Tomasz Dziubak

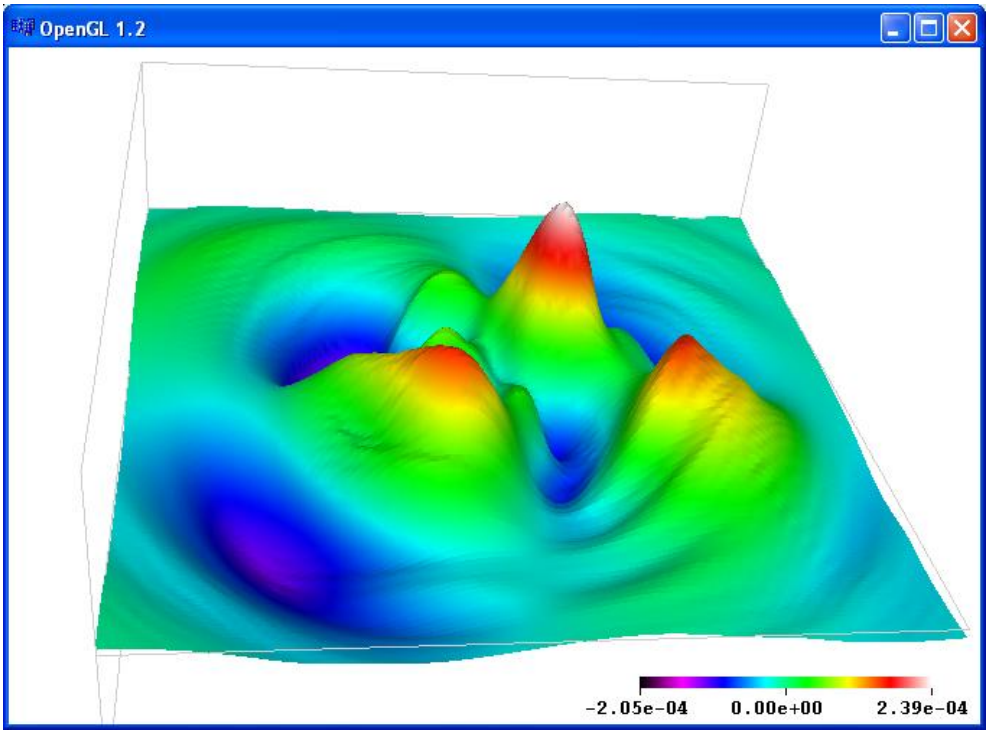
Studium przypadku: wizualizacja danych 3D (C++Builder 6)

26 listopada 2007

1. Wstęp

Skrypt opisuje przygotowanie programu służącego do prezentacji wczytanej z pliku funkcji dwóch zmiennych w trójwymiarowej przestrzeni (por. rysunek 1). Do zrozumienia omówionego niżej kodu wystarczy podstawowa wiedza o C++ oraz znajomość OpenGL w zakresie wyznaczonym przez skrypt Jacka Matulewskiego dostępny pod adresem http://www.fizyka.umk.pl/~jacek/dydaktyka/3d/OpenGL_BCB.pdf . Opisany program opiera się także na projekcie-szablonie aplikacji korzystającej z OpenGL opisanym we wspomnianym kursie (do ściągnięcia z http://www.fizyka.umk.pl/~jacek/dydaktyka/3d/szablon_bcb.zip).

Rysunek 1
Cel tego tutoriału czyli program plot3D w działaniu (z wyłączonym antyaliasingiem)



2. Założenia dotyczące programu

Program (rysunek 1), który napiszemy w tym tutoriale będzie wyświetlał funkcje falowe będące rozwiązaniami dwuwymiarowego równania Schrödingera (w zasadzie, można go będzie wykorzystać do wyświetlenia dowolnych danych umieszczonych w plikach tekstowych, w postaci kolumn i zawierających dowolną liczbę linii komentarza w nagłówku). Funkcje te zapisane są w postaci tekstowej w odpowiednich plikach. Są one zdyskretyzowane na sieci przestrzennej o parametrach czytanych również z pliku. Nagłówki plików zawierające funkcje falowe są przedstawione na listingach od 1 do 3. Pliki o rozszerzeniu *plt* (listing 1) zawierają funkcję falową, która w tym przypadku jest rzeczywista oraz współrzędne przestrzenne. Nagłówek zawiera parametry przestrzenne sieci (druga linia), które będziemy wczytywać do programu. W tym przypadku parametry sieci są następujące: zakres sieci dla osi OX jak i OY jest taki sam i wynosi od –10 do 10, także i ilość węzłów sieci jest równa dla obu osi i wynosi 10. Oczywiście jest to tylko przykładowy plik i dla innych plików parametry te mogą być inne. Kolejnym typem plików są *plt*.dat* (listing 2), które podobnie jak pliki o rozszerzeniu *plt* zawierają współrzędne przestrzenne sieci. Pliki te w odróżnieniu do wcześniej omawianych zawierają kwadrat modułu funkcji falowej. Ostatnim typem plików są pliki *.dat* (listing 3), które zawierają tylko funkcję falową bez współrzędnych przestrzennych (dokładniej jej część rzeczywistą i urojoną). W przypadku tych plików możliwe będzie wyświetlenie jedynie części urojonej lub rzeczywistej funkcji falowej (ewentualnie kwadrat części rzeczywistej lub urojonej). Natomiast, nie będzie można wyświetlić kwadratu modułu całej funkcji falowej. We

wszystkich tych plikach funkcja falowa jest zapisana w taki sposób, że najpierw zapisywane są wartości funkcji dla ustalonej wartości współrzędnej *y*, i wszystkich możliwych wartości *x*. Następnie zmieniana jest współrzędna *y* na kolejną i dla tej wartości przebiegamy po wszystkich możliwych wartościach *x*. Jest to ważne przy wczytywaniu plików typu **.dat*, w których, jak już wspominałem, nie ma zapisanych wartości współrzędnych przestrzennych. Na końcu pliku zapisane są cztery liczby określające zakres sieci dla funkcji falowej znajdującej się w tym pliku.

Listing 1. *Pliki typu *.plt zawierające funkcje falowe wraz ze współrzędnymi przestrzennymi*

```
# norm= 1   n= 1
# nx = 10      ny = 10      range = [10,10]
#          x                    y                psi(x,y,z)
-1.0000000000000e+01   -1.0000000000000e+01   6.0100769704691e-09
-9.7979797979798e+00   -1.0000000000000e+01   1.9250132270683e-08
-9.5959595959596e+00   -1.0000000000000e+01   3.3470263349353e-08
-9.3939393939394e+00   -1.0000000000000e+01   4.8019543772410e-08
-9.1919191919192e+00   -1.0000000000000e+01   6.2987546525226e-08
.
.
.
```

Listing 2. *Pliki typu plt*.dat zawierające kwadrat modułu funkcje falowe wraz ze współrzędnymi przestrzennymi*

```
#Funkcja falowa (wykres) utworzona z pliku 200plt0001.dat
#Kwadrat modułu |Psi(x,y)|^2
#Parametry sieci x: 2048 (-200,200), y: 2048 (-200,200), t: 24000 (0,150.796)
#
#x           y   norm(Psi(x,y))
-200       -200    1.00836e-14
-199.805   -200    9.97368e-19
-199.609   -200    6.12203e-17
-199.414   -200    3.41487e-15
-199.218   -200    3.39721e-15
.
.
.
```

Listing 3. *Pliki typu *.dat zawierające funkcje falowe wraz ze współrzędnymi przestrzennymi*

```
# norm= 1   n= 2
#   re                      im
3.290152556458731e-16       0
1.214861838381894e-14       0
2.351863031998781e-14       0
3.171370104506601e-14       0
.
.
.
8.532291688730832e-15       0
1.704650555501268e-14       0
1.328619602704041e-14       0
-2.000000000000000e+02
2.000000000000000e+02
-2.000000000000000e+02
2.000000000000000e+02
```

Listing 4. *Struktura plików typu *.par zawierających parametry sieci przestrzennej (oraz czasowej – ostatnia linia)*

```
<wersja pliku>
<nx> <x_min> <x_max>
<ny> <y_min> <y_max>
<nt> <t_min> <t_krok>
```

Parametry sieci w przypadku plików o rozszerzeniu *plt* będziemy wczytywać bezpośrednio z nich, natomiast dla pozostałych typów plików, parametry sieci wczytywać będziemy z osobnego pliku, którego struktura jest przedstawiona na listingu 4. Pliki te mają rozszerzenie *par*. Nas interesuje tylko trzecia i czwarta linia tego pliku, zawierające od lewej: ilość węzłów, minimalną i maksymalną wartość współrzędnych dla danej osi. Wiedząc już, z jakimi plikami będziemy mieć doczynienia możemy przystąpić do implementowania odpowiednich metod. Najpierw jednak, przygotujemy odpowiednie struktury, w których będziemy przechowywać informacje potrzebne Nam, w dalszej części tworzenia programu. Struktury te są umieszczone na listingach od 4 do 6. Na listingu 4 przedstawiona jest struktura `fpsi`, która będzie przechowywać wartość funkcji falowej w danym

punkcie przestrzeni (pole `psi_value`) oraz składowe koloru odpowiadającemu tej wartości (struktura `kolor`). Dzięki temu nie będziemy musieli, za każdym razem, gdy odświeżamy wyświetlany obraz, ponownie wyznaczać koloru dla danej wartości funkcji. Zaoszczędzi nam to czas potrzebny na kilka milionów wywoływań metody wyznaczającej kolor w oparciu o wartość funkcji falowej. Na listingu 5 przedstawiona jest struktura, która posłuży nam do przechowywania składowych *R*, *G* i *B* koloru. Ostatnią strukturą jest `FileInfo`, przechowująca informacje o wczytywanym pliku. Tak jak widać na listingu 6 struktura zawiera między innymi takie dane jak ilość haszy w nagłówku, informację czy plik zawiera parametry sieci (tylko pliki przedstawione na listingu 1) oraz strukturę `naglowek` zawierającą linie nagłówka z wczytywanego pliku. W dalszej części napiszemy metodę wyznaczającą odpowiednie wartości dla tych pól w zależności od wczytywanego pliku. Wszystkie struktury umieszczamy w pliku nagłówkowym *Unit1.h*. Musimy jeszcze zdefiniować odpowiednie typy dla dwuwymiarowej tablicy, w której przechowywać będziemy informacje o wartościach funkcji falowej oraz wskaźnik na tą tablicę. W tym celu do pliku nagłówkowego dodajemy jeszcze linie zaznaczone poniżej:

```
typedef fpsi* pfpsi;
class TForm1 : public TGLForm
{
...
private:      // User declarations
        fpsi **psi;
...
        GridParam *gp;
};
```

Jak wiemy w C++ (oraz C) nie ma tablic wielowymiarowych, są tylko „tablice tablic”. Właśnie `**psi` jest wskaźnikiem na tablicę, której elementami są wskaźniki na inne tablice zawierające elementy typu `fpsi`. Nasza dwuwymiarowa tablica będzie miała wymiar, jak nie trudno się domyślić, *nx*×*ny*, gdzie *nx* i *ny* są liczbą węzłów sieci dla odpowiednio osi OX i OY. Przygotowaniem pamięci dla tej tablicy zajmiemy się w dalszej części tego tutoriału. Wskaźnik `psi` inicjujemy wartością *NULL* w konstruktorze klasy *TForm1*. W przedostatniej linii kodu mamy jeszcze wskaźnik na obiekt klasy `GridParam`, który posłuży nam do przechowywania informacji o parametrach sieci. W konstruktorze klasy *TForm1* inicjujemy go wartością *NULL*. Nie będę tu omawiał klasy `GridParam`. Pliki (nagłówkowy oraz *cpp*) z tą klasą znajduje się w załączonym do tego tutoriału pliku zip (konieczne jest dodatnie pliku *cpp* do projektu). Jest ona dość intuicyjna, więc nie powinno być problemów ze zrozumieniem jej. Oczywiście musimy dodać odpowiedni plik nagłówkowy *gridparam.h* do *Unit1.h*. W tym momencie cały projekt powinien dać się bez problemów skompilować.

Listing 4. *Struktura `fpsi` przechowująca wartość funcji falowej w danym punkcie przestrzeni oraz odpowiadające jej składowe koloru*

```
typedef struct {
    float psi_value;
    rgb_triplet kolor;
} fpsi;
```

Listing 5. *Struktura `rgb_triplet` przechowująca składowe koloru dla danej wartości funkcji falowej*

```
typedef unsigned char cm_byte;
struct rgb_triplet {
    cm_byte r;
    cm_byte g;
    cm_byte b;
};
```

Listing 6. *Struktura `FileInfo` przechowująca dane o pliku*

```
typedef struct {
```

```
    int ilosc_haszy_w_naglowku;
    int ilosc_kolumn;
    bool zawiera_param_sieci;
    vector<AnsiString> naglowek;
} FileInfo;
```

Ogólny schemat naszego programu wygląda następująco:

1. określenie formatu pliku z funkcją falową (liczba linii nagłówka, liczba kolumn)
2. wczytanie parametrów sieci z pliku
3. określenie minimalnego oraz maksymalnego indeksu wiersza oraz kolumny tablicy, w których znajdują się dane z zakresu przestrzennego podanego przez użytkownika
4. wczytanie wartości funkcji falowej z pliku
5. wyznaczenie minimalnej oraz maksymalnej wartości funkcji falowej z wyznaczonego fragmentu tablicy
6. wyznaczenie składowych koloru odpowiadających danej wartości funkcji
7. wyświetlenie funkcji falowej z zadanego przez użytkownika przedziału

3. Wczytanie parametrów sieci z pliku

Przystąpimy teraz do napisania procedury wczytującej parametry sieci z pliku (klasa *GridParam* posiada metodę wczytującą parametry sieci z pliku, ale musi on mieć trochę inną strukturę niż ta przedstawiona na listingu 4). Nie zaczynamy, co prawda w ten sposób od punktu pierwszego naszego schematu, ale w tym przypadku nie ma to wielkiego znaczenia. A więc do pliku *Unit1.cpp* dodajemy następujący kod:

Listing 7. *Metoda* `TForm1::CzytajGrid(char *filename)` *wczytująca parametry sieci z pliku filename*

```
void __fastcall TForm1::CzytajGrid(char *filename)
{
    ifstream input(filename);
    double tmp;
    input>>tmp;
    double nx,xmin,xmax;
    double ny,ymin,ymax;
    input>>nx>>xmin>>xmax;
    input>>ny>>ymin>>ymax;

    gp = new GridParam(xmin, xmax, nx, ymin, ymax, ny, 0, 0, 0);
}
```

Dodajemy także deklarację tej metody do pliku nagłówkowego *Unit1.h*. Kod tej procedury jest tak prosty, że nie będę jej tutaj omawiać. Wspomnę może tylko o jej ostatniej linii. Tworzy ona obiekt klasy *GridParam*, który przechowuje parametry dwuwymiarowej sieci, a wskaźnik na ten obiekt przypisuje do `gp`. Kolejny etap to określenie formatu pliku z funkcją falową, czyli wyznaczenie liczby linii nagłówka i liczby kolumn (nasz program będzie potrafił wyświetlić dane z dowolnej kolumny znajdującej się w pliku). Jednak zanim przystąpimy do implementacji odpowiedniej metody, dodamy do projektu nową formę. Umieścimy na niej komponent *CSpinEdit* z palety *Samples*. Za pomocą tego komponentu użytkownik naszego programu będzie przekazywał informację o numerze kolumny, z której dane chce wyświetlić (później umieścimy także parę innych kontrolek umożliwiających sterowanie programem). Domyślnie będzie to ostatnia kolumna dla plików z

liczbą kolumn większą od dwóch i pierwsza dla pozostałych plików. A więc, z menu *File* wybieramy *New->Other* a następnie z zakładki *Dialogs* wybieramy *Standard Dialog (Vertical)*. Na naszą nową formę „wrzucamy" wcześniej wspomniany komponent. Dla tego komponentu ustawiamy w *Object Inspector* wartość pola `MinValue` na 1 (ponieważ zakładamy, że w pliku jest chociaż jedna kolumna). Nasza nowa forma jest tworzona w pliku projektu *Project1.cpp*:

```
Application->CreateForm(__classid(TOKRightDlg), &OKRightDlg);
```

My jednak usuwamy tą linię, ponieważ formę będziemy tworzyć w konstruktorze klasy *TForm1*. Modyfikujemy więc konstruktor klasy *TForm1* w sposób pokazany poniżej:

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TGLForm(Owner)
{
        debug_mode=true;
        NatezenieSwiatlaOtoczenia=0.3f;
        OKRightDlg = new TOKRightDlg(this);
}
```

Nowy *unit* zapisujemy pod nazwą *okcancel2*. Plik nagłówkowy *okcancel2.h* włączamy do sekcji include w pliku *Unit1.cpp*. Oczywiście skoro sami przydzielamy dla naszej nowej formy pamięć, sami ją też musimy zwolnić w odpowiednim miejscu. Dlatego też w zdarzeniu `OnClose` formy `Form1` wpisujemy następującą linię:

```
delete OKRightDlg;
```

Przy okazji dopiszemy także linię zwracającą pamięć do systemu po naszym obiekcie `gp`, dla którego pamięć przydzieliliśmy w metodzie `CzytajGrid()`:

