After having replaced two triangles with two better ones that form the same quadrangle, we repeat the search process, searching all triangles again. After all, a triangle created by diagonal ßipping may later successfully be combined with another adjacent triangle. This is implemented in the class Polygon2D, listed below.

```
// Polygon2D: A class to triangulate a polygon.
import java.util.Vector;

public class Polygon2D {
    Point2D[] vertices;
    int[] nrs;
    int n;

    Polygon2D(Point2D[] p) {
        vertices = p;
        n = p.length;
        nrs = new int[n];
        for (int i=0; i<n; i++)
            nrs[i] = i;
    }

    Polygon2D(Point2D[] p, int[] index) { // Used for 3D applications
        vertices = p;
            n = index.length;
            nrs = index;
    }

    ßoat angle(Point2D a, Point2D b, Point2D c){
      // Return angle ABC, using dot product          u . v =  |u||v|cos angle
       double xBA = a.x - b.x, yBA = a.y - b.y,
              xBC = c.x - b.x, yBC = c.y - b.y,
              dotproduct = xBA * xBC + yBA * yBC,
              lenBA = Math.sqrt(xBA * xBA + yBA * yBA),
              lenBC = Math.sqrt(xBC * xBC + yBC * yBC),
              cosB = dotproduct / (lenBA * lenBC);
       return (ßoat)Math.acos(cosB);
    }

    boolean ßippingDesirable(int iP, int iQ, int iR, int iS) {
        // Currently, diagonal PR divides quadrangle PQRS into
        // two triangles. Is the alternative diagonal QS a
        // better choice?
        Point2D vP = vertices[iP], vQ = vertices[iQ],
                vR = vertices[iR], vS = vertices[iS];
```

```
    // Compute the angles at the opposite vertices Q and S.
    // Flipping is desirable if (angle Q) + (angle S) > pi
    return angle(vP, vQ, vR) + angle(vR, vS, vP) > Math.PI;
}

boolean anyFlipping(Tria[] trias) {
    if (trias.length < 2)
        return false;
    for (int i=0; i<trias.length; i++){
        int[] t = {trias[i].iA, trias[i].iB, trias[i].iC};
        for (int j=i+1; j<trias.length; j++){
            int[] u = {trias[j].iA, trias[j].iB, trias[j].iC};
            // Look for a common edge of triangles t and u
            for (int h=0; h<3; h++){
                for (int k=0; k<3; k++){
                    if (t[h] == u[k] &&
                       t[(h+1)%3] == u[(k+2)%3]){
                        int iP = t[(h+1)%3], iQ = t[(h+2)%3],
                              iR = t[h], iS = u[(k+1)%3];
                        if (ßippingDesirable(iP, iQ, iR, iS)){
                            trias[i] = new Tria(iP, iQ, iS);
                            trias[j] = new Tria(iS, iQ, iR);
                            return true;
                        }
                    }
                }
            }
        }
    }
    return false;
}

Tria[] triangulate(){
    Tria[] tr = new Tria[n - 2];
    int[] next = new int[n];
    for (int i = 0; i < n; i++)
        next[i] = (i + 1) % n;

    for (int k = 0; k < n - 2; k++) {
        // Find a suitable triangle, consisting of two edges
        // and an internal diagonal:
        Point2D a, b, c;
        boolean triaFound = false;
        int iA = 0;
```

```
            int count = 0;
            while (!triaFound && ++count < n) {
                int iB = next[iA], iC = next[iB];
                int nA = Math.abs(nrs[iA]),
                      nB = Math.abs(nrs[iB]),
                      nC = Math.abs(nrs[iC]);
                a = vertices[nA];
                b = vertices[nB];
                c = vertices[nC];
                if (Tools2D.area2(a, b, c) >= 0) {
                    // Edges AB and BC; diagonal AC.
                    // Test to see if no other polygon vertex
                    // lies within triangle ABC:
                    int j = next[iC];
                    int nj = Math.abs(nrs[j]);
                    while (j != iA &&
                            (nj == nA || nj == nB || nj == nC) ||
                             !Tools2D.insideTriangle(a, b, c,
                                 vertices[nj])){
                        j = next[j];
                        nj = Math.abs(nrs[j]);
                    }
                    if (j == iA) {
                        // Triangle ABC contains no other vertex:
                        tr[k] = new Tria(nA, nB, nC);
                        next[iA] = iC;
                        triaFound = true;
                    }
                }
                iA = next[iA];
            }

            if (count == n)
            {   System.out.println("Not a simple polygon" +
                    " or vertex sequence not counter-clockwise.");
                System.exit(1);
            }
        }

        while (anyFlipping(tr))
            ; // Keep ßipping diagonals as long as this is
              // desirable
        return tr;
    }
}
```

