10

Geometrische Algorithmen

Störe meine Kreise nicht!

Archimedes (287–212 v. Chr.)

Die Geometrie braucht nicht vorgestellt zu werden. Der anschaulichste Zweig der Mathematik ist immer im Spiel, wenn Bilder auf dem Bildschirm dargestellt werden. In diesem Kapitel untersuchen wir Algorithmen für Aufgaben wir diese:

Imagemaps im Web
: Wie kann man feststellen, ob der Mauszeiger innerhalb oder außerhalb einer unregelmäßig geformten Figur liegt? Siehe den Abschnitt »Enthaltensein in einem Polygon«.

Anordnen von Fenstern
: Wie öffnet man ein neues Fenster auf dem Bildschirm, so daß es andere Fenster so wenig wie möglich überdeckt? Siehe den Abschnitt »Begrenzungen«.

Kartographie
: Ein Menge von verstreuten Punkten sei gegeben (zum Beispiel Pfosten einer Umzäunung) und der Umriß des durch sie definierten Gebietes soll gezeichnet werden. Siehe den Abschnitt »Begrenzungen«.

Simulationen
: Welche zwei von 10 000 Punkten liegen am nächsten beieinander und sind daher in Gefahr, zusammenzustoßen? Siehe den Abschnitt »Dichtestes Punktepaar«.

In diesem Kapitel geht es um geometrische Probleme und Algorithmen. Wir stellen nur die Bausteine zur Verfügung, mit denen Sie Programme aufbauen; wir können nicht Lösungen für jeden erdenklichen Fall vorsehen und voraussehen. Wir bleiben bei fast allen Beispielen in zwei Dimensionen. Wir besprechen Splines zwar in Kapitel 16, *Numerische Analysis*, überlassen aber weitergehende Themen wie Ray-Tracing, Beleuchtung, Reflexion, Animation und Darstellung von Texturen spezialisierten Büchern zur Computergraphik. Für ein tieferes Eintauchen in das Thema empfehlen wir

Computer Graphics: Principles and Practice von Foley, van Dam, Feiner und Hughes und die Bücher der Reihe *Graphics Gems*. Für praktischer veranlagte Naturen geben wir am Ende des Kapitels einige Hinweise zu verschiedenen Graphik-Toolkits, zu Geschäftsgraphiken, OpenGL (eine 3-D-Graphiksprache und VRML (Virtual Reality Markup Language).

Der Einfachheit halber erwarten fast alle Programme in diesem Kapitel Koordinaten in Form einer einfachen Liste von Zahlen. Für den Einsatz in Ihren eigenen existierenden Programmen müssen Sie diese abändern, so daß sie mit den Array- oder Hashreferenzen aus ihrer Darstellung von Punkten, Linien und Polygonen zurechtkommen. Wenn es viele Koordinaten sind, ist das auch schneller. Mehr dazu erfahren Sie im Abschnitt »Referenzen« in Kapitel 1, der *Einführung*.

Und noch eine Warnung: Viele geometrische Probleme haben Spezialfälle, die gesondert behandelt werden müssen. Zum Beispiel funktionieren viele Algorithmen nicht bei konkaven Objekten; man muß dann konvexe Einzelteile davon behandeln und diese wieder zusammensetzen. Komplizierte Objekte wie Menschen, Bäume oder die intergalaktischen Raumschiffe der Klasse F/X, die gegen die tentakelbewehrten Monster vom Orion-Gürtel ankämpfen, werden mittels Polygonen dargestellt (meist Dreiecke, oder Tetraeder bei drei Dimensionen) und Kollisionen dieser Objekte mit einer *Bounding Box* oder mit der *konvexen Hülle*. Mehr dazu folgt später in diesem Kapitel.

Distanz

Einer der fundamentalen geometrischen Begriffe ist die *Distanz* oder *Entfernung*, die Menge von Raum zwischen zwei Körpern.

Euklidische Distanz

Entfernungen kann man nach vielerlei Kriterien messen, die üblichste und wohl intuitiv am leichtesten verständliche ist die *euklidische Distanz*,[1] die Länge der geraden Strecke zwischen zwei Punkten, der Luftlinie. Mathematisch gesprochen bilden wir die Differenz der Koordinaten auf beiden Achsen, quadrieren und ziehen die Wurzel aus der Summe. In zwei Dimensionen entspricht dies dem altbekannten *Satz von Pythagoras*:[2] $d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$. Abbildung 10-1 illustriert die euklidische Distanz in verschiedenen Dimensionen. Dabei sind die letzten zwei Fälle notwendigerweise Projektionen auf die Ebene, im letzten Fall eine zweimalige Projektion.

Wir berechnen die euklidische Distanz für alle Dimensionen mit dieser Subroutine:

```
# distance( @p )
#    Berechnet aus 2*d Koordinaten die euklidische Distanz
#    zwischen zwei Punkten im d-dimensionalen Raum.
#    Zwei 3-D-Punkte werden so angegeben: ( $x0, $y0, $z0, $x1, $y1, $z1 ).
```

1 Euklid, um 300 v. Chr.

2 Pythagoras, ca. 570–490 v. Chr.

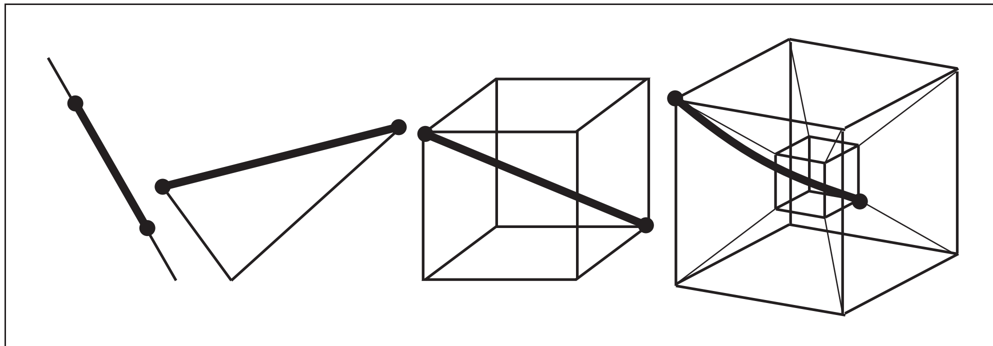

Abbildung 10-1: Euklidische Distanz in 1, 2, 3 und 4 Dimensionen

```
sub distance {
    my @p = @_;                         # Die Koordinaten der Punkte.
    my $d = @p / 2;                     # Anzahl Dimensionen.

    # Der Fall für zwei Dimensionen ist optimiert.
    return sqrt( ($_[0] - $_[2])**2 + ($_[1] - $_[3])**2 )
        if $d == 2;

    my $S = 0;                          # Die Summe von zwei Quadraten.
    my @p0 = splice @p, 0, $d;          # Anfangspunkt.

    for ( my $i = 0; $i < $d; $i++ ) {
        my $di = $p0[ $i ] - $p[ $i ];  # Differenz ...
        $S += $di * $di;                #  ... quadriert und summiert.
    }

    return sqrt( $S );
}
```

Die euklidische Distanz zwischen den Punkten (3, 4) und (10,12) ist also:

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

Manhattan-Distanz

Für manche Situationen sind Entfernungen anderer Art erforderlich. Abbildung 10-2 illustriert die *Manhattan-Distanz*. Der Name bezieht sich auf die meist rechtwinklige Anordnung der Straßen in Manhattan. Gute New Yorker Taxifahrer denken in Manhattan-Distanzen; Helikopterpiloten eher in euklidischen.

Zur Berechnung werden die Absolutwerte der Differenzen der Koordinaten auf allen Achsen summiert:

```
# manhattan_distance( @p )
#    Berechnet die Manhattan-Distanz zwischen zwei Punkten in d Dimensionen.
#    Zwei 3-D-Punkte werden so angegeben: ( $x0, $y0, $z0, $x1, $y1, $z1 ).
```

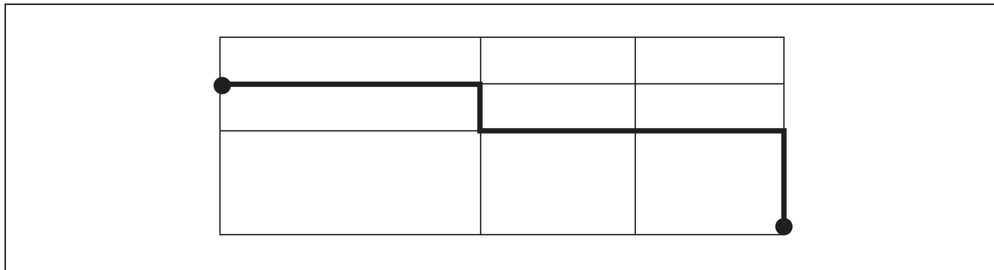

Abbildung 10-2: Manhattan-Distanz

```
sub manhattan_distance {
    my @p = @_;                 # Koordinaten der Punkte.
    my $d = @p / 2;             # Anzahl der Dimensionen.

    my $S = 0;                  # Summe.
    my @p0 = splice @p, 0, $d;  # Startpunkt aus der Liste holen.

    for ( my $i = 0; $i < $d; $i++ ) {
        my $di = $p0[ $i ] - $p[ $i ];  # Differenz ...
        $S += abs $di;                  #  ... Summe der Absolutwerte.
    }

    return $S;
}
```

Beispielsweise ist die Manhattan-Distanz zwischen (3, 4) und (10, 12):

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

Maximaldistanz

Manchmal wird als »Entfernung« einfach die größte Koordinatendifferenz benutzt: $d = \max d_i$, wobei d_i die i-te Koordinatendifferenz ist.

Man kann sich die Manhattan-Distanz als eine Approximation ersten Grades denken und die euklidische als eine zweiten Grades. Der Grenzwert dieser Approximation wäre dann die Maximaldistanz:

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

Mit anderen Worten: Bei wachsendem k dominiert die größte Koordinatendifferenz immer mehr; bei ∞ vollständig.

Sphärische Distanz

Die Entfernung zweier Punkte auf einer Kugeloberfläche heißt auch *Großkreis-Distanz*. Die Formel dafür ist eine gute Übung in sphärischer Trigonometrie; der ungeduldige Programmierer kann sich aber die Zeit sparen und – sofern er Perl 5.005_03 oder neuer benutzt – die Funktion `great_circle_distance()` aus dem Modul Math::Trig benutzen. Ältere Versionen hatten zwar das Modul, aber nicht die `great_circle_distance()`-Funktion. So berechnet man die Distanz zwischen London (51,3° N, 0,5° W) und Tokio (35,7° N, 139,8° O) in Kilometern:

```
#!/usr/bin/perl

use Math::Trig qw(great_circle_distance deg2rad);

# Achtung: Am Nordpol ist Phi = 0, daher 90 Grad minus die Breite.
@london = (deg2rad(-  0.5), deg2rad(90 - 51.3));
@tokio  = (deg2rad( 139.8), deg2rad(90 - 35.7));

# 6378 ist der Radius der Erde in km am Äquator.
print great_circle_distance(@london, @tokio, 6378);
```

Das Resultat

```
9605.26637021388
```

ist nicht beliebig genau, weil die Erde keine perfekte Kugel ist und weil 0,1° in diesen Breitengraden etwa 8 km ausmachen.

Wir subtrahieren die geographische Breite von 90°, weil bei `great_circle_distance()` *azimutale sphärische Koordinaten* benutzt werden: $\phi = 0$ ist oben am Nordpol, während bei geographischen Koordinaten der Äquator bei 0° liegt (siehe die Dokumentation von Math::Trig für Genaueres).

Fläche und Umfang

Wir kennen jetzt Distanzen, wir betreten nun Flächen und schreiten Umfänge ab.

Dreieck

Die Fläche eines Dreiecks kann auf mehrere Arten berechnet werden, je nachdem, welche Teile des Dreiecks bekannt sind. In Abbildung 10-3 zeigen wir eine der ältesten Methoden, die *Heronsche Formel*.[3]

3 Heron lebte um 65–120.

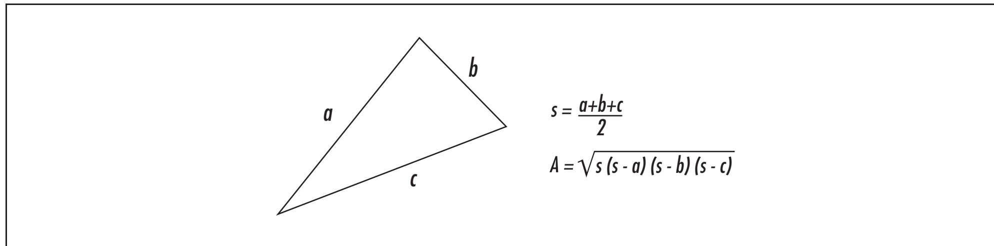

Abbildung 10-3: Dreiecksfläche nach der Heronschen Formel aus den Seitenlängen.

