UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY
A INFORMATIKY

BAKALÁŘSKÁ PRÁCE

2009 Jan Novotný

Univerzita Pardubice

Fakulta Elektrotechniky a Informatiky

Bézierovy křivky

Jan Novotný

Bakalářská práce

2009

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Katedra informačních technologií
Akademický rok: **2008/2009**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Jan NOVOTNÝ**

Studijní program: **B2646 Informační technologie**

Studijní obor: **Informační technologie**

Název tématu: **Bézierovy křivky**

Zásady pro vypracování:

Teoretická část: Nadefinovat a popsat vlastnosti Bezierových křivek. Beziérovy křivky v Matlabu (popis implementovaných algoritmů)
Implementační část: Prezentovat konkrétní užití Beziérových křivek počítačové grafice. Vytvoření programu pro výpočet a zobrazení Beziérových křivek v prostředí Delphi. Porovnání časové náročnosti vytvořeného programu a matlabovských algoritmů.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

J.Kobza: Splajny, Skriptum UP, Olomouc 1993 F.Drdla: Metody modelování křivek a ploch v počítačové geometrii, Skriptum UP, Olomouc 1992 ***Dušek F.,: Matlab a Simulink - úvod do používání, Pardubice 2000*** **Moderní počítačová grafika, Jiří Žára, Bedřich Beneš, Jiří Sochor, Petr Felkel, vydáno: leden 2005, Nakladatel: Computer Press, ISBN 80-251-0454-0 Výpočty a simulace v programech Matlab a Simulink, Pavel Karban, vydáno: prosinec 2006, Nakladatel: Computer Press, ISBN 978-80-251-1448-3 Základy teorie splinů, Karel Najzar, vydáno: prosinec 2006, Nakladatel: Karolinum, ISBN 80-246-1287-9 Handbook on Splines for the User, E. V. Shikin, Alexander I. Plis, vydáno: 1995, Nakladatel: CRC Press, ISBN 084939404X**

Vedoucí bakalářské práce: **Mgr. Jana Heckenbergerová**
Katedra informačních technologií

Datum zadání bakalářské práce: **15. ledna 2009**

Termín odevzdání bakalářské práce: **15. května 2009**

Univerzita Pardubice Česká republika

L.S.

doc. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan
vedoucí katedry

V Pardubicích dne 31. března 2009

Obsah

Seznam obrázků

Souhrn

Tato práce je věnována algoritmům pro výpočet Bézierových křivek. Cílem této práce je implementovat tyto algoritmy do prostředí Matlabu. Dále porovnat časy potřebné k výpočtu bodů křivky v Matlabu s programem v Delphi.

Klíčová slova

Bézierovy křivky, de Casteljau, Matlab, Bernsteinovy polynomy

Title

Bézier curves

Abstract

This work is dedicated to algorithms used for computation of Bézier Curves. The purpose of this work is to implement these algorithms in to a Matlab. Then comparison of time spent with computation in Matlab, with time spent in Delphi program.

Keywords

Bézier curves, de Casteljau, Matlab, Bernstein polynomials

1 Úvod

Křivky používané v počítačové grafice je možné rozdělit podle způsobu interpretace řídících bodů, které určují tvar křivek, na dva druhy: interpolační a aproximační. Interpolační křivky řídícími body procházejí zatímco aproximační křivky jimi procházet můžou, ale i nemusí.

Mezi nejznámější křivky v počítačové grafice patří Bézierovy křivky, pojmenované po Dr. Pierre Etienne Bézierovi. Tyto křivky byly představeny v teoretické matematice dávno před počítači – stáli za nimi francouzský matematik Charles Hermite a jeho ruský kolega Sergej Bernstein. Křivky, které využívají Bernsteinových polynomů, se staly známé v počítačové grafice díky práci pana Béziera a Paula de Casteljau. Pierre Bézier i Paul de Casteljau pracovali v automobilovém průmyslu – první z nich pracoval u firmy Renault, druhý pro firmu Citroen, kde tyto křivky používali za účelem vývoje automobilových karoserií.

Bézierovy křivky jsou běžně používány v grafických programech jako jsou Adobe Photoshop, Adobe Illustrator, Gimp nebo AutoCAD. Bézierovy křivky 2. stupně jsou používány pro definici fontů True Type, bézierovy kubiky – křivky 3. stupně jsou používány pro definici fontů Type 1. Křivky 3. stupně se rovněž používají pro definici tvarů písma v programovacím jazyku Metafont.

V této práci se budu nejprve věnovat definici těchto křivek. Na několika příkladech předvedu vlastnosti Bézierových křivek konkrétního stupně. V další kapitole pak definuji seznam vlastností obecných Bézierových křivek. Poté uvedu algoritmy, které se používají pro jejich výpočet a jejich příklady v pseudokódu. Nakonec popíši mou implementaci těchto algoritmů do Matlabu.

2 Formulace křivky

Křivky mohou být formulovány buď explicitně, implicitně nebo parametricky. Jak si ukážeme, ne všechny formulace jsou vhodné pro počítačovou grafiku.

Explicitní formulování křivky v dvourozměrném euklidovském prostoru má tvar: $y = f(x)$. Toto zadání křivky lze použít jen pro křivky, které jsou zároveň funkcí – tedy křivky, u kterých pro jednu hodnotu parametru x existuje právě jedna hodnota parametru y. To znamená, že by nebylo křivkou možné charakterizovat tvary jako tvar osmičky, apod.

Implicitní formulování má tvar: $F(x, y) = 0$. Ani takové zadání křivky není příliš vhodné pro použití v počítačové grafice, protože neumožňuje postupný výpočet křivky. Toto zadání má svůj význam při hledání průsečíků přímky s křivkou.

Poslední způsob formulace křivky, tedy parametrický tvar, je v počítačové grafice nejpoužívanější. Křivka je zde chápána jako dráha pohybujícího se bodu. Souřadnice tohoto bodu je určena funkcí parametru t (času). Tento parametr nabývá hodnot z intervalu $t \in \langle t_{\min}, t_{\max} \rangle$, který je většinou volen v rozsahu $t \in \langle 0,1 \rangle$.

3 Obecná Bézierova křivka n-tého stupně

Bézierova křivka n-tého stupně je zadána pomocí n+1 bodů, které tvoří kontrolní (řídící) polygon $P = \{P_0, ..., P_n\}$ funkce. Výsledná křivka se nachází v konvexním obalu tohoto polygonu. Vychází z bodu P_0 ve směru přímky P_0P_1 a končí v bodě P_n, kde má směr přímky $P_{n-1}P_n$. Svým tvarem mimo okrajové body napodobuje tvar řídícího polygonu. Rovnicí obecné Bézierovy křivky stupně n, zadané řídícím polygonem $P = \{P_0, ..., P_n\}$ na intervalu $t \in \langle 0,1 \rangle$, je parametrická křivka $v(t)$, která je dána předpisem:

$$v(t) = \sum_{i=0}^{n} B_i^n(t) P_i ; t \in \langle 0,1 \rangle ; i = 0, ..., n \qquad (1)$$

Kde B_i^n jsou Bernsteinovy polynomy n-tého stupně, které jsou dány předpisem:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} ; t \in \langle 0,1 \rangle ; i = 0, ..., n \qquad (2)$$

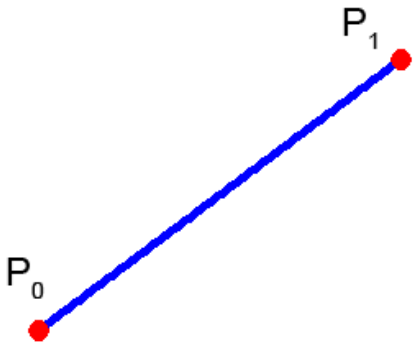
Bernsteinovy polynomy mají tyto vlastnosti:

1. $n \in N \cap n > 0 \cap i \in \langle 0, n \rangle \cap t \in \langle 0,1 \rangle \Rightarrow B_i^n(t) \geq 0$ - polynomy jsou nezáporné
2. $\sum_{i=0}^{n} B_i^n(t) = 1; t \in \langle 0,1 \rangle$ - tato vlastnost zaručuje, že výsledná křivka bude vůči kontrolnímu polynomu konvexní
3. $B_i^n(t) = (1-t) \cdot B_i^{n-1}(t) + t \cdot B_{i-1}^{n-1}(t)$ - rekurentní definice Bernsteinova polynomu n-tého stupně pomocí lineární kombinace dvou polynomu stupně n-1; této vlastnosti se využívá v algoritmu de Casteljau. [1.]

4 Příklady Bézierových křivek

4.1 Křivka 1. stupně – lineární Bézierova křivka

Nejjednodušším příkladem Bézierovy křivky je křivka 1. stupně. Taková křivka je zadána dvěma body. Pokud tedy zvolíme dva libovolné body P_0 a P_1 uvidíme, že Bézierova křivka bude přímkou spojující tyto dva body.



