовано логічну структуру зберігання інформації відповідно до потреб програмного забезпечення. Критеріями вибору СУБД стали особливості даних, їх структурованість, обмежений обсяг, низький рівень транзакційної активності та переважання операцій читання над записом. Розгляд альтернативних СУБД MySQL та PostgreSQL, засвідчив їхню надмірність для заданих вимог, тоді як SQLite продемонструвала оптимальне співвідношення між функціональністю та ресурсними потребами.

Реалізована інфологічна модель описує два ключові об’єкти предметної області, гравців та результати зіграних матчів. Об’єкти мають чітко визначені зв’язки та обмеження цілісності. Структура таблиць забезпечує мінімізацію надмірності даних, підтримку логічної цілісності і можливість побудови запитів для отримання статистики та аналітики. Конструкція бази передбачає потенціал до подальшого розширення функціональності без суттєвого ускладнення схеми.

Проведене проектування бази даних відповідає основним принципам реляційної моделі, включає необхідні засоби логічного зв’язування даних і забезпечує належну гнучкість при роботі в середовищі локального застосунку. Обрані рішення дозволяють реалізувати систему з механізмом збереження інформації, низьким порогом входу для підтримки та обслуговування, а також простотою у перенесенні або резервному копіюванні.

РОЗДІЛ IV
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1. Інтерфейс та алгоритм гри

Інтерфейс гри реалізується як візуальний шар програмного забезпечення, що забезпечує взаємодію користувача з функціональними модулями системи. Його побудова базується на принципах структуризації елементів керування. Мінімізовано кількість дій для виконання типових операцій та забезпечення логічної послідовності подій у межах ігрового сценарію.

Інтерфейс побудований із урахуванням принципів ергономіки зменшує навантаження на користувача під час взаємодії та підвищує загальну ефективність сприйняття інформації. Усі інтерактивні елементи мають достатній розмір, візуальні маркери активного стану, а також логічне групування відповідно до функціонального призначення.

При запуску програми відкривається меню гри. В ньому користувач може обрати режим гри (1 на 1 або проти бота) або закрити програму (Рис. 4.1).

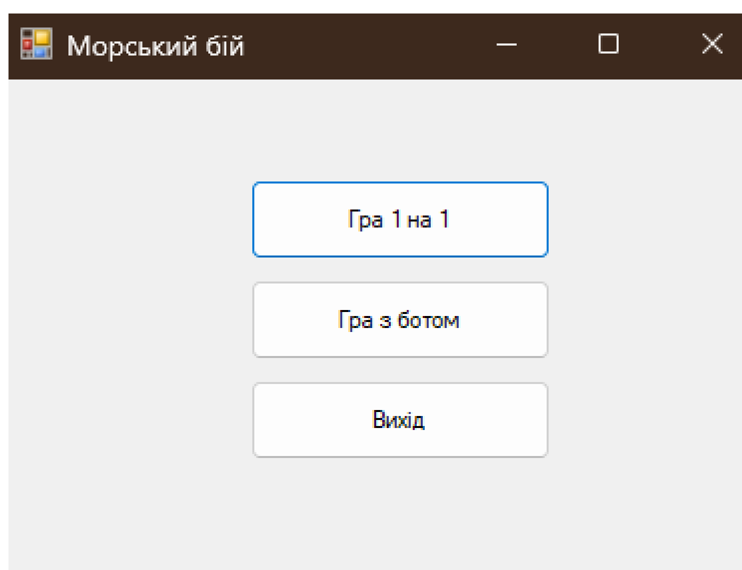

Рис. 4.1. Меню гри Морський бій

Після вибору режиму гри користувачу одразу пропонується обрати тип розстановки - вручну або автоматично (Рис. 4.2).

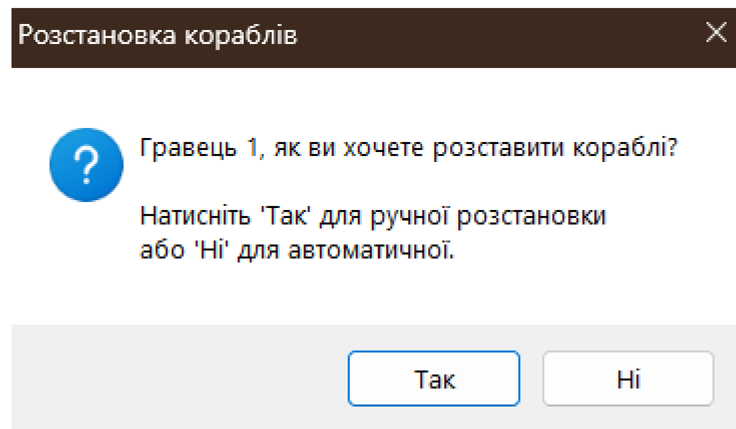

Рис. 4.2. Варіанти розстановки флоту гри Морський бій

При виборі ручного режиму на екран виводиться зона підготовки, де кораблі відображаються у заданих координатах на панелі праворуч від ігрового поля. Користувач має змогу перетягувати кожен корабель мишею та змінювати його орієнтацію через обробку системних подій кліку, руху, подвійного кліку миші.

Корабель складається з окремих частин, кожна з яких промальовується окремо з урахуванням її положення та типу. Система спершу визначає координати кожної частини на основі розміру корабля та його орієнтації, вертикальної або горизонтальної. Потім кожна з частин відображається у вигляді кольорового прямокутника з додатковими декоративними елементами, які залежать від її типу: ніс, корма, середина або одиночна частина.

Для кожного типу секції застосовуються унікальні графічні елементи, носова частина має трикутний виступ і вказує на напрямок корабля, корма містить окремий елемент у вигляді виступу і стилізує задню частину. Середина відображається як прямокутник з темною вставкою, а одиночна частина має додаткове коло в центрі. Вона позначається тільки у малому човні.

Візуалізація кораблів виконується за допомогою OpenGL-примітивів: прямокутники створюються за допомогою Quads, трикутники - через Triangles, а кола - за допомогою TriangleFan. Кожна частина корабля має також чорну рамку, яка обводиться окремо для чіткого відображення меж корабля на полі.

Ігрове поле реалізоване як графічний контролер на базі OpenGL. Ініціалізація побудована так, щоб забезпечувати апаратне прискорення рендерингу та оновлення зображення в реальному часі. Координатна сітка, межі, клітинки і самі кораблі відображаються за допомогою викликів OpenGL-функцій (Рис. 4.3).

