Rozdział 13

Krzywa Béziera i inne krzywe składane

Listingi z tego rozdziału znajdują się na CD-ROM-ie w katalogu: Béziers and Other Splines.

Co to jest krzywa składana (ang. *spline*)? * – a więc „krzywik". Ta definicja przywodzi na myśl obraz inżyniera, który trzyma w ręku niewygodne narzędzie i – zgarbiony nad arkuszem papieru milimetrowego – próbuje dopasować krzywą do rozproszonego zbioru punktów. Obecnie lepszą definicją tego słowa jest „krzywa obliczana przez funkcję matematyczną, która gładko łączy odrębne punkty... *Zobacz także* krzywa Béziera"**.

Pierre Etienne Bézier urodził się w Paryżu w 1910 roku w rodzinie inżynierów. W 1930 roku uzyskał dyplom inżyniera mechanika, a rok później – dyplom inżyniera elektryka. W 1933 roku zaczął pracować we francuskiej firmie samochodowej Renault, gdzie pozostał do 1975 roku. W latach pięćdziesiątych Bézier był odpowiedzialny za wdrożenia pierwszych maszyn do wiercenia i frezowania, które działały pod kontrolą NC, czyli sterowania numerycznego (ang. *numerical control* – dziś rzadko używa się tego terminu).

Od roku 1960 praca Béziera była związana głównie z programem UNISURF, wczesnym systemem CAD/CAM używanym w Renault do interaktywnego projektowania części samochodowych. System ten wymagał matematycznej definicji skomplikowanych krzywych, którymi projektanci mogliby manipulować bez odwoływania się do wzorów i które można by wykorzystać w procesach produkcyjnych. Z tych prac narodziła się krzywa, która dziś nosi nazwisko Béziera. Pierre Bézier zmarł w 1999 roku.***

Krzywa Béziera zyskała duże znaczenie w grafice komputerowej i plasuje się tuż za linią prostą oraz łukiem eliptycznym. W PostScripcie krzywa Béziera opisuje *wszystkie* krzywe – aproksymuje nawet łuki eliptyczne. Krzywe Béziera służą

---

* Angielskie słowniki definiują słowo *spline* jako „giętki kawałek drzewa, twardej gumy lub metalu używany do rysowania krzywych" *American Heritage Dictionary of the English Language*, 4th ed. (Boston: Houghton Mifflin, 2000).

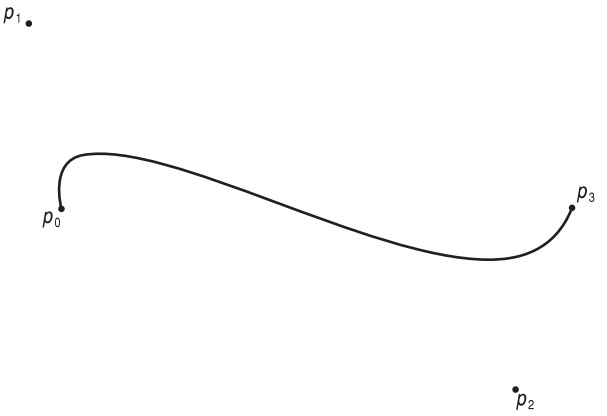
** *Microsoft Computer Dictionary*, 4th ed. (Redmond, WA: Microsoft Press, 1999).

*** Większość informacji biograficznych zaczerpnięto z artykułu Pierra Béziera „Style, Mathematics and NC", *Computer-Aided Design* 22, nr 9 (listopad 1990): 523. Na angielski przełożono dwie książki Béziera: Pierre Bézier, *Numerical Control: Mathematics and Application* (Londyn: John Wiley & Sons, 1972) oraz *The Mathematical Basis of the UNISURF CAD System* (Londyn: Butterworths, 1986). Zobacz także Pierre Bézier, „How a Simple System Was Born" w Gerald Farin, *Curves and Surfaces for Computer-Aided Geometric Design: A Practical Guide*, 4th ed. (San Diego: Academic Press, 1997).

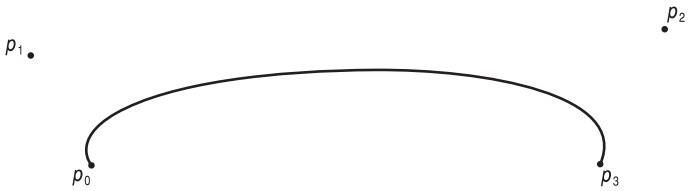
również do definiowania obrysów czcionek postscriptowych (czcionki TrueType używają prostszego i szybszego rodzaju krzywej).

Krzywa Béziera w praktyce

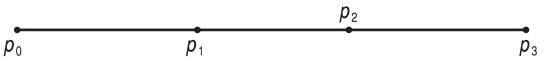
Pojedynczą krzywą Béziera jednoznacznie identyfikują cztery punkty, które nazwiemy p_0, p_1, p_2 i p_3. Krzywa zaczyna się w punkcie p_0 i kończy w p_3; p_0 jest zatem punktem *początkowym*, a p_3 – punktem *końcowym* (zbiorczo punkty p_0 i p_3 są często nazywane końcowymi). Punkty p_1 i p_2 to punkty *kontrolne*. Punkty kontrolne działają jak magnesy i przyciągają do siebie krzywą. Oto przykładowa krzywa Béziera z dwoma punktami końcowymi i dwoma kontrolnymi:



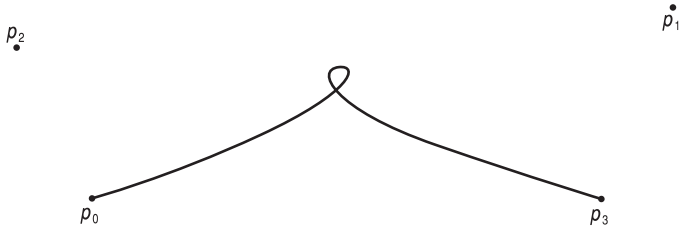
Zauważ, że krzywa zaczyna się w punkcie p_0 i zmierza w stronę p_1, ale potem porzuca ten szlak i kieruje się w stronę p_2. Nie dotykając p_2, kończy się w punkcie p_3. Oto inna krzywa Béziera:



Rzadko się zdarza, żeby krzywa Béziera przechodziła przez punkty kontrolne. Jeśli jednak umieścisz oba punkty kontrolne między punktami końcowymi, krzywa Béziera zmieni się w linię prostą i będzie przechodzić przez oba punkty:



Można też wybrać punkty, które zawiną krzywą Béziera w pętlę:



Aby narysować krzywą Béziera w Windows Forms, musisz określić cztery punkty: albo jako cztery struktury *Point* lub *PointF*, albo jako osiem wartości typu *float*:

Metody *DrawBezier* klasy *Graphics*

```
void DrawBezier(Pen pen, Point pt0, Point pt1, Point pt2, Point pt3)
void DrawBezier(Pen pen, PointF ptf0, PointF ptf1, PointF ptf2,
                         PointF ptF3)
void DrawBezier(Pen pen, float x0, float y0, float x1, float y1,
                         float x2, float y2, float x3, float y3)
```

Czasem wygodniej jest określić cztery punkty jako tablicę struktur *Point* lub *PointF*. Pozwalają na to dwie wersje metody *DrawBeziers* (zwróć uwagę na liczbę mnogą). Możesz przekazać do metody *DrawBeziers* tablicę czterech struktur *Point* lub *PointF* albo użyć tej metody do rysowania wielu połączonych krzywych Béziera:

Metody *DrawBeziers* klasy *Graphics*

```
void DrawBeziers(Pen pen, Point[] apt)
void DrawBeziers(Pen pen, PointF[] aptf)
```

Gdy rysujesz wiele krzywych Béziera, punkt końcowy jednej krzywej staje się punktem początkowym następnej, co oznacza, że każda dodatkowa krzywa wymaga określenia trzech punktów. Jeśli chcesz narysować *N* krzywych Béziera, liczba punktów w tablicy musi być równa $3N + 1$. Gdy rozmiar tablicy jest inny niż $3N + 1$, dla $N \quad 1$, metoda zgłasza wyjątek.

