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

Короля Вячеслава Ігоровича
(прізвище, ім'я, по батькові здобувача)

на тему «Розробка гри-головоломки "2048"»
(повна назва теми)

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

науковий керівник к.е.е., доц. _____ Лисенко В. С.
(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 Лисенко В.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, Король Вячеслав Ігорович, підтримую політику Державного університету економіки і технологій з академічної доброчесності і відкритого доступу.

Засвідчую, що кваліфікаційна бакалаврська робота «Розробка гри-головоломки “2048”» 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-головоломки “2048”»

Керівник роботи к.е.н., доцент Лисенко В.С.
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: гра 2048

Предмет дослідження: алгоритми головоломки

Мета кваліфікаційної роботи: *розробка програмного забезпечення гри-головоломки 2048*

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 _____ __Лисенко В.С.___
(підпис) (прізвище та ініціали)

Завдання одержав _____ ___Король. В.І.
_
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

на кваліфікаційну бакалаврську роботу

«Розробка гри-головоломки “2048”»

Короля Вячеслава Ігоровича

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-головоломки “2048”: десктопна та веб версія. Програмний додаток розроблено на мові C# з використанням бібліотеки OpenGL на основі GlControl панелі для роботи з тривимірною графікою. Веб-додаток розроблений з використанням WebGL та технології MySQL для роботи з базами даних.

Ключові слова: ГРА, ГОЛОВОЛОМКА, ГРАФІКА, 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

Головоломка 2048 є цифровою грою-симуляцією дискретного процесу. Кожен крок змінює стан ігрового поля за простими правилами. Результат залежить від попередньо виконаних дій. Ігровий процес складається з задачі з обмеженим числом комбінацій і випадковими початковими умовами. Це створює середовище для побудови моделей із простими правилами, але складним простором рішень.

У наукових колах гра 2048 використовується як тестова платформа для оцінки алгоритмів навчання. Наприклад, у дослідженні Гунга Гуея запропоновано оптимістичне навчання з різницею тимчасових оцінок, яке дозволило досягти високих результатів у грі [1]. Також досліджуються інші підходи, такі як n-кортежні мережі, пошук Монте-Карло та глибоке навчання штучних інтелектів.

2048 не є просто розважальним продуктом. Це модель, що дає змогу спостерігати динаміку у замкненому просторі рішень. Такі задачі часто виникають у прикладній інформатиці, наприклад, в аналізі даних, логістичних задачах або плануванні обмежених ресурсів.

З математичної точки зору, гра 2048 є NP-складною. Це означає, що знайти оптимальну стратегію для досягнення певного результату є обчислювально складним завданням. Така складність робить гру привабливою для дослідження алгоритмів.

У межах дипломної роботи буде розроблено дві програмні реалізації гри. Перша — десктопна, створена мовою C# із використанням OpenGL і GlControl. Друга — веб-версія, що базується на WebGL. Для збереження даних використано MySQL. Програмні модулі реалізовано незалежно, що дозволяє адаптувати їх до інших середовищ.

Об'єкт дослідження — гра 2048. Предмет дослідження — алгоритми, що лежать в основі механіки гри. Мета роботи — створення програмного забезпечення, яке реалізує гру 2048 з урахуванням її алгоритмічних особливостей.

Завдання дослідження:

1) Сформулювати модель гри як дискретну систему зі змінним станом.
2) Побудувати алгоритм обробки ходів і конфліктів.
3) Реалізувати графіку гри на двох платформах.
4) Розробити базу даних для зберігання результатів.
5) Забезпечити взаємодію між логікою гри та графічним інтерфейсом.

Актуальність завдання полягає в поєднанні практичного програмування з теоретичним аналізом алгоритмів. У задачах із динамічним середовищем виникає потреба у стабільних та передбачуваних алгоритмах взаємодії з простором станів. Ігри з послідовною зміною конфігурацій використовуються у дослідженнях адаптивних стратегій. У таких структурах вивчається залежність між локальними діями та довгостроковими результатами. Гра 2048 виконує функцію обмеженої симуляції, де стан системи змінюється після кожного циклу. Механізм гри дозволяє моделювати вплив локальної дії на глобальний результат без зовнішніх вхідних параметрів.

У межах прикладного програмування така задача дозволяє протестувати способи організації внутрішньої логіки, синхронізації графічної частини з процесором подій, а також реалізацію взаємодії з базою даних. У середовищах, орієнтованих на перенесення функціоналу між платформами, виникає потреба у паралельному проектуванні під різні технології. Порівняння реалізацій для десктопу та браузера буде проведене з метою оцінити технічні обмеження та варіанти розв’язання аналогічних задач в умовах різних середовищ.

РОЗДІЛ 1
ПОСТАНОВКА ЗАДАЧІ

1.1. Алгоритмічна модель гри

Гра 2048 моделюється як дискретна стохастична система з обмеженим простором станів. Ігрове поле розміром 4×4 представлено як матриця, де кожна клітинка містить ціле числу ступеню двійки. Кожен хід змінює конфігурацію поля через детерміновані правила переміщення та об'єднання клітинок, а також через додавання нових плиток у випадкові позиції.

Кожен стан гри визначається розташуванням чисел у клітинках поля. Загальна кількість можливих станів обмежена, але велика, тому необхідний повний перебір. З математичної точки зору, це означає появу елементів недетермінізму. Таку систему можна описати термінами MDP — Markov Decision Process [2], а переходи між станами — ребрам з певними ймовірностями. Тут стан системи не фіксується повністю дією гравця, а формується частково під впливом імовірнісних факторів. Для аналізу таких систем використовують політики — правила вибору дій, що максимізують очікувану вигоду.

Додавання нових плиток після кожного ходу моделюється як стохастичний процес з відомими ймовірностями: плитка з числом 2 з'являється з імовірністю 0.9, а з числом 4 — з імовірністю 0.1.

У загальному випадку кількість можливих конфігурацій поля надзвичайно велика. Але простір станів лишається скінченним. Цей простір можна описати як орієнтований граф. Вершини графа — це всі допустимі конфігурації поля. Ребра відповідають можливим переходам між станами в результаті ходів гравця та генерації нових плиток.

Граф має стохастичну природу. Не кожен хід має один наслідок. Додавання плитки відбувається з випадковим розміщенням і числовим значенням. Таку модель можна теж розглядати як дискретний Марковський

процес. У цьому процесі ймовірність наступного стану залежить лише від поточного. Це дозволяє використовувати методи марковського планування для аналізу та прогнозування гри.

На практиці досліджуються два основні типи стратегій — жадібні та багатокрокові. Жадібні обирають хід, що дає найбільший приріст за одне оновлення. Багатокрокові оцінюють наслідки кількох послідовних дій, формуючи дерево варіантів. Для таких стратегій застосовуються методи пошуку з обмеженням глибини, наприклад expectimax. Цей алгоритм комбінує дії гравця та випадкові події, обчислюючи середнє очікуване значення виграшу.

При кожному ході всі плитки на полі переміщуються в обраному напрямку до максимально можливого ступеня. Якщо дві плитки з однаковими числами стикаються під час переміщення, вони об'єднуються в одну плитку з подвоєним значенням. Повторне злиття у межах одного ходу не дозволяється. Кожна пара може бути об’єднана лише один раз за цикл.

Операція злиття — детермінована. Її можна описати як послідовність перестановок з умовною логікою. Перша частина алгоритму проходить по рядках або стовпцях, визначаючи напрямок руху. Друга — виконує складання чисел і позначення оброблених комірок. Після завершення переміщення та об'єднання, у випадковій порожній клітинці з'являється нова плитка зі значенням 2 або 4.

Після завершення ходу в поле додається нова плитка. Це створює розгалуження дерева можливих станів. Кожен хід генерує кілька потенційних результатів. Ця властивість змушує гравця враховувати не тільки поточну оцінку стану, а й ризики зміни конфігурації.

Крім пошуку, застосовуються евристичні функції. Їхня мета — оцінити поточний стан без повного перебору. Типові функції враховують такі характеристики, як кількість порожніх клітинок, монотонність рядків, розміщення великих чисел в одному куті. Такі функції дають змогу

скоротити час обчислення та зменшити потребу в повному аналізі дерева станів.

1.2. Вивчення алгоритмів існуючих ігор

Гра 2048 у класичному вигляді представлена у вигляді браузерного застосунку [3]. Оригінальна реалізація, опублікована на відкритому репозиторії GitHub, забезпечує повноцінне функціонування гри через взаємодію користувача з HTML-інтерфейсом. У цій версії використано мову JavaScript для логіки, DOM-структуру для візуалізації та подієво-орієнтовану модель управління.

Архітектура гри поділена на логічні та візуальні компоненти. Основна логіка реалізована у вигляді класів, які описують ігрову сітку, плитки, правила злиття та генерацію нових елементів. Обробка натискань клавіш ініціює зміну стану гри. Алгоритм обробки ходу включає три етапи: переміщення, злиття та вставку нової плитки.

Візуальна частина взаємодіє з DOM через оновлення CSS-класів. Анімація побудована на зміні класів та трансформаційних властивостей стилів. Візуал не впливає на логіку, але забезпечує приємну картинку та гарне сприйняття елементів.

Ігрове поле моделюється як двовимірний масив (рис. 1.1.). Кожен елемент може містити числове значення кратне двійці. Зсув плиток виконується шляхом ітерації у напрямку ходу. Алгоритм перевіряє, чи можливе злиття, зберігаючи інформацію про оброблені клітинки, аби уникнути повторного об'єднання у межах одного циклу.

Рис. 1.1. Ігрове поле 2048

Після зсуву та злиття нова плитка генерується випадковим чином (Рис. 1.2.). Використовується псевдовипадковий генератор JavaScript. Імовірність появи плитки зі значенням 2 перевищує ймовірність появи плитки 4. Положення нової плитки обирається з доступних порожніх клітинок.

Рис. 1.2. Генерування нової плитки

Гра завершується у двох випадках: або коли досягнута плитка зі значенням 2048 (рис. 1.3.), хоча гру можна продовжити після перемоги, тому гра фактично безкінечна, або коли жодна дія не призводить до зміни стану (рис. 1.4.). Для перевірки наявності можливого ходу виконується перебір усіх напрямків із симуляцією — без змін до основного стану. Якщо жодна з симуляцій не створює нового стану, гра припиняється.

Рис. 1.3. Перемога в грі 2048

Поточний стан гри зберігається у локальному сховищі браузера. Це реалізовано з цілю збереження прогресу між сесіями. Кожен об'єкт (плитка, рахунок, історія) зберігається у вигляді JSON. Ця модель забезпечує розділення логіки гри та механізму збереження без надлишкового дублювання даних.

У реалізації гри, створеній Ґабріелем Чіруллі, рахунок змінюється виключно в момент об’єднання плиток. При кожному злитті двох плиток зі значенням n утворюється нова плитка зі значенням $2n$. До поточного рахунку

додається саме це нове значення. Наприклад, злиття двох плиток 128 дає плитку 256, і до рахунку додається 256.

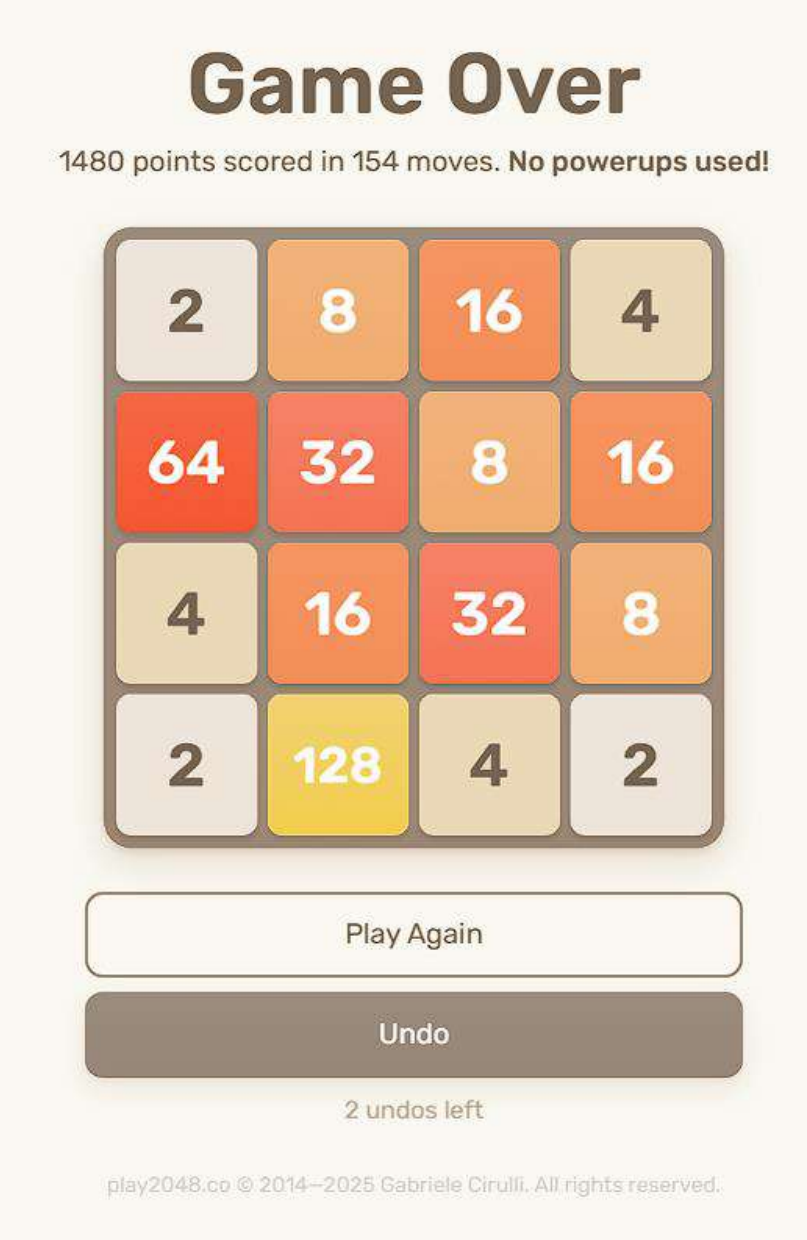

Рис. 1.4. Програш у грі 2048

Це створює асиметричну систему винагороди, де кожне злиття генерує експоненційно зростаючий внесок. Рахунок не залежить від кількості ходів або тривалості гри. Він визначається лише сумою значень усіх успішних об’єднань (рис. 1.5.).

На практиці це стимулює утворення великих плиток. Алгоритм підрахунку не враховує інші аспекти, як-от кількість порожніх клітинок, кількість доступних ходів або кількість дій без злиття. У цьому сенсі рахунок

відображає лише одну характеристику гри — накопичення значень через злиття.

Рис. 1.5. Рахунок гри

У вихідному коді гра веде облік не лише поточного, а й найкращого досягнутого рахунку. Цей показник зберігається у браузерному сховищі. При перезапуску гри зберігається тільки best score. Таким чином реалізується базова система прогресу без серверної частини.

У модифікованих версіях застосовують додаткові критерії оцінки. Один із них — кількість ходів. Цей показник демонструє ступінь оптимальності послідовності дій. Зменшення числа ходів до заданої мети розглядається як індикатор раціональності вибору. Відсутність штрафів за безрезультатні ходи дозволяє досліджувати баланс між швидкістю та результативністю.

Інший підхід передбачає врахування часу, витраченого на виконання ходів. Часові обмеження або оцінка швидкості впливають на стратегії гравців і автоматичних агентів. Це актуально при моделюванні реальних умов експлуатації систем з обмеженими ресурсами.

Деякі дослідження пропонують комплексні евристичні функції для оцінки стану ігрового поля. Параметри включають кількість порожніх клітинок, рівень монотонності значень плиток, різницю між сусідніми значеннями. Вагові коефіцієнти застосовуються для балансування цих характеристик. Результатом стає багатовимірна метрика. Вона інтегрує якісні і кількісні аспекти.