```
delete OKRightDlg;
delete gp;
```

Teraz zajmiemy się implementacją metody określającej format pliku z funkcją falową. Metodę tą przedstawiam na listingu 8. Są w niej wykorzystane podstawowe wiadomości z zakresu programowania w C/C++, a celem tego tutorialu nie jest nauka programowania w C++, dlatego też nie będę jej omawiał. Poza tym jest ona dość dobrze opisana w postaci komentarzy. Metoda z listingu 8 potrafi wczytać parametry sieci bezpośrednio z pliku z funkcją falową, jeśli tylko plik ten ma format przedstawiony na listingu 1. Dodatkowo wykorzystuje ona metodę `ile_kolumn(char *linia)`, która na podstawie pierwszej linii z bloku zawierającego wartości funkcji potrafi wyznaczyć ilość kolumn w pliku. Przedstawiona jest ona na listingu 9.

Listing 8. *Metoda `TForm1::Info()` wyznaczająca format pliku z funkcją falową (mdz. in. liczbę kolumn i linii nagłówka)*

```
bool __fastcall TForm1::Info()
{
    int ilosc=0;
    char linia[255];
    ifstream in(OpenDialog->FileName.c_str());
    if (in.eof()) return 0; //brak danych w pliku: BłAD

    //czyścimy strukturę przechowująca linie nagłówka pliku
    InfoPliku.naglowek.clear();
    //wykonujemy pętle dopóki na początku linii znajduje się znak #
    for (;;) {
        if (!in.eof()) in.getline(linia,255);
            else break;
        if (linia[0]=='#') {
```

```
            ++ilosc;
            InfoPliku.naglowek.push_back(linia);
        } else break;

        //po wczytaniu drugiej linii sprawdzamy czy są w niej parametry sieci
        if (ilosc==2) {
            if (strstr(linia,"nx = "))
                InfoPliku.zawiera_param_sieci = 1;
            else
                InfoPliku.zawiera_param_sieci = 0;
        }
    }
    //określamy ilość kolumn w pliku
    InfoPliku.ilosc_kolumn = ile_kolumn(linia);
    //ustawiamy wartość MaxValue komponentu OKRightDlg->CSpinEdit1 na
    //ilość kolumn w pliku
    OKRightDlg->CSpinEdit1->MaxValue = InfoPliku.ilosc_kolumn;
    if (InfoPliku.ilosc_kolumn<=2)
        //jeżeli w pliku są dwie lub jedna kolumna to
        //ustawiamy wartość Value komponentu OKRightDlg->CSpinEdit1 na 1
        OKRightDlg->CSpinEdit1->Value = 1;
    else
        //w przeciwnym razie ustawiamy na OKRightDlg->CSpinEdit1->MaxValue
        OKRightDlg->CSpinEdit1->Value = OKRightDlg->CSpinEdit1->MaxValue;
    in.close(); //zamykamy plik

    InfoPliku.ilosc_haszy_w_naglowku = ilosc;
    return 1;   //OK
}
```

Listing 9. *Metoda `TForm1::ile_kolumn(char *linia)` wyznaczająca liczbę kolumn w pliku*

```
int __fastcall TForm1::ile_kolumn(char *linia)
{
    //ustawienie zmiennej określającej liczbę kolumn na 0
    int ile_kolumn=0;

    //wyznaczenie pierwszej pozycji w tablicy "linia", na której występuje
    //jakikolwiek znak z "1234567890-+" oraz przypisanie wskaźnika do tej pozycji do tp
    char *tp=strpbrk(linia,"1234567890-+");

    //dopóki tp jest różne NULL
    while (tp) {
        ++ile_kolumn; //zwiększamy liczbę kolumn o jeden

        //dopóki pod tp występuje jakikolwiek znak różny od spacji, tabulatora lub
        //znaku końca tablicy wykonujemy pętlę wstawiając za
```

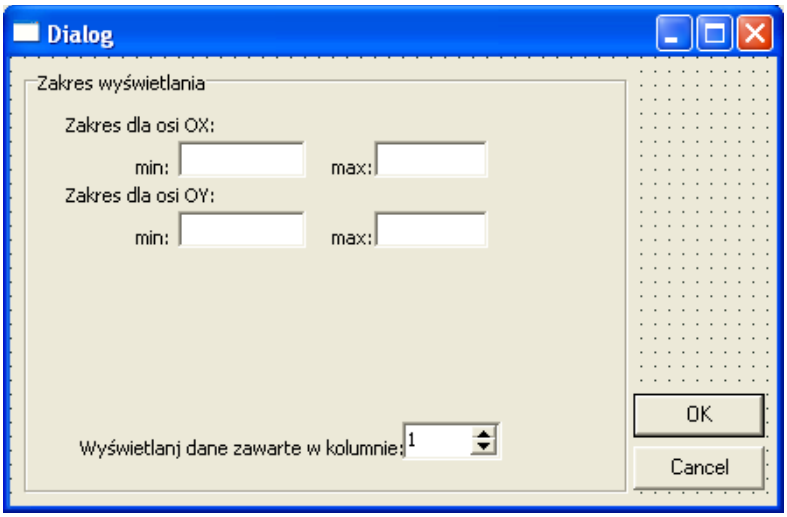
```
            // wartość spod tp znak spacji ' '
            for (;*tp!=' ' && *tp!='\t' && *tp!='\0'; tp++)
                *tp=' ';
            //strpbrk zwraca NULL, jeśli w linia nie występuje choć
            //jeden znak z "1234567890-+"
            //w przeciwnym razie zwraca wskaźnik na ten znak
            tp=strpbrk(linia,"1234567890-+");
        }

        return ile_kolumn;
    }
```

Metoda działa w ten sposób, że w ciągu znaków szuka pierwszego wystąpienia jednego ze znaków 1234567890-+ a następnie zamazuje wszystko aż do napotkania znaku spacji, tabulatora lub końca tablicy ze znakami (pętla `for`). Wcześniej następuje zwiększenie o jeden zmiennej zawierającej liczbę kolumn, ponieważ sam fakt wejścia do ciała pętli `while` oznacza, że w pliku jest przynajmniej jedna kolumna. Następnie metoda sprawdza czy w ciągu znaków występuje jeszcze jedna pozycja zawierająca jeden ze znaków 1234567890-+ (przypisanie `tp=strpbrk(linia,"1234567890-+")` w pętli `while`). Jeśli tak jest to kod w pętli `while` wykonuje się kolejny raz. Po dodaniu metody z listingu 9 do pliku *Unit1.cpp*, odpowiedniej deklaracji tej metody do pliku *Unit1.h* oraz struktury `InfoPliku` typu `FileInfo` do sekcji `private` cały projekt powinien się skompilować.

4. Wyznaczenie zakresu tablicy, w którym znajdują się dane do wyświetlania

Mając już przygotowane wszystkie niezbędne struktury oraz przedstawione powyżej dwie metody przystępujemy do zaimplementowania metody wyznaczającej minimalny oraz maksymalny indeks wiersza oraz kolumny tablicy, w których znajdują się dane do wyświetlenia, z zakresu przestrzennego podanego przez użytkownika. Najpierw jednak, do formy `OKRightDlg` dodajemy cztery komponenty `Edit`, umożliwiające użytkownikowi określenie przedziału przestrzennego, z którego ma być wyświetlona funkcja falowa. Nasza forma `OKRightDlg` w tym momencie powinna wyglądać mniej więcej jak na rysunku 2.



Rysunek 2
Okno umożliwiające zminę zakresu przestrzennego wyświetlanej funkcji falowej

Do sekcji private klasy *TForm1* dodajemy jeszcze osiem pól: `Xmin, Xmax, Ymin, Ymax` typu *double* oraz `imin, imax, jmin, jmax` typu *int*. Pierwsze cztery pola posłużą nam do przechowywania zakresu przestrzennego wyświetlanej funkcji falowej, natomiast kolejne cztery określają zakres dla wierszy (`imin,`

imax) oraz kolumn (jmin, jmax), w których znajdują się dane do wyświetlenia. W tym momencie możemy przystąpić do realizacji kolejnego punktu:

3. określenie minimalnego oraz maksymalnego indeksu wiersza (imin, imax) oraz kolumny (jmin, jmax) tablicy (psi), w których znajdują się dane z zakresu przestrzennego podanego przez użytkownika (Xmin, Xmax, Ymin, Ymax)

Metodę, która wykonuje ten punkt przedstawiam na listingu 10. Procedura ta nie jest zbyt skomplikowana. Do wyznaczenia odpowiednich wartości indeksów (imin, imax, jmin, jmax) wykorzystuje ona metody GetIndexY() oraz GetIndexX(), które są składowymi klasy GridParam. Metoda dba także o to, aby pola Xmin, Xmax, Ymin, Ymax miały odpowiednią wartość w przypadku, gdy użytkownik ustawi zbyt małą lub zbyt dużą wartość któregoś z parametrów wywołania X_min, X_max, Y_min, Y_max metody SetRange().

Listing 9. *Metoda TForm1::SetRange(double X_min, double X_max, double Y_min, double Y_max) wyznaczająca realizująca punkt 3*

```
void __fastcall TForm1::SetRange(double X_min, double X_max, double Y_min, double Y_max)
{
    Xmin = X_min; Xmax = X_max;
    Ymin = Y_min; Ymax = Y_max;

    //jeśli Xmin jest mniejsze od minimalnej wartości
    //sieci przestrzennej na osi OX przypisz Xmin  najmniejsza możliwa wartość
    if (Xmin<gp->GetXmin()) Xmin=gp->GetXmin();
    imin = gp->GetIndexX(Xmin); //wyznaczamy indeks odpowiadający wartości Xmin

    //jeśli Xmax jest większe od największej wartości
    //sieci przestrzennej na osi OX przypisz Xmax  największa możliwa wartość
    if (Xmax>gp->GetXmax()) Xmax=gp->GetXmax();
    imax = gp->GetIndexX(Xmax); //wyznaczamy indeks odpowiadający wartości Xmax

    //jeśli Ymin jest mniejsze od minimalnej wartości
    //sieci przestrzennej na osi OY przypisz Ymin  najmniejsza możliwa wartość
    if (Ymin<gp->GetYmin()) Ymin=gp->GetYmin();
    jmin = gp->GetIndexY(Ymin); //wyznaczamy indeks odpowiadający wartości Ymin

    //jeśli Ymax jest większe od największej wartości
    //sieci przestrzennej na osi OY przypisz Ymax  największa możliwa wartość
    if (Ymax>gp->GetYmax()) Ymax=gp->GetYmax();
    jmax = gp->GetIndexY(Ymax); //wyznaczamy indeks odpowiadający wartości Ymax

}
```

5. Wczytanie danych

I to by było na tyle, jeśli chodzi o punkt 3. Teraz przechodzimy do implementowania metody wykonującej kolejny punkt:

4. wczytanie wartości funkcji falowej z pliku do pamięci operacyjnej komputera

Metodę nazwiemy `CzytajPsi()`, będzie ona dość ogólną metodą tzn. będzie wczytywać dane z pliku o dowolnej liczbie linii nagłówka (zaczynających się od #) oraz będzie wczytywać dowolną, wybraną przez użytkownika, kolumnę z danymi do wyświetlenia. Dane przechowywać będziemy w dwuwymiarowej tablicy o rozmiarze *nx×ny* (gdzie jak już wspominałem, *nx* i *ny* są liczbą węzłów dla osi odpowiednio OX i OY). Dodajemy więc wskaźnik `**psi` do sekcji *private* klasy *TForm1*, wskazujący na dwuwymiarową tablicę, której elementami będą struktury typu `fpsi`.

```
private:
        fpsi **psi;
        .
        .
        .
```

Do formy *Form1* dodajemy jeszcze komponent *OpenDialog* z palet *Dialogs* i zmieniamy jago nazwę na `OpenDialog`. Komponent ten posłuży nam do wskazania pliku, z którego dane chcemy wyświetlić. Przystępujemy teraz do implementacji metody `CzytajPsi()` (listing 10). Na początku przydzielamy pamięć dla naszej dwuwymiarowej tablicy:

```
psi = new pfpsi[gp->GetNx()];
for(int i=0; i<gp->GetNx(); i++) psi[i]=new fpsi[gp->GetNy()];
```

Następnie musimy pobrać linie nagłówka z pliku, aby móc wczytać interesujące nas dane:

```
for (int i=0; i<InfoPliku.ilosc_haszy_w_naglowku; i++)
        input.getline(line,255);
```

Teraz możemy przystąpić do wczytywania danych z kolumny wskazanej przez użytkownika za pomocą `OKRightDlg->CSpinEdit1->Value`:

```
for (int j=0; j<gp->GetNy(); j++)
         for (int i=0; i<gp->GetNx(); i++){
             double tmp;
             //przechodzimy przez dane, których nie mamy wyświetlać
             for (int k=1; k<OKRightDlg->CSpinEdit1->Value; k++)
                 input>>tmp;
             //wczytujemy wartość z pożądanej kolumny
             input>>(psi[i][j].psi_value);
             //przechodzimy przez resztę kolumn
             for (int k=OKRightDlg->CSpinEdit1->Value+1;
                         k<=OKRightDlg->CSpinEdit1->MaxValue; k++) input>>tmp;
         }
```

Najpierw wczytujemy dane do zmiennej pomocniczej `tmp` z kolumn począwszy od pierwszej aż do kolumny, z której dane chcemy wyświetlić. Następnie wczytujemy wartości z interesującej nas kolumny do tablicy `psi`. Wartości z kolejnych kolumn wczytujemy ponownie do `tmp`. Kompletny kod tej metody przedstawiam na listingu 10.

Listing 10. *Metoda `CzytajPsi()` wczytująca dane z pliku.*

```
void __fastcall TForm1::CzytajPsi()
{
    Screen->Cursor=crAppStart; //ustawiamy typ kursora na klepsydrę
```

```
        //przygotowujemy pamięć dla tablicy 2D
        psi = new pfpsi[gp->GetNx()];
        for(int i=0; i<gp->GetNx(); i++) psi[i]=new fpsi[gp->GetNy()];

        //otwieramy plik o nazwie wskazanej przez OpenDialog->FileName
        ifstream input(OpenDialog->FileName.c_str());

        char line[255]; //przechowuje linię tekstu z nagłówka

        //pobieramy pierwsze linie w pliku zawierające #
        for (int i=0; i<InfoPliku.ilosc_haszy_w_naglowku; i++)
            input.getline(line,255);

        //wczytujemy dane do tablicy
        for (int j=0; j<gp->GetNy(); j++)
            for (int i=0; i<gp->GetNx(); i++){
                double tmp;
                //przechodzimy przez dane, których nie mamy wyświetlać
                for (int k=1; k<OKRightDlg->CSpinEdit1->Value; k++)
                    input>>tmp;
                //wczytujemy wartość z pożądanej kolumny
                input>>(psi[i][j].psi_value);
                //przechodzimy przez resztę kolumn
                for (int k=OKRightDlg->CSpinEdit1->Value+1;
                          k<=OKRightDlg->CSpinEdit1->MaxValue; k++) input>>tmp;
            }
        input.close();  //zamykamy plik
        Screen->Cursor=crDefault;  //ustawiamy kursor na domyślny
    }
```

Mając już metodę wczytującą dane z pliku do naszego programu, musimy ją gdzieś wywołać. Proponuję „dorzucić" do formy *Form1* komponent *PopupMenu* z palety *Standard*. To podręczne menu łączymy z naszą formą poprzez ustawienie własność *PopupMenu* w inspektorze obiektów na *PopupMenu1*. Oczywiście *PopupMenu1* nie zawiera żadnych opcji, dlatego też klikając prawym przyciskiem myszy na formie nie zobaczymy naszego menu. Jednak za chwilę ulegnie to zmianie. Jeżeli klikniemy dwa razy na naszym komponencie pojawi się dodatkowe okno. W oknie tym możemy zdefiniować nasze menu podręczne. Wpisujemy więc we własności *Caption*, przycisku widocznego w tym nowym oknie, tekst *Wczytaj*. Teraz musimy odpowiednio „zaprogramować" zdarzenie *OnClick* dla tego przycisku. Klikamy więc dwa razy na tym zdarzeniu i do metody, która nam się pojawiła w pliku *Unit1.cpp* wpisujemy kod z listingu 11. Metoda przedstawiona na tym listingu wywołuje funkcję `UstawParametrySieci()`, która wczytuje parametry sieci z pliku (domyślnie o nazwie *2d-dyn.par*), wykorzystując metodę z listingu 7 lub metodę `GetSetup()`, która pobiera dane bezpośrednio z pliku, z funkcją falową, jeżeli tylko plik ma format jak z listingu 1.

Listing 11. *Metoda `TForm1::Wczytaj1Click(TObject *Sender)` wywoływana w wyniku zdarzenia OnClick dla przycisku z menu podręcznego*

```
void __fastcall TForm1::Wczytaj1Click(TObject *Sender)
{
    //jeżeli wybrano plik
```

```
if (OpenDialog->Execute()){
        if (psi) { //jeżeli wczytano już wcześniej jakieś dane
            for(int i=0; i<gp->GetNx(); i++) { //zwolnij pamięć zajmowana
                                               //przez te dane
                delete[] psi[i];
                psi[i]=0;
            }
            delete[] psi;
            psi=0;
            delete gp; //zwolnij pamięć zajmowana przez parametry sieci
            gp=0;
        }
        //wyznaczamy format pliku z danymi
        Info();
        //wczytujemy parametry sieci
        UstawParametrySieci();

