Krzywe Béziera

Czas porozmawiać o genialnym w swojej prostocie matematycznym konstrukcie, obecnym bodaj w każdym obszarze informatyki związanym w jakikolwiek sposób z grafiką komputerową.

RENAULT VS CITROEN

Niełatwo jest znaleźć kogoś, kto nie miał nigdy do czynienia z krzywymi Béziera. Są one obecne w aplikacjach do edycji grafiki wektorowej (np. Inkscape, CorelDRAW), rastrowej (GIMP, Adobe Photoshop), trójwymiarowej (Blender, 3D Studio Max, Maya), w aplikacjach typu CAD (np. Fusion 360, AutoCAD), edytorach filmów i animacji (Shotcut, Adobe Premiere), a nawet w pakietach biurowych (Microsoft Office, LibreOffice). W dwóch słowach, jeżeli program poważnie podchodzi do tematu i ma cokolwiek wspólnego z grafiką, z pewnością korzysta w pewnym momencie z krzywych Béziera.

Przyzwyczaiłem się już trochę do tego, że większość sprytnych matematycznych wynalazków jest dziełem, no cóż, matematyków – teoretyków oraz pracowników akademickich. Tymczasem autorami opisywanego rewolucyjnego odkrycia w zakresie konstruowania, przechowywania i wyświetlania krzywych jest dwóch francuskich inżynierów z branży automotive: pracujący w przedsiębiorstwie Renault Pierre Étienne Bézier oraz (niezależnie) pracownik Citroena, Paul de Faget de Casteljau. Co ciekawe, prace dotyczące projektowania karoserii samochodów prowadzili oni równolegle, a swoją nazwę krzywe Béziera zawdzięczają temu pierwszemu z nich tylko dlatego, że to przedsiębiorstwo Renault jako pierwsze (pod koniec lat 60. XX wieku) zdecydowało się ujawnić wyniki prowadzonych tam prac.

APARAT MATEMATYCZNY

Krzywe Béziera korzystają z wielomianów bazowych Bernsteina, zaś do zrozumienia tych ostatnich będziemy musieli odświeżyć sobie nieco wiedzę na temat symbolu Newtona. Nie jest tego wcale aż tak dużo, więc miejmy szybko tę matematykę już za sobą.

Symbol Newtona

Symbolem Newtona nazywamy wartość wyznaczoną przy pomocy wzoru:

$$\binom{n}{k} = \frac{n!}{(n-k)!\cdot k!}, \text{ dla } 0 \le k \le n, \text{ gdzie}$$

$$n! = 1\cdot 2\cdot 3\cdot \ldots \cdot (n-1)\cdot n \text{ oraz } 0! \overset{def}{=} 1$$

Wzór 1. Symbol Newtona

Silnia (zapis n!) jest funkcją, która rośnie przeraźliwie szybko – szybciej nawet od funkcji wykładniczej. Na przykład 2! = 2, 5! = 120, 10! = 3628800, 13! nie mieści się już w zakresie typu `uint`, a 21! – w zakresie typu `ulong` (64-bitowa liczba bez znaku). Na szczęście jednak symbol Newtona zachowuje się nieco łagodniej i nie próbuje uciekać do nieskończoności w aż tak gwałtownym tempie.

Aby uniknąć liczenia silni (lub, co nieco sprytniejsze, skracania ułamków i obliczania iloczynów), możemy skorzystać z ciekawego matematycznego konstruktu, jakim jest trójkąt Pascala. Wyobrażamy go sobie następująco: konstruujemy wiersze zawierające ciągi liczb. W pierwszym wierszu znajduje się jedynka, a każdy kolejny wiersz zawiera zawsze o jedną wartość więcej niż poprzedni. Na skraju znajdują się zawsze jedynki, zaś wartości pomiędzy nimi stanowią sumę dwóch wartości znajdujących się bezpośrednio powyżej, w poprzednim wierszu. Budowę Trójkąta Pascala możemy zobaczyć na Rysunku 1 (czerwone i niebieskie liczby obrazują sposób obliczania wartości w kolejnych wierszach).

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

Rysunek 1. Trójkąt Pascala

Jeżeli teraz rozpiszemy trójkąt Pascala do odpowiedniej wysokości, możemy w łatwy sposób wyznaczyć dwumian Newtona dla wartości n i k: bierzemy po prostu n-ty wiersz (licząc od 0), a potem w ramach tego wiersza k-tą wartość (również licząc od 0). I tyle!

Wielomiany bazowe Bernsteina

Wielomiany bazowe Bernsteina zostały nazwane od nazwiska Siergieja Bernsteina, który skonstruował je w 1912 roku w celu udowodnienia twierdzenia Weierstrassa o przybliżaniu funkcji ciągłych. W oryginale służyły one do zbudowania wielomianu Bernsteina, ale nam przydadzą się w nieco innym celu.

Wielomian bazowy Bernsteina definiujemy dla wartości naturalnych n oraz i takich, że $0 \le i \le n$.

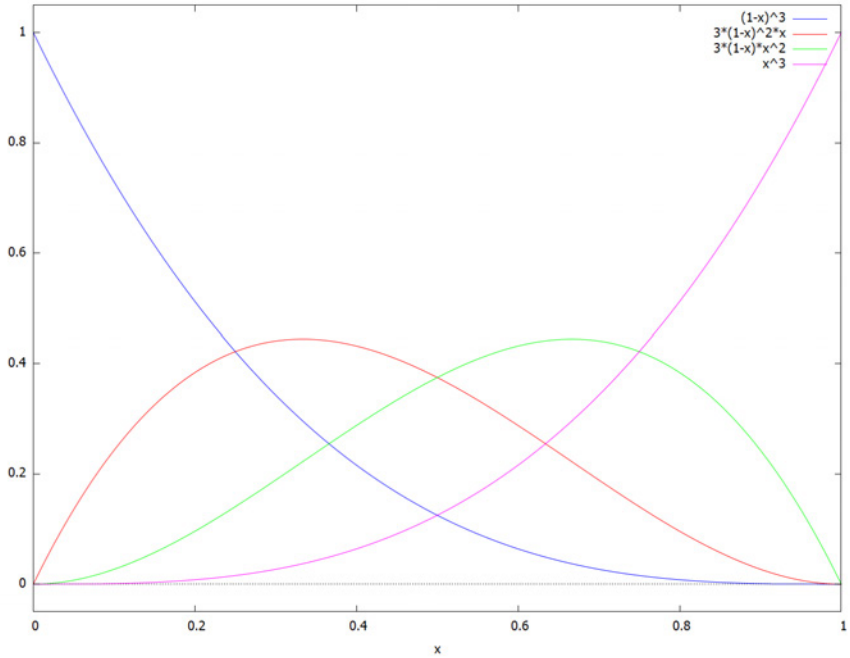
$$B_i^n(x) = \binom{n}{i}\cdot x^i \cdot (1-x)^{n-i}$$