Obrázek 4.1 – křivka 1. stupně

Dosadíme-li do rovnice dostaneme:

$$v(t) = B_0^1(t)P_0 + B_1^1(t)P_1,$$

Bernsteinovy polynomy 1. stupně, pro i=0 a i=1 vypočítáme takto:

$$B_0^1(t) = \binom{1}{0} t^0 (1-t)^1 = (1-t)$$

$$B_1^1(t) = \binom{1}{1} t^1 (1-t)^{1-1} = t$$

Po dosazení do parametrické rovnice křivky dostaneme:

$v(t) = (1-t)P_0 + tP_1$.

Pro krajní hodnoty parametru t z intervalu $t \in \langle 0,1 \rangle$ nabývá funkce hodnot:

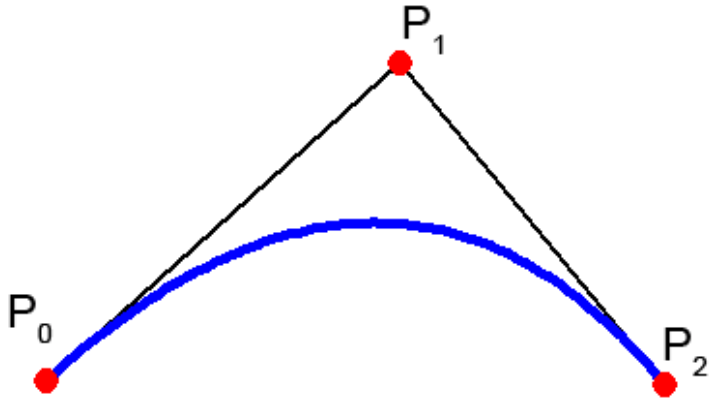
$v(0) = P_0$

$v(1) = P_1$.

Z výše uvedeného vidíme, že funkce prochází zadanými body. Pro hodnoty parametru, které jsou mezi krajními hodnotami, nabývá funkce $v(t)$ hodnot váženého průměru mezi body P_0 a P_1, kde má bod P_0 váhu $1-t$ a bod P_1 váhu t. Pro hodnotu parametru $t = 1/2$ má tak funkce $v(t)$ hodnotu $v\left(\frac{1}{2}\right) = \left(1-\frac{1}{2}\right)P_0 + \frac{1}{2}P_1 = \frac{1}{2}(P_0 + P_1)$, tedy bod, který leží v polovině úsečky dané body P_0 a P_1.

4.2 Křivka 2.stupně – kvadratická Bézierova křivka

Tato křivka je zadána kontrolním polygonem se třemi body - P0, P1, P2.



Obrázek 4.2 – křivka 2. stupně

Její parametrická rovnice má tvar:

$$v(t) = B_0^2(t)P_0 + B_1^2(t)P_1 + B_2^2(t)P_2$$

$$B_0^2(t) = \binom{2}{0} t^0 (1-t)^2 = (1-t)^2$$

$$B_1^2(t) = \binom{2}{1} t^1 (1-t)^1 = 2t(1-t)$$

$$B_2^2(t) = \binom{2}{2} t^2 (1-t)^0 = t^2$$

$$v(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2$$

Pro t=0 a t=1, tedy okraje křivky pak funkce nabývá hodnot:

$$v(0) = P_0$$

$$v(1) = P_2 .$$

První derivace parametrické rovnice křivky 2. stupně:

$$v'(t) = -2(1-t)P_0 + 2(1-t)P_1 - 2tP_1 + 2tP_2 = 2\left[-(1-t)P_0 + (1-t)P_1 - tP_1 + tP_2\right] =$$
$$= 2\left[(1-t)(P_1 - P_0) + t(P_2 - P_1)\right]$$

První derivace pro parametr t=0:

$$v'(0) = 2\left[(1-0)(P_1 - P_0) + 0(P_2 - P_1)\right] = 2(P_1 - P_0)$$

Směrnice tečny k funkci $v(t)$ v bodě P_0 má směr úsečky P_0,P_1 a dvojnásobnou velikost. První derivace pro parametr t=1:

$$v'(1) = 2\left[(1-1)(P_1 - P_0) + 1(P_2 - P_1)\right] = 2(P_2 - P_1)$$

Směrnice tečny k funkci $v(t)$ v bodě P_1 má směr úsečky P_2,P_1 a dvojnásobnou velikost.

4.3 Křivka 3.stupně – Bézierova kubika

V počítačové grafice asi nejčastější typ Bézierovy křivky. Je zadána čtyřmi body – P_0, P_1, P_2, P_3. Parametrická rovnice takové křivky má tvar:

$$v(t) = B_0^3(t)P_0 + B_1^3(t)P_1 + B_2^3(t)P_2 + B_3^3(t)P_3$$

$$B_0^3(t) = \binom{3}{0} \cdot t^0 (1-t)^3 = (1-t)^3$$

$$B_1^3(t) = \binom{3}{1} \cdot t^1 (1-t)^2 = 3t \cdot (1-t)^2$$

$$B_2^3(t) = \binom{3}{2} \cdot t^2 (1-t)^1 = 3t^2 \cdot (1-t)$$

$$B_3^3(t) = \binom{3}{3} \cdot t^3 (1-t)^0 = t^3$$

$$v(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_2$$

První derivace parametrické rovnice křivky 3. stupně:

$$v'(t) = -3(1-t)^2 P_0 + 3(1-t)^2 P_1 - 6t(1-t)P_1 + 6t(1-t)P_2 - 3t^2 P_2 + 3t^2 P_3 =$$
$$3\left[(1-t)^2 (P_1 - P_0) + 2t(1-t)(P_2 - P_1) + t^2 (P_3 - P_2)\right]$$

První derivace pro parametr t=0:

$$v'(0) = 3 \cdot \left[(1-0)^2 \cdot (P_1 - P_0) + 2 \cdot 0 \cdot (1-0) \cdot (P_2 - P_1) + 0^2 \cdot (P_3 - P_2)\right] = 3 \cdot (P_1 - P_0)$$

První derivace pro parametr t=1:

$$v'(1) = 3 \cdot \left[(1-1)^2 \cdot (P_1 - P_0) + 2 \cdot 1 \cdot (1-1) \cdot (P_2 - P_1) + 1^2 \cdot (P_3 - P_2)\right] = 3 \cdot (P_3 - P_2)$$

Z těchto rovnic opět vyplývá, že směrnice tečny v okrajových bodech křivky odpovídá směru úseček tvořených krajními body řídícího polygonu. Dále můžeme vidět, že velikost směrnice bude n-násobná vůči velikosti úsečky tvořené krajními body.

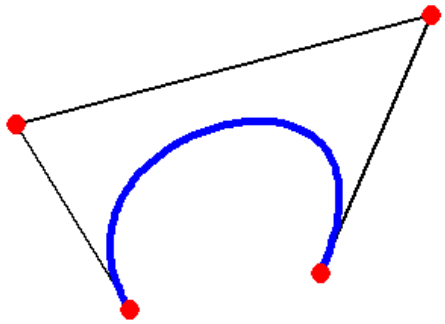
5 Vlastnosti Bézierových Křivek

Na základě předchozích odstavců můžeme definovat vlastnosti obecné Bézierovy křivky:

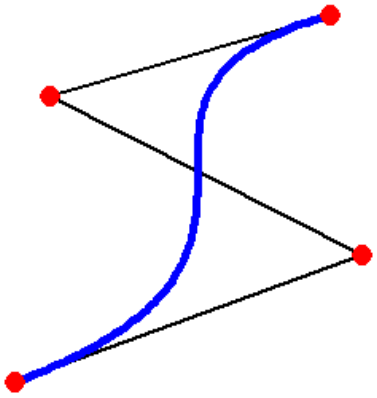
1. Křivka n-tého stupně začíná v prvním bodě kontrolního polygonu $P = \{P_0, ..., P_n\}$; tedy v bodě P_0. První derivace křivky v tomto bodě odpovídá n-násobku vzdálenosti krajních bodů řídícího polygonu - P0,P1. Platí vztah: $\left[\frac{dv}{dt}\right]_{t=0} = n(P_1 - P_0)$.
2. Křivka končí v posledním bodě kontrolního polygonu - bodě P_n. První derivace křivky v tomto bodě odpovídá n-násobku

vzdálenosti krajních bodů řídícího polygonu P_{n-1},P_n. Platí vztah: $\left[\frac{dv}{dt}\right]_{t=1} = n(P_n - P_{n-1})$.

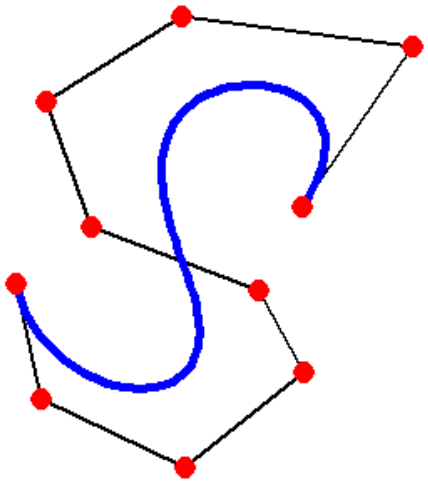
3. Křivka je konvexní vůči bodům řídícího polygonu.
4. Křivka se svým tvarem blíží tvaru řídícího polygonu (Obrázek 5.1, Obrázek 5.2, Obrázek 5.3). Pokud leží všechny body řídícího polygonu na přímce je křivkou úsečka.