Nie ma metod *FillBezier* ani *FillBeziers*. Jeśli chcesz użyć krzywych Béziera do wypełniania zamkniętych obszarów, musisz skorzystać ze ścieżek graficznych, które omawiam w rozdziale 15.

Żeby zyskać wprawę w manipulowaniu krzywymi Béziera, poeksperymentuj z poniższym programem:

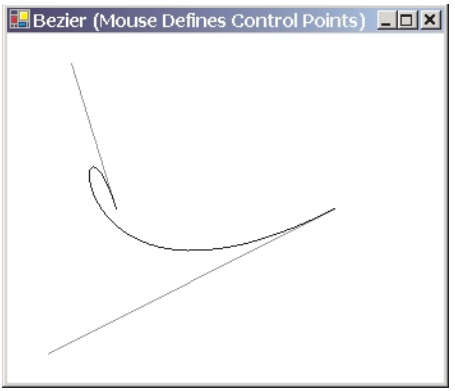
Listing Bezier.cs

```
//-------------------------------------
// Bezier.cs © 2001 by Charles Petzold
//-------------------------------------
using System;
using System.Drawing;
```

```
using System.Windows.Forms;

class Bezier: Form
{
     protected Point[] apt = new Point[4];

     public static void Main()
     {
          Application.Run(new Bezier());
     }
     public Bezier()
     {
          Text = "Bezier (Mouse Defines Control Points)";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          ResizeRedraw = true;

          OnResize(EventArgs.Empty);
     }
     protected override void OnResize(EventArgs ea)
     {
          base.OnResize(ea);

          int cx = ClientSize.Width;
          int cy = ClientSize.Height;

          apt[0] = new Point(    cx / 4,     cy / 2);
          apt[1] = new Point(    cx / 2,     cy / 4);
          apt[2] = new Point(    cx / 2, 3 * cy / 4);
          apt[3] = new Point(3 * cx / 4,     cy / 2);
     }
     protected override void OnMouseDown(MouseEventArgs mea)
     {
          Point pt;

          if (mea.Button == MouseButtons.Left)
               pt = apt[1];

          else if (mea.Button == MouseButtons.Right)
               pt = apt[2];

          else
               return;

          Cursor.Position = PointToScreen(pt);
     }
     protected override void OnMouseMove(MouseEventArgs mea)
     {
          if (mea.Button == MouseButtons.Left)
          {
               apt[1] = new Point(mea.X, mea.Y);
               Invalidate();
          }
          else if (mea.Button == MouseButtons.Right)
          {
               apt[2] = new Point(mea.X, mea.Y);
               Invalidate();
          }
     }
     protected override void OnPaint(PaintEventArgs pea)
```

```
        {
            Graphics grfx = pea.Graphics;

            grfx.DrawBeziers(new Pen(ForeColor), apt);

            Pen pen = new Pen(Color.FromArgb(0x80, ForeColor));

            grfx.DrawLine(pen, apt[0], apt[1]);
            grfx.DrawLine(pen, apt[2], apt[3]);
        }
}
```

Położenie punktów końcowych jest stałe, a dwoma punktami kontrolnymi można sterować za pomocą myszy. Lewy przycisk myszy kontroluje punkt p_1, a prawy – punkt p_2. W programie zrealizowałem funkcję „przyciągania": kiedy naciśniesz lewy lub prawy przycisk myszy, program używa statycznej właściwości *Cursor.Position*, żeby przesunąć wskaźnik myszy do odpowiedniego punktu kontrolnego. Program rysuje także szare linie od punktów końcowych do kontrolnych. Oto typowe wyniki:



Krzywe Béziera mają kilka cech, które predestynują je do zastosowania w projektowaniu wspomaganym komputerowo. Po pierwsze, przy odrobinie wprawy zwykle da się dopasować krzywą do żądanego kształtu.

Po drugie, krzywa Béziera jest bardzo dobrze kontrolowana. Niektóre krzywe nie przechodzą przez żaden z definiujących je punktów. Krzywa Béziera jest zawsze zakotwiczona w dwóch punktach końcowych (jak się niebawem przekonamy, jest to jedno z założeń używanych podczas wyprowadzania wzorów Béziera). Co więcej, pewne rodzaje krzywych mają osobliwości, w których krzywa ucieka do nieskończoności (co w komputerowym projektowaniu jest raczej niepożądane). Krzywa Béziera jest znacznie bardziej zdyscyplinowana. Zawsze ogranicza ją czworokąt (nazywany *powłoką wypukłą*) utworzony przez połączenie punktów końcowych i kontrolnych. (Sposób łączenia punktów w celu utworzenia powłoki wypukłej zależy od konkretnej krzywej).

Trzecia właściwość krzywej Béziera dotyczy związku między punktami końcowymi i kontrolnymi. W punkcie początkowym krzywa jest zawsze styczna do linii prostej biegnącej od punkt początkowego do pierwszego punktu kontrolnego i ma ten sam kierunek (tę zależność ilustruje graficznie nasz przykładowy program). W punkcie końcowym krzywa jest zawsze styczna do linii prostej biegnącej od drugiego punktu kontrolnego do punktu końcowego i ma ten sam kierunek. To kolejne dwa założenia używane do wyprowadzania wzorów Béziera.

Po czwarte, krzywe Béziera często wywołują pozytywne wrażenia estetyczne. Wiem, że to dość subiektywne kryterium, ale nie jestem jedyną osobą, która uważa krzywe Béziera za pełne wdzięku.

Bardziej stylowy zegar

Minęły dwie dekady od napisania pierwszego programu zegara analogowego, a programy tego typu wciąż wyglądają prawie identycznie. Niemal zawsze programista używa prostych wielokątów do rysowania wskazówek zegara. Pora zbadać nowe perspektywy, rysując wskazówki zegara za pomocą krzywych Béziera.

Jak pamiętasz, program AnalogClock z rozdziału 10 – „Zegar i czas" używał kontrolki, którą zrealizowałem w klasie *ClockControl*. Na szczęście byłem dość przewidujący, aby wyodrębnić rysujący wskazówki kod w chronionych wirtualnych metodach tej klasy. Oto klasa *BezierClockControl*, która wywołuje metody *DrawBeziers* w nowych metodach *DrawHourHand* i *DrawMinuteHand*.

