Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Práce s vektorovou grafikou v .NET

Filip Růžička

© 2016 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Filip Růžička

Informatika

Název práce

Práce s vektorovou grafikou v .NET

Název anglicky

Working with vector graphics in .NET

Cíle práce

Bakalářská práce je zaměřena na problematiku práce s vektorovou grafikou v prostředí .NET. Hlavním cílem je představit prostředky, které .NET nabízí pro práci s vektorovou grafikou a na praktickém příkladu demonstrovat přínosy a případné nedostatky těchto prostředků.

Metodika

Metodika řešení bakalářské práce je založena na analyticko-syntetickém přístupu. Bude provedena analýza odborných a technických informačních zdrojů, dále bude provedena syntéza zjištěných poznatků a na jejím základě budou popsány specifické prostředky, které poskytuje .NET pro práci s vektorovou grafikou. Dále budou navrženy a implementována ukázková aplikace řešící stejný problém s využitím a bez využití specializovaných prostředků .NET pro práci s vektorovou grafikou. Následně budou obě metody porovnány a zhodnoceny.

Doporučený rozsah práce

35-40 stran

Klíčová slova

.NET Framework, C#, SVG, XML, vektorová grafika, křivka

Doporučené zdroje informací

Eisenberg, David J. Feb. 2002. SVG Essentials. 1st. Sebastopol : O'Reilly, Feb. 2002. p. 364. ISBN: 0-596-00223-8.

Farin, Gerald E., Hoschek, Josef and Kim, Myung Soo. 2002. Handbook of Computer Aided Geometric Design. Amsterdam : Elsevier B.V., 2002. ISBN: 978-0-494-51104-1.

Martišek, Dalibor. 2000. Počítačová geometrie a grafika. Brno : skripta Vysoké učení technické Brno, 2000. str. 89. ISBN: 80-214-1632-7.

POKORNÝ, J. – ČESKÁ SPOLEČNOST PRO SYSTÉMOVOU INTEGRACI, – MLÝNKOVÁ, I. *XML technologie : principy a aplikace v praxi.* Praha: Grada, 2008. ISBN 978-80-247-2725-7.

Sharp, John. 2010. Microsoft Visual C# 2010 – krok za krokem. Brno : Computer Press a.s., 2010. ISBN: 978-80-251-3147-3.

Skřenek, Martin. 2015. Jazyk C# – programování II. Praha : GOPAS, 2015.

Špička, Ivo a Frischer, Robert. 2012. Počítačová geometrie a grafika – Teoretické základy. Ostrava : Technická univerzita Ostrava – Vysoká škola báňská, 2012. str. 114. ISBN: 978-80-248-2590-8.

Thai, Thuan L. and Lam, Hoang Q. 2002. .NET Framework Essentials. 2nd. Sebastopol : O'Reilly, 2002. p. 320. ISBN: 0-596-00302-1.

Předběžný termín obhajoby

2015/16 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 28. 02. 2016

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Práce s vektorovou grafikou v .NET" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 4.3.2016 ____________________________

Poděkování

Rád bych touto cestou poděkoval vedoucímu bakalářské práce Ing. Jiřímu Brožkovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování této bakalářské práce.

Práce s vektorovou grafikou v .NET

Souhrn

Současná verze Microsoft .NET Framework (4.6) poskytuje přístup k základní funkcionalitě grafického rozhraní GDI+ (Graphical Device Interface) pomocí více než padesáti elementů kódu (tříd, rozhraní, struktur, výčtových typů a delegátů) obsažených ve jmenném prostoru *System.Drawing*. Řada pokročilých funkcí je pak nabízena v dalších třech vnořených jmenných prostorech, a to *Drawing2D*, *Imaging* a *Text*. Tato práce demonstruje na komplexním příkladu užití vybrané množiny těchto knihovních funkcí a srovnává jejich výstup, časovou a paměťovou náročnost s vlastní parametrizovatelnou implementací odpovídajících rovinných geometrických útvarů, techniky vyhlazování hran a vyplňování uzavřených oblastí. Cílem práce je vytvoření aplikace – jednoduchého prohlížeče formátu SVG (Scalable Vector Graphics) – která bude schopná v grafickém uživatelském prostředí interpretovat příkazy SVG souboru oběma výše uvedenými způsoby a pro každý z nich poskytovat informace o výpočetní složitosti, na jejichž základě lze usuzovat na vhodnost použití jednoho či druhého přístupu, tedy knihovních funkcí či funkcí uživatelsky definovaných s možností jejich další optimalizace.

Klíčová slova: .NET Framework, C#, SVG, XML, vektorová grafika, křivka

Working with vector graphics in .NET Framework

Summary

Current version of the Microsoft .NET Framework (4.6) provides access to the basic functionality of GDI+ (Graphical Device Interface) with more than fifty of code elements (classes, interfaces, structures, enumerated types, and delegates) contained in the *System.Drawing* namespace. Many advanced functions are then offered in other three nested namespaces, namely Drawing2D, Imaging and Text. This work demonstrates on a complex example using of a selected set of these library functions and compares their output, time and memory requirements with its own parameterizable implementation of the corresponding planar geometric shapes, antialiasing techniques and filling closed areas. The goal of this work is to create an application - a simple SVG format (Scalable Vector Graphics) viewer - which will be able to interpret SVG file commands in a graphical user interface by both of the above methods and for each of them to provide information about the computational complexity, on the basis of which we can show the appropriateness of using one or the other approach, library functions or user-defined functions with the possibility of further optimization.

Keywords: .NET Framework, C#, SVG, XML, vector graphics, curve

Obsah

1 Úvod **10**

2 Cíl práce a metodika **13**
2.1 Cíl práce 13
2.2 Metodika 13

3 Teoretická východiska **14**
3.1 Stručný úvod do .NET 14
3.1.1 Historie 14
3.1.2 Architektura .NET Framework 14
3.1.3 Nástroje vektorové grafiky v .NET 15
3.1.4 Jmenné prostory pro 2D grafiku 17
3.1.5 Jmenný prostor System.Drawing 17
3.1.5.1 Kreslení úsečky 18
3.1.5.2 Kreslení Bézierovy křivky 24
3.2 Stručný úvod do SVG 33
3.2.1 Historie 33
3.2.2 Popis SVG dokumentu 34
3.2.2.1 Viewport 34
3.2.2.2 Souřadnicový systém 34
3.2.2.3 Základní rovinné útvary 35
3.2.2.4 SVG Line - <line> 35
3.2.2.5 SVG Polyline - <polyline> 36
3.2.2.6 SVG Rectangle - <rect> 36
3.2.2.7 SVG Circle - <circle> 37
3.2.2.8 SVG Ellipse - <ellipse> 37
3.2.2.9 SVG Polygon - <polygon> 37
3.2.2.10 SVG Path - <path> 38
3.3 Vlastní práce 40
3.3.1 Jednoduchý prohlížeč SVG souborů - SvgViewer 40
3.3.1.1 Jádro aplikace – třída *SvgViewer* 40
3.3.1.2 Třída Curves 50
3.3.1.3 Grafické uživatelské rozhraní 57
3.3.1.4 Nástroje pro ladění a měření výkonu 63

4 Výsledky a diskuse ... 65
4.1 Rasterizace úsečky a Bézierovy křivky ... 65
4.2 Aplikace SvgViewer ... 66
4.2.1 Individuální vykreslování cest ... 66
4.2.2 Testování ... 67
5 Závěr ... 68
6 Seznam použitých zdrojů ... 70
7 Přílohy ... 71

Seznam obrázků

Obrázek 1 - Komponenty .NET framework ... 14
Obrázek 2 - Graphical Device Interface ... 16
Obrázek 3 - Hodnoty směrnice k v oktantech souřadnicové soustavy ... 19
Obrázek 4 - spojité a diskrétní vykreslení úsečky ... 20
Obrázek 5 - Varianty souřadnic pro nový bod úsečky ... 22
Obrázek 6 - Kvadratická a kubická Bézierova křivka ... 26
Obrázek 7 - Bernsteinovy polynomy 3.stupně ... 28
Obrázek 8 - Vrcholy řídícího polygonu ... 29
Obrázek 9 - Nové body po 1.interpolaci ... 29
Obrázek 10 - Interpolace ve 2.kroku ... 30
Obrázek 11 - Poslední krok - bod na křivce ... 30
Obrázek 12 - SvgPathCommand ... 41
Obrázek 13 - Nastavení parametrů vykreslování ... 43
Obrázek 14 - Využití metody GetTranslatedCommad v GUI ... 44
Obrázek 15 - Třída SvgPathElement ... 45
Obrázek 16 - Umístění komponenty SvgViewer v Toolboxu ... 46
Obrázek 17 - Individuální vykreslování cest ... 61
Obrázek 18 - Stavové informace z komponenty SvgViewer ... 63
Obrázek 19 - Srovnání vlastních a vestavěných metod ... 65

Seznam tabulek

Tabulka 1 - Tabulka jmenných prostorů pro 2D grafiku ... 17
Tabulka 2 - Přetížení metody DrawLine ... 18
Tabulka 3 - Přetížení metody DrawBezier ... 25
Tabulka 4 - Příkazy atributu d v elementu <path> ... 39
Tabulka 5 - Vlastnosti a metody třídy SvgPath ... 42
Tabulka 6 - Metody třídy SvgPathElementsFactory ... 45
Tabulka 7 - Vlastnosti a metody třídy SvgViewer ... 49

1 Úvod

Bez ohledu na operační systém či použitou platformu je vektorová grafika jednou ze dvou základních forem prezentace obrazových dat v počítačové grafice. Vždy je tvořena sadou tzv. grafických primitiv, což jsou základní geometrické útvary, jako je bod, úsečka, kružnice, elipsa, křivka, mnohoúhelník atd. Každý tento útvar má svůj formální matematický popis vycházející z analytické geometrie a společně jsou označovány jako vektory, proto mluvíme o grafice vektorové. V této práci budeme předpokládat útvary rovinné, i když teoreticky lze uvažovat i o jejich prostorových analogiích. Z těchto elementů je pak složen celý obraz, v němž mohou být dále aplikovány metody vyplňování ploch barvou, barevným přechodem či texturou, techniky vyhlazování čar a křivek (anti-aliasing) apod. Na rozdíl od grafiky rastrové (bitmapové), ve které je grafická informace reprezentována jako množina obrazových bodů (pixelů) v ortogonální matici, jejichž základními parametry jsou souřadnice a barva, pracuje vektorová grafika s objekty daných analytických, častěji však parametrických předpisů. Z této skutečnosti vyplývají některé obecné vlastnosti vektorové grafiky. V první řadě je to poměrně nízká paměťová a časová složitost, avšak od určité míry komplexnosti vektorového obrazu je ve srovnání s bitmapovou grafikou náročnost na paměť i procesor vyšší. Kvalita vektorového obrazu je invariantní vzhledem ke změně jeho velikosti, což je podstatná vlastnost např. u aplikací umožňujících změnu měřítka kreslícího plátna (Zoom). U rastrových obrázků má tato operace za následek jejich rozostření, při vyšších hodnotách změny měřítka vede dokonce ke ztrátě původní obrazové informace. Při práci s vektorovým obrazem je možno též manipulovat s jeho jednotlivými objekty, instantně měnit jejich velikost, polohu, tvar, barvu a řadu dalších vlastností. To lze pochopitelně pouze s použitím editorů vektorové grafiky, které jsou dostupné jako proprietární software (Adobe® Illustrator, Corel Draw®, Zoner Draw) či jako produkty pro volné použití (Inkscape, OpenOffice Draw).

Využití vektorové grafiky je velmi široké a pokrývá rozsáhlou oblast systémového i aplikačního programového vybavení. Pravděpodobně prvním kontaktem uživatelů s vektorovou grafikou obvykle bývají okna grafického uživatelského rozhraní (GUI) operačního systému, ať už je jedná o MS Windows, MacOS nebo grafické nadstavby operačních systémů unixového typu, a to včetně jejich klonů pro mobilní zařízení nebo vestavěné systémy. Popisem pomocí vektorů jsou také realizovány některé typy počítačových písem – fontů, což dalo vzniknout standardu TrueType, následně pak

OpenType jako integrace osvědčených vlastností z TrueType a konkurenčního standardu PostScript. Dále jsou to již zmiňované editory vektorové grafiky, které umožňují tvorbu grafiky pro web, pre-print, tvorbu logotypů, diagramů, ilustrací atd. Významným segmentem pro uplatnění vektorové grafiky jsou aplikace typu CAD[1] pro podporu projektování ve stavebnictví, architektuře, strojírenství a elektrotechnice, CASE[2] – pro počítačovou podporu při vývoji software, GIS - geografických informačních systémů, SCADA[3] - systémů pro dispečerské řízení a sběr dat atd.

Tato práce se z širokého spektra aplikací vektorové grafiky zaměřuje na její uplatnění v prostředí Internetu, speciálně na sémantické prvky značkovacího jazyka SVG (Scalable Vector Graphics) a především na interpretaci uložené kódované grafické informace. Jazyk SVG popisuje dvoudimenzionální vektorovou grafiku pomocí XML a od svého vzniku v roce 2001, kdy byl poměrně dlouhou dobu opomíjen v prohlížečích Internet Explorer společnosti Microsoft na úkor vlastního formátu VML[4] zažívá opět renesanci, a to především díky své relativní jednoduchosti, čitelnosti, nízkým nárokům na výkon a nezávislosti na rozlišení zobrazovacích zařízení. Zvláštní pozornost bude v praktické části této práce věnována zpracování elementu `path`, zejména jeho atributu `d` (data), který popisuje cesty – uzavřené či otevřené systémy rovinných křivek. Ukázková aplikace však rozhodně neaspiruje na plnohodnotný SVG prohlížeč s dokonalým parserem a interpretem uložené grafické informace, ale pouze naznačuje směr, jakým by se mohl seriózní vývoj takového softwarového nástroje ubírat.

Vzhledem ke skutečnosti, že SVG vychází z formátu XML, lze pro jeho zpracování s výhodou využít nástroje pro syntaktickou a lexikální analýzu XML souborů. Tyto nástroje jsou často standardně zařazeny do knihoven moderních programovacích jazyků, infrastruktur a integrovaných vývojových prostředí. Platforma Microsoft .NET Framework není v tomto směru výjimkou a nabízí v pěti jmenných prostorech komplexní sadu tříd pro vývoj aplikací podporujících zpracování souborů tohoto formátu. Stejně robustně je implementována i podpora vektorové grafiky. Cílem této práce je demonstrovat na uceleném příkladu efektivitu použití knihovních grafických funkcí ve srovnání s jejich uživatelskou implementací dle známých obecných algoritmů. Důraz bude kladen nejen na komparaci

[1] Computer Aided Design
[2] Computer Aided Software Engineering
[3] Supervisory Control And Data Acquisition
[4] Vector Markup Language

aspektů časových a paměťových, ale též na vhodnost použití konkrétní metody v kontextu právě řešené problémové domény. Pro vývoj ukázkové aplikace byl z bohatého portfolia programovacích jazyků platformy .NET zvolen jazyk C#, jakožto reprezentant moderního, expresivně silného, multiparadigmatického a objektově orientovaného programovacího jazyka.

2 Cíl práce a metodika

2.1 Cíl práce

Bakalářská práce je zaměřena na problematiku práce s vektorovou grafikou v prostředí .NET. Hlavním cílem je představit prostředky, které .NET nabízí pro práci s vektorovou grafikou a na praktickém příkladu demonstrovat přínosy a případné nedostatky těchto prostředků.

2.2 Metodika

Metodika řešení bakalářské práce je založena na analyticko-syntetickém přístupu. Bude provedena analýza odborných a technických informačních zdrojů, dále bude provedena syntéza zjištěných poznatků a na jejím základě budou popsány specifické prostředky, které poskytuje .NET pro práci s vektorovou grafikou. Dále budou navrženy a implementována ukázková aplikace řešící stejný problém s využitím a bez využití specializovaných prostředků .NET pro práci s vektorovou grafikou. Následně budou obě metody porovnány a zhodnoceny.

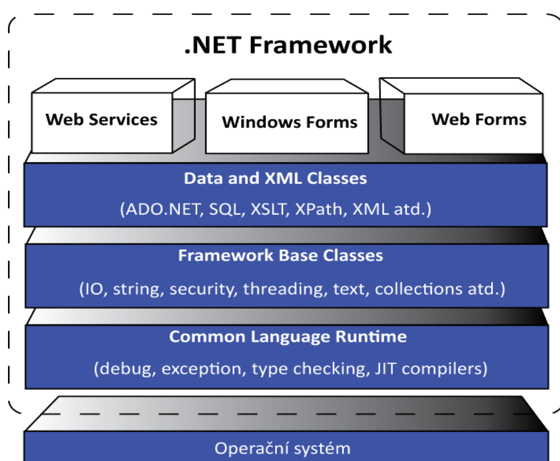
3 Teoretická východiska

3.1 Stručný úvod do .NET

3.1.1 Historie

Platforma .NET, i když na tomto místě lze ještě hovořit o vizi .NET, byla poprvé odborné veřejnosti představena v roce 2000 jako nový vývojový rámec s novým programovým rozhraním pro využívání služeb operačního systému Windows a jeho API, ve kterém je integrována řada technologií, jež vzešly ze softwarových laboratoří Microsoftu v průběhu devadesátých let minulého století. Do rámce .NET jsou začleněny technologie COM+ (Component Services), ASP[5] (framework pro vývoj webových aplikací), XML a objektově orientovaný návrh, podpora nových protokolů webových služeb jako jsou SOAP, WSDL[6] a UDDI[7] a především je zde patrný silný důraz na využití Internetu. (Thai, et al., 2002)

3.1.2 Architektura .NET Framework



Obrázek 1 - Komponenty .NET framework
Zdroj: (Thai, et al., 2002 p. 12)

První úrovní je operační systém. Nad ním je vrstva CLR *(Common Language Runtime)*, která představuje nejdůležitější komponentu celého frameworku a často bývá svým významem přirovnávána k JVM *(Java Virtual Machine)* ve světě programování v jazyce Java. CLR je srdcem architektury .NET a zajišťuje fundamentální operace

[5] Active Server Pages
[6] Web Services Description Language
[7] Universal Description, Discovery and Integration

s objekty, tj. jejich aktivaci (inicializaci), ověřování z hlediska bezpečnosti, umísťování do paměti, spouštění a uvolňování z paměti. Koncepčně jsou si CLR i JVM podobné v tom, že obě běhové infrastruktury abstrahují rozdíly v platformách (operačních systémech), nad kterými jsou provozovány, avšak JVM podporuje pouze jazyk Java. CLR podporuje všechny jazyky, které mohou být reprezentovány v tzv. CIL *(Common Intermediate Language)*. Další konceptuální rozdíl mezi těmito dvěma infrastrukturami je ten, že Java běží na řadě platforem, obecně na takových, na které je portovaný JVM. Kód .NET beží pouze na platformě Windows s CLR, i když v současné době existují produkty (Mono, DotGNU) nezávislých open source iniciativ implementujících .NET runtime pro operační systémy unixového typu (Linux, MacOS).

Nad vrstvou CLR se nachází množina základních tříd rámce .NET. Tato množina tříd je velmi podobná sadě tříd známých z STL[8], MFC[9], ATL[10] nebo Java. Tyto třídy zajišťují základní funkcionalitu vstupu a výstupu, manipulaci s řetězci znaků, správu zabezpečení, síťovou komunikaci, správu vláken a řadu dalších bazálních operací.

