Vícevláknová multiplatformní grafická knihovna

Multithreaded Multiplatform Graphics Library

Bc. Petr Kobalíček

Diplomová práce
2009

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2008/2009

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Petr KOBALÍČEK**
Studijní program: **N 3902 Inženýrská informatika**
Studijní obor: **Informační technologie**

Téma práce: **Vícevláknová multiplatformní grafická knihovna**

Zásady pro vypracování:

1. **Vytvořte literární rešerši na téma rasterizace geometrických objektů a možné optimalizace SW algoritmů pro její urychlení. Zhodnoťte současný stav a schopnosti grafických subsystémů (Cairo, GDI+, Antigrain, ...) současných nejrozšířenějších operačních systémů (MS Windows, Linux, MacOS).**
2. **Vytvořte multiplatformní grafickou knihovnu schopnou využít více vláken a rozšíření CPU pro vykreslování grafických objektů. Zvažte a případně implementujte možnosti JIT kompilace nízkoúrovňových grafických funkcí.**
3. **Navrhněte a implementujte základní objekty grafické knihovny (písmo, obrázek, region, grafický kontext, framework pro načítání / ukládání obrázků).**
4. **Navrhněte a implementujte funkce pro operace s rastrem na nejnižší vrstvě (optimalizované funkce, využití technologií MMX, SSE2). Navrhněte a implementujte optimalizace vykreslení gradientu pomocí MMX a SSE2.**
5. **Srovnejte rychlost renderování a možnosti výsledné softwarové knihovny s jinými podobnými knihovnami.**
6. **Navrhněte metodiku integrace nového grafického subsystému s dalšími grafickými knihovnami a GUI toolkity pracujícími na vyšší úrovni (wxWidgets, Qt, GTK, FLTK, ...).**
7. **Vytvořte programovou dokumentaci Vašeho grafického systému.**

Rozsah práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. **HYDE, Randall. The Art of Assembly Language. First Edition, 2004. No Starch Press. ISBN 1-886-41197-2.**
2. **HARRIS, Simon – ROSS, James. Beginning Algorithms. First Edition, 2006. Wiley Publishing, Inc. ISBN 0-764-59674-8.**
3. **MATTHEW, Nail – STORES, Richard. Beginning Linux Programming. Third Edition, 2004. Wiley Publishing, Inc. ISBN 1-861-00297-1.**
4. **HART, Johnson. Windows System Programming. Third Edition, 2005. ISBN 0-321-25619-0.**
5. **HYDE, Randall. Write Great Code Volume II – Thinking Low Level, Writing High Level. First Edition, 2006. Addison Wesley Professional. ISBN 1-593-27065-8.**
6. **Microsoft Developer Network [online]. URL: (http://msdn.microsoft.com).**
7. **AntiGrain Geometry [online]. URL: (http://www.antigrain.com/).**
8. **Cairo Graphics [online]. URL: (http://cairographics.org/).**

Vedoucí diplomové práce: **Ing. Michal Bližňák, Ph.D.**
Ústav aplikované informatiky

Datum zadání diplomové práce: **20. února 2009**

Termín odevzdání diplomové práce: **27. května 2009**

Ve Zlíně dne 13. února 2009

L.S.

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Cílem práce bylo vytvořit literární rešerši na téma rasterizace geometrických objektů a možné optimalizace SW algoritmů pro její urychlení. Dále zhodnotit současný stav a schopnosti grafických subsystémů Cairo, GDI+ a AntiGrain, které se používají v nejrozšířenějších operačních systémech. Praktická část se zabývá návrhem a implementací vlastní grafické knihovny v jazyce C++, která bude schopná využít více vláken a multimediální instrukce procesoru x86 a x86_64 pro maximalizaci výkonu. Součástí návrhu a implementace je popis navržených tříd v knihovně, jejich optimalizace a demonstrační ukázky výstupu knihovny. Praktická část se dále zabývá porovnáním výkonu navržené knihovny a návrhem metodiky integrace s jinými grafickými knihovnami a GUI toolkity.

Klíčová slova: grafická knihovna, rasterizace, C++, JIT kompilace, MMX, SSE2, vlákna.

ABSTRACT

The aim of this work was to create literature research about rasterization of geometric objects and possible optimizations of software based rasterization algorithms. In addition, to assess the current status and capabilities of graphics libraries Cairo, GDI+ and AntiGrain, which are used in most operating systems. The practical part deals with design and implementation of new graphics library in C++ language, which will be able to use multithreading and special x86 or x86_64 CPU instructions to maximize its performance. Part of library design and implementation is description about its classes, optimizations and demonstration examples of library output. The practical part also deals with comparing the performance of the proposed library and integration with other graphics libraries and GUI toolkits.

Keywords: graphics library, rasterization, C++, JIT compilation, MMX, SSE2, threads.

Tímto bych chtěl poděkovat panu Ing. M. Bližňákovi Ph.D. za odborné vedení při zpracování diplomové práce. Chtěl bych také poděkovat všem ostatním, kteří se o navrženou knihovnu zajímali a podporovali mě při vývoji.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.

Ve Zlíně ……

Podpis diplomanta

OBSAH

ÚVOD

Každý počítačový program, který komunikuje s uživatelem pomocí grafického rozhraní (nebo jen prezentuje výsledky v grafické formě) musí obsahovat programovou vrstvu, která umožňuje samotnou realizaci požadavků na grafický výstup. Tyto programy zpravidla neimplementují grafickou vrstvu samy, ale používají API cílových operačních systémů nebo knihovny třetích stran. Jen ve výjimečných případech se vyplatí při psaní programu implementace vlastní grafické knihovny.

V minulosti probíhalo vykreslování v aplikacích velmi často na rastrové úrovni, dnešní trendy směřují spíše k vektorově orientované grafice, která přináší nové možnosti ve vizualizaci a interakci s uživatelem. Rozmach vektorové grafiky ale také znamená vyšší výpočetní výkon pro vykreslení geometrických objektů, proto se většina grafických knihoven píše v jazyce C/C++ a jejich kritické části i v assembleru.

I přesto, že vytváření grafických knihoven je časově i finančně náročné, existuje mnoho aplikací, které obsahují vlastní implementace mini-knihoven, které se starají o grafickou prezentaci nebo alespoň její část. Mezi nejčastější důvody patří odstranění závislosti na knihovnách operačních systémů (multiplatformní aplikace), vysoké nároky na kvalitu a přesnost výstupu (např. mapy), ale i vysoké požadavky na množství funkcí v grafické knihovně.

Protože jen málo knihoven je ve svém návrhu orientovaných čistě na rychlost, nabízí se otázka, jestli by taková knihovna nemohla konkurovat ostatním grafickým knihovnám. Není škoda, že v době vícejádrových procesorů s pokročilými technologiemi jako MMX nebo SSE2 tyto technologie mnohdy nejsou využity pro grafické prezentace aplikací? Často slýcháme v souvislosti s výkonem i o JIT (just in time) kompilaci, která se často používá na urychlení skriptovacích jazyků, ale na urychlení grafických knihoven se zatím používá jen minimálně nebo vůbec.

Grafická knihovna Fog (Fair Object Graphics) byla navržena jako pokus využít všechny zmíněné technologie s cílem maximalizovat výkon vykreslování vektorové grafiky. Knihovna podporuje operační systémy Windows a Linux, a byla navržena s ohledem na budoucí vývoj i pro jiné operační systémy (BSD, Mac OS X) a rozšiřitelnost samotné knihovny o nové funkce.

I. TEORETICKÁ ČÁST

1 RASTERIZACE GEOMETRICKÝCH OBJEKTŮ

1.1 Úvod

Rasterizace je proces, při kterém se z rastrově nebo vektorově definovaných vstupních dat vytvoří rastrový výstup. Rastr je dvojrozměrné pole, jehož základní jednotkou je pixel. Zdroj [8] uvádí definici rasterizace jako převod grafických prvků do posloupnosti pixelů. Existuje několik rasterizačních algoritmů, které se člení v závislosti na typu vstupních dat a typu výstupní operace. V souvislosti s počítačovou grafickou budeme mluvit o rasterizaci 2D objektů rastrového i vektorového charakteru.

Při rasterizaci rozeznáváme tzv. rastrové a vektorové souřadnice. Rastrové souřadnice popisují fyzické umístění pixelů na rastru a jsou celočíselného typu. Vektorové souřadnice popisují logické umístění a jsou většinou definována formou reálných čísel. Logické data většinou představují světové (world) souřadnice, zatímco fyzické data klientské (client) souřadnice. O světových a klientských souřadnicích většinou mluvíme v souvislosti s afinními transformacemi, které je možné aplikovat na vektorově definované souřadnice a získat tak souřadnice fyzické. Pokud není umožněno aplikovat afinní nebo jiné transformace na vstupní souřadnice, můžeme termíny klientské a světové úplně vynechat.

1.2 Formáty pixelů

V souvislosti s vznikem počítačové grafiky vznikaly i formáty pixelů. Formát pixelu je definice způsobu uložení hodnoty v pixelu, která reprezentuje jeho barvu. V dnešní době je pixel definován většinou jako 24bitová RGB nebo 32bitová ARGB entita popisující jeho jednotlivé barevné složky:

- R (red) – červená,
- G (green) – zelená,
- B (blue) – modrá,
- A (alpha) – alfa kanál (průhlednost).

V minulosti byla ale počítačová paměť omezená a v počítačové grafice byly pixely reprezentovány 1, 4 a 8 bity. Následoval rozmach hicolor grafiky, která definovala pixel

jako 16bitovou RGB entitu popisující jeho jednotlivé složky barev (5 bitů pro červenou, 6 bitů pro zelenou a 5 bitů pro modrou).

Následující tabulka popisuje barevné hloubky pixelů a způsob uložení jeho hodnot.

Hloubka	Formát	Popis
1 bit	MONO1	Monochromatický formát, který obsahuje jen 2 barvy, které jsou většinou definované jako černá a bílá.
4 bity	GREY4	16 stupňů odstínů šedi.
4 bity	INDEX4	16 barev, jejichž RGB hodnoty jsou definované v externí paletě barev. Hodnota pixelu je jen index do palety barev.
8 bitů	GREY8	256 stupňů odstínů šedi.
8 bitů	INDEX8	256 barev, jejichž RGB hodnoty jsou definované v externí paletě barev. Hodnota pixelu je jen index do palety barev.
16 bitů	RGB565	65536 barev. Hodnota barvy definuje zároveň její RGB hodnotu (červená 5 bitů, zelená 6 bitů, modrá 5 bitů). Tento model se někdy označuje termínem hicolor.
16 bitů	RGB555	32768 barev. Hodnota barvy definuje zároveň její RGB hodnotu (červená 5 bitů, zelená 5 bitů, modrá 5 bitů).
24 bitů	RGB24	16 milionů barev, přesněji 16777216. Hodnota každé barevné složky RGB je definována jako 8bitová hodnota. Každá barevná složka má tedy 256 odstínů. Tento formát je označován termínem truecolor.
32 bitů	ARGB32	Tento formát je podobný 24bitové hloubce, ale obsahuje navíc alfa kanál, který definuje průhlednost pixelu.
48 bitů	RGB48	RGB formát, ale každá barevná složka je definována jako 16bitová hodnota (tedy 65536 odstínů).
64 bitů	ARGB64	RGBA formát, ale každá barevná složka včetně alfa kanálu je definována jako 16bitová hodnota (tedy 65536 odstínů).

Kromě zmíněných formátů v tabulce ještě existuje mnoho různých formátů, které můžou být definovány jejich kombinací nebo jiným počtem bitů na barevné složky. Prakticky kdokoliv si může definovat svůj vlastní formát, je tedy nemožné je vypsat všechny. V souvislosti s počítačovou grafickou se ještě používá 128bitový formát založený na reálných číslech (float), kde je barva definována v intervalu od 0 do 1. Tento formát se používá velice často jako výstup z rasterizace pomocí metody raytracing.

V souvislosti s formátem pixelů, který obsahuje alfa kanál se ještě používají 2 termíny:

- premultiplied (přednásobený),
- nonpremultiplied (standardní).

Přednásobený formát se používá velice často, pokud chceme smíchat rastrové obrázky na základě hodnot jejich alfa kanálů (průhlednosti). Tento druh operace se nazývá kompozitní. Přednásobený formát je definován jako součin barevných složek pixelu s jeho alfa kanálem [6]. Tedy

$$\begin{aligned} c_{ra} &= c_r * c_a \\ c_{ga} &= c_g * c_a \\ c_{ba} &= c_b * c_a \end{aligned} \tag{1}$$

kde

- c_r, c_g a c_b jsou barevné složky RGB,
- c_a je alfa kanál,
- c_{ra}, c_{ga} a c_{ba} jsou barevné složky RGB vynásobené alfa kanálem.

Pokud je tedy poloprůhledná červená barva definována jako RGBA entita v podobě (1, 0, 0, 0.5), bude po přednásobení vypadat jako (0.5, 0, 0, 0.5). Kompozitní operace jsou probrány v kapitole 1.6 a v kapitole 5.2.3.7 je možné vidět výsledky různých kompozitních operací nad 2 obrázky s alfa kanálem.

Pokud chceme převést přednásobený formát pixelů na standardní, je třeba barevné složky vydělit hodnotou alfa kanálu, pokud je jeho hodnota větší než 0 [6], tedy:

$$\begin{aligned} c_r = c_a > 0 &\rightarrow c_{ra} / c_a : 0 \\ c_g = c_a > 0 &\rightarrow c_{ga} / c_a : 0 \\ c_b = c_a > 0 &\rightarrow c_{ba} / c_a : 0 \end{aligned} \tag{2}$$

1.3 Rastrové vs. vektorové data

Jak už bylo zmíněno, rastrové data jsou reprezentována na rastrové úrovni ve formě dvojrozměrného pole pixelů, z nichž každý má přesnou pozici a hodnotu. Výsledný obraz je tedy složen z jednotlivých pixelů, a platí, že z čím většího počtu pixelů je obraz složen, tím je kvalitnější a má větší rozlišení, ale z pohledu datové velikosti zabírá více paměťového prostoru [5]. Pokud chceme zvětšit nebo zmenšit rastrový obrázek, musíme provést jeho interpolaci a kvalita výsledného obrazu už nebude nikdy taková, jako originál.

Naproti tomu vektorové data jsou definována pomocí matematického modelu. Aby mohl být takový obrázek zobrazen, je nutné provést rasterizaci, protože výstupní zařízení většinou pracují na rastrové úrovni. Matematický model zajišťuje, že vektorově definovaný obrázek je možné jakkoliv transformovat bez zhoršení jeho kvality, protože se rasterizuje vždy výsledek po aplikaci afinních nebo jiných transformací. Z toho plyne, že vektorový obrázek lze zvětšovat nebo zmenšovat bez ztráty na jeho kvalitě [5].

1.4 Přehled rasterizačních algoritmů

Rasterizační algoritmy se rozdělují do 3 kategorií:

- analytický rasterizer,
- tesselator,
- raytracing.

Analytický rasterizer pracuje na principu vytváření seznamu pixelů z polygonální uzavřené oblasti. Tesselator vytváří z polygonální uzavřené oblasti trojúhelníky (triangles) nebo rovnoběžníky (trapezoids), které slouží jako vstupní data pro grafické karty nebo jednodušší rasterizer, který pracuje jen na základě trojúhelníků. Tesselator se vždy používá v souvislosti s předáním informací grafické kartě (pro rozhraní openGL nebo DirectX). Tesselator, který vytváří rovnoběžníky je implementovaný například v grafické knihovně Cairo, naproti tomu analytický rasterizer je součástí knihovny AntiGrain.

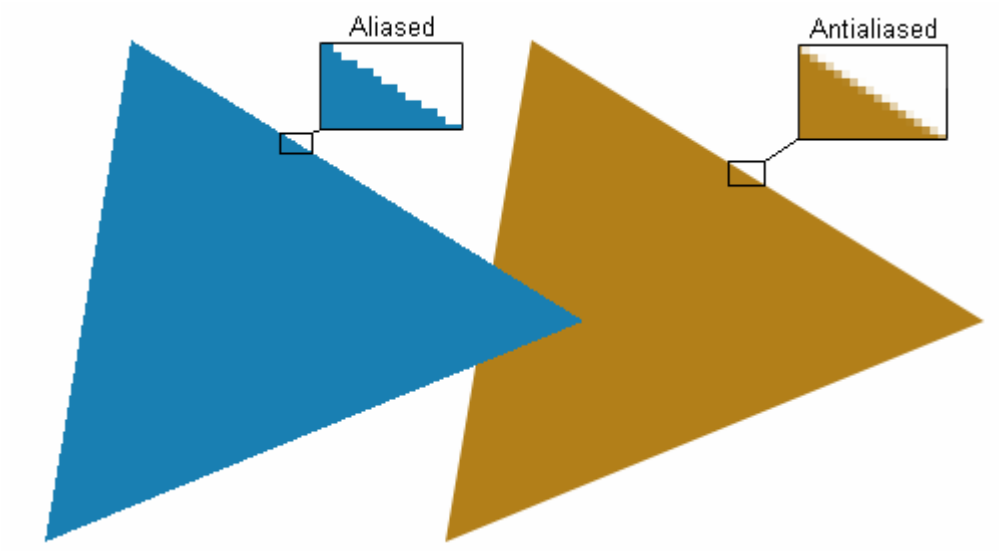
Raytracing je metoda založená na sledování paprsků světla, která se používá ve vysoce kvalitní 3D grafice a její výsledek jsou opravdu věrné obrázky. Zde je tento působ rasterizace uveden jen pro kompletnost.

1.5 Aliasing / Antialiasing

Způsob rasterizace může být rozdělen do 2 skupin:

- výsledek včetně informací o průhlednosti pixelů (antialiasing),
- výsledek bez informací o průhlednosti pixelů (binární rasterizery).

Rozdíl mezi způsobem rasterizace je znázorněn na obrázku (*Obr. 1*).



Obr. 1: Aliasing vs. antialiasing.

V době vektorové grafiky se velice často používají rasterizery, jejichž výsledkem jsou data včetně průhlednosti (antialiasing), protože velice věrně vykreslují hrany mezi vyrenderovaným geometrickým objektem a původním obrázkem (ať už se jedná o pozadí nebo jiné dříve vyrenderované objekty). V souvislosti s antialiasingem mluvíme velice často o tzv. kompozitních operacích, které jsou popsány v následující kapitole.

1.6 Kompozitní operace

Kompozitní operace definují způsob míchání 2 pixelů, které obsahují alfa kanál (průhlednost). Tyto operace definovali Porter a Duff v roce 1984 [6]. V počítačové grafice, kde výstup z rasterizeru i cílový rastrový obrázek může obsahovat alfa kanál bylo zapotřebí definovat způsob jejich kompozice (smíchání).

Kompozitní operace tedy definují způsob, jakým se smíchají barvy a alfa kanál zdrojového a cílového pixelu [6]. Kompozitní operace jsou definovány v barevném prostoru ARGB, kde složky RGB jsou přednásobené hodnotou alfa kanálu (premultiplied) [6]. Zdrojový

pixel můžeme označit jako *A* a cílový jako *B*. Přednásobené barevné složky zdrojového a cílového pixelu označíme jako A_c a B_c a jejich alfa kanál jako A_a a B_a. Výsledná průhlednost pixelu po kompozici je popsána v tabulce (*Tab. 1*).

Tab. 1: Matematický popis výsledné průhlednosti kompozice 2 pixelů [6].

B	A	oblast	Popis	Možnosti
0	0	0	$\overline{A} \cap \overline{B} = (1 - A_a) * (1 - B_a)$	0
0	1	A	$A \cap \overline{B} = (A_a) * (1 - B_a)$	0, A
1	0	B	$\overline{A} \cap B = (1 - A_a) * (B_a)$	0, B
1	1	AB	$A \cap B = (A_a) * (B_a)$	0, A, B

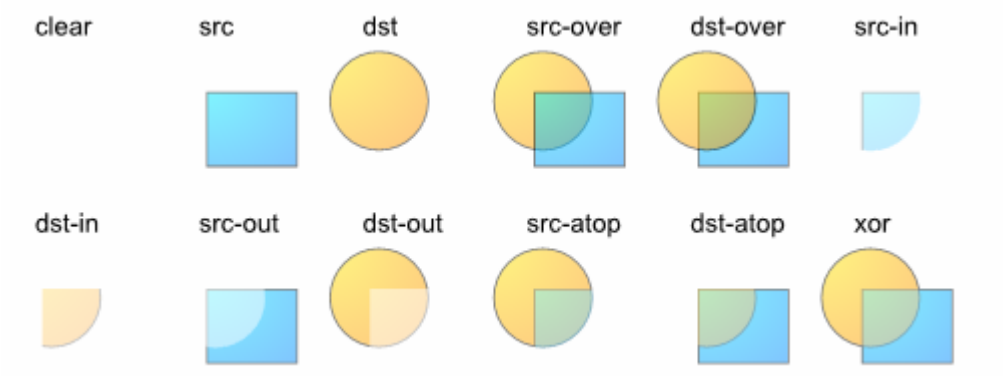
Možnosti v tabulce (*Tab. 1*) označují vliv alfa kanálu na jeho výslednou hodnotu. Výsledná hodnota barevných složek je komplikovanější a originální materiál [6] popisuje 12 možných kombinací. Každá kombinace je popsána pomocí tzv. quadruple (čtyřnásobek) kde přispívá každý subpixel v oblasti 0, A, B, AB (*Tab. 1*). Jednotlivé kompozitní operace jsou definovány v tabulce (*Tab. 2*).

