Console Minesweeper

Sprawozdanie Projektowe

Kamil Kotorc

Autor projektu

Temat

Nazwa

Konsolowy Saper w C++

Opis

Tematem projektu jest prosta gra komputerowa, bazująca na popularnej grze Saper (z ang. Minesweeper). To dobrze znany program wydany już w 1989 roku, który zawitał na stałe jako akcesorium w systemie Windows 7, gdzie zyskał na sporej popularności.

Jest to już kultowa gra, a sam schemat jej działania jest dla mnie bardzo fascynujący. Dlatego postanowiłem zrobić swoją odmienną wersję, napisaną od podstaw w C++, z wykorzystaniem bibliotek omówionych na zajęciach laboratoryjnych.

Gra Saper ma prostą, ale angażującą mechanikę, która polega na odkrywaniu pól na planszy, unikając przy tym min. Jest to gra logiczna, która wymaga od gracza umiejętności analitycznego myślenia i przewidywania, co sprawia, że jest ona zarówno wyzwaniem, jak i satysfakcjonującą rozrywką.

Analiza tematu

Schemat działania

Gra Saper to nieskomplikowana logiczna gra jednoosobowa.

Plansza

Gra toczy się na planszy o określonych wymiarach, która składa się z siatki pól. Gracz może wybrać poziom trudności, co determinuje rozmiar planszy oraz liczbę min na niej rozmieszczonych.

Rodzaje pól

- Puste pola: Pola, które nie zawierają miny i nie sąsiadują z żadną miną.
- Pola z minami: Pola, które zawierają minę. Odkrycie takiego pola kończy grę przegraną.
- Pola z liczbami: Pola, które wskazują liczbę min znajdujących się w sąsiednich polach (do ośmiu sąsiednich pól).

Rozpoczęcie gry

Na początku gry wszystkie pola są zakryte, a gracz nie wie, co się pod nimi znajduje. Gracz wybiera pole do odkrycia, starając się unikać min. Odkryte pole ujawnia, czy jest puste, z miną, czy z liczbą.

Podpowiedzi

Pola z liczbami działają jak podpowiedzi, wskazując graczowi, ile min znajduje się w sąsiedztwie danego pola. Na podstawie tych podpowiedzi gracz musi logicznie rozważyć, które pola są bezpieczne, a które mogą zawierać miny.

Flagi

Gracz może oznaczać podejrzane pola flagami, sygnalizując, że podejrzewa, iż znajduje się tam mina. Flagi pomagają graczowi śledzić podejrzane pola i unikać ich odkrywania.

Warunki wygranej i przegranej

- ✓ Gra kończy się wygraną, gdy wszystkie pola bez min zostaną odkryte, a wszystkie miny zostaną poprawnie oznaczone flagami.
- ✗ Gra kończy się przegraną, gdy gracz odkryje pole z miną – mina wybucha i gra się kończy.

Interfejs użytkownika

Gra działa w trybie tekstowym w konsoli, gdzie plansza jest reprezentowana przez znaki. Użytkownik wprowadza komendy za pomocą klawiatury, aby odkrywać pola, stawiać flagi, zapisywać stan gry, wczytywać zapisany stan i wyświetlać pomoc.

Uzasadnienie wyboru klas

Cell

Podstawowa klasa reprezentująca pojedyncze pole na planszy. Dziedziczą po niej klasy specjalizujące się w różnych typach pól (MineCell, EmptyCell, NumberCell). Umożliwia to zastosowanie polimorfizmu, co ułatwia zarządzanie różnymi typami pól.

MineCell, EmptyCell, NumberCell

Klasy dziedziczące po Cell, reprezentujące odpowiednio pole z miną, puste pole oraz pole z liczbą wskazującą na ilość sąsiadujących min. Podział ten jest naturalny i intuicyjny, odzwierciedla różnorodność pól w grze.

Board

Klasa zarządzająca stanem gry i planszy. Przechowuje stan wszystkich komórek, inicjalizuje planszę, zarządza odkrywaniem pól, stawianiem flag, zapisywaniem i wczytywaniem stanu gry. Jest to centralna klasa zawierająca logikę gry.