Unser Programm für die Heronsche Formel erwartet als Argumente entweder drei Seitenlängen oder aber die Koordinaten der drei Ecken des Dreiecks – in diesem Fall werden in `triangle_area_heron()` die Seitenlängen mit der euklidischen Distanz berechnet:

```
#!/usr/bin/perl

# triangle_area_heron( $laenge_a, $laenge_b, $laenge_c )
#    Berechnet die Fläche eines Dreiecks aus den drei Seitenlängen, oder – bei sechs
#    Parametern – aus den drei Eckpunkten, einer Liste von drei Koordinatenpaaren.
#    Gibt die Fläche des Dreiecks zurück.

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

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

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

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

Das ergibt:

```
6 2
```

Fläche eines Polygons

Die Fläche eines konvexen Polygons (eines ohne »Einbuchtungen«) kann man berechnen, indem man das Polygon in Dreiecke zerlegt und deren Fläche summiert, wie auf der linken Seite von Abbildung 10-4 gezeigt.

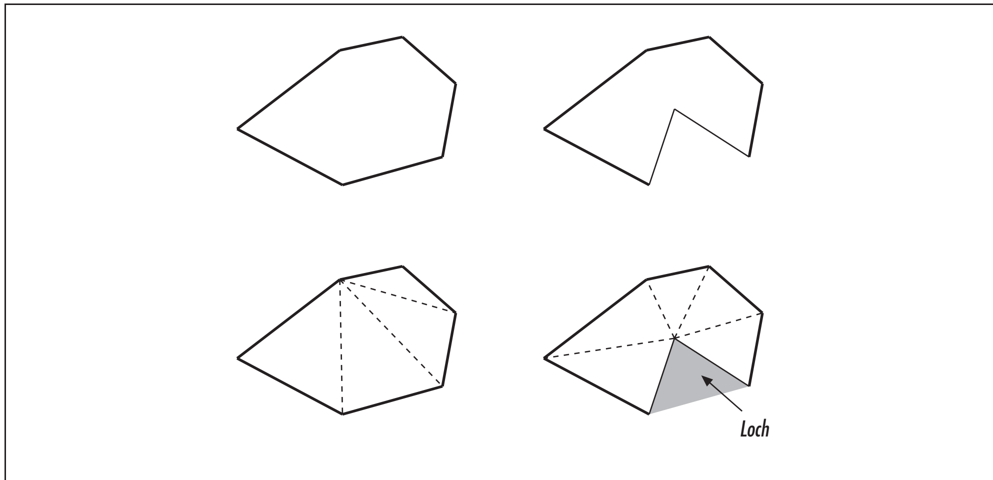

Abbildung 10-4: Konvexe (links) und konkave Polygone, in Dreiecke aufgeteilt

Bei konkaven Polygonen ist die Situation mühsamer: wir müssen die »Löcher« abziehen. Auf viel elegantere Weise erfolgt die Berechnung mit Determinanten (siehe den Abschnitt »Berechnung der Determinante« in Kapitel 7, *Matrizen*), wie Abbildung 10-5 und die folgende Formel zeigen:

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

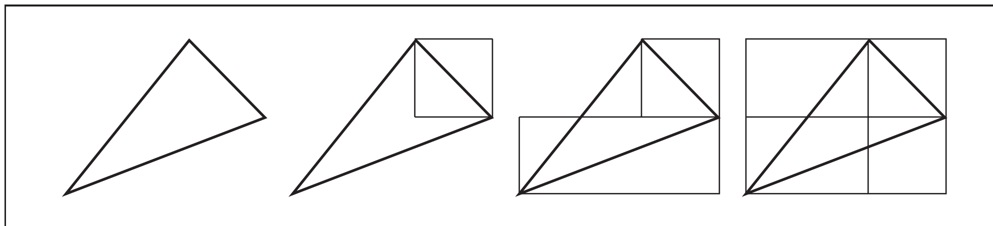

Abbildung 10-5: Aus den Determinanten ergibt sich die Fläche des Polygons.

Jede der Determinanten ergibt die Fläche eines Rechtecks, das durch zwei Eckpunkte des Polygons definiert wird. Weil die entsprechende Kante des Polygons das Rechteck diagonal halbiert, müssen wir jeweils die Hälfte der Fläche nehmen. Die Überlappung von Rechtecken (unten links in Abbildung 10-5) kann ignoriert werden, weil sich deren Beiträge aufheben.

Man beachte, wie die Formel einen Ring vom letzten Punkt (x_{n-1}, y_{n-1}) zurück zum ersten (x_0, y_0) schließt. Das ist nur natürlich, schließlich folgen wir allen n Kanten des Polygons, und wir brauchen dazu n Determinanten. Wir benötigen lediglich die Determinante einer 2×2-Matrix, die ganz einfach zu berechnen ist:

```
# determinant( $x0, $y0, $x1, $y1 )
#    Berechnet die Determinante aus den vier Elementen einer Matrix.
#
sub determinant { $_[0] * $_[3] - $_[1] * $_[2] }
```

Mit der Determinante können wir die Fläche eines Polygons berechnen:

```
# polygon_area( @xy )
#    Berechnet die Fläche (area) eines Polygons mittels Determinanten.
#    Die Eckpunkte werden als Liste ( $x0, $y0, $x1, $y1, $x2, $y2, ...) übergeben.
#
sub polygon_area {
    my @xy = @_;

    my $A = 0;                              # Die Fläche.

    # Statt am Ende schließen wir den Kreis gleich zu Beginn.
    # [-2, -1] ist das letzte Koordinatenpaar.
    for ( my ( $xa, $ya ) = @xy[ -2, -1 ];
          my ( $xb, $yb ) = splice @xy, 0, 2;
          ( $xa, $ya ) = ( $xb, $yb ) ) { # Zum nächsten Punkt.
        $A += determinant( $xa, $ya, $xb, $yb );
    }

    # Falls die Punkte im Uhrzeigersinn angegeben wurden, wird die Fläche $A
    # hier negativ, also nehmen wir den Absolutbetrag.

    return abs $A / 2;
}
```

Zum Beispiel kann man damit die Fläche des Fünfecks berechnen, das durch die fünf Punkte (0, 1), (1, 0), (3, 2), (2, 3) und (0, 2) gegeben ist:

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

Das Resultat:

```
5
```

Die angegebenen Punkte *müssen* in der richtigen Reihenfolge, im Uhrzeiger- oder im Gegenuhrzeigersinn angegeben werden; im Abschnitt »Uhrzeiger- und Gegenuhrzeigersinn« wird genauer beschrieben, was damit gemeint ist. Wenn die Punkte in einer anderen Reihenfolge angegeben werden, beschreiben sie ein anderes Polygon:

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

Durch Verschieben des letzten Punktes in die Mitte der Liste bekommen wir ein anderes Resultat:

```
1
```

Umfang eines Polygons

Mit der gleichen Schleife über alle Punkte kann statt der Fläche auch der Umfang berechnet werden. Man summiert einfach die Streckenlängen statt der Determinanten:

```
# polygon_perimeter( @xy )
#    Berechnet den Umfang (perimeter) eines Polygons.
#    Die Eckpunkte werden als Liste ( $x0, $y0, $x1, $y1, $x2, $y2, ...) übergeben.
#
sub polygon_perimeter {
    my @xy = @_;

    my $P = 0;                                  # Länge des Umfangs.

    # Statt am Ende schließen wir den Kreis gleich zu Beginn.
    # [-2, -1] ist das letzte Koordinatenpaar.
    for ( my ( $xa, $ya ) = @xy[ -2, -1 ];
          my ( $xb, $yb ) = splice @xy, 0, 2;
          ( $xa, $ya ) = ( $xb, $yb ) ) { # Zum nächsten Punkt.
        $P += distance( $xa, $ya, $xb, $yb );
    }

    return $P;
}
```

Als Beispiel berechnen wir den Umfang des Fünfecks aus dem letzten Abschnitt:

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

Das Resultat ist:

```
8.89292222699217
```

Uhrzeiger- und Gegenuhrzeigersinn

Manchmal müssen wir wissen, welche Dinge rechts von uns (im *Uhrzeigersinn*) oder links von uns (im *Gegenuhrzeigersinn*) liegen, zum Beispiel, um herauszufinden, ob ein bestimmter Punkt innerhalb eines Dreiecks liegt oder nicht. Wir behandeln hier das Problem nur in zwei Dimensionen; in drei ist die Bedeutung von »rechts« und »links« nur gegeben, wenn zuvor definiert wird, was oben und unten ist.

Zu gegebenen drei Punkten kann man immer angeben, ob sie einen Weg im Uhrzeigersinn oder einen im Gegenuhrzeigersinn beschreiben oder keines von beiden tun: Dann liegen sie auf einer Geraden. In Abbildung 10-6 beschreiben die Punkte (1, 1), (4, 3) und (4, 4) einen Weg im Gegenuhrzeigersinn, der Weg dreht nach links ab. Der Weg durch die Punkte (1, 1), (4, 3) und (7, 4) weist nach rechts, im Uhrzeigersinn.

Die Routine `clockwise()` erwartet drei Punkte als Parameter und gibt eine Zahl zurück. Das Resultat ist positiv, falls die drei Punkte einen Weg im Uhrzeigersinn beschreiben, negativ beim Gegenuhrzeigersinn und eine Zahl sehr nahe bei 0, falls die Punkte auf einer Geraden liegen.

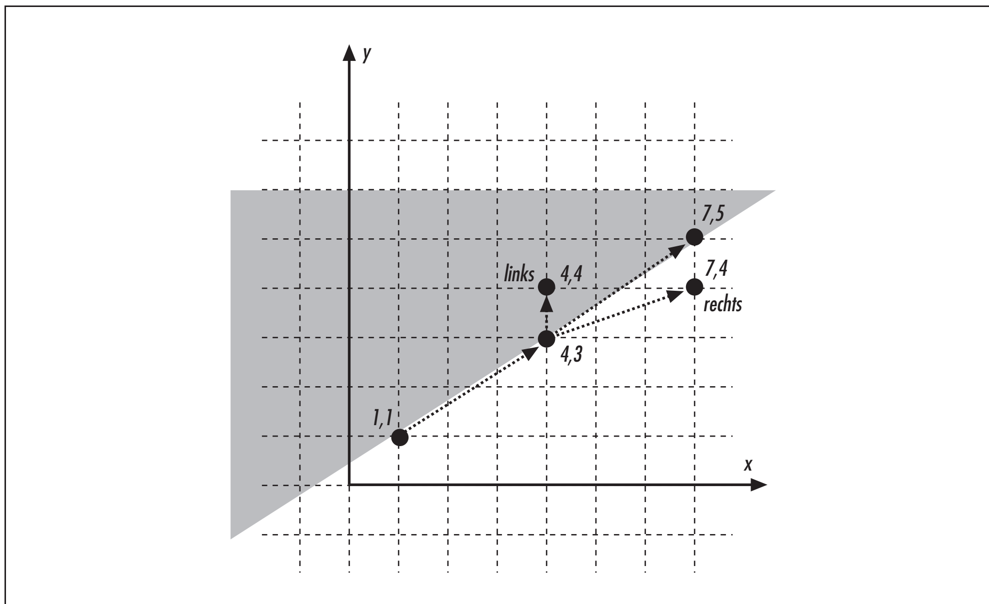

Abbildung 10-6: Uhrzeiger- und Gegenuhrzeigersinn, links und rechts

```
# clockwise( $x0, $y0, $x1, $y1, $x2, $y2 )
#     Gibt eine positive Zahl zurück, wenn der Weg von p0 (x0, y0) via p1 nach p3 nach
#     rechts weist (im Uhrzeigersinn), und eine negative Zahl, wenn der Weg nach links
#     (im Gegenuhrzeigersinn) verläuft. Gibt Null zurück, wenn die drei Punkte auf
#     einer Geraden liegen – aber Vorsicht vor Rundungsfehlern!
#
sub clockwise {
    my ( $x0, $y0, $x1, $y1, $x2, $y2 ) = @_;
    return ( $x2 - $x0 ) * ( $y1 - $y0 ) - ( $x1 - $x0 ) * ( $y2 - $y0 );
}
```

Zum Beispiel ergeben die Punkte aus Abbildung 10-6:

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

die Ausgabe:

```
-3
0
3
```

In Worten ausgedrückt: Der Punkt (4, 4) ist links (negativ) des Vektors von (1, 1) nach (4, 3), der Punkt (7, 5) ist genau auf diesem Vektor, und der Punkt (7, 4) liegt rechts (positiv) davon.

Die Routine `clockwise()` ist so etwas wie eine plattgedrückte Version des *Kreuzproduktes* oder *Vektorproduktes* aus der Vektoralgebra. Das Kreuzprodukt ist ein Objekt im dreidimensionalen Raum, ein Vektor, der senkrecht auf der durch die Vektoren $p_0 - p_1$ und $p_1 - p_2$ gebildeten Ebene steht.

Schnittprobleme

In diesem Abschnitt werden wir bei Fließkomma-Berechnungen öfters `epsilon` benutzen. ε steht für eine sehr kleine Zahl, für »Null« innerhalb der Rechengenauigkeit. Wie groß ε sein soll, müssen Sie entscheiden, wir empfehlen ein Zehn-Milliardstel:

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

oder mit der etwas schnelleren Version:

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

Mehr dazu erfahren Sie im Abschnitt »Rechengenauigkeit« im Kapitel 11, *Zahlensysteme*.

Schnitt von Geraden

Es gibt zwei Arten von Schnittproblemen bei Geraden. Im allgemeinen Fall können die Geraden jede mögliche Steigung haben, in der eingeschränkteren Version geht es nur um horizontale und vertikale Geraden, um die »*Manhattan-Kreuzungen*«.

Schnittpunkt von Geraden: Allgemeiner Fall

Das Bestimmen des Schnittpunktes zweier Geraden ist recht einfach. Man löst das Gleichungssystem der zwei Gleichungen $y_0 = b_0x+a_0$ und $y_1 = b_1x+a_1$, die die Geraden beschreiben. Die Methoden dazu sind im Abschnitt »Gaußsches Eliminationsverfahren« in Kapitel 7, *Matrizen*, bzw. im Abschnitt »Gleichungen lösen« in Kapitel 16, *Numerische Analysis*, beschrieben. Aber diese allgemeinen Verfahren funktionieren nicht immer. Wenn wir »Division-durch-Null«-Fehler vermeiden wollen, müssen wir die Spezialfälle gesondert behandeln, bei denen eine der Garden horizontal oder vertikal ist oder bei denen die Geraden parallel sind. Abbildung 10-7 illustriert einige Schnittprobleme.

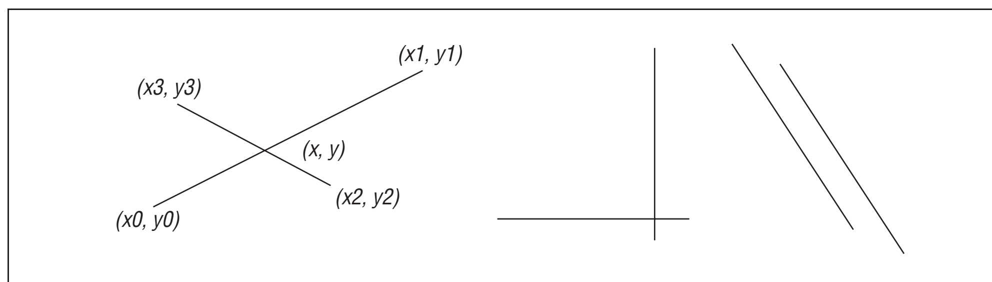

Abbildung 10-7: Schnitt von Geraden: Allgemein; horizontale, vertikale und parallele Fälle

Mit all diesen Sonderfällen ist die Bestimmung des Schnittpunktes von zwei Geraden gar nicht mehr so simpel, wie es scheint. Unsere Implementation ist erstaunlich lang:

```
# line_intersection( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 )
#
#    Berechnet den Schnittpunkt zweier Geraden, die definiert sind durch:
#      - Acht Werte: Zwei Strecken, gebildet aus den vier Punkten
#        (x0, y0) – (x1, y1) und (x2, y2) – (x3, y3).
#      - Vier Werte: Die Steigungen und y-Achsenabschnitte (a0, b0, a1 b1)
#        der zwei Geraden in der Parameterdarstellung y = ax + b.
#
#    Rückgabewert:
#    Entweder ein Zahlentripel ($x, $y, $s) für den Schnittpunkt, wobei $x und
#    $y die Koordinaten des Schnittpunkts sind und $s besagt, ob der Schnittpunkt
#    auf den Strecken selbst liegt (1) oder nur in deren Verlängerung (0).
#
#    Oder ein String, der eine Ausnahmesituation beschreibt:
#       "out of bounding box"    - Außerhalb der Bounding Box
#       "parallel"
#       "parallel collinear"     - Die zwei Geraden sind auf einer Linie
#       "parallel horizontal"
#       "parallel vertical"
#    Wegen des Tests auf Enthaltensein in der Bounding Box können die Fälle
#    »parallel horizontal« und »parallel vertical« gar nicht auftreten.
#    (Bounding Boxes werden etwas später behandelt.)
#
sub line_intersection {
 use constant epsilon => 1e-14;
 my ( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 );

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

     # Die Bounding Boxes teilen die Geraden in Segmente auf.
     # bounding_box() wird später definiert.
     my @box_a = bounding_box( 2, $x0, $y0, $x1, $y1 );
     my @box_b = bounding_box( 2, $x2, $y2, $x3, $y3 );

     # Ohne die folgenden zwei Zeilen überprüft die Routine nur, ob sich die zwei
     # Geraden schneiden; evtl. schneiden sich nur die Verlängerungen der Strecken.
     # bounding_box_intersect() wird später definiert.
     return "out of bounding box"
         unless bounding_box_intersect( 2, @box_a, @box_b );
 } elsif ( @_ == 4 ) { # Parameterdarstellung der Geraden.
     $x0 = $x2 = 0;
     ( $y0, $y2 ) = @_[ 1, 3 ];
     # $enough muß genügend groß sein, damit die Punkte in der x-Richtung weit
     # auseinanderliegen – wegen Rechenungenauigkeiten.
     my $abs_y0 = abs $y0;
     my $abs_y2 = abs $y2;
     my $enough = 10 * ( $abs_y0 > $abs_y2 ? $abs_y0 : $abs_y2 );
```

```
    $x1 = $x3 = $enough;
    $y1 = $_[0] * $x1 + $y0;
    $y3 = $_[2] * $x3 + $y2;
}

my ($x, $y);                       # Der noch unbestimmte Schnittpunkt.

my $dy10 = $y1 - $y0;              # $dyPQ und $dxPQ sind die
my $dx10 = $x1 - $x0;              # Koordinatendifferenzen zwischen
my $dy32 = $y3 - $y2;              # den Punkten P und Q.
my $dx32 = $x3 - $x2;

my $dy10z = abs( $dy10 ) < epsilon; # Ist die Differenz $dy10 gleich »Null«?
my $dx10z = abs( $dx10 ) < epsilon;
my $dy32z = abs( $dy32 ) < epsilon;
my $dx32z = abs( $dx32 ) < epsilon;

my $dyx10;                         # Die Steigungen.
my $dyx32;

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

# Wir haben alle Differenzen und Steigungen; jetzt können die horizontalen und
# vertikalen Spezialfälle isoliert werden (z. B. Steigung = 0 ist eine horizontale Linie).

unless ( defined $dyx10 or defined $dyx32 ) {
    return "parallel vertical";
} elsif ( $dy10z and not $dy32z ) { # Erste Gerade horizontal.
    $y = $y0;
    $x = $x2 + ( $y - $y2 ) * $dx32 / $dy32;
} elsif ( not $dy10z and $dy32z ) { # Zweite Gerade horizontal.
    $y = $y2;
    $x = $x0 + ( $y - $y0 ) * $dx10 / $dy10;
} elsif ( $dx10z and not $dx32z ) { # Erste Gerade vertikal.
    $x = $x0;
    $y = $y2 + $dyx32 * ( $x - $x2 );
} elsif ( not $dx10z and $dx32z ) { # Zweite Gerade vertikal.
    $x = $x2;
    $y = $y0 + $dyx10 * ( $x - $x0 );
} elsif ( abs( $dyx10 - $dyx32 ) < epsilon ) {
    # Die Steigungen sind fast identisch: parallele Geraden, eventuell kollineare.

    # Durch den »Bounding-Box«-Test wurden die Fälle »parallel horizontal«
    # und »parallel vertical« bereits ausgeschlossen.

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

    return "parallel collinear" if abs( $ya - $yb ) < epsilon;
    return "parallel";
```

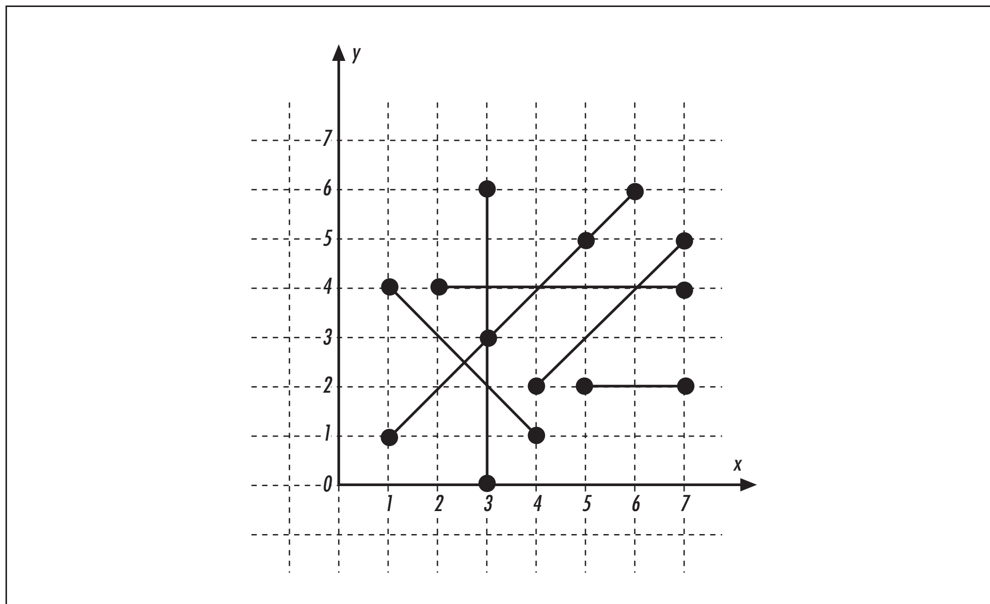

Abbildung 10-8: Schnittpunkt von Geraden: Beispiele

```
    } else {
        # Kein Spezialfall: ein normaler, ehrlicher Schnittpunkt.

        $x = ($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);
}
```

Abbildung 10-8 illustriert mit einer Anzahl von Strecken die verschiedenen Fälle, wie Geraden sich schneiden können (oder eben nicht).

Wir untersuchen sechs mögliche Schnittpunkte mit `line_intersection()`:

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

Das Resultat:

```
2.5 2.5 1
4 4 1
3 3 1
2 2
parallel
parallel collinear
```

Wenn wir nur daran interessiert sind, *ob* sich zwei Strecken schneiden, ist die Berechnung der Koordinaten des Schnittpunktes zu aufwendig. Es genügt dafür, die Vorzeichen der Beträge der Kreuzprodukte $(p_2 - p_0)\times(p_1 - p_0)$ und $(p_3 - p_0)\times(p_1 - p_0)$ zu vergleichen. Die Routine `line_intersect()` gibt einen einfachen Booleschen Wert zurück, je nachdem, ob sich die Strecken innerhalb der Bounding Box schneiden:

```
# line_intersect( $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 )
#    Gibt wahr zurück, wenn sich die durch die vier Punkte definierten Strecken schneiden.
#    Benutzt epsilon zur Abschätzung der Grenzfälle.

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

    my @box_a = bounding_box( 2, $x0, $y0, $x1, $y1 );
    my @box_b = bounding_box( 2, $x2, $y2, $x3, $y3 );

    # Wenn sich nicht einmal die Bounding Boxes überschneiden, sofort aufgeben.

    return 0 unless bounding_box_intersect( 2, @box_a, @box_b );

    # Wenn sich die Vorzeichen der zwei Determinanten (die Absolutbeträge der
    # Kreuzprodukte) unterscheiden, schneiden sich die Strecken.

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

    my $det_a = determinant( $x2 - $x0, $y2 - $y0, $dx10, $dy10 );
    my $det_b = determinant( $x3 - $x0, $y3 - $y0, $dx10, $dy10 );

    return 1 if -$det_a > epsilon and  $det_b > epsilon or
                 $det_a > epsilon and -$det_b > epsilon;

    if ( abs( $det_a ) < epsilon ) {
        if ( abs( $det_b ) < epsilon ) {
            # Beide Kreuzprodukte »Null«.
            return 1;
        } elsif ( abs( $x3 - $x2 ) < epsilon and
                  abs( $y3 - $y2 ) < epsilon ) {
            # Das erste Kreuzprodukt ist »Null«, und der zweite Vektor
            # (von (x2,y2) nach (x3,y3)) hat die Länge »Null«.
            return 1;
        }
```

```
        } elsif ( abs( $det_b < epsilon ) ) {
            # Das zweite Kreuzprodukt und der erste Vektor sind »Null«.
            return 1 if abs( $dx10 ) < epsilon and abs( $dy10 ) < epsilon;
        }

        return 0; # Voreinstellung: Keine Überschneidung.
    }
```

Wir testen `line_intersect()` mit zwei Paaren von Strecken. Das erste Paar kreuzt bei (3, 4), das zweite überhaupt nicht, weil die Strecken parallel sind:

```
print "Schnittpunkt\n"
    if     line_intersect( 3, 0,  3, 6,  1, 1,  6, 6 );
print "Kein Schnittpunkt\n"
    unless line_intersect( 1, 1,  6, 6,  4, 2,  7, 5 );
Schnittpunkt
Kein Schnittpunkt
```

Schnittpunkt von Geraden: Nur horizontale und vertikale Geraden

Oft ist der allgemeine Fall des Schnittpunkts von Geraden *zu* allgemein: Bei der »Manhattan«-Geometrie, bei der nur senkrechte und waagrechte Linien vorkommen, können ganz andere Algorithmen verwendet werden.

Die Lösung besteht darin, *binäre Bäume* zu verwenden, wie wir sie in Kapitel 3, *Komplexe Datenstrukturen*, eingeführt haben. Wir schieben dabei eine gedachte horizontale Linie von unten nach oben über die Zeichenebene und bauen bei diesem Vorgang unseren binären Baum auf. Der entstandene binäre Baum enthält die vertikalen Strecken, nach ihrer x-Koordinate sortiert, deshalb heißt er auch x-Baum. Der x-Baum wird nach folgendem Schema aufgebaut:

- Die Punkte werden von unten nach oben durchgegangen, zuerst die an senkrechten Strecken beteiligten und danach die an horizontalen Strecken beteiligten von links nach rechts. Das bedeutet, daß die beiden Endpunkte einer horizontalen Strecke gleichzeitig gesehen werden, die Endpunkte von vertikalen Linien nacheinander.
- Wenn der untere Endpunkt einer vertikalen Linie erkannt wird, wird dieser Knoten in den binären Baum aufgenommen, mit seiner x-Koordinate als Wert. Dadurch werden die Punkte im Baum von links nach rechts aufgeteilt; wenn die Strecke a links von der Strecke b liegt, dann liegt auch der Knoten für den Anfangspunkt von a im Baum links des entsprechenden Punktes von b.
- Wenn der obere Endpunkt einer senkrechten Linie passiert wird, wird der Knoten für die entsprechende Linie aus dem Baum entfernt.
- Wenn der Algorithmus eine horizontale Linie antrifft, werden die Knoten im Baum (die aktiven senkrechten Linien) daraufhin überprüft, ob sie diese waagerechte Linie schneiden. Die horizontalen Strecken werden nicht in den Baum eingefügt, sie werden nur dafür benutzt, die Kreuzungstests zu starten.

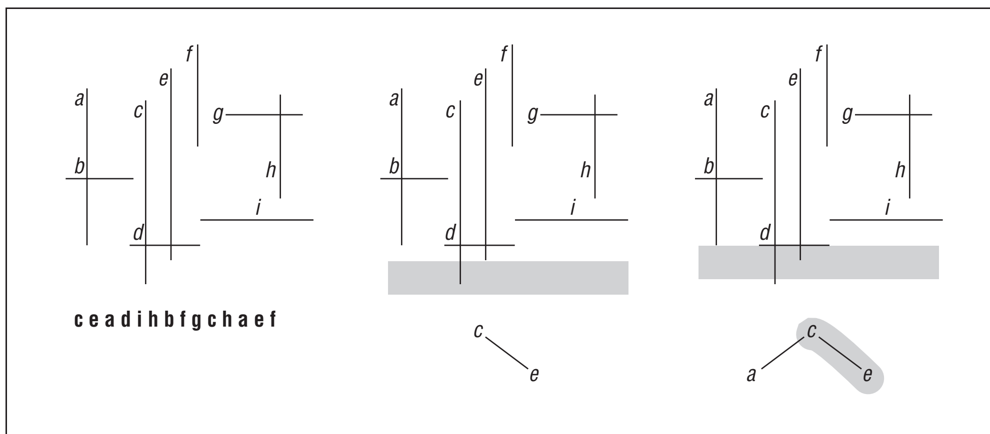

Abbildung 10-9: Schnittpunkte bei horizontalen und vertikalen Linien

Abbildung 10-9 zeigt, wie der *x*-Baum aufgebaut wird, während sich die gedachte (grau dargestellte) Grenze von unten nach oben schiebt. Im linken Bild wird nur dargestellt, in welcher Reihenfolge die Liniensegmente behandelt werden: Zuerst *c*, dann *e* usw. Das mittlere Teilbild zeigt, wie der *x*-Baum aussieht, nachdem *e* gefunden wurde; das Teilbild rechts stellt die Situation dar, nachdem *a* hinzugefügt wurde. Die Strecke *d* wird nicht in den Baum eingefügt. Wenn sie gefunden wird, wird der Baum nach möglichen Kreuzungspunkten abgesucht.

Der Algorithmus ist in `manhattan_intersection()` implementiert:

```
# manhattan_intersection( @lines )
#    Schnittpunkte von horizontalen und vertikalen Strecken finden.
#    Benötigt die Routinen basic_tree_add(), basic_tree_del() und
#    basic_tree_find() aus Kapitel 3, Komplexe Datenstrukturen.
#
sub manhattan_intersection {
    my @op; # Die Koordinaten werden hier in Operationen verwandelt.

