Gliwice, 07.06.2019

**Laboratorium**

**Programowania Komputerów**

Temat:

Onslaught

Autor: Patryk Frączyński

Informatyka, semestr 4, grupa 6

Prowadzący: Dr inż. Jolanta Kawulok

# 1. Temat

Onslaught – gra typu Tower Defence z użyciem biblioteki SFML.

# 2. Analiza, projektowanie

**2.1 Algorytmy, struktury danych, ograniczenia specyfikacji**

* struktury danych:

Do programu została użyta tylko tablica dwuwymiarowa o wymiarze [8][8], gdyż gra nie wymaga bardziej złożonych struktur danych, np. dynamicznych.

* algorytmy:

Do wykonania programu została użyta duża liczba algorytmów, które były projektowane głównie dla osiągnięcia jak najlepszej złożoności obliczeniowej, lecz złożoność czasowa także była ważnym elementem, lecz w tym konkretnym przypadku nie stanowi zbyt dużego utrudnienia, gdyż algorytmy są stosunkowo krótko, toteż z małymi wymaganiami czasowymi. Złożoność pamięciowa była najmniej brana pod uwagę, gdyż pamięć jest w małym stopniu wykorzystywana.

* ograniczenia specyfikacji:

Gra warcaby posiada jednoznaczne zasady, które program musi przestrzegać, dlatego niedozwolone było dopuszczenie jakichkolwiek ograniczeń, które powodowały by sprzeczność z zasadami. W tym programie istniała jedynie możliwość rozszerzenia funkcjonalności, jak zliczanie ilości zbitych pionków, pokazywanie czyja kolej ruchu oraz wyświetlanie napisu oznaczającego koniec gry.

**2.2 Analiza problemu, podstawy teoretyczne**

Program został stworzony w taki sposób, by stosował się do wszystkich zasad obowiązujących w grze warcaby, nie dopuszczając do nieprawidłowych ruchów pionkami, przez co znacznie wspomaga użytkownika, lecz on także powinien posiadać wiedzę na temat wszystkich zasad obowiązujących w tej grze.

Obowiązujące zasady to:

* Gra rozgrywana jest na ciemnych polach planszy o rozmiarze 8 × 8 pól.
* Gra warcaby klasyczne (określana też warcabami brazylijskimi) jest rozgrywana na planszy 8x8 pól pokolorowanych na przemian na kolor jasny i ciemny.
* Każdy gracz rozpoczyna grę z dwunastoma pionami (jeden koloru białego, drugi – czerwonego) ustawionymi na ciemniejszych polach planszy, w sposób przedstawiony na poniższej ilustracji.
* Jako pierwszy ruch wykonuje grający pionami białymi, po czym gracze wykonują na zmianę kolejne ruchy.
* Celem gry jest zbicie wszystkich pionów przeciwnika (w tym damek – patrz niżej) albo zablokowanie wszystkich, które pozostają na planszy, pozbawiając przeciwnika możliwości wykonania ruchu. Jeśli żaden z graczy nie jest w stanie tego osiągnąć (każdy z graczy wykona po 15 ruchów damkami bez zmniejszania liczby pionów pozostających na planszy), następuje remis.
* Piony mogą poruszać się o jedno pole do przodu po przekątnej (na ukos) na wolne pola.
* Bicie pionem następuje przez przeskoczenie sąsiedniego pionu (lub damki) przeciwnika na pole znajdujące się tuż za nim po przekątnej (pole to musi być wolne). Zbite piony są usuwane z planszy po zakończeniu ruchu.
* Piony mogą bić zarówno do przodu, jak i do tyłu.
* W jednym ruchu wolno wykonać więcej niż jedno bicie tym samym pionem, przeskakując przez kolejne piony (damki) przeciwnika.
* Bicia są obowiązkowe.
* Podczas bicia nie można przeskakiwać więcej niż jeden raz przez ten sam pion (damkę).
* Pion, który dojdzie do ostatniego rzędu planszy, staje się damką, przy czym jeśli w jednym ruchu w wyniku wielokrotnego bicia przejdzie przez ostatni rząd, ale nie zakończy na niej ruchu, to nie staje się damką i kończy ruch jako pionek.
* Kiedy pion staje się damką, kolejny ruch przypada dla przeciwnika.
* Damki mogą poruszać się w jednym ruchu o dowolną liczbę pól do przodu lub do tyłu po przekątnej, zatrzymując się na wolnych polach.
* Bicie damką jest możliwe z dowolnej odległości po linii przekątnej i następuje przez przeskoczenie pionu (lub damki) przeciwnika, za którym musi znajdować się co najmniej jedno wolne pole – damka przeskakuje na dowolne z tych pól i może kontynuować bicie (na tej samej lub prostopadłej linii).
* Kiedy istnieje kilka możliwych bić, gracz musi wykonać maksymalne (tzn. takie, w którym zbije największą liczbę pionów lub damek przeciwnika).