You may have noticed that this class contains two Polygon2D constructors. In this chapter, we will use only the first. When dealing with 3D objects in Chap. 6, we will use the second, which is useful if we want to use only one array of Point2D objects to store the vertices of several polygons. Then the second argument, index, of this constructor, indicates which of all those vertices comprise the polygon that is to be triangulated. Actually, the first constructor, with only the array as an argument, builds such an array, internally named nrs. Initially, for a polygon of n vertices, it is assumed that this polygon has the vertex numbers 0, ..., n − 1, which are stored in the array nrs. So both now and in later chapters, the polygon has the following vertex numbers:

vertices[nrs[0]], vertices[nrs[1]], ..., vertices[nrs[n − 1]]

We also use the array next, so we can easily deal with the vertices in counter-clockwise order. Initially,

$$next[i] = i + 1 \text{ for } i = 0, 1, \ldots, n - 2$$
$$next[n - 1] = 0$$

This is achieved by means of the following fragment:

```
for (int i = 0; i < n; i++)
    next[i] = (i + 1) % n;
```

This use of the modulo operator % for cyclic traversing the vertices of polygons and triangles is often used in this program, so it is essential to understand it. The introduction of this array next enables us to deal with reduced polygons after triangles have been cut off. You can see this happening in the fragment.

```
if (j == iA) { // Triangle ABC contains no other vertex:
    tr[k] = new Tria(nA, nB, nC);
      next[iA] = iC;
    triaFound = true;
}
```

For example if the polygon has 5 vertices, numbered 0, 1, 2, 3, 4, and triangle 2-3-4 is cut off the polygon, vertex 4 replaces vertex 3 as the successor of vertex 2, so we make next[2] = 4.

To see this class Polygon2D and the triangulate method in action, we will use a demonstration program, PolyTria.java. It allows the user to define a polygon, in the same way as we did with program DefPoly.java in Sect. 1.4, but this time this polygon will be divided into triangles, which appear in different colors.

