Gliwice, 07.06.2019

**Laboratorium**

**Programowania Komputerów**

Temat:

Onslaught

Autor: Patryk Frączyński

Informatyka, semestr 4, grupa 6

Prowadzący: Dr inż. Jolanta Kawulok

# Temat

Onslaught – gra typu „defense your base” z użyciem biblioteki SFML. Gra polega na budowaniu i ulepszaniu wieżyczek, które będą bronić bazę przed nadchodzącymi wrogami.

# Analiza tematu

## Struktura danych

W programie została wykorzystana lista, ponieważ część danych jest zawsze wyłącznie liniowo przeszukiwana a wstawianie elementów odbywa się tylko na koniec. Wykorzystany został także wektor z powodu jego krótkiego czasu dostępu do środkowych elementów oraz możliwości łatwego dodawania nowych elementów w odpowiednie miejsca. Zostało utworzone także wiele klas niezbędnych do poprawnego działania programu.

## Biblioteki

Wykorzystano bibliotekę graficzną SFML z powodu jej prostoty, szybkości działania oraz pełnej wystarczalności do utworzenia tej gry. W celu wstrzymywania pracy programu na pewien czas skorzystano z bibliotek chrono oraz thread, a w celu zapisu i odczytu stanu gry skorzystano z biblioteki fstream. Wykorzystano także biblioteki kontenerów STL takie jak vector oraz list.

## Algorytmy

Do wykonania programu została użyta duża liczba algorytmów, które były projektowane głównie dla osiągnięcia jak najlepszej złożoności obliczeniowej, ponieważ jest to prosta gra, więc powinna bardzo mało obciążać procesor. Złożoność pamięciowa była w małym stopniu brana pod uwagę z powodu małej ilości grafik.

# Specyfikacja zewnętrzna

## Instrukcja dla użytkownika

Po uruchomieniu gry pojawia się menu, w którym należy wybrać preferowaną mapę. Po dokonaniu wyboru rozpoczyna się gra, potwory zaczynają wychodzić z jednego punktu przy krawędzi mapy i zadaniem gracza jest umiejętne kupowanie i budowanie wieżyczek, w sklepie znajdującym się na dole ekranu, by nie dopuścić potworów do bazy. Nie wolno budować wieżyczek na drodze ani bazie, a potwory przemieszczają się zawsze zaznaczoną drogą. Co 20 sekund zaczyna wychodzić nowa grupa dziesięciu potworów, które z każdym poziomem są silniejsze, ale także zabijanie ich przynosi coraz większe zyski. Każda wieżyczka posiada swój własny sposób zadawania obrażeń oraz własne ceny i wartości ulepszeń. Można przyśpieszyć pojawianie się nowych potworów poprzez kliknięcie zegara znajdującego się w prawym dolnym rogu. Przyciskiem ESC można zatrzymać grę poprzez pojawienie się menu pauzy, w którym można wczytać zapisaną grę na danej mapie, zapisać aktualny stan gry, kontynuować rozgrywkę, rozpocząć grę od nowa oraz wyłączyć aplikację.

# Specyfikacja wewnętrzna

## Klasy

### Znaczenie obiektu

### Powiązania z innymi klasami

### Istotne pola i metody

## Diagram hierarchii klas

## Istotne struktury danych i algorytmy

## Techniki obiektowe

## Ogólny schemat działania programu

# Testowanie i uruchamianie

Podczas prac nad projektem czasem pojawiały się małe problemy, co jest rzeczą naturalną, ale największy problem był z płynnymi animacjami kolizji, gdyż SFML nie posiada zbyt rozbudowanych możliwości w tym zakresie. Wraz z kolejnymi zmianami kodu gra była rozbudowywana, ulepszana oraz testowana. Po zakończeniu prac gra była testowana przez kilka osób, które podpowiedziały parę zmian, które zostały wprowadzone. Nie ma wycieków pamięci, a w przypadku braku któregokolwiek z plików aplikacja odpowiedni reaguje.

# Uwagi i wnioski

Stworzenie tej gry okazało się bardziej pracochłonne niż początkowo sądziłem, ale z drugiej strony jest bardzo dużo sytuacji, które należy przewidzieć i zaimplementować sposób postępowania podczas nich, oraz zaimplementować wszystkie niezbędne funkcjonalności gry, a jest ich dużo, by gra była przyjemna. Myślę, że korzystając z jakiegoś silnika graficznego mniej czasu zajęłaby mi praca nad tą grą, ale celowo wybrałem SFML, żeby mieć dużo styczności z kodem a nie z gotowym środowiskiem, który ma już dużo gotowych rzeczy zaimplementować i wystarczy je włączyć. Praca nad tym projektem dużo pokazała mi w związku z projektowaniem klas, co na pewno mi się w przyszłości przyda.

