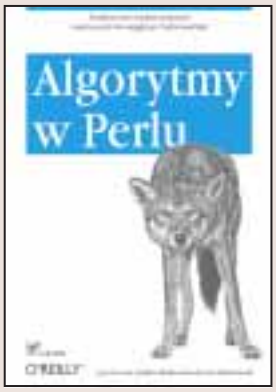
Algorytmy w Perlu



Autorzy: Jon Orwant, Jarkko Hietaniemi, John Macdonald
Tłumaczenie: Sławomir Dzieniszewski, Marcin Jędrysiak
ISBN: 83-7197-913-4
Tytuł oryginału: Mastering Algorithms with Perl
Format: B5, stron: 680

Wielu programistów poszukuje książki, która przedstawiłaby implementacje znanych algorytmów w Perlu. Niestety w podręcznikach do tego języka trudno znaleźć informacje na ten temat. Informatycy opracowali wiele technik związanych z często spotykanymi problemami, takimi jak:

- Przybliżone dopasowywanie tekstów (uwzględniające literówki)
- Znajdowanie korelacji w zbiorach danych
- Algorytmy związane z grami
- Przewidywanie zjawisk (np. obciążenia serwera WWW)
- Dopasowywanie wielomianowe i za pomocą funkcji sklejanych
- Szyfrowanie informacji

Dzięki algorytmom przedstawionym w niniejszej książce będziesz mógł poradzić sobie z tymi problemami używając wydajnego i łatwego do nauczenia się języka, jakim jest Perl.

Autorzy zakładają, że opanowałeś już składnię Perla i znasz jego podstawowe funkcje. Książka „Algorytmy w Perlu" przystępnie objaśni Ci, kiedy używać klasycznych technik programistycznych i w jakich rodzajach aplikacji znajdują one swoje zastosowanie, a przede wszystkim pokaże Ci, jak je implementować w Perlu.

Jeśli jesteś początkującym programistą, poznasz najważniejsze algorytmy, które pozwolą Ci rozwiązywać problemy programistyczne w sposób profesjonalny. Nawet jeśli znasz już podstawy algorytmiki, będziesz zapewne zaskoczony z jaką łatwością można je zastosować w Perlu. W książce znajdziesz nawet obowiązkowy program rysujący fraktale.

Jest to pierwsza książka spośród licznych pozycji poświęconych algorytmom, która demonstruje ich użycie za pomocą Perla.

Autorami są m.in. Jon Orwant, redaktor The Perl Journal i Jarkko Hietaniemi – zarządzający biblioteką modułów CPAN. Wszyscy autorzy są stałymi współpracownikami CPAN, stąd wiele z przytoczonych tu fragmentów kodu możesz znaleźć w tej bibliotece.

„Poświęciłem lekturze wiele czasu przeznaczonego na sen – tak ekscytująca jest ta książka" – Tom Christiansen

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

Spis treści

10

Algorytmy geometryczne

Nie niszcz moich kół!

— Archimedes (287 – 212 p.n.e.)

Geometria nie wymaga wprowadzenia. Jest to najbardziej wizualna dziedzina matematyki, dzięki której możliwe jest wyświetlanie obrazu na ekranie monitora. Ten rozdział przedstawia algorytmy pozwalające na wykonanie poniższych zadań:

Tworzenie grafiki zawierającej hiperodnośniki (mapy obrazu)
: W jaki sposób można sprawdzić, czy użytkownik kliknął myszą w obszarze o dziwnych kształtach? Szczegóły można odnaleźć w podrozdziale „Zawieranie punktów i wielokątów".

Nakładanie okien
: Jak można otworzyć nowe okno, aby w minimalnym stopniu zasłaniało już istniejące okna? Zobacz podrozdział „Granice".

Kartografia
: W jaki sposób można oznaczyć obszar ograniczający grupę rozsianych punktów? Zobacz podrozdział „Granice".

Symulacje
: Które z 10 000 punktów znajdują się najbliżej siebie, co oznacza niebezpieczeństwo zderzenia? Odpowiedź znajduje się w podrozdziale „Najbliższa para punktów".

W tym rozdziale przedstawiane są różne wzory i algorytmy geometryczne. Znajdujące się tu przykłady stanowią tylko komponenty, które powinny być ulepszane przez Ciebie, Czytelniku, gdyż nie jesteśmy w stanie przewidzieć wszystkich przykładów ich zastosowania. Większość programów została ograniczona do pracy tylko w dwóch wymiarach. Nie są tu również poruszane zaawansowane zagadnienia, które można znaleźć w książkach poświęconych grafice komputerowej, takie jak śledzenie promieni (ang. *ray tracing*), oświetlanie sceny, animacja lub mapowanie tekstur. Jedynym wyjątkiem są tu krzywe złożone,

które zostały omówione w rozdziale 16. poświęconym analizie numerycznej. Zalecana literatura poświęcona grafice to m.in. książka „Computer Graphics: Principles and Practice" (autorstwa Foleya, van Dama, Feinera i Hughesa) oraz seria książek „Graphics Gems". Pod koniec rozdziału znajdują się informacje dotyczące obsługi okien, tworzenia grafiki biznesowej, języka OpenGL (język grafiki trójwymiarowej) oraz języka VRML (Virtual Reality Markup Language).

Prawie wszystkie procedury z tego rozdziału akceptują współrzędne przekazywane jako płaskie listy liczb. Aby możliwe było utworzenie interfejsu do już istniejących programów, może wystąpić konieczność dokonania modyfikacji tych procedur, dzięki czemu możliwe będzie przekazywanie punktów, prostych i wielokątów poprzez tablice lub tablice asocjacyjne. Ta metoda będzie także szybsza w przypadku dużej ilości danych. Więcej informacji na ten temat można znaleźć w rozdziale 1.

Należy pamiętać o pewnej bardzo istotnej kwestii. Wiele problemów geometrycznych ma specjalne przypadki, które wymagają zwrócenia dodatkowej uwagi. Wiele algorytmów nie działa poprawnie dla wklęsłych obiektów, przez co przed użyciem algorytmu należy podzielić takie obiekty na mniejsze, wypukłe fragmenty. Skomplikowane obiekty, takie jak drzewa, ludzie oraz potwory z filmów SF, są zwykle reprezentowane w postaci wielokątów (najczęściej trójkątów lub czterościanów, jeśli są to obiekty trójwymiarowe). Zderzenia takich obiektów są sprawdzane przy użyciu *powłok wypukłych prostopadłościanów ograniczających*. Więcej informacji na ten temat można znaleźć w dalszej części rozdziału.

Odległość

Jedną z podstawowych koncepcji geometrycznych jest *odległość* między dwoma obiektami.

Odległość euklidesowa

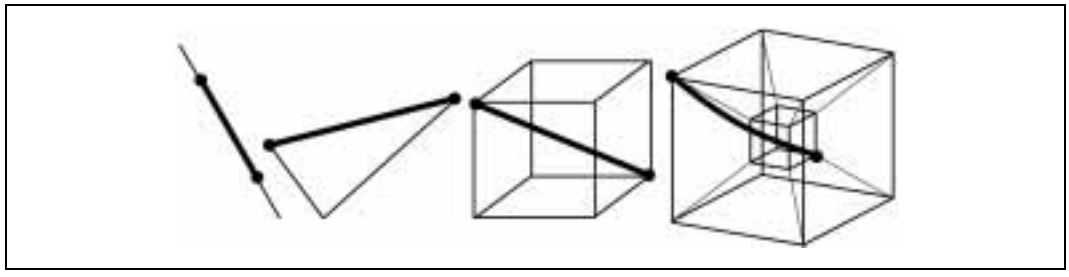
Istnieje wiele sposobów definiowania odległości między dwoma punktami, ale najczęściej używaną i najbardziej intuicyjną definicją jest *odległość euklidesowa*, czyli odległość w linii prostej[1]. Aby uzyskać tę odległość, należy obliczyć różnice w położeniu dla każdej osi, zsumować kwadraty różnic i obliczyć pierwiastek kwadratowy tej sumy. W przypadku dwóch wymiarów sprowadza się to do znajomego *twierdzenia Pitagorasa*[2]:

$$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

Rysunek 10.1 przedstawia odległość euklidesową dla różnej liczby wymiarów. Ostatnie dwa przykłady są jedynie projekcją trzech i czterech wymiarów na kartkę papieru.

[1] Euklides — ok. 370 p.n.e.

[2] Pitagoras — 750 – 490 p.n.e.



Rysunek 10.1. Odległość euklidesowa w jednym, dwóch, trzech i czterech wymiarach

Przedstawiona poniżej procedura służy do obliczania odległości euklidesowej w dowolnej liczbie wymiarów.

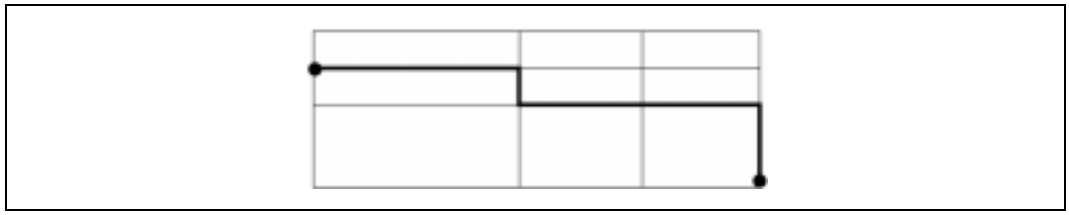
```
# Procedura odleglosc( @p ) oblicza odleglosc euklidesowa pomiedzy dwoma
# punktami o d wymiarow, dla ktorych istnieje 2 * d wspolrzednych.
# Dla przykladu, para punktow trojwymiarowych powinna byc przekazana do tej
# procedury jako ( $x0, $y0, $z0, $x1, $y1, $z1 ).

sub odleglosc {
    my @p = @_;                 # Wspolrzedne punktow.
    my $d = @p / 2;             # Liczba wymiarow.

    # Procedura zostala zoptymalizowana dla przypadku z dwoma wymiarami.
    return sqrt( ($_[0] - $_[2])**2 + ($_[1] - $_[3])**2 )
        if $d == 2;

    my $s = 0;                  # Suma kwadratow.
    my @p0 = splice @p, 0, $d;  # Uzyskanie punktu startowego.

    for ( my $i = 0; $i < $d; $i++ ) {
        my $di = $p0[ $i ] - $p[ $i ];  # Roznica...
        $s += $di * $di;                # ...podniesiona do kwadratu i zsumowana.
    }

    return sqrt( $s );
}
```

Odległość euklidesowa pomiędzy punktami (3, 4) i (10, 12) może być obliczona w następujący sposób:

```
print odleglosc( 3,4, 10,12);
10.6301458127346
```

Odległość Manhattan

Kolejną miarą odległości jest *odległość Manhattan*, która została przedstawiona graficznie na rysunku 10.2. Nazwa tej odległości jest związana z prostokątną siatką, zgodnie z którą ułożone są ulice w Nowym Jorku. Taksówkarze jeżdżący po tym mieście zwykle mierzą wszystko odległością Manhattan, podczas gdy piloci śmigłowców są przyzwyczajeni do odległości euklidesowej.



Rysunek 10.2. Odległość Manhattan

W celu obliczenia odległości Manhattan zamiast potęgowania różnic między punktami należy dokonać ich sumowania. Poniżej przedstawiono odpowiednią procedurę.

```
# odleglosc_manhattan( @p )
# Oblicza odleglosc Manhattan pomiedzy dwoma
# punktami o d wymiarow, dla ktorych istnieje 2 * d wspolrzednych.
# Dla przykladu, para punktow trojwymiarowych powinna byc przekazana do tej
# procedury jako ( $x0, $y0, $z0, $x1, $y1, $z1 ).

sub odleglosc_manhattan {
    my @p = @_;                 # Wspolrzedne punktow.
    my $d = @p / 2;             # Liczba wymiarow.

    my $S = 0;                  # Suma kwadratow.
    my @p0 = splice @p, 0, $d;  # Uzyskanie punktu startowego.

    for ( my $i = 0; $i < $d; $i++ ) {
        my $di = $p0[ $i ] - $p[ $i ];  # Roznica...
        $S += abs $di;                  # ...zsumowanie absolutnej wartosci.
    }

    return $S;
}
```

Oto przykładowa odległość Manhattan między punktami (3, 4) i (10, 12):

```
print odleglosc_manhattan( 3, 4, 10, 12);
15
```

Maksymalna odległość

Czasami odległość najlepiej zdefiniować jako największą różnicę między współrzędnymi: $d = max\ d_i$, gdzie d_i jest różnicą i-tej współrzędnej.

Jeśli odległość Manhattan stanowi przybliżenie pierwszego stopnia odległości (ponieważ różnice między współrzędnymi są podnoszone do potęgi 1), to odległość euklidesowa jest przybliżeniem drugiego stopnia. Granicą tego ciągu jest maksymalna odległość:

$$\sqrt[k]{\sum_{k=1}^{\infty} d_i^k}$$

Innymi słowy, wraz ze wzrostem k zaczyna dominować maksymalna odległość, natomiast w nieskończoności dominuje ona już całkowicie.

Odległość na powierzchni kulistej

Najkrótsza odległość na powierzchni kulistej nosi nazwę *odległości koła wielkiego*. Uzyskanie prawidłowego wzoru stanowi dobre zadanie trygonometryczne, ale programista może po prostu skorzystać z funkcji `great_circle_distance()`, która jest dostępna w module Math::Trig dołączonym do Perla w wersji 5.005_03 lub nowszej. Choć ten moduł znajduje się także we wcześniejszych wersjach Perla, to nie zawiera funkcji `great_circle_distance()`. Poniżej pokazano sposób, w jaki można obliczyć przybliżoną odległość w kilometrach między Londynem (51,3° N, 0,5° W) oraz Tokio (35,7° N, 139,8° E).

```
#!/usr/bin/perl

use Math::Trig qw(great_circle_distance deg2rad);

# Prosze zauwazyc odejmowanie szerokosci geograficznej od 90 stopni;
# fi zero znajduje sie na biegunie polnocnym.
@londyn = (deg2rad(-  0.5), deg2rad(90 - 51.3));
@tokio  = (deg2rad( 139.8), deg2rad(90 - 35.7));

# 6378 to promien rownikowy Ziemi.
print great_circle_distance(@londyn, @tokio, 6378);
```

Odległość między Londynem i Tokio wynosi:

```
9605.26637021388
```

Szerokość geograficzna jest odejmowana od 90, ponieważ funkcja `great_circle_distance()` wykorzystuje *azymutowe współrzędne sferyczne*. ϕ=0 wskazuje kierunek z bieguna północnego, podczas gdy w innych miejscach świata jest to kierunek od równika. Z tego powodu należy odwrócić współrzędne o 90 stopni. Więcej informacji na ten temat można odnaleźć w dokumentacji modułu Math::Trig.

Oczywiście wynik nie jest dokładny, ponieważ Ziemia nie jest idealną kulą, a 0,1° stopnia na tych szerokościach geograficznych wynosi około 8 km.

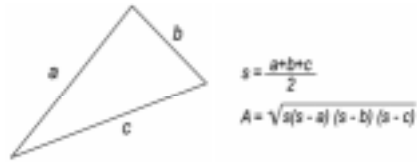
Pole, obwód i objętość

Ponieważ uzyskaliśmy już niezbędne informacje na temat odległości, możemy zająć się tematyką pola, obwodu i objętości.