Listing BezierClockControl.cs

```
//-----------------------------------------------
// BezierClockControl.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

namespace Petzold.ProgrammingWindowsWithCSharp
{
class BezierClockControl: ClockControl
{
     protected override void DrawHourHand(Graphics grfx, Pen pen)
     {
          GraphicsState gs = grfx.Save();
          grfx.RotateTransform(360f * Time.Hour / 12 +
                               30f * Time.Minute / 60);

          grfx.DrawBeziers(pen, new Point[]
          {
               new Point(  0, -600), new Point(   0, -300),
               new Point(200, -300), new Point(  50, -200),
               new Point( 50, -200), new Point(  50,    0),
               new Point( 50,    0), new Point(  50,   75),
               new Point(-50,   75), new Point( -50,    0),
```

```
                new Point(-50,    0), new Point( -50, -200),
                new Point(-50, -200), new Point(-200, -300),
                new Point(  0, -300), new Point(   0, -600)
           });
           grfx.Restore(gs);
     }
     protected override void DrawMinuteHand(Graphics grfx, Pen pen)
     {
           GraphicsState gs = grfx.Save();
           grfx.RotateTransform(360f * Time.Minute / 60 +
                                6f * Time.Second / 60);

           grfx.DrawBeziers(pen, new Point[]
           {
                new Point(  0, -800), new Point(   0, -750),
                new Point(  0, -700), new Point(  25, -600),
                new Point( 25, -600), new Point(  25,    0),
                new Point( 25,    0), new Point(  25,   50),
                new Point(-25,   50), new Point( -25,    0),
                new Point(-25,    0), new Point( -25, -600),
                new Point(-25, -600), new Point(   0, -700),
                new Point(  0, -750), new Point(   0, -800)
           });
           grfx.Restore(gs);
     }
}
}
```

W obu wywołaniach *DrawBeziers* przekazywana jest tablica szesnastu struktur *Point* w celu narysowania pięciu krzywych Béziera. (Pamiętaj, że pierwsza krzywa Béziera rysowana przez *DrawBeziers* wymaga czterech punktów, a każda kolejna – dalszych trzech).

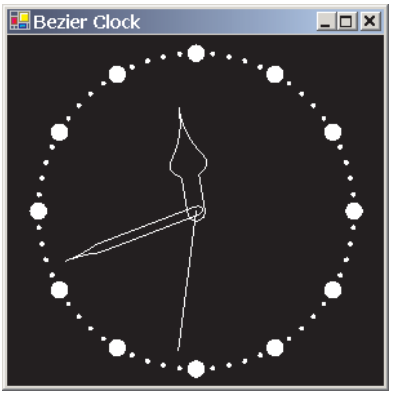
Pierwotny program AnalogClock był bardzo mały, więc zdecydowałem, że nie ma sensu wyprowadzać z niego podklasy. Oto zupełnie nowy program BezierClock, który wykorzystuje klasę *BezierClockControl*.

Listing BezierClock.cs

```
//----------------------------------------
// BezierClock.cs © 2001 by Charles Petzold
//----------------------------------------
using Petzold.ProgrammingWindowsWithCSharp;
using System;
using System.Drawing;
using System.Windows.Forms;

class BezierClock: Form
{
     BezierClockControl clkctl;

     public static void Main()
     {
          Application.Run(new BezierClock());
     }
     public BezierClock()
     {
          Text = "Bezier Clock";
```

```
            clkctl = new BezierClockControl();
            clkctl.Parent    = this;
            clkctl.Time      = DateTime.Now;
            clkctl.Dock      = DockStyle.Fill;
            clkctl.BackColor = Color.Black;
            clkctl.ForeColor = Color.White;

            Timer timer = new Timer();
            timer.Interval = 100;
            timer.Tick += new EventHandler(OnTimerTick);
            timer.Start();
        }
        void OnTimerTick(object obj, EventArgs ea)
        {
            clkctl.Time = DateTime.Now;
        }
}
```

A oto program w działaniu:



Zakrzywione zakończenie wskazówki składa się z dwóch krzywych Béziera, jednej po każdej stronie. Proste części wskazówki to kolejna para krzywych Béziera, a zaokrąglona część w środku to jeszcze jedna krzywa – co w sumie daje pięć.

Współliniowe krzywe Béziera

Choć połączone krzywe Béziera współdzielą punkty końcowe, jest możliwe, że punkt, w którym jedna krzywa się kończy, a druga zaczyna, nie będzie gładki. W języku matematyki złożoną krzywą nazywa się gładką tylko wtedy, gdy jej pierwsza pochodna jest ciągła – to znaczy nie zachodzą w niej żadne nagłe zmiany.

Gdy rysujesz wiele krzywych Béziera, możesz sobie życzyć, żeby złożona krzywa była gładka w punktach, w których zaczynają się i kończą poszczególne odcinki – ale niekoniecznie. We wskazówkach zegara potrzebna jest kombinacja gładkości i nieciągłości. W szczytowym punkcie, w którym zbiegają się dwie krzywe Béziera, pierwsza pochodna jest nieciągła. Podobną nieciągłość obserwu-

jemy tam, gdzie zakrzywiona część wskazówki spotyka się z linią prostą. Jednakże linie proste gładko łączą się z zaokrągloną częścią na środku zegara.

Jeżeli chcesz, żeby krzywe Béziera były gładko połączone, poniższe trzy punkty muszą być współliniowe (tzn. leżeć na tej samej prostej):

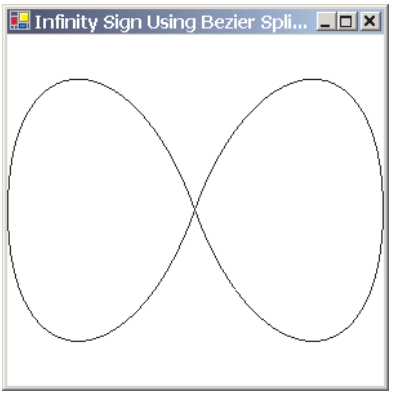
- drugi punkt kontrolny pierwszej krzywej Béziera;
- punkt końcowy pierwszej krzywej Béziera (zarazem będący punktem początkowym drugiej krzywej Béziera);
- pierwszy punkt kontrolny drugiej krzywej Béziera.

Oto program rysujący cztery połączone krzywe Béziera, które są gładkie w każdym punkcie łączącym. Koniec czwartej krzywej pokrywa się z początkiem pierwszej, tworząc krzywą zamkniętą.

Listing Infinity.cs

```
//---------------------------------------
// Infinity.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class Infinity: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new Infinity());
     }
     public Infinity()
     {
          Text = "Infinity Sign Using Bezier Splines";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          cx--;
          cy--;

          Point[] apt =
          {
               new Point(0,          cy / 2),     // Begin
               new Point(0,          0),          // Control
               new Point(    cx / 3, 0),          // Control
               new Point(    cx / 2, cy / 2),     // End/Begin
               new Point(2 * cx / 3, cy),         // Control
               new Point(    cx,     cy),         // Control
               new Point(    cx,     cy / 2),     // End/Begin
               new Point(    cx,     0),          // Control
               new Point(2 * cx / 3, 0),          // Control
               new Point(    cx / 2, cy / 2),     // End/Begin
               new Point(    cx / 3, cy),         // Control
               new Point(0,          cy),         // Control
               new Point(0,          cy / 2)      // End
          };
          grfx.DrawBeziers(new Pen(clr), apt);
     }
}
```

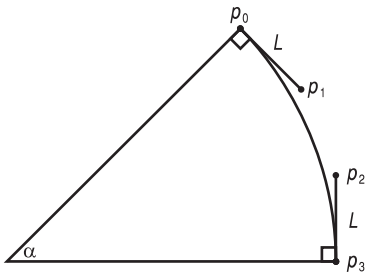
W powyższej tablicy każdy punkt oznaczony komentarzem Begin, End, albo End/Begin jest współliniowy z punktami kontrolnymi po obu stronach. Cztery krzywe tworzą kształt, który nieco przypomina znak nieskończoności:



Rysowanie okręgów i łuków za pomocą krzywych Béziera

Jak wspomniałem już w tym rozdziale, PostScript używa krzywych Béziera do rysowania łuków eliptycznych. W rozdziale 15 dowiesz się, że tak samo postępuje Windows Forms, przynajmniej kiedy trzeba przechować łuki i elipsy na ścieżce graficznej.