Wzór 2. Wielomian bazowy Bernsteina

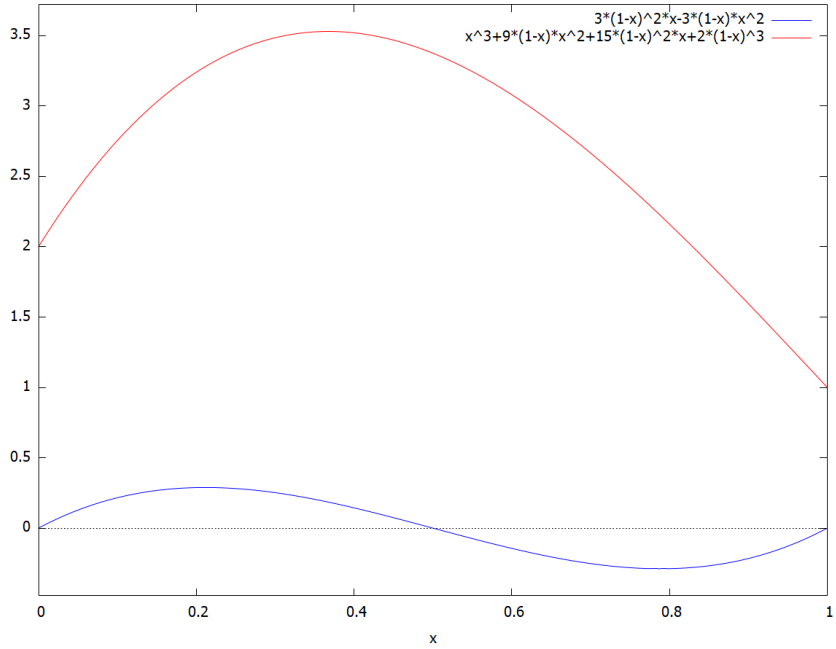
Dla przykładu, kolejne wielomiany bazowe Bernsteina dla n=3 wyglądają następująco:

$$(1-x)^3,\ 3\cdot x\cdot (1-x)^2,\ 3\cdot x^2\cdot (1-x), oraz\, x^3$$

Kiedy popatrzymy na kolejne wielomiany, da się dostrzec pewną prawidłowość. Jest ona znacznie bardziej wyraźna, gdy narysujemy wykresy tych funkcji dla zakresu $x \in [0, 1]$:



Rysunek 2. Wykres wszystkich wielomianów Bernsteina dla n=3



Rysunek 3. Przykładowe krzywe Béziera 3. stopnia

KRZYWE BÉZIERA

Mamy już wszystkie części układanki, żeby zaprezentować wzór krzywych Béziera. Co istotne, mówimy o *krzywych* Béziera, a nie o *krzywej* Béziera, ponieważ jest to tak naprawdę rodzina funkcji różniących się od siebie stopniem.

Wzór krzywych Béziera stopnia n wygląda następująco:

$$p(t) = \sum_{i=0}^{n} p_i \cdot B_i^n(t) \; dla \; t \in [0,1]$$

Wzór 3. Krzywe Béziera stopnia n

Rozbijmy sumę, aby wzór stał się nieco bardziej czytelny:

$$p(t) = p_1 \cdot B_1^n(t) \cdot p_2 \cdot B_2^n(t) + \ldots + p_n B_n^n(t)$$

Wzór 4. Nieco uproszczony wzór krzywej Béziera stopnia n

Aby przekonać czytelnika, że nie mamy tu do czynienia ze skomplikowaną matematyką,, zobaczmy jeszcze, zanim przejdziemy dalej, jak wyglądają wzory krzywych Béziera stopnia 2 i 3.

$$p_i \cdot (1-x)^2 + p_2 \cdot 2 \cdot (1-x) \cdot x + p_3 \cdot x^2$$
$$p_1 \cdot (1-x)^3 + p_2 \cdot 3 \cdot (1-x)^2 \cdot x + p_3 \cdot 3 \cdot (1-x) \cdot x^2 + p_4 \cdot x^3$$

Wzór 5. Krzywe Béziera 2 i 3 stopnia

Spróbujmy teraz zagłębić się nieco we wzory, by zobaczyć, czym tak naprawdę są krzywe Béziera. Oprócz powtarzających się zapisów B_i^n oznaczających wielomiany bazowe Bernsteina, we wzorze pojawiły się tam nowe wartości oznaczone jako p_1, p_2, …, p_n, stanowiące parametry krzywej Béziera n-tego stopnia. Zatrzymajmy się przy nich przez chwilę.

Aby zrozumieć, dlaczego figurują one we wzorze, musimy przypomnieć sobie, że krzywe Béziera powstały przede wszystkim po to, by można było w łatwy sposób je modelować wewnątrz aplikacji graficznych. Mówimy przecież o mechanizmie, który miał pomóc w projektowaniu karoserii samochodów – co komu po krzywej, która wprawdzie pozwala wymodelować dowolnie skomplikowane kształty, ale jednocześnie wymaga ścisłej, matematycznej wiedzy, by ją zastosować?

Dlatego też każda krzywa Béziera (danego stopnia) zdefiniowana jest przez szereg wartości liczbowych, które bezpośrednio wpływają na jej kształt. Na przykład krzywe Béziera 3. stopnia dla parametrów (0, 1, -1, 0) oraz (2, 3, 5, 1) przedstawiono na Rysunku 3.

Jasnym jest, że wartości parametrów wpływają bezpośrednio na kształt skonstruowanej krzywej. Wciąż jednak operujemy na wykresach funkcji w przedziale [0, 1], a przecież krzywe Béziera mają najwięcej sensu na płaszczyźnie. Wystarczy jednak dokonać niewielkiej modyfikacji, aby przenieść się do przestrzeni dwuwymiarowej.

DRUGI WYMIAR

Jesteśmy przyzwyczajeni do tego, że wykres funkcji budujemy poprzez przesuwanie się wzdłuż poziomej osi i wyznaczanie wartości na pionowej. W przypadku wykresów dwuwymiarowych coś takiego nie ma już racji bytu, ponieważ na płaszczyźnie wykres może zawracać, zapętlać się i układać w dość dowolny sposób. Dlatego też funkcje dwuwymiarowe definiujemy nieco inaczej, niż jednowymiarowe.

