helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ

SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

# C++ Builder.

## 20 efektownych programów

Autor: Andrzej Stasiewicz
ISBN: 83-7197-656-9
Nośnik: CD
Liczba stron: 224

Książka zawiera bardzo przystępne opisy niezwykle efektownych zjawisk z pogranicza różnych nauk przyrodniczych oraz ich komputerowe realizacje w dialekcie C++ Builder firmy Borland. Stanowi ona zbiór ćwiczeń do wykorzystania na szkolnym kółku komputerowym, ale zapewne zainteresuje też wykładowców i studentów kierunków przyrodniczych. Od Czytelnika wymagamy wiedzy na poziomie szkoły średniej, a niekiedy zaledwie gimnazjum. Opi-sywane zagadnienia często są ledwie zarysowane i pozostawiają Czytelnikowi ogromne możliwości dalszego, samodzielnego eksperymentowania.
W realizacji pomysłów Autor po mistrzowsku posługuje się najprostszym, a przy tym w pełni obiektowym i bardzo nowoczesnym sposobem programowania komputerów.

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

# Spis treści


Notka wydawnicza ... 5
Wstęp ... 7
Rozdział 1. Dywany na ustalonej powierzchni ... 9
Rozdział 2. Grafika rozpinanej nici ... 17
Rozdział 3. Serwetka z cykloid ... 23
Rozdział 4. Skalowanie ... 31
Rozdział 5. Składanie drgań poprzecznych ... 43
Rozdział 6. Sumowanie drgań ... 51
Rozdział 7. Dywany iterowane ... 61
Rozdział 8. Dywany afiniczne ... 69
Rozdział 9. Grafika układu współrzędnych ... 87
Rozdział 10. Konkurencja międzygatunkowa ... 97
Rozdział 11. Przyszłość nie do przewidzenia ... 111
Rozdział 12. Algorytm barwy fizycznej ... 119
Rozdział 13. Grafika wykładników Lapunowa ... 127
Rozdział 14. Fraktal Mandelbrota ... 137
Rozdział 15. Eksplorator Mandelbrota ... 147
Rozdział 16. Otwarty kosmos ... 161
Rozdział 17. Gra w życie ... 171
Rozdział 18. Epidemia ... 185
Rozdział 19. Mrowisko pełne automatów ... 201
Rozdział 20. Jednowymiarowy automat komórkowy ... 215

## Rozdział 2.
# Grafika rozpinanej nici

Jest pod Białymstokiem artystka, która niegraficznymi technikami tworzy dziwne grafiki. Wbija ta niemłoda już babcia setki gwoździków, a potem rozpina na nich kolorowe nitki. Nie jest to jednak takie łatwe, jak mogłoby się wydawać — artystce z pewnością należy się uznanie. Siłą tych obrazów jest ich matematyczna precyzja — drobne niejednorodności w prowadzeniu nici, nierównomierne odstępy nasze oczy wychwytują natychmiast.

Będziemy wbijać gwoździki w wirtualną deskę wirtualnego obrazu. Wbić gwoździk będzie znaczyło tyle, co wyliczyć jakimś algorytmem jego współrzędne *(x, y)*. Rozpiąć nitkę między dwoma gwoździkami będzie znaczyło tyle, co pociągnąć kolorową linię od jednego punktu do drugiego.

Algorytm rozpinania nici musi mieć jakiś taki kształt:

```
FUNKCJA NITKI()
```

**STAŁE:**

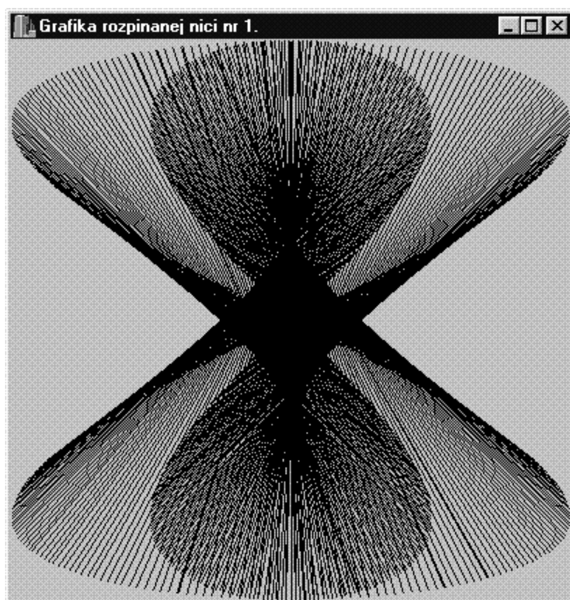
```
int MAX_IL = 1000;
```

**ZMIENNE:**

```
int i
REAL R, x1, x2, y1, y2;

R = 100;
for i = 0 to MAX_IL
   x1 = R * sin( i / 20.) * cos( i / 20.);
   y1 = R * cos( i / 20.);
   x2 = -2 * R * sin( i / 20.) * cos( i / 20.);
   y2 = -R * cos( i / 20.);
   Color = RED;
   Line( x1, y1, x2, y2);
next i
```

Po wstępnych deklaracjach, ustaleniu liczby rozpinanych nici *MAX_IL*, wyliczamy cztery współrzędne dwóch gwoździków, a potem rozpinamy linię — nitkę między nimi. Zmienna *R* oraz cała ta plątanina funkcji trygonometrycznych to kaprys programisty. Cała sztuka polega na dobraniu takich formuł na cztery współrzędne końców linii, by zamknięta w pętli całość złożyła się na miłą oku grafikę. Znów nie ma żadnych reguł, gwarantujących sukces artystyczny. Nazwijmy to programowaniem eksperymentalnym...