Obrázek 5.1 – Bézierova kubika



Obrázek 5.2 – Bézierova kubika



Obrázek 5.3 – křivka9. stupně

5. Křivka bude mít stejný tvar i pokud prohodíme pořadí bodů kontrolního polygonu. $P_0, P_1, ..., P_{n-1}, P_n \longrightarrow P_n, P_{n-1}, ..., P_1, P_0$
6. Změna polohy jednoho bodu kontrolního polygonu má vliv na tvar celé křivky.
7. Počet průsečíku přímky s rovinnou Bézierovou křivkou odpovídá počtu průsečíku přímky s jejím kontrolním polygonem.
8. Hladkého spojení dvou křivek docílíme, pokud budou tečné vektory křivek identické. První a druhý bod následující křivky tak závisí na posledních dvou bodech křivky předchozí.
9. Křivka je invariantní vůči afinní transformaci souřadnic. Tyto transformace křivky lze provádět pomocí transformací kontrolního polygonu a opětovným vykreslením křivky. [8.]
10. Přidání bodů do kontrolního polygonu umožňuje lépe vystihnout požadovaný tvar křivky. Zvýšení stupně křivky má za následek to, že řídící polygon konverguje k výsledné křivce.
11. Odebrání bodů z kontrolního polygonu má za následek snížení náročnosti výpočtů.

6 Algoritmy pro výpočet Bézierových křivek

6.1 Přímý algoritmus

Přímý algoritmus je nejjednodušším algoritmem pro výpočet Bézierovy křivky. Spočívá v postupném dosazování parametru $t \in \langle 0,1 \rangle$ do parametrické rovnice křivky. Výsledkem je množina bodů, které tvoří výslednou křivku.

Příklad algoritmu v pseudokódu pro křivku 3. stupně danou kontrolním polygonem s body P0, P1, P2, P3:

```
//inicializace

t=0; //Parametr t

i=0; //Pomocná proměnná

krok=0.001; //Krok o který zvyšujeme parametr t

While t<=1 do:

Begin

        Krivka[i].X=P[0].X*(1-t)^3+P[1].X*3*t*(1-t)^2+P[2].X*3*t^2*(1-t)+P[3].X*3*t^3;

        Krivka[i].Y=P[0].Y*(1-t)^3+P[1].Y*3*t*(1-t)^2+P[2].Y*3*t^2*(1-t)+P[3].Y*3*t^3;

        t=t+krok;

        i=i+1;

End;

nakresliKřivku(Krivka);
```

Konstantní krok, o který při výpočtu zvyšujeme parametr t, má vliv na to, s jakou přesností bude křivka vypočítána. Výhodou tohoto postupu pro výpočet křivky je to, že díky volbě velikosti kroku dopředu známe počet bodů. Počet bodů můžeme určit jako $pocet = 1/krok$ - pro tento případ kdy jsme velikost kroku zvolili 0.001, je to tedy 1000 bodů, kterými bude výsledná křivka tvořena. Vykreslení křivky probíhá tak, že vypočítané body spojíme úsečkami a dostáváme výslednou křivku.

Tento algoritmus nám umožňuje realizovat výpočty pouze pro křivky zadané čtyřmi body. Pokud by jsme chtěli algoritmus upravit pro obecné Bézierovy křivky stupně n, musíme pro každou hodnotu parametru t a pro každý bod vypočítat hodnotu Bernsteinova polynomu.

Příklad algoritmu pro výpočet obecné Bézierovy křivky stupně n v pseudokódu:

```
Function BernsteinPoly(n:integer; i:integer; t:real):real
//Funkce pro výpočet bernsteinova polynomu
Begin
    KombinacniCis=faktoriál(n)/faktoriál(n-i)*faktoriál(i);
    BernsteinPoly=KombinacniCis*t^2*(1-t)^n-i;
End;
```

Funkce „BernsteinPoly“ je pomocná funkce, která má za účel vypočítat hodnotu Bernsteinova polynomu.

```
Procedure BezierPrimy(n:integer,P:Polygon)
Begin
    //inicializace
    t=0; //Parametr t
    j=0;//Pomocná proměnná
    krok=0.001; //Krok o který zvyšujeme parametr t
    while t<=1 do:
    Begin
        For i=0 to n do:
        Begin
            Krivka[j].X= Krivka[j].X+BernsteinPoly(n,i,t)*P[i].X;
            Krivka[j].Y= Krivka[j].Y+BernsteinPoly(n,i,t)*P[i].Y;
```

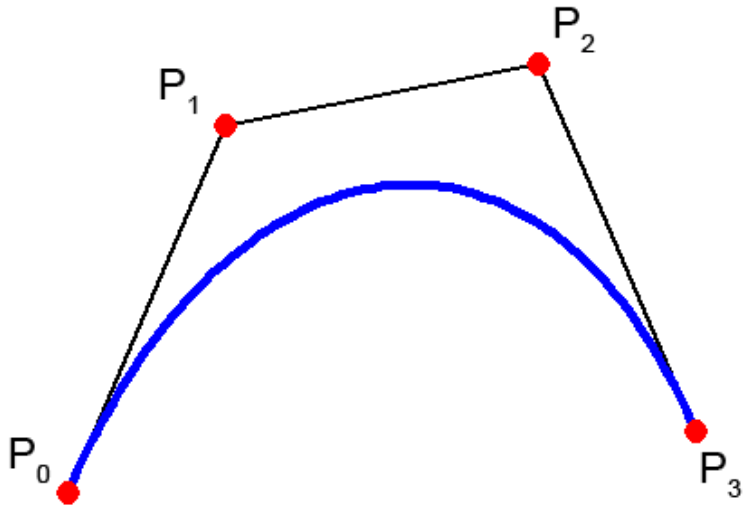
```
                End;

                t=t+krok;

                j=j+1;

        End;

        nakresliKřivku(Krivka);

End;
```

Procedura „BezierPrimy“ obsahuje cyklus, jehož vstupní podmínkou je $t \leq 1$. Pokud je tato podmínka splněna, program vstoupí do cyklu typu „for“, který proběhne od nuly do hodnoty proměnné „n“. V tomto cyklu program pomocí funkce „BernsteinPoly“ postupně vypočítává hodnotu bodu Bézierovy křivky. Po skončení „for“ cyklu je zvýšena hodnota proměnné „t“ a pomocná proměnná „j“. Po skončení „while“ cyklu, který je limitován podmínkou $t \leq 1$, máme v proměnné „Krivka“ uloženy vypočtené body křivky a můžeme jí vykreslit.

6.2 Algoritmus de Casteljau

Tento algoritmus spočívá v přidávání bodů do kontrolního polygonu, dokud není dosaženo přesnosti, požadované k zobrazení křivky. Využívá rekurzivní vlastnosti Bernsteinových polynomů. Jeho vstupem je řídící polygon budoucí křivky, dalším vstupním parametrem může být požadovaná přesnost křivky. Algoritmus vypočítá nový řídící polygon, který se tvarem přibližuje požadované Bézierově křivce.

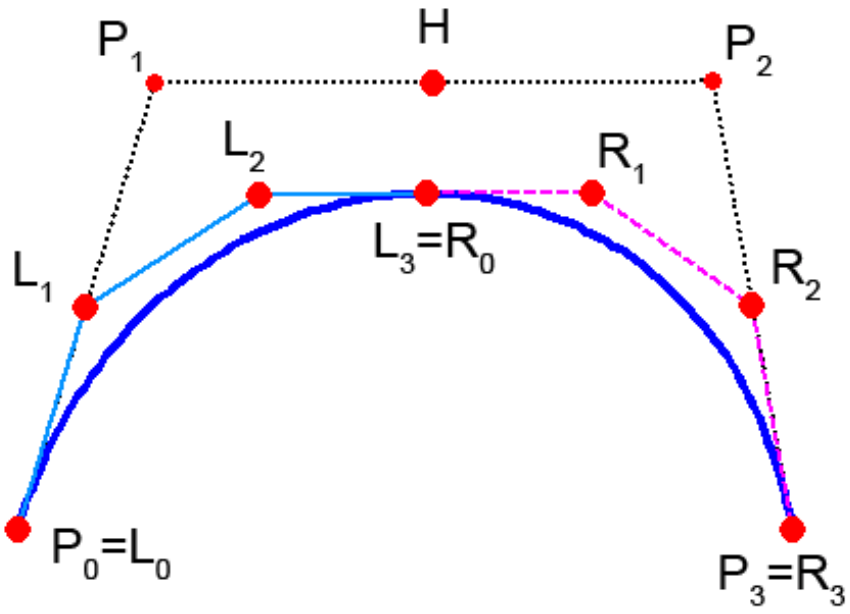
Vezměme Bézierovu kubiku, tedy křivku 3. stupně se 4-mi řídícími body – P_0, P_1, P_2, P_3 (Obrázek 6.1)