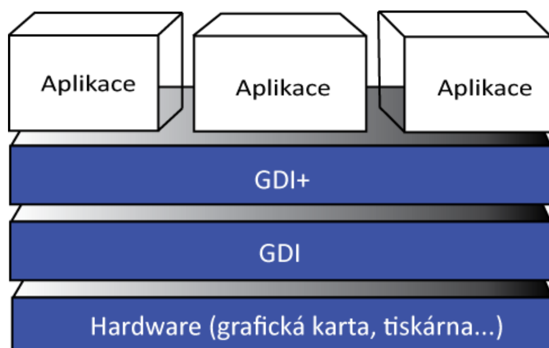
Nad touto sadou základních tříd je vrstva tříd, které ji dále rozšiřují a poskytují podporu správy dat a manipulaci s XML. Jsou to především třídy specializované na manipulaci s perzistentními daty uloženými v databázích (SQL, ADO.NET) a třídy zaměřené na vytváření, čtení, prohledávání a transformaci XML. (Thai, et al., 2002)

3.1.3 **Nástroje vektorové grafiky v .NET**

Při práci s vektorovou grafikou, ať už je jedná o programování vizuálních komponent, grafickou prezentaci dat či jen prosté kreslení statických geometrických útvarů, je celkem samozřejmě používána celá řada pomocných objektů, jako jsou pera pro definování základních charakteristik čar a křivek (barva, tloušťka, typ…), štětce pro vyplňování ploch nebo různé styly písma. Vlastní programový kód pro takové kreslení je pak poměrně jednoduchý, což je dáno tím, že se použité metody opírají o technologii nazvanou GDI+[11]. Tato technologie je fyzicky realizována knihovnou se sadou tříd, které jsou k dispozici pro účely reprezentace grafických objektů a jejich transformací do výstupních zařízení jako jsou monitory nebo tiskárny. Stejně jako ostatní třídy v .NET jsou

[8] Standard Template Library
[9] Microsoft Foundation Classes
[10] Active Template Library
[11] Graphical Device Interface

i třídy v GDI+ založeny na velmi intuitivním a snadno použitelném objektovém modelu. Jak je naznačeno na Obrázek 2, GDI+ je vlastně jakousi obálkou kolem GDI.



Obrázek 2 - Graphical Device Interface
Zdroj: vlastní tvorba

GDI maskuje rozdíly mezi různými grafickými adaptéry (od různých výrobců, obvykle s rozdílnou instrukční sadou), což umožňuje jednoduše volat API funkce pro vykonání specifické úlohy, přičemž GDI interně řeší, jak tyto abstraktní funkce transponovat na instrukční kód konkrétní grafické karty. Další významnou vlastností GDI je, že lze poměrně snadno využívat více výstupních zařízení, tj. kromě monitoru je možno přesměrovat výstup na tiskárnu s obdržením identické reprezentace grafických objektů. Tato vlastnost je základem způsobu editace dokumentů na počítači, který je nazýván WYSIWYG[12].

Ačkoli GDI zpřístupňuje vývojářům relativně vysokou úroveň API, nepatří její použití mezi triviální a pohodlné, a to především kvůli nutnosti volat funkce v jazyce C, mnohdy se značným počtem parametrů. Z toho důvodu je mezi GDI a aplikacemi vrstva GDI+, která poskytuje více intuitivní a na dědičnosti založený objektový model. I když je GDI+ prezentována pouze jako wrapper GDI, Microsoft dokázal jejím prostřednictvím nabídnout programátorům nové vlastnosti a zajistil také v určité oblasti výkonové zdokonalení. (Microsoft, 2012)

[12] WYSIWYG je akronym anglické věty „What You See Is What You Get“, tj. „To co vidíš, to dostaneš.“

3.1.4 Jmenné prostory pro 2D grafiku

Namespace	Popis
`System.Drawing`	Většina tříd, struktur, výčtových typů a delegátů nesoucích základní funkcionalitu vykreslování v .NET.
`System.Drawing.Drawing2D`	Sada více specializovaných tříd poskytujících pokročilejší efekty a techniky vykreslování dvoudimenzionální a vektorové grafiky.
`System.Drawing.Imaging`	Třídy podporující zpracování a manipulaci s obrazem (podpora grafických formátů BMP, GIF, JPEG, PNG, WMF…)
`System.Drawing.Printing`	Poskytuje služby související s tiskem z prostředí aplikací Windows Forms.
`System.Drawing.Design`	Obsahuje třídy, které rozšiřují logiku návrhu uživatelského rozhraní (předdefinovaná dialogová okna apod.)
`System.Drawing.Text`	Poskytuje pokročilou GDI+ typografickou funkcionalitu. Třídy v tomto jmenném prostoru umožňují uživatelům vytvářet a využívat kolekce fontů.

Tabulka 1 - Tabulka jmenných prostorů pro 2D grafiku
Zdroj: vlastní tvorba

3.1.5 Jmenný prostor System.Drawing

Tato kapitola se blíže zaměřuje na jmenný prostor *System.Drawing* frameworku .NET. Ze všech 38 tříd, které tento obor názvů[13] obsahuje, bude zvláštní pozornost věnována zejména třídě *Graphics*, ve které jsou mimo jiné implementovány metody vykreslování základních liniových i plošných grafických prvků, jako např. úsečky, lomené čáry, obdélníky, obecné mnohoúhelníky, elipsy, oblouky, textové řetězce[14] apod. Nechybí však ani možnosti kreslení aproximačních a interpolačních křivek, segmentů vzniklých napojením těchto křivek a dále pak obecných grafických cest. Tyto nástroje pak obvykle dobře poslouží k vytvoření i značně složitých grafických vektorových útvarů. Z výše uvedeného výčtu grafických funkcí budou zdůrazněny především ty, které jsou využity v praktické části, tj. v jednoduchém prohlížeči souborů ve formátu SVG.

[13] Překlad anglického *namespace* není vždy jednotný. Např. Microsoft uvádí ve své dokumentaci jako český ekvivalent „obor názvů“, jiná literatura naopak upřednostňuje termín „jmenný prostor“.
[14] Tyto grafické prvky označujeme angl. termínem *output primitives*.

3.1.5.1 Kreslení úsečky

Graphics.DrawLine

Na vykreslení úsečky nabízí .NET Framework ve třídě *Graphics* metodu `DrawLine`. Tato metoda nakreslí úsečku spojující dva body. Způsob zadání souřadnic bodů může být s ohledem na řešení konkrétního problému různý, stejně jako jejich datový typ. Metoda je však na různé varianty zadání vstupních parametrů připravena. Slovy objektově-orientovaného programování říkáme, že je přetížena. V případě metody `DrawLine` je její základní deklarace přetížena dokonce třikrát. To ukazuje následující tabulka.

Varianty přetížení metody `DrawLine`	
Deklarace	**Popis**
`public void DrawLine(Pen pen, Point pt1, Point pt2)`	pt1 a pt2 představují struktury Point, jejichž prvky X a Y jsou celočíselné hodnoty. pt1 definuje počáteční a pt2 koncový bod úsečky
`public void DrawLine(Pen pen, PointF pt1, PointF pt2)`	Platí totéž co pro výše uvedené DrawLine pouze s tím rozdílem, že struktura PointF uchovává hodnoty souřadnic X a Y jako reálná čísla.
`public void DrawLine(Pen pen, int x1, int y1, int x2, int y2)`	Vstupem jsou čtyři celočíselné hodnoty. Dvojice x1, y1 určuje počáteční bod úsečky a x2,y2 koncový bod úsečky.
`public void DrawLine(Pen pen, float x1,float y1, float x2,float y2)`	Vstupem jsou čtyři reálné hodnoty souřadnic bodů. Stejně jako v předchozím případě určuje x1,y1 počáteční bod a x2,y2 koncový bod.

Tabulka 2 - Přetížení metody DrawLine
Zdroj: https://msdn.microsoft.com/en-us/library/system.drawing.graphics.drawline%28v=vs.110%29.aspx

Co se týče parametru `pen`, který vystupuje ve všech čtyřech implementacích téže metody `DrawLine` jako instance třídy *Pen*, ten určuje barvu, sílu a styl čáry. V následujících kapitolách popisujících některé významné elementy SVG z hlediska vykreslování grafických primitiv bude zřejmé, že parametr `pen` je analogií atributu `style` těchto elementů. Je vidět, že vykreslení čáry, resp. úsečky představuje s použitím metody .NET velmi triviální operaci. Jiná situace by však nastala, pokud by mělo být vykreslování řízeno uživatelsky definovanými algoritmy.

Rasterizace úsečky

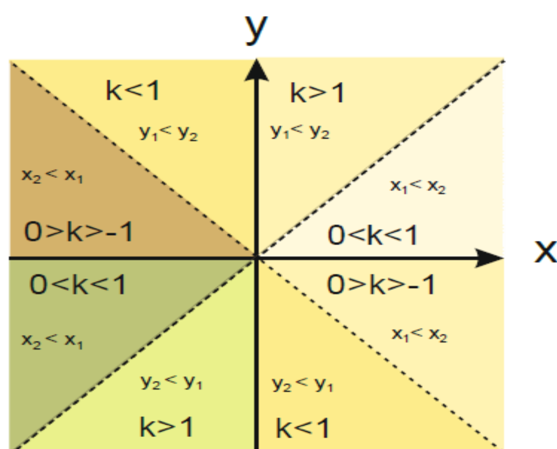
Rasterizací se obecně rozumí proces převodu vektorových objektů do systému posloupnosti obrazových bodů. Tento proces lze chápat jako výpočet polohy a barvy obrazových bodů – pixelů (Špička, a další, 2012). Ačkoli se práce zabývá problematikou vektorových grafických útvarů, je třeba si uvědomit, že zobrazovací zařízení je tvořeno rastrem, v němž je možno se v každé situaci pohybovat vždy jen diskrétně, po celých pixelech. Z toho vyplývá, že úsečky (ani jiná grafická primitiva) nemohou být do tohoto rastru zakresleny úplně přesně a uvedené algoritmy budou řešit, jak umístit pixely co nejefektivněji při požadované kvalitě a přijatelné výpočetní složitosti. S odkazem na praktickou část práce zde budou uvedeny dva základní algoritmy pro rasterizaci úsečky – DDA[15] a Bresenhamův algoritmus. Oba algoritmy vycházejí ze směrnicového tvaru přímky, tj.

$y = kx + q \quad (1)$

Je-li úsečka zadána body $P_1[x_1, y_1]$ a $P_2[x_2, y_2]$, pak pro směrnici k platí vztah:

$$k = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} \qquad (2)$$

Hodnota směrnice určuje sklon úsečky a číselně je rovna tangentě úhlu, který úsečka svírá s kladnou částí osy *x*. Následující obrázek ukazuje hodnoty směrnice v jednotlivých oktantech kartézské souřadnicové soustavy.

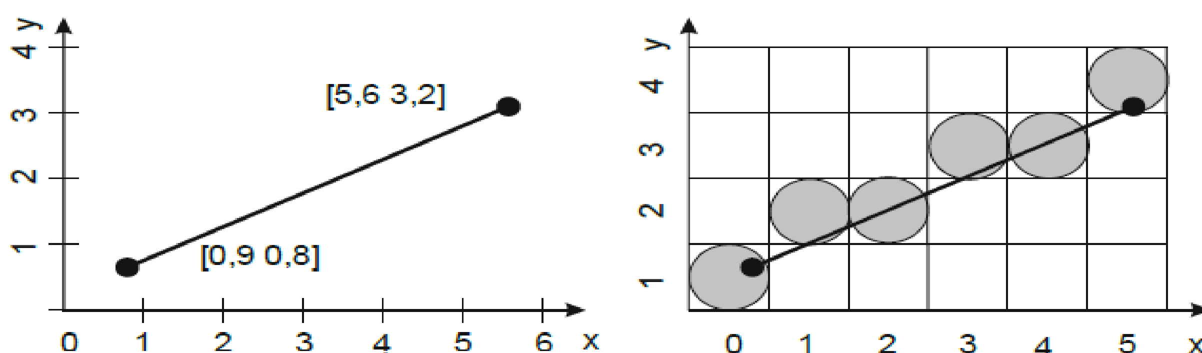


Obrázek 3 - Hodnoty směrnice k v oktantech souřadnicové soustavy
Zdroj: (Špička, a další, 2012)

---

[15] Digital Differential Analyzer

Z obrázku je zřejmé, že pokud je $|k| < 1$, pak úsečka „přiléhá" k ose *x*, v opačném případě k ose *y*. Pokud úsečka k ose přiléhá, pak se tato osa nazývá osou řídící a v jejím směru je přičítán vždy jeden pixel. Diagonální úsečky, tj. ty, jejichž $|k| = 1$ mohou mít řídící osu libovolnou. Z definice směrnice přímky (úsečky) plyne, že je to obecně reálné číslo. Stejně tak i některé varianty (přetížení) metody `DrawLine` předpokládají vstup neceločíselných souřadnic. V zobrazovacím rastru však dojde k jejich zaokrouhlení a chyba tohoto zaokrouhlení nebývá významná, pokud se nejedná o koncové body a není tím narušena návaznost lomené čáry či křivky. Následující obrázky znázorňují spojité a diskrétní vykreslení úsečky v rovině (Špička, a další, 2012).



Obrázek 4 - spojité a diskrétní vykreslení úsečky
Zdroj: (Špička, a další, 2012)

Algoritmus DDA

Tento velmi jednoduchý algoritmus je založen na přičítání konstantních přírůstků k oběma souřadnicím. Velikost přírůstků závisí na tom, která osa je osou řídící. Pokud je řídící osou *x*, pak k *x*-ové souřadnici přičítáme jedničku a k *y*-ové hodnotu směrnice *k*. Pokud je řídící osou *y*, pak k *y*-ové souřadnici přičítáme jedničku a k *x*-ové hodnotu $\frac{1}{k}$.
Hodnotu souřadnice, ke které byl přičten neceločíselný přírůstek hodnoty směrnice (či její hodnoty reciproké) je nutno před vykreslením pixelu zaokrouhlit. Formálně lze algoritmus DDA popsat následujícím způsobem: (Špička, a další, 2012)
Předpoklad: úsečka je zadána počátečním a koncovým bodem $P_1[x_1, y_1]$ a $P_2[x_2, y_2]$ v celočíselných souřadnicích.

1) **Jestliže** je $x_1 = x_2$ a současně $y_1 = y_2$, **pak** se jedná o bod. *Plot*(x_1, y_1) a ukonči činnost.
2) **Jestliže** je $x_1 = x_2$ a současně $y_1 \neq y_2$, **pak** se jedná o svislou úsečku.
   2a) **Jestliže** $y_2 < y_1$, **potom** zaměň hodnoty y_1, y_2

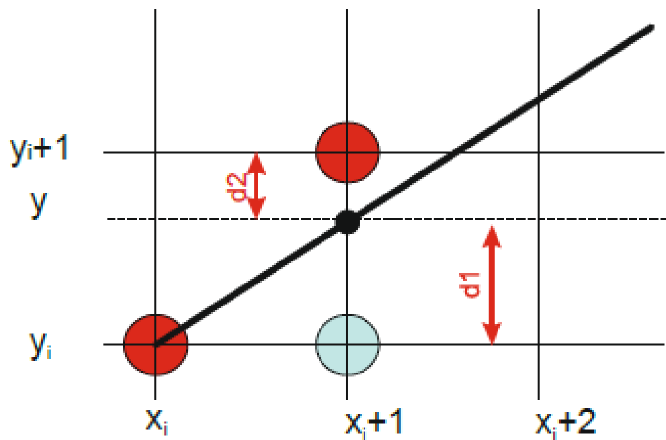
2b) **Inicializuj** $y = y_1$
2c) **Dokud** je $y \leq y_2$, **opakuj**
$y = y + 1;$
Plot(x_1, y)
2d) **Ukonči** činnost
3) **Pokud** je $x_1 \neq x_2$ a současně $y_1 = y_2$, **pak** se jedná o vodorovnou úsečku.
3a) **Jestliže** $x_2 < x_1$, **potom** zaměň hodnoty x_1, x_2
3b) **Inicializuj** $x = x_1$
3c) **Dokud** je $x \leq x_2$, **opakuj**
$x = x + 1;$
Plot(x, y_1)
3d) **Ukonči** činnost
4) **Spočítej** směrnici k podle výše uvedeného vzorce (2).
Jestliže $k \in < -1,1 >$, pak řídící osa je osa x
4a) **Jestliže** $x_2 < x_1$, **potom** zaměň hodnoty x_1, x_2 a y_1, y_2
4b) **Inicializuj** $y = y_1$
4c) **Dokud** je $x \leq x_2$, **opakuj**
$x = x + 1;$
Plot$(x, Round(y))$; $y = y + k$
5c) **Ukonči** činnost
5) Jestliže $|k| > 1$, pak řídící je osa y
5a) **Jestliže** $y_2 < y_1$, **potom** zaměň hodnoty x_1, x_2 a y_1, y_2
5b) **Inicializuj** $x = x_1$
5c) **Dokud** je $y \leq y_2$, **opakuj**
$y = y + 1;$
Plot$(Round(x), y)$; $x = x + \frac{1}{k}$
5d) **Ukonči** činnost

Bresenhamův algoritmus

Tento algoritmus navrhl v šedesátých letech minulého století americký inženýr pracující pro společnost IBM Jack Elton Bresenham. Motivací bylo sestavit algoritmus pro vykreslování obrazu na plotteru, kde bylo potřeba maximalizovat hospodárnost a efektivitu pohybu jeho pera. I v dnešní době patří mezi nejběžnější algoritmy užívané pro rasterizaci úsečky. Jeho nespornou výhodou oproti algoritmu DDA nebo algoritmu triviálnímu[16] je fakt, že pracuje pouze s celočíselnou aritmetikou (nepoužívá operace dělení ani práci s datovým typem reprezentujícím čísla v plovoucí řádové čárce) (Špička, a další, 2012).

[16] Ten zde uveden není, neboť se neobjevuje v praktické části, ale jde o jednoduchý algoritmus, který vychází z parametrického vyjádření úsečky.

Princip algoritmu je velmi jednoduchý a je založen na výpočtu rozhodovací proměnné, jejíž znaménko určí v každé iteraci, která z možných variant pro souřadnice bodu bude použita. Algoritmus se snáze pochopí s využitím následujícího obrázku.



Obrázek 5 - Varianty souřadnic pro nový bod úsečky
Zdroj: (Špička, a další, 2012)

Nechť je dána situace znázorněná výše. Tučně je na obrázku vyznačena „ideální" úsečka a řídící osou je v tomto případě osa *x*, neboť hodnota směrnice je jistě z intervalu (0,1). Je dán bod na souřadnicích $[x_i, y_i]$. Nyní je třeba rozhodnout, který další bod bude vykreslen. Může to být buď bod na souřadnicích $[x_i + 1, y_i]$ nebo bod $[x_i + 1, y_i + 1]$. Z těchto dvou bodů je vybrán ten, jehož vzdálenost od ideální úsečky je menší. Rozdíly mezi souřadnicemi *y* středů uvedených bodů a souřadnicí *y* na úsečce v bodě $x_i + 1$ jsou označeny d_1a d_2, jak je zřejmé z obrázku. Dosazením souřadnice $x_i + 1$ do směrnicové rovnice přímky $y = kx + q$ je získána souřadnice *y*, tj.

$$y = k(x_i + 1) + q \quad (3)$$

Pro vzdálenosti d_1 a d_2 pak platí:

$$d_1 = y - y_i = k(x_i + 1) + q - y_i \quad (4)$$
$$d_2 = y_i + 1 - y = y_i + 1 - k(x_i + 1) - q \quad (5)$$

Rozdíl těchto dvou vzdáleností je vyjádřen následovně:

$$\Delta d = d_1 - d_2 = k(x_i + 1) + q - y_i - y_i - 1 + k(x_i + 1) + q \quad (6)$$

a po úpravě:

$$\Delta d = 2k(x_i + 1) - 2y_i + 2q - 1 \quad (7)$$

Znaménko Δd určí, který ze dvou uvažovaných bodů leží blíže ideální úsečce. Pokud je hodnota Δd záporná, potom leží blíže bod na souřadnicích $[x_i + 1, y_i]$. Je-li hodnota Δd kladná, blíže bude pixel na souřadnicích $[x_i + 1, y_i + 1]$.