        //ustawiamy imin, imax, jmin oraz jmax
        SetRange(Xmin, Xmax, Ymin, Ymax);

        //czytamy dane z pliku, jeżeli zostały wczytane parametry sieci ()
        if (gp) CzytajPsi();
    }
}
```

Metoda `UstawParametrySieci()`, w razie braku pliku *2d-dyn.par* lub braku parametrów w pliku z funkcją falową, wyświetla formę `OKRightDlg` w celu podania parametrów sieci przez użytkownika. Jednak zanim będzie to możliwe, musimy dodać do formy `OKRightDlg` sześć komponentów *Edit* oraz dwa *Button*. Nazwy komponentów *Button* zmieniamy na *BEdytuj* oraz *BOpen*. Nasza forma `OKRightDlg` w tym momencie powinna wyglądać mniej więcej jak na rysunku 3. Pozostaje nam jeszcze zaimplementować odpowiednio metodę `UstawParametrySieci()`. Metoda ta jest przedstawiona na listingu 12. Korzysta ona z prostej funkcji `GetSetup()`, która pobiera dane bezpośrednio z pliku zawierającego funkcję falową. Nie będę jej tutaj omawiam, ponieważ, jak już pisałem, celem tego tutoriału nie jest nauka języka C++. Funkcję tą przedstawiam na listingu 13. Mając już zaimplementowane wszystkie te funkcje nasz projekt powinien się skompilować.

Rysunek 3
Okno umożliwiające zminę zakresu przestrzennego wyświetlanej funkcji falowej oraz parametrów sieci

Listing 12. *Metoda `TForm1:: UstawParametrySieci()` ustawiająca parametry sieci przestrzennej*

```
void __fastcall TForm1::UstawParametrySieci()
{
    if (InfoPliku.zawiera_param_sieci) //jeżeli plik z funkcja falowa zawiera
                                       //parametry sieci
        //czytaj parametry sieci bezpośrednio z pliku z funkcja falowa
        GetSetup(OpenDialog->FileName.c_str());
    //jeżeli istnieje plik parametrów sieci (2d-grid.par)
    else if (FileExists(ExtractFilePath(OpenDialog->FileName)+"2d-grid.par"))
{
        //czytaj parametry sieci z pliku 2d-grid.par
        CzytajGrid((ExtractFilePath(OpenDialog->FileName)+"2d-
grid.par").c_str());
    } else {
        //wyświetl komunikat, jeżeli brak parametrów sieci
        ShowMessage("Brak parametrow sieci w pliku "+
  ExtractFileName(OpenDialog->FileName) +
                    ". Podaj parametry sieci lub plik z parametrami sieci.");
        //pokaż formę OKRightDlg w celu podania parametrów sieci przez
użytkownika
        OKRightDlg->ShowModal();
    }
}
```

Listing 13. *Metoda `TForm1::GetSetup(char *out)` wczytująca parametry sieci z pliku zawierającego dane do wyświetlenia*

```
void __fastcall TForm1::GetSetup(char *out)
{
    ntyp dim=1;
    char temp[201];
    for(int i=0; i<201; i++) temp[i]='\0';
    ifstream plik(out);
    plik.getline(temp,200); //pobierz pierwszą linię komentarza
    plik.getline(temp,200); //pobierz drugą linię komentarza

    if (strstr(temp,"ny")) { //jeżeli w drugiej linii komentarza występuje
„ny”
        ++dim; //zwiększ wymiar przestrzenny o jeden
        if (strstr(temp,"nz")) //jeżeli w drugiej lini komentarza występuje
„nz”
++dim; //zwiększ wymiar przestrzenny o jeden
    }
    char *tp,
   l[10]; //przechowuje liczbę w postaci znakowej (char)
    ntyp n[6];  //przechowuje odczytane parametry sieci
```

```
        n[0]=n[1]=n[2]=n[3]=n[4]=n[5]=0;
        for(int i=0; i<dim*2; i++){
            int j=0;
            tp=strpbrk(temp,"1234567890"); //znajdź pierwsze wystąpienie jednego
ze
         //znaków "1234567890"
        //umieść pierwszą napotkaną liczbę w tablicy o nazwie „l” typu char
            while ((*tp!=' ') && (*tp!=',') && (*tp!=']')) {
                  l[j]=*tp;
               *tp=' ';
               ++j;
               ++tp;
            }
            l[j]='\0';
            n[i]=StrToInt(l); //zamień liczbę przechowywana w tablicy typu char na
int
        }
        delete gp;
        switch (dim) {
            case 1 : gp= new GridParam(-n[1],n[1],n[0],0,0,0,0,0,0); break;
            case 2 : gp= new GridParam(-n[2],n[2],n[0],-
n[3],n[3],n[1],0,0,0);break;
            case 3 : gp= new GridParam(-n[3],n[3],n[0],-n[4],n[4],n[1],-
n[5],n[5],n[2]);
                     break;
        }
        plik.close();
}
```

Jeżeli dysponujemy plikiem *2d-grid.par* albo plik z funkcją falową zawiera już parametry sieci, to możemy już wczytać interesujące nas dane (ale oczywiście nie zobaczymy jeszcze żadnego wykresu). W przeciwnym wypadku nie uda nam się niczego wczytać, ponieważ nie mamy odpowiednich metod w klasie *TOKRightDlg*, odpowiedzialnych za ustawienie parametrów sieci przez użytkownika. Tym właśnie, zajmiemy się teraz.

Najlepiej, jeśli klasę *TOKRightDlg* zaprzyjaźnimy z klasą *TForm1*, dzięki czemu w klasie *TOKRightDlg* będziemy mieli dostęp do prywatnych pól i metod klasy *TForm1*. Ułatwi nam to znacznie pracę. W tym celu do pliku nagłówkowego *Unit1.h* włączamy plik nagłówkowy *okcancel2.h* i dodajemy poniższy kod:

```
class TForm1 : public TGLForm
{
    friend class TOKRightDlg;
      ...
```

Musimy teraz zaimplementować zdarzenie *OnShow* dla formy *OKRightDlg*. Kod, który należy wpisać przedstawiam na listingu 14. W zasadzie nie robi on nic szczególnego. Jeżeli obiekt `Form1->gp` już istnieje, to przepisuje on parametry sieci w odpowiednie kontrolki, oraz wyłącza możliwość zmiany wartości w tych kontrolkach. Ustawia także, `CancelBtn->Enabled` na `true`. Umożliwi nam to rozpoznanie czy forma *OKRightDlg* została wyświetlona z powodu braku parametrów sieci, czy w wyniku wybrania z menu podręcznego opcji `Zakres`, którą teraz dodajemy. W zdarzeniu *OnClick* dla tej opcji wpisujemy poniższy kod:

```
OKRightDlg->ShowModal();
```

Metoda z listingu 14, korzysta z `ChangeEnable`. Jest to bardzo prosta metoda (listing 15), dzięki której nie będziemy musieli powtarzać zawartego w niej kodu w innych częściach programu.

Listing 14. *Metoda `TOKRightDlg::FormShow(TObject *Sender)` dla formy `OKRightDlg`*

```
void __fastcall TOKRightDlg::FormShow(TObject *Sender)
{
//jeżeli istnieje obiekt gp (oznacza to tyle, że udało się wczytać
// parametry sieci)
    if (Form1->gp) {
        Edit5->Text=FloatToStr(Form1->gp->GetXmin());
        Edit6->Text=FloatToStr(Form1->gp->GetXmax());
        Edit7->Text=IntToStr(Form1->gp->GetNx());
        Edit8->Text=FloatToStr(Form1->gp->GetYmin());
        Edit9->Text=FloatToStr(Form1->gp->GetYmax());
        Edit10->Text=IntToStr(Form1->gp->GetNy());
        ChangeEnable(false);
        BEdytuj->Enabled=true;
        CancelBtn->Enabled=true;
    }
    //w przeciwnym razie parametry sieci muszą zostać podane przez użytkownika
    else {
        ChangeEnable(true);
        BEdytuj->Enabled=false;
        CancelBtn->Enabled=false;
    }
}
```

Listing 15. *Metoda `TOKRightDlg:: ChangeEnable(bool)` dla zmieniająca własność `Enabled` komponentów wyświetlających parametry sieci*

```
void __fastcall TOKRightDlg::ChangeEnable(bool enable)
{
    Edit5->Enabled=enable;
    Edit6->Enabled=enable;
    Edit7->Enabled=enable;
    Edit8->Enabled=enable;
    Edit9->Enabled=enable;
    Edit10->Enabled=enable;
}
```

Zaimplementujemy teraz zdarzenia *OnClick* dla komponentów *BOpen* i *BEdytuj*. Zaczniemy od najprostszego, czyli *BEdytuj* (chociaż metoda dla *BOpen* też nie jest trudna). Wygenerowany automatycznie kod należy uzupełnić o poniższe instrukcje:

```
if (MessageBox(0,"Nieprawidłowe parametry sieci mogą spowodować niepoprawne "
               "działanie programu. Czy pomimo tego edytować parametry sieci?",
               "UWAGA !!!",MB_YESNO)==IDYES) {
   //umożliwiamy zmianę parametrów sieci na wyraźne życzenia użytkownika
```

```
        ChangeEnable(true);
        BEdytuj->Enabled = false;
    }
```

Kod dla zdarzenia *OnClick* komponentu *BOpen* zamieszczam na listingu 16. Metoda ta, w zależności od tego czy użytkownik zamierza zmienić wczytane już parametry sieci (wówczas własność `CancelBtn->Enabled` jest ustawiona na `true`), czy też parametry sieci mają być podane przez użytkownika (ze względu na bark pliku z parametrami sieci oraz brak tych parametrów w pliku z wczytywaną funkcją falową, wówczas `CancelBtn->Enabled=false`) wyświetla lub nie odpowiednie ostrzeżenie. Metoda wykorzystuje komponent *OpenDialog*, który musimy umieścić na formie `OKRightDlg` oraz zmienić jego nazwę na `ODGrid`, aby projekt dał się skompilować.

Listing 16. *Metoda `TOKRightDlg::BOpenClick(TObject *Sender)` wywoływana w wyniki kliknięcia komponentu BOpen*

```
void __fastcall TOKRightDlg::BOpenClick(TObject *Sender)
{
    switch (CancelBtn->Enabled) {
        //jeżeli parametry sieci są już wczytane
        case true  :  if (MessageBox(0,"Nieprawidłowe parametry sieci mogą
spowodować niepoprawne "
                          "działanie programu. Czy pomimo tego edytować
parametry
sieci ?",
                          "UWAGA !!!",MB_YESNO)==IDYES);
                      //przerywamy wczytywanie parametrów sieci jeśli
użytkownik
                      //wciśnie NIE
                      else break;
        //wczytujemy parametry sieci ze wskazanego przez użytkownika pliku
        case false :  if (ODGrid->Execute()) {
                            ChangeEnable(true);
                            BEdytuj->Enabled = false;
                            ifstream input(ODGrid->FileName.c_str());
                            double tmp;
                            input>>tmp;
                            double nx,xmin,xmax;
                            double ny,ymin,ymax;
                            input>>nx>>xmin>>xmax;
                            input>>ny>>ymin>>ymax;
                            Edit5->Text = FloatToStr(xmin);
                            Edit6->Text = FloatToStr(xmax);
                            Edit8->Text = FloatToStr(ymin);
                            Edit9->Text = FloatToStr(ymax);
                            Edit7->Text = FloatToStr(nx);
                            Edit10->Text = FloatToStr(ny);
                      };
    }
```

```
}
```

W tym momencie brakuje nam jeszcze zmiany wartości odpowiednich pól obiektu `gp` (lub ewentualnie utworzenie go) na te, podane przez użytkownika. Kod, który będzie to realizował umieszczamy w metodzie obsługującej zdarzenie *OnClick* dla komponentu *OKBtn*. Kod ten przedstawiam na listingu 17.

Listing 17. *Metoda `TOKRightDlg::OKBtnClick(TObject *Sender)` wywoływana w wyniki kliknięcia komponentu OKBtn*

```
void __fastcall TOKRightDlg::OKBtnClick(TObject *Sender)
{
    bool czytaj = false; //zakładamy, że nie będziemy musieli
                         //wczytywać ponownie danych

    if (!BEdytuj->Enabled) {//jeśli edytowane są parametry sieci
        //jeśli parametry sieci zostały wczytane, ale użytkownik zmienił
        //wartości nx i/lub ny
        if (Form1->gp && Form1->gp->GetNx()!=Edit7->Text.ToInt() &&
                         Form1->gp->GetNy()!=Edit10->Text.ToInt()) {
            //usuwamy wcześniej przydzielona pamięć dla tablicy przechowującej
dane
            for(int i=0; i<Form1->gp->GetNx(); i++) {
                delete[] Form1->psi[i];
                Form1->psi[i]=0;
            }
            delete[] Form1->psi;
            Form1->psi=0;
            czytaj = true; //ustawiamy czytaj na true, ponieważ będziemy
musieli
                           //ponownie wczytać dane dla innych wartości
parametrów
                           //nx i/lub ny
        } else Form1->gp = new GridParam(); //jeśli obiekt gp nie istnieje,
czyli
                                            //parametry sieci nie zostały
wczytane
                                            //to tworzymy obiekt gp
        //ustawiamy nowe parametry sieci
        Form1->gp->SetXRange(Edit5->Text.ToDouble(),Edit6->Text.ToDouble(),
Edit7->Text.ToDouble());
        Form1->gp->SetYRange(Edit8->Text.ToDouble(),Edit9->Text.ToDouble(),
Edit10->Text.ToDouble());
        ChangeEnable(false);
        BEdytuj->Enabled=true;
    }

    //jeżeli czytaj = false oznacza to, że nie było parametrów sieci i że
    //dane wczytamy wykonując pozostała część metody
```

```
    //TForm1::Wczytaj1Click(TObject *Sender)
    //jeżeli czytaj = true, parametry sieci były wczytane, ale zostały one
    // zmienione przez użytkownika w związku z czym musimy ponownie wczytać
dane
    if (czytaj){
        //zm_kolumny=false;
        Form1->Info();//parametry sieci już są (zostały podane przez
użytkownika)
        Form1->SetRange(Edit1->Text.ToDouble(),Edit2->Text.ToDouble(),
                        Edit3->Text.ToDouble(),Edit4->Text.ToDouble());
        Form1->CzytajPsi();

    } else
        //jeżeli przycisk CancelBtn jest dostępny, czyli jeżeli nie trzeba
        //wczytywać ponownie danych, ponieważ nie zmieniły się parametry sieci
        if (CancelBtn->Enabled) {
        //mógł się natomiast zmienić zakres wyświetlanej funkcji
            Form1->SetRange(Edit1->Text.ToDouble(),Edit2->Text.ToDouble(),
                            Edit3->Text.ToDouble(),Edit4->Text.ToDouble());
        }

    Close();
}
```

Metoda ta, rozpoznaje czy forma `OKRightDlg` została wyświetlona w wyniku braku parametrów sieci czy też na życzenie użytkownika po tym, jaką wartość ma wskaźnik `Form1->gp`. Jeżeli wskaźnik ten mam wartość NULL, oznacza to, że parametry sieci nie zostały wczytane. W przypadku, gdy forma została wyświetlona na życzenie użytkownika, możliwe jest zrezygnowanie ze zmiany jakichkolwiek parametrów poprzez wciśnięcie przycisku *Cancel*. W przypadku zmiany wczytanych już parametrów sieci (dokładniej, to zmiany wartości *nx* i/lub *ny*) metoda wywołuje funkcję `Form1->CzytajPsi()` w celu ponownego wczytania danych, ale tym razem dla innych parametrów sieci. Gdy brak było parametrów sieci, przeciwnie, metoda `Form1->CzytajPsi()` nie jest wywoływana w `TOKRightDlg::OKBtnClick(TObject *Sender)`, ponieważ wywoływana jest ona w metodzie `TForm1::Wczytaj1Click(TObject *Sender)`. Pod względem złożoności jest to, moim zdaniem, najbardziej złożona metoda w tym tutoriale, kolejne będą znacznie prostsze. Program powinien się teraz bez problemów skompilować. Jeśli chodzi o wczytywanie danych to pozostało nam jeszcze umożliwienie użytkownikowi wyboru kolumny, z której dane chce wyświetlić (poprzez ustawienie odpowiedniej wartości w komponencie `CSpinEdit1`). Dodajemy więc zmienną `bool zm_kolumny` do sekcji *private* klas *TOKRightDlg*. Umożliwi nam ona sprawdzenie czy użytkownik zmienił numer wyświetlanej kolumny. Jeżeli tak, to będziemy musieli wczytać ponownie dane z pliku. Zmienną tą ustawiamy na *false* za każdym razem, gdy zostaje wyświetlona forma `OKRightDlg`:

```
void __fastcall TOKRightDlg::FormShow(TObject *Sender)
{
    zm_kolumny = false;
    .
    .
    .
```

W metodzie obsługującej zdarzenie *OnChange* komponentu *CSpinEdit1* ustawiamy zmienną `zm_kolumny` na *true*. Musimy teraz, zmienić trochę metodę z listingu 17. Dochodzą nam dwie nowe sytuacje:

- nie udało się wczytać parametrów sieci, w związku z czym o podanie parametrów sieci został poproszony użytkownik. Przy okazji zmienił on domyślny numer kolumny, z której mają zostać wczytane dane
- forma *OKRightDlg*, została wyświetlona z inicjatywy użytkownika (w wyniku wybrania opcji *zakres* z menu podręcznego)

Musimy odróżnić w programie te dwie sytuacje, ponieważ w pierwszym przypadku dane zostaną wczytane w wyniku wykonania dalszej części metody `TForm1::Wczytaj1Click(TObject *Sender)`, co nie będzie miało miejsca w drugim przypadku. Aby odróżnić od siebie te dwie sytuacje, w metodzie z listingu 17 musimy wprowadzić dodatkową zmienną lokalną, która będzie przyjmowała wartość *false* dla pierwszej sytuacji i *true* dla drugiej. Metodę `TOKRightDlg::OKBtnClick(TObject *Sender)` już po wprowadzeniu odpowiednich zmian przedstawiam na listingu 18. Zmienna, o której była mowa, nazywa się `gp_istnieje`. Na listingu 18 zaznaczyłem zmiany, które należy wykonać w metodzie.

Listing 18. *Uaktualniona metoda `TOKRightDlg::OKBtnClick(TObject *Sender)` wywoływana w wyniki kliknięcia komponentu OKBtn*

```
void __fastcall TOKRightDlg::OKBtnClick(TObject *Sender)
{
    bool czytaj = false; //zakładamy, że nie będziemy musieli
                         //wczytywać ponownie danych

    bool gp_istnieje=false; //zakładamy, że parametry sieci nie zostały
wczytane
    if (Form1->gp) gp_istnieje = true; //weryfikujemy założenie powyżej
    if (!BEdytuj->Enabled) {//jeśli edytowane sa parametry sieci
        //jeśli parametry sieci zostały wczytane, ale użytkownik zmienił
        //wartości nx i/lub ny
        if (Form1->gp && Form1->gp->GetNx()!=Edit7->Text.ToInt() &&
                         Form1->gp->GetNy()!=Edit10->Text.ToInt()) {
            //usuwamy wcześniej przydzielona pamięć dla tablicy przechowującej
dane
            for(int i=0; i<Form1->gp->GetNx(); i++) {
                delete[] Form1->psi[i];
                Form1->psi[i]=0;
            }
            delete[] Form1->psi;
            Form1->psi=0;
            czytaj = true; //ustawiamy czytaj na true, ponieważ będziemy
musieli
                           //ponownie wczytać dane dla innych wartości
parametrów
                           //nx i/lub ny
        } else Form1->gp = new GridParam(); //jeśli obiekt gp nie istnieje,
czyli
                                            //parametry sieci nie zostały
wczytane
                                            //to tworzymy obiekt gp
        //ustawiamy nowe parametry sieci
        Form1->gp->SetXRange(Edit5->Text.ToDouble(),Edit6->Text.ToDouble(),
```

```
Edit7->Text.ToDouble());
        Form1->gp->SetYRange(Edit8->Text.ToDouble(),Edit9->Text.ToDouble(),
Edit10->Text.ToDouble());
        ChangeEnable(false);
        BEdytuj->Enabled=true;
    }

    //jeżeli czytaj = false oznacza to, że nie było parametrów sieci i że
    //dane wczytamy wykonując pozostała część metody
    //TForm1::Wczytaj1Click(TObject *Sender)
    //jeżeli czytaj = true, parametry sieci były wczytane
    //ale zostały one zmienione
    //przez użytkownika w związku z czym musimy ponownie wczytać dane
    if (czytaj || (zm_kolumny && gp_istnieje)){
          //ten kod należy wykonać tylko wtedy, gdy czytaj=true lub użytkownik
zmienił
   //wcześniej wczytane parametry sieci. Nawet, jeżeli zm_kolumny ma wartość
  //true, to jeżeli parametry sieci nie zostały wczytane automatycznie
  //z pliku (gp_istnieje=false) nie wykonujemy tej części kodu, ponieważ
  //zostanie on wykonane w dalszej części metody
  //TForm1::Wczytaj1Click(TObject *Sender)
        zm_kolumny=false;
        Form1->Info();//parametry sieci już są (zostały podane przez
użytkownika)
        Form1->SetRange(Edit1->Text.ToDouble(),Edit2->Text.ToDouble(),
                        Edit3->Text.ToDouble(),Edit4->Text.ToDouble());
        Form1->CzytajPsi();

    } else
        //jeżeli przycisk CancelBtn jest dostępny czyli jeżeli nie trzeba
        //wczytywać ponownie danych, ponieważ nie zmieniły się parametry sieci
        if (CancelBtn->Enabled) {
            //mógł się natomiast zmienić zakres wyświetlanej funkcji
            Form1->SetRange(Edit1->Text.ToDouble(),Edit2->Text.ToDouble(),
                            Edit3->Text.ToDouble(),Edit4->Text.ToDouble());
        }

    Close();
}
```

Aby zakończyć proces implementowania metod dla formy `OKRightDlg` dodajmy jeszcze do jej konstruktora kod przedstawiony poniżej:

```
Edit1->Text=FloatToStr(Form1->Xmin);
```

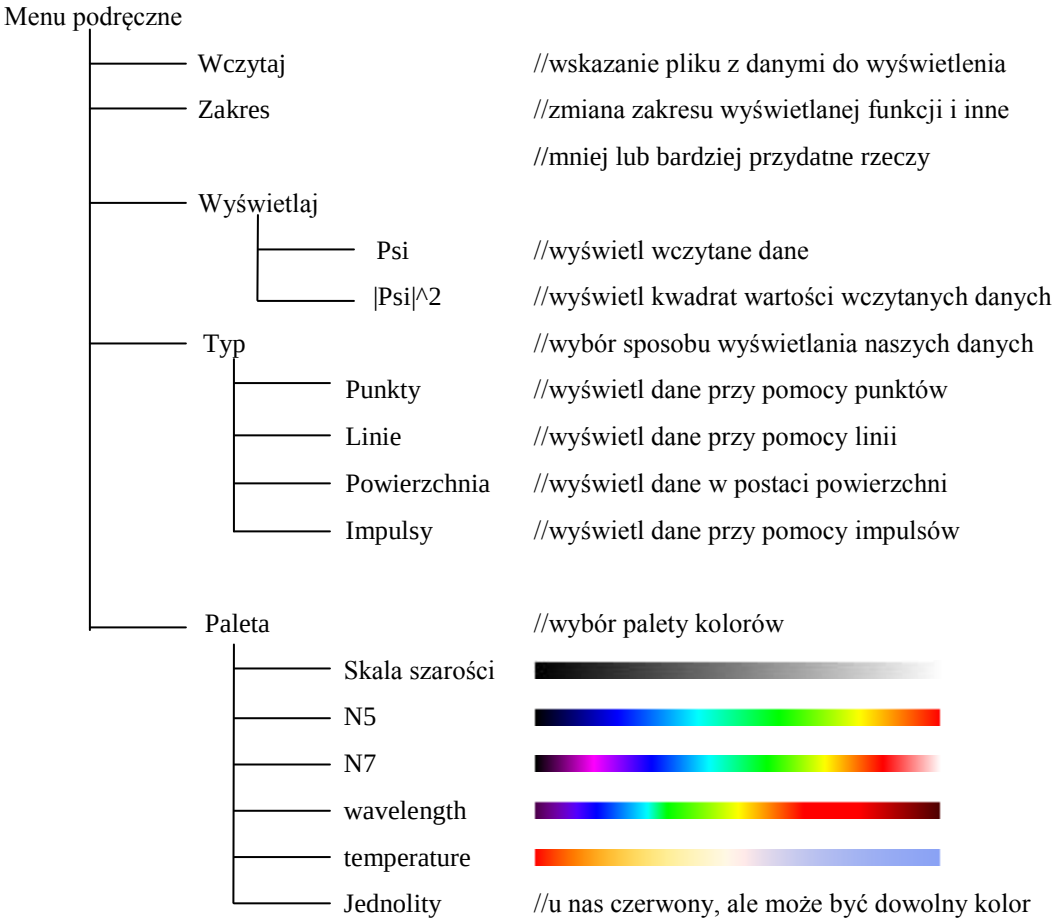
```
    Edit2->Text=FloatToStr(Form1->Xmax);
    Edit3->Text=FloatToStr(Form1->Ymin);
    Edit4->Text=FloatToStr(Form1->Ymax);
```

Dzięki temu, w momencie tworzenia formy do odpowiednich komponentów typu *Edit* zostaną wczytane aktualne wartości zakresu przestrzennego sieci. Zmiennym `Xmin`, `Xmax`, `Ymin`, `Ymax` najlepiej przypisać jakieś domyślne wartości w konstruktorze formy *Form1*, ale koniecznie przed utworzeniem obiektu `OKRightDlg`. Ja im przypisałem następujące wartości:

```
__fastcall TForm1::TForm1(TComponent* Owner)
       : TGLForm(Owner)
{
    Xmin = -12.0f;  Xmax =  12.0f;
    Ymin = -12.0f;  Ymax =  12.0f;
       .
       .
       .
```

W tym momencie mamy już wszystkie niezbędne metody, aby wczytać dane z pliku do naszego programu. Plik może mieć dowolną liczbę kolumn jak i ilość linii komentarza w nagłówku. Nie może mieć natomiast komentarzy w bloku zawierającym dane. Nie będziemy już także dodawać nowych metod do klasy *TOKRightDlg*, ale będziemy jeszcze modyfikować niektóre z już istniejących. Możemy więc zająć się bardziej przyjemnymi rzeczami, czyli przygotowaniem metod umożliwiających wyświetlenie naszych danych na monitorze (punkty 5-7 naszego schematu).

Zanim jednak, zaczniemy pisać jakikolwiek kod, dodamy do naszego menu podręcznego nowe opcje, między innymi takie jak *Wyświetlaj*, *Typ* czy *Paleta*. Pełen spis opcji, które należy dodać zamieszczam na grafie poniżej.



6. Wyznaczenie wartości minimalnej i maksymalnej

Dość dziwne nazwy palet *N5* i *N7* wyjaśnią się w dalszej części tego tutoriału. Teraz zajmijmy się punktem piątym:

5. wyznaczenie minimalnej oraz maksymalnej wartości funkcji falowej z wyznaczonego fragmentu tablicy

Minimalnej i maksymalnej wartości z naszych danych nie będziemy szukać w całym ich zakresie, ale tylko w przedziale wyznaczonym przez `imin`, `imax`, `jmin` i `jmax`. Przy okazji, od razu znajdziemy także minimalny i maksymalny kwadrat wartości (proszę zauważyć, że nie jest to kwadrat wartości, odpowiednio minimalnej i maksymalnej). Dzięki temu nie będziemy musieli przeszukiwać jeszcze raz tablicy, w przypadku, gdy użytkownik wybierze z menu podręcznego opcję *Wyświetlaj->|Psi|^2*. Dodajemy więc cztery dodatkowe pola do sekcji *private* klasy *TForm1*:

```
double minpsi, //wartość minimalna
       maxpsi; //wartość maksymalna
       _min,   //minimalny kwadrat wartości
       _max;   //maksymalny kwadrat wartości
```

Pola `minpsi` oraz `maxpsi` przechowywać będą, odpowiednio minimalną oraz maksymalną wartość z danego zakresu tablicy `psi`. Zmiennym `_min` i `_max` przypiszemy minimalny oraz maksymalny kwadrat wartości z tego samego zakresu tablicy `psi`. Fragment tablicy, w której musimy poszukiwać tych wartości jest wyznaczony przy pomocy metody `SetRange(double X_min, double X_max, double Y_min, double Y_max)`, składowej klasy *TForm1*.Odpowiednią metodę wyznaczającą te wartości przedstawiam na listingu 19.

Listing 19. *Metoda `TForm1::MinMax2D()` wyznaczająca wartości minimalne i maksymalne z pewnego przedziału tablicy 2D*

```
void __fastcall TForm1::MinMax2D()
{
    minpsi=maxpsi=psi[imin][jmin].psi_value;
    _min=_max=minpsi*minpsi;
    //szukamy w wierszach tablicy psi począwszy od imin aż do imax
    for (int i=imin; i<=imax; i++)
        //szukamy w kolumnach tablicy psi począwszy od jmin aż do jmax
        for (int j=jmin; j<=jmax; j++){
        //poszukujemy wartości minimalne oraz maksymalnej
              if (psi[i][j].psi_value<minpsi) minpsi=psi[i][j].psi_value;
              else if (psi[i][j].psi_value>maxpsi) maxpsi=psi[i][j].psi_value;

       //poszukujemy maksymalnego oraz minimalnego kwadratu wartości
           if (psi[i][j].psi_value*psi[i][j].psi_value<_min)
_min=psi[i][j].psi_value*psi[i][j].psi_value;
           else if (psi[i][j].psi_value*psi[i][j].psi_value>_max)
_max=psi[i][j].psi_value*psi[i][j].psi_value;
    }

}
```

Powyższa metoda jest bardzo prosta. Zakładamy w niej, że wartości, które poszukujemy znajdują się pod pozycją (`imin`, `jmin`). Następnie weryfikujemy nasze założenie przechodząc przez wyznaczony fragment tablicy. W przypadku znalezienia wartości mniejszej lub większej przypisujemy ją do odpowiednich zmiennych i kontynuujemy poszukiwania z tą wartością. `MinMax2D()` wywołujemy wewnątrz `Wczytaj1Click(TObject *Sender)` w miejscu wskazanym poniżej:

```
if (gp) {
    //czytamy dane z pliku jeżeli zostały wczytane parametry sieci ()
    CzytajPsi();

    //wyznaczamy wartości minimalna i maksymalna
    MinMax2D();
}
```

Wywołujemy ją również w metodzie `TOKRightDlg::OKBtnClick(TObject *Sender)`:

```
if (czytaj || (zm_kolumny && gp_istnieje)){
    zm_kolumny=false;
    Form1->Info();//parametry sieci już są (zostały podane przez użytkownika)
    Form1->SetRange(Edit1->Text.ToDouble(),Edit2->Text.ToDouble(),
                    Edit3->Text.ToDouble(),Edit4->Text.ToDouble());
    Form1->CzytajPsi();
    Form1->MinMax2D();
} else
```

```
    //jeżeli przycisk CancelBtn jest dostępny czyli jeżeli nie trzeba
    //wczytywać ponownie danych ponieważ nie zmieniły się parametry sieci
    if (CancelBtn->Enabled) {
        //mógł się natomiast zmienić zakres wyświetlanej funkcji
        Form1->SetRange(Edit1->Text.ToDouble(),Edit2->Text.ToDouble(),
                        Edit3->Text.ToDouble(),Edit4->Text.ToDouble());
        Form1->MinMax2D();
    }
```

Do klasy *TForm1* dodajemy także pole określające czy chcemy, wyświetlić wartości podniesione do kwadratu. Pole to o nazwie `mod_psi` będzie typu `bool`. Będzie ono równe 1 jeżeli chcemy aby zostały wyświetlone kwadraty wartości. Zmiennej tej nadajemy wartość 0 w konstruktorze klasy *TForm1*. Musimy jeszcze zaprogramować mechanizm zmiany tej wartości w zależności od opcji wybranej przez użytkownika. Klikamy więc dwa razy na opcji *Wyświetlaj->Psi* i do automatycznie wygenerowanego szkieletu metody dodajemy poniższy kod:

```
SetPsi(0);
```

Zadaniem `SetPsi` będzie ustawienie zmiennej `mod_psi` na `false` oraz ustawienie „ptaszka” przy opcji *Wyświetlaj->Psi*. Kod tej metody zamieszczam na listingu 20. Powinniśmy jeszcze ustawić własność *Checked* dla opcji *Wyświetlaj->Psi* na *true*, ponieważ w konstruktorze klasy *TForm1* przypisaliśmy zmiennej `mod_psi` wartość *false*. `SetPsi` wywołujemy również w metodzie obsługującej kliknięcie opcji *Wyświetlaj->|Psi|^2*, lecz w tym przypadku jako argument podajemy liczbę 1. Jak widać, dzięki `SetPsi` nie musimy powtarzać kodu w metodach obsługujących wybranie jednej z opcji *Wyświetl*.

Listing 20. *Metoda `TForm1::SetPsi(int mode)` przypisująca zmiennej `mod_psi` odpowiednią wartość*

```
void __fastcall TForm1::SetPsi(int mode)
{
    //przełącz własność Checked poprzednio wybranej opcji na false
    switch (mod_psi) {
        case 0 : Psi1->Checked = 0; break;
        case 1 : Psi21->Checked = 0; break;
    }
    mod_psi = mode; //przypisz zmiennej mod_psi wartość mode
    //przełącz własność Checked wybranej opcji na true
    switch (mod_psi) {
        case 0 : Psi1->Checked = 1; break;
        case 1 : Psi21->Checked = 1; break;
    }
    return;
}
```

7. Wyznaczenie składowych koloru

Zajmiemy się teraz wyznaczeniem składowych kolorów odpowiadających wartością z wyznaczonego zakresu tablicy *psi* (jest to punkt szósty naszego schematu programu). Do klasy *TForm1* dodajemy pole określające, którą paletę barw chcemy użyć:

