Uniwersytet Łódzki  
Wydział Matematyki i Informatyki

Artur Plesiński  
Nr albumu: 359784

Implementacja gry Saper  
o heksagonalnych polach  
w języku C#

Praca napisana pod kierunkiem  
dr Sebastiana Lindnera  
Katedra Funkcji Rzeczywistych

1. Wstęp

Większość osób, które miały do czynienia ze starszymi systemami operacyjnymi firmy Microsoft, zapewne w pewnym momencie użytkowania platformy spotykało się z zakładką „Gry”. Znajdowały się tam różne proste umilacze czasu, takie jak kierki, pasjans, warcaby, pinball czy saper, które w choćby najmniejszym stopniu pozwalały uporać się z nudą. To właśnie na saperze spędziłem zdecydowanie najwięcej czasu śrubując możliwie najszybsze czasy rozbrojenia planszy.

Saper (ang. Minesweeper) to gra stworzona w 1989 r. przez Curta Johnsona a następnie przeportowana przez Roberta Donnera.

Pierwotnie gra była przeznaczona na system operacyjny IBMu - OS/2. To Robert Donner, po otrzymaniu od Johnsona kodu źródłowego oryginału, przeportował Sapera na OS Microsoftu - Windows 3.0.

Oryginalna wersja gry stawiała graczowi za zadanie rozbrojenie pola minowego w kształcie prostokąta. Siatka była wypełniana kwadratowymi komórkami, z których niektóre stawały się minami. Gracz mógł odkrywać kolejne pola, a także zaznaczać miejsca, gdzie według niego znajdować się mogła mina. Jeżeli odkryte pole sąsiadowało z miną, na polu wyświetlana była liczba odpowiadająca ilości min sąsiadujących z polem. Celem gry było więc odkrycie wszystkich bezpiecznych pól. Celem drugorzędnym było zrobienie tego w jak najkrótszym czasie.

2. Cel pracy

Celem pracy jest zaprezentowanie implementacji gry „Saper” w języku programowania C#.

Moja wersja gry została napisana w języku C# z pomocą Windows Forms – graficznego interfejsu użytkownika udostępniającego elementy zawarte w interfejsie graficznym systemu operacyjnego Microsoft Windows. Postanowiłem nieco zmienić oryginalny zamysł gry, dlatego też stworzyłem wariant z polami w kształcie heksagonów.

3. Wybór narzędzi

a) język programowania – C#

Przy realizacji celu pracy chciałem skupić się na logice gry. Jest to kluczowe dla grywalności produktu. Taki element jak interfejs użytkownika także jest niewątpliwie ważny, jednak do stworzenia przejrzystej i prostej aplikacji zdecydowanie wystarczą znane już rozwiązania. Język C# ma wsparcie od GUI Windows Forms, który dostarcza kontrolki znane z systemu Windows np. okna, guziki, pola na tekst.

Do stworzenia projektu wystarczyłyby inne znane mi języki programowania takie jak Java czy choćby C++. Trzeba jednak nadmienić, że znane mi GUI do Javy (np. JavaFX) nie grzeszą szybkością działania w porównaniu właśnie do WF w C#. C# w porównaniu do C++ jest wygodniejszy – dealokacja pamięci leży po stronie garbage collectora. Łatwo jest też przenieść aplikację między platformami Microsoft np. Windows, Windows Phone czy konsola Xbox.

b) IDE (środowisko programistyczne) – Microsoft Visual Studio Community 2017

Osoba chcąca pracować w języku C# nie musi się wiele zastanawiać przy wyborze narzędzia do pisania kodu. Opcje są właściwie dwie – Visual Studio oraz Rider. Oba programy są bardzo rozwinięte, jednak dla domowego użytkownika za tym pierwszym przemówić może fakt, że jest on dostępny za darmo w wersji Community.