Pokud je směrnice k ve výše uvedeném vztahu přepsána, přejde rovnice na tvar:

$$\Delta d = 2\frac{\Delta y}{\Delta x}(x_i + 1) - 2y_i + 2q - 1 \quad (8)$$

Pro odstranění jediné neceločíselné hodnoty, tj. podílu $\frac{\Delta y}{\Delta x}$, je třeba vynásobit celou rovnici Δx.Tím se smysl použití znaménka nemění.

$$\Delta d\Delta x = 2\Delta y(x_i + 1) - 2\Delta x y_i + 2\Delta x q - \Delta x$$

$$\Delta d\Delta x = 2\Delta y x_i + 2\Delta y - 2\Delta x y_i + 2\Delta x q - \Delta x$$

$$p_i = \Delta d\Delta x = 2\Delta y x_i - 2\Delta x y_i + 2\Delta y + \Delta x(2q - 1) \quad (9)$$

Hodnota p_i se nazývá i-tá predikce. Červeně označená část rovnice je konstanta.

Pro výpočet predikce p_{i+1} v následujícím kroku platí:

$$p_{i+1} = 2\Delta y x_{i+1} - 2\Delta x y_{i+1} + 2\Delta y + \Delta x(2q - 1) \quad (10)$$

Rozdíl prediktorů $p_{i+1} - p_i$ je pak následující:

$$p_{i+1} - p_i = 2\Delta y x_{i+1} - 2\Delta x y_{i+1} + konst - 2\Delta y x_i + 2\Delta x y_i - konst$$

$$p_{i+1} - p_i = 2\Delta y(x_{i+1} - x_i) - 2\Delta x(y_{i+1} - y_i) \quad (11)$$

Pokud je řídící osou osa x, pak platí $x_{i+1} - x_i = 1$ (krok v řídící ose je vždy 1). Pro prediktor p_{i+1} pak lze psát:

$$p_{i+1} = p_i + 2\Delta y - 2\Delta x(y_{i+1} - y_i) \quad (12)$$

Je-li $p_i < 0$, potom $y_i = y_{i+1}$ a $p_{i+1} = p_i + 2\Delta y$

Je-li $p_i \geq 0$, potom $p_{i+1} = p_i + 2\Delta y - 2\Delta x(y_{i+1} - y_i)$

Počáteční hodnota $p_1 = 2\Delta y - \Delta x$ se získá dosazením souřadnic $[x_1, y_1]$ počátečního bodu úsečky do rovnice pro výpočet predikce p_i (9).

Po nastudování příslušných teoretických partií zformuloval autor této práce Bresenhamův algoritmus bez vazby na konkrétní programovací jazyk následujícím způsobem:

Předpoklad: úsečka je zadána počátečním a koncovým bodem $P_1[x_1, y_1]$ a $P_2[x_2, y_2]$ v celočíselných souřadnicích. První tři kroky jsou shodné s popisem algoritmu DDA a řeší

situace, kdy je úsečka pouhým bodem nebo je její směrnice nulová. Začíná se tedy až krokem číslo 4, kde už se algoritmy liší.

4) **Spočítej** $\Delta x = |x_2 - x_1|$ a $\Delta y = |y_2 - y_1|$
  4a) **Pokud** je $\Delta x > \Delta y$, **pak** je řídící osa x
  4b) **Inicializuj** konstanty:
    $k_1 = 2 * \Delta y$
    $k_2 = 2 * \Delta y - \Delta x$
    $predictor = k_1 - \Delta x$
    incY = 1
  4c) **Jestliže** je $x_2 < x_1$, **potom** zaměň x_1, x_2 a y_1, y_2
  4d) **Jestliže** je $y_2 < y_1$, potom incY = -1
5) $Plot(x_1, y_1)$
6) **Inicializuj** $y = y_1$
7) **Dokud** je $x \leq x_2$, **opakuj**:
  7a) **Jestliže** $predictor \leq 0$, **potom** $predictor = predictor + k_1$ **jinak** $predictor = predictor + k_2$
  $y = y$ +incY
  Plot(x,y)
  x = x +1
8) Jestliže je $\Delta x < \Delta y$, pak je řídící osa y
9) **Inicializuj** konstanty:
  $k_1 = 2 * \Delta x$
  $k_2 = 2 * (\Delta x - \Delta y)$
  $predictor = k_1 - \Delta y$
  incX = 1
10) **Jestliže** je $y_2 < y_1$, **potom** zaměň x_1, x_2 a y_1, y_2
11) **Jestliže** $x_2 < x_1$, **potom** incX = -1
12) **Inicializuj** $x = x_1$
13) **Dokud** je $y \leq y2$, **opakuj**
  13a) **Jestliže** $predictor \leq 0$, **potom** $predictor = predictor + k_1$ **jinak** $predictor = predictor + k_2$
  $x = x$ +incX
  Plot(x,y)
  y = y +1
14) **Konec**

3.1.5.2 Kreslení Bézierovy křivky

Graphics.DrawBezier

Třída *Graphics* ve jmenném prostoru *System.Drawing* poskytuje pro vykreslování Bézierových křivek metodu `DrawBezier`. Metoda však neumožňuje realizaci křivek obecného stupně n, ale pouze tzv. Bézierových kubik, tedy křivek třetího stupně, jež budou popsány dále. Metoda `DrawBezier` je, podobně jako `DrawLine`, přetížená, tj. existují

varianty metody téhož jména, ale s rozdílnou sadou parametrů volání. Přehled přetížení metody `DrawBezier` přináší následující tabulka.

Varianty přetížení metody `DrawBezier`	
Deklarace	**Popis**
`public void DrawBezier(Pen pen, Point pt1, Point pt2, Point pt3, Point pt4)`	Zadání souřadnic s využitím čtyř struktur typu Point. pt1 a p4 určují počáteční a koncový bod křivky, pt2 a pt3 první a druhý tzv. řídící bod.
`public void DrawBezier(Pen pen, PointF pt1,PointF pt2, PointF pt3,PointF pt4)`	Platí totéž co výše pouze s tím rozdílem, že struktura `PointF` uchovává souřadnice bodu jako hodnoty datového typu `float`.
`public void DrawBezier(Pen pen, float x1,float y1, float x2,float y2, float x3,float y3, float x4,float y4)`	Bézierova kubika zadaná čtyřmi páry souřadnic, jejichž význam byl uveden již v první deklaraci. Tento způsob zadání souřadnic bude s předcházejícím široce využíván v praktické části této práce.

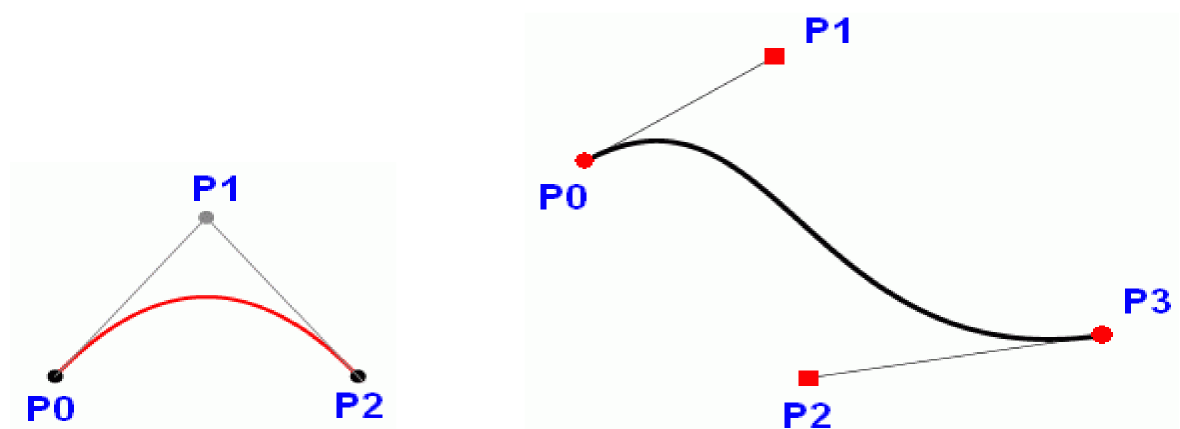
Tabulka 3 - Přetížení metody DrawBezier
Zdroj: https://msdn.microsoft.com/en-us/library/system.drawing.graphics.drawbezier%28v=vs.110%29.aspx

Dříve než bude objasněn princip nejběžnějších algoritmů pro výpočet bodů Bézierových křivek, je nutné nejprve položit alespoň minimální teoretické základy pro přiblížení a pochopení dané problematiky. V dalším textu bude velmi frekventovaným pojmem termín křivka. Při velmi zjednodušené formální definici se křivou k rozumí v matematice spojité zobrazení intervalu reálných čísel do nějakého matematického prostoru (Euklidovského, variety, topologického). Toto zobrazení se nazývá parametrické vyjádření křivky a je z hlediska počítačového modelování křivek zdaleka nejvýhodnější. Množina $\{k(x); x \in I\}$ se nazývá geometrický obraz křivky. Mají-li složky k_i křivky k na otevřeném intervalu (a, b) spojité derivace až do r-tého řádu, pak to znamená, že se jedná o křivku r-té třídy (Rektorys, 1988). Rovnici obrazu rovinné křivky lze často vyjádřit také explicitně ve formě funkční závislosti proměnných x a y, tj. $y = f(x)$ nebo implicitně předpisem $f(x, y) = 0$. Bodová rovnice křivky se zapisuje ve tvaru $Q(t) = [x(t); y(t)]$. Křivky mohou být určeny také pomocí tzv. řídících (opěrných) bodů, obvykle s doplněním dalších okrajových podmínek. V zásadě existují dva různé přístupy k vytváření takových křivek:

- Interpolace – jsou zadány řídící body a (obvykle) okrajové podmínky a výsledná hladká křivka prochází všemi opěrnými body. Typickými příklady jsou Lagrangeova interpolace, Hermitova interpolace atd.
- Aproximace – jsou zadány řídící body a výsledkem je typicky hladká křivka, která prochází pouze některými, v krajním případě žádnými, řídícími body. Mezi aproximační křivky patří Bézierovy křivky, Coonsovy kubiky, spline křivky, B-spline křivky a neuniformní racionální spline – NURBS.

Bézierovy křivky zavedli v šedesátých letech minulého století francouzští inženýři Paul de Casteljau zaměstnaný u firmy Citroen a Pierre Bézier z konkurenční automobilky Renault. Ačkoli de Casteljau vypracoval matematický model křivek a ploch o tři roky dříve než skupina kolem P. Béziera, dodnes jsou křivky nazývány Bézierovými, neboť Bézier v roce 1962 výsledky své práce publikoval jako první. Bézierovy křivky jsou v současné době asi nejpopulárnější aproximační křivky. Používají se nejen pro modelování ve 2D a 3D, ale např. také pro popis TrueType a PostScript fontů (Žára, a další, 2004).

Bézierovy křivky jsou určeny tzv. řídícím polygonem – lomenou čárou určenou polohovými vektory bodů $P_0, P_1, \dots, P_n$.



Obrázek 6 - Kvadratická a kubická Bézierova křivka
Zdroj: http://www.root.cz/clanky/vytvarime-krivky-v-postscriptu/

Bézierova křivka stupně n určená řídícím polygonem $P_0, P_1, \dots P_n$ je dána vztahem

$Q(t) = \sum_{i=0}^{n} B_i^n(t) P_i = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i P_i$ a $t \in \langle 0,1 \rangle$ (13)

kde $B_i^n(t)$ jsou tzv. Bernsteinovy polynomy stupně n, které tvoří bázi prostorů polynomů stupně n. Bernsteinovy polynomy mají následující vlastnosti: (Žára, a další, 2004)

- $B_i^n(t) > 0$ pro $t \in \langle 0,1 \rangle$; $i = 0,1, \dots n$ (14)

- Bernsteinovy polynomy je možné generovat rekurzivně podle vztahu:

$$B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) \quad (15)$$

- Platí: $\sum_{i=0}^{n} B_i^n(t) = 1$ pro $\forall t$ (16)
- Symetrie: $B_i^n(t) = B_{n-1}^n(1-t)$ (17)

Dále jsou uvedeny některé důležité vlastnosti Bézierových křivek: (Žára, a další, 2004)

- Počátečním bodem Bézierovy křivky je bod P_0 řídícího polygonu, koncovým bodem je bod P_n řídícího polygonu.
- Bézierova křivka se v počátečním bodě dotýká první strany řídícího polygonu a v koncovém bodě se dotýká poslední strany řídícího polygonu.
- Mějme dvě Bézierovy křivky: $P(t)$ určenou řídícím polygonem $P_0, P_1, \dots P_n$ a křivku $Q(t)$ určenou řídícím polygonem $Q_0, Q_1, \dots Q_m$. Jestliže $P_n = Q_0$, potom jsou Bézierovy křivky napojeny ve třídě C_0. Pokud leží body $P_{n-1}, P_n = Q_0, Q_1$ na jedné přímce, potom jsou Bézierovy křivky napojeny ve třídě G_1. Pokud navíc platí, že $n(P_{n-1} - P_n) = m(Q_1 - Q_0)$, potom jsou napojeny ve třídě C_1.
- Jelikož libovolný polynom stupně n je možné vyjádřit jako lineární kombinaci Bernsteinových polynomů, je možné libovolnou polynomiální parametrickou křivku vyjádřit jako Bézierovu křivku.
- Bézierova křivka je vždy obsažena v konvexním obalu svého řídícího polygonu.
- Bézierovy křivky vykazují tzv. afinní invariantnost, tj. nezáleží na tom, zda afinně transformujeme Bézierovu křivku nebo nejprve transformujeme její řídící polygon a následně křivku dopočteme. V obou případech dostaneme tytéž výsledky.
- Pokud jsou řídící body Bézierovy křivky kolineární, tj. leží na přímce, potom má Bézierova křivka tvar úsečky.[17]
- Počet průsečíků libovolné přímky s Bézierovou křivkou je nejvýše roven počtu průsečíků této přímky s řídícím polygonem.[18]
- Pokud přidáme řídící bod nebo změníme polohu některého řídícího bodu, změní se tvar celé křivky.

[17] Tento případ nazýváme angl. termínem linear precision
[18] Tuto vlastnost označujeme jako variation diminishing property.

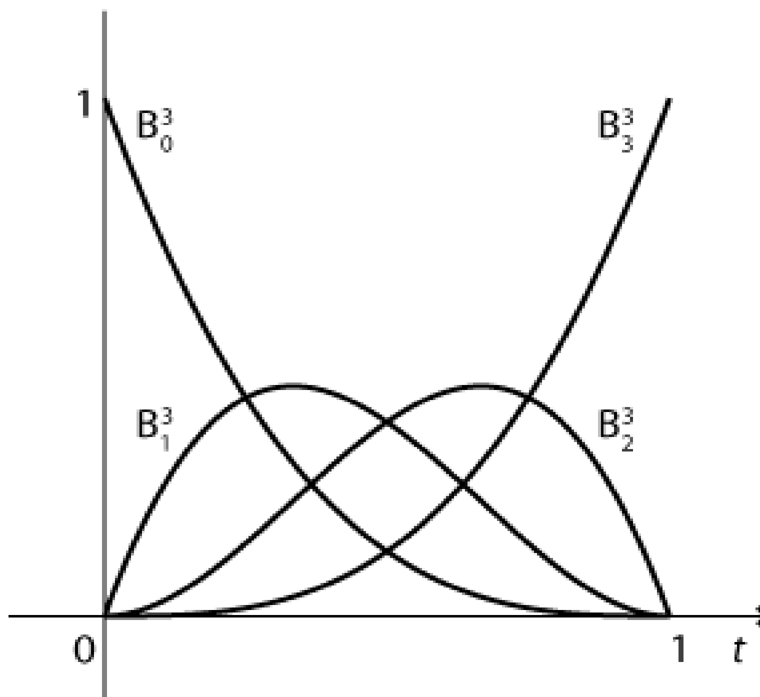
Naprosto nejpoužívanější variantou Bézierových křivek v počítačové grafice jsou Bézierovy kubiky, tj. Bézierovy křivky třetího stupně. Třetí stupeň je nejnižší stupeň, který může zajistit spojitost až do třídy C_2, což je pro potřeby počítačové geometrie zcela postačující.

Bézierova kubika je určena vztahem:

$$Q(t) = \sum_{i=0}^{3} B_i^3(t) P_i \quad (18)$$

přičemž Bernsteinovy polynomy třetího stupně mají tvar:

$$B_0^3(t) = (1-t)^3$$
$$B_1^3(t) = 3t(1-t)^2$$
$$B_2^3(t) = 3t^2(1-t)$$
$$B_3^3(t) = t^3$$



Obrázek 7 - Bernsteinovy polynomy 3.stupně
Zdroj: http://home.zcu.cz/~smolik/zpg/cviceni/cv_08.html

Bézierovu kubiku můžeme také zapsat v maticovém tvaru:

$$Q(t) = [t^3 \; t^2 \; t \; 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \quad (19)$$

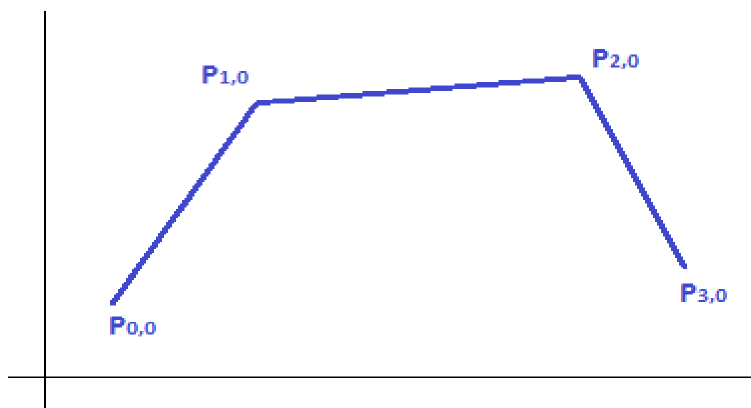
(Žára, a další, 2004)

Algoritmus de Casteljau

Motivací pro sestavení tohoto algoritmu byla provázková konstrukce paraboly. Algoritmus je založený na opakovaném použití lineární interpolace a zobecňuje speciální případ konstrukce paraboly pro křivky vyšších stupňů (Žára, a další, 2004).

Pro lepší pochopení algoritmu bude postup rozložen na jednotlivé kroky.

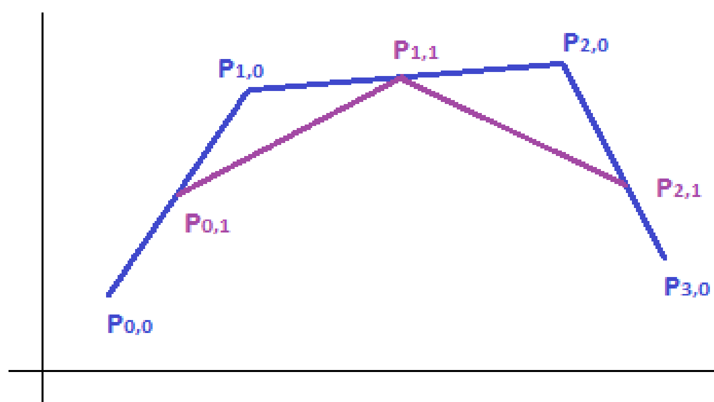
1) Je dán pro jednoduchost řídící polygon o čtyřech vrcholech $P_{0,0}, P_{0,1}, P_{0,2}, P_{0,3}$ a parametr $t \in \langle 0,1 \rangle$



Obrázek 8 - Vrcholy řídícího polygonu
Zdroj: vlastní tvorba

V prvním kroku se provede lineární interpolace podle následujících vztahů pro všechny dvojice po sobě jdoucích bodů a tím jsou získány body nové.

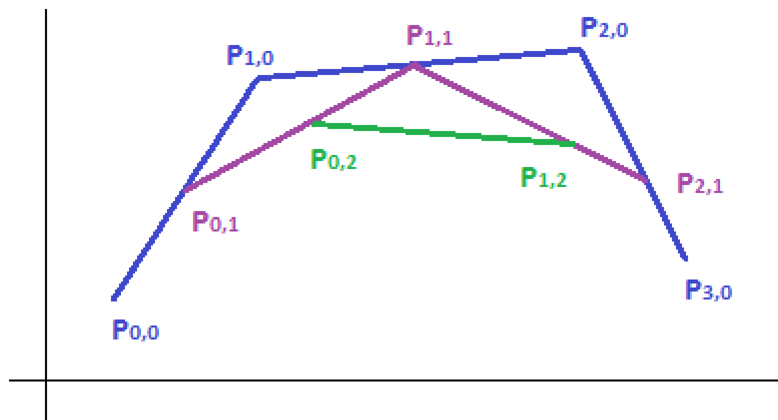
$$P_{0,1} = (1-t)P_{0,0} + tP_{1,0}$$
$$P_{1,1} = (1-t)P_{1,0} + tP_{2,0}$$
$$P_{2,1} = (1-t)P_{2,0} + tP_{3,0}$$



Obrázek 9 - Nové body po 1.interpolaci
Zdroj: vlastní tvorba

2) Ve druhém kroku se postup opakuj pro nově získané body z předchozího kroku algoritmu pro stejný parametr t.