Spróbujmy zaimplementować nasz algorytm w dialekcie C++ Buildera. Jak zwykle, zaczynamy od wydania polecenia *New Application*, po którym Builder oczyszcza swoje wnętrze z dotychczasowych programów i jest gotów do pracy nad nowym zagadnieniem.




**Rysunek 6.** *Babcia wbiła w wirtualną deskę 1000 gwoździków i rozpięła między nimi 500 czarnych nici. Pozycje gwoździków nie są przypadkowe — dostarcza je niezbyt złożona kombinacja funkcji trygonometrycznych.*

W okienku edytora, które na początek jest ukryte pod okienkiem z formą, od razu doklejmy nagłówek modułu z algorytmami matematycznymi. Oto fragment kodu, który powinniśmy najpierw zlokalizować w górnej części pliku *CPP*, potem uzupełnić o frazę doklejania nagłówka:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "fnici1.h"
#include "math.h"
...
```

Grafikę oczywiście umieścimy w uzgodnionej z systemem operacyjnym funkcji — reakcji na zdarzenie *OnPaint* — chcę rysować. Tylko wtedy nasz program automatycznie odnowi swoją grafikę, gdy jego okienko nagle wyłoni się spod Worda czy Excela. Odszukajmy więc *Inspektora obiektów*, przejdźmy na jego zakładkę *Events* — zdarzenia — i dwukrotnym kliknięciem wygenerujmy funkcję — reakcję na zdarzenie *OnPaint*. Gdy Builder wykreuje puste ciało tej funkcji, niezwłocznie spiszmy jej algorytm, przekładając wcześniejsze, ogólne frazy na dialekt C++ Buider:

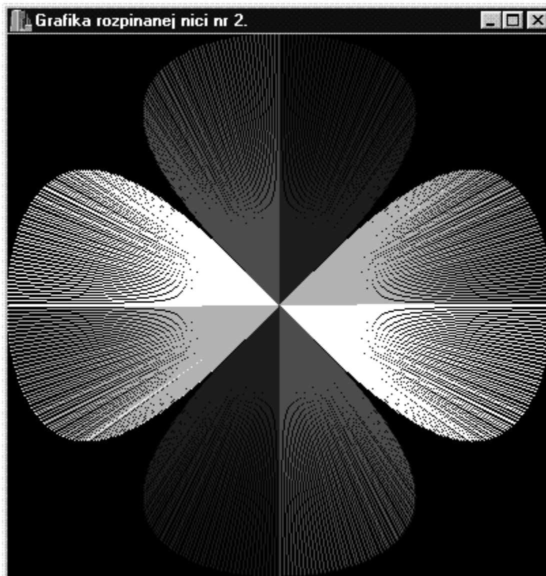
```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
 int i;
 double x1, x2, y1, y2, R, A, B;
 R = ClientWidth / 2;
 A = ClientWidth / 2;
 B = ClientHeight / 2;

 for( i = 0; i < 500; i ++)
 {
   x1 = R * sin( i / 20.) * cos( i / 20.);
   y1 = R * cos( i / 20.);
   x2 = -2 * R * sin( i / 20.) * cos( i / 20.);
   y2 = -R * cos( i / 20.);

   Canvas -> MoveTo( x1 + A, y1 + B);
   Canvas -> LineTo( x2 + A, y2 + B);
 }
}
```

Jest to w zasadzie ten sam algorytm, ale wypowiedziany w innym języku i osadzony w konkretnym okienku. Zmienne *R*, *A* i *B* mają znaczenie czysto techniczne — *R* rozciąga grafikę, *A* i *B* ją pozycjonuje w okienku Windows. Parametry *ClientWidth* i *ClientHeight* zadają rozpiętość graficznej powierzchni okienka. Zauważmy też, że u Borlanda nie ma czteroargumentowej funkcji *Line()*, za to jest para funkcji *MoveTo()* — idź do punktu i *LineTo()* — ciągnij stamtąd linię. Ta para funkcji z powodzeniem zastępuje klasyczną funkcję *Line()*.

Pora na wskazanie kilku możliwości modyfikacji algorytmu. Przede wszystkim włóżmy trochę koloru w grafikę rozpinanej nici. Mam taki pomysł:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
 double t, x1, x2, y1, y2, R, A, B;
 R = ClientWidth / 2;
 A = ClientWidth / 2;
 B = ClientHeight / 2;

 for( t = -M_PI; t < M_PI; t += M_PI/200)
 {
   x1 = R * sin( t) * cos( t);
   y1 = R * cos( t);
   x2 = -R * sin( t) * cos( t);
   y2 = -R * cos( t);

   Canvas -> Pen -> Color = ( t < M_PI/2 ? clBlue : clRed);
   Canvas -> MoveTo( x1 + A, y1 + B);
   Canvas -> LineTo( x2 + A, y2 + B);

   x1 = R * sin( t);
   y1 = R * cos( t) * sin( t);
   x2 = -R * sin( t);
   y2 = -R * cos( t) * sin( t);
```

```
        Canvas -> Pen -> Color = ( t < M_PI/2 ? clWhite : clAqua);
        Canvas -> MoveTo( x1 + A, y1 + B);
        Canvas -> LineTo( x2 + A, y2 + B);
    }
  }
```