Game

Klasa zarządzająca interakcjami użytkownika oraz kontrolująca przepływ gry. Odpowiada za wyświetlanie menu, odbieranie komend od użytkownika oraz aktualizację stanu gry.

Uzasadnienie wyboru algorytmów

Losowe rozmieszczenie min

Użycie `std::rand()` do losowego rozmieszczania min na planszy.
Zapewnia losowość i nieprzewidywalność, co jest kluczowe dla gry Saper.

Rekurencyjne odkrywanie pustych pól

Algorytm rekurencyjny do odkrywania sąsiednich pustych pól.
Umożliwia odkrywanie dużych obszarów planszy po odkryciu pustego pola.

Sprawdzanie sąsiednich komórek

Użycie pętli zagnieżdżonych do iteracji przez sąsiednie komórki.
Pozwala na efektywne zliczanie min wokół danego pola oraz odkrywanie sąsiednich pól.

Walidacja komend za pomocą wyrażeń regularnych

Użycie wyrażeń regularnych do sprawdzania poprawności wpisanych komend przez użytkownika, co zapewnia lepszą obsługę błędów i zabezpiecza program przed nieprawidłowymi danymi wejściowymi.

Uzasadnienie wyboru bibliotek

<regex>

Użycie tej biblioteki pozwala na walidację wejściowych komend użytkownika przy użyciu wyrażeń regularnych, co zwiększa czytelność i elastyczność kodu.

<thread>

Biblioteka wątków umożliwia asynchroniczne uruchomienie licznika czasu gry, co pozwala na śledzenie czasu rozgrywki bez zakłócania interakcji użytkownika.

<ranges>

Użycie tej biblioteki pozwala na bardziej idiomatyczne i czytelne iterowanie po kontenerach. Ułatwia operacje na zakresach.

<filesystem>

Umożliwia wygodne operacje na plikach i ścieżkach, takie jak sprawdzanie istnienia plików, tworzenie ścieżek i zarządzanie plikami.

<iostream>, <vector>, <memory>, <fstream>, <algorithm>

Standardowe biblioteki C++ używane do wejścia/wyjścia, zarządzania kontenerami, dynamicznego zarządzania pamięcią oraz podstawowych operacji algorytmicznych.

Specyfikacja zewnętrzna

Instrukcja dla użytkownika

1. *Uruchom program.*
2. *Wybierz poziom trudności wpisując odpowiednią liczbę (1, 2 lub 3).*
3. *Użyj dostępnych komend do interakcji z grą:*
    - ***help** - Wyświetla listę dostępnych komend.*
    - ***save** - Zapisuje aktualny stan gry do pliku.*
    - ***load** - Wczytuje zapisany stan gry z pliku.*
    - ***exit** - Kończy bieżącą grę i wraca do menu głównego.*
    - ***![coordinates]**: Stawia flagę na podanych koordynatach (np. `!D5`) lub usuwa flagę, jeśli już tam jest.*
    - ***[coordinates]**: Odkrywa pole na podanych koordynatach (np. `D5`).*
4. *Gra kończy się wygraną, jeśli poprawnie oznaczysz wszystkie pola z minami, lub przegraną, jeśli odkryjesz pole z miną.*

Przykłady działania

Przykład 1: Start gry

```
 * MINESWEEPER *

Select difficulty level:
1. Easy (6x6 with 5 mines)
2. Medium (8x8 with 10 mines)
3. Hard (10x10 with 20 mines)
Enter your choice: 1
```

Przykład 2: Wyświetlanie planszy i interakcje użytkownika

```
  0 1 2 3 4 5
A . . . . . .
B . . . . . .
C . . . . . .
D . . . . . .
E . . . . . .
F . . . . . .

Flags remaining: 5
Type 'help' for a list of commands.
Enter action and coordinates: D3
```