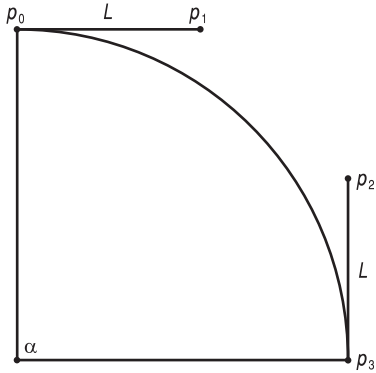
Dostępnych jest kilka artykułów opisujących przybliżanie łuków eliptycznych za pomocą krzywych Béziera*. Pierwszy artykuł opisuje dość prostą technikę, której można użyć do rysowania fragmentów okręgu. Przypuśćmy, że chcesz użyć krzywej Béziera do narysowania łuku kołowego o określonym promieniu i kącie . Wiesz, że musisz ustawić p_0 i p_3 na początek i koniec łuku, ale jak ustawić p_1 i p_2? Jak pokazuje poniższy wykres, problem sprowadza się do wyznaczenia odległości między punktami końcowymi a kontrolnymi – tę odległość oznaczymy literą *L*:



* Tor Dokken, et al., „Good Approximation of Circle by Curvature-Continuous Bézier Curves", *Computer Aided Geometric Design* 7 (1990), 33-41. Michael Goldap, „Approximation of Circular Arcs by Cubic Polynomials", *Computer Aided Geometric Design* 8 (1991), 227-238.

Na rysunku zaznaczyłem, że linie łączące punkty końcowe z kontrolnymi tworzą kąt prosty z promieniami. Skąd to wiemy? Ponieważ do zachowania gładkości niezbędna jest współliniowość. Gdybyś chciał użyć krzywej Béziera do narysowania kolejnego, przyległego łuku o tym samym środku i promieniu, wspólny punkt końcowy i dwa przyległe punkty kontrolne musiałyby być współliniowe. Oznacza to, że linia biegnąca od punktu końcowego do punktu kontrolnego musi tworzyć kąt prosty z promieniem okręgu.

Jeśli znasz L, do obliczenia współrzędnych p_1 i p_2 wystarczy podstawowa trygonometria. Spójrz jednak, jak łatwe stają się obliczenia p_1 i p_2, gdy używasz kąta 90 stopni wyrównanego do współrzędnych poziomych i pionowych:



Obliczenia p_1 i p_2 są banalnie proste także wtedy, gdy używasz kąta 180 stopni.

Pierwszy zacytowany przeze mnie artykuł dowodzi, że dobre przybliżenie można uzyskać, mnożąc

$$L \quad \frac{4}{3} tg \quad \frac{1}{4}$$

przez promień okręgu.

Program BezierCircles rysuje dwa pełne okręgi, używając tego przybliżenia, najpierw za pomocą dwóch krzywych Béziera, a potem (dokładniej) – za pomocą czterech krzywych.

Listing BezierCircles.cs

```
//-----------------------------------------
// BezierCircles.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BezierCircles: PrintableForm
{
     public new static void Main()
     {
```

```
            Application.Run(new BezierCircles());
        }
        public BezierCircles()
        {
            Text = "Bezier Circles";
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            int iRadius = Math.Min(cx - 1, cy - 1) / 2;

            grfx.DrawEllipse(new Pen(clr), cx / 2 - iRadius, cy / 2 - iRadius,
                                           2 * iRadius, 2 * iRadius);

                // Two-segment (180-degree) approximation

            int L = (int) Math.Round(iRadius * 4f / 3 * Math.Tan(Math.PI / 4));

            Point[] apt = {
                              new Point(cx / 2,     cy / 2 - iRadius),
                              new Point(cx / 2 + L, cy / 2 - iRadius),
                              new Point(cx / 2 + L, cy / 2 + iRadius),
                              new Point(cx / 2,     cy / 2 + iRadius),
                              new Point(cx / 2 - L, cy / 2 + iRadius),
                              new Point(cx / 2 - L, cy / 2 - iRadius),
                              new Point(cx / 2,     cy / 2 - iRadius)
                          };
            grfx.DrawBeziers(Pens.Blue, apt);

                // Four-segment (90-degree) approximation

            L = (int) Math.Round(iRadius * 4f / 3 * Math.Tan(Math.PI / 8));

            apt = new Point[]
                        {
                            new Point(cx / 2,           cy / 2 - iRadius),
                            new Point(cx / 2 + L,       cy / 2 - iRadius),
                            new Point(cx / 2 + iRadius, cy / 2 - L),
                            new Point(cx / 2 + iRadius, cy / 2),
                            new Point(cx / 2 + iRadius, cy / 2 + L),
                            new Point(cx / 2 + L,       cy / 2 + iRadius),
                            new Point(cx / 2,           cy / 2 + iRadius),
                            new Point(cx / 2 - L,       cy / 2 + iRadius),
                            new Point(cx / 2 - iRadius, cy / 2 + L),
                            new Point(cx / 2 - iRadius, cy / 2),
                            new Point(cx / 2 - iRadius, cy / 2 - L),
                            new Point(cx / 2 - L,       cy / 2 - iRadius),
                            new Point(cx / 2,           cy / 2 - iRadius)
                        };
            grfx.DrawBeziers(Pens.Red, apt);
        }
}
```

Program demonstruje też, jak przybliżenie Béziera różni się od metody *DrawEllipse*. Program zaczyna przetwarzanie *DoPage* od wywołania metody *DrawEllipse* w celu narysowania czarnej elipsy. Przybliżenie z dwoma krzywymi Béziera jest rysowane na niebiesko, a wersja z czterema krzywymi – na czerwono. Pamiętaj, że argumenty funkcji trygonometrycznych z klasy *Math* są wyrażone w radianach, więc zamiast dzielić kąt przez 4 zgodnie z wzorem na *L*, używam wyrażenia opartego na stałej *Math.PI*.

Sztuka Béziera

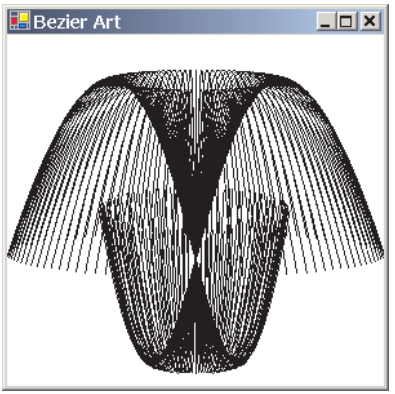
Wiele osób – w tym sam Pierre Bézier* – używało krzywych Béziera do tworzenia interesujących wzorów i deseni. Tego rodzaju prace zwykle określa się mianem „sztuki Béziera". Nie ma tu żadnych reguł, choć zazwyczaj korzysta się z pętli *for*. Oto przykład:

Listing BezierArt.cs

```
//---------------------------------------
// BezierArt.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BezierArt: PrintableForm
{
     const int iNum = 100;

     public new static void Main()
     {
          Application.Run(new BezierArt());
     }
     public BezierArt()
     {
          Text = "Bezier Art";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Pen      pen  = new Pen(clr);
          PointF[] aptf = new PointF[4];

          for (int i = 0; i < iNum; i++)
          {
               double dAngle = 2 * i * Math.PI / iNum;

               aptf[0].X =     cx / 2 + cx /  2 * (float) Math.Cos(dAngle);
               aptf[0].Y = 5 * cy / 8 + cy / 16 * (float) Math.Sin(dAngle);

               aptf[1] = new PointF(cx / 2,    -cy);
               aptf[2] = new PointF(cx / 2, 2 * cy);

               dAngle += Math.PI;

               aptf[3].X = cx / 2 + cx /  4 * (float) Math.Cos(dAngle);
               aptf[3].Y = cy / 2 + cy / 16 * (float) Math.Sin(dAngle);

               grfx.DrawBeziers(pen, aptf);
          }
     }
}
```

* Niektóre prace Pierra Béziera można obejrzeć w witrynie WWW profesora Briana Barsky'ego pod adresem *http://www.cs.berkeley.edu/~barsky/gifs/bezier.html*.