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

        if ($line[1] == $line[3]) {      # Horizontal.
            push @op, [ @line, \&range_check_tree, 2 ];
        } else {                         # Vertikal.
            # Punkte vertauschen, wenn die Strecke von oben nach unten verläuft.
            @line = @line[0, 3, 2, 1] if $line[1] > $line[3];

            push @op, [ @line[0, 1, 2, 1], \&basic_tree_add, 1 ];
            push @op, [ @line[0, 3, 2, 3], \&basic_tree_del, 3 ];
        }
    }
```

```
    my $x_tree; # x-Baum.
    # Vergleichsfunktion: Sortiert nach x-Koordinaten.
    my $compare_x = sub { $_[0]->[0] <=> $_[1]->[0] };
    my @intersect; # Schnittpunkte.

    foreach my $op (sort { $a->[1] <=> $b->[1] ||
                           $a->[5] <=> $b->[5] ||
                           $a->[0] <=> $b->[0] }
                           @op) {
        if ($op->[4] == \&range_check_tree) {
            push @intersect, $op->[4]->( \$x_tree, $op, $compare_x );
        } else { # Zufügen oder löschen.
            $op->[4]->( \$x_tree, $op, $compare_x );
        }
    }

    return @intersect;
}

# range_check_tree( $tree_link, $horizontal, $compare )
#    Gibt eine Liste der Knoten zurück, deren x-Koordinaten zwischen
#    $horizontal->[0] und $horizontal->[2] liegen. Verwendet binäre Bäume
#    wie die Routinen aus Kapitel 3, Komplexe Datenstrukturen.
#
sub range_check_tree {
    my ( $tree, $horizontal, $compare ) = @_;

    my @range          = ( );   # Der Rückgabewert.
    my $node           = $$tree;
    my $vertical_x     = $node->{val};
    my $horizontal_lo  = [ $horizontal->[ 0 ] ];
    my $horizontal_hi  = [ $horizontal->[ 2 ] ];

    return unless defined $$tree;

    push @range, range_check_tree( \$node->{left}, $horizontal, $compare )
        if defined $node->{left};

    push @range, $vertical_x->[ 0 ], $horizontal->[ 1 ]
        if $compare->( $horizontal_lo, $vertical_x ) <= 0 &&
           $compare->( $horizontal_hi, $vertical_x ) >= 0;

    push @range, range_check_tree( \$node->{right}, $horizontal, $compare )
        if defined $node->{right};

    return @range;
}
```

`manhattan_intersection()` ist von der Komplexität $O(N \log N + k)$, wobei k die Anzahl der Schnittpunkte ist (wovon es nicht mehr als $(N/2)^2$ geben kann).

Als Beispieldaten für `manhattan_intersection()` nehmen wir die Strecken aus Abbildung 10-10.

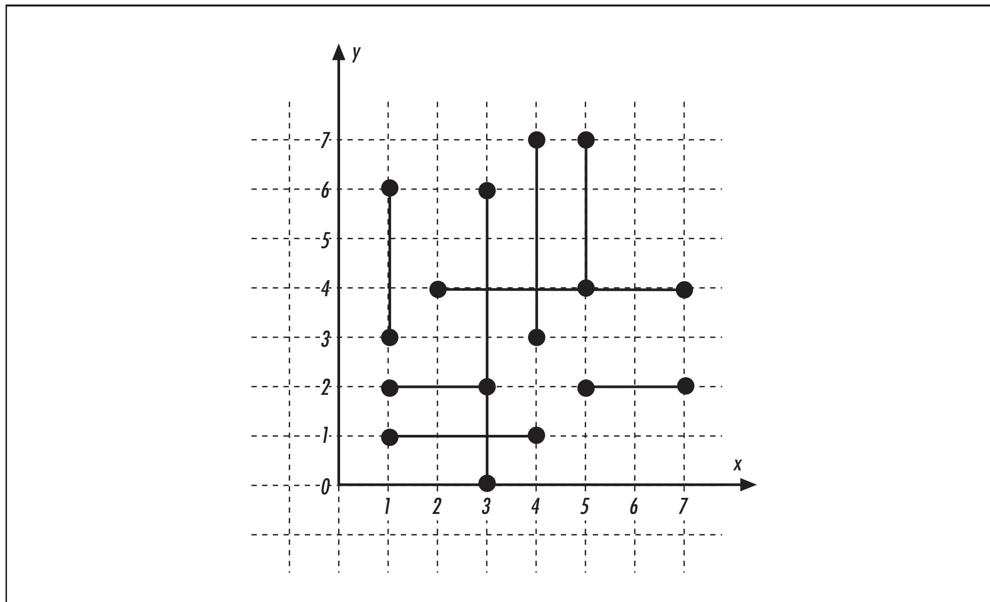

Abbildung 10-10: Strecken für das Programmbeispiel des Manhattan-Algorithmus

Die Strecken aus Abbildung 10-10 werden in einem Array abgelegt und auf Schnittpunkte überprüft:

```
@lines = ( 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(" ", manhattan_intersection(@lines)), "\n";
```

Wir bekommen:

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

Wir bekommen also sechs Schnittpunkte: (3, 1) ist der unterste links, und (5, 4) ist der oberste rechts.

Enthaltensein in einem Polygon

In diesem Abschnitt geht es darum herauszufinden, ob ein Punkt *innerhalb* eines Polygons liegt – ob er in dem Polygon enthalten ist. Darauf aufbauend können wir beispielsweise untersuchen, ob eine Strecke in einem Polygon vollständig oder nur teilweise enthalten ist.

Punkt in einem Polygon

Um herauszufinden, ob ein Punkt innerhalb eines Polygons liegt, verfolgt man einen »Strahl« von diesem Punkt aus in die Unendlichkeit (oder zu einem Punkt, von dem bekannt ist, daß er außerhalb des Polygons liegt). Der Algorithmus ist einfach: Man zählt die Schnittpunkte des Strahls mit den Kanten des Polygons. Wenn diese Anzahl ungerade ist (bei den Punkten *e*, *f*, *h* und *j* in Abbildung 10-11), liegt der Punkt innerhalb des Polygons, sonst außerhalb (*a*, *b*, *c*, *d*, *g* und *i*). Es gibt ein paar verzwickte Sonderfälle (bei geometrischen Algorithmen geht es kaum je ohne Spezialfälle ab): was passiert, wenn der Strahl auf eine Ecke des Polygons fällt (Punkte *d*, *f*, *g* und *j*)? Oder, noch schlimmer, auf eine Kante (Punkt *j*)? Der hier vorgestellte Algorithmus funktioniert sicher für die Punkte, die klar innerhalb oder außerhalb des Polygons liegen. Bei den Grenzfällen hängt es davon ab, wie die »Fast-Zusammenstöße« gezählt werden.

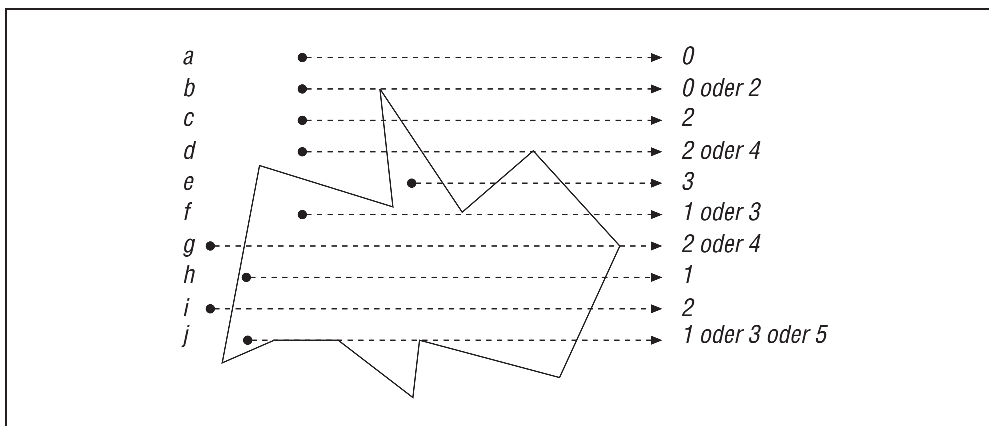

Abbildung 10-11: Liegt der Punkt im Polygon? Zählen Sie die Schnittpunkte!

Die Routine `point_in_polygon()` gibt einen wahren Wert zurück, wenn der Punkt (durch die ersten zwei Argumente definiert) im Innern des Polygons liegt, das durch den Rest der Argumente gegeben ist.

```
# point_in_polygon ( $x, $y, @xy )
#
#    Gegeben: Punkt ($x, $y), Polygon ($x0, $y0, $x1, $y1, ... ) als Array @xy.
#    Rückgabewert: 1 für Punkte, die sicher innerhalb, 0 für Punkte, die sicher außerhalb
#    des Polygons liegen. Für die Grenzfälle wird die Situation komplex und sprengt
#    den Rahmen dieses Buches. Die Grenzfälle werden immerhin in einer Hinsicht
#    korrekt behandelt: Wenn die Ebene in mehrere Polygone aufgeteilt wird, gehört jeder
#    Punkt zu genau einem Polygon.
#
#    Abgeleitet von einem Algorithmus aus der FAQ zu comp.graphics.algorithms.
#    Mit freundlicher Genehmigung von Wm. Randolph Franklin.
#
```

```
sub point_in_polygon {
    my ( $x, $y, @xy ) = @_;

    my $n = @xy / 2;                        # Anzahl der Ecken des Polygons.
    my @i = map { 2 * $_ } 0 .. (@xy/2);    # Die geraden Indizes von @xy .
    my @x = map { $xy[ $_ ]     } @i;       # Gerade Indizes: x-Koordinaten.
    my @y = map { $xy[ $_ + 1 ] } @i;       # Ungerade Indizes: y-Koordinaten.

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

    my $side = 0;                           # 0 = außerhalb, 1 = innerhalb.

    for ( $i = 0, $j = $n - 1 ; $i < $n; $j = $i++ ) {
        if (
            (
             # Wenn y innerhalb der y-Grenzen liegt ...
             ( ( $y[ $i ] <= $y ) && ( $y < $y[ $j ] ) ) ||
             ( ( $y[ $j ] <= $y ) && ( $y < $y[ $i ] ) )
            )
            and
            #  ... und der Strahl von (x, y) in die Unendlichkeit die Kante vom
            # i-ten zum j-ten Punkt kreuzt ...
            ($x
             <
             ( $x[ $j ] - $x[ $i ] ) *
             ( $y - $y[ $i ] ) / ( $y[ $j ] - $y[ $i ] ) + $x[ $i ] )) {
          $side = not $side; #  ... dann die Seite wechseln.
      }
    }

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

Um herauszufinden, ob die Anzahl der Schnittpunkte gerade oder ungerade ist, müssen wir sie gar nicht zählen; es genügt, wenn wir bei jedem Schnittpunkt den Booleschen Wert `$side` ins Gegenteil verkehren.

Mit dem Polygon aus Abbildung 10-12 testen wir, ob die neun Punkte innerhalb oder außerhalb des Polygons liegen:

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

Das Resultat:

```
( 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
```

Also sind die Punkte (3, 4), (3, 1) und (7, 1) im Polygon enthalten, die anderen nicht.

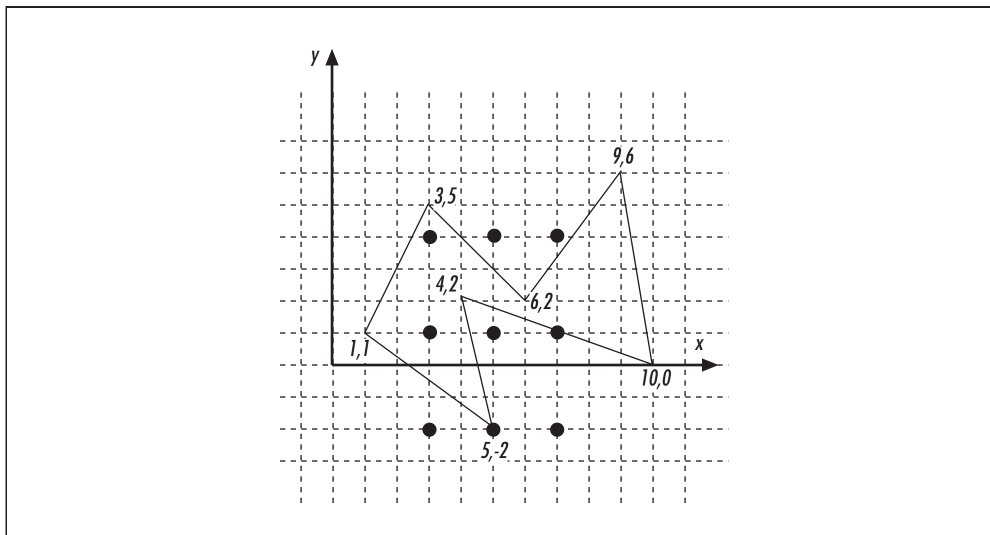

Abbildung 10-12: Das Polygon aus dem Programmbeispiel mit Punkten inner- und außerhalb

Punkt in einem Dreieck

Für ein einfaches Polygon wie das Dreieck gibt es auch andere Algorithmen. Wir beginnen mit einer Seite des Dreiecks und bestimmen, ob der gesuchte Punkt links oder rechts davon liegt. Dann gehen wir zur nächsten Seite und wiederholen den Vorgang. Wenn der Punkt auf verschiedenen Seiten der ersten und der zweiten Dreiecksseite liegt, ist er sicher außerhalb des Dreiecks. Wenn nicht, wiederholen wir das Verfahren mit der dritten Seite; wenn der Punkt auch da auf der gleichen Seite liegt, ist er im Dreieck enthalten. Wenn wir feststellen, daß der Punkt exakt auf einer Kante liegt, zählen wir das als »innerhalb«.

In Abbildung 10-13 wird angenommen, daß wir die Seiten des Dreiecks im Gegenuhrzeigersinn durchgehen. Punkte innerhalb des Dreiecks liegen dann immer links. Wenn ein Punkt einmal auf der anderen Seite liegt, ist er außerhalb.