This program, PolyTria.java, uses the above classes Tria and Tools2D, as well as the class CvDefPoly, which occurs in program DefPoly.java of Sect. 1.4. In our subclass CvPolyTria, we apply the method ccw, discussed in Sect. 2.4, to the given polygon to examine the orientation of its vertex sequence. If this happens to be clockwise, we put the vertices in reverse order in the array p, so that the vertex sequence will be counter-clockwise in this array, which we can then safely pass on to the Polygon2D constructor:

```
// PolyTria.java: Drawing a polygon and dividing it into triangles.
// Uses: CvDefPoly, Point2D (Section 1.4),
//       Tools2D (Section 2.3), Tria, Polygon2D (discussed above).

import java.awt.*;
import java.awt.event.*;

public class PolyTria extends Frame {
    public static void main(String[] args) {new PolyTria();}

    PolyTria() {
        super("Deþne polygon vertices by clicking");
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);}
        });
        setSize(1500, 900);
        add("Center", new CvPolyTria());
        setCursor(Cursor.getPredeþnedCursor
        (Cursor.CROSSHAIR_CURSOR));
        setVisible(true);
    }
}

class CvPolyTria extends CvDefPoly { // see Section 1.4
    public void paint(Graphics g) {
        int n = v.size(); // v is deþned in superclass CvDefPoly
        if (n >= 3 && ready) {
            Point2D[] p = new Point2D[n];
            for (int i = 0; i < n; i++)
                p[i] = (Point2D) v.elementAt(i);
            // If not counter-clockwise, reverse the order:
            if (!ccw(p))
                for (int i = 0; i < n; i++)
                    p[i] = (Point2D) v.elementAt(n - i - 1);
```

```
            Polygon2D polygon = new Polygon2D(p);
            Tria[] t = polygon.triangulate();
            initgr();
            if (t != null) {
                for (int j = 0; j < t.length; j++) {
                    g.setColor(new Color(rand(), rand(), rand()));
                    int iA = t[j].iA, iB = t[j].iB, iC = t[j].iC;
                    int[] x = new int[3], y = new int[3];
                    x[0] = iX(p[iA].x); y[0] = iY(p[iA].y);
                    x[1] = iX(p[iB].x); y[1] = iY(p[iB].y);
                    x[2] = iX(p[iC].x); y[2] = iY(p[iC].y);
                    g.fillPolygon(x, y, 3);
                }
            }
        }
        g.setColor(Color.black);
        super.paint(g);
    }

    int rand() {return (int) (Math.random() * 256);}

    static boolean ccw(Point2D[] p) {
        int n = p.length, k = 0;
        for (int i = 1; i < n; i++)
            if (p[i].x <= p[k].x &&
                (p[i].x < p[k].x || p[i].y < p[k].y))
                k = i;
        // p[k] is a convex vertex.
        int prev = k - 1, next = k + 1;
        if (prev == -1) prev = n - 1;
        if (next == n) next = 0;
        return Tools2D.area2(p[prev], p[k], p[next]) > 0;
    }
}
```

The canvas class CvPolyTria is a subclass of CvDefPoly so that the construction of a polygon with vertices specified by the user is done in the same way as in Sect. 1.4. In this subclass, we call the method triangulate to construct the array t of triangles. These are then displayed in colors generated with a random number generator, so that we can clearly distinguish them, as shown in Fig. 2.15.

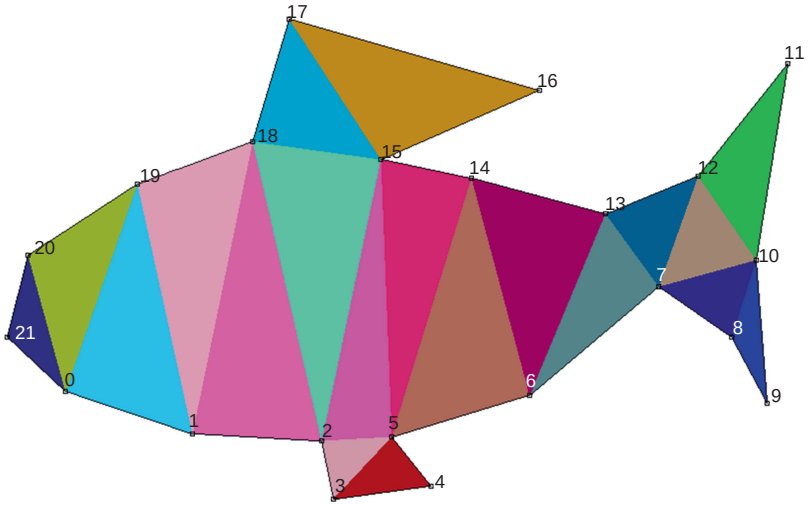

Fig. 2.15 Triangulation of a polygon

2.7 Point-on-Line Test

Testing whether a point P lies on a given line is very simple if this line is given as an equation, say,

$$ax + by = h \qquad (2.3)$$

Then all we need to do is to test whether the coordinates of P satisfy this equation. Due to the discrete nature of the screen (as discussed in Chap. 1), such a test may fail if P is not exactly one of the pixels on the line. It may therefore be wise to be a little tolerant, so that we may write.

```
if (Math.abs(a * p.x + b * p.y - h) < eps) // P on the line
```

where eps is a small positive value, such as 10^{-5}.