Przykład 3: Wygrana

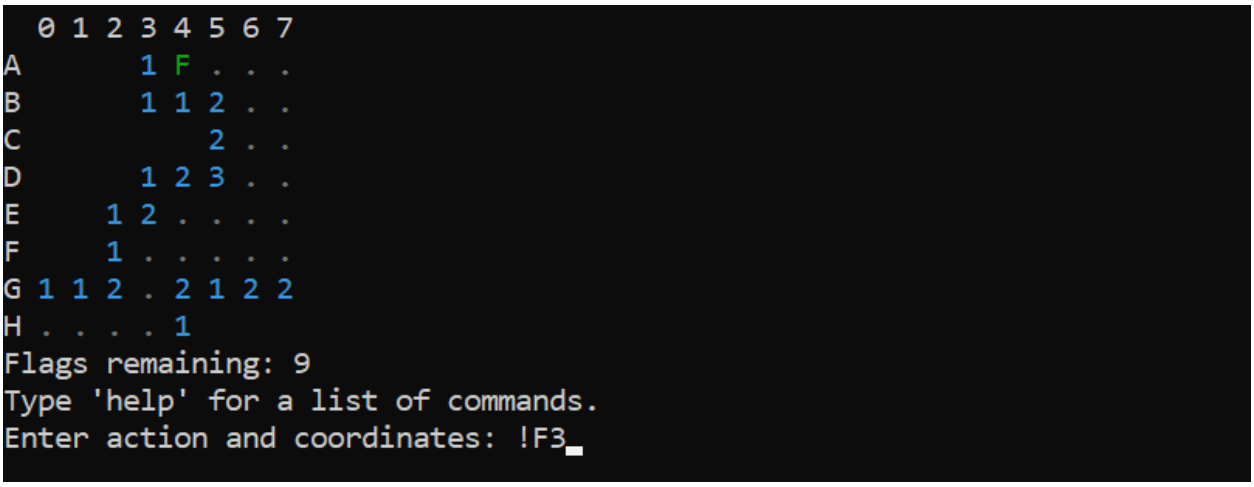
```
Congratulations! You flagged all the mines correctly. You win!
Total game time: 04:08
```

Przykład 4: Przegrana

```
You hit a mine! Game over. :(
Total game time: 01:24
Do you want to play again? (y/n):
```