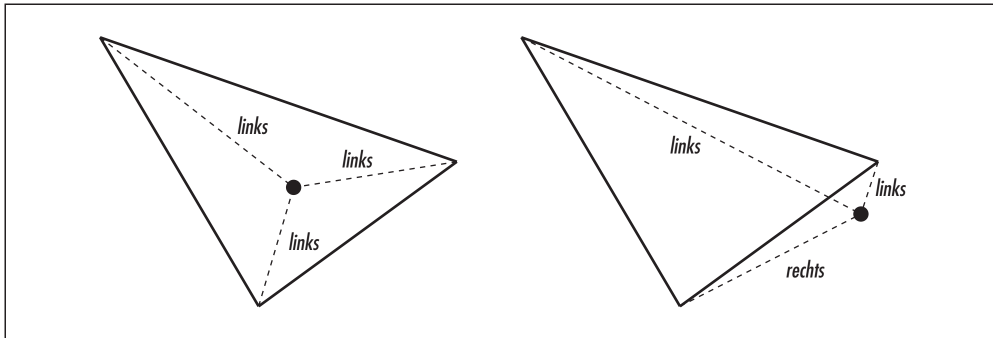

Abbildung 10-13: Punkte inner- und außerhalb eines Dreiecks

Der Algorithmus ist in `point_in_triangle()` ausgeführt:

```
# point_in_triangle( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2 )
#     Gibt wahr zurück, wenn der Punkt ($x, $y) innerhalb des durch die folgenden
#     Punkte gebildeten Dreiecks liegt.
#
sub point_in_triangle {
    my ( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2 ) = @_;

    # clockwise() wurde weiter vorn in diesem Kapitel definiert.
    my $cw0 = clockwise( $x0, $y0, $x1, $y1, $x, $y );
    my $cw1 = clockwise( $x1, $y1, $x2, $y2, $x, $y );
    my $cw2 = clockwise( $x2, $y2, $x0, $y0, $x, $y );

    # Links = -1, kollinear = 0, rechts = 1.
    $cw0 = abs( $cw0 ) < epsilon ? 0 : $cw0 < 0 ? -1 : 1;
    $cw1 = abs( $cw1 ) < epsilon ? 0 : $cw1 < 0 ? -1 : 1;
    $cw2 = abs( $cw2 ) < epsilon ? 0 : $cw2 < 0 ? -1 : 1;

    # Im allgemeinen Fall ist bei Punkten innerhalb die Summe 3 oder -3.
    # Bei Punkten auf einer Kante ist die Summe 2 oder -2.
    return 1 if abs( $cw0 + $cw1 + $cw2 ) >= 2;

    # Punkte, bei denen zwei $cwX gleich Null sind, sind die Ecken
    # des Dreiecks – alle anderen Punkte liegen außerhalb.
    return abs( $cw0 ) + abs( $cw1 ) + abs( $cw2 ) <= 1;
}
```

Wir definieren ein Dreieck mit den Ecken (1, 1), (5, 6) und (9, 3) und überprüfen sieben Punkte:

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

```
print "(5, 1): ", point_in_triangle( 5, 1,  @triangle ), "\n";
print "(5, 2): ", point_in_triangle( 5, 2,  @triangle ), "\n";
```

Wir erhalten:

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

Die Punkte (1, 2), (3, 4) und (5, 1) liegen innerhalb des Dreiecks, die anderen außerhalb.

Punkt in einem Viereck

Jedes konvexe Viereck (ein Polygon mit vier Seiten, dazu gehören Quadrat, Rechteck, Rhombus, Trapez usw.) läßt sich durch eine Linie zwischen gegenüberliegenden Ecken in zwei Dreiecke aufteilen. Damit und mit der `point_in_triangle()`-Routine können wir sehr einfach feststellen, ob ein Punkt innerhalb eines Vierecks liegt. (Vorsicht vor entarteten Fällen: Vierecke mit zusammenfallenden Ecken werden zu Dreiecken, Strecken oder sogar Punkten.) Diese Aufteilung in zwei Dreiecke ist in Abbildung 10-14 illustriert.

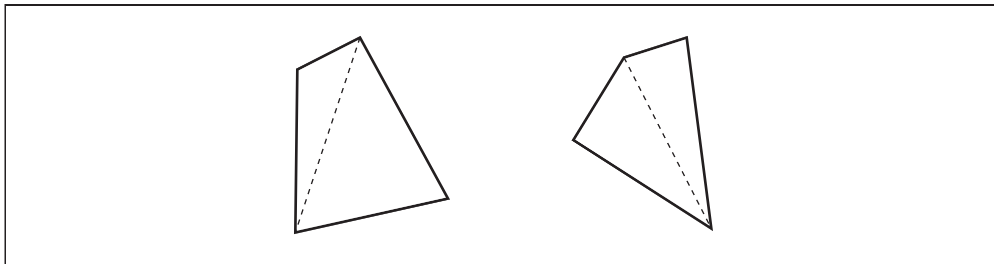

Abbildung 10-14: Aufteilung eines Vierecks in zwei Dreiecke

Die Routine `point_in_quadrangle()` ruft einfach die Routine `point_in_triangle()` zweimal auf:

```
# point_in_quadrangle( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 )
#    Gibt wahr zurück, wenn der Punkt ($x, $y) innerhalb des durch die
#    Punkte p0 ($x0, $y0), p1, p2 und p3 gebildeten Vierecks liegt.
#    Benutzt einfach point_in_triangle().
#
sub point_in_quadrangle {
    my ( $x, $y, $x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3 ) = @_;

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

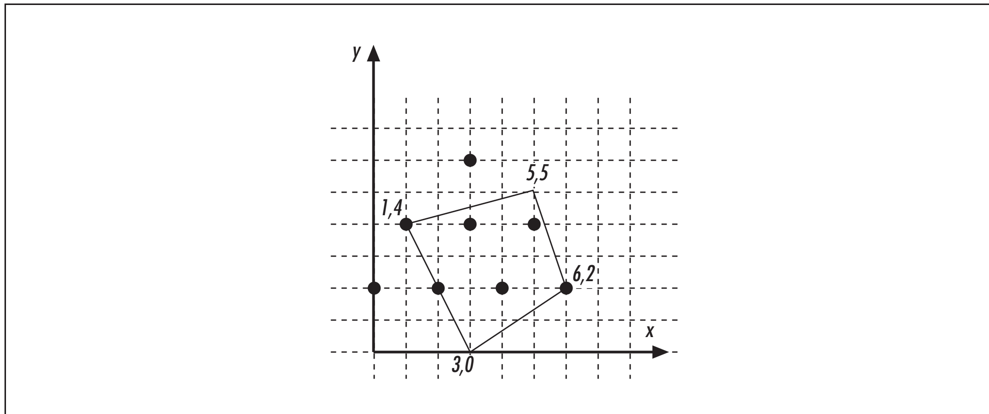

Abbildung 10-15: Welche Punkte liegen innerhalb des Vierecks?

Wir testen die Routine `point_in_quadrangle()` mit den Daten aus Abbildung 10-15. Die Ecken des Vierecks sind (1, 4), (3, 0), (6, 2) und (5, 5):

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

Die Ausgabe:

```
(0, 2): 0
(1, 4): 1
(2, 2): 1
(3, 6): 0
(3, 4): 1
(4, 2): 1
(5, 4): 1
(6, 2): 1
```

Die Punkte (3, 4), (4, 2) und (5, 4) liegen sicher innerhalb des Vierecks, die Punkte (0, 2) und (3, 6) sicher außerhalb.

Begrenzungen

In diesem Abschnitt geht es um die die Abgrenzungen von geometrischen Objekten, mit denen man ermitteln kann, ob sich zwei Objekte zu überlagern scheinen. Wir sagen »scheinen«, weil die hier behandelten Abgrenzungen nur eine erste Approximation geben: Bei konkaven Objekten wird die Sache viel komplizierter.

Bounding Box

Die *Bounding Box* eines geometrischen Objekts ist die kleinste *d*-dimensionale »Kiste«, in die das *d*-dimensionale Objekt hineinpaßt, und deren Kanten parallel zu den Koordinatenachsen sind. Die Bounding Box wird etwa bei Videospielen benutzt, um festzustellen, ob zwei Objekte gerade zusammengestoßen sind. Abbildung 10-16 zeigt ein Polygon und seine Bounding Box.

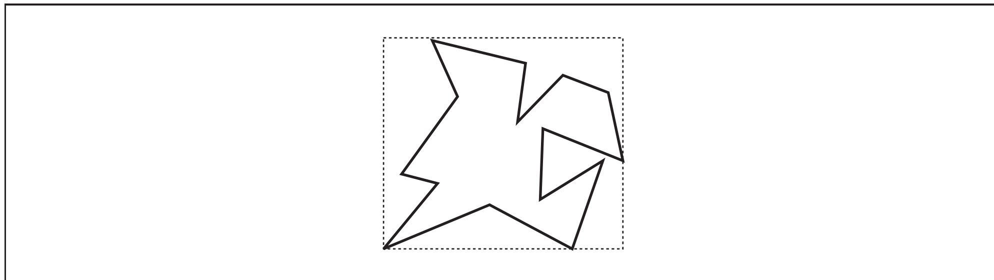

Abbildung 10-16: Ein Polygon und seine Bounding Box (gestrichelt)

Die Subroutine `bounding_box()` gibt ein Array von Punkten zurück. Bei `$d=2` Dimensionen ist die Bounding Box ein Rechteck, daher gibt `bounding_box()` vier Werte zurück: zwei gegenüberliegende Ecken des Rechtecks.

```
# bounding_box_of_points($d, @p)
#     Gibt die Bounding Box zurück, die die $d-dimensionale Punktmenge @p enthält.

sub bounding_box_of_points {
    my ($d, @points) = @_;

    my @bb;

    while (my @p = splice @points, 0, $d) {
        @bb = bounding_box($d, @p, @bb); # Weiter unten definiert.
    }

    return @bb;
}

# bounding_box($d, @p [,@b])
#     Gibt die Bounding Box der Punktmenge @p in $d Dimensionen zurück.
#     Der optionale Parameter @b gibt einen Startwert für die Bounding Box,
#     so daß wir Bounding Boxes kumulativ aus früheren zusammen mit weiteren Punkten
#     aufbauen können. Dies wird von bounding_box_of_points() benutzt.
#
#     Die Bounding Box wird als Liste zurückgegeben. Die ersten $d Elemente sind
#     die Minimum-Koordinaten, die letzten $d die größten Koordinatenwerte.
```

```
sub bounding_box {
    my ( $d, @bb ) = @_; # $d ist die Anzahl der Dimensionen.
    # Punkte herauslösen, Bounding Box in @bb zurücklassen.
    my @p = splice( @bb, 0, @bb - 2 * $d );

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

    # Jede Koordinatenrichtung durchgehen und Extremalwerte notieren.
    for ( my $i = 0; $i < $d; $i++ ) {
        for ( my $j = 0; $j < @p; $j += $d ) {
            my $ij = $i + $j;
            # Die Minima ...
            $bb[ $i      ] = $p[ $ij ] if $p[ $ij ] < $bb[ $i      ];
            #  ... und Maxima.
            $bb[ $i + $d ] = $p[ $ij ] if $p[ $ij ] > $bb[ $i + $d ];
        }
    }

    return @bb;
}

# bounding_box_intersect($d, @a, @b)
#    Gibt wahr zurück, wenn sich die zwei $d-dimensionalen Bounding Boxes @a und @b
#    überlagern. Wird von line_intersection() benutzt.

sub bounding_box_intersect {
    my ( $d, @bb ) = @_; # Anzahl Dimensionen, Koordinaten der Boxen.
    my @aa = splice( @bb, 0, 2 * $d ); # Erste Box (@bb ist die zweite Box).

    # Die Boxen müssen sich in allen Dimensionen überschneiden.
    for ( my $i_min = 0; $i_min < $d; $i_min++ ) {
        my $i_max = $i_min + $d; # Index für das Maximum.
        return 0 if ( $aa[ $i_max ] + epsilon ) < $bb[ $i_min ];
        return 0 if ( $bb[ $i_max ] + epsilon ) < $aa[ $i_min ];
    }

    return 1;
}
```

Zur Veranschaulichung berechnen wir die Bounding Box des Polygons aus Abbildung 10-17. Wir rufen `bounding_box_of_points()` mit 21 Argumenten auf, der Dimension (2) und den 10 Koordinatenpaaren der Punkte aus Abbildung 10-17:

```
@bb = bounding_box_of_points(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";
```

Wir bekommen die linke untere und die rechte obere Ecke der Bounding Box, die hier ein Quadrat ist:

```
1 1 7 7
```

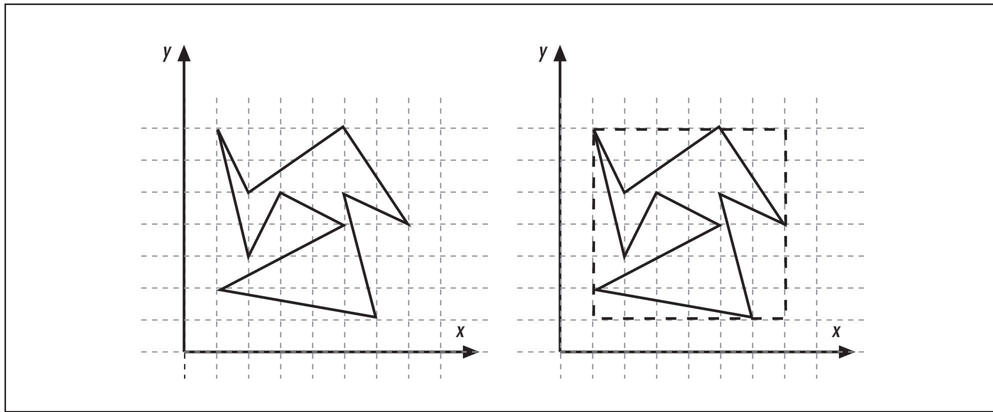

Abbildung 10-17: Ein Polygon und seine Bounding Box

Konvexe Hülle

Die *konvexe Hülle* ähnelt der Bounding Box darin, daß sie alle Punkte enthält; sie ist aber keine Box. Sie umfaßt die äußersten Punkte des Objekts, ähnlich wie ein Gummiband, das eine Anzahl Nägel in einem Nagelbrett umfaßt. Oder stellen Sie sich von Christo eingehüllte Bäume vor: die Plastikfolie stellt die konvexe Hülle dar.

Im zweidimensionalen Raum besteht die konvexe Hülle aus den Seiten eines konvexen Polygons. In drei Dimensionen sind es die Flächen eines konvexen Polyeders, bei dem alle Seitenflächen Dreiecke sind. Abbildung 10-18 zeigt ein Beispiel einer zweidimensionalen konvexen Hülle.

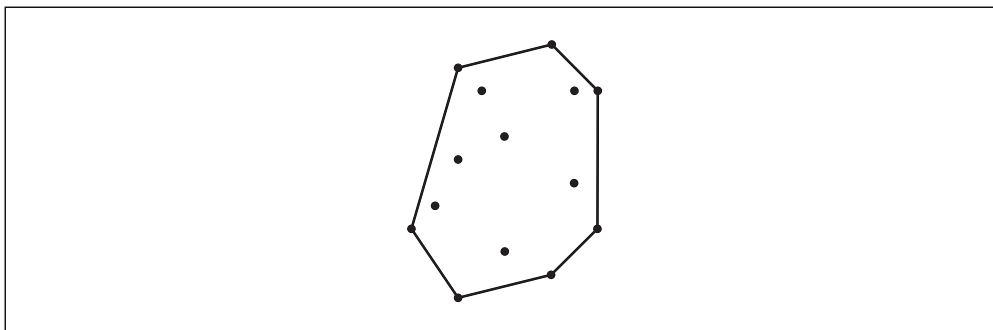

Abbildung 10-18: Konvexe Hülle einer Punktmenge

Der bekannteste Algorithmus zur Bestimmung der konvexen Hülle ist das *Durchsuchen nach Graham*. Man beginnt mit einem Punkt, für den feststeht, daß er zur konvexen Hülle gehört; meist nimmt man den Punkt mit der kleinsten *x*- oder *y*-Koordinate, wie in Abbildung 10-19(a) gezeigt.

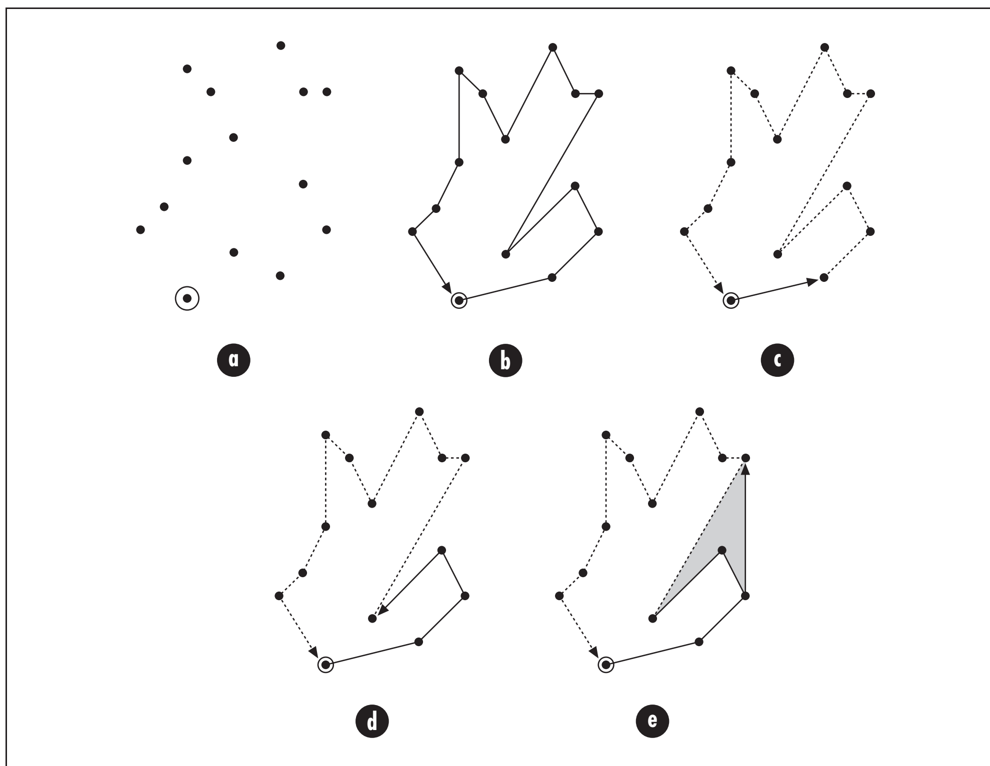

Abbildung 10-19: Durchsuchen nach Graham

Alle anderen Punkte werden nach dem Winkel sortiert, den ein Strahl vom Startpunkt aus durch diesen Punkt zur positiven *x*-Achse bildet, wie in Abbildung 10-19(b) dargestellt. Weil wir den Punkt mit der niedrigsten *y*-Koordinate als Startpunkt gewählt haben, ist sichergestellt, daß dieser Winkel zwischen 0 und π (Bogenmaß) liegt.

Der Anfang der gesuchten Hülle besteht zuerst aus der Strecke zum ersten dieser sortierten Punkte. Eine Komplikation kann entstehen, wenn zwei Punkte exakt den gleichen Winkel bilden; diese wird beseitigt, indem man eine ausgefeiltere Sortierfunktion benutzt, die bei gleichem Winkel nach *x*- und dann nach *y*-Koordinaten sortiert.

Jetzt gehen wir zum nächsten Punkt. Wenn wir dazu nach links abdrehen müssen, wird dieser nächste Punkt zur Hülle hinzugefügt.

Wenn wir uns allerdings nach rechts drehen müssen, dann war der eben hinzugefügte Punkt doch nicht einer der Hülle und muß wieder entfernt werden. Vielleicht müssen noch weitere Punkte entfernt werden, bis zum nächsten Punkt eine Linksdrehung erforderlich ist. Dieses Auf- und wieder Abbauen der Hülle legt es nahe, als Datenstruktur einen Stapel zu verwenden (wie im Abschnitt »Stapel« in Kapitel 2, *Grundlegende Datenstrukturen*, beschrieben).

Wie man in Abbildung 10-19(e) sieht, wird mit dieser »Rechtsabbiegen verboten«-Strategie rückwärts aus konkaven Gegenden (schattiert dargestellt) des Polygons herausmanövriert; übrig bleibt die konvexe Hülle. Dieser Vorgang wird mit Punkten nach steigenden Winkel so lange weitergeführt, bis der Startpunkt erreicht ist. Die Subroutine `convex_hull_graham()` berechnet die konvexe Hülle nach Graham:

```
# convex_hull_graham( @xy )
#     Berechnet die konvexe Hülle der Punktmenge @xy nach Graham.
#     Gibt die konvexe Hülle als Punktliste ($x, $y, ... ) zurück.

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

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

    # Startpunkt suchen: von den Punkten mit kleinstem y der mit kleinstem x.

    # $ymin ist der bisher kleinste y-Wert, $xmini enthält den Index des Punktes
    # mit dem bisher kleinsten x-Wert unter denen, die $ymin als y-Wert haben.
    my ( $ymin, $xmini, $xmin ) = ( $y[ 0 ], 0, $x[ 0 ] );

    for ( my $i = 1; $i < $n; $i++ ) {
        if ( $y[ $i ] + epsilon < $ymin ) {           # Neuer kleinster y-Wert.
            $ymin  = $y[ $i ];
            $xmin  = $x[ $i ];
            $xmini = $i;
        } elsif ( abs( $y[ $i ] - $ymin ) < epsilon  # Gleiches kleinstes y ...
                  and $x[ $i ] + epsilon < $xmin ) { #  ... aber kleineres x.
            $xmin  = $x[ $i ];
            $xmini = $i;
        }
    }

    splice @x, $xmini, 1;      # Startpunkt entfernen.
    splice @y, $xmini, 1;

    my @wink = map {           # Winkel zum Startpunkt berechnen.
                     atan2( $y[ $_ ] - $ymin,
                            $x[ $_ ] - $xmin)
                   } 0 .. $#x;

    # Eine ungewöhnliche Schwartzsche Transformation.
    # Wir erhalten die sortierten Indizes, so daß wir die Sortierung mehrfach,
    # für x und für y, anwenden können – eine Index-Permutation.
```

```
    my @j = map { $_->[ 0 ] }
                sort {          # Nach Winkeln sortieren, dann nach x, dann nach y.
                     return $a->[ 1 ] <=> $b->[ 1 ]              ||
                            $x[ $a->[ 0 ] ] <=> $x[ $b->[ 0 ] ] ||
                            $y[ $a->[ 0 ] ] <=> $y[ $b->[ 0 ] ];
                     }
                    map { [ $_, $wink[ $_ ] ] } 0 .. $#wink;

    @x = @x[ @j ];              # Permutieren.
    @y = @y[ @j ];

    unshift @x, $xmin;          # Startpunkt wieder in die Liste der Punkte einfügen.
    unshift @y, $ymin;

    my @h = ( 0, 1 );           # Die Hülle.

    # Zurückmanövrieren: Hülle verkleinern, solange wir rechts abbiegen müßten.
    for ( $i = 2; $i < $n; $i++ ) {
        while (@h >= 2 and      # Immer mindestens zwei Punkte in der Hülle belassen.
            clockwise( $x[ $h[ -2 ] ],
                       $y[ $h[ -2 ] ],
                       $x[ $h[ -1 ] ],
                       $y[ $h[ -1 ] ],
                       $x[ $i ],
                       $y[ $i ] ) > - epsilon ) {
            pop @h;
        }
        push @h, $i;            # Punkt in die Hülle aufnehmen.
    }

    # x- und y-Koordinaten paarweise zur einer Liste zusammensetzen.
    return map { ( $x[ $h[ $_ ] ], $y[ $h[ $_ ] ] ) } 0 .. $#h;
}
```

Das Durchsuchen nach Graham läßt sich optimieren, indem man die Anzahl der Punkte einschränkt, die untersucht werden müssen. Eine Möglichkeit ist das Entfernen von inneren Punkten: die *innere Elimination*. Punkte, von denen bekannt ist, daß sie nicht zur konvexen Hülle gehören können, werden gar nicht erst berücksichtigt. Von welchen Punkten man das wissen kann, hängt von der Verteilung der Punkte ab. Wenn die Verteilung in allen Richtungen etwa gleich ist (eine Gleichverteilung in einem Rechteck), dann ist ein Viereck, das aus dem Punkten gebildet wird, die den Eckpunkten dieses Rechtecks am nächsten sind, ein idealer Eliminator. Die Punkte im Inneren dieses Vierecks brauchen gar nicht erst berücksichtigt zu werden, wie aus Abbildung 10-20 hervorgeht.

Die den Eckpunkten nächsten Punkte kann man durch Summen- und Differenzbildung ziemlich einfach finden:

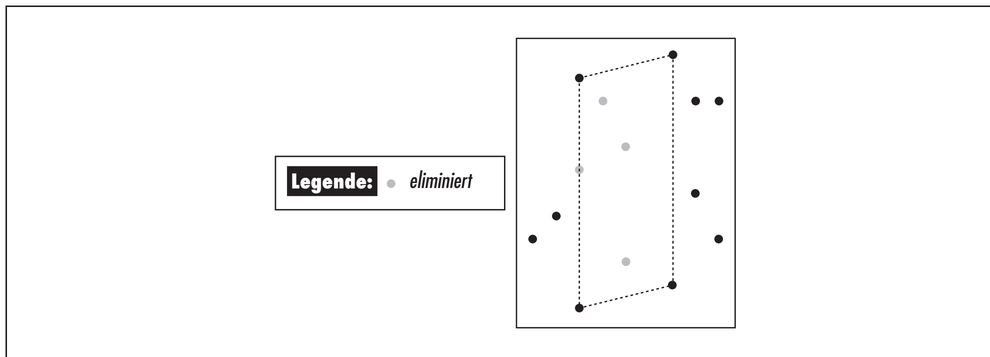

Abbildung 10-20: Durchsuchen nach Graham: Eliminieren von offensichtlich inneren Punkten

- kleinste Summe: Ecke unten links
- größte Summe: Ecke oben rechts
- kleinste Differenz: Ecke oben links
- größte Differenz: Ecke unten rechts

In Perl formuliert sieht das etwa so aus:

```
# Größte und kleinste Summen und Differenzen finden (oder vielmehr die Indizes
# der entsprechenden Punkte in den Arrays @x und @y).