Obrazki zawierające wiele narysowanych linii lub krzywych zwykle wyglądają lepiej po wydrukowaniu, ale poniżej zamieszczam wersję wideo wyników programu:



Choć w tej książce muszę się ograniczyć do czarno-białych obrazków, nie zapomnij o kolorach, gdy będziesz tworzył własne dzieła.

Wyprowadzenie wzoru na krzywą Béziera

Czasem warto znać podstawowe wzory, których system graficzny używa do wyświetlania krzywych. Może kiedyś zechcesz ułożyć inne elementy graficzne (np. znaki tekstowe) względem krzywej narysowanej przez system. Zresztą, wzór warto wyprowadzić choćby po to, żebyś nie myślał, że pewnego dnia spadł z nieba.

Krzywa Béziera to wielomian trzeciego stopnia. Jak wszystkie wielomiany trzeciego stopnia, krzywą Béziera jednoznacznie określają cztery punkty, które nazwaliśmy p_0 (punkt początkowy), p_1 i p_2 (dwa punkty kontrolne) oraz p_3 (punkt końcowy). Te cztery punkty można również oznaczyć jako (x_0, y_0), (x_1, y_1), (x_2, y_2) i (x_3, y_3).

Ogólna, parametryczna postać wielomianu trzeciego stopnia dwóch zmiennych to:

$$x(t) = a_x \cdot t^3 + b_x \cdot t^2 + c_x \cdot t + d_x$$

$$x(t) = a_y \cdot t^3 + b_y \cdot t^2 + c_y \cdot t + d_y$$

gdzie a_x, b_x, c_x, d_x, a_y, b_y, c_y oraz d_y to stałe, a t przybiera wartości od 0 do 1. Każda krzywa Béziera jest jednoznacznie określona przez te osiem stałych. Wartości stałych zależą od czterech punktów definiujących krzywą. Celem tego ćwiczenia jest wyprowadzenie wzorów na obliczanie ośmiu stałych w zależności od położenia czterech punktów.

Pierwsze założenie jest takie, że krzywa Béziera zaczyna się w punkcie (x_0, y_0), gdy t jest równe 0:

$$x(0) = x_0$$

$$y(0) = y_0$$

Nawet to proste założenie pozwala nam uczynić pierwszy krok na drodze do wyprowadzenia stałych. Jeśli podstawimy 0 za t w równaniach parametrycznych, otrzymamy:

$$x(0) = d_x$$

$$y(0) = d_y$$

Oznacza to, że dwie stałe są po prostu współrzędnymi punktu początkowego:

$$d_x = x_0 \tag{1a}$$

$$d_y = y_0 \tag{1b}$$

Drugie założenie związane z krzywą Béziera jest takie, że kończy się ona w punkcie (x_3, y_3), gdy t jest równe 1:

$$x(1) = x_3$$

$$y(1) = y_3$$

Podstawiając 1 za t w równaniach parametrycznych, otrzymujemy:

$$x(1) = a_x + b_x + c_x + d_x$$

$$y(1) = a_y + b_y + c_y + d_y$$

Oznacza to, że między stałymi a współrzędnymi punktu końcowego zachodzi następujący związek:

$$a_x + b_x + c_x + d_x = x_3 \tag{2a}$$

$$a_y + b_y + c_y + d_y = y_3 \tag{2b}$$

Pozostałe dwa założenia dotyczą pierwszych pochodnych równań parametrycznych, które opisują nachylenie krzywej. Pierwsze pochodne uogólnionych równań parametrycznych wielomianu trzeciego stopnia liczone względem t to:

$$x'(t) = 3a_x t^2 + 2b_x t + c_x$$

$$y'(t) = 3a_y t^2 + 2b_y t + c_y$$

Interesuje nas zwłaszcza nachylenie krzywej w dwóch punktach końcowych. W punkcie początkowym krzywa Béziera jest styczna do linii biegnącej od punktu początkowego do pierwszego punktu kontrolnego i ma ten sam kierunek. Tę linię prostą zwykle definiowałoby równanie parametryczne:

$$x(t) = (x_1 - x_0)\, t + x_0$$

$$y(t) = (y_1 - y_0)\, t + y_0$$

dla t z zakresu od 0 do 1. Jednakże inną reprezentacją tej linii prostej mogą być poniższe równania parametryczne:

$$x(t) = 3\,(x_1 - x_0)\, t + x_0$$

$$y(t) = 3\,(y_1 - y_0)\, t + y_0$$

gdzie t przybiera wartości od 0 do 1/3. Dlaczego 1/3? Ponieważ fragment krzywej Béziera, który jest styczny do linii prostej od p_0 do p_1 i ma ten sam kierunek, to

mniej więcej 1/3 całej krzywej. Oto pierwsze pochodne tych zmodyfikowanych równań parametrycznych:

$$x'(t) = 3\,(x_1 - x_0)$$
$$y'(t) = 3\,(y_1 - y_0)$$

Chcemy, żeby te równania reprezentowały nachylenie krzywej Béziera, gdy t jest równe zeru, a zatem

$$x'(0) = 3\,(x_1 - x_0)$$
$$y'(0) = 3\,(y_1 - y_0)$$

Podstawiamy 0 za t w uogólnionych pierwszych pochodnych i otrzymujemy:

$$x'(0) = c_x$$
$$y'(0) = c_y$$

Oznacza to, że:

$$c_x = 3\,(x_1 - x_0) \quad (3a)$$
$$c_y = 3\,(y_1 - y_0) \quad (3b)$$

Ostatnie założenie jest takie, że w punkcie końcowym krzywa Béziera jest styczna do linii prostej biegnącej od drugiego punktu kontrolnego do punktu końcowego i ma taki sam kierunek. Innymi słowy:

$$x'(1) = 3\,(x_3 - x_2)$$
$$y'(1) = 3\,(y_3 - y_2)$$

Z uogólnionych wzorów wiemy, że:

$$x'(1) = 3a_x + 2b_x + c_x$$
$$y'(1) = 3a_y + 2b_y + c_y$$

a zatem:

$$3a_x + 2b_x + c_x = 3\,(x_3 - x_2) \quad (4a)$$
$$3a_y + 2b_y + c_y = 3\,(y_3 - y_2) \quad (4b)$$

Równania 1a, 2a, 3a i 4a to układ czterech równań z czterema niewiadomymi, który pozwala nam obliczać a_x, b_x, c_x i d_x na podstawie x_0, x_1, x_2 i x_3. Po kilku prostych przekształceniach otrzymujemy:

$$a_x = -x_0 + 3x_1 - 3x_2 + x_3$$
$$b_x = 3x_0 - 6x_1 + 3x_2$$
$$c_x = 3x_0 + 3x_1$$
$$d_x = x_0$$

Równania 1b, 2b, 3b i 4b pozwalają obliczyć w ten sam sposób współczynniki y. Możemy teraz podstawić stałe do uogólnionych równań parametrycznych trzeciego stopnia:

$$x(t) = (-x_0 + 3x_1 - 3x_2 + x_3)\,t^3 + (3x_0 - 6x_1 + 3x_2)\,t^2 + (3x_0 + 3x_1)\,t + x_0$$
$$y(t) = (-y_0 + 3y_1 - 3y_2 + y_3)\,t^3 + (3y_0 - 6y_1 + 3y_2)\,t^2 + (3y_0 + 3y_1)\,t + y_0$$

To już właściwie wszystko. Zwykle przestawia się jednak składniki, żeby otrzymać bardziej eleganckie i łatwiejsze w użyciu równania parametryczne:

$$x(t) = (1-t)^3 x_0 + 3t(1-t)^2 x_1 + 3t^2 (1-t) x_2 + t^3 x_3$$

$$y(t) = (1-t)^3 y_0 + 3t(1-t)^2 y_1 + 3t^2 (1-t) y_2 + t^3 y_3$$

Właśnie takimi równaniami zwyczajowo opisuje się krzywą Béziera.

