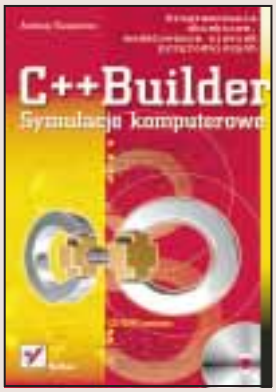
C++Builder. Symulacje komputerowe



Autor: Andrzej Stasiewicz
ISBN: 83-7361-052-9
Format: B5, stron: 238
Zawiera CD-ROM

Nowoczesne techniki programowania i projektowania pozwalają pisać złożone aplikacje także osobom nie będącym informatykami. Pasjonatom nauk przyrodniczych przychodzi z pomocą projektowanie obiektowe: dzięki zastosowaniu gotowych obiektów mogą oni symulować komputerowo zjawiska występujące w przyrodzie. Programowanie zorientowane obiektowo wymaga przede wszystkim bardzo dokładnych opisów funkcjonalnych obiektów; szczegóły techniczne, realizatorskie i znajomość ich konstrukcji wewnętrznej nie są tu ważne.

Książka C++Builder. Symulacje komputerowe przedstawia kilkanaście programów symulujących rozmaite zjawiska występujące w przyrodzie. Programy te zostały napisane w języku C++ (użyto dialektu C++Builder Borlanda). Zastosowano w nich gotowe klasy, które możesz odnaleźć na dołączonym do książki krążku CD, można je rozbudowywać i wykorzystywać we własnych programach. Osoby zainteresowane tajnikami programowania obiektowego poznają szczegóły konstrukcji obiektów, przyrodnicy mogą pominąć bardziej techniczne fragmenty i skoncentrować się na modelowaniu zjawisk przyrodniczych.

Programy opisane w książce dotyczą:

- Widma światła białego
- Drgań i fal prostych
- Fal na wodzie i ich interferencji
- Interferencji światła
- Postrzegania głębi i geometrii 3D
- Fotografii relatywistycznej
- Algorytmów wzrostu
- Tworzenia wirtualnych przestrzeni za pomocą techniki śledzenia promieni (ray-tracing)

Programowanie obiektowe jest to jedyna technika szybkiego tworzenia aplikacji z wykorzystaniem istniejących, uniwersalnych algorytmów. Jeśli jesteś interesujesz się fizyką czy biologią, książka udowodni Ci, że nie musisz kończyć studiów informatycznych, by modelować komputerowo interesujące Cię zjawiska.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 3.

Fale na wodzie

Poprzedni rozdział był poświęcony doskonaleniu umiejętności włączania obiektów `TSkalowanie` i `TDiagram` do swoich aplikacji. Pretekstem były nieskomplikowane zjawiska falowe, konkretnie tzw. fale proste. Teraz zmierzymy się z nieco poważniejszym zagadnieniem — falami powierzchniowymi. Fizyka opisywanych zjawisk nie będzie przy tym zbyt trudna, ponieważ ograniczymy się do rzeczy zapewne znanych. Jednak komputerowa wizualizacja fal powierzchniowych będzie wymagała włączenia do naszego warsztatu nowych obiektów, zdolnych do wykreślenia obrazów płacht funkcyjnych.

Powierzchniowa fala płaska

Powierzchnia wody jest dwuwymiarowym, sprężystym ośrodkiem, w którym mogą rozchodzić się zaburzenia. Cząsteczka (w rzeczywistości raczej grupa takich cząsteczek) wody wytrącona, np. kijem, z położenia równowagi drga rytmicznie w górę i w dół. Za pośrednictwem sił oddziaływania międzycząsteczkowego porusza inne, sąsiednie cząsteczki. Wówczas mamy do czynienia z rozchodzeniem się fali na powierzchni wody.

Przedstawimy taki obraz na ekranach monitorów. Nie będziemy przy tym modelować tych wszystkich cząsteczek i ich oddziaływań, tylko wymyślimy matematyczny kształt dwuwymiarowej funkcji, której wykres możliwie wiernie odda obraz prawdziwej fali.

Aplikacja pokaże obraz powierzchniowej fali płaskiej. Taka fala powstaje wtedy, gdy czynnik zaburzający spokojną taflę wody ma kształt długiej linijki. Gdyby nagle poruszyła się jedna ze ścian basenu, po powierzchni wody zaczęłaby wędrować fala płaska. Termin *fala płaska* — cokolwiek oznacza — powstał głównie po to, by odróżnić taką falę od powierzchniowej fali kolistej. Do fal kolistych przejdziemy potem, a teraz włączmy komputery.

Zaczniemy od eksploatacji nieco zubożonego algorytmu, zamkniętego w klasie `TWykres-Pseudo3d`. Algorytm ten, opisany ze strony funkcjonalnej w dodatku na końcu książki, służy do prostego obrazowania powyginanych, dwuwymiarowych powierzchni (rysunek 3.1). Teraz skoncentrujmy się na sposobie włączania go do aplikacji.

Rysunek 3.1. *Obiekt TWykresPseudo3d nie nadaje się do bezpośredniego wykorzystania, bo potrafi tylko wykreślać błahe, proste, płaskie powierzchnie. Na szczęście, funkcja odpowiedzialna za wykres jest zapowiedziana jako wirtualna, a to znaczy, że powinniśmy utworzyć obiekt potomny, który przejmie cały ustrój TWykresPseudo3d, ale z własną, już niewirtualną funkcją, odpowiadającą za kształt wykresu*

Przypomnijmy raz jeszcze schemat działania, prowadzący do uzyskania *stanu zerowego* aplikacji z dodatkowym modułem. Należy:

- założyć nowy katalog roboczy,
- skopiować do niego pliki nowego modułu, tym razem *pseudo3d.h* i *pseudo3d.cpp* będące nośnikiem klasy `TWykresPseudo3d`,
- uruchomić Buildera i włączyć do projektu *pseudo3d.cpp*,
- dopisać frazę `#include "pseudo3d.h"`,
- zapisać wszystko w katalogu roboczym.