Крім того, існують підходи, які фокусуються на властивостях стратегічної глибини. Максимальна глибина дерева ходів і середня довжина послідовності об’єднань оцінюють інтелектуальний рівень прийняття рішень.

Ці характеристики особливо релевантні для задач навчання з підкріпленням. Їх інтерпретація вимагає контексту і не завжди корелює з користувацьким досвідом.

1.3. Вимоги до гри-головоломки

Веб-версія має моделювати гру як дискретну стохастичну систему з обмеженим простором станів. Ігрове поле представлено матрицею 4×4. Кожна клітинка містить ціле число — ступінь двійки. Кожен хід змінює конфігурацію матриці. Зміни відбуваються за детермінованими правилами переміщення та об’єднання клітинок. Після ходу додається нова плитка. Поява нової плитки є стохастичним процесом із фіксованими ймовірностями. Ймовірність появи плитки з числом 2 становить 90 %, з числом 4 — 10 %. Вибір позиції для нової плитки обмежений порожніми клітинками. Відбір позиції відбувається випадковим чином.

Правила переміщення плиток передбачають поетапну обробку рядків або стовпців. Алгоритм виконує перестановки чисел і об’єднання плиток. Об’єднання для кожної пари плиток допускається лише один раз за хід. Веб-версія повинна підтримувати коректне виконання цього механізму. Відсутність пропусків або дублювань у злитті плиток є обов’язковою.

Простір станів можна описати у вигляді орієнтованого графа. Вершини графа відповідають конфігураціям поля. Ребра графа відображають переходи між станами. Переходи формуються внаслідок дій гравця та стохастичної генерації нових плиток. Стохастична природа гри зумовлює множину можливих наступних станів після кожного ходу.

Веб-додаток повинен реалізувати механізми обчислення евристичних функцій для оцінки стану поля. Функції оцінки мають враховувати кількість порожніх клітинок, монотонність розташування чисел та концентрацію великих значень у кутах. Використання таких функцій сприяє потенційному застосуванню алгоритмів пошуку оптимальних ходів.

Інтерфейс гри відображає актуальний стан матриці після кожного ходу. Відображення синхронізоване з внутрішньою моделлю. Підтримка відповідності між даними моделі та візуалізацією забезпечує коректність гри.

Механізм випадкового вибору позиції та значення плитки реалізує статистичну відповідність заданим ймовірностям. Тестування цього механізму виявляє рівномірність розподілу випадкових подій.

Архітектура веб-реалізації базується на математичній моделі дискретного марковського процесу. Переходи між станами залежать від дій користувача та випадкових факторів.

Десктопний додаток має бути побудований на базі мови програмування C# із використанням OpenGL через GlControl. Необхідно реалізувати графічний інтерфейс із підтримкою двовимірної візуалізації.

Ігровий процес організований через матрицю розміром 4×4, де кожна клітинка зберігає числове значення у вигляді ступеню двійки. Логіка переміщення плиток реалізується послідовно, із застосуванням операцій перестановки та об’єднання чисел відповідно до правил гри. Кожна пара плиток з однаковими значеннями обробляється один раз за хід.

Додаток має обробляти події користувача, зокрема натискання клавіш, для керування напрямком руху плиток. Взаємодія з користувачем повинна відбуватися у реальному часі без затримок.

Графічний інтерфейс повинен мати код відтворення руху плиток із плавними анімаціями. Візуальні ефекти зміни у стані гри повинні прауювати без затримок та лагів. Використання OpenGL забезпечує апаратне прискорення візуалізації.

Внутрішня логіка стохастичного додавання плиток відповідає заданим ймовірностям появи чисел 2 та 4. Відповідність цим ймовірностям перевіряється шляхом статистичного аналізу результатів генерації.

Архітектурно код має бути модульним із розмежуванням логіки гри, обробки введення і відображення. Така структура підвищує зрозумілість, полегшує налагодження та подальше розширення функціоналу.

Орієнтація на десктопну платформу зумовлює оптимізацію ресурсів системи, забезпечення сумісності з ОС Windows, а також підтримку різних роздільностей екрана.

1.4. Теоретичні відомості інструментів розробки

Розробка десктопної версії гри 2048 виконана на мові програмування C#. Ця мова є частиною платформи .NET і відзначається статичною типізацією та об’єктно-орієнтованою парадигмою. C# підтримує сучасні конструкції мови, які спрощують розробку складних програмних систем. Мова надає можливість використання делегатів, подій і асинхронного програмування полегшує обробку користувацьких подій та взаємодію із графічним інтерфейсом. За даними офіційної документації Microsoft, C# забезпечує високу продуктивність завдяки компіляції у проміжний код (Intermediate Language) із подальшою оптимізацією на етапі виконання (Just-In-Time compilation) [4].

Для реалізації графічного інтерфейсу та візуалізації використано бібліотеку OpenGL у поєднанні з GlControl — компонентом з можливостю інтеграції OpenGL-контексту у Windows Forms додатки. OpenGL (Open Graphics Library) є кросплатформеним API для роботи з 2D та 3D графікою. В основі лежить модель керування графічним конвеєром, яка дозволяє програмісту оперувати вершинами, текстурами, шейдерами та іншими елементами візуалізації. Відповідно до джерел, OpenGL забезпечує низькорівневий доступ до графічного апаратного забезпечення. Це робить бібліотеку придатною для реалізації інтерактивних додатків із високими вимогами до продуктивності [5].

Компонент GlControl слугує обгорткою для OpenGL у середовищі .NET, спрощуючи ініціалізацію контексту і взаємодію з формами Windows. Інтеграція дозволяє поєднати апаратне прискорення рендерингу із стандартними елементами управління Windows Forms. Таким чином

спрощується створення користувацького інтерфейсу. Використання GlControl є оптимальним рішенням для проектів, які потребують 3D-візуалізації в десктопних додатках без переходу на більш складні фреймворки.

Для веб-версії гри застосовано технологію WebGL. WebGL — це стандарт, який розширює можливості браузерів, дозволяючи виконувати апаратно прискорену графіку без необхідності встановлення додаткових плагінів. WebGL базується на OpenGL ES, адаптованій версії OpenGL для вбудованих систем. За результатами особистих досліджень, WebGL підтримує роботу із шейдерами, буферами і текстурами, що дозволяє створювати як двовимірні, так і тривимірні сцени у браузерному середовищі.

Для збереження і обробки даних веб-додатку використано систему управління базами даних MySQL. MySQL — це реляційна СУБД із відкритим вихідним кодом, яка широко застосовується для зберігання структурованої інформації. У контексті гри, MySQL забезпечує надійне збереження результатів і статистики. Архітектура СУБД базується на таблицях із чіткою схемою, дає змогу виконувати складні SQL-запити для отримання потрібних даних.

Таким чином, вибір набору інструментів для розробки спрямований на баланс між продуктивністю, функціональністю і сумісністю. Було вирішено використовувати такі програмні засоби, як C# і OpenGL у десктопній версії, та застосовувати WebGL і MySQL для веб-додатку.

Висновки до розділу 1

Отже, у цьому розділі було сформульовано алгоритмічну модель гри 2048 як стохастичної системи з дискретним простором станів. Проаналізовано правила руху і об’єднання плиток, а також механізм підрахунку очок. Визначено вимоги до функціональності гри для різних платформ. Описано інструменти розробки, їхні особливості та роль у створенні десктопної та веб-версій гри.

РОЗДІЛ 2
РОЗРОБКА АЛГОРИТМУ РОЗВ’ЯЗАННЯ ЗАДАЧІ

У першому розділі було розглянуто низку підходів до моделювання та аналізу гри 2048, моделі на основі Марковських процесів, жадібні та багатокрокові стратегії, а також евристичні методи оцінювання станів гри. Цей розділ присвячено детальному аналізу цих алгоритмічних концепцій із метою побудови власного алгоритму гри, що відображатиме специфіку механіки та стохастичної природи 2048.

Рух плиток у грі 2048 відбувається за двома послідовними діями — зміщенням та об’єднанням. Спершу непорожні плитки зсуваються максимально до одного боку поля, утворюючи суцільний ряд без пропусків. Після цього починається процес злиття. Кожна пара плиток з однаковими значеннями з’єднується один раз за хід, а результатом стає плитка з подвоєним числом. Для уникнення повторних злиттів застосовують маркування оброблених елементів.

Вставка нових плиток залежить від випадкового вибору порожньої клітинки і значення — найчастіше 2, рідше 4. Для цього створюється список вільних позицій, а вибір виконується з урахуванням заданих ймовірностей. Відсутність передбачуваності у розміщенні плиток змінює хід гри.

Інтерфейс між користувачем та грою реалізується як набір команд для кожного напрямку руху. Кожна команда ініціює оновлення стану поля: виконує зміщення, злиття та додавання нової плитки. Керування відбувається через події або функції, що зв’язують логіку з відображенням.

Керування гри буде організоване через обробку подій, які виникають у відповідь на дії користувача. Вхідні сигнали — натискання клавіш напрямків або взаємодія із графічним інтерфейсом. Ці сигнали перетворюються на команди зміщення плиток у відповідну сторону. Вони запускають послідовність логічних операцій над ігровим полем.

Для уникнення дублювання ходів під час обробки вводів треба система блокування, яка дозволяє прийняти нову команду тільки після завершення всіх операцій попередньої. Кожна команда передбачає виклик процедур, що здійснюють зсув і злиття плиток, а також додавання нових елементів у випадкові позиції. Результат кожного кроку негайно відображається у візуальній частині додатку, забезпечуючи відчуття реактивності системи.

Обробка команд виконується у межах циклу подій з підтримкою асинхронності взаємодії з користувачем і гарантує своєчасну реакцію на його дії. Керування станом гри супроводжується перевірками на завершення, результату у вигляді перемоги або поразки. В свою чергу результати перевірок повертають відповідні повідомлення чи дії.

В багатокрокових стратегіях хід розглядається разом з можливими наслідками. Для цього створюється дерево варіантів, де на кожному рівні чергуються дії гравця і випадкові події. Щоб знизити обчислювальне навантаження, можна застосувати збереження проміжних результатів і обмеження глибини аналізу.

Оцінка стану гри використовується для формування рекомендацій. Ця функція аналізує матрицю поля за кількома параметрами: кількістю порожніх клітинок, розташуванням великих плиток, послідовністю значень у рядках і стовпцях. Значення оцінки передається в алгоритм вибору кроку.

Код буде розділено на декілька частин. Кожний шматок буде відповідати за окрему дію: переміщення плиток, об’єднання, додавання випадкових елементів, оцінка стану та управління ходом.

Відповідно, основа реалізації містить послідовність керованих процедур, у яких чергуються детерміновані операції і випадкові події. Мета сформувати динамічний розвиток гри з урахуванням стохастичних факторів і багатокрокового планування.

Враховуючи плюси та мінуси аналізованих методів, пропонується алгоритм гри із використанням евристичних функцій оцінки для зменшення

обчислювальної складності, а також врахуванням ймовірнісної моделі появи нових плиток.

Основні етапи алгоритму:

1) Отримання поточного стану гри. Ігрове поле представлено матрицею 4×4, в якій кожна клітинка містить або 0 (порожня), або ступінь двійки.
2) Генерація можливих ходів. Для кожного з чотирьох напрямків (вгору, вниз, вліво, вправо) симулюються переміщення плиток з урахуванням правил злиття.
3) Оцінка результатів ходів. Для кожного результату застосовуються евристичні функції:
    a) Підрахунок кількості порожніх клітин.
    b) Оцінка монотонності рядків і стовпців.
    c) Розташування найбільших плиток у кутах.
4) Врахування стохастичного фактора. Для кожного ходового стану моделюється ймовірність додавання плитки 2 або 4 у порожні клітинки. Розраховується середнє очікуване значення стану з урахуванням ймовірностей.
5) Вибір ходу. Вибирається напрямок з найбільшим очікуваним значенням за результатами кроків 3-4.
6) Оновлення стану. Виконується обраний хід, відбувається додавання нової плитки.
7) Перевірка завершення гри. Аналізуються можливі ходи. Якщо всі напрямки блоковані, гра завершується.

Описаний алгоритм візуалізований у вигляді блок-схеми (рис. 2.1.).

Рис. 2.1. Алгоритм гри 2048

Схема ілюструє послідовність операцій, умови переходів і ключові рішення. Створений алгоритм буде взято за основу для подальшої імплементації та тестування в рамках розробки гри 2048.

Висновки до розділу 2

Розділ 2 зосереджувався на формуванні алгоритмічної основи для розв’язання задачі гри 2048.

РОЗДІЛ 3
ОРГАНІЗАЦІЯ ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Структура ігрових даних

Гра реалізується через дискретне середовище, представлене квадратною матрицею розміру 4×4. У кожній клітинці зберігається ціле число. Значення 0 відповідає порожній клітинці, інші значення є ступенями числа 2, що виникають внаслідок злиття однакових плиток. Така структура дозволяє зберігати повний стан гри в одній змінній типу list[list[int]].

У кожному такті ігрового циклу відбуваються трансформації цієї матриці. Напрямки переміщення реалізуються як різні режими зсуву та об'єднання рядків або стовпців. Для цього використовуються додаткові функції обробки рядків, які імітують злиття плиток без втрати даних.

Матриця гри є основним вектором стану. Для забезпечення гнучкої обробки, поле зберігається у вигляді вкладеного списку, але для алгоритмічних операцій воно може тимчасово перетворюватися в одномірний вектор.

```
field = [  [2, 0, 2, 4],
           [0, 4, 0, 0],
           [8, 8, 2, 0],
           [0, 0, 0, 0]]
```

Під час кожного ходу відбувається:

1) створення копії масиву для збереження попереднього стану;
2) трансформація рядків або стовпців залежно від напрямку;
3) оновлення масиву з урахуванням нових значень;
4) запис нового елемента у випадкову порожню клітинку.

Матриця поля функціонує як єдине джерело істинного стану гри. Усі зміни, пов’язані з переміщенням, об’єднанням або генерацією плиток, реалізуються через перетворення цього масиву. Початковий стан матриці

формується шляхом ініціалізації нульовими значеннями з наступним записом двох випадкових плиток зі значенням 2 або 4.

Для кожного такту гри передбачено алгоритмічну послідовність, яка включає:

1) створення копії масиву до змін;
2) визначення вектору переміщення;
3) перетворення рядків або стовпців відповідно до напрямку;
4) застосування правил злиття елементів однакового значення;
5) фіксацію зміненого стану;
6) додавання нового елемента у випадкову порожню клітинку.

Злиття відбувається згідно з принципом одноразової агрегації: при одній ітерації рядка або стовпця допускається не більше одного злиття для кожної пари плиток. Кінцевий результат обчислюється без проміжних виводів, з опорою лише на вихідні значення відповідної лінії.

Напрямки обробки (вгору, вниз, вліво, вправо) реалізуються через функції, які трансформують вихідну матрицю в уніфіковану форму для обчислення, після чого виконується базова обробка рядків, і, у разі потреби, результат повертається у вихідну орієнтацію.

Механізм генерації нових плиток базується на пошуку порожніх клітинок із значенням 0 та виборі однієї з них випадковим чином. Значення нової плитки визначається з урахуванням імовірнісного розподілу.

Стан гри визначається винятково значеннями у полі. Додаткові змінні, як поточний рахунок і лічильник ходів, зберігаються окремо й не впливають на основну структуру ігрових даних. У випадках необхідності серіалізації або логування, матриця формується у вигляд одномірного масиву та рядка із фіксованим роздільником.

Операції над матрицею не передбачають змін типів даних, щоб зберігалася компактність і сумісність із форматами збереження. Усі значення залишаються цілими числами. Переважне використання вкладених списків

аргументується простотою доступу до координатних позицій та низькою вартістю модифікації елементів за індексом.

