29.01.2025

Joanna Kruk 334209

Jan Dardziński 337547

gra Saper („Minesweeper”)

przedmiot: Jezyki i metody programowania 1

link: https://github.com/jandardzinski/Saper

Wstęp projektu

Celem projektu jest zaimplementowanie gry "Saper" (ang. Minesweeper), która jest klasyczną grą logiczną wymagającą od gracza umiejętności dedukcji i planowania. Gra polega na odkrywaniu pól na planszy w taki sposób, aby unikać min, które zostały losowo rozmieszczone. Każde odkryte pole informuje gracza o liczbie min znajdujących się w sąsiedztwie danego pola, co stanowi wskazówkę do podejmowania dalszych decyzji.

Gra oferuje kilka poziomów trudności, od łatwego po trudny, a także możliwość definiowania własnych ustawień planszy, co umożliwia graczowi dostosowanie rozgrywki do własnych preferencji. Kluczowe cechy gry to obsługa flagowania pól, zapis wyników najlepszych graczy, obsługa ruchów gracza zarówno interaktywnie, jak i z pliku, oraz zapewnienie mechanizmu zapobiegającego trafieniu na minę podczas pierwszego ruchu.

Dodatkowo projekt zakłada spełnienie następujących wymagań technicznych:

- Plansza i wyniki są wypisywane w terminalu.
- Obsługiwane są polecenia użytkownika do odkrywania pól oraz flagowania min.
- Wyniki gracza są obliczane na podstawie liczby odkrytych pól i mnożnika zależnego od poziomu trudności.
- Implementacja uwzględnia modułowy podział kodu, możliwość testowania aplikacji za pomocą testów jednostkowych oraz korzystanie z narzędzia make do budowania aplikacji.
- Repozytorium kodu znajduje się na platformie GitHub, zapewniając wersjonowanie i przejrzystość rozwoju projektu.

Gra "Saper" stanowi świetne ćwiczenie w zakresie programowania, łącząc elementy algorytmiki, obsługi danych wejściowych/wyjściowych, testowania kodu oraz interakcji z użytkownikiem. Projekt jest również idealnym przykładem zastosowania dobrych praktyk inżynierii oprogramowania.

podręcznik użytkownika programu (opis zasad gry, sposób uruchomienia programu, opis poszczególnych opcji)

Podręcznik użytkownika programu – Gra Saper

Wprowadzenie

Gra **Saper** to klasyczna gra logiczna, w której gracz ma za zadanie odkrywać pola na planszy, unikając min. Na każdym odkrytym polu wyświetlana jest liczba min sąsiadujących z tym polem. Celem gry jest odkrycie wszystkich pól niebędących minami.

Zasady gry

1. **Plansza i miny**:
    a. Gra rozpoczyna się na prostokątnej planszy o wymiarach zależnych od wybranego trybu gry.
    b. Na planszy rozmieszczone są bomby (miny) w losowych pozycjach.
    c. Liczba bomb oraz rozmiar planszy zależą od trybu gry:
        i. **Łatwy**: 9x9 pól, 10 bomb.
        ii. **Średni**: 16x16 pól, 40 bomb.
        iii. **Trudny**: 16x30 pól, 99 bomb.
        iv. **Tryb niestandardowy**: Użytkownik definiuje liczbę pól i bomb.
2. **Cele gry**:
    a. Odkryć wszystkie pola niebędące minami.
    b. Unikać odkrycia pola zawierającego minę – odkrycie miny kończy grę porażką.
3. **Informacje o polach**:
    a. Odkryte pole może zawierać:
        i. Liczbę od 1 do 8 – wskazującą, ile min znajduje się w sąsiedztwie.
        ii. Pustą przestrzeń (0) – brak min w sąsiedztwie. Takie pola są automatycznie odkrywane razem z sąsiednimi polami.
        iii. Symbol miny (*) – pojawia się, gdy gracz odkryje pole z bombą (gra kończy się przegraną).
    b. Nieodkryte pole jest puste.
4. **Flagowanie pól**:
    a. Pola, które gracz podejrzewa o zawieranie min, można oznaczyć flagą. Pola z flagą nie mogą zostać przypadkowo odkryte.
    b. Flagi można włączać i usuwać w dowolnym momencie.
5. **Wygrana**:
    a. Gracz wygrywa, gdy wszystkie pola niebędące minami zostaną odkryte.