Trójkąt

Pole trójkąta może być obliczone na wiele sposobów, które są zależne od dostępnych informacji. Na rysunku 10.3 przedstawiono jedną z najstarszych metod obliczania pola trójkąta, znaną jako *wzór Herona*.[3]

[3] Heron żył w latach 65 – 120 n.e.



Rysunek 10.3. Wzór Herona pozwala na obliczenie pola trójkąta, jeśli znana jest długość jego boków

Parametrami poniższej procedury implementującej wzór Herona mogą być zarówno długości boków trójkąta, jak i jego wierzchołki. W tym drugim przypadku procedura `pole_trojkata_heron()` obliczy długości boków z użyciem odległości euklidesowej:

```
#!/usr/bin/perl
# pole_trojkata_heron( $dlugosc_pierwszego_boku,
#                      $dlugosc_drugiego_boku,
#                      $dlugosc_trzeciego_boku )
#   Mozliwe jest takze podanie szesciu argumentow, ktore beda trzema parami
#   wspolrzednych (x,y) rogow trojkata.
# Procedura zwraca pole trojkata.

sub pole_trojkata_heron {
    my ( $a, $b, $c );

    if ( @_ == 3 ) { ( $a, $b, $c ) = @_ }
    elsif ( @_ == 6 ) {
        ( $a, $b, $c ) = ( odleglosc( $_[0], $_[1], $_[2], $_[3] ),
                           odleglosc( $_[2], $_[3], $_[4], $_[5] ),
                           odleglosc( $_[4], $_[5], $_[0], $_[1] ) );
    }

    my $s = ( $a + $b + $c ) / 2;              # Parametr posredni.
    return sqrt( $s * ( $s - $a ) * ( $s - $b ) * ( $s - $c ) );
}

print pole_trojkata_heron(3, 4, 5), " ",
      pole_trojkata_heron( 0, 1,   1, 0,   2, 3 ), "\n";
```

Uruchomienie procedury da następujące wyniki:

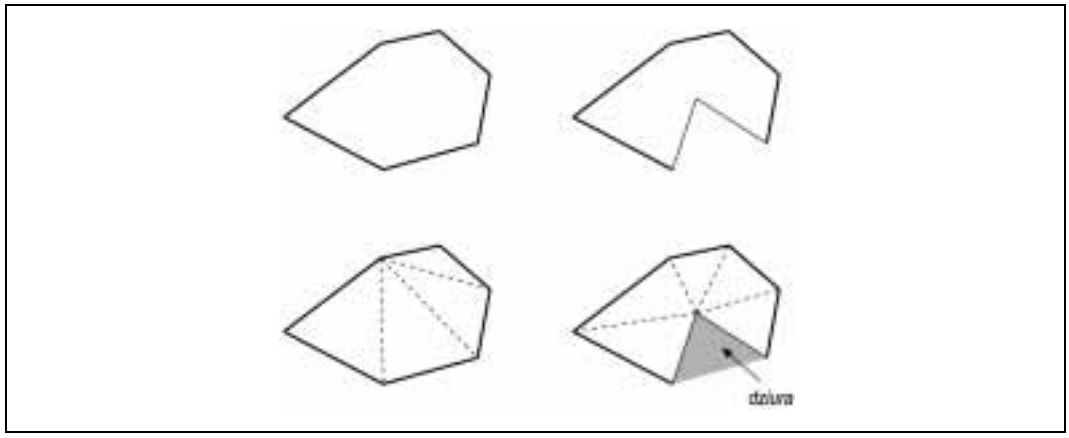
```
6 2
```

Pole wielokąta

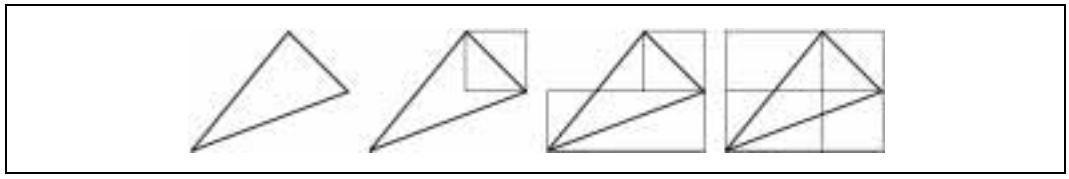
Pole wielokąta wypukłego może być obliczone poprzez podzielenie go na kilka trójkątów, a następnie zsumowanie ich pól, co pokazano na rysunku 10.4.

Sytuacja staje się bardziej skomplikowana w przypadku wielokątów wklęsłych, gdyż konieczne jest zignorowanie „dziur". Znacznie prostszym sposobem jest użycie wyznaczników, które zostały omówione bliżej w rozdziale 7. poświęconym macierzom. Tę metodę przedstawiają rysunek 10.5 oraz poniższy wzór:

$$A = \frac{1}{2}\left(\begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \end{vmatrix} + \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} + \ldots + \begin{vmatrix} x_{n-1} & y_{n-1} \\ x_0 & y_0 \end{vmatrix}\right)$$



Rysunek 10.4. Podzielone na trójkąty wielokąty wypukłe i wklęsłe



Rysunek 10.5. Wyznaczanie pól przez wyznaczniki

Każdy wyznacznik wyznacza pole prostokąta zdefiniowanego przez dwa wierzchołki wielokąta. Ponieważ każda krawędź wieloboku dzieli prostokąt na pół, to obliczone pole prostokąta należy podzielić przez 2. Nakładające się fragmenty prostokątów (prawy wielokąt na rysunku 10.5) mogą być zignorowane.

Proszę zauważyć, jak przedstawiony powyżej wzór łączy ostatni punkt — (x_{n-1}, y_{n-1}) — z pierwszym punktem — (x_0, y_0). Jest to naturalne, ponieważ należy uwzględnić wszystkie n krawędzi wielokąta, co oznacza zsumowanie dokładnie n wyznaczników. W tym przypadku potrzebny jest wyznacznik macierzy 2 x 2, co jest proste. Poniżej przedstawiono odpowiednią procedurę.

```
# wyznacznik( $x0, $y0, $x1, $y1 )
# Procedura oblicza wyznacznik dla czterech elementow macierzy
# podanych jako argumenty.
#
sub wyznacznik { $_[0] * $_[3] - $_[1] * $_[2] }
```

Teraz możemy już obliczyć pole wielokąta:

```
# pole_wielokata( @xy )
#   Procedura oblicza pole wielokata z uzyciem wyznacznika. Do procedury
#   nalezy przekazac punkty w postaci ( $x0, $y0, $x1, $y1, $x2, $y2, ...).
#

sub pole_wielokata {
    my @xy = @_;
```

```
    my $A = 0;                          # Pole.

    # Polaczenie ostatniego i pierwszego punktu jest wykonywane juz na poczatku
    # procedury, a nie na jej koncu (punkt [-2, -1] ponizej).
    for ( my ( $xa, $ya ) = @xy[ -2, -1 ];
          my ( $xb, $yb ) = splice @xy, 0, 2;
           ( $xa, $ya ) = ( $xb, $yb ) ) { # Przejscie do kolejnego punktu.
        $A += wyznacznik( $xa, $ya, $xb, $yb );
    }

    # Jesli punkty zostaly podane w kierunku przeciwnym do ruchu wskazowek
    # zegara, to zmienna $A bedzie miala wartosc ujemna. Z tego powodu nalezy
    # obliczyc wartosc absolutna.

    return abs $A / 2;
}
```

Pole pięciokąta zdefiniowanego przez punkty (0, 1), (1, 0), (3, 2), (2, 3) i (0, 2) może być obliczone w następujący sposób:

```
print pole_wielokata( 0, 1,  1, 0,  3, 2,  2, 3,  0, 2 ), "\n";
```

Wynik wynosi:

```
5
```

Należy pamiętać, że kolejne punkty muszą być podane w kolejności zgodnej lub przeciwnej do kierunku ruchu wskazówek zegara (bliższe wyjaśnienia można odnaleźć w kolejnym podrozdziale). Przekazanie punktów do procedury w innej kolejności oznacza inną definicję prostokąta, co pokazano na poniższym przykładzie:

```
print pole_wielokata( 0, 1,  1, 0,  0, 2,  3, 2,  2, 3 ), "\n";
```

Przeniesienie ostatniego punktu do środka dało w efekcie wynik równy:

```
1
```

Obwód wielokąta

Procedura służąca do obliczania pola wielokąta może być użyta także do uzyskania jego obwodu. W tym celu wystarczy zsumować długości, a nie wyznaczniki. Poniżej przedstawiono odpowiednią procedurę:

```
# obwod_wielokata( @xy )
#   Procedura oblicza dlugosc obwodu wielokata. Do procedury nalezy
#   przekazac punkty w postaci ( $x0, $y0, $x1, $y1, $x2, $y2, ...).
#

sub obwod_wielokata {
    my @xy = @_;

    my $P = 0;                          # Obwod.

    # Polaczenie ostatniego i pierwszego punktu jest wykonywane juz na poczatku
    # procedury, a nie na jej koncu (punkt [-2, -1] ponizej).
    for ( my ( $xa, $ya ) = @xy[ -2, -1 ];
```

```
            my ( $xb, $yb ) = splice @xy, 0, 2;
            ( $xa, $ya ) = ( $xb, $yb ) ) { # Przejscie do kolejnego punktu.
        $A += dlugosc( $xa, $ya, $xb, $yb );
    }

    return $P / 2;
}
```

Obwód pięciokąta z poprzedniego przykładu można obliczyć w następujący sposób:

```
print obwod_wielokata( 0, 1,  1, 0,  3, 2,  2, 3,  0, 2 ), "\n";
```

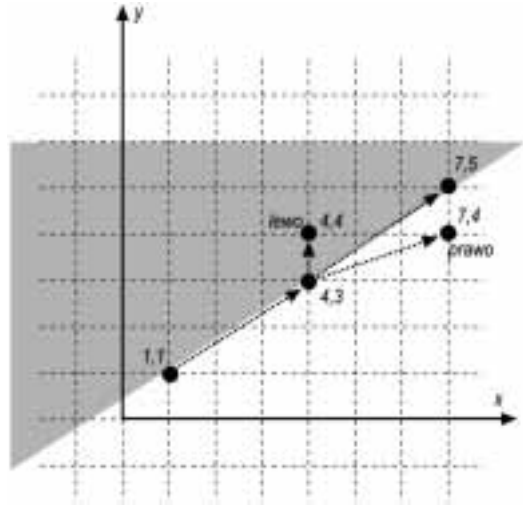
Wynik to:

```
8.89292222699217
```

Kierunek

Czasami warto wiedzieć, czy obiekty znajdują się na prawo (zgodnie z kierunkiem ruchu wskazówek zegara) lub na lewo od nas (przeciwnie do kierunku ruchu wskazówek zegara). Może to być przydatne na przykład do ustalenia, czy dany punkt znajduje się wewnątrz trójkąta. W tym rozdziale ograniczymy się jedynie do dwóch wymiarów, ponieważ trzy wymiary zmieniają znaczenie określeń „lewo" i „prawo" w zależności od kierunku wybranego jako „góra".

Dla dowolnych trzech punktów możliwe jest ustalenie, czy tworzą one ścieżkę zgodną lub przeciwną do kierunku ruchu wskazówek zegara (a być może nie istnieje żadna taka ścieżka). Punkty (1, 1), (4, 3) i (4, 4) na rysunku 10.6 tworzą ścieżkę zgodną z kierunkiem ruchu wskazówek zegara (ścieżka kieruje się w lewo), natomiast punkty (1, 1), (4, 3) i (7, 4) tworzą ścieżkę o przeciwnym kierunku (skierowaną w prawo).



Rysunek 10.6. Dwie ścieżki — jedna skierowana w lewo, druga w prawo

Parametrami procedury `kierunek()` są trzy punkty, natomiast zwracana jest pojedyncza liczba. Jeśli zwrócona zostanie wartość dodatnia, to ścieżka przechodząca przez wszystkie trzy punkty jest zgodna z kierunkiem ruchu wskazówek zegara. Liczba ujemna oznacza ścieżkę przeciwną do kierunku ruchu wskazówek zegara, natomiast wartość bliska 0 oznacza, że trzy punkty znajdują się na linii prostej.

```
# kierunek( $x0, $y0, $x1, $y1, $x2, $y2 )
#    Procedura zwraca wartosc dodatnia, jesli przesuwajac sie z p0 (x0, y0)
#    do przez p1 do p2 nalezy skrecic w prawo, wartosc ujemna, jesli nalezy
#    skrecic w lewo.
#    Zero jest zwracane w przypadku, gdy wszystkie trzy punkty leza na tej samej
#    linii prostej. Nalezy jednak uwazac na bledy zmiennoprzecinkowe.
#

sub kierunek {
    my ( $x0, $y0, $x1, $y1, $x2, $y2 ) = @_;
    return ( $x2 - $x0 ) * ( $y1 - $y0 ) - ( $x1 - $x0 ) * ( $y2 - $y0 );
}
```

Np. poniższe wywołania:

```
print kierunek( 1, 1,   4, 3,   4, 4 ), "\n";
print kierunek( 1, 1,   4, 3,   7, 5 ), "\n";
print kierunek( 1, 1,   4, 3,   7, 4 ), "\n";
```

zwrócą wartości:

```
-3
0
3
```

Innymi słowy, punkt (4, 4) znajduje się na lewo (wartość ujemna) od wektora przebiegającego od (1, 1) do (4, 3), punkt (7, 5) znajduje się *na* tym wektorze (wartość zero), natomiast punkt (7, 4) znajduje się na prawo (wartość dodatnia) od tego wektora.

Procedura `kierunek()` stanowi spłaszczoną, dwuwymiarową wersję iloczynu wektorowego. *Iloczyn wektorowy* jest obiektem trójwymiarowym, który wskazuje na zewnątrz od płaszczyzny zdefiniowanej przez wektory $p_0 - p_1$ oraz $p_1 - p_2$.

Przecięcie

W tej części rozdziału będziemy często wykorzystywali podprocedurę `epsilon()` do obliczeń zmiennoprzecinkowych. Można wybrać dowolną wartość epsilon, ale zalecamy użycie wartości równej 1e–10:

```
sub epsilon () { 1E-10 }
```

Szybsza wersja tej procedury to:

```
sub constant epsilon => 1E-10;
```

Więcej informacji na ten temat znajdziesz w rozdziale 11. poświęconym systemom liczbowym.

Punkt przecięcia prostych

Istnieją dwie odmiany przecięcia prostej. W ogólnym przypadku linie mogą mieć dowolne nachylenie, natomiast w bardziej restrykcyjnym przypadku wszystkie linie muszą być poziome lub pionowe; ten przypadek jest nazywany *przecięciem Manhattan*.

Przecięcie prostych — ogólny przypadek

W celu odnalezienia punktu przecięcia dwóch prostych wystarczy tylko odnaleźć miejsce, w którym krzyżują się proste $y_0 = b_0x + a_0$ i $y_1 = b_1x + a_1$. Pomocne tu będą techniki opisane w rozdziałach 7. i 16. poświęcone rozwiązywaniu równań i eliminacji Gaussa, ale trzeba pamiętać o możliwości wystąpienia pewnych problemów. Aby uniknąć błędów dzielenia przez 0, należy zwrócić uwagę na przypadek, w którym jedna z linii jest pozioma lub pionowa. Specjalnym przypadkiem są również proste równoległe. Rysunek 10.7 ilustruje różne rodzaje przecięcia prostych.