If the line is not given by an equation but by two points A and B on it, we can use the above test after deriving an equation for the line, writing

$$\begin{vmatrix} x & y & 1 \\ x_A & y_A & 1 \\ x_B & y_B & 1 \end{vmatrix} = 0$$

which gives the following coefficients for Eq. (2.3):

$$a = y_A - y_B$$
$$b = x_B - x_A$$
$$h = x_B y_A - x_A y_B$$

Instead, we can benefit from the area2 method. After all, if and only if P lies on the line through A and B, the triangle ABP is degenerated and has a zero area. We can therefore write.

```
if (Math.abs(Tools2D.area2(a, b, p)) < eps) // P on line AB
```

A Point on a Line Segment

Given three points A, B and P, we may want to determine whether P lies on the closed line segment AB. The adjective *closed* here means that we include the endpoints A and B, so that the question is to be answered affirmatively if P lies between A and B or coincides with one of these points. We assume that A and B are different points, i.e. $x_A \neq x_B$ or $y_A \neq y_B$. If $x_A \neq x_B$ we test whether x_P lies between x_A and x_B; if not, we test whether y_P lies between y_A and y_B, where in both cases the word *between* includes the points A and B themselves. This test is sufficient if P lies on the infinite line AB. Otherwise, we also have to perform the above test, which is done in the following method:

```
static boolean onSegment(Point2D a, Point2D b, Point2D p) {
    double eps = 1e-2;
    return
    (a.x != b.x &&
        (a.x <= p.x && p.x <= b.x || b.x <= p.x && p.x <= a.x)
    || a.x == b.x &&
    (a.y <= p.y && p.y <= b.y || b.y <= p.y && p.y <= a.y))
    && Math.abs(Tools2D.area2(a, b, p)) < eps;
}
```

The expression following *return* relies on the and-operator && having higher precedence than the or-operator ||. Since both operators && and || evaluate the second operand only if it is necessary, this test is more efficient than it looks. For example, if $x_A \neq x_B$ the test on the line of the form a(y < ...) is not evaluated at all. The positive constant eps = 1e-2 ($= 10^{-2}$) in the above code may be replaced with a smaller or larger one, depending on the application needs.

Instead of testing if P lies on the segment AB, we may want to apply a similar test to the projection P' of P on AB, as Fig. 2.16 shows. We can solve this problem by computing the dot product of the vectors AB and AP. This dot product AB · AP is equal to 0 if $P' = A$ (P' coincides with A) and it is equal to AB · AB = AB^2 if

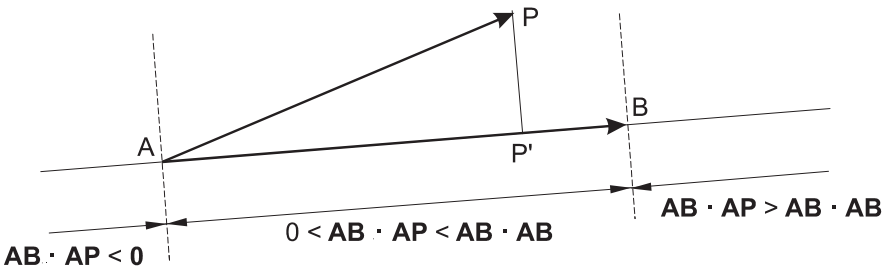

Fig. 2.16 Projection P' of P on line AB between A and B

P' B. For any value of this dot product between these two values, P lies between A and B. We can write this as follows in a program, where len2 corresponds to AB AB, inprod corresponds to AB AP, and eps is some very small positive value:

```
// Does P' (P projected on AB) lie on the closed segment AB?
static boolean projOnSegment(Point2D a, Point2D b, Point2D p) {
    double eps = 1e-2,
        ux = b.x - a.x, uy = b.y - a.y,
        len2 = ux * ux + uy * uy,
        inprod = ux * (p.x - a.x) + uy * (p.y - a.y);
    return inprod > -eps && inprod < len2 + eps;
}
```

To determine whether P' lies on the open segment AB (not including A and B), we replace the return statement with

```
return inprod > eps && inprod < len2 - eps;
```

2.8 Projection of a Point on a Line

Suppose that again a line l and a point P (not on l) are given and that we want to compute the projection P' of P on l (see Fig. 2.16). This point P' has three interesting properties:

1. P' is the point on l that is closest to P.
2. The length of PP' is the distance between P and l (see also Sect. 2.9).
3. PP' and l are perpendicular.