Obsługa programu

1. **Uruchamianie gry**:
    Aby uruchomić program, wpisz w terminalu:
    ```
    ./saper
    ```
    Możesz również uruchomić grę w trybie testowym, wczytując planszę i ruchy z pliku:
    ```
    ./saper test
    ```

2. **Wybór trybu gry**:
    a. Po uruchomieniu gry bez argumentów program zapyta o tryb gry:

        i. **1**: Mała plansza.
        ii. **2**: Średnia plansza.
        iii. **3**: Duża plansza.
        iv. **4**: Niestandardowa plansza (gracz podaje liczbę wierszy, kolumn i bomb).
3. **Sterowanie w grze**:
    a. W grze gracz wprowadza polecenia w formacie:
        i. r x y: Odkryj pole o współrzędnych (x, y).
        ii. f x y: Oznacz pole (x, y) flagą. Jeśli pole jest już oznaczone, usunięcie flagi następuje po ponownym wpisaniu polecenia.
    b. Współrzędne (x, y) są liczbami całkowitymi oznaczającymi odpowiednio wiersz i kolumnę planszy.
4. **Zakończenie gry**:
    a. **Wygrana**:
        i. Po odkryciu wszystkich pól niezawierających min gra wyświetli wynik gracza i zaprosi do wpisania nazwy na listę najlepszych wyników.
    b. **Przegrana**:
        i. Po odkryciu pola z bombą gra wyświetli wszystkie bomby na planszy oraz wynik gracza.
5. **Wyniki**:
    a. Wynik gracza obliczany jest na podstawie liczby odkrytych pól i trybu gry:
        i. Wynik = 10 × liczba odkrytych pól × mnożnik (1 dla łatwego, 2 dla średniego, 3 dla trudnego).
    b. Tablica najlepszych wyników zawiera maksymalnie 5 wpisów i jest zapisywana w pliku scoreboard.txt.

Przykłady interakcji

1. **Rozpoczęcie gry**:

Wybierz tryb gry, podając numer trybu:
1. MALA PLANSZA
2. SREDNIA PLANSZA
3. DUZA PLANSZA
4. NIESTANDARDOWA PLANSZA
Wprowadź tryb w którym chcesz zagrać: 1

2. **Ruch gracza**:

r 3 4

3. **Flagowanie pola**:

f 5 6

4. **Przegrana**:

You lost
Your score: 120

5. **Wygrana**:

You won
Your score: 350

Wskazówki dla gracza

1. Pierwszy ruch jest zawsze bezpieczny – nie trafi na minę.
2. Wykorzystuj flagi, aby oznaczać pola, które na pewno zawierają miny.
3. Analizuj sąsiadujące pola, aby dedukować położenie min.
4. Uważaj na błędy w podawanych współrzędnych – ruchy poza planszą są ignorowane.

Potencjalne problemy

1. **Niepoprawne polecenia**:
    a. Program zignoruje błędne ruchy, takie jak odkrywanie pól poza planszą.
2. **Plik wyników**:
    a. Jeśli plik scoreboard.txt nie istnieje, zostanie utworzony automatycznie.

Szczegóły implementacji programu

Podział projektu na moduły

Projekt składa się z kilku modułów (plików źródłowych), które są odpowiedzialne za różne funkcjonalności programu. Pliki są skompilowane razem w jeden plik wykonywalny saper. Główne moduły programu to:

1. **generacja_planszy.c** - odpowiedzialny za generowanie planszy do gry.
2. **wczytywanie_planszy.c** - zajmuje się wczytywaniem planszy i ewentualnych danych wejściowych z plików.
3. **przebieg_gry.c** - implementuje logikę rozgrywki, w tym obsługę ruchów gracza, flagowanie pól oraz zakończenie gry.

4. **main.c** - główny plik programu, który inicjuje działanie gry, integruje wszystkie moduły i steruje przepływem programu.

Budowa programu za pomocą Makefile

Plik Makefile zawiera instrukcje do kompilacji i zarządzania projektem:

- **Flagi kompilatora (CFLAGS)**:
  - -Wall oraz -Wextra włączają dodatkowe ostrzeżenia kompilatora, co pomaga w identyfikowaniu potencjalnych problemów.
  - -std=c99 ustawia standard języka C na C99.
- **Reguły**:
  - saper: Kompiluje wszystkie moduły w jeden plik wykonywalny o nazwie saper.
  - gdb: Kompiluje program z flagami debugowania (-ggdb), umożliwiając analizę programu za pomocą debuggera GDB.
  - run: Uruchamia skompilowany program.
  - test: Uruchamia program w trybie testowym, w którym domyślnie wykonuje zaplanowane testy gry.
  - clean: Usuwa plik wykonywalny saper.

Sposób połączenia modułów

Każdy moduł odpowiada za oddzielny aspekt działania gry, a ich funkcje są wywoływane z pliku main.c, co zapewnia przejrzystą strukturę i łatwość modyfikacji. Na przykład:

- Funkcje do generowania planszy i rozmieszczania min są zdefiniowane w module generacja_planszy.c, a ich wyniki są używane w main.c.
- Moduł wczytywanie_planszy.c może ładować dane wejściowe lub plansze z plików, co umożliwia obsługę różnych trybów gry.
- Moduł przebieg_gry.c zarządza logiką rozgrywki, np. odsłanianiem pól i obsługą końca gry.

Testowanie programu

Plik Makefile definiuje regułę test, która uruchamia program w trybie testowym. Testy prawdopodobnie są zaimplementowane w kodzie i sprawdzają poprawność działania różnych funkcjonalności gry, takich jak poprawność generacji planszy, flagowanie pól i rozgrywka.

Szczegóły implementacji modułu wczytywanie_planszy.c

Moduł wczytywanie_planszy.c odpowiada za wczytywanie planszy i ruchów gracza z pliku. Jest to kluczowa część programu, umożliwiająca testowanie oraz analizę gry w trybie

odczytu danych wejściowych z wcześniej przygotowanego pliku. Poniżej znajduje się szczegółowy opis funkcji oraz używanych struktur.

Funkcjonalność modułu

1. **Wczytywanie planszy z pliku**:
    a. Moduł prosi użytkownika o nazwę pliku wejściowego, a następnie otwiera plik za pomocą funkcji fopen().
    b. Jeśli plik nie istnieje, użytkownik ma możliwość ponownego wprowadzenia nazwy pliku.
2. **Odczyt danych z pliku**:
    a. Plik wejściowy zawiera w pierwszych trzech liniach:
        i. Liczbę wierszy (row) i kolumn (col) planszy.
        ii. Liczbę bomb (bombs).
    b. Na podstawie tych danych moduł:
        i. Tworzy planszę o podanych wymiarach za pomocą funkcji create_board().
        ii. Ustawia tryb gry (mode) w zależności od liczby wierszy.
3. **Rozmieszczenie bomb**:
    a. Z pliku odczytywane są współrzędne bomb, które są następnie umieszczane na planszy poprzez przypisanie wartości '*' do odpowiednich pól.
4. **Logika rozgrywki**:
    a. Plik wejściowy zawiera również sekwencję ruchów gracza w formacie:
        i. r x y: odkrycie pola o współrzędnych (x, y).
        ii. f x y: ustawienie lub usunięcie flagi na polu (x, y).
    b. Każdy ruch jest analizowany i odpowiednia funkcja (np. reveal_box() lub place_flag()) jest wywoływana.
5. **Zakończenie gry**:
    a. Gra kończy się w dwóch przypadkach:
        i. Trafienie na bombę (game_over = true) – wypisanie liczby poprawnych ruchów oraz wyniku gracza.
        ii. Odkrycie wszystkich pól, które nie zawierają bomb – gra wygrana.
    b. Jeśli plik nie zawiera ruchów kończących grę, wypisuje się wynik aktualnego stanu.

Kluczowe funkcje i struktury

1. **void load_board(box_t **board)**:
    a. Funkcja główna modułu, która:
        i. Wczytuje dane z pliku.
        ii. Tworzy planszę za pomocą funkcji create_board().

iii. Odczytuje ruchy gracza i aktualizuje stan gry.

2. **Struktura box_t**:
    a. Odpowiada za przechowywanie informacji o polu na planszy, np.:
        i. Czy pole zawiera bombę (value == '*').
        ii. Czy pole zostało oflagowane (flagged).
3. **Wywoływane funkcje z innych modułów**:
    a. create_board(int rows, int cols): Tworzy planszę o określonych wymiarach.
    b. set_bomb(int bombs): Rozmieszcza bomby na planszy.
    c. count_bombs_around(int rows, int cols, box_t **board): Liczy bomby wokół każdego pola.
    d. reveal_box(box_t **board, int rows, int cols, int x, int y): Odkrywa wskazane pole.
    e. place_flag(box_t **board, int x, int y, int rows, int cols): Ustawia lub usuwa flagę na danym polu.
    f. get_revealed(): Zwraca liczbę odkrytych pól.

Obsługa błędów

- Sprawdzenie istnienia pliku:
    - Jeśli plik nie istnieje, użytkownik jest proszony o ponowne podanie nazwy pliku.
- Walidacja danych wejściowych:
    - Program zakłada, że dane w pliku są poprawne (brak dodatkowych mechanizmów walidacji).

Szczegóły implementacji modułu generacja_planszy.c

Moduł generacja_planszy.c odpowiada za tworzenie, zarządzanie oraz obsługę planszy gry. Kluczowe funkcjonalności obejmują generowanie planszy, rozmieszczanie bomb, zliczanie bomb wokół pól oraz wypisywanie planszy w terminalu. Poniżej znajduje się szczegółowy opis funkcji oraz ich ról w programie.

Funkcjonalność modułu

1. **Tworzenie planszy**:
    a. Plansza jest reprezentowana jako dynamicznie alokowana dwuwymiarowa tablica wskaźników do struktur box_t. Każda komórka planszy jest strukturą zawierającą informacje o stanie pola (czy jest odkryte, czy zawiera bombę, jaka jest jego wartość).
2. **Zarządzanie pamięcią**:

    a. Funkcja create_board() alokuje pamięć dla całej planszy, a funkcja free_board() zwalnia ją po zakończeniu gry.
3. **Wybór trybu gry**:
    a. Funkcja choose_mode_of_the_game() pozwala użytkownikowi wybrać jeden z trzech standardowych trybów gry (mała, średnia, duża plansza) lub tryb niestandardowy, w którym użytkownik definiuje własne parametry planszy.
4. **Generowanie bomb**:
    a. Funkcja generate_bombs() rozmieszcza bomby na planszy w losowych pozycjach, zapewniając, że nie trafiają na już odkryte pola.
5. **Zliczanie bomb wokół pól**:
    a. Funkcja count_bombs_around() oblicza liczbę bomb w sąsiedztwie każdego pola, co jest kluczowe dla logiki gry.
6. **Wypisywanie planszy w terminalu**:
    a. Funkcja print_board() wyświetla aktualny stan planszy, pokazując odkryte pola, flagi oraz pola nieodkryte.
7. **Obsługa błędów**:
    a. Funkcja check_arguments() weryfikuje poprawność danych wejściowych (rozmiary planszy i liczba bomb) w trybie niestandardowym.

Kluczowe funkcje i ich opis

1. **box_t** create_board(int row, int col)**
    a. Tworzy dynamicznie alokowaną dwuwymiarową tablicę wskaźników do struktur box_t.
    b. Każde pole planszy jest inicjalizowane jako nieodkryte, bez bomby i z wartością '0'.
2. **void free_board(int row, int col, box_t** board)**
    a. Zwalnia pamięć zajmowaną przez planszę, w tym każdą strukturę box_t.
3. **void choose_mode_of_the_game(int* row, int* col, int* num_of_bombs)**
    a. Umożliwia wybór trybu gry.
    b. Tryby standardowe:
        i. Mała plansza (9x9, 10 bomb).
        ii. Średnia plansza (16x16, 40 bomb).
        iii. Duża plansza (16x30, 99 bomb).
    c. Tryb niestandardowy pozwala na dowolne rozmiary i liczbę bomb, z walidacją poprawności danych.
4. **void print_board(int row, int col, box_t** board)**
    a. Wyświetla aktualny stan planszy, ukazując:
        i. Odkryte pola z ich wartością.
        ii. Pola z flagami.
        iii. Nieodkryte pola.
5. **void generate_bombs(int row, int col, box_t** board, int num_of_bombs)**
    a. Rozmieszcza losowo bomby na planszy, upewniając się, że:
        i. Pole nie zostało wcześniej odkryte.

ii. Pole nie zawiera już bomby.

6. **void count_bombs_around(int row, int col, box_t** board)**
    a. Oblicza liczbę bomb w sąsiedztwie każdego pola.
    b. Działa na zasadzie iteracji po sąsiednich polach w promieniu 1 od danego pola.
7. **int check_arguments(int *value)**
    a. Weryfikuje, czy podane przez użytkownika dane (np. liczba wierszy, kolumn, bomb) są liczbami całkowitymi i spełniają minimalne wymagania (np. min. 9 wierszy).

Ważniejsze struktury

- **Struktura box_t**:
    - Reprezentuje pojedyncze pole na planszy.
    - Pola struktury:
        - bool revealed: Czy pole zostało odkryte.
        - bool bomb: Czy pole zawiera bombę.
        - char value: Wartość pola ('0'-'8' dla liczby bomb w sąsiedztwie, '*' dla bomby, 'F' dla flagi).

Obsługa wyjątków i potencjalne usprawnienia

1. **Obsługa błędów**:
    a. Zabezpieczenie przed błędami alokacji pamięci w funkcji create_board().
    b. Walidacja danych wejściowych w trybie niestandardowym.
2. **Potencjalne usprawnienia**:
    a. Ulepszenie wizualizacji planszy w terminalu (np. dodanie współrzędnych).
    b. Obsługa wprowadzenia błędnych danych przy wyborze trybu gry (np. niepoprawne wartości w trybie niestandardowym).

Szczegóły implementacji modułu przebieg_gry.c

Moduł przebieg_gry.c odpowiada za zarządzanie logiką rozgrywki w grze Saper. Obsługuje działania gracza, takie jak odkrywanie pól, flagowanie, przetwarzanie ruchów oraz zarządzanie stanem gry (zwycięstwo, przegrana). Moduł także zapisuje i wyświetla najlepsze wyniki graczy.

Funkcjonalność modułu

1. **Logika gry**:
    a. Zarządzanie odkrywaniem pól (funkcja reveal_box()), z uwzględnieniem mechaniki automatycznego odkrywania pustych pól.
    b. Obsługa flagowania pól (place_flag()), z możliwością odflagowywania.
    c. Odkrywanie wszystkich bomb w przypadku przegranej (reveal_bombs()).
2. **Zarządzanie rozgrywką**:
    a. Funkcja process_arguments() pozwala na przetwarzanie ruchów gracza (odkrywanie lub flagowanie pól).
    b. Mechanizm pierwszego ruchu zapewniający, że gracz nie trafi na bombę (first_move).
3. **Zarządzanie wynikami**:
    a. Funkcje win() i loss() obsługują zakończenie gry, w tym zapis wyniku gracza do tablicy wyników.
    b. Funkcje add_to_scoreboard() i show_scoreboard() zapisują wyniki w pliku tekstowym oraz wyświetlają najlepszych pięciu graczy.

Kluczowe funkcje i ich opis

1. **void reveal_box(box_t** board, int row, int col, int x, int y)**
    a. Odpowiada za odkrywanie pola na planszy.
    b. Jeśli pole zawiera 0, automatycznie odkrywa sąsiadujące pola rekurencyjnie.
    c. Aktualizuje liczbę odkrytych pól (revealed).
2. **void place_flag(box_t** board, int x, int y, int row, int col)**
    a. Oznacza pole jako oflagowane lub usuwa flagę z pola.
    b. Dla oflagowanych pól wartość pola jest ustawiana na 'F'.
3. **void reveal_bombs(box_t** board, int row, int col)**
    a. Odkrywa wszystkie bomby na planszy w przypadku zakończenia gry.
4. **void process_arguments(int row, int col, box_t** board)**
    a. Przetwarza ruchy gracza, takie jak odkrywanie pola (r x y) lub flagowanie (f x y).
    b. Obsługuje pierwsze odkrycie, gwarantując, że nie trafi ono na bombę.
5. **void win(box_t** board, int row, int col)**
    a. Obsługuje zakończenie gry w przypadku zwycięstwa.
    b. Zapisuje wynik gracza do tablicy wyników i wyświetla go.
6. **void loss(box_t** board, int row, int col)**
    a. Obsługuje zakończenie gry w przypadku przegranej.
    b. Wyświetla wynik gracza oraz listę najlepszych wyników.
7. **void add_to_scoreboard(int new_score)**
    a. Dodaje wynik gracza do listy najlepszych wyników przechowywanych w pliku scoreboard.txt.
    b. Wyniki są sortowane malejąco.

8. **void show_scoreboard()**
    a. Wyświetla najlepsze wyniki (do 5 graczy) z pliku scoreboard.txt.

Ważniejsze zmienne globalne

1. **static int revealed**:
    a. Przechowuje liczbę odkrytych pól.
2. **static int mode**:
    a. Określa tryb gry (łatwy, średni, trudny).
3. **static int bomb**:
    a. Przechowuje liczbę bomb na planszy.
4. **static bool first_move**:
    a. Flaga określająca, czy gracz wykonał pierwszy ruch.

Mechanizmy gry

1. **Pierwszy ruch**:
    a. Mechanizm gwarantuje, że pierwszy ruch gracza nigdy nie trafi na bombę.
    b. W momencie pierwszego ruchu bomby są rozmieszczane na planszy, a liczba bomb wokół pól jest obliczana.
2. **System punktacji**:
    a. Punkty są obliczane na podstawie liczby odkrytych pól, trybu gry (mode) i mnożnika punktowego (10x).
3. **Tablica wyników**:
    a. Wyniki są zapisywane w pliku scoreboard.txt.
    b. Nowy wynik jest dodawany do listy wyników w odpowiedniej pozycji, w zależności od jego wartości.

Obsługa wyjątków i potencjalne usprawnienia

1. **Obsługa błędów**:
    a. Brak mechanizmów obsługi błędów przy odczycie pliku scoreboard.txt.
    b. Warto dodać zabezpieczenie przed próbą odczytu lub zapisu na nieistniejącym pliku.
2. **Usprawnienia**:
    a. Dodanie obsługi większej liczby graczy (obecnie tylko 5 wyników jest przechowywanych).
    b. Możliwość ręcznego resetowania tablicy wyników.

Szczegóły implementacji modułu main.c

Moduł main.c jest punktem wejścia do programu, integrującym wszystkie moduły i zarządzającym przepływem gry. Jego główne zadania to inicjalizacja parametrów gry, wybór trybu gry, obsługa testów oraz uruchamianie podstawowych mechanizmów programu.

Funkcjonalność modułu

1. **Inicjalizacja gry**:
    a. Ustawia lokalne ustawienia (np. dla znaków Unicode) za pomocą setlocale(LC_ALL, "").
    b. Używa generatora liczb pseudolosowych (srand(time(NULL))) do losowego rozmieszczania bomb.
2. **Obsługa trybu gry**:
    a. Gdy program jest uruchamiany bez argumentów, umożliwia użytkownikowi wybór trybu gry (np. mała, średnia, duża lub niestandardowa plansza) poprzez wywołanie funkcji choose_mode_of_the_game().
3. **Obsługa testów**:
    a. Gdy program jest uruchamiany z argumentem test, wczytuje planszę oraz ruchy z pliku za pomocą funkcji load_board().
4. **Przepływ gry**:
    a. Tworzy planszę za pomocą create_board() i ustawia liczbę bomb za pomocą set_bomb().
    b. Wyświetla początkowy stan planszy i przekazuje kontrolę do funkcji process_arguments(), która obsługuje ruchy gracza.

Kluczowe funkcje i ich opis

1. **int main(int argc, char** argv)**
    a. Główna funkcja programu, integrująca wszystkie kluczowe moduły.
    b. Parametry:
        i. argc: Liczba argumentów przekazanych do programu.
        ii. argv: Tablica argumentów (np. argv[1] może zawierać "test").
    c. Logika:
        i. Gdy brak argumentów:
            1. Umożliwia wybór trybu gry.
            2. Tworzy planszę i rozpoczyna rozgrywkę.
        ii. Gdy argument test:
            1. Wczytuje planszę oraz ruchy z pliku za pomocą load_board().

Kluczowe wywoływane funkcje z innych modułów

1. **choose_mode_of_the_game()** (moduł generacja_planszy.c):
    a. Umożliwia użytkownikowi wybór trybu gry i ustawia odpowiednie parametry (liczba wierszy, kolumn, bomb).
2. **create_board()** (moduł generacja_planszy.c):
    a. Tworzy dynamiczną planszę gry.
3. **set_bomb()** (moduł przebieg_gry.c):
    a. Ustawia liczbę bomb na planszy.
4. **print_board()** (moduł generacja_planszy.c):
    a. Wyświetla aktualny stan planszy.
5. **process_arguments()** (moduł przebieg_gry.c):
    a. Obsługuje interakcje gracza (odkrywanie pól, flagowanie) i zarządza przebiegiem gry.
6. **load_board()** (moduł wczytywanie_planszy.c):
    a. Wczytuje planszę i ruchy z pliku w trybie testowym.

Obsługa argumentów programu

1. **Bez argumentów (argc < 2)**:
    a. Tryb interaktywny: użytkownik wybiera tryb gry, a gra jest uruchamiana normalnie.
2. **Z argumentem test (argc > 1 i strcmp(argv[1], "test") == 0)**:
    a. Program wczytuje planszę i ruchy z pliku, umożliwiając testowanie działania gry na danych wejściowych.

Potencjalne usprawnienia

1. **Walidacja argumentów**:
    a. Dodanie komunikatu błędu w przypadku niepoprawnych argumentów (np. nieznanego argumentu zamiast test).
2. **Obsługa zaawansowanych trybów**:
    a. Możliwość uruchamiania gry z dodatkowymi argumentami, np. określenie planszy z poziomu wiersza poleceń.
3. **Mechanizm testowania**:
    a. Rozbudowa trybu testowego o bardziej szczegółowe raporty (np. szczegółowe logi ruchów).

Podział pracy w zespole

Projekt został zrealizowany przez nasz dwuosobowy zespół – Asię i Jaśka. Od początku staraliśmy się podzielić obowiązki, jednak podział ten nie był ściśle określony. W praktyce nasze zadania często się przeplatały, ponieważ zależało nam na wspólnym zaangażowaniu w realizację projektu i wzajemnym wsparciu przy trudniejszych elementach.

Zakres obowiązków

Jasiek zrealizował większą część implementacji kodu, skupiając się na:

- Logice gry, w tym obsłudze ruchów gracza, takich jak odkrywanie pól (reveal_box()) oraz flagowanie (place_flag()).
- Mechanizmach związanych z zakończeniem gry (zwycięstwo i przegrana) oraz integracją modułów w pliku main.c.
- Obsłudze systemu wyników, w tym zapisywania najlepszych wyników do pliku i wyświetlania tablicy liderów.

Asia zajęła się implementacją niektórych elementów, takich jak generowanie planszy, rozmieszczanie bomb (generate_bombs()) oraz obliczanie liczby min w sąsiedztwie pól (count_bombs_around()). Była także odpowiedzialna za wczytywanie planszy i ruchów z pliku w trybie testowym. Asia przygotowała sprawozdanie z projektu, dokumentując zarówno proces realizacji, jak i szczegóły kodu.

Współpraca

Choć podział obowiązków nie był precyzyjnie określony, nie przeszkodziło to w efektywnej realizacji projektu dzięki dobrej komunikacji i współpracy. Regularnie omawialiśmy postępy, konsultowaliśmy się przy implementacji bardziej złożonych funkcji i wspólnie testowaliśmy działanie poszczególnych modułów. Wiele problemów udało się rozwiązać dzięki wspólnej analizie kodu.

W pracy nad projektem korzystaliśmy z systemu kontroli wersji git, co pozwoliło nam synchronizować zmiany i uniknąć konfliktów w kodzie. Często pracowaliśmy jednocześnie nad różnymi częściami programu, co wymagało bieżącej wymiany informacji i ustalania priorytetów.

Podsumowanie

Chociaż podział pracy nie był całkowicie jasny, dobra komunikacja i współpraca w zespole pozwoliły nam skutecznie zrealizować projekt. Jasiek przejął większość odpowiedzialności za implementację, podczas gdy Asia zajmowała się generowaniem planszy, testowaniem i dokumentacją. Oba obszary uzupełniały się wzajemnie, a wspólne zaangażowanie przyczyniło się do osiągnięcia zamierzonych celów. Realizacja projektu była dla nas wartościowym doświadczeniem i okazją do rozwijania umiejętności pracy zespołowej.

Podsumowanie projektu Saper

1. Ocena realizacji funkcjonalności

W ramach projektu udało się zaimplementować większość wymaganych funkcjonalności określonych w specyfikacji. Gra działa zgodnie z założeniami, pozwalając na:

- Odkrywanie pól i wyświetlanie liczby min w sąsiedztwie.
- Flagowanie podejrzanych pól w celu ochrony przed przypadkowym odkryciem miny.
- Obsługę różnych poziomów trudności oraz możliwość wyboru niestandardowej planszy.
- Zapisywanie wyników i wyświetlanie najlepszych graczy.
- Możliwość wczytywania planszy i ruchów z pliku w trybie testowym.
- Zapewnienie, że pierwszy ruch gracza nigdy nie trafi na minę.
- Podział kodu na moduły, co ułatwia zarządzanie i rozwój gry.
- Obsługę gry z wiersza poleceń i przechowywanie wyników w pliku scoreboard.txt.

Niektóre funkcjonalności, takie jak bardziej zaawansowana walidacja wejścia użytkownika czy bardziej rozbudowana obsługa błędów, mogłyby zostać ulepszone w przyszłości.

2. Napotkane trudności i sposoby ich rozwiązania

Podczas realizacji projektu napotkaliśmy kilka trudności technicznych i organizacyjnych, które udało nam się przezwyciężyć.

a) Zarządzanie dynamiczną pamięcią

- Początkowo mieliśmy problem z alokacją pamięci dla tablicy dwuwymiarowej wskaźników do struktur box_t.
- Rozwiązanie: Dokładne przeanalizowanie procesu alokacji i zwalniania pamięci oraz testowanie kodu w poszukiwaniu wycieków pamięci. Użycie valgrind do sprawdzenia poprawności zarządzania pamięcią.

b) Rekurencyjne odkrywanie pól

- Podczas implementacji funkcji reveal_box() natknęliśmy się na problem nieskończonej rekurencji, gdy gra próbowała odkrywać sąsiednie pola dla pól o wartości 0.
- Rozwiązanie: Wprowadziliśmy warunki zapobiegające ponownemu odkrywaniu tych samych pól.

c) Obsługa wejścia użytkownika

- Podczas wczytywania danych użytkownika niepoprawne wejście powodowało błędy.
- Rozwiązanie: Dodanie walidacji poprawności wprowadzonych danych (np. sprawdzanie, czy wartości są liczbami oraz czy mieszczą się w zakresie planszy).

d) Obsługa pliku wyników (scoreboard.txt)

- Pierwotna implementacja powodowała duplikaty wyników w pliku oraz błędne sortowanie wyników.
- Rozwiązanie: Wprowadziliśmy mechanizm, który dodaje wyniki do listy w odpowiednim miejscu, a następnie zapisuje nową wersję pliku, zastępując starą.

e) Synchronizacja pracy w zespole

- Kod był tworzony przez dwie osoby, co wymagało podziału pracy i synchronizacji.
- Rozwiązanie: Ustaliliśmy podział modułów i regularnie wymienialiśmy się informacjami na temat postępów. Korzystaliśmy z systemu kontroli wersji git, co ułatwiło współpracę i integrację zmian.

3. Wnioski

1. **Modularność kodu ułatwia rozwój i modyfikację**
    a. Podział na moduły (generacja_planszy.c, przebieg_gry.c, wczytywanie_planszy.c, main.c) sprawił, że kod był bardziej przejrzysty i łatwiejszy do debugowania.
2. **Dobre zarządzanie pamięcią jest kluczowe**
    a. Użycie dynamicznych struktur danych wymaga staranności w alokacji i zwalnianiu pamięci. Narzędzia takie jak valgrind pomogły wykryć błędy.
3. **Praca zespołowa wymaga dobrej komunikacji**
    a. Regularne spotkania i dzielenie się postępami pomogły w skutecznym ukończeniu projektu.
4. **Obsługa błędów i walidacja wejścia użytkownika są bardzo ważne**
    a. Początkowo nie uwzględniliśmy wszystkich możliwych błędnych wejść użytkownika, co prowadziło do nieoczekiwanych błędów.
5. **Dalsze usprawnienia są możliwe**
    a. Można dodać bardziej rozbudowane menu, obsługę zapisu i wczytywania gry oraz lepsze formatowanie planszy.

Podsumowując, projekt zakończył się sukcesem, a wszystkie kluczowe funkcjonalności udało się wdrożyć. Było to wartościowe doświadczenie, które pozwoliło nam lepiej zrozumieć programowanie w języku C oraz wyzwania związane z implementacją bardziej złożonych aplikacji.

Wnioski ogólne

Realizacja projektu była dla nas cennym doświadczeniem, zarówno pod względem technicznym, jak i pracy zespołowej. Udało nam się zaimplementować wszystkie kluczowe funkcjonalności, a dobra komunikacja w zespole pozwoliła nam sprawnie rozwiązywać napotkane problemy.

Podczas pracy nad projektem szczególnie doceniliśmy znaczenie modularnej struktury kodu oraz systemu kontroli wersji git, które ułatwiły organizację i współpracę. Mimo że podział pracy nie był precyzyjnie określony, dzięki regularnym konsultacjom i wspólnemu testowaniu udało nam się efektywnie ukończyć projekt.

W przyszłości moglibyśmy jeszcze bardziej skupić się na walidacji danych wejściowych oraz optymalizacji niektórych fragmentów kodu. Ogólnie rzecz biorąc, projekt przebiegł pomyślnie i dostarczył nam dużo satysfakcji oraz nowych umiejętności.