# Specyfikacja wewnętrzna

1. **4.1** **Zmienne** 
   1. W programie nie korzystano ze zmiennych globalnych oraz zrezygnowano również ze stałych kompilacji. W programie występują następujące zmienne:

|  |  |  |  |
| --- | --- | --- | --- |
| 1. typ | 1. nazwa | 1. znaczenie | 1. zakres/wartości |
| 1. BITMAP \* | 1. bufor | 1. wskaźniki do bitmap zawierających grafiki | 1. na początku: wartość NULL; następnie adresy bitmap w pamięci |
| 1. plansza |
| 1. pb |
| 1. pcz |
| 1. db |
| 1. dcz |
| 1. wyjscie |
| 1. koniec |
| 1. char [8][8] | 1. tab | 1. tablica przechowywująca informacje o rozmieszczeniu pionków | 1. na początku: rozmieszczenie pionków zgodnie z zasadami (‘0’ – pole ciemne puste, ‘a’ – pole jasne, 0 – pole z pionkiem białym, 1 – pole z pionkiem czarnym); następnie: zależnie od rozwoju rozgrywki (dodatkowo: 2 – pole z damką białą, 3 – pole z damką czarną) |
| 1. int | 1. i, j, x | 1. liczniki pętli | 1. 0…8 |
| 1. double | 1. x | 1. licznik pętli, zmienna wykorzystywana do animacji zakończenia gry | 1. 1 - 350 |
| 1. char | 1. ruch | 1. zmienna określająca kolor pionków wykonujących aktualny ruch | 1. 0 – białe, 1 – czarne |
| 1. int | 1. ilosc\_pionkow\_ 2. danego\_koloru | 1. zmienna przechowywująca ilość pozostałych pionków na planszy danego koloru | 1. 0…12 |
| 1. int | 1. ilosc\_ruchow\_ 2. bez\_bicia | 1. zmienna przechowywująca ilość wykonanych ruchów pod rząd, podczas których nie wykonano żadnego bicia | 1. 0…15 |
| 1. int/int \* | 1. my | 1. numer rzędu na planszy klikniętego pionka | 1. 0…7 |
| 1. int/int \* | 1. mx | 1. numer kolumny na planszy klikniętego pionka | 1. 0…7 |
| 1. int/int \* | 1. tmp\_my | 1. numer rzędu na planszy klikniętego pustego pola | 1. 0…7 |
| 1. int/int \* | 1. tmp\_my | 1. numer kolumny na planszy klikniętego pustego pola | 1. 0…7 |
| 1. char [2] | 1. zbite | 1. ilość zbitych pionków (adres: 0-pionki białe, 1-pionki czarne) | 1. 0…12 |
| 1. int | 1. rozmiar | 1. rozmiar w pikselach bitmapy przechowywującej grafiki pionków | 1. 96 |
| 1. int | 1. szerokosc | 1. szerokość okna w pikselach | 1. 1000 |
| 1. int | 1. wysokosc | 1. wysokość okna w pikselach | 1. 768 |
| 1. char \* | 1. elem | 1. zmienna przechowywująca zawartość konkretnej komórki tablicy tab | 1. ‘0’ – pole ciemne puste, ‘a’ – pole jasne, 0 – pole z pionkiem białym, 1 – pole z pionkiem czarnym, 2 – pole z damką białą, 3 – pole z damką czarną |
| 1. int/int \* | 1. zbijane | 1. ilość przeskoczonych pionków podczas bicia damką | 1. 1…5 |
| 1. int | 1. a, b | 1. zmienne wykorzystywane do obliczeń w funkcjach |  |
| 1. int/int \* | 1. powodzenie | 1. zmienna przechowywująca informację o zbitych pionkach podczas bicia damką | 1. 0 – nie zbito pionka, 1 – zbito pionek |

1. **4.2 Funkcje**

|  |
| --- |
| 1. void wyswietl(BITMAP \*\* bufor, BITMAP \*\* plansza, BITMAP \*\* pb, BITMAP \*\* pcz, BITMAP \*\* db, BITMAP \*\* dcz, BITMAP \*\* wyjscie, char tab[8][8], char ruch) |
| 1. Procedura odpowiedzialna za wyświetlanie stanu planszy na ekran, korzystając z bitmap, przekazywanych jako argumenty funkcji, oraz tablicy zawierającej aktualne rozmieszczenie pionków i zmiennej zawierającej informację o kolorze pionków, które aktualnie wykonują ruch. Funkcja umieszcza wszystko na buforze, a następnie gotowy bufor wyświetla na ekran. |

|  |
| --- |
| 1. int czy\_wyjscie(int tmp\_my, int tmp\_mx) |
| 1. Funkcja sprawdzająca, czy podczas kliknięcia myszą wskaźnik znajduje się na przycisku wyjście. |

|  |
| --- |
| 1. void pobierz\_wspolrzedne(int \*tmp\_mx, int \*tmp\_my) |
| 1. Procedura upewnia się, że przycisk myszy nie jest wciśnięty (jeśli jest to czeka aż zostanie zwolniony), a następnie czeka aż użytkownik kliknie LPM i zapisuje wtedy położenie kursora do zmiennych przekazywanych jako argumenty. Wykorzystana została w niej funkcja rest, w celu spowolnienia wykonywania pętli pętli, dla optymalizacji obliczeniowej. |