Ponieważ z oczywistego powodu nie możemy już wybrać żadnej „osi", wzdłuż której będziemy się przesuwać, wprowadzamy ją w sposób sztuczny. Wyobrażamy sobie więc wirtualną oś „t", wzdłuż której „przesuwamy się", wyznaczając kolejne wartości x oraz y naszego wykresu. Wzór funkcji dwuwymiarowej wygląda następująco:

$$\begin{cases} x = f_1(t) \\ y = f_2(t) \end{cases}$$

Wzór 6. Funkcja dwuwymiarowa

Na przykład wzór wykresu przedstawiającego okrąg o promieniu r oraz środku w punkcie (a, b) zdefiniujemy tak:

$$\begin{cases} x = r \cdot \sin(t) + a \\ y = r \cdot \cos(t) + b \end{cases}$$

Wzór 7. Wzór okręgu na płaszczyźnie

Wróćmy jednak do krzywych Béziera. Wiemy już, że do wyznaczenia krzywej n-tego stopnia potrzebujemy (n+1) parametrów. Ponieważ jednak będziemy musieli zdefiniować dwa osobne wykresy dla współrzędnych x oraz y, liczba parametrów też się podwoi. Dlatego dwuwymiarową krzywą Béziera zapiszemy następującym wzorem:

$$\begin{cases} x(t) = \sum_{i=0}^{n} px_i \cdot B_i^n(t) \\ y(t) = \sum_{i=0}^{n} py_i \cdot B_i^n(t) \end{cases} \quad dla\, t \in [0, 1]$$

Wzór 8. Dwuwymiarowa krzywa Béziera

Matematyka jest dosyć elastyczna, jeśli chodzi o zapis. Jeżeli pogrupujemy wszystkie parametry w pary i przyjmiemy, że P_i oznacza punkt o współrzędnych (px_i, py_i), to powyższy wzór możemy skrócić:

$$P(t) = \sum_{i=0}^{n} P_i \cdot B_i^n(t)$$

Wzór 9. Skrócona dwuwymiarowa krzywa Béziera

Wróciliśmy tym sposobem do oryginalnego zapisu, ale w trakcie tego procesu przekształciliśmy parametry p w punkty P (nazywane punktami kontrolnymi krzywej Béziera). I o ile może się wydawać, że jest to kosmetyczna zmiana skracająca zapis, wbrew pozorom ma kolosalne znaczenie dla możliwości manualnego kształtowania przebiegu krzywej, bo w miejsce manipulacji parametrami liczbowymi wprowadza ona możliwość umieszczenia punktów kontrolnych wraz z krzywą na ekranie i obserwowania, w jaki sposób ich przemieszczanie wpływa na jej kształt.

JAK TO DZIAŁA?

Dużo matematycznych konstruktów wymaga skomplikowanej wiedzy, by dokładnie zrozumieć, jak naprawdę one działają. W przypadku krzywych Béziera jest zupełnie inaczej – jest tu obecna tak prosta matematyka, że bez większych problemów powinniśmy ją rozszyfrować.

Kluczem do zrozumienia całego algorytmu jest pewien ciekawy fakt dotyczący wielomianów bazowych Bernsteina. Otóż dla stałej wartości t i wybranego stopnia n wszystkie wielomiany bazowe Bernsteina tego stopnia sumują się zawsze do 1. Dodajmy też, że w zakresie od 0 do 1 wielomiany bazowe Bernsteina są zawsze dodatnie.

W definicji krzywej Béziera n-tego stopnia widzimy, że punkty kontrolne (czy też parametry) są przemnażane po kolei przez wszystkie wielomiany bazowe Bernsteina (które – jak się właśnie dowiedzieliśmy – sumują się do 1). Czy kojarzycie inny konstrukt matematyczny, w którym serię liczb mnożymy przez zbiór wartości sumujących się do 1, a następnie dodajemy do siebie? Podejrzewam, że większość z czytelników miała z nim kiedyś do czynienia. To przecież zwykła średnia ważona!

Przyjrzyjmy się jeszcze raz Rysunkowi 2. Gdy t = 0, waga pierwszego punktu jest równa 1, zaś wszystkie pozostałe – 0. Możemy stąd wyciągnąć wniosek, że na samym początku konstruowana krzywa pokryje się z pierwszym punktem kontrolnym. Gdy zaczniemy powoli zwiększać wartość t, zwrócimy uwagę, że waga pierwszego punktu zaczyna stopniowo maleć, ale jednocześnie rośnie waga pierwszego punktu kontrolnego (choć nie osiąga ona 1, czyli nie mamy gwarancji, że krzywa przez ten punkt przejdzie – jakkolwiek jest to możliwe). Potem drugi punkt kontrolny traci na znaczeniu, a na kształt krzywej zaczyna wpływać trzeci punkt kontrolny, aż na końcu waga ostatniego punktu kontrolnego rośnie do 1 i krzywa kończy się dokładnie w miejscu, gdzie ten punkt się znajduje.

WŁASNOŚCI

Choć stosunkowo nieskomplikowane, krzywe Béziera charakteryzują się całym szeregiem ciekawych własności.

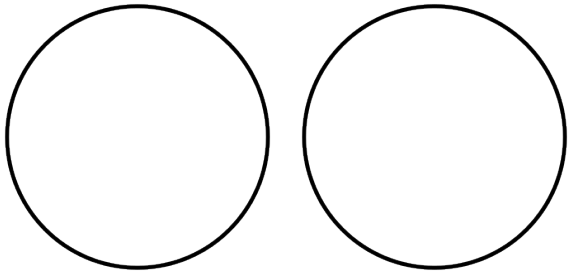
Na początku, z uwagi na ich specyficzną konstrukcję – jako średniej ważonej – warto zauważyć, że poza t = 0 oraz t = 1 *każdy* punkt kontrolny wpływa na kształt *całej* krzywej. Wynika to z faktu, iż poza wspomnianymi miejscami waga każdego z punktów kontrolnych jest niezerowa. Z drugiej zaś strony punkty t = 0 oraz t = 1 są kontrolowane tylko i wyłącznie przez skrajne punkty kontrolne, co daje nam gwarancję, że krzywa w nich się rozpocznie i zakończy.

Patrząc na wykresy wielomianów bazowych Bernsteina, możemy wywnioskować też, że skonstruowana, końcowa krzywa powinna być *gładka* (bo stanowi sumę gładkich funkcji). Matematycznie oznacza to, że funkcja jest nieskończenie różniczkowalna, ale można tę własność wyjaśnić trochę bardziej przyziemnie – krzywa nie będzie miała żadnych przerw ani też ostrych kantów.

Każda krzywa Béziera jest zawsze zawarta wewnątrz wypukłej otoczki swoich punktów kontrolnych. Innymi słowy, jeżeli w prostokątnym obszarze znajdują się wszystkie punkty kontrolne krzywej, to krzywa ta w całości zmieści się w tym obszarze.

Idąc dalej, krzywe Béziera są zawsze styczne w punktach t = 0 oraz t = 1 do odcinków zbudowanych ze skrajnych punktów kontrolnych (odpowiednio, pierwszego i drugiego oraz przedostatniego i ostatniego). Dzięki temu łączenie krzywych Béziera w bardziej złożone krzywe jest bardzo łatwe, o ile tylko zadbamy o współliniowość odcinka tworzonego przez dwa ostatnie punkty pierwszej krzywej i dwa pierwsze drugiej.

