LABORATORIUM SYSTEMÓW MOBILNYCH

STWORZENIE MODUŁU POBIERAJĄCEGO WEKTOROWE DANE PRZESTRZENNE Z BAZY DANYCH I PRZYGOTOWUJĄCEGO JE DO WYŚWIETLENIA

I. Temat ćwiczenia

Stworzenie modułu przygotowującego dane do wyświetlenia i pobierającego dane z bazy danych.

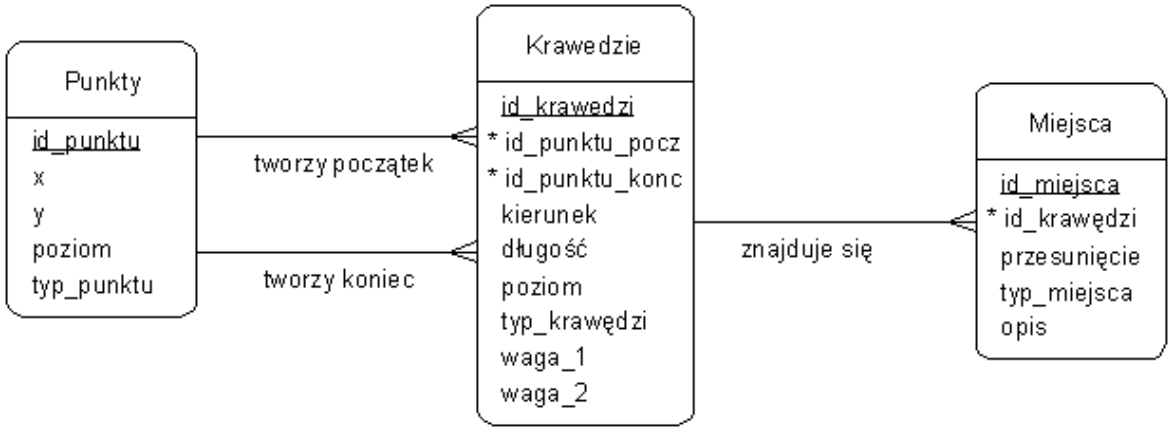
II. Wymagania

Znajomość SQL-a
Znajomość dostępu i odczytywania danych z bazy relacyjnej (**SQL Server CE**) – na podstawie schematu z ćwiczenia „Implementacja modułu konwersji danych przestrzennych”

III. Ćwiczenie

1. Stworzenie modułu do odczytywania danych z bazy SQL

Należy stworzyć moduł **MapDB** do odczytywania danych z bazy danych SQL.
Schemat bazy danych zamieszczony jest poniżej:



Rysunek 1 – Schemat relacji bazy danych przechowywania danych przestrzennych

W celu stworzenia modułu do obsługi bazy danych należy dodać referencje do następujących bibliotek:

- System.Data.dll
- System.Data.SqlClient.dll
- System.Data.SqlServerCe.dll

Stworzona klasa **MapDB** powinna umożliwiać:

- Otwarcie połączenia z baza danych,
- Zamknięcie połączenia z baza danych,
- Odczytywanie danych.

Przykładowe funkcje otwierające i zamykające połączenie z bazą danych:

```
public void OpenConnection()
{
      connection = new SqlConnection(this.ConnectrionString);
      connection.Open();
}
```