**Rysunek 7.** *Czarne tło uzyskałem, opracowując w Inspektorze obiektów właściwość o nazwie Color. Modyfikacja algorytmu wbijania gwoździków polega na innej organizacji pętli for() — teraz pętla nie liczy linii, a przebiega jakoś zakreśloną dziedzinę, akceptowalną przez wykorzystane tutaj funkcje trygonometryczne. Stała o nazwie M_PI (liczba pi) jest z dużą dokładnością zdefiniowana w doklejanym nagłówku matematycznym math.h.*

Co się tutaj zmieniło? Algorytm jest dwa razy dłuższy — w każdym obiegu pętli kreślimy nie jedną, a dwie linie. Sama pętla przebiega od wartości *–3.14* do *+3.14* (symbol *M_PI* to zdefiniowana w pliku nagłówkowym *math.h* liczba *pi*) ze skokiem równym *pi/200*. Dlaczego taki zakres pętli? Nie wiadomo — proszę spróbować rozegrać to inaczej. Na tym między innym polega wielka sztuka — trzeba coś zrobić po swojemu.

Przed wykreśleniem każdej z dwóch linii pojawia się fraza dobierania koloru. Za kolor linii w aparacie *Canvas* (płótno malarskie) odpowiada obiekt *Pen* (pióro), który z kolei ma zmienną *Color*. Kolor ustala bardzo sprytna i chętnie wykorzystywana instrukcja warunkowego przypisania:

```
...
Canvas -> Pen -> Color = ( t < M_PI/2 ? clBlue : clRed);
...
```

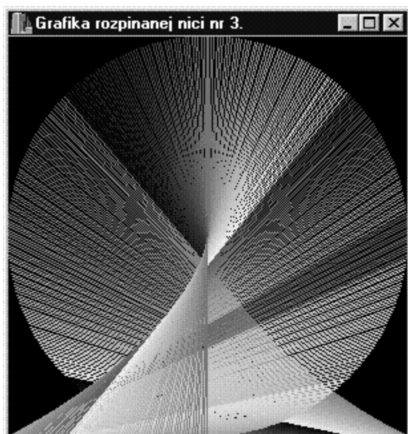
Dlaczego lubimy tę instrukcję? Bo daje się wbudowywać bezpośrednio w wyrażenia. Za pomocą klasycznego warunku logicznego powyższe zapisalibyśmy np. tak:

```
...
if( t < M_PI/2)
   Canvas -> Pen -> Color = clBlue;
```

```
else
   Canvas -> Pen -> Color = clRed;
...
```

co oczywiście też jest dobrą, choć mniej profesjonalną, szkołą programowania.

Inny pomysł to wprowadzenie linii barwionej nie kolorem dobranym arbitralnie, a wyliczanym w smakowitej funkcji *RGB()*. Konieczne do wykonania modyfikacje są naprawdę proste:




**Rysunek 8.** *Nasza babcia — artystka — w swoją grafikę zaczęła wplatać kolorowe nici. Tu powoli kończy się analogia z realem (tak znajomi nazywają ten świat za oknem...). Nie ma kompletu nici o barwach tożsamych i równie bogatych, co kolorystyka funkcji RGB().*

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
 double t, x1, x2, y1, y2, R, A, B;
 int r, g, b;

 R = ClientWidth / 2;
 A = ClientWidth / 2;
 B = ClientHeight / 2;

 for( t = -M_PI; t < M_PI; t += M_PI/200)
 {
   x1 = R * sin( t);
   y1 = R * cos( t);
   x2 = -R * sin( t);
   y2 = R;