Spróbujmy wszystko zrobić tak jak dotychczas. Wygenerujmy funkcję — reakcję na zdarzenie `OnPaint` (jest to funkcja odpowiedzialna za grafikę okienka). Zadeklarujmy tam egzemplarz klasy `TWykresPseudo3d`, tak jak podpowiada to intuicja i dotychczasowe doświadczenie z obiektami `TSkalowanie` albo `TDiagram`. Wreszcie, uruchomimy `iterator` kreślący wykres:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
 int marg = 10, xe1, ye1, xe2, ye2;   //współrzędne ekranowe odcinka
 TWykresPseudo3d w( marg, marg , ClientWidth - ClientHeight - 2 * marg,
ClientHeight - 2 * marg, 0, 0, 50, 50);
 w.inicjuj_iterator();               //uruchomienie iteratora
 while( w.iterator( xe1, ye1, xe2, ye2))  //pętla kreśląca wykres
 {
  Canvas -> MoveTo( xe1, ye1);       //wykreślenie odcinka wykresu
  Canvas -> LineTo( xe2, ye2);
 }
}
```

Konstruktor tworzy obiekt typu `TWykresPseudo3d`. Publiczna funkcja `inicjuj_iterator()` rozpoczyna bieg algorytmu. Publiczny `iterator` dostarcza współrzędne wszystkich odcinków, które mają złożyć się na wykres. Tylko gdzie jest nasza fala płaska?

Algorytmy obiektu, zanim wykreślą jakikolwiek punkt płachty funkcyjnej, muszą przepytać jakąś funkcję o jej wartość we wszystkich, istotnych dla kreślenia punktach. Taka funkcja jest wbudowana w obiekt. To rozwiązanie ma jednak oczywistą wadę — obiekt kreśli zawsze tę samą płachtę, tutaj płaski prostokąt. Wiem, co mówię. Oto wbudowana w TWykresPseudo3d, trywialna funkcja, której wykres rysuje obiekt:

```
double TWykresPseudo3d :: fun( double x, double y)
{
 return 0;
}
```

Musimy ją podmienić. W świecie obiektów przewidziano taką sytuację. Po pierwsze, niech wykreślana funkcja będzie rzeczywiście na stałe wbudowana do obiektu, ale niech to będzie funkcja *wirtualna*, czyli możliwa do przedefiniowania:

```
class TWykresPseudo3d
{
 private:
   ...
 public:
   TWykresPseudo3d( ...);      //konstruktor
   ...
   virtual double fun( double x, double y);
   ...
};
```

Po drugie, użytkownicy naszego obiektu nie będą bezpośrednio korzystać z klasy TWykresPseudo3d, tylko tzw. *klasy potomnej*. Klasa potomna *odziedziczy* wszystko po TWykresPseudo3d, za wyjątkiem funkcji *wirtualnej*. Na jej miejsce napiszemy własną, która nie będzie już *wirtualna*:

```
class TMojWykres : TWykresPseudo3d     //dziedziczenie
{
 private:
   ...
 public:
   TMójWykres( ...);         //konstruktor
   ...
   double fun( double x, double y);   //nowa funkcja
   ...
};
```

Tak naprawdę, użytkownik naszego obiektu TWykresPseudo3d (w powyższej sytuacji zwanego obiektem bazowym) niewiele się napracuje. Sama fraza dziedziczenia wymaga nikłego zaangażowania w pisanie — dwukropek i przytoczenie nazwy obiektu. Napisanie konstruktora okaże się bardzo łatwe, gdyż konstruktor TMojWykres() ograniczy się do wywołania konstruktora TWykresPseudo3d(). A co z napisaniem funkcji fun()? No cóż, akurat to jest konieczne, ponieważ właśnie nasz użytkownik ma pomysł na płachtę funkcyjną.

Co można powiedzieć o *dziedziczeniu* i *funkcjach wirtualnych*? Jest to prawdziwa rewolucja w programowaniu. Dostajemy nowy, pachnący farbą obiekt, który możemy zmieniać, nie bojąc się, że coś popsujemy. Algorytmy wirtualne są po to, by je zmieniać, a ściślej — by na ich miejsce wprowadzać własne.

Postawmy się na chwilę w roli programisty (np. Borlanda) — producenta obiektów. Jednym z jego najbardziej odpowiedzialnych zadań jest dokonanie wyboru, które algorytmy pisanego obiektu mają być wirtualne, czyli możliwe do przedefiniowania przez przyszłych użytkowników. Jeśli zabraknie gdzieś słowa `virtual`, obiekt już zawsze będzie niezbyt elastyczny w użyciu, np. zawsze będzie rysował wykres płaskiej powierzchni. Dzięki wirtualności mogą powstawać obiekty prawdziwie uniwersalne. Dopiero ostateczny użytkownik zadecyduje, co tak naprawdę ma robić ów obiekt.

Jeśli w opisie obiektu widzimy jakieś algorytmy opatrzone frazą `virtual`, należy się zastanowić, czy nie warto utworzyć zupełnie nowego obiektu, który będzie potomkiem bazowego oryginału. W ten sposób mamy możliwość napisania nowej funkcji, która zastąpi wirtualny oryginał.

Teraz napiszemy tę aplikację wzorcowo. Do powyższego schematu działania dodamy tworzenie obiektu potomnego z konkretną (niebędącą już wirtualną) funkcją, opisującą kształt płaskiej fali powierzchniowej.

W części *h* modułu z formą (mógłby to też być nowy, specjalnie powołany do życia moduł) dopiszemy deklarację nowej klasy:

```
#include "pseudo3d.h"
...
class TFalaPlaska : public TWykresPseudo3d
{
 public:
   TFalaPlaska( int xe0, int ye0, int eszer, int ewys, double xr0, double yr0,
double rszer, double rwys);
   double fun( double x, double y);
};
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:      // IDE-managed Components
    void __fastcall FormPaint(TObject *Sender);
 private:        // User declarations
 public:        // User declarations
    __fastcall TForm1(TComponent* Owner);
};
```

Część *h* zawiera więc deklaracje dwóch klas — naszej, powstałej z `TWykresPseudo3d` i klasy okienka, dodanej automatycznie przez Builder. Uważajmy, by niczego nie popsuć w kodzie automatycznie wygenerowanym przez Borlanda. Nie zapomnijmy też o frazie z pierwszej linii powyższego algorytmu, która informuje kompilator, co oznacza napis `TWykresPseudo3d`.

Deklaracja obiektu o nazwie `TFalaPlaska` wprowadza dwa nowe algorytmy: swój konstruktor i funkcję przedefiniowującą wirtualny oryginał. Musimy te algorytmy spisać w części *CPP* modułu:

```
TFalaPlaska :: TFalaPlaska( int xe0, int ye0, int eszer, int ewys,  double xr0,
double yr0, double rszer, double rwys)
     : TWykresPseudo3d( xe0, ye0, eszer, ewys, xr0, yr0, rszer, rwys)
{
```

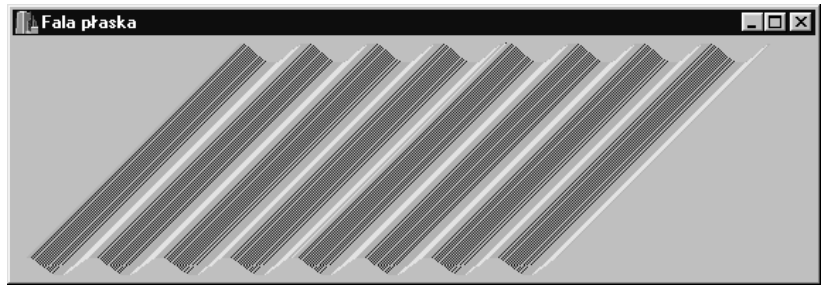
```
}
//---------------------------------------------------------
double TFalaPlaska :: fun( double x, double y)
{
 return  7 * cos( x);
}
```

Pierwsza z tych funkcji jest konstruktorem nowej klasy (dlatego, że nazywa się tak samo jak klasa). Konstruktor ten nie robi nic poza wywołaniem konstruktora klasy bazowej i przekazaniem mu wszystkich swoich argumentów. Fraza z dwukropkiem, łącząca dwa nagłówki konstruktorów, jest wymagana przez składnię języka.

Druga funkcja ma nagłówek identyczny z jej wirtualnym oryginałem i wniknie w niuanse algorytmów, w których uczestniczył jej pierwowzór. Obiekty nawet nie poznają, że ktoś zamienił funkcje.

Niewielka zmiana jest też w funkcji — reakcji na zdarzenie `OnPaint`. Tworzymy tam konkretny obiekt naszej, niewirtualnej `TFalaPlaska`, która nie jest już klasą `TWykresPseudo3d`:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
 TFalaPlaska fp( marg, marg , ClientWidth - ClientHeight - 2 * marg, ClientHeight -
2 * marg, 0, 0, 50, 50);
 ...
 fp.inicjuj_iterator();
 ...
```



Rysunek 3.2. *Jeśli klasa posiada wirtualne algorytmy, jest to sygnał, że prawdopodobnie możemy głęboko ingerować w jej funkcjonowanie. Należy tylko zrobić dwie rzeczy: utworzyć klasę potomną i spisać w niej nowe wersje wirtualnych algorytmów. Ta blacha falista to niezbyt ciekawy obraz płaskiej fali powierzchniowej. Iterator wykresu umożliwia sprawdzenie, czy kreślony odcinek podjeżdża w górę czy też opada w dół. Zastosowano tu inny schemat barwienia zboczy rosnących i malejących. Szczegóły znajdują się na płycie CD*

Jeżeli ktoś chce eksperymentować, to (gdy program będzie już gotowy) może wrócić jeszcze raz do tekstu źródłowego i usunąć słowo `virtual` z deklaracji bazowej klasy `TWykresPseudo3d` — czekający na modyfikację. Całość powinna się skompilować i uruchomić, lecz na ekranie znów pojawi się płaska powierzchnia. Dlaczego? Obiekt potomny zawiera teraz dwie funkcje: głęboko ukryty, niewirtualny oryginał i nadpisujący go nowy egzemplarz. Jednocześnie wszystkie operacje tworzenia wykresu znajdują się w klasie bazowej, zatem ważny jest bazowy, nieciekawy, ale bliższy egzemplarz funkcji. Schemat podmieniania bazowych algorytmów nie zadziałał.

Fala kolista

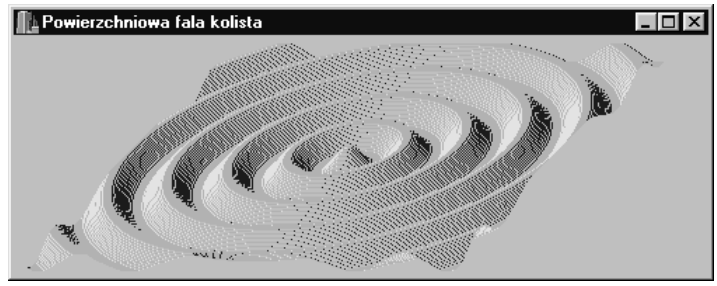
Powierzchnia wody jest środowiskiem dwuwymiarowym. Dzięki sprężystości oddziaływań między cząsteczkami wody mogą rozchodzić się w tym środowisku dwuwymiarowe fale poprzeczne (tzn. takie, w których wychylenie nie pokrywa się z kierunkiem rozprzestrzeniania). Dzięki swej powszechności, woda jest idealnym miejscem, umożliwiającym obserwację i badanie fal powierzchniowych.

