```
    x = p[0].x, y = p[0].y, x0, y0;
    for (int i=1; i<=n; i++)
    {  float t = i * dt;
       x0 = x; y0 = y;
       x = ((cx3 * t + cx2) * t + cx1) * t + cx0;
       y = ((cy3 * t + cy2) * t + cy1) * t + cy0;
       g.drawLine(iX(x0), iY(y0), iX(x), iY(y));
    }
}
```

O cálculo de x e y nesse método é uma aplicação da *regra de Horner*, segundo a qual podemos calcular eficientemente polinômios usando o lado direito, e não o esquerdo, da seguinte equação:

$$a_3t^3 + a_2t^2 + a_1t + a_0 = ((a_3t + a_2)t + a_1)t + a_0$$

Embora *bezier2* não pareça mais simples que *bezier1*, é muito mais eficiente devido ao número reduzido de operações aritméticas no laço *for*. Com um grande número de passos, como $n = 200$ nessas versões de *bezier1* e *bezier2*, é o número de operações dentro do laço que conta, e não as ações preparatórias que precedem o laço.

4.6.3 Curvas 3D

Embora as curvas discutidas aqui sejam bidimensionais, curvas tridimensionais podem ser geradas da mesma forma. Simplesmente adicionamos um componente z a $B(t)$ e aos pontos de controle, que será calculado da mesma forma que os componentes x e y. A possibilidade de gerar curvas que não estejam localizadas em um plano está relacionada com o grau dos polinômios que temos discutido. Se os quatro pontos não estiverem no mesmo plano, o segmento de curva cúbica gerado também não estará. Em contraste, curvas quadráticas são determinadas por apenas três pontos, que definem de forma única um plano (a menos que eles sejam colineares); a curva quadrática através desses três pontos se localiza nesse plano. Em outras palavras, curvas polinomiais podem ser não-planares apenas se forem pelo menos de terceiro grau.

4.7 AJUSTE DE CURVAS B-SPLINE

Além das técnicas discutidas na seção anterior, há outras formas de gerar curvas $x = f(t)$, $y = g(t)$, em que f e g são polinômios de terceiro grau em t. Uma técnica popular, conhecida como *B-splines*, possui a característica de que a curva gerada normalmente não passa pelos pontos dados. Chamaremos todos esses pontos de *pontos de controle*. Um único segmento de tal curva, baseado em quatro pontos de controle A, B, C e D, parece bastante desapontador pois dá a impressão de estar relacionado apenas a B e C. Isso é mostrado na Figura 4.17, na qual, da esquerda para a direita, os pontos A, B, C e D estão marcados novamente com pequenos quadrados.

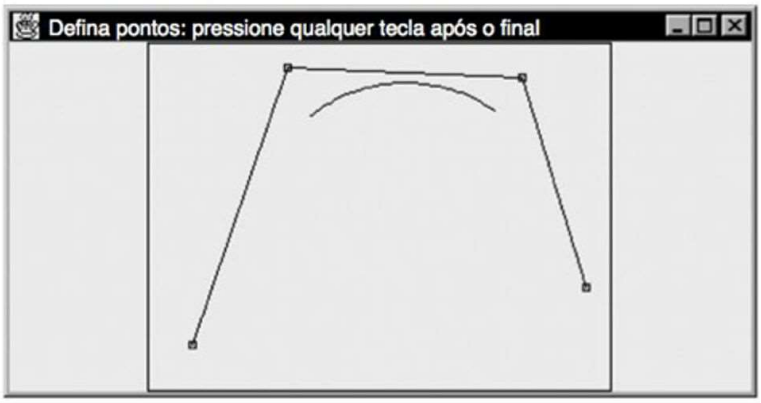

Figura 4.17: Segmento B-spline único, baseado em quatro pontos

Todavia, um ponto forte a favor de B-splines é que essa técnica facilita o desenho de curvas muito suaves que consistem em muitos segmentos de curva. Para evitar confusão, observe que cada segmento de curva consiste em muitos segmentos de reta. Por exemplo, a Figura 4.17 mostra um segmento de curva que consiste em 50 segmentos de retas. Como quatro pontos de controle são necessários para um único segmento de curva, temos