Rysunek 10.7. Przecięcia prostych: przypadek ogólny, prosta pozioma i pionowa oraz proste równoległe

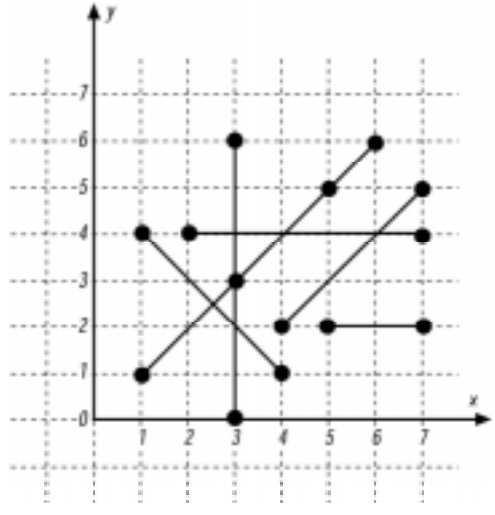
Obecność tych specjalnych przypadków sprawia, że algorytm obliczający punkt przecięcia prostych staje się dość skomplikowany. Również implementacja tego algorytmu wymaga dużej ilości kodu:

```
# przeciecie_prostych( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 )
#
#    Procedura oblicza punkt przeciecia odcinkow
#    (x0,y0)-(x1,y1) i (x2,y2)-(x3,y3).
#
#    Mozliwe jest takze podanie czterech argumentow decydujacych
#    o nachyleniu obu prostych oraz punktach przeciecia z osia y. Innymi slowy,
#    jesli obie proste zostana przedstawione jako y = ax+b, to nalezy podac
#    dwie wartosci 'a' i dwie wartosci 'b'.
#
#    Procedura przeciecie_prostych() zwraca trzy wartosci ($x, $y, $s) dla punktu
#    przeciecia, gdzie $x i $y to wspolrzedne tego punktu, a $s jest prawda,
#    jesli odcinki przecinaja sie, lub falsz, jesli odcinki nie maja punktu
#    przeciecia (ale ekstrapolowane proste przecinalyby sie).
#
#    W innych przypadkach zwracany jest ciag opisujacy przyczyne, dla ktorej
#    odcinki nie przecinaja sie:
#      "poza prostopadloscianem ograniczajacym"
#      "rownolegle"
#      "rownolegle wspolliniowe"
#      "rownolegle poziome"
#      "rownolegle pionowe"
```

```
#    Ze wzgledu na kontrole prostopadloscianow ograniczajacych przypadki
#    "rownolegle poziome" i "rownolegle pionowe" nigdy nie wystepuja.
#    (Prostopadlosciany ograniczajace zostana omowione w dalszej czesci
#    rozdzialu.)
#
sub przeciecie_prostych {
 my ( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 );

 if ( @_ == 8 ) {
     ( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 ) = @_;

     # Prostopadlosciany ograniczajace dziela proste na odcinki.
     # Procedura prostopadloscian_ograniczajacy() zostanie zdefiniowana
     # w dalszej czesci rozdzialu.
     my @prostokat_a = prostopadloscian_ograniczajacy( 2, $x0, $y0, $x1, $y1 );
     my @prostokat_b = prostopadloscian_ograniczajacy( 2, $x2, $y2, $x3, $y3 );

     # Po usunieciu tego testu odcinki stalyby sie nieskonczonymi prostymi.
     # Procedura prostopadloscian_ograniczajacy_przeciecie() zostanie
     # zdefiniowana w dalszej czesci rozdzialu.
     return "poza prostopadloscianem ograniczajacym"
         unless prostopadloscian_ograniczajacy_przeciecie( 2, @prostokat_a,
            @prostokat_b );
 } elsif ( @_ == 4 ) { # Forma parametryczna.
     $x0 = $x2 = 0;
     ( $y0, $y2 ) = @_[ 1, 3 ];
     # Nalezy pomnozyc przez 'mnoznik', aby uzyskac wystarczajaca wielkosc.
     my $abs_y0 = abs $y0;
     my $abs_y2 = abs $y2;
     my $mnoznik = 10 * ( $abs_y0 > $abs_y2 ? $abs_y0 : $abs_y2 );
     $x1 = $x3 = $mnoznik;
     $y1 = $_[0] * $x1 + $y0;
     $y3 = $_[2] * $x2 + $y2;
 }

 my ($x, $y);  # Jeszcze nieustalony punkt przeciecia.

 my $dy10 = $y1 - $y0; # dyPQ, dxPQ to roznice wspolrzednych
 my $dx10 = $x1 - $x0; # miedzy punktami P i Q.
 my $dy32 = $y3 - $y2;
 my $dx32 = $x3 - $x2;

 my $dy10z = abs( $dy10 ) < epsilon;  # Czy roznica $dy10 jest zerowa?
 my $dx10z = abs( $dx10 ) < epsilon;
 my $dy32z = abs( $dy32 ) < epsilon;
 my $dx32z = abs( $dx32 ) < epsilon;

 my $dyx10;                           # Nachylenie.
 my $dyx32;

 $dyx10 = $dy10 / $dx10 unless $dx10z;
 $dyx32 = $dy32 / $dx32 unless $dx32z;

 # Po uzyskaniu wszystkich roznic i nachylen mozna wykonac rozpoznanie
 # specjalnych przypadkow z poziomymi i pionowymi prostymi.
 # Nachylenie rowne zero oznacza pozioma prosta.

 unless ( defined $dyx10 or defined $dyx32 ) {
     return "rownolegle pionowe";
 } elsif ( $dy10z and not $dy32z ) { # Pierwsza prosta pozioma.
     $y = $y0;
     $x = $x2 + ( $y - $y2 ) * $dx32 / $dy32;
```

```
    } elsif ( not $dy10z and $dy32z ) { # Druga prosta pozioma.
        $y = $y2;
        $x = $x0 + ( $y - $y0 ) * $dx10 / $dy10;
    } elsif ( $dx10z and not $dx32z ) { # Pierwsza prosta pionowa.
        $x = $x0;
        $y = $y2 + $dyx32 * ( $x - $x2 );
    } elsif ( not $dx10z and $dx32z ) { # Druga prosta pionowa.
        $x = $x2;
        $y = $y0 + $dyx10 * ( $x - $x0 );
    } elsif ( abs( $dyx10 - $dyx32 ) < epsilon ) {
        # Obie wartosci nachylenia sa zaskakujaco zblizone.
        # Prawdopodobnie jest to przypadek rownoleglych prostych wspolliniowych
        # lub zwykle proste rownolegle.

        # Kontrola prostokatow ograniczajacych spowodowala juz odrzucenie
        # przypadkow "rownolegle poziome" i "rownolegle pionowe".

        my $ya = $y0 - $dyx10 * $x0;
        my $yb = $y2 - $dyx32 * $x2;

        return "rownolegle wspolliniowe" if abs( $ya - $yb ) < epsilon;
        return "rownolegle";
    } else {
        # Nie wystapil zaden specjalny przypadek.
        # Obie proste rzeczywiscie sie przecinaja.

        $x = ($y2 - $y0 + $dyx10*$x0 - $dyx32*$x2)/($dyx10 - $dyx32);
        $y = $y0 + $dyx10 * ($x - $x0);
    }

    my $h10 = $dx10 ? ($x - $x0) / $dx10 : ($dy10 ? ($y - $y0) / $dy10 : 1);
    my $h32 = $dx32 ? ($x - $x2) / $dx32 : ($dy32 ? ($y - $y2) / $dy32 : 1);

    return ($x, $y, $h10 >= 0 && $h10 <= 1 && $h32 >= 0 && $h32 <= 1);
}
```

Na rysunku 10.8 przedstawiono kilka przykładów przecinających się prostych.



Rysunek 10.8. Przykłady przecinających się prostych

Procedura `przeciecie_prostych()` zostanie wykorzystana do sprawdzenia sześciu potencjalnych przypadków przecinających się prostych:

```
print "@{[przeciecie_prostych( 1, 1,  5, 5,  1, 4,  4, 1 )]}\n";
print "@{[przeciecie_prostych( 1, 1,  5, 5,  2, 4,  7, 4 )]}\n";
print "@{[przeciecie_prostych( 1, 1,  5, 5,  3, 0,  3, 6 )]}\n";
print "@{[przeciecie_prostych( 1, 1,  5, 5,  5, 2,  7, 2 )]}\n";
print     przeciecie_prostych( 1, 1,  5, 5,  4, 2,  7, 5 ), "\n";
print     przeciecie_prostych( 1, 1,  5, 5,  3, 3,  6, 6 ), "\n";
```

Poniżej przedstawiono wyniki:

```
2.5 2.5 1
4 4 1
3 3 1
2 2
rownolegle
rownolegle wspolliniowe
```

Obliczenie dokładnego punktu przecięcia czasem nie jest wymagane, gdyż wystarczy informacja o tym, że dwie proste przecinają się. Do uzyskania tej informacji wystarczy zbadanie znaków dwóch iloczynów wektorowych, a mianowicie $(p_2 - p_0) \times (p_1 - p_0)$ oraz $(p_3 - p_0) \times (p_1 - p_0)$. Przedstawiona poniżej procedura `przecinanie_prostych()` zwraca prawdę lub fałsz w zależności od tego, czy dwie linie proste przecinają się.

```
# przecinanie_prostych( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 )
#    Procedura zwraca prawde, jesli dwie linie proste zdefiniowane
#    przez podane punkty przecinaja sie.
#    W przypadkach granicznych o wyniku decyduje wartosc epsilon.

sub przecinanie_prostych {

    my ( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 ) = @_;

    my @prostokat_a = prostopadloscian_ograniczajacy( 2, $x0, $y0, $x1, $y1 );
    my @prostokat_b = prostopadloscian_ograniczajacy( 2, $x2, $y2, $x3, $y3 );

    # Jesli nawet prostopadlosciany ograniczajace nie przecinaja sie, to mozna
    # natychmiast przerwac prace procedury.

    return 0 unless prostopadloscian_ograniczajacy_przeciecie( 2, @prostokat_a,
       @prostokat_b );

    # Jesli znaki obu wyznacznikow (wartosci absolutnych lub dlugosci
    # iloczynow wektorowych) roznia sie, to proste przecinaja sie.

    my $dx10 = $x1 - $x0;
    my $dy10 = $y1 - $y0;

    my $wyzn_a = wyznacznik( $x2 - $x0, $y2 - $y0, $dx10, $dy10 );
    my $wyzn_b = wyznacznik( $x3 - $x0, $y3 - $y0, $dx10, $dy10 );

    return 1 if $wyzn_a < 0 and $wyzn_b > 0 or
                $wyzn_a > 0 and $wyzn_b < 0;

    if ( abs( $wyzn_a ) < epsilon ) {
        if ( abs( $wyzn_b ) < epsilon ) {
            # Oba iloczyny wektorowe sa zerowe.
            return 1;
        } elsif ( abs( $x3 - $x2 ) < epsilon and
                  abs( $y3 - $y2 ) < epsilon ) {
```

```
            # Jeden z iloczynow wektorowych ma wartosc zerowa,
            # a drugi wektor (od (x2,y2) do (x3,y3))
            # jest rowniez zerowy.
            return 1;
        }
    } elsif ( abs( $wyzn_b < epsilon ) ) {
        # Jeden z iloczynow wektorowych ma wartosc zerowa,
        # a drugi wektor jest rowniez zerowy.
        return 1 if abs( $dx10 ) < epsilon and abs( $dy10 ) < epsilon;
    }

    return 0; # Domyslny wynik to brak przeciecia.
}
```

Przetestujmy procedurę `przecinanie_prostych()` dla dwóch par linii prostych. Pierwsza para przecina się w punkcie (3, 4), natomiast druga para prostych nie krzyżuje się w ogóle, ponieważ są to linie równoległe.

```
print "Przeciecie\n"
    if     przecinanie_linii( 3, 0,  3, 6,  1, 1,  6, 6 );
print "Brak przeciecia\n"
    unless przecinanie_linii( 1, 1,  6, 6,  4, 2,  7, 5 );
Przeciecie
Brak przeciecia
```

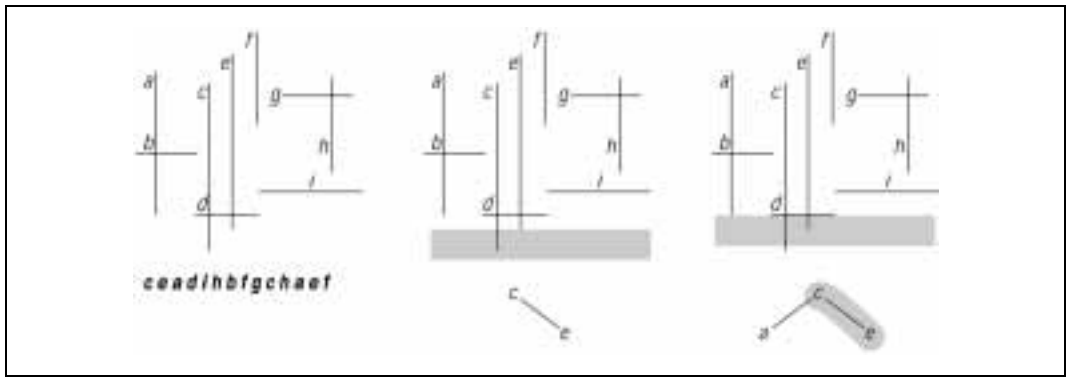
Przecięcie prostych pionowych i poziomych

Bardzo często ogólny przypadek przecięcia prostych jest *zbyt* ogólny; jeśli linie proste są zgodne z zasadami geometrii Manhattan (to znaczy są pionowe lub poziome), to dostępna jest zupełnie inna metoda wyszukiwania punktów przecięcia.

W tym przypadku wykorzystywane są *drzewa binarne*, które zostały przedstawione w rozdziale 3. poświęconym zaawansowanym strukturom danych. Pozioma prosta jest przesuwana od dołu do góry płaszczyzny, co daje w efekcie drzewo binarne pionowych linii prostych posortowanych według ich współrzędnej x. Z tego powodu takie drzewo nosi nazwę *drzewa x*. Drzewo x jest tworzone w następujący sposób:

- Punkty są przetwarzane od dołu do góry i od lewej do prawej. Linie pionowe są przetwarzane przed liniami poziomymi. Oznacza to, iż oba punkty końcowe poziomej prostej są widoczne jednocześnie, podczas gdy punkty końcowe prostej pionowej będą widoczne oddzielnie.
- Każde pojawienie się dolnego punktu końcowego pionowej prostej powoduje dodanie tego węzła do drzewa binarnego. Wartością tego węzła jest współrzędna x. Powoduje to podzielenie punktów w drzewie w następujący sposób: jeśli prosta a znajduje się na lewo od prostej b, to węzeł a znajdzie się w drzewie na lewo od węzła b.
- Każde pojawienie się górnego punktu końcowego pionowej prostej powoduje usunięcie odpowiadającego mu węzła z drzewa binarnego.
- Po napotkaniu poziomej prostej węzły drzewa (aktywne proste pionowe) są sprawdzane w celu ustalenia, czy występuje przecięcie z tą prostą. Poziome linie proste nie są dodawane do drzewa, gdyż ich jedynym zadaniem jest wywołanie kontroli przecięcia.