Opis trwającej rozgrywki

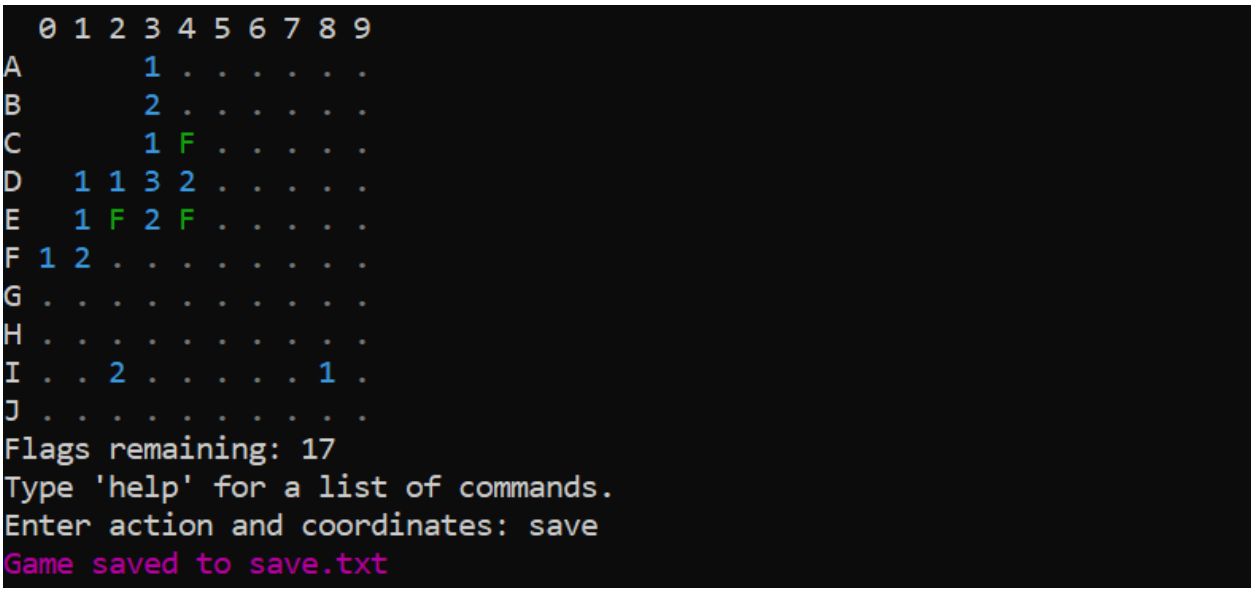
Zrzut ekranu



Opis ruchów

Gracz, odkrywając puste pole, odkrył również pola dookoła. Po analizie pól z liczbami min wokół, stwierdza, że mina na pewno znajduje się na pozycji A4, dlatego oznaczył to pole flagą. Następnym potencjalnym polem z miną może być E4, ze względu na „1” w polu D3 i fakt, że wszystkie pola wokół zostały już odkryte.

Zrzut ekranu



Opis ruchów

Gracz, wybrał trudną planszę. Po kilku rundach stwierdza, że jednak musi kończyć rozgrywkę. Zapisuje grę, aby mógł ją dokończyć później, o czym zostaje poinformowany fioletowym tekstem.

Specyfikacja wewnętrzna

Klasy

Cell

Znaczenie obiektu: Reprezentuje pojedynczą komórkę na planszy gry.

Powiązania z innymi klasami: Klasa bazowa dla MineCell, EmptyCell, NumberCell.

```
class Cell {
protected:
    bool revealed;
    bool flagged;

public:
    Cell() : revealed(false), flagged(false) {}
    virtual ~Cell() = default;

    virtual void display() const = 0;
    virtual CellType getType() const = 0;

    bool isRevealed() const { return revealed; }
    bool isFlagged() const { return flagged; }
    void reveal() { revealed = true; }
    void flag() { flagged = true; }
    void unflag() { flagged = false; }

    virtual bool isMine() const = 0;
    virtual std::string toString() const = 0;
    static std::unique_ptr<Cell> fromString(const std::string& str);
};
```

NumberCell

Znaczenie obiektu: Reprezentuje komórkę z liczbą wskazującą ilość sąsiadujących min.

Powiązania z innymi klasami: Dziedziczy po klasie Cell.

```
class NumberCell : public Cell {
    int mineCount;

public:
    NumberCell(int count) : mineCount(count) {}
    void display() const override;
    CellType getType() const override { return CellType::Number; }
    bool isMine() const override { return false; }
    std::string toString() const override { return "N"+std::to_string(mineCount); }
    int getMineCount() const { return mineCount; }
};
```

MineCell

Znaczenie obiektu: Reprezentuje komórkę zawierającą minę.

Powiązania z innymi klasami: Dziedziczy po klasie Cell.

```
class MineCell : public Cell {
public:
    void display() const override;
    CellType getType() const override { return CellType::Mine; }
    bool isMine() const override { return true; }
    std::string toString() const override { return "M"; }
};
```

EmptyCell

Znaczenie obiektu: Reprezentuje pustą komórkę.

Powiązania z innymi klasami: Dziedziczy po klasie Cell.

```
class EmptyCell : public Cell {
public:
    void display() const override;
    CellType getType() const override { return CellType::Empty; }
    bool isMine() const override { return false; }
    std::string toString() const override { return "E"; }
};
```

Game

Znaczenie obiektu: Zarządza interakcjami użytkownika i kontroluje przepływ gry.

Powiązania z innymi klasami: Zawiera instancję Board.

```
class Game {
    Board board;
    bool isGameOver;

public:
    Game(int rows, int cols);
    Game(int rows, int cols, int mines);
    ~Game();
    Game(Game&& other) noexcept;
    Game& operator=(Game&& other) noexcept;

    void play();
    void reset(int rows, int cols, int mines);
    bool checkWinCondition() const;
    void printHelp() const;
};
```

Board

Znaczenie obiektu: Zarządza stanem gry i planszy. Zawiera także główne funkcje wykonujące operacje na komórkach.

Powiązania z innymi klasami: Zawiera instancje komórek Cell.