3.2. Управління ігровим циклом

Ігровий цикл описується як послідовність дискретних тактів, кожен з яких відповідає за прийом вхідної команди, оновлення стану гри та вивід результату. Цикл реалізується у вигляді умовно-перервного процесу з тривалістю до досягнення одного з термінальних станів.

Спочатку отримується одна з допустимих команд переміщення. У контексті ручного управління – з клавіатури або графічного інтерфейсу. Перш ніж змінити стан поля, виконується симуляція ходу. У випадку, коли зсув не призводить до жодної зміни у розташуванні чи значеннях клітинок, цикл повертається на фазу очікування, не зберігаючи зміни.

При валідному ході ініціюється трансформація – переміщення плиток у вибраному напрямку з подальшим злиттям однакових значень. Після завершення операцій обирається одна з доступних порожніх клітинок, у яку записується нове значення 2 або 4.

Аналізується поле на предмет завершення гри. У разі досягнення певного граничного значення 2048 фіксується перемога, з можливістю продовження гри. Якщо немає доступних ходів і злиттів, фіксується поразка. Усі логічні перевірки реалізуються окремо і не блокують основний цикл.

Новий стан гри, разом з інформацією про зміну очок, завершеність гри або інші сигнали, передається на інтерфейсний рівень. Таким чином, на кожному такті можна виділити наступні фази:

1) Зчитування вхідної дії
2) Симуляція та перевірка допустимості
3) Оновлення стану гри
4) Перевірка ігрового стану
5) Формування результату такту

Цикл реалізується синхронно, без використання багатопоточності. У разі наявності графічного виводу, анімації або затримки між тактами обробляються окремим модулем візуалізації. Логіка залишає поле незмінним, доки не завершиться повний розрахунок такту.

3.3. Побудова бази даних

Запис проміжних та фінальних результатів, зберігання конфігурацій, журналювання подій гри — усе це потребує впровадження бази даних. У перспективі передбачається використання реляційної системи управління базами даних на основі MySQL.

Початкове проектування здійснюється на рівні інфологічної моделі, де визначаються ключові сутності, зв’язки між ними та основні атрибути. Дані описуються з урахуванням потенційної підтримки мультикористувацької гри, можливості статистичного аналізу та подальшої аналітики на стороні бекенда.

Користувач буде базовим елементом персоніфікованого доступу. В ньому зберігається унікальний ідентифікатор, нікнейм, хеш пароля, атрибути безпеки. Сесія гри — окремий ігровий процес, прив’язаний до користувача. Має часову мітку початку й завершення, фінальний рахунок, статус гри.

Необхідно обов’язково зберігати хід гри. Ця сутність слугуватиме, як одиниця трансформації поля. Для кожної сесії може зберігатися послідовність ходів, включаючи напрямок дії, отриманий рахунок за такт та стан поля до/після. Конфігурація гри — це звичайні параметри запуску.

Подія — додатковий елемент для фіксації системних чи внутрішніх подій.

Із зазначених сутностей формується початкова реляційна схема:

1) Користувач (id, username, password_hash, created_at)
2) Сесія (id, user_id, started_at, ended_at, final_score, status, config_id)

3) Хід (id, session_id, move_index, direction, score_delta, field_before, field_after)
4) Конфігурація (id, board_size, rule_variant)
5) Подія (id, session_id, timestamp, event_type, payload)

На основі системи створюються запити на реалізацію бази даних у СУБД MySQL. Кожним стовпцям задаються типи даних, встановлюються зв’язки між таблицями. Це все утворює схему бази даних.

Таблиця конфігурація гри (табл. 3.1) зберігає налаштування гри, які можуть повторно використовуватися у різних сесіях. Дозволяє гнучко змінювати параметри без дублювання.

Таблиця 3.1

Структура таблиці Конфігурація гри

Поле	Тип даних	Опис
id	Ціле (INT)	Унікальний ідентифікатор конфігурації (PK)
board_size	Ціле (INT)	Розмір ігрового поля
rule_variant	Текстовий (VARCHAR(50))	Варіант правил гри

Користувач (табл. 3.2) зберігає інформацію про користувачів системи. Ключ id унікальний та використовується для зв’язку з іншими таблицями. Поле password_hash містить захищене представлення пароля.

Таблиця 3.2

Структура таблиці Користувач

Поле	Тип даних	Опис
id	Ціле (INT)	Унікальний ідентифікатор користувача (PK)

Продовження табл. 3.2

username	Текстовий (VARCHAR(50))	Логін або нікнейм користувача
password_hash	Текстовий (VARCHAR(255))	Хешований пароль користувача
created_at	Дата/час (DATETIME)	Дата та час реєстрації

Таблиця сесії гри (табл. 3.3) відображає конкретний ігровий процес

Таблиця 3.3

Структура таблиці Сессія гри

Поле	Тип даних	Опис
id	Ціле (INT)	Унікальний ідентифікатор сесії (PK)
user_id	Ціле (INT)	Ідентифікатор користувача (FK на User.id)
started_at	Дата/час (DATETIME)	Час початку сесії
ended_at	Дата/час (DATETIME)	Час завершення сесії (може бути NULL, якщо незавершена)
final_score	Ціле (INT)	Підсумковий рахунок
status	Текстовий (VARCHAR(20))	Статус гри (виграш, поразка, незавершена)
config_id	Ціле (INT)	Ідентифікатор конфігурації (FK на Configuration.id)

. Зв’язок із користувачем та конфігурацією дозволяє відслідковувати індивідуальні ігри та їх параметри.

Хід гри (табл. 3.4) фіксує кожен крок у межах сесії, дозволяючи відтворити логіку гри або провести детальний аналіз. Зберігаються поля до і після трансформації для відновлення стану.

Таблиця 3.4

Структура таблиці Хід гри

Поле	Тип даних	Опис
id	Ціле (INT)	Унікальний ідентифікатор ходу (PK)
session_id	Ціле (INT)	Ідентифікатор сесії (FK на Session.id)
move_index	Ціле (INT)	Номер ходу у сесії (послідовність)
direction	Текстовий (VARCHAR(10))	Напрямок руху (up, down, left, right)
score_delta	Ціле (INT)	Збільшення рахунку в результаті ходу
field_before	Текстовий (TEXT)	Стан поля перед ходом (серіалізований у JSON)
field_after	Текстовий (TEXT)	Стан поля після ходу (серіалізований у JSON)

Таблиця подій (табл. 3.5) відображає внутрішні або системні події, що виникають під час сесії. Буде використана для налагодження, аудиту або логування.

Ключові зв’язки реалізуються за допомогою зовнішніх ключів:

Сесія.user_id і Користувач.id

Хід.session_id і Сесія.id

Сесія.config_id і Конфігурація.id

Подія.session_id і Сесія.id

Таблиця 3.5

Структура таблиці Хід гри

Поле	Тип даних	Опис
id	Ціле (INT)	Унікальний ідентифікатор події (PK)
session_id	Ціле (INT)	Ідентифікатор сесії (FK на Session.id)
timestamp	Дата/час (DATETIME)	Час події
event_type	Текстовий (VARCHAR(50))	Тип події (помилка, повідомлення, завершення)
payload	Текстовий (TEXT)	Додаткова інформація про подію (у вільному форматі)

Зараз у системі використовується лише частина таблиць із запропонованої структури бази даних. Частина сутностей поки що планується використовувати в основній логіці гри, оскільки побудований алгоритм не передбачає їх активного використання. Проте ці таблиці були спроектовані заздалегідь, з урахуванням можливого розвитку гри.

База даних повинна мати повну структуру на етапі проектування тому що це дозволять одразу врахувати всі можливі варіанти використання інформації в майбутньому. Знижується ризик виникнення необхідності суттєвих змін у схемі під час розширення функціоналу і не буде необхідності змінювати структуру вже існуючих таблиць даних. В іншому випадку виникає загроза крашу програми. Якщо виникне потреба аналізувати хід ігор,

впроваджувати відновлення сесій або додавати аналітичні функції, існуюча структура вже підтримуватиме такі задачі без серйозних змін.

Висновки до розділу 3

У третьому розділі розглянуто основні компоненти формування основи програмного забезпечення для гри. У підрозділі про структуру ігрових даних описано модель представлення ігрового поля у вигляді квадратної матриці з цілочисельними значеннями, які відображають поточний стан кожної клітинки.

Було проаналізовано особливості управління ігровим циклом, викладено механізми реалізації послідовності дій гри. Визначено, як відбувається обробка подій користувача, зміна стану гри та оновлення візуального представлення. Розроблена модель циклу враховує обробку анімації, зміщення плиток і логіку їх злиття, що забезпечує узгоджене функціонування ігрового процесу.

Було також описано створення комплексної структури даних на етапі проектування. Наведено опис сутностей, які відповідають основним об’єктам гри, а також визначено реляційні зв’язки між ними.

РОЗДІЛ 4
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1. Розробка C#-застосунку

У цьому підрозділі розглянуто архітектуру та особливості реалізації застосунку гри «2048», створеного з використанням мови C# і бібліотеки OpenTK для графічного виведення (Додаток А). Застосунок є типовим Windows Forms-додатком із використанням GLControl для малювання OpenGL-графіки.

Основні змінні гри:

- tiles — двовимірний масив типу Tile, який представляє собою сітку 4×4, у якій зберігаються плитки.
- textureIds — словник, у якому зберігаються ідентифікатори текстур, створених з чисел.
- score — змінна для підрахунку набраних балів.
- gameStartTime — мітка часу, коли почалася гра, для розрахунку тривалості.
- animationTimer — системний таймер, що виконує періодичну перевірку наявності анімацій і оновлення часу.
- glControl1 — елемент керування від OpenTK, у якому відбувається малювання гри.

Основним засобом рендерингу гри є бібліотека OpenGL, яка використовується через інтерфейс GLControl. Під час ініціалізації елемента glControl1 виконується налаштування ортографічної проекції для 2D-відображення:

GL.Ortho(0, glControl1.ClientSize.Width, glControl1.ClientSize.Height, 0, -1, 1);

Кожна клітинка сітки розміром 4×4 виводиться як прямокутник із заданим кольором. Плитки, які містять числові значення, малюються поверх

сітки (рис. 4.1.). Графічне представлення плиток здійснюється з використанням текстур, які створюються динамічно за допомогою методу LoadTextureFromText.

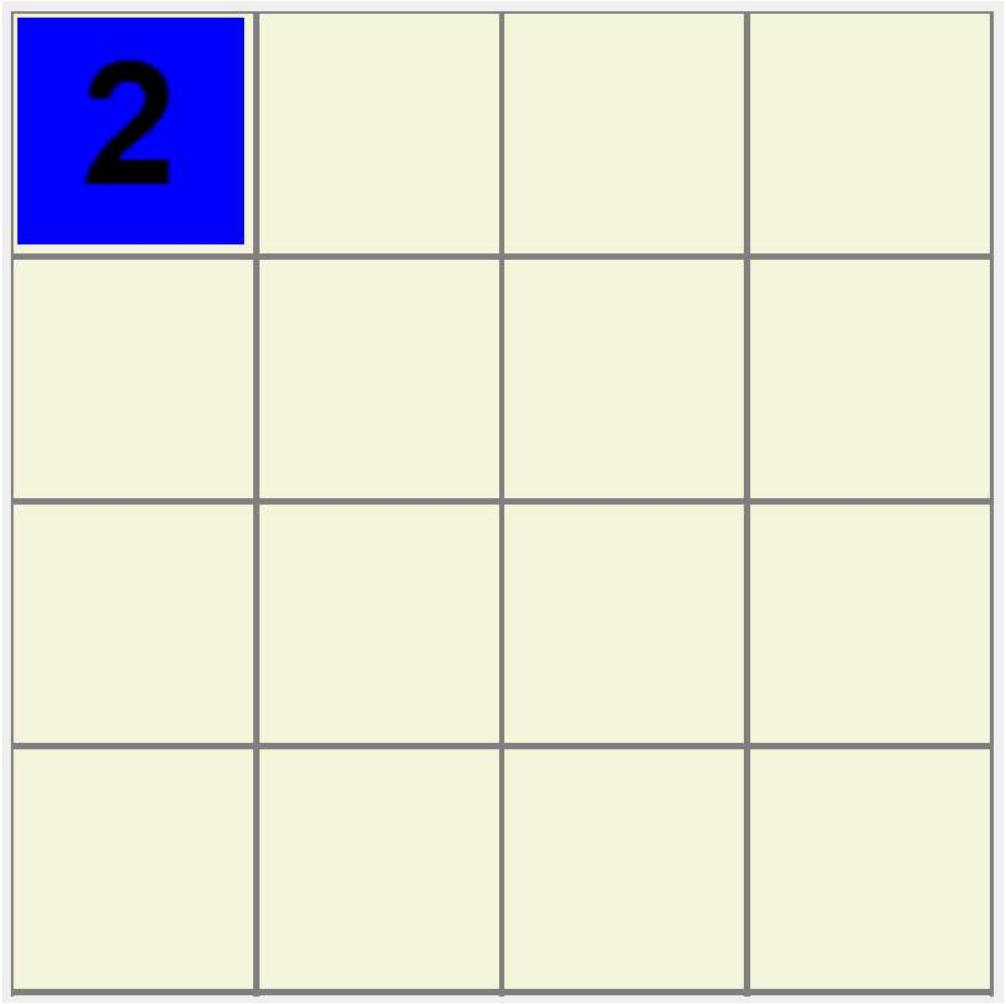

Рис. 4.1. Ігровий контрол

Плитка має колір, який залежить від її числового значення. Наприклад, плитка зі значенням 4 має помаранчевий колір (рис. 4.2.), тоді як плитка 16 має червоний колір (рис. 4.3.).

Рис. 4.2. Плитка четвірки

Рис. 4.3. Плитка шістнадцятки

Присвоєння кольору реалізовано за допомогою конструкції switch, який визначає колір плитки в залежності від її значення. Тип Color4 є частиною бібліотеки OpenTK і представляє кольори у форматі RGBA.

Метод використовує сучасну конструкцію switch expression, яка була введена у C# 8.0. У даному випадку умова — це числове значення, яке містить плитка гри (наприклад, 2, 4, 8 і т.д.).

```
 private Color4 GetTileColor(int value)
{
   return value switch
   {
      2 => Color4.Blue,
      4 => Color4.Coral,
      8 => Color4.Orange,
      16 => Color4.OrangeRed,
      32 => Color4.DarkOrange,
      64 => Color4.DarkRed,
      128 => Color4.Lavender,
      256 => Color4.MediumPurple,
      512 => Color4.Purple,
      1024 => Color4.Green,
      2048 => Color4.DarkGreen,
      _ => Color4.Brown,
   };
```

```
}
```

Кожне числове значення має власний колір.Таким чином можна легко відрізняти плитки між собою. Плитка з числом 2 зафарбовується в синій колір, плитка з числом 8 — у жовтий, а плитка з числом 2048, число при якому досягається перемога, має темно-зелений колір. Значення, які не вказані явно у списку, 4096 або більше, автоматично отримують коричневий колір.

Клас Tile інкапсулює логіку кожної окремої плитки, її позицію, числове значення та анімаційні властивості. Основні атрибути:

1) X, Y — логічні координати плитки.
2) drawX, drawY — координати, які використовуються для плавного переміщення (анімації).
3) Value — числове значення плитки.
4) IsMoving — булевий індикатор, що вказує, чи триває анімація руху плитки.
5) HasMerged — позначає, чи вже була об’єднана плитка в поточному ході.

Анімація переміщення плитки відбувається поступово, шляхом наближення drawX і drawY до визначених параметрів X, Y:

```
drawX += (X - drawX) * 0.2f;
drawY += (Y - drawY) * 0.2f;
```

Користувач керує плитками за допомогою клавіш W, A, S, D. Подія Form1_KeyDown виконує відповідний виклик методу MoveTiles, який в свою чергу перевіряє можливість руху та виконує зміщення:

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
   switch (e.KeyCode)
   {
      case Keys.Up:
      case Keys.W:
```

```
            MoveTiles(0, -1);
            break;
        case Keys.Down:
        case Keys.S:
            MoveTiles(0, 1);
            break;
        ...
    }
    glControl1.Invalidate();
}
```

Програма містить два методи для перевірки стану гри:

1) HasWon() — визначає, чи досягнуто плитки 2048.
2) HasLost() — перевіряє, чи відсутні ходи і порожні клітинки.

У разі перемоги чи поразки користувач бачить відповідне вікно повідомлення (MessageBox), а таймер анімації зупиняється.

Кожна плитка містить числове значення, яке рендериться як текст на текстурі OpenGL. Це реалізується в методі LoadTextureFromText, що створює Bitmap, записує в нього текст і перетворює у текстуру за допомогою функцій OpenGL:

```
g.DrawString(text, font, Brushes.Black, new RectangleF(0, 0, bitmap.Width, bitmap.Height), format);
GL.TexImage2D(...);
```

Оновлення позицій плиток виконується кожні 16 мілісекунд завдяки таймеру animationTimer, що викликає метод AnimationTick. У ньому перевіряється, чи є активні анімації плиток, оновлюється позиція та перерисовується поле.

Основна логіка гри реалізує класичні правила популярної головоломки 2048. Гравець керує плитками на квадратному полі, виконуючи дії за допомогою стрілок клавіатури. При кожному натисканні однієї з чотирьох

клавіш (вліво, вправо, вгору або вниз) всі плитки на полі переміщуються одночасно у відповідному напрямку (рис. 4.4.).

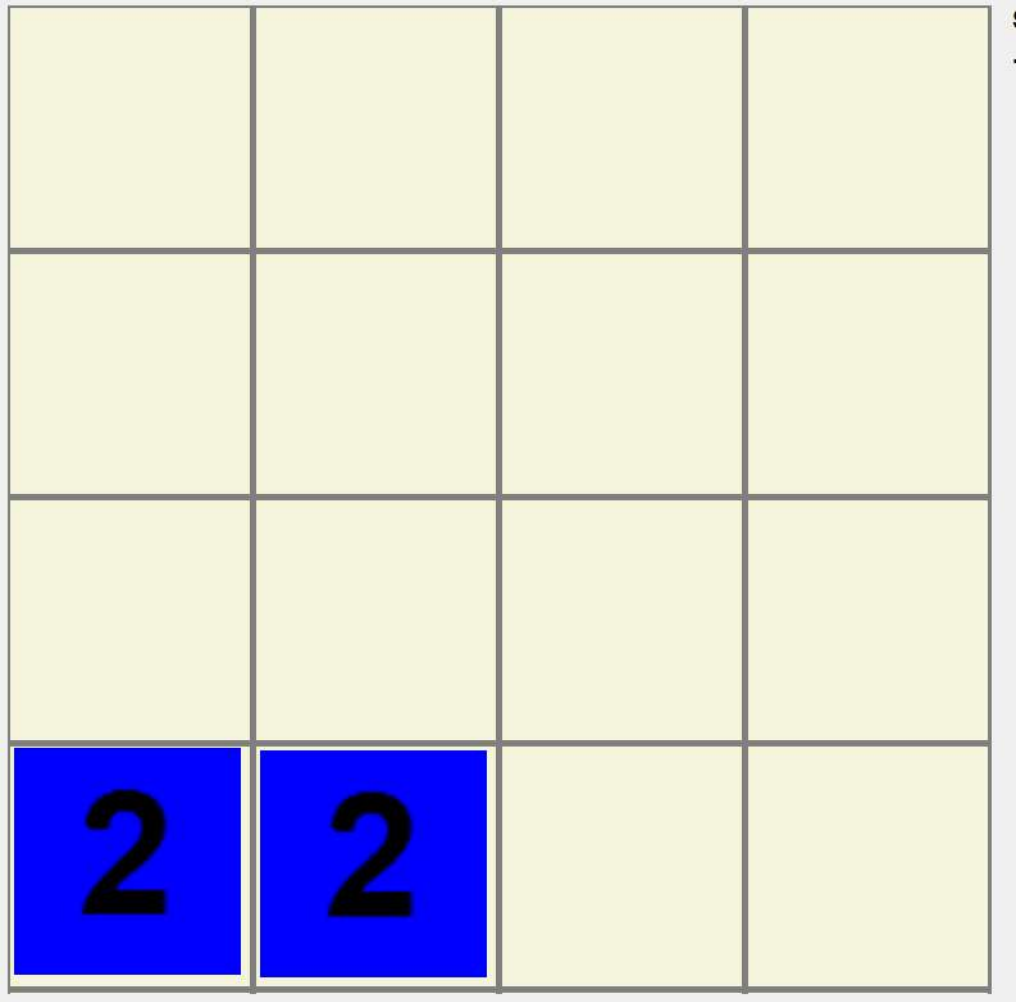

Рис. 4.4. Плитки переміщуються в одну сторону

У разі, якщо на шляху руху дві плитки з однаковими числовими значеннями стикаються, вони об’єднуються в одну. Число на новоутвореній плитці дорівнює сумі значень обох вихідних плиток. Таким чином, дві плитки з числом 2 формують одну з числом 4, дві з числом 4 утворюють 8 і так далі (рис. 4.5.).

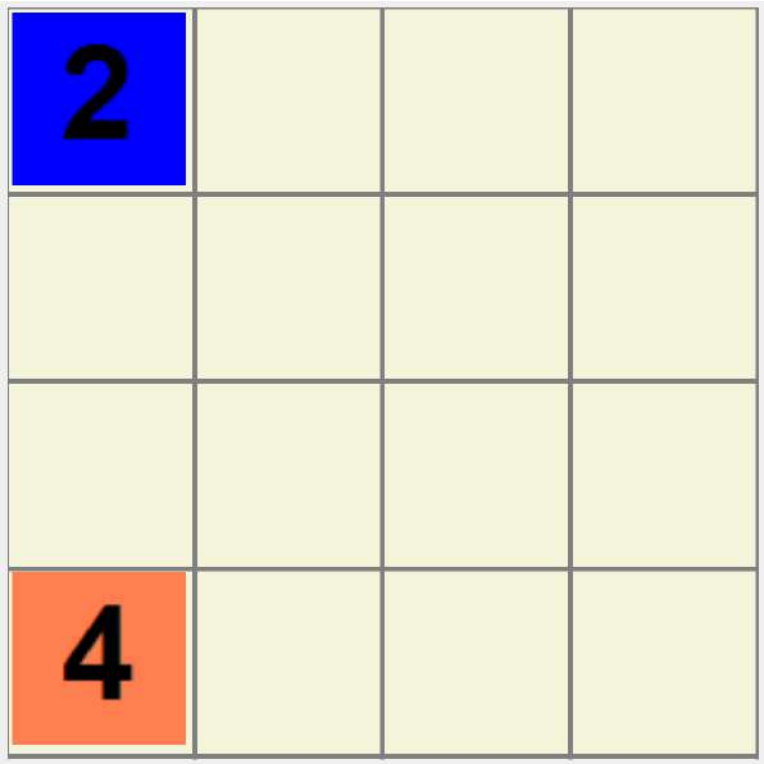

Рис. 4.5. Дві двійки утворили 4-ку

Після кожного ходу на полі випадковим чином з’являється нова плитка. Вона має значення 2 або 4. Найчастіше з’являється саме 2, однак є ймовірність появи 4, що дещо змінює баланс гри та ускладнює досягнення перемоги (рис. 4.6.).

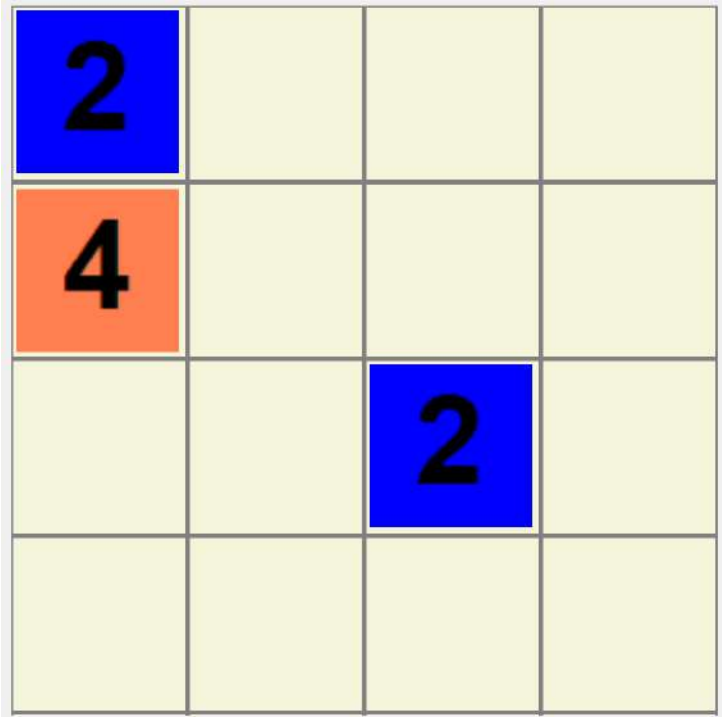

Рис. 4.6. В випадковому вільному місці з’явилася 2-ка

У випадку, коли гравець вичерпав усі можливості для ходу, гра автоматично завершується. Це відбувається, коли поле заповнене (рис. 4.7.), і жодні плитки не можуть бути об’єднані. У цей момент відображається спеціальне вікно повідомлення про поразку (рис. 4.8.).

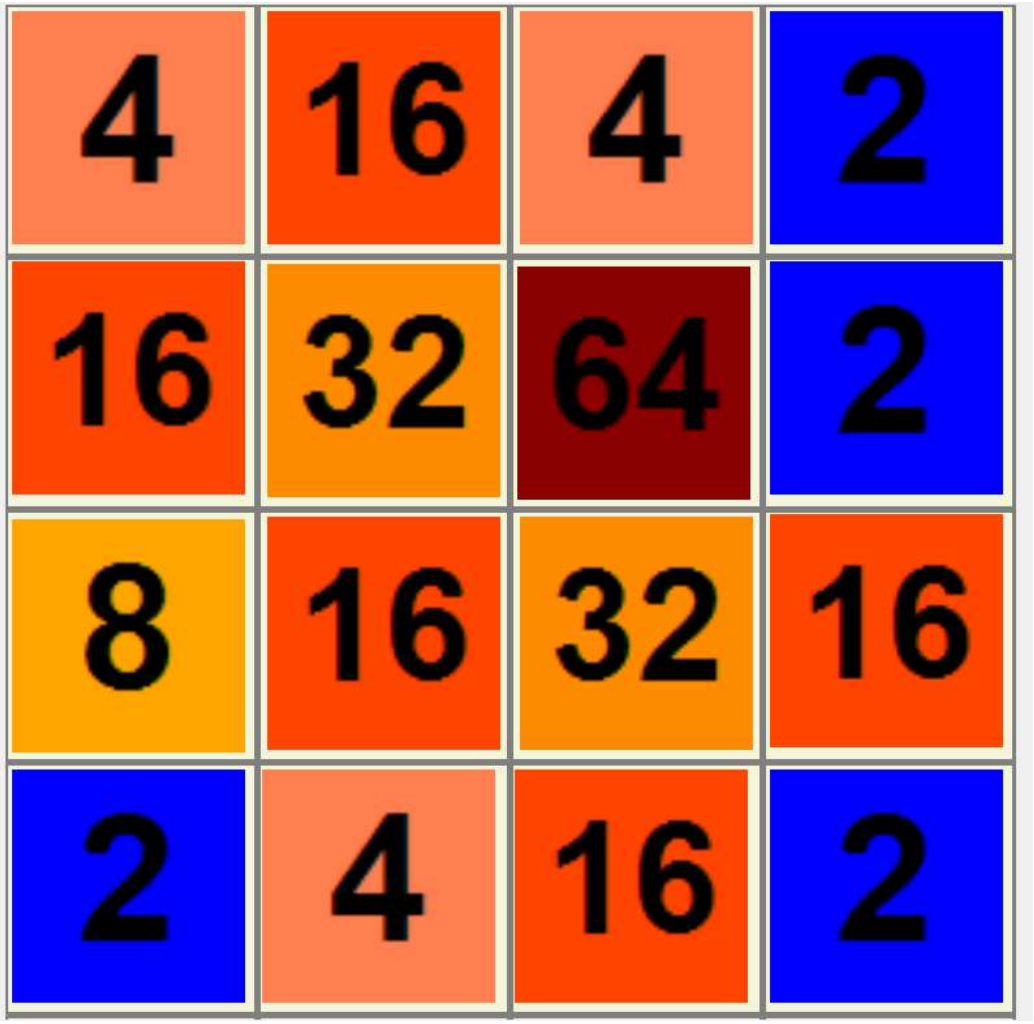

Рис. 4.7. Умови програшу

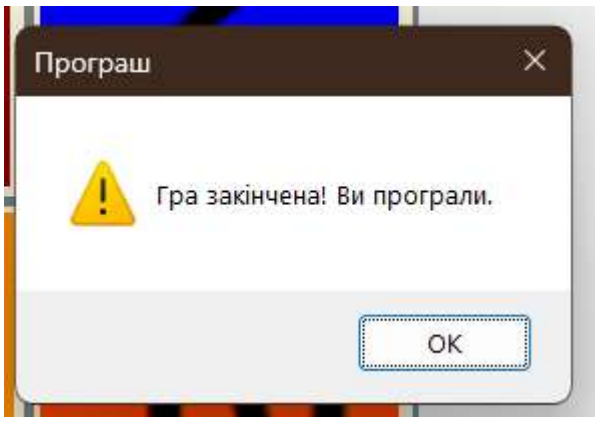

Рис. 4.8. Вікно програшу

Це вікно з’являється поверх основного ігрового інтерфейсу. Воно має форму модального повідомлення — такого, що блокує взаємодію з фоном, поки користувач не підтвердить його закриття. Основним візуальним елементом є текстове повідомлення, яке інформує гравця про завершення гри та факт поразки.

Окрім тексту, у вікні також розміщується кнопка підтвердження, за допомогою якої користувач може закрити повідомлення. Після цього додаток завершує роботу.

Досягнення плитки зі значенням 2048 є головною метою гри. Як тільки гравцеві вдається об’єднати плитки таким чином, щоб на полі з’явилася клітинка з числом 2048, гра фіксує момент перемоги (рис. 4.9.). У відповідь на цю подію система автоматично викликає вікно повідомлення про перемогу (рис. 4.10.).

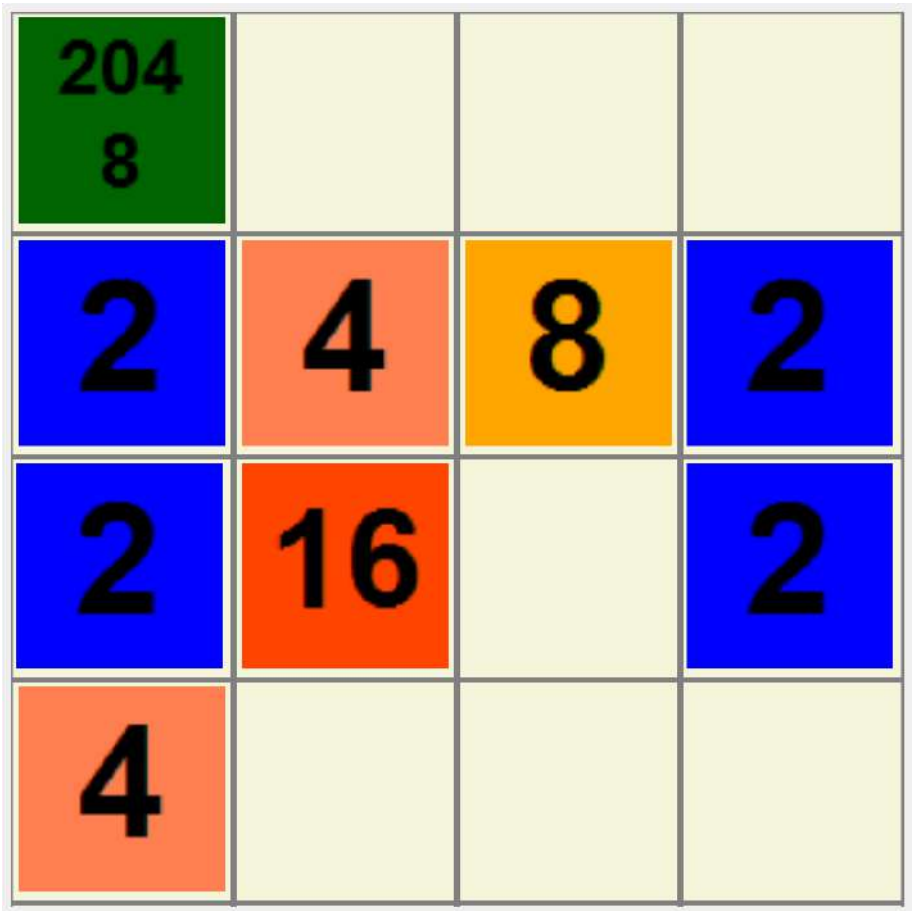

Рис. 4.9. Умови виграшу

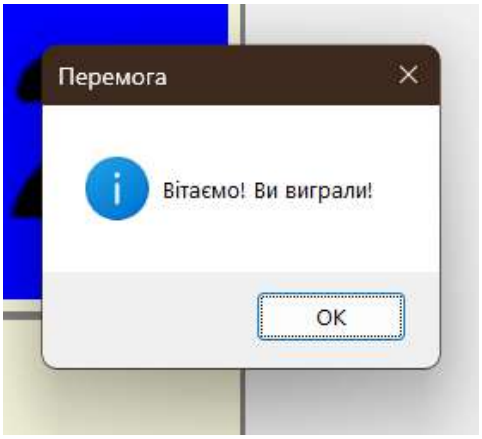

Рис. 4.10. Вікно про перемогу

Це вікно накладається поверх основного інтерфейсу у вигляді модального діалогового повідомлення. На відміну від повідомлення про поразку, вікно перемоги не завершує гру остаточно. Його функція — поінформувати гравця про досягнення мети та надати вибір: завершити гру або продовжити її далі.

У центрі вікна розміщується текстове повідомлення. Воно містить стислу фразу на кшталт «Вітаємо! Ви виграли!».

Нижче за текстом розташовано кнопку “Ок”. Вона дозволяє гравцеві залишити повідомлення та повернутися до ігрового поля. У такому разі всі функції гри залишаються активними, і гравець може намагатися досягти ще вищих значень плиток — 4096, 8192 тощо.

Підрахунок очок у грі реалізовано на основі об’єднання плиток з однаковими значеннями. Кожен раз, коли дві плитки з однаковим числом зливаються в одну, їх значення подвоюється, і це подвоєне значення додається до загального рахунку гравця (Рис. 4.11.). Таким чином, отримання більшої плитки приносить більшу кількість очок.

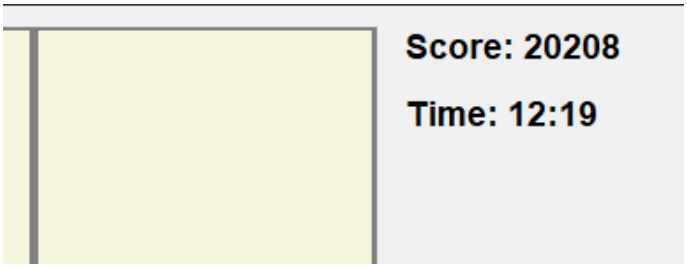

Рис. 4.11. Підрахунок очок

Процес оновлення рахунку відбувається автоматично в момент злиття плиток. Наприклад, при об’єднанні плиток зі значенням 8 і 8 утворюється плитка 16, і до рахунку додається 16 очок.

Значення поточного рахунку зберігається у приватній змінній класу, відповідаючій за логіку гри. Після кожного оновлення очок відбувається відображення нового значення на екрані у вигляді текстової мітки (Label). Вона розташована у правій частині вікна гри і постійно інформує користувача про поточний стан рахунку.

Оновлення тексту мітки здійснюється викликом спеціального методу, який форматує рядок із числом і передає його у властивість Text мітки, з метою підтримувати інформацію про очки у актуальному вигляді протягом всієї ігрової сесії.

Облік часу в грі ведеться за допомогою двох основних компонентів: збереження початкового моменту старту гри та регулярне оновлення показників часу, що минув. При початку гри у змінну зберігається значення поточного часу системи, яке вважається початком ігрової сесії (рис. 4.12.).

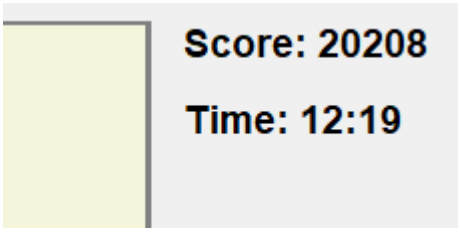

Рис. 4.12. Таймер гри

Для відображення часу, що минув, використовується таймер із коротким інтервалом (приблизно 16 мілісекунд), який запускає оновлення у циклі. Кожного разу при спрацьовуванні таймера обчислюється різниця між поточним системним часом і початковим значенням.

Результатом цього обчислення є тривалість у форматі хвилини та секунди. Це значення у вигляді рядка відображається у спеціальній текстовій мітці, розташованій поруч із рахунком гравця. Форматування передбачає вивід двозначних чисел з провідними нулями.

4.2. Розробка Веб-застосунку

У цьому підрозділі розглянуто архітектуру та особливості реалізації застосунку гри «2048», створеного з використанням мови Java Script і бібліотеки WebGL для графічного виведення (Додаток Б).

Основою візуального відображення є елемент <canvas>, у якому через WebGL рендериться ігрове поле та плитки. Ініціалізація гри починається з отримання контексту WebGL у елементі canvas, після чого задаються параметри сцени: колір фону встановлюється у світло-бежевий, а також включається прозорість для плавного відображення плиток (рис. 4.13.).

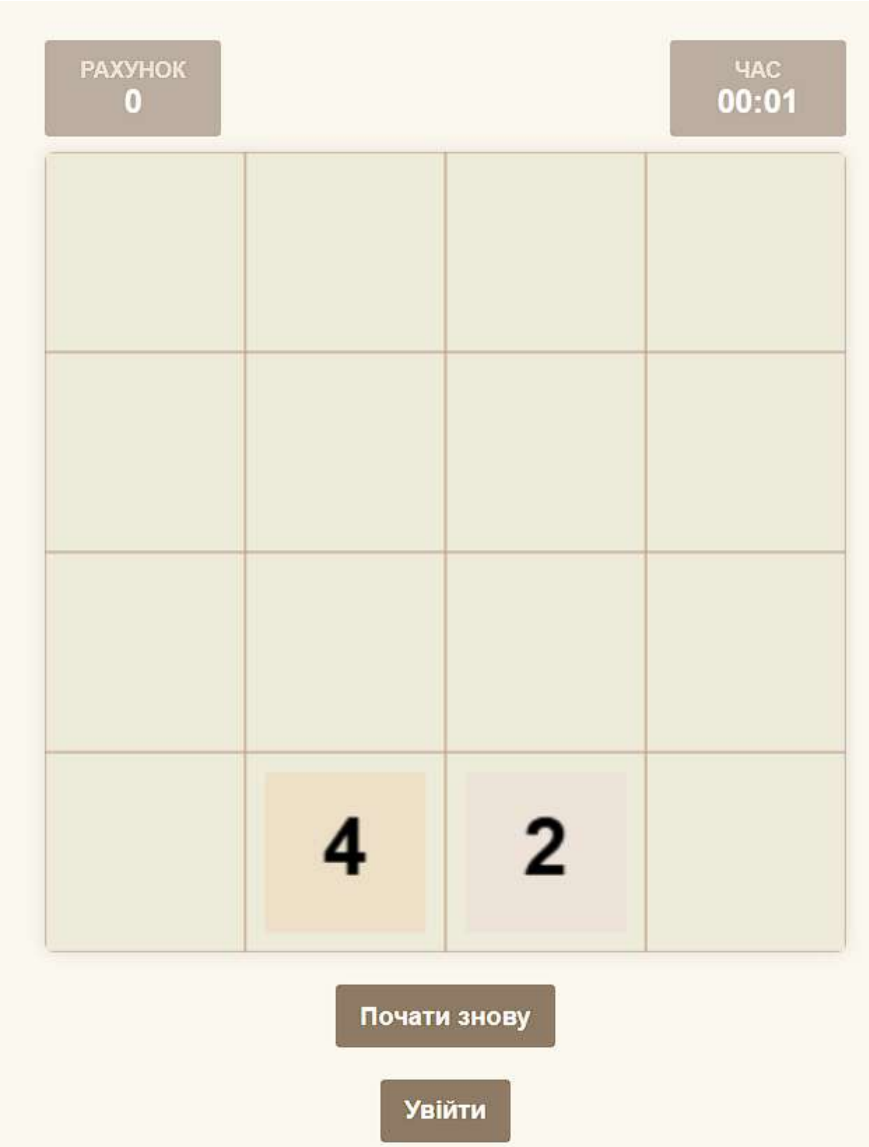

Рис. 4.13. Ігрова сцена

Відповідальність за малювання поля та плиток несуть два шейдерні програми: перша — для сітки ігрового поля, друга — для відображення плиток з текстурами чисел. Кожна плитка — це квадрат із кольором і накладеним текстом числа, що генерується у вигляді WebGL-текстури на основі канваса з намальованим числом (рис. 4.14.).

Рис. 4.14. Плитка з числом

На початку роботи коду визначаються основні параметри гри, такі як розмір сітки (GRID_SIZE), розмір кожної клітинки у відносних координатах (CELL_SIZE), відступи між клітинками (CELL_MARGIN) та швидкість анімації (ANIMATION_SPEED):

```
const GRID_SIZE = 4;
const CELL_SIZE = 1 / GRID_SIZE;
const CELL_MARGIN = 0.05;
const ANIMATION_SPEED = 0.2;
```

Код для створення і компіляції шейдерних програм виділено у функцію createShaderProgram. Визначено два набори шейдерів: для сітки і для плиток.

```
const tileVertexShaderSource = `
    attribute vec2 position;
    attribute vec2 texcoord;
    varying vec2 vTexcoord;
    uniform mat4 matrix;
    void main() {
        vTexcoord = texcoord;
        gl_Position = matrix * vec4(position, 0.0, 1.0);
    }
```

```
`;
const tileFragmentShaderSource = `
    precision mediump float;
    varying vec2 vTexcoord;
    uniform vec4 color;
    uniform sampler2D sampler;
    uniform bool hasTexture;
    void main() {
        if (hasTexture) {
            vec4 texColor = texture2D(sampler, vTexcoord);
            gl_FragColor = texColor * color;
        } else {
            gl_FragColor = color;
        }
    }
`;
```

Для управління грою використовується обробка подій клавіатури та дотиків (для мобільних пристроїв). Клавіші зі стрілками та кнопки WASD відповідають за переміщення плиток у відповідних напрямках. Аналогічно реалізовано обробку свайпів: порівняння координат початку і кінця дотику визначає напрямок руху плиток.

```
function handleKeyDown(event) {
    if (keys.hasOwnProperty(event.key) && !gameOver) {
        keys[event.key] = true;
        if (event.key === 'ArrowUp' || event.key === 'w') {
            moveTiles(0, -1);
        } else if (event.key === 'ArrowDown' || event.key === 's') {
            moveTiles(0, 1);
        } else if (event.key === 'ArrowLeft' || event.key === 'a') {
            moveTiles(-1, 0);
```

```
        } else if (event.key === 'ArrowRight' || event.key === 'd') {
            moveTiles(1, 0);
        }
        event.preventDefault();
    }
}
function handleKeyUp(event) {
    if (keys.hasOwnProperty(event.key)) {
        keys[event.key] = false;
    }
}
```

Рендеринг кожного кадру відбувається через requestAnimationFrame. Завдяки методу анімації плиток виходять плавні. Під час рендеру виконується малювання сітки, плиток та оновлення інформації про рахунок.

Важливою частиною є функція завантаження текстур чисел — вона створює для кожного можливого значення плитки текстуру на основі канваса, на якій намальовано число.

```
function createTextureFromText(text) {
    return new Promise(resolve => {
        const canvas = document.createElement('canvas');
        canvas.width = 64;
        canvas.height = 64;
        const ctx = canvas.getContext('2d');
        ctx.fillStyle = 'rgba(255, 255, 255, 1)';
        ctx.fillRect(0, 0, canvas.width, canvas.height);
        let fontSize;
        if (text.length === 1) fontSize = 32;
        else if (text.length === 2) fontSize = 28;
        else if (text.length === 3) fontSize = 20;
        else fontSize = 16;
```

```
            ctx.fillStyle = 'black';
            ctx.font = `bold ${fontSize}px Arial`;
            ctx.textAlign = 'center';
            ctx.textBaseline = 'middle';
            ctx.fillText(text, canvas.width / 2, canvas.height / 2);
            const texture = gl.createTexture();
            gl.bindTexture(gl.TEXTURE_2D, texture);
            gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, canvas);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.LINEAR);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
gl.LINEAR);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);
            resolve(texture);
        });
    }
```

Ігровий контейнер займає центральну частину сторінки (рис. 4.15.). У верхній частині розміщена інформаційна панель з двома блоками: один показує поточний рахунок, другий — час гри. Ці дані оновлюються під час ігрового процесу. Основна область — це холст, на якому відображається ігрове поле.

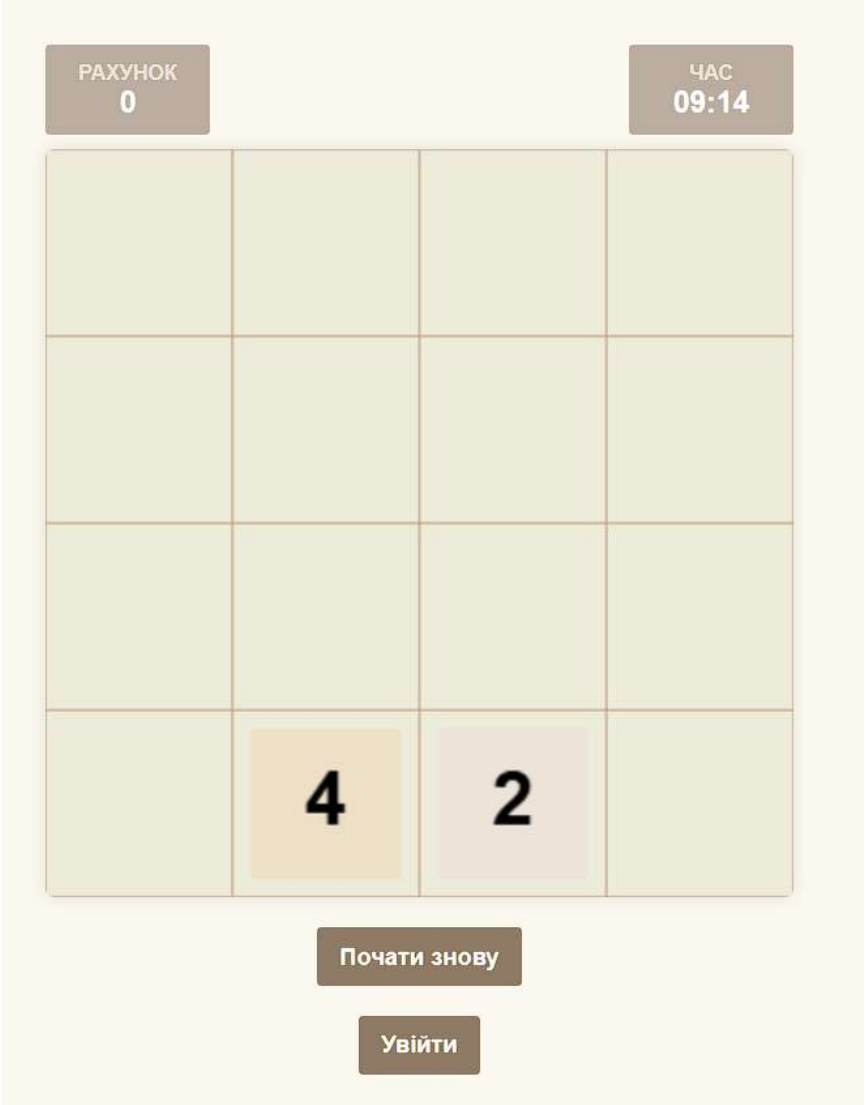

Рис. 4.15. Сторінка гри

При завершенні гри поверх холста з’являється оверлей з повідомленням про кінець гри та кнопкою для початку нового раунду (рис. 4.16.). Ця панель блокує взаємодію з ігровим полем до перезапуску.

Внизу сторінки розташовані кнопки управління: одна для перезапуску гри, інша — для входу в акаунт. Вони забезпечують швидкий доступ до основних дій.

Передбачений адаптивний дизайн, щоб користувачі з розміром різних пристроїв могли грати в гру. Основний ігровий контейнер автоматично підлаштовує свої розміри відповідно до ширини вікна браузера. Це досягається за допомогою JavaScript-функції, яка змінює висоту канваса

пропорційно до ширини контейнера, підтримуючи квадратну форму ігрового поля.

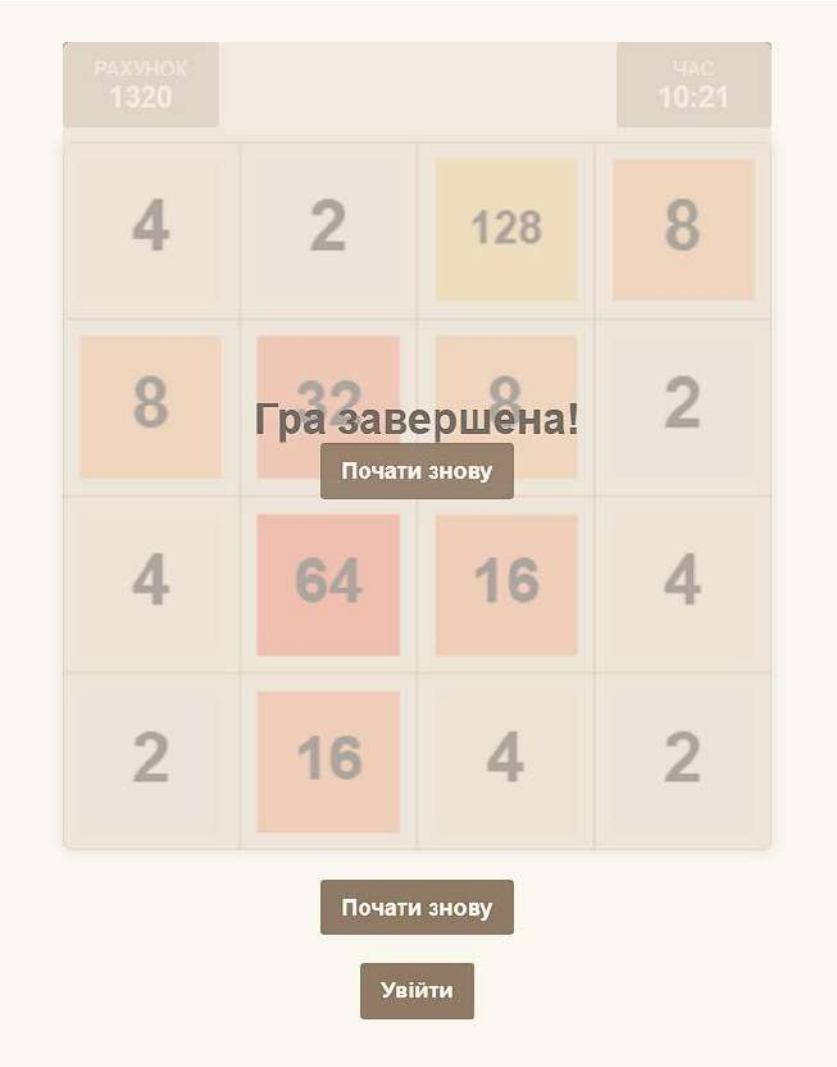

Рис. 4.16. Кінець гри

Сторінка аутентифікації розташована по центру вікна браузера завдяки flex-контейнеру, який вирівнює контент по вертикалі та горизонталі. Основний блок з формами має білий фон, заокруглені кути та легку тінь, що додає об’єму та привабливості.

Верхня частина містить заголовок “2048 - Авторизація”, який виділений більшим шрифтом та сірим кольором.

Інтерфейс включає дві форми: для входу (рис. 4.17.) та для реєстрації (рис. 4.18.). Спочатку відображається форма входу, яка містить поля для введення логіна та пароля, а також дві кнопки — для підтвердження входу і

переходу до реєстрації. Поля мають однаковий стиль з рамками, внутрішнім відступом і зручним розміром шрифту.

Рис. 4.17. Форма авторизації

Форма реєстрації прихована за замовчуванням і відкривається за натисканням відповідної кнопки. Вона містить додаткове поле для підтвердження пароля і кнопки для реєстрації та повернення до входу. Усі кнопки мають округлі краї, фон у теплих відтінках коричневого, що узгоджується з кольорами гри, і змінюють прозорість при наведенні курсора.

В обох формах реалізовано базову валідацію полів за допомогою JavaScript. Це означає що коли користувач вводить некоректні дані, відображається повідомлення про помилку (рис. 4.19.) Коли користувач ввів дані коректно відображається повідомлення про успіх (рис. 4.20.) Повідомлення про помилки виділені помаранчевим кольором для привернення уваги, а повідомлення про успішну дію — зеленим.

Рис. 4.18. Форма реєстрації

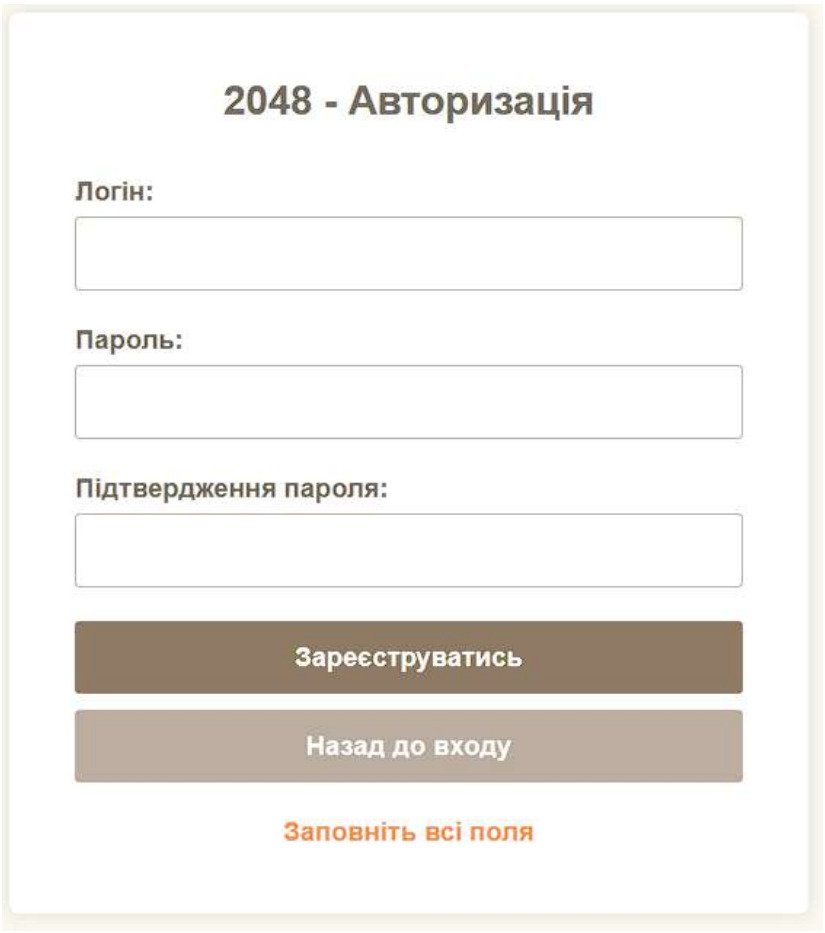

Рис. 4.19. Повідомлення про помилку

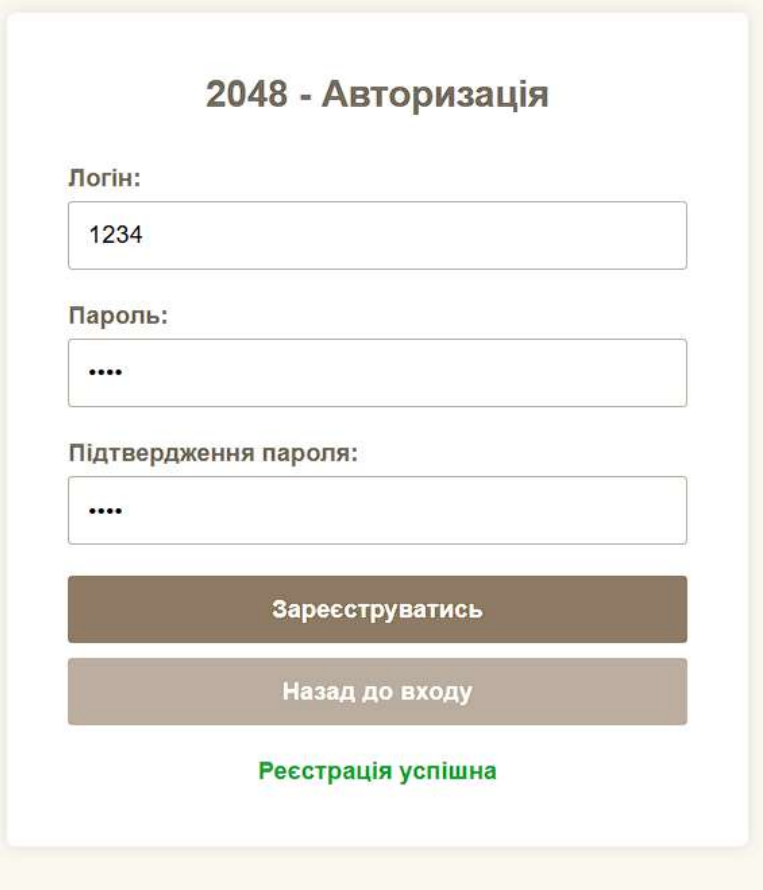

Рис. 4.20. Повідомлення про успішну дію

Для форм передбачена базова валідація введених даних: перевірка заповненості полів, збіг паролів та мінімальна довжина пароля.

Висновки до розділу 4

У розділі було розглянуто процес розробки веб-застосунку гри 2048. Було описано основний інтерфейс гри, включно з відображенням рахунку, таймера і самої ігрової сітки, яка рендериться за допомогою OpenGL.

ВИСНОВКИ

У межах виконання дипломної роботи було послідовно пройдено низку етапів, які охоплюють як теоретичний аналіз, так і практичну реалізацію програмного забезпечення для гри 2048. Кожен етап виконував окрему функцію у формуванні цілісної, багатоаспектної розробки.

Спочатку було поставлено задачу. Розглянуто алгоритмічну модель гри 2048 — описано механіку гри як дискретну систему, де кожен крок змінює конфігурацію поля за чітко визначеними правилами. Далі проведено аналіз існуючих реалізацій гри, що дало змогу оцінити різноманітні підходи до побудови логіки та графіки. Було також сформовано перелік вимог до власного проєкту — як функціональних, так і нефункціональних. Наприкінці розділу розглянуто вибрані технології розробки — C#, OpenGL, WebGL, MySQL — і пояснено доцільність їх застосування.

Потім було здійснено розробку алгоритмів, які реалізують основну логіку гри. Було визначено, як саме обробляються ходи, об’єднання клітинок, генерація нових чисел і перевірка завершення гри. Алгоритми були адаптовані до обмежень і переваг кожної з платформ, що планувалися до реалізації.

Потім було приділено час на організацію інформаційного забезпечення. Було розроблено структуру даних для збереження стану гри та результатів. Також описано логіку ігрового циклу, включно з переходами між станами гри, обробкою дій користувача та оновленням інтерфейсу. Потім будувалася база даних. Описано її схему, таблиці та логіку взаємодії з нею з боку програми.

Четвертий етап — найоб’ємніший, присвячений безпосередній розробці програмного забезпечення. Розроблено десктопну версію гри на C# із використанням бібліотек OpenGL і GlControl. Зокрема, створено графічний інтерфейс, реалізовано логіку гри, а також забезпечено інтеграцію з базою даних. Паралельно розроблено веб-версію, засновану на HTML5, JavaScript і WebGL.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. G. G. G. Guei. Temporal Difference Learning for 2048 with Optimistic Initialization. arXiv preprint arXiv:1506.08843, 2015.
2. Szubert, M., Jaśkowski, W. Temporal difference learning of n-tuple networks for the game 2048. IEEE Transactions on Computational Intelligence and AI in Games, 2014.
3. Wu, Y., & Tian, Y. Training agent for 2048 using reinforcement learning. arXiv preprint arXiv:1506.07361, 2015.
4. Kocsis, L., & Szepesvári, C. Monte-Carlo Tree Search for 2048. In Proceedings of ECML, 2006.
5. Russell, S., & Norvig, P. Artificial Intelligence: A Modern Approach. Pearson, 4th Edition, 2020.
6. Heaton, J. Introduction to Neural Networks for Java, C# and Python. Heaton Research, 2011.
7. Troelsen, A., & Japikse, P. Pro C# 9 with .NET 5. Apress, 2021.
8. Flanagan, D. JavaScript: The Definitive Guide. O’Reilly Media, 2020.
9. Freeman, E., & Robson, E. Head First HTML and CSS. O'Reilly Media, 2012.
10. Angel, E., & Shreiner, D. Interactive Computer Graphics with WebGL. Addison-Wesley, 2019.
11. Ullman, J. D., & Widom, J. A First Course in Database Systems. Pearson, 3rd Edition, 2008.
12. Welling, L., & Thomson, L. PHP and MySQL Web Development. Addison-Wesley, 5th Edition, 2016.
13. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
14. Ray, R. 2048 AI Challenge: Game mechanics and algorithm strategies. [Online] https://github.com/nneonneo/2048-ai
15. GitHub. 2048 Web Implementation by Gabriele Cirulli. [Online] https://github.com/gabrielecirulli/2048

ДОДАТКИ

Додаток А

C#-застосунок гри

```
using OpenTK.Graphics.OpenGL;
using OpenTK.Mathematics;
using System.Drawing.Imaging;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;

namespace _2048v1
{
    public partial class Form1 : Form
    {
        private System.Windows.Forms.Timer animationTimer;
        private const int GridSize = 4;
        private const int AnimationSpeed = 10;
        private Tile[,] tiles = new Tile[GridSize, GridSize];
        private Dictionary<int, int> textureIds = new Dictionary<int, int>();
        private int score = 0;
        private Label timeLabel;

        public Form1()
        {
            InitializeComponent();
            this.KeyPreview = true;
            this.KeyDown += Form1_KeyDown;
            this.Activated += (s, e) => this.Focus();

            glControl1.Load += glControl1_Load;
            glControl1.Paint += glControl1_Paint;

            animationTimer = new System.Windows.Forms.Timer();
```

```
    animationTimer.Interval = 16;
    animationTimer.Tick += AnimationTick;
    animationTimer.Start();

    gameStartTime = DateTime.Now;

    tiles[0, 0] = new Tile(0, 0, 2);
}


private void UpdateScore(int points)
{
    score += points;
    scoreLabel.Text = $"Score: {score}";
}

private void UpdateTime()
{
    TimeSpan elapsed = DateTime.Now - gameStartTime;
    timeLabel.Text = $"Time: {elapsed.Minutes:D2}:{elapsed.Seconds:D2}";
}

private void glControl1_Load(object sender, EventArgs e)
{
    GL.Enable(EnableCap.DebugOutput);
    GL.Enable(EnableCap.Texture2D);
    GL.MatrixMode(MatrixMode.Projection);
    GL.LoadIdentity();
    GL.Ortho(0, glControl1.ClientSize.Width, glControl1.ClientSize.Height, 0, -1, 1);
    GL.MatrixMode(MatrixMode.Modelview);
    GL.LoadIdentity();
    LoadNumberTextures();
```

```
    glControl1.Invalidate();
}

private void LoadNumberTextures()
{
    for (int i = 1; i <= 2048; i *= 2)
    {
        int textureId = LoadTextureFromText(i.ToString());
        textureIds[i] = textureId;
    }
}

public int LoadTextureFromText(string text)
{
    Bitmap bitmap = new Bitmap(64, 64);
    using (Graphics g = Graphics.FromImage(bitmap))
    {
        g.Clear(Color.White);

        int fontSize = text.Length switch
        {
            1 => 32,
            2 => 28,
            3 => 20,
            _ => 16
        };

        using (Font font = new Font("Arial", fontSize, FontStyle.Bold))
        using (StringFormat format = new StringFormat { Alignment =
StringAlignment.Center, LineAlignment = StringAlignment.Center })
        {
            g.DrawString(text, font, Brushes.Black, new RectangleF(0, 0, bitmap.Width,
bitmap.Height), format);
```

```
            }
        }

        int textureId = GL.GenTexture();
        GL.BindTexture(TextureTarget.Texture2D, textureId);

        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter,
(int)TextureMinFilter.Linear);
        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter,
(int)TextureMagFilter.Linear);

        BitmapData data = bitmap.LockBits(new Rectangle(0, 0, bitmap.Width, bitmap.Height),
            ImageLockMode.ReadOnly,
System.Drawing.Imaging.PixelFormat.Format32bppArgb);

        GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, bitmap.Width,
bitmap.Height,
            0, OpenTK.Graphics.OpenGL.PixelFormat.Bgra, PixelType.UnsignedByte,
data.Scan0);

        bitmap.UnlockBits(data);
        bitmap.Dispose();

        return textureId;
    }

    private void glControl1_Paint(object sender, PaintEventArgs e)
    {
        glControl1.MakeCurrent();
        GL.Viewport(0, 0, glControl1.ClientSize.Width, glControl1.ClientSize.Height);
        GL.ClearColor(Color4.Beige);
        GL.Clear(ClearBufferMask.ColorBufferBit);

        DrawGrid();
```

```
    DrawTiles();

    glControl1.SwapBuffers();
}

private void DrawGrid()
{
    float cellSize = glControl1.ClientSize.Width / (float)GridSize;
    GL.Color4(Color4.Gray);
    GL.LineWidth(5);

    GL.Begin(PrimitiveType.Lines);
    for (int i = 0; i <= GridSize; i++)
    {
        float pos = i * cellSize;
        GL.Vertex2(pos, 0);
        GL.Vertex2(pos, glControl1.ClientSize.Height);
        GL.Vertex2(0, pos);
        GL.Vertex2(glControl1.ClientSize.Width, pos);
    }
    GL.End();
}

private void DrawTiles()
{
    float cellSize = glControl1.ClientSize.Width / (float)GridSize;

    foreach (var tile in tiles)
    {
        if (tile != null)
        {
            DrawTile(tile, cellSize);
```

```
        }
    }
}

private void DrawTile(Tile tile, float cellSize)
{
    float x = tile.GetDrawX(cellSize);
    float y = tile.GetDrawY(cellSize);
    float margin = 5;
    float tileSize = cellSize - 2 * margin;

    Color4 tileColor = GetTileColor(tile.Value);

    GL.Disable(EnableCap.Texture2D);

    GL.Color4(tileColor);
    GL.Begin(PrimitiveType.Quads);
    GL.Vertex2(x + margin, y + margin);
    GL.Vertex2(x + tileSize, y + margin);
    GL.Vertex2(x + tileSize, y + tileSize);
    GL.Vertex2(x + margin, y + tileSize);
    GL.End();

    if (textureIds.ContainsKey(tile.Value))
    {
        int textureId = textureIds[tile.Value];
        GL.Enable(EnableCap.Texture2D);
        GL.BindTexture(TextureTarget.Texture2D, textureId);

        GL.Begin(PrimitiveType.Quads);
        GL.TexCoord2(0, 0); GL.Vertex2(x + margin, y + margin);
        GL.TexCoord2(1, 0); GL.Vertex2(x + tileSize, y + margin);
```

```
            GL.TexCoord2(1, 1); GL.Vertex2(x + tileSize, y + tileSize);
            GL.TexCoord2(0, 1); GL.Vertex2(x + margin, y + tileSize);
            GL.End();

            GL.Disable(EnableCap.Texture2D);
        }
    }

    private Color4 GetTileColor(int value)
    {
        return value switch
        {
            2 => Color4.Blue,
            4 => Color4.Coral,
            8 => Color4.Orange,
            16 => Color4.OrangeRed,
            32 => Color4.DarkOrange,
            64 => Color4.DarkRed,
            128 => Color4.Lavender,
            256 => Color4.MediumPurple,
            512 => Color4.Purple,
            1024 => Color4.Green,
            2048 => Color4.DarkGreen,
            _ => Color4.Brown,
        };
    }

    private void AnimationTick(object sender, EventArgs e)
    {
        bool needRedraw = false;

        foreach (var tile in tiles)
```

```
        {
            if (tile != null && tile.IsMoving)
            {
                tile.UpdatePosition();
                needRedraw = true;
            }
        }

        UpdateTime();

        if (needRedraw)
            glControl1.Invalidate();
    }

    private void AddRandomTile()
    {
        List<Point> emptyTiles = new List<Point>();
        for (int x = 0; x < GridSize; x++)
        {
            for (int y = 0; y < GridSize; y++)
            {
                if (tiles[x, y] == null)
                    emptyTiles.Add(new Point(x, y));
            }
        }
        if (emptyTiles.Count > 0)
        {
            Random rand = new Random();
            Point newTilePos = emptyTiles[rand.Next(emptyTiles.Count)];
            tiles[newTilePos.X, newTilePos.Y] = new Tile(newTilePos.X, newTilePos.Y,
rand.Next(10) < 9 ? 2 : 4);
        }
```

```
}

private bool TryMoveTile(int x, int y, int dx, int dy)
{
    if (tiles[x, y] == null)
        return false;

    int newX = x + dx;
    int newY = y + dy;

    while (IsInsideGrid(newX, newY) && tiles[newX, newY] == null)
    {
        tiles[newX, newY] = tiles[x, y];
        tiles[x, y] = null;
        tiles[newX, newY].MoveTo(newX, newY);

        x = newX;
        y = newY;
        newX += dx;
        newY += dy;
    }

    if (IsInsideGrid(newX, newY) && tiles[newX, newY] != null &&
        tiles[newX, newY].Value == tiles[x, y].Value &&
        !tiles[newX, newY].HasMerged && !tiles[x, y].HasMerged)
    {
        int mergedValue = tiles[x, y].Value * 2;
        tiles[newX, newY].DoubleValue();
        tiles[newX, newY].HasMerged = true;
        tiles[x, y] = null;
        UpdateScore(mergedValue);
        return true;
```

```
    }

    return false;
}

private void ResetMergeFlags()
{
    foreach (var tile in tiles)
    {
        if (tile != null)
            tile.HasMerged = false;
    }
}

private bool IsInsideGrid(int x, int y)
{
    return x >= 0 && x < GridSize && y >= 0 && y < GridSize;
}

private void MoveTiles(int dx, int dy)
{
    bool moved = true;
    ResetMergeFlags();

    if (dx == 1)
    {
        for (int y = 0; y < GridSize; y++)
        {
            for (int x = GridSize - 2; x >= 0; x--)
            {
                moved |= TryMoveTile(x, y, dx, dy);
            }
```

```
    }
}
else if (dx == -1)
{
    for (int y = 0; y < GridSize; y++)
    {
        for (int x = 1; x < GridSize; x++)
        {
            moved |= TryMoveTile(x, y, dx, dy);
        }
    }
}
else if (dy == 1)
{
    for (int x = 0; x < GridSize; x++)
    {
        for (int y = GridSize - 2; y >= 0; y--)
        {
            moved |= TryMoveTile(x, y, dx, dy);
        }
    }
}
else if (dy == -1)
{
    for (int x = 0; x < GridSize; x++)
    {
        for (int y = 1; y < GridSize; y++)
        {
            moved |= TryMoveTile(x, y, dx, dy);
        }
    }
}
```

```
    if (moved)
    {
        AddRandomTile();
        CheckGameOver();
    }
}

private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    switch (e.KeyCode)
    {
        case Keys.Up:
        case Keys.W:
            MoveTiles(0, -1);
            break;
        case Keys.Down:
        case Keys.S:
            MoveTiles(0, 1);
            break;
        case Keys.Left:
        case Keys.A:
            MoveTiles(-1, 0);
            break;
        case Keys.Right:
        case Keys.D:
            MoveTiles(1, 0);
            break;
    }

    glControl1.Invalidate();
}
```

```
    private void CheckGameOver()
    {
        if (HasWon())
        {
            MessageBox.Show("Вітаємо! Ви виграли!", "Перемога", MessageBoxButtons.OK,
MessageBoxIcon.Information);
            animationTimer.Stop();
        }
        else if (HasLost())
        {
            MessageBox.Show("Гра закінчена! Ви програли.", "Програш",
MessageBoxButtons.OK, MessageBoxIcon.Warning);
            animationTimer.Stop();
        }
    }

    private bool HasWon()
    {
        foreach (var tile in tiles)
        {
            if (tile != null && tile.Value == 2048)
            {
                return true;
            }
        }
        return false;
    }

    private bool HasLost()
    {
        for (int x = 0; x < GridSize; x++)
        {
```

```
        for (int y = 0; y < GridSize; y++)
        {
            if (tiles[x, y] == null)
                return false;
        }
    }

    for (int x = 0; x < GridSize; x++)
    {
        for (int y = 0; y < GridSize; y++)
        {
            if (x < GridSize - 1 && tiles[x, y].Value == tiles[x + 1, y].Value)
                return false;
            if (y < GridSize - 1 && tiles[x, y].Value == tiles[x, y + 1].Value)
                return false;
        }
    }

    return true;
}

public class Tile
{
    public int X { get; private set; }
    public int Y { get; private set; }
    private float drawX, drawY;
    public int Value { get; private set; }
    public bool IsMoving => Math.Abs(drawX - X) > 0.01f || Math.Abs(drawY - Y) > 0.01f;
    public bool HasMerged { get; set; }

    public Tile(int x, int y, int value)
    {
```

```
            X = x;
            Y = y;
            drawX = x;
            drawY = y;
            Value = value;
            HasMerged = false;
        }

        public void DoubleValue()
        {
            Value *= 2;
        }

        public void MoveTo(int newX, int newY)
        {
            X = newX;
            Y = newY;
        }

        public void UpdatePosition()
        {
            drawX += (X - drawX) * 0.2f;
            drawY += (Y - drawY) * 0.2f;
        }

        public float GetDrawX(float cellSize) => drawX * cellSize;
        public float GetDrawY(float cellSize) => drawY * cellSize;
    }

  }
}
```

Додаток Б

Веб-застосунок

```
const GRID_SIZE = 4;
  const CELL_SIZE = 1 / GRID_SIZE;
  const CELL_MARGIN = 0.05;
  const ANIMATION_SPEED = 0.2;

  let canvas, gl;
  let gridProgram, tileProgram;
  let tiles = [];
  let score = 0;
  let startTime = null;
  let gameTime = 0;
  let gameRunning = false;
  let gameOver = false;
  let victory = false;
  let textures = {};
  let matrixLocation;
  let colorLocation;
  let tileMatrixLocation;
  let tileColorLocation;
  let textureLocation;
  let hasTextureLocation;

  const keys = {
      ArrowUp: false,
      ArrowDown: false,
      ArrowLeft: false,
      ArrowRight: false,
      w: false,
      a: false,
      s: false,
      d: false
```

```
};

let touchStartX = 0;
let touchStartY = 0;

window.onload = function() {
    initializeGame();
    setupEventListeners();
    loadTextures().then(() => {
        startGame();
        requestAnimationFrame(render);
    });
};



function restoreGameState(gameState) {
    tiles = [];
    score = gameState.score;

    gameState.tiles.forEach(tileData => {
        const tile = new Tile(tileData.x, tileData.y, tileData.value);
        setTileAt(tileData.x, tileData.y, tile);
    });

    document.getElementById('score').textContent = score;
    startTime = Date.now() - (gameState.gameTime * 1000);
}

function initializeGame() {
    canvas = document.getElementById('game-canvas');
    canvas.width = 500;
    canvas.height = 500;
```

```
    gl = canvas.getContext('webgl');
    if (!gl) {
        alert('WebGL не підтримується вашим браузером.');
        return;
    }

    gl.clearColor(0.93, 0.92, 0.85, 1.0);
    gl.enable(gl.BLEND);
    gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);

    initShaders();
}

function setupEventListeners() {
    window.addEventListener('keydown', handleKeyDown);
    window.addEventListener('keyup', handleKeyUp);

    canvas.addEventListener('touchstart', handleTouchStart);
    canvas.addEventListener('touchend', handleTouchEnd);

    document.getElementById('restart-btn').addEventListener('click', restartGame);
    document.getElementById('restart-btn-overlay').addEventListener('click', restartGame);
    document.getElementById('auth-btn').addEventListener('click', pageAuthOpen);

    window.addEventListener('resize', resizeCanvas);
    resizeCanvas();
}

function resizeCanvas() {
    const container = document.querySelector('.game-container');
    const width = container.clientWidth;
```

```
    canvas.style.height = width + 'px';
}

function handleKeyDown(event) {
    if (keys.hasOwnProperty(event.key) && !gameOver) {
        keys[event.key] = true;

        if (event.key === 'ArrowUp' || event.key === 'w') {
            moveTiles(0, -1);
        } else if (event.key === 'ArrowDown' || event.key === 's') {
            moveTiles(0, 1);
        } else if (event.key === 'ArrowLeft' || event.key === 'a') {
            moveTiles(-1, 0);
        } else if (event.key === 'ArrowRight' || event.key === 'd') {
            moveTiles(1, 0);
        }

        event.preventDefault();
    }
}

function handleKeyUp(event) {
    if (keys.hasOwnProperty(event.key)) {
        keys[event.key] = false;
    }
}

function handleTouchStart(event) {
    touchStartX = event.touches[0].clientX;
    touchStartY = event.touches[0].clientY;
    event.preventDefault();
}
```

```
function handleTouchEnd(event) {
    if (gameOver) return;

    const touchEndX = event.changedTouches[0].clientX;
    const touchEndY = event.changedTouches[0].clientY;

    const dx = touchEndX - touchStartX;
    const dy = touchEndY - touchStartY;

    if (Math.abs(dx) > Math.abs(dy)) {
        if (dx > 30) {
            moveTiles(1, 0);
        } else if (dx < -30) {
            moveTiles(-1, 0);
        }
    } else {
        if (dy > 30) {
            moveTiles(0, 1);
        } else if (dy < -30) {
            moveTiles(0, -1);
        }
    }

    event.preventDefault();
}

function initShaders() {
    const gridVertexShaderSource = `
        attribute vec2 position;
        uniform mat4 matrix;
        void main() {
```