Klasa *BezierManual* w poniższym programie przesłania klasę *Bezier* z programu zamieszczonego wcześniej w tym rozdziale i rysuje drugą krzywą Béziera – tym razem obliczaną „ręcznie" według właśnie wyprowadzonych równań.

Listing BezierManual.cs

```
//-----------------------------------------
// BezierManual.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BezierManual: Bezier
{
     public new static void Main()
     {
          Application.Run(new BezierManual());
     }
     public BezierManual()
     {
          Text = "Bezier Curve \"Manually\" Drawn";
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          base.OnPaint(pea);

          BezierSpline(pea.Graphics, Pens.Red, apt);
     }
     void BezierSpline(Graphics grfx, Pen pen, Point[] aptDefine)
     {
          Point[] apt = new Point[100];

          for (int i = 0; i < apt.Length; i++)
          {
               float t = (float) i / (apt.Length - 1);

               float x = (1 - t) * (1 - t) * (1 - t) * aptDefine[0].X +
                         3 * t  * (1 - t) * (1 - t) * aptDefine[1].X +
                         3 * t * t        * (1 - t) * aptDefine[2].X +
                         t * t * t                  * aptDefine[3].X;

               float y = (1 - t) * (1 - t) * (1 - t) * aptDefine[0].Y +
                         3 * t  * (1 - t) * (1 - t) * aptDefine[1].Y +
                         3 * t * t        * (1 - t) * aptDefine[2].Y +
                         t * t * t                  * aptDefine[3].Y;

               apt[i] = new Point((int) Math.Round(x), (int) Math.Round(y));
          }
          grfx.DrawLines(pen, apt);
     }
}
```

Metoda *OnPaint* w klasie *BezierManual* wywołuje metodę *OnPaint* w klasie podstawowej (czyli *Bezier*), a następnie metodę *BezierSpline* we własnej klasie. Metoda *BezierSpline* jest zdefiniowana podobnie jak *DrawBezier*, z tym że przyjmuje obiekt *Graphics* jako pierwszy argument i potrafi obsłużyć tylko jedną krzywą Béziera. Metoda wykorzystuje tablicę 100 struktur *Point*, oblicza każdy punkt na podstawie wyprowadzonych wyżej równań parametrycznych, a następnie składa krzywą z odcinków. Ręcznie obliczona krzywa jest rysowana na czerwono, żebyś mógł ją porównać z wersją rysowaną przez Windows Forms. Krzywe nie pokrywają się dokładnie, ale nigdy nie różnią się więcej niż o jeden piksel.

Krzywa kanoniczna

Klasa *Graphics* zawiera jeszcze jeden typ krzywej, nazywany krzywą *kanoniczną*, czyli *standardową* lub *normalną*. Krzywe kanoniczne rysuje się za pomocą jednej z metod *DrawCurve*. Metoda *DrawCurve* ma siedem wersji, ale prawdopodobnie najczęściej będziesz używał poniższych czterech:

Metody *DrawCurve* klasy Graphics (wybór)

```
DrawCurve(Pen pen, Point[] apt)
DrawCurve(Pen pen, PointF[] aptf)
DrawCurve(Pen pen, Point[] apt, float fTension)
DrawCurve(Pen pen, PointF[] aptf, float fTension)
```

Potrzebne są co najmniej dwa punkty. Jeśli tablica zawiera tylko dwa punkty, metoda *DrawCurve* rysuje linię prostą od pierwszego punktu do drugiego. Jeśli punktów jest więcej, metoda rysuje zakrzywioną linię łączącą wszystkie punkty.

Krzywa kanoniczna różni się od krzywej Béziera przede wszystkim tym, że przechodzi przez każdy punkt określony w tablicy. Krzywe między poszczególnymi parami punktów są czasem nazywane *segmentami* pełnej krzywej. Kształt każdego segmentu krzywej jest określony przez punkty na jego początku i końcu (oczywiście), ale także przez pozostałe dwa przyległe punkty. Na przykład w tablicy struktur *Point* o nazwie *apt* na kształt segmentu między *apt[3]* i *apt[4]* mają wpływ także punkty *apt[2]* i *apt[5]*.

Na krzywą wpływa także *naprężenie*, które jest jawnym argumentem niektórych wersji metody *DrawCurve*. Jeśli porównamy krzywą do drewnianych lub metalowych krzywików, naprężenie będzie odpowiednikiem sztywności krzywika. Wartość domyślna to 0,5. Naprężenie równe zeru daje w wyniku linie proste: *DrawCurves* zmienia się w *DrawLines*. Przy naprężeniach większych od 0,5 krzywa staje się bardziej zakrzywiona. Można ustawić naprężenie mniejsze od 0, ale często prowadzi to do powstawania pętli. Naprężenia znacznie większe od 1 również mogą tworzyć pętle.

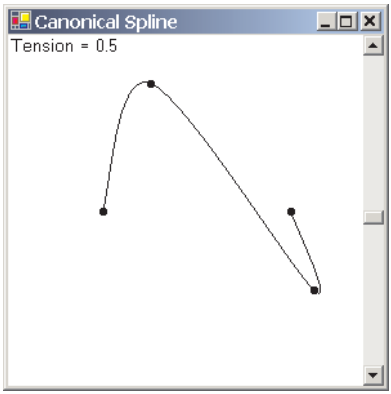
Poeksperymentujmy. Poniższy program jest podobny do programu Bezier, ale zawiera także pasek przewijania do ustawiania naprężenia i daje większą swobodę przesuwania punktów.