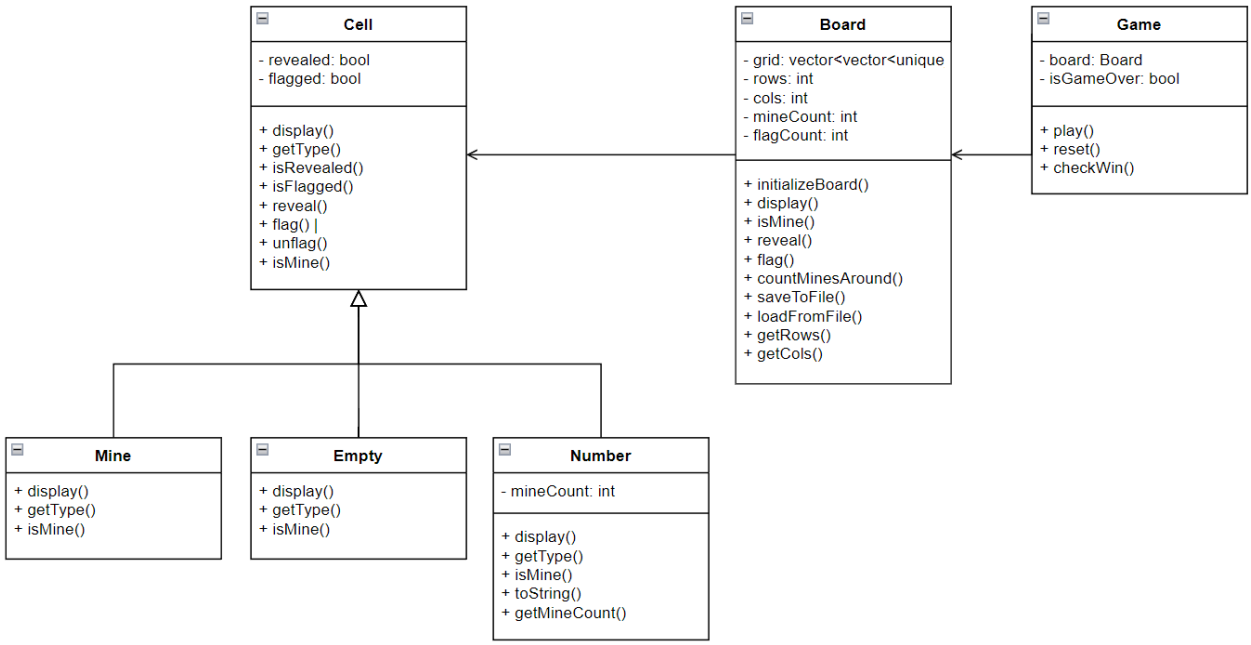
```
class Board {
    std::vector<std::vector<std::unique_ptr<Cell>>> grid;
    int rows;
    int cols;
    int mineCount;
    int flagCount;

public:
    Board(int r, int c);
    Board(int r, int c, int mines);
    Board(const Board&) = delete;
    Board& operator=(const Board&) = delete;
    Board(Board&&) noexcept;
    Board& operator=(Board&&) noexcept;

    void initializeBoard();
    void initializeBoardWithMines(int mines);
    void display() const;
    bool isMine(int x, int y) const;
    bool reveal(int x, int y);
    void flag(int x, int y);
    void revealAllMines();
    void revealAdjacentCells(int x, int y);
    int countMinesAround(int x, int y) const;
    void saveToFile(const std::string& filename) const;
    void loadFromFile(const std::string& filename);
    int getFlagCount() const;
    bool allMinesFlagged() const;
    int getRows() const { return rows; }
    int getCols() const { return cols; }

};
```

Diagram hierarchii klas



Istotne struktury danych i algorytmy

Struktury danych

- **vector<vector<unique_ptr<Cell>>> grid**
  Dwuwymiarowy wektor dynamicznie alokowanych unikalnych wskaźników na obiekty klasy Cell. Reprezentuje planszę gry.
- **int rows, cols**
  Liczby całkowite reprezentujące wymiary planszy.
- **int mineCount**
  Liczba całkowita przechowująca liczbę min na planszy.
- **Int flagCount**
  Liczba całkowita przechowująca liczbę dostępnych flag.