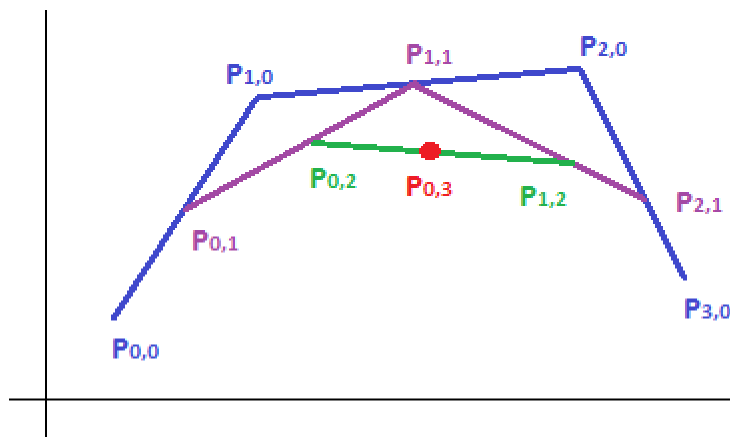
$$P_{0,2} = (1 - t)P_{0,1} + tP_{1,1}$$
$$P_{1,2} = (1 - t)P_{1,1} + tP_{2,1}$$



Obrázek 10 - Interpolace ve 2.kroku
Zdroj: vlastní tvorba

3) Ve třetím kroku se postup znovu opakuje pro body z druhého kroku, opět se stejným parametrem t. Vypočítaný bod již je na Bézierově křivce.

$$P_{0,3} = (1 - t)P_{0,2} + tP_{1,2}$$



Obrázek 11 - Poslední krok - bod na křivce
Zdroj: vlastní tvorba

4) Pro získání všech bodů Bézierovy křivky, je nutné postup opakovat pro všechna $t \in \langle 0,1 \rangle$.

Protože $P(0) = P_{0,0}$ a $P(1) = P_{3,0}$ prochází Bézierova křivka vždy prvním a posledním bodem řídícího polygonu. Pokud se postupně dosadí do posledního vztahu algoritmu ze všech předcházejících, pak platí:

$$P(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t) t^2 P_2 + t^3 P_3 = \sum_{i=0}^{3} \binom{3}{i} (1-t)^{3-i} t^i P_i \quad (20)$$

Jelikož stupeň parametru t je nejvýše 3, jedná se o kubickou Bézierovu křivku. Je zřejmé, že opakování postupu lineární interpolace vytváří trojúhelníkové schéma.

$$\begin{matrix} P_{0,0} & & & \\ & P_{0,1} & & \\ P_{1,0} & & P_{0,2} & \\ & P_{1,1} & & P_{0,3} \\ P_{2,0} & & P_{1,2} & \\ & P_{2,1} & & \\ P_{3,0} & & & \end{matrix} \quad (21)$$

Proces je možné zobecnit pro libovolný počet bodů řídícího polygonu. Pokud má řídící polygon n bodů, je nutné provést $n-1$ kroků algoritmu, aby se získal bod na křivce. Při výpočtu se postupuje podle rekurentního vztahu: (Žára, a další, 2004)

$$P_{j,i}(t) = \begin{cases} (1-t)P_{j-1,i-1}(t) + tP_{j-1,i}(t); pro\ j > 0 \\ jinak\ P_i \end{cases} \quad (22)$$

(Špička, a další, 2012)

Autor, opět s užitím výše uvedeného matematického aparátu, slovně formuloval algoritmus následujícím způsobem:

1) **Inicializuj** pole *Vrcholy* vrcholy řídícího polygonu
2) **Inicializuj** parametr $t = 0$
3) **Inicializuj** konstantu $eps = 0.05$
3) **Inicializuj** proměnnou $n = Vrcholy.Počet - 1$
4) **Dokud** je $t \leq 1$, **opakuj**:
    $novýBod = VypočítejNovýBod(n, 0, t)$
    $Plot(novýBod.X, novýBod.Y)$
    $t = t + eps$
5) **Konec**

Pro výpočet nového bodu na křivce je použit rekurzivní algoritmus.

1) **Nastav** hodnoty proměnných $i, j\ a\ param$
2) **Jestliže** $i = 0$, **potom** vrať hodnotu prvku $Vrcholy[j]$
3) **Spočítej** souřadnice bodu $p1$ pomocí tohoto postupu s hodnotami $i-1, j, param$
4) **Spočítej** souřadnice bodu $p2$ pomocí tohoto postupu s hodnotami $i-1, j+1, param$

5) **Vrať** souřadnice bodu spočítané podle vzorce:

$(1 - t) * p1.X + t * p2.X, (1 - t) * p1.Y + t * p2.Y$

6) **Konec**

Algoritmus s využitím Bernsteinových polynomů

Analytický postup výpočtu bodu Bézierovy křivky byl uveden výše, tudíž lze zformulovat sled operací, kterými bude vykreslena Bézierovu křivka stupně n. (Hardy, et al., 2008) Algoritmus je opět pro přehlednost a srozumitelnost dekomponován na několik dílčích částí.

// Hlavní blok algoritmu

1) **Inicializuj** pole *Vrcholy* vrcholy řídícího polygonu
2) **Inicializuj** proměnnou $t = 0$
3) **Inicializuj** konstantu $eps = 0.05$
4) **Inicializuj** proměnnou $n = Vrcholy.Počet - 1$
5) **Inicializuj** proměnnou $k = 0$
6) **Dokud** je $t \leq 1$, **opakuj**:

    Dokud je $k \leq n$, **opakuj**:

    $novýBod.X = novýBod.X + VypočítejVáhu(n, k, t) * Vrcholy[k].X$

    $novýBod.Y = novýBod.Y + VypočítejVáhu(n, k, t) * Vrcholy[k].Y$

    $k = k + 1$

    $Plot(novýBod.X, novýBod.Y)$

    $t = t + eps$

7) **Konec**

$VypočítejVáhu(n, k, t)$

1) **Vypočítej** $C = KombinačníČíslo(n, k)$ kombinační číslo n nad k.
2) **Vypočítej** $TK = t^k$
3) **Vypočítej** $TNK = (1 - t)^{n-k}$
4) **Vrať** výsledek $C * TK * TNK$
5) **Konec**

$KombinačníČíslo(n, k)$

1) **Vypočítej** $fakn$ = faktoriál n
2) **Vypočítej** $fakk$ =faktoriál k
3) **Vypočítej** $faknk$ =faktoriál $n - k$
4) **Vrať** výsledek $\frac{fakn}{fakk * faknk}$
5) **Konec**

3.2 Stručný úvod do SVG

V následujícím textu je stručně představen formát SVG ve smyslu jeho historie a motivace vzniku, koncepce, struktury dokumentu a značek relevantních pro praktickou část této bakalářské práce. Z poměrně širokého portfolia elementů a jejich atributů, kterými SVG disponuje jsou vybrány ty, které se obvykle vyskytují v SVG dokumentu vzniklém např. exportem vektorového obrazu z některého grafického editoru jako je např. Adobe Illustrator®, Corel DRAW®, Inkscape apod. Je vynechán popis animace elementů, stylování dokumentu, skriptování, podmíněného zpracování a mnoho dalšího. Komplexní uchopení a zpracování problematiky interpretace SVG dokumentu je netriviální a přesáhlo by rámec této práce.

3.2.1 Historie

V roce 1998 byla konsorciem W3C[19] založena pracovní skupina, jejímž cílem bylo vytvořit reprezentaci vektorové grafiky s využitím již existujícího a osvědčeného formátu XML, což by umožnilo ukládat grafickou informaci o obraze jako prostý text a současně využívat všech výhod, které formát XML poskytuje, tj. otevřenost, přenositelnost a interoperabilitu. Vzniká tak značkovací jazyk a současně velmi kompaktní formát SVG[20] - škálovatelná vektorová grafika. Zájem o tento nový způsob uložení grafických dat stabilně vzrůstá a nástroje na jeho zpracování se staly již standardní výbavou softwarových produktů většiny firem operujících ve světě grafických aplikací, ačkoli pro samotné vytvoření a úpravu souboru SVG lze vystačit s obyčejným textovým editorem běžně dostupným na libovolné platformě. Na rostoucí popularitu formátu podepřenou vysokou flexibilitou, integrací ostatních standardů W3C (DOM[21], XSL[22]), možností animace elementů i atributů zareagovali také všichni významní hráči na poli vývoje moderních webových prohlížečů a implementovali nativní podporu SVG do renderovacích jader svých produktů (Microsoft Internet Explorer, Mozilla Firefox, Google Chrome, Opera atd.). V HTML5 je možno vkládat kód SVG přímo (*inline embedding*) a využít jej alternativně např. k elementu `<canvas>` či k proprietární technologii Flash, která je pro otevřený standard SVG přímým konkurentem (Eisenberg, Feb. 2002).

---

[19] World Wide Web Consortium
[20] Scalable Vector Graphics
[21] Document Object Model
[22] Extensible Stylesheet Language

3.2.2 **Popis SVG dokumentu**

Z hlediska kompozice představuje SVG formát, který je složený z tzv. **fragmentů**. Fragmentem se rozumí libovolný (dokonce i nulový) počet grafických prvků uzavřených v párovém elementu `<svg>`. Ještě obecněji lze fragment definovat jako podstrom XML dokumentu, který začíná tagem `<svg>`. Fragment může existovat jako samostatný soubor, může být součástí jiného rodičovského XML dokumentu, případně může být vnořen do jiného fragmentu.

Před vlastním detailním popisem elementů SVG, které nesou informace o vykreslování 2D grafických primitiv, je třeba nejprve nadefinovat některé důležité značky a jejich atributy a zavést souřadnicový systém. (Eisenberg, Feb. 2002)

3.2.2.1 Viewport

Ve světě SVG je základním objektem pro výstup tzv. **Canvas**, někdy se též hovoří o plátnu. Canvas bývá také definován jako rovina neomezená v žádném směru. Tento objekt je fyzicky implementován ve Windows GDI+ API a je prostřednictvím struktury *device context* zpřístupněn programátorům[23]. Canvas je teoreticky nekonečný, proto se na něm vymezuje pouze určitá oblast, do které je směřován výstup, resp. ve které se očekává výstup SVG prohlížečů. Tato oblast určená atributy elementu `<svg>` `width` (šířka) a `height` (výška) se označuje jako **Viewport[24]**. (Eisenberg, Feb. 2002)

3.2.2.2 Souřadnicový systém

V SVG jsou rozlišovány v zásadě dva typy souřadnicových systémů. Jednak je to souřadnicový systém zavedený na Viewport (*Viewport Coordinate System*), jehož definice byla uvedena výše, tj. prostřednictvím atributů `width` a `height` elementu `<svg>`. Prohlížeč nastaví tento kartézský ortogonální souřadnicový systém dle obvyklých konvencí pro zobrazovací zařízení, tj. osa x začíná hodnotou nula v levém horním rohu obrazovky a její kladná část narůstá směrem do pravého horního rohu. Osa y taktéž začíná v levém horním rohu hodnotou nula a její kladná část roste směrem dolů do levého dolního rohu obrazovky. Tento souřadnicový systém je podobný souřadnicovému systému zavedenému na HTML element v rámcovém modelu CSS. Druhým souřadnicovým systémem je systém uživatelský

---

[23] Tato práce se zaměřuje na využití vektorové grafiky v operačním systému MS Windows

[24] Kromě elementu `<svg>` definují nový Viewport ještě tagy `<symbol>`, `<image>` a `<foreignObject>`

(*User Coordinate System*). Ten je zaveden na Canvas SVG a není-li uvedeno jinak, je identický se systémem Viewportu. Použitím atributu `viewBox` elementu `<svg>` může být tento uživatelský souřadnicový systém, který je také znám pod názvem „Aktuální souřadnicový systém" (*Current Coordinate System*), modifikován. Atributem `viewBox` tak definujeme jakési virtuální okno nebo rám, kterým se lze na dokument dívat (Eisenberg, Feb. 2002).

3.2.2.3 Základní rovinné útvary

Po objasnění významu objektu Canvas, definování Viewportu a viewBoxu v SVG dokumentu a zavedení souřadnicových soustav na těchto objektech, lze přejít k popisu elementárních předdefinovaných geometrických útvarů, které jsou v SVG k dispozici. Jsou to především čáry, složené čáry, obdélníky (včetně obdélníků se zaoblenými rohy), mnohoúhelníky, kružnice a elipsy. Dále je možno k těmto základním grafickým prvkům řadit ještě element `<text>` a z hlediska realizace arbitrárního vektorového obrazu velmi komplexní element `<path>` - cesta, jehož konstrukcemi lze nahradit všechny výše uvedené jednoduché vektorové útvary. Zpracování a interpretace elementu `<path>` je proto jedním z klíčových témat v praktické části této práce.

3.2.2.4 SVG Line - <line>

Pro nakreslení úsečky stačí specifikovat její krajní body, jak vyplývá z geometrické definice. Pokud jsou hodnoty souřadnic krajních bodů uvedeny bez identifikátoru jednotek, jedná se o tzv. souřadnice uživatelské. Hodnoty souřadnic však můžeme opatřit jednotkami definovanými ve standardu SVG 1.1

syntaxe:

```
<line x1="start-x" y1="start-y" x2="end-x" y2="end-y">
```

např.

```
<svg width = "200px" height = "200px" viewBox = "0 0 200 200">
     <line x1 = "20" y1 = "20" x2 = "200" y2 = "180" style="stroke:
black; stroke-width: 3; />
 </svg>
```

3.2.2.5 SVG Polyline - <polyline>

Pokud vzniká potřeba vykreslení série čar, lze použít v požadovaném počtu prvek `<line>`, ale výhodnější je pro tento případ aplikovat element `<polyline>`. Posloupnost hodnot atributu `points` je interpretována jako seznam dvojic určujících *x*-ovou a *y*-ovou souřadnici bodů, které budou následně propojeny. Hodnoty jsou oddělené čárkou nebo mezerou.

syntaxe:

```
<polyline points="x1,y1 x2,y2 x3,y3 … xn,yn" />
```

např.

```
<svg height="200" width="500">
  <polyline points="20,20 40,25 60,40 80,120 120,140 200,180"
  style="fill:none;stroke:black;stroke-width:3" />
</svg>
```

3.2.2.6 SVG Rectangle - <rect>

Dalším prvkem ze souboru základních geometrických útvarů v SVG je obdélník. Je určen souřadnicemi levého horního rohu, šířkou a výškou. Pokud je uveden atribut `fill`, je vnitřek obdélníku vyplněn dle hodnoty tohoto atributu (barva, barevný přechod, vzor) a případně modifikován způsobem, který určují atributy `fill-rule` a `fill-opacity`. Pokud atribut `fill` uveden není, je vnitřní plocha ohraničená obdélníkem vyplněna černou barvou. Dále lze volitelně nastavit poloměry r_x a r_y zakřivení rohů obdélníka. Maximální hodnota poloměru r_x je polovina šířky obdélníka a maximální hodnota pro poloměr r_y je polovina jeho výšky. Je-li uveden pouze jeden z atributů, předpokládá se, že hodnota druhého je ekvivalentní.

syntaxe:

```
<rect x="left-x" y="top-y" width="width" height="height" [ rx="corner-x-radius" | ry="corner-y-radius" ] />
```

např.

```
<svg width="400" height="180">
  <rect x="50" y="20" rx="20" ry="20" width="150" height="150"
  style="fill:red;stroke:black;stroke-width:5;opacity:0.5" />
</svg>
```

3.2.2.7 SVG Circle - <circle>

Kružnice je definována jako množina všech bodů, které mají od jednoho pevně daného bodu (střed) stejnou vzdálenost. V SVG je kružnice určena souřadnicemi středu a poloměrem. Stejně jako u obdélníku platí, že není-li uvedeno jinak, je implicitní výplň kružnice černá. Záporná hodnota poloměru způsobí chybu při renderování obrazu. Je možné vynechat souřadnice středu kružnice. Pak se předpokládá, že jsou obě hodnoty nulové.

syntaxe:

```
<circle [ cx="center-x" cy="center-y" ] r="radius" />
```

např.

```
<svg height="100" width="100">
  <circle cx="50" cy="50" r="40" stroke="black" stroke-width="3"
fill="red" />
</svg>
```

3.2.2.8 SVG Ellipse - <ellipse>

Dle definice je elipsa množina všech bodů, které mají od dvou pevně zvolených bodů (ohnisek), konstantní součet vzdáleností. Z pohledu SVG je však elipsa uzavřená křivka v rovině s daným středem a horizontálním a vertikálním poloměrem. Speciálním případem elipsy s ekvivalentními hodnotami poloměrů je kružnice. Pro výplň elipsy platí totéž co pro kružnici a obdélník.

syntaxe:

```
<ellipse cx="center-x" cy="center-y" rx="x-radius" ry="y-radius" />
```

např.

```
<svg height="140" width="500">
  <ellipse cx="200" cy="80" rx="100" ry="50"
  style="fill:yellow;stroke:purple;stroke-width:2" />
</svg>
```

3.2.2.9 SVG Polygon - <polygon>

Mnohoúhelník, tedy prvek `<polygon>` se používá pro tvorbu uzavřených planárních objektů tvořených z lomených čar. Nejjednodušším typem polygonu je trojúhelník. Analogicky, jako je tomu u lomených čar, má element `<polygon>` atribut `points`, který specifikuje posloupnost hodnot, které pak v páru představují souřadnice vrcholů mnohoúhelníku. Z toho plyne, že nutným předpokladem je, aby byl počet hodnot atributu `points` sudý. Jak je to v SVG obvyklé, mohou být hodnoty odděleny čárkou nebo mezerou.

Vyplňování ploch ohraničených mnohoúhelníkem může být triviální a jednoznačné, pokud se čáry, které tvoří hranici mnohoúhelníku navzájem neprotínají. Pokud ano, můžeme strategii vyplnění řídit vlastností `fill-rule` atributu `style`.

syntaxe:

`<polygon points="`x_1,y_1 x_2,y_2 x_3,y_3 …x_n,y_n`" />`

např.

```
<svg height="210" width="500">
  <polygon points="100,10 40,198 190,78 10,78 160,198"
  style="fill:lime;stroke:purple;stroke-width:5;fill-rule:nonzero;" />
</svg>
```

3.2.2.10SVG Path - <path>

Všechny předcházející základní rovinné útvary lze považovat za zjednodušené formy obecnějšího a velmi komplexního elementu `<path>`. Pomocí tohoto prvku je možné nakreslit zcela libovolné tvary propojením čar, oblouků a křivek. Vzhled cesty může být modifikován užitím atributu `style` stejně jako prvky ostatních základních útvarů. Vymezuje-li cesta uzavřenou oblast, je možné ji vyplnit jedním z již uvedených způsobů vyplňování. Veškerá fundamentální data popisující vektorový obraz jsou v rámci elementu `<path>` soustředěna do atributu `d` (d - data). Data jsou reprezentována sérií jednopísmenných příkazů následovaných odpovídajícím počtem argumentů - hodnot souřadnic, které příkaz zpracovává (Eisenberg, Feb. 2002). Následující tabulka uvádí přehled příkazů atributu `d` v elementu `<path>` a jejich stručný popis. Některé z nich budou detailně probrány v praktické části této práce. Velikost písmene – identifikátoru příkazu určuje, zda bude výstup příkazu realizován vzhledem k absolutním souřadnicím (velké písmeno), tj. k počátku aktuálního souřadnicového systému nebo k souřadnicím relativním (malé písmeno), v tomto případě chápáno vzhledem k aktuálním souřadnicím pomyslného pera.