```
class TForm1 : public TGLForm
```

```
{
...
private:
       short int paleta;
       .
       .
       .
```

Ponieważ nie możemy się już doczekać wyników w postaci wspaniałego wykresu będącego graficzną reprezentacją naszych danych, dlatego też zajmiemy się implementacją, najprostszej z możliwych palety kolorów – jednakowy kolor dla wszystkich naszych danych. Przygotowujemy więc metodę `rgb_triplet __fastcall TForm1::SetColorJednolity()`, która zwracać będzie kolor czerwony (oczywiście można wybrać dowolny inny). Zmienna `paleta` dla tej palety będzie miała wartość 5 (licząc od zera jest to piąta paleta na diagramie przedstawionym powyżej) i taką też wartością inicjujemy ją w konstruktorze klasy.

Listing 21. *Metoda `rgb_triplet __fastcall TForm1::SetColorJednolity()`, zwracająca strukturę przechowującą składowe R, G i B koloru czerwonego*

```
rgb_triplet __fastcall TForm1::SetColorJednolity()
{
    rgb_triplet rgb={255,0,0}; //R, G, B
    return rgb;
}
```

Myśląc o implementacji pozostałych palet przygotujemy jeszcze jedną metodę, która w zależności od wartości zmiennej `paleta` będzie wykonywała odpowiedni kod. Metoda ta, przedstawiam na listingu 22, jest na razie bardzo banalna, ale będzie ona rozbudowywana w miarę dodawania nowych palet do naszego programu. Dzięki niej, w przyszłości, nie będziemy musieli powtarzać sporej ilości kodu, w metodach implementujących nasze palety barw. Jako argumenty pobiera ona między innymi wartość minimalną, maksymalną (dzięki temu jest ona bardziej „elastyczna"). Na razie wartości te nie są wykorzystywane, lecz będą nam one potrzebne przy implementacji pozostałych palet.

Listing 22. *Metoda `rgb_triplet __fastcall TForm1::Kolor(double minpsi, double maxpsi, double psi)`, zwracająca odpowiedni kolor, wzależności od wybranej palety barw*

```
rgb_triplet __fastcall TForm1::Kolor(double minpsi, double maxpsi, double psi)
{
    switch (paleta) {
        case 5 : return SetColorJednolity(); //wybrano paletę jednobarwną
    }
}
```

Przy pomocy powyższej metody wyznaczymy kolory dla wszystkich wartości w tablicy `psi`, z zakresu (*imin:imax; jmin:jmax*). Sposób realizacji tego kroku demonstruję na listingu 23. Przy wyznaczaniu koloru dla danej wartość uwzględniana jest zmienna `mod_psi`. Oczywiście można by przechowywać kwadrat wartości od razu, w tablicy *psi*, jednak podejście takie sprawdzałoby się znacznie gorzej w momencie gdy użytkownik wybierałby na przemian opcję *Wyświetlaj->Psi* oraz *Wyświetlaj->|Psi|^2*. W takim przypadku istniałaby konieczność ponownego wczytywania danych przy każdej zmianie opcji na drugą (wyciągnięcie pierwiastka nic by nie dało, ponieważ nie potrafimy ustalić znaku), a wczytywanie danych z pliku jest najwolniejszym krokiem w procesie wyświetlenia danych na monitorze, w postaci wykresu.

Listing 23. *Metoda `TForm1::SetColor()`, wyznaczająca kolory dla wartości z zakresu (imin:imax; jmin:jmax) tablicy psi*

```
void __fastcall TForm1::SetColor()
{
    //wyznaczenie wartości minimalnej i maksymalnej z uwzględnieniem
    //wartości zmiennej mod_psi
    float min_value = mod_psi?_min:minpsi,
          max_value = mod_psi?_max:maxpsi;

    //dla wszystkich wartości w kolumnach z zakresu jmin:jmax
    for (int j=jmin; j<=jmax; j++)
        //dla wszystkich wartości w wierszach z zakresu imin:imax
        for (int i=imin; i<=imax; i++){
            //przypisz psi_value wartość spod psi[i][j].psi_value lub kwadrat
            //tej wartości jeśli mod_psi = true
            float psi_value = mod_psi?psi[i][j].psi_value*psi[i][j].psi_value:
psi[i][j].psi_value; //wsp. z-owa
            //wyznacz kolor odpowiadający wartości psi_value
            psi[i][j].kolor = Kolor(min_value, max_value, psi_value);
        }
}
```

Metodę `SetColor` wywołujemy zawsze po wywołaniu `SetRange` ponieważ musimy wyznaczyć ponownie kolory po każdej zmianie zakresu wyświetlanych danych. Oczywiście nie wywołujemy jej bezpośrednio po `SetRange` ponieważ najpierw musimy wyznaczyć wartości minimalną oraz maksymalną z danego zakresu tablicy za pomocą metody `MinMax2D()` (zgodnie z naszym schematem). Musimy więc, wywołać ją w metodzie `TOKRightDlg::OKBtnClick(TObject *Sender)` w następującym miejscu:

```
...
if (czytaj || (zm_kolumny && gp_istnieje)){
    zm_kolumny=false;
    Form1->Info();//parametry sieci już sa (zostały podane przez użytkownika)
    Form1->SetRange(Edit1->Text.ToDouble(),Edit2->Text.ToDouble(),
                    Edit3->Text.ToDouble(),Edit4->Text.ToDouble());
    Form1->CzytajPsi();
    Form1->MinMax2D();
    Form1->SetColor();
} else
    //jeżeli przycisk CancelBtn jest dostępny czyli jeżeli nie trzeba
    //wczytywać ponownie danych ponieważ nie zmieniły się parametry sieci
    if (CancelBtn->Enabled) {
        //mógł się natomiast zmienić zakres wyświetlanej funkcji
        Form1->SetRange(Edit1->Text.ToDouble(),Edit2->Text.ToDouble(),
                        Edit3->Text.ToDouble(),Edit4->Text.ToDouble());
        Form1->MinMax2D();
        Form1->SetColor();
```

```
}
...
```

A także w metodzie `TForm1::Wczytaj1Click(TObject *Sender)`:

```
...
if (gp) {
    //czytamy dane z pliku jeżeli zostały wczytane parametry sieci ()
    CzytajPsi();

    //wyznaczamy wartości minimalna i maksymalna
    MinMax2D();

    //wyznaczamy kolory dla wartości z wcześniej wyznaczonego
    //przedziału tablicy psi
    SetColor();
}
...
```

Ostatnimi metodami, w których musimy wywołać `SetColor` są metody obsługujące kliknięcie opcji *Wyświetlaj->Psi* oraz *Wyświetlaj->|Psi|^2*. Jest to konieczne, ponieważ za każdym razem, gdy wybieramy jedną z tych opcji, zmienia się przedział wartości naszych danych. `SetColor` w tych metodach wywołujemy zaraz po wywołaniu metody `SetPsi`.

8. Wyświetlenie danych w postaci wykresu

Mając już metodę `SetColor`, szybko przeskakujemy do ostatniego punktu naszego schematu. Tutaj także, zaimplementujemy najprostszy sposób wyświetlenia naszych danych, tak aby jak najszybciej zobaczyć wykres. Nasze dane wyświetlimy w postaci punktów. Podobnie jak dla palety kolorów, wprowadzamy zmienną `short int typ`, która będzie informowała jaki styl wyświetlania danych (punkty, linie, itd.) wybrał użytkownik. Implementujemy także metodę `SetTyp` (listing 24), która będzie wykonywała podobne czynności jak `SetPsi`, lecz tym razem dla innej zmiennej i opcji. Zmiennej `typ` przypisujemy wartość 0 w konstruktorze klasy, ponieważ opcja *Punkty* jest, licząc od zera, na tej właśnie pozycji (ten sposób wyświetlania naszych danych będzie ustawiony jako domyślny) .

Listing 24. *Metoda `TForm1::SetTyp()`, przypisująca zmiennej `typ` odpowiednią wartość*

```
void __fastcall TForm1::SetTyp(int wybrany_typ)
{
    //przełacz własność Checked poprzednio wybranej opcji na false
    switch (typ) {
        case 0 : Punkty1->Checked = 0; break;
        case 1 : Linie1->Checked = 0; break;
        case 2 : Powierzchnia1->Checked = 0; break;
        case 3 : Impulsy1->Checked = 0; break;
    }
    typ = wybrany_typ; //przypisz zmiennej mod_psi wartość mode
    //przełacz własność Checked wybranej opcji na true
```

```
        switch (typ) {
            case 0 : Punkty1->Checked = 1; break;
            case 1 : Linie1->Checked = 1; break;
            case 2 : Powierzchnia1->Checked = 1; break;
            case 3 : Impulsy1->Checked = 1; break;
        }
        return;
    }
```

Do pełnego sukcesu pozostało nam jeszcze przygotowanie metody wyświetlającej nasze dane na monitorze w postaci jednokolorowych punktów. Metodę, wykonującą to zadanie przedstawiam na listingu 25. Wyświetla ona wszystkie wartości z zakresu (*imin:imax; jmin:jmax*) tablicy `psi` w postaci punktów, uwzględniając wartość zmiennej `mod_psi`. Ponieważ nie przetrzymujemy w pamięci współrzędnych *x* i *y* a jedynie *z*, metoda z listingu 25 wyznacza współrzędne (*x, y*) w oparciu o parametry sieci oraz numer kolumny i wiersza, z którego pochodzi wartość przeznaczona do wyświetlenia. Takie podejście zaoszczędza nam sporej ilości pamięci operacyjnej. Współrzędne (*x, y, z*) przechowywane są w lokalnej tablicy *pkt1[3]*. Do wyświetlenia punktu na monitorze użyłem funkcji *glVertex3fv*, która jako parametr pobiera właśnie wskaźnik na tablicę *pkt1[3]*.

Listing 25. *Metoda `TForm1::RysujPsiPunkty()`, wykreślająca nasze dane na monitorze w postaci punktów*

```
void __fastcall TForm1::RysujPsiPunkty()
{
    //każdy vertex ma być traktowany jako punkt
    glBegin(GL_POINTS);
    //dla wsz
    for (int j=jmin; j<=jmax; j++) {
        float pkt1[3]; //tablica przechowująca współrzędne x, y, z
        pkt1[1]=gp->GetYmin()+j*gp->GetDy();  //wyznaczenie wsp. y
        for (int i=imin; i<=imax; i++){
            pkt1[0]=gp->GetXmin()+i*gp->GetDx();  //wyznaczenie wsp. x
            //wyznaczenie wsp. z, z uwzględnieniem wartości zmiennej mod_psi
             pkt1[2]=mod_psi?psi[i][j].psi_value*psi[i][j].psi_value:
psi[i][j].psi_value;
        //ustawienie odpowiedniego koloru rysowania
            glColor3ub(psi[i][j].kolor.r,psi[i][j].kolor.g,psi[i][j].kolor.b);
        //wyrysowanie punktu
            glVertex3fv(pkt1);
         }
    }
    glEnd();
}
```

Mając już powyższą metodę musimy wywołać *SetTyp* wewnątrz metody obsługującej wybór (kliknięcie) opcji *Typ->Punkty*. Argumentem wywołania metody *SetTyp* jest liczba 0 (z przyczyn podanych wyżej). Tak naprawdę to w tym momencie zmienna typ nie jest nam do niczego potrzebna, ale przyda się ona, gdy zaimplementujemy kolejny sposób wyświetlania naszych danych.

Nareszcie możemy wyświetlić nasz dane. W tym celu wywołujemy metodę `RysujPsiPunkty()` wewnątrz `TForm1::RysujScene()`:

```
void __fastcall TForm1::RysujScene()
```

```
{
	//przeksztalcenia macierzy model-widok i rysowanie figur

	//jeżeli udało się wczytać parametry sieci
	//wyświetl dane na monitorze
	if (gp) RysujPsiPunkty();
}
```

Zabezpieczenie przed brakiem parametrów sieci jest konieczne ponieważ `RysujScene()` jest wywoływana pomimo niepowodzenia we wczytaniu parametrów sieci wewnątrz metody `TGLForm::GL_RysujScene()`. Brak tego zabezpieczenia powodowałby próbę odczytania parametrów sieci z nieistniejącego obiektu `gp`. Możemy teraz uruchomić nasz program i wczytać jakieś dane (np. z pliku dostarczonego razem z tutoriałem). Wczytujemy więc nasze dane i co widzimy. Piękny, złożony z punktów wykres naszych danych. Dziwne jest tylko, że wszystkie punkty leżą na płaszczyźnie. Wygląda to tak jakby dane miały tą samą wartość w każdym punkcie. Zapewniam jednak, że tak nie jest. Powodem tego, że nie widzimy żadnych struktur, a jedynie płaszczyznę składającą się z punktów, są zbyt małe wartości danych (rzędu 10^{-2}) w porównaniu z rozmiarami sieci. Aby móc zobaczyć nasz wykres, musimy przeskalować nasze dane do jakiejś większej liczby. My zrobimy to w taki sposób, aby użytkownik sam mógł wybrać do jakiej liczby chce skalować wykres. Komponenty umożliwiające ten wybór umieścimy na nowej formie. Dodajemy więc nową formę do projektu podobnie jak dodawaliśmy *OKRightDlg*. Tym razem zamiast *Standard Dialog (Vertical)* wybieramy *Form* z zakładki *New*. Zmieniamy nazwę naszej nowej formy na *FormSettings* i zapisujemy ją pod nazwą *settings*. Podobnie jak *OKRightDlg* czynimy ją zaprzyjaźnioną z naszą klasą *TForm1*:

```
#include "settings.h"
class TForm1 : public TGLForm
{
    friend class TOKRightDlg;
    friend class TFormSettings;
```

Usuwamy także, kod tworzący obiekt tej klasy z *Project1.cpp*:

```
	try
	{
		Application->Initialize();
		Application->CreateForm(__classid(TForm1), &Form1);
		Application->CreateForm(__classid(TFormSettings), &FormSettings);
```

Tworzymy go sami w konstruktorze klasy *TForm1*.

```
    OKRightDlg = new TOKRightDlg(this);
    FormSettings = new TFormSettings(this);
```

Nie zapominamy także o usunięciu go z pamięci w *FormClose* klasy *TForm1*. Dodajemy również nową opcję *Ustawienia* do naszego menu podręcznego umożliwiającą wyświetlenie tej formy. W metodzie obsługującej wybranie tej opcji wywołujemy *FormSettings->Show()*. Teraz do naszej nowej formy dodajemy komponenty *TrackBar* oraz *Label*, których używać będziemy do zmiany oraz wyświetlania wartości, do której skalowany jest nasz wykres (ustawionej za pomocą *TrackBar*). Nazwę komponentu *TrackBar* zmieniamy na *TBSkala*. Maksymalną, możliwą do ustawienia wartość dla *TBSkala*, ustawiamy w *Object Inspector* na 50, natomiast minimalną na 1. Dodajemy także zmienną `double skala` do sekcji *private* klasy *TForm1*. Zmienną tą inicjujemy wartością 6 w konstruktorze tej klasy (przed utworzeniem obiektu klasy *TFormSettings*). W metodzie obsługującej zdarzenie *OnCreate* dla naszej nowej formy wpisujemy kod ustawiający pozycję suwaka komponentu *TBSkala* na wartość zmiennej skala (listing 26).

Listing 25. *Metoda `TFormSettings::FormCreate(TObject *Sender)`, ustawiająca suwak komponentu* TBSkala *na odpowiedniej pozycji*

```
void __fastcall TFormSettings::FormCreate(TObject *Sender)
```

```
{
    TBSkala->Position=Form1->skala;
    Label5->Caption=IntToStr(TBSkala->Position);
}
```

Musimy jeszcze dodać kod zmieniający wartość zmiennej skala podczas zmiany pozycji suwaka *TBSkala*. W tym celu uzupełniamy metodę obsługującą zdarzenie *OnChange* dla komponentu *TBSkala* o poniższy kod:

```
//ustawiamy wartość zmiennej skala na wskazana przez TBSkala
Form1->skala = TBSkala->Position;
//wyświetlamy wybrana wartość za pomoca komponentu Label
Label2->Caption=IntToStr(TBSkala->Position);

if (CBPodglad->Checked)
//wykonujemy GL_RysujScene() aby zobaczyć wykres po zmianie wartości
//zmiennej TBSkala
    Form1->GL_RysujScene();
```

Komponent `CBPodglad` dodałem w celu umożliwienia użytkownikowi wyboru czy chce ujrzeć od razu wynik zmiany wartości zmiennej `skala` czy też dopiero po zamknięciu formy *Settings*. Podejście takie wymaga jednak wywołania `Form1->GL_RysujScene()`, w metodzie obsługującej zdarzenie *OnClose* formy. Jest to jednak zalecane, ponieważ w przypadku wyświetlania dużej ilości danych użytkownik może ustawić wszystkie możliwe parametry (u nas na razie tylko jeden) i dopiero wtedy wyświetlić dane (dzięki temu nie musi on czekać aż zakończą się obliczenia związane ze zmianą wartości jakiegoś parametru po to aby zmienić wartość następnego).

Mając już mechanizm umożliwiający zmianę wartości, do której skalowany jest wykres, przystępujemy do zaimplementowania samego skalowania wykresu. Najpierw dodajemy do klasy *TForm1* zmienną `double wsp`, która „mówić" nam będzie przez ile należ pomnożyć największą wartość, co do wartości bezwzględnej, aby była ona równa zmiennej `skala`. Wszystkie pozostałe wartości, również mnożyć będziemy przez tą wartość zmiennej *wsp*. Przy wyznaczaniu wartości zmiennej *wsp* uwzględniamy oczywiście zmienna *mod_psi*. Kod wyznaczający wartość *wsp* zamieszczam na listingu 26. Metoda z tego listingu wyznacza również wartości zmiennych *min_value* oraz *max_value* typu *double*. Zmienne te przechowują minimalną oraz maksymalną wartość naszych danych uwzględniając wartości *mod_psi* oraz *skala*. Deklaracje tych zmiennych dodajemy do sekcji *private* klasy *TForm1*.

Listing 26. *Metoda `TForm1::SetSkala()`, wyznaczająca wartości zmiennych wsp, min_value oraz max_value*