Algorytmy

Losowe rozmieszczenie min:

- Użycie std::rand() do losowego wybierania pozycji dla min.
- W pętli do-while losowo wybierane są pozycje dla min, dopóki liczba rozmieszczonych min nie osiągnie wymaganej liczby.

Rekurencyjne odkrywanie pustych pól:

- Metoda revealAdjacentCells wykorzystuje rekurencję do odkrywania sąsiednich pustych pól, jeśli bieżące pole jest puste.
- Iteracja po sąsiednich polach za pomocą zagnieżdżonych pętli for.

Sprawdzanie sąsiednich komórek:

- Metoda countMinesAround iteruje po sąsiednich komórkach wokół danej pozycji, zliczając liczbę min.
- Zagnieżdżone pętle for do iteracji po sąsiednich komórkach.

Dodatkowe funkcjonalności

Wielowątkowość:

- Użycie wątku (std::thread) do śledzenia czasu gry bez zakłócania interakcji użytkownika.
- std::atomic<bool> do synchronizacji zakończenia wątku.

Regex:

- Użycie wyrażeń regularnych (std::regex) do walidacji wejściowych komend użytkownika, co zwiększa czytelność i elastyczność kodu.

Wykorzystane techniki obiektowe

Dziedziczenie

Klasy MineCell, EmptyCell i NumberCell dziedziczą po klasie bazowej Cell.

Pozwala to na zastosowanie wspólnych metod oraz dzielenie się wspólnymi atrybutami.

Polimorfizm

Metody display i isMine są zadeklarowane jako wirtualne w klasie `Cell` i mają różne implementacje w klasach dziedziczących (MineCell, EmptyCell, NumberCell).

Pozwala to na dynamiczne wywoływanie odpowiednich metod w zależności od rzeczywistego typu obiektu.

Kapsułkowanie

Atrybuty klas są prywatne lub chronione, a dostęp do nich odbywa się za pomocą publicznych metod.

Chroni wewnętrzny stan obiektów przed nieautoryzowanym dostępem i modyfikacją.

Kompozycja

Klasa `Board` zawiera wektor unikalnych wskaźników do obiektów klasy Cell, tworząc relację "has-a".

Pozwala to na dynamiczne zarządzanie komórkami planszy.

Ogólny schemat działania programu

Inicjalizacja gry

Użytkownik uruchamia program i wybiera poziom trudności.

Program tworzy planszę o odpowiednich wymiarach i liczbie min.

Interakcje użytkownika

Użytkownik wprowadza komendy, aby odkrywać pola, stawiać flagi, zapisywać stan gry, wczytywać zapisany stan i wyświetlać pomoc.

Program aktualizuje stan planszy zgodnie z komendami użytkownika.

Logika gry

Program sprawdza, czy użytkownik odkrył pole z miną (przegrana) lub poprawnie oznaczył wszystkie miny flagami (wygrana).

W przypadku odkrycia pustego pola program rekurencyjnie odkrywa sąsiednie pola.

Zapisywanie i wczytywanie gry

Program zapisuje stan gry do pliku lub wczytuje zapisany stan z pliku przy użyciu biblioteki `<filesystem>`.

Zakończenie gry

Gra kończy się wygraną lub przegraną, a użytkownik ma możliwość rozpoczęcia nowej gry lub wyjścia.

Testowanie

Opis

Aby poradzić sobie z wykryciem błędów i potencjalnych problemów jakie może napotkać użytkownik podczas rozgrywki, testowałem nowo dodane funkcje, co okazało się bardzo pomocne. W kontekście tego projektu miało to szczególne znaczenie ze względu na liczne skomplikowane interakcje między klasami oraz operacje na danych. Pracując nad implementacją gry , napotykałem na różne wyzwania, zwłaszcza te związane z walidacją komend użytkownika, zarządzaniem stanem gry oraz operacjami wejścia/wyjścia z plików.