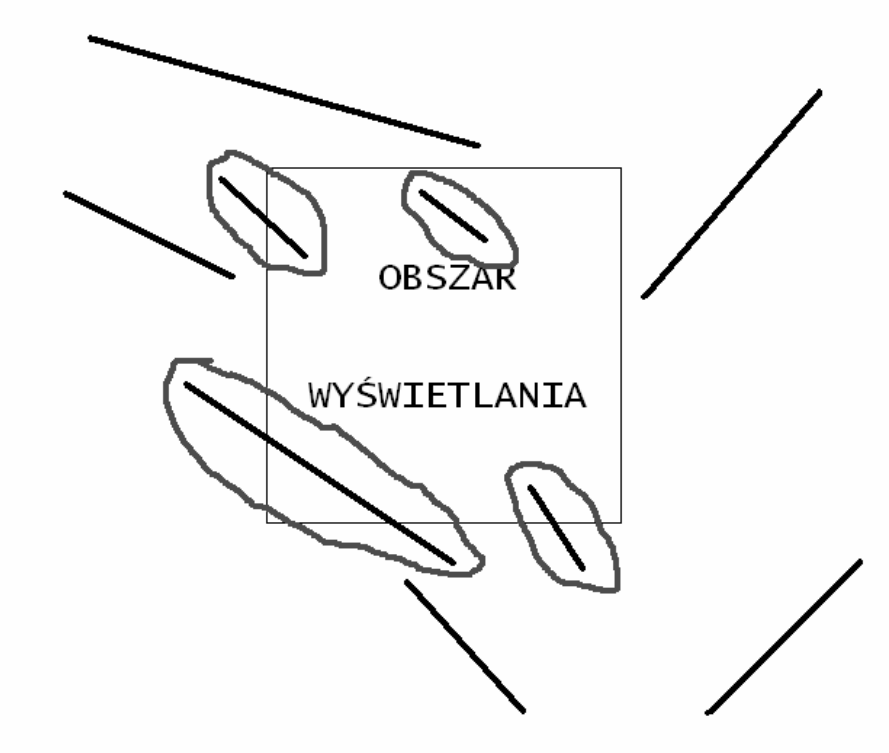
```
public void CloseConnection()
{
      connection.Close();
}
```

Do odczytywania danych należy użyć następujący wzorzec funkcji:

```
public Krawedzie[] ReadEdges(double x1, double y1, double x2,
                             double y2, TypKrawedzi typ)
```

Odczytywane danych powinno obsługiwać następujące sposoby odczytywania danych:

- Odczyt listy krawędzi określonego typu (umożliwia przy danym poziomie powiększenia odczytywanie tylko danych dotyczących głównych ulic, z pominięciem ulic mniejszych lub ścieżek),
- Pominięcie w odczytywaniu krawędzi, które rozpoczynają się i kończą poza obszarem wyświetlania (współrzędne X i Y punktu początku i końca) – **UWAGA! Należy wziąć pod uwagę krawędzie, które rozpoczynają się i kończą poza obszarem wyświetlania, ale przechodzą przez ten obszar !!!** (**patrz rysunek poniżej**) Funkcja powinna pobierać aktualny obszar wyświetlania i na jego podstawie generować odpowiednie zapytanie SQL pobierające krawędzie z bazy danych.



Rysunek 2 – Przykład krawędzi, które powinny zostać pobrane z bazy

- Odczyt listy miejsc znajdujących się przy odczytanych krawędziach:

```
public Places[] ReadPlaces(int x1, int y1, int x2, int y2,
                           TypKrawedzi typ, TypMiejsca typ2)
```

Z uwagi na ograniczenia związane z urządzeniami mobilnymi, moduł MapDB powinien być zoptymalizowany pod katem:

- Małej ilości zapytań SQL pobierających dane,
- Dużej selektywności zapytań (należy pobierać tylko krawędzie i miejsca, które będą wyświetlane).

2. Stworzenie modułu przygotowującego dane do wyświetlenia

Moduł **MapPrepare** powinien pobierać dane z bazy danych (za pośrednictwem modułu MapDB) i przygotowywać je do wyświetlenia.

Ponieważ ograniczenia wynikające z urządzeń mobilnych typu **Pocket PC** lub **SmartPhone** są dość znaczące, a moc przetwarzania mała, duże znaczenie ma odpowiednie przygotowanie danych przed ich wyświetleniem.

Wcześniejsze odpowiednie przygotowanie modułu **MapPrepare** sprawia, iż możliwe jest późniejsze płynne wyświetlanie, powiększanie i przewijanie mapy.

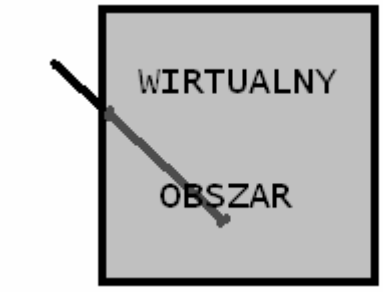
Z punktu widzenia aplikacji mobilnych, operacjami najwolniejszymi są operacje rysowania bezpośrednio na formatce (ekranie). Zamiast nich, wszędzie gdzie to możliwe należy wykorzystać wyświetlanie „gotowej”, przygotowanej wcześniej bitmapy.