$$\text{número de pontos de controle} = \text{número de segmentos de curva} + 3$$

nesse caso. Essa equação também se aplica se houver diversos segmentos de curva. À primeira vista, a Figura 4.18 parece violar essa regra, já que há cinco segmentos de curva, e parece haver apenas seis pontos de controle. Entretanto, dois deles foram usa-

dos duas vezes. Essa curva foi desenhada clicando-se primeiro no vértice inferior esquerdo (= ponto 0), seguido por um clique no superior esquerdo (= ponto 1), depois no superior direito (= ponto 2) e assim por diante, seguindo o polígono no sentido horário. Os oito pontos de controle foram selecionados na seguinte ordem: 0, 1, 2, 3, 4, 5, 0, 1. Se, após a seleção, uma tecla for pressionada, apenas a curva é redesenhada, e não os pontos de controle nem os segmentos de reta que os conectam.

Se tivéssemos clicado em outro ponto de controle (ponto 2, no alto, à direita), teríamos obtido uma curva fechada. De modo geral, para se produzir uma curva fechada, deve haver três vértices sobrepostos, isto é, dois lados de polígono sobrepostos. Como você pode ver na Figura 4.18, a curva é de fato bastante suave: temos continuidade de segunda ordem, conforme discutido na seção anterior. Lembre-se de que isso implica que até a curvatura é contínua nos pontos em que dois segmentos de curva adjacentes se encontram. Como mostra a parte da curva próxima do vértice inferior direito, podemos tornar a distância entre uma curva e os pontos dados muito pequena ao fornecer diversos pontos próximos uns dos outros.

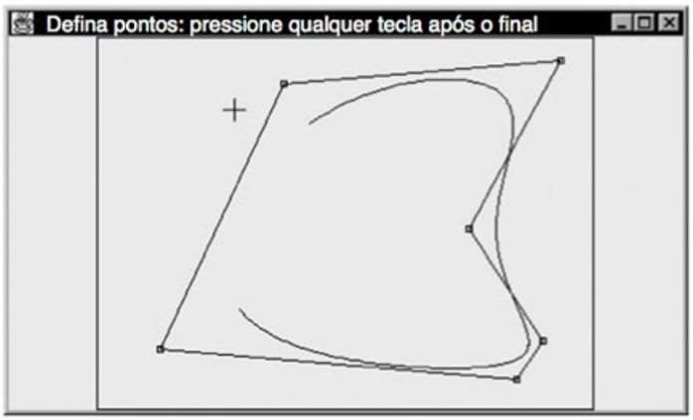

Figura 4.18: Curva B-spline consistindo em cinco segmentos de curva

A matemática de B-splines pode ser expressa pela seguinte equação matricial, semelhante à Equação (4.20):

$$B(t) = \tfrac{1}{6}[\ t^3 \quad t^2 \quad t \quad 1\ ] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{4.21}$$

Se tivermos n pontos de controle $P_0, P_1, ..., P_{n-1}$ ($n \geq 4$), então, estritamente falando, a Equação (4.21) se aplica apenas ao primeiro segmento de curva. Para o segundo, temos que substituir os pontos P_0, P_1, P_2 e P_3 no vetor coluna por P1, P2, P3 e P4, e assim por diante. Da mesma forma que com curvas de Bézier, a variável t varia de 0 a 1 para cada segmento de curva. Multiplicando a matriz 4×4 anterior pelo vetor coluna que a segue, obtemos

$$B(t) = \tfrac{1}{6}[\ t^3 \quad t^2 \quad t \quad 1\ ] \begin{bmatrix} -P_0 + 3P_1 - 3P_2 + P_3 \\ 3P_0 - 6P_1 + 3P_2 \\ -3P_0 + 3P_2 \\ P_0 + 4P_1 + P_2 \end{bmatrix}$$

ou

$$B(t) = \tfrac{1}{6}(-P_0 + 3P_1 - 3P_2 + P_3)t^3 + \tfrac{1}{2}(P_0 - 2P_1 + P_2)t^2 + \tfrac{1}{2}(-P_0 + P_2)t + \tfrac{1}{6}(P_0 + 4P_1 + P_2)$$