We discuss two solutions: one for a line l given by two points A and B, and the other for l given as the equation $x n = h$.

With given points A and B on line l, the situation is as shown in Fig. 2.16. Recall that in Sect. 2.7 we discussed the method projOnSegment to test if the projection P' of P on the line through A and B lies between A and B. In that method, we did not

actually compute the position of P'. We will now do this (see Fig. 2.16), first by introducing the vector u of length 1 and direction AB:

$$u = \frac{1}{|AB|} AB$$

Then the length of the projection AP' of AP is equal to

$$= AP \cdot u$$

which we use to compute

$$AP' = (AP \cdot u)\, u$$

Doing this straightforwardly would require a square-root operation in the computation of the distance between A and B, used in the computation of u. Fortunately, we can avoid this by rewriting the last equation, using the two preceding ones:

$$AP' = (AP \cdot u)\, u = \left(AP \cdot \frac{1}{|AB|} AB\right) \frac{1}{|AB|} AB = \frac{1}{|AB|^2} (AP \cdot AB)\, AB$$

The advantage of the last form is that the square of the segment length AB is easier to compute than that length itself. The following method, which returns the projection P' of P on AB, demonstrates this:

```
// Compute PÕ (P projected on AB):
static Point2D projection(Point2D a, Point2D b, Point2D p) {
    ßoat vx = b.x - a.x, vy = b.y - a.y, len2 = vx * vx + vy * vy,
       inprod = vx * (p.x - a.x) + vy * (p.y - a.y);
    return new Point2D(a.x + inprod * vx/len2,
                       a.y + inprod * vy/len2);
}
```

Let us now consider a line L given by its equation, which we again write as

$$x \cdot n = h$$

where

$$n = \begin{pmatrix} a \\ b \end{pmatrix} \quad \text{and} \quad x = \begin{pmatrix} x \\ y \end{pmatrix}$$

and

$$\sqrt{a^2 + b^2} = 1$$

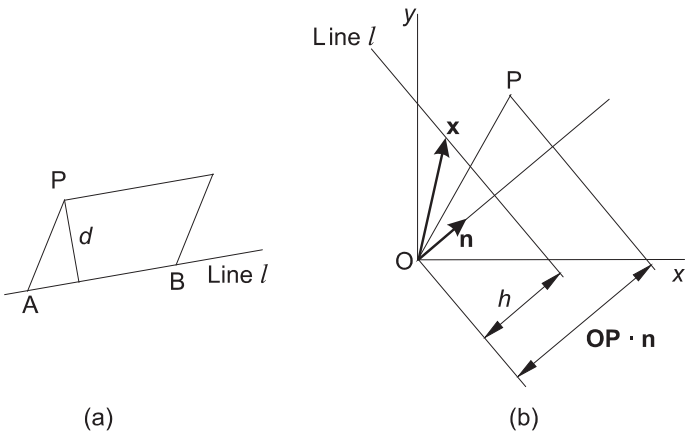

Fig. 2.17 Distance between point P and line l

Using the 'signed distance'

$$d = OP \cdot n - h$$

as illustrated by Fig. 2.17 of the next section, we can write the following vector equation to compute the desired projection P' of P on l:

$$OP' = OP - dn = \begin{matrix} x_P \\ y_P \end{matrix} - d \begin{matrix} a \\ b \end{matrix}$$

This should make the following method clear:

```
// Compute P', the projection of P on line l given as
// ax + by = h, where a * a + b * b = 1
static Point2D projection(float a, float b, float h, Point2D p) {
    float d = p.x * a + p.y * b - h;
    return new Point2D(p.x - d * a, p.y - d * b);
}
```

2.9 Distance Between a Point and a Line

We can find the distance between a point P and a line in different ways, depending on the way the line is specified. If two points A and B of the line are given, we can find the distance between a point P and the (infinite) line through A and B by using the method `Tools2D.area2` defined in Sect. 2.3:

$$\text{distance between P and line AB} = d = \frac{|\text{area2}(A, B, P)|}{|AB|}$$

This follows from the fact that the absolute value of $a_2(A, B, P)$ denotes the area of the parallelogram formed by A, B and P, as shown in Fig. 2.17a. This area is also equal to the product of the parallelogram's base AB and its height d. We can therefore compute d in the above way.