Příkazy atributu `d` elementu `<path>` (SVG 1.1 Second Edition)		
Příkaz	**Argumenty**	**Výstup**
M **m**	*x y*	Interpretujeme jako *moveto*. Přesune virtuální pero na souřadnice *x y*.
L **l**	*x y*	Interpretujeme jako *lineto*. Nakreslí čáru s počátkem v aktuální pozici pera do bodu o souřadnicích *x y*. Pokud uvedeme sérii souřadnic, dosáhneme efektu, jaký má prvek `<polyline>`.

H **h**	*x*	Vykreslí vodorovnou čáru z aktuální pozice např. (x_1, y_1) do koncového bodu o souřadnicích $(x_1 + x, y_1)$
V **v**	*y*	Vykreslí svislou čáru z aktuální pozice např. (x_1, y_1) do koncového bodu o souřadnicích $(x_1, y_1 + y)$
A **a**	*rx ry* *x-axis-rotation* *large-arc sweep x y*	Kreslí eliptický oblouk z aktuální pozice do bodu (x, y). Body oblouku leží na elipse s poloměry *rx* a *ry*. Elipsa je otočena o *x-axis-rotation* stupňů. Jestliže má oblouk méně než 180 stupňů, je hodnota *large-arc* nulová. V opačném případě 1. Je-li oblouk kreslen v kladném smyslu (proti směru hodinových ručiček), je hodnota *sweep* rovna 1, jinak je rovna 0.
Q **q**	*x1 y1 x y*	Kreslí kvadratickou Bézierovu křivku z aktuálního bodu do bodu (*x*,*y*) s využitím bodu (x_1,y_1) jako bodu řídícího.
T **t**	*x y*	Kreslí kvadratickou Bézierovu křivku z aktuálního bodu do bodu (*x*,*y*). Řídící bod bude bod středově souměrný s řídícím bodem předchozího příkazu Q (resp. q). Pokud předcházejícím příkazem nebyl příkaz Q (q), pak je řídícím bodem aktuální bod.
C **c**	*x1 y1 x2 y2 x y*	Kreslí kubickou Bézierovu křivku z aktuálního bodu do bodu (*x*,*y*) s využitím bodu (x_1,y_1) jako řídícího bodu počátku křivky a bodu (x_2,y_2) jako řídícího bodu konce křivky.
S **s**	*x2 y2 x y*	Kreslí Bézierovu křivku z aktuálního bodu do bodu (*x*,*y*) s využitím bodu (x_2,y_2) jako řídícího bodu konce křivky. První řídící bod bude bod středově souměrný s řídícím bodem konce křivky předcházejícího příkazu C (resp. c). Pokud předchozím příkazem nebyl příkaz C (c), pak je prvním řídícím bodem aktuální bod.

Tabulka 4 - Příkazy atributu d v elementu <path>

Zdroj: (Eisenberg, Feb. 2002 p. 92)

3.3 Vlastní práce

3.3.1 Jednoduchý prohlížeč SVG souborů - SvgViewer

Předmětem této a následujících kapitol bude představení výsledku vlastní tvůrčí činnosti, tj. jednoduchého prohlížeče vektorové grafiky uložené v souboru formátu SVG. Motivací pro naprogramování této jednoduché aplikace byla snaha o porozumění obsahu některých elementů v souboru SVG, speciálně hodnotám atributu `d` prvku `<path>`. Tyto hodnoty jsou v případě vektorových obrázků exportovaných do SVG z grafických editorů (Adobe Illustrator®, CorelDraw, Inkscape apod.) představovány značným počtem číselných údajů – souřadnic a poměrně úzkou množinou jednopísmenných příkazů, které hodnoty souřadnic přijímají na svůj vstup jako parametry. Po bližším prozkoumání struktury dat v atributu `d`, a pochopitelně s odpovídajícím teoretickým aparátem o syntaxi standardu škálovatelné vektorové grafiky a též nutným minimem znalostí z počítačové geometrie, bylo zjevné, že nebude větší problém převést takto kódované grafické objekty do „řeči“ metod z grafických tříd, kterými .NET disponuje. Jak již bylo řečeno v úvodu této práce, byl pro vývoj aplikace zvolen jazyk C# integrovaný do vývojového prostředí Microsoft Visual Studio 2015.

Celá práce lze logicky rozdělit do následujících čtyř oblastí:

- Jádro aplikace – třída *SvgViewer*
- Vlastní implementace rasterizačních algoritmů – třída *Curves*
- Grafické uživatelské rozhraní
- Nástroje pro ladění a měření výkonu

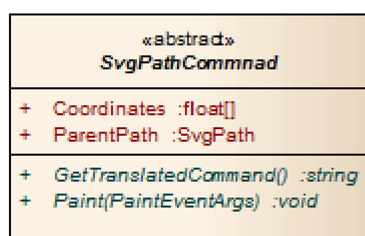
3.3.1.1 Jádro aplikace – třída *SvgViewer*

Pro zvýšení univerzálnosti je aplikace koncipována jako vrstevnatá. Nad výkonným jádrem, které je tvořeno třídou *SvgViewer* odvozenou (zděděnou) od třídy *ScrollableControl* z jmenného prostoru *System.Windows.Forms* je vrstva grafického uživatelského rozhraní, jehož prostřednictvím uživatel modifikuje vlastnosti komponenty (předává jí cestu a název SVG souboru, odkazy na rasterizační algoritmy v případě, kdy nejsou použity standardní metody třídy *Graphics*), volá metody pro vykreslování, čte hodnoty stavových proměnných (časové údaje o době triviální lexikální a syntaktické analýzy a vykreslování, údaje o celkovému počtu elementů `<path>` a počtu příkazů v nich obsažených) atd.

Ačkoli v sobě třída *SvgViewer*, která z hlediska integrace do vývojového prostředí reprezentuje vizuální komponentu, zahrnuje veškerou funkcionalitu, využívá též sadu datových struktur, tříd a metod, které zajišťují komplexní zpracování dat atributu `d`. Výsledkem je kompaktní kolekce transformovaných příkazů a souřadnic do podoby přímo použitelných metod třídy *Graphics* z jmenného prostoru *System.Drawing* nebo metod vlastní třídy *Curves*.

Abstraktní třída *SvgPathCommand*

Základní myšlenkou celého procesu zpracování dat obsažených v atributu `d` je nejprve obecně definovat strukturu prvku, který je součástí nějaké cesty ve smyslu cesty určené elementem `<path>`. V terminologii SVG se takový prvek označuje jako *path command* (Eisenberg, Feb. 2002). Pokud bude takový objekt modelován a následně implementován v jazyce C#, je zřejmé, že bude mít nějaké souřadnice a bude jistě součástí nějakého segmentu cesty. Abstraktní třída, kterou je objekt popsán bude ještě obsahovat klíčovou abstraktní metodu `Paint` a abstraktní metodu `GetTranslatedCommand`. Metoda `Paint` pak v potomkovi třídy *SvgPathCommand* implementuje konkrétní grafické primitivum, které bude nakresleno na Canvas komponenty *SvgViewer*, např. úsečku, Bézierovu křivku apod. Obdobně metoda `GetTranslatedCommand` v potomkovi implementuje textový popis použité metody třídy *Graphics* nebo *Curves*, a to včetně transformovaných souřadnic. Tento první krok je graficky i v kódu C# vyjádřen následujícím způsobem:



Obrázek 12 - SvgPathCommand

```
public abstract class SvgPathCommand
{
 public float[] Coordinates;
 public SvgPath ParentPath;
 public abstract void Paint(PaintEventArgs g);
 public abstract string GetTranslatedCommand();
}
```

Třída *SvgPath*

Ještě dříve než bude přistoupeno k popisu potomků abstraktní třídy *SvgPathCommand*, je třeba deklarovat významnou třídu *SvgPath*. Ta je zde v roli datového typu, jak je zřejmé z datové položky[25] *ParentPath* třídy *SvgPathCommand*. Dále má pak

[25] Pro termín datová položka či datový člen se běžněji užívá angl. označení *Field*.

agregační vazbu na třídu *SvgViewer*, která vytváří kolekci instancí této třídy, jež představuje množinu všech transformovaných elementů `<path>`, které se v souboru SVG vyskytují.

Třída *SvgPath*	
Vlastnosti	
`EnableFill`	Stavová proměnná, které indikuje, zda bude cesta vyplněna. Je nastavena na hodnotu `true`, pokud je uveden atribut `fill` elementu `path` a jeho hodnota je různá od `none`.
`Fill`	Hodnota atributu `fill`. Pokud to není barva nebo je to barva zapsaná v jiném formátu než definuje HTML specifikace, je vyvolána výjimka.
`PathId`	Hodnota atributu `id` elementu `path`.
`Stroke`	Hodnota atributu `stroke`. V SVG určuje barvu tahu. Platí totéž co u vlastnosti `Fill`. Neúspěšná konverze vyvolá výjimku: `"Nastala výjimka při konverzi řetězce na barvu"`. Barva je pak nastavena na hodnotu *#000000*.
`StrokeWidth`	Šířka tahu, resp. tloušťka čáry. Je to hodnota atributu `stroke-width`. Je obvykle udávána jako desetinné číslo. Pokud selže konverze, systém vyvolá výjimku: `"Nastala výjimka při konverzi řetězce na desetinné číslo. "`
Datové položky	
`graphicsPath`	Instance třídy *GraphicsPath*, která slouží pro alternativní vytváření cesty pomocí metod `AddLine`, `AddBezier` apod.
`pathCommandList`	Seznam instancí tříd zděděných z nadtřídy *SvgPathCommand*, které reprezentují transformované příkazy SVG tvořící cestu.
Metody	
`SvgPath`	Konstruktor, ve kterém je inicializována kolekce *pathCommandList* a datová položka *graphicsPath*.

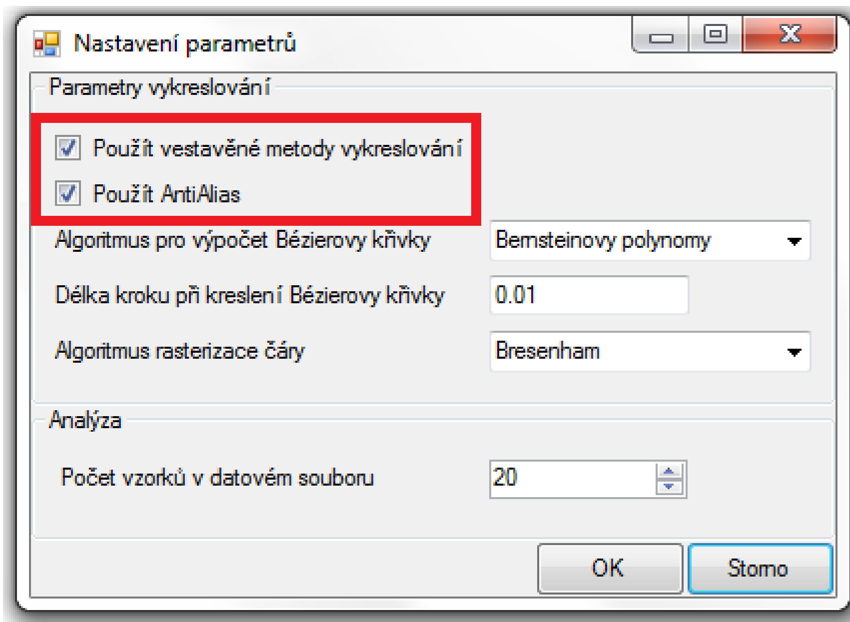
Tabulka 5 - Vlastnosti a metody třídy SvgPath
Zdroj: Vlastní tvorba

Třídy *SvgCommandLine* a *SvgCommandBezier*

Obě třídy jsou potomky abstraktní třídy *SvgPathCommand*. V těchto třídách jsou „přepsány“ metody `Paint` a `GetTranslatedCommand`.

Třída *SvgCommandLine* řeší v metodě `Paint` vykreslení úsečky, která je určená souřadnicemi počátečního a koncového bodu, barvou a tloušťkou čáry a v závislosti na nastavení proměnné `UseBuiltInDrawingMethods` ze třídy *SvgViewerGlobalSettings* rozhoduje, zda bude vykreslování realizováno „vestavěnou“ metodou `DrawLine` třídy *Graphics* nebo jedním ze dvou algoritmů pro rasterizaci úsečky ze třídy *Curves*. V druhém případě se sada parametrů volání metody rozšiřuje ještě o typ algoritmu. Jak je vidět z kódu,

v přepsané metodě `Paint` se uplatňuje ještě proměnná `ApplyAntiAlias` ve funkci přepínače, který povoluje či zakazuje použití techniky anti-alias ze jmenného prostoru *System.Drawing.Drawing2D*. Tyto dvě proměnné jsou zvýrazněny červeně.

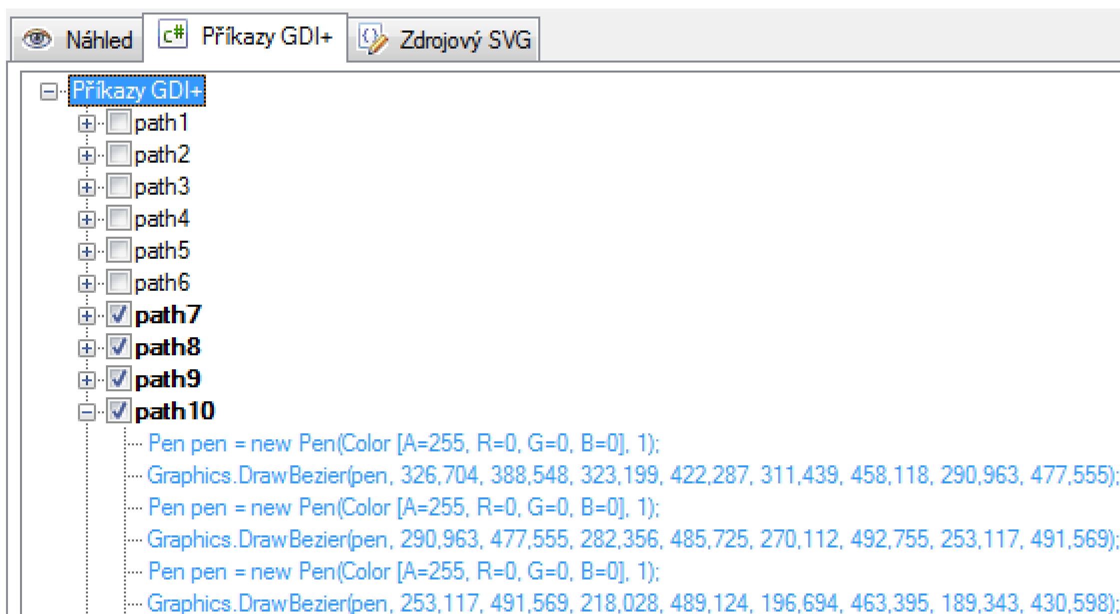


Obrázek 13 - Nastavení parametrů vykreslování

```
public override void Paint(PaintEventArgs g)
{
  Pen pen = new Pen(ParentPath.Stroke, ParentPath.StrokeWidth);
  if (SvgViewerGlobalSettings.ApplyAntiAlias)
     g.Graphics.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
  if (SvgViewerGlobalSettings.UseBuiltInDrawingMethods)
     g.Graphics.DrawLine(pen, this.Coordinates[0], this.Coordinates[1],
       this.Coordinates[2], this.Coordinates[3]);
  else
    Curves.Line(g.Graphics, pen, (int)this.Coordinates[0],
      (int)this.Coordinates[1], (int)this.Coordinates[2],
      (int)this.Coordinates[3], SvgViewerGlobalSettings.LineAlgorithm);
}
```

Metoda `GetTranslatedCommand` je spíše diagnostická a na vlastní proces vykreslování nemá žádný vliv. Vrací řetězec, jehož obsahem je popis volání zvolené grafické metody. Je to vlastně „přeložený“ příkaz elementu `<path>` souboru SVG, a to včetně transformovaných souřadnic. Podává informaci o tom, kterou metodu a s jakými souřadnicemi použil mechanismus vykreslování třídy *SvgViewer*. Úplný seznam těchto elementárních vykreslovacích zpráv pak podává obraz o celkovém počtu tagů `<path>` v souboru a o celkovém počtu příkazů v nich použitých. V grafickém uživatelském rozhraní je pak možné tyto zprávy zobrazit ve stromové struktuře a dokonce podle nich individuálně zobrazovat

jednotlivé segmenty cesty, což dává poměrně přesnou představu o tom, v jakém sledu grafických operací se výsledný vektorový obraz formátu SVG vykresloval.



Obrázek 14 - Využití metody GetTranslatedCommad v GUI

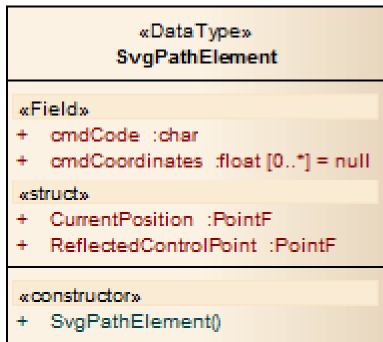
U třídy *SvgCommandBezier* je situace analogická. Její metoda `Paint` pracuje s jinou sadou souřadnic, speciálně pro tzv. Bézierovu kubiku, o níž byla řeč v teoretické partii této práce. Je to sada osmi hodnot, které specifikují počáteční kotevní bod křivky, dva řídící body a koncový kotevní bod. K vykreslení Bézierovy křivky je ve třídě *Graphics* k dispozici metoda `DrawBezier`. Ze třídy *Curves* je možnost volby ze tří standardních algoritmů – Bézierova kubika, obecná Bézierova křivka s využitím Bernsteinových polynomů a rekurzivní adaptivní algoritmus de Casteljau. Pokud se pro kreslení využijí algoritmy třídy *Curves*, je možno na formuláři „Nastavení parametrů" volit nejen algoritmus, ale i iterační krok, pokud jej algoritmus používá.

Třída *SvgPathElement*

Tato třída je podobně jako třída *SvgPath* v roli datového typu a vymezuje element cesty z pohledu souřadnic. Pracuje s aktuální pozicí pera a středově souměrného řídícího bodu pro případ tzv. hladké[26] Bézierovy křivky. Zatímco abstraktní třída *SvgPathCommand* spíše obecně předepisovala, jak bude libovolný příkaz cesty vykreslen, třída

[26] Tyto křivky bývají označovány jako *smooth* nebo *shorthand*

SvgPathElement řeší, kolik a jaké souřadnice je třeba pro již konkrétní příkaz připravit a zda je třeba uvažovat i souřadnice příkazu předcházejícího.




```
public class SvgPathElement
{
    public SvgPathElement()
    {
        cmdCoordinates = new List<float>();
    }
    public char cmdCode;
    public List<float> cmdCoordinates = null;
    public PointF CurrentPosition;
    public PointF ReflectedControlPoint;
}
```

Obrázek 15 - Třída SvgPathElement

Třída *SvgPathElementsFactory*

O třídě *SvgPathElementsFactory* lze s trochou nadsázky hovořit jako o „továrně na zpracování cest“. Tato třída totiž implementuje tři velmi důležité metody, a sice `ParsePath`, `TransformPath` a `ExtractCoordinates`. Celý proces zpracování cesty začíná přiřazením řetězce s cestou do property[27] `PathString`. Setter této property následně volá metody `ParsePath` a `TransformPath`. Výsledkem těchto operací je naplněná kolekce `PathElement`, která obsahuje instance třídy *SvgPathElement*.