Obrázek 6.1 – Bézierova kubika

Pomocí jednoduchého půlení úseček tvořených body řídícího polygonu $P = \{P_0, P_1, P_2, P_3\}$ získáme 8 nových řídících bodů – L_0 až L_3 a R_0 až R_3. Těchto 8 bodů vypočítáme pomocí následujících vztahů:

$$L_0 = P_0$$
$$L_1 = (P_0 + P_1)/2$$
$$H = (P_1 + P_2)/2$$
$$L_2 = (L_1 + H)/2$$
$$R_2 = (P_2 + P_3)/2$$
$$R_1 = (H + R_2)/2$$
$$R_0 = L_3 = (L_2 + R_1)/2$$
$$R_3 = P_3$$



Obrázek 6.2 – de Casteljau – první krok

Získáváme tedy dva nové řídící polygony – levý $L=\{L_0,L_1,L_2,L_3\}$ (na obrázku plnou čarou) a pravý $R=\{R_0,R_1,R_2,R_3\}$ (na obrázku čárkovaně). Poslední bod prvního (levého) polygonu je totožný s prvním bodem druhého (pravého). Vidíme, že tvar těchto polygonů se přibližuje výslednému tvaru požadované Bézierovy křivky. Každý z těchto dvou řídících polygonů může být znovu vstupem algoritmu. Výpočet se opakuje a je dosaženo jemnějšího rozdělení. Algoritmus je zastaven v momentě, kdy je dosaženo požadované přesnosti tvaru křivky. V případě počítačové grafiky to muže být ve chvíli, kdy je vzdálenost dvou bodů menší, nebo rovna velikosti úhlopříčky pixelu. Jako jiné kritérium pro zastavení výpočtu rekurzivního algoritmu můžeme použít velikost plochy řídícího polygonu. Díky použití kritéria pro zastavení výpočtu, produkuje tento algoritmus menší objem dat. Rovné úseky křivky mohou být nahrazovány úsečkou, zatímco v zakřivených částech je rozdělení jemnější.

Příklad v pseudokódu:

```
Procedure Rozdel(RidiciPolygon:Polygon, Levy:Polygon, Pravy:Polygon)

Begin
```

```
        PolovinaRidicihoPolygonu=(RidiciPolygon[1]+RidiciPolygon[2])/2;

        Levy[0]=RidiciPolygon[0];

        Levy[1]=(RidiciPolygon[0]+RidiciPolygon[1])/2;

        Levy[2]=(Levy[1]+PolovinaRidicihoPolygonu)/2;

        Pravy[2]=(RidiciPolygon[2]+RidiciPolygon[3])/2;

        Pravy[1]=(PolovinaRidicihoPolygonu+Pravy[2])/2;

        Pravy[3]=RidiciPolygon[3];

        Levy[3]=Pravy[0]=(Levy[2]+Pravy[1])/2;

End;
```

Procedura „Rozdel“ dostává jako vstup řídící polygon. Výstupem procedury jsou pak dva nově vzniklé polygony levý a pravý. Tyto polygony vypočítáme pomocí vztahů(odkaz na rovnice které jsou výše).

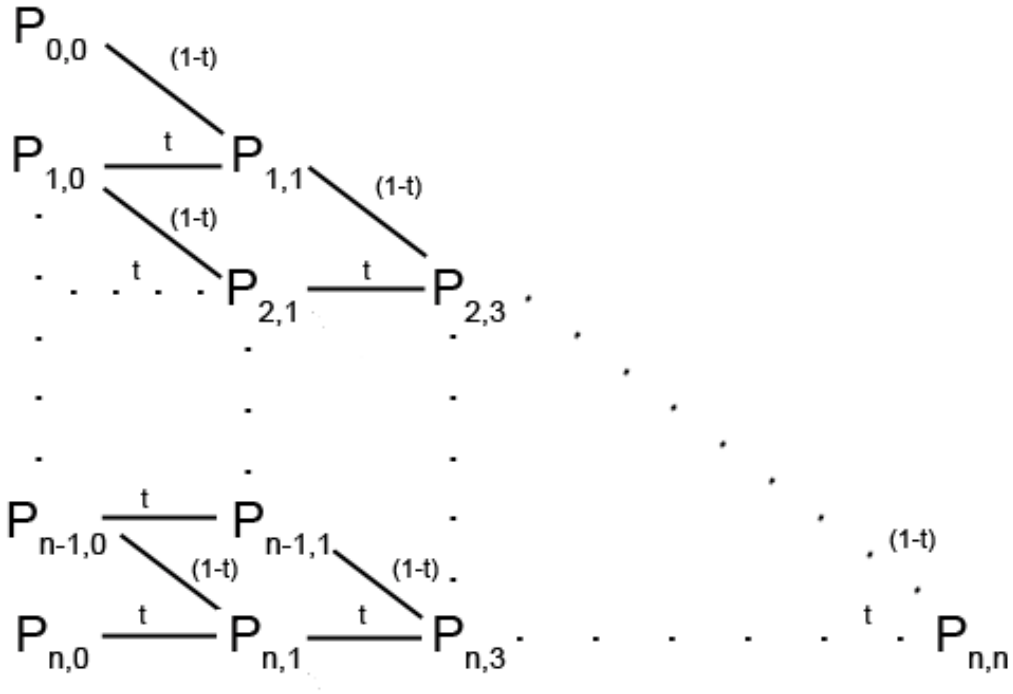
```
Procedure DeCasteljau(RidiciPolygon:Polygon; Presnost)

Begin

        IF ( SplnujePresnost(RidiciPolygon,Presnost) )

        Begin

                Nakresli( RidiciPolygon );

                Konec;

        End;

        Rozdel( RidiciPolygon,Levy,Pravy );

        DeCasteljau( Levy,Presnost );

        DeCasteljau( Pravy,Presnost );
```

```
End;
```

Procedura „DeCasteljau“ má jako vstupní parametr řídící polygon bézierovy křivky a požadovanou přesnost pro její vykreslení. Nejprve kontrolujeme zda polygon, který procedura dostala jako vstup, splňuje požadovanou přesnost. Pokud ji splňuje, můžeme polygon vykreslit a proceduru ukončit. Pokud požadovanou přesnost nesplňuje, zavoláme proceduru „Rozdel“, která rozdělí vstupní polygon na levý a pravý. V dalším kroku zavoláme proceduru „DeCasteljau“, které jako parametr postupně předáme nové polygony a vše se opakuje.

Takové řešení je však použitelné jen pro Bézierovy křivky 4. stupně. Pokud bychom chtěli algoritmus využívající dělení řídícího polygonu a jeho přibližování k výsledné křivce použít pro obecné křivky stupně n, je třeba ho modifikovat.

Do rekurentního vztahu, který je definován takto : $P_{j,i}(t) = (1-t)\cdot P_{j-1,i-1} + t\cdot P_{j,i-1}; i = 1,2,...,n;\ j = i, i+1,...n$ budeme postupně dosazovat parametr $t \in \langle 0,1 \rangle$. Pomocí rekurentního vztahu postupně vypočítáváme body $P_{j,i}(t)$, jak je naznačeno na obrázku (Obrázek 6.3).



Obrázek 6.3 – rekurentní vztah - postup výpočtu

Poslední bod - bod $P_{n,n}(t)$ odpovídá hodnotě křivky pro parametr t - $P_{n,n}(t)=v(t)$. Algoritmus spočívá v postupném zvyšování parametru t o konstantní krok a jeho dosazování do rekurentního vztahu, který vypočítá bod křivky. Na rozdíl od přímého algoritmu, kde jsme také zvyšovali parametr o konstantní krok, je tento algoritmus méně výpočetně náročný. Není třeba počítat kombinační číslo ani mocniny.

Příklad algoritmu využívající rekurentní vztah v pseudokódu:

```
Function Rekurent(P:Polygon, _n:integer,t:real):Polygon

Begin

        IF (_n=0)

        Begin

                Casteljau:=P;

        End

        Else Begin

                i:=n-_n;

                For i to i<n-1 do

                Begin

                        Result[i+1].X:=(1-t)*P[i].X+t*P[i+1].X;

                        Resul[i+1].Y:=(1-t)*P[i].Y+t*P[i+1].Y;

                End;

                Rekurent(Result,_n-1,t);

        End;

End;
```

Toto je pomocná funkce, v které pomocí rekurentního vztahu postupně vypočítáváme body křivky. Jako vstupní parametr slouží řídící polygon, parametr „t" a pomocná celočíselná proměnná. Tato funkce je rekurzivní, volá ve svém těle sama sebe – pomocná proměnná „_n", kterou předáváme jako parametr nám slouží k zastavení rekurze. Při volání této funkce z hlavního programu bude tato proměnná odpovídat stupni křivky. Na začátku zjišťujeme, zda je vstupní parametr „n" roven nule. Pokud ano, tak je výsledkem funkce vstupní polygon a funkce je ukončena. Pokud ne, inicializujeme si pomocnou proměnou „_i". V cyklu „for" pak postupně vypočítáváme pomocí bodů ze vstupního polygonu, parametru t a rekurentního vztahu body nového polygonu a ukládáme si je do pole nazvaného „Result". Po skončení cyklu zavoláme funkci znovu, jako parametr ji předáme pole s novými body a pomocnou proměnou „_n" sníženou o jedna.