   r = x1 - y2;
   g = y1 - x2;
   b = x1 - y1;
   Canvas -> Pen -> Color = (TColor)RGB( r, g, b);
   Canvas -> MoveTo( x1 + A, y1 + B);
   Canvas -> LineTo( x2 + A, y2 + B);
 }
}
```

Wyliczanie współrzędnych pod gwoździki zawiera szczególnie proste wyrażenia. Nie będziemy tego omawiać, bo jest to pole do popisu dla domowych programistów, a w dodatku nie mam godnej polecenia recepty, co należałoby tam wpisać.

W algorytmie pojawiły się trzy dodatkowe zmienne — amplitudy trzech barw podstawowych. W powyższym programie każda z tych amplitud jest jakąś funkcją współrzędnych gwoździków. Nad sposobem wyliczania amplitud koloru, podobnie jak nad sposobem znajdowania miejsc na gwoździki, niewątpliwie należy solidnie popracować.

## Zadania i problemy

1. Stała *M_PI* jest zadeklarowana i zainicjowana w pliku *math.h*. Spróbuj odszukać ten plik w katalogu Buildera, otwórz go w edytorze, obejrzyj. Tylko niczego tam nie popsuj!
2. Trzymamy się tutaj kurczowo funkcji trygonometrycznych, co jest uzasadnione ich przewidywalnym przebiegiem, ale jeszcze są funkcje: *log()* (logarytm), *fabs()* (wartość bezwzględna), *exp()* (funkcja wykładnicza), *sqrt()* (pierwiastek kwadratowy). Czy uda Ci się coś z tego tworzywa zbudować?

## Rozwiązania

1. W katalogu, w którym jest zainstalowany Builder, prawdopodobnie (podczas instalacji można zmieniać katalogi) znajduje się podkatalog o nazwie *Include*. Jest tam plik *math.h*. Jego odpowiednik z algorytmami, prawdopodobnie o nazwie *math.cpp*, nie jest w wersji źródłowej udostępniany przez Borlanda — ot mają tam pewnie jakieś tajemnice implementacyjne. Jest dostarczany w postaci skompilowanej, tak by już nikt nie był w stanie tego rozszyfrować.
2. Z pewnością, ale nie wiem jak (informatyka doświadczalna!). Należy jedynie uważać, by pętla podawała takie wartości, które będą do przyjęcia dla plątaniny tych funkcji.

# Rozdział 12.
# Algorytm barwy fizycznej

W poprzednich rozdziałach poznaliśmy system dobierania koloru do wykreślania jakiegoś obiektu graficznego. Centralne znaczenie ma tam funkcja *RGB()*, sterująca natężeniem trzech niezależnych barw kineskopowych:

```
...
kolor = RGB( 100, 100, 100);
...
```

Funkcja ta oczekuje trzech argumentów — natężeń czerwieni, zieleni i błękitu. Oprócz funkcji *RGB()*, mogącej zsyntetyzować praktycznie dowolny kolor, mamy w Builderze zestaw indywidualnie nazwanych barw, np. *clRed*, *clGreen*, *clYellow*. Cały zbiór nazw możemy zobaczyć w *Inspektorze obiektów*, ot choćby we właściwości *Color*, należącej do formy.

Mimo tego bogactwa zaopatrzony w kolorowy monitor przyrodnik napotka prędzej czy później poważną trudność — nie będzie potrafił uzyskać koloru fizycznego, wypełniającego cały otaczający nas świat. Kolor fizyczny określa się długością fali światła albo — alternatywnie — jej częstotliwością. Światło czerwone ma dłuższe fale niż światło żółte. Z kolei światło żółte ma dłuższą falę niż fiolet. Gdyby wreszcie wykreślić barwy wszystkich długości fal (wszystkich oczywiście się nie da), otrzymalibyśmy na ekranie tęczę. Tymczasem istniejący, biblioteczny aparat koloru w żaden sposób nie umożliwia nam wykreślenia tęczy.

Gdzie leży zasadnicza trudność? Monitory posługują się trzema barwami podstawowymi, gdyż jakimś cudem trzy zmieszane barwy podstawowe potrafią doskonale oszukać nasze przebiegłe mózgi. Wrażenie uzyskane w wyniku wpuszczenia do oka światła czerwonego i zielonego w pełni odpowiada sytuacji, gdyby wpuszczono tam światło czysto żółte. Mieszanina czerwieni i zieleni jest dla oka tym samym, czym żółć. Dla oka, ale nie dla Natury. Jak fachowo mówimy, oko nie ma właściwości spektralnych — nie potrafi analizować mieszanin barw. Wystarczy spojrzeć na ekran przez spektroskop, by odkryć najważniejszą mistyfikację dwudziestego wieku: kolor żółty wcale nie jest żółty. Mało który kolor jest prawdziwy. W naszych komputerach i monitorach tylko pięć kolorów jest prawdziwych.

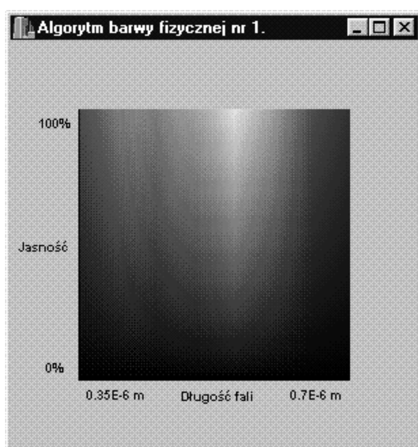
Napiszemy nowy obiekt o nazwie *TWidmo*. Obiekt *TWidmo* będzie dostarczał barwę spektralną (czystą). Wystarczy podać długość fali światła — byle z zakresu widzialnego — a obiekt *TWidmo* wytworzy odpowiednią barwę. Czy barwa ta będzie rzeczywiście czysta? Jasne, że nie. Wszak nasze kineskopy mają tylko trzy barwne działka elektronowe. Barwa nie będzie czysta, ale uzyskiwane wrażenie — tak.

Najważniejszym elementem obiektu *TWidmo* będzie publiczna funkcja o następującym prototypie (nagłówku):