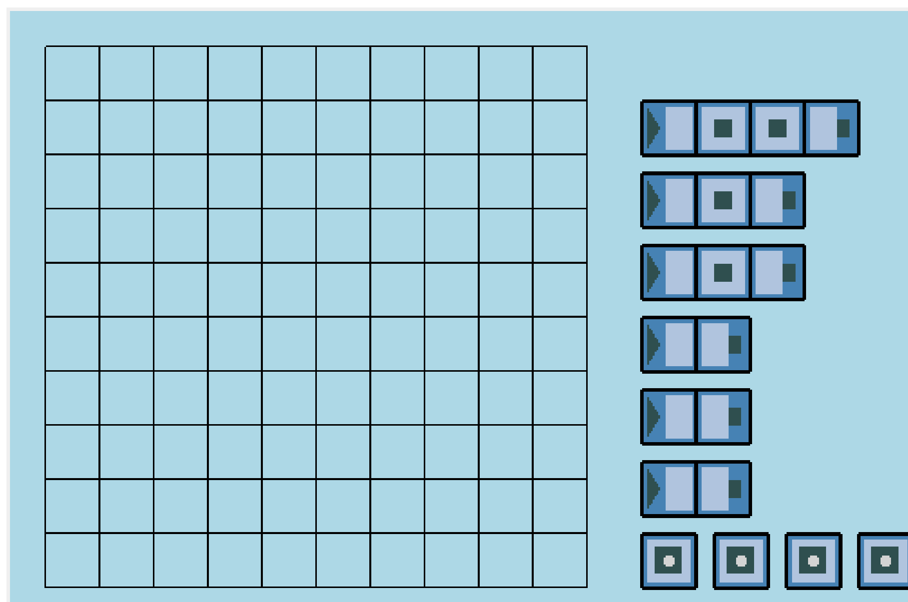

Рис. 4.3. Графічний контролер та флот гри Морський бій

Коли гравець розташовує корабель на сітку — клітинки навколо нього фарбується в червоний колір. Таким чином користувачу повідомляється, що ставити корабель в ці клітинки заборонено. Якщо ж все таки користувач захоче розташувати корабель в заборонену позицію — корабль повернеться в іcходну позицію. Алгоритм передбачає такий сценарій подій, тому у коді відбувається перевірка координат, перед відмальовкою корабля на сітці.

Тобто перевіряється чи валідна клітина для розташування в ній корабля, чи ні (Рис. 4.4).

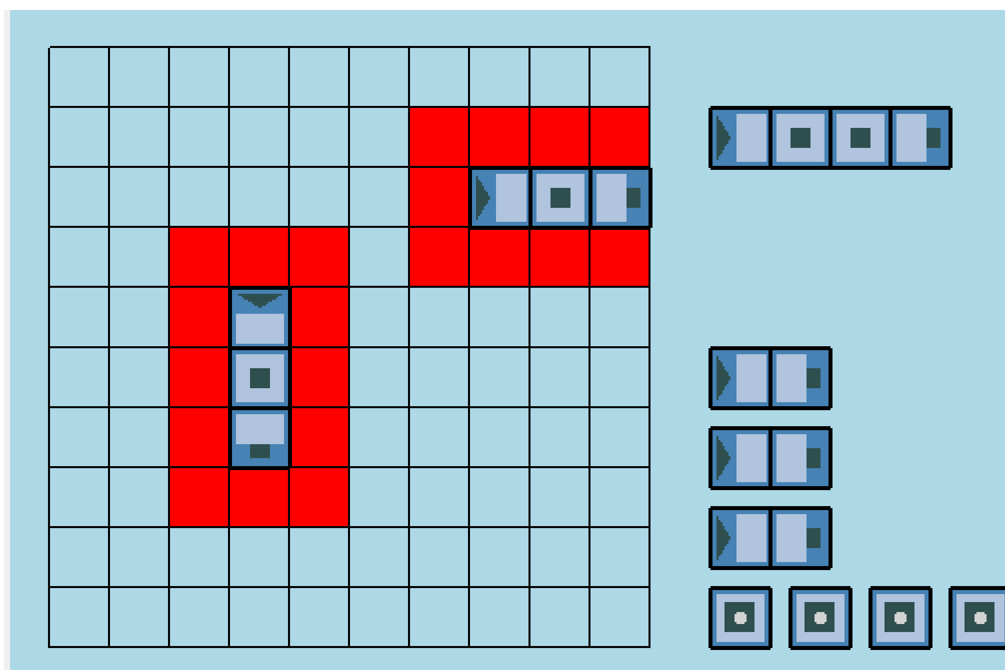

Рис. 4.4. Позначка клітин в яких заборонено розташовувати кораблі

Також передбачена перевірка на те, чи усі кораблі розташував гравець. Якщо кораблі мають початкові координати - це означає, що користувач не розташував флот повністю. В такому разі система повертає помилку, а користувач бачить діалогове вікно з проханням розташувати усі кораблі на гральну сітку (Рис. 4.5).

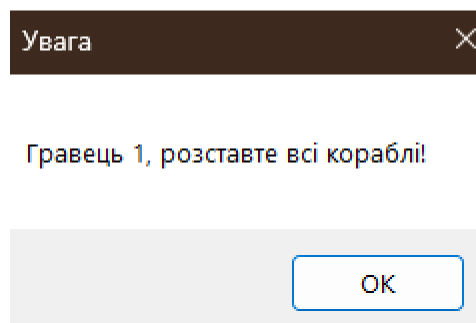

Рис. 4.5. Прохання розташувати усі кораблі

У разі автоматичного розміщення викликається метод, який проходить через набір фіксованих розмірів кораблів і для кожного з них випадковим чином генерує координати, орієнтацію та перевіряє допустимість розміщення на основі вже зайнятих клітин. Алгоритм використовує структуру клітин, які вже містять корабель, для швидкого доступу до координат зайнятих клітин і перевірки суміжних позицій за допомогою методу, який забезпечує

дотримання стандартних правил гри — заборону на сусідство кораблів, заборону на розташування поза сіткою, тощо (Рис. 4.6).

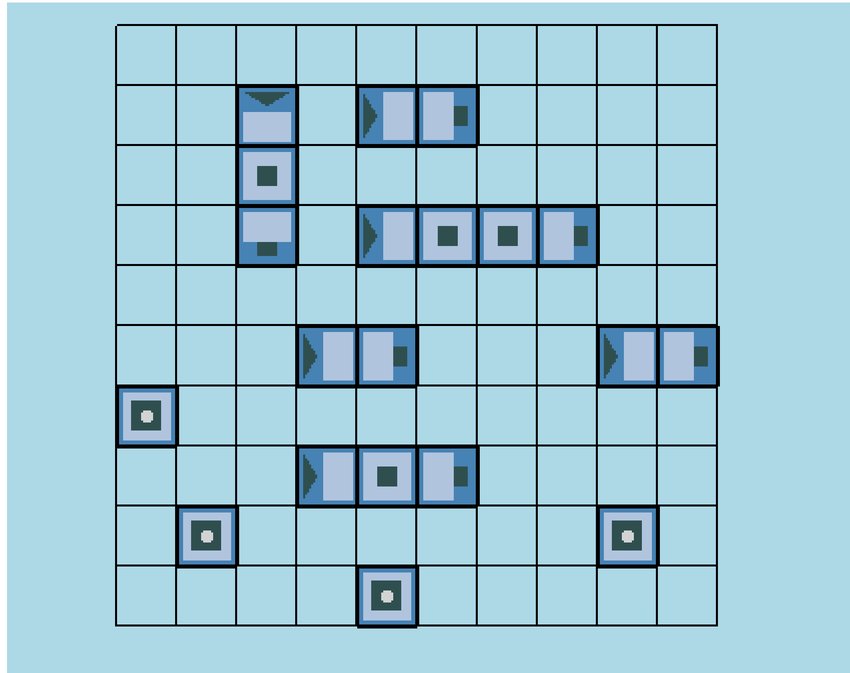

Рис. 4.6. Автоматична розстановка флоту гравця

У випадку неуспішного розміщення, користувачеві виводиться діалогове повідомлення з пропозицією повторити автоматичну генерацію.

Коли гравець розставив усі кораблі та натиснув кнопку далі пропонується метод розташування другому гравцеві [Додаток Д] теж автоматичний або ручний. Або ж у випадку вибору режиму гри з ботом, алгоритм розставляє кораблі за бота поза екраном, а гра одразу починається.