Zaczniemy od matematycznego opisu takiej fali. Opisać falę, to znaczy podać wychylenie powierzchni wody w dowolnym punkcie `(x, y)`. Fale na wodzie (a raczej ich statyczny, zatrzymany w czasie obraz) są dane następującym równaniem:

```
F( x, y) = A * sin( 2 * pi / L * r + fi);
```

Dzięki takiemu równaniu potrafimy wyliczyć wychylenie powierzchni wody w dowolnym punkcie. W powyższym równaniu parametr `A` oznacza amplitudę, `L` — długość fali i `fi` — jej fazę początkową. Parametry te są niemal identyczne z ich odpowiednikami we wcześniej omawianych falach prostych. O ile jednak uprzednio opisane ciągi falowe zależały od jednej współrzędnej `x`, tak koliste fale na wodzie (rysunek 3.3) zależą od odległości od centrum, w którym znajduje się drgający punkt, wymuszający powstawanie fali. Odległość ta jest zwyczajowo oznaczana literą `r`:

```
double r = sqrt( x * x + y * y);
```



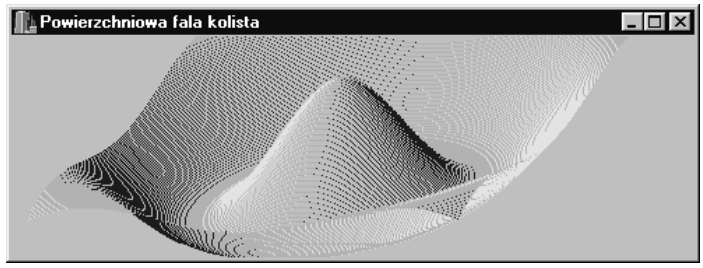
Rysunek 3.3. *Powierzchniowa fala kolista, której formuła zawiera dość istotny błąd — amplituda nie maleje wraz z odległością*

Jeszcze raz użyjemy mało efektownej klasy `TWykresPseudo3d`. Program właściwie niczym nie różni się od poprzedniego — jedynie wprowadzana jest funkcja o innym kształcie matematycznym:

```
//  przedefiniowana funkcja obiektu podstawowego
double TFalaKolista :: fun( double x, double y)
{
 double r = sqrt( x * x + y * y);
 return  7 * cos( r);
}
```

Prawdziwe, koliste fale na wodzie mają jeszcze jedną cechę — są tym słabsze, im dalej odpłynęły (rysunek 3.4). Zatem ich amplituda zależy też od odległości:

```
F( x, y) = A( r) * sin( 2 * pi / L * r + fi);
```



Rysunek 3.4. *Realistyczny obraz fali powierzchniowej musi zawierać czynnik tłumiący amplitudę wraz ze wzrostem odległości — im dalej, tym mniejsza fala. Oto wykres funkcji F(x, y) = 50 * exp(–r / 100) * cos(r). Funkcja ta maleje wraz z odległością*

W powyższym równaniu amplituda jest malejącą funkcją odległości.

Teraz sporządzimy bardziej realistyczny wykres fali kolistej. Zmienimy narzędzie kreślące płachtę i, zamiast ubogiego typu `TWykresPseudo3d`, wprowadzimy bardziej zaawansowany `TWykres3d`. Klasa ta jest potomkiem obiektu `T3d`, zajmującego się wykreślaniem perspektywicznych rzutów terenu. Implementuje też oświetlenie płachty. Przebadajmy kształty publicznych algorytmów `TWykres3d`.

Najważniejszy jest konstruktor — algorytm służący do tworzenia konkretnego egzemplarza obiektu typu `TWykres3d`:

```
TWykres3d(
TPunkt obs,                    //pozycja obserwatora
TPunkt osw,                    //pozycja punktu oświetlającego
double xr0, yr0, rszer, rwys,  //obszar funkcji
int il_x, il_y,                //liczba podziałów dziedziny
int xe0, ye0, eszer, ewys,     //okno ekranowe
double odl_ekr=0.5, szer_ekr=0, wys_ekr=0);   //opis ekranu
```

Trzy ostatnie argumenty mają swoje wartości domyślne, zatem w konkretnej sytuacji może ich po prostu nie być. Oznacza to pozostanie przy predefiniowanych wartościach odległości od ekranu i jego rozpiętości w metrach.

Obiekt `TWykres3d` (a raczej jego przodek `T3d`) zakłada, że obserwowana rzeczywistość spoczywa w pobliżu początku układu współrzędnych. Mówiąc inaczej, stożek widzenia jest zawsze skierowany do środka. To pewne uproszczenie w stosunku do takich bibliotek, jak np. OpenGL. Jednak dzięki temu wystarczy zadać pozycję obserwatora, którego współrzędne widzimy jako trzy pierwsze argumenty konstruktora.

Trzy kolejne argumenty to pozycja punktu oświetlającego. Obiekt analizuje wzajemną zależność położeń obserwatora, punktu oświetlającego oraz usytuowania obserwowanej powierzchni i na podstawie tych danych wylicza jasność oświetlenia.

Potem widzimy cztery parametry określające wycinek matematycznej powierzchni, nad którą jest zadana funkcja (czyli *dziedzinę* funkcji).

Dalej mamy dwa parametry określające liczbę podziałów płachty na elementarne czworokąty i wreszcie położenie oraz rozmiary okienka ekranowego (przeznaczonego na grafikę).

Oto typowa fraza definiowania zmiennej typu `TWykres3d`:

```
TPunkt obserwator(100., 100., 50.);
TPunkt oswietlenie(-5., 10., 0.);
TWykres3d w3d( obserwator,          //pozycja obserwatora
 oswietlenie,                       //pozycja punktu oświetlającego
 0., 0., 50., 50.,                  //obszar funkcji
 80, 80,                            //dokładność kreślenia
 0, 0,                              //lewy górny róg okienka
 ClientWidth, ClientHeight);        //okno ekranowe
```