```
void __fastcall TForm1::SetSkala()
{
    //jeżeli mod_psi = false
    if (!mod_psi) {
        //jeżeli minpsi!=0 i |minpis| jest większa od |maxpsi|
        if (minpsi && (fabs(minpsi)>fabs(maxpsi))) wsp=skala/fabs(minpsi);
        //jeżeli maxpsi!=0 i |maxpis| jest większa badź równa |minpsi|
        else if (maxpsi && (fabs(minpsi)<=fabs(maxpsi)))
wsp=skala/fabs(maxpsi);
        //jeżeli minpsi i maxpsi sa równe 0
        else wsp=0.0;
    } else { //jeżeli mod_psi  = true
        //jeżeli _max jest różne od zera
        if (_max)
            wsp=skala/_max;
```

```
            //w przeciwnym razie
            else wsp=0.0;
        }
        //wyznaczamy wartości minimalna oraz maksymalna uwzględniając
        //zmienne mod_psi oraz wsp (skala)
        min_value=(mod_psi?_min:minpsi)*wsp;
        max_value=(mod_psi?_max:maxpsi)*wsp;
    }
```

Metodę *SetScala* wywołujemy za każdym razem, gdy wyznaczamy wartości minimalną oraz maksymalną, dlatego też jej wywołanie umieszczamy na samym końcu metody *MinMax2D*. Wywołujemy ją także przy każdej zmianie wartości zmiennej `skala`:

```
void __fastcall TFormSettings::TBSkalaChange(TObject *Sender)
{
    //ustawiamy wartość zmiennej skala na wskazana przez TBSkala
    Form1->skala = TBSkala->Position;

    //wyznaczamy wartość współczynnika skalującego
    Form1->SetSkala();
...
```

oraz na końcu metody `SetPsi` obsługującej zmianę opcji *Wyświetlaj->Psi* na *Wyświetlaj->|Psi|^2* i odwrotnie.

Znając już wartość współczynnika skalującego, musimy zmienić odrobinę kod metody *SetColor*. W tym momencie mając już zmienne które określają maksymalną oraz minimalną wartość wykresu, uwzględniające wartości *skala* oraz *mod_psi* usuwamy z niej deklaracje tych zmiennych oraz mnożymy przez *wsp* współrzędną *z*:

```
void __fastcall TForm1::SetColor()
{
    //wyznaczenie wartości minimalnej i maksymalnej z uwzględnieniem
    //wartości zmiennej mod_psi
    float min_value = mod_psi?_min:minpsi,
          max_value = mod_psi?_max:maxpsi;
...
     float psi_value = mod_psi?psi[i][j].psi_value*psi[i][j].psi_value:
                                                    psi[i][j].psi_value; //wsp. z
     psi_value*=wsp;
...
```

Pozostało na jeszcze uwzględnić nasze skalowanie podczas wyświetlania danych. Wyznaczając współrzędną *z* musimy pomnożyć ją przez *wsp*:

```
void __fastcall TForm1::RysujPsiPunkty()
{
    ...
    for (int j=jmin; j<=jmax; j++) {
        float pkt1[3]; //tablica przechowujaca współrzędne x, y, z
        pkt1[1]=gp->GetYmin()+j*gp->GetDy();//wyznaczenie wsp. y
```

```
        for (int i=imin; i<=imax; i++){
            pkt1[0]=gp->GetXmin()+i*gp->GetDx();  //wyznaczenie wsp. x
            //wyznaczenie wsp. z, z uwzględnieniem wartości zmiennej mod_psi
             pkt1[2]=mod_psi?psi[i][j].psi_value*psi[i][j].psi_value:
                                                    psi[i][j].psi_value;
             pkt1[2]*=wsp;

            //ustawienie odpowiedniego koloru rysowania
            glColor3ub(psi[i][j].kolor.r,psi[i][j].kolor.g,psi[i][j].kolor.b);
    ...
```

W tym momencie, po skompilowaniu naszego programu i wczytaniu do niego naszych danych powinniśmy ujrzeć wykres w całej jego okazałości. Oglądając nasz wykres, zwłaszcza przy dużym zakresie wyświetlanych danych można zauważyć, że urywa się w pewnym momencie. Spowodowane jest to zbyt małą wartością ostatniego parametru metody *glFrustrum*. Proponuję ustawić tą wartość na 1000, dzięki czemu będziemy mogli podziwiać cały nasz wykres a nie tylko fragment.

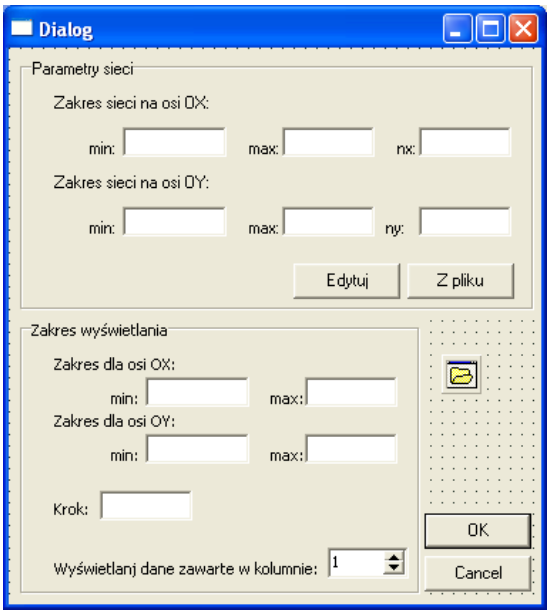
Udało się nam więc wyświetlić dane na monitorze komputera za pomocą *OpenGL*. Ponieważ ilość danych nie jest zbyt duża, możliwe jest łatwe manipulowanie wykresem (obracanie, oddalanie itp.) Jednak, gdyby ilość danych w danym zakresie sieci, była znacznie większa manipulowanie wykresem byłoby utrudnione. Przydałaby się więc możliwość wyświetlania danych z danego zakresu z pewnym, ustalonym przez użytkownika krokiem. Zajmiemy się teraz dodaniem takiej możliwości do naszego programu. Sterowanie wielkością kroku będzie możliwe za pomocą komponentu *Edit*, który umieścimy na formie *OKRightDlg* (rysunek 4). Zmienną `int every` przechowującą wartość kroku deklarujemy w klasie *TForm1*. Zmiennej tej nadajemy wartość 1 w konstruktorze tejże klasy (koniecznie przed utworzeniem obiektu klasy `TOKRightDlg`, ponieważ wartość tej zmiennej będzie przepisywana do kontrolki umieszczonej na tej formie). Musimy jeszcze zadbać o zmianę jej wartości na tą podaną przez użytkownika. W tym celu umieszczamy w metodzie `TOKRightDlg::OKBtnClick(TObject *Sender)` kod przypisujący zmiennej `every` wartość wpisaną do *Edit11*:

```
void __fastcall TOKRightDlg::OKBtnClick(TObject *Sender)
{
    bool czytaj = false; //zakł., że nie będziemy musieli wczytywać ponownie danych

    Form1->every=StrToFloat(Edit11->Text);
...
```

Aby podczas pierwszego wyświetlenia `OKRightDlg` zobaczyć obecną wartość tej zmiennej, umieszczamy kod `Edit11->Text=FloatToStr(Form1->every);` w konstruktorze tej formy.

Rysunek 4
Okno umożliwiające zminę zakresu przestrzennego wyświetlanej funkcji falowej oraz parametrów sieci po dodaniu kontrolki umozliwiającej zmianę kroku wyświetlanych danych



Musimy jeszcze zmienić kod metod *SetColor* (tak aby określanie kolorów odbywało się tylko dla wyświetlanych wartości) oraz *RysujPsiPunkty* (aby wyświetlane były dane z uwzględnieniem wartości kroku). Zmiany te są bardzo proste. Zamiast zwiększać o jeden zmienne iteracyjne *i* i *j* dodajemy do nich wartość zmiennej *every*:

```
for (int j=jmin; j<=jmax; j+=every) {
    ...
    for (int i=imin; i<=imax; i+=every){
```

Teraz możemy już wyświetlać dane z dowolnego zakresu używając dowolnej wartości kroku. Proponuję teraz pobawić się trochę naszym programem i zobaczyć czy wszystko dobrze funkcjonuje. Przede wszystkim proponuję zmianę zakresu wyświetlanych danych na np. $x \in [-50; -40]$ i $y \in [-50; -40]$. Dla takiego zakresu nic nie jest wyświetlane na monitorze. Tak naprawdę to jest, ale bardzo daleko od środka okna (które ma współrzędne [0,0,0]). Manipulując odpowiednio wykresem można go tak ustawić aby zobaczyć te dane. Powinniśmy jednak tak zmienić współrzędne x i y wyświetlanych danych aby środek przedziału znajdował się dokładnie w środku naszego okna. Nie jest to zadanie wcale trudne. Po prostu obliczamy współrzędne x_{sr} i y_{sr} środka zakresu wyświetlanych danych. Następnie od współrzędnych x i y wyświetlanych wartości odejmujemy współrzędne, odpowiednio x_{sr} i y_{sr}. Wyznaczeniem współrzędnych x_{sr} i y_{sr} zajmiemy się w metodzie `TForm1::SetRange(double X_min, double X_max, double Y_min, double Y_max)` dodając na końcu następujący kod (oczywiście trzeba mieć zadeklarowane zmienne `x_sr` i `y_sr`):

```
//wyznaczamy środek przedziału na osi OX
x_sr = (Xmax+Xmin)/2.0;
//wyznaczamy środek przedziału na osi OY
y_sr = (Ymax+Ymin)/2.0;
```

Współrzędne `x_sr` i `y_sr` odejmujemy od współrzędnych odpowiednio x i y wyświetlanych danych (listing 27).

Listing 27. *Metoda `TForm1::RysujPsiPunkty()`, uwzględniająca współrzędne środka wyświetlanego zakresu danych*

```
void __fastcall TForm1::RysujPsiPunkty()
{
    //każdy vertex ma być traktowany jako punkt
    glBegin(GL_POINTS);
    for (int j=jmin; j<=jmax; j+=every) {
        float pkt1[3]; //tablica przechowujaca współrzędne x, y, z
        pkt1[1]=gp->GetYmin()+j*gp->GetDy()-y_sr; //wyznaczenie wsp. y
        for (int i=imin; i<=imax; i+=every){
```

```
                pkt1[0]=gp->GetXmin()+i*gp->GetDx()-x_sr;  //wyznaczenie wsp. x
                //wyznaczenie wsp. z, z uwzględnieniem wartości zmiennej mod_psi
                pkt1[2]=mod_psi?psi[i][j].psi_value*psi[i][j].psi_value:psi[i][j].psi_value;
                pkt1[2]*=wsp;

                //ustawienie odpowiedniego koloru rysowania
                glColor3ub(psi[i][j].kolor.r,psi[i][j].kolor.g,psi[i][j].kolor.b);
                //wyrysowanie punktu
                glVertex3fv(pkt1);
            }
        }
        glEnd();
    }
```

W ten oto sposób nasz program jest już w pełni funkcjonalny. Potrafi on wyświetlić dane z dowolnego pliku tekstowego zawierającego dowolną ilość linii komentarza i z dowolnej kolumny w nim zawartej. Dane są wyświetlane za pomocą czerwonych punktów. Można dowolnie zmieniać zakres wyświetlanych danych oraz krok. Jednak ochłonąwszy trochę z wrażenia jakie na nas wywarł nasz pierwszy wykres, stwierdzamy że nie jest on taki idealny. Po pierwsze przydałoby się zróżnicować kolor wyświetlanych punktów w zależności od tego jaką wartość współrzędnej *z* posiadają. Po drugie wyświetlanie za pomocą punktów, choć proste, pod względem wizualnym nie jest wcale tak efektowne. Przydałoby się zaimplementować inny sposób wyświetlania naszych danych.

9. Dodatkowe palety barw oraz style wyświetlania danych

Zaczniemy od zaimplementowania kolejnej palety barw (ja ją nazwałem *N5*). Po pierwsze musimy umożliwić użytkownikowi wybór, innej niż dotychczas używana, palety barw (zmiana wartości zmiennej `paleta`). Zrobimy to w podobny sposób jak dla opcji z *Wyświetlaj*. Przygotowujemy metodę `TForm1::SetPaleta(short int nr_palety)`, która będzie zmieniała wartość zmiennej paleta oraz dodatkowo będzie ona ustawiała ptaszek przy wybranej palecie barw (listing 28).

Listing 28. *Metoda `TForm1::SetPaleta(short int nr_palety)`, zmieniająca wartość zmiennej palet i ustawiająca ptaszek przy wypranej palecie*

```
void __fastcall TForm1::SetPaleta(short int nr_palety)
{
    //usuń ptaszek dla wybranej wcześniej palety
    switch (paleta) {
        case 0 : Skalszaroci1->Checked=0; break;
        case 1 : N51->Checked=0; break;        //opcja N5
        case 2 : N71->Checked=0; break;
        case 3 : wlength1->Checked=0; break;
        case 4 : temp1->Checked=0; break;
        case 5 : Jednolity1->Checked=0; break; //opcja jednolity
    }
    paleta=nr_palety;
    //ustaw ptaszek dla obecnie wybranej palety
    switch (paleta) {
```

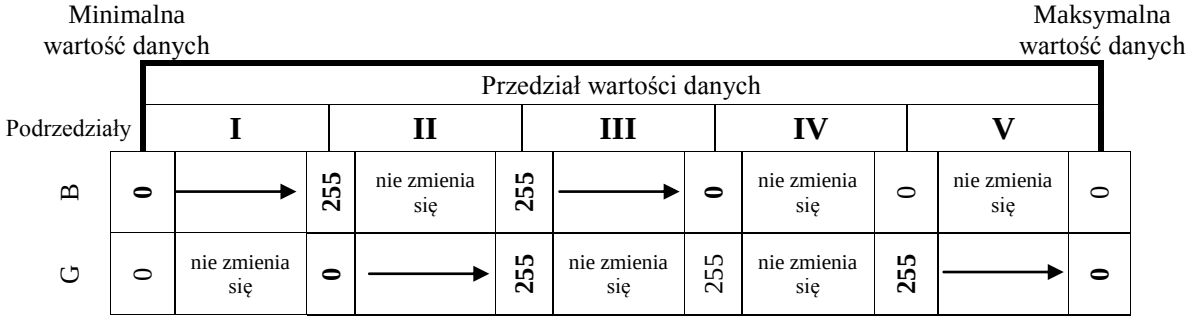
```
            case 0 : Skalszaroci1->Checked=1; break;
            case 1 : N51->Checked=1; break;          //opcja N5
            case 2 : N71->Checked=1; break;
            case 3 : wlength1->Checked=1; break;
            case 4 : temp1->Checked=1; break;
            case 5 : Jednolity1->Checked=1; break;  //opcja jednolity
        }
    }
```

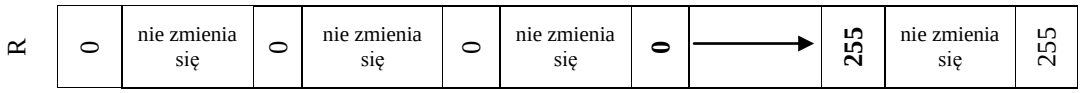
Musimy jeszcze wywołać powyższą metodę wewnątrz metod obsługujących wybór opcji *Paleta->N5* i *Paleta->Jednolity*. Klikamy więc dwa razy na tych opcjach a do automatycznie wygenerowanego kodu dopisujemy wywołanie *SetPaleta* z odpowiednim numerem jako argumentem (dla *Paleta->Jednolity* jest to 5 a dla *Paleta->N5* 1 - zgodnie z numerem pozycji, na której występują w menu, licząc od zera). Po każdym wywołaniu metody *SetPaleta* wywołujemy *SetColor*, ponieważ zmiana palety barw wymaga ponownego wyznaczenia kolorów dla wyświetlanych danych oraz *GL_RysujScene* - aby zobaczyć wykres w nowej palecie barw (listing 29).

Listing 29. *Metody obsługujące wybranie opcji Paleta->Jednolity lub Paleta->N5*