Розташування кораблів для бота [Додаток А] реалізовано за допомогою алгоритму випадкової генерації координат із перевіркою коректності для кожного судна. У цьому алгоритмі фіксований набір кораблів проходиться послідовно: для кожного типу обирається випадкова позиція та орієнтація, після чого перевіряється можливість розміщення в межах поля без конфлікту з уже зайнятими клітинками. Перевірка включає не лише саму позицію судна, а й наявність відступів — згідно з правилами, жоден корабель не повинен стикатися з іншим. Для забезпечення швидкої перевірки використовується множина зайнятих клітин, яка оновлюється при кожному успішному розміщенні. У разі неможливості знайти валідну позицію за певну кількість

спроб, алгоритм перезапускає процес з початку, обнуляючи поле. Після успішного завершення генерації кораблі зберігаються в об'єкті, відповідаючому за логіку комп'ютерного гравця.

Якщо режим передбачає участь двох людей, аналогічне вікно з вибором способу розташування кораблів відкривається для другого гравця, як вже було зазначено вище. У випадку з ботом цей крок пропускається, і після розстановки кораблів першого гравця одразу ініціалізується основний ігровий цикл.

На цьому етапі формується об'єкт гри, в який передаються обидва гравці та відповідні графічні контролери для візуалізації стану поля. Контролер, відповідальний за відображення поля противника, приховує кораблі, щоб одразу два гравця могли грати з одного пристрою, без необхідності відволікання від екрану. У формі доступно одразу 2 ігрових сітки. Також зверху пишется повідомлення яке інформує гравців, чий хід перший (Рис. 4.7).

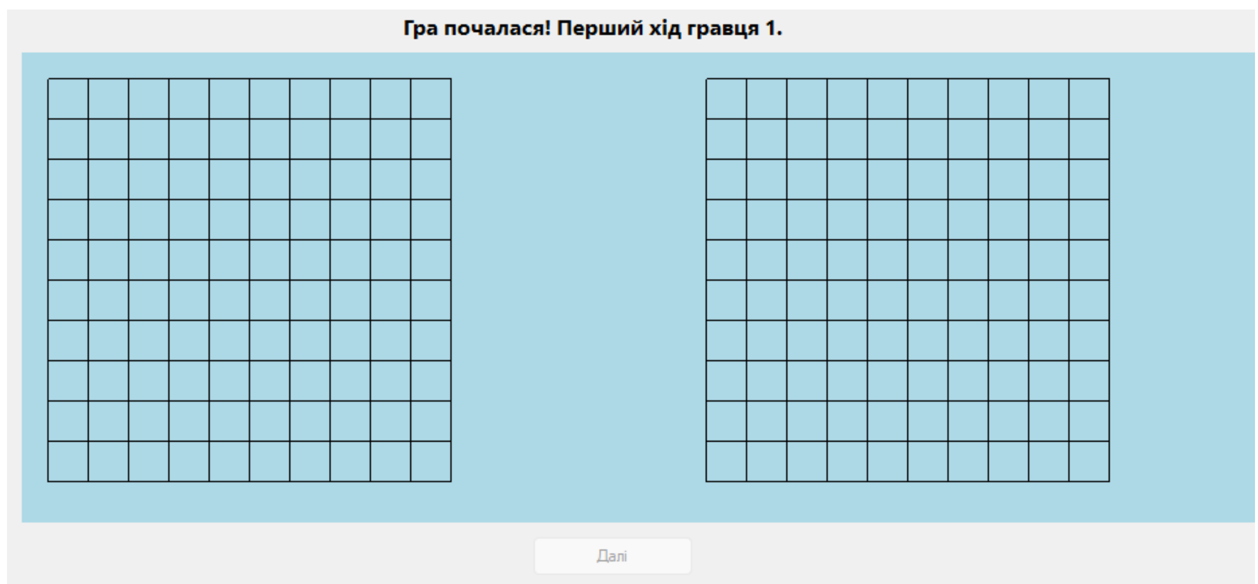

Рис. 4.7. Початок гри Морський бій

Ходи або ж, за термінологією гри, вистріли відбуваються за допомогою звичайного кліку лівої кнопки миші [Додаток Г]. Коли гравець виконує такий клік у свій хід в межах сітки супротивника, програма перевіряє, чи ця клітинка ще не була обстріляна раніше. Якщо вона є новою для поточного гравця, її координати додаються до списку атакованих, і відбувається перевірка: чи потрапив гравець у частину корабля.

Якщо координата співпадає з однією з частин корабля противника, ця частина вважається ураженою, а відповідна клітинка на полі супротивника зафарбовується червоним кольором (Рис. 4.8).

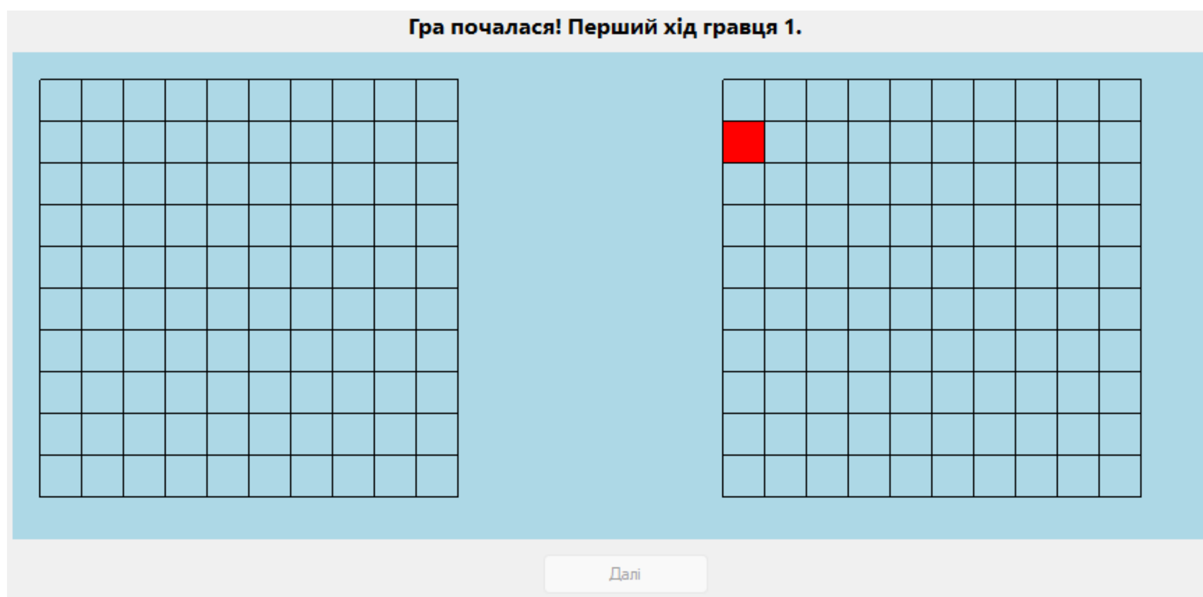

Рис. 4.8. Графічне зображення попадання по кораблю

Коли гравець ране корабель, він має право на додатковий хід, і так до тих пір, поки він не промахнеться. У разі якщо було уражено всі частини одного корабля, корабель вважається потопленим. Додатково навколо нього автоматично позначаються сусідні клітинки, які були заборонені і підсвічувалися червоним під час етапу розстановки. Це означає що стріляти в них більше не потрібно — вони зафарбовуються синім (Рис. 4.9).