Nyní si v pseudokódu ukažme proceduru, kterou budeme volat v hlavním těle programu. Tato procedura bude používat funkci, kterou jsme si popsali v předchozím odstavci:

```
Procedure Casteljau(RidiciPolygon:Polygon,n:integer)

Begin

        t:=0; i:=0; krok:=0.001;

        while t<=1 do

        Begin

                PomocnyPol:=RidiciPolygon;

                PomocnyPol:=Rekurent(PomocnyPol,n,t);

                Krivka[i]:=PomocnyPol[n];

                t:=t+krok;

                i:=i+1;
```

```
        End;

        NakresliKrivku(Krivka);

End;
```

Proceduře budeme předávat jako parametry řídící polygon křivky a stupeň křivky. Nejprve si inicializujeme proměnné a nastavíme velikost kroku, o který budeme zvyšovat parametr t. Následuje cyklus, jehož vstupní podmínkou je, že hodnota parametru t je menší, nebo rovna jedné. V tomto cyklu si nejprve do pomocného pole uložíme řídící polygon. Zavoláme funkci „Rekurent“, které jako parametr předáme pomocný polygon (v tuto chvíli je v něm uložen řídící polygon křivky), stupeň křivky a parametr t. Výsledek funkce si uložíme do pomocného pole. V něm se na poslední pozici (n-té pozici) nachází výsledná hodnota křivky pro odpovídající parametr. Nakonec zvýšíme parametr t o krok a pomocnou proměnou sloužící pro pohyb v poli výsledku zvýšíme o jedna. Po ukončení cyklu máme v poli „Krivka“ uloženy body výsledné křivky a můžeme pole vykreslit.

7 Implementace do Matlabu

Matlab sám o sobě neobsahuje žádné funkce umožňující výpočet bézierovy křivky. Musel jsem si tedy funkce pro oba popsané typy algoritmu napsat sám. V této kapitole popíši, jak jednotlivé funkce fungují.

7.1 Přímý algoritmus v Matlabu

Přímý algoritmus byl na interpretaci zřejmě nejjednodušší. Jen bylo třeba dopsat funkci pro výpočet Bernsteinových polynomů, protože Matlab takovou funkci nenabízí. Kód pomocné funkce pro výpočet Bernsteinova polynomu, kterou jsem nazval „bernstein_poly“ a uložil jako soubor „bernstein_poly.m“ vypadá tedy takto:

```
function [b] = bernstein_poly(i,n,t)

% bernstein_poly

%Funkce pro výpočet Bernsteinova polynomu pro Bézierovy křivky
```

```
%   VSTUPY: i – pořadí bodu v kontrolním polygonu
%           n - stupeň bézierovy křivky
%           t - parametr t
%   VÝSTUP: b - hodnota bernsteinova polynomu
% Jan Novotný 2009
b= nchoosek(n,i) *power(t,i)*power((1-t),(n-i));
end
```

Funkce z předaných parametrů vypočítá hodnotu bernsteinova polynomu. Pro výpočet kombinačního čísla jsem použil funkci „nchoosek“, které jako parametr předávám proměnné „n“ a „i“. Do výstupní proměnné vypočítám hodnotu Bernsteinova polynomu jako součin kombinačního čísla, a příslušných mocnin. Pro výpočet mocnin používám funkci „Power“, která má jako první parametr mocněnec a jako druhý parametr exponent.

S využitím této funkce už není problém napsat skript, který vypočítá body Bézierovy křivky. Kód funkce, kterou jsem nazval „primy_alg“ a uložil jako soubor „primy_alg.m“, vypadá takto:

```
function [ krivka ] = primy_alg( polygon,t )
%primy_alg
%   Funkce pro výpočet bodů Beziérovy křivky pomocí přímého algoritmu
%   VSTUPY: polygon - dvourozměrné pole s kontrolním polygonem
%           t - jednorozměrné pole s parametrem t
%   VÝSTUP: krivka - dvourozměrné pole s body bézierovy křivky
%  Jan Novotný 2009
```

```
n= length (polygon);

m=length(t);

krivka=zeros(m,2);



for j = 1:m

   for i = 1:n

      krivka(j,1)=krivka(j,1)+bernstein_poly( i-1 , n-1 , t(j) ) * polygon(i,1);

      krivka(j,2)=krivka(j,2)+bernstein_poly( i-1 , n-1 , t(j) ) * polygon(i,2);

   end

end
```

Funkci předáváme jako parametr řídící polygon bézierovy křivky a vektor s hodnotami parametru „t“, ten nabývá hodnot od nuly do jedničky. Na začátku si zinicializujeme proměnné. Do proměnné „n“ si uložíme pomocí funkce „length“ velikost vstupního polygonu – hodnota bude odpovídat stupni křivky zvýšenému o jedničku (to kvůli tomu, že Matlab indexuje pole od čísla jedna). Do proměnné „m“ si uložíme, opět voláním funkce length, délku vektoru s parametrem t – tím zjistíme, kolik bodů je třeba vypočítat. Nakonec si pomocí funkce zeros alokujeme dvourozměrné pole, do kterého budeme ukládat vypočtené body. První rozměr je určen hodnotou proměnné „m“ (počet bodů výsledné křivky) a druhý rozměr je dva (pohybujeme se v rovině – tedy x,y).

Následují dva vnořené cykly typu for. První, pro proměnou „j“, se bude provádět od čísla 1 až do hodnoty proměnné „m“, v níž je počet bodů, kterými bude tvořena křivka. Druhý for-cyklus pro proměnou „i“ se bude provádět od čísla 1 až do hodnoty proměnné „n“ – proběhne tedy postupně pro všechny body kontrolního polygonu. Uvnitř tohoto cyklu také probíhá vlastní výpočet bodů křivky. Z rovnice křivky víme, že funkční hodnota křivky

odpovídá sumě součinů bodů kontrolního polygonu a hodnoty Bernsteinova polynomu pro konkrétní bod. Ve for-cyklu postupně určujeme voláním funkce „bernstein_poly“ hodnotu Bernsteinova polynomu, kterou vynásobíme s konkrétním bodem kontrolního polygonu. Takto vypočtenou hodnotu přičteme k hodnotě v poli „krivka“ (to je inicializováno nulami). Po skončení tohoto vnitřního cyklu se ve vnějším cyklu posuneme k dalšímu bodu křivky a výpočet se opakuje.

7.2 Algoritmus de Casteljau v Matlabu

Díky tomu, jak Matlab umožňuje pracovat s polem, je implementace tohoto algoritmu oproti příkladu v pseudokódu, který jsem uváděl jednodušší. Vše spočívá v jediné funkci, kterou jsem nazval „casteljau“, a uložil jako soubor „casteljau.m“. Její kód vypadá takto:

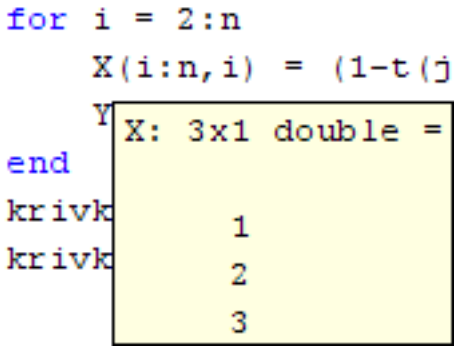
```
function [krivka] = casteljau(polygon,t)

% casteljau

% Funkce pro výpočet bodů Bézierovy křivky pomocí algoritmu „de
Casteljau“

% VSTUPY: polygon - řídící polygon křivky - dvourozměrné pole

%         t - jednorozměrné pole s parametrem t

% VYSTUP: krivka - body výsledné křivky - dvourozměrné pole

% Jan Novotný 2009



n = length(polygon);

m = length(t);

krivka = zeros(m,2);

X(:,1) = polygon(:,1);

Y(:,1) = polygon(:,2);
```

```
for j = 1:m
   for i = 2:n
      X(i:n,i) = (1-t(j))*X(i-1:n-1,i-1) + t(j)*X(i:n,i-1);
      Y(i:n,i) = (1-t(j))*Y(i-1:n-1,i-1) + t(j)*Y(i:n,i-1);
   end
   krivka(j,1) = X(n,n);
   krivka(j,2) = Y(n,n);
end
```

Funkci předáváme řídící polygon křivky a vektor obsahující hodnoty parametru „t“. Nejprve si inicializujeme proměnné. Podobně, jako u přímého algoritmu, si do proměnných nazvaných „n“ a „m“ uložíme délky vstupních polí. Poté si alokujeme dvourozměrné pole nazvané „krivka“, do kterého budeme ukládat vypočtené body. Do pomocného pole „X“ si uložíme x-ové složky bodů řídícího polygonu. Stejnou operaci provedeme s Y-ovými souřadnicemi, které uložíme do pomocného pole „Y“.