Rysunek 10.9 przedstawia sposób tworzenia drzewa *x* w czasie przesuwania wyimaginowanej prostej od dołu do góry płaszczyzny. Rysunek po lewej stronie przedstawia jedynie kolejność wykrywania poszczególnych odcinków — najpierw odcinek *c*, potem *e* itd. Środkowy rysunek ilustruje drzewo *x* tuż po wykryciu odcinka *e*, natomiast rysunek po prawej przedstawia drzewo binarne po wykryciu odcinków *a* i *d*. Zwróć uwagę, że odcinek *d* nie został dodany do drzewa, ponieważ służy on tylko do wywołania kontroli przecięcia.



Rysunek 10.9. Przecinanie poziomych i pionowych linii prostych w geometrii Manhattan

Przedstawiona tutaj procedura `przeciecie_manhattan()` implementuje opisany wcześniej algorytm.

```
# przeciecie_manhattan ( @proste )
#   Procedura wyszukuje punkty przeciecia poziomych i pionowych linii prostych.
#   Ta procedura wymaga funkcji proste_dodawanie_do_drzewa(),
#   proste_usuwanie_z_drzewa() i proste_przeszukiw_drzewa(), ktore zostaly
#   zdefiniowane w rozdziale 3.
#
sub przeciecie_manhattan {
    my @op; # Wspolrzedne sa tutaj przeksztalcane jak operacje.

    while (@_) {
        my @prosta = splice @_, 0, 4;

        if ($prosta[1] == $prosta[3]) {     # Pozioma prosta.
            push @op, [ @prosta, \&drzewo_sprawdzenia_przedzialu ];
        } else {                            # Pionowa prosta.
            # Odwrocenie, jesli do gory nogami.
            @prosta = @prosta[0, 3, 2, 1] if $prosta[1] > $prosta[3];

            push @op, [ @prosta[0, 1, 2, 1], \&proste_dodawanie_do_drzewa ];
            push @op, [ @prosta[0, 3, 2, 3], \&proste_usuwanie_z_drzewa ];
        }
    }

    my $drzewo_x; # Drzewo sprawdzania przedzialu.
    # Procedura porownujaca wspolrzedne x.
    my $porownaj_x = sub { $_[0]->[0] <=> $_[1]->[0] };
    my @przeciecie; # Przeciecia prostych.

    foreach my $op (sort { $a->[1] <=> $b->[1] ||
                           $a->[4] == \&drzewo_sprawdzenia_przedzialu ||
                           $a->[0] <=> $b->[0] }
                           @op) {
```

```
            if ($op->[4] == \&drzewo_sprawdzenia_przedzialu) {
                push @przeciecie, $op->[4]->( \$drzewo_x, $op, $porownaj_x );
            } else { # Dodanie lub usuniecie.
                $op->[4]->( \$drzewo_x, $op, $porownaj_x );
            }
        }

        return @przeciecie;
    }

    # drzewo_sprawdzenia_przedzialu( $powiazanie_drzewa, $pozioma, $porownanie )
    #    Ta podprocedura zwraca liste wezlow drzewa, ktore znajduja sie w przedziale
    #    od $pozioma->[0] do $pozioma->[1]. Procedura jest zalezna od drzew
    #    binarnych, ktore zostaly przedstawione w rozdziale 3.
    #
    sub drzewo_sprawdzenia_przedzialu {
        my ( $drzewo, $pozioma, $porownanie ) = @_;

        my @przedzial      = ( );   # Wartosc zwrotna.
        my $wezel          = $$drzewo;
        my $pionowa_x      = $wezel->{val};
        my $pozioma_dolny  = [ $pozioma->[ 0 ] ];
        my $pozioma_gorny  = [ $pozioma->[ 1 ] ];

        return unless defined $$drzewo;

        push @przedzial, drzewo_sprawdzenia_przedzialu( \$wezel->{left}, $pozioma,
           $porownanie )
            if defined $wezel->{left};

        push @przedzial, $pionowa_x->[ 0 ], $pozioma->[ 1 ]
            if $porownanie->( $pozioma_dolny, $pozioma ) <= 0 &&
               $porownanie->( $pozioma_gorny, $pozioma ) >= 0;

        push @przedzial, drzewo_sprawdzenia_przedzialu( \$wezel->{right}, $pozioma,
           $porownanie )
            if defined $wezel->{right};

        return @przedzial;
    }
```

Procedura `przeciecie_manhattan()` działa w czasie nie dłuższym niż $O(N \log N + k)$, gdzie k to liczba przecięć, która nie może być większa niż $(N/2)^2$.

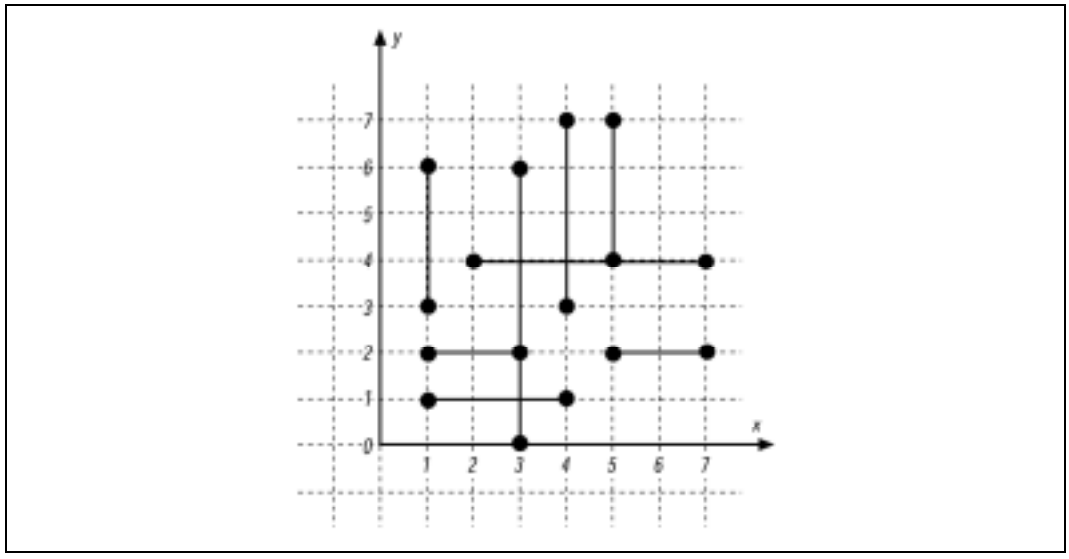
Sposób użycia procedury `przeciecie_manhattan()` zostanie przedstawiony na przykładzie odcinków z rysunku 10.10.

Poszczególne odcinki zostają zapisane w tablicy. Wyszukiwanie przecięć odbywa się w następujący sposób:

```
@proste = ( 1, 6,  1, 3,  1, 2,  3, 2,  1, 1,  4, 1,
            2, 4,  7, 4,  3, 0,  3, 6,  4, 3,  4, 7,
            5, 7,  5, 4,  5, 2,  7, 2 );

print join(" ", przeciecie_manhattan (@proste)), "\n";
```

Uzyskane wyniki to:

```
3 1 3 2 1 4 3 4 4 4 5 4
```



Rysunek 10.10. Odcinki dla przykładu zastosowania algorytmu Manhattan

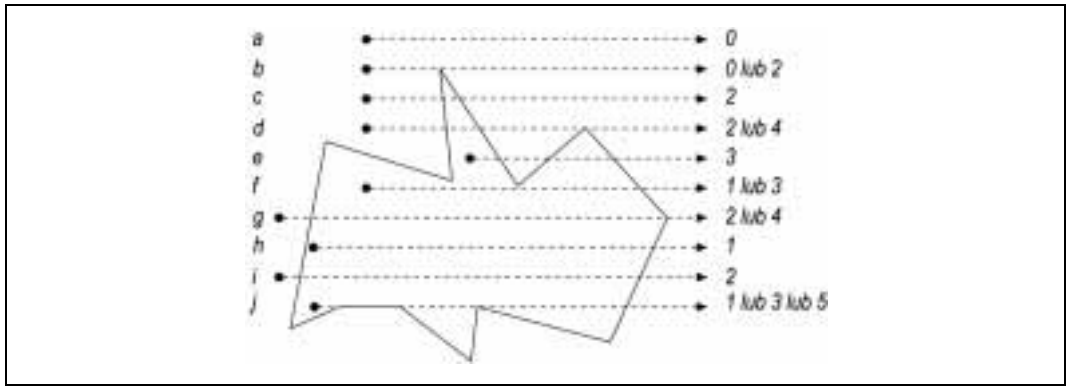
Na podstawie uzyskanych wyników można stwierdzić, że istnieje sześć punktów przecięcia; na przykład punkt (3, 1) stanowi najniższy punkt przecięcia, natomiast (5, 4) to najwyższy taki punkt.

Zawieranie punktów i wielokątów

W tej części rozdziału przedstawimy metody umożliwiające ustalenie, czy punkt znajduje się *wewnątrz* wielokąta. Pozwala to na przeprowadzenie bardziej skomplikowanych operacji, takich jak sprawdzenie, czy wewnątrz wielokąta znalazł się cały odcinek czy tylko jego część.

Punkt wewnątrz wielokąta

Sprawdzenie, czy punkt znajduje się wewnątrz figury geometrycznej, wymaga wyprowadzenia prostej od tego punktu do „nieskończoności" (czyli do dowolnego punktu, który na pewno znajduje się poza obszarem wielokąta). Algorytm jest bardzo prosty — wystarczy ustalić, ile razy prosta przecina krawędzie wielokąta. Jeśli ta liczba będzie nieparzysta (punkty *e*, *f*, *h* i *j* na rysunku 10.11), to punkt znajduje się wewnątrz figury. W przeciwnym razie punkt jest umieszczony na zewnątrz (punkty *a*, *b*, *c*, *d*, *g* i *i* na tym samym rysunku). Istnieją jednak pewne specjalne przypadki (bardzo rzadko zdarza się algorytm geometryczny, który nie powoduje problemów) wymagające specjalnego potraktowania. Co się stanie, jeśli wyprowadzona prosta przetnie wierzchołek wielokąta (punkty *d*, *f*, *g* i *j*)? W jeszcze gorszym przypadku prosta może przebiegać wzdłuż krawędzi figury geometrycznej (punkt *j*). Przedstawiony tu algorytm gwarantuje zwrócenie prawdy dla punktów naprawdę znajdujących się wewnątrz wielokąta lub fałszu dla pozostałych przypadków. Przypadki graniczne są traktowane zgodnie z wybranym sposobem liczenia.



Rysunek 10.11. Czy punkt znajduje się wewnątrz wielokąta? Wystarczy policzyć przecięcia krawędzi

Procedura `punkt_wewnatrz_wielokata()` zwraca prawdę, jeśli dany punkt (pierwsze dwa argumenty) znajduje się wewnątrz wielokąta opisanego przez kolejne argumenty:

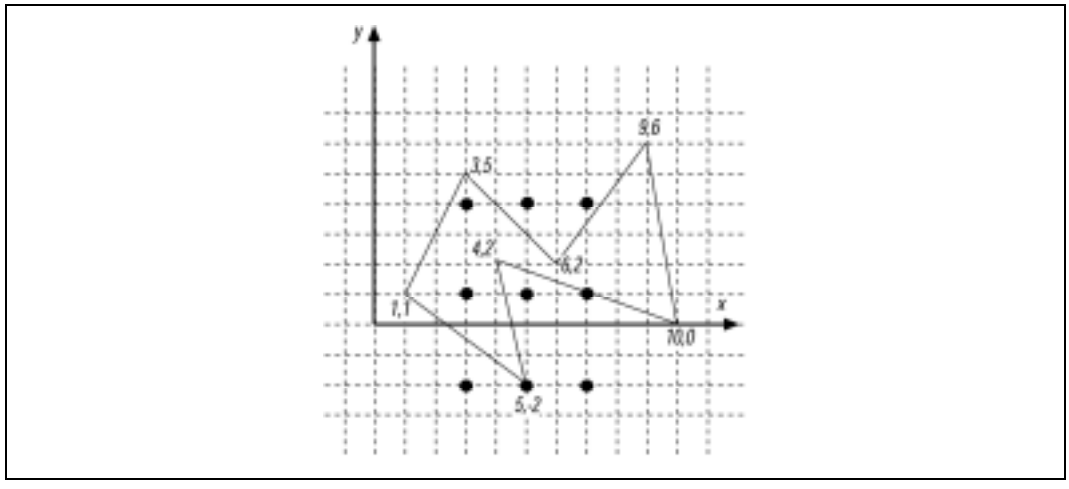
```
# punkt_wewnatrz_wielokata ( $x, $y, @xy )
#
#    Parametry to punkt ($x,$y) oraz wielokat ($x0, $y0, $x1, $y1, ...) w @xy.
#    Procedura zwraca 1 dla punktow znajdujacych sie wewnatrz wielokata
#    i 0 dla punktow poza wielokatem. W przypadku punktow granicznych sytuacja
#    jest bardziej skomplikowana, a jej omowienie wykraczaloby poza tematyke
#    tej ksiazki. Punkty graniczne sa jednak ustalane jednoznacznie;
#    jesli plaszczyzna zostanie podzielona na wiele wielokatow, to kazdy punkt
#    znajdzie sie dokladnie w jednym wielokacie.
#
#    Procedura wywodzi sie z dokumentu FAQ dla grupy dyskusyjnej
#    comp.graphics.algorithms i zostala udostepniona przez Wm. Randolpha
#    Franklina.
#
sub punkt_wewnatrz_wielokata {
    my ( $x, $y, @xy ) = @_;

    my $n = @xy / 2;                        # Liczba punktow wielokata.
    my @i = map { 2 * $_ } 0 .. (@xy/2);    # Parzyste indeksy @xy.
    my @x = map { $xy[ $_ ]     } @i;       # Parzyste indeksy: wspolrzedne x.
    my @y = map { $xy[ $_ + 1 ] } @i;       # Nieparzyste indeksy: wspolrzedne y.

    my ( $i, $j );                          # Indeksy.

    my $polozenie = 0;                      # 0 = na zewnatrz, 1 = wewnatrz.

    for ( $i = 0, $j = $n - 1 ; $i < $n; $j = $i++ ) {
        if (
            (

             # Jesli y znajduje sie w granicach (y-)...
             ( ( $y[ $i ] <= $y ) && ( $y < $y[ $j ] ) ) ||
             ( ( $y[ $j ] <= $y ) && ( $y < $y[ $i ] ) )
            )
            and
            # ... i prosta poprowadzona od punktu (x,y) do nieskonczonosci
            # przecina krawedz pomiedzy i-tym i j-tym punktem...
            ($x
             <
             ( $x[ $j ] - $x[ $i ] ) *
```

```
                ( $y - $y[ $i ] ) / ( $y[ $j ] - $y[ $i ] ) + $x[ $i ] )) {
            $polozenie = not $polozenie; # Zmiana polozenia.
        }
    }

    return $polozenie ? 1 : 0;
}
```

Sprawdzenie parzystości lub nieparzystości liczby przecięć nie wymaga nawet zliczania tych przecięć, gdyż istnieje znacznie szybsza metoda polegająca na przełączaniu wartości zmiennej logicznej `$polozenie`.