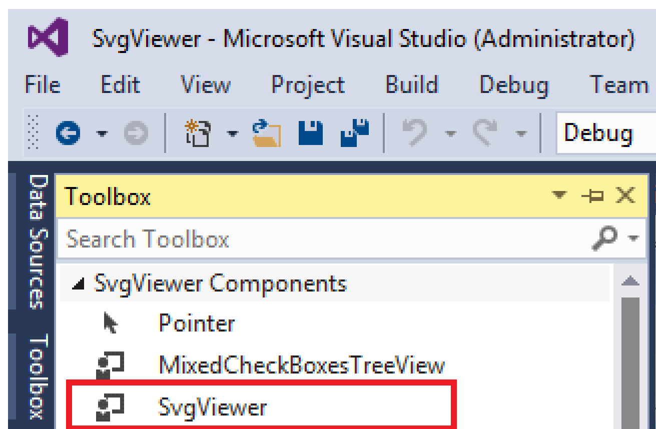
Metody třídy *SvgPathElementsFactory*	
`ParsePath`	Metoda rozdělí řetězec cesty elementu `<path>` na jednotlivé příkazy včetně jejich souřadnic. Metoda aktuálně reaguje na sedm základních příkazů, přičemž malé písmeno indikuje relativní souřadnice vzhledem k aktuální pozici pera a velké písmeno absolutní souřadnice vzhledem k počátku souřadnicového systému.
`ExtractCoordinates`	Tato metoda je volána v rámci metody `ParsePath`. Metoda „vytěží“ z řetězce souřadnice, které jsou zapsány dle specifikace SVG 1.1. Oddělovačem může být čárka, mezera nebo znaménko mínus v případě záporné relativní souřadnice.
`TransformPath`	Metoda převádí absolutní a relativní souřadnice jednotlivých příkazů v cestě na souřadnice pro vykreslovací metody ze třídy *Graphics* nebo *Curves*.

Tabulka 6 - Metody třídy SvgPathElementsFactory

[27] Property je speciální datová položka v rámci třídy. Bývá překládána jako „vlastnost“.

Třída *SvgViewer*

Tato třída reprezentuje vizuální komponentu, která je potomkem třídy *ScrollableControl*. Na následujícím obrázku je patrné, že se komponenta po přeložení a sestavení kódu skutečně umístila do Toolboxu vývojového prostředí Visual Studio, a pokud by byla např. instalována do GAC[28] mohla by být použita i v kontextu jiné aplikace než této eponymní práce.



Obrázek 16 - Umístění komponenty SvgViewer v Toolboxu

Dříve než budou popsány jednotlivé vlastnosti (properties) a metody komponenty *SvgViewer*, je pro formální úplnost uvedena hierarchie dědičnosti pro rodičovskou třídu *ScrollableControl*:

```
[ComVisibleAttribute(true)]
[ClassInterfaceAttribute(ClassInterfaceType.AutoDispatch)]
public class ScrollableControl : Control, IComponent, IDisposable
```

Třída *SvgViewer* v sobě integruje nejen mechanismy pro vykreslování objektů připravených třídou *SvgPathElementsFactory*, ale umožňuje také zaznamenávat údaje časové náročnosti algoritmů při vykreslování vektorového obrazu, poskytuje informace o počtu zpracovaných elementů `<path>` a celkovém počtu příkazů v nich obsažených, umožňuje konvertovat a ukládat vykreslený obraz do řady běžných rastrových formátů, nabízí možnost tisku a náhledu před tiskem.

Co se týče vykreslování, to je realizováno metodou `DrawSvg`. Tato metoda je přetížená, takže jejím vstupním parametrem může být buď název souboru ve formátu SVG nebo obecně

[28] Global Assemby Cache – speciální adresář, kde se shromažďují assemblies.

jakýkoli stream[29]. Jak bude ukázáno dále, otevření souboru zajistí instance třídy *FileStream* a pokud nedojde k vyvolání výjimky, předá se tento stream přetížené metodě, která z něj čte data a zpracovává je.

```
public void DrawSvg(string fileName)
public void DrawSvg(Stream input)
```

Varianta metody `DrawSvg` s parametrem streamu na vstupu zajišťuje průchod strukturou souboru SVG, vyhledání potřebných prvků a jejich atributů a na základě zpracování dat metodami třídy *SvgPathElementsFactory* vytváří seznam cest složený z jednotlivých segmentů – instancí třídy *SvgPath*. Současně s lexikálním rozkladem souboru probíhá měření času tohoto zpracování a jsou inkrementována příslušná počítadla podle počtu příkazů zařazených do cesty a podle počtu cest zařazených do seznamu cest. Následující část kódu ukazuje, jak je rozpoznán příkaz (*path command*) C,c pro vykreslení kubické Bézierovy křivky. Zcela obdobně jsou dekódovány ostatní příkazy cesty. Po ukončení zpracování všech nalezených příkazů v cestách je ukončeno měření času lexikální analýzy a zpracování dat a výsledek v milisekundách je dostupný v property `ParsingTime`.

```
string dAttr = reader.GetAttribute("d");
SvgPathElementsFactory factory = new SvgPathElementsFactory();
factory.PathString = dAttr;
foreach (SvgPathElement command in factory.PathElement)
  {
     switch (command.cmdCode)
        {
           case 'c':
           case 'C':
           SvgCommandBezier cmdCubicBezier = new SvgCommandBezier();
           cmdCubicBezier.Coordinates = new float[8];
           cmdCubicBezier.Coordinates[0] = command.CurrentPosition.X;
           cmdCubicBezier.Coordinates[1] = command.CurrentPosition.Y;
           cmdCubicBezier.Coordinates[2] = command.cmdCoordinates[0];
           cmdCubicBezier.Coordinates[3] = command.cmdCoordinates[1];
           cmdCubicBezier.Coordinates[4] = command.cmdCoordinates[2];
           cmdCubicBezier.Coordinates[5] = command.cmdCoordinates[3];
           cmdCubicBezier.Coordinates[6] = command.cmdCoordinates[4];
           cmdCubicBezier.Coordinates[7] = command.cmdCoordinates[5];
           cmdCubicBezier.ParentPath = path;
           path.pathCommandList.Add(cmdCubicBezier);
           _totalSVGCommands++;
           break;
…
```

[29] Stream je abstrakce pro práci s posloupností bytů za použití standardizovaných metod. Můžeme se setkat i s označením datový proud.

Naprosto klíčovým okamžikem celého procesu sběru, dekódování a transformace dat se pak stává volání metody `Invalidate` třídy *SvgViewer*. Ta je zděděná z předka, tj. třídy *ScrollableControl* a vynutí překreslení Canvasu komponenty spuštěním události[30] `OnPaint`. V těle této eventy je implementován mechanismus měření času vykreslování, ale především dojde k volání metody `DoPaint(PaintEventArgs pea)`. Tato metoda prochází všechny cesty v seznamu cest a v rámci těchto cest všechny příkazy. U nich pak volá metodu `Paint`, která nakreslí požadovaný objekt v závislosti na konkrétní implementaci této metody. Následující zjednodušený kód celou situaci zřetelně objasňuje.

```
private void DoPaint(PaintEventArgs pea)
{
  foreach (SvgPath pl in svgPathList)
   {
     foreach (var pc in pl.pathCommandList)
      {
         pc.Paint(pea);
      }
    }
}
```

Stručný popis datových položek (fields), vlastností (properties) a metod třídy *SvgViewer* je pak přehledně shrnut do tabulkové formy.

Vlastnosti a metody třídy *SvgViewer*	
Vlastnosti	
`GDIPlus`	Pokud je property `SuppressGDIOutput` nastavena na hodnotu `false`, lze z této property získat řetězec, který reprezentuje příkazy v cestě ve formě metod tříd *Graphics* nebo *Curves*. Getter této property volá metodu `TranslatedCommandList`.
`SuppressGDIOutput`	Potlačuje nebo povoluje výstup transformovaných příkazu cesty do řetězce, který je pak dostupný v property `GDIPlus`.
`ParsingTime`	Doba trvání lexikální analýzy SVG souboru a zpracování dat v atributech elementu path. Čas je v milisekundách.
`DrawingTime`	Pokud jsou v metodě `SetStyle` nastaveny parametry `DoubleBuffer` a `AllPaintingInWmPaint` na `true`, potom je obsahem této property čas v milisekundách, po který se obraz vykresluje do paměti. V opačném případě je to časová režie vykreslení na obrazovku.
`TotalSVGPaths`	Určuje celkový počet zpracovaných cest v souboru formátu SVG.

[30] Běžnější označení pro událost je angl. termín Event.

`TotalSVGCommands`	Celkový počet příkazů ve všech cestách, které byly nalezeny a zpracovány v souboru SVG.
Metody	
`ActionTime`	Pomocná metoda, která měří čas vykreslení. Před vlastním spuštěním stopek (metody `StartNew` a `Stop` třídy *StopWatch*) zajistí „úklid" všech tří generací garbage collectoru a nastaví procesu vykreslování nejvyšší prioritu.
`Clear`	Metoda vyprázdní kolekce *PathCommandList* a *SvgPathList* a překreslí Canvas komponenty. Efekt metody `Clear` je smazání aktuálně vykresleného obrazu na Canvasu.
`DoPaint`	Metoda volaná ze spuštěné eventy `OnPaint` po zavolání metody `Invalidate`. Prochází kolekce *PathList* a v nich obsažené kolekce *PathCommandList* a volá konkrétní implementaci metody `Paint` pro každý objekt v seznamu *PathCommandList*.
`DrawSvg`	Vlastní metoda vykreslování popsaná výše. Prochází vstupní SVG soubor (nebo stream) a v součinnosti s metodami třídy *SvgPathElementsFactory* plní kolekce *PathCommandList* a *PathList*, které jsou poté zpracovány metodou `DoPaint` po volání metody `Invalidate`.
`Print`	V závislosti na hodnotě vstupního parametru zajišťuje náhled před tiskem nebo samotný tisk. Oba přístupy však pracují na stejném principu, a to na překreslení vektorového obrazu do bitmapy, která je pak předaná k dalšímu zpracování události `PrintPage` třídy *PrintDocument*.
`SaveToFile`	Umožňuje uložit vykreslený vektorový obrázek do souboru jednoho z osmi běžných rastrových formátů. K dispozici jsou formáty BMP, EMF, GIF, Icon, Jpeg, WMF, PNG, TIFF.
`SaveToStream`	Je zobecněním ukládání do souboru. Používá ji metoda *SaveToFile*, která manipuluje se souborem prostřednictvím třídy *FileStream*. Ukládat však můžeme např. do *MemoryStream*, *StringStream* apod.
`TranslatedCommandList`	Metoda, kterou volá getter property GDIPlus. Prochází kolekce *PathList* a *PathCommandList* stejně jako metoda `DoPaint` a skládá do řetězce výstupy volání metody `GetTranslatedCommand`. Tato metoda poskytuje dekódovaný příkaz cesty z atributu `d` elementu `path` do metod třídy *Graphics* či *Curves* včetně transformovaných souřadnic.

Tabulka 7 - Vlastnosti a metody třídy SvgViewer

3.3.1.2 Třída *Curves*

V úvodu této kapitoly je nutné předeslat, že navzdory značnému rozsahu a bohatosti celého jmenného prostoru *System.Drawing* v něm není implementována metoda na vykreslení bodu. Jednou z možností je volat metody `PutPixel` a `GetPixel` přímo z knihovny *gdi.dll* nebo aproximovat bod elipsou či úsečkou velmi malé délky. V následujících příkladech je zvolena druhá varianta. Bod je úsečka s délkou 0.5 pixelu. Další simplifikací řešené problematiky je předpoklad, že všechny vykreslované grafické objekty nezohledňují tloušťku čáry. Přestože je atribut stroke-width elementu `<path>` rozpoznán a jeho hodnota uložena, všechny čáry a křivky jsou kresleny jako vlasové (hair line), tj. s šířkou tahu 1. Ačkoli je třída pojmenovaná *Curves*, tedy křivky, zdaleka neobsahuje jen metody vykreslování křivek (i když jisté extrémní případy křivek mohou mít podobu úsečky, např. Bézierova křivka prvního stupně, tj. bez řídícího bodu nebo se všemi řídícími body na jedné přímce viz.pozn. 17). Třída *Curves* je statická, tj. není třeba ji před použitím instanciovat a obsahuje kromě metod vykreslovacích také několik metod ryze numerických. Pouze dvě metody ze statické třídy *Curves* jsou veřejné, a to metody `Line` a `Bezier`. Ostatní metody jsou privátní. Tyto veřejné metody jsou v závislosti na nastavení globální proměnné `UseBuiltInDrawingMethods` volány z metody `Paint` konkrétní implementace příkazu cesty. Následující text je orientován na implementaci algoritmů popsaných v teoretické části této práce, a to v jazyce C#. Budou to algoritmy na rasterizaci úsečky (DDA, Bresenham) a Bézierovy křivky (Bézierova kubika, obecná Bézierova křivka s využitím výpočtu Bernsteinových polynomů a výpočet bodů Bézierovy křivky rekurzivním algoritmem de Casteljau). V dále uvedených ukázkách implementací nebude bez ztráty na obecnosti v některých případech zdrojový kód kompletní, ale bude redukován např. na řešení v jednom kvadrantu, resp. oktantu. Řešení v ostatních oktantech bývá většinou velmi podobné. Úplný zdrojový kód v jazyce C# je přístupný v příloze této práce.

Implementace algoritmu DDA (Digital Differential Analyzer)

Princip tohoto algoritmu je popsán v teoretické části. Podle hodnoty směrnice úsečky se rozhodne, která osa je osou řídící. Ve směru této osy je pak délka kroku vzorkování rovna jedné. Na vedlejší ose se vzorkuje s přírůstkem rovným směrnici úsečky (v případě řídící osy x) nebo s reciprokou hodnotou směrnice (je-li řídící osou y). Uvedený příklad je pro úsečku, jejíž směrnice $k \neq 0$.

```
private static void LineDDA(Graphics g, Pen p, int x1, int y1, int x2, int y2)
{
   double deltaX = x2 - x1;
   double deltaY = y2 - y1;
   double k = deltaY / deltaX;
   if (Math.Abs(k) <= 1) // Ridici osa je osa x - k je v intervalu <-1;1>
      {
         if (x2 < x1)
            {
                Swap<int>(ref x1, ref x2);
                Swap<int>(ref y1, ref y2);
            }
          double y = (double)y1;
          for (int x = x1; x <= x2; x++)
             {
                int roundedY = (int)Math.Round(y);
                Plot(g, p, x, roundedY);
                y += k;
              }
       }
       else // Ridici osa je osa y
         {
            if (y2 < y1)
              {
                 Swap<int>(ref x1, ref x2);
                 Swap<int>(ref y1, ref y2);
              }
              double x = (double)x1;
              for (int y = y1; y <= y2; y++)
                {
                  int roundedX = (int)Math.Round(x);
                  Plot(g, p, roundedX, y);
                  x += (1 / k);
                }
       }
}
```

Implementace Bresenhamova algoritmu

Popis algoritmu včetně základních odvození je opět uveden v teoretickém partu práce. Pouze pro připomenutí lze uvést, že algoritmus je založen na iteračním výpočtu tzv. prediktoru (někdy se též hovoří o rozhodovací proměnné – decision variable), jehož znaménko rozhoduje, která ze dvou možných variant souřadnic bodů v okolí originální úsečky bude zvolena. Algoritmus by měl být ze všech zmiňovaných algoritmů pro rasterizaci úsečky nejrychlejší, neboť pracuje pouze s celočíselnou aritmetikou. Níže uvedená ukázka implementace se omezuje pouze na první oktant souřadnicového systému, tj. pro směrnici úsečky v intervalu $k = (0; 1)$.

```
public static void LineBresenham(Graphics g, Pen p, int x1, int y1, int x2, int y2)
 {
    int deltaX = Math.Abs(x2 - x1);
    int deltaY = Math.Abs(y2 - y1);
    int k1, k2, predictor;
```

```
    if (deltaX > deltaY) // Ridici osa je osa x
      {
         k1 = 2 * deltaY;
         k2 = 2 * (deltaY - deltaX);
         predictor = k1 - deltaX;
         int incY = 1;
         if (x2 < x1)
           {
               Swap<int>(ref x1, ref x2);
               Swap<int>(ref y1, ref y2);
            }
          if (y2 < y1) incY = -1;
          Plot(g, p, x1, y1);
          int y = y1;
          for (int x = x1; x <= x2; x++)
           {
             if (predictor <= 0) predictor += k1;
               else  { predictor += k2; y += incY; }
             Plot(g, p, x, y);
            }
        }
}
```

Implementace Bézierovy kubiky

Bézierova kubika je nejčastěji používanou variantou Bézierovy křivky. Pro připomenutí je nejprve uveden obecný tvar Bézierovy aproximační křivky:

$$Q(t) = \sum_{k=0}^{n} P_i B_k^n(t)$$

kde B_k^n jsou Bernsteinovy polynomy n-tého stupně.

$$B_k^n = \binom{n}{k} t^i (1-t)^{n-k}; t \in \langle 0,1 \rangle; k = 0,1 \dots n$$

Pro kubiku, tedy křivku třetího stupně, mají Bernsteinovy polynomy následující tvar:

$$w_0 = B_0^3 = \binom{3}{0} t^0 (1-t)^3 = (1-t)^3$$

$$w_1 = B_1^3 = \binom{3}{1} t^1 (1-t)^2 = 3t(1-t)^2$$

$$w_2 = B_2^3 = \binom{3}{2} t^2 (1-t)^1 = 3t^2(1-t)$$

$$w_3 = B_3^3 = \binom{3}{3} t^3 (1-t)^0 = t^3$$

(Špička, a další, 2012)

Pro výpočet souřadnic bodu na Bézierově kubice při dané hodnotě parametru t je použita pomocná metoda `EvaluateCubic`. Hodnoty $w_0 \dots w_3$ se nazývají funkční váhy.

```
private static PointF EvaluateCubic(PointF[] vertices, double t)
{
   double w0 = (1 - t) * (1 - t) * (1 - t);
   double w1 = 3 * t * (1 - t) * (1 - t);
   double w2 = 3 * t * t * (1 - t);
   double w3 = t * t * t
   double coordX = w0 * vertices[0].X + w1 * vertices[1].X + w2 * vertices[2].X +
              w3 * vertices[3].X;
   double coordY = w0 * vertices[0].Y + w1 * vertices[1].Y + w2 * vertices[2].Y +
              w3 * vertices[3].Y;
   return new PointF((float)coordX, (float)coordY);
}
```

Vlastní vykreslení pak realizuje následující metoda.

```
private static void CubicBezier(Graphics g, Pen p, PointF[] controlPoints)
{
    double t = 0.0;
    PointF origPoint = new PointF(0f, 0f);
    PointF newPoint = new PointF(0f, 0f);
    origPoint = EvaluateCubic(controlPoints, t);
     t += SvgViewerGlobalSettings.BezierCurveLoopIncrement;
     for (; t <= 1; t += SvgViewerGlobalSettings.BezierCurveLoopIncrement)
     {
        newPoint = EvaluateCubic(controlPoints, t);
        g.DrawLine(p, origPoint.X, origPoint.Y, newPoint.X, newPoint.Y);
        origPoint = newPoint;
      }
      t = 1.0;
      newPoint = EvaluateCubic(controlPoints, t);
      g.DrawLine(p, origPoint.X, origPoint.Y, newPoint.X, newPoint.Y);
  }
```

Globální proměnná `BezierCurveLoopIncrement` ze třídy *SvgViewerGlobalSettings* je charakteristickým znakem všech algoritmů pro rasterizaci Bézierovy křivky. Určuje totiž krok, kterým se prochází interval parametru $t \in \langle 0,1 \rangle$. Ten je implicitně nastaven na hodnotu 0.01, ale uživatel jej může libovolně měnit v grafickém uživatelském rozhraní, jak bude ukázáno později. Z kódu je také zjevné, že pro zajištění spojitosti křivky jsou vypočítané body spojené úsečkou. Také zde není záměrně použita funkce `Pow` na umocňování čísel. Cílem bylo co možná nejvíce minimalizovat výpočetní složitost.