**3. Specyfikacja zewnętrzna**

Uruchomiony program od razu jest gotowy do rozgrywki i zgodnie z zasadami czeka, aż gracz z pionkami białymi wykona pierwszy ruch, a następnie rozgrywa się dalsza część gry.

**3.1 Obsługa programu**

Program obsługuje się przy pomocy myszy, klikając LPM na odpowiednie pola. Po kliknięciu na pole z pionkiem, należy następnie kliknąć na wolne, docelowe pole, na które może, zgodnie z zasadami, ruszyć się pionek. W przypadku wielokrotnego bicia jednym pionkiem, należy kliknąć najpierw pole z pionkiem, a następnie po kolei klikać wolne pola, przez które musi przejść pionek. Gra kończy się automatycznie zgodnie z zasadami, lecz istnieje możliwość jej wcześniejszego zakończenia, klikając na przycisk „wyjście” w dolnym prawym rogu okna.

**3.2 Format danych wejściowych**

Program pobiera z plików zewnętrznych grafiki potrzebne do reprezentowania stanu gry na ekranie, takich jak grafiki pionków, planszy, przycisku „wyjście” oraz napisu „koniec”. Ilość, rodzaj i rozmiar tych plików są na stałe wpisane do programu.

**3.3 Komunikaty**

W celu poprawy efektu wizualnego programu został zastosowany bufor, na którym wszystko jest po kolei wyświetlane, a gdy zostanie to ukończone to dopiero wtedy, jego zawartość trafia na ekran. W przypadku błędu podczas tworzenia tego bufora zostanie wyświetlony komunikat:

**,,** **Blad tworzenia bufora !"**

Gdy zabraknie plików graficznych zostanie wyświetlony komunikat:

**,,** **Brak wszystkich niezbednych plikow graficznych !"**

Po owych komunikatach program zawsze kończy działanie, by użytkownik rozwiązał problem.

**4. Specyfikacja wewnętrzna**

**4.1** **Zmienne**

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

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

**4.2 Funkcje**

|  |
| --- |
| void wyswietl(BITMAP \*\* bufor, BITMAP \*\* plansza, BITMAP \*\* pb, BITMAP \*\* pcz, BITMAP \*\* db, BITMAP \*\* dcz, BITMAP \*\* wyjscie, char tab[8][8], char ruch) |
| 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. |

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

|  |
| --- |
| void pobierz\_wspolrzedne(int \*tmp\_mx, int \*tmp\_my) |
| 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. |

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

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

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

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

|  |
| --- |
| int ruchf(char tab[8][8], int \*my, int \*mx, int \*tmp\_my, int \*tmp\_mx, char ruch, char zbite[2]) |
| 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. |

|  |
| --- |
| int czy\_przeskoczony\_pionek(char \*elem, char ruch, int \*zbijane) |
| 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. |

|  |
| --- |
| 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) |
| 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. |

|  |
| --- |
| int bicie(char tab[8][8], int \*my, int \*mx, int \*tmp\_my, int \*tmp\_mx, char ruch, char zbite[2]) |
| 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. |

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

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

|  |
| --- |
| int main(void) |
| 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. |

**6. Testowanie**

Programbył testowany wielokrotnie ze względu na poprawność jego działania, zarówno przez bardziej jak i znacznie mniej doświadczonych graczy w warcaby. Wszelkie znalezione błędy zostały wyeliminowane. Program nie wysypuje się w żadnym przypadku, zawsze działa zgodnie z zasadami. W przypadku jakichkolwiek niedociągnięć ze strony użytkownika (np. skasowanie któregokolwiek pliku z grafiką) program wyświetlał stosowny komunikat.

**7. Wnioski**

Projekt Warcaby początkowo wydawał się prosty w realizacji, lecz obowiązujące zasady gry bardzo to zmieniły. Wydaje mi się, że program jest przygotowany na wszystkie możliwe do wystąpienia sytuacje, lecz dalej będzie przeze mnie udoskonalany (np. tworzenie nowych animacji), ale aktualnie program jest już w pełni funkcjonalny.