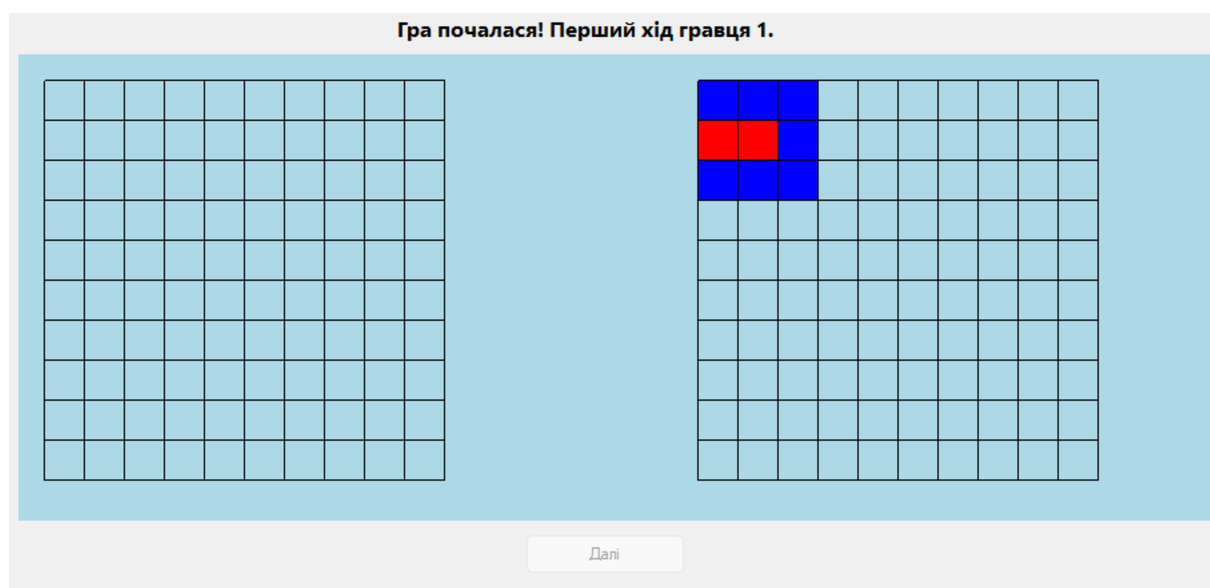

Рис. 4.9. Потоплений корабель

У випадку промаху, тобто коли користувач клікає по клітині і вона не відноситься до частина жодного корабля, і він не знаходиться в цій клітинці, вона просто позначається синім кольором. Після цього хід передається іншому гравцеві, і лише йому дозволяється здійснювати подальші дії. Уся взаємодія з полем суворо контролюється: гравець не може атакувати під час чужого ходу або повторно стріляти в одну й ту саму клітинку (Рис. 4.10).

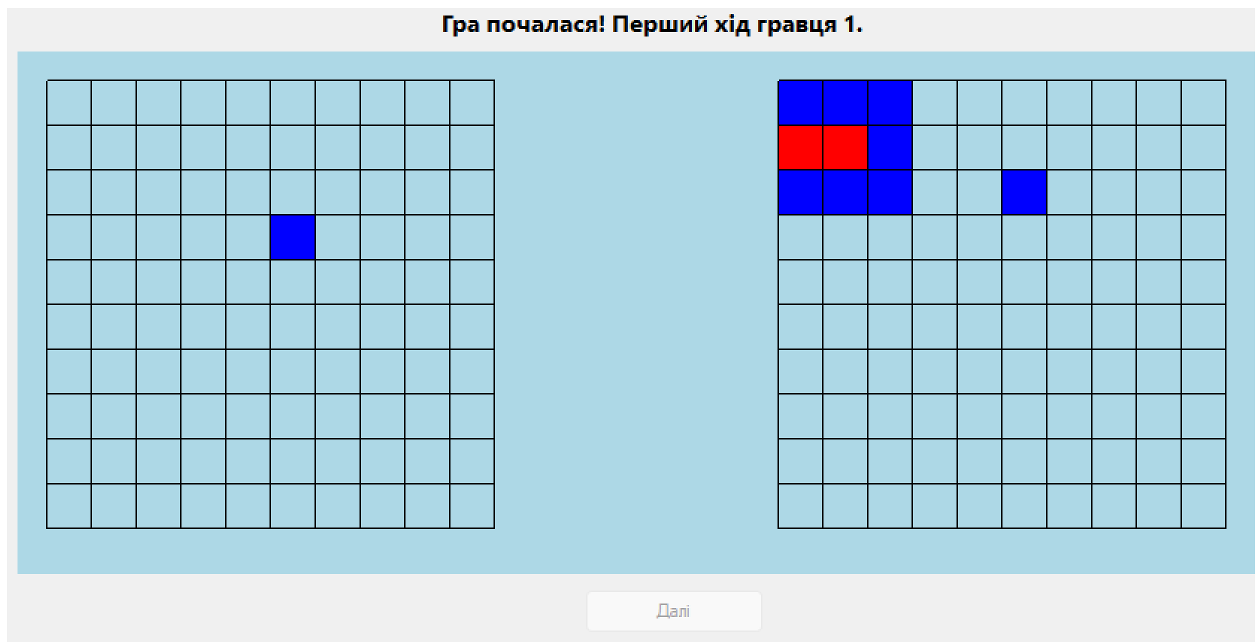

Рис. 4.10. Промашні постріли

Бот виконує ходи автоматично після гравця. Після завершення ходу гравця, система перевіряє чи настав його черговий хід. Якщо так, бот вибирає клітинку для атаки серед тих, які ще не були атаковані. Він не стріляє навмання в уже обрані координати: спочатку перевіряється список попередніх пострілів, і з-поміж вільних клітин вибирається наступна ціль.

На базовому рівні логіка роботи бота є простою - він випадковим чином обирає одну з клітинок, які залишилися, і робить постріл - це викликає таку ж обробку, як і при кліку живого гравця: система визначає, чи була це частина корабля, фарбує клітинку відповідним кольором і перевіряє, чи корабель знищено. Якщо бот промахнувся, право ходу повертається до гравця. Якщо ж він потрапив — ходить знову, доки не промахнеться.

Після завершення гри система переходить до обробки фінального результату. Коли один з гравців знищує всі кораблі суперника, гра

автоматично визначає переможця та програвшого. Ця інформація передається у вигляді спеціального об'єкта, який зберігає обох гравців і кількість ходів, які були здійснені протягом гри.

На цьому етапі відкривається вікно з результатом, окрема форма, в якій відображається підсумкова інформація: хто переміг, хто програв і скільки було зроблено ходів. Якщо в грі брав участь бот, система автоматично підставляє його ім’я, і якщо воно відсутнє в базі даних, створює запис про нього (Рис. 4.11).