Tab. 2: Definice kompozitních operací Porter & Duff.

Název operace	Quadruple	F_A	F_B
clear	(0, 0, 0, 0)	0	0
A	(0, A, 0, A)	1	0
B	(0, 0, B, B)	0	1
A over B	(0, A, B, A)	1	1 - A_a
B over A	(0, A, B, B)	1 - B_a	1
A in B	(0, 0, 0, A)	B_a	0
B in A	(0, 0, 0, B)	0	A_a
A out B	(0, A, 0, 0)	1 - B_a	0
B out A	(0, 0, B, 0)	0	1 - A_a

A atop B	(0, 0, B, A)	B_a	1 - A_a
B atop A	(0, A, 0, B)	1 - B_a	A_a
A xor B	(0, A, B, 0)	1 - B_a	1 - A_a

Ilustrace základních Porter & Duff kompozitních operací z tabulky (*Tab. 2*) je zobrazena na obrázku (*Obr. 2*). Tento obrázek je ze specifikace standardu SVG, který je na Porter & Duff kompozici založený [14]. Kompozitní operace dvou rastrových obrázků je zobrazena i v kapitole 5.2.3.7 na obrázku (*Obr. 24*).



Obr. 2: Kompozitní operace Porter & Duff [14].

1.7 Základní optimalizace SW algoritmů

Tato kapitola slouží jako úvod k optimalizaci rasterizačních algoritmů. Nejsou zde uvedeny konkrétní algoritmy, ale optimalizační tipy, které platí téměř pro jakýkoliv algoritmus. Při optimalizaci rasterizačních algoritmů je velmi důležité porozumět počítačové architektuře, protože se většinou jedná o velké paměťové přesuny a optimalizace se velice často věnují jen minimalizování čtení nebo zápisu do náhodných úseků v paměti a maximalizaci čtení nebo zápisu z úseku sekvenčního. Dále je potřeba vzít v úvahu, že aplikace metod zmíněných v této kapitole by neměla být prováděna na základní rasterizační algoritmy, ale na výplně objektů polygonálního charakteru.

1.7.1 Optimalizace analytického rasterizeru

Tato podkapitola obsahuje základní optimalizace zjištěné ze zdrojových kódů analytického rasterizeru v knihovně antigrain [9], protože se jedná o velmi výkonnou knihovnu. Ještě

než se začneme věnovat samotným optimalizacím, je nutné popsat proces rasterizace v knihovně AntiGrain:

- převod křivek na úsečky,
- aplikování stylu čar na vstupní úsečky a vytvoření uzavřeného polygonálního objektu,
- aplikování afinních transformací na polygonální objekt,
- převod uzavřeného polygonálního objektu na pixely,
- výplň.

Téměř každý z výše uvedených kroků je v knihovně AntiGrain optimalizován. Převod křivek na úsečky je optimalizován pomocí adaptivního dělení bézierových křivek [7]. Následuje převod vstupního objektu na uzavřený polygonální objekt. Tento převod je vynechán v případě, že se již o uzavřený objekt jedná (výplň). Dále jsou aplikované afinní transformace na současný polygonální objekt. Na této operaci se s výjimkou specializace na různé případy nevztahují žádné optimalizace (pokud nevyužijeme SIMD instrukce).

Zbývá převod uzavřeného polygonálního objektu na seznam pixelů. Tento krok je jeden z nejdůležitějších kroků a v souvislosti s knihovnou AntiGrain se na něho vztahují tyto optimalizace:

- proces převodu je generován pomoci algoritmu, který pracuje s 32bitovými celými čísly (nepracuje se tedy s plovoucí desetinnou čárkou). 32bitové číslo je rozděleno na 24 bitů reprezentující část před desetinnou čárkou (souřadnice rastru) a 8 bitů, které definují prostor za desetinnou čárkou (subpixel) - v počítačové literatuře se tento termín nazývá jako fixed point 24.8,
- algoritmus generuje seznam pixelů, který je ukládán do pole pro každý řádek obrazu zvlášť – jedná se tedy o scanline (řárkový) algoritmus,
- při dokončení rasterizace se seznam pixelů pro každý řádek setřídí podle X souřadnic,
- následuje vykreslení pixelů řádek po řádku.

Můžeme si všimnout, že výsledek rasterizace je uložen do rastrového obrazu řádek po řádku. Toto je asi jedna z nejdůležitějších vlastností knihovny AntiGrain, protože tento přístup je optimální pro moderní paměťové sběrnice, kde sekvenční přístup k paměti (sequential access) je mnohem rychlejší než náhodný (random access).

V souvislosti s optimalizací rasterizačních algoritmů byl v roce 2007 publikován algoritmus nazvaný „Scanline edge-flag algorithm for antialiasing“ [4], který v testech dosahuje ještě lepších výsledků než analytický rasterizer, který je v knihovně AntiGrain. Tento algoritmus sice produkuje jen 64 stupňů šedi (antigrain plných 256), ale výsledek je lidským okem nepozorovatelný [4].

1.7.2 Optimalizace operací na rastrové úrovni

Při optimalizaci na rastrové úrovni se většinou využívá rozšíření cílového procesoru. Na platformě x86 a x86_64 existují rozšíření MMX a SSE2, které umožňují zpracovat více dat v jediné instrukci. Více o této optimalizaci je napsáno v praktické části, kapitola 5.3.

2 GRAFICKÉ KNIHOVNY

Grafická knihovna je soubor datových struktur a funkcí, které umožňují aplikaci výstup v podobě rastrové nebo vektorové grafiky. Grafické knihovny se liší tím, jestli používají HW akceleraci k urychlení grafického výstupu nebo jen SW akceleraci. Dále se mohou lišit podporou programovacích jazyků a operačních systémů. Tato kapitola obsahuje popis 3 grafických knihoven, které používají k renderování čistě SW akceleraci, a které se používají v operačních systémech Windows a Linux / BSD.

2.1 Cairo

Cairo je čistě vektorová grafická knihovna napsaná v jazyce C [12], která vznikla původně pro X Window System (okenní systém používaný v operačním systému Linux / BSD). Časem se ale z této knihovny stala multiplatformní knihovna podporující všechny nejpoužívanější operační systémy. Cairo obsahuje robustní návrh výstupních modulů, díky kterému je možné grafický výstup serializovat například i do PDF nebo postscriptu (vektorové grafické formáty). V současnosti se cairo používá jako grafický subsystém v toolkitu GTK+ a v internetovém prohlížeči Firefox.

Obr. 3: Logo knihovny Cairo [12].

Cairo používá pro nízkoúrovňovou manipulaci s pixely knihovnu pixman, která vznikla ze zdrojových kódu implementující framebuffer v X serveru, konkrétně XRender Extension. Knihovna pixman implementuje základní datové struktury a algoritmy pro nízkoúrovňovou práci s pixely, která probíhá na scanline úrovni ve formátu ARGB32-Premultiplied. Průběh rasterizace není ale na scanline úrovni a používají se data ve formě rovnoběžníku (trapezoid). Cairo pomocí tesselatoru převede vektorové data na trojúhelníky a ty se rasterizují pomocí knihovny pixman do pomocného bufferu, který se použije jako maska pro provedení kompozitní operace na cílový rastrový buffer.

Z pohledu diplomové práce je knihovna pixman zajímavější než samotné cairo, protože cairo je jen vysokoúrovňová vrstva postavená nad knihovnou pixman. Pixman může být

pro některé programátory nebo firmy zajímavá knihovna i kvůli licenci, která se drží licence X Serveru (MIT Licence) a umožňuje tedy téměř jakékoliv použití této knihovny.

2.1.1 Možnosti knihovny Cairo

Knihovna Cairo definuje poměrně jednoduché programové rozhraní, díky kterému je možné provádět grafické operace. Kromě grafického kontextu obsahuje i vybavení pro práci s rastrovým obrazem, písmem, maticové transformace a tzv. povrch (surface). Surface je z pohledu programátora výstupní zařízení, do kterého jsou serializované grafické operace, kromě paměťového bufferu se může jednat o PDF dokument, PostScript, PNG nebo SVG obrázek [13].

Grafický kontext v knihovně Cairo je založený na stavech. Ještě než dojde k rasterizaci a vykreslení geometrického objektu, je nutné nastavit jeho vlastnosti (například tloušťka čáry, vzor, ...), a až poté se volá funkce na samotné vykreslení. Po vykreslení zůstávají stavy nezměněné, je tedy možné velice jednoduše vykreslit sadu objektů se stejnými vlastnostmi za sebou. Při vykreslování geometrických objektů se rozlišuje mezi výplní (fill) a tahem (stroke).

Cairo podporuje všechny kompozitní grafické operace, které definovali Porter & Duff [6] a některé (Add a Saturate) definované v SVG specifikaci [14]. Výchozí operace je CAIRO_OPERATOR_OVER (kreslit přes), která patří zároveň mezi ty nejčastěji používané.

2.1.2 Ukázka použití v knihovny Cairo v C jazyce

Následující programová ukázka slouží pro vykreslí obdélníku do grafického kontextu v knihovny Cairo:

```
// Hlavičkový soubor pro knihovnu Cairo.
#include <cairo.h>

// Vytvoření grafického kontextu (surf je typu cairo_surface_t).
cairo_t* cr = cairo_create(surf);

// Nastavení barvy na bílou.
cairo_set_source_rgba(cr, 1.0, 1.0, 1.0, 1.0);

// Přidání obdélníku do současné cesty.
cairo_rectangle(cr, 50.0, 50.0, 100.0, 100.0);

// Výplň.
cairo_fill(cr);

// Zničení grafického kontextu.
cairo_destroy(cr);
```

2.1.3 Souhrn možností knihovny Cairo

- vykreslení a výplň jednoduchých objektů a grafických cest,
- vykreslení s použitím vzorů (lineární nebo radiální gradient, rastrový obrázek),
- vykreslení písma,
- možnost aplikovat maticové transformace na vykreslovaný objekt nebo vzor (rotace, zvětšení, zmenšení, posuv),
- možnost zvolit grafický výstup (PDF dokument, PostScript, SVG dokument, PNG obrázek nebo paměťový buffer),
- knihovna je plně vektorová (v reálných číslech se definují i barvy).

2.2 GDI+

Grafická knihovna GDI+ tvoří v současnosti jeden z grafických subsystémů v operačním systémů Windows, který vylepšuje starší subsystém GDI o vektorovou grafiku a podporu pro antialiasing. Knihovna GDI+ je napsaná v čistém C jazyce, ale obsahuje vrstvy pro C++ a .NET framework [15].

Knihovna GDI+ byla poprvé distribuována s operačním systémem Windows XP a byla backportovaná i na starší Windows 2000. Jejím cílem bylo vyplnit místo standardní vektorově orientované grafické knihovny, která pod Windows tehdy chyběla. Svým designem se jedná o knihovnu s jasně definovaným API, která navazuje na starší GDI a umožňuje kreslit mnohem zajímavější a lépe vypadající ovládací prvky než bylo možné dříve. Knihovna GDI+ je uzavřená a její zdrojové kódy nejsou k dispozici.

2.2.1 Možnosti knihovny GDI+

Knihovna GDI+ představuje ucelený balík funkcí pro počítačovou grafiku. Samozřejmostí je vykreslení a výplň vektorově definovaných grafických cest, jejich afinní transformace a nastavení stylu pera a štětce. Oproti jiným grafickým knihovnám ale podporuje i načítání a ukládání rastrových obrázků všech možných formátů a aplikaci matice barev (color matrix) na jejich vykreslení.

Grafická knihovna GDI+ je nestavová (s výjimkou afinních transformací). Grafický kontext přebírá při renderování vždy co nejvíce informací popisující renderování

geometrického objektu a jeho stylu (např. definici pera, štětce, písma). Oproti knihovně Cairo se tedy jedná o rozdílný přístup v koncepci grafického kontextu. Jednou z výhod knihovny GDI+ je velice dobrá integrace s operačním systémem Windows a možnost tisku grafických příkazů pomocí stejných tříd jaké jsou využívány ke kreslení do rastru.

2.2.2 Ukázka použití knihovny GDI+

Následující ukázka použije grafický kontext GDI+ k vykreslení obdélníku:

```
// Vytvoření grafického kontextu
Gdiplus::Graphics gr(dc);

// Nastavení barvy na bílou
Gdiplus::Color c(0xFFFFFFFF);

// Vytvoření štětce.
Gdiplus::SolidBrush br(c);

// Vyplnění obdélníku.
gr.FillRectangle(&br, 50.0, 50.0, 100.0, 100.0);
```

2.2.3 Souhrn možností knihovny GDI+

- vykreslení a výplň jednoduchých objektů a grafických cest (*GraphicsPath*),
- ovlivnit nastavení pera (změna stylů čáry, barva, vzor),
- vykreslení s použitím štětců (lineární gradient, rastrový obrázek),
- vykreslení písma a nahrazení znaků písmen (unikátní funkce),
- možnost aplikovat maticové transformace na vykreslovaný objekt nebo štětec (rotace, zvětšení, zmenšení, posuv),
- možnost kreslit do rastrového obrázku nebo do zařízení reprezentující tiskárnu,
- možnost transformace barev pomocí matice (color matrix),
- možnost načítat a ukládat rastrové obrázky různého typu (.bmp, .jpeg, .png, ...),
- velmi dobrá integrace s Windows.

2.3 AntiGrain

AntiGrain je v současné době velmi populární grafická knihovna, která je svým návrhem naprosto odlišná od ostatních. Celý koncept kolem knihovny AntiGrain je založený na C++ šablonách, díky kterým je možné specializovat rasterizační pipeline na míru vstupních dat.

AntiGrain je označován jako renderovací engine, který produkuje rastrové data z vektorového vstupu [9].

Obr. 4: Logo knihovny AntiGrain [9].

Základní vlastnosti knihovny AntiGrain [9]:

- antialiasing,
- nejvyšší možná kvalita výstupu,
- vysoký výkon,
- platformní nezávislost,
- flexibilita a rozšiřitelnost,
- jednoduchý, ale velice mocný design knihovny,
- spolehlivost a přesnost.

Rasterizace v knihovně AntiGrain probíhá na scanline úrovni (řádek po řádku). Z vektorových dat se vytvoří seznam, který obsahuje souřadnice pixelů (x, y), šířku (width) a masku. V počítačové grafice se taková datová struktura nazývá span nebo cell (pokud neobsahuje šířku). Díky tomu, že pomocí knihovny AntiGrain je možné vytvořit seznam spanů, se kterými lze pokračovat v renderování, je tato knihovna ideální jako základ pro další grafické knihovny. Pipeline se v knihovně AntiGrain označuje jeden krok v rasterizačním procesu (může to být převod křivek na úsečky, transformace, ...).

2.3.1 Možnosti knihovny AntiGrain

AntiGrain je knihovna prakticky neomezených možností, která se stala základem mnoha aplikací a renderovacích enginů (např. Gnash, KDE-SVG, ...). Aplikační rozhraní knihovny AntiGrain je definováno jako soubor šablon a programovacím stylem velmi připomíná standardní C++ knihovnu. Knihovna AntiGrain tedy nedefinuje API jako ostatní knihovny, ale ponechává využití a kombinaci tříd na aplikaci využívající právě AntiGrain.

Pomocí knihovny AntiGrain je možné stejně jako pomocí knihovny Cairo rasterizovat uzavřené i otevřené grafické cesty (otevřené se musí nejdříve uzavřít pomocí kresby tahem), generovat vzory a aplikovat afinní transformace. Seznam je uveden v podkapitole 2.3.3.

2.3.2 Ukázka použití knihovny AntiGrain

Jak už bylo řečeno v úvodu, knihovna AntiGrain je založená na C++ šablonách. následující malá ukázka ukazuje její použití a vykreslení obdélníku:

```
// Připravíme si typy šablon.
typedef agg::pixfmt_argb32 pixfmt;
typedef agg::renderer_base<pixfmt> renderer_base;
typedef agg::renderer_scanline_aa_solid<renderer_base> renderer_aa;

// Vytoříme rasterizační instance (do paměťového bufferu).
pixfmt pixf(buffer);
renderer_base rb(pixf);
renderer_aa ren_aa(rb);

// Přidáme obdélník do grafické cesty.
agg::path_storage path;
path.move_to(50.0, 50.0);
path.line_to(150.0, 50.0);
path.line_to(150.0, 150.0);
path.line_to(50.0, 150.0);
path.close_polygon();

// Nastavíme barvu na bílou.
ren_aa.color(agg::rgba(1.0, 1.0, 1.0, 1.0));

// Přidáme cestu do rasterizeru.
m_ras.add_path(path);
// Vyrenderujeme do paměťového bufferu.
agg::render_scanlines(m_ras, m_sl_p8, ren_aa);
```

Na první pohled je patrné, že architektura knihovny AntiGrain je mnohem složitější a její použití vyžaduje mnohem vyšší úroveň znalosti jazyka C++ (zejména šablon).

2.3.3 Souhrn možností knihovny AntiGrain

- vykreslení a výplň jednoduchých objektů a grafických cest (path),
- vykreslení písma (nutná podpora WinAPI nebo FreeType knihovny)
- vykreslení s použitím vzorů (různé typy gradientů, rastrový obrázek),
- nastavit gamma funkci,
- možnost aplikovat maticové transformace na vykreslovaný objekt nebo vzor (rotace, zvětšení, zmenšení, posuv),
- možnost poskládat vlastní pipeline pro rasterizaci objektu pomocí C++ šablon (součástí rasterizace může být převod křivek na přímky, transformace, pomocí

šablon jakákoliv operace). Cílem je vyhodit výpočty operací, které nejsou pro konkrétní objekt potřeba,

- rozmazání (blur) rasterizovaného objektu,
- a mnoho dalších funkcí, které nejsou dostupné v jiných knihovnách.

2.4 Souhrn

Popsané grafické knihovny vznikly každá s jinými požadavky a původně pro rozdílné operační systémy. V operačním systému Windows se hodně používá knihovna GDI+ dodávaná přímo s operačním systémem Windows XP (a vyšší), a stala se výchozí knihovnou pro vektorovou grafiku v prostředí .NET. V operačních systémech Linux a BSD se použití grafických knihoven liší podle použitého grafického toolkitu a podle zkušeností nebo preferencí programátora (žádná standardní grafická knihovna neexistuje). Použití knihovny Cairo je tedy jen jedna z možností, jak pracovat s vektorovou grafikou.

Následující tabulka (*Tab. 3*) shrnuje licence, otevřenost zdrojových kódů a podporované operační systémy zmíněných knihoven.

Tab. 3: Srovnání licencí, otevřenost kódu a podporované operační systémy grafických knihoven Cairo, GDI+ a Antigrain.

Název knihovny	Licence	Otevřený kód	Podporované operační systémy
Cairo	LGPL	Ano	Windows, Linux, BSD, Mac OS X
GDI+	Commercial, Redistributable	Ne	Windows 2000 a novější
AntiGrain	Public Domain	Ano	Windows, Linux, BSD, Mac OS X

Pro výběr knihovny je také důležité srovnání rychlosti, které bylo provedeno a je popsáno v praktické části diplomové práce (kapitola 6), kde jsou srovnány knihovny GDI+, Cairo a navržená knihovna Fog.

2.5 Prostor pro nové knihovny, důvody pro návrh knihovny Fog

Zmíněné grafické knihovny GDI+ a Cairo patří mezi nejpoužívanější v operačních systémech Windows a Linux / BSD. Knihovna GDI+ je prakticky standard v počítačové grafice pod Windows a použití v prostředí .NET ji zaručilo velmi dlouhou životnost a podporu. Na druhé straně knihovna Cairo se snaží stát se standardem ve světě operačních systémů Linux / BSD, ale nejedná se o jediné řešení, a této knihovně je velmi často vytýkaná právě rychlost (viz test rychlosti v kapitole 6.1).

Dalo by se říct, že pro operační systém Linux a BSD neexistuje knihovna, která by byla po výkonnostní stránce na velmi vysoké úrovni, ale namísto toho vznikly pomalejší knihovny, které ale problém vektorové grafiky řeší. Snahou tedy je napsat knihovnu, která by byla schopná pomocí použití jakýchkoliv programovacích a platformě závislých postupů a docílit tak maximálního výkonu při vykreslování rastrové a vektorově orientované grafiky.