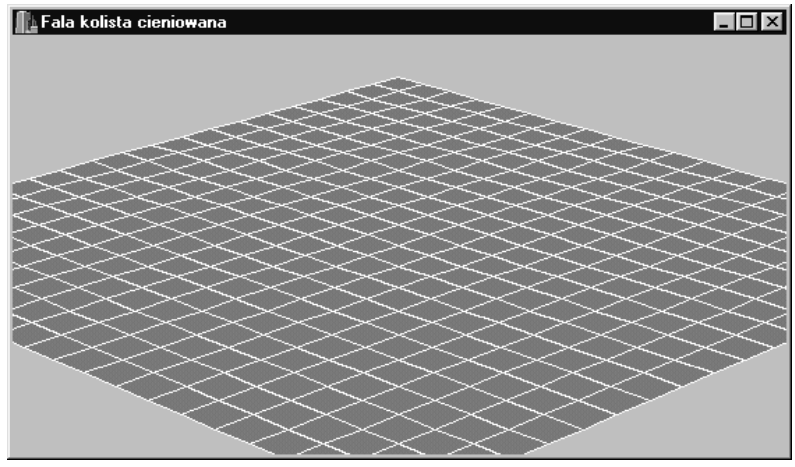
Jak przed chwilą, wykreślany kształt zadajemy wirtualną funkcją dwóch zmiennych:

```
virtual double fun( double x, double y);    //funkcja do wyrysowania
```

Funkcja ta musi być określona w dziedzinie niedawno zdefiniowanej w konstruktorze klasy. Wirtualność funkcji sugeruje, że będziemy musieli utworzyć obiekt potomny, czyli obiekt dziedziczący właściwości niniejszego obiektu. Jeśli zaniechamy tej czynności, będziemy zmuszeni do obserwacji błahego pierwowzoru oryginalnej funkcji:

```
double fun( double x, double y)
{
 return 0;
}
```

Pierwowzór ten znów jest niezwykle prostą płachtą, spełniającą wymagania algorytmu.



Rysunek 3.5. *Funkcja wirtualna niekiedy nic nie robi (ma puste ciało) lub robi zbyt mało, jak np. na powyższym rysunku, gdzie widać wykres funkcji f(x, y) = 0. Obecność funkcji wirtualnej jest zaproszeniem do tworzenia obiektu potomnego, w którym funkcję wirtualną zastąpi pełnowartościowy algorytm*

Za kreślenie płachty funkcyjnej odpowiada specjalny `iterator`. Stanowi go para funkcji:

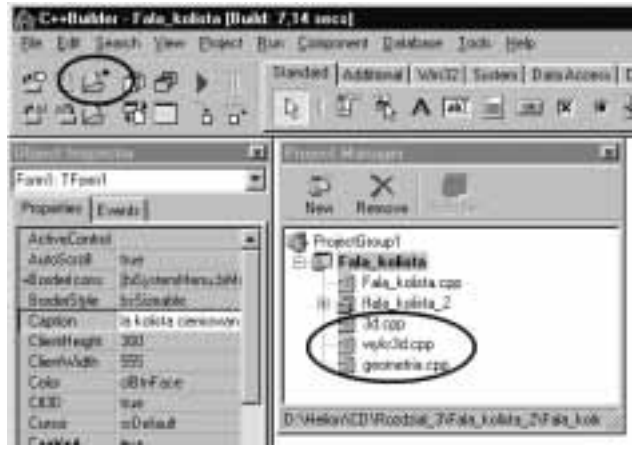
```
void inicjuj_iterator( void);               //początek kreślenia
bool iterator(        //kreślenie element po elemencie, dopóki true
  int &xe1, &ye1, &xe2, &ye2, &xe3, &ye3, &xe4, &ye4, &jasnosc);
```

`Iterator` kolejno dostarcza współrzędne wszystkich wielokątów, które złożą się na płachtę, a dodatkowo informuje o procentowej jasności ich oświetlenia. Typowe frazy kreślenia płachty `iteratorem` to:

```
int xe1, ye1, xe2, ye2, xe3, ye3, xe4, ye4, jasn;
w3d.inicjuj_iterator();
while( w3d.iterator( xe1, ye1, xe2, ye2, xe3, ye3, xe4, ye4, jasn))
{
  "zamień jasn na dowolny kolor";
  Rectangle( xe1, ye1, xe2, ye2, xe3, ye3, xe4, ye4);
}
```

Ten, nieco umowny, algorytm ilustruje to, co trzeba, czyli uruchomienie `iteratora`, pozyskiwanie współrzędnych i koloru każdego z elementarnych wielokątów płachty i wykreślanie go.

Napiszmy teraz prawdziwy program, ilustrujący srebrzystą powierzchnię wody, którą w jednym punkcie ktoś dotknął kijem. W katalogu roboczym niech znajdą się pliki źródłowe klasy `TWykres3d`, a w związku z tym także klasy `T3d`, bo to ona jest przodkiem `TWykres3d`. Ponieważ pojawił się też napis `TPunkt`, do katalogu dodamy też moduł *geometria*. Po uruchomieniu Buildera do przestrzeni aplikacji (do *projektu*) dołączymy pliki *cpp* tych trzech modułów (rysunek 3.6).



Rysunek 3.6. *Wszystkie moduły, nawet te, do których nie odwołujemy się bezpośrednio, muszą być wyszczególnione w projekcie. Projekt jest spisem treści aplikacji*

Dysponując *stanem zerowym* aplikacji, czyli poprawnie skompletowanym projektem, tworzymy klasę potomną. W części *H* modułu z okienkiem wpisujemy frazy dziedziczenia:

```
class TWykres : public TWykres3d
{
 private:
 public:
   TWykres(TPunkt obs, TPunkt osw,
           double xr0, double yr0, double rszer, double rwys,
           int il_x, int il_y,
           int xe0, int ye0, int eszer, int ewys);
   double fun( double x, double y);
};
```

Powyższa deklaracja zapowiada klasę potomną, zatem przejmującą całe wnętrze klasy TWykres3d, zaopatrzoną jednak w dwa własne algorytmy. W części *CPP* spiszemy ciała tych algorytmów:

```
//  Konstruktor obiektu kreślącego
TWykres :: TWykres(
    TPunkt obs, TPunkt osw,
    double xr0, double yr0, double rszer, double rwys,
    int il_x, int il_y,
    int xe0, int ye0, int eszer, int ewys)
  : TWykres3d( obs, osw,
                   xr0, yr0, rszer, rwys,
                   il_x, il_y, xe0, ye0, eszer, ewys)
{
}
//---------------------------------------------------------
// Przedefiniowana funkcja obiektu podstawowego
// Fala kolista wygasająca wykładniczo.
double TWykres :: fun( double x, double y)
{
 double r = sqrt(x * x + y * y);
 return 3.0 * exp( -r * 0.1) * cos( r);
}
```

Konstruktor nie robi nic, a ściślej ogranicza się do wywołania konstruktora klasy bazowej. Funkcja zastępująca wirtualny oryginał wykreśla płachtę, będącą powierzchniowym kosinusem, wykładniczo gasnącym wraz z odległością.