Pomimo wielkiej elastyczności w zakresie konstruowanych kształtów krzywe Béziera budowane są z wielomianów, co oznacza, że przy ich pomocy nie jest możliwe modelowanie krzywych eliptycznych, czyli na przykład okręgów, elips, hiperbol i tak dalej. Warto jednak mieć na uwadze, że krzywe Béziera wykorzystywane są zwykle do wszelkiego rodzaju wizualizacji, co oznacza, że zwykle muszą one aproksymować jakiś kształt tylko na tyle dobrze, żeby nie dało się dostrzec różnicy gołym okiem. Dla przykładu, na Rysunku 4 przedstawiony jest okrąg oraz aproksymujące go cztery sklejone krzywe Béziera 3. stopnia (w sumie 12 punktów kontrolnych). Spróbujcie ocenić, który jest który.



Rysunek 4. Okrąg i sklejane krzywe Béziera

IMPLEMENTACJA

Czy samodzielne zaimplementowanie krzywych Béziera jest trudne? W żadnej mierze. Jeżeli nie przewidujemy zbyt dużych wahań w zakresie stopnia krzywych, to dwumian Newtona możemy zaimplementować przy pomocy trójkąta Pascala, a wtedy wielomiany bazowe Bernsteina oraz sama krzywa Béziera pozostają już tylko formalnością.

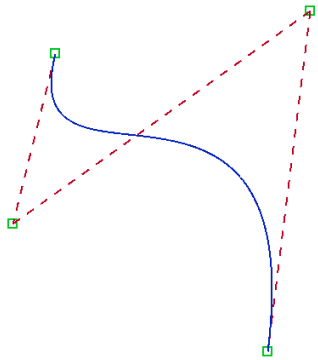
Listing 1. Implementacja krzywych Béziera w C#/.NET 6

```
public static class Bezier
{
    private static List<int[]> pascal = new();

    private static int Newton(int n, int k)
    {
        if (n < 0)
            throw new ArgumentOutOfRangeException(nameof(n));
        if (k < 0 || k > n)
            throw new ArgumentOutOfRangeException(nameof(k));

        while (pascal.Count <= n)
        {
            if (pascal.Count == 0)
                pascal.Add(new int[] { 1 });
            else
            {
                var row = new int[pascal.Count + 1];
                row[0] = row[^1] = 1;
                for (int i = 1; i < row.Length - 1; i++)
                {
                    row[i] = pascal[pascal.Count - 1][i - 1] +
                        pascal[pascal.Count - 1][i];
                }

                pascal.Add(row);
            }
        }

        return pascal[n][k];
    }

    private static float Bernstein(int i, int n, float t)
    {
        if (n < 1)
            throw new ArgumentOutOfRangeException(nameof(n));
        if (i < 0 || i > n)
            throw new ArgumentOutOfRangeException(nameof(i));
        if (t < 0 || t > 1)
            throw new ArgumentOutOfRangeException(nameof(t));

        return (float)(Newton(n, i)
            * Math.Pow(t, i)
            * Math.Pow(1 - t, n - i));
    }

    public static Vector2 Evaluate(Vector2[] points, float t)
    {
        if (points.Length < 2)
            throw new ArgumentOutOfRangeException(nameof(points));
        if (t < 0 || t > 1)
            throw new ArgumentOutOfRangeException(nameof(t));

        Vector2 result = Vector2.Zero;

        for (int i = 0; i < points.Length; i++)
            result += points[i]
                * Bernstein(i, points.Length - 1, t);

        return result;
    }
}
```

Listing 2. Przykładowy program rysujący krzywe

```
public class Program
{
    static void Main(string[] args)
    {
        Bitmap bitmap = new Bitmap(512, 512,
            PixelFormat.Format32bppArgb);
        using Graphics g = Graphics.FromImage(bitmap);
```

```
        using Brush pointBrush =
            new SolidBrush(Color.FromArgb(0, 196, 32));
        using Brush controlBrush =
            new SolidBrush(Color.FromArgb(192, 0, 32));
        using Brush curveBrush =
            new SolidBrush(Color.FromArgb(0, 32, 192));

        using Pen pointPen =
            new Pen(pointBrush, 2);
        using Pen controlPen =
            new Pen(controlBrush, 2)
        {
            DashPattern = new[] { 5.0f, 5.0f }
        };
        using Pen curvePen = new Pen(curveBrush, 2);

        // Modify points here
        var points = new Vector2[]
        {
            new Vector2(100, 100),
            new Vector2(50, 300),
            new Vector2(400, 50),
            new Vector2(350, 450)
        };

        // Estimate number of segments
        float lengthSum = 0.0f;
        for (int i = 0; i < points.Length - 1; i++)
            lengthSum += (points[i + 1] - points[i]).Length();
        int segmentCount = (int)lengthSum;

        // Draw background
        g.FillRectangle(Brushes.White,
            new RectangleF(0, 0, bitmap.Width, bitmap.Height));

        // Draw points
        for (int i =0; i < points.Length; i++)
        {
            g.DrawRectangle(pointPen,
                new RectangleF(points[i].X - 5,
                    points[i].Y - 5,
                    10,
                    10));
        }

        // Draw control
        for (int i = 0; i < points.Length - 1; i++)
        {
            g.DrawLine(controlPen,
                new PointF(points[i].X, points[i].Y),
                new PointF(points[i + 1].X, points[i + 1].Y));
        }

        // Draw curve
        for (int i = 0; i < segmentCount; i++)
        {
            float t1 = (float)i / segmentCount;
            float t2 = (float)(i + 1) / segmentCount;

            var p1 = Bezier.Evaluate(points, t1);
            var p2 = Bezier.Evaluate(points, t2);

            g.DrawLine(curvePen, new PointF(p1.X, p1.Y),
                new PointF(p2.X, p2.Y));
        }

        bitmap.Save(@"D:\Bezier.png");
    }
}
```



Rysunek 5. Efekt działania programu z Listingu 1

REALIA

Zaprezentowany chwilę wcześniej program jest bardzo uniwersalny i pozwala skonstruować krzywą Béziera dowolnego stopnia. Pamiętajmy jednak, że krzywe te powstały przede wszystkim po to, by ułatwić manualne ich konstruowanie w programach typu CAD, a przecież manipulowanie kilkunastoma punktami kontrolnymi w celu uzyskania określonego kształtu nie brzmi wcale jak proste zadanie – tym bardziej gdy przypomnimy sobie, że krzywe można łatwo ze sobą sklejać. Dlatego też najczęściej spotykanym rodzajem krzywych Béziera są krzywe 3. stopnia, które stanowią bardzo dobry balans pomiędzy prostotą używania a elastycznością w zakresie konstruowania pożądanych kształtów.