Listing CanonicalSpline.cs

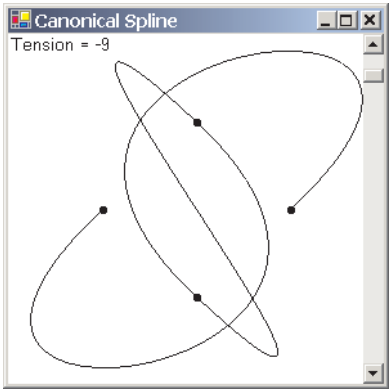
```
//---------------------------------------------
// CanonicalSpline.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CanonicalSpline: Form
{
     protected Point[] apt      = new Point[4];
     protected float   fTension = 0.5f;

     public static void Main()
     {
          Application.Run(new CanonicalSpline());
     }
     public CanonicalSpline()
     {
          Text = "Canonical Spline";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          ResizeRedraw = true;

          ScrollBar scroll = new VScrollBar();
          scroll.Parent = this;
          scroll.Dock = DockStyle.Right;
          scroll.Minimum = -100;
          scroll.Maximum =  109;
          scroll.SmallChange =  1;
          scroll.LargeChange = 10;
          scroll.Value = (int) (10 * fTension);
          scroll.ValueChanged += new EventHandler(ScrollOnValueChanged);

          OnResize(EventArgs.Empty);
     }
     void ScrollOnValueChanged(object obj, EventArgs ea)
     {
          ScrollBar scroll = (ScrollBar) obj;

          fTension = scroll.Value / 10f;

          Invalidate(false);
     }
     protected override void OnResize(EventArgs ea)
     {
          base.OnResize(ea);

          int cx = ClientSize.Width;
          int cy = ClientSize.Height;

          apt[0] = new Point(    cx / 4,     cy / 2);
          apt[1] = new Point(    cx / 2,     cy / 4);
          apt[2] = new Point(    cx / 2, 3 * cy / 4);
          apt[3] = new Point(3 * cx / 4,     cy / 2);
     }
     protected override void OnMouseDown(MouseEventArgs mea)
     {
          Point pt;

          if (mea.Button == MouseButtons.Left)
```

```
            {
                if (ModifierKeys == Keys.Shift)
                    pt = apt[0];
                else if (ModifierKeys == Keys.None)
                    pt = apt[1];
                else
                    return;
            }
            else if (mea.Button == MouseButtons.Right)
            {
                if (ModifierKeys == Keys.None)
                    pt = apt[2];
                else if (ModifierKeys == Keys.Shift)
                    pt = apt[3];
                else
                    return;
            }
            else
                return;

            Cursor.Position = PointToScreen(pt);
        }
        protected override void OnMouseMove(MouseEventArgs mea)
        {
            Point pt = new Point(mea.X, mea.Y);

            if (mea.Button == MouseButtons.Left)
            {
                if (ModifierKeys == Keys.Shift)
                    apt[0] = pt;
                else if (ModifierKeys == Keys.None)
                    apt[1] = pt;
                else
                    return;
            }
            else if (mea.Button == MouseButtons.Right)
            {
                if (ModifierKeys == Keys.None)
                    apt[2]= pt;
                else if (ModifierKeys == Keys.Shift)
                    apt[3] = pt;
                else
                    return;
            }
            else
                return;

            Invalidate();
        }
        protected override void OnPaint(PaintEventArgs pea)
        {
            Graphics grfx  = pea.Graphics;
            Brush    brush = new SolidBrush(ForeColor);

            grfx.DrawCurve(new Pen(ForeColor), apt, fTension);

            grfx.DrawString("Tension = " + fTension, Font, brush, 0, 0);

            for (int i = 0; i < 4; i++)
                grfx.FillEllipse(brush, apt[i].X - 3, apt[i].Y - 3, 7, 7);
        }
}
```

Podobnie jak w programie Bezier, lewy i prawy przycisk myszy służą do zmiany położenia punktów p_1 i p_2. Program CanonicalSpline pozwala ponadto zmienić położenie punktów p_0 i p_3 za pomocą przycisków myszy używanych łącznie z klawiszem [Shift]. Oto typowy wynik:



Naprężenie reguluje się za pomocą paska przewijania; wartość jest wyświetlana w lewym górnym rogu okna. Zakres regulacji wynosi od -10 do 10, więc możesz sam zobaczyć, jak skrajne wartości wpływają na kształt krzywej. Oto jeden z moich ulubionych kształtów, uzyskany przy domyślnym ustawieniu tablicy *Point*:



Można także użyć podzbioru tablicy punktów w poniższych metodach *DrawCurve*:

Metody *DrawCurve* klasy *Graphics* (wybór)

```
DrawCurve(Pen pen, PointF[] aptf, int iOffset, int iSegments)
DrawCurve(Pen pen, Point[] apt, int iOffset, int iSegments,
          float fTension)
DrawCurve(Pen pen, PointF[] aptf, int iOffset, int iSegments,
          float fTension)
```

Argument *iOffset* to indeks do tablicy *Point* lub *PointF*. Od tego punktu zaczyna się krzywa. Argument *iSegments* określa liczbę rysowanych segmentów, a także liczbę dodatkowych struktur *Point* lub *PointF*, których użyje metoda. Przypuśćmy, że *aptf* to tablica struktur *PointF*. Wywołanie:

```
grfx.DrawCurve(pen, aptf, 2, 3);
```

rysuje trzy segmenty: od *aptf[2]* do *aptf[3]*, od *aptf[3]* do *aptf[4]* i od *aptf[4]* do *aptf[5]*. Wyniki nie są takie same, jak w przypadku wywołania prostszej wersji *DrawCurve* z tymi czterema punktami. Wersje przyjmujące argumenty *iOffset* oraz *iSegments* używają punktu *aptf[1]* do określenia kształtu krzywej między punktami *aptf[2]* i *aptf[3]*, a punktu *aptf[6]* – do określenia kształtu krzywej między *aptf[4]* i *aptf[5]*.

Metody *DrawClosedCurve* łączą ostatni punkt w tablicy z pierwszym punktem za pomocą dodatkowej krzywej:

Metody *DrawClosedCurve* klasy Graphics

```
DrawClosedCurve(Pen pen, Point[] apt)
DrawClosedCurve(Pen pen, PointF[] aptf)
DrawClosedCurve(Pen pen, Point[] apt, float fTension, FillMode fm)
DrawClosedCurve(Pen pen, PointF[] aptf, float fTension, FillMode fm)
```

Metoda *DrawClosedCurve* nie rysuje po prostu dodatkowego odcinka. Pierwszy segment rysowany przez *DrawClosedCurve* różni się od segmentu rysowanego przez *DrawCurve*, ponieważ ma na niego wpływ ostatni punkt w tablicy; podobnie, na przedostatnią krzywą wpływa pierwszy punkt w tablicy.

Dwie wersje metody *DrawClosedCurve* mają argument *FillMode*. Z pewnością pamiętasz *FillMode*, typ wyliczeniowy zdefiniowany w przestrzeni nazw *System.Drawing.Drawing2D* i używany w metodzie *DrawPolygon* w celu określenia obszarów, które mają zostać wypełnione:

Wyliczenie *FillMode*

Składowa	Wartość	Komentarz
Alternate	0	Wartość domyślna; na zmianę obszary wypełnione i niewypełnione
Winding	1	Wypełnia większość wewnętrznych obszarów

Do czego jednak – zapytasz – służy tryb wypełniania w metodzie, która po prostu rysuje linie i niczego nie wypełnia? To tajemnica, a metody zdają się działać tak samo bez względu na ustawienie *FillMode*.

Argument *FillMode* ma znacznie więcej sensu w metodach *FillClosedCurve*:

Metody *FillClosedCurve* klasy Graphics

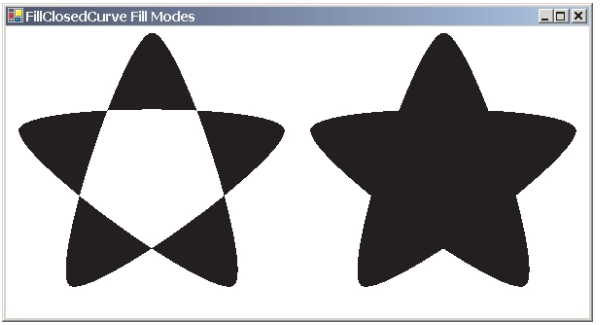
```
FillClosedCurve(Brush brush, Point[] apt)
FillClosedCurve(Brush brush, PointF[] aptf)
FillClosedCurve(Brush brush, Point[] apt, FillMode fm)
FillClosedCurve(Brush brush, PointF[] aptf, FillMode fm)
FillClosedCurve(Brush brush, Point[] apt, FillMode fm, float fTension)
FillClosedCurve(Brush brush, PointF[] aptf, FillMode fm, float fTension)
```

Poniższy program ClosedCurveFillModes jest niemal identyczny z programem FillModesClassical z rozdziału 5. Rysuje dwie pięcioramienne gwiazdy, żeby pokazać różnicę między trybami *FillMode.Alternate* i *FillMode.Winding*.