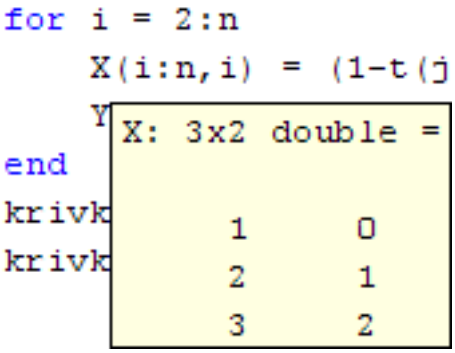
Následují dva for-cykly. Vnější cyklus pro proměnou „m“ slouží pro pohyb v poli výsledku – „krivka“ a pro pohyb v poli s hodnotami parametru „t“. Ve vnitřním cyklu se provádí samotný výpočet, ke kterému využíváme rekurentní vztah. Cyklus začíná od čísla dvě, důvod je zřejmý z obrázku 8, na kterém je postup výpočtu pomocí rekurentního vztahu. Zápisem X(i:n,i) zajistíme to, že se výpočet provede pro prvky s indexem odpovídajícím hodnotě „i“, až po prvky s indexem, který odpovídá hodnotě „n“. Díky tomu nemusíme použít další for-cyklus. Pro lepší ilustraci toho jak probíhá, nám poslouží následující příklad. Mějme křivku 2. stupně s kontrolním polygonem, jehož body mají souřadnice P_0=[1,1], P_1=[2,2], P_2=[3,3]. Hledáme funkční hodnotu křivky pro velikost parametru t=0,5. Cyklus se bude provádět od čísla 2 až do hodnoty proměnné „n“, která je v tomto případě 3. Cyklus

se tedy provede 2-krát (poprvé pro i=2, podruhé pro i=3). Při vstupu do cyklu máme v proměnné „X“ uloženy x-ové složky bodů, tvořících kontrolní polygon – rozměr tohoto pole je 3 x 1 (Obrázek 7.1).



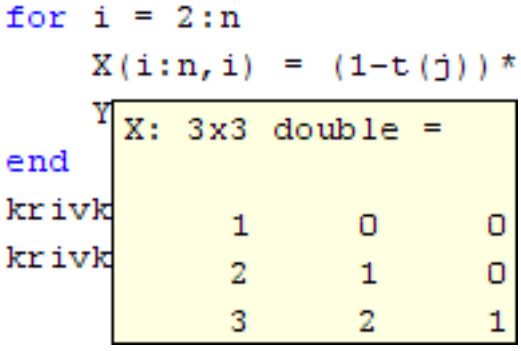
Obrázek 7.1 – rekurentní výpočet – 1. krok

Po prvním průchodu cyklem a provedením výpočtu se rozměr pole rozšíří na 3 x 2. V druhém sloupci jsou uloženy nově vypočtené hodnoty (Obrázek 7.2).



Obrázek 7.2 – rekurentní výpočet – 2. krok

Po posledním průchodu cyklem má pole rozměr 3 x 3 a v jeho třetím sloupci, na třetím řádku je uložena hledaná hodnota (Obrázek 7.3). Stejně probíhá výpočet i pro proměnou „Y“.



Obrázek 7.3 – rekurentní výpočet – 3. krok

Po skončení vnitřního cyklu si do proměnné „krivka“ ukládáme hodnotu, která se nachází na posledním sloupci a posledním řádku v poly „X“ a „Y“.

7.3 Demonstrace algoritmů

Pro demonstraci funkčnosti algoritmů jsem napsal další dvě funkce, jednu pro přímý algoritmus a druhou pro algoritmus de Casteljau. Tyto dvě funkce se od sebe liší jen v řádku, kde se volá funkce pro výpočet bézierovy křivky. Jednou je to funkce „primy_alg“ (pro přímý algoritmus), podruhé je to funkce „casteljau“ (pro algoritmus de casteljau).

Funkci pro demonstraci přímého algoritmu jsem nazval „Bezier_primy_alg“ a je uložena v souboru „Bezier_primy_alg.m“. Funkce pro demonstraci algoritmu de Casteljau se jmenuje „Bezier_casteljau“ a uložena je v souboru „Bezier_casteljau.m“.

Kód funkce „Bezier_casteljau“ vypadá takto:

```
function Bezier_casteljau(n)



%Bezier_casteljau

%   Funkce pro demonstraci bézierovy křivky pomocí algoritmu de Casteljau

%   VSTUP: n - stupeň bézierovy křivky

%   Jan Novotný 2009



close all;

presnost = 100;

cas=0;

figure(1);

hold on;
```

```
set(gca,'Fontsize',12);

title(['Bézierova křivka stupně ',num2str(n)]);

t=linspace(0,1,presnost);

axis([0 10 0 10]);



for i = 1:n + 1

    polygon(i,:) = ginput(1);

    plot(polygon(:,1),polygon(:,2),'k-','LineWidth',2);

    plot(polygon(:,1),polygon(:,2),'ro','MarkerSize',6,'MarkerFaceColor','r');

end



tic;

krivka = casteljau(polygon,t);

cas=toc;

plot(krivka(:,1),krivka(:,2),'b-','LineWidth',3);



cas=cas*1000;

title(['Bézierova křivka stupně ',num2str(n),', doba výpočtu: ',num2str(cas),'
ms']);
```

Funkci předáváme jako parametr stupeň křivky. Nejprve provedeme inicializaci. Příkazem „close all“ uzavřeme všechny figure okna, která jsou otevřená. Do proměnné „presnost“ nastavíme počet bodů, kterými bude výsledná křivka tvořena. Dále je vynulována proměnná „cas“, do které

se bude ukládat informace o tom, jak dlouho trval výpočet bodů křivky. Příkazem „figure(1)“ vytvoříme grafické okno figure. Příkaz „hold on“ zajistí to, že do jednoho obrázku budeme moci nakreslit více grafů (kontrolní polygon, křivka). V dalším kroku si nastavíme velikost písma, která bude v obrázku použita. V tomto případě je velikost nastavena na 12 bodů. Pomocí příkazu „title“ si nastavíme nadpis grafu. Funkce „linspace“ vrátí do proměnné „t“ vektor s počtem prvků, který odpovídá proměnné „presnost“ a má hodnoty od nuly do jedné. Poté nastavíme osy grafu. Na ose x tak budeme mít hodnoty od nuly do deseti, stejně tak na ose y.

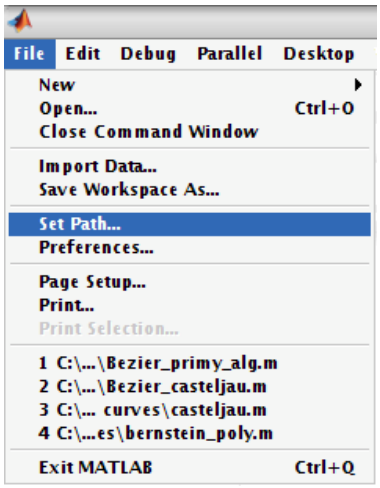
Následuje for-cyklus, ve kterém se z grafického vstupu načítají body kontrolního polygonu. Cyklus probíhá od jedné do hodnoty n+1, protože křivka n-tého stupně je tvořena n+1 body. Funkcí „ginput(1)“ načteme z grafického vstupu jeden bod a uložíme ho na i-tou pozici v kontrolním polygonu. Následující dvě funkce „plot“ vykreslí kontrolní polygon a do souřadnic bodu nakresli červený puntík.

Když už máme body řídícího polygonu, je třeba vypočítat body křivky. Protože nás zajímá, jak dlouho bude výpočet trvat, tak před voláním funkce pro výpočet bodů křivky použijeme funkci „tic“. Nyní zavoláme funkci „casteljau“, která má jako parametry řídící polygon a vektor s parametrem „t“. Pro funkci, která demonstruje přímý algoritmus voláme v tomto místě funkci „primy_alg“. Výsledek funkce pro výpočet bodů Bézierovy křivky uložíme do proměnné „krivka“. Potom zavoláme funkci „toc“ a dobu trvání výpočtu křivky uložíme do proměnné „cas“.

Nyní, když máme body křivky, stačí ji vykreslit. K tomu využijeme funkci „plot“, která pomocí předaných parametrů vykreslí křivku modrou barvou. Nakonec už jen vynásobíme hodnotu proměnné „cas“ tisícem, čímž dostaneme čas v milisekundách. Poté, už jen upravíme voláním funkce „title“ nadpis grafu, do něhož přidáme informaci o trvání výpočtu.