O programa a seguir se baseia nessa equação. O usuário pode clicar qualquer número de pontos, que são usados como os pontos $P_0, P_1, ..., P_{n-1}$. O primeiro segmento de curva aparece imediatamente após o quarto ponto de controle, P_3, ter sido definido, e cada ponto de controle adicional faz com que um novo segmento de curva apareça. Para mostrar apenas a curva, o usuário pode pressionar qualquer tecla, o que também termina o processo de entrada. Após isso, podemos gerar outra curva clicando o mouse novamente. As Figuras 4.17 e 4.18 foram produzidas por esse programa:

```
// Bspline.java: Ajuste de curvas B-spline.
// Usa: Point2D (Seção 1.5).

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

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

```
   Bspline()
   {  super("Defina pontos: pressione qualquer tecla após o final");
      addWindowListener(new WindowAdapter()
         {public void windowClosing(WindowEvent e){System.exit(0);}});
      setSize (500, 300);
      add("Center", new CvBspline());

      setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
      show();
   }
}

class CvBspline extends Canvas
{  Vector V = new Vector();
   int np = 0, centerX, centerY;
   float rWidth = 10.0F, rHeight = 7.5F, eps = rWidth/100F, pixelSize;
   boolean ready = false;

   CvBspline()
   {  addMouseListener(new MouseAdapter()
      {  public void mousePressed(MouseEvent evt)
         {  float x = fx(evt.getX()), y = fy(evt.getY());
            if (ready)
            {  V.removeAllElements();
               np = 0;
               ready = false;
            }
            V.addElement(new Point2D(x, y));
            np++;
            repaint();
         }
      });

      addKeyListener(new KeyAdapter()
      {  public void keyTyped(KeyEvent evt)
         {  evt.getKeyChar();
            if (np >= 4) ready = true;
            repaint();
         }
      });
   }

   void initgr()
   {  Dimension d = getSize();
      int maxX = d.width - 1, maxY = d.height - 1;
      pixelSize = Math.max(rWidth/maxX, rHeight/maxY);
      centerX = maxX/2; centerY = maxY/2;
   }
   int iX(float x){return Math.round(centerX + x/pixelSize);}
   int iY(float y){return Math.round(centerY - y/pixelSize);}
   float fx(int x){return (x - centerX) * pixelSize;}
   float fy(int y){return (centerY - y) * pixelSize;}

   void bspline(Graphics g, Point2D[] p)
   {  int m = 50, n = p.length;
      float xA, yA, xB, yB, xC, yC, xD, yD,
         a0, a1, a2, a3, b0, b1, b2, b3, x=0, y=0, x0, y0;
      boolean first = true;
      for (int i=1; i<n-2; i++)
      {  xA=p[i-1].x; xB=p[i].x; xC=p[i+1].x; xD=p[i+2].x;
         yA=p[i-1].y; yB=p[i].y; yC=p[i+1].y; yD=p[i+2].y;
         a3=(-xA+3*(xB-xC)+xD)/6; b3=(-yA+3*(yB-yC)+yD)/6;
         a2=(xA-2*xB+xC)/2;       b2=(yA-2*yB+yC)/2;
         a1=(xC-xA)/2;            b1=(yC-yA)/2;
         a0=(xA+4*xB+xC)/6;       b0=(yA+4*yB+yC)/6;
         for (int j=0; j<=m; j++)
         {  x0 = x;
            y0 = y;
            float t = (float)j/(float)m;
            x = ((a3*t+a2)*t+a1)*t+a0;
            y = ((b3*t+b2)*t+b1)*t+b0;
            if (first) first = false;
            else
               g.drawLine(iX(x0), iY(y0), iX(x), iY(y));
         }
      }
   }
```