If the line l is given as an equation, we assume this to be in the form

$$ax + by = h$$

where

$$\sqrt{a^2 + b^2} = 1$$

If the latter condition is not satisfied, we only have to divide a, b and h by the above square root. We can then write the above equation of line l as the dot product

$$x \cdot n = h$$

where

$$n = \begin{pmatrix} a \\ b \end{pmatrix} \quad \text{and} \quad x = \begin{pmatrix} x \\ y \end{pmatrix}$$

The normal vector n is perpendicular to line l and has length 1. For any vector x, starting at O, the dot product $x \cdot n$ is the projection of vector x on n. This also applies if the endpoint of x lies on line l, as shown in Fig. 2.17b; in this case we have $x \cdot n = h$. We find the desired distance between point P and line l by projecting OP also on n and computing the difference of the two projections:

$$\text{Distance between P and line } l = |OP \cdot n - h| = |ax_P + by_P - h|$$

Although Fig. 2.17b applies to the case $h > 0$, this equation is also valid if h is negative or zero, or if O lies between line l and the line through P parallel to l. Both $OP \cdot n$ and h are scale factors for the same vector n. The absolute value of the algebraic difference of these two scale factors is the desired distance between P and l.

Exercises

2.1. Write a program that draws a square ABCD. The points A and B are arbitrarily specified by the user by clicking the mouse button. The orientation of the points A, B, C and D should be counter-clockwise.

2.2. Write a program that, for four points A, B, C and P,

- draws a triangle formed by ABC and a small cross showing the position of P; and

¥ displays a line of text indicating which of the following three cases applies: P lies (a) inside ABC, (b) outside ABC, or (c) on an edge of ABC.

The user will specify the four points by clicking.

2.3. The same as Exercise 2.2, but, instead of displaying a line of text, the program computes the distances of P to the (inÞnite) lines AB, BC and CA, and draws the shortest possible line that connects P with the nearest of those three lines.

2.4. Write a program that computes the intersection point of two (inÞnite) lines AB and CD. The user will specify the points A, B, C and D by clicking. Draw a small circle around the intersection point. If the two lines AB and CD do not have a unique intersection point (because they are parallel or coinciding), display a line of text indicating this.

2.5. Write a program that constructs the bisector of the angle ABC (which divides the angle at B into two equal angles). After the user has speciÞed the points A, B and C by clicking, the program should compute the intersection point D of this bisector with the opposite side AC, and draw both triangle ABC and bisector BD.

2.6. Write a program that draws the circumscribed circle (also known as the circumcircle) of a given triangle ABC; this circle passes through the points A, B and C. These points will be speciÞed by the user by clicking the mouse button. Remember, the three perpendicular bisectors of the three edges of a triangle all pass through one point, the *circumcenter*, which is the center of the circumscribed circle.

2.7. Write a program that, for three given points P, Q and R speciÞed by the user, draws a circular arc, starting at P, passing through Q and ending at R (see also Exercise 2.1).

2.8. Construct a Þllet to replace a sharp corner with a rounded one, as illustrated by the solid lines and the arc in Fig. 2.14. The four points A, B, C and D are speciÞed by the user by clicking the mouse. Point D may or may not lie on AB; if it does not, it is projected onto AB, giving D′, as Fig. 2.18 illustrates. The arc starts at point D.