Rider jest produktem firmy JetBrains, do którego trzeba wykupić licencję. Jest też możliwość otrzymania rocznej subskrybcji na produkty JetBrains jeśli jest się studentem. Myślę jednak, że Visual Studio nie jest gorsze, zwłaszcza że jego twórca – Microsoft – jest też autorem języka C#.

c) GUI – Windows Forms

Jest to najłatwiej dostępna (domyślnie WF jest zainstalowane w Visual Studio) opcja, która dostarcza kontrolki potrzebne do stworzenia przejrzystego interfejsu.

4. Konstrukcja aplikacji

a) założenia

Saper Heksagonalny jest luźnie inspirowany produktem Microsoftu. Do mojej aplikacji został przeniesiony system flag. Są też obecne stoper oraz licznik pozostałych bomb. Tu także za przycisk „restartu” gry będzie robił obrazek przedstawiający charakterystyczną „buźkę”. Pola odkrywa się lewym przyciskiem myszy, a flagę stawia się prawym przyciskiem myszy.

b) dwuwarstwowość

Moja aplikacja została wyraźnie podzielona na 2 niezależne od siebie warstwy. Pierwszą z nich jest interfejs stworzony z pomocą Windows Forms. To właśnie WF dostarcza wszystkie widoczne na ekranie gry kontrolki - przyciski, panel, w którym rysowana jest gra, czasomierz, licznik, a także kolejne okienka. Gra składa się z trzech okien:

- okno gry - okno główne wyświetlane w czasie gry;

- okno ustawień - wyświetlane przy starcie gry i przy jej restarcie. Pozwala wybrać liczbę kolumn, wierszy oraz min;

- okno końca gry - wyświetlane przy wygranej oraz przegranej. Pozwala zakończyć grę bądź ją zresetować.

Wybrane w oknie ustawień parametry są zapisywane w ramach jednej instancji aplikacji. Przy restarcie gry, do nowej rozgrywki są przekazywane parametry z poprzedniej rozgrywki.

Drugą z warstw jest logika gry wyodrębniona w osobnej klasie (Map.cs). Znajdują się tu wszystkie metody, które są odpowiedzialne za zachowanie gry w zgodzie z jej zasadami. Dzięki tym metodom m.in. losowo rozstawimy bomby na mapie czy wyliczymy tablicę sąsiadów.

Dwuwarstwowość aplikacji niesie za sobą znaczącą korzyść, mianowicie możliwość łatwego przeniesienia gry na inne języki. Oczywiście wciąż trzeba zmodyfikować kod, żeby pasował do składni danego języka, ale jest to niski koszt w porównaniu z zyskiem.

5. Wykonanie aplikacji

a) Interfejs

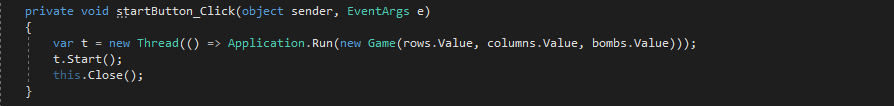
Aplikacja jest stworzona przy pomocy Windows Forms. Wszelkie komponenty aplikacji, łącznie z samymi oknami, są dostarczone przez ten framework.

Saper Heksagonalny składa się ze wspomnianych trzech okien, z czego każdy jest osobnym formularzem systemu Windows. W ramach każdego formularza znajduje się kilka kontrolek.

Komunikacja między formularzami ma dwa zadania - wywoływanie innego formularza oraz przesyłanie do niego zapisanych danych.

Aplikacja rozpoczyna się wywołaniem formularza Options (Form2.cs), który prosi użytkownika o podanie liczbę wierszy, kolumn oraz min na mapie. Po kliknięciu przycisku START Options się zamyka, a otwiera się formularz Game (Form1.cs). Tutaj rozpoczyna się właściwa gra. Formularz może się zakończyć na 2 sposoby (pomijając oczywiście wyjście z gry) - poprzez ukończenie gry (zarówno porażka jak i wygrana) oraz poprzez wcisnięcie „buźki”. Druga opcja przenosi użytkownika z powrotem do formularza Options, natomiast ta pierwsza otwiera prosty formularz GameOver (Form3.cs), z którego można po prostu zakończyć grę bądź ją zrestartować, co przeniesie użytkownika do formularza Options.