Wykorzystując wielokąt z rysunku 10.12, możemy sprawdzić, które z dziewięciu punktów znajdują się wewnątrz tej figury:

```
@wielokat = ( 1, 1,  3, 5,  6, 2,  9, 6,  10, 0,  4,2,  5, -2);
print "( 3, 4): ", punkt_wewnatrz_wielokata( 3, 4, @wielokat ), "\n";
print "( 3, 1): ", punkt_wewnatrz_wielokata( 3, 1, @wielokat ), "\n";
print "( 3,-2): ", punkt_wewnatrz_wielokata( 3,-2, @wielokat ), "\n";
print "( 5, 4): ", punkt_wewnatrz_wielokata( 5, 4, @wielokat ), "\n";
print "( 5, 1): ", punkt_wewnatrz_wielokata( 5, 1, @wielokat ), "\n";
print "( 5,-2): ", punkt_wewnatrz_wielokata( 5,-2, @wielokat ), "\n";
print "( 7, 4): ", punkt_wewnatrz_wielokata( 7, 4, @wielokat ), "\n";
print "( 7, 1): ", punkt_wewnatrz_wielokata( 7, 1, @wielokat ), "\n";
print "( 7,-2): ", punkt_wewnatrz_wielokata( 7,-2, @wielokat ), "\n";
```



Rysunek 10.12. Przykładowy wielokąt z punktami wewnętrznymi i zewnętrznymi

Oto uzyskane wyniki:

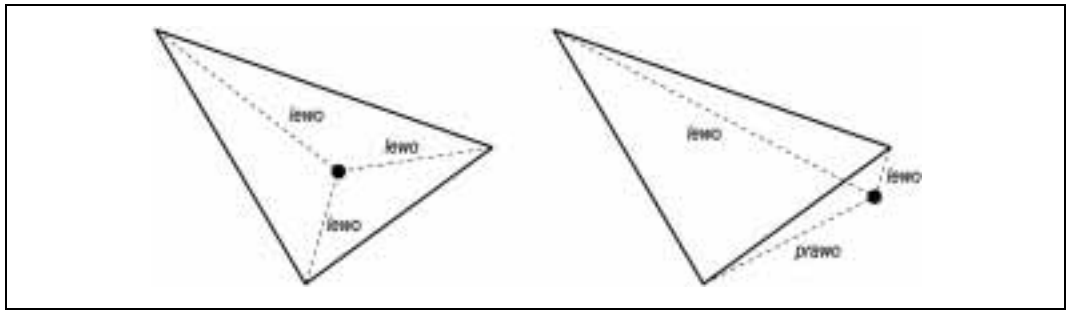
```
(3, 4): 1
(3, 1): 1
(3,-2): 0
(5, 4): 0
(5, 1): 0
(5,-2): 0
(7, 4): 0
(7, 1): 1
(7,-2): 0
```

Udało się nam stwierdzić, że punkty (3, 4), (3, 1) i (7, 1) są położone wewnątrz wielokąta, natomiast pozostałe punkty leżą na zewnątrz.

Punkt wewnątrz trójkąta

W przypadku prostszych wielokątów, takich jak trójkąty, możliwe jest zastosowanie alternatywnego algorytmu. Po wybraniu jednego z wierzchołków trójkąta odbywa się sprawdzanie, czy punkt jest widoczny po lewej czy po prawej stronie. Teraz należy przejść do drugiego wierzchołka i powtórzyć procedurę. Jeśli punkt widoczny jest po innej stronie niż poprzednio, to nie może znajdować się wewnątrz trójkąta. Operacja jest powtarzana także dla trzeciego wierzchołka. Jeśli punkt będzie widoczny zawsze po tej samej stronie (lub leży na krawędzi trójkąta), to można bezpiecznie przyjąć, że znajduje się wewnątrz trójkąta.

Rysunek 10.13 stanowi ilustrację powyższego algorytmu. Kolejne wierzchołki trójkąta są badane w kolejności przeciwnej do kierunku ruchu wskazówek zegara. Punkty znajdujące się wewnątrz trójkąta powinny być widoczne po lewej stronie. Jeśli punkt znajduje się na zewnątrz, to będzie możliwe zaobserwowanie zmiany kierunku.



Rysunek 10.13. Sprawdzanie, czy punkt znajduje się wewnątrz trójkąta

Przedstawiona powyżej metoda została zaimplementowana w podprocedurze `punkt_wewnatrz_trojkata()`:

```
# punkt_wewnatrz_trojkata( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2 )
# Procedura zwraca prawde, jesli punkt ($x,$y) znajduje sie wewnatrz
# trojkata zdefiniowanego przez punkty ($x0, $y0, $x1, $y1, $x2, $y2).
#

sub punkt_wewnatrz_trojkata {
    my ( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2 ) = @_;

    # Procedura kierunek() pochodzi z wczesniejszej czesci rozdzialu.
    my $cw0 = kierunek( $x0, $y0, $x1, $y1, $x, $y );
    return 1 if abs( $cw0 ) < epsilon; # Pierwsza krawedz.

    my $cw1 = kierunek( $x1, $y1, $x2, $y2, $x, $y );
    return 1 if abs( $cw1 ) < epsilon; # Druga krawedz.

    # Niepowodzenie w przypadku zmiany znaku.
    return 0 if ( $cw0 < 0 and $cw1 > 0 ) or ( $cw0 > 0 and $cw1 < 0 );
```

```
    my $cw2 = kierunek( $x2, $y2, $x0, $y0, $x, $y );
    return 1 if abs( $cw2 ) < epsilon; # Trzecia krawedz.

    # Niepowodzenie w przypadku zmiany znaku.
    return 0 if ( $cw0 < 0 and $cw2 > 0 ) or ( $cw0 > 0 and $cw2 < 0 );

    # Sukces!
    return 1;
}
```

Zdefiniujmy teraz trójkąt o wierzchołkach (1, 1), (5, 6) i (9, 3), a następnie sprawdźmy, czy siedem przykładowych punktów znajduje się wewnątrz tego trójkąta:

```
@trojkat = ( 1, 1,  5, 6,  9, 3 );
print "(1, 1): ", punkt_wewnatrz_trojkata( 1, 1,  @trojkat ), "\n";
print "(1, 2): ", punkt_wewnatrz_trojkata( 1, 2,  @trojkat ), "\n";
print "(3, 2): ", punkt_wewnatrz_trojkata( 3, 2,  @trojkat ), "\n";
print "(3, 3): ", punkt_wewnatrz_trojkata( 3, 3,  @trojkat ), "\n";
print "(3, 4): ", punkt_wewnatrz_trojkata( 3, 4,  @trojkat ), "\n";
print "(5, 1): ", punkt_wewnatrz_trojkata( 5, 1,  @trojkat ), "\n";
print "(5, 2): ", punkt_wewnatrz_trojkata( 5, 2,  @trojkat ), "\n";
```

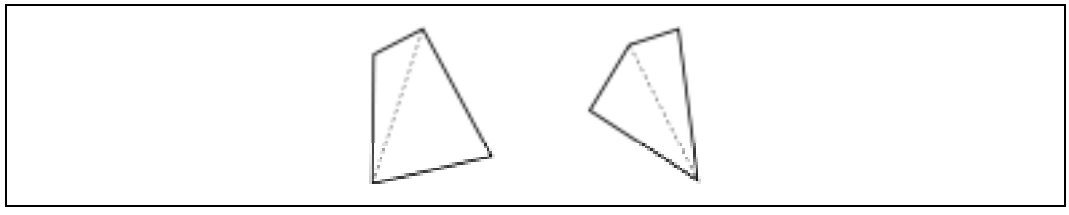
Oto uzyskane wyniki:

```
(1, 1): 1
(1, 2): 0
(3, 2): 1
(3, 3): 1
(3, 4): 0
(5, 1): 0
(5, 2): 1
```

Punkty (1, 2), (3, 4) i (5, 1) znajdują się na zewnątrz trójkąta, natomiast pozostałe punkty są umieszczone wewnątrz niego.

Punkt wewnątrz czworokąta

Każdy *czworokąt* wypukły (wszystkie kwadraty i prostokąty są czworokątami) może być podzielony na dwa trójkąty poprzez połączenie dwóch przeciwległych wierzchołków. Wykorzystując tę cechę i podprocedurę `punkt_wewnatrz_trojkata()`, możemy sprawdzić, czy dany punkt znajduje się wewnątrz czworokąta. Należy uważać jedynie na specjalny rodzaj czworokątów z nakładającymi się wierzchołkami, które mogą ulec redukcji do trójkątów, odcinków, a nawet punktów. Podział czworokąta na dwa trójkąty przedstawiono na rysunku 10.14.

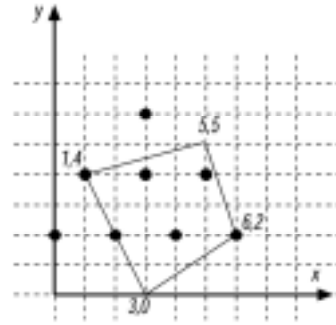


Rysunek 10.14. Podział czworokąta na dwa trójkąty

Poniższa podprocedura `punkt_wewnatrz_czworokata()` dwukrotnie wywołuje procedurę `punkt_wewnatrz_trojkata()` dla każdego trójkąta, który powstał po podziale:

```
# punkt_wewnatrz_czworokata( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 )
#   Procedura zwraca prawde, jesli punkt ($x,$y) znajduje sie wewnatrz
#   czworokata
#   zdefiniowanego przez punkty p0 ($x0,$y0), p1, p2 oraz p3.
#   Wykorzystywana jest podprocedura punkt_wewnatrz_trojkata().
#

sub punkt_wewnatrz_czworokata {
    my ( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 ) = @_;

    return punkt_wewnatrz_trojkata( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2 ) ||
           punkt_wewnatrz_trojkata( $x, $y, $x0, $y0, $x2, $y2, $x3, $y3 )
}
```

Działanie procedury `punkt_wewnatrz_czworokata()` zostanie zademonstrowane na przykładzie czworokąta i punktów pokazanych na rysunku 10.15.



Rysunek 10.15. Ustalenie punktów znajdujących się wewnątrz czworokąta

Wierzchołki czworokąta to (1, 4), (3, 0), (6, 2) i (5, 5), a więc procedurę `punkt_wewnatrz_czworokata()` należy wywołać w następujący sposób:

```
@czworokat = ( 1, 4,  3, 0,  6, 2,  5, 5 );
print "(0, 2): ", punkt_wewnatrz_czworokata( 0, 2,  @czworokat ), "\n";
print "(1, 4): ", punkt_wewnatrz_czworokata( 1, 4,  @czworokat ), "\n";
print "(2, 2): ", punkt_wewnatrz_czworokata( 2, 2,  @czworokat ), "\n";
print "(3, 6): ", punkt_wewnatrz_czworokata( 3, 6,  @czworokat ), "\n";
print "(3, 4): ", punkt_wewnatrz_czworokata( 3, 4,  @czworokat ), "\n";
print "(4, 2): ", punkt_wewnatrz_czworokata( 4, 2,  @czworokat ), "\n";
print "(5, 4): ", punkt_wewnatrz_czworokata( 5, 4,  @czworokat ), "\n";
print "(6, 2): ", punkt_wewnatrz_czworokata( 6, 2,  @czworokat ), "\n";
```

Uzyskane wyniki to:

```
(0, 2): 0
(1, 4): 1
(2, 2): 1
(3, 6): 0
(3, 4): 1
```

```
(4, 2): 1
(5, 4): 1
(6, 2): 1
```

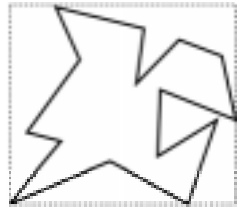
Jedyne punkty znajdujące się na zewnątrz czworokąta to (0, 2) i (3, 6).

Granice

W tej części rozdziału zajmiemy się badaniem granic figur geometrycznych, co może być przydatne do sprawdzenia, czy figury nakładają się. Należy być jednak bardzo uważnym, gdyż uzyskane granice są tylko pierwszym przybliżeniem, a obiekty wklęsłe mogą sprawiać problemy.

Prostopadłościany ograniczające

Prostopadłościan ograniczający to obiekt geometryczny definiowany jako najmniejszy prostopadłościan *d*-wymiarowy zawierający obiekt *d*-wymiarowy, którego krawędzie są równoległe do osi. Prostopadłościany ograniczające są często wykorzystywane w grach komputerowych do wykrywania kolizji obiektów. Przykład takiego prostopadłościanu znajduje się na rysunku 10.16.



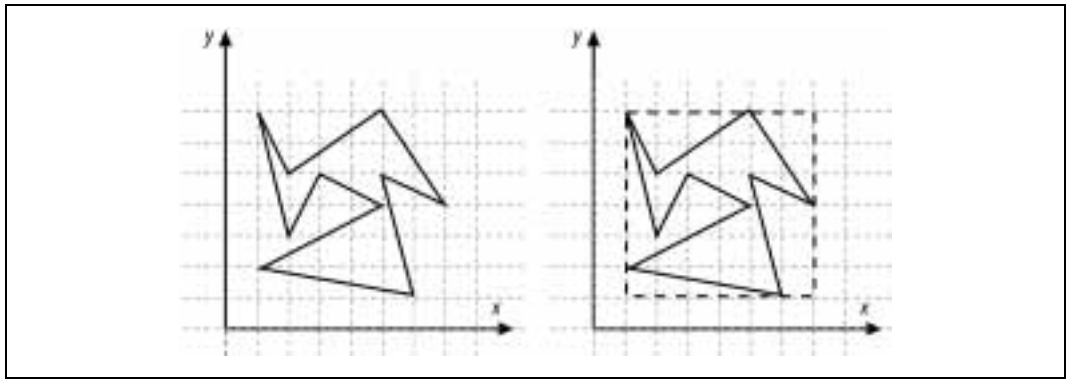
Rysunek 10.16. Wielokąt i jego prostopadłościan ograniczający (zaznaczony kropkowaną linią)

Poniższa podprocedura `prostopadloscian_ograniczajacy()` zwraca tablicę punktów. Jeśli zmienna *$d* ma wartość 2 (dwa wymiary), to prostopadłościan ograniczający będzie miał postać prostokąta. W tym przypadku procedura zwróci cztery elementy będące dwoma wierzchołkami prostokąta.

```
# prostopadloscian_ograniczajacy_punkty($d, @p)
#   Procedura zwraca prostopadloscian ograniczajacy dla zbioru $d-wymiarowych
#   punktow @p.

sub prostopadloscian_ograniczajacy_punkty {
    my ($d, @punkty) = @_;

    my @bb;

    while (my @p = splice @punkty, 0, $d) {
        @bb = prostopadloscian_ograniczajacy($d, @p, @bb); # Zdefiniowane
                                                           # ponizej.
    }
```

```
    return @bb;
}

# prostopadloscian_ograniczajacy($d, @p [,@b])
#   Procedura zwraca prostopadloscian ograniczajacy dla punktow @p w $d
#   wymiarach.
#   @b to opcjonalny, wstepny prostopadloscian ograniczajacy, ktory moze byc
#   uzyty do utworzenia zbiorczego prostopadloscianu ograniczajacego
#   zawierajacego prostopadlosciany odnalezione przez wczesniejsze wywolania
#   tej procedury (ta funkcja jest wykorzystywana przez procedure
#   prostopadloscian_ograniczajacy_punkty()).
#   Prostopadloscian ograniczajacy jest zwracany w postaci listy. Pierwsze
#   $d elementow to minimalne wspolrzedne, a ostatnie $d elementow to
#   wspolrzedne maksymalne.

sub prostopadloscian_ograniczajacy {
    my ( $d, @bb ) = @_; # $d to liczba wymiarow.
    # Usuniecie punktow i pozostawienie prostopadloscianu ograniczajacego.
    my @p = splice( @bb, 0, @bb - 2 * $d );

    @bb = ( @p, @p ) unless @bb;

    # Przeszukanie wszystkich wspolrzednych i zapamietanie ekstremow.
    for ( my $i = 0; $i < $d; $i++ ) {
        for ( my $j = 0; $j < @p; $j += $d ) {
            my $ij = $i + $j;
            # Minima.
            $bb[ $i      ] = $p[ $ij ] if $p[ $ij ] < $bb[ $i      ];
            # Maksima.
            $bb[ $i + $d ] = $p[ $ij ] if $p[ $ij ] > $bb[ $i + $d ];
        }
    }

    return @bb;
}

# prostopadloscian_ograniczajacy_przeciecie($d, @a, @b)
#   Procedura zwraca prawde, jesli podane prostopadlosciany @a i @b przecinaja
#   sie w $d wymiarach. Podprocedura wykorzystana przez funkcje
#   przeciecie_prostych().

sub prostopadloscian_ograniczajacy_przeciecie {
    my ( $d, @bb ) = @_; # Liczba wymiarow i wspolrzedne prostopadloscianow.
    my @aa = splice( @bb, 0, 2 * $d ); # Pierwszy prostopadloscian.
    # (@bb to drugi prostopadloscian.)

    # Prostopadlosciany musza przecinac sie we wszystkich wymiarach.
    for ( my $i_min = 0; $i_min < $d; $i_min++ ) {
        my $i_max = $i_min + $d; # Indeks dla maksimum.
        return 0 if ( $aa[ $i_max ] + epsilon ) < $bb[ $i_min ];
        return 0 if ( $bb[ $i_max ] + epsilon ) < $aa[ $i_min ];
    }

    return 1;
}
```