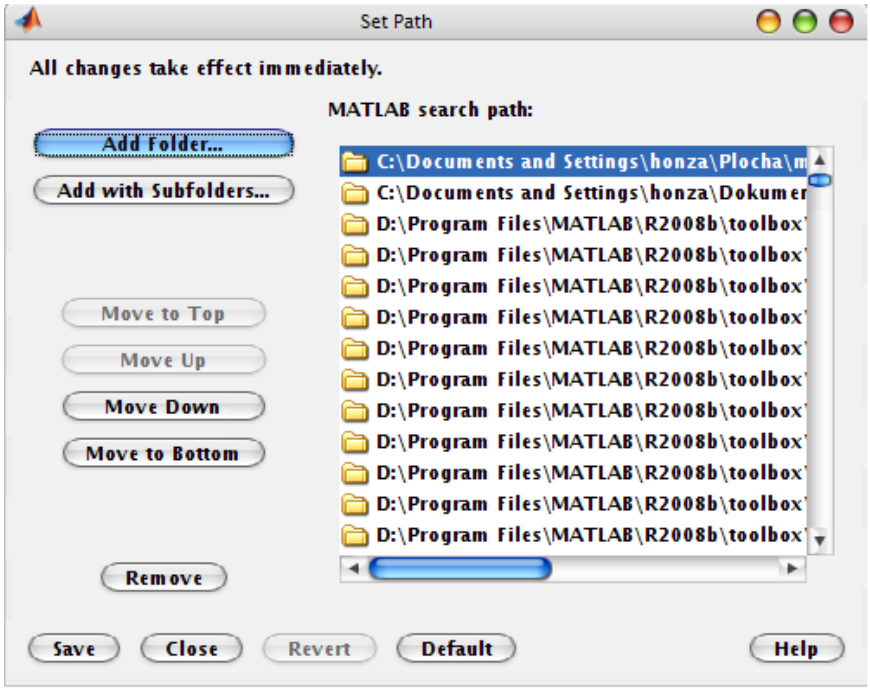
7.4 Použití v Matlabu

Pro to, aby bylo možné použít tyto funkce v Matlabu, je třeba přidat adresář, ve kterém je máme uloženy do cesty. To uděláme v záložce „File“ (obrázek 7.4), kde je položka „Set path“.



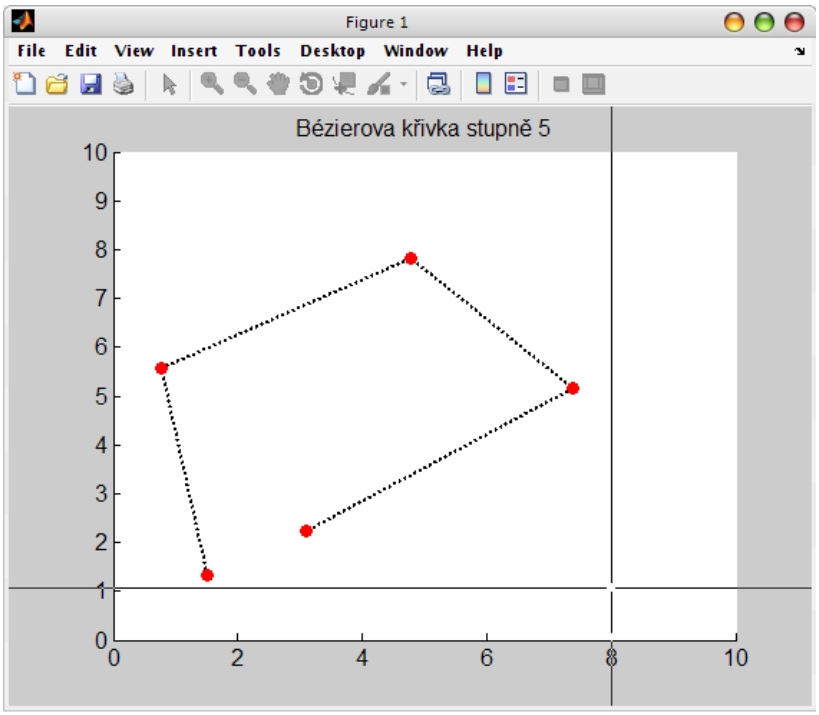
Obrázek 7.4 – nastavení cesty 1

V dialogovém okně „Set Path“ (obrázek 7.5) stiskneme tlačítko „Add folder…“ a zvolíme složku, v které máme uloženy soubory s funkcemi.



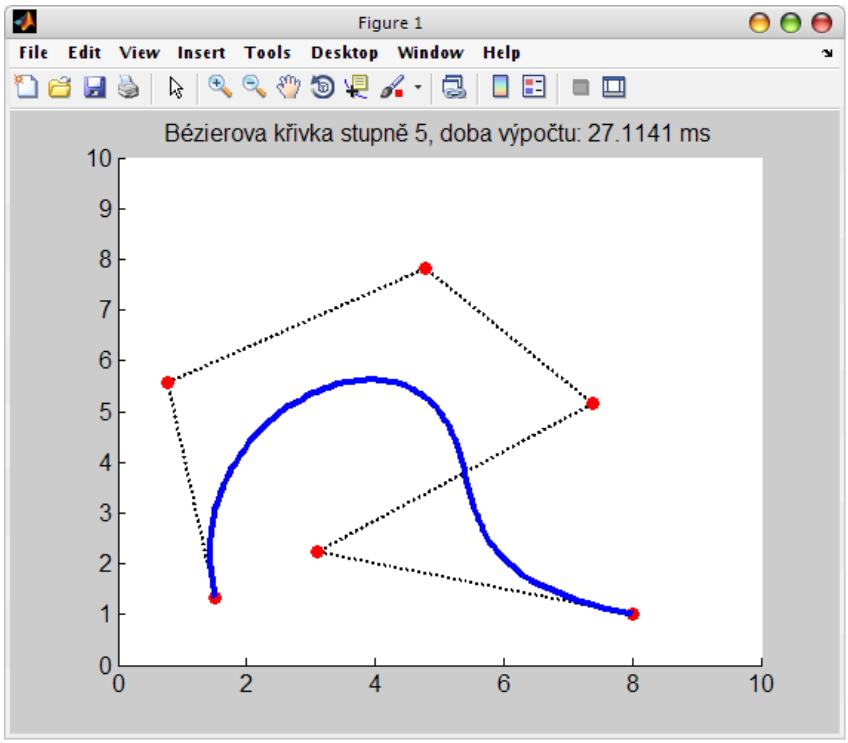
Obrázek 7.5 – nastavení cesty 2

Teď, když máme nastavenou cestu, můžeme v příkazovém okně zavolat funkci „Bezier_primy“ (případně „Bezier_casteljau“), které jako parametr předáme stupeň křivky. Po tom, co příkaz potvrdíme stisknutím klávesy „enter“, se nám otevře „figure“ okno (obrázek 7.6). V tomto okně postupně zadáváme body kontrolního polygonu.



Obrázek 7.6 – Figure okno

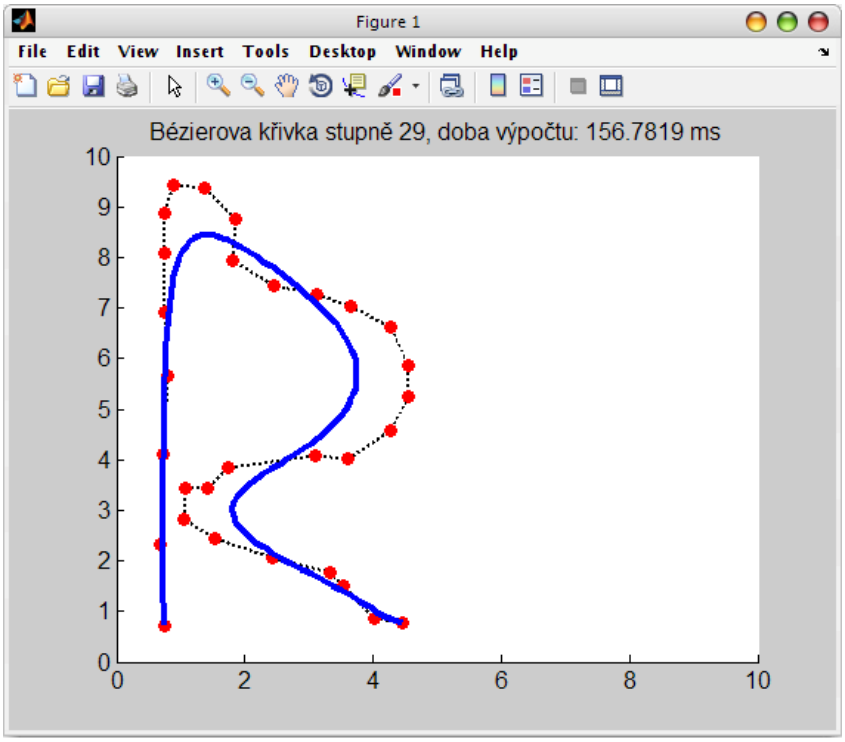
Po zadání odpovídajícího počtu bodů, je vykreslena křivka a v titulu grafu se objeví informace o tom, jak dlouho trval výpočet bodů křivky (obrázek 6.7).