Zaimplementować funkcję generującą obrazek **Bitmap** na podstawie danych odczytanych z bazy.

Przykładowa funkcja została umieszczona poniżej:

```
public Image RenderImage(Rectangle rect)
{
      // tworzenie bitmapy
      Image bmp = new Bitmap(rect.Width, rect.Height);

      // pobranie wirtualnego kontekstu urzadzenia
      Graphics gr = Graphics.FromImage(bmp);

      // rysowanie na bitmapie
      gr.Clear(Color.White);

      Pen blackPen = new Pen(Color.Black);

      // dla kazdej linii
      gr.DrawLine(blackPen, x1, y1, x2, y2);

      // dla kazdego miejsca
      gr.DrawRectangle(blackPen, x1, y1, x2, y2);

      // zwolnienie kontekstu urzadzenia
      gr.Dispose();

      // zwrocenie wygenerowanej bitmapy
      return bmp;
}
```

Drugą rzeczą pod względem „pracochłonności” i „czasochłonności” zajmują procedury służące do rysowania z wykorzystaniem wirtualnego kontekstu urządzenia pobranego z obiektu bitmapy (rysowanie na obiekcie w pamięci). Dlatego też należy zwrócić szczególną uwagę na to, aby wcześniej odpowiednio „przyciąć” w sposób matematyczny początki i końce rysowanych linii, aby przekazywać do metody DrawLine tylko części linii, które znajdują się dokładnie na rysunku (patrz poniżej).



Rysunek 3 – Obcinanie punktu początku i końca krawędzi przed wyświetleniem

Należy tak przerobić funkcję **RenderImage()**, aby rysowała w sposób optymalny (z zastosowaniem ręcznego przycinania).

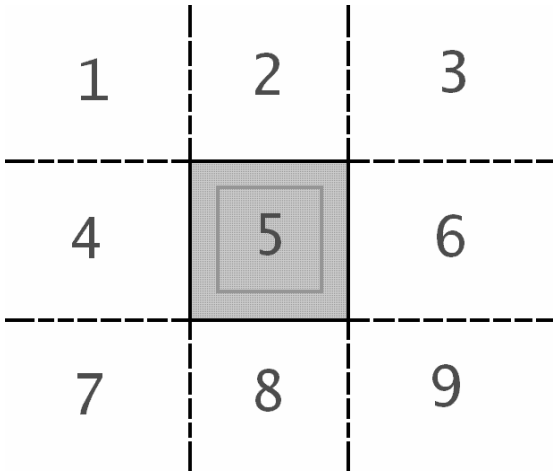
Wykorzystanie wygenerowanej w ten sposób bitmapy wydaje się proste z punktu widzenia użytkownika aplikacji. Wystarczy po prostu wygenerować odpowiedni (aktualnie wyświetlany) wycinek mapy i wyświetlić tak wyrenderowany obrazek na formatce.

Problem pojawia się natomiast w momencie wykonywania operacji przesuwania mapy. Każda operacja przesuwania mapy przez użytkownika (lub nawet automatycznie przez program – np. podczas śledzenia ruchu samochodu) wiąże się za każdym razem z ponownym przerysowaniem wyświetlanego obszaru.

Dobrym pomysłem na optymalizację jest generowanie większego obszaru, i wyświetlanie w danym momencie fragmentu wygenerowanego obrazka. Likwiduje to w ogólności wspomniany problem, jednakże zauważyć możemy, iż problem ten znowu się pojawi w momencie dojścia do granicy wygenerowanego obszaru.

Innym pomysłem na optymalizację tej operacji (i często wykorzystywanym w profesjonalnych zastosowaniach) jest przechowywanie jednocześnie wygenerowanych sąsiednich obszarów, tak aby w momencie przesuwania bitmapy można było w sposób płynny przejść do sąsiednich obszarów.

Należy rozszerzyć implementację klasy **MapPrepare** tak, aby zawsze przechowywała wygenerowane i gotowe do wyświetlenia 9 bitmap, które będą pokrywać aktualnie wyświetlany obszar i obszary sąsiednie, dzięki czemu możliwe będzie potem płynne przewijanie mapy.