Wykorzystajmy powyższą procedurę do odnalezienia prostopadłościanu ograniczającego wielokąt z rysunku 10.17. Do procedury `prostopadloscian_ograniczajacy_punkty()` przekazywanych jest 21 argumentów: wymiar 2 oraz 10 par współrzędnych opisujących kolejne wierzchołki figury z rysunku:



Rysunek 10.17. Wielokąt i jego prostopadłościan ograniczający

```
@bb = prostopadloscian_ograniczajacy_punkty(2,
                                1, 2,  5, 4,  3, 5,  2, 3,  1, 7,
                                2, 5,  5, 7,  7, 4,  5, 5,  6, 1), "\n";
print "@bb\n";
```

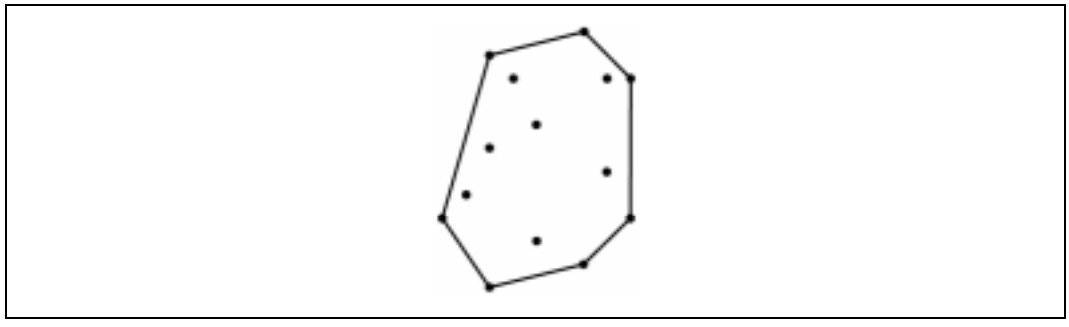
Wynikiem jest lewy dolny (1, 1) i prawy górny (7, 7) wierzchołek kwadratu:

```
1 1 7 7
```

Powłoka wypukła

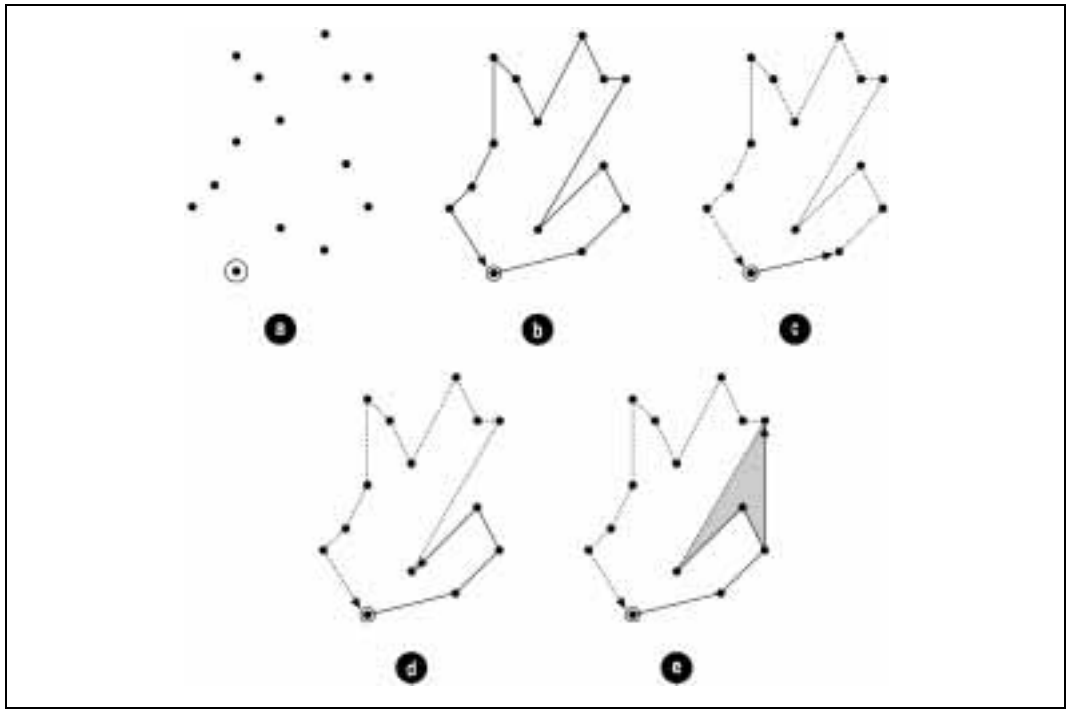
Powłoka wypukła przypomina prostopadłościan ograniczający, ale lepiej otacza wybraną figurę geometryczną, ponieważ nie musi być prostopadłościanem. Powłoka wypukła jest „rozciągana" wokół punktów zewnętrznych obiektu. Aby to sobie wyobrazić, przypomnij sobie artystę nazwiskiem Christo, który owija folią słynne budowle. Folia tworzy właśnie powłokę wypukłą.

Powłoka wypukła w dwóch wymiarach to zbiór krawędzi tworzących wielokąt wypukły, natomiast w trzech wymiarach jest to wielościan wypukły, którego ściany mają trójkątny kształt. Dwuwymiarową powłokę wypukłą przedstawiono na rysunku 10.18.



Rysunek 10.18. Powłoka wypukła dla zbioru punktów

Najbardziej znany algorytm do tworzenia powłoki wypukłej w dwóch wymiarach nosi nazwę *algorytmu Grahama*. Najpierw należy wyszukać jeden punkt, który na pewno stanowi część powłoki. Zwykle jest to punkt o najmniejszej współrzędnej x lub y, co pokazano na rysunku 10.19(a).



Rysunek 10.19. Algorytm Grahama — wyszukiwanie punktu początkowego

Wszystkie pozostałe punkty są następnie sortowane w zależności od kąta tworzonego w stosunku do punktu startowego, co pokazano na rysunku 10.19(b). Dzięki użytej metodzie wyboru punktu startowego miary wszystkich kątów zawierają się w przedziale od 0 do π.

Tworzenie powłoki rozpoczyna się od punktu startowego, po czym następuje przejście do pierwszego z posortowanych punktów. Pewien problem może powstać, jeśli kolejny punkt znajduje się bezpośrednio na linii prostej. Wymaga to użycia bardziej zaawansowanego sortowania; jeśli kąty są równe, to należy posortować punkty według współrzędnych x i y.

Jeśli w czasie przechodzenia do kolejnego punktu należy skręcić w lewo, to ten punkt zostaje dodany do powłoki.

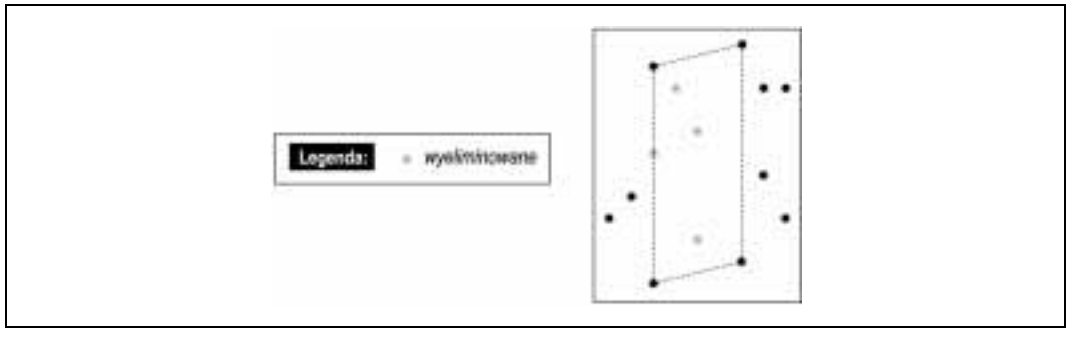
W przypadku skrętu w prawo ostatni dodany punkt nie może stanowić części powłoki i musi być usunięty. Oznacza to często również konieczność usunięcia poprzednich punktów aż do momentu, w którym znowu możliwy będzie skręt w lewo. Ciągłe zwiększanie i zmniejszanie wielkości powłoki sugeruje zastosowanie stosu opisanego w rozdziale 2. poświęconym strukturom danych.

Jak łatwo zauważyć, reguła zakazująca skrętu w prawo pozwala na pominięcie wszystkich wklęsłości (zacieniony obszar na rysunku 10.19). Cały proces wyszukiwania punktów jest powtarzany aż do momentu powrotu do punktu startowego. Poniższa procedura `powloka_wypukla_graham()` stanowi implementację algorytmu Grahama:

```
# powloka_wypukla_graham( @xy )
#   Procedura oblicza powloke wypukla dla punktow @xy z uzyciem
#   algorytmu Grahama. Procedura zwraca kolejne punkty powloki w postaci.
#   listy ($x,$y,...).

sub powloka_wypukla_graham {
    my ( @xy ) = @_;

    my $n = @xy / 2;
    my @i = map { 2 * $_ } 0 .. ( $#xy / 2 ); # Parzyste indeksy.
    my @x = map { $xy[ $_ ]     } @i;
    my @y = map { $xy[ $_ + 1 ] } @i;

    # Najpierw nalezy odnalezc najmniejsza wartosc y z najmniejsza wartoscia x.

    # $ymin to najmniejsza wartosc y w danym momencie, @xmini zawiera indeksy
    # najmniejszych wartosci y, $xmini to indeks najmniejszej wartosci x,
    # a $xmin to najmniejsza wartosc x.
    my ( $ymin, $xmini, $xmin, $i );

    for ( $ymin = $ymax = $y[ 0 ], $i = 1; $i < $n; $i++ ) {
        if ( $y[ $i ] + epsilon < $ymin ) {
            $ymin  = $y[ $i ];
            @xmini = ( $i );
        } elsif ( abs( $y[ $i ] - $ymin ) < epsilon ) {
            $xmini = $i # Zapamietanie indeksu najmniejszej wartosci x.
                if not defined $xmini or $x[ $i ] < $xmini;
        }
    }

    $xmin  = $x[ $xmini ];
    splice @x, $xmini, 1;         # Usuniecie punktu minimum.
    splice @y, $xmini, 1;

    my @a = map {  # Posortowanie punktow zgodnie z wartosciami katow.
                   atan2( $y[ $_ ] - $ymin,
                          $x[ $_ ] - $xmin)
                } 0 .. $#x;

    # Rzadki przypadek transformacji Schwartza, ktora daje w wyniku posortowane
    # indeksy. Pozwala to na wielokrotne sortowanie, czyli uzycie permutacji.

    my @j = map { $_->[ 0 ] }
               sort {      # Posortowanie wedlug katow, wartosci x i wartosci y.
                      return $a->[ 1 ] <=> $b->[ 1 ]             ||
                             $x[ $a->[ 0 ] ] <=> $x[ $b->[ 0 ] ] ||
                             $y[ $a->[ 0 ] ] <=> $y[ $b->[ 0 ] ];
                     }
                    map { [ $_, $a[ $_ ] ] } 0 .. $#a;

    @x = @x[ @j ];          # Permutacja.
    @y = @y[ @j ];
    @a = @a[ @j ];

    unshift @x, $xmin;      # Przywrocenie punktu minimum.
    unshift @y, $ymin;
    unshift @a, 0;
```

```
    my @h = ( 0, 1 );        # Powloka.
    my $cw;

  # Cofniecie: zmniejszenie powloki, jesli nalezy skrecic w prawo lub jesli
  # nie ma mozliwosci skretu.
    for ( $i = 2; $i < $n; $i++ ) {
        while (
            kierunek( $x[ $h[ $#h - 1 ] ],
                      $y[ $h[ $#h - 1 ] ],
                      $x[ $h[ $#h ] ],
                      $y[ $h[ $#h ] ],
                      $x[ $i ],
                      $y[ $i ] ) > epsilon
             and @h >= 2 ) {  # Te 2 punkty zawsze beda stanowily czesc powloki.
           pop @h;
        }
        push @h, $i;  # Zwiekszenie powloki.
    }

  # Przeniesienie wartosci x i y powloki z powrotem na liste, a nastepnie powrot.
    return map { ( $x[ $_ ], $y[ $_ ] ) } @h;
}
```

Algorytm Grahama można przyspieszyć poprzez zredukowanie liczby badanych punktów. Jednym ze sposobów dokonania tego jest *eliminacja wnętrza* — odrzucane są wszystkie punkty, które absolutnie *nie* mogą stanowić części powłoki. Eliminacja punktów jest zależna od ich rozkładu; jeśli rozkład jest typu losowego lub punkty rozciągają się w dwóch kierunkach, to bardzo pomocne będzie zastosowanie prostokąta rozciągającego się między punktami znajdującymi się najbliżej rogów. Wszystkie punkty znajdujące się wewnątrz tego prostokąta można bezpiecznie usunąć, co pokazano na rysunku 10.20.



Rysunek 10.20. Algorytm Grahama — eliminacja wewnętrznych punktów

Punkty znajdujące się najbliżej rogów można odnaleźć poprzez minimalizację i maksymalizację sum i różnic punktów:

- najmniejsza suma — lewy dolny róg,
- największa suma — prawy górny róg,
- najmniejsza różnica — lewy górny róg,
- największa różnica — prawy dolny róg.