Listing ClosedCurveFillModes.cs

```
//-----------------------------------------------
// ClosedCurveFillModes.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ClosedCurveFillModes: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new ClosedCurveFillModes());
     }
     ClosedCurveFillModes()
     {
          Text = "FillClosedCurve Fill Modes";
          ClientSize = new Size(2 * ClientSize.Height, ClientSize.Height);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush   brush = new SolidBrush(clr);
          Point[] apt   = new Point[5];

          for (int i = 0; i < apt.Length; i++)
          {
               double dAngle = (i * 0.8 - 0.5) * Math.PI;
               apt[i] = new Point(
                              (int)(cx *(0.25 + 0.24 * Math.Cos(dAngle))),
                              (int)(cy *(0.50 + 0.48 * Math.Sin(dAngle))));
          }
          grfx.FillClosedCurve(brush, apt, FillMode.Alternate);

          for (int i = 0; i < apt.Length; i++)
               apt[i].X += cx / 2;

          grfx.FillClosedCurve(brush, apt, FillMode.Winding);
     }
}
```

Choć kształty rysowane przez program nadal można uznać za gwiazdy, mają bardziej miękki wygląd:



Przypominają ciasteczka w kształcie gwiazdy, które zaraz po wycięciu miały proste krawędzie, ale zaokrągliły się nieco podczas pieczenia.

Wyprowadzenie wzoru na krzywą kanoniczną

Podobnie jak krzywa Béziera, krzywa kanoniczna jest wielomianem trzeciego stopnia, więc jej uogólnione wzory parametryczne to:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

dla t z zakresu od 0 do 1. Pierwsze pochodne to:

$$x'(t) = 3a_x t^2 + 2b_x t + c_x$$

$$y'(t) = 3a_y t^2 + 2b_y t + c_y$$

Rozważmy cztery punkty, p_0, p_1, p_2 i p_3. Zamierzam wyprowadzić wzór na segment między punktami p_1 i p_2. Kształt segmentu zależy od tych dwóch punktów, a także od dwóch sąsiednich – p_0 i p_3. Zakładamy, że krzywa zaczyna się w punkcie p_1 i kończy w p_2:

$$x(0) = x_1$$

$$y(0) = y_1$$

$$x(1) = x_2$$

$$y(1) = y_2$$

Z uogólnionych wzorów parametrycznych możemy wyprowadzić następujące równania:

$$d_x = x_1$$

$$d_y = y_1$$

$$a_x + b_x + c_x + d_x = x_2$$

$$a_y + b_y + c_y + d_y = y_2$$

Pozostałe dwa założenia określają nachylenie krzywej w punktach p_1 i p_2. Nachylenie w punkcie p_1 to iloczyn naprężenia (które określimy literą T) oraz nachylenia linii prostej między punktami p_0 i p_2. Podobnie, nachylenie w punkcie p_2 to naprężenie razy nachylenie linii prostej między p_1 i p_3:

$$x'(0) = T(x_2 - x_0)$$
$$y'(0) = T(y_2 - y_0)$$
$$x'(1) = T(x_3 - x_1)$$
$$y'(1) = T(y_3 - y_1)$$

Z pierwszych pochodnych uogólnionych wzorów parametrycznych wnioskujemy, że:

$$c_x = T(x_2 - x_0)$$
$$c_y = T(y_2 - y_0)$$
$$3a_x + 2b_x + c_x = T(x_3 - x_1)$$
$$3a_y + 2b_y + c_y = T(y_3 - y_1)$$

Po rozwiązaniu tego układu równań otrzymujemy:

$$a_x = T(x_2 - x_0) + T(x_3 - x_1) + 2x_1 - 2x_2$$
$$a_y = T(y_2 - y_0) + T(y_3 - y_1) + 2y_1 - 2y_2$$
$$b_x = -2T(x_2 - x_0) - T(x_3 - x_1) - 3x_1 + 3x_2$$
$$b_y = -2T(y_2 - y_0) - T(y_3 - y_1) - 3y_1 + 3y_2$$
$$c_x = T(x_2 - x_0)$$
$$c_y = T(y_2 - y_0)$$
$$d_x = x_1$$
$$d_y = y_1$$

Program CanonicalSplineManual dowodzi, że te stałe są prawidłowe.

Listing CanonicalSplineManual.cs

```
//-----------------------------------------------------
// CanonicalSplineManual.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CanonicalSplineManual: CanonicalSpline
{
     public new static void Main()
     {
          Application.Run(new CanonicalSplineManual());
     }
     public CanonicalSplineManual()
     {
          Text = "Canonical Spline \"Manually\" Drawn";
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          base.OnPaint(pea);

          CanonicalSpline(pea.Graphics, Pens.Red, apt, fTension);
     }
     void CanonicalSpline(Graphics grfx, Pen pen, Point[] apt, float T)
```

```
    {
        CanonicalSegment(grfx, pen, apt[0], apt[0], apt[1], apt[2], T);
        CanonicalSegment(grfx, pen, apt[0], apt[1], apt[2], apt[3], T);
        CanonicalSegment(grfx, pen, apt[1], apt[2], apt[3], apt[3], T);
    }
    void CanonicalSegment(Graphics grfx, Pen pen, Point pt0, Point pt1,
                          Point pt2, Point pt3, float T)
    {
        Point[] apt = new Point[10];

        float SX1 = T * (pt2.X - pt0.X);
        float SY1 = T * (pt2.Y - pt0.Y);
        float SX2 = T * (pt3.X - pt1.X);
        float SY2 = T * (pt3.Y - pt1.Y);
        float AX = SX1 + SX2 + 2 * pt1.X - 2 * pt2.X;
        float AY = SY1 + SY2 + 2 * pt1.Y - 2 * pt2.Y;
        float BX = -2 * SX1 - SX2 - 3 * pt1.X + 3 * pt2.X;
        float BY = -2 * SY1 - SY2 - 3 * pt1.Y + 3 * pt2.Y;
        float CX = SX1;
        float CY = SY1;
        float DX = pt1.X;
        float DY = pt1.Y;

        for (int i = 0; i < apt.Length; i++)
        {
            float t = (float)i / (apt.Length - 1);
            apt[i].X = (int) (AX * t * t * t + BX * t * t + CX * t + DX);
            apt[i].Y = (int) (AY * t * t * t + BY * t * t + CY * t + DY);
        }
        grfx.DrawLines(pen, apt);
    }
}
```

Kilka uwag dotyczących tego programu: metoda *CanonicalSpline* obsługuje tylko tablicę czteroelementową i trzykrotnie wywołuje metodę *CanonicalSegment*, za każdym razem wyświetlając jeden segment. Pierwszy i ostatni segment trzeba traktować specjalnie, ponieważ krzywą wyznaczają tylko trzy, a nie cztery punkty.

Metoda *CanonicalSegment* rysuje segment, wykorzystując tablicę zaledwie 10 struktur *Point*. To za mało, żeby krzywa była gładka, ale wystarcza, by dowieść, że metoda rzeczywiście imituje działanie metody *DrawCurve* z klasy *Graphics*.

W rozdziałach 15 i 19 znajdziesz więcej przykładowych programów wykorzystujących krzywe Béziera i krzywe kanoniczne.