Jeżeli przyjmiemy stały stopień krzywych, implementacja oczywiście radykalnie się uprości:

Listing 3. Implementacja krzywych Béziera 3. stopnia

```
public static class Bezier3
{
    public static Vector2 Evaluate(Vector2 p1,
        Vector2 p2,
        Vector2 p3,
        Vector2 p4,
        float t)
    {
        if (t < 0 || t > 1)
            throw new ArgumentOutOfRangeException(
                nameof(t));

        return p1 * (float)Math.Pow(1 - t, 3) +
            p2 * 3 * (float)Math.Pow(1 - t, 2) * t +
            p3 * (1 - t) * (float)Math.Pow(t, 2) +
            p4 * (float)Math.Pow(t, 3);
    }

    public static Vector2 Evaluate(Vector2[] points,
        float t)
    {
        if (points.Length != 4)
            throw new ArgumentOutOfRangeException(
                nameof(points));

        return Evaluate(points[0],
            points[1],
            points[2],
            points[3],
            t);
    }
}
```

RENDEROWANIE

Przypuśćmy, że zajdzie potrzeba ręcznego wyrenderowania krzywej Béziera. Najłatwiej jest zrobić to, traktując ją jako bardzo gęstą łamaną – czyli dzieląc na małe odcinki. Może to brzmieć jak rozwiązanie prymitywne, ale w większości przypadków jest ono całkowicie wystarczające – o ile tylko we właściwy sposób dobierzemy granularność podziału. Ta natomiast zależy od medium, na którym chcemy wyrenderować kształt. Jeśli jest to ekran komputera, wystarczy, że najkrótszy odcinek będzie nie dłuższy niż jeden piksel.

Jednym ze sposobów na oszacowanie gęstości podziału jest wyznaczenie długości krzywej, ale nie jest to niestety zbyt łatwe zadanie. Wzorem na długość krzywej opisanej dwiema funkcjami $x=f_1(t)$ i $y = f_2(t)$ jest:

$$L = \int_a^b \sqrt{(f_1'(t))^2 + (f_2'(t))^2 dt}$$

Wzór 10. Długość krzywej parametrycznej

Jeżeli teraz podstawimy pod f_1 i f_2 wzory krzywej Béziera trzeciego stopnia, otrzymamy formułę tak skomplikowaną, że obliczenia jej odmówiła zarówno Maxima, jak i Wolfram Alpha.

W miejsce dokładnych obliczeń możemy zastosować pewne przybliżenie: długość łamanej zbudowanej z kolejnych punktów kontrolnych (widocznych na Rysunku 5 jako czerwone, przerywane linie). Ponieważ wiemy, że krzywa Béziera przechodzi jedynie przez skrajne punkty kontrolne, a pozostałe tylko przybliża, możemy wywnioskować, że długość łamanej będzie zawsze większa lub w skrajnym przypadku równa długości krzywej. Niestety jednak, często jest to wartość dosyć mocno zawyżona. W przypadku krzywej przedstawionej na Rysunku 5 łamana ma długość 1039.3844 pikseli, co stanowi przeszło dwukrotność długości krzywej, o czym zaraz się przekonamy.

Istnieje jednak prosty sposób na obliczenie długości krzywej z zadaną dokładnością. Zaczynamy od poprowadzenia odcinka od pierwszego do ostatniego punktu kontrolnego (jak pamiętamy, krzywa zawsze przez nie przebiega). Długość tego odcinka stanowi oczywiście jakieś przybliżenie długości krzywej, ale oczywiście nie ma co spodziewać się cudów.

Jeżeli przyjmiemy, że f(t) jest funkcją opisującą naszą krzywą, to odcinek, który skonstruowaliśmy, przebiega od punktu f(0) do punktu f(1). Wprowadzamy teraz dodatkowy punkt, dzielący istniejący przedział na pół, czyli f(0,5) (tu dygresja: choć koncepcja taka jest kusząca, punkt ten wcale niekoniecznie leży w geometrycznym środku krzywej!).

Mamy teraz trzy punkty – prowadzimy więc dwa odcinki: jeden od f(0) do f(0,5) oraz drugi od f(0,5) do f(1). Teraz obliczamy ich długości i sumujemy – otrzymaliśmy nieco lepsze oszacowanie długości krzywej. Następnie dzielimy istniejące przedziały na połowy i otrzymujemy cztery odcinki: od f(0) do f(0,25), od f(0,25) do f(0,5), od f(0,5) do f(0,75) i wreszcie od f(0,75) do f(1). Proces ten możemy powtarzać tak długo, aż nie wyznaczymy długości krzywej z zadowalającą nas dokładnością.

Listing 4. Szacowanie długości krzywej Béziera

```
public static class BezierMath
{
    public static (float length, int divisions) EvalLength(
        Func<Vector2[], float, Vector2> bezier,
        Vector2[] points,
        float precision = 0.1f)
    {
        int divisions = 0;

        float lastLength = float.MaxValue;
        bool precisionReached;

        do
        {
            divisions = divisions == 0 ? 1 : divisions * 2;

            float length = 0.0f;

            for (int i = 0; i < divisions; i++)
            {
                var t1 = (float)i / divisions;
                var t2 = (float)(i + 1) / divisions;

                var p1 = bezier(points, t1);
                var p2 = bezier(points, t2);

                length += (p2 - p1).Length();
            }

            precisionReached =
                Math.Abs(length - lastLength) < precision;
            lastLength = length;
        }
        while (!precisionReached);

        return (lastLength, divisions);
    }
}
```

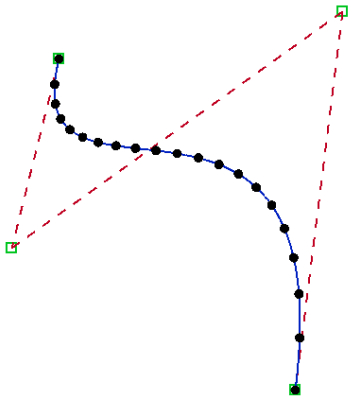
Po uruchomieniu powyższej metody dla przykładowej krzywej otrzymałem następujące wyniki:

Dokładność	Długość	Liczba segmentów	Liczba podziałów
0,1	474,9997	64	5
0,01	475,0245	256	8
0,001	475,02634	1024	10

Tabela 1. Wyniki pomiaru długości krzywej

Miejmy na uwadze, że jednostką długości są piksele – obliczenie długości z dokładnością do 0.1 piksela powinno wystarczyć w przeważającej liczbie przypadków.

Na podstawie osiągniętych wyników można byłoby teraz wysnuć wniosek, że wystarczy podzielić krzywą na 475 odcinków, by każdy z nich był krótszy od piksela. Niestety, również i to nie jest prawdą. Przyczyną takiego stanu rzeczy jest fakt, iż krzywe Béziera mają zmienną prędkość. Wyrenderujmy raz jeszcze naszą przykładową krzywą, wstawiając 20 markerów rozmieszczonych równomiernie względem wartości t (czyli w punktach 0, 0.05, 0.1, …, 0.85, 0.9, 1).



Rysunek 6. Zmienna prędkość krzywej

Widać wyraźnie, jak krzywa na początku zwalnia, by potem przyspieszyć i osiągnąć swoją maksymalną prędkość na końcowym odcinku. Jeżeli więc podzielimy ją na 475 odcinków, okaże się, że najkrótszy mierzyć będzie 0,6344416 piksela, zaś najdłuższy – aż 2,542091.