|  |
| --- |
| 1. void pobierz\_polozenie(int \*tmp\_mx, int \*tmp\_my, int rozmiar) |
| 1. Procedura pobiera położenie wskaźnika myszy i przerabia je na adres pola na planszy, nad którym znajduje się wskaźnik. |

|  |
| --- |
| 1. void kapowe(char \*tab, int \*i, int \*j, char ruch, char zbite[2]) |
| 1. Procedura usuwająca pionki, które powinny zostać wykorzystane do bicia |

|  |
| --- |
| 1. int ilosc\_pionkow\_danego\_koloru(char tab[8][8], char ruch) |
| 1. Funkcja zliczająca ile pionków danego koloru pozostało jeszcze na planszy. |

|  |
| --- |
| 1. int czy\_przeskoczony\_pionek(char \*elem, char ruch, int \*zbijane) |
| 1. Funkcja sprawdzająca, czy został przeskoczony poprawny pionek. W przypadku przeskoczenia złego pionka funkcja przerywa bicie. |

|  |
| --- |
| 1. int ruchf(char tab[8][8], int \*my, int \*mx, int \*tmp\_my, int \*tmp\_mx, char ruch, char zbite[2]) |
| 1. Funkcja odpowiedzialna za wykonywanie ruchów bez bić. Najpierw sprawdza, czy wykonywany ruch jest poprawny (jeśli nie to kończy swoje działanie i działanie programu wraca do funkcji głównej). Następnie funkcja sprawdza, czy w przypadku ruchu damką nie został przeskoczony żaden pionek. W kolejnym kroku funkcja sprawdza, czy nie zostało niezauważone bicie. Ostatnią czynnością tej procedury jest przestawienie pionka w nowe miejsce. |

|  |
| --- |
| 1. int czy\_przeskoczony\_pionek(char \*elem, char ruch, int \*zbijane) |
| 1. Funkcja sprawdzająca, czy został przeskoczony pionek podczas ruchu damką. Jeśli przeskoczono pionek przeciwnego koloru to funkcja inkrementuje wartość wskazywaną przez wskaźnik zbijane, a jeśli przeskoczono własny pionek funkcja zeruje wartość tego wskaźnika i w tym przypadku funkcja zwraca wartość 0, a w pozostałych przypadkach wartość 1. |

|  |
| --- |
| 1. void zbijanie\_damka(char tab[8][8], int \*tmp\_my, int \*tmp\_mx, int \*my, int \*mx, int a, int b, char ruch, char zbite[2], int \*powodzenie) |
| 1. Procedura zostaje wywołana, gdy podczas bicia damką został przeskoczony tylko jeden poprawny pionek i przestawia damkę w nowe miejsce oraz usuwa zbity pionek z tablicy. |

|  |
| --- |
| 1. int bicie(char tab[8][8], int \*my, int \*mx, int \*tmp\_my, int \*tmp\_mx, char ruch, char zbite[2]) |
| 1. Funkcja odpowiadająca za bicie pionków. W przypadku bicia zwykłym pionkiem (lub damką, lecz tak jak zwykłym pionkiem) funkcja najpierw sprawdza czy prawidłowe jest do bicie i jeśli tak, to przestawia pionki oraz sprawdza, czy można dalej zbijać (w tym przypadku funkcja czeka, aż użytkownik dokończy zbijanie). Jeśli pierwsze sprawdzanie da negatywny wynik, to funkcja sprawdza, czy wykonywane jest bicie damką, wykorzystując dwie poprzednie funkcje. Jeśli nie było możliwości bicia, funkcja kończy swoje działanie i program wraca do funkcji głównej, a w przeciwnym przypadku, funkcja sprawdza, czy istnieje możliwość dalszego bicia (uzyskując pozytywny wynik funkcja pobiera od użytkownika adres kolejnego pola, na który można przestawić pionek w celu wykonania bicia. Gdy wykonano bicie funkcja zwraca wartość 1, w przeciwnym przypadku wartość 0. |

|  |
| --- |
| 1. void koniecf(BITMAP \*\* bufor, BITMAP \*\* koniec) |
| 1. Funkcja wyświetlająca końcowy napis po zakończeniu gry, wykorzystując funkcję cosinus do animacji napisu. |

|  |
| --- |
| 1. int czy\_mozliwy\_ruch(char tab[8][8], char ruch) |
| 1. Funkcja odpowiedzialna za sprawdzenie, czy dany kolor pionków posiada możliwość ruchu. Jeśli jej nie posiada to następuje koniec gry. |

|  |
| --- |
| 1. int main(void) |
| 1. Główna funkcja programu nadzorująca kolejne etapy gry. Najpierw inicjuje bibliotekę allegro, tworzy wskaźniki na bitmapy, przypisuje im adresy w pamięci, tworzy tablicę zawierającą rozmieszczenie pionków na planszy, przygotowuje się do obsługi myszy, wyświetla ilości zbitych pionków i zajmuje się nadzorowaniem przebiegu gry. |