Każdy z formularzy działa na osobnym wątku.

(obrazek)

W powyższym przykładzie zostało pokazane wywołanie formularza Game przy kliknięciu przycisku START w formularzu Options. Poprzez konstruktor zostaje tworzony formularz z parametrami, które mówią o ilości wybranych wierszach, kolumnach oraz minach. Obiekty rows, columns i bombs to nic innego jak kontrolki typu NumericUpDown - w formularzu to miejsca, w których użytkownik wybiera parametry gry. Pole Value to wpisana w danym miejscu wartość.

Wartości parametrów są zapisywane w ramach instancji formularza Game (a następnie przekazywane następnemu formularzowi), żeby przy restarcie gry użytkownik nie musiał ponownie wpisywać interesujących go specyfikatów.

Głównym elementem formularza Game jak i właściwie całej aplikacji jest PictureBox - kontrolka służąca do wyświetlania obrazu. To dzięki niej wyświetlana jest siatka heksagonów. Najważniejszym zdarzeniem dotyczącym PictureBoxa jest zdarzenie Paint. Paint jest wywoływane, gdy aplikacja uważa, że narysowane na ekranie informacje nie są poprawnym odzwierciedleniem danych. Innymi słowy rysowanie w PictureBoxie wywołuje się, gdy rzeczywiście coś się w przedstawionych danych zmienia.

b) Logika gry

Większość logiki gry została umieszczona w klasie Map.cs. Formularz Game dokłada do niej jedynie obsługę błędów w postaci sprawdzania, czy kliknięte pole może posłużyć za dane potrzebne przy wywoływaniu metod. Po prostu zezwala bądź odmawia wywołania jakiejś metody w zależności od klikniętego miejsca.

W klasie Map znajdują się 3 dwuwymiarowe tablice będące odzwierciedleniem pola gry. Są to:

- tablica pól i min,

- tablica zaznaczonych flag,

- tablica liczby sąsiadujących min.

Komórki w tablicy odkrytych pól i min są wypełniane liczbą. Oto możliwe wartości:

⦁ -1: Komórka jest odkryta.

⦁ 0: Komórka nieodkryta, bez bomby.

⦁ 1: Komórka nieodkryta, z bombą.

Tablica jest sprawdzana przy kliknięciu lewym przyciskiem myszy, a także przy wyliczaniu tablicy sąsiadów.

Tablica zaznaczonych flag jest prostą tablicą, w której są tylko dwie wartości - jedna oznaczająca postawioną flagę oraz - siłą rzeczy - drugą, która oznacza brak flagi.

Ostatnia tablica odzwierciedla liczbę sąsiadujących z danym polem min. Jest to tablica łańcuchów, w której jest możliwych 7 wartości. Jest to zbiór {"0","1","2","3","4","5","6"}. Oczywiście jest to podyktowane tym, że większość pól (pomijając te usytuowane na krawędzi siatki) może mieć od 0 do 6 sąsiadujących z nimi min.

Ważnym polem klasy Map dla logiki gry jest pole typu int, *remainingTiles*. Jego wartość obliczana jest przy tworzeniu siatki. Jest to liczba wszystkich, możliwych do odkrycia, pól wyłączając te z miną:

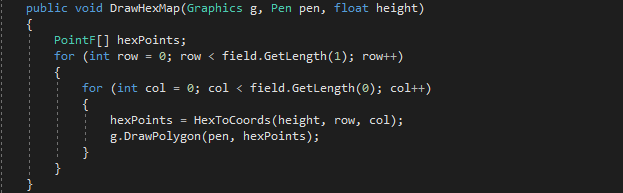
remainingTiles = [szerokość siatki] \* [wysokość siatki] - [liczba min];

Jego wartość jest zmniejszana o 1 za każdym razem, gdy odkrywane jest kolejne pole w siatce tj., gdy wartość komórki w tablicy odkrytych pól zmienia się z 0 na -1. Warunkiem zwycięstwa w grze jest zmniejszenie wartości pola do 0. Oznacza to, że gracz odkrył wszystkie możliwe pola bez zdetonowania miny. Warto tu jedynie wspomnieć o warunku porażki, gdyż jest krótki i prosty - kliknięcie na pole o wartości 1 - oznacza zdetonowanie miny.