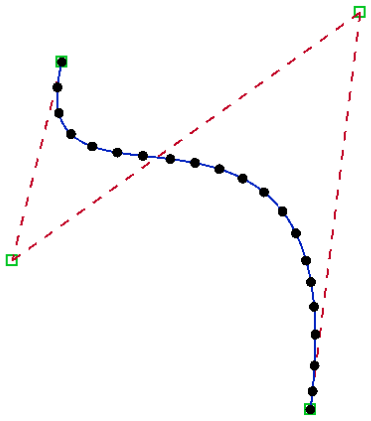
Wbrew pozorom nie jest to jednak wcale aż tak wielki problem w kontekście renderowania. Zauważmy bowiem, że w miejscach, w których odcinki są najdłuższe, krzywa „porusza się" z największą prędkością, co w efekcie bardzo ją w tym miejscu wypłaszcza. Wyrenderowanie jej z mniejszą granularnością nie wpłynie więc zbytnio na końcowy efekt.

Zmienna prędkość krzywej Béziera jest oczywiście również problemem w sytuacji, w której chcielibyśmy wykorzystać ją jako tor ruchu jakiegoś obiektu przemieszczającego się z jednostajną prędkością. Z uwagi na brak analitycznych rozwiązań pozostaje nam stablicowanie długości segmentów i wyznaczenie odpowiednich wartości

t ręcznie. Co – dodajmy – nie jest wcale najłatwiejszym zadaniem, w dużej mierze z powodu nawarstwiających się błędów numerycznych (Rysunek 7).

Listing 5. Równomierne rozmieszczenie markerów na krzywej (fragment metody)

```
int markerCount = 20;
(float length, int divisions) = BezierMath.EvalLength(
    Bezier.Evaluate, points, 0.1f);

float[] fromStart = new float[divisions];
float sum = 0.0f;
for (int i = 0; i < divisions; i++)
{
    var t1 = (float)i / divisions;
    var t2 = (float)(i + 1) / divisions;

    var p1 = Bezier.Evaluate(points, t1);
    var p2 = Bezier.Evaluate(points, t2);

    sum += (p2 - p1).Length();
    fromStart[i] = sum;
}

var marker = -1;
for (int i = 0; i < fromStart.Length; i++)
{
    var newMarker = (int)(markerCount *
        (fromStart[i] / sum));
    if (newMarker > marker)
    {
        var t = (float)i / (fromStart.Length - 1);
        var p = Bezier.Evaluate(points, t);
        g.FillEllipse(markerBrush, new RectangleF(
            p.X - 5, p.Y - 5, 10, 10));
        marker = newMarker;
    }
}
```



Rysunek 7. (Bardziej) równomiernie rozmieszczone markery

Nieco bardziej precyzyjne wyniki osiągniemy, jeżeli podzielimy krzywą na jeszcze większą liczbę segmentów.

ALGORYTM DE CASTELJEAU

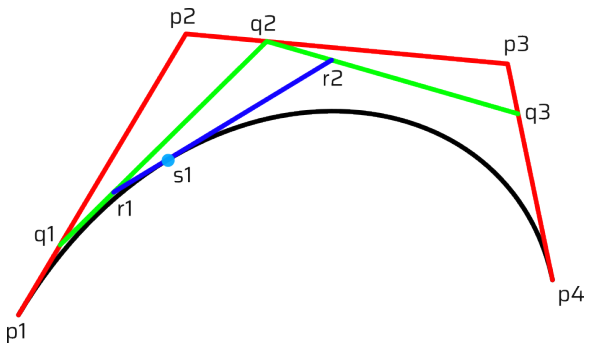
Krzywe Béziera otrzymały swoją nazwę od nazwiska pracownika Renault, ponieważ to właśnie ta firma jako pierwsza zdecydowała się opublikować wyniki jego badań. Na szczęście jednak Paul de Faget de Casteljau nie został zapomniany, a to za przyczyną wymyślonego przez niego bardzo ciekawego algorytmu, pozwalającego na wyznaczenie punktów krzywej w alternatywny, geometryczny sposób.

Załóżmy, że mamy dane cztery punkty kontrolne krzywej Beziera, p_1, p_2, p_3 oraz p_4 oraz wartość parametru t, wskazującą na konkretny punkt na krzywej, który chcemy wyznaczyć.

Konstruujemy teraz trzy odcinki łączące kolejne punkty kontrolne, a następnie wyznaczamy na nich punkty stanowiące podział tych odcinków w miejscu wyznaczonym parametrem t (czyli jeśli t = 0, będzie to punkt początkowy odcinka, dla t =1 – punkt końcowy, t = 0,5 oznacza środek odcinka itp.).

Tym sposobem otrzymamy trzy nowe punkty: nazwijmy je q_1, q_2 oraz q_3. Powtarzamy teraz cały proces, budując dwa odcinki pomiędzy tymi punktami, a następnie dzieląc je znów w miejscu wyznaczonym przez parametr t; teraz otrzymaliśmy dwa punkty: r_1 oraz r_2. Na koniec dzielimy ostatni odcinek, otrzymując punkt s_1, równoważny szukanemu punktowi na krzywej dla zadanego parametru t.

Na Rysunku 8 przedstawiono symbolicznie realizację algorytmu de Casteljeau dla wartości t = 0,25.



Rysunek 8. Symulacja algorytmu de Casteljeau

Można byłoby zapytać: po co stosować algorytm de Casteljeau, jeśli wystarczy wstawić parametr t do wzoru krzywej, by natychmiast otrzymać odpowiedni punkt?

Cóż, jeśli zależy nam tylko na wyznaczeniu punktu na krzywej, to oczywiście w takim przypadku łatwiej jest zastosować wzór. Przy pomocy algorytmu de Casteljeau możemy jednak nie tylko wyznaczać punkty na krzywej, ale również dzielić ją na mniejsze w wyznaczonym miejscu!

Spójrzmy na Rysunek 8. Punktami kontrolnymi oryginalnej krzywej są oczywiście p_1, p_2, p_3 i p_4. Gdybyśmy teraz chcieli podzielić ją w punkcie t=0,25, wystarczy zrealizować algorytm de Casteljeau, by otrzymać dwa nowe zestawy nowych punktów kontrolnych: p_1, q_1, r_1 oraz s_1 dla pierwszej krzywej oraz s_1, r_2, q_3 oraz p_4 dla drugiej. Sumą dwóch krzywych wyznaczonych tymi punktami kontrolnymi będzie wówczas nasza oryginalna krzywa.

NOTACJA SVG

Przyjrzeliśmy się już krzywym Béziera w skali mikro, a teraz popatrzmy jeszcze na nie przez chwilę w skali makro.

Jak wspomniałem na wstępie, krzywe Béziera są obecne bodaj w każdej aplikacji związanej w jakikolwiek sposób z grafiką. W praktyce może więc zajść sytuacja, w której efekty działania naszego programu chcielibyśmy zaimportować do takiej aplikacji. Jednym z najprostszych sposobów osiągnięcia tego celu jest skorzystanie z otwartego, opartego na XML i szeroko zaimplementowanego formatu SVG, który pozwala w stosunkowo łatwy sposób zapisać wektorowe obrazy – a co za tym idzie, również i krzywe.

Przyjrzyjmy się na początku ogólnej strukturze pliku SVG:

Listing 6. Prosty plik SVG

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg
   xmlns="http://www.w3.org/2000/svg"
   xmlns:svg="http://www.w3.org/2000/svg"
   width="32" height="32" viewBox="0 0 32 32"
   version="1.1" id="svg5">
  <defs id="defs1" />
  <g id="layer1">
    <path style="fill:#80a0e0;fill-opacity:1;stroke:none"
          d="M 4 4 L 28 4 L 28 28 L 4 28 Z"
          id="path2" />
  </g>
</svg>
```

Sam format jest zaskakująco prosty – przypomina w dużym stopniu HTML i CSS, tyle że wyspecjalizowany jest oczywiście w kierunku zapisywania obrazów wektorowych. Co zainteresuje nas jednak tym razem najbardziej, to specjalny sposób zapisu kształtu widoczny w atrybucie `d` węzła `path`. Jest to mini-język służący do opisu wektorowych kształtów, zarówno otwartych, jak i zamkniętych, obecny w wielu różnych miejscach, niekoniecznie związanych z formatem SVG (jest on na przykład respektowany przez obiekt `PathData` w ramach frameworka WPF).

Język ten cechują dwa kluczowe aspekty. Po pierwsze, składa się on z serii komend, z których każda odpowiedzialna jest za opisanie kolejnego elementu kształtu. Po drugie zaś, znany jest on z usankcjonowania wielu skrótów, by zapewnić maksymalną zwięzłość zapisu. Na przykład ciąg, który nieco bardziej *explicite* zapisałem w zaprezentowanym wcześniej przykładzie, można skrócić do `M 4 4 28 4 28 28 4 28 Z` lub nawet jeszcze krócej, do `M4,4H28V28H4Z`.

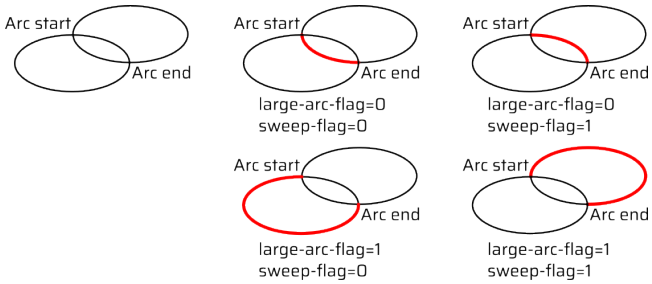
Każde polecenie (za wyjątkiem kilku skrótów) zaczyna się literą określającą rodzaj graficznego składnika, a następnie szeregiem parametrów. Wszystkie liczby są traktowane jako zmiennoprzecinkowe, ze znakiem kropki jako separatora dziesiętnego. Elementy (polecenia i liczby) mogą być oddzielane spacją lub przecinkiem, ale zgodnie ze standardem odstępy są obowiązkowe tylko tam, gdzie jest to naprawdę konieczne (zwykle gdy mamy obok siebie dwie liczby). W każdym innym przypadku służą one tylko zwiększeniu czytelności zapisu.

Każde polecenie może być wydane przy pomocy wielkiej lub małej litery. Wielka litera oznacza, że współrzędne są bezwzględne, zaś mała – że są przyrostowe w stosunku do miejsca, w którym zakończyła się ostatnia komenda (dlatego wielkość liter w opisywanych poniżej poleceniach jest istotna!).

Do dyspozycji mamy kilka klas poleceń. Kontrolne (`Mm`, `Zz`), linie (`Ll`, `Hh`, `Vv`), krzywa Béziera drugiego stopnia (`Cc`, `Ss`), krzywa Béziera trzeciego stopnia (`Qq`, `Tt`) oraz łuk eliptyczny (`Aa`).

W kolejności, możemy wydawać następujące polecenia:

`M x y [x y ...]` `m dx xy [dx dy ...]`	Rozpoczęcie nowego pod-kształtu od zadanych współrzędnych. Jeżeli po poleceniu `M` zostanie wprowadzonych więcej współrzędnych, wszystkie kolejne będą traktowane jako niejawne polecenia `L` lub `l`. Jeśli pierwszym poleceniem całego kształtu jest `m`, współrzędne będą traktowane jako bezwzględne, ale ewentualne kolejne współrzędne będą traktowane jako niejawne polecenia `l`
`Z` `z`	Zamyka bieżący pod-kształt, łącząc go z jego pierwszym punktem. Ponieważ `Z` nie przyjmuje parametrów, obie jego wersje działają tak samo
`L x y [x y ...]` `l dx dy [dx dy ...]`	Rysuje prostą linię rozpoczynającą się od miejsca, w którym zakończyło się poprzednie polecenie, do miejsca wskazanego współrzędnymi
`H x [x ...]` `h dx [dx ...]`	Rysuje poziomą linię rozpoczynającą się od miejsca, w którym zakończyło się poprzednie polecenie, do współrzędnej `x` wskazanej parametrem. Możliwe jest przekazanie wielu współrzędnych, choć zwykle nie ma to większego sensu
`V y [y ...]` `v dy [dy ...]`	Rysuje pionową linię rozpoczynającą się od miejsca, w którym zakończyło się poprzednie polecenie, do współrzędnej `y` wskazanej parametrem. Możliwe jest przekazanie wielu współrzędnych, choć zwykle nie ma to większego sensu
`C x1 y1 x2 y2 x y [...]` `c dx1 dy1 dx2 dy2 dx dy [...]`	Rysuje krzywą Béziera trzeciego stopnia. Pierwszym punktem jest miejsce, w którym zakończyło się poprzednie polecenie, zaś `x1 y1`, `x2 y2` i `x y` wyznaczają miejsce pozostałych trzech punktów kontrolnych. Co ważne, wszystkie relatywne współrzędne są wyznaczane w stosunku do poprzedniego punktu (a nie względem siebie!)
`S x2 y2 x y [...]` `s dx2 dy2 dx dy [...]`	Rysuje krzywą Béziera trzeciego stopnia. Pierwszym punktem kontrolnym jest miejsce, w którym zakończyło się poprzednie polecenie. Drugi punkt kontrolny wyznaczany jest jako lustrzane odbicie przedostatniego punktu kontrolnego poprzedniej krzywej. Jeśli poprzednim poleceniem nie była krzywa, drugi punkt kontrolny pokryje się z pierwszym. Dwa pozostałe punkty kontrolne definiowane są parametrami `x2 y2` oraz `x y`
`Q x1 y1 x y [...]` `q dx1 dy1 dx dy [...]`	Rysuje krzywą Béziera drugiego stopnia. Pierwszy punkt kontrolny pokrywa się z miejscem, w którym zakończyło się poprzednie polecenie, zaś dwa pozostałe wskazane są przez parametry
`T x y [...]` `t dx dy [...]`	Rysuje krzywą Béziera drugiego stopnia. Pierwszy punkt kontrolny pokrywa się z miejscem, w którym zakończyło się poprzednie polecenie, drugi obliczany jest tak samo, jak w poleceniach `S` i `s`. Trzeci punkt kontrolny wskazany jest parametrem
`A rx ry x-axis-rotation large-arc-flag sweep-flag x y [...]` `a rx ry x-axis-rotation large-arc-flag sweep-flag dx dy [...]`	Rysuje fragment łuku eliptycznego do punktu wskazanego współrzędnymi `x` oraz `y`. Rozmiar i orientacja elipsy wskazana jest dwoma promieniami `rx` oraz `ry` (środek elipsy obliczany jest automatycznie na bazie przekazanych parametrów). `X-axis-rotation` określa, o jaki kąt obrócona jest oś `X` elipsy w stosunku do bieżącego układu współrzędnych. Wreszcie `large-arc-flag` oraz `sweep-flag` określają, który z czterech możliwych łuków ma zostać wybrany (Rysunek 9)



Rysunek 9. Działanie parametrów large-arc-flag oraz sweep-flag (źródło: https://www.w3.org/TR/SVG2/paths.html#DProperty)

Przykład

Zabrałem się ostatnio za standaryzowanie ikon wszystkich moich aplikacji. Zainspirowałem się znalezionym gdzieś w Internecie rysunkiem sześciokątów zbudowanych z trójkątów, które tworzą wielokierunkowe gradienty. Oczywiście dałoby się narysować takie ikony w Inkscape, ale zależało mi na tym, żeby wygenerowanie kolejnych – dla następnych pisanych przeze mnie aplikacji – zajmowało możliwie jak najmniej czasu, więc napisałem program, który buduje je na podstawie zbioru parametrów.

Dla przykładu, poniższy fragment kodu prezentuje generowanie ciągu opisującego duży sześciokąt z zaokrąglonymi wierzchołkami, który stanowi maskę dla całego obrazu.

Listing 7. Automatyczne generowanie wektorowych kształtów