```
void __fastcall TForm1::N51Click(TObject *Sender)
{
    SetPaleta(1);
    SetColor();
    GL_RysujScene();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Jednolity1Click(TObject *Sender)
{
    SetPaleta(5);
    SetColor();
    GL_RysujScene();
}
```

Pozostało nam jeszcze zaimplementować samą paletę barw *N5*. Otwieramy więc program *Paint* i edytujemy jakiś kolor. Otwieramy zakładkę *Definiuj kolory niestandardowe*. Teraz musimy się zastanowić nad sposobem implementacji palety barw. Na początek wybierzmy kolor niebieski (0,0,255). Teraz, za pomocą suwaka zmniejszajmy powoli składową B. To co możemy zaobserwować, to że przez cały czas składowe R i G są równe zero. Ustawmy teraz z powrotem B na 255. Spróbujmy teraz powoli przesuwać krzyżykiem na palecie barw w lewą stronę, ale uważając aby przesuwał się on dokładnie po górnej krawędzi. Widzimy, że najpierw zaczyna rosnąć wartość składowej G, aż do momentu gdy osiągnie 255. Następnie maleje do zera wartość składowej niebieskiej. Gdy składowa niebieska osiągnie zero rośnie składowa czerwona. Po osiągnięciu składowej czerwonej wartości 255, malej do zera wartość składowej zielonej. Taki właśnie schemat kolorów spróbujemy zaimplementować. Nazwa *N5* pochodzi od liczby przedziałów jakie tu wyróżniliśmy. Poniżej przedstawiam nasze spostrzeżenia w postaci graficznej.





Algorytm jest następujący:

- dzielimy nasz przedział wartości danych na pięć podprzedziałów
- każdy podprzedział odpowiada zmianie jednej ze składowych (R,G,B) koloru
- wyznaczamy przedział, do którego zalicza się dana wartość z naszych danych, a następnie wyznaczamy z proporcji wartość zmieniającej się składowej

Kod realizujący paletę barw *N5* przedstawiam na listingu 30.

Listing 30. *Metoda implementująca paletę barw N5*

```
rgb_triplet __fastcall TForm1::SetColorN5(ftyp min, ftyp max, ftyp psi)
{
 unsigned char przedzial; //numer podprzedziału
   rgb_triplet Kolor;       //przechowuje składowe koloru
   ftyp delta=max-min,      //różnica pomiędzy maksymalna a minimalna
                            //wartościa danych
        dr=delta/5.0,dp;    //szerokość podprzedziału

   Kolor.r=Kolor.g=Kolor.b=0; //wyzerowanie składowych
   //wyznaczenie podprzedziału, do którego zalicza sie wartość psi
        if ((psi>=min) && (psi<(min+dr))) przedzial=1;
        else if ((psi>=min+dr) && (psi<(min+2*dr))) przedzial=2;
        else if ((psi>=min+2*dr) && (psi<(min+3*dr))) przedzial=3;
        else if ((psi>=min+3*dr) && (psi<(min+4*dr))) przedzial=4;
        else przedzial=5;

   if (!delta) return Kolor; //jeżeli min=max zwracamy kolor czarny
   switch (przedzial) {
       //przedział pierwszy
       //R=G=0 B=0->255
       case 1 : Kolor.r=0; Kolor.g=0;
                dp=psi-min;
                Kolor.b=(255*dp/dr);
                break;
       //przedział drugi
       //R=0 B=255 G=0->255
       case 2 : Kolor.r=0; Kolor.b=255;
                dp=psi-(min+dr);
                Kolor.g=(255*dp/dr);
                break;
       //przedział trzeci
       //R=0 G=255 B=255->0
       case 3 : Kolor.r=0; Kolor.g=255;
                dp=psi-(min+2*dr);
                Kolor.b=255-(255*dp/dr);
```

```
                break;
            //przedział czwarty
            //G=255 B=0 R=0->255
            case 4 : Kolor.g=255; Kolor.b=0;
                dp=psi-(min+3*dr);
                Kolor.r=(255*dp/dr);
                break;
            //przedział piaty
            //R=255 B=0 G=255->0
             case 5 : Kolor.r=255; Kolor.b=0;
                dp=psi-(min+4*dr);
                Kolor.g=255-(255*dp/dr);
                break;
        }
        return Kolor;
    }
```

Mając już zaimplementowana kolejną paletę barw, musimy wywołać ją w metodzie `Kolor`:

```
switch (paleta) {
    case 1 : return SetColorN5(minpsi, maxpsi, psi); //wybrano paletę N5
    case 5 : return SetColorJednolity(); //wybrano paletę jednobarwna
}
```

Paleta *N7* jest zaimplementowana w podobny sposób jak *N5*, z tym że w tym przypadku mamy więcej przedziałów (i tym samym kolorów). Natomiast odcienie szarości to najprostsza z zaprezentowanych tu palet. W przypadku tej palety mamy doczynienia z jednym przedziałem, w którym wartości wszystkich składowych są zawsze sobie równe. Wartość tych składowych wyznacza się z prostej proporcji. Pozostałe dwie palet wykorzystują bibliotekę funkcji graficznych z pliku *colormaps.cpp*, której autorem jest dr Jacek Matulewski. Nie będziemy implementowali pozostałych palet w tym tutoriale. Wszystkie one są zaimplementowane w dostarczonym programie. Implementacja ich nie wniosłaby niczego nowego do tego tutoriału. Zajmiemy się natomiast implementacją pozostałych stylów wyświetlania naszych danych. Najpierw zaimplementujemy styl *Impulsy*. Polegać on będzie na tym, że poprowadzimy linię od poziomu *z*=0 aż do wartości, którą chcemy wyświetlić. Do tego celu użyjemy oczywiście GL_LINES. Metoda (listing 31) realizująca ten sposób wyświetlania naszych danych jest bardzo podobna do `RysujPsiPunkty`.

Listing 31. *Metoda `TForm1::RysujPsiImpulsy()` implementująca styl wyświetlania Impulsy*

```
void __fastcall TForm1::RysujPsiImpulsy()
{
    rgb_triplet zero = Kolor(min_value, max_value, 0.0); //kolor poziomu z=0
    //dla wszystkich danych z przedziału jmin:jmax z krokiem every
    for (int j=jmin; j<=jmax; j+=every) {
        float pkt1[3];
        pkt1[1]=gp->GetYmin()+j*gp->GetDy()-y_sr; //wsp. y
        //dla wszystkich danych z przedziału imin:imax z krokiem every
        for (int i=imin; i<=imax; i+=every){
            pkt1[0]=gp->GetXmin()+i*gp->GetDx()-x_sr;  //wsp. x
            glBegin(GL_LINES);
            //pierwszy punkt linii (słupka) na poziomie z=0
            glColor3ub(zero.r,zero.g,zero.b);
            pkt1[2]=0.0;
```

```
                glVertex3fv(pkt1);

                //wyznaczamy wsp. z dla psi[i][j]
                //drugi punkt linii (słupka)
                pkt1[2]=mod_psi?psi[i][j].psi_value*psi[i][j].psi_value:
                                                    psi[i][j].psi_value; //wsp. z
                pkt1[2]*=wsp; //przeskalowanie
                glColor3ub(psi[i][j].kolor.r,psi[i][j].kolor.g,psi[i][j].kolor.b);
                glVertex3fv(pkt1);
                glEnd();
            }
        }
    }
```

Teraz, podobnie jak przy `RysujPsiPunkty` musimy wywołać `SetTyp` w metodzie obsługującej wybranie (kliknięcie) opcji *Typ->Impulsy*. W tym przypadku argumentem wywołania `SetTyp` jest liczba 3 (ponieważ *Impulsy* są na pozycji trzeciej licząc od zera). Aby możliwe było wyświetlenie naszych danych w postaci impulsów musimy jeszcze odpowiednio zmodyfikować metodę `TForm1::RysujScene()`. Musi ona wywoływać `RysujPsiPunkty` lub `RysujPsiImpulsy` w zależności od tego jaką wartość ma zmienna `typ`. Możemy to osiągnąć modyfikując kod w następujący sposób:

```
if (gp)
    switch (typ) {
        case 0 : RysujPsiPunkty(); break;
        case 3 : RysujPsiImpulsy (); break;
    }
```

Możemy teraz sprawdzić poprawność naszego programu. Po zmianie stylu wyświetlania naszych danych na *Impulsy* trzeba przyznać, że nie należy on do najładniejszych. Dlatego zaimplementujemy teraz wyświetlanie naszych danych za pomocą linii łączących punkty o tej samej współrzędnej *y*. Metoda ta (listing 32) jest znacznie mniej skomplikowana niż metoda rysująca impulsy. W zasadzie jest to przerobiona metoda `RysujPsiPunkty`, w której *GL_POINTS* zostało zamienione na *GL_LINE_STRIP* oraz wywołanie *glBegin* i *glEnd* jest umieszczone wewnątrz pierwszej pętli for (listing 32). Argument *GL_LINE_STRIP* w *glBegin* oznacz, że wszystkie werteksy zawarte pomiędzy *glBegin* a *glEnd* należy połączyć linią tworząc jedną krzywą. Funkcja *glBegin(GL_LINE_STRIP)* musi być wywołana wewnątrz pierwszej pętli for, ponieważ gdyby została wywołana za zewnątrz pętli, koniec pierwszej krzywej zostałby połączony z początkiem drugiej itd. Wywołanie jej w drugiej pętli zupełnie mijałoby się z celem. Tak jak i dla poprzednich stylów wyświetlania naszych danych, musimy wywołać *SetTyp* z argumentem 1 w metodzie obsługującej wybranie opcji *Typ->Linie*. Musimy także zmodyfikować metodę *RysujScene* tak aby wywoływała `RysujPsiLinie` jeśli wartość zmiennej `typ` jest równa 1.

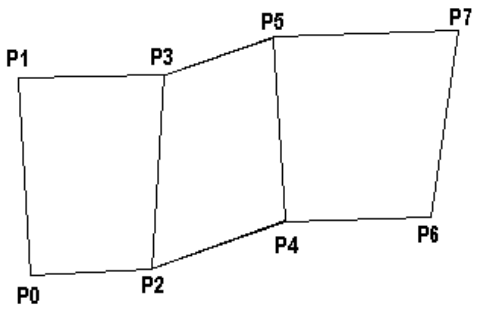
Listing 32. *Metoda `TForm1::RysujPsiLinie()` implementująca styl wyświetlania Linie*

```
void __fastcall TForm1::RysujPsiLinie()
{
    for (int j=jmin; j<=jmax; j+=every) {
        float pkt1[3]; //tablica przechowująca wsp. x, y i z
        pkt1[1]=gp->GetYmin()+j*gp->GetDy()-y_sr; //wsp. y
        glBegin(GL_LINE_STRIP); //połącz werteksy tworząc krzywa
        for (int i=imin; i<=imax; i+=every){
            pkt1[0]=gp->GetXmin()+i*gp->GetDx()-x_sr; //wsp. x
            pkt1[2]=mod_psi?psi[i][j].psi_value*psi[i][j].psi_value:psi[i][j].psi_value;
//wsp. z
            pkt1[2]*=wsp; //przeskalowanie
```

```
                glColor3ub(psi[i][j].kolor.r,psi[i][j].kolor.g,psi[i][j].kolor.b);
                glVertex3fv(pkt1);
            }
            glEnd();
        }
    }
```

Mamy już trzy z czterech stylów wyświetlania naszych danych. Teraz musimy zaimplementować ostatni styl *Powierzchnia* (listing 33). Zostawiłem go na sam koniec ponieważ jest on najtrudniejszy. W tym przypadku musimy wyznaczać normalne do powierzchni aby wykres ładnie wyglądał przy włączonym oświetleniu. Do stworzenia wykresu w tym stylu wykorzystamy GL_QUAD_STRIP, czyli paski złożone z połączonych czworokątów (rysunek 5). Podobnie jak w przypadku stylu *Linie*, *glBegin* i *glEnd* musi być wywołane wewnątrz pierwszej pętli *for* (gdyby były one wywołane na zewnątrz pętli koniec pierwszego paska byłby połączony z początkiem drugiego itd.). Paski będą tworzone ze wszystkich werteksów wywołanych w trakcie wykonywania się drugiej pętli *for*. Do wyznaczenia jednostkowego wektora normalnego wykorzystamy funkcję *JednostkowyWektorNormalny3fv* napisaną na kursie *Grafika 3D* (do projektu należy dodać plik *GLWektory.cpp*). Wektory normalne wyznaczać będziemy we wszystkich punktach. Najprawdopodobniej najlepiej byłoby wykorzystać także technikę uśredniania normalny, jednak my jej nie będziemy stosować (można potraktować to jako zadanie domowe). Przy wyborze punktów do wyświetlenia należy uwzględnić wartość kroku (zmienna `every`). Może to jednak spowodować, że wyjdziemy poza zakres przestrzenny sieci. Musimy o tym pamiętać i odpowiednio zabezpieczyć nasz program przed taką ewentualnością.

Rysunek 5
Numeracja wierzchołków w GL_QUAD_STRIP (UWAGA!!! Punkty tworzące jeden czworokąt nie muszą znajdować się na jednej płaszczyźnie)



Przedstawiony tu sposób wyświetlania płaszczyzny za pomocą werteksów nie jest zbyt optymalny. Wszystkie punkty wspólne dla dwóch sąsiednich, połączonych ze sobą, pasków są „rysowane" na monitorze dwa razy – raz dla jednego i drugi raz dla drugiego paska. Ponieważ takich połączonych pasków jest bardzo dużo to jest także bardzo dużo wyświetlanych podwójnie werteksów. Niestety jest to wada tej metody. Aby uniknąć niepotrzebnego wyświetlania werteksów musielibyśmy użyć zupełnie innego sposobu tworzenia płaszczyzny. Jedną z takich metod jest wykorzystanie tablicy wierzchołków. Ale jak to często bywa metoda ta wymaga większej ilość pamięci na dane, ponieważ oprócz współrzędnej *z*, musimy przechowywać także wartości współrzędnych *x* i *y*.

Listing 33. *Metoda `TForm1::RysujPsiPowierzchnia()` implementująca styl wyświetlania Powierzchnia*

```
void __fastcall TForm1::RysujPsiPowierzchnia()
{
   float pkt1[3], pkt2[3], pkt3[3];
    for (int j=jmin; j<=jmax-1; j+=every) {
        double y=gp->GetYmin()+j*gp->GetDy()-y_sr; //wsp. y
        pkt1[1]=y; pkt2[1]=y;
        glBegin(GL_QUAD_STRIP);
        for (int i=imin; i<=imax; i+=every){
            double x=gp->GetXmin()+i*gp->GetDx()-x_sr; //wsp. x
            float normalna[3]; //tablica przechowujaca współrzędne wektora normalnego
            //sparawdzamy czy nie wyjdziemy poza zakres sieci
            if (i+every<gp->GetNx() && j+every<gp->GetNy()) {
                //dolny lewy punkt
```

```
                    pkt1[0]=x;
                    pkt1[2]=(mod_psi?psi[i][j].psi_value*psi[i][j].psi_value:
                                                    psi[i][j].psi_value)*wsp; //wsp. z

                    //dolny prawy punkt
                    pkt2[0]=x+every*gp->GetDx(); //wsp. x
                    pkt2[2]=(mod_psi?psi[i+every][j].psi_value*psi[i+every][j].psi_value:
                                        psi[i+every][j].psi_value)*wsp; //wsp. z

                    //gorny lewy punkt
                    pkt3[0]=x; pkt3[1]=y+every*gp->GetDy();
                    pkt3[2]=(mod_psi?psi[i][j+every].psi_value*psi[i][j+every].psi_value:
                                                psi[i][j+every].psi_value)*wsp; //wsp. z

                    //wyznaczamy wektor normalny do płaszczyzny
                    //zawierającej pkt1, pkt2 i pkt3
                    JednostkowyWektorNormalny3fv(pkt1, pkt2, pkt3, normalna);

                    //ustawiamy wektor normalny
                    glNormal3fv(normalna);
                    glColor3ub(psi[i][j].kolor.r,psi[i][j].kolor.g,psi[i][j].kolor.b);
                    //"rysujemy" pierwszy punkt
                    glVertex3fv(pkt1);

    glColor3ub(psi[i][j+every].kolor.r,psi[i][j+every].kolor.g,psi[i][j+every].kolor.b);
                    //"rysujemy" drugi punkt
                    glVertex3fv(pkt3);
                }
            }
            glEnd();
  }
}
```

Oczywiście tak, jak w przypadku implementacji poprzednich stylów i tu musimy wywołać metodę `SetTyp` tym razem z argumentem 2 oraz dodać wywołanie metody z listingu 33 do `RysujScene`:

```
    if (gp)
        switch (typ) {
            case 0 : RysujPsiPunkty();        break;
            case 1 : RysujPsiLinie();         break;
            case 2 : RysujPsiPowierzchnia(); break;
            case 3 : RysujPsiImpulsy ();      break;
        }
```

10. Legenda i inne „bajery”

Mając zaimplementowany ostatni styl wyświetlania danych, nasz program jest już prawie skończony. Pozostało nam jedynie wyświetlenie legendy (jaki kolor jakiej wartości odpowiada) oraz zrobienie osi. Najpierw zrobimy wyświetlanie legendy, gdyż jest to procedura bardzo podobna do wyświetlania danych w postaci dwuwymiarowej powierzchni. W przypadku legendy będzie to kolorowy prostokąt (oczywiście kolory będą zależały od wybranej palet barw), zawsze zwrócony w naszą stronę. W związku z powyższym będziemy ją musieli wyświetlić, zanim dokonamy jakiejkolwiek transformacji (obrotu czy przesunięcia). Największym problemem będzie odpowiednie ustawienie naszej legendy w oknie, tak aby było ją całą widać oraz żeby nie była zbyt duża. Powodem przez, który te czynnością stwarzają trochę problemów jest fakt, że nie używamy rzutowania ortogonalnego lecz perspektywy. Dlatego współrzędne, w których należy umieścić legendę podam bez żadnego zagłębiania się w to skąd one się wzięły (po prostu zostały wyznaczone metodą prób i błędów). Na początku proponuję ustawienie szerokości i wysokość naszego okna odpowiednio na 681 i 504 (takie rozmiary miało moje okno gdy wyznaczałem współrzędne legendy). Metodę wyświetlająca legendę na monitorze przedstawiam poniżej.

Listing 34. *Metoda `TForm1::RysujLegende()` implementująca wyświetlająca legende na monitorze*

```
void __fastcall TForm1::RysujLegende()
{
    if (paleta==5) return; //jeśli jest wybrana paleta jednobarwna
                           //to nie wyświetlamy legendy
    double max_wsp_x = 0.12, //maksymalna współrzędna x legendy (prawa krawędź)
           min_wsp_x=0.04;   //minimalna współrzędna x legendy (lewa krawędź)
    glBegin(GL_QUAD_STRIP);
    for (double x=min_wsp_x; x<=max_wsp_x; x+=0.00025){
            //kolor krawędzi
            rgb_triplet kolor = Kolor(min_wsp_x,max_wsp_x,x);
            glColor3ub(kolor.r,kolor.g,kolor.b);
            //punkty wyznaczające krawędź
            glVertex3f(x,-0.080,-0.4);
            glVertex3f(x,-0.083,-0.4);
    }
    glEnd();
}
```

Tak jak już pisałem legenda musi być wyświetlona zanim zaczniemy wykonywać jakiekolwiek przekształcenia (ma być ona wyświetlana zawsze w tym samym miejscu okna). W związku z tym nie możemy jej wywołać w metodzie `RysujScene` ale bezpośrednio w `TGLForm::GL_RysujScene()`. Wywołujemy ją przed określeniem współrzędnych kamery:

```
...
Oswietlenie();

RysujLegende(); //rysuje legendę

//kamera
gluLookAt(0,0,KameraR,  //położenie kamery/oka
          0,0,0,    //punkt, na który skierowana jest kamera/oko
          0,1,0);  //kierunek "do góry" kamery (polaryzacja)
...
```

Jednak aby nasz program dało się skompilować metodę `RysujLegende` musimy uczynić abstrakcyjną metodą klasy *TGLForm*. Gdy już to zrobimy nasz program bez problemu powinien się skompilować. W programie

dostarczonym wraz z tym tutoriałem zdefiniowałem także dodatkowe światło (listing 35), które „włączam” w `TForm1::Oswietlenie()`, wyłączając pozostałe.

Listing 35. *Metoda `TForm1::Swiatlo3()` ustawiająca własności nowego światła rozproszonego*

```
void __fastcall TForm1::Swiatlo3()
{
    const float kolor3_rozproszone[]={0.5,0.5,0.5,1.0};

    glLightfv(GL_LIGHT3,GL_DIFFUSE,kolor3_rozproszone);
    glEnable(GL_LIGHT3);
}
```

Ostatnia rzecz jaka nam została do zrobienia to osie wyznaczaj układ współrzędnych. Nie będą to jednak osie OX, OY i OZ, lecz boks, wewnątrz którego, będzie znajdował się nasz wykres (patrz: rysunek 1). Zaczniemy od obrysowania naszego wykresu czworokątem leżącym w płaszczyźnie z=0. Do tego celu użyjemy GL_LINE_STRIP. W sumie wywołamy pięć razy funkcję *glVertex3f*: raz dla punktu, od którego zaczniemy rysowanie, trzy razy dla kolejnych punktów i ostatni raz ponownie dla punktu, od którego zaczęliśmy. Dzięki temu uzyskamy prostokąt obrysowujący nasz wykres na wysokości z=0. Metodę realizującą tą część przedstawiam na listingu 36. Wyznaczenie współrzędnych x i y rogów jest rzeczą bardzo prostą. Od współrzędnych wyznaczających zakres wyświetlanych danych odejmujemy po prostu współrzędne x_{sr} i y_{sr} środka wyświetlanego zakresu.

Listing 36. *Metoda `TForm1::RysujOsie()` obrysowująca nasz wykres prostokątem na wysokości z=0*

```
void __fastcall TForm1::RysujOsie()
{
    double Xmin=Form1->Xmin-x_sr, //min. wartość wsp. x przesunięta o x_sr
           Xmax=Form1->Xmax-x_sr, //max. wartość wsp. x przesunięta o x_sr
           Ymin=Form1->Ymin-y_sr, //min. wartość wsp. y przesunięta o y_sr
           Ymax=Form1->Ymax-y_sr; //max. wartość wsp. y przesunięta o y_sr

    glBegin(GL_LINE_STRIP);
    glColor3ub(255,255,255);
    glVertex3f(Xmin,Ymin,0); //współrzędne lewego dolnego rogu
    glVertex3f(Xmin,Ymax,0); //współrzędne lewego górnego rogu
    glVertex3f(Xmax,Ymax,0); //współrzędne prawego górnego rogu
    glVertex3f(Xmax,Ymin,0); //współrzędne prawego dolnego rogu
    glVertex3f(Xmin,Ymin,0); //współrzędne lewego dolnego rogu

    glEnd();
}
```

Aby zobaczyć efekt naszej pracy modyfikujemy odpowiednio funkcję `TForm1::RysujScene()`:

```
if (gp) {
    switch (typ) {
        ...
    }
    RysujOsie();
}
```

Teraz przerobimy naszą metodę tak aby wyświetlała pionowe „ściany” wzdłuż krawędzi *x*=*Xmin* oraz *y*=*Ymax*. Ich współrzędne *x* i *y* będą identyczne jak dla prostokąta obrysowującego nasz wykres na poziomie *z*=0. Jeśli `min_value` będzie mniejsze od zera, a `max_value` większe, wówczas będą się one rozciągać od *z*=`min_value`, aż do *z*=`max_value`. Natomiast jeśli `min_value` i `max_value` będą większe od zera wówczas będą się one rozciągać od *z*=0 do *z*=`max_value`, w przeciwnym wypadku od *z*=`min_value` do *z*=0. Również i w tym przypadku użyjemy GL_LINE_STRIP. Kod metody zawierający już niezbędne poprawki przedstawiam na listingu 37.

Listing 37. *Metoda `TForm1::RysujOsie()` rysująca boks, wewnątrz którego znajduje się nasza funkcja*

```
void __fastcall TForm1::RysujOsie()
{
    double minpsi=min_value,
           maxpsi=max_value;
    double Xmin=Form1->Xmin-x_sr, //minimalna wartość współrzędnej x przesunięta o x_sr
           Xmax=Form1->Xmax-x_sr, //maksymalna wartość współrzędnej x przesunięta o x_sr
           Ymin=Form1->Ymin-y_sr, //minimalna wartość współrzędnej y przesunięta o y_sr
           Ymax=Form1->Ymax-y_sr; //maksymalna wartość współrzędnej y przesunięta o y_sr

    if (minpsi>=0.0 && maxpsi>=0.0) minpsi=0.0;
    if (maxpsi<=0.0 && minpsi<=0.0) maxpsi=0.0;

    glColor3ub(255,255,255);
    //prostokąt z=0
    glBegin(GL_LINE_STRIP);
    glVertex3f(Xmin,Ymin,0); //współrzędne lewego dolnego rogu
    glVertex3f(Xmin,Ymax,0); //współrzędne lewego górnego rogu
    glVertex3f(Xmax,Ymax,0); //współrzędne prawego górnego rogu
    glVertex3f(Xmax,Ymin,0); //współrzędne prawego dolnego rogu
    glVertex3f(Xmin,Ymin,0); //współrzędne lewego dolnego rogu
    glEnd();

    //prostokat x=Xmin
    glBegin(GL_LINE_STRIP);
    glVertex3f(Xmin,Ymin,minpsi); //współrzędne dolnego bardziej oddalonego rogu
    glVertex3f(Xmin,Ymin,maxpsi); //współrzędne dolnego rogu
    glVertex3f(Xmin,Ymax,maxpsi); //współrzędne górnego rogu
    glVertex3f(Xmin,Ymax,minpsi); //współrzędne górnego bardziej oddalonego rogu
    glVertex3f(Xmin,Ymin,minpsi); //współrzędne dolnego bardziej oddalonego rogu
    glEnd();

    //prostokat y=Ymax
    glBegin(GL_LINE_STRIP);
    glVertex3f(Xmin,Ymax,minpsi); //współrzędne lewego bardziej oddalonego rogu
    glVertex3f(Xmin,Ymax,maxpsi); //współrzędne lewego rogu
    glVertex3f(Xmax,Ymax,maxpsi); //współrzędne prawego rogu
    glVertex3f(Xmax,Ymax,minpsi); //współrzędne prawego bardziej oddalonego rogu
    glVertex3f(Xmin,Ymax,minpsi); //współrzędne lewego bardziej oddalonego rogu
    glEnd();
```

```
}
```

Na koniec pozostało nam opisać odpowiednio legendę oraz stworzony przed chwilą boks. W tym tutoriale pokażę tylko jak opisać legendę. Opisanie osi pozostawiam jako zadanie domowe. Użyjemy w tym celu funkcji z pliku *GLNapisy.cpp*. Do funkcji zawartych w tym pliku dodamy jeszcze *glPrint* o zmiennej liście argumentów, która swym działaniem będzie przypominała funkcję *printf* z *C*. Procedurę tą znalazłem na http://aklimx.sppieniezno.pl/nehepl/display.php?id=13 (przetłumaczona na język polski strona http://nehe.gamedev.net), a jedyną zmianą jakiej dokonałem jest dodanie dodatkowego argumentu wywołania tej funkcji `GLuint podstawa` (znaczenie tej zmiennej wyjaśnię później). Dodajemy również funkcję `KillFont`, której zadaniem będzie usunięcie listy wcześniej przygotowanych znaków. Obie funkcje przedstawiam na listingu 38. Nie będę ich tutaj omawiać, a zainteresowanych zapraszam do odwiedzenia stron internetowych podanych powyżej. Na koniec przygotowujemy metodę `TForm1::OpisLegendy`, która będzie odpowiednio opisywać naszą legendę (wartość minimalna, maksymalna oraz wartość znajdująca się pośrodku przedziału wyznaczonego przez te wartości). Ponieważ używać będziemy czcionek rastrowych dlatego przy wyświetlaniu napisów wykorzystamy funkcję *glRasterPos3f* ustawiającą pozycję naszego tekstu w oknie. Metoda `TForm1::OpisLegendy` jest przedstawiona na listingu 39.

Listing 38. *Funkcje `glPrint()` oraz `KillFont()`*

```
void glPrint(GLuint podstawa, const char *fmt, ...)
      {
        char           text[256];
        va_list        ap;

        if (fmt == NULL)
               return;

        va_start(ap, fmt);
            vsprintf(text, fmt, ap);
       va_end(ap);

        glPushAttrib(GL_LIST_BIT);
        glListBase(podstawa - 32);
        glCallLists(strlen(text), GL_UNSIGNED_BYTE, text);
        glPopAttrib();
      }

void KillFont(GLuint podstawa){
       glDeleteLists(podstawa, 256);
}
```

Listing 39. *Metoda `TForm1::OpisLegendy()` wyświetlająca opis legendy na monitorze*

```
      void __fastcall TForm1::OpisLegendy()
      {
          double max_wsp_x = 0.12,
                 min_wsp_x=0.04;
                 //określenie minimalnej i maksymalnej wartości
                 //(uwzględnienie wartości mod_psi)
          double min = mod_psi?_min:minpsi,
                 max = mod_psi?_max:maxpsi;

          //opis legendy - wartosc minimalna
          glColor3ub(255, 255, 255);
          glRasterPos3f(min_wsp_x-0.015, -0.09, -0.4);
```

```
        glPrint(podstawa, "%4.2e", min);

        //opis legendy - wartosc pośrodku przedziału minimalna:maksymalna
        glRasterPos3f(min_wsp_x+(max_wsp_x-min_wsp_x)/2-0.015, -0.09, -0.4);
        glPrint(podstawa, "%4.2e", (max+min)/2.0);

        //opis legendy - wartosc maksymalna
        glRasterPos3f(max_wsp_x-0.015, -0.09, -0.4);
        glPrint(podstawa, "%4.2e", max);
    }
```

Zmodyfikujmy jeszcze odrobinę metodę `TForm1::RysujLegende()` tak aby wyświetlane były znaczniki, wskazujące dokładnie miejsce, do którego odnosi się opis legendy. Będą to tylko trzy znaczniki, ponieważ mamy zamiar wyświetlić tylko trzy wartości: minimalną, maksymalną oraz wartość znajdującą się pośrodku. Oczywiście w realizacji tych znaczników użyjemy GL_LINES. Opis legendy umieścimy dokładnie w takim miejscu aby znaczniki wskazywały środek napisu. Kod, o który należy uzupełnić metodę `TForm1::RysujLegende()` jest następujący (umieszczamy go na samym końcu metody):

```
    glColor3ub(255, 255, 255);
    glBegin(GL_LINES);
        //znacznik przy lewej krawędzi legendy
        glVertex3f(min_wsp_x,-0.083,-0.4);
        glVertex3f(min_wsp_x,-0.085,-0.4);

        //znacznik umieszczony na środku legendy
        glVertex3f(min_wsp_x+(max_wsp_x-min_wsp_x)/2,-0.083,-0.4);
        glVertex3f(min_wsp_x+(max_wsp_x-min_wsp_x)/2,-0.085,-0.4);

        //znacznik przy prawej krawędzi legendy
        glVertex3f(max_wsp_x,-0.083,-0.4);
        glVertex3f(max_wsp_x,-0.085,-0.4);
    glEnd();
```

Aby zobaczyć opis legendy musimy w odpowiednich miejscach wywołać odpowiednie metody. Najpierw wywołamy funkcję *StworzCzcionkeBitmapowa* budującą zestaw znaków ASCII, na samym końcu metody `TGLForm::GL_UstawienieSceny()`. Funkcja ta jako pierwszy argument przyjmuje uchwyt *Handle* naszego okna. Następnym argumentem jest nazwa czcionki (ja ustawiłem *Courier New*). Ostatnie dwa argumenty to *Pogrubienie* i *Kursywa* (u mnie, odpowiednio, 1 i 0). Funkcja ta zwraca wartość, która przechowuje numer pierwszej listy wyświetlania (ang. display list), którą tworzymy. Wartość tą zapisujemy pod zmienną o nazwie `podstawa`. Oczywiście należy ją zadeklarować najlepiej w sekcji *protected* klasy *TGLForm*. Zmienna ta będzie nam potrzebna, gdy używać będziemy funkcji *glPrint*. Funkcję `KillFont` wywołujemy w destruktorze klasy *TGLForm*. Jako argument podajemy zmienną `podstawa`. W tym momencie możemy już wywołać ostatnią metodę `TForm1::OpisLegendy()` wyświetlającą opis legendy. Wywołujemy ją na samym końcu metody rysującej legendę: `TForm1::RysujLegende()`. Teraz możemy skompilować projekt, wczytać jakieś dane i podziwiać efekty naszej pracy. Pierwsze co rzuca się w oczy to zbyt duży rozmiar czcionek. Zmniejszamy go więc w funkcji `StworzCzcionkeBitmapowa` z 30 na –14 i jeszcze raz uruchamiamy program. Ujemna wartość rozmiaru czcionki oznacza, że windows ma szukać czcionki bazującej na wysokości znaku. Jeżeli znak byłby dodatni, czcionka byłaby szukana na podstawie wysokości komórki (plamki). Tym razem wszystko wygląda dobrze (przynajmniej z grubsza).

Podsumowując. Udało nam się zrobić program, który potrafi wyświetlić dane używając do tego *OpenGL*. Dane te są wczytywane z plików tekstowych. Pliki mogą mieć dowolną ilość kolumn oraz linii komentarza umieszczonych na początku i zaczynających się od znaku #. Można wyświetlić dane z dowolnej kolumny. Zaimplementowaliśmy cztery style wyświetlania naszych danych oraz dwie palety barw. To co można jeszcze

zrobić aby przyśpieszyć działanie naszego programu to wykorzystać tablice wierzchołków oraz VBO. **V**ertex **B**uffer **O**bject (VBO) jest to rozszerzenie OpenGL umożliwiające tworzenie obiektów zawierających opis geometrii. Rozszerzenie to umożliwia szybkie renderowanie sceny i znacznie zwiększa wydajność aplikacji. Niestety może nie działać na starszych kartach graficznych. Na zakończenie przytoczę jeden cytat, specjalnie dla osób, którym coś nie wyszło w tworzeniu programu przedstawionego w tym tutoriale: *Tylko niepotrzebny nikomu program przebiega bez zakłóceń*.

Wszelkie uwagi odnośnie skryptu oraz programu proszę kierować na adres tomek@fizyka.umk.pl.