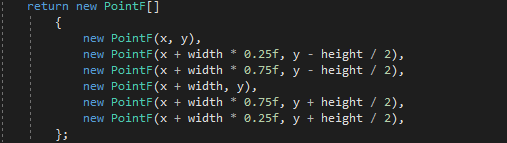
c) Rysowanie pola gry

Za rysowanie całej siatki odpowiadają 4 metody w klasie Map:

* DrawHexMap(Graphics, Pen, float) - wywoływana na starcie gry.



Jest to proste narysowanie kolejnych heksagonów, jeden po drugim, według rozmiarów tablicy. Ważnym elementem jest dodatkowa funkcja HexToCoords(float, float, float), która zwraca tablicę typu PointF o rozmiarze 6. Każdy z punktów w zwróconej tablicy oznacza kolejny wierzchołek heksagonu:



Metoda, znając wysokość pojedynczego heksagonu oraz jego kolejność na mapie, jest w stanie podać jego dokładne koordynaty.  
Ostatecznie jeden heksagon w pętli rysowany jest przy pomocy metody DrawPolygon() z klasy Graphics.

* DrawBombs(Graphics, Brush, float) - metoda działająca identycznie jak DrawHexMap, z tym, że wyznaczony heksagon jest rysowany z wypełnieniem, bez kontur (metoda FillPolygon()). Metoda wywoływana, gdy użytkownik odkryje pole z miną - odkrywane są wszystkie pola mające wartość 1 w tablicy odkrytych pól i min.
* DrawRevealed(Graphics, Brush, float) - bliźniacza do DrawBombs() metoda wywoływana przy odkrywaniu pól. Jeśli pole nie sąsiaduje z miną, jest wypełniane kolorem. Jeśli sąsiaduje z bombą, to prócz wypełnienia wstawiany w heksagon jest również plik graficzny reprezentujący liczbę. Tu przydaje się też tablica liczby sąsiadujących min.
* DrawFlags(Graphics, Brush, float) - rysuje flagi. Identyczna funkcja jak DrawBombs, z tym że wywoływana przy kliknięciu pola prawym przyciskiem myszy.

d) Algorytm powodzi (flood fill)

Jest to algorytm służący do wypełniania zadanego obszaru zgodnie z ustalonymi warunkami. W saperze służy do tego, by użytkownik nie musiał odkrywać wszystkich pól, które nie sąsiadują z bombą. Po odsłonięciu takiego pola zostają odkrywane sąsiednie komórki również nie będące w styku z miną. Algorytm kończy się, gdy natrafi na pole sąsiadujące z miną.

Pseudokod algorytmu dla mojej aplikacji można zapisać następująco:

RevealNeighbours(kolumna, wiersz):  
jeżeli (punkt jest miną bądź na punkcie jest postawiona flaga) return;  
jeżeli (na punkcie nie ma miny, ale punkt sąsiaduje z bombą) odkryj pole, return;  
jeżeli (na punkcie nie ma miny, punkt nie sąsiaduje z bombą) odkryj pole, wywołaj RevealNeighbours dla wszystkich sąsiadujących pól  
jeżeli (pole było już odkryte) return;

Algorytm wywołuje się rekurencyjnie. Oczywiście pseudokod nie przedstawia całej metody. Tworzony kod musiał być odporny na błędy, dlatego też trzeba było obsłużyć przypadki krańcowe. Mowa o przypadkach, gdy heksagon jest na krawędzi siatki. Trzeba wtedy wyznaczyć innych sąsiadów pola, żeby nie wyjść poza zakres tablicy. Do tego dochodzi fakt, że pole ma różnych sąsiadów w zależności od tego, czy numer jego kolumny jest parzysty bądź nieparzysty.