```
        gl_Position = matrix * vec4(position, 0.0, 1.0);
    }
`;

const gridFragmentShaderSource = `
    precision mediump float;
    uniform vec4 color;
    void main() {
        gl_FragColor = color;
    }
`;

const tileVertexShaderSource = `
    attribute vec2 position;
    attribute vec2 texcoord;
    varying vec2 vTexcoord;
    uniform mat4 matrix;
    void main() {
        vTexcoord = texcoord;
        gl_Position = matrix * vec4(position, 0.0, 1.0);
    }
`;

const tileFragmentShaderSource = `
    precision mediump float;
    varying vec2 vTexcoord;
    uniform vec4 color;
    uniform sampler2D sampler;
    uniform bool hasTexture;
    void main() {
        if (hasTexture) {
            vec4 texColor = texture2D(sampler, vTexcoord);
```

```
            gl_FragColor = texColor * color;
        } else {
            gl_FragColor = color;
        }
    }
`;

    gridProgram = createShaderProgram(gridVertexShaderSource, gridFragmentShaderSource);
    matrixLocation = gl.getUniformLocation(gridProgram, 'matrix');
    colorLocation = gl.getUniformLocation(gridProgram, 'color');

    tileProgram = createShaderProgram(tileVertexShaderSource, tileFragmentShaderSource);
    tileMatrixLocation = gl.getUniformLocation(tileProgram, 'matrix');
    tileColorLocation = gl.getUniformLocation(tileProgram, 'color');
    textureLocation = gl.getUniformLocation(tileProgram, 'sampler');
    hasTextureLocation = gl.getUniformLocation(tileProgram, 'hasTexture');
}

function createShaderProgram(vertexSource, fragmentSource) {
    const vertexShader = gl.createShader(gl.VERTEX_SHADER);
    gl.shaderSource(vertexShader, vertexSource);
    gl.compileShader(vertexShader);

    if (!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)) {
        console.error('Помилка компіляції шейдера:', gl.getShaderInfoLog(vertexShader));
        return null;
    }

    const fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
    gl.shaderSource(fragmentShader, fragmentSource);
    gl.compileShader(fragmentShader);
```

```
        if (!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)) {
            console.error('Помилка компіляції шейдера:',
gl.getShaderInfoLog(fragmentShader));
            return null;
        }

        const program = gl.createProgram();
        gl.attachShader(program, vertexShader);
        gl.attachShader(program, fragmentShader);
        gl.linkProgram(program);

        if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
            console.error('Помилка компіляції шейдера:', gl.getProgramInfoLog(program));
            return null;
        }

        return program;
    }

    async function loadTextures() {
        const values = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048];

        for (const value of values) {
            textures[value] = await createTextureFromText(value.toString());
        }
    }

    function createTextureFromText(text) {
        return new Promise(resolve => {
            const canvas = document.createElement('canvas');
            canvas.width = 64;
```

```
            canvas.height = 64;
            const ctx = canvas.getContext('2d');

            ctx.fillStyle = 'rgba(255, 255, 255, 1)';
            ctx.fillRect(0, 0, canvas.width, canvas.height);

            let fontSize;
            if (text.length === 1) fontSize = 32;
            else if (text.length === 2) fontSize = 28;
            else if (text.length === 3) fontSize = 20;
            else fontSize = 16;

            ctx.fillStyle = 'black';
            ctx.font = `bold ${fontSize}px Arial`;
            ctx.textAlign = 'center';
            ctx.textBaseline = 'middle';
            ctx.fillText(text, canvas.width / 2, canvas.height / 2);

            const texture = gl.createTexture();
            gl.bindTexture(gl.TEXTURE_2D, texture);
            gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
canvas);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);

            resolve(texture);
        });
    }
```

```
function startGame() {
    tiles = [];
    score = 0;
    gameOver = false;
    victory = false;
    document.getElementById('score').textContent = '0';
    document.getElementById('game-over').style.display = 'none';

    addRandomTile();
    addRandomTile();

    if (!gameRunning) {
        startTime = Date.now();
        gameRunning = true;
        updateTimer();
    }
}

function restartGame() {
    startGame();
}

function pageAuthOpen() {
    window.location.href = 'auth.html';
}

function updateTimer() {
    if (!gameRunning) return;

    const currentTime = Date.now();
    gameTime = Math.floor((currentTime - startTime) / 1000);
```

```
        const minutes = Math.floor(gameTime / 60);
        const seconds = gameTime % 60;

        document.getElementById('time').textContent = `${minutes.toString().padStart(2, '0')}:${seconds.toString().padStart(2, '0')}`;

        setTimeout(updateTimer, 1000);
    }

    function render() {
        gl.viewport(0, 0, canvas.width, canvas.height);
        gl.clear(gl.COLOR_BUFFER_BIT);

        drawGrid();
        drawTiles();

        requestAnimationFrame(render);
    }

    function drawGrid() {
        gl.useProgram(gridProgram);

        const projectionMatrix = mat4.create();
        mat4.ortho(projectionMatrix, -1, 1, 1, -1, -1, 1);
        gl.uniformMatrix4fv(matrixLocation, false, projectionMatrix);

        for (let i = 0; i <= GRID_SIZE; i++) {
            const pos = i / GRID_SIZE * 2 - 1;

            drawLine(pos, -1, pos, 1, [0.8, 0.7, 0.6, 1.0]);

            drawLine(-1, pos, 1, pos, [0.8, 0.7, 0.6, 1.0]);
```

```
    }
}

function drawLine(x1, y1, x2, y2, color) {
    gl.uniform4fv(colorLocation, color);

    const positionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

    const positions = [x1, y1, x2, y2];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);

    const positionAttributeLocation = gl.getAttribLocation(gridProgram, 'position');
    gl.enableVertexAttribArray(positionAttributeLocation);
    gl.vertexAttribPointer(positionAttributeLocation, 2, gl.FLOAT, false, 0, 0);

    gl.drawArrays(gl.LINES, 0, 2);
}

function drawTiles() {
    gl.useProgram(tileProgram);

    const projectionMatrix = mat4.create();
    mat4.ortho(projectionMatrix, -1, 1, 1, -1, -1, 1);

    for (const tile of tiles) {
        if (tile) {
            updateTilePosition(tile);
            drawTile(tile, projectionMatrix);
        }
    }
}
```

```
function updateTilePosition(tile) {
    if (tile.isMoving()) {
        tile.drawX += (tile.x - tile.drawX) * ANIMATION_SPEED;
        tile.drawY += (tile.y - tile.drawY) * ANIMATION_SPEED;

        if (Math.abs(tile.drawX - tile.x) < 0.01 && Math.abs(tile.drawY - tile.y) < 0.01) {
            tile.drawX = tile.x;
            tile.drawY = tile.y;
        }
    }
}

function drawTile(tile, projectionMatrix) {
    const normalizedX = tile.drawX / GRID_SIZE * 2 - 1 + CELL_SIZE;
    const normalizedY = tile.drawY / GRID_SIZE * 2 - 1 + CELL_SIZE;
    const normalizedSize = (CELL_SIZE * 2) - (CELL_MARGIN * 2);

    const modelMatrix = mat4.create();
    mat4.translate(modelMatrix, projectionMatrix, [normalizedX, normalizedY, 0]);
    mat4.scale(modelMatrix, modelMatrix, [normalizedSize, normalizedSize, 1]);

    gl.uniformMatrix4fv(tileMatrixLocation, false, modelMatrix);

    const tileColor = getTileColor(tile.value);
    gl.uniform4fv(tileColorLocation, tileColor);

    const positionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

    const positions = [
        -0.5, -0.5,
```

```
    0.5, -0.5,
    0.5, 0.5,
    -0.5, 0.5
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);

  const positionAttributeLocation = gl.getAttribLocation(tileProgram, 'position');
  gl.enableVertexAttribArray(positionAttributeLocation);
  gl.vertexAttribPointer(positionAttributeLocation, 2, gl.FLOAT, false, 0, 0);

  const texcoordBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, texcoordBuffer);

  const texcoords = [
    0.0, 0.0,
    1.0, 0.0,
    1.0, 1.0,
    0.0, 1.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(texcoords),
gl.STATIC_DRAW);

  const texcoordAttributeLocation = gl.getAttribLocation(tileProgram, 'texcoord');
  gl.enableVertexAttribArray(texcoordAttributeLocation);
  gl.vertexAttribPointer(texcoordAttributeLocation, 2, gl.FLOAT, false, 0, 0);

  const indexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);

  const indices = [
    0, 1, 2,
    0, 2, 3          ];
```

```
        gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
gl.STATIC_DRAW);

        if (textures[tile.value]) {
            gl.uniform1i(hasTextureLocation, 1);
            gl.activeTexture(gl.TEXTURE0);
            gl.bindTexture(gl.TEXTURE_2D, textures[tile.value]);
            gl.uniform1i(textureLocation, 0);
        } else {
            gl.uniform1i(hasTextureLocation, 0);
        }

        gl.drawElements(gl.TRIANGLES, 6, gl.UNSIGNED_SHORT, 0);
    }

    function getTileColor(value) {
        switch(value) {
        case 2: return [0.93, 0.89, 0.85, 1.0];
        case 4: return [0.93, 0.88, 0.78, 1.0];
        case 8: return [0.95, 0.69, 0.47, 1.0];
        case 16: return [0.96, 0.58, 0.39, 1.0];
        case 32: return [0.96, 0.49, 0.37, 1.0];
        case 64: return [0.96, 0.37, 0.23, 1.0];
        case 128: return [0.93, 0.81, 0.45, 1.0];
        case 256: return [0.93, 0.78, 0.38, 1.0];
        case 512: return [0.93, 0.75, 0.31, 1.0];
        case 1024: return [0.93, 0.72, 0.23, 1.0];
        case 2048: return [0.93, 0.69, 0.16, 1.0];
        case 4096: return [0.45, 0.33, 0.84, 1.0];
        case 8192: return [0.29, 0.0, 0.51, 1.0];
        case 16384: return [0.13, 0.55, 0.13, 1.0];
        case 32768: return [0.0, 0.39, 0.0, 1.0];
```

```
        default: return [0.2, 0.2, 0.2, 1.0];
    }
}
    class Tile {
        constructor(x, y, value) {
            this.x = x;
            this.y = y;
            this.drawX = x;
            this.drawY = y;
            this.value = value;
            this.hasMerged = false;
        }

        isMoving() {
            return Math.abs(this.drawX - this.x) > 0.01 || Math.abs(this.drawY - this.y) > 0.01;
        }

        doubleValue() {
            this.value *= 2;
            updateScore(this.value);
        }

        moveTo(newX, newY) {
            this.x = newX;
            this.y = newY;
        }
    }

    function addRandomTile() {
        const emptyPositions = [];

        for (let x = 0; x < GRID_SIZE; x++) {
```

```
        for (let y = 0; y < GRID_SIZE; y++) {
            if (!getTileAt(x, y)) {
                emptyPositions.push({x, y});
            }
        }
    }

    if (emptyPositions.length > 0) {
        const position = emptyPositions[Math.floor(Math.random() * emptyPositions.length)];
        const value = Math.random() < 0.9 ? 2 : 4;
        const newTile = new Tile(position.x, position.y, value);
        setTileAt(position.x, position.y, newTile);
    }
}

function getTileAt(x, y) {
    return tiles[y * GRID_SIZE + x];
}

function setTileAt(x, y, tile) {
    tiles[y * GRID_SIZE + x] = tile;
}

function removeTileAt(x, y) {
    tiles[y * GRID_SIZE + x] = null;
}

function moveTiles(dx, dy) {
    let moved = false;
    resetMergeFlags();

    const processOrder = [];
```

```
if (dx === 1) {
    for (let y = 0; y < GRID_SIZE; y++) {
        for (let x = GRID_SIZE - 1; x >= 0; x--) {
            processOrder.push({x, y});
        }
    }
} else if (dx === -1) {
    for (let y = 0; y < GRID_SIZE; y++) {
        for (let x = 0; x < GRID_SIZE; x++) {
            processOrder.push({x, y});
        }
    }
} else if (dy === 1) {
    for (let x = 0; x < GRID_SIZE; x++) {
        for (let y = GRID_SIZE - 1; y >= 0; y--) {
            processOrder.push({x, y});
        }
    }
} else if (dy === -1) {
    for (let x = 0; x < GRID_SIZE; x++) {
        for (let y = 0; y < GRID_SIZE; y++) {
            processOrder.push({x, y});
        }
    }
}

for (const {x, y} of processOrder) {
    if (tryMoveTile(x, y, dx, dy)) {
        moved = true;
    }
}
```

```
    if (moved) {
        resetMergeFlags();
        addRandomTile();
        checkGameOver();
    }
}

function tryMoveTile(x, y, dx, dy) {
    const tile = getTileAt(x, y);
    if (!tile) return false;

    let newX = x + dx;
    let newY = y + dy;
    let moved = false;

    while (isInsideGrid(newX, newY) && !getTileAt(newX, newY)) {
        removeTileAt(x, y);
        setTileAt(newX, newY, tile);
        tile.moveTo(newX, newY);

        x = newX;
        y = newY;
        newX += dx;
        newY += dy;
        moved = true;
    }

    if (isInsideGrid(newX, newY)) {
        const targetTile = getTileAt(newX, newY);
```

```
            if (targetTile && targetTile.value === tile.value && !targetTile.hasMerged &&
!tile.hasMerged) {
                removeTileAt(x, y);
                targetTile.doubleValue();
                targetTile.hasMerged = true;
                moved = true;
            }
        }

        return moved;
    }

    function isInsideGrid(x, y) {
  return x >= 0 && x < GRID_SIZE && y >= 0 && y < GRID_SIZE;
}

function resetMergeFlags() {
  for (const tile of tiles) {
    if (tile) {
      tile.hasMerged = false;
    }
  }
}

function updateScore(points) {
  score += points;
  document.getElementById('score').textContent = score;
}

function checkGameOver() {
  if (hasWon()) {
```

```
        if (!victory) {
            victory = true;
            showGameOverMessage('Ви перемогли!', true);
        }
        return;
    }

    if (hasLost()) {
        gameOver = true;
        showGameOverMessage('Гру закінчено!', false);
        return;
    }
}

function hasWon() {
    for (const tile of tiles) {
        if (tile && tile.value === 2048) {
            return true;
        }
    }
    return false;
}

function hasLost() {
    for (let x = 0; x < GRID_SIZE; x++) {
        for (let y = 0; y < GRID_SIZE; y++) {
            if (!getTileAt(x, y)) {
                return false;
            }
        }
    }
```

```
    for (let x = 0; x < GRID_SIZE; x++) {
        for (let y = 0; y < GRID_SIZE; y++) {
            const tile = getTileAt(x, y);

            if (x < GRID_SIZE - 1) {
                const rightTile = getTileAt(x + 1, y);
                if (rightTile && tile.value === rightTile.value) {
                    return false;
                }
            }

            if (y < GRID_SIZE - 1) {
                const bottomTile = getTileAt(x, y + 1);
                if (bottomTile && tile.value === bottomTile.value) {
                    return false;
                }
            }
        }
    }

    return true;
}

function showGameOverMessage(message, isWin) {
    const gameOverElement = document.getElementById('game-over');
    const messageElement = document.getElementById('game-over-message');

    messageElement.textContent = message;

    if (isWin) {
        messageElement.classList.add('win-message');
```

```
    } else {
        messageElement.classList.remove('win-message');
    }

    gameOverElement.style.display = 'flex';
    gameRunning = false;
}

function saveHighScore() {
    const highScore = localStorage.getItem('2048_highscore') || 0;

    if (score > highScore) {
        localStorage.setItem('2048_highscore', score);
    }
}
```