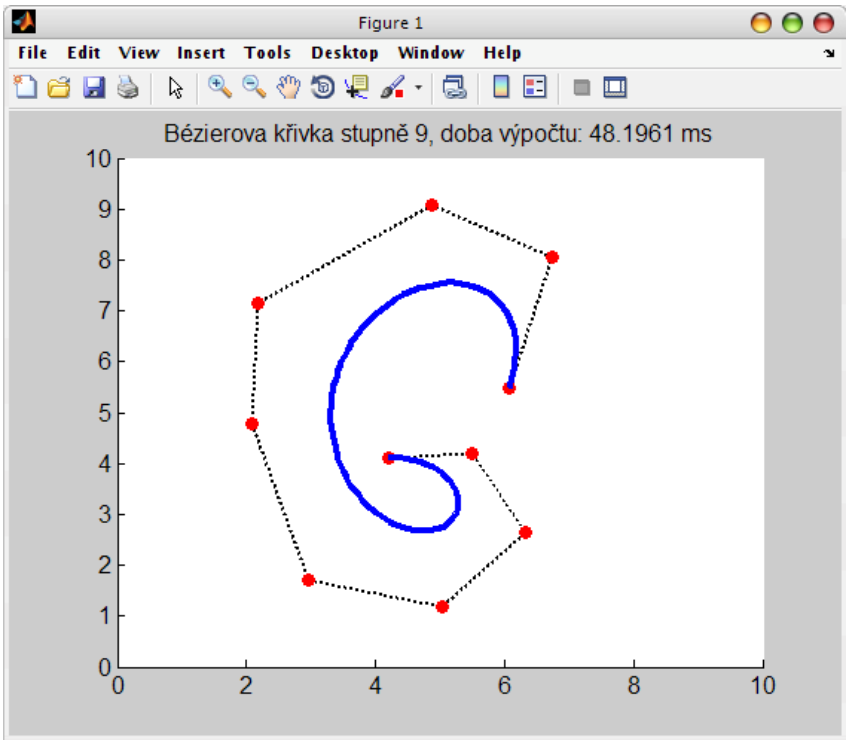
Obrázek 7.7 – Figure okno – křivka 5. stupně

8 Závěr

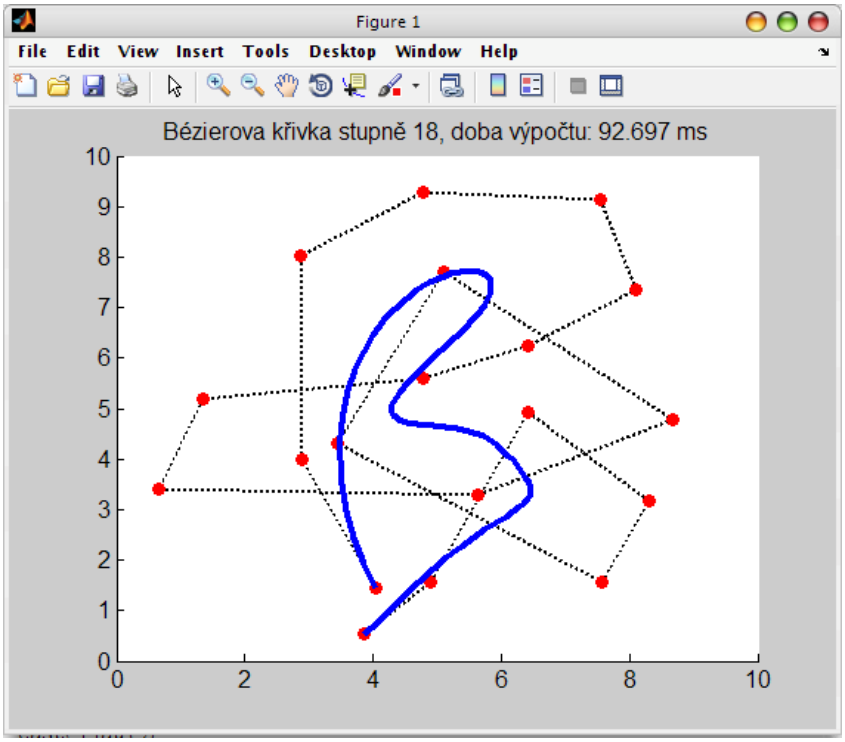
S Matlabem jsem se poprvé setkal až při zpracování mé bakalářské práce. Musím říci, že práce s ním je velmi intuitivní a není vůbec složitá. Velmi jsem ocenil jeho debugger, kde při krokování funkce stačí pouze myší najet nad název proměnné, která nás zajímá, a hned můžeme vidět její hodnoty. Výsledkem mé práce jsou funkce, které mohou být použity pro výpočet a prezentaci Beziérových křivek v prostředí Matlabu. Pomocí funkcí, které jsem vytvořil, je možno snadno zadat a zobrazit křivky libovolného stupně spolu s jejich kontrolním polygonem a informací o trvání výpočtu.



Obrázek 8.1 - písmeno "R" křivkou 29. stupně



Obrázek 8.2 - písmeno "G" křivkou 9. stupně



Obrázek 8.3 - písmeno "B" křivkou 18. stupně

Podle očekávání dopadl v porovnání doby potřebné k výpočtu lépe algoritmus de Casteljau, a to jak v Delphi, tak i v Matlabu. Zatímco Bézierova kubika pomocí algoritmu de Casteljau trvala Matlabu přibližně 16ms, stejná křivka pomocí přímého algoritmu mu zabrala přibližně 90ms. Podobně na tom je i program v Delphi. Očekával jsem, že výpočty budou obecně

rychlejší v Matlabu, než v Delphi. Nejspíš hlavně proto, že implementace algoritmů v Matlabu mi přišla mnohem jednodušší. Čekal jsem, že Matlab bude výkonnější při práci s poli. To proto, že v moji implementaci v Delphi provádím během výpočtu alokace polí apod. Ale výpočty Delphi byly v porovnání nakonec mnohem rychlejší, než v Matlabu. Výpočet Bézierovy kubiky tvořené tisícem bodů pomocí algoritmu de Casteljau zabere Matlabu běžně něco okolo 150ms. Zatímco Delphi vypočítá stejnou křivku za necelé 4ms.

Při měření času potřebného pro výpočet bodů křivky jsem v Delphi nejprve naivně použil funkci „GetTime“, která vrací aktuální systémový čas. Jednoduše jsem si uložil čas před spuštěním algoritmu a po jeho ukončení. Rozdíl těchto časů byl k mému překvapení roven nule. Hledal jsem funkci pro přesnější měření a nalezl jsem funkci Windows API „GetTickCount“. Tato funkce vrací počet milisekund, které uběhly od spuštění systému. Výsledek byl, ale stále nula. Zjistil jsem, že hodnota funkce „GetTickCount“ je zvyšována po 16ms a to pro změření nestačí. Našel jsem tedy další funkci Windows API „QueryPerformanceCounter“, která vrací aktuální hodnotu časového registru. Tato hodnota je však závislá na frekvenci a je tedy nutné ji vždy dělit hodnotou, kterou vrací funkce „QueryPerformanceFrequency“. Řešení s využitím těchto funkcí již bylo dostatečně přesné.

V Matlabu bylo měření času o dost jednodušší v porovnání s Delphi. Matlab pro účely měření časové náročnosti výpočtů poskytuje funkce „tic“ a „toc“. První z nich stačí dát na začátek části kódu, kterou chceme měřit. Druhou pak dáme na konec a její hodnotu si uložíme do nějaké proměnné. V proměnné pak máme uložen potřebný čas k výpočtu v sekundách.

9 Zdroje

1. ŽÁRA, Jiří, et al. *Moderní počítačová grafika*. Brno: Computer Press, 2004. 609 s. ISBN 80-251-0454-0.
2. NAJZAR, Karel. *Základy teorie splinů*. Praha: Karolinum, 2006. 203 s. ISBN 80-246-1287-9.
3. KARBAN, Pavel. *Výpočty a simulace v programech Matlab a Simulink*. Brno : Computer Press, 2006. 220 s. ISBN 80-251-1301-9.

4. SVOBODA, Luďek, et al. *1001 tipů a triků pro Delphi*. Brno: Computer Press, 2003. 546 s. ISBN 80-7226-488-5.

5. CASSELMAN, Bill. From Bézier to Bernstein. *American Mathematical Society* [online]. 2008 [cit. 2009-04-03]. Dostupný z WWW: <http://www.ams.org/featurecolumn/archive/bezier.html>.

6. GetTickCount. *Wikipedia* [online]. 2007 [cit. 2009-04-07]. Dostupný z WWW: <http://en.wikipedia.org/wiki/GetTickCount>.

7. ALEXANDR, Lubomír. Diplomová práce: Výuka počítačové grafiky cestou WWW. *VUT Brno* [online]. 1999 [cit. 2009-04-07]. Dostupný z WWW: <http://lubovo.misto.cz/_MAIL_/curves/obsah.html>.

8. KOBZA, Jiří. *Počítačová geometrie*. Olomouc: Univerzita Palackého, 2008. 61 s. Dostupný z WWW: <http://phoenix.inf.upol.cz/esf/ucebni/pgm1.pdf>.

9. SHIKIN, E. V., PLIS, Alexander I. *Handbook on splines for the user*. [s.l.] : CRC Press, 1995. 240 s. ISBN 978-0849394041.

10. BREINER, Moshe, BIRAN, Adrian. *MATLAB 6 for engineers*. [s.l.] : Prentice Hall, 2002. 800 s. ISBN 978-0130336316.

11. About Timers . *Microsoft Developer Network* [online]. 2009 [cit. 2009-04-07]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms644900(VS.85).aspx#high_resolution>.

12. DUŠEK,F. *MATLAB a SIMULINK - úvod do používání.* (druhé rozšířené vydání) [Skriptum], Univerzita Pardubice 2002, 158 s., ISBN 80-7194-475-0