Implementace obecné Bézierovy křivky *n*-tého stupně

Předcházející případ Bézierovy křivky třetího stupně byl na implementaci velmi jednoduchý. Pro všechny čtyři řídící body byly vypočítány váhové funkce a nový bod na Bézierově křivce je pak stanoven jako součet součinů těchto vah s příslušnými *x*-ovými, resp. *y*-ovými souřadnicemi řídících bodů. Nyní je třeba celou situaci zobecnit pro libovolné hodnoty n. Pro lepší čitelnost a srozumitelnost kódu jsou napsány dvě pomocné funkce, a to

funkce `Factorial` a `CombinationNumber`. `Factorial` provádí rekurzivní výpočet faktoriálu kladného celého čísla n. Návratová hodnota, stejně tak vstupní parametr jsou datového typu `int`. Tento typ je použit s ohledem na skutečnost, že ve formátu SVG figurují Bézierovy křivky nejvýše stupně 3 (příkazy C a S). V jiných případech by byl samozřejmě vhodnější datový typ `long`. Funkce `CombinationNumber` slouží pro výpočet kombinačního čísla:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \qquad (23)$$

Funkce má dva vstupní parametry typu `int` a návratová hodnota je typu `double`. Renderování křivky je podobně jako v předchozím případě rozděleno do dvou částí. Numerická část poskytuje prostřednictvím metody `EvaluateBernstein` (a pomocných metod `Bernstein`, `CombinationNumber` a `Factorial`) souřadnice bodu na křivce při dané hodnotě parametru t. Vlastní kód metody pro vykreslení Bézierovy křivky je pak s využitím pomocných funkcí opět velmi jednoduchý.

```
private static PointF EvaluateBernstein(PointF[] vertices, double t)
{
   double coordX = 0.0;
   double coordY = 0.0;
   int n = vertices.Count() - 1;
   for (int k = 0; k <= n; k++)
   {
      coordX += Bernstein(n, k, t) * vertices[k].X;
      coordY += Bernstein(n, k, t) * vertices[k].Y;
    }
    return new PointF((float)coordX, (float)coordY);
}

private static void BernsteinBezier(Graphics g, Pen p, PointF[] controlPoints)
{
    double t = 0.0;
    PointF origPoint = new PointF(0f, 0f);
    PointF newPoint = new PointF(0f, 0f);
    origPoint = EvaluateBernstein(controlPoints, t);
    t += SvgViewerGlobalSettings.BezierCurveLoopIncrement;
    for (; t<=1; t += SvgViewerGlobalSettings.BezierCurveLoopIncrement)
    {
        newPoint = EvaluateBernstein(controlPoints, t);
        g.DrawLine(p, origPoint.X, origPoint.Y, newPoint.X, newPoint.Y);
        origPoint = newPoint;
     }
     t = 1.0;
     newPoint = EvaluateBernstein(controlPoints, t);
     g.DrawLine(p, origPoint.X, origPoint.Y, newPoint.X, newPoint.Y);
}
```

Cyklus `for` prochází interval parametru $t \in \langle 0,1 \rangle$ s krokem určeným globální proměnnou `BezierCurveLoopIncrement.` Pro zajištění hladkosti a spojitosti křivky (na tomto místě jsou hladkost a spojitost chápány jako vizuální vlastnosti křivky) jsou opět vypočítané body spojeny úsečkou. Vstupním parametrem tentokrát není osm hodnot určujících čtyři dvojice souřadnic bodů řídícího polygonu, ale pole struktur `PointF`.

Implementace Bézierovy křivky algoritmem de Casteljau

Vhodnou metodou pro výpočet bodů obecné Bézierovy křivky stupně *n* je rekurzivní algoritmus de Casteljau. V této práci je pro jednoduchost, a opět s ohledem na formát SVG, v němž je Bézierova křivka maximálně stupně $n = 3$, implementován algoritmus pro Bézierovu kubiku. Pro připomenutí je opět uveden základní vztah, podle kterého výpočet probíhá. Bod křivky o souřadnicích $Q(t)$ je vypočten z rekurentního vztahu: (Špička, a další, 2012)

$$P_{j,i}(t) = (1-t)P_{j-1,i-1} + tP_{j,i-1}$$

kde $i = 1,2 \dots n$ a $j = i, i+1 \dots n$

Pokud by bylo žádoucí vyhnout se rekurzi i za cenu ztráty na obecnosti, lze při výpočtu bodu postupovat následujícím způsobem:

1) Vstupní body řídícího polygonu Bézierovy kubiky: $P_{0,0}\ P_{1,0} P_{2,0} P_{3,0}$

2) 1. iterace:

$$Q_{0,1} = (1-t)P_{0,0} + tP_{1,0}$$
$$Q_{1,1} = (1-t)P_{1,0} + tP_{2,0}$$
$$Q_{2,1} = (1-t)P_{2,0} + tP_{3,0}$$

3) 2. iterace

$$R_{0,2} = (1-t)Q_{0,1} + tQ_{1,1}$$
$$R_{1,2} = (1-t)Q_{1,1} + tQ_{2,1}$$

4) 3. iterace

$$S_{0,3} = (1-t)R_{0,2} + tR_{1,2}$$

5) konec výpočtu – bod $S_{0,3}$ je bodem na Bézierově křivce.

Je vidět, že tento postup je poměrně zdlouhavý. O eleganci řešení nemůže být ani řeč. Proto je použita rekurze, i když „moderní“ teorie o agilním programování před použitím rekurze

varují. (Sommerville, 2013) V tomto případě je však možné ji použít bez obav z přílišné časové i paměťové náročnosti.

Co se týče vlastní implementace, ta se skládá ze dvou metod. Je to metoda `CalculateDeCasteljau`, která má na vstupu dvě celočíselné hodnoty i a j, jejichž význam je zřejmý z výše uvedeného postupu a samozřejmě z rekurentního vzorce. Dále je to parametr t datového typu `float`. Funkce rekurzivně počítá hodnotu bodu na křivce a výsledek vrací ve struktuře `PointF`, tj. jako dvojici hodnot typu `float`. O samotné vykreslení se pak postará metoda nazvaná `DeCasteljauBezier`. Ta má mimo jiné na vstupu pole struktur typu `PointF` definujících konvexní obálku kubické Bézierovy křivky. V tomto místě je záměrně potlačena obecnost řešení pro křivky libovolného stupně n. Důvody byly v této kapitole již dvakrát uvedeny, tj. SVG formát nemá ve své výbavě nástroj pro vykreslování Bézierových křivek vyššího stupně než 3. Toto „úzké hrdlo" potlačené obecnosti však může být snadno odstraněno a jednoduchou změnou signatury metody můžeme vykreslovat křivky n-tého stupně, neboť vlastní výpočetní mechanismus je na takovou situaci připraven.

Rekurzivní metoda `CalculateDeCasteljau`:

```
private static PointF CalculateDeCasteljau(int i, int j, float t)
{
   if (i == 0) return DeCasteljauInit[j];
   PointF p1 = CalculateDeCasteljau(i - 1, j, t);
   PointF p2 = CalculateDeCasteljau(i - 1, j + 1, t);

   return new PointF(((1 - t) * p1.X + t * p2.X), ((1 - t) * p1.Y + t * p2.Y));
}
```

`DeCasteljauInit` je instance kolekce `List<PointF>`, která je naplněna souřadnicemi bodů řídícího polygonu.

Kód metody pro vlastní vykreslení má následující podobu:

```
private static void DeCasteljauBezier(Graphics g, Pen p, PointF[] controlPoints)
{
   float t = 0f;
   PointF origPoint = CalculateDeCasteljau(DeCasteljauInit.Count - 1, 0, t);
   t += SvgViewerGlobalSettings.BezierCurveLoopIncrement;
   for (; t <= 1; t += SvgViewerGlobalSettings.BezierCurveLoopIncrement)
   {
       newPoint = CalculateDeCasteljau(DeCasteljauInit.Count - 1, 0, t);
       g.DrawLine(p, origPoint.X, origPoint.Y, newPoint.X, newPoint.Y);
       origPoint = newPoint;
    }
    t = 1f;
    newPoint = CalculateDeCasteljau(DeCasteljauInit.Count - 1, 0, t);
    g.DrawLine(p, origPoint.X, origPoint.Y, newPoint.X, newPoint.Y);
}
```

Kód je velmi jednoduchý, srozumitelný a efektivní. Na řádku, kde je volána rekurzivní metoda `CalculateDeCasteljau` lze hodnotu `DeCateljauInit.Count` – 1 nahradit konstantou 3. Hodnota 0 na pozici parametru j říká, že výsledek je očekáván v bodě $Q(t) = S_{0,3}$. To je dobře patrné z jednotlivých iteračních kroků, kterými rekurze prochází, tj. výpočet bodů $Q_{j,i} - S_{j,i}$, který je uveden výše.

Jak již bylo řečeno, pouze dvě metody třídy *Curves* jsou veřejné. Jsou volány z metody `Paint` ve třídách *SvgCommandLine* a *SvgCommandBezier* za předpokladu, že je nastavena globální proměnná `UseBuiltInDrawingMethods` na hodnotu `false`. Jsou to metody `Line` a `Bezier`. Ty se volají s příslušných počtem parametrů specifikujících souřadnice a s parametrem určujícím typ algoritmu, který bude pro vykreslování použit. Např. pro úsečku vypadá kód metody `Line` následujícím způsobem:

```
public static void Line(Graphics g, Pen pen, int x1, int y1, int x2, int y2,
LineRasterizationAlgorithm Algorithm)
{
   switch(Algorithm)
   {
     case LineRasterizationAlgorithm.DDA:
         LineDDA(g, pen, x1, y1, x2, y2);
         break;

      case LineRasterizationAlgorithm.Bresenham:
          LineBresenham(g, pen, x1, y1, x2, y2);
          break;

   }

 }
```

3.3.1.3 Grafické uživatelské rozhraní

Grafické uživatelské rozhraní tvoří nadstavbu komponenty *SvgViewer*. Jeho smysl je primárně v usnadnění ovládání komponenty, tj. předávání parametrů, prohlížení, tisk a konverze grafického výstupu a v neposlední řadě sledování některých stavových proměnných, např. čas vykreslování, počet příkazů v SVG souboru apod. Umožňuje však také individuální vykreslování jednotlivých cest, ze kterých je obraz složen a nabízí jednoduchý prohlížeč zdrojového souboru s možností vyhledávání řetězce znaků včetně obarvení jeho výskytů v textu. Celé uživatelské rozhraní je sestaveno ze standardních komponent dostupných ve Visual Studiu 2015. Jedinou výjimku tvoří volně šiřitelná komponenta *ZedGraph*, jejíž účel bude popsán později.

Na hlavním formuláři je umístěno pět základních vizuálních komponent:

- *MenuStrip* – hlavní roletová nabídka s položkami Soubor, Volby a Nápověda
- *ToolStrip* – obsahuje tlačítka s ikonou zastupující některé často používané funkce z hlavní nabídky
- *ListView* – Zobrazuje názvy souborů formátu SVG, které jsou ve vybrané složce adresářové struktury
- *TabControl* – hlavní kontajner celé aplikace. Na záložce **Náhled** je umístěna komponenta *SvgViewer*, záložka **Příkazy GDI+** obsahuje modifikovanou komponentu *TreeView* a *Panel*, na který jsou vykreslovány cesty z SVG souboru podle toho, zda jsou kliknutím označeny ve stromové struktuře. Poslední záložka **Zdrojový SVG** obsahuje komponentu *RichTextBox* pro zobrazení textu zdrojového souboru.
- *StatusStrip* – zobrazuje stavové informace o vybrané adresářové složce, časech parsování a vykreslení a údaje o celkovém počtu cest a příkazů v nich.

Některé parametry aplikace, jako např. pozice a rozměry formuláře, naposledy otevřená složka se soubory SVG, generování příkazů GDI+ apod. jsou při ukončení ukládány do registrů a naopak při spuštění aplikace jsou z registrů čteny. To sice aplikaci poněkud hendikepuje z hlediska možné portace na jiné operační systémy (např. projekt Mono v systémech Linuxového typu), ale na druhou stranu jistý drobný komfort to přináší. Jazyk C# navíc disponuje standardními nástroji pro práci s registry, i když na tomto místě by bylo vhodnější použít třídu *IsolatedStorageFile* a informace ukládat a načítat pomocí tohoto přístupu.

Hlavní nabídka

Hlavní nabídka aplikace představovaná lištou roletového menu nabízí pouze tři položky, Soubor, Volby a Nápověda. Položka **Soubor** obsahuje submenu o dalších šesti položkách:

- **Otevřít složku** – Otevře standardní dialogové okno systému Windows pro výběr složky ze stromové adresářové struktury (třída *FolderBrowserDialog*). Pokud uživatel potvrdí výběr nějaké složky tlačítkem OK, je zavolána metoda `ListSvgFilesInFolder`, jejímž vstupním parametrem je právě zvolená složka. Tato metoda vyhledá v zadaném adresáři (ne však v adresářích vnořených) všechny soubory s příponou *.svg a zobrazí jejich názvy v komponentě *ListView* umístěné v levé části formuláře.

- **Uložit jako…**Pokud je již komponentou *SvgViewer* zpracován nějaký soubor ve formátu SVG, je vykreslen na její Canvas, zpřístupní se položka Uložit jako…Ta otevře standardní dialogové okno pro uložení souboru ve zvoleném formátu rastrové grafiky. Je možno vybrat z osmi nejběžnějších formátů: WMF (Windows Metafile), TIFF (Tagged Image File Format), PNG (Portable Network Graphics), BMP (Bitmap), EMF (Enhanced MetaFile), GIF (Graphics Interchange Format), Icon a JPEG.
- **Zavřít** – Uplatňuje se v případě, že na plátně komponenty *SvgViewer* je již vykreslen obraz, který chceme odstranit. V takovém případě je volána metoda `Clear` komponenty *SvgViewer* a princip této metody byl vysvětlen dříve.
- **Náhled před tiskem** – Název položky přesně vystihuje její funkci. Umožňuje ověřit, jak bude vykreslený obraz vypadat po vytištění na tiskárně na papír formátu A4 s orientací Portrait, tj. na výšku. Použité standardní dialogové okno náhledu, které je třídy *PrintPreviewDialog* bohužel neumožňuje ani volbu formátu papíru ani jeho orientaci.
- **Tisk** – Podobně jako položka Náhled před tiskem volá nabídka Tisk metodu `Print` komponenty *SvgViewer*, která překreslí její Canvas do bitmapy a po nastavení parametrů tisku ve standardním tiskovém dialogu ji vytiskne na vybranou tiskárnu.
- **Konec** – uloží stav hlavního okna aplikace do registrů a pak jej zavře.

Další položkou hlavní nabídky je položka **Volby**, která též obsahuje podnabídku s následujícími prvky:

- **Generovat příkazy GDI+** - Volba této položky nevykoná žádnou akci. Má funkci přepínače, což se vizuálně projeví symbolem „zaškrtnutí“, pokud je pozitivně nastavena hodnota její vlastnosti `Checked`. Před vykreslením obrázku, tj. před voláním metody `DrawSvg` je stav tohoto přepínače testován a je dle něj nastavena vlastnost `SuppressGDIOutput`. Ta, jak již bylo uvedeno v přehledu vlastností a metod třídy *SvgViewer*, povolí nebo naopak potlačí produkci textového řetězce, jenž obsahuje dekódované a přeložené příkazy elementu `<path>` v souboru SVG včetně transformovaných souřadnic.

- **Pořídit datový soubor** – Po kliknutí na tuto položku dojde k vyvolání oznámení prostřednictvím tzv. *MessageBoxu*[31], že po vybrání souboru SVG ze seznamu *ListView* bude spuštěn sběr dat pro srovnání časové režie vykreslovacích metod ze třídy *Graphics* (v této práci mnohdy označovaných jako metody built-in) a vlastních grafických metod třídy *Curves*. Také dojde v dočasnému nastavení globální proměnné `GeneratingDataFile` ze třídy *SvgViewerGlobalSettings*.
- **Nastavení…**- Otevře dialogové okno s možností nastavení parametrů pro metody třídy *Curves* (např. volba algoritmu pro kreslení úseček a Bézierových křivek)

Poslední položkou z hlavního menu je **Nápověda**.

- **Dokumentace** – Otevře okno internetového prohlížeče, který je v systému nastaven jako výchozí a v něm zobrazí dokumentaci metod ve formátu XML, která je vygenerovaná z komentářů nástrojem SandCastle HelpFile Builder.
- **O aplikaci…**Otevře okno s informacemi o autorovi této bakalářské práce, vedoucím, informačních zdrojích použitých při vývoji apod.

TabControl

Záložka **Náhled**

Na této záložce je umístěna komponenta *SvgViewer* a zcela logicky zaujímá největší plochu hlavního okna aplikace, neboť je do ní směrován veškerý grafický výstup. Pokud by nastala situace, že zobrazený vektorový obraz přesahuje aktuální hranice Canvasu, je automaticky přidán vodorovný a svislý posuvník (ScrollBar).

Záložka **Příkazy GDI+**

Hlavním ovládacím prvkem na této záložce je upravená komponenta *TreeView*, u které je každý uzel 1.úrovně opatřen kromě názvu také prvkem pro označení – CheckBox. Tato datová struktura je naplněna za předpokladu, že je zakázáno potlačení výstupu transformovaných příkazů formátu SVG pomocí property `SuppressGDIOutput`. V takovém případě je pak nultou úrovní stromu (tedy kořenový uzel) pouze popiska „Příkazy GDI+“.

[31] Třída ze jmenného prostoru *System.Windows.Forms*. Je známá také jako tzv. Dialog Box – modální okno, které textem a ikonou informuje a instruuje uživatele v případě, že nastane některá z následujících událostí: chyba (Error), varování (Warning), oznámení (Information) a reakce na dotaz (Confirm).

Po jejím rozbalení je v první úrovni tolik uzlů, kolik bylo nalezeno a zpracováno elementů `path`. V každém tomto uzlu, tedy ve druhé úrovni stromu, je pak takový počet listů, který odpovídá počtu příkazů rodičovského uzlu zastupujícího příslušný prvek `path`. Lépe bude celá situace zřejmá z následujícího obrázku. Zaškrtnutím nebo odškrtnutím[32]uzlu první úrovně je možné zobrazit tvar příslušné cesty a současně mít informace o grafických příkazech, kterými byla realizována. Tato, na naprogramování poměrně jednoduchá, inspekce vykreslování se ukázala jako vysoce účinná a užitečná ve fázi ladění procesu transformace souřadnic ze zdrojového textu souboru SVG. Navíc činí tento prohlížeč distinktivní od softwaru podobného zaměření.

Obrázek 17 - Individuální vykreslování cest

Záložka **Zdrojový SVG**

Zde je umístěna komponenta *RichTextBox*, která představuje velmi jednoduchý prohlížeč zdrojového text souboru. Záměrně je nastavením property `ReadOnly` zakázána modifikace souboru, i když by jisté opodstatnění měla i možnost provádění změn v souboru s vazbou na eventuální překreslení upraveného zdrojového kódu. Tento jednoduchý read-

[32] Běžně se užívá angl. termínu check/uncheck, které jsou do přijatelného českého ekvivalentu vyjadřujícího danou situaci téměř nepřeložitelné.

only textový editor je doplněn o možnost vyhledávání a barevného zvýraznění nalezených výskytů textového řetězce.

Volby – Nastavení…

Formulář pro nastavení parametrů je rozdělen na dvě logické části. Parametry, které kvalitativně ovlivňují vlastní metody vykreslování a parametr určující počet vzorků při pořizování datového souboru. Skupina komponent označená titulkem „Parametry vykreslování" obsahuje pět položek, jejichž prostřednictvím uživatel mění hodnoty globálních proměnných, které jsou součástí algoritmů implementovaných ve třídě *Curves*.

- **Použít vestavěné metody vykreslování** – řídí obsah globální proměnné `UseBuiltInDrawingMethods`. Po spuštění aplikace je nastavena na hodnotu `true`, což znamená, že k vykreslování SVG souborů budou použity metody třídy *Graphics* .NET Frameworku. Pokud chce uživatel používat statické metody třídy *Curves*, je třeba zaškrtnutí (kliknutím) zrušit.
- **Použít AntiAlias** – implicitně taktéž nastavená na hodnotu `true`. Nastavuje globální proměnnou `ApplyAntiAlias`, která pak zakazuje či povoluje nastavení property `SmoothingMode` třídy *Graphics* na hodnotu `SmoothingMode.AntiAlias`. Důsledkem je pak povolení či potlačení procesu vyhlazování hran čar a křivek.
- **Algoritmus pro výpočet Bézierovy křivky** – z rozbalovací nabídky[33]může uživatel vybrat jednu ze tří metod, kterou budou vykreslovány Bézierovy kubiky v případě, že nebude uplatněna volba „Použít vestavěné metody vykreslování". Je nastavena globální proměnná `BezierAlgoritm`
- **Délka kroku při kreslení Bézierovy křivky** – hodnota tohoto ovládacího prvku je přenesena do globální proměnné `BezierCurveLoopIncrement`. Její význam byl vysvětlen v kapitole o třídě *Curves*, která obsahuje implementace vykreslovacích algoritmů. Hodnota této proměnné je po spuštění aplikace nastavena na 0.01. Její změna na zásadní vliv na rychlost a kvalitu vykreslovaných aproximačních křivek.
- **Algortimus rasterizace čáry** – je analogií k nabídce volby algoritmu pro Bézierovu křivku jen s tím rozdílem, že zde uživatel vybírá algoritmus pro kreslení úsečky. Dostupné jsou implementace dvou nejběžnějších algoritmů – DDA a Bresenhamův. Je nastavena globální proměnná `LineAlgorithm`.

[33] Tato komponenta se prakticky ve všech vývojových prostředích označuje jako ComboBox.

Skupina s označením „Analýza“ obsahuje pouze jeden prvek, a sice „Počet vzorků v datovém souboru“. Ten je implicitně nastaven na hodnotu 20, která se přenáší do globální proměnné `SamplesInDataFile`. Její význam bude vysvětlen v následující kapitole. Formulář pro nastavení parametrů byl již uveden na Obrázek 13.

StatusStrip

Posledním grafickým prvkem, kterému bude v této kapitole věnována pozornost je komponenta *StatusStrip*. Je de facto standardem pro zobrazování stavových informací podle zaměření aplikace. U textových editorů to může být např. aktuální pozice kurzoru nebo stav kláves Num Lock, Caps Lock apod., u grafických editorů počet objektů na pracovní ploše atd. V tomto případě se zde zobrazuje název aktuálně vybrané složky v adresářové struktuře, kde jsou umístěny soubory formátu SVG, časové hodnoty zpracování souboru a jeho vykreslení, dále pak informace o celkovém počtu elementů `path` a celkovém počtu zpracovaných příkazů obsažených v těchto cestách. Pro případ operace pořizování datového souboru je na *StatusStripu* ještě indikátor postupu[34] vzorkování, ale ten není za běžných okolností viditelný. Vzhled pruhu stavových informací je na následujícím obrázku.

Vybraná složka: D:\CS App\BP\SvgViewer\svg Čas parsování: 8,4284 ms Čas vykreslování: 3,9634 ms Celkem cest: 36 Celkem příkazů: 514

Obrázek 18 - Stavové informace z komponenty SvgViewer

3.3.1.4 Nástroje pro ladění a měření výkonu

Součástí grafického uživatelského rozhraní aplikace *SvgViewer* je mechanismus vytvoření datového souboru, jehož prvky jsou dvojice hodnot udávajících časovou složitost grafických metod .NET Frameworku a metod implementovaných ve třídě *Curves*. Soubor v tomto kontextu je chápán ve statistickém slova smyslu, nikoli jako soubor fyzicky uložený na disku či jiném médiu. Procedura vytvoření souboru je poměrně jednoduchá. V požadovaném počtu opakování, který je dán globální proměnnou `SamplesIndataFile`, o níž již byla v předchozím textu zmínka, volá metodu `DrawSvg` s tím, že je před jejím voláním střídavě nastavována globální proměnná `UseBuiltInDrawingMethods` na hodnotu `true` a `false`. Vlastní měření času vykreslování probíhá poněkud komplikovaněji než by se očekávalo. Po volání metody `Invalidate` a následném spuštění kódu v těle události `OnPaint` se zavolá metoda `ActionTime`, jejímž vstupním parametrem je speciální delegát

[34] Komponenta označovaná jako ProgressBar.

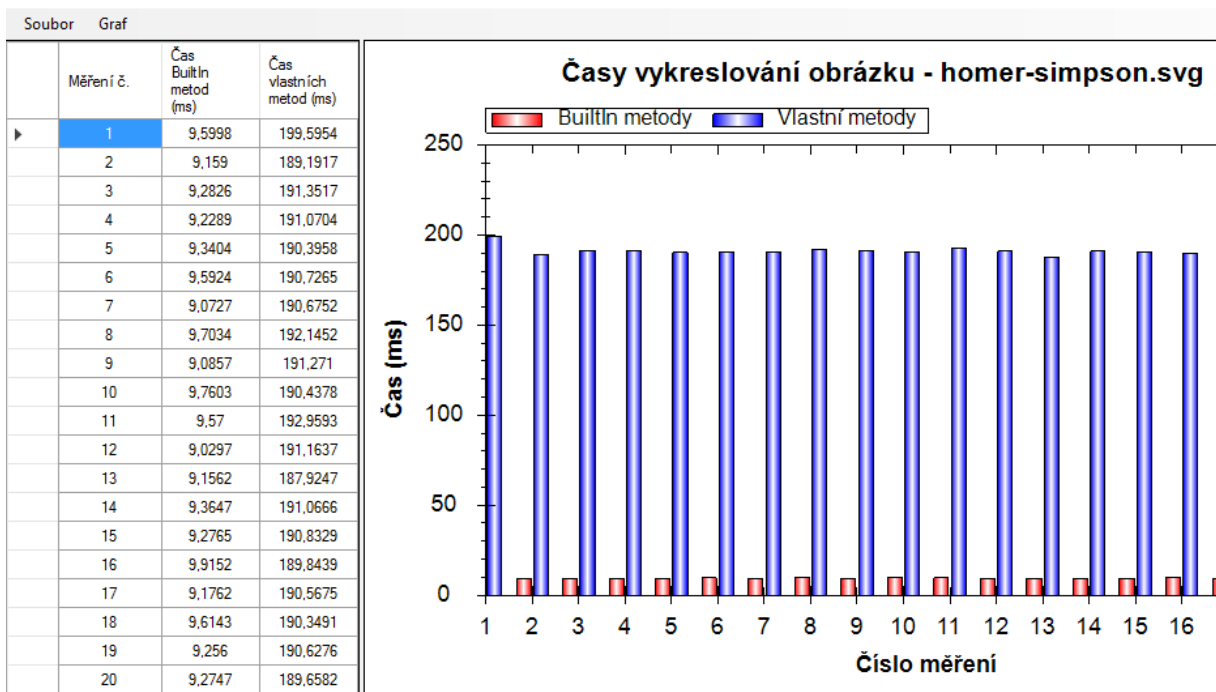
Action, který nemá žádné vstupní parametry a jeho návratová hodnota je `void`. Této anonymní metodě je přiřazena, např. použitím lambda výrazu, metoda `DoPaint`, která provádí vykreslování. Metoda `ActionTime`, která vrací strukturu *TimeSpan*, zajistí před spuštěním akce vyčištění garbage collectoru a nastavení nejvyšší priority procesu vykreslování. Čas je měřen v milisekundách s využitím třídy *StopWatch* ze jmenného prostoru *System.Diagnostics*. Pro názornost je uveden zdrojový kód:

```
private static TimeSpan ActionTime(Action action)
{
   GC.Collect();
   GC.WaitForPendingFinalizers();
   GC.Collect();
   Process.GetCurrentProcess().ProcessorAffinity = new IntPtr(2);
   Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;
   Thread.CurrentThread.Priority = ThreadPriority.Highest;

   Stopwatch sw = Stopwatch.StartNew();
   action();
   sw.Stop();
   return sw.Elapsed;
 }


protected override void OnPaint(PaintEventArgs pe)
{
   TimeSpan actionTime = ActionTime(() =>  { DoPaint(pe);  });
   DrawingTime = actionTime.TotalMilliseconds;
}
```

Bezprostředně po vytvoření datového souboru dojde k otevření formuláře, kde je uvedena přehledná tabulka naměřených hodnot a sloupcový nebo bodový graf pro jejich vizualizaci. I s přihlédnutím k řadě faktorů, které mohou naměřené hodnoty zkreslovat, jsou k dispozici poměrně relevantní informace o tom, jak si vlastní grafické metody stojí ve srovnání s metodami vestavěnými. Je zřejmé, že jisté kombinace algoritmů a jejich vstupních parametrů mohou poskytovat výsledky, které se kvalitou a rychlostí metodám .NET Frameworku nejen blíží, ale mohou je i překonat. Tato analýza je však již nad rámec této práce.



Měření č.	Čas BuiltIn metod (ms)	Čas vlastních metod (ms)
1	9,5998	199,5954
2	9,159	189,1917
3	9,2826	191,3517
4	9,2289	191,0704
5	9,3404	190,3958
6	9,5924	190,7265
7	9,0727	190,6752
8	9,7034	192,1452
9	9,0857	191,271
10	9,7603	190,4378
11	9,57	192,9593
12	9,0297	191,1637
13	9,1562	187,9247
14	9,3647	191,0666
15	9,2765	190,8329
16	9,9152	189,8439
17	9,1762	190,5675
18	9,6143	190,3491
19	9,256	190,6276
20	9,2747	189,6582

Obrázek 19 - Srovnání vlastních a vestavěných metod

4 Výsledky a diskuse

4.1 Rasterizace úsečky a Bézierovy křivky

V praktické části byly implementovány v jazyce C# algoritmy DDA a Bresenham pro rasterizaci úsečky. Pro modelování aproximační Bézierovy křivky byl implementován rekurzivní algoritmus de Casteljau a algoritmus s využitím Bernsteinových polynomů. Při vhodně zvoleném kroku v implementacích křivky, který leží zhruba v intervalu $\langle 0.01; 0.05 \rangle$ bylo co do kvality dosaženo v podstatě totožných výsledků zobrazení jako u metod, které jsou interně v této práci označovány jako vestavěné (bulit-in), tj. metody třídy *Graphics*. Při srovnání časové režie jednotlivých přístupů vykreslování však bylo zjištěno, že vlastní implementace se rychlostně té vestavěné řádově blíží až při nastavení takového kroku, že jsou Bézierovy křivky téměř nahrazeny sérií lomených čar. Otázkou ovšem zůstává, zda je dobře zvolen a naprogramován mechanismus měření času zpracování. Dále se v této oblasti otevírá poměrně široký prostor pro experimentování s různými kombinacemi nastavení vstupních parametrů, tedy typ algoritmu pro úsečku a Bézierovu křivku, iterační krok, vyhlazení hran. Při dané složitosti SVG souboru, tj. počtu elementů `<path>` a příkazů v nich,

by bylo zajímavé sledovat, zda právě zvolená kombinace parametrů není vhodnější (a rychlejší) než metody .NET Frameworku.

4.2 Aplikace SvgViewer

Od původní snahy o interpretaci všech elementů a jejich atributů, které se mohou vyskytnout v kódu SVG bylo nakonec upuštěno, jelikož převážná většina obrázků ve formátu SVG, která vznikla jako výstup nějakého grafického vektorového editoru popisuje obraz příkazy elementu `<path>`, resp. jeho atributu `d`. Z množiny všech deseti příkazů cesty dokáže komponenta *SvgViewer* identifikovat a zpracovat sedm. Záměrně byly vynechány příkazy: A,a – eliptický oblouk, Q,q – kvadratická Bézierova křivka a T,t – navazující kvadratická Bézierova křivka. Z předcházejícího textu, zejména teoretické části pojednávající o obecných Bézierových křivkách stupně *n* vyplývá, že všechny tři uvedené geometrické útvary lze nahradit minimálně kubickou Bézierovou křivkou. To je také důvodem, proč se tyto tři příkazy cesty vyskytují v SVG souborech velice zřídka. Pokud by se ovšem v datech objevily, syntaktický analyzér komponenty by je nezachytil, netransformoval jejich souřadnice a výsledný obraz by byl silně deformovaný. V této fázi vývoje také není vyřešen převod barvy zadaný identifikátorem IRI. To se netýká pochopitelně jen barvy, ale též barevných přechodů. Dalším ústupkem od původních cílů práce je vynechání vlastní implementace algoritmů pro vyhlazování grafických objektů a vyplňování uzavřených grafických cest. Důvodem je poměrně komplikované teoretické pozadí (a s ním spojená následná implementace), které by překročilo rámec této práce.

4.2.1 Individuální vykreslování cest

Na tomto místě je nejprve potřeba uvést podstatnou korekci použité terminologie. Ačkoli je příslušná záložka na komponentě TabControl pojmenována „Příkazy GDI+“, není vždy obsahem datové struktury *TreeView* na této záložce seznam použitých metod implementovaných v GDI+, resp. ve třídě *Graphics* knihovny *System.Drawing*. To se děje pouze za předpokladu, že je nastavena globální proměnná `UseBuiltInDrawingMethods` na hodnotu `true` a současně je hodnota property `SuppressGDIOutput` rovna `false`. V opačném případě je stromová struktura naplněna popisem volání metod statické třídy *Curves*. Sestavení grafických cest do stromové struktury je nástroj grafického uživatelského rozhraní, který stojí za povšimnutí. Jeho prostřednictvím může uživatel nejen nahlížet na

příkazy tříd *Graphics* nebo *Curves*, které cestu tvoří, ale může si nechat vykreslit pouze vybrané cesty. Tento „inspektor cest“ lze považovat za poměrně efektní a efektivní doplněk, který činí aplikaci v podstatě výjimečnou mezi ostatním softwarem podobného zaměření. Tento nástroj se ukázal jako vysoce užitečný např. při ladění aplikace, zejména pak při psaní transformačních algoritmů pro souřadnice.

4.2.2 **Testování**

V současnosti je nedílnou součástí vývoje software také vývoj různých druhů verifikačních testů, které jeho funkčnost komplexně prověří. Takové testy vznikají dokonce ještě dříve než software samotný. V případě komponenty *SvgViewer* žádná sada testovacích procedur napsána ani použita nebyla. Testování však probíhalo formou komparace výstupů z prohlížeče MS Internet Explorer v.11 na totožných souborech vytvořených ve vektorovém grafickém editoru Adobe® Illustrator, aby se ověřila správnost použitých metod a zejména mechanismus transformace relativních a absolutních souřadnic ze zdrojového SVG souboru.

5 Závěr

Ačkoli primárním cílem této práce bylo seznámení s prostředky vektorové grafiky, které nabízí .NET Framework, nakonec je výsledek tématicky přeci jen poněkud bohatší. V prvních kapitolách se čtenář dozvěděl stručná historická fakta o frameworku .NET jako takovém, následně byl obeznámen s jeho architekturou a jednou z klíčových komponent, kterou představuje běhové prostředí pro aplikace – Common Language Runtime. Další etapou při směřování k hlavnímu tématu práce bylo představení knihovny GDI+, jež formou relativně jednoduchých metod abstrahuje používání a ovládání hardwarových prostředků jako jsou grafické karty, tiskárny apod. Jakousi obálkou GDI+ pro prostředí .NET je pak mimo jiné jmenný prostor *System.Drawing*, který je pro tuto práci klíčový. V něm pak zcela fundamentální roli hraje třída *Graphics*.V další kapitole byl stručně popsán formát SVG, ze kterého čerpá data ukázková aplikace, která je součástí této práce. Při zpracování elementů a atributů SVG souboru jsou ze zdrojového kódu patrné i techniky a nástroje pro zpracování formátu XML, z něhož SVG vychází. Na teoretický part věnovaný třídě *Graphics* bezprostředně navazuje popis dvou metod, které mají svůj obraz právě ve formátu SVG, tedy metody `DrawLine` pro vykreslování úsečky a metody `DrawBezier` pro kubickou Bézierovu křivku. Pro oba tyto grafické vektorové útvary byly, s nastudovanou podporou matematického aparátu počítačové geometrie, teoreticky popsány nejběžnější algoritmy pro jejich rasterizaci, tj. pro výpočet a vykreslování bodů do rastru zobrazovacího zařízení. Přes původní ambice uvedení širšího spektra principů a metod modelování křivek, které nacházejí své uplatnění při zpracování grafických dat uložených v souboru formátu SVG, byly nakonec vysvětleny jen dva algoritmy pro úsečku a dva pro Bézierovu křivku. Ukázalo se totiž, že i s tak relativně omezenou paletou nástrojů lze zpracovat velkou část vektorových obrázků, které jsou dostupné po exportování z vektorových grafických editorů. Ve vlastní práci pak byly teoretické poznatky komplexně zpracovány do kompaktní aplikace nazvané SvgViewer, která umožňuje jednoduchým a intuitivním způsobem vybírat uložené SVG soubory a jejich grafická data interpretovat pomocí metod třídy *Graphics* či vlastní implementací renderovacích algoritmů v jazyce C#. Zajímavou přidanou hodnotou, která dostávala v průběhu vlastní programátorské práce konkrétnější kontury, je měření časové náročnosti obou přístupů vykreslování vektorových obrazů. V tabulkové i grafické podobě jsou k dispozici výsledky měření, které poskytují solidní základ nejen pro rychlé vizuální srovnání, ale i pro případnou statistickou analýzu. Za současného stavu výkonu uživatelsky

implementovaných vykreslovacích algoritmů je bohužel nutné konstatovat, že výsledky metod zapouzdřených ve frameworku .NET jsou výrazně lepší. Optimalizace vlastních metod je však nadále otevřená a výkonové přiblížení velmi reálné.

6 Seznam použitých zdrojů

Eisenberg, David J. Feb. 2002. *SVG Essentials.* 1st. Sebastopol : O'Reilly, Feb. 2002. p. 364. ISBN: 0-596-00223-8.

Farin, Gerald E., Hoschek, Josef and Kim, Myung Soo. 2002. *Handbook of Computer Aided Geometric Design.* Amsterdam : Elsevier B.V., 2002. ISBN: 978-0-494-51104-1.

Hardy, Alexandre and Steeb, Willi Hans. 2008. *Mathematical Tools In Computer Graphics with C# Implementations.* University of Johannesburg : World Scientific Publishing Co. Pte. Ldt., 2008. p. 493. ISBN-13 978-981-279-102-3.

Martišek, Dalibor. 2000. *Počítačová geometrie a grafika.* Brno : skripta Vysoké učení technické Brno, 2000. str. 89. ISBN: 80-214-1632-7.

Microsoft. 2012. GDI+ Reference (Windows). *Windows Dev Center.* [Online] 3 6, 2012. https://msdn.microsoft.com/en-us/library/windows/desktop/ms533799%28v=vs.85%29.aspx.

Mlýnková, Irena, a další. 2008. *XML technologie - Principy a aplikace v praxi.* Praha : Grada Publishing a.s., 2008. str. 272. ISBN: 978-80-247-2725-7.

Rektorys, Karel. 1988. *Přehled užité matematiky.* Praha : SNTL, 1988. p. 607. Vol. I. ISBN: 04-022-88-01.

Robinson, Simon, et al. 2001. *Professional C#.* 2nd Edition. Birmingham : Wrox Press Limited, 2001. p. 1200. ISBN: 1861004990.

Sharp, John. 2010. *Microsoft Visual C# 2010 - krok za krokem.* Brno : Computer Press a.s., 2010. ISBN: 978-80-251-3147-3.

Skřenek, Martin. 2015. *Jazyk C# - programování II.* Praha : GOPAS, 2015.

Sommerville, Ian. 2013. *Softwarové inženýrství.* [trans.] Jakub Goner. Brno : Computer Press, 2013. ISBN: 978-80-251-3826-7.

Špička, Ivo a Frischer, Robert. 2012. *Počítačová geometrie a grafika - Teoretické základy.* Ostrava : Technická univerzita Ostrava - Vysoká škola báňská, 2012. str. 114. ISBN: 978-80-248-2590-8.

Tabor, Robert. 2002. *Microsoft .NET XML Web Services.* s.l. : Sams, 2002. ISBN: 0-672-32088-6.

Thai, Thuan L. and Lam, Hoang Q. 2002. *.NET Framework Essentials.* 2nd. Sebastopol : O'Reilly, 2002. p. 320. ISBN: 0-596-00302-1.

Žára, Jiří, et al. 2004. *Moderní počítačová grafika.* Brno : Computer Press, 2004. ISBN: 80-251-0454-0.

7 Přílohy

CD s aplikací a zdrojovými kódy