Smyslem práce tedy nebylo jen navrhnout a implementovat grafickou knihovnu, ale i experimentovat s použitím různých optimalizačních technik k docílení co nejlepšího výkonu. Protože knihovna AntiGrain obsahuje implementaci všech vektorových algoritmů potřebných pro renderovací jádro, byla zvolena jako kompletní základ pro knihovnu Fog. Koncept šablon zcela vyhovuje filozofii knihovny Fog využít jen velmi malou část knihovny AntiGrain (rasterizer) a na vše ostatní napojit vlastní renderovací funkce.

3 CÍL PRÁCE

Cílem diplomové práce s názvem Vícevláknová multiplatformní grafická knihovna je:

- v teoretické části popsat rasterizaci geometrických objektů a možné optimalizace SW algoritmů pro její urychlení,
- zhodnotit současný stav a schopnosti grafických subsystémů současných nejrozšířenějších operačních systémů,
- v praktické části navrhnout multiplatformní grafickou knihovnu, která bude schopná využít více vláken a rozšíření CPU pro vykreslování grafických objektů,
- navrhnout a implementovat základní objekty grafické knihovny (písmo, obrázek, region, grafický kontext, framework pro načítání / ukládání obrázků),
- navrhnout a implementovat funkce pro operace s rastrem na nejnižší vrstvě,
- navrhnout a implementovat optimalizace vykreslení gradientu pomocí MMX a SSE2,
- srovnat rychlost renderování a možnosti navržené softwarové knihovny s jinými podobnými knihovnami,
- navrhnout metodiku integrace grafické knihovny s dalšími grafickými knihovnami a GUI toolkity pracujícími na vyšší úrovni,
- vytvořit programovou dokumentaci grafické knihovny.

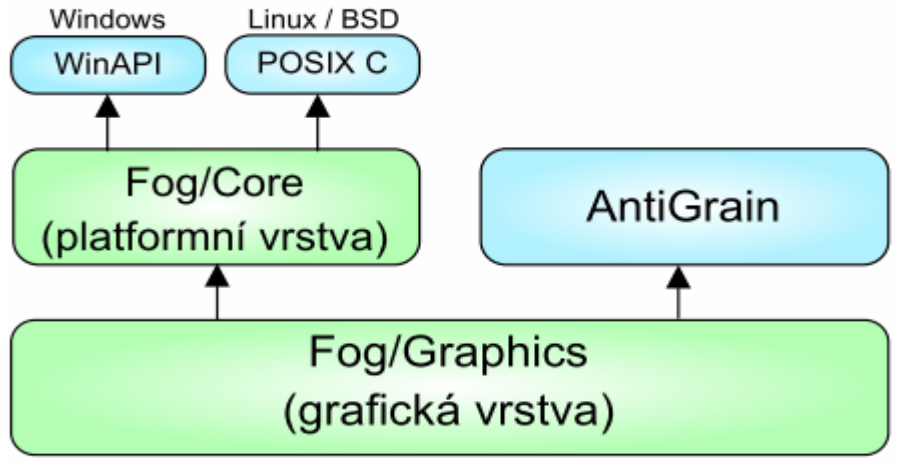
II. PRAKTICKÁ ČÁST

4 NÁVRH GRAFICKÉ KNIHOVNY FOG

Tato část se zabývá návrhem grafické knihovny nazvané Fog – Fair Object Graphics. Grafická knihovna Fog je svým návrhem odlišná od ostatních knihoven tím, že umožňuje rasterizaci grafických objektů ve více vláknech za účelem zvýšení výkonu. Koncept vícevláknové knihovny musí umožnit efektivní sdílení dat mezi vlákny tak, aby nedocházelo k jejich zbytečnému kopírování nebo nadbytečnému používání synchronizačních prostředků (například zámky).

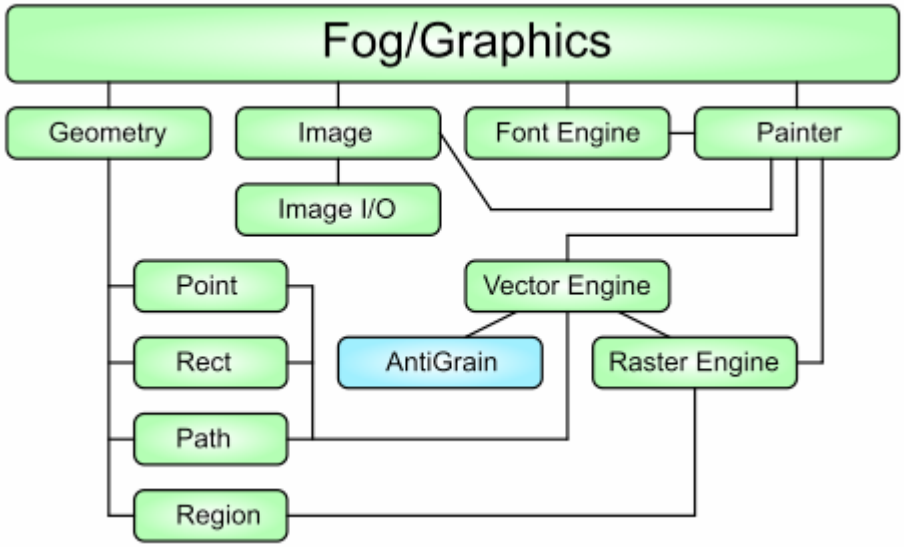
4.1 Schéma navržené knihovny

Knihovna byla rozvržena do několika vrstev, jejichž vztahy ilustruje obrázek (*Obr. 5*). Schéma popisuje vrstvy knihovny Fog (znázorněné zeleně) a jejich závislosti na ostatních knihovnách (znázorněné modře). Jednotlivé vrstvy knihovny Fog můžou být chápány jako samostatné knihovny (ve zdrojovém kódu jsou oddělené).



Obr. 5: Schéma navržené knihovny Fog.

Obr. 5 je jen obecné schéma návrhu knihovny Fog. Podrobné schéma včetně všech subsystémů knihovny Fog/Graphics ilustruje obrázek (*Obr. 6*), ve kterém je možné vidět jednotlivé moduly a jejich propojení v grafické části knihovny.



Obr. 6: Detailní schéma knihovny Fog/Graphics.

Na obrázku (*Obr. 6*) můžeme vidět logické uspořádání knihovny Fog/Graphics a vazby mezi jednotlivými částmi. Kreslení probíhá pomocí třídy Painter, do které se serializují grafické operace. V závislosti na typu operace se použije vektorový nebo rastrový engine. Vektorový engine funguje tak, že pomocí knihovny AntiGrain rasterizuje požadovaný geometrický objektu a výsledek rasterizace se pošle do rastrového engine.

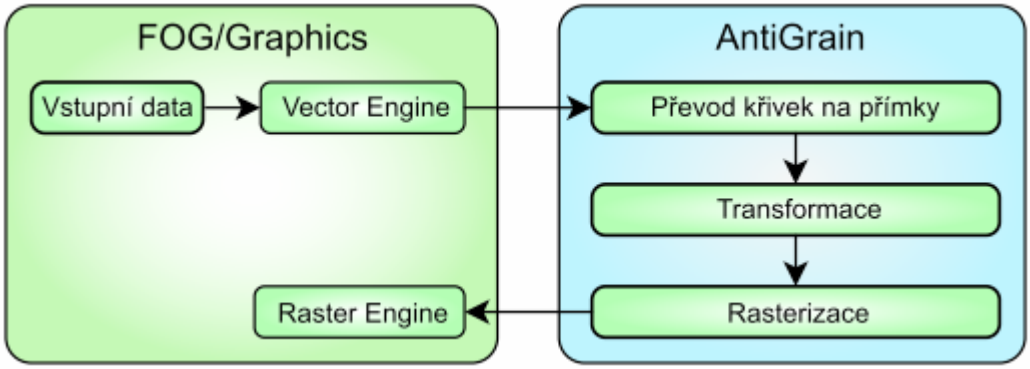
4.2 Vektorový engine

O vektorový engine knihovny Fog/Graphics se stará knihovna AntiGrain, protože svým konceptem umožňuje použít jen ty části, které jsou opravdu využitelné knihovnou Fog - rasterizer a transformace. Knihovna AntiGrain je ale použita jen interně a před programátorem využívající knihovnu Fog je skryta. Všechny třídy, které umožňují vektorovou reprezentaci geometrických objektů (například cesty, čtverce, přímky) jsou implementované v knihovně Fog bez vazeb na AntiGrain a při rasterizaci se mapují pomocí šablon na třídy kompatibilní s rozhraním AntiGrain.

Renderování vektorové grafiky probíhá v několika fázích:

- převod křivek na přímky,
- transformace,
- rasterizace.

Průběh renderování vektorového geometrického objektu je zobrazen na obrázku (*Obr. 7*).



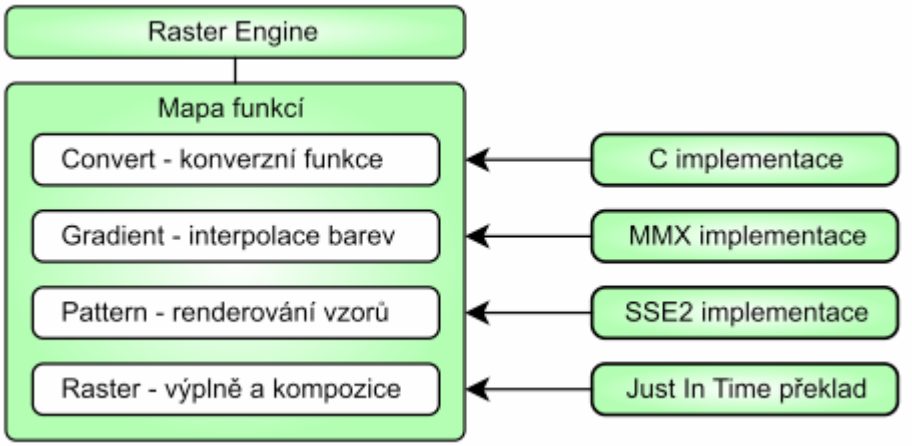
Obr. 7: Průběh renderování geometrického objektu.

Vektorový engine obsahuje i optimalizace, pomocí kterých je možné obejít některé kroky při renderování geometrického objektu (například transformace) nebo kompletně obejít AntiGrain (renderování jednoduchých výplní, textur, obrázků a písma).

4.3 Rastrový engine

Rastrový engine se stará o nízkoúrovňovou manipulaci s pixely. Vstupem pro funkce v rastrovém enginu jsou vždy ukazatele na grafické data a proměnné reprezentující jejich délku. Vstupem a výstupem rastrového enginu jsou tedy čistě pixelové data a jejich rozměry.

Protože rastrový engine hodně pracuje s pamětí a jeho výsledkem jsou často kompozitní operace mezi pixely, vyplatí se optimalizace využívající technologie MMX a SSE2. Z implementačního hlediska se jedná o velkou mapu funkcí, která je při inicializaci knihovny nastavená na výchozí funkce, které jsou napsané pouze v C++ jazyce. Při inicializaci knihovny je provedena detekce MMX a SSE2 rozšíření a v případě úspěchu se některé funkce přenastaví tak, aby tyto technologie využily.



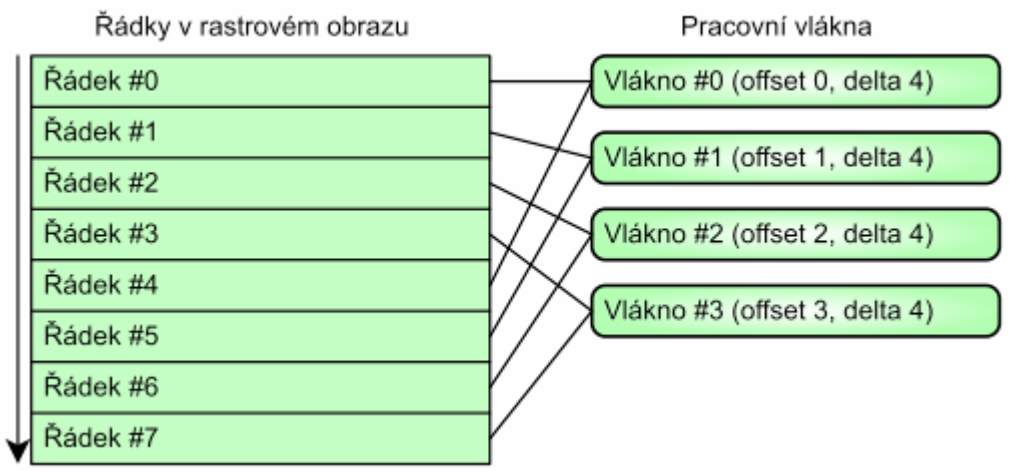
Obr. 8: Mapa funkcí v knihovně Fog.

Díky mapě funkcí je možné, aby každá část knihovny využila ty nejlepší implementace rastrových funkcí bez jediného řádku kódu na detekci CPU uvnitř grafického kontextu.

4.4 Vícevláknová architektura

Knihovna Fog umožňuje vykreslení geometrických objektů paralelně s využitím více vláken. Na procesorech, které obsahují více jader se může díky paralelnímu zpracování zvýšit rychlost vykreslení grafické scény. Při návrhu vícevláknové architektury byl navrhnut velice jednoduchý algoritmus, který je založený na přeskakováním řádků (scanline skipping). Každé vlákno má nastavený posun (offset) a delta hodnotu. Posun udává první pracovní řádek vlákna a delta udává, o kolik se posunout v případě, že byl právě dokončen současný řádek.

Logicky vyplývá, že posun musí mít každé vlákno různé (musí být provedeny operace na každém řádku a vlákna si nesmí vzájemně překážet) a nemůže se rovnat nebo přesáhnout hodnotu delta. Delta je v našem případě celkový počet vláken, který by měl odpovídat celkovému počtu jader v procesoru, přesněji součet všech jader všech procesorů. Delta hodnota nesmí být nikdy nulová a nastavení na 1 nemá smysl (s výjimkou testování). Ilustrace vláken a jejich pracovních řádků pro čtyřjádrový procesor je zobrazena na obrázku (*Obr. 9*).



Obr. 9: Rozdělení pracovních vláken pro čtyřjádrový procesor.

Vícevláknovou architekturou se zabývá i kapitola 5.4, kde je popsaný způsob implementace.

5 IMPLEMENTACE GRAFICKÉ KNIHOVNY FOG

Tato kapitola rozebírá třídy v knihovně Fog a jejich použití včetně krátkých příkladů a visuálních výstupů. Všechny demonstrační obrázky jsou vykresleny knihovnou Fog.

5.1 Platformní vrstva

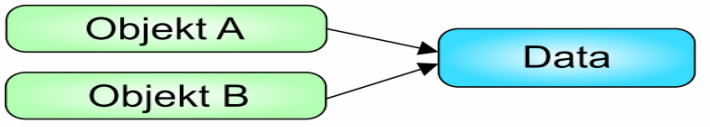
Platformní vrstva má za úkol sjednotit rozhraní pro volání funkcí, které jsou závislé a cílovém operačním systému. Základem jsou abstrakce, které umožňují psát jeden kód, který bude podporovat více operačních systémů, překladačů a architektur.

5.1.1 Atomické operace a copy-on-write

Atomická operace je nedělitelná operace, která musí proběhnout celá nebo nesmí proběhnout vůbec. Každá hardwarová architektura, která umožňuje běh více procesů (a tedy i vláken) obsahuje na platformě závislé postupy, pomocí kterých lze atomičnosti docílit.

Aby bylo možné používat atomické operace, je nutné použít specifické funkce konkrétního překladače C/C++ jazyka nebo inline assembler (z důvodu rychlosti). Byly použity obě varianty podle cílové architektury, překladače a operačního systému.

Korektní zvládnutí implementace atomických operací umožňuje implicitní sdílení dat spolu s technikou copy-on-write. Implicitním sdílením dat se rozumí sdílení paměťového prostoru dvou nebo více objektů, které vystupují zcela samostatně. Ke sdílení dat dojde v případě, že obsah jednoho objektu chceme zkopírovat do jiného, ale místo fyzického zkopírování se jen přičte počítadlo (reference count). Pokud v budoucnu dojde na modifikaci sdíleného objektu, vytvoří se nejdřív kopie, která se následně modifikuje (copy-on-write).



Obr. 10: Implicitní sdílení dat mezi objekty.

Spojení implicitního sdílení a atomických operací dovoluje sdílet i paměť objektů, ke kterým se přistupuje z různých vláken, bez nutnosti používat synchronizační prostředky. Z pohledu vícevláknové knihovny se jedná o důležitou optimalizaci, která zabraňuje

zbytečnému kopírování dat pro pracující vlákna, a zbytečnému zamykání objektů, ke kterým se přistupuje jen pro čtení.

Atomický typ je implementovaný ve třídě *Fog::Atomic*. Jedná se o šablonu, ve které je možné zvolit typ hodnoty (např. *Fog::Atomic<int>*).

5.1.2 Základní třídy a kontejnery

Platformní vrstva obsahuje základní třídy a kontejnery pro programování v C++, které jsou používané v celé knihovně Fog. Důvod pro naprogramování těchto tříd byl ten, že základní knihovny pro C/C++ neobsahují podporu pro unicode, který je velmi potřeba v grafické knihovně, ve které chceme pracovat i s textem. Další důvod byl ten, aby nebyla knihovna závislá na základní C++ knihovně, která používá výjimky a RTTI (knihovna Fog tyto vlastnosti jazyka C++ nevyužívá). Jako poslední je potřeba zmínit, že základní třídy a kontejnery používají implicitní sdílení dat a techniku copy-on-write.

Mezi základní třídy a kontejnery patří třídy uvedené v tabulce (*Tab. 4*).

Tab. 4: Základní třídy a kontejnery.

Třída	Popis
Fog::Char	Třída pro abstrakci jednoho znaku v 8bitové, 16bitové a 32bitové formě.
Fog::Delegate	Implementace delegátů v C++. Jedná se o převzatý kód dostupný na portálu codeproject.com jako public domain [2].
Fog::Hash	Implementace hash tabulky, která používá implicitní sdílení a copy-on-write.
Fog::List	Implementace pole dat, které umožňuje přidávat / odebírat položky ze začátku a konce v konstantním čase.
Fog::Lazy	Šablona, která umožňuje vytvářet zpožděné instance tříd až při prvním přístupu (používají se atomické operace).
Fog::Locale	Třída, která obsahuje preferované znaky pro převod čísel na řetězce a zpět.
Fog::Memory	Statická třída, která obsahuje funkce pro správu a práci s pamětí. Obsahuje i optimalizované funkce jako například bswap.

Fog::OS	Statická třída, která umožňuje získat základní informace o operačním systému (jméno, verze, ...).
Fog::String	Implementace řetězce v 8bitové, 16bitové a 32bitové formě. Interně se v celé knihovně používají 32bitové řetězce v unicode kódování, ale někdy je potřeba řetězce převést do 8bitové nebo 16bitové formy a zpět.
Fog::TextCodec	Implementace převodu řetězců z jednoho kódování do jiného.
Fog::Time	Třída pro práci s časem, obsahuje pomocné třídy Fog::TimeTicks a Fog::TimeDelta.
Fog::UserInfo	Statická třída, která umožňuje získat základní informace o uživateli.
Fog::Value	Třída, která v sobě umožňuje uchovat různé typy hodnot (řetězec, celé číslo, desetinné číslo, null, ...)
Fog::Vector	Implementace pole dat, které umožňuje přidávat / odebírat položky pouze z konce seznamu v konstantním čase.

Zmíněné třídy v mnoha případech zcela nahrazují standardní C++ knihovnu, ale slouží hlavně jako multiplatformní základ pro celou knihovnu. Následuje pár jednoduchých příkladů, které demonstrují použití a smysl zmíněných tříd:

```
// Použití třídy String a TextCodec na konverzi řetězce ze systémového
// kódování do unicode, které se používá uvnitř knihovny Fog.
Fog::String32 string; // Cílový řetězec v unicode
const char* input;    // Vstupní řetězec
err_t error;          // Chybový stav

error = Fog::TextCodec::local8().toUtf32(output, input)
if (error) {
  // Ošetření chyby.
}

// Převod řetězce 'string' do kódování UTF-8.
Fog::String8 utf8;
error = Fog::TextCodec::utf8().fromUtf32(utf8, string);
if (error) {
  // Ošetření chyby.
}

// Použití obecného typu Fog::Value, vytvoření Value z ASCII C řetězce.
Fog::Value val = Fog::Value::fromString(Fog::StubAscii8("128"));

// Typ hodnoty lze zjisti pomocí metody type().
if (val.type() == Fog::Value::TypeString) {}

// Je ale možné vyžádat si hodnotu jako úplně jiný typ.
int32_t valueAsInt32;
error = val.toInt32(&valueAsInt32);
if (error) {
  // Ošetření chyby.
}
```

Z příkladů je možné vidět, že knihovna je postavena na chybových stavech a jakýkoliv neúspěch volání funkcí je možné ihned zjistit. Koncept chybových stavů místo výjimek a RTTI má řadu výhod, z nichž stojí za zmínku menší velikost knihovny (v případě, že překládáme bez podpory výjimek a RTTI) a možné použití knihovny i v projektech, které tyto vlastnosti také nepoužívají. Nezávislost na standardní C++ knihovně má stejný význam.

5.1.3 Vlákna, synchronizace a cyklus událostí

Platformní část obsahuje i vrstvu, která se stará o vytváření vláken, synchronizačních prostředků a cyklus událostí (event loop). Programy využívající knihovnu Fog čistě na grafickou část tyto třídy nebudou nikdy potřebovat, ale programy, které by z knihovny chtěli použít mnohem víc, než jen grafickou část, jsou tyto třídy nachystané k použití.

Seznam tříd pro práci s vlákny a jejich synchronizaci je uveden v tabulce (*Tab. 5*).

Tab. 5: Seznam tříd pro práci s vlákny a jejich synchronizaci.

Třída	Popis
Fog::AutoLock	Automatické zamykání zámku v bloku C++ kódu.
Fog::AutoUnlock	Automatické odemykání zámku v bloku kódu.
Fog::Lock	Zámek, slouží pro zamykání konkrétního úseku kódu proti provedení jiným vláknem
Fog::Thread	Třída, pomocí které lze jednoduše vytvářet vlákna.
Fog::ThreadCondition, *Fog::ThreadEvent*	Synchronizační prostředky, umožňují posílat signál nebo čekat na událost poslanou z jiného vlákna.
Fog::ThreadLocalStorage	Implementace lokálních dat pro každé vlákno.
Fog::ThreadPool	Zásobník vláken, minimalizuje režii na vytváření a uvolňování vláken pro krátkodobou činnost.

Jména tříd odpovídají jejich zažitým termínům nebo se dají velice snadno odvodit. Pro vytvoření vlákna se používá třída *Fog::Thread*. Pro synchronizaci přístupu k proměnným se používá *Fog::Lock* spolu s pomocnými třídami *Fog::AutoLock*

a *Fog:AutoUnlock*. Pro synchronizaci více vláken se používá i třída *Fog::ThreadCondition*, která se chová podobně jako objekt *pthread_cond_t*, který je součástí POSIX specifikace a umožňuje synchronizaci vláken na základě hodnoty proměnné [1].

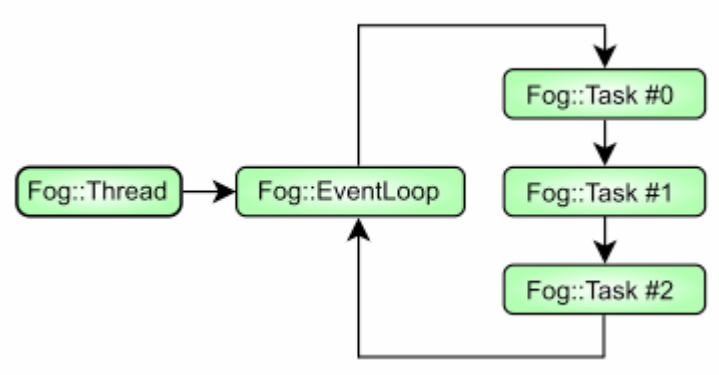
Knihovna Fog implementuje i cyklus událostí, do kterého lze zasílat úkoly a zprávy. Cyklus událostí se může použít jako hlavní cyklus událostí aplikace nebo jako lokální cyklus pro vyřizování úkolů v pracujících vláknech.

Seznam tříd souvisejících s cyklem událostí je uveden v tabulce (*Tab. 6*).

Tab. 6: Seznam tříd definující cyklus událostí.

Třída	Popis
Fog::Application	Třída aplikace, obsahuje hlavní cyklus událostí a metody ke spuštění a ukončení tohoto cyklu.
Fog::Event	Hlavní třída implementující událost, může být zaslána jen třídě, která vychází z Fog::Object.
Fog::EventLoop	Cyklus událostí, jen jeden pro každé vlákno.
Fog::EventPump	Implementační detail konkrétního cyklu událostí (existuje obecná implementace a implementace pro Windows a X11)
Fog::Task, *Fog::CancelableTask*	Zpráva, kterou je možné zaslat na zpracování do hlavního nebo lokálního cyklu událostí.
Fog::Timer	Časovač, který vyvolává události v závislosti na nastaveném intervalu.

V některých multiplatformních knihovnách se implementace vláken řeší způsobem reimplementace virtuálních metod (většinou mají název *run* nebo *main*). Knihovna Fog tento přístup podporuje také (reimplementace metody *Fog::Thread::main*), ale je doporučeno použít lokální cyklus událostí a vláknu poslat úkol (*Fog::Task*). Proces zpracování úkolů vláknem pomocí nekonečného cyklu událostí je zachycen na obrázku (*Obr. 11*).



Obr. 11: Cyklus zpracování úkolů ve vlákně.

5.1.4 Objektová vrstva

Knihovna Fog obsahuje i základní implementaci objektu, který může sloužit jako hlavní třída pro vytvoření aplikačních komponent. *Fog::Object* je třída, která implementuje základní metody pro práci s objekty a vazeb mezi nimi. Třída *Fog::Object* se používá jako posluchač (listener) nebo generátor (emitter) událostí, které mohou být zaslány přímo prostřednictvím metody *sendEvent()* nebo prostřednictvím cyklu událostí.

Základní objekt obsahuje i vlastní implementaci RTTI, která není závislá na implementaci v C++ a umožňuje například přetypovat objekt jen na základě jména v řetězci. Možnosti demonstruje následující malý příklad, ve kterém je vstup do funkce jen *Fog::Object* a pomocí jeho metod můžeme zjistit konkrétní typ:

```
void func(Fog::Object* obj)
{
  // Zjistí, jestli je obj typ Fog::Timer.
  if (obj->isClassOf("Fog::Timer"))
  {
  }

  // Zjistit typ je možné i pomocí šablon.
  Fog::Timer* timer = fog_object_cast<Fog::Timer*>(obj);
  if (timer)
  {
  }
}
```

5.1.5 Práce se soubory a I/O

Jedna z posledních věcí, kterou je potřeba představit v platformní vrstvě je práce se souborovým systémem. Knihovna Fog umožňuje vytvářet datové proudy, procházet adresáře a testovat existenci souborů.

Implementace datových proudů je ve třídě *Fog::Stream*, umožňuje otevřít regulární soubor na souborovém systému (pomoci platformě závislého API) a zapisovat nebo číst do úseku

v paměti. Knihovna Fog obsahuje i doplňkové třídy *Fog::FileSystem*, pomocí které je možné testovat existenci souborů a adresářů včetně jejich práv a třídu *Fog::DirIterator*, která slouží pro procházení adresářů.

Efektivní práce se soubory a souborovým systémem byla potřeba pro načítání / ukládání obrázků v grafické vrstvě a správu písma pod operačním systémem Linux.

5.2 Grafická vrstva

Grafická vrstva je funkční celek, pomocí kterého je možné pracovat s obrázky, písmem a vykreslovat geometrické objekty. Jedná se o soubor tříd, které reprezentují geometrické objekty na rastrové nebo vektorové úrovni. Seznam veřejných tříd v grafické vrstvě je uveden v tabulce (*Tab. 7*).

Tab. 7: Třídy v grafické vrstvě.

Třída	Popis
Fog::AffineMatrix	2D matice pro afinní transformace.
Fog::DitherMatrix	Matice, která se používá při snížení barevné hloubky obrázku (dithering).
Fog::Font, *Fog::FontFace*	Třídy související s písmem. Fog::Font je hlavní třída pro nastavení písma a souvisejících parametrů (velikost, kurzíva, ...).
Fog::Glyph, *Fog::GlyphCache*, *Fog::GlyphSet*	Třídy související s uchováním písma v bitmapovém formátu. Tyto třídy slouží zejména k urychlení renderování písma bez jakýchkoliv transformací.
Fog::GradientStop, *Fog::GradientStops*	Třídy reprezentující seznam barev, pomocí kterých se vykreslují barevné přechody (kompatibilní s SVG specifikací [14]).
Fog::Point[F], *Fog::Rect[F]*, *Fog::Region*	Geometrické třídy pro rastrovou i vektorovou grafiku. Třídy, které končí sufixem F používají reálná čísla.
Fog::Image, *Fog::ImageIO*	Rastrový obrázek a třídy související s načítáním a ukládáním rastrových obrázků.

Fog::Painter	Vektorový grafický kontext.
Fog::Palette	Paleta barev, která se používá pro indexované formáty obrázků.
Fog::Pattern	Vzor, který slouží pro výplně nebo kreslení ve vektorovém kontextu.
Fog::Raster	Implementace rastrových operací.
Fog::Region	Pole YX setříděných obdélníků.
Fog::Rgba	Barva definovaná jako intenzita jednotlivých složek RGB a alfa kanál.

5.2.1 Geometrické objekty (Fog::Point, Fog::Rect, Fog::Path)

Knihovna Fog obsahuje třídy, které implementují základní i komplexní geometrické objekty. Mezi základní objekty patří bod (*Fog::Point*), obdélník (*Fog::Rect*) a region (*Fog::Region*). Mezi komplexní objekty patří grafické cesta (*Fog::Path*). Základní geometrické objekty jsou jen struktury pro uchování hodnot pohromadě, třída *Fog::Point* obsahuje souřadnice *X* a *Y*, třída *Fog::Rect* (obdélník) obsahuje počáteční souřadnice X_0, Y_0, šířku (width) a výšku (height). V knihovně Fog se rozlišují rastrové geometrické objekty a vektorové grafické objekty.

Seznam rastrových geometrických objektů:

- *Fog::Point* – bod (*X*, *Y*),
- *Fog::Rect* – obdélník (X_0, Y_0, *W*, *H*),
- *Fog::Region* – setříděný seznam obdélníků.

Seznam vektorových geometrických objektů:

- *Fog::PointF* – bod (*X*, *Y*),
- *Fog::RectF* – obdélník (X_0, Y_0, *W*, *H*),
- *Fog::Path* – grafická cesta definovaná jako pole úseček a křivek.

Použití geometrických objektů je spojeno s vektorovým grafickým kontextem popsaném v kapitole 5.2.3.

5.2.2 Obrázek (Fog::Image)

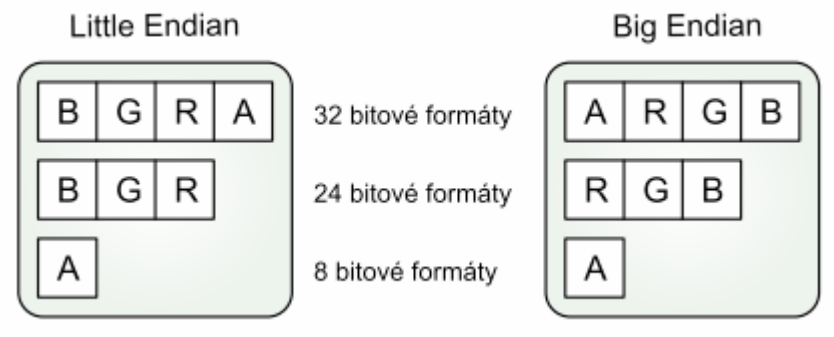
Obrázek je jeden ze základních rastrových objektů v počítačové grafice. V knihovně Fog jsou obrázky implementované jako dvojrozměrné pole pixelů, které je uložené v souvislém úseku paměti. Obrázek reprezentuje třída *Fog::Image*, pomocí které je možné vytvářet obrázky a provádět nad nimi základní rastrové operace. Data třídy *Fog::Image* jsou implicitně sdílené a používají techniku copy-on-write.

5.2.2.1 Formáty pixelů a jejich uložení v třídě Fog::Image

Existuje mnoho způsobů uložení jednotlivých pixelů v paměti a třída *Fog::Image* jich podporuje celkem 6:

- ARGB32 – 32bitový RGB formát včetně alfa kanálu,
- PRGB32 – 32bitový RGB formát včetně alfa kanálu, hodnoty složek R, G a B jsou vynásobeny hodnotou alfa kanálu. Jedná se o základní formát pro kompozici pixelů,
- RGB32 – 32bitový RGB formát, který neobsahuje údaj o průhlednosti. 8 bitů v každém pixelu je nevyužitých,
- RGB24 – 24bitový RGB formát,
- A8 – 8bitový formát, který obsahuje jen hodnotu alfa kanálu (průhlednosti). Tento formát se používá k uložení písmen,
- I8 – 8bitový formát, kde hodnota pixelu odpovídá indexu v paletě barev. V minulosti hodně používaný formát v počítačové grafice (hlavně u her), dnes se téměř nepoužívá (je omezený na 256 barev).

Pořadí bytů v pixelu odpovídá pořadí bytů, se kterým pracuje procesor. Pro reprezentaci pixelu se v knihovně Fog používá datový typ *uint32_t* (32bitové neznaménkové celé číslo) a pomocí hexadecimálního zápisu je možné zapsat všechny složky pixelu. Hexadecimální zápis je v případě knihovny Fog platformě nezávislý a je definován ve tvaru *0xAARRGGBB*. Například plně sytá červená barva se zapíše jako *0xFFFF0000*. Uložení pixelu v paměti ve třídě *Fog::Image* je znázorněno na obrázku (*Obr. 12*).



Obr. 12: Uložení pixelu ve třídě Fog::Image.

Pořadí bytů Little Endian používají procesory architektury x86 a x86_64, Big Endian se používá v mobilních zařízeních (architektura ARM), ale i na desktopovém nebo serverovém zařízení (architektura PowerPC).

5.2.2.2 Vytváření obrázků a základní úpravy

Třída *Fog::Image* obsahuje metody, pomocí kterých lze vytvářet obrázky a provádět nad nimi základní grafické úpravy. Všechny funkce pracují čistě na rastrové úrovni a slouží spíše k předpřipravení dat pro vektorový grafický kontext.

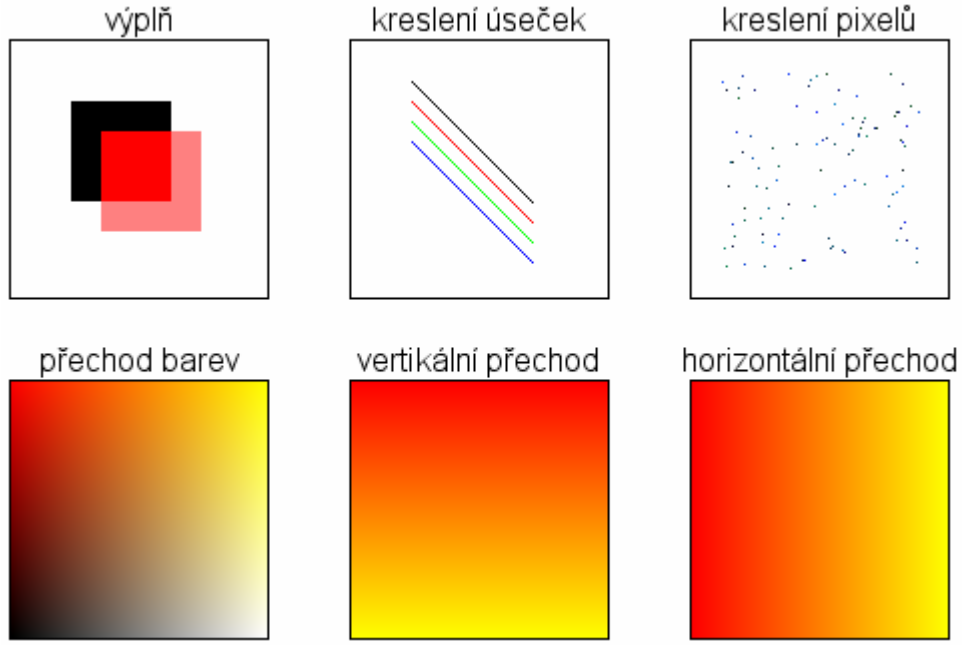
Obrázek lze vytvořit pomocí metody *create()* nebo pomocí jeho konstruktoru. Pokud nám nevyhovuje současný formát pixelů, tak pomocí metod *convert()*, *premultiply()*, *demultiply()* a *to8bit()* můžeme formát změnit.

Mezi základní operace patří:

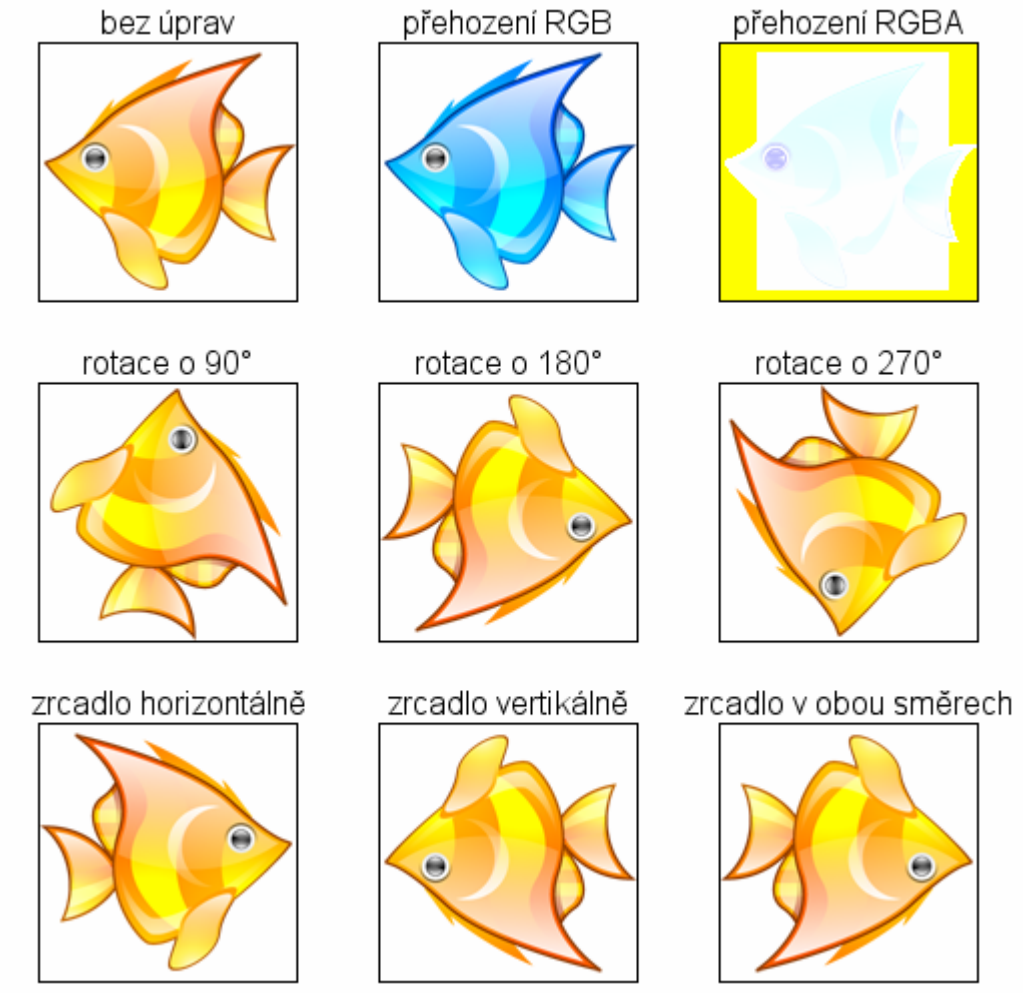
- vyplnit obrázek nebo obdélník barvou – *clear()*, *fillRect()*,
- nakreslit pixel – *drawPixel()*,
- nakreslit úsečku – *drawLine()*,
- výplně pomocí barevného přechodu – *fillQGradient()*, *fillHGradient()*, *fillVGradient()*.
- prohození barev – *swapRgb()*, *swapRgba()*,
- vynásobení / vydělení barevných složek alfa kanálem – *premultiply()*, *demultiply()*,
- inverze barev – *invert()*,
- rotace o 90°, 180° a 270° – *rotate()*,

- zrcadlení – *mirror()*.

Základní operace jsou znázorněné na obrázku (*Obr. 13*) a (*Obr. 14*).



Obr. 13: Základní operace s obrázkem (výplně, úsečky a pixely).



Obr. 14: Základní operace s obrázkem (barvy, rotace, zrcadlení).

5.2.2.3 Načítání a ukládání obrázků

I při vektorově orientované grafice jsou velice často potřebné data definované na rastrové úrovni. Knihovna Fog obsahuje framework pro načítání a ukládání obrázků více formátů:

- Windows/OS2 Bitmap (.bmp, .ras) – implementace v knihovně Fog,
- ZSoft Paintbrush (.pcx) – implementace v knihovně Fog,
- Graphics Interchange Format (.gif) – implementace v knihovně Fog,
- Portable Network Graphics (.png) – pomocí knihovny libpng,
- Joint Photographic Experts Group (.jpeg, .jfif) – pomocí knihovny libjpeg.

Pro načítání se v knihovně používá termín dekodér (Decoder) a pro ukládání enkodér (Encoder). Třída, které umožňuje vytvořit dekodér a enkodér pro konkrétní formát se nazývá Provider. Instance třídy Provider je vytvořena jen 1x pro každý formát

při inicializaci knihovny. Při načítání nebo ukládání obrázku se tato třída jednoduše vyhledá v seznamu dostupných providerů a použije pro vytvoření enkodéru nebo dekodéru.

Pro načítání obrázku slouží metody *readFile()*, *readStream()* a *readMemory()*, pro ukládání *writeFile()* a *writeStream()*.

Příklad načtení obrázku fish.bmp a uložení jako fish.png:

```
Fog::Image i;
err_t err;

err = i.readFile(Fog::StubAscii8("fish.bmp"));
if (err) {
  // Zpracování chyby.
}

err = i.writeFile(Fog::StubAscii8("fish.png"));
if (err) {
  // Zpracování chyby.
}
```

5.2.3 Vektorový grafický kontext (Fog::Painter)

Vektorový grafický kontext slouží k vykreslení vektorově orientované grafiky do cílového objektu. Knihovna Fog v současnosti podporuje jako cílový objekt pouze rastrový obrázek (*Fog::Image*).

5.2.3.1 Zahájení a ukončení kreslení

Vytvořením instance třídy *Fog::Painter* vznikne jen prázdná instance, která při zavolání jakékoliv metody neudělá vůbec nic. Aby bylo možné kreslit, je nejdříve nutné připravit si cílový obrázek, nastavit mu rozměry a formát. Třída *Fog::Painter* umí kreslit i do libovolného paměťového bufferu, je tedy možné kreslit i do úplně jiné třídy než *Fog::Image*.

Pro zahájení kreslení slouží metoda *begin()*, a pro ukončení metoda *end()*. Alternativně lze k zahájení kreslení použít přetížený konstruktor a předat tak stejné parametry jako pomocí metody *begin()*.

Ukázka připravení rastrového obrázku a nastavení grafického kontextu:

```
// Vytvoření cílového obrázku o rozměrech 320x200
Image image(320, 200, Image::FormatPRGB32);

Painter p;       // Vytvoření grafického kontextu.
p.begin(image);  // Začátek kreslení do instance 'image'.
// Kreslení ...
p.end();         // Konec kreslení
```

Alternativně lze použít konstruktor a metodu *end()* úplně vynechat, pokud končí platnost třídy Fog::*Painter* (konec bloku nebo zavoláním destruktoru):

```
// Vytvoření cílového obrázku o rozměrech 320x200
Image image(320, 200, Image::FormatPRGB32);
{
  Painter p(image);// Vytvoření grafického kontextu a zahájení kreslení.
  // Kreslení ...
} // Konec kreslení, destruktor Fog::Painter zavolá end() za nás.
```

V následujících příkladech bude instance *p* vždy chápána jako již zinicializovaný grafický kontext.

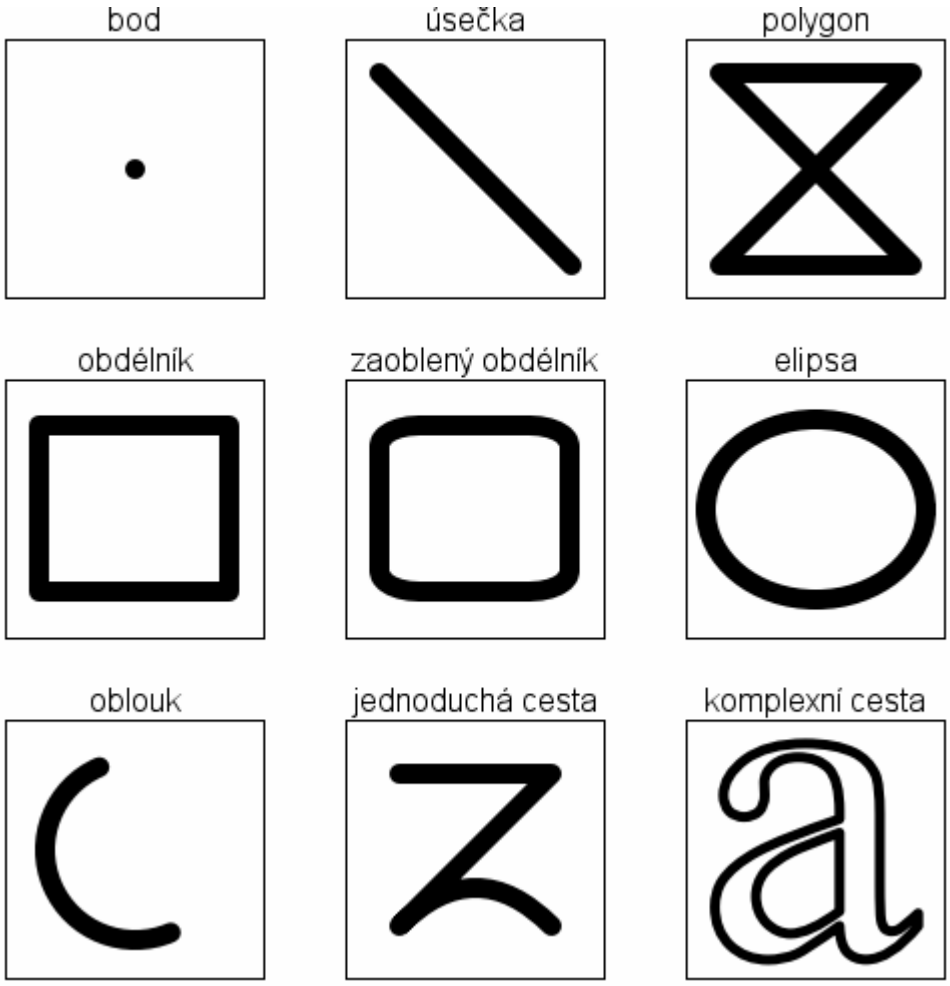
5.2.3.2 Kreslení tahem (stroke)

Kreslení tahem (stroke) patří mezi základní operace grafického kontextu. Výstup je kresba grafické cesty perem včetně aplikování různých stylů čáry a její tloušťky.

Podpora geometrických objektů:

- kresba bodu – *drawPoint()*,
- kresba úsečky – *drawLine()*,
- kresba polygonu – *drawPolygon()*,
- kresba obdélníku – *drawRect()*, *drawRects()*,
- kresba obdélníku se zaoblenými rohy – *drawRound()*,
- kresba elipsy – *drawEllipse()*,
- kresba oblouku – *drawArc()*,
- kresba libovolné grafické cesty – *drawPath()*.

Na obrázku (*Obr. 15*) je zobrazen výstup kreslení podporovaných geometrických objektů. Pomocí grafické cesty je možné nakreslit libovolně složitý grafický objekt a interně jsou všechny kreslící funkce implementovány tak, že se nejprve vytvoří grafická cesta pro požadovaný objekt a ten se poté serializuje na výstup.

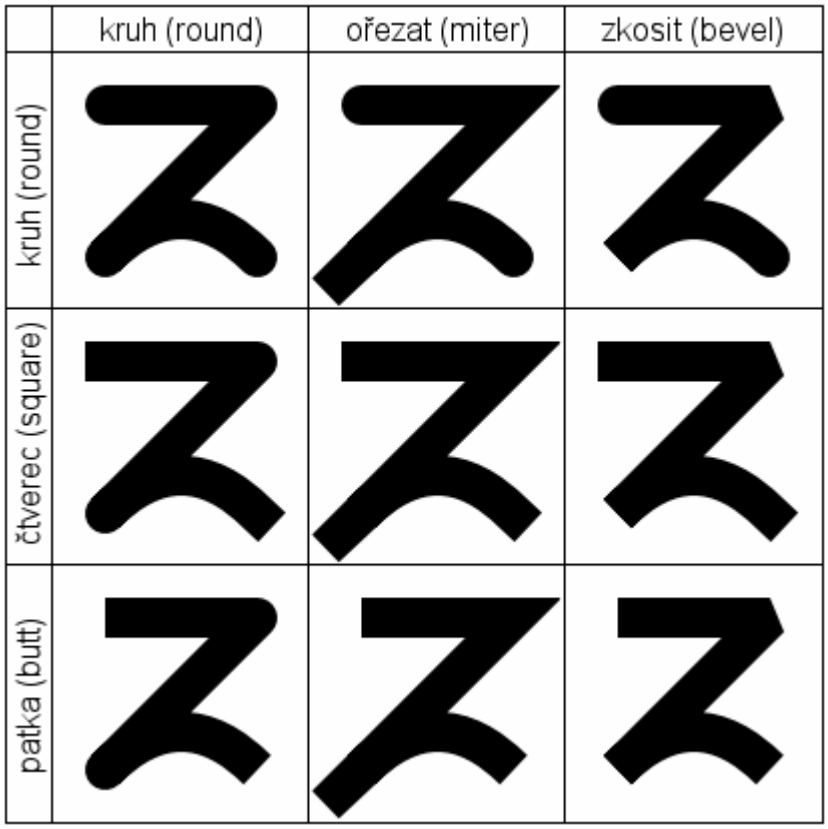


Obr. 15: Kreslení geometrických objektů tahem.

Volby při kreslení tahem:

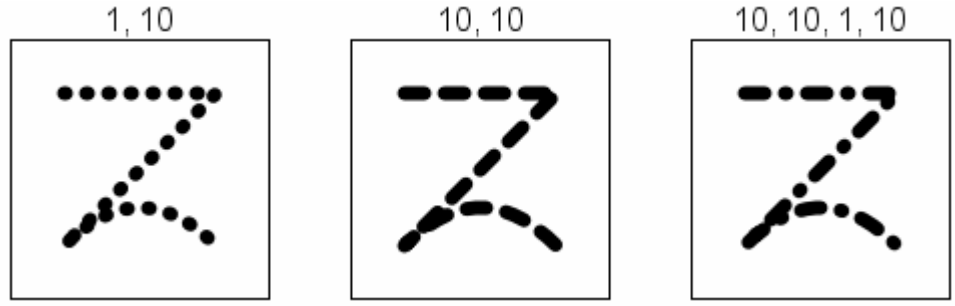
- nastavení tloušťky čáry – *setLineWidth()*,
- nastavení začátku čáry – *setLineCap()*,
- nastavení spojení čar – *setLineJoin()*,
- nastavení stylu čáry – *setLineDash()*.

Kreslení tahem včetně nastavení všech kombinací začátku a spojení čar je zobrazeno na obrázku (*Obr. 16*).



Obr. 16: Nastavení začátku (vertikální osa) a spojení (horizontální osa) čáry.

Knihovna Fog umožňuje nastavit styl čáry na přerušovaný, čerchovaný nebo libovolně nastavit délku přerušení a tahu. K nastavení délky přerušení a tahu slouží funkce *setLineDash()* a k nastavení délky mezi prvním tahem *setLineDashOffset()*. Na obrázku (*Obr. 17*) je ukázka různého nastavení tahu (první parametr) a přerušení (druhý parametr).



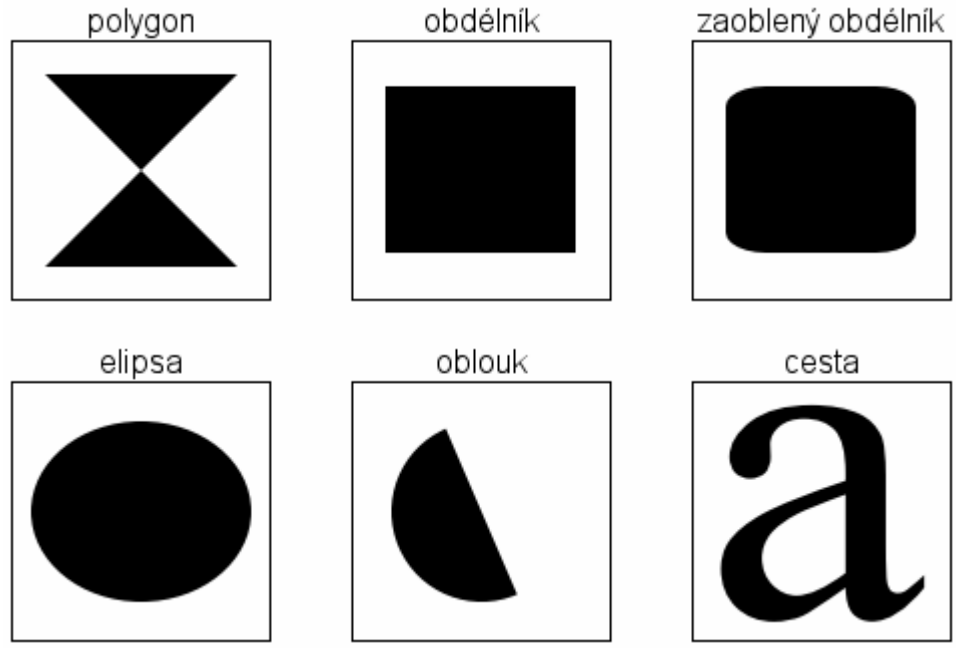
Obr. 17: Nastavení délky tahu a délky přerušení čáry.

5.2.3.3 Výplň (fill)

Výplň je nejdůležitější operace v grafické knihovně Fog. Obecně lze nahlížet na jakékoliv kreslení grafických cest jako na výplň uzavřené oblasti. Výplň je i z hlediska implementace jednodušší část, protože se na ni nevztahují nastavení stylu a tloušťky čáry.

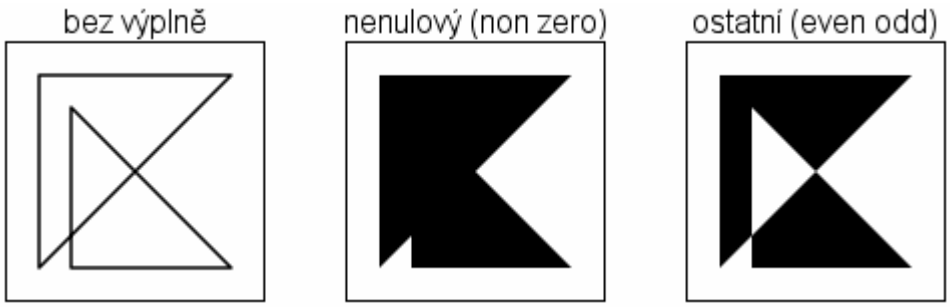
Výplň lze aplikovat na podobné množství objektů jako u kreslení tahem (*Obr. 18*):

- výplň polygonu – *fillPolygon()*,
- výplň obdélníku – *fillRect()*, *fillRects()*,
- výplň obdélníku se zaoblenými rohy – *fillRound()*,
- výplň elipsy – *fillEllipse()*,
- výplň oblouku – *fillArc()*,
- výplň libovolné uzavřené grafické cesty – *drawPath()*.



Obr. 18: Výplň geometrických objektů.

Kreslení lze ovlivnit pouze nastavením módu výplně pomocí metody *setFillMode()* na nenulové (non zero) a ostatní (even odd). Rozdíl módu výplně je možné zaznamenat jen při překrývání grafických cest a je zobrazen na obrázku (*Obr. 19*).



Obr. 19: Rozdíl mezi módem výplně.

5.2.3.4 Aplikování vzorů

Knihovna Fog umožňuje kromě kreslení tahem a vyplněním oblasti stejnou barvou i nastavit vzor (pattern). Vzor lze nastavit pomocí metody *setPattern()* a jako parametr se předává instance třídy *Fog::Rgba* (nastavení barvy) nebo *Fog::Pattern* (nastavení vzoru). Vzor je aplikován na kreslení tahem i výplně (nerozlišuje se tedy mezi perem a štětcem a tato terminologie se v knihovně Fog nepoužívá). Pro vytvoření vzoru a nastavení požadovaných vlastností je tedy nutné použít třídu *Fog::Pattern* a její metody.

Seznam vzorů:

- žádný – nevykreslí se nic (*Fog::Pattern::Null*),
- jednotná barva (*Fog::Pattern::Solid*),
- lineární gradient (*Fog::Pattern::LinearGradient*),
- radiální gradient (*Fog::Pattern::RadialGradient*),
- konický gradient (*Fog::Pattern::ConicalGradient*),
- rastrový obrázek (*Fog::Pattern::Texture*).

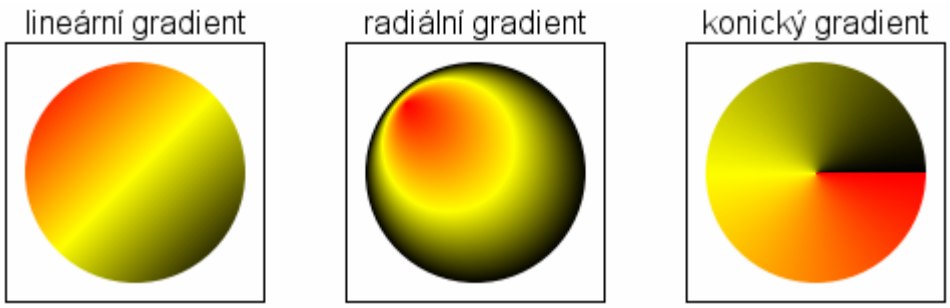
Vzory, které jsou definované jako gradientní výplň používají k definici průběhu výplně barevné značky (gradient stops), které se skládají z pozice (0 až 1) a barvy definované pomocí složek ARGB. Množství takových značek není omezené, minimum pro kreslení jsou ale 2. Výplně pomocí gradientu jsou kompatibilní se specifikací SVG [14].

Ukázka zdrojového kódu pro vytvoření a nastavení vzorů:

```
// Vytvoření instance třídy Pattern.
Pattern pat;

// Nastavení gradientu.
pat.setType(Fog::Pattern::LinearGradient);
// Pokud zvolíme gradient, je potřeba nastavit počáteční
```

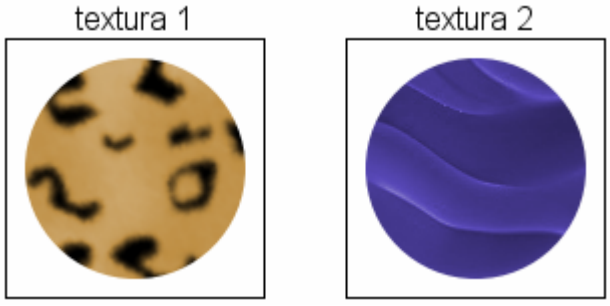
```
// a koncové souřadnice (hranice).
pat.setPoints(PointF(10.0, 10.0), PointF(90.0, 90.0));
// U gradientu musíme přidat i barevné značky (gradient stops).
pat.addGradientStop(GradientStop(0.0, Rgba(0xFFFF0000))); // Červená.
pat.addGradientStop(GradientStop(1.0, Rgba(0xFFFFFF00))); // Žlutá.

// ...

// Nastavení textury.
pat.setTexture(texture);
// Nastavení počáteční souřadnice.
pat.setStartPoint(PointF(10.0, 10.0));
```

Aplikování jednotlivých vzorů je zobrazeno na obrázku (*Obr. 20*). Gradient se ve všech případech skládá ze 3 barev a jsou měněny pouze hranice a typ gradientu. Vykreslená je vždy vyplněná kružnice, ale vykreslování s použitím vzorů není omezené na ostatní objekty.



Obr. 20: Použití gradientního vzoru.

Dále je možné použít jako vzor rastrový obrázek (texturu), jehož vykreslení je z hlediska výkonové náročnosti nejmenší. Ukázka použití takového vzoru je zobrazena na obrázku (*Obr. 21*).

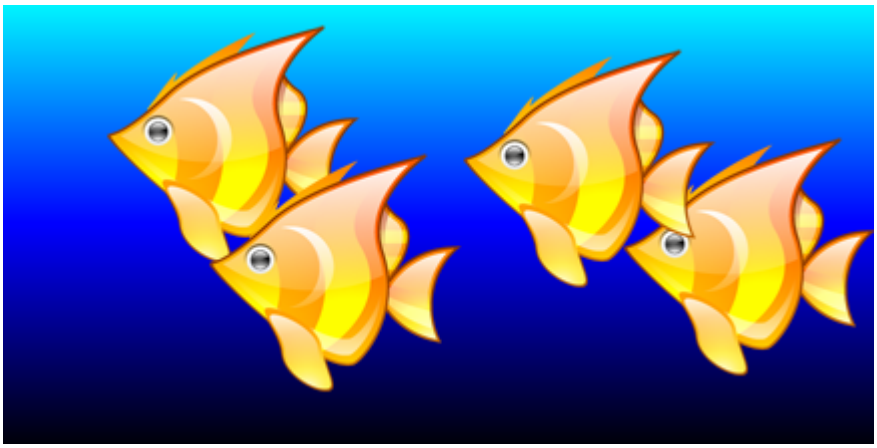


Obr. 21: Použití texturového vzoru.

5.2.3.5 Kreslení rastrových obrázků

Knihovna Fog umožňuje kombinovat vektorové kreslení s kreslením rastrových obrázků. Rastrový obrázek sice lze nastavit jako vzor, ale ve většině případů je jednodušší a hlavně rychlejší použít zabudované funkce pro jeho přímé vykreslení. K vykreslení slouží funkce *drawImage()*, která akceptuje pozici a referenci na zdrojový obrázek. Ukázka jednoduché

scény, která se skládá z vykreslení několika obrázku je na obrázku (*Obr. 22*). Jedná se o obrázek, který obsahuje údaj o průhlednosti jednotlivých pixelů – alfa kanál. Souřadnice obrázků byly zvoleny náhodně.



Obr. 22: Vykreslení obrázku s alfa kanálem.

5.2.3.6 Kreslení písma

Schopnost vykreslení písma patří k základním požadavkům na grafickou knihovnu. Knihovna Fog byla původně navržena jako grafický toolkit pro tvorbu uživatelského rozhraní v aplikacích, renderování písma je teda maximálně optimalizováno a interně je každý vykreslený znak uchováván v paměti pro co nejrychlejší sekundární vykreslení. Pro vykreslení písma slouží metoda *drawText()*.

Písmo reprezentuje třída Fog::Font a pomocí jejich metod lze nastavit font (např. arial), velikost písma a styl (tučné, kurzíva, ...). Zjištění dostupných písem a styl renderování písma je platformě závislý a knihovna Fog využívá pod operačním systémem Windows GDI subsystém, pod operačním systémem Linux se používá dvojice knihoven FontConfig (pro zjištění typů písem a jejich umístění) a FreeType2 k načtení fontů a jejich rasterizaci.

Ukázka nastavení písma požadované velikosti a stylu:

```
// Vytvoření instance třídy Fog::Font.
Font font;
// Nastavení typu písma.
font.setFamily(StubAscii8("arial"));
// Nastavení velikosti.
font.setSize(20);
// Změna stylu na kurzívu.
font.setItalic(true);
```

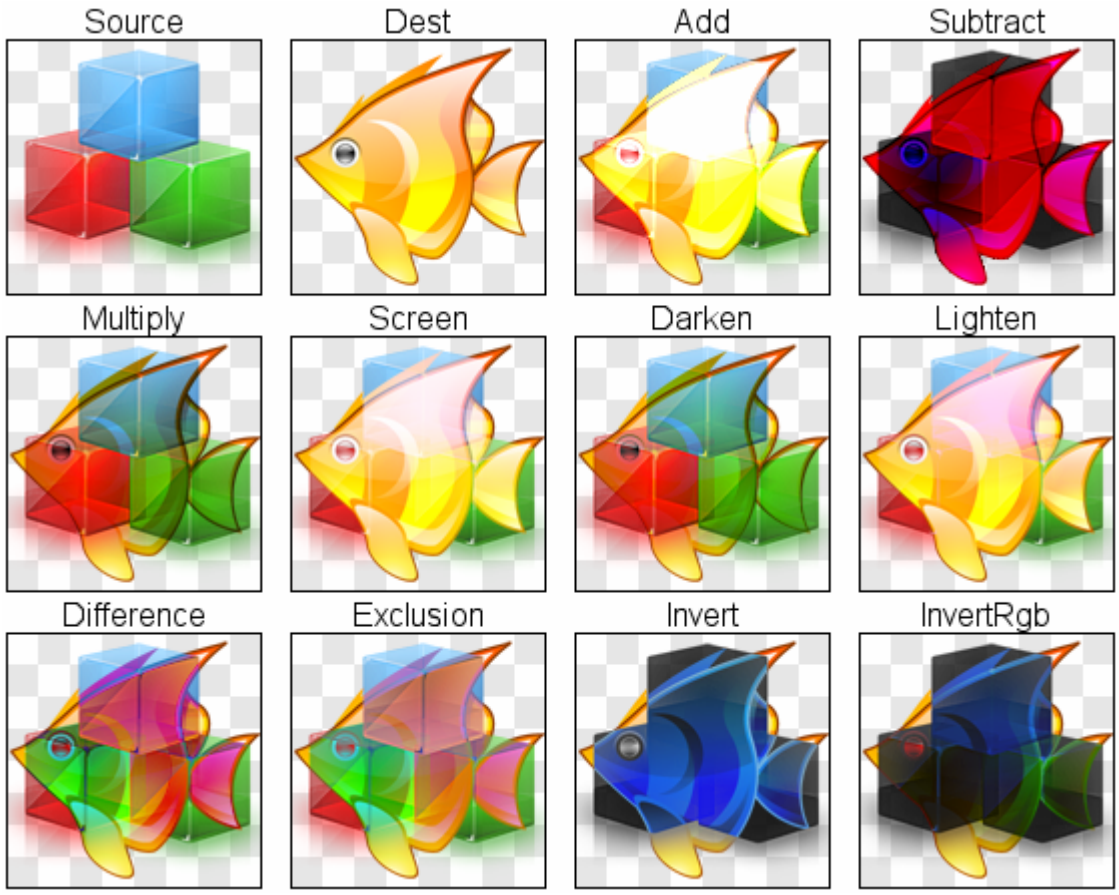
Na vykreslení písma se vztahují i vzory, takže je možné vykreslit písmo včetně různých dekorací. Použití jednotné barvy i vzorů je zobrazeno na obrázku (*Obr. 23*).

Obr. 23: Vykreslení textu barvou a pomocí vzorů.

5.2.3.7 Kompozitní operace

Grafický kontext v knihovně Fog umožňuje zvolit při kreslení různé kompozitní operace. Operace se zvolí pomocí metody *setOp()* a jejich název a význam je znázorněn na obrázku (*Obr. 24*) a (*Obr. 25*). Kompozitní operace je možné používat pouze při kreslení do obrázku, který obsahuje alfa kanál. Obrázky bez alfa kanálu používají pouze jedinou základní operaci známou jako Source Over (alpha blending). Kompozitní operace jsou založeny na specifikaci SVG a jejich podrobný popis včetně jejich matematického vyjádření je její součástí [14].

Obr. 24: Kompozitní operace - Porter & Duff.



Obr. 25: Kompozitní operace – rozšíření.

5.2.3.8 Afinní transformace

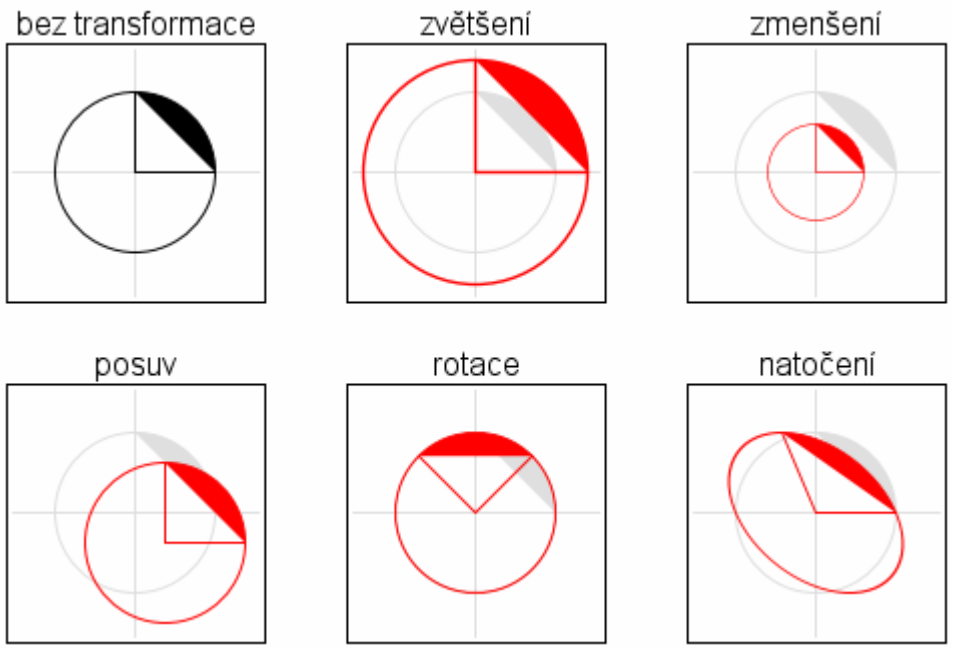
Afinní transformace jsou lineární transformace definované v kartézských souřadnicích (striktně řečeno se nejedná jen o kartézské souřadnice, ale v knihovně Fog jsou aplikované na kartézské souřadnice vždy). Obsahují rotaci, zvětšování / zmenšování, posuv a natočení. Výhoda afinních transformací z hlediska implementace spočívá v tom, že všechny operace a jejich kombinace lze aplikovat na kartézské souřadnice pouze vynásobením souřadnic afinní maticí (4x násobení).

V knihovně Fog jsou afinní transformace implementovány ve třídě *Fog::AffineMatrix*. Afinní transformace lze kombinovat přímo prostřednictvím afinní matice a voláním metod třídy *Fog::AffineMatrix* nebo prostřednictvím grafického kontextu, který obsahuje stejné metody shodné funkcionality.

Grafický kontext (nebo afinní matice) tedy umožňuje:

- rotaci – *rotate(angle)*,
- zvětšení / zmenšení – *scale(sx, sy)*,
- posuv – *translate(dx, dy)*,
- natočení – *skew(sx, sy)*.
- vynásobit afinní maticí – *affine(matrix)*,
- nastavit afinní matici – *setMatrix(matrix)*,
- zrušit všechny transformace – *resetMatrix()*.

Jednotlivé operace jsou znázorněné na obrázku (*Obr. 26*).



Obr. 26: Afinní transformace.

5.2.3.9 Předcházení problémům s vícevláknovou architekturou

Protože třída *Fog::Painter* může použít vlákna k urychlení vykreslování, bylo nutné implementovat synchronizační prostředek, který může použít i programátor bez znalostí, jestli jsou nebo nejsou vlákna aktuálně použité. Třída *Fog::Painter* obsahuje metodu *flush()*, která zaručí, že všechny grafické operace jsou vykresleny a všechny zdroje použité pro kreslení písma, cest a obrázků uvolněny. Vnitřně je metoda *flush()* implementována tak, že počká na dokončení všech zbývajících operací všech vláken.

Teoreticky by vícevláknové vykreslování nikdy nemělo způsobit problémy, protože knihovna Fog používá techniku copy-on-write ke sdílení dat i v grafických třídách. V praxi se ale může nesprávné použití projevit degradací výkonu zbytečným kopírováním velkých sdílených paměťových oblastí.

Následuje pár tipů pro správné použití třídy *Fog::Painter*:

- nikdy nepoužívat jako zdroj pro kreslení stejný obrázek jako je ten cílový (výstup je v takovém případě nedefinovaný),
- snažit se nemodifikovat obrázek, který byl použit jako zdroj pro kreslení, ať už se jedná o vzor (*Fog::Pattern*) nebo samotný obrázek (*Fog::Image*),
- vypnout vícevláknové vykreslování, pokud zdroj dat obsahuje jen velmi malé regiony ke kreslení (grafický kontext není navržen pro kresbu pixel po pixelu).

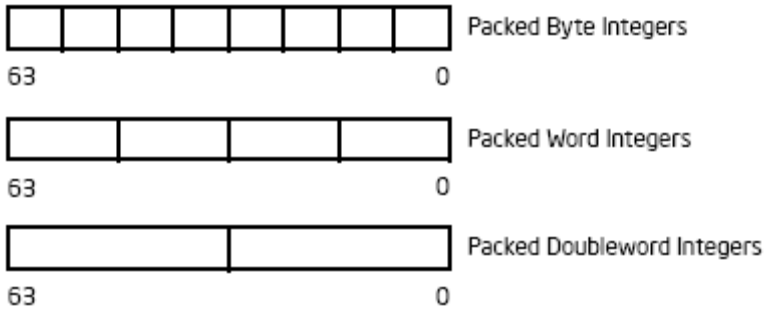
Ukázka nesprávného použití třídy *Fog::Painter* a řešení:

```
using namespace Fog;

// Vytvoření cílového obrázku o rozměrech 320x200
Image dst(320, 200, Image::FormatPRGB32);

// Vytvoření zdrojového obrázku o rozměrech 320x200
Image src(320, 200, Image::FormatPRGB32);

// Vytvoření grafického kontextu.
Painter p(dst);

// Vykreslení zdrojového obrázku
p.drawImage(Point(0, 0), src);

// Modifikace zdrojového obrázku zapříčiní vytvoření kopie v případě,
// že je obrázek používán ještě alespoň jedním vláknem (pravděpodobné).
src.fillRect(Rect(0, 0, 100, 100));

// Pokud se chceme problému vyhnout, je nutné před src.fillRect() zavolat
// p.flush().
```

5.3 Optimalizace pomocí MMX / SSE2

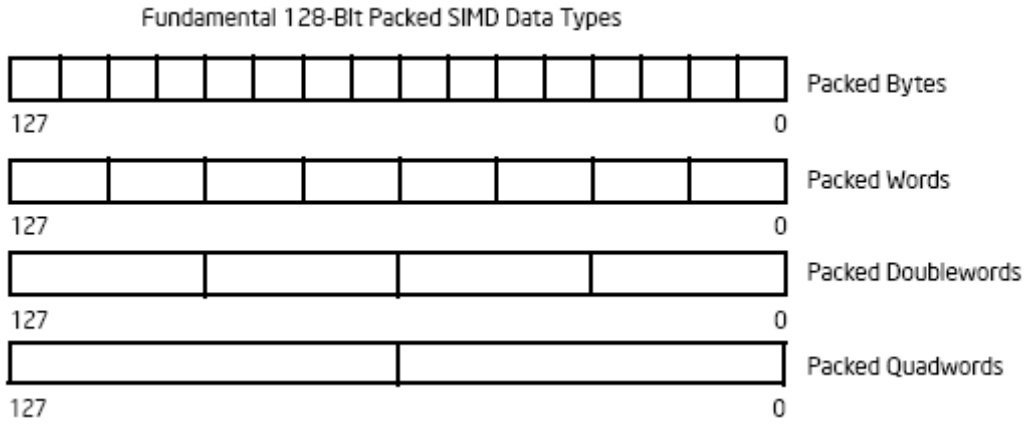
Pokud chceme optimalizovat jakýkoliv kód, je nejprve nutné prozkoumat algoritmickou složitost daného úseku kódu a zjistit, které funkce spotřebují nejvíce procesorového času. U grafických knihoven je největší výkon spotřebovaný na rasterizaci vstupních dat a na kompozici pixelů. Pomocí speciálních optimalizací, které využívají instrukce MMX a SSE2 lze vylepšit výkon oproti standardní C implementaci.

MMX technologie definuje jednoduchý, ale flexibilní model k paralelnímu zpracování dat, které jsou uložené v 64bitovém registru [3]. V jedné instrukci je možné nezávisle na sobě zpracovat celá čísla dlouhé 8, 16 nebo 32 bitů (*Obr. 27*).



Obr. 27: MMX registr [3].

SSE2 technologie rozšiřuje paralelní zpracování na 128bitové registry, které jsou zcela nezávislé (takže je možné kombinovat MMX i SSE2) [3]. SSE2 technologie rozšiřuje i nejmenší entitu, kterou je možné zpracovat na 64 bitů, takže je možné i na 32bitovém procesoru provádět plně 64bitové operace s celými čísly (*Obr. 28*).



Obr. 28: SSE2 registr [3].

Z pohledu grafické knihovny, která zpracovává pixely uložené jako 8bitové RGB nebo ARGB entity je nejvýhodnější používat 16bitové paralelní data (Packed Words), díky kterým je možné násobit dvě 8bitové čísla a získat tak 16bitový výsledek. Technologie MMX umožňuje zpracovat jeden 32bitový ARGB pixel na registr a technologie SSE2 umožňuje zpracovat tyto pixely 2. V hlavní smyčce se většinou používá paralelní zpracování 2 registrů, takže při použití SSE2 technologie se pracuje se 4 pixely na jeden hlavní cyklus.

5.3.1 Optimalizace výplně pomocí MMX / SSE2

Výplň je jedna z nejdůležitějších nízkoúrovňových grafických operací, která je volána velice často, pokud vykreslujeme geometrické objekty stejnou barvou. Optimalizace pomocí technologie MMX a SSE2 využívá šířky MMX nebo SSE2 registru k uložení více pixelů současně. Podmínka pro efektivní průběh je zarovnání cílového bufferu na 8 nebo 16 bytů (podle použité technologie).

C++ kód, který využívá technologii SSE2 k výplni pole 32 bitových pixelů bude vypadat následovně (pouze hlavní smyčka):

```
while (i >= 16)
{
  _mm_store_si128((__m128i*)(dst), src0mm);
  _mm_store_si128((__m128i*)(dst + 16), src0mm);
  _mm_store_si128((__m128i*)(dst + 32), src0mm);
  _mm_store_si128((__m128i*)(dst + 48), src0mm);

  dst += 64;
  i -= 16;
}
```

Kde *dst* je cílový buffer zarovnaný na 16 bytů, *src0mm* jsou zabalené 4 pixely v jednom *xmm* registru a *i* je počítadlo. Jeden průchod smyčky uloží do paměti 16 pixelů o velikosti 32 bitů. MMX kód by vypadal podobně, pouze by se ukládalo po 8 bytech (šířka MMX registru). Úplnou implementaci výplně s využitím SSE2 je možné nalézt ve funkci raster_rgb32_span_solid_sse2() v souboru ./Fog/Graphics/Raster/Raster_SSE2.cpp. Ve zmíněném souboru jsou implementované veškeré SSE2 optimalizace, které jsou součástí knihovny Fog.

5.3.2 Optimalizace výpočtu gradientu pomocí MMX / SSE2

Gradienty jsou v současnosti hodně používané v počítačové grafice, protože jejich renderování není nijak složité a vizuálně působí velice příjemně. Renderování gradientu spočívá v lineární interpolaci dvou nebo více barev. Knihovna Fog používá k interpolaci barev vždy funkci, která vypočítá interpolaci mezi dvěma barvami. Při interpolaci více barev se zavolá tato funkce pro každý pár barev samostatně.

Prototyp funkce pro interpolaci vypadá následovně

```
void gradient(void* dst, Rgba c0, Rgba c1, int w, int x1, int x2)
```

kde

- *dst* je ukazatel na cílový buffer,

- c_0 a c_1 jsou barvy (primární a sekundární),
- w je délka interpolace (v pixelech),
- x_1 a x_2 jsou startovní a koncové pozice ($x_1 < x_2$).

Jednotlivé barvy na pozici 0 až w můžeme vyjádřit vztahem

$$c_p = c_0 + (c_1 - c_0)\, p / w \tag{3}$$

kde

- c_p je cílová barva na pozici p,
- c_0 a c_1 jsou barvy přechodu (viz prototyp funkce pro interpolaci),
- p je současná pozice (v rozsahu 0 až w),
- w je délka interpolace (číslo větší než 0).

pro pozici menší než 0 se použije startovní barva c_0 a pro pozici větší než w se použije cílová barva c_1. Základní implementace by se tedy měla specializovat na případy výpočtu pro $p < 0$, p v intervalu 0 až w a $p > w$. Pokud implementujeme tyto 3 speciální případy, vyhneme se podmínkám v hlavím cyklu, jedná se ovšem jen o první krok k maximalizaci výkonu.

Pokud počítáme interpolaci pro více barev (nejčastější použití), můžeme využít lineárnosti interpolace a místo násobení a dělení v každém cyklu použít pouze sčítání. Rovnici je možné vyjádřit vztahy

$$\begin{aligned} c_i &= (c_1 - c_0) / w \\ c_p &= c_{p-1} + c_i \end{aligned} \tag{4}$$

kde

- c_i je přírůstek mezi jedním krokem interpolace,
- c_p je cílová barva na pozici p.

Pro výpočet každého kroku je pouze nutné přičíst proměnnou c_i k barvě z předchozího kroku. Na matematické úrovni už nelze interpolaci dále zjednodušit, ale na programové úrovni ano. Jeden z prvních kroků je nepočítat interpolaci pomocí reálních čísel (pohyblivá desetinná čárka), ale použít fixed point (reprezentace čísel s fixní desetinou čárkou). Výchozí typ *int* je na 32bitové a 64bitové x86 architektuře 32 bitů a pro výpočet přechodu

jedné složky barvy naprosto postačuje. Tento typ si rozdělíme na 16 bitů pro celočíselnou část a 16 bitů pro desetinnou část. Jedná se o reprezentaci označovanou jako 16.16 fixed point.

Implementaci je možné najít v souboru Fog/Fog/Raster/Raster_C.cpp ve funkci gradient_gradient_argb32(). Hlavní smyčka vypadá následovně:

```
do {
  set4(dstCur,
    ((uint32_t)(grCur[0] & (0xFF << 16)) >> 16) |
    ((uint32_t)(grCur[1] & (0xFF << 16)) >>  8) |
    ((uint32_t)(grCur[2] & (0xFF << 16))      ) |
    ((uint32_t)(grCur[3] & (0xFF << 16)) <<  8) );
  dstCur += 4;

  grCur[0] += grStp[0];
  grCur[1] += grStp[1];
  grCur[2] += grStp[2];
  grCur[3] += grStp[3];
} while (--w);
```

Je nutné vypočítat každou složku interpolace v RGBA pixelu. Pro urychlení počítání je možné využít technologii MMX a SSE2, pomocí kterých je možné počítat více složek barev současně. Technologie MMX umožňuje výpočet 2 složek a technologie SSE2 4 složky. Hlavní smyčka optimalizovaného cyklu pro SSE2 pro výpočet 4 pixelů současně je zobrazena v následujícím bloku kódu:

```
while (i >= 4)
{
  xmm2 = xmm0;                                  // xmm2 = [xAxxxRxxxGxxxBxx]
  xmm0 = _mm_add_epi32(xmm0, xmm1);             // xmm0 += xmm1

  xmm3 = xmm0;                                  // xmm3 = [xAxxxRxxxGxxxBxx]
  xmm0 = _mm_add_epi32(xmm0, xmm1);             // xmm0 += xmm1

  xmm4 = xmm0;                                  // xmm4 = [xAxxxRxxxGxxxBxx]
  xmm0 = _mm_add_epi32(xmm0, xmm1);             // xmm0 += xmm1

  xmm5 = xmm0;                                  // xmm5 = [xAxxxRxxxGxxxBxx]
  xmm0 = _mm_add_epi32(xmm0, xmm1);             // xmm0 += xmm1

  xmm2 = _mm_packus_epi16(xmm2, xmm3);          // xmm2 = [AxRxGxBxAxRxGxBx]
  xmm4 = _mm_packus_epi16(xmm4, xmm5);          // xmm4 = [AxRxGxBxAxRxGxBx]
  xmm2 = _mm_srli_epi16(xmm2, 8);               // xmm2 = [0A0R0G0B0A0R0G0B]
  xmm4 = _mm_srli_epi16(xmm4, 8);               // xmm4 = [0A0R0G0B0A0R0G0B]

  xmm2 = _mm_packus_epi16(xmm2, xmm4);          // xmm2 = [ARGBARGBARGBARGB]

  _mm_store_si128((__m128i *)dstCur, xmm2);

  dstCur += 16;
  i -= 4;
}
```

Jednotlivé funkce začínající prefixem *_mm* jsou přeloženy jako instrukce CPU, kód tedy v jazyce assembler zabírá jen pár instrukcí a vykonává toho nepřekonatelně více než původní verze napsaná v čistém C.

5.4 Optimalizace pomocí použití více vláken

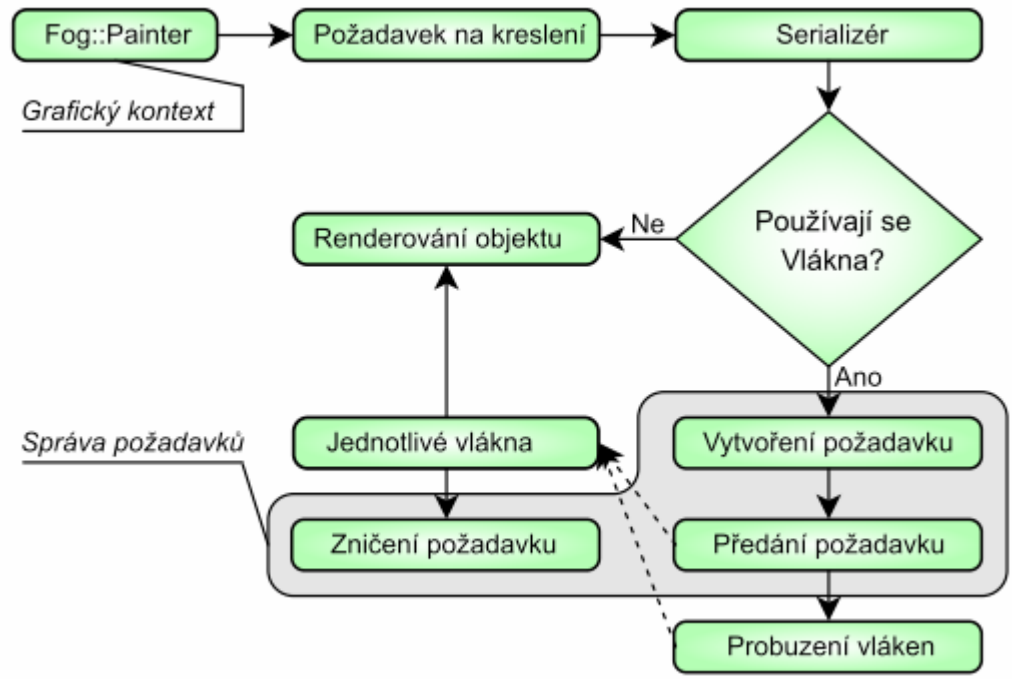
Optimalizace vykreslování pomocí použití více vláken byla základní myšlenka při návrhu knihovny Fog. Jedná se o zatím málo prozkoumanou oblast v oblasti vektorových grafických knihoven a knihovna Fog obsahuje referenční implementaci, kterou je možné použít jako základ pro jinou grafickou knihovnu. Při implementaci byl kladen důraz na jednoduchost a budoucí rozšiřitelnost.

Návrh implementace je popsán v kapitole 4.4 (*Obr. 9*), tato kapitola tedy upřesňuje samotnou implementaci a triky použité k maximalizaci výkonu vícevláknové architektury. Základem při implementaci se staly atomické operace, které umožnily pohodlně sdílet paměťové struktury mezi vlákny. Ovšem vývoj nebyl jen o atomických operacích a strukturách, byl navržen mechanizmus pro rychlou serializaci požadavku na grafický výstup a jeho rozdělení mezi vlákna. Knihovna Fog je implementována tak, že všechny grafické funkce je možné kategorizovat do 4 typů. Každý typ funkce má svůj serializér, který je schopen vykreslit ihned požadovanou operaci (v případě, že se nepoužívají vlákna) nebo předat detail operace včetně všech potřebných dat ostatním vláknům, které se o vykreslení postarají.

Typ operací, které mají svůj serializér:

- Grafické cesty - *_serializePath()*,
- Obdélníková výplň - *_serializeBoxes()*,
- Vykreslení obrázku - *_serializeImage()*,
- Vykreslení písma - *_serializeGlyphs()*.

Mnoho operací použije serializér pro grafickou cestu, ovšem pro vykreslování textu a obrázků bylo potřeba mnohem výkonnější řešení. Výplň obdélníků má svůj vlastní serializér pouze z důvodu rychlosti. Proces serializace pro jednovláknové a vícevláknové kreslení je znázorněn na obrázku (*Obr. 29*).



Obr. 29: Proces serializace grafického požadavku.

Je možné si všimnout, že při vícevláknovém vykreslování je potřeba mnohem více operací pro vykreslení jakéhokoliv grafického objektu. Zhodnocení včetně testů rychlosti jsou probrány v kapitole 6.

5.5 Optimalizace pomocí JIT kompilace

JIT (just in time) kompilace je v dnešní době trend zejména v oblasti programovacích jazyků. V počítačové grafice se můžeme částečně setkat s JIT kompilací v podobě shaderů v OpenGL a DirectX, ale v 2D vektorové grafice se tato technika pro zvýšení výkonu nepoužívá (přesněji ne v dostupných open source řešeních).

Mezi pozitiva na JIT kompilaci můžeme zařadit maximalizaci výkonu vykonání často se opakující smyčky v programu. Negativních aspektů je bohužel mnohem víc, a mezi nejsilnější negativa patří psaní platformě závislého kódu a mnohem více času potřebného na přepsání i velmi jednoduché smyčky do podoby JIT. Z hlediska času programátora (a tedy i nákladů na vývoj SW) podpora pro JIT kompilaci prodlouží dobu potřebnou na vývoj SW.

V rámci diplomové práce bylo přispěno do 2 knihoven, které nejsou nijak spojené s knihovnou Fog. Mezi první patří knihovna AsmJit. Tato knihovna umožňuje psát JIT

assembler přímo v C++ jazyce. Další knihovna se jmenuje BlitJit, která se zabývá JIT kompilací grafických funkcí pro kompozici pixelů.

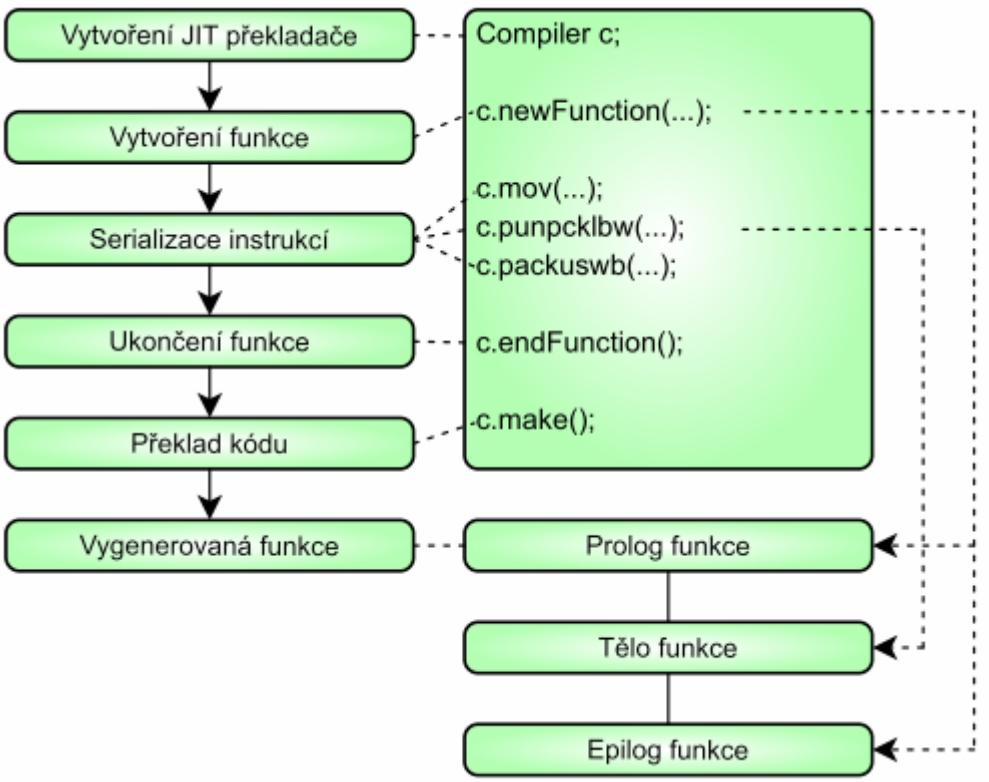
5.5.1 AsmJit

AsmJit je kompletní x86/x64 JIT assembler pro jazyk C++. Podporuje instrukční sadu FPU, MMX, 3dNow, SSE, SSE2, SSE3 a SSE4. Obsahuje překladač, pomocí kterého je možné psát multiplatformní kód pro operační systémy Windows, Linux a Mac OS X, pro 32bitový i 64bitový režim. Knihovna AsmJit může být použitá pro vytvoření funkcí spustitelných v C/C++ jazyce nebo z již vygenerovaného kódu [10].

Ukázka použití knihovny AsmJit:

```
// Používáme prostor jmen AsmJit.
using namespace AsmJit;

// Prototyp funkce, kterou budeme generovat.
typedef void (*MyFn)(int*);

// Vytvoří instanci třídy AsmJit::Compiler.
Compiler c;
// Vytvoří prototyp funkce: void f(int*).
c.newFunction(CALL_CONV_DEFAULT, BuildFunction1<int*>());
// Proměnná prvního parametru funkce.
PtrRef a1(f.argument(0));
// Uložíme do ukazatele (první parametr) hodnotu 1024.
c.mov(dword_ptr(a1.r()), imm(1024));
// Konec funkce.
c.endFunction();

// Vytvoření JIT funkce.
MyFn fn = function_cast<MyFn>(c.make());
// Zavolání funkce.
int x;
fn(&x);
```

Průběh generování funkce pomocí knihovny AsmJit je zobrazen na obrázku (*Obr. 30*).



Obr. 30: Průběh generování funkce pomocí knihovny AsmJit.

Knihovna AsmJit je základem pouze pro knihovnu BlitJit a v knihovně Fog se nepoužívá. Cílem bylo umožnit použití této knihovny pro různé projekty, které nejsou nijak spojené s knihovnou Fog nebo BlitJit.

5.5.2 BlitJit

BlitJit je vysoce výkonná knihovna, která slouží pro práci s pixely na rastrové úrovni. Je napsaná v jazyce C++ s úmyslem využít pokročilé rozšíření procesoru a JIT kompilaci. Knihovna BlitJit používá pro JIT kompilaci knihovnu AsmJit a rozšiřuje tak funkcionalitu této knihovny pro vývoj vysoce výkonných funkcí pro kompozici pixelů. Díky knihovně AsmJit je knihovna BlitJit kompatibilní s operačními systémy Windows, Linux a Mac OS X (32bitový i 64bitový režim) [11].

Knihovna BlitJit v současné době podporuje pouze 32bitový formát pixelů, lze ji tedy použít pouze pro vygenerování funkcí pro formáty ARGB32, PRGB32 a RGB32. Výhodou je, že podporuje všechny kompozitní operace, které podporuje i knihovna Fog, takže

je možné zkompilovat pro zmíněné formáty pixelů téměř všechny funkce z mapy funkcí (viz kapitola 4.3, *Obr. 8*).

5.5.3 Souhrn

Knihovny AsmJit a BlitJit nabízí opravdu hodně možností, jak v budoucnu optimalizovat kód. V současnosti je možné popsat knihovnu AsmJit jako použitelnou a stabilní knihovnu pro psaní JIT kódu. Na druhé straně knihovna BlitJit by potřebovala ještě hodně práce k podpoře všech různých formátů pixelů a operací s nimi. V rámci diplomové práce se stala knihovna BlitJit použitelná pro projekt Fog, ale zatím se nejedná o kompletní řešení a knihovna Fog se bez knihovny BlitJit obejde (je potřeba jen velmi málo kódu k integraci, kterou lze vypnout).

6 SROVNÁNÍ RYCHLOSTI

Pro srovnání rychlosti knihovny Fog, GDI+ a Cairo byla napsána aplikace, která vykoná sto tisíc grafických operací za sebou pro každý typ testu. Byl spočítán procesorový čas potřebný na dokončení každé sady testů a čas celkový. Testování knihovny Fog proběhlo ve vícevláknovém i jednovlíknovém režimu.

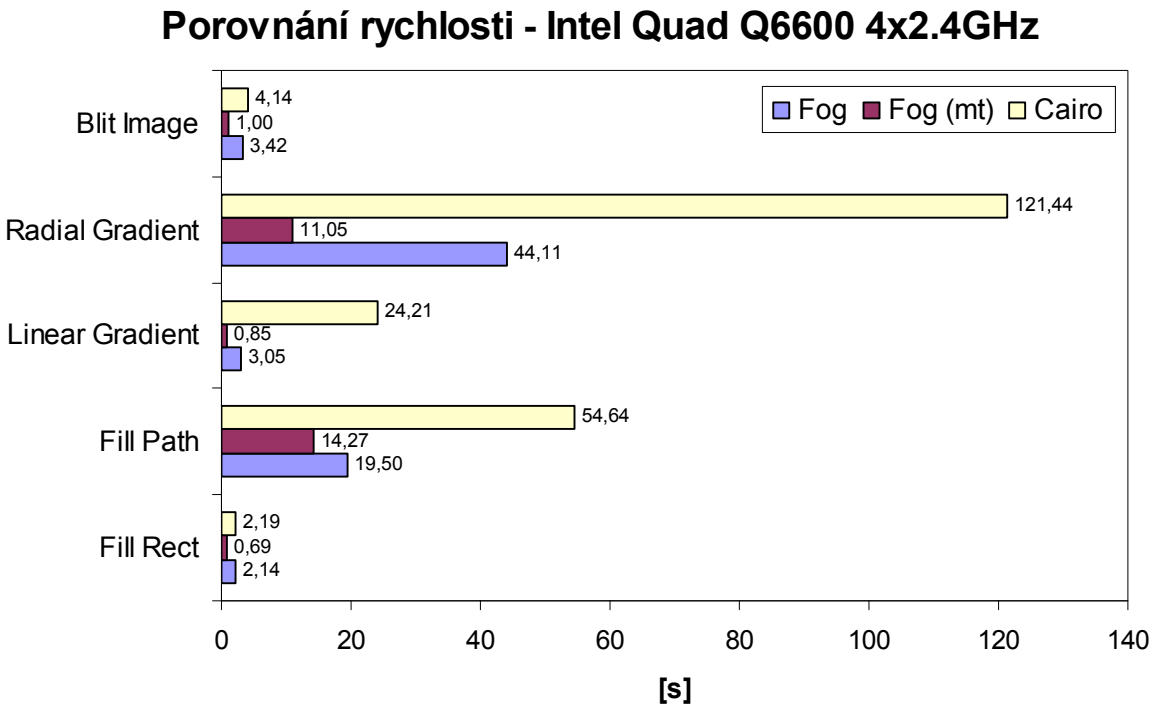
Testovací aplikace zahrnuje tyto grafické operace:

- výplň obdélníku (ve všech knihovnách se jedná o speciální případ),
- výplň grafické cesty,
- kreslení rastrových obrázků,
- kreslení gradientů (lineární a radiální, knihovna GdiPlus umožňuje pouze lineární).

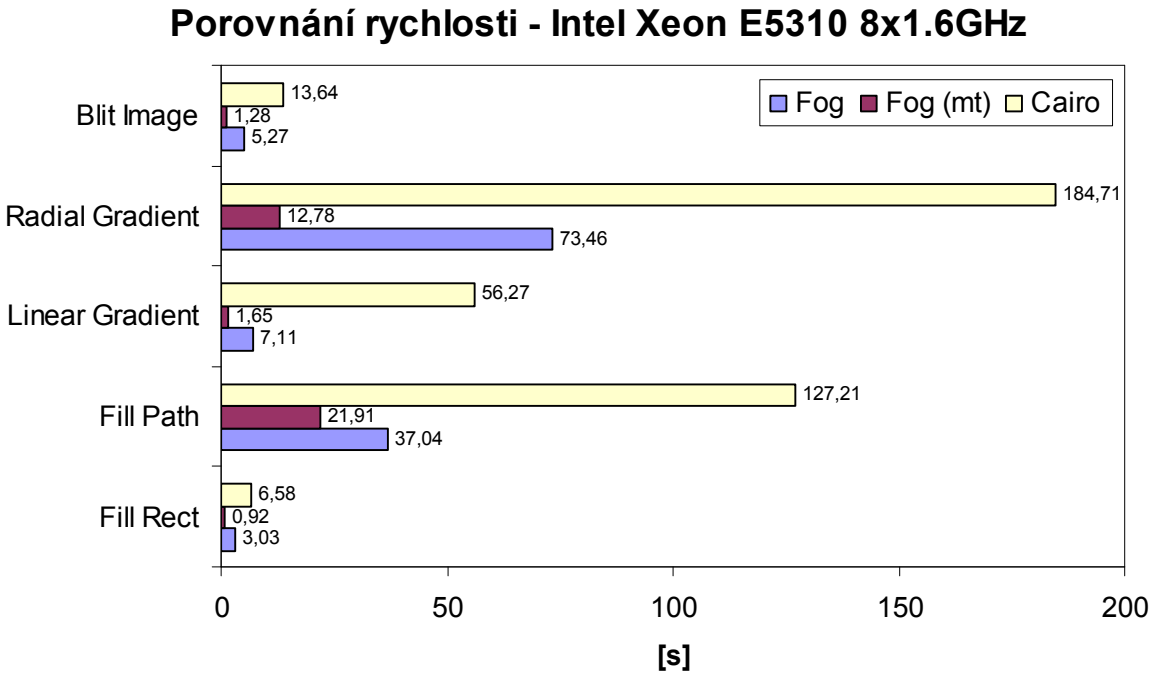
Kreslení proběhlo v rastrovém obrázku o velikosti 640x480 pixelů o bitové hloubce 32 bitů. Všechny operace s výjimkou vykreslování grafických cest byly o velikosti 128x128 pixelů a jejich souřadnice a barvy byly zvoleny pomocí generátoru pseudonáhodných čísel. Generátor náhodných čísel byl před každým testem resetován, aby bylo docíleno stejné posloupnosti náhodně generovaných souřadnic a barev. Test vykreslování grafické cesty probíhal generováním 3 náhodných uzavřených křivek a jejich výplní. Ve všech testech je zobrazen celkový čas potřebný na vykreslení 100.000 objektů, čím menší čas, tím lepší výsledek.

6.1 Fog vs. Cairo - Linux

První výsledky byly naměřeny na počítačové sestavě s procesorem Intel Quad Q6600 4x2.4GHz s operačním systémem Gentoo Linux 64-bit (*Obr. 31*). Dále byly naměřeny výsledky na počítačové sestavě s procesorem Intel Xeon E5310 8x@1.6GHz s operačním systémem Ubuntu Linux 64-bit (*Obr. 32*).



Obr. 31: Porovnání rychlosti – Intel Quad Q6600, Gentoo Linux 64-bit.



Obr. 32: Porovnání rychlosti – Intel Xeon E5310 8x1.6GHz, Ubuntu Linux 64-bit.

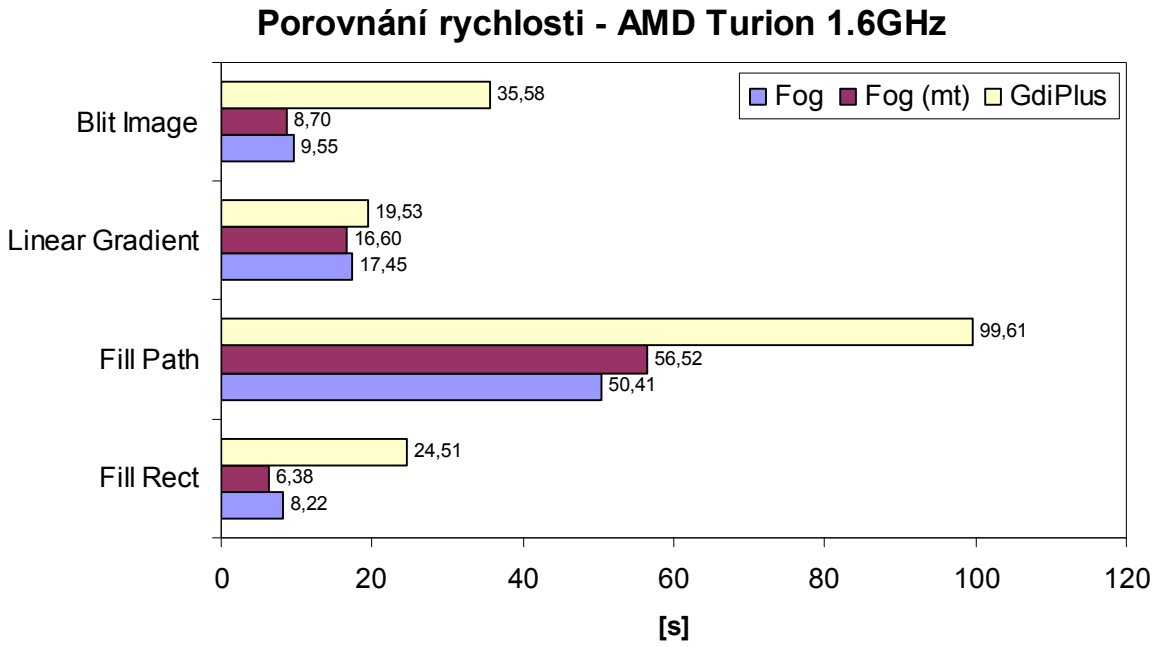
Z naměřených výsledů je patrné, že vícevláknová architektura opravdu zrychlila vykreslování testovaných vzorů a geometrických primitiv. Některé grafické operace (např.

výplň obdélníku a vykreslování gradientů) bylo možné maximálně paralelizovat a dosáhnout tím opravdu maximálnímu zvýšení výkonu. Výplň grafické cesty obsahuje příliš mnoho kódu, který je sekvenční (nedá se tedy tak dobře paralelizovat) a nárůst výkonu není nijak dramatický (ale rozhodně stojí za povšimnutí).

Dále bylo zjištěno, že knihovna Fog dokazuje mnohem lepších výsledků než knihovna Cairo i bez použití vláken, architektura knihovny Fog je tedy navržena více s ohledem na rychlost knihovny.

6.2 Fog vs. GdiPlus – Windows

Pod Windows byla k dispozici pouze sestava s jednojádrovým procesorem, bylo tedy možné změřit pouze rychlost knihovny Fog a GdiPlus bez konkurenčního běhu více vláken současně. Měření se odehrálo na sestavě s procesorem AMD Turion 1.6GHz s operačním systémem Windows XP 32-bit (*Obr. 33*).



Obr. 33: Porovnání rychlosti – AMD Turion 1.6GHz, Windows XP 32-bit.

Z výsledku testu je patrné, že knihovna Fog byla rychlejší ve všech testech. Velmi zajímavý je i fakt, že vícevláknová architektura neměla negativní dopad na rychlost při použití více vláken i na jednojádrovém procesoru a někdy byly výsledky překvapivě lepší.

7 METODIKA INTEGRACE DO JINÝCH KNIHOVEN

V této kapitole je popsána kompatibilita knihovny Fog a možnosti integrace do jiných knihoven a GUI toolkitů. Knihovna Fog byla navržena jako kompletní řešení pro vektorovou grafiku a správu písma, jedná se tedy hlavně o integraci na rastrové úrovni (integrace obrázků a kreslení do cizího bufferu). Jako nejlepší forma integrace se jeví přímá podpora knihovny Fog ostatními knihovnami, ale to je v ranné fázi vývoje knihovny Fog nemožné.

Integrace tedy záleží na tom, jestli ostatní knihovny podporují formát pixelů, který se používá v knihovně Fog. Knihovna Fog používá formát pixelů, který je platformě závislý, a který je kompatibilní s okenními systémy Windows a X11. Následující tabulka (*Tab. 8*) obsahuje prozkoumané knihovny a stav kompatibility.

Tab. 8: Stav kompatibility knihovny Fog s ostatními knihovnami.

Knihovna / Subsystém	Kompatibilita	Komentář
WinAPI, GDI	Ano	Kompatibilní 32bitové i 24bitové formáty. Je to dáno tím, že WinAPI používá nativní formáty pixelů jako knihovna Fog.
X Window System (X11)	Ano	Kompatibilní 32bitové formáty. Platí to samé co pro WinAPI.
Qt Toolkit	Ano	Kompatibilní 32bitové, 24bitové i 8bitové formáty. I Qt Toolkit se drží platformě závislých formátů.
Gtk (Gimp Toolkit) (konkrétně gdk_pixbuf)	Ne	Gdk definuje vlastní formát pixelů, který je sice 32bitový, ale na platformě Little Endian je pozice pixelů opačná (BGRA).
Cairo	Ano	Cairo definuje jen několik formátů a všechny jsou kompatibilní s knihovnou Fog (konkrétně RGB32 a PRGB32).
Fltk	Ano	Fltk může být zkompilované s podporou knihovny Cairo, podporuje tedy i formáty

		knihovny Fog (RGB32 a PRGB32).
wxWidgets	Ne	Knihovna wxWidgets definuje vlastní formát, který není ve světě počítačové grafiky příliš používaný. Jedná se o 24bitový RGB formát (Big Endian) a možnost přidat alfa kanál jako samostatný buffer.
SDL	Ano	Knihovna SDL podporuje téměř jakýkoliv formát, s knihovnou Fog nenastal problém a její první verze byly dokonce testovány přes knihovnu SDL.

Z tabulky je možné vyčíst, že většina grafických knihoven a GUI toolkitů se drží platformě závislých formátů k dosažení maximální kompatibility s formáty použitých operačních a okenních systémů. V následující části budou popsány postupy integrace s jednotlivými knihovnami.

7.1 Integrace s WinAPI, GDI

Tato podkapitola se zabývá integrací knihovny Fog se subsystémem GDI, který se používá jako výchozí v operačním systému Windows. Knihovna GDI umožňuje vytvořit tzv. DibSection, což je platformě nezávislá reprezentace dvojrozměrného rastrového obrázku [16]. Pokud jsme vytvořili DibSection, je možné předat ukazatel na rastrové data knihovně Fog (včetně rozměrů a formátu pixelů) a ihned začít pomocí knihovny Fog do předaného bufferu kreslit.

Příklad vytvoření grafického kontextu nad GDI obrázkem:

```
bool beginDibPainter(Fog::Painter* p, HBITMAP hbm)
{
  // Potřebujeme informace o DIBSECTION.
  DIBSECTION info;
  if (GetObject(hbm, sizeof(DIBSECTION), &info) == 0) return false;

  // Pokud se nejedná o 24bitový nebo 32bitový obrázek, končíme.
  if (info.dsBm.bmBitsPixel != 24 && info.dsBm.bmBitsPixel != 32) return false;

  // Vnutíme vektorovému grafickému kontextu data získané z DIBSECTION.
  p->begin((uint8_t*)info.dsBm.bmBits,
    info.dsBm.bmWidth, info.dsBm.bmHeight, info.dsBm.bmWidthBytes,
    info.dsBm.bmBitsPixel == 24
      ? Fog::Image::FormatRGB24
      : Fog::Image::FormatPRGB32);
  return true;
}
```

7.2 Integrace s X Window System (X11)

Knihovna Fog je kompatibilní s formátem, který definuje X Window System (X11), ale v rámci této práce není možné poskytnout ukázku kódu, který by umožnil spolupráci obou knihoven. Je to dáno tím, že knihovna Xlib je jen prostředník pro komunikaci s X Serverem a vytvoření sdílené paměti mezi procesy a synchronizace kreslení do takové paměti přesahuje rozsah této práce. Pokud by se někdo o tento postup zajímal, tak je potřeba prozkoumat funkce *shmget()*, *shmat()*, *XShmAttach()* a *XShmCreatePixmap()*.

7.3 Integrace s knihovnou Qt

Qt Toolkit obsahuje třídu *QImage*, do které je možné kreslit pomocí knihovny Fog. Opět je nutné zavolat metodu *Fog::Painter::begin()* s ukazatelem na rastrové data a jejich definici.

Příklad vytvoření grafického kontextu nad *QImage*:

```
bool beginQImagePainter(Fog::Painter* p, QImage* qimage)
{
  int fmt = -1;

  // Převod formátu z QImage na Fog:
  switch (qimage->format())
  {
    case QImage::Format_RGB32:
      fmt = Fog::Image::FormatRGB32; break;
    case QImage::Format_ARGB32:
      fmt = Fog::Image::FormatARGB32; break;
    case QImage::Format_ARGB32_Premultiplied:
      fmt = Fog::Image::FormatPRGB32; break;
    case QImage::Format_RGB888:
      fmt = Fog::Image::FormatRGB24; break;
    default: // Žádný vhodný
      return false;
  }

  // Vnutíme vektorovému grafickému kontextu data získané
  // z QImage.
  p->begin((uint8_t*)qimage->bits(),
    qimage->width(), qimage->height(),
    qimage->bytesPerLine(),
    fmt);

  return true;
}
```

7.4 Integrace s knihovnou Cairo

Knihovna Cairo obsahuje naprosto totožné formáty pixelů jako knihovna Fog, je tedy možné vytvořit vektorový grafický kontext opět pomocí vnucení rastrových dat z Cairo obrázku do vektorového grafického kontextu.

Příklad vytvoření grafického kontextu nad Cairo obrázkem:

```
bool beginCairoPainter(Fog::Painter* p, cairo_surface_t csurf)
{
  int fmt = -1;

  // Pokud Cairo vrátí NULL, nejedná se o rastrový obrázek.
  uint8_t* bits = cairo_image_surface_get_data(csurf);
  if (bits == NULL) return false;

  // Převod formátu z knihovny Cairo na Fog:
  switch (cairo_image_surface_get_format(csurf))
  {
    case CAIRO_FORMAT_ARGB32:
      // Výchozí formát v knihovně Cairo.
      fmt = Fog::Image::FormatRGB32; break;
    case CAIRO_FORMAT_RGB24:
      // Pozor, Cairo nepodporuje 24bitový formát,
      // jen ho tak označuje.
      fmt = Fog::Image::FormatRGB32; break;
    default: // Žádný vhodný
      return false;
  }

  // Vnutíme vektorovému grafickému kontextu data získané
  // z Cairo_surface_t.
  p->begin(bits,
    cairo_image_surface_get_width(csurf),
    cairo_image_surface_get_height(csurf),
    cairo_image_surface_get_stride(csurf), fmt);

  return true;
}
```

7.5 Integrace s knihovnou SDL

Knihovna SDL umožňuje vytvořit prakticky jakýkoliv formát pixelů. Pokud je tedy požadavek pro kreslení do SDL objektů pomocí knihovny Fog, je potřeba pohlídat si, aby cílový rastrový obrázek splňoval formát pixelů knihovny Fog.

Příklad vytvoření grafického kontextu nad SDL obrázkem:

```
bool beginSdlPainter(Fog::Painter* p, SDL_Surface* surf)
{
  int fmt = -1;

  // Převod SDL formátu na Fog:
  SDL_PixelFormat* f = surf->format;

  if (f->Rmask == 0x00FF0000 &&
      f->Gmask == 0x0000FF00 &&
      f->Bmask == 0x000000FF)
  {
    if (f->BitsPerPixel == 32)
    {
      fmt = (f->Amask == 0xFF000000) ? Image::FormatARGB32 : Image::FormatRGB32;
    }
    else if (f->BitsPerPixel == 24)
    {
      fmt = Image::FormatRGB24;
    }
  }
  // Neplatný formát...
  if (fmt == -1) return false;

  // Vnutíme vektorovému grafickému kontextu data z SDL_Surface:
  p->begin((uint8_t*)surf->pixels,
    surf->w, surf->h, surf->pitch, fmt)

  return true;
}
```

7.6 Souhrn

Z příkladů je možné vidět, že knihovna Fog umožňuje integraci praktiky s jakoukoliv knihovnou s kompatibilním formátem pixelů knihovny Fog. Ve všech ukázkových příkladech je směr integrace knihovny Fog nad jinou grafickou knihovnou. Je ale možné integrovat knihovnu Fog i opačně, například vnutit rastrové data z třídy *Fog::Image* grafické knihovně Cairo. Knihovny je tedy ve většině případů možné integrovat v obou směrech, což je určitě do budoucna velmi výhodné.

ZÁVĚR

Diplomová práce se zabývala zhodnocením současného stavu grafických knihoven, návrhem a implementací zcela nové grafické knihovny, která bude schopná využít instrukce moderních procesorů a bude podporovat paralelní vykreslování ve více vláknech. Primární ukazatel pro vývoj knihovny byl výkon.

Podařilo se navrhnout a implementovat knihovnu Fog v programovacím jazyce C++, která svou architekturou umožňuje nejen paralelní vykreslování, ale i věnovat se velmi podrobně optimalizacím na všech úrovních vykreslovacího procesu. Nejvíce času bylo věnováno na optimalizaci funkcí na rastrové úrovni, které spotřebují v mnoha případech největší množství procesorového času, dále na vývoj efektivní vícevláknové architektury.

Součástí diplomové práce bylo i porovnání rychlosti navržené grafické knihovny Fog s nejpoužívanějšími grafickými knihovnami v operačních systémech Windows (GDI+) a Linux (Cairo) popsanými v teoretické části práce. Bylo zjištěno, že navržená knihovna Fog je nejrychlejší knihovna v porovnání s ostatními grafickými knihovnami, a v případě použití čtyřjádrového nebo osmijádrového procesoru se výkon ostatních testovaných knihoven propadl až o jeden řád. Z navržené grafické knihovny Fog se tedy stala jedna z nejvýkonnějších grafických knihoven, na jejichž vývoj by bylo vhodné vynaložit další čas a prostředky.

Součástí diplomové práce je i mnoho obrázků, které popisují architekturu knihovny a demonstrují její grafický výstup. Všechny demonstrační obrázky grafického výstupu byly vykresleny knihovnou Fog a můžou být chápány jako praktické využití této knihovny pro vykreslování rastrové i vektorově orientované počítačové grafiky.

CONCLUSION

Thesis dealt with assessment of the current state of graphics libraries, design and implementation of an entirely new graphics library, which will be able to use the features available in modern processors and that will support parallel rendering in multiple threads. The primary indicator for the development was the performance.

Managed to design and implement a library in the C++ programming language called Fog, which allows not only to render graphics objects in parallel, but also allows to address all details in whole rendering process. Most time was devoted to optimize the raster-level functions that can take the most of CPU time and to implement efficient multithreading architecture.

Part of thesis was also to propose and compare the speed of implemented Fog library compared to graphics libraries in Windows (GDI+) and Linux (Cairo) that was described in the theoretical part. It was found that the proposed library Fog is the fastest library compared to other graphics libraries, and when benchmarked using quad core or eightfold core processor the speed difference was even bigger. The proposed graphical library Fog thus became one of the most efficient library for vector graphics and it would be appropriated to spend additional time and resources to improve it.

Part of thesis is a lot of images that describe the architecture of the library and demonstrate its graphical output. All demonstration images were drawn by Fog library and can be understood as the practical use of the library for rendering raster and vector based computer graphics.

SEZNAM POUŽITÉ LITERATURY

[1] BUTENHOF, D. Programming with POSIX Threads. Addison Wesley, 1997. ISBN 02-0163-392-2

[2] CLUGSTON, D. Member Function Pointers and the Fastest Possible C++ Delegates [online]. Dostupný z WWW:

http://www.codeproject.com/KB/cpp/FastDelegate.aspx

[3] INTEL, Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1 – Basic Architecture. Intel Corporation, 2008. 500 s.

[4] KALLIO, K. Scanline edge-flag algorithm for antialiasing. Dostupný z WWW:

http://mlab.taik.fi/~kkallio/antialiasing/

[5] NAVRÁTIL, P. Počítačová grafika a multimédia. Computer Media, Kralice na Hané, 2007. 112 s. ISBN 80-86686-77-9

[6] PORTER, T., DUFF, T. Compositing Digital Images,

Computer Graphics Volume 18, Number 3 July 1984 pp 253-259

[7] SHEMANAREV, M. Adaptive Subdivision of Bezier Curves, An attempt to achieve perfect result in Bezier curve approximation. 2005. Dostupný z WWW:

http://www.antigrain.com/research/adaptive_bezier/index.html

[8] ŽÁRA, J., LIMPOUCH, A., BENEŠ, B., WERNER, T. Pocítacová grafika - principy a algoritmy. Grada a.s., Praha, 1992. 472 s. ISBN 80-85623-00-5

[9] Anti-Grain Geometry [online]. Dostupný z WWW:

http://www.antigrain.com/

[10] AsmJit Library. Dostupný z WWW:

http://code.google.com/p/asmjit/

[11] BlitJit Library. Dostupný z WWW:

http://code.google.com/p/blitjit/

[12] Cairo Graphics homepage [online]. Dostupný z WWW:

http://cairographics.org/

[13] Cairo Graphics documentation [online]. Dostupný z WWW: http://cairographics.org/manual/

[14] Scalable Vector Graphics (SVG) 1.1 Specification [online]. Dostupný z WWW: http://www.w3.org/TR/SVG/

[15] Windows GDI+ [online]. Dostupný z WWW: http://msdn.microsoft.com/en-us/library/ms533798.aspx

[16] Windows API documentation - DibSection, dostupný z WWW: http://msdn.microsoft.com/en-us/library/dd183567(VS.85).aspx

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API Application Interface – Aplikační rozhraní (soubor struktur, tříd, funkcí).

CPU Central Processing Unit – Centrální výpočetní jednotka (procesor).

GDI Graphics Device Interface – Grafický subsystém ve Windows.

GTK The Gimp Toolkit – Multiplatformní GUI toolkit napsaný v jazyce C.

GUI Graphical User Interface – Grafické uživatelské rozhraní.

JIT Just In Time – termín používající se pro kompilaci části aplikace za běhu.

MIT Massachusetts Institute of Technology – Univerzita v USA a stejnojmenná licence používaná pro softwarové knihovny a aplikace.

MMX Matrix Math Extension – Rozšíření procesoru x86 o nové 64bitové registry a instrukce pro celočíselné výpočty.

PDF Portable Document Format – Formát pro přenos dokumentů.

POSIX Portable Operating System Interface for Unix – Standard definující API unixových operačních systémů.

RGB Red, Green, Blue – Barevné kanály v pixelu.

RGBA Red, Green, Blue, Alpha – Barevné kanály v pixelu včetně průhlednosti.

RTTI Run Time Type Information – Volitelná součást jazyka C++, která umožňuje získat typ instance třídy až za běhu aplikace.

SIMD Single Instruction Multiple Data – Paralelní zpracování dat v jedné instrukci.

SSE2 Streaming Simd Extensions 2 – Rozšíření procesoru x86 o nové 128bitové registry a instrukce pro práci s celočíselnými i reálnými čísly.

SVG Scalable Vector Graphics – Vektorový grafický formát definovaný konsorciem W3C (http://www.w3.org/TR/SVG/).

SEZNAM OBRÁZKŮ

SEZNAM TABULEK