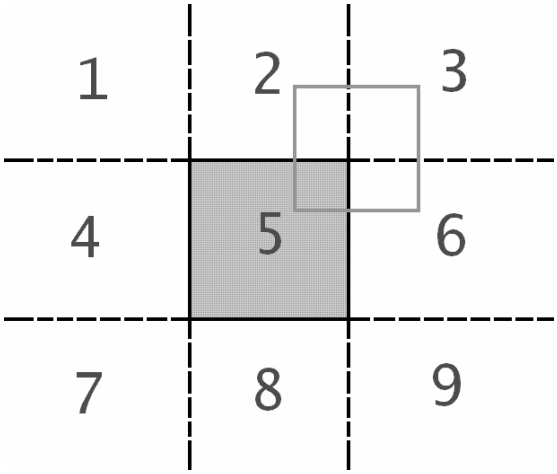
Rysunek 4 – Generowane obszary (bitmapy)

Każdy z obszarów od 1-9 powinien mieć taką samą wielkość. Warto zauważyć, iż realny obszar wyświetlania na **Pocket PC** (zaznaczony na **zielono**) powinien być nieco mniejszy od całego obszaru głównej bitmapy. Dzięki temu, możliwe będzie (jak użytkownik będzie przewijał mapę, od razu wyświetlenie mu nowej (przesuniętej mapy), a dopiero później (w tle), pobranie nowego obszaru (oznaczonego na rysunku numerem 5).

Wszystkie obszary 1-9 powinny być przechowywane w postaci wygenerowanej w obiekcie **MapPrepare**. Sama natomiast aplikacja powinna przechowywać u siebie jedynie obszar 5 i wywoływać funkcję **MapPrepare.GetMainRegion(x1, y1, x2, y2)** celem pobrania głównego obszaru bitmapy (obszaru 5), którego część będzie wyświetlana użytkownikowi na ekranie urządzenia.

Na uwagę zasługuje problem przejścia między obszarami sąsiednimi.

Przykład wywołania funkcji dla obszaru pośredniego pokazano poniżej:



Rysunek 5 – Generowanie głównego obszaru po przesunięciu (bitmapy)

W przypadku przykładowego przesunięcia obszaru w kierunku prawego górnego rogu funkcja **GetMainRegion()** powinna być zaimplementowana w następujący sposób:

- Generować nowy **obszar 5** zwracany aplikacji, na podstawie części danych z obszarów **2, 3, 5, 6** (w przypadku przesunięcia tak jak na rysunku).

Aby nie generować jeszcze raz i nie rysować danych, należy brać już wygenerowane bitmapy sąsiednie i z nich „sklejać" nowy obszar „5" przekazywany do aplikacji w celu wyświetlenia.

Do sklejania obszarów przydatne będą następujące funkcje:

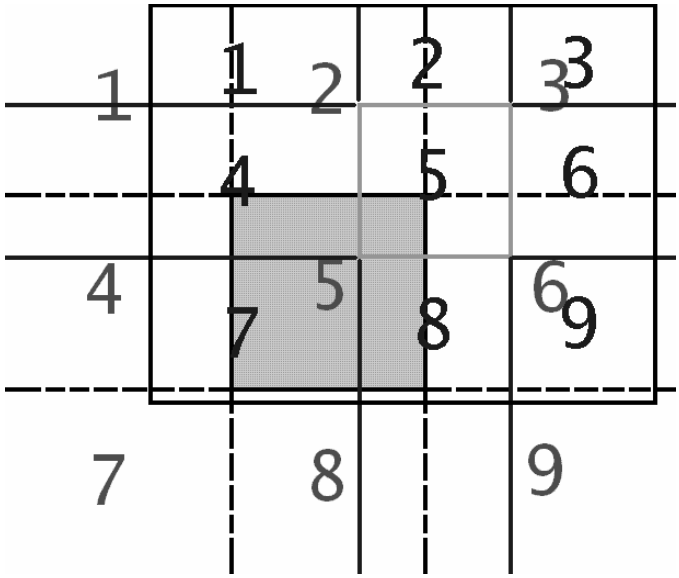
```
// stworzenie nowego obszaru
Image nowy_obszar = new Bitmap(szerokosc_obszaru_5, wysokosc_obszaru_5);
// otwarcie obszaru do rysowania
System.Drawing.Graphics gr = Graphics.FromImage(nowy_obszar);
// skopiowanie czesci bitmapy
gr.DrawImage(stary_obszar_2,
      new Rectangle(1, 1, 10, 10),
      new Rectangle(11, 11, 20, 20),
      GraphicsUnit.Pixel);
// zamkniecie obszaru do rysowania
gr.Dispose();
```

Listing 1 – Przykładowe generowanie bitmapy

W powyższy sposób należy zrealizować funkcję **GetMainRegion**, aby w zależności od przesunięcia tworzyła i wyświetlała nowy obszar użytkownikowi sklejając go odpowiednio z części obszarów sąsiednich.