```
private static string BuildRoundedHexagon(IconModel icon)
{
    StringBuilder result = new();

    Vector2 start = icon.Start.AsVector;
    Vector2 span = icon.End.AsVector - icon.Start.AsVector;
    Vector2 horizontal = span / 2.0f;
    Vector2 upwards = horizontal.RotateLeft(60);
    Vector2 downwards = horizontal.RotateRight(60);

    List<(Vector2 point, Vector2 span)> edges = new();
    edges.Add((start, downwards));
    edges.Add((edges.Last().point + edges.Last().span,
        horizontal));
    edges.Add((edges.Last().point + edges.Last().span,
        upwards));
    edges.Add((edges.Last().point + edges.Last().span,
        -downwards));
    edges.Add((edges.Last().point + edges.Last().span,
        -horizontal));
    edges.Add((edges.Last().point + edges.Last().span,
        -upwards));

    result.AppendLine($"    <path style=\"fill:#ffffff;" +
        "fill-opacity:1;stroke:none\"");
    result.Append("          d=\"");

    Vector2 p;

    for (int i = 0; i < edges.Count; i++)
    {
        var current = edges[i];
        var next = edges[(i + 1) % edges.Count];

        if (i == 0)
            result.Append("M ");

        p = current.point +
            current.span * icon.RoundingEdgeFactor;
        result.Append(FormattableString.Invariant(
            $"{p.X},{icon.Height - p.Y} "));

        if (i > 0)
            result.Append("L ");

        p = current.point +
            current.span * (1.0f - icon.RoundingEdgeFactor);
        result.Append(FormattableString.Invariant(
            $"{p.X},{icon.Height - p.Y} "));

        result.Append("C ");

        p = current.point +
            current.span * (1.0f -
                icon.RoundingControlPointFactor);
        result.Append(FormattableString.Invariant(
            $"{p.X},{icon.Height - p.Y} "));

        p = next.point +
            next.span * icon.RoundingControlPointFactor;
        result.Append(FormattableString.Invariant(
            $"{p.X},{icon.Height - p.Y} "));
    }

    p = edges[0].point +
        edges[0].span * icon.RoundingEdgeFactor;

    result.AppendLine(FormattableString.Invariant(
        $"{p.X},{icon.Height - p.Y} Z\""));
    result.AppendLine($" id=\"path{idGenerator++}\" />");

    return result.ToString();
}
```

Końcowy efekt działania całego programu – nową ikonę mojej aplikacji Dev.Editor – możemy zobaczyć na Rysunku 10.

Rysunek 10. Kształty wygenerowane z poziomu aplikacji C#

NA KONIEC

W świetle faktu, iż większość szanujących się bibliotek graficznych pozwala na renderowanie krzywych Béziera, zasadnym jest zadać sobie pytanie, czy implementowanie ich od zera ma jakikolwiek sens.

Odpowiedzią jest bodaj najbardziej uniwersalne stwierdzenie w świecie IT: to zależy. Jeśli chcemy tylko renderować krzywe otwarte lub zamknięte, prawdopodobnie nie ma większego sensu zaprzątać sobie głowy implementowaniem wszystkiego od początku. Czasami jednak krzywych Béziera możemy chcieć użyć do innych celów niż tylko rysowanie rastrowych obrazów: w takim przypadku warto przyswoić sobie nieco matematycznych podstaw i wykorzystać ten wszechstronny matematyczny konstrukt, by znacząco ułatwić sobie pracę. Otwartą opcją pozostaje również skorzystanie z gotowych mechanizmów – jak format SVG i język opisu wektorowych kształtów – by zrealizować najtrudniejszą część operacji przy pomocy napisanego przez siebie programu, a resztę pracy wykonać w dedykowanej, przeznaczonej do tego aplikacji.

WOJCIECH SURA

wojciechsura@gmail.com

Programuje 30 lat, z czego 15 komercyjnie; ma na koncie aplikacje desktopowe, webowe, mobilne i wbudowane – pisane w C#, C++, Javie, Delphi, PHP, JavaScript i w jeszcze kilku innych językach. Obecnie pracuje w SII – największym w Polsce dostawcy usług doradztwa technologicznego, transformacji cyfrowej, Business Process Outsourcing i inżynierii.

Z ARCHIWUM CVE | PRAKTYCZNE ZASTOSOWANIE LLM HOSTOWANEGO SAMODZIELNIE

INDEX: 285358

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

programista

1/2024 (111)

Cena 28,90 zł (w tym VAT 8%)

SIECI NEURONOWE OD PODSTAW

JUŻ W EMPIKACH

Kolory (dla opornych)

Kafka w .NET i Docker

Kierowanie kierownikiem

Passkey, klucze fizyczne i passwordless

HAL9000 @ localhost, czyli programujemy lokalne LLMy

ISSN 2084-9400

9 772084 940404 01