Oto implementacja tej metody w Perlu:

```
# Wyszukanie największych i najmniejszych sum oraz roznic
# (a raczej indeksow tych punktow).

my @sortowanie_wedlug_sumy =
       map { $_->[ 0 ] }
           sort {$a->[ 1 ] <=> $b->[ 1 ] }
              map { [ $_, $x[ $_ ] + $y[ $_ ] ] } 0..$#x;

my @sortowanie_wedlug_roznicy =
       map { $_->[ 0 ] }
           sort {$a->[ 1 ] <=> $b->[ 1 ] }
              map { [ $_, $x[ $_ ] - $y[ $_ ] ] } 0..$#x;

my $ld = @sortowanie_wedlug_sumy [ 0 ];    # Lewy dolny rog prostokatu do
                                           # usuwania punktow.
My $pg = @sortowanie_wedlug_sumy [-1 ];    # Prawy gorny rog.
my $lg = @sortowanie_wedlug_roznicy [ 0 ]; # Lewy gorny rog.
My $pd = @sortowanie_wedlug_roznicy [-1 ]; # Prawy dolny rog.
```

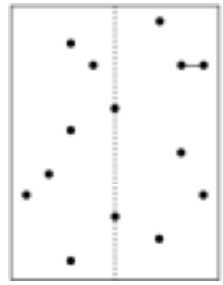
Stosując tę metodę, należy pamiętać o pewnych ograniczeniach. Bezpiecznie można wyeliminować *tylko* te punkty, które znajdują się wewnątrz prostokąta. Punkty znajdujące się na bokach prostokąta mogą stanowić część powłoki, natomiast punkty w rogach z całą pewnością *są* jej częścią. Na wszelki wypadek można utworzyć drugi prostokąt, który będzie minimalnie mniejszy od pierwszego (*epsilon*). Jeśli wartość epsilon zostanie wybrana prawidłowo, to punkty wewnątrz mniejszego prostokąta będzie można usunąć bez obaw.

Procedura oparta na algorytmie Grahama działa w czasie $O(N \log N)$, co jest optymalnym wynikiem.

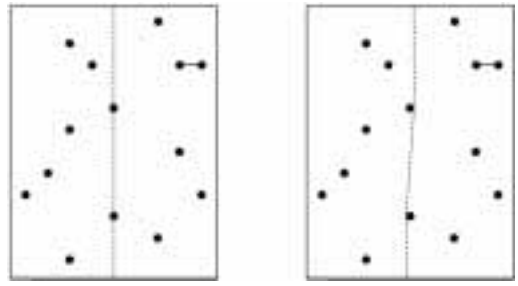
Najbliższa para punktów

W jaki sposób można wybrać dwa punkty, które leżą najbliżej siebie? Narzuca się tu oczywiste rozwiązanie polegające na obliczeniu odległości dla każdej możliwej pary punktów. Choć ta metoda zadziała poprawnie, to jej czas pracy wynosi aż $O(N2)$. Wyszukiwanie najbliższych punktów ma wiele rzeczywistych zastosowań; przykładem może być oprogramowanie do symulacji ruchu lotniczego – dwa odrzutowce nie powinny znaleźć się w tym samym miejscu. Prostopadłościany ograniczające mogą być wykorzystane do wykrycia kolizji, natomiast sprawdzanie najbliższych punktów uniemożliwi katastrofę. Do przedstawienia tego problemu użyjemy zbioru punktów z rysunku 10.21.

W jaki sposób można odnaleźć najbliższą parę punktów? Przydatna tu będzie pewna wewnętrzna właściwość punktów, która mówi, że punkt znajdujący się po lewej stronie będzie prawdopodobnie bliższy innym punktom po tej samej stronie, niż punktom po stronie przeciwnej. Po raz kolejny skorzystamy tu z paradygmatu „dziel i rządź", który został przedstawiony w podrozdziale poświęconym rekurencji. Grupa punktów zostanie podzielona na lewą i prawą połowę, co pokazano na rysunku 10.22.



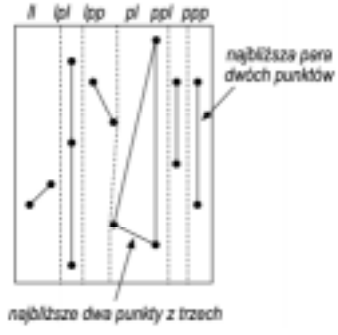
Rysunek 10.21. Grupa punktów i najbliższa para



Rysunek 10.22. Rekurencyjne dzielenie na pół — widok fizyczny i logiczny

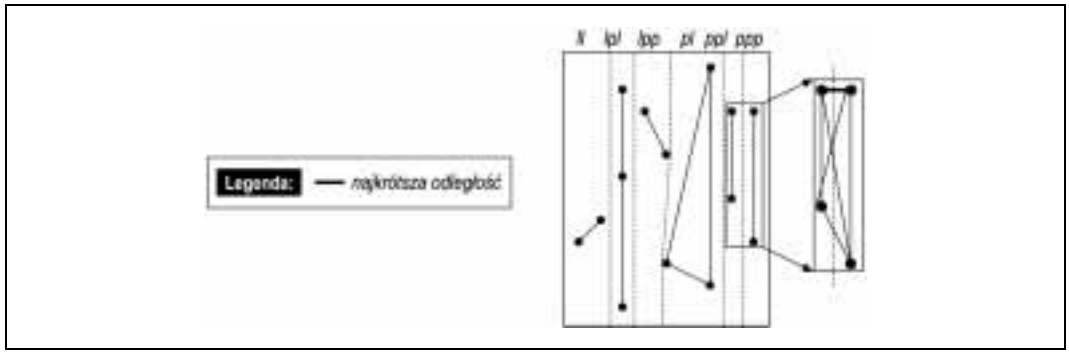
Dlaczego linia podziału logicznego na rysunku 10.22 jest zakrzywiona? Dzieje się tak, gdyż linia podziału zawiera dwa punkty mające dokładnie taką samą współrzędną x. Dokonano więc drobnych zmian, aby lepiej rozdzielić obie połówki.

Na rysunku 10.23 przedstawiono pionowe wycinki utworzone przez rekurencyjny podział zbioru. Poszczególne wycinki są odpowiednio oznakowane, na przykład *lpp* oznacza wycinek powstały poprzez podział w lewo, po którym nastąpiły dwa podziały w prawo.



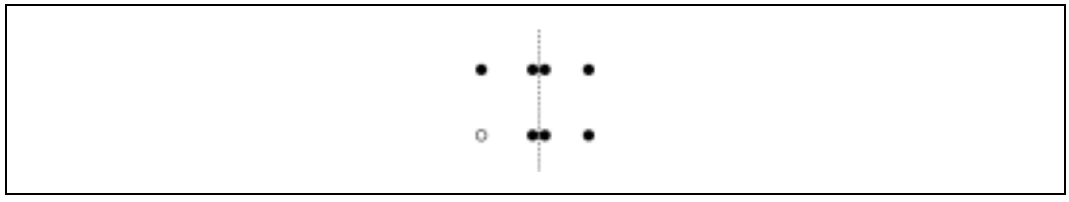
Rysunek 10.23. Pionowe wycinki i najbliższe pary punktów w poszczególnych wycinkach

Rekurencja zostaje przerwana, kiedy wycinek zawiera tylko dwa lub trzy punkty. W takim przypadku najkrótsza odległość (czyli najbliższa para punktów) może być odnaleziona w bardzo prosty sposób, co pokazano na rysunku 10.24.



Rysunek 10.24. Łączenie wycinków podzielonych rekurencyjnie

Jaką operację należy wykonać powracając z rekurencji? W każdym wycinku odnaleziono najbliższą parę punktów, ale nie można po prostu wybrać najkrótszej odległości dla wszystkich wycinków, gdyż najbliższa para punktów z całego zbioru może znaleźć się po dwóch stronach linii podziału. Tę sytuację pokazano na rysunku 10.25.



Rysunek 10.25. Przeszukiwanie w czasie łączenia; aktualny punkt oznaczono białym kolorem

Sztuka polega na tym, aby w przylegających połówkach wyszukać punkty, których odległość od linii podziału jest mniejsza, niż najmniejsza odnaleziona dotąd odległość. Po wykonaniu tych operacji dla każdej linii podziału należy zbadać wszystkie uzyskane punkty w kolejności zależnej od współrzędnej y. Dla każdego punktu należy sprawdzić maksymalnie siedem innych punktów, co pokazano na rysunku 10.25.

Kod Perla implementujący opisaną powyżej metodę będzie dość skomplikowany, ponieważ konieczne jest jednoczesne sortowanie zbiorów punktów według wielu kryteriów. Punkty są przechowywane w oryginalnej kolejności oraz posortowane poziomo (w ten sposób zbiór punktów jest dzielony horyzontalnie) i pionowo. Wszystkie te widoki tego samego zbioru punktów są zaimplementowane poprzez obliczenie różnych *wektorów permutacji* jako tablic Perla, na przykład tablica `@yoi` zawiera kolejne punkty posortowane od dołu do góry zgodnie ze współrzędnymi y.

Zwróć uwagę na to, że zastosowanie podstawowej techniki „dziel i rządź" sprawia, że czas pracy algorytmu będzie dość długi — $O(N \log N)$, aczkolwiek rekurencja wymaga

tylko $O(N)$ operacji. Ciągłe sortowanie danych wewnątrz kodu rekurencyjnego mogłoby znacznie wpłynąć na ocenę $O(N \log N)$, przez co sortowanie poziome i pionowe jest wykonywane tylko raz, przed rozpoczęciem rekurencji.

Oto przerażająco długa podprocedura `najblizsze_punkty()`:

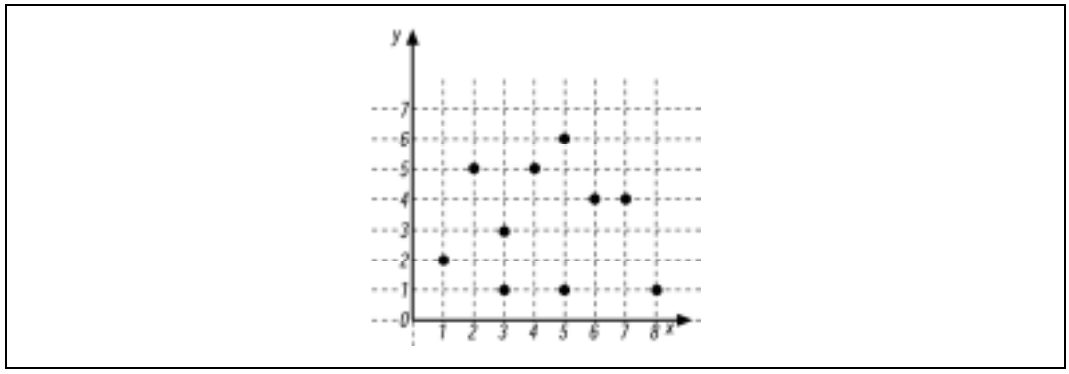
```
sub najblizsze_punkty {
    my ( @p ) = @_;

    return () unless @p and @p % 2 == 0;

    my $nieposortowane_x = [ map { $p[ 2 * $_     ] } 0..$#p/2 ];
    my $nieposortowane_y = [ map { $p[ 2 * $_ + 1 ] } 0..$#p/2 ];

    # Oblicza permutacje i indeksy porzadkowe.

    # Indeks permutacji wspolrzednej x.
    #
    # Jesli tablica @$nieposortowane_x to (18, 7, 25, 11), to tablica @xpi
    # bedzie zawierala (1, 3, 0, 2), np. $xpi[0] == 1 oznacza, ze
    # $posortowane_x[0] znajduje sie w $nieposortowane_x->[1].
    #
    # Ta operacja jest wykonywana, aby umozliwic sortowanie @$nieposortowane_x
    # do @posortowane_x z zachowaniem mozliwosci przywrocenia oryginalnej
    # kolejnosci w postaci @posortowane_x[@xpi].
    # Jest to niezbedne ze wzgledu na koniecznosc sortowania punktow wedlug
    # wspolrzednych x i y oraz identyfikacji wyniku na podstawie oryginalnych
    # indeksow punktow: "12 punkt i 45 punkt to najblizsza para ".

    my @xpi = sort { $nieposortowane_x->[ $a ] <=> $nieposortowane_x->[ $b ] }
                   0..$#$nieposortowane_x;

    # Indeks permutacji wspolrzednych y.
    #
    my @ypi = sort { $nieposortowane_y->[ $a ] <=> $nieposortowane_y->[ $b ] }
                   0..$#$nieposortowane_y;

    # Indeks porzadkowy wspolrzednych y.
    #
    # Indeks porzadkowy to odwrotnosc indeksu permutacji. Jesli tablica
    # @$nieposortowane_y to (16, 3, 42, 10) a @ypi to (1, 3, 0, 2), to tablica
    # @yoibedzie miala postac (2, 0, 3, 1), na przyklad $yoi[0] == 1 oznacza,
    # ze $nieposortowane_y->[0] to $posortowane_y[1].

    my @yoi;
    @yoi[ @ypi ] = 0..$#ypi;

    # Rekurencja w celu odnalezienia najblizszych punktow.
    my ( $p, $q, $d ) = _najblizsze_punkty_rekurencja( [
       @$nieposortowane_x[@xpi] ],
                                                       [
       @$nieposortowane_y[@xpi] ],
                                                       \@xpi, \@yoi, 0, $#xpi
                                                    );

    my $pi = $xpi[ $p ];                            # Odwrotna permutacja.
    my $qi = $xpi[ $q ];

    ( $pi, $qi ) = ( $qi, $pi ) if $pi > $qi;       # Najpierw mniejszy
                                                    # identyfikator.
    return ( $pi, $qi, $d );
}
```

```
sub _najblizsze_punkty_rekurencja {
    my ( $x, $y, $xpi, $yoi, $x_l, $x_p ) = @_;

    # $x, $y:  odwolania tablicowe do wspolrzednych x i y punktow
    # $xpi:    indeksy permutacji x: obliczone przez
    # najblizsze_punkty_rekurencja()
    # $yoi:    indeksy kolejnosci y: obliczone przez
    # najblizsze_punkty_rekurencja()
    # $x_l:    lewa granica aktualnie sprawdzanego zbioru punktow
    # $x_p:    prawa granica aktualnie sprawdzanego zbioru punktow
    #          Oznacza to, iz sprawdzane beda tylko punkty $x->[$x_l..$x_p]
    #          i $y->[$x_l..$x_p].

    my $d;     # Odnaleziona najblizsza odleglosc.
    my $p;     # Indeks drugiego konca minimalnej odleglosci.
    my $q;     # Jak wyzej.

    my $N = $x_p - $x_l + 1;      # Liczba interesujacych punktow.

    if ( $N > 3 ) {               # Duza liczba punktow. Rekurencja!
        my $x_lp = int( ( $x_l + $x_p ) / 2 ); # Prawa granica lewej polowki.
        my $x_pl = $x_lp + 1;                  # Lewa granica prawej polowki.

        # Najpierw rekurencja w celu zbadania polowek.

        my ( $p1, $q1, $d1 ) =
            _najblizsze_punkty_rekurencja( $x, $y, $xpi, $yoi, $x_l, $x_lp );
        my ( $p2, $q2, $d2 ) =
            _najblizsze_punkty_rekurencja( $x, $y, $xpi, $yoi, $x_pl, $x_p );

        # Teraz polaczenie wynikow obu polowek.

        # Ustawienie $d, $p, $q na najkrotsza odleglosc oraz indeksy
        # odnalezionej najblizszej pary punktow.

        if ( $d1 < $d2 ) { $d = $d1;  $p = $p1;  $q = $q1 }
        else             { $d = $d2;  $p = $p2;  $q = $q2 }

        # Sprawdzenie obszaru laczenia.

        # Wspolrzedna x miedzy lewa i prawa polowka.
        my $x_d = ( $x->[ $x_lp ] + $x->[ $x_pl ] ) / 2;

        # Indeksy potencjalnych punktow: te pary punktow znajduja sie po obu
        # stronach linii podzialu i moga znajdowac sie blizej siebie, niz
        # dotychczasowa najlepsza para punktow.
        #
        my @xi;

        # Odnalezienie potencjalnych punktow z lewej polowki.

        # Lewa granica lewego segmentu z potencjalnymi punktami.
        my $x_ll;

        if ( $x_lp == $x_l ) { $x_ll = $x_l }
        else {                                      # Przeszukiwanie binarne.
            my $x_ll_lo = $x_l;
            my $x_ll_hi = $x_lp;
            do { $x_ll = int( ( $x_ll_lo + $x_ll_hi ) / 2 );
                 if ( $x_d - $x->[ $x_ll ] > $d ) {
                    $x_ll_lo = $x_ll + 1;
                 } elsif ( $x_d - $x->[ $x_ll ] < $d ) {
                    $x_ll_hi = $x_ll - 1;
                 }
```

```
    } until $x_ll_lo > $x_ll_hi
        or ( $x_d - $x->[ $x_ll ] < $d
             and ( $x_ll == 0 or
                   $x_d - $x->[ $x_ll - 1 ] > $d ) );
}
push @xi, $x_ll..$x_lp;

# Odnalezienie potencjalnych punktow z prawej polowki.

# Prawa granica prawego segmentu z potencjalnymi punktami.
my $x_pp;

if ( $x_pl == $x_p ) { $x_pp = $x_p }
else {                                          # Przeszukiwanie binarne.
    my $x_pp_lo = $x_pl;
    my $x_pp_hi = $x_p;
    do { $x_pp = int( ( $x_pp_lo + $x_pp_hi ) / 2 );
         if ( $x->[ $x_pp ] - $x_d > $d ) {
            $x_pp_hi = $x_pp - 1;
         } elsif ( $x->[ $x_pp ] - $x_d < $d ) {
            $x_pp_lo = $x_pp + 1;
         }
    } until $x_pp_hi < $x_pp_lo
        or ( $x->[ $x_pp ] - $x_d < $d
             and ( $x_pp == $x_p or
                   $x->[ $x_pp + 1 ] - $x_d > $d ) );
}
push @xi, $x_pl..$x_pp;

# Uzyskano juz liste potencjalnych punktow. Czy spelnia one nasze
# warunki?
# Sprawdzanie jest dosc skomplikowane.

# Najpierw posortowanie punktow wedlug oryginalnych indeksow.

my @x_by_y   = @$yoi[ @$xpi[ @xi ] ];
my @i_x_by_y = sort { $x_by_y[ $a ] <=> $x_by_y[ $b ] }
               0..$#x_by_y;
my @xi_by_yi;
@xi_by_yi[ 0..$#xi ] = @xi[ @i_x_by_y ];

my @xi_by_y = @$yoi[ @$xpi[ @xi_by_yi ] ];
my @x_by_yi = @$x[ @xi_by_yi ];
my @y_by_yi = @$y[ @xi_by_yi ];

# Zbadanie kazdej potencjalnej pary punktow (pierwszy punkt
# z lewej polowki, drugi punkt z prawej polowki).

for ( my $i = 0; $i <= $#xi_by_yi; $i++ ) {
    my $i_i = $xi_by_y[ $i ];
    my $x_i = $x_by_yi[ $i ];
    my $y_i = $y_by_yi[ $i ];
    for ( my $j = $i + 1; $j <= $#xi_by_yi; $j++ ) {
        # Pominiecie punktow, dla ktorych odleglosc nie moze byc
        # mniejsza, niz dla dotychczasowej najlepszej pary.
        last if $xi_by_y[ $j ] - $i_i > 7; # Zbyt daleko?
        my $y_j = $y_by_yi[ $j ];
        my $dy = $y_j - $y_i;
        last if $dy > $d;                  # Zbyt wysoko?
        my $x_j = $x_by_yi[ $j ];
        my $dx = $x_j - $x_i;
        next if abs( $dx ) > $d;           # Zbyt szeroko?
```

```
                # Program dotarl az tutaj?  Byc moze odnaleziono zwyciezce.
                # Nalezy sprawdzic odleglosc i zaktualizowac zmienne.
                my $d3 = sqrt( $dx**2 + $dy**2 );
                if ( $d3 < $d ) {
                    $d = $d3;
                    $p = $xi_by_yi[ $i ];
                    $q = $xi_by_yi[ $j ];
                }
            }
        }
    } elsif ( $N == 3 ) {      # Tylko trzy punkty? Rekurencja nie jest
                               # potrzebna.
        my $x_m = $x_l + 1;
        # Porownanie kwadrator sum. Pierwiastkowanie zostanie wykonane pozniej.
        my $s1 = ($x->[ $x_l ]-$x->[ $x_m ])**2 +
                 ($y->[ $x_l ]-$y->[ $x_m ])**2;
        my $s2 = ($x->[ $x_m ]-$x->[ $x_p ])**2 +
                 ($y->[ $x_m ]-$y->[ $x_p ])**2;
        my $s3 = ($x->[ $x_l ]-$x->[ $x_p ])**2 +
                 ($y->[ $x_l ]-$y->[ $x_p ])**2;
        if ( $s1 < $s2 ) {
            if ( $s1 < $s3 )  { $d = $s1;  $p = $x_l;  $q = $x_m }
            else              { $d = $s3;  $p = $x_l;  $q = $x_p }
        } elsif ( $s2 < $s3 ) { $d = $s2;  $p = $x_m;  $q = $x_p }
          else                { $d = $s3;  $p = $x_l;  $q = $x_p }

        $d = sqrt $d;
    } elsif ( $N == 2 ) {         # Tylko dwa punkty? Rekurencja nie jest
                                  # potrzebna.
        $d = sqrt(($x->[ $x_l ]-$x->[ $x_p ])**2 +
                  ($y->[ $x_l ]-$y->[ $x_p ])**2);
        $p = $x_l;
        $q = $x_p;
    } else {                      # Mniej niz dwa punkty?  Dziwne.
        return ( );
    }

    return ( $p, $q, $d );
}
```