Następnie funkcja **GetMainRegion** powinna:

- Wygenerować nowe obszary **1-9**. Należy zauważyć, iż dzięki optymalizacji polegającej na „**sklejaniu**" starych obszarów, obszary **4,6,7,8** mogą być wygenerowane w sposób **automatyczny** na podstawie starych obszarów (rysunek poniżej – nowe obszary zostały zaznaczone na **niebiesko**),



Rysunek 6 – Nowe obszary 1-9

- Pozostałe obszary **1,2,3,6,9** (które nie są od razu widoczne w całości) należy ponownie wygenerować na podstawie informacji odczytanych z bazy danych (w przypadku niekompletności informacji, należy odczytać z bazy brakujące krawędzie i miejsca).

Procedura generowania obszarów sąsiednich powinna być wykonywana w odpowiednio stworzonym wątku, tak aby użytkownik mógł od razu wrócić i kontynuować pracę z aplikacją.

Przykład tworzenia i uruchamiania wątku został zamieszczony poniżej:

```
public class ThreadExample
{
      public static void ThreadProc()
      {
            for (int i = 0; i < 10; i++)
            {
                  Console.WriteLine("ThreadProc: {0}", i);
                  Thread.Sleep(0);
            }
      }
      public static void Main()
      {
            // stworzenie watku
            Thread t = new Thread(new ThreadStart(ThreadProc));
            // uruchomienie watku
            t.Start();
      }
}
```

Listing 2 – Przykład wykorzystania wątków

Niestety z wewnątrz wątku dostęp do zmiennych z innego wątku może okazać się problemem, dlatego dobrą metodą jest wykorzystanie funkcji Invoke() na stworzonym delegacie będącym częścią obiektu do którego się chcemy odwołać.

Całość ilustruje poniższy przykład:

```
public delegate void SwitchBitmapDelegate(int nr, Image bmp);

public void SwitchBitmap(int nr, Image bmp)
{
      // jakas implementacja
}

public void ThreadProc()
{
      Image bmp = new Bitmap(10, 10);

      SwitchBitmapDelegate myDelegate =
            new SwitchBitmapDelegate(this.SwitchBitmap);

      form.Invoke(myDelegate, new Object[] {
                (int) 10,
                (Image) bmp
            }
      );
}
```

Listing 3 – Przykład wykorzystania delegatów

3. Inne sposoby optymalizacji

Dodatkowo można zaimplementować inne optymalizacje:

- Buforowanie większej ilości obszarów niż tylko 9,
- Stworzenie osobnych obszarów dla sąsiednich powiększeń (np. gdy aktualne powiększenie wynosi 4, można się spodziewać, że użytkownik może przejść na poziom 3 lub 5. Można więc trzymać i generować dodatkowo „obszary 5” dla sąsiednich powiększeń. W przypadku, gdy użytkownik zamiast przesunięcia wykona powiększenie lub zmniejszenie, można od razu wyświetlić mu gotową bitmapę, i dopiero wówczas wygenerować dla danego poziomu sąsiednie obszary 1,2,3,4,6,7,8,9.

4. Zadanie

Należy zrealizować funkcję **GetMainRegion**, aby w zależności od przesunięcia tworzyła i wyświetlała nowy obszar użytkownikowi sklejając go odpowiednio z części obszarów sąsiednich.

Następnie funkcja **GetMainRegion** powinna:

- Wygenerować nowe obszary 1-9,
- Pozostałe obszary 1,2,3,6,9 (które nie są od razu widoczne w całości) należy ponownie wygenerować na podstawie informacji odczytanych z bazy danych (w przypadku niekompletności informacji, należy odczytać z bazy brakujące krawędzie i miejsca).

Procedura generowania obszarów sąsiednich powinna być wykonywana w odpowiednio stworzonym wątku, tak aby użytkownik mógł od razu wrócić i kontynuować pracę z aplikacją.