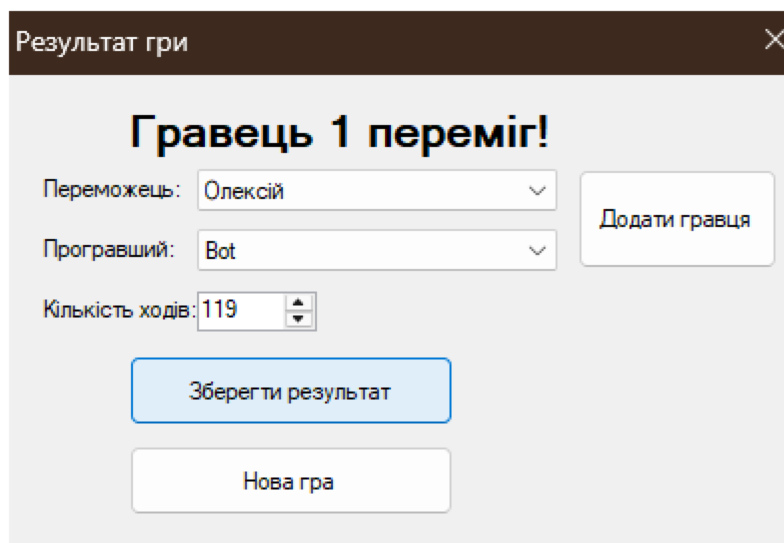

Рис. 4.11. Вікно результатів ги

Гравцеві пропонується зберегти результат, після підтвердження інформації натисканням відповідної кнопки, дані зберігаються у базі: імена гравців, дата, кількість ходів. Якщо гравець не обрав обох учасників або вибрав одного й того самого для обох ролей, система виведе відповідне попередження. Якщо дані введено коректно, система виводе повідомлення про успішне збереження результату (Рис. 4.12).

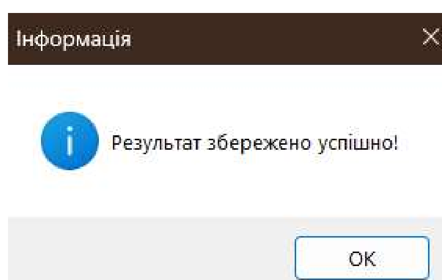

Рис. 4.12. Успішне збереження результату

Форма додавання нового учасника відкривається у вигляді окремого вікна, яке дозволяє користувачу ввести ім’я гравця для подальшого збереження в базі даних [Додаток Б]. У цьому вікні є просте текстове поле для введення імені, кнопка для підтвердження додавання, а також кнопка скасування (Рис. 4.13).

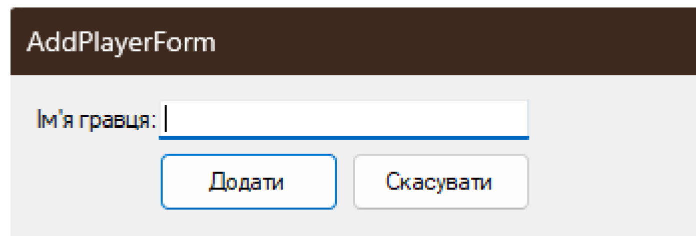

Рис. 4.13. Додавання нового гравця

Користувач має ввести ім’я, якщо це поле залишене порожнім або містить лише пробіли, система виводить попередження з проханням ввести коректне значення (Рис. 4.14).

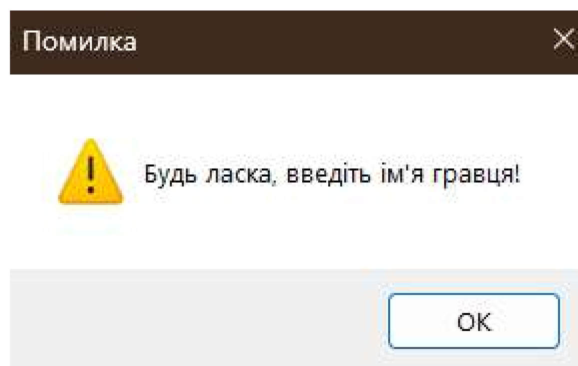

Рис. 4.14. Повідомлення про некоректне введення гравця

У разі успішного введення ім’я зберігається, форма закривається, і новий гравець одразу стає доступним у списках вибору переможця або програвшого в підсумковій формі результатів гри.

Під час запуску гри система перевіряє, чи існує локальна база даних. Якщо база відсутня, вона створюється автоматично, включаючи всі необхідні таблиці. У базі зберігається інформація про гравців та результати зіграних матчів. Таблиці формуються із заздалегідь визначеною структурою: одна з них відповідає за список гравців, інша за збереження результатів кожної гри, включно з переможцем, переможеним, кількістю ходів і часом завершення матчу.

У випадку, якщо база створюється з нуля, у список гравців одразу додається кілька стандартних записів для демонстрації або тестування. Дані зберігаються у форматі, придатному для подальшого аналізу або статистики.

Після завершення кожної гри система фіксує її підсумки: зберігається інформація про те, хто переміг, хто програв, скільки ходів було зроблено, а також точний час завершення партії. Усі дані додаються до відповідної таблиці за допомогою SQL-запитів із параметрами.

Крім запису результатів, система може повертати список усіх зареєстрованих гравців. Це дає змогу підключати імена в інтерфейсі гри, відображати статистику або формувати таблицю лідерів. Також передбачена можливість додавати нових гравців вручну, при створенні нової гри або після завершення гри.

4.2 Інтерфейс та алгоритм розробки сайту

Серверна частина сайту реалізована на базі фреймворку ASP.NET Core і побудована за принципами REST API. Основна функція бекенду приймати запити від клієнтської частини, обробляти їх, взаємодіяти з базою даних та повертати структуровані відповіді. Метою було розробити сервер, який працює з ігровими результатами, зокрема їх збереженням, фільтрацією, сортуванням та статистичним аналізом.

Коли користувач відкриває сторінку з таблицею результатів, фронтенд надсилає запит до сервера із параметрами ідентифікатору гравця, періодом дат, сторінкою та розміром сторінки. Сервер аналізує надані параметри, формує SQL-запит через ORM (Entity Framework Core), виконує вибірку з бази, і повертає відповідь у вигляді JSON-об’єкта. У цій відповіді містяться лише ті записи, які відповідають умовам, а також загальна кількість результатів, що важливо для реалізації пагінації на клієнті.

Під час збереження нового результату гри фронтенд надсилає об’єкт, який містить інформацію про учасників, дату, тривалість гри, місце,

переможця та інші ключові показники. Бекенд приймає цей об’єкт, перевіряє його коректність на обов’язковість полів, допустимі значення, після чого додає його до бази даних і зберігає. Якщо збереження пройшло успішно, сервер повертає підтвердження.

Для оновлення вже наявного запису використовується подібна логіка: клієнт надсилає оновлену інформацію, а сервер шукає відповідний запис у базі, змінює його значення та зберігає оновлення. Якщо запис не знайдено або передані некоректні дані, повертається відповідна помилка.

Фронтенд створено на базі HTML та JS. Він виконує завдання для взаємодії з користувачем, відображенням інтерфейсу, надсилання запитів до сервера та відображення отриманих результатів у зручному вигляді. Інтерфейс поділений на кілька функціональних блоків: таблиця з результатами, фільтри, блоки статистики.

Коли користувач відкриває сторінку з таблицею результатів, відразу ініціюється запит до сервера. У цьому запиті передаються параметри фільтрації, сортування, номер сторінки та кількість записів на сторінці. Після отримання відповіді дані обробляються, і таблиця відображає лише потрібні записи, а також оновлюється інформація про кількість сторінок.

На сторінці перш за все розташовується заголовок з назвою турнірної таблиці "Морский Бій". Він знаходиться у верхній частині інтерфейсу, візуально виділений кольором та лінією, і слугує основним заголовком всієї сторінки.

Нижче розміщується блок зі статистикою, який складається з трьох інформаційних карток: загальна кількість ігор, найкращий гравець і середня кількість ходів у партіях. Ці показники оновлюються автоматично, використовуючи дані з сервера.

Під блоком статистики знаходиться панель фільтрів. Вона дозволяє користувачу обрати конкретного гравця, вказати часовий проміжок (від і до), після чого можна застосувати фільтри або скинути їх. Ця панель впливає на результати, які відображаються в таблиці нижче.

Основну частину займає таблиця з результатами ігор (Рис. 4.15).

Рис. 4.15. Сторінка з турнірною таблицею

У ній вказано дату і час гри, переможця, того хто програв, а також кількість ходів у матчі. Рядки в таблиці оновлюються згідно з фільтрами або після отримання нових даних із сервера.

Під таблицею розташовані кнопки пагінації для переходу між сторінками результатів. Кнопки дозволяють перемикатися між попередньою та наступною сторінками, якщо кількість записів перевищує обсяг однієї сторінки.

Таблиця результатів оновлюється динамічно кожні 30 секунд, щоб користувач завжди бачив актуальні дані без потреби оновлювати сторінку вручну. Це забезпечує зручність та своєчасне відображення нових ігор, змін у статистиці та оновлень після фільтрації чи редагування записів.

Висновки до розділу 4

У процесі розробки гри було реалізовано класичну логіку «Морського бою» з підтримкою двох гравців, поетапного розміщення флоту, обробки пострілів та визначення переможця за умов знищення всіх кораблів

супротивника. Основна ігрова механіка була реалізована з урахуванням валідації координат, запобігання перетину кораблів та перевірки коректності розміщення відповідно до стандартних правил гри.

Турнірна таблиця реалізована як окремий модуль, який взаємодіє з базою даних, отримуючи список завершених ігор. Було реалізовано базову аналітику: підрахунок загальної кількості ігор, визначення найуспішнішого гравця за кількістю перемог та обчислення середньої кількості ходів на гру. Інтерфейс таблиці включає систему фільтрації за гравцем та датами, а також функціонал пагінації для зручності перегляду великого обсягу даних.

ВИСНОВКИ

У результаті виконання дипломної роботи було повністю реалізовано веб застосунок, що поєднує класичну гру «Морський бій» з системою збереження й аналізу результатів у форматі турнірної таблиці. Проект охоплює розробку клієнтської та серверної частин, взаємодію з базою даних, а також побудову динамічного інтерфейсу для користувача.

Було успішно реалізовано ігрову логіку із дотриманням правил гри, механізми розміщення флоту, обробки ходів та визначення переможця. Графічний інтерфейс гри забезпечує взаємодію користувача з полем бою. Кожна завершена гра зберігається у базі даних із повною інформацією про учасників, кількість ходів та часову мітку.

Система турнірної таблиці включає функції фільтрації, сортування та пагінації, що дозволяє зручно аналізувати ігрову статистику. Було реалізовано автоматичне оновлення таблиці кожні 30 секунд без перезавантаження сторінки, що підвищує зручність перегляду результатів у режимі майже реального часу.

У роботі застосовано сучасні технології веброзробки, включаючи HTML, CSS, JavaScript для фронтенду, а також серверну частину з використанням відповідного середовища виконання та бази даних. Проект структуровано таким чином, щоб забезпечити підтримуваність коду й можливість подальшого розширення функціоналу.

Поставлені завдання дипломної роботи були виконані повністю, а її результати демонструють практичне застосування знань у галузі вебпрограмування, розробки інтерактивних інтерфейсів та роботи з даними.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Абрамян М. Э. Visual C# на примерах / Абрамян М. Э. – СПб. : БХВ-Петербург, 2008. – 496 с.
2. Агуров П. В. C#. Разработка компонентов в MS Visual Studio 2005/2008 / Агуров П. В. – СПб. : БХВ-Петербург, 2008. – 480 с.
3. Зеленков Ю. Введение в базы данных / Зеленков Ю.– СПб. : Питер, 2003.
4. Кузнецов С. Д. Основы современных баз данных / КузнецовС. Д. – М. : Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний. – 2007. – 484 с.
5. Нейгел К. C# 2005 для профессионалов / Нейгел К., Ивьен Б., Глинн Дж., Уотсон К. – Диалектика, 2006. – 763 с.
6. Павловская Т. А. C#. Программирование на языке высокого уровня : Учебник для вузов / Павловская Т. А. – СПб. : Питер, 2007. – 432 с.
7. Петцольд Ч. Программирование для Microsoft Windows на C# / Петцольд Ч.– М. : ИТД «Русская Редакция», 2002. – 576 с.
8. Прайс Дж. Visual C# NET. Полное руководство / Дж. Прайс, М. Гандэрлой. – КОРОНА-принт, 2004. – 960 с.
9. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft NET Framework 4.5 на языке C# /Рихтер Дж. – Питер, 2013. – 896 с.
10. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft NET Framework 2.0 на языке C# / Рихтер Дж. – Питер, 2007. – 656 с.
11. Цехнер Марио. Программирование игр под Android / Цехнер Марио. – Питер–Москва, 2012. – 688 с.
12. Платформа ADO. NET Entity Framework [Електронний ресурс]. – Режим доступа : http://msdn.microsoft.com/ru-ru/library/bb399572.aspx.
13. SQLite Documentation [Електронний ресурс]. – Режим доступа : http://www.sqlite.org/docs.html.
14. System.Data.SQLite [Електронний ресурс]. – Режим доступа : http://sqlite.phxsoftware.com/readme.html.

ДОДАТКИ

Додаток А

Об'єкт реалізуючий бота

```
public partial class SeaBattleWindowBot : Form
{
    private IGame currentGame;
    private Player player1;
    private BotPlayer bot;
    private SeaBattleGridControl grid1;
    private SeaBattleGridControl grid2;
    private Button actionButton;
    private Label statusLabel;
    private bool playerPlaced = false;

    public SeaBattleWindowBot()
    {
        InitializeComponent();

        player1 = new Player(1);
        bot = new BotPlayer(2);

        grid1 = new SeaBattleGridControl(500, 350, player1) { Location = new Point(20, 40) };
        grid2 = new SeaBattleGridControl(510, 350, bot) { Location = new Point(510, 40) };

        grid2.HideShips();

        Controls.Add(grid1);
        Controls.Add(grid2);

        actionButton = new Button
        {
            Text = "Далі",
            Location = new Point(400, 400),
            Size = new Size(120, 30),
            Enabled = false
        };
        actionButton.Click += ActionButton_Click;
        Controls.Add(actionButton);

        statusLabel = new Label
        {
            Text = "Гравець: розстановка кораблів",
```

```
            Location = new Point(300, 10),
            Size = new Size(300, 20),
            Font = new Font("Segoe UI", 10, FontStyle.Bold)
        };
        Controls.Add(statusLabel);

        AskShipPlacementMethod();
    }

    private void AskShipPlacementMethod()
    {
        var result = MessageBox.Show(
            "Як ви хочете розставити кораблі?\n\nНатисніть 'Так' для ручної розстановки\nабо 'Ні' для
автоматичної.",
            "Розстановка кораблів",
            MessageBoxButtons.YesNo,
            MessageBoxIcon.Question
        );

        if (result == DialogResult.Yes)
        {
            MessageBox.Show("Розташуйте кораблі вручну і натисніть 'Далі'.", "Вручну");
        }
        else
        {
            grid1.DrawRandomShips();
            MessageBox.Show("Кораблі розставлені автоматично. Натисніть 'Далі'.", "Авто");
        }

        actionButton.Enabled = true;
    }

    private void ActionButton_Click(object sender, EventArgs e)
    {
        if (!playerPlaced)
        {
            if (!grid1.AreShipsPlaced())
            {
                MessageBox.Show("Будь ласка, розставте всі кораблі!", "Увага");
                return;
            }
```

Продовження додатку А

```
            grid1.HideShips();
            playerPlaced = true;

            StartGame();
        }
    }

    private void StartGame()
    {
        currentGame = new PlayerVsBotGame(player1, bot, grid1, grid2);
        currentGame.GameOver += CurrentGame_GameOver;
        currentGame.StartGame();

        actionButton.Enabled = false;
        statusLabel.Text = "Гра почалася! Перший хід гравця.";
    }

    private void CurrentGame_GameOver(object sender, GameOverEventArgs e)
    {
        using (var resultForm = new GameResultForm(e.Winner, e.Loser, e.MovesCount))
        {
            if (resultForm.ShowDialog() == DialogResult.OK)
            {               ResetGame();            }        }     }

    private void ResetGame()
    {
        player1 = new Player(1);
        bot = new BotPlayer(2);

        grid1.ResetGrid(player1);
        grid2.ResetGrid(bot);
        grid2.HideShips();

        actionButton.Text = "Далі";
        actionButton.Enabled = false;
        statusLabel.Text = "Гравець: розстановка кораблів";
        playerPlaced = false;

        AskShipPlacementMethod();
    }
}
```

Додаток Б

Клас-помічник для роботи з БД

```
internal class DatabaseHelper
{
    private string connectionString;

    public DatabaseHelper(string connectionString)
    {
        this.connectionString = connectionString;
        EnsureDatabaseExists();
    }

    private void EnsureDatabaseExists()
    {
        string dbPath = connectionString.Replace("Data Source=", "").Replace("|DataDirectory|\\", "");

        if (connectionString.Contains("|DataDirectory|"))
        {
            string baseDir = AppDomain.CurrentDomain.BaseDirectory;
            dbPath = Path.Combine(baseDir, dbPath);
        }

        if (!File.Exists(dbPath))
        {
            SQLiteConnection.CreateFile(dbPath);

            using (SQLiteConnection connection = new SQLiteConnection($"Data
Source={dbPath};Version=3;"))
            {
                connection.Open();

                string createPlayersTable = @"
                CREATE TABLE IF NOT EXISTS Players (
                    PlayerId INTEGER PRIMARY KEY AUTOINCREMENT,
                    Name TEXT NOT NULL
                )";

                using (SQLiteCommand command = new SQLiteCommand(createPlayersTable,
connection))
                {
                    command.ExecuteNonQuery();
```

```
                }

                string createGameResultsTable = @"
                CREATE TABLE IF NOT EXISTS GameResults (
                    ResultId INTEGER PRIMARY KEY AUTOINCREMENT,
                    WinnerPlayerId INTEGER NOT NULL,
                    LoserPlayerId INTEGER NOT NULL,
                    GameDateTime TEXT NOT NULL,
                    MovesCount INTEGER NOT NULL,
                    FOREIGN KEY (WinnerPlayerId) REFERENCES Players (PlayerId),
                    FOREIGN KEY (LoserPlayerId) REFERENCES Players (PlayerId)
                )";

                using (SQLiteCommand command = new SQLiteCommand(createGameResultsTable,
connection))
                {
                    command.ExecuteNonQuery();
                }

                string addDefaultPlayers = @"
                INSERT OR IGNORE INTO Players (PlayerId, Name) VALUES
                (1, 'Гравець 1'),
                (2, 'Гравець 2'),
                (3, 'Гравець 3'),
                (4, 'Гравець 4')";

                using (SQLiteCommand command = new SQLiteCommand(addDefaultPlayers,
connection))
                {
                    command.ExecuteNonQuery();
                }
            }
        }
    }

    public void SaveGameResult(int winnerPlayerId, int loserPlayerId, DateTime gameDateTime, int
movesCount)
    {
        using (SQLiteConnection connection = new SQLiteConnection(connectionString +
";Version=3;"))
        {
```

Продовження додатку Б

```
            try
            {
                connection.Open();
                string query = @"
                INSERT INTO GameResults (WinnerPlayerId, LoserPlayerId, GameDateTime,
MovesCount)
                VALUES (@WinnerPlayerId, @LoserPlayerId, @GameDateTime, @MovesCount)";

                using (SQLiteCommand command = new SQLiteCommand(query, connection))
                {
                    command.Parameters.AddWithValue("@WinnerPlayerId", winnerPlayerId);
                    command.Parameters.AddWithValue("@LoserPlayerId", loserPlayerId);
                    command.Parameters.AddWithValue("@GameDateTime",
gameDateTime.ToString("yyyy-MM-dd HH:mm:ss"));
                    command.Parameters.AddWithValue("@MovesCount", movesCount);
                    command.ExecuteNonQuery();
                }
            }
            catch (Exception ex)
            {
                throw new Exception($"Помилка збереження результатів: {ex.Message}", ex);
            }
        }
    }

    public System.Data.DataTable GetPlayers()
    {
        var dataTable = new System.Data.DataTable();

        using (SQLiteConnection connection = new SQLiteConnection(connectionString +
";Version=3;"))
        {
            connection.Open();
            string query = "SELECT PlayerId, Name FROM Players ORDER BY PlayerId";

            using (SQLiteCommand command = new SQLiteCommand(query, connection))
            {
                using (SQLiteDataAdapter adapter = new SQLiteDataAdapter(command))
                {
                    adapter.Fill(dataTable);
                }
```

```
            }
        }

        return dataTable;
    }

    public int AddPlayer(string playerName)
    {
        using (SQLiteConnection connection = new SQLiteConnection(connectionString +
";Version=3;"))
        {
            connection.Open();
            string query = "INSERT INTO Players (Name) VALUES (@Name); SELECT
last_insert_rowid();";

            using (SQLiteCommand command = new SQLiteCommand(query, connection))
            {
                command.Parameters.AddWithValue("@Name", playerName);
                return Convert.ToInt32(command.ExecuteScalar());
            }
        }
    }
}
```

Додаток В

Інтерфейс гри

```
internal interface IGame
{
    void StartGame();
    bool IsGameOver();
    void HandleCellClick(Point target);
    int GetCurrentPlayerIndex();
    event EventHandler<GameOverEventArgs> GameOver;
}
```

Додаток Г

Логіка гри з людиною

```
  internal class PlayerVsPlayerGame : IGame
{
    private Player player1;
    private Player player2;
    private int currentPlayerIndex = 1;
    private bool isGameOver = false;
    private HashSet<Point> attackedCellsPlayer1 = new HashSet<Point>();
    private HashSet<Point> attackedCellsPlayer2 = new HashSet<Point>();
    private int totalMoves = 0;

    private SeaBattleGridControl grid1;
    private SeaBattleGridControl grid2;

    public event EventHandler<GameOverEventArgs> GameOver;

    public PlayerVsPlayerGame(Player p1, Player p2, SeaBattleGridControl grid1, SeaBattleGridControl
grid2, bool isBotGame = false)
    {
        player1 = p1;
        player2 = p2;
        this.grid1 = grid1;
        this.grid2 = grid2;

        grid1.OnCellClicked += HandleCellClick;
        grid2.OnCellClicked += HandleCellClick;
    }

    public Player GetCurrentPlayer()
    {
        return currentPlayerIndex == 1 ? player1 : player2;
    }

    public void StartGame()
    {
        currentPlayerIndex = 1;
        isGameOver = false;
        totalMoves = 0;
        attackedCellsPlayer1.Clear();
        attackedCellsPlayer2.Clear();
```

```
        grid1.SetCanClick(false);
        grid2.SetCanClick(true);
    }

    public bool IsGameOver()
    {
        return isGameOver;
    }

    public void HandleCellClick(Point target)
    {
        if (isGameOver) return;

        HashSet<Point> currentPlayerAttackedCells = (currentPlayerIndex == 1) ? attackedCellsPlayer1 :
attackedCellsPlayer2;

        if (currentPlayerAttackedCells.Contains(target)) return;

        totalMoves++;
        currentPlayerAttackedCells.Add(target);
        bool hit = false;

        Player opponent = (currentPlayerIndex == 1) ? player2 : player1;
        SeaBattleGridControl opponentGrid = (currentPlayerIndex == 1) ? grid2 : grid1;

        foreach (var ship in opponent.GetShips())
        {
            foreach (var part in ship.Parts)
            {
                int x = (part.X - 20) / 30;
                int y = (part.Y - 20) / 30;

                if (x == target.X && y == target.Y)
                {
                    ship.RegisterHit(target);
                    opponentGrid.UpdateCellColor(target, Color.Red);
                    hit = true;
                    break;
                }
            }
```

```
        if (hit)
        {
            if (ship.IsSunk())
            {
                MarkShipAsSunk(ship, currentPlayerAttackedCells);
                var surroundingCells = opponentGrid.GetSurroundingCells(ship);

                foreach (var cell in surroundingCells)
                {
                    if (!currentPlayerAttackedCells.Contains(cell))
                    {
                        opponentGrid.UpdateCellColor(cell, Color.Blue);
                    }
                }
            }
            break;
        }
    }

    if (!hit)
    {
        opponentGrid.UpdateCellColor(target, Color.Blue);
    }

    CheckGameOver();

    if (!isGameOver && !hit)
    {
        currentPlayerIndex = (currentPlayerIndex == 1) ? 2 : 1;
    }

    grid1.SetCanClick(currentPlayerIndex == 2);
    grid2.SetCanClick(currentPlayerIndex == 1);
}

private void CheckGameOver()
{
    if (player1.GetShips().All(ship => ship.IsSunk()))
    {
        isGameOver = true;
        OnGameOver(player2, player1, totalMoves);
```

```
    }
    else if (player2.GetShips().All(ship => ship.IsSunk()))
    {
        isGameOver = true;
        OnGameOver(player1, player2, totalMoves);
    }
  }

  protected virtual void OnGameOver(Player winner, Player loser, int moves)
  {
    GameOver?.Invoke(this, new GameOverEventArgs(winner, loser, moves));
  }

  private void MarkShipAsSunk(Ship ship, HashSet<Point> currentPlayerAttackedCells)
  {
    foreach (var part in ship.Parts)
    {
        currentPlayerAttackedCells.Add(part);
    }
  }

  public int GetCurrentPlayerIndex()
  {
    return currentPlayerIndex;
  }

  public Player GetPlayer()
  {
    return currentPlayerIndex == 1 ? player1 : player2;
  }
}
```

Додаток Д

Логіка гри з ботом

```
internal class PlayerVsBotGame : IGame
{
    private Player player1;
    private BotPlayer bot;
    private int currentPlayerIndex = 1;
    private bool isGameOver = false;
    private HashSet<Point> attackedCellsPlayer1 = new HashSet<Point>();
    private HashSet<Point> attackedCellsBot = new HashSet<Point>();
    private int totalMoves = 0;

    private SeaBattleGridControl grid1;
    private SeaBattleGridControl grid2;

    public event EventHandler<GameOverEventArgs> GameOver;

    public PlayerVsBotGame(Player p1, BotPlayer bot, SeaBattleGridControl grid1,
SeaBattleGridControl grid2)
    {
        player1 = p1;
        this.bot = bot;
        this.grid1 = grid1;
        this.grid2 = grid2;

        grid1.OnCellClicked += HandleCellClick;
        grid2.OnCellClicked += HandleCellClick;
    }

    public void StartGame()
    {
        currentPlayerIndex = 1;
        isGameOver = false;
        totalMoves = 0;
        attackedCellsPlayer1.Clear();
        attackedCellsBot.Clear();

        grid1.SetCanClick(false);
        grid2.SetCanClick(true);

        bot.PlaceShipsRandomly();
```

```
        }

        public bool IsGameOver()
        {
            return isGameOver;
        }

        public void HandleCellClick(Point target)
        {
            if (isGameOver) return;

            HashSet<Point> currentPlayerAttackedCells = (currentPlayerIndex == 1) ? attackedCellsPlayer1
: attackedCellsBot;

            if (currentPlayerAttackedCells.Contains(target)) return;

            totalMoves++;
            currentPlayerAttackedCells.Add(target);
            bool hit = false;

            Player opponent = (currentPlayerIndex == 1) ? bot : player1;
            SeaBattleGridControl opponentGrid = (currentPlayerIndex == 1) ? grid2 : grid1;

            foreach (var ship in opponent.GetShips())
            {
                foreach (var part in ship.Parts)
                {
                    int x = (part.X - 20) / 30;
                    int y = (part.Y - 20) / 30;

                    if (x == target.X && y == target.Y)
                    {
                        ship.RegisterHit(target);
                        opponentGrid.UpdateCellColor(target, Color.Red);
                        hit = true;
                        break;
                    }
                }

                if (hit)
                {
```

```
            if (ship.IsSunk())
            {
                MarkShipAsSunk(ship, currentPlayerAttackedCells);
                var surroundingCells = opponentGrid.GetSurroundingCells(ship);

                foreach (var cell in surroundingCells)
                {
                    if (!currentPlayerAttackedCells.Contains(cell))
                    {
                        opponentGrid.UpdateCellColor(cell, Color.Blue);
                    }
                }
            }
            break;
        }
    }

    if (!hit)
    {
        opponentGrid.UpdateCellColor(target, Color.Blue);
    }

    CheckGameOver();

    if (!isGameOver && !hit)
    {
        currentPlayerIndex = (currentPlayerIndex == 1) ? 2 : 1;
    }

    if (!isGameOver && currentPlayerIndex == 2)
    {
        HandleBotTurn();
    }
}

private void HandleBotTurn()
{
    Point botMove = bot.MakeMove();
    HandleCellClick(botMove);

    if (!isGameOver)
```

```
        {
            currentPlayerIndex = 1;
            grid2.SetCanClick(true);
        }
    }

    private void CheckGameOver()
    {
        if (player1.GetShips().All(ship => ship.IsSunk()))
        {
            isGameOver = true;
            OnGameOver(bot, player1, totalMoves);
        }
        else if (bot.GetShips().All(ship => ship.IsSunk()))
        {
            isGameOver = true;
            OnGameOver(player1, bot, totalMoves);
        }
    }

    protected virtual void OnGameOver(Player winner, Player loser, int moves)
    {
        GameOver?.Invoke(this, new GameOverEventArgs(winner, loser, moves));
    }

    private void MarkShipAsSunk(Ship ship, HashSet<Point> currentPlayerAttackedCells)
    {
        foreach (var part in ship.Parts)
        {
            currentPlayerAttackedCells.Add(part);
        }
    }

    public int GetCurrentPlayerIndex()
    {
        return currentPlayerIndex;
    }
}
```