```
    public void paint(Graphics g)
    {  initgr();
       int left = iX(-rWidth/2), right = iX(rWidth/2),
           bottom = iY(-rHeight/2), top = iY(rHeight/2);
       g.drawRect(left, top, right - left, bottom - top);
       Point2D[] p = new Point2D[np];
       V.copyInto(p);
       if (!ready)
       {  for (int i=0; i<np; i++)
          {  // Mostra pequeno retângulo em torno do ponto:
             g.drawRect(iX(p[i].x)-2, iY(p[i].y)-2, 4, 4);
             if (i > 0)
                // Desenha reta p[i-1]p[i]:
                g.drawLine(iX(p[i-1].x), iY(p[i-1].y),
                           iX(p[i].x), iY(p[i].y));
          }
       }
       if (np >= 4) bspline(g, p);
    }
}
```

Para ver por que B-splines são tão suaves, você deve derivar $B(t)$ duas vezes e verificar que, para qualquer segmento que não seja o final, os valores de $B(1)$, $B'(1)$ e $B''(1)$ no ponto final desses segmentos são iguais aos valores $B(0)$, $B'(0)$ e $B''(0)$ no ponto inicial do próximo segmento de curva. Por exemplo, para a continuidade da própria curva, encontramos

$$\begin{aligned} B(1) &= \tfrac{1}{6}(-P_0 + 3P_1 - 3P_2 + P_3) + \tfrac{1}{2}(P_0 - 2P_1 + P_2) + \tfrac{1}{2}(-P_0 + P_2) + \tfrac{1}{6}(P_0 + 4P_1 + P_2) \\ &= \tfrac{1}{6}(P_1 + 4P_2 + P_3) \end{aligned}$$

para o primeiro segmento, baseado em P_0, P_1, P_2 e P_3, enquanto podemos ver imediatamente que obtemos exatamente esse valor se calcularmos $B(0)$ para o segundo segmento de curva, baseado em P_1, P_2, P_3 e P_4.

EXERCÍCIOS

4.1 Substitua o método *drawLine* baseado no algoritmo de Bresenham e listado próximo ao final da Seção 4.1 por uma versão ainda mais rápida que se beneficie da simetria das duas metades do segmento de reta. Por exemplo, com pontos extremos P e Q satisfazendo a Equação (4.1) e usando o valor inteiro x_{Mid} exatamente no meio entre x_P e x_Q, podemos deixar que a variável x varie de x_P a x_{Mid} e, ao mesmo tempo, usar uma variável x_2, que varie de x_Q a x_{Mid}. Em cada iteração do laço, x é incrementado em 1 e x_2 é decrementado em 1. Observe que haverá um ou dois pontos no meio do segmento de reta, dependendo do número de pixels a serem colocados ser par ou ímpar. Assegure-se de que nenhum pixel do segmento de reta seja omitido ou colocado duas vezes na tela. Para testar essa última situação, você pode usar o modo XOR para que escrever o mesmo pixel duas vezes tenha o mesmo efeito que omitir o pixel.

4.2 Generalize o método *doubleStep2* (da Seção 4.2), para fazê-lo funcionar com qualquer reta.

4.3 Como pixels normais são muito pequenos, eles não mostram muito claramente quais deles são selecionados pelos algoritmos de Bresenham. Use uma grade de pontos para simular uma tela com resolução muito baixa e demonstrar tanto o método *drawLine* da Seção 4.1 (com *g* como seu primeiro argumento) quanto o método *drawCircle* da Seção 4.3. Apenas os pontos da grade devem ser usados como centros dos 'superpixels'. Codifique um novo método *putPixel* para desenhar um pequeno círculo com tal centro, tendo como diâmetro a distância *dGrid* entre dois pontos vizinhos da grade.

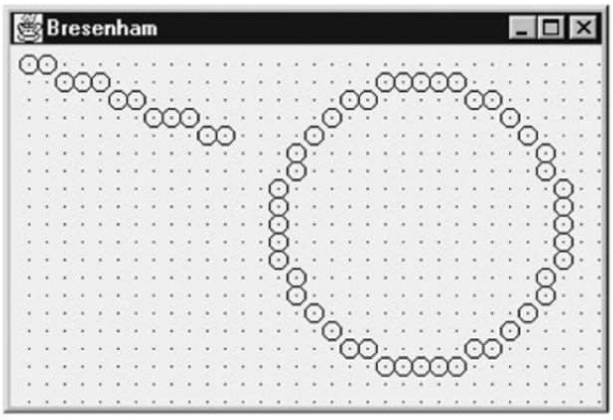

Figura 4.19: Algoritmos de Bresenham para uma reta e para um círculo (veja também as Figuras 4.1 e 4.6)