```
TColor lambda_to_kolor( int jasnosc, double lambda);
```

W fizyce symbolem *lambda* zwyczajowo określa się długość fali. Zatem powyższa funkcja ma przetworzyć rzeczywistą długość fali w kolor. Parametr *jasnosc* dodatkowo pozwoli operować jaskrawością światła, zwiększając lub zmniejszając intensywność koloru, ale tak, by nie popsuć barwy. Jeśli komuś z Państwa uda się dobrze napisać i udokumentować to, o czym za chwilę powiemy, myślę że Borland to kupi.

Algorytmy nowego obiektu lokujemy w nowym module — robiliśmy to już dwukrotnie, rozpoczynając prace nad obiektami *TSkalowanie* i *TDiagram*. Bardzo ważna jest nazwa plików takiego modułu — powinna być czytelna, by w przyszłości już na pierwszy rzut oka było oczywiste, co znajduje się w tym module. Niech nazwa ta brzmi *widmo*.




**Rysunek 51.** *Zastanówmy się przez chwilę, co leży u źródeł naszych kłopotów z tęczą. Nasze oczy postrzegają fale elektromagnetyczne z zakresu od 0.4 do 0.65 mikrometra. Dzięki zróżnicowanej budowie receptorów potrafią też rozróżniać długości fal z powyższego zakresu — różnice te postrzegamy jako barwę (ale zauważmy, że oczy łatwo dają się oszukać — istnieje np. czysta zieleń spektralna (0.5 mikrometra) i taka sama co do koloru mieszanina światła niebieskiego i żółtego). To, że poszczególnym długościom fal elektromagnetycznych odpowiadają takie, a nie inne kolory, jest już sprawą naszych mózgów i zawartych tam algorytmów interpretujących elektryczne bodźce pochodzące z oka. Jest nawet prawdopodobne, że każdy człowiek inaczej widzi kolory. Co jest czerwone, a co niebieskie, uzgodniliśmy wiele lat temu tylko i wyłącznie drogą ustnej wymiany poglądów na ten temat. Tymczasem komputery dysponują trzema drutami prowadzącymi do monitora. Biegną po nich trzy amplitudy kolorów, z których da się zbudować każdą barwę, albo lepiej — którymi da się oszukać każdy mózg, tak by myślał, że widzi pełną paletę. Te kolory to: czerwony, niebieski i zielony. Taka triada nazywa się barwami dopełniającymi. Moglibyśmy nazwać je też barwami bazowymi. Wystarczy wziąć lupę i spojrzeć na trójkolorową strukturę pojedynczego punktu ekranu, by doświadczalnie zweryfikować te słowa. Jeśli chodzi o kolory, to jesteśmy brutalnie oszukiwani.*

Oto treść pliku *widmo.h*, zawierającego deklarację nowej klasy:

```
//---------------------------------------------------------------------------
#ifndef widmoH
#define widmoH
//---------------------------------------------------------------------------
// Obiekt przeliczający fizyczną długość fali na amplitudy R, G, B.
// Zaimpementowano też jasność światła, zadawaną procentowo (od 0 do 100).
class TWidmo
{
 private:
    int tecza[ 15][ 3];          //rozkład wybranych punktów tęczy na składniki RGB
    double Askal, Bskal, Cskal;  //współczynniki pomocnicze (skalujące)

 public:
    TWidmo( void);                       //konstruktor
    TColor lambda_to_kolor( int jasnosc, double L);
    double LAMBDA_MIN, LAMBDA_MAX;//skrajne długości fal światła widzialnego
};
#endif
```

Powyższa klasa zapowiada dwie funkcje publiczne — jak zwykle oczekiwanego konstruktora i równie oczekiwanej funkcji konwertującej fizyczną długość fali na borlandowski kolor. Konstruktor nie ma parametrów — to się zdarza w sytuacjach, w których nie ma nic do zainicjowania. Bezparametrowy konstruktor nie jest czymś niezwykłym. Ale w tym zagadnieniu konstruktor, choć bezparametrowy, będzie musiał wykonać niezwykle istotną część pracy — będzie musiał zainicjować tablicę *tecza[][]*, gdzie umieścimy definicje 15 kluczowych barw tęczy.

Publiczne parametry *LAMBDA_MIN* i *LAMBDA_MAX* to prezent dla użytkowników naszej klasy. Parametry te opisują fizyczne granice widzialnej części widma fal elektromagnetycznych. Zostały upublicznione, bo może komuś się przydadzą.

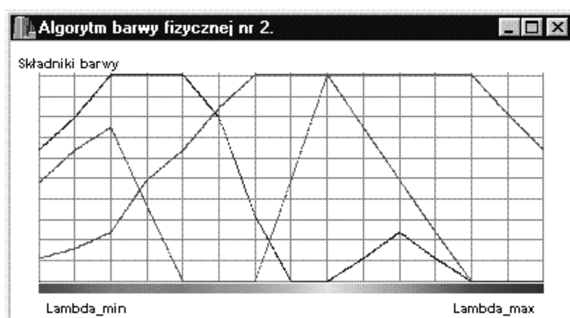
Takiej deklaracji należało się spodziewać. Zobaczmy zatem, jak zrealizowano szczegóły implementacyjne obiektu *TWidmo*, czyli zajrzyjmy do pliku *CPP*:

```
#include <vcl.h>
#pragma hdrstop

#include "widmo.h"
//---------------------------------------------------------------------------
// Konstruktor inicjuje wewnętrzny ustrój obiektu.
TWidmo :: TWidmo( void)
{
 int il_czysty_kolor = 15;               //liczba wejść do tabeli tecza

 LAMBDA_MIN = 0.35E-6;           //granica nadfioletowa
 LAMBDA_MAX = 0.7E-6;            //granica podczerwona
 Askal = (double)(il_czysty_kolor-1) / (LAMBDA_MAX - LAMBDA_MIN);
 Bskal = -Askal * LAMBDA_MIN;
 Cskal = 4. / 100.;                      //skalowanie jasności

 tecza[0][0] = 30; tecza[0][1] = 7; tecza[0][2] = 40;
 tecza[1][0] = 40; tecza[1][1] = 10; tecza[1][2] = 50;
 tecza[2][0] = 47; tecza[2][1] = 15; tecza[2][2] = 63;
 tecza[3][0] = 23; tecza[3][1] = 31; tecza[3][2] = 63;
```

```
 tecza[4][0] = 0; tecza[4][1] = 40; tecza[4][2] = 63;
 tecza[5][0] = 0; tecza[5][1] = 53; tecza[5][2] = 50;
 tecza[6][0] = 0; tecza[6][1] = 63; tecza[6][2] = 20;
 tecza[7][0] = 31; tecza[7][1] = 63; tecza[7][2] = 0;
 tecza[8][0] = 63; tecza[8][1] = 63; tecza[8][2] = 0;
 tecza[9][0] = 63; tecza[9][1] = 47; tecza[9][2] = 7;
 tecza[10][0] = 63; tecza[10][1] = 31; tecza[10][2] = 15;
 tecza[11][0] = 63; tecza[11][1] = 15; tecza[11][2] = 7;
 tecza[12][0] = 63; tecza[12][1] = 0; tecza[12][2] = 0;
 tecza[13][0] = 51; tecza[13][1] = 0; tecza[13][2] = 0;
 tecza[14][0] = 40; tecza[14][1] = 0; tecza[14][2] = 0;
}
//---------------------------------------------------------
// Najważniejsza funkcja, zamieniająca L i jasność w kolor
TColor TWidmo :: lambda_to_kolor( int jasnosc, double L)
{
 int pierwszy_kolor, r, g, b;
 double prawdziwy_kolor, ulamek_drugiego, a1, a2;
 double jasn;

 if( L <= LAMBDA_MIN || L >= LAMBDA_MAX)
   return clBlack;                              //poza zakresem widzenia

 if( jasnosc > 100)                             //zbyt duża jasność
    jasnosc = 100;
 if( jasnosc < 0)                               //zbyt mała jasność
    jasnosc = 0;

 jasn = Cskal * jasnosc;               //przeskalowanie jasności

 prawdziwy_kolor = Askal * L + Bskal;
 pierwszy_kolor = (int)prawdziwy_kolor; //część całkowita, np. 7
 ulamek_drugiego = prawdziwy_kolor - pierwszy_kolor;//ułamek 2. koloru, np. 0.35

 a1 = tecza[ pierwszy_kolor][ 0] * jasn;
 a2 = tecza[ pierwszy_kolor + 1][ 0] * jasn;
 r = (int)( a1 + ulamek_drugiego * (a2 - a1));

 a1 = tecza[ pierwszy_kolor][ 1] * jasn;
 a2 = tecza[ pierwszy_kolor + 1][ 1] * jasn;
 g = (int)( a1 + ulamek_drugiego * (a2 - a1));

 a1 = tecza[ pierwszy_kolor][ 2] * jasn;
 a2 = tecza[ pierwszy_kolor + 1][ 2] * jasn;
 b = (int)( a1 + ulamek_drugiego * (a2 - a1));
 return RGB( r, g, b);
}
```

Konstruktor, jak to czynią wszystkie konstruktory, inicjuje wewnętrzne (czyli prywatne) zmienne obiektu. Najważniejsze jest zainicjowanie tablicy *tecza[][]*, zawierającej definicje 15 kluczowych kolorów tęczy.

Ciekawsze rzeczy dzieją się w funkcji transformującej fizyczną długość fali światła i jego jasność w kolor.




**Rysunek 52.** *Spektralnie czystą tęczę, gdzie każdy kolor ma ściśle określoną długość fali, udajemy odpowiednio dobranymi mieszankami barw czerwonej, zielonej i niebieskiej. Jak dobrać amplitudy tych trzech barw w każdym punkcie tęczy? Jak skonstruować umieszczoną w konstruktorze TWidmo() tabelę definiującą rozkład tęczy na barwy bazowe? Trzeba zawołać kilkoro dzieci i posadzić je przed jakimś programem typu Paint czy Corel. Są tam narzędzia, które pozwalają na budowę własnego koloru, przy okazji pokazując zawartość sygnałów RGB. Dzieci doskonale wiedzą, jak powinien wyglądać spektralny kolor pomarańczowy czy fioletowy. Nam pozostaje wynotować amplitudy R, G i B, składające się na takie specjalne kolory. Amplitudy te wbudowujemy do tabeli tecza[][].*

*Widoczny tutaj wykres, w istocie skonstruowany doświadczalnie, pokazuje jakąś prawdę o fizjologii widzenia barw. Tęczę na dolnej osi wykreślono algorytmem opisanym w tekście.*

Idea algorytmu jest taka: mamy jakąś długość fali światła *lambda* i szukamy trójki liczb *R*, *G*, *B*, które po wymieszaniu dadzą wrażenie światła o długości fali *lambda*. Wchodzimy z wartością tej długości fali do tablicy *tecza[ ][ ]* i szukamy najbliższej jej trójki amplitud *R*, *G*, *B*. Jeśli najbliższej nie ma, aproksymujemy amplitudy pomiędzy dwiema sąsiednimi trójkami. Wyliczamy przybliżone wartości *R*, *G* i *B*. Jeśli potrzebny nam kolor pomiędzy pomarańczowo-czerwonym a czerwonym, bierzemy wartości amplitud *R*, *G*, *B* gdzieś spomiędzy obu tych trójek.

Współczynniki *Askal* i *Bskal* mają tak dobrane wartości, by fraza:

```
prawdziwy_kolor = Askal * L + Bskal;
```

dostarczyła numer koloru z zakresu od *0* do *15*. Powinno się to kojarzyć nam z 15 elementami tablicy *tecza[][]*. Zmienna *prawdziwy_kolor* jest więc numerem trójki *R*, *G*, *B* w tablicy *tecza[ ][ ]* (ale zazwyczaj z ogonkiem ułamkowym, który posłuży do korekty barwy), zmienna *L* to długość fali światła.

Współczynnik *Cskal* zamieni procentową wartość jaskrawości na większy lub mniejszy zestaw amplitud *R*, *G*, *B*. Ma tak dobraną wartość, by najwyższa amplituda z tablicy *tecza[ ][ ]* (czyli *63*) przemnożona przez najwyższą jaskrawość (czyli *100%*) mieściła się w zakresie tolerowanym przez biblioteczną funkcję *RGB()*:

```
(Cskal * jasność * tęcza[][]) <= 255
```

Funkcja *lambda_to_kolor()* wykorzystuje wartości wcześniej zainicjowanych współczynników i nie traci już czasu na nic, poza przeliczeniem rzeczywistej wartości długości fali w naturalne wartości amplitud barw składowych i zebranie tych składowych do ostatecznego koloru.

Najpierw następuje kontrola, czy podana długość fali wypada gdzieś w widzialnym skrawku nieskończonego widma fal elektromagnetycznych. Jeśli długość fali nie mieści się w zakresie widzialnym, zwracanym kolorem jest czerń — nic nie widać.

Potem kontrolujemy, czy jaskrawość mieści się w przyzwoitym zakresie procentowym i wyliczamy pomocniczy współczynnik o nazwie *jasn*. Wreszcie obliczamy wejście do tablicy kolorów *tecza[ ][ ]*:

```
...
prawdziwy_kolor = Askal * L + Bskal;            //np. 7.35
pierwszy_kolor = (int)prawdziwy_kolor;          //część całkowita, np. 7
ulamek_drugiego = prawdziwy_kolor - pierwszy_kolor;//ułamek, np. 0.35
...
```

Zmienna *prawdziwy_kolor* zazwyczaj jest wartością ułamkową. Jej część całkowita, uchwycona w zmiennej *pierwszy_kolor*, oznacza czysty kolor z tablicy *tecza[ ][ ]*. Jednak *prawdziwy_kolor* ma także część ułamkową — złamanie barwy w stronę następnego elementu tablicy *tecza[ ][ ]*. Tak oto dokonujemy złamania czystego koloru ułamkiem drugiego:

```
...
a1 = tecza[ pierwszy_kolor][ 0] * jasn;
a2 = tecza[ pierwszy_kolor + 1][ 0] * jasn;
R = (int)( a1 + ulamek_drugiego * (a2 - a1));
...
```

Powyższy fragment algorytmu dotyczył wyliczenia amplitudy czerwieni, stąd indeks zero w elementach tablicy *tecza[ ][ ]*. Potem jeszcze powtarzamy tę recepturę dla zieleni i błękitu. Wreszcie syntetyzujemy ostateczny kolor.

Tak naprawdę, sztuką nie jest napisanie powyższego programu. Jeśli ktoś dobrze przemyślał obiekt *TSkalowanie*, nie ma tu wiele do roboty. Jakaś tajemnica tkwi jednak w tablicy *tecza[][]*, definiującej 15 kolejnych barw z tęczy w rozkładzie na komputerowe amplitudy czerwieni, zieleni i błękitu. W tablicy tej krzyżują się dwa światy: prosty, logicznie skonstruowany świat komputerów i cała nasza niezbadana fizjologia, która falę o takiej a takiej długości każe postrzegać jako taką a taką barwę. Dobre tablice *tecza[][]* pewnie są sporo warte dla grafika — przyrodnika, który potrzebuje barwy światła o długości fali, powiedzmy *0.000000512 m*. Skąd wziąłem tę tablicę? Znalazłem w jakiejś książce fotografię tęczy. Leciutko, by nie niszczyć książki, narysowałem na tęczy 15 równoodległych linii. Mówiąc inaczej, zaznaczyłem na niej 15 spektralnych barw bazowych. To był pierwszy punkt mojego chytrego algorytmu.

W drugim punkcie poprosiłem o pomoc kilkoro małych dzieci. Posadziłem je przed komputerem i położyłem przed nimi fotografię tęczy z zaznaczonymi punktami. Na ekranie był program, umożliwiający syntezę barwy z nastaw trzech suwaków, ot taki mieszacz kolorów. Dzieci miały zadanie dobrania takiego położenia suwaków czerwieni, zieleni i błękitu, by wymieszany kolor odpowiadał kolorowi z zaznaczonego punktu tęczy. Wystarczyło wynotować nastawy suwaków i wpisać je do tablicy *tecza[][]*.

Przy okazji zauważyłem ciekawą właściwość małych dzieci — dokładnie wiedzą, kiedy kolor jest odpowiedni i kiedy trzeba przerwać poszukiwania. Dorosły poprawia, marudzi, doskonali i traci czas. Dziecko po prostu wie.

Każdy dopiero co napisany obiekt trzeba przetestować. W funkcji — reakcji na zdarzenie *OnPaint* — umieścimy algorytm kreślenia wszystkich barw tęczy we wszystkich jaskrawościach. Zamalujemy tęczą wydzielony z okienka, prostokątny obszar. Oto treść funkcji:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
 const int xe0 = 50, ye0 = 50, eszer = ClientWidth - 100, ewys = ClientHeight -
100;
 double lambda, jasn;
 int x, y;
 TWidmo widmo;
 double widmo_szer = widmo.LAMBDA_MAX - widmo.LAMBDA_MIN;
 TSkalowanie skal( xe0, ye0, eszer, ewys,
                  widmo.LAMBDA_MIN + widmo_szer / 2, 50, widmo_szer, 100);
 for( x = xe0; x < xe0 + eszer; x ++)
 {
   lambda = skal.daj_real_x( x);
   for( y = ye0; y < ye0 + ewys; y ++)
   {
      jasn = skal.daj_real_y( y);
      Canvas -> Pixels[ x][ y] = widmo.lambda_to_kolor( jasn, lambda);
   }
 }
}
```

Ten program wymaga dołączenia do projektu plików modułów *skala* i *widmo* — pamiętajmy zatem o wstępnych operacjach *Add File to Project* — dodaj pliki do programu. Nie zapomnijmy o doklejeniu dwóch nagłówków, które zapowiedzą kompilatorowi, co oznaczają napisy *TSkalowanie* i *TWidmo*.

W funkcji — reakcji na *OnPaint* — najpierw parametryzujemy obszar okienka przeznaczony pod grafikę, ot taki rodzaj porządku. Potem deklarujemy zmienną typu *TWidmo* — jest to oczywiście zmienna obiektowa, złożona. Najważniejsze jest utworzenie zmiennej typu *TSkalowanie* — w jej konstruktorze mamy precyzyjne opisanie teatru ekranowego i fizycznego. Teatr ekranowy nie wymaga specjalnych komentarzy — jest to okienko o miłych oku rozmiarach i położeniu. Teatr rzeczywisty natomiast zawiera się między *LAMBDA_MIN* i *LAMBDA_MAX* w poziomie oraz między wartością *0* i *100* w pionie. W poziomie odłożymy wszystkie możliwe długości światła widzialnego, w pionie wszystkie możliwe jaskrawości światła.

Potem mamy popis możliwości obiektu skalującego. Dwie pętle przebiegają punkt po punkcie cały obszar okienkowy, ale następuje transformacja współrzędnych każdego punktu ekranowego do długości fali i do jaskrawości. Parametry te służą do wyliczenia barwy, czyli do przetestowania poprawności działania obiektu *TWidmo*.

## Zadania i problemy

1. Obiekt *TWidmo* nie nadaje się do pracy z grafiką 256-kolorową (bo w tym systemie mamy kolory arbitralnie ponumerowane, nie dynamicznie tworzone funkcją *RGB())*. Jednak okazuje się, że także kolory 16-bitowe (czyli niepełne *RGB()*), aczkolwiek tworzone poprawnie, wykazują jakąś ziarnistość. Receptą jest domieszanie niewielkiej składowej pseudolosowej do każdej barwy — niech kolor będzie żółty, ale z ewentualną domieszką okolicznych barw. Domieszka niech nie będzie duża — np. 1% odchylenia od barw wyliczonych.

   Jak to zaimplementować?

## Odpowiedzi

1. Należy zaimplementować rozmywanie zarówno jaskrawości, jak i długości fali. Gdzieś na samym początku funkcji *lambda_to_kolor()* dodajmy dwa wiersze, lekko psujące otrzymane argumenty:

```
...
jasnosc += (1 - random( 3));
L += 0.1E-8 * (1.0 - (double)random( 2000) / 1000.0);
...
```

   Pierwszy wiersz psuje całkowitą wartość zmiennej *jasność*, dodając do niej 1, 0 lub –1 (czyli na głębokość +/– 1%). Drugi wiersz psuje rzeczywistą wartość zmiennej *lambda* na głębokość +/– 0.1E–8 (czyli o dwa rzędy mniej niż podawana długość fali). To rzeczywiście pomaga usunąć ziarnistość odwzorowania barwy!