Czas pracy procedury `najblizsze_punkty()` wynosi $O(M \log M)$, co jest dobrą wiadomością. Przetestujmy działanie programu wykorzystując do tego punkty z rysunku 10.26.



Rysunek 10.26. Wyszukiwanie najbliższej pary

Wybranie najbliższej pary punktów ze zbioru punktów przedstawionych na rysunku 10.26 może być wykonane w następujący sposób:

```
@para_punktow = najblizsze_punkty( 1, 2,  2, 5,  3, 1,  3, 3,  4, 5,
                                   5, 1,  5, 6,  6, 4,  7, 4,  8, 1 ), "\n";
print "@para_punktow \n";
```

Uzyskany wynik to:

```
7 8 1
```

Udało się nam stwierdzić, że ósmy (6, 4) i dziewiąty (7, 4) punkt stanowią najbliższą parę punktów (tablice Perla są indeksowane od 0), a odległość między nimi wynosi 1.

Algorytmy geometryczne — podsumowanie

Algorytmy geometryczne są często oparte na znajomych wzorach, ale w czasie ich implementacji należy zachować ostrożność. Przekształcanie wzorów geometrycznych do postaci programów komputerowych nie zawsze jest takie proste, jak to może się wydawać. Głównym źródłem problemów jest konflikt między idealnymi wartościami matematycznymi i niedokładnymi reprezentacjami liczb rzeczywistych w komputerach. Choć z teoretycznego punktu widzenia wydaje się, że dany punkt leży na przecięciu prostych $x - 1$ i $1 - 2x$, to komputer może uważać inaczej. Również koło o promieniu 1 nie zawiera dokładnie π pikseli.

Moduły graficzne CPAN

Przedstawione w tym rozdziale algorytmy tak naprawdę nie wyświetliły na ekranie ani jednego piksela. Aby tak się jednak stało, należy wykorzystać jeden z pakietów graficznych dostępnych pod adresem *http://www.perl.com/CPAN/modules*. Większość tych modułów stanowi interfejsy do zewnętrznych bibliotek, które również powinny być zainstalowane w systemie. Więcej informacji na ten temat można znaleźć w dokumentacji poszczególnych modułów.

Grafika dwuwymiarowa

W bibliotece CPAN dostępnych jest pięć modułów służących do manipulacji obrazami dwuwymiarowymi: Perl-Gimp, GD, Image::Size, PerlMagick i PGPLOT.

Perl-Gimp

Gimp to potężne narzędzie dla systemu Linux, przypominające trochę Adobe Photoshopa; więcej informacji znajduje się na stronie *http://www.gimp.org*. Napisany przez Marca Lehmana Perl-Gimp to interfejs API, który pozwala na uzyskanie zaawansowanych efektów graficznych bezpośrednio w Perlu.

GD

Utworzony przez Lincolna D. Steina moduł GD stanowi interfejs do biblioteki *libgd*, która umożliwia „rysowanie" obrazów GIF. Poniższy kod tworzy obraz GIF zawierający okrąg.

```
use GD;

# Utworzenie rysunku.
my $gif = new GD::Image(100, 100);

# Alokacja kolorow.
my $bialy = $gif->colorAllocate(255, 255, 255);
my $czerwony = $gif->colorAllocate(255,   0,   0);

# Kolor tla.
$gif->transparent($bialy);

# Okrag.
$gif->arc(50, 50,        # Srodek x, y.
          30, 30,        # Szerokosc, wysokosc.
          0, 360,        # Poczatkowy i koncowy kat.
          $czerwony)     # Kolor.

# Zapisanie rysunku.
open(GIF, ">okrag.gif") or die "otwarcie pliku nie powiodlo sie: $!\n";
binmode GIF;
print GIF $gif->gif;
close GIF;
```

Image::Size

Randy J. Ray napisał specjalny moduł o nazwie Image::Size, który podgląda pliki graficzne i pobiera informacje o ich wielkości i rozmiarach. Może się wydawać, że to mało przydatne dane, ale są one bardzo często wykorzystywane. Kiedy serwer internetowy przesyła stronę WWW, informacja o wielkości rysunków powinna być przekazana jak najszybciej (przed przesłaniem samych plików), dzięki czemu przeglądarka internetowa natychmiast przygotuje miejsce dla tych rysunków. Pozwoli to na znacznie szybsze renderowanie strony, ponieważ układ strony nie zmieni się gwałtownie po odebraniu wszystkich obrazków.

PerlMagick

Moduł PerlMagick, autorstwa Kyle'a Shortera, stanowi interfejs do *ImageMagick* — bardzo rozbudowanej biblioteki do konwersji i obróbki obrazów. Dzięki temu modułowi możliwa jest konwersja plików graficznych między różnymi formatami, a także stosowanie różnego rodzaju filtrów — od zmiany nasycenia barw do efektów specjalnych. Więcej informacji znajduje się pod adresem *http://www.perldoc.com/cpan/Image/Magick.html*.

PGPLOT

Napisany przez Karla Glazebrooka moduł PGPLOT jest interfejsem do biblioteki graficznej PGPLOT, która może być wykorzystana na przykład do rysowania obrazów z etykietami. PGPLOT w połączeniu z językiem numerycznym PDL (kolejny moduł Perla; jego

krótki opis znajdziesz w rozdziale 7.) staje się bardzo potężnym narzędziem, ponieważ wszystkie funkcje Perla stają się dostępne w PDL. Więcej informacji można uzyskać pod adresem *http://www.ast.cam.ac.uk/AAO/local/www/kgb/pgperl/*.

Wykresy i grafika biznesowa

Jeśli przez słowo „grafika" rozumiesz „grafikę biznesową" (różnego rodzaju wykresy kołowe i słupkowe), to warto zapoznać się z modułami Chart i GIFgraph autorstwa Davida Bonnera i Martiena Verbruggena. Oba moduły można wykorzystać na przykład do szybkiego tworzenia raportów. Wymagana jest obecność modułu GD.

Modelowanie trójwymiarowe

Jeszcze kilka lat temu rzeczywiste modelowanie trójwymiarowe nie było możliwe na komputerach, na które każdy może sobie pozwolić. Do tego typu zadań można wykorzystać trzy moduły CPAN o nazwach OpenGL, Renderman i VRML.

OpenGL

Stan Melax napisał moduł OpenGL, który implementuje interfejs OpenGL w Perlu. OpenGL to otwarta wersja języka GL firmy Silicon Graphics; jest to język modelowania trójwymiarowego, który pozwala na stworzenie kompletnych „światów" ze złożonymi obiektami i oświetleniem. Popularna gra Quake została wyrenderowana właśnie poprzez OpenGL. Dostępna jest także darmowa implementacja OpenGL o nazwie Mesa; więcej informacji znajduje się pod adresem *http://www.mesa3d.org*.

Renderman

Moduł Renderman, autorstwa Glenna M. Lewisa, to interfejs do fotorealistycznego systemu modelowania Renderman firmy Pixar. Teraz każdy może rozpocząć samodzielne tworzenie filmu „Toy Story" w Perlu.

VRML

Hartmut Palm zaimplementował interfejs Perla dla języka Virtual Reality Markup Language (VRML), który pozwala na tworzenie świata trójwymiarowego. Goście odwiedzający witrynę internetową z kodem VRML mogą spacerować po wirtualnym świecie (oczywiście jeśli mają odpowiednią wtyczkę).

Graficzny interfejs użytkownika

Do utworzenia własnej aplikacji graficznej, która będzie niezależna od Internetu, wymagane jest użycie jednego z opisanych tu pakietów. Perl/Tk to zdecydowanie najbardziej zaawansowany system oferujący największą liczbę funkcji.

Perl/Tk

Moduł Perl/Tk, autorstwa Nicka Ing-Simmonsa, to najlepsze narzędzie do tworzenia elementów graficznego interfejsu użytkownika w Perlu.[4] Moduł działa w systemie X11 oraz w systemach Windows 95, 98, NT i 2000.

Praca w Perl/Tk jest bardzo prosta. Poniżej przedstawiono bardzo krótki program, który wyświetla przycisk.

```
use Tk;
$MW = MainWindow->new;
$witaj = $MW->Button(
       -text    => 'Witaj swiecie',
       -command => sub { print STDOUT "Witaj swiecie!\n"; exit; },
);
$witaj->pack;
MainLoop;
```

Z przyciskiem powiązana jest określona akcja — po jego wciśnięciu w oknie terminala wyświetlany jest napis `Witaj swiecie!`. Tk pozwala na utworzenie rozbudowanego graficznego interfejsu użytkownika. Bardzo dobry opis Tk można znaleźć w książce „Learning Perl/Tk" Nancy Walsh (O'Reilly & Associates).

Inne narzędzia

Perl pozwala na użycie także innych narzędzi do obsługi graficznego interfejsu użytkownika. Te narzędzia działają jednak głównie w systemie X Window, który jest dostępny w środowiskach Unix. Poniżej przedstawiono kilka takich programów:

- Gnome (GNU Object Model Environment) autorstwa Kennetha Albanowskiego (*http://www.gnome.org*).
- Gtk — napisany przez Kennetha Albanowskiego przybornik, używany także w Gimpie.
- Sx (Simple Athena Widgets for X) autorstwa Frederica Chaveau.
- X11::Motif — napisany przez Kena Foxa przybornik Motif.

[4] Moduł Tk nie powinien być mylony z przybornikiem (ang. *toolkit*) Tk, który został utworzony przez Johna Ousterhosta, i który współpracuje z jego językiem programowania Tcl. Przybornik Tk jest niezależny od stosowanego języka i dlatego może być wykorzystany na przykład w Perlu. Moduł Perl/Tk stanowi interfejs do tego przybornika.