Należy jeszcze wygenerować i napisać najważniejszą funkcję — reakcję na zdarzenie OnPaint:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
 int marg = 10;
 TPunkt obs(100., 100., 50.), osw(-5., 10., 0.);
 TWykres w3d( obs,                  //pozycja obserwatora
    osw,                            //pozycja punktu oświetlającego
    0., 0., 50., 50.,               //obszar funkcji
    80, 80,                         //dokładność kreślenia
    marg, marg,
    ClientWidth - 2 * marg, ClientHeight - 2 * marg);//okno ekranowe
 int xe1, ye1, xe2, ye2, xe3, ye3, xe4, ye4, jasn;
 int a;
 TPoint p[ 5];
 TColor kolor;

 w3d.inicjuj_iterator();            //początek kreślenia
 while( w3d.iterator( xe1, ye1, xe2, ye2, xe3, ye3, xe4, ye4, jasn))
 {
  p[ 0].x = xe1;                    //opis elementarnego czworokąta
  p[ 0].y = ye1;
  p[ 1].x = xe2;
  p[ 1].y = ye2;
  p[ 2].x = xe3;
  p[ 2].y = ye3;
  p[ 3].x = xe4;
```

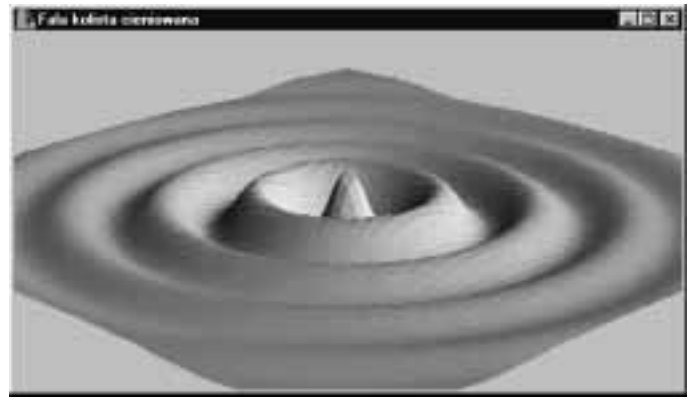
```
    p[ 3].y = ye4;
    p[ 4].x = xe1;
    p[ 4].y = ye1;

    a = (int)((double)jasn * 2.5);    //pomocnicze
    kolor = (TColor)RGB( a, a, a);    //synteza koloru
    Canvas -> Pen -> Color = kolor;   //kolor linii
    Canvas -> Brush -> Color = kolor; //kolor wypełnienia czworokącika
    Canvas -> Polygon( p, 4);
   }
  }
```

W powyższym algorytmie biblioteczna funkcja `Polygon` (wielokąt) zmusza do wprowadzenia tablicy zmiennych typu `TPoint`, ponieważ wymaga jej jako argumentu.

Do barwienia elementów wykresu wykorzystaliśmy funkcję zamieniającą amplitudy czerwieni, zieleni i błękitu na windowsową reprezentację koloru:

```
TColor RGB( int r, int g, int b);
```



Rysunek 3.7. *Wykres powierzchniowej fali kolistej, cieniowanej światłem. Liczba podziałów dziedziny, czyli liczba elementarnych wielokątów, określona w konstruktorze obiektu potomnego TWykres, celowo jest za duża. Dzięki temu lepiej widzimy strukturę płachty*

Interferencja fal na wodzie

No dobrze, ale przecież na wodzie powstają nie tylko fale koliste czy płaskie. Obraz pomarszczonej wody potrafi być bardzo złożony. W takiej sytuacji wykorzystujemy ważne prawo superpozycji ciągów falowych:

> Dowolny układ zmarszczek można wyrazić jako sumę zbioru odpowiednio dobranych, prostych fal kolistych.

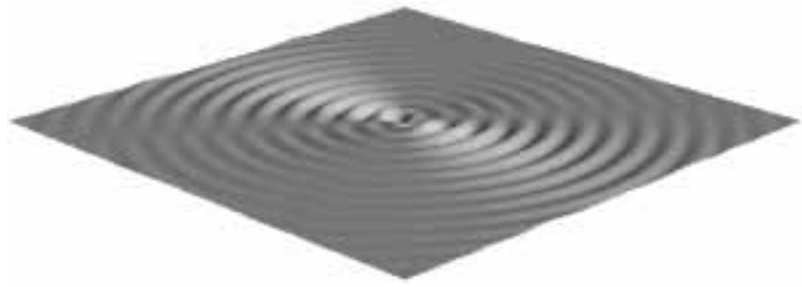
Zasada ta nosi nazwę *zasady superpozycji fal* albo *zasady Huyghensa*. U jej podstaw leży teza, że fala to jakaś mniej lub bardziej skomplikowana gra prostych wychyleń cząsteczek z położenia równowagi. A wychylenia się sumują pod warunkiem, że nie są zbyt duże.

Jeśli na powierzchni wody mamy dwa źródła fal kolistych lub więcej, jest oczywistym, że każdy konkretny punkt powierzchni będzie falował pod wpływem każdej z tych fal. Możliwe są różne sytuacje, np. punkt może falować z dużą amplitudą, o ile spotykają się tam góry (albo doliny) fal kolistych, może też zupełnie nie falować, o ile góra jednej fali wpadnie tam w dolinę drugiej.

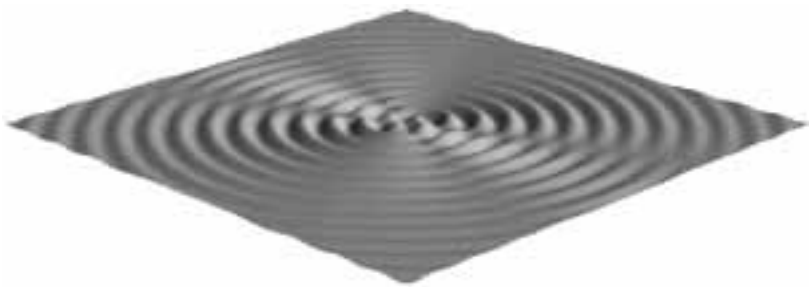
Przy jakimś nieregularnym rozkładzie źródeł fal kolistych obraz zaburzeń powierzchni będzie bardzo złożony i raczej nieciekawy. Istnieją jednak specjalne konfiguracje źródeł, którym warto się przyjrzeć.

Z najprostszą sytuacją mamy do czynienia, gdy na wodzie znajdują się dwa źródła fal kolistych, które drgają w tej samej fazie. Możemy sobie wyobrazić, że harmonicznie (znaczy sinusoidalnie) uderzamy w wodę dwuzębnym widelcem. Okazuje się, że w takiej, najprostszej sytuacji obserwujemy wielką regularność rozkładów wzmocnień i osłabień fali wypadkowej.

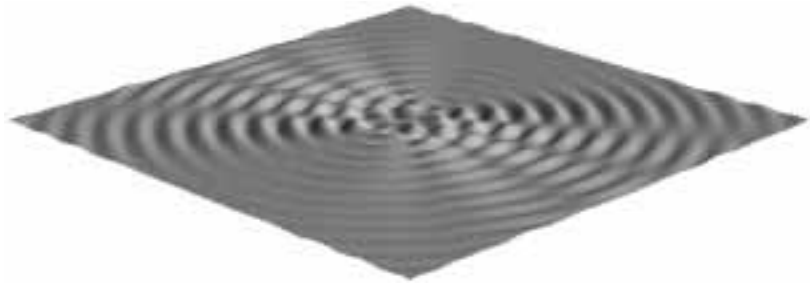
Komputerowe fotografie 3.8 – 3.10 ilustrują właśnie taką sytuację. Na kolejnych zwiększa się odległość między źródłami fal (oznaczona literą d). Proszę zauważyć, że obszary interferencji konstruktywnej, czyli wzmacniania fali, rozkładają się promieniście od centrum obrazu, w którym są umieszczone źródła fal. Obszary interferencji destruktywnej, czyli wygaszania się fal, też są rozłożone promieniście i przedzielają obszary interferencji konstruktywnej.



Rysunek 3.8. *Fala o długości 4 cm jest wzbudzana w dwóch miejscach odległych od siebie o 4 cm. Słabo widoczne są obszary interferencyjnych wzmocnień i osłabień — odległość źródeł jest zbyt mała w porównaniu z długością fali*

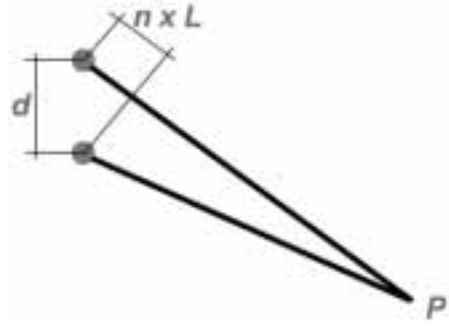


Rysunek 3.9. *Takie same fale, ale wzbudzane w miejscach odległych od siebie o 10 cm. Teraz obszary oddziaływań interferencyjnych są znacznie lepiej widoczne*



Rysunek 3.10. *Źródła fal są odległe o 20 cm. Wyraźnie widać kilka obszarów wzmocnień i osłabień, tzw. prążków interferencyjnych*

Zjawiska te mają bardzo proste wyjaśnienie: wzmocnienie fal następuje tam, gdzie spotykają się one w zbliżonej fazie (góra z górą, dolina z doliną). To z kolei następuje wtedy, gdy różnica dróg przebytych przez obie fale równa się zero lub jest wielokrotnością długości fali (rysunek 3.11).



Rysunek 3.11. *W punkcie P nastąpi wzmocnienie falowania (interferencja konstruktywna), gdy różnica dróg, po których docierają tam fale cząstkowe, będzie równa całkowitej wielokrotności długości fali L*

Program, który wytworzył te obrazy, posługuje się obiektem `TWykres3d`, a ściślej obiektem klasy potomnej. Jest to program bardzo podobny do poprzedniego. Jedynie funkcja, odpowiadająca za kształt płachty, ma tutaj postać nieco bardziej złożoną, bo opisuje dwie fale koliste, które rozchodzą się z punktów przesuniętych od siebie na odległość `d`. Obliczany oraz ostatecznie zwracany rezultat jest sumą tych dwóch fal:

```
double TWykres :: fun( double x, double y)
{
 double L = 4.0, d = 4.0;          // długość fali i odl. źródeł
 double k, r, r1, r2;

 r = sqrt( x*x+y*y);               //odległość od punktu między źródłami
 r1 = sqrt((x-d/2)*(x-d/2)+y*y);   //odległość od pierwszego źródła ...
 r2 = sqrt((x+d/2)*(x+d/2)+y*y);   // .. i od drugiego
 k = 0.5 * M_PI / L;

 return exp(-r/100.0)*( cos( k*r1) + cos( k*r2)); //wartość funkcji
}
```

Po przetestowaniu, co dzieje się, gdy dwa źródła identycznych fal kolistych stopniowo odsuwamy od siebie, spojrzymy na sytuację, w której takich źródeł jest coraz więcej.

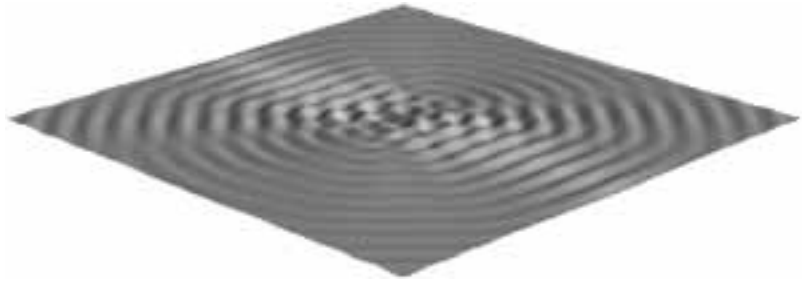
Program komputerowy wymaga jedynie drobnej zmiany w kształcie funkcji zadającej płachtę powierzchni wody:

```
double TWykres :: fun( double x, double y)
{
 double A, L = 4.0, d = 8.0, k;
 double r, r1, r2, r3, r4, r5, r6;

 r  = sqrt( x * x + y * y);
 r1 = sqrt((x - 4 * d) * (x - 4 * d) + y * y);
 r2 = sqrt((x - 2 * d) * (x - 2 * d) + y * y);
 r3 = sqrt((x - d) * (x - d) + y * y);
 r4 = sqrt((x + d) * (x + d) + y * y);
 r5 = sqrt((x + 2 * d) * (x + 2 * d) + y * y);
 r6 = sqrt((x + 4 * d) * (x + 4 * d) + y * y);
 A  = exp( -r / 100.0);
 k = 0.5 * M_PI / L;
 return A*(cos(k*r1)+cos(k*r2)+cos(k*r3)+cos(k*r4)+cos(k*r5)+cos(k*r6));
}
```

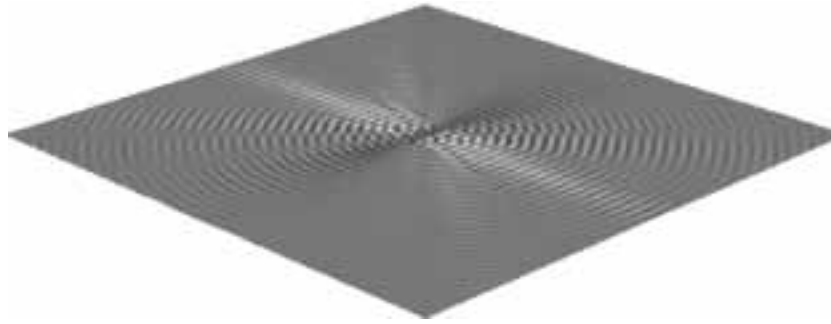
Zmiana funkcji ma charakter ilościowy, bowiem przybyło równoodległych źródeł fal.

Jak zwiększanie liczby źródeł wpływa na rozkład interferencyjnych wzmocnień i osłabień falowania? Okazuje się, że obszary wzmocnień i osłabień są tym wyraźniej zaznaczone, im więcej jest źródeł. Przede wszystkim zyskuje na sile prążek zerowego rzędu, leżący na wprost źródeł fal kolistych. Zjawisko to, wynikające wyłącznie ze wspomnianych zasad sumowania się fal przybywających do każdego, konkretnego punktu w różnych fazach, ma ogromne zastosowania w optyce, przy produkcji tzw. *siatek dyfrakcyjnych*. Wkrótce zbadamy doświadczalnie ten temat (rysunek 3.12, 3.13).

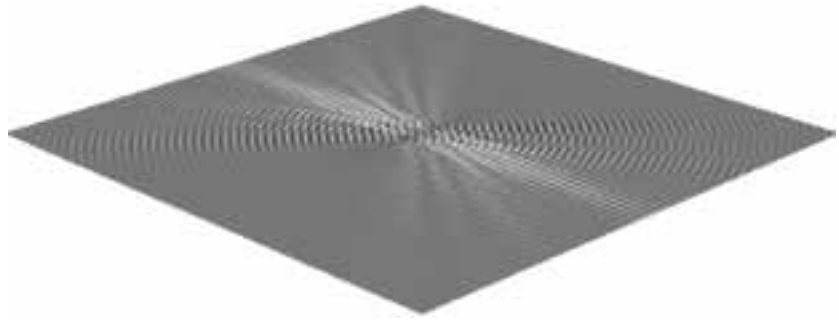


Rysunek 3.12. *Obraz powierzchni wody pobudzanej do harmonicznych drgań w pięciu równoodległych punktach*

Co się stanie, gdy przy tej samej konfiguracji źródeł fal kolistych będziemy zwiększać długość fal? Rozumiejąc istotę powstawania obrazów interferencyjnych na powierzchni, nietrudno przewidzieć odpowiedź. Okazuje się, że w miarę zwiększania długości fal, obszary wzmocnień i osłabień falowania odsuwają się od siebie (rysunek 3.14).



Rysunek 3.13. *Wraz z przybywaniem liczby punktów, źródeł fal kolistych, interferencja konstruktywna koncentruje się w obszarach przed i za otworami. Interferencja jest tym wyraźniejsza, im więcej źródeł regularnie rozmieszczonych. Scena z rysunku 3.13 jest oglądana z dalszej perspektywy, wyznaczonej pozycją obserwatora (5, 5, 3) metry*



Rysunek 3.14. *Długość fali jest dwukrotnie krótsza niż na rysunku 3.13, pozostałe parametry są bez zmian*

Na rysunku 3.14 powinniśmy zauważyć, że obszary interferencji koncentrują się na wprost układu punktów wymuszających falowanie. Zjawisko to ma wielkie znaczenie w optyce, gdzie interferują ze sobą fale świetlne, np. wypuszczane z milionów otworków na powierzchni siatki dyfrakcyjnej. Fale światła o różnej długości (czyli różnej barwie) interferują konstruktywnie w różnych miejscach. Siatka dyfrakcyjna, dzięki zjawisku, które właśnie zbadaliśmy, dokonuje rozszczepienia światła na barwy spektralne. Na tej podstawie astronom potrafi odpowiedzieć, jaki jest skład chemiczny świecącej materii.