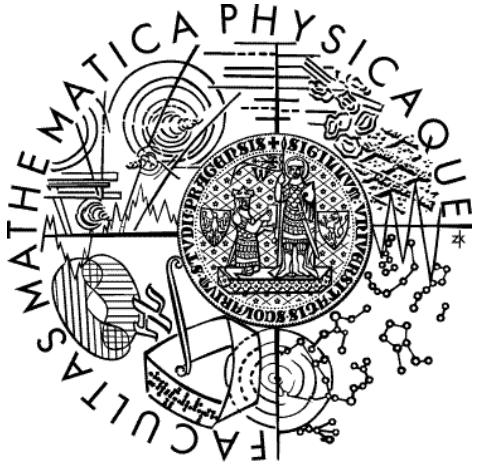
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

Bakalářská práce



Martin Kruliš

Vektorový editor se zaměřením na animace

Katedra Softwarového Inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D.

Studijní program: Informatika, Programování

2007

Děkuji panu RNDr. Filipu Zavoralovi, Ph.D. za odborné vedení mé práce, za rady a za čas, který mi během jejího vypracování věnoval. Rovněž děkuji lidem, kteří si tuto práci přečetli a pomohli mi s korekturami.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Martin Kruliš

Obsah

Název práce: Vektorový editor se zaměřením na animace

Autor: Martin Kruliš

Katedra: Katedra Softwarového Inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D.

E-mail vedoucího: filip.zavoral@mff.cuni.cz

Abstrakt: Předmětem této práce je implementace vektorového editoru s podporou animací. Program je určen pro specifickou oblast uživatelů, kteří potřebují vytvářet jednoduché animace a je pro ně zbytečné pořizovat komplexní komerční nástroje. V aplikaci bude možné navrhovat základní vektorové objekty, editovat je a nechat jejich body pohybovat po křivkách.

Aplikace se vyznačuje především svou modularitou a flexibilitou, které umožní její snadné budoucí rozšiřování a přizpůsobení požadavkům uživatele.

Klíčová slova: grafika, vektor, animace

Title: Vector editor for creating animations

Author: Martin Kruliš

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D.

Supervisor's e-mail address: filip.zavoral@mff.cuni.cz

Abstract: The object of this work is to design and implement a vector editor with animation support. This program is designated for a very specialized group of users that want to create simple animations but find buying complex commercial tools unnecessary. The application will allow user to design basic vector objects, edit them and set them into motion using curved trajectories.

Main features of the application are its modularity and flexibility, making it ready for future extending and customization.

Keywords: graphics, vector, animation

1. Úvod

Se vzrůstající kvalitou zobrazovacích zařízení a rostoucím výkonem osobních počítačů jsou kladeny stále větší nároky na grafický vzhled aplikací, prezentací a dalších elektronických děl. Ruku v ruce tomuto trendu rostou i požadavky grafiků na kvalitu a množství funkcí grafických editorů. Na poli aplikací pro rastrovou grafiku je situace celkem příznivá. V komerční sféře kraluje firma Adobe s aplikací Photoshop. V oblasti freewarových aplikací je momentálně na výsluní poměrně známý nástroj GIMP. Rastrová grafika je vhodná pouze pro některé druhy práce. V ostatních případech může být výhodnější pracovat s parametrickou reprezentací vykreslovaných objektů – tedy s vektorovou grafikou.

Mezi komerčními aplikacemi je nejznámější sada aplikací CorelDraw, které zvládají téměř vše, na co si může grafik vzpomenout. U freewarových aplikací je už situace o poznání horší. Většina editorů je zaměřena na určitou speciální oblast (např. Dia – kreslení schémat), nebo neumožňuje tvorbu animací (např. Vrr – vektorový editor, který vznikl na MFF-UK).

Cílem této práce je navrhnout a naprogramovat jednoduchý vektorový editor, který by uměl vytvářet obrázky i animace. Při zpracování bude kladen důraz především na nalezení efektivních algoritmů rastrovaní vektorových objektů a také na vytvoření dostatečně flexibilní struktury aplikace, která umožní její snadné budoucí rozšiřování.

Motivací pro vznik tohoto projektu je jednak nedostatek volně šiřitelných aplikací, které by pokryly mé speciální požadavky na tvorbu grafiky a také má touha proniknout lépe do problematiky dvojrozměrné grafiky a animací.

1.1 Rozdělení práce

Celá práce se skládá ze tří hlavních částí:

- popis použitého matematického aparátu, odvození a přehled použitých vzorců (kapitola 2.)
- návrh a formální definice použitých algoritmů a datových struktur (kapitola 3.)
- popis samotné implementace (kapitola 4. a 5.)

Potřebný matematický aparát vychází ze základů analytické geometrie, která je zahrnuta ve středoškolské výuce matematiky. Díky tomu jsem si dovolil omezit popis matematické části pouze na definice a vzorce. U složitějších vzorců je zpravidla uveden i postup odvození, avšak i u nich si čtenář vystačí se základními znalostmi.

Většina potřebných algoritmů a datových struktur již existuje a je dobře zdokumentována (viz např. [1]). Mou prací bylo především vybrat z existujících možností ty nejlepší, upravit je pro potřeby aplikace a efektivně je naimplementovat.

Na samotné implementaci byl nejdůležitější výběr použitého programovacího jazyka. Výběr jsem omezil na objektově orientované jazyky, protože objektová struktura aplikace nejlépe odráží skutečný stav řešeného problému a usnadňuje rozdělení aplikace do modulů. Rychlost je u náročných grafických výpočtů nezbytnou součástí, takže interpretované jazyky nepřichází v úvahu. Nakonec jsem zvolil C++, protože se jedná o moderní jazyk, který je poměrně hodně rozšířen a bude bez potíží přeložitelný na různých platformách.

Nyní se budu jednotlivým částem věnovat podrobněji v následujících kapitolách.

2. Použité matematické vzorce

V této kapitole hodlám uvést potřebné matematické vzorce a definice, které budou dále použity při implementaci vektorových objektů. Všeobecně známé vzorce si dovoluji uvést bez odvození. Čtenář by měl být obeznámen s matematikou na středoškolské úrovni.

2.1 Značení a zápis vzorců

V následujících výpočtech a vzorcích budu chápat bod v rovině jako vektor od počátku souřadnic. Jak bývá v počítačové grafice zvykem, počátek souřadnic je umístěn v levém horním rohu obrazu, hodnoty souřadnice x rostou směrem doprava a hodnoty y rostou směrem dolů.

Operace s body a operace s vektory jsou identické (tj. např. sčítání a odčítání je definováno po složkách). Pro bod A (resp. vektor $\vec{v}$) budu značit $A(x)$ (resp. $\vec{v}(y)$) hodnotu x-ové souřadnice bodu (resp. y-ové složky vektoru).

Parametrickou funkci křivky budu zpravidla značit $Q(t)$, kde t bude vždy reprezentovat proměnnou z intervalu $[0,1]$. Hodnota této funkce je bod (tedy prvek z množiny uspořádaných dvojic $\mathbb{R}\times\mathbb{R}$).

2.2 Výpočet reálných kořenů kvadratické rovnice

K výpočtu použiji všeobecně známý vzorec a kořeny kvadratické rovnice ve tvaru $ax^2+bx+c=0$ vypočítám jako:

$x_{1,2}=\frac{-b\pm\sqrt{D}}{2a}$, kde $D=b^2-4ac$ představuje *diskriminant* rovnice.

Pokud je $D\geq 0$ má rovnice pouze reálné kořeny. Komplexní řešení mě nebudou zajímat.

Při implementaci může dojít k nepříjemnostem v podobě násobení a dělení velice malým číslem (např. velice malá hodnota koeficientu a způsobí nepřesnosti ve výpočtech. Z tohoto důvodu se před samotným

výpočtem všechny koeficienty, které jsou v absolutní hodnotě menší nebo rovny 10^{-10}, zaokrouhlí na 0.

Speciální případ této rovnice $(a=0)$ je vyřešen zvlášť převodem na lineární rovnici $bx+c=0$, kde x se vypočítá jako:

$x=-\frac{c}{b}$ (pokud je $b=0$, rovnice nemá žádná zajímavá řešení)

2.3 Křivky

Základní představa křivek se kterou budu nadále pracovat:

Def*: Křivka* je dvourozměrný geometrický útvar definovaný funkcí $Q:[0,1]\rightarrow \mathbb{R}\times\mathbb{R}$, která pro danou hodnotu proměnné $t\in[0,1]$ vrátí souřadnice bodu křivky $Q(t)=(x,y)$, kde $x,y\in\mathbb{R}$.

Křivky jsou zpravidla ohraničené dvěma body (souřadnice těchto bodů odpovídají hodnotám parametrické funkce $Q(t)$ pro t rovné 0 a 1. Křivky mohou být dále určeny i jinými prvky (vektory, dalšími body apod.), ze kterých se určují hodnoty vnitřních parametrů funkce křivky.

Aby bylo možné s křivkou rozumně pracovat v rámci aplikace, musí existovat jednoduchý způsob, jak spočítat tyto její vlastnosti:

- délka křivky
- tečna v bodě t
- bounding-box (viz níže)

Def: *bounding-box* je nejmenší obdélník, který má strany rovnoběžné s osami x, y, a všechny body dané křivky leží uvnitř tohoto obdélníku.

Def: *rozkladový řádek* je řádek pixelů, jehož části se vyplňují při aplikaci algoritmu na vyplňování polygonu (kapitola 3.3.1). Pro účely této kapitoly postačí zjednodušení, kde si představíme rozkladový řádek jako přímku rovnoběžnou s osou x.

2.4 Úsečky

Úsečka je definována dvěma body a z geometrického hlediska se jedná o nejkratší spojnici těchto dvou bodů v rovině. Parametrická definice úsečky mezi body A, B je dána předpisem $Q(t)=A+(\vec{v})t$ kde $\vec{v}=B-A$ je směrnice a zároveň tečný vektor všech bodů úsečky.

Délka úsečky l je rovna vzdálenosti bodů A, B v Eukleidovské metrice:

$$l=\|A-B\|=\sqrt{(A(x)-B(x))^2+(A(y)-B(y))^2}$$

Bounding-box je určen pouze krajními body úsečky. Tzn. souřadnice levé strany je rovna $min(A(x), B(x))$ a souřadnice pravé strany je $max(A(x), B(x))$. Analogicky se určí hodnoty y-ových souřadnic horní a dolní strany. V případě že $A(x)=B(x)$ nebo $A(y)=B(y)$, bude mít obdélník nulovou šířku nebo výšku.

Úsečky jsou navíc používány k definici polygonů. U algoritmu vyplňování polygonu je třeba umět spočítat průsečík úsečky a rozkladového řádku (přímky rovnoběžné s osou x). Průsečík může být jeden, nebo žádný. Situace, kdy úsečka leží celá na rozkladovém řádku (a má tak nekonečně mnoho průsečíků), je brána stejně jako situace, kdy úsečka nemá průsečík žádný.

Pro danou úsečku AB (bez újmy na obecnosