Krzysztof Dąbrowski i Jakub Bogusz

Specyfikacja implementacyjna – Gra w życie

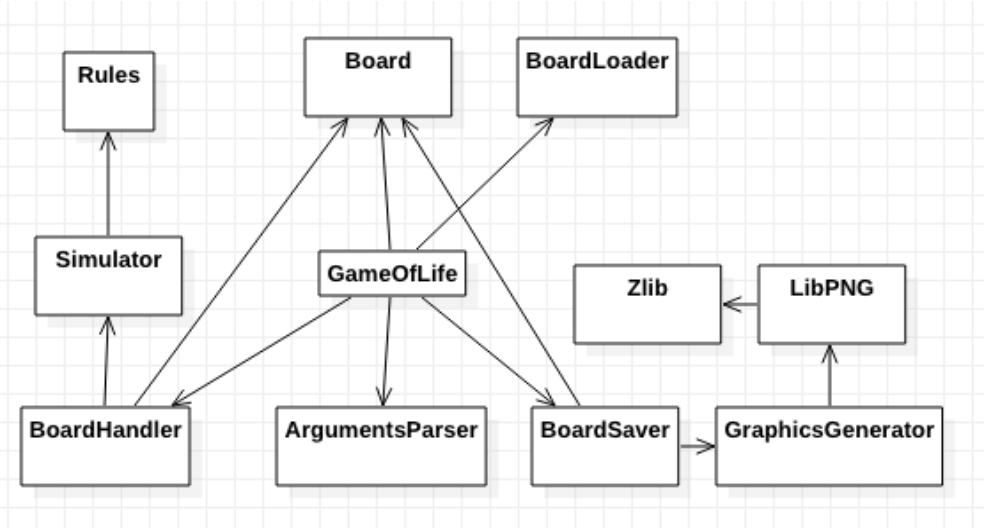
23 marca 2019

Spis treści

1. Opis przepływu sterowania

Lista kolejnych czynności wykonywanych przez program:

1. Wczytanie argumentów wywołania programu
    1.1 Pobranie argumentów
    1.2 Przekonwertowanie argumentów do struktury ustawień



2. Generacja planszy początkowej
    a) na podstawie podanego pliku
        2.a.1 Wczytanie danych z pliku
        2.b.1 Przekonwertowanie wczytanych danych do struktury planszy
    b) losowe wygenerowanie planszy
3. Uruchomienie symulacji z odpowiednimi ustawieniami
    3.1 Generacja kolejnych pokoleń
    3.2 Ewentualny zapis stanu planszy do pliku tekstowego
    3.3 Ewentualna konwersja wygenerowanych plików tekstowych na pliki odpowiednie pliki graficzne
4. Wyczyszczenie zaalokowanej pamięci

2. Algorytm

Opis algorytmu służącego do generacji następnego pokolenia:

Iteracja po planszy:
1. Wycięcie fragmentu planszy *(w razie wyjścia poza tablicę planszy, uzupełnienie odpowiednich krawędzi obszaru martwymi komórkami)*,
2. Zliczenie "żywych" komórek w tym obszarze z pominięciem elementu środkowego tego fragmentu,
3. Ustalenie stanu elementu środkowego fragmentu,
4. Zapisanie ustalonego stanu do planszy następnego pokolenia.

3. Podział na moduły

Program będzie podzielony na współdziałające moduły. Pozwoli to na łatwiejszą modyfikację programu oraz dodawanie nowych funkcjonalności.

3.1. GameOfLife

Główny moduł kontrolujący przepływ sterowania i danych między pozostałymi modułami.

3.1.1. Funkcje

`int main(int argc, char** args)` – Punkt startowy programu. Z niej wywoływane będą kolejne funkcje. Przyjmować będzie 2 argumenty – argumenty wsadowe programu:

`int argc` – liczba argumentów,
`char** args` – tablica napisów – faktycznych argumentów wywołania programu.

3.2. ArgumentsParser

Moduł odpowiadający za interpretacje podanych wsadowo argumentów programu, konwersji ich oraz zapisu do utworzonej w tym celu struktury.

3.2.1. Struktury

Typ wyliczeniowy zawierający dostępne stany zapisu.

```
typedef enum{
   GIF,
   PNG,
   TXT
}Type;
```

Struktura zawierająca ustawienia symulacji:

```
typedef struct{
   int help;
   char* file;
   char* output_dest;
   Type type;
   int number_of_generations;
   int step;
   int delayMs;
}Config;
```

`help` – informuje o tym czy ma być wyświetlana pomoc,
`file` – informuje o tym jaka jest ścieżka do pliku wejściowego,
`output_dest` – informuje o tym jaka jest ścieżka dla pliku/plików wyjściowych,
`type` – informuje o tym jaki jest typ zwracanych wyników,
`number_o_generations` – informuje o tym ile pokoleń komórek zostanie wygenerowane,
`step` – informuje o tym co ile pokoleń zapisywane będą dane wyjściowe,
`delay` – informuje w jakich odstępach czasu mają się wyświetlać kolejne generacje komórek.

3.2.2. Funkcje

`Config* parseArgs(int argc, char** argv)` – będzie przetwarzać argumenty wsadowe programu podane w postaci flag na strukturę zawierającą ustawienia symulacji. Przyjmować będzie 2 argumenty – argumenty wsadowe programu:

`int argc` – liczba argumentów,
`char** args` – tablica napisów, będących faktycznymi argumentami wywołania programu.

Zwracać będzie strukturę zawierająca ustawienia symulacji.

3.3. BoardSaver

Moduł odpowiadający za obsługę zapisu wyników działania programu w zależności od argumentów.

3.3.1. Funkcje

`void save(Board* b, Config* c)` – będzie uruchamiać konkretne funkcję zapisujące w zależności od podanych flag. Jako argument będzie przyjmować wskaźnik na strukturę zawierającą opis planszy:

`Board* b` – wskaźnik na strukturę zawierającą opis planszy.

`void saveTxt(Board* b, Config* c)` – Będzie zapisywać dany stan planszy w pliku tekstowym, mogącym w przyszłości służyć za plik wejściowy. Jako argument będzie przyjmować wskaźnik na strukturę zawierającą opis planszy:

`Board* b` – wskaźnik na strukturę zawierającą opis planszy.

3.4. GraphicsGenerator

Moduł odpowiadający za obsługę zapisu wyników działania programu w zależności od argumentów.

3.4.1. Funkcje

`void savePng(char* data, Config* c)` – będzie generować plik .png będący reprezentacją planszy:

`char* data` – dane opisujące planszę, potrzebne do generacji pliku .png,
`Config* c` – struktura zawierająca parametry pliku wyjściowego.

`void saveGif(char* data, Config* c)` – będzie generować plik .gif będący reprezentacją planszy:

`char* data` – dane opisujące planszę, potrzebne do generacji pliku .gif,
`Config* c` – struktura zawierająca parametry pliku wyjściowego.

3.5. BoardHandler

Moduł definiujący operacje wykonywane na strukturze `Board`.

3.5.1. Funkcje

`Board* createRandomBoard(int x, int y)` – Tworzy planszę o podanej wielkości z komórkami o losowych stanach.

`int x` – szerokość planszy
`int y` – wysokość planszy

Zwracany jest wskaźnik na nowy obiekt `Board` utworzony na stercie.

`void disposeBoard(Board *Board)` – Zwalnia dynamicznie alokowane pola struktury `Board`. Powinna zostać wywołana przed zwolnieniem dowolnej zmiennej typu `Board`. Po zastosowaniu tej metody zmienna wskazana przez argument `Board` nie nadaje się już do użytku.

`Board *Board` – Adres planszy, która będzie zwalniana

`char* boardToString(Board *Board)` – Generuje ciąg znaków reprezentujący stan wskazanej planszy.

`Board *Board` – Adres planszy, której reprezentacja ma zostać wygenerowana

Zwrócony napis należy po wykorzystaniu zwolnić.

`char* serializeBoard(Board *Board)` – Generuje ciąg znaków reprezentujący wskazaną strukturę `Board`.

`Board *Board` – Adres planszy, której reprezentacja ma zostać wygenerowana

Zwrócony napis zawiera wszystkie niezbędne informacje do rekonstrukcji struktury na jego podstawie. Zwrócony napis należy po wykorzystaniu zwolnić.

3.6. Board

Moduł definiujący strukturę planszy.

3.6.1. Struktury

Typ wyliczeniowy zawierający możliwe stany pojedynczej komórki.

```
typedef enum{
   DEAD = 0,
   ALIVE = 1
}CellState;
```

`DEAD` – Komórka jest martwa
`ALIVE` – Komórka jest żywa

Struktura przechowująca stan pojedynczego pokolenia.

```
typedef struct{
   int sizeX;
   int sizeY;
   CellState *cells;
}Board;
```

`sizeX` – Szerokość planszy
`sizeY` – Wysokość planszy
`cells` – Tablica zawierająca stany wszystkich komórek, wypisane wiersz po wierszu w tablicy 1-wymiarowej - adres komórki o współrzędnych (x,y): y * sizeX + x.