Niektóre ze sposobów wykrywania błędów z wycinkami kodu

Testowanie interakcji użytkownika

- Wprowadzanie poprawnych i niepoprawnych komend, odkrywanie pól, stawianie flag.
- Wypisywanie na konsoli informacji o aktualnym stanie planszy, liczbie pozostałych flag i wynikach akcji użytkownika.

```
std::cout << "Enter action and coordinates: ";
std::cin >> input;
std::cout << "You entered: " << input << std::endl;
```

Testowanie warunków wygranej i przegranej

- Odkrycie pola z miną, poprawne oznaczenie wszystkich min flagami.
- Wypisywanie na konsoli informacji o liczbie pozostałych min i flag.

```
if (board.reveal(row, col)) {
    std::cout << "You hit a mine! Game over.\n";
}
else {
    std::cout << "Safe move.\n";
}
```

Testowanie zapisywania i wczytywania gry

- Zapisywanie stanu gry do pliku i wczytywanie zapisanego stanu.
- Weryfikacja poprawności danych poprzez porównywanie stanów planszy przed zapisaniem i po wczytaniu.

```
board.saveToFile("save.txt");
std::cout << "Game saved to save.txt\n";
board.loadFromFile("save.txt");
std::cout << "Game loaded from save.txt\n";
```

Testowanie rekurencyjnego odkrywania pól

- Odkrywanie pustego pola i sprawdzanie, czy sąsiednie pola są poprawnie odkrywane.
- Wypisywanie na konsoli stanu sąsiednich pól przed i po odkryciu.

```
board.reveal(row, col);
std::cout << "Revealed cell at (" << row << ", " << col << ")\n";
```

Testowanie wielowątkowości:

- Uruchamianie wątku do śledzenia czasu gry i sprawdzanie, czy czas jest poprawnie mierzony i wyświetlany po zakończeniu gry.
- Wypisywanie na konsoli czasu gry co sekundę.

```
auto start = std::chrono::steady_clock::now();
// Wątek śledzący czas gry
std::thread timerThread([&]() {
    while (!isGameOver) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        auto now = std::chrono::steady_clock::now();
        auto elapsed = std::chrono::duration_cast<std::chrono::seconds>(now -
start);
        std::cout << "Time elapsed: " << elapsed.count() << " seconds\n";
    }
});

timerThread.detach();
```

Testowanie wyrażeń regularnych:

- Wprowadzanie różnych komend i sprawdzanie, czy są poprawnie rozpoznawane i przetwarzane przez program.
- Wypisywanie na konsoli wyników walidacji komend.

```
if (std::regex_match(input, flagRegex)) {
    std::cout << "Valid flag command\n";
}
else if (std::regex_match(input, revealRegex)) {
    std::cout << "Valid reveal command\n";
}
else {
    std::cout << "Invalid command\n";

}
```

Kompilacja programu

g++ -std=c++20 -fmodules-ts -c time.ixx -o time.o

g++ -std=c++20 main.cpp time.o Cell.cpp Board.cpp Game.cpp -o saper

Wnioski

Podczas pisania projektu musiałem stawić czoła wielu wyzwaniom i dokładnie zapoznać się z zasadami programowania obiektowego w języku C++. Największym problemem okazało się jednak debugowanie programu. Często podczas implementacji nowych metod, wcześniej poprawnie działające funkcje wymagały całkowitego przeprojektowania. Wiele nowo poznanych bibliotek okazało się bardzo przydatnych. Na przykład, użycie wyrażeń regularnych było bardziej efektywne i prostsze niż standardowe pętle porównań do sprawdzania poprawności danych wpisywanych przez użytkownika. Zastąpienie standardowych pętli iteratorami z biblioteki ranges również zaoszczędziło kilka linijek kodu, a co najważniejsze, uczyniło go bardziej czytelnym. Ze względu na to, że program był wyświetlany w standardowej konsoli, nie było dużego pola do popisu w kwestii grafiki. Jednakże, użycie kolorowych tekstów przełamało monotonię szarości i dodało kontrastu. Po wyeliminowaniu błędów i zakończeniu projektu, pozostało jedynie cieszyć się rozgrywką, która, jak można się domyślić, okazała się dość wciągająca.

Uwagi

Szczególne wyrazy uznania dla dr inż. Piotra Pecki za świetną organizację zajęć, pomoc w rozwiązaniu niektórych problemów i koordynacje projektu. Znacząco wpłynął na realizacje projektu i przekaz materiału jaki był do przeprowadzenia.