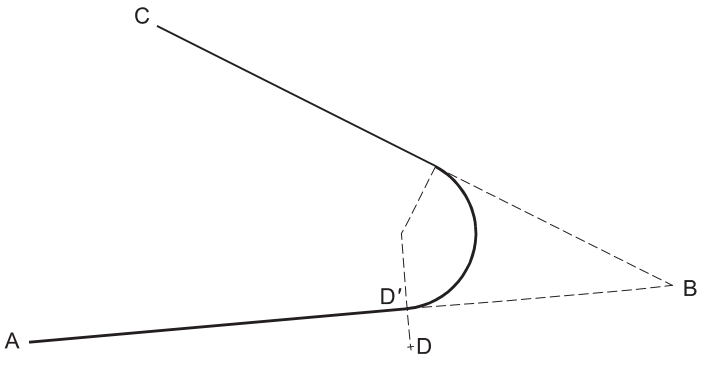

Fig. 2.18 Fillet

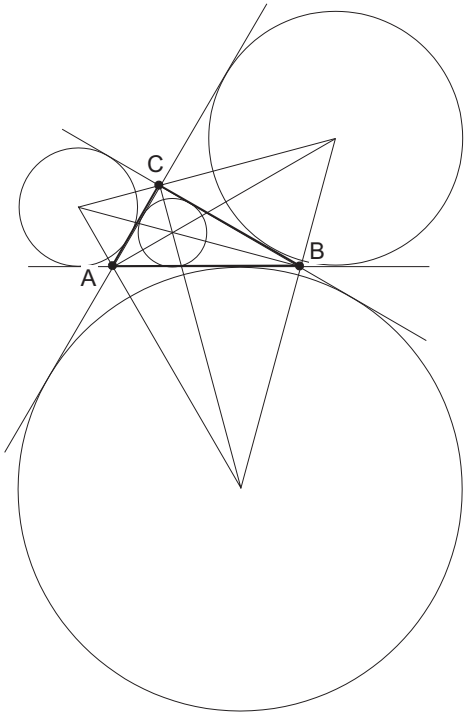

Fig. 2.19 Incircle and excircles of triangle ABC

2.9. Construct the inscribed circle (or incircle) of a given triangle ABC. The center of this circle lies on the point of intersection of the (internal) bisectors of the three angles A, B and C. Draw also the three excircles, which, like the incircle, are tangent to the sides of the triangle, as shown in Fig. 2.19. The centers of the excircles lie on the points of intersection of the external bisectors of the angles A, B and C.

2.10. Write a program that draws a tree of Pythagoras as shown in Fig. 2.20. Two vertices A and B, the basis of the tree, are specified by the user by pressing a mouse button. Then both a square ABCD and an isosceles, right-angled triangle DCE, with right angle E, is constructed. The orientation of both

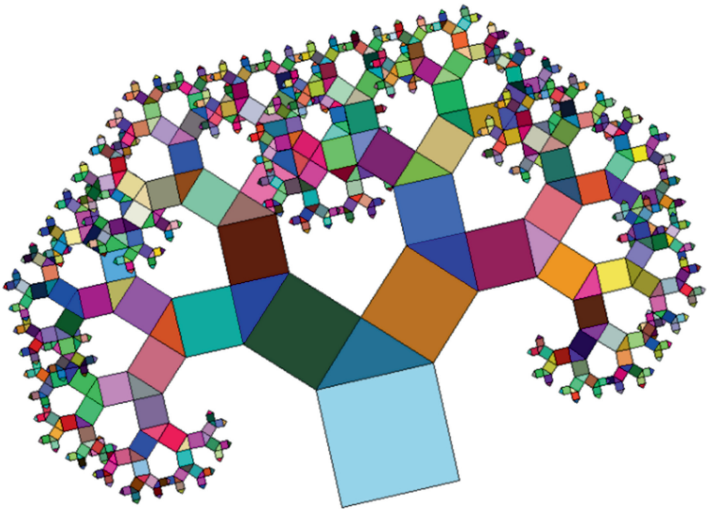

Fig. 2.20 Tree of Pythagoras

ABCD and DCE is counter-clockwise. Finally, the points D and E form the basis of another tree of Pythagoras, and so do the points E and C. Use a recursive method, which does nothing at all if the two supplied basis points are closer together than some limit.