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

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

my $ul = $sort_by_sum [  0 ]; # Index der Ecke unten links des Eliminator-Vierecks.
my $or = $sort_by_sum [ -1 ]; # oben rechts.
my $ol = $sort_by_diff[  0 ]; # oben links.
my $ur = $sort_by_diff[ -1 ]; # unten rechts.
```

Dieser Ansatz birgt eine Gefahr. Wir dürfen nur die Punkte eliminieren, die *streng* innerhalb des Vierecks liegen. Punkte auf den Kanten des Vierecks könnten zur Hülle gehören, die Ecken des Vierecks selbst gehören sogar sicher dazu. Ein Ausweg wäre es, ein um ein »kleines bißchen« kleineres Viereck zu nehmen. Wenn dieses ε gut gewählt wird, gehen dabei kaum Punkte verloren, und es können sofort viele Punkte eliminiert werden.

Die zeitliche Komplexität von `convex_hull_graham` ist $O(N \log N)$, das ist optimal.

Dichtestes Punktepaar

Welche zwei Punkte in einer Punktemenge liegen am nächsten beieinander? Der naive Ansatz – einfach die Distanzen zwischen allen Punktepaaren berechnen – funktioniert, ist aber langsam: $O(N^2)$. Eine praktische Anwendung ließe sich z. B. in der Flugverkehrssimulation oder -kontrolle denken: Zwei Jumbo-Jets sollten sich besser nicht am exakt gleichen Ort aufhalten. Mit Bounding Boxes lassen sich Kollisionen ermitteln; mit der Suche nach den Punkten, die am dichtesten beieinander liegen, kann man sie voraussehen und vielleicht vermeiden. Wir werden die Punkte aus Abbildung 10-21 als Beispiel verwenden.

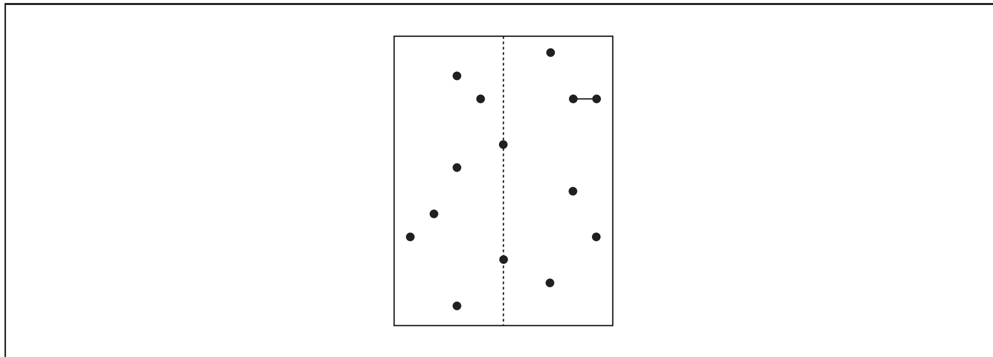

Abbildung 10-21: Eine Punktmenge und das dichteste Punktepaar

Wir können von einem Lokalitätskriterium ausgehen: Wenn wir ein Gebiet aufteilen, ist es wahrscheinlich, daß ein Punkt auf der linken Seite anderen Punkten auf der linken Seite näher liegt als solchen zur Rechten. Wir wenden einmal mehr die »Teile-und-Herrsche«-Strategie an (vergleiche dazu den Abschnitt »Wiederkehrende Themen bei Algorithmen« in Kapitel 1, der *Einführung*) und teilen die Punktemenge wiederholt und rekursiv in linke und rechte Teile auf, wie in Abbildung 10-22 dargestellt.

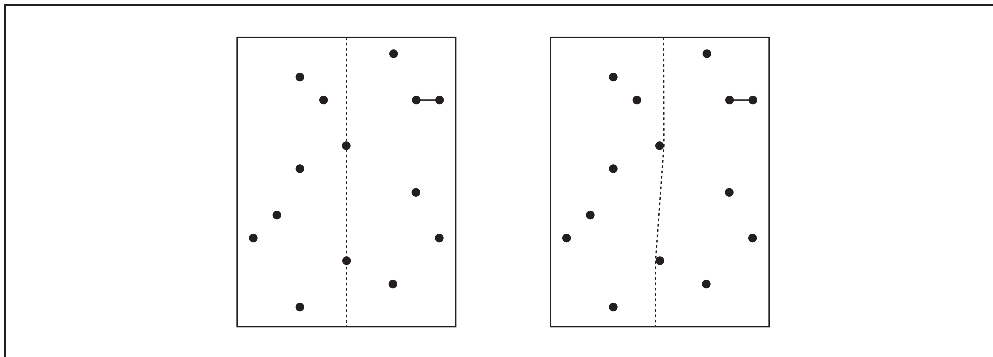

Abbildung 10-22: Rekursives Aufteilen: Geometrische und logische Betrachtungsweise

Sie wundern sich über die geknickte Linie im rechten Teil von Abbildung 10-22? Die Trennungslinie fällt exakt auf zwei Punkte, die die gleiche *x*-Koordinate haben. Daher zeigen wir auch die »logische« Ansicht, in der jeder Punkt klar zur einen oder zur anderen Hälfte gehört.

In Abbildung 10-23 sind die senkrechten Streifen dargestellt, wie sie aus der rekursiven Links-Rechts-Aufteilung entstehen. Die Streifen sind beschriftet: `lrr` ist beispielsweise der Streifen, der aus einer Links-Teilung, gefolgt von zwei Rechts-Teilungen resultiert.

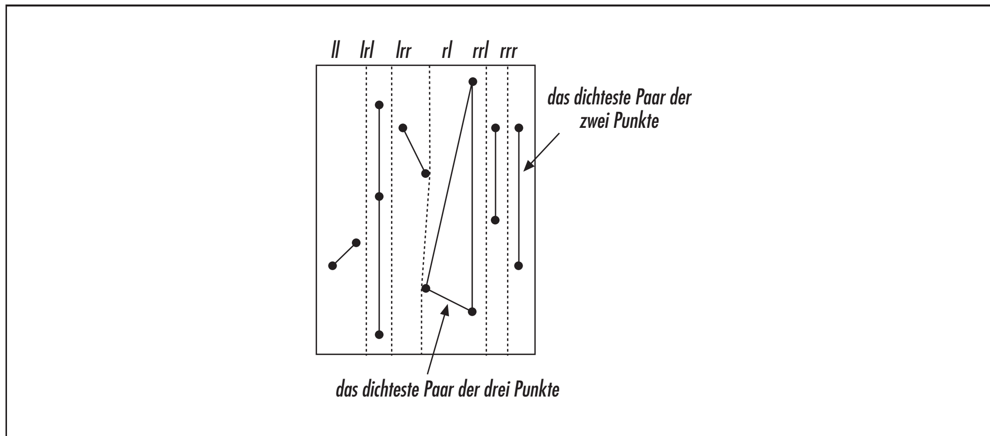

Abbildung 10-23: Aus rekursiver Aufteilung entstandene Streifen und dichteste Paare darin

Wenn ein Streifen nur noch zwei oder drei Punkte enthält, wird die Rekursion abgebrochen. In diesen Fällen ist es trivial einfach, die kürzeste Distanz zu finden – oder eben das dichteste Punktepaar (siehe Abbildung 10-23).

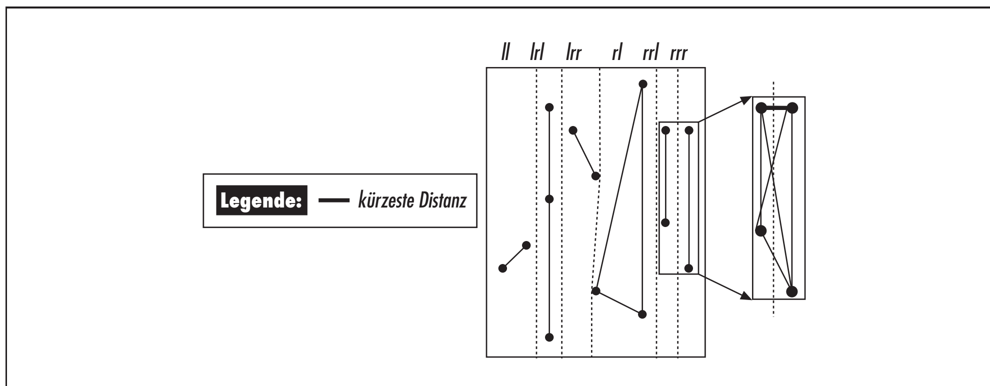

Abbildung 10-24: Die aus der rekursiven Teilung entstandenen Streifen zusammenführen

Aber was ist zu tun, wenn wir aus der Rekursion wieder zurückkehren? In jedem Streifen gibt es nun eine private »kürzeste Distanz«. Wir können aber nicht einfach die

kürzeste unter diesen nehmen, weil die wirklich kürzeste vielleicht eine Teilungslinie überschreitet, wie jene in Abbildung 10-24.

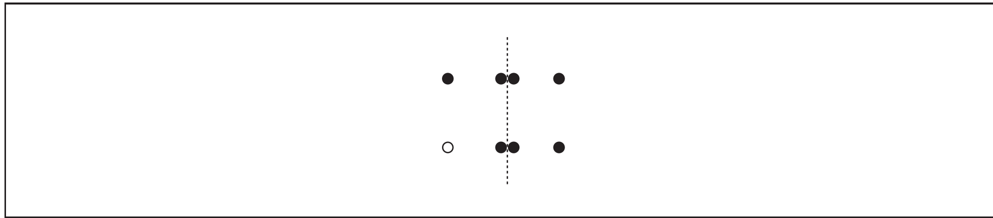

Abbildung 10-25: Der komplizierteste Fall beim Zusammenführen von Streifen. Der aktuelle Punkt ist weiß markiert.

Wir verfahren wie folgt: Bei jeder Trennungslinie müssen wir nur die Punkte berücksichtigen, die näher an der Trennungslinie liegen als die kürzeste bisher gefundene Distanz. Diese Punkte gehen wir in der Reihenfolge ihrer y-Koordinaten durch. Pro Punkt müssen wir im ungünstigsten Fall die Distanzen zu den sieben anderen in Abbildung 10-25 gezeigten Punkten berechnen.

Das entsprechende Perl-Programm ist etwas kompliziert, weil die Punktemenge gleichzeitig auf verschiedene Arten sortiert werden muß: in der ursprünglichen Ordnung, die Punkte horizontal sortiert (so nehmen wir die Aufteilung in Streifen vor), und in vertikaler Sortierung (so suchen wir nach Distanzen, die die Trennungslinien überschreiten). Diese verschiedenen Darstellungen der gleichen Punktemenge werden mit *Permutationsvektoren* realisiert, wofür wir in Perl Arrays nehmen. Zum Beispiel enthält `@yoi` den »vertikalen Rang« jedes Punktes, von oben nach unten.

Es scheint, daß die »Teile-und-Herrsche«-Strategie zu einem $O(N \log N)$-Algorithmus führt. Das stimmt aber nur, wenn die Rekursion selbst von der Komplexität $O(N)$ ist. Wir können daher nicht innerhalb der Rekursion `sort()` benutzen, sonst ist unser angestrebtes $O(N \log N)$-Resultat dahin. Wir sortieren deshalb nur einmal horizontal und einmal vertikal und rufen dann die rekursive Teilung auf.

Hier kommt sie nun, unsere angsteinflößend lange `closest_points()`-Subroutine:

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

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

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

    # Berechnung der Permutations- und Ordinalindizes.

    # xpi: X-Permutationsindex.
    #
    # Wenn @$unsorted_x (18, 7, 25, 11) ist, dann wird @xpi zu (1, 3, 0, 2),
    # d. h. $xpi[0] == 1 bedeutet, daß das nach x sortierte $sorted_x[0] in
    # $unsorted_x->[1] zu finden ist.
```

```
    # Wir benutzen diese Index-Transformation, weil wir so einerseits @$unsorted_x
    # nach @sorted_x sortieren können, andererseits aber die ursprüngliche Reihenfolge
    # mit @sorted_x[@xpi] wiederherstellen können.
    # Das ist notwendig, weil wir die Punkte sowohl nach x als auch nach y sortieren
    # wollen, wir wollen aber auch die gefundenen Punkte in den ursprünglichen
    # Arrays finden: »Der 12. und der 42. Punkt bilden das dichteste Paar.«

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

    # ypi: Y-Permutationsindex.
    #
    my @ypi = sort { $unsorted_y->[ $a ] <=> $unsorted_y->[ $b ] }
                   0..$#$unsorted_y;

    # yoi: Y-Ordinalindex.
    #
    # Der Ordinalindex ist das Gegenstück zum Permutationsindex: Wenn @$unsorted_y
    # gleich (16, 3, 42, 10) und @ypi gleich (1, 3, 0, 2) ist, dann wird @yoi zu
    # (2, 0, 3, 1); mit anderen Worten, $yoi[0] == 1 bedeutet, daß $unsorted_y->[0]
    # das gleiche Element ist wie $sorted_y[1] .

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

    # Dichteste Punkte mit rekursivem Aufruf bestimmen.
    my ( $p, $q, $d ) = _closest_points_recurse( [ @$unsorted_x[@xpi] ],
                                                 [ @$unsorted_y[@xpi] ],
                                                 \@xpi, \@yoi, 0, $#xpi
                                               );

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

    ( $pi, $qi ) = ( $qi, $pi ) if $pi > $qi; # Niedrigeren Index zuerst.
    return ( $pi, $qi, $d );
}

sub _closest_points_recurse {
    my ( $x, $y, $xpi, $yoi, $x_l, $x_r ) = @_;

    # $x, $y: Referenzen auf die Arrays mit den x- und y-Koordinaten, nach x sortiert.
    # $xpi:   X-Permutationsindizes, von closest_points_recurse() berechnet.
    # $yoi:   Y-Ordinalindizes, wurden von closest_points_recurse() berechnet.
    # $x_l:   Linke Grenze des im Moment interessierenden Teils der Punktemenge.
    # $x_r:   Rechte Grenze des im Moment interessierenden Teils der Punktemenge.
    #         Das heißt, nur die Punkte $x->[$x_l..$x_r] und $y->[$x_l..$x_r]
    #         werden in diesem Schritt untersucht.
```

```
my $d;      # Die kleinste bisher gefundene Distanz.
my $p;      # Index des Punktes am einen Ende der Minimaldistanz ...
my $q;      #  ... und des anderen Punktes.

my $N = $x_r - $x_l + 1;                     # Anzahl der Punkte im Streifen.

if ( $N > 3 ) {                              # Noch zu viele – rekursiv aufteilen!
    my $x_lr = int( ( $x_l + $x_r ) / 2 ); # Rechte Grenze der linken Hälfte.
    my $x_rl = $x_lr + 1;                    # Linke Grenze der rechten Hälfte.

    # Zuerst herausfinden, was die Teile für Resultate zurückliefern.

    my ( $p1, $q1, $d1 ) =
        _closest_points_recurse( $x, $y, $xpi, $yoi, $x_l, $x_lr );
    my ( $p2, $q2, $d2 ) =
        _closest_points_recurse( $x, $y, $xpi, $yoi, $x_rl, $x_r );

    # Resultate von beiden Hälften zusammenführen.

    # $d, $p und $q nachführen: Die kürzeste bisher gefundene Distanz und die
    # Indizes der daran beteiligten Punkte.

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

    # Überlappungsbereich überprüfen.

    # Die x-Koordinate auf halbem Weg zwischen der linken und der rechten Hälfte.
    my $x_d = ( $x->[ $x_lr ] + $x->[ $x_rl ] ) / 2;

    # Die Indizes von »möglichen« Punkten, d. h. von Punktepaaren, die die
    # Trennungslinie überschreiten und vielleicht näher beieinander liegen als die
    # bisher kürzeste Distanz.
    my @xi;

    # Mögliche Punkte aus der linken Hälfte finden.

    # Linke Grenze des linken Segments mit möglichen Punkten.
    my $x_ll;

    if ( $x_lr == $x_l ) { $x_ll = $x_l }
    else {                                       # Binäre Suche.
        my $x_ll_lo = $x_l;
        my $x_ll_hi = $x_lr;
        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_lr;

# Mögliche Punkte aus der rechten Hälfte finden.

# Rechte Grenze des rechten Segments mit möglichen Punkten.
my $x_rr;

if ( $x_rl == $x_r ) { $x_rr = $x_r }
else {                                      # Binäre Suche.
    my $x_rr_lo = $x_rl;
    my $x_rr_hi = $x_r;
    do { $x_rr = int( ( $x_rr_lo + $x_rr_hi ) / 2 );
         if ( $x->[ $x_rr ] - $x_d > $d ) {
            $x_rr_hi = $x_rr - 1;
         } elsif ( $x->[ $x_rr ] - $x_d < $d ) {
            $x_rr_lo = $x_rr + 1;
         }
    } until $x_rr_hi < $x_rr_lo
        or ( $x->[ $x_rr ] - $x_d <= $d
             and ( $x_rr == $x_r or
                   $x->[ $x_rr + 1 ] - $x_d >= $d ) );
}
push @xi, $x_rl..$x_rr;

# Jetzt haben wir die »möglichen« Punkte. Taugen sie etwas?
# Achtung: Dichter Nebel voraus!

# Zunächst Punkte nach ihren ursprünglichen Indizes sortieren.

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 ];

# Jedes mögliche Punktepaar untersuchen (der erste Punkt aus der linken
# Hälfte, der zweite aus der rechten).
```

```
        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++ ) {
                # Punktepaare überspringen, die gar nicht näher als die bisher
                # kürzeste Distanz sein können.
                last if $xi_by_y[ $j ] - $i_i > 7; # Indizes zu verschieden?
                my $y_j = $y_by_yi[ $j ];
                my $dy = $y_j - $y_i;
                last if $dy > $d;                  # Distanz zu hoch (in y)?
                my $x_j = $x_by_yi[ $j ];
                my $dx = $x_j - $x_i;
                next if abs( $dx ) > $d;           # Distanz in x zu breit?
                # Tortur überlebt! Vielleicht haben wir einen Gewinner. Distanz
                # pythagoräisch-klassisch berechnen; wenn kürzer, nachführen.
                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 ) {        # Nur drei Punkte? Rekursion abbrechen.
        my $x_m = $x_l + 1;
        # Distanz-Quadrate vergleichen und Wurzel nur einmal ziehen.
        my $s1 = ($x->[ $x_l ]-$x->[ $x_m ])**2 +
                 ($y->[ $x_l ]-$y->[ $x_m ])**2;
        my $s2 = ($x->[ $x_m ]-$x->[ $x_r ])**2 +
                 ($y->[ $x_m ]-$y->[ $x_r ])**2;
        my $s3 = ($x->[ $x_l ]-$x->[ $x_r ])**2 +
                 ($y->[ $x_l ]-$y->[ $x_r ])**2;
        if ( $s1 < $s2 ) {
            if ( $s1 < $s3 )  { $d = $s1;  $p = $x_l;  $q = $x_m }
            else              { $d = $s3;  $p = $x_l;  $q = $x_r }
        } elsif ( $s2 < $s3 ) { $d = $s2;  $p = $x_m;  $q = $x_r }
          else                { $d = $s3;  $p = $x_l;  $q = $x_r }

        $d = sqrt $d;
    } elsif ( $N == 2 ) {           # Nur zwei Punkte? Rekursion abbrechen.
        $d = sqrt(($x->[ $x_l ]-$x->[ $x_r ])**2 +
                  ($y->[ $x_l ]-$y->[ $x_r ])**2);
        $p = $x_l;
        $q = $x_r;
    } else {                        # Weniger als zwei Punkte – etwas ist faul.
        return ( );
    }

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

Die zeitliche Komplexität von `closest_points()` ist $O(N \log N)$, was in der Zwischenzeit ein bekannter Ausdruck sein dürfte und abgesehen davon ein erfreuliches Resultat darstellt. Wir testen die Routine mit den Punkten aus Abbildung 10-26.

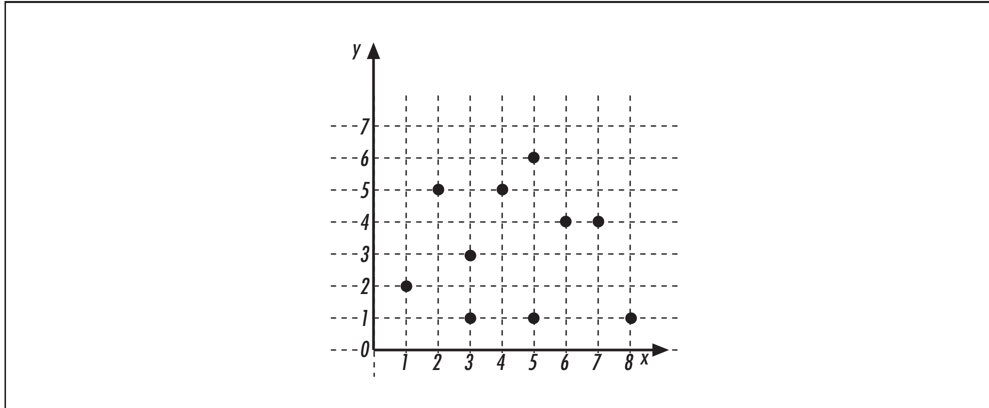

Abbildung 10-26: Beispieldaten für das Problem des dichtesten Punktepaars

Wir suchen die am dichtesten beieinanderliegenden Punkte aus den zehn Punkten von Abbildung 10-26 wie folgt:

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

Das Resultat:

```
7 8 1
```

Das besagt, daß der 8. und der 9. Punkt – (6, 4) und (7, 4), weil die Arrays in Perl ab 0 numeriert werden – am nächsten beieinander liegen und daß die Distanz 1 beträgt.

Geometrische Algorithmen – Zusammenfassung

Geometrische Algorithmen beruhen oft auf bekannten Formeln aus der Geometrie. Aber Achtung: Die Übersetzung einer Formel in ein Programm ist oft vertrackter, als es den Anschein hat. Der eigentliche Grund für die Probleme ist der Unterschied zwischen den exakten Werten aus der Mathematik und der ungenauen Darstellung im Computer (der Euphemismus für diese unvermeidliche Übersetzung lautet *diskrete Darstellung*). Ein Punkt liegt genau auf der Kreuzung von $x - 1$ und $1 - 2x$, denkt man, aber die Maschine denkt anders. Auch ein Kreis mit Radius 1 besteht nicht aus π Pixeln.

Graphik-Module auf dem CPAN

Die in diesem Kapitel besprochenen Algorithmen zeichnen nie etwas auf den Bildschirm. Dafür brauchen Sie eines der Pakete aus diesem Abschnitt. Die meisten von ihnen sind Perl-Schnittstellen zu externen Bibliotheken; die Bibliothekspakete müssen zuerst installiert werden. Wo man diese findet, steht in der Dokumentation der einzelnen Module; die Module selbst finden Sie, natürlich, auf *http://www.perl.com/CPAN/modules*.

Zweidimensionale Graphik

Es gibt fünf Pakete auf dem CPAN, mit denen man zweidimensionale Bilder behandeln kann: Perl-Gimp, GD, Image::Size, PerlMagick und PGPLOT.

Perl-Gimp

Der Gimp ist das bekannte Programm aus der Linux-Welt, das ähnlich wie Photoshop von Adobe funktioniert; siehe *http://www.gimp.org*. Perl-Gimp von Marc Lehmann ist das Perl-API dazu. Man kann damit Bilder verzerren, entflecken, Schatten hinzufügen und tausend andere Effekte erzielen.

GD

Das GD-Modul von Lincoln D. Stein ist eine Schnittstelle zu *libgd*, einer Sammlung von Routinen, mit der man GIF-Bilder »zeichnen« kann.[4] Mit diesem Beispiel würde man ein GIF mit einem Kreis erzeugen:

```
use GD;

# Neues Bild erzeugen.
my $gif = new GD::Image(100, 100);

# Farben allozieren.
my $weiss = $gif->colorAllocate(255, 255, 255);
my $rot   = $gif->colorAllocate(255,   0,   0);

# Hintergrundfarbe.
$gif->transparent($weiss);

# Kreis zeichnen.
$gif->arc(50, 50,          # Koordinaten des Kreismittelpunktes.
          30, 30,          # Breite, Höhe.
          0, 360,          # Anfangs- und Endwinkel.
          $rot);           # Farbe.
```

4 Neuere Versionen von *libgd* und damit auch das GD-Modul benutzen aus Lizenzgründen nicht mehr GIF, sondern PNG (Portable Network Graphics). Alle neueren Webbrowser und viele andere Programme können auch mit PNG umgehen, zudem sind die Dateien meist kompakter. Anm. d. Ü.

```
# Bild ausgeben.
open(GIF, ">circle.gif") or die "Kann Datei nicht öffnen: $!\n";
binmode GIF;
print GIF $gif->gif;
close GIF;
```

Image::Size

Das Image::Size-Modul von Randy J. Ray ist ein spezialisiertes Modul, das die Abmessungen von Bildern vieler Formate herausfindet. Das mag ziemlich abwegig klingen, es gibt dafür aber eine wichtige Anwendung im World Wide Web. Wenn ein Webserver eine Seite absendet, sollte er so früh wie möglich die Abmessungen der darin enthaltenen Bilder mitschicken; vor den eigentlichen Bildern, deren Übertragung oft lange dauert. So kann der Browser Platz für die Bilder bereitstellen und die Seite vorerst mit leeren Kästen darstellen. Wenn die Bilder wirklich übertragen sind, braucht das Layout nicht mehr abrupt umgestellt zu werden.

PerlMagick

Das PerlMagick-Modul von Kyle Shorter ist eine Schnittstelle zu *ImageMagick*, einer umfangreichen Bibliothek zur Konversion und Manipulation von Bildern. Man kann damit von einem Graphikformat in andere übersetzen und dabei alle Arten von Filtern anwenden; beispielsweise Filter für die Farbabstimmung oder irgendwelche Spezialeffekte. Siehe *http://www.wizards.dupont.com/cristy/www/perl.html*.

PGPLOT

Das Modul PGPLOT von Karl Glazebrook ist das Bindeglied von Perl zur PGPLOT-Graphikbibliothek. Mit PGPLOT kann man Zeichnungen von Kurven und Beschriftungen anfertigen; wirklich interessant aber wird PGPLOT im Zusammenspiel mit PDL, einer Sprache für numerische Probleme (auch PDL ist ein Perl-Modul, siehe Kapitel 7, *Matrizen*). Und weil in PDL auch alle Möglichkeiten von Perl vorhanden sind, ist die Kombination schon fast beängstigend gut. Mehr Information zum PGPLOT-Modul gibt es auf *http://www.ast.cam.ac.uk/AAO/local/www/kgb/pgperl/*.

Balkendiagramme, Geschäftsgraphiken

Wenn Sie unter »Graphik« nur »Geschäftsgraphik« verstehen (Balken-, Kuchendiagramme usw.), dann benötigen Sie vielleicht das Chart- oder GIFgraph-Modul von David Bonner und Martien Verbruggen. Damit kann man etwa Statistiken über den Datenverkehr eines Webservers auf Anfrage erstellen. Beide benutzen das GD-Modul.

Dreidimensionale Graphik

Erst in den letzten Jahren wurde es möglich, ernstzunehmende, realistische 3-D-Graphik auch auf Computern zu erzeugen, die sich jeder leisten kann. Zu drei Toolkits gibt es frei erhältliche CPAN-Module: OpenGL, Renderman und VRML.

OpenGL

Das OpenGL-Modul von Stan Melax implementiert die OpenGL-Sprache für Perl. OpenGL ist die herstellerunabhängige Version der GL-Sprache von Silicon Graphics; eine 3-D-Modellierungssprache, in der man »Welten« mit komplexen Objekten und Beleuchtungsbedingungen beschreiben kann. Das bekannte Computerspiel *Quake* wurde mit OpenGL gemacht. Es gibt eine frei zugängliche Implementation von OpenGL namens *Mesa* von *http://www.mesa3d.org/*.

Renderman

Das Renderman-Modul ist die Perl-Schnittstelle zu Renderman, einem System für photorealistische Modellierung. Schreiben Sie Ihre Toy Story in Perl!

VRML

Hartmut Palm hat eine Perl-Schnittstelle zur Virtual Reality Markup Language geschrieben, mit der man dreidimensionale Welten beschreiben und VRML-Output erzeugen kann. Wenn ein Besucher Ihrer Website das entsprechende Plug-In installiert hat, kann er in Ihrer Welt herumwandern. Das Modul nennt sich – wen erstaunt's – VRML.

Graphische Oberflächen, Widget-Systeme

Wenn Sie vom Web unabhängige Perl-Programme mit graphischen Oberflächen schreiben wollen, benötigen Sie ein Paket aus diesem Abschnitt. Perl/Tk ist sicher das umfangreichste und portabelste System.

Perl/Tk

Perl/Tk von Nick Ing-Simmons ist das bekannteste und wohl beste System für graphische Oberflächen in Perl.[5] Es funktioniert unter dem X11-Window-System und unter Windows 95/98/NT/2k.

Einfache Dinge sind in Perl/Tk einfach zu programmieren. Hier ein Minimal-Programm, das eine Taste (*Button*) erzeugt:

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

5 Verwechseln Sie das *Modul* Tk nicht mit dem Tk-*Toolkit*. Dieses wurde ursprünglich von John Ousterhout für seine Sprache Tcl geschrieben. Das Tk-Toolkit ist aber sprachunabhängig, und deshalb gibt es eine Anbindung an Perl, wie auch an andere Sprachen. Das Perl/Tk-Modul ist die Schnittstelle zum Tk-Toolkit.

Der Button ist mit einer *Aktion* verbunden. Wenn er betätigt wird, wird `Hello, world!` auf die Standardausgabe geschrieben. Man kann mit Tk komplexe Anwendungen und graphische Applikationen programmieren, aber das geht bei weitem über den Rahmen dieses Buches hinaus. Perl/Tk ist ein ganzes Buch wert: *Einführung in Perl/Tk* von Nancy Walsh (O'Reilly, 2000).

Andere Window-Toolkits

Es gibt Anbindungen für Perl für eine ganze Reihe von anderen Window-Programmiersystemen. Die meisten sind nur unter dem X11-Window-System lauffähig, das vor allem in der Unix-Welt benutzt wird, aber zu manchen soll es bald Portierungen auf Windows geben (Gtk, Mitte 1999).

Gnome, von Kenneth Albanowski
: Das GNU Object Model Environment (*http://www.gnome.org*).

Gtk, von Kenneth Albanowski
: Die Bibliothek, die ursprünglich nur für den Gimp verwendet wurde.

Sx, von Frederic Chaveau
: Simple *Athena Widgets* für X.

X11::Motif, von Ken Fox
: Ein Motif-Toolkit.