3.7. Simulator

Moduł odpowiadający za przeprowadzenie właściwej symulacji zgodnie z regułami gry w życie.

3.7.1. Funkcje

`Board* simulate(Board* b, Config* p)` – będzie przeprowadzać symulacje
całej gry w życie:

`Board* b` – wskaźnik na strukturę zawierającą stan planszy,
`Config* p` – struktura zawierająca ustawienia symulacji.

Zwracać będzie strukturę zawierająca końcowy stan planszy.

`Board* nextGen(Board* b)` – będzie generować planszę odpowiadającą następnemu pokoleniu komórek. Jako argument będzie przyjmować:

`Board* b` – wskaźnik na strukturę zawierającą stan planszy,

Zwracać będzie wskaźnik na strukturę zawierająca stan planszy w następnym pokoleniu.

`CellState* getArea(Board* b, int x, int y, int size)` – będzie wycinać fragment planszy na potrzeby określenia stanu środkowej komórki. Funkcja będzie przyjmować argumenty:

`Board* b` – wskaźnik na strukturę zawierającą opis planszy, z której zostanie pobrany wycinek,
`int x, int y` – współrzędne środka wycinka,
`int size` – rozmiar wycinanego obszaru (długość boku wycinanego kwadratu), liczba nieparzysta.

Zwracać będzie tablicę stanów komórek na danym obszarze.

3.8. Rules

Moduł odpowiedzialny za ustalenie stanu komórki na podstawie jej otoczenia.

3.8.1. Funkcje

`CellState nextState(CellState* area)` – będzie określać następny stan komórki na środku badanego obszaru:

`CellState* area` – tablica stanów komórek reprezentująca badany obszar.

Zwracać będzie stan komórki na środku obszaru.

3.9. Loader

Moduł odpowiedzialny za wczytanie planszy początkowej z pliku tekstowego i zapisanie jej do struktury.

3.9.1. Funkcje

`Board* load(char* path)` – będzie przetwarzać plik wejściowy i zapisywać go do struktury. Jako argument będzie przyjmować:

`char* path` – ścieżka do pliku wejściowego,

Zwracać będzie wskaźnik na strukturę opisującą początkowy stan planszy.

`int* getSize(char* path)` – będzie wczytywać rozmiar planszy zapisanej w pliku wejściowym:

`char* path` – ścieżka do pliku wejściowego,

Zwracać będzie dwuelementowy wektor zawierający rozmiary planszy początkowej.

4. Kompilacja

W celu automatyzacji procesu kompilacji wykorzystane zostanie narzędzie *makefile*. Umożliwi ono przygotowanie scenariuszy kompilacji na różne środowiska.

4.1. Środowisko testowe

Środowisko to ma na celu optymalizację prędkości kompilacji i umieszczenie symboli pozwalających na sprawne debugowanie kodu w plikach wynikowych. Przy kompilacji testowej zostanie zdefiniowana stała `DEBUG`. Dzięki temu wykorzystując mechanizm kompilacji warunkowej możliwe będzie wyświetlenie dodatkowych informacji o działaniu programu w trybie testowym.

Lista flag:

— `-O0` – Optymalizacja czasu kompilacji
— `-std=c11` – Ustawienie standardu języka C
— `-Wall` – Wypisywanie wszystkich ostrzeżeń
— `-g3` – Umieszczenie pełnych symboli do debugowania
— `-D DEBUG` – Ustawienie stałej preprocesora o nazwie `DEBUG`

4.2. Środowisko produkcyjne

Środowisko to ma na celu optymalizację zasobów wykorzystywanych przez program w trakcie działania.

Lista flag:

— `-Ofast` – Optymalizacja zasobów kosztem czasu kompilacji
— `-std=c11` – Ustawienie standardu języka C
— `-g0` – Nie umieszczenie symboli do debugowania

5. Testy

Poprawność działania funkcji zawartych w modułach programu będzie testowana przy pomocy testów jednostkowych. W celu organizacji oraz uspójnienia testów zostanie wykorzystany framework **CUnit**.

5.1. Stuktóra testów

Testy zostaną podzielone na zestawy. Każdy zestaw będzie grupował testy dotyczące danej części projektu (najczęściej modułu, ale koniecznie). Dzięki podziałowi testów łatwiej będzie zidentyfikować kod wywołujący problemy.

5.2. Planowane testy

5.2.1. Test zamiany planszy na ciąg znaków

Test ten będzie polegał na ręcznym utworzeniu struktury planszy o konkretym stanie komórek. Następnie zostanie utworzona reprezentacja tekstowa planszy przy pomocy funkcji
`char* boardToString(Board *Board)`. Powstały w ten sposób tekst zostanie porównany z ręcznie wpisaną spodziewaną wartością.

Przy danym obiekcie Board board o wartościach sizeX = 3, sizeY = 2, cells = 1, 1, 0, 0, 0, 1

Po wywołaniu boardToString(&board)

Zostanie zwrócony napis `"1 1 0\n0 0 1\n"`

5.2.2. Test parsowania planszy

Ten test będzie polegał na utworzeniu tymczasowego pliku zawierającego tekst w formacie pliku przechowującego stan planszy. Następnie zostanie wywołana funkcja `Board* load(char* path)`. Kolejnym krokiem będzie sprawdzenie czy szerokość, długość i wartości komórek wczytanej planszy są takie jak spodziewane. Jeśli te warunki zostaną spełnione wynik testu będzie pozytywny. Pod koniec testu usunięty zostanie plik tymczasowy.

Przy danym pliku dane.txt
3 3
1 1 0
1 0 1

0 1 1

Po wywołaniu load(”dane.txt”)

Zostanie zwrócony wskaźnik na obiekt typu Board o wartościach sizeX = 3, sizeY = 3, cells = 1, 1, 0, 1, 0, 1, 0, 1, 1

5.2.3. Test reguł

Ten test będzie weryfikował czy zasady generacji komórek następnego pokolenia przetwarzają poprawnie otrzymany obszar. W jego trakcie do modułu zasad zostanie przekazany obszar hipotetycznej planszy. Wynik realizacji reguł generacji następnego pokolenia na danym obszarze zostanie porównany ze spodziewanym obszarem. Jeśli wszystkie komórki będą miały identyczne stany test zakończy się pozytywnie.

Przy danym obszarze CellState* area = 1, 1, 0, 1, 0, 1, 0, 1, 0

Po wywołaniu nextState(area)

Zostanie zwrócony stan DEAD

5.2.4. Test symulacji pokolenia

Ten test ma za zadanie zweryfikować proces wytworzenia całego nowego pokolenia na podstawie poprzedniego. W tym celu zostanie utworzona plansza o znanym stanie komórek. Będzie ona przekazana funkcji `Board* nextGen(Board* b)`. Nowa plansza powstała w wyniku działania funkcji symulującej zostanie porównana ze spodziewanym stanem planszy. Jeśli jakakolwiek z komórek będzie różna między wynikiem symulacji a wartością spodziewaną test zakończy się niepowodzeniem.

Przy danej planszy Board board z wartościami sizeX = 2 sizeY = 2 CellState* area = 1, 1, 0, 1,

Po wywołaniu simulate(&board)

Zostanie zwrócony wskaźnik na nowy obiekt typu Board z wartościami sizeX = 2 sizeY = 2 CellState* area = 1, 1, 1, 1,

5.2.5. Test wczytania argumentów wsadowych

Ten test będzie polegał na utworzeniu napisu reprezentującego flagi wprowadzone przez użytkownika oraz przekazaniu go do funkcji `Config* parseArgs(int argc, char** argv)`. Zwrócona struktura ustawień zostanie porównana pole po polu ze spodziewanym wynikiem. Jeśli wszystkie pola spodziewanych ustawień będą takie same jak tych powstały w wyniku parsowania flag programu test zakończy się powodzeniem.

Przy danym ciagu argumentów argv = ”-f”, ”zapis.txt”, ”-o”, ”wyniki”, ”–number_of_generations”, ”8” oraz argc = 6

Po wywołaniu parseArgs(argc, argv)

Zostanie zwrócony wskaźnik na strukturę typu Config o wartościach: help = 0, file = ”zapis.txt”, output_dest = ”wyniki”, type = GIF, number_of_generations = 6, step = 1, delayMs = 0.

6. Biblioteki zewnętrzne

Program będzie korzystał z bibliotek nie będących częścią standardu języka C. Umożliw to dostarczenie szerszej funkcjonalności.

6.1. zlib.h

Biblioteka zawierająca darmową implementację algorytmów kompresji i dekompresji *deflate*. Jest ona wykorzystywana wewnętrznie przez bibliotekę `libpng.h`.

6.2. libpng.h

Biblioteka ta umożliwia tworzenie oraz odczytywanie plików graficznych takich jak *png* czy *gif*.

6.3. cunit.h

Jest to framework służący do tworzenia i uruchamiania testów jednostkowych. Zostanie wykorzystany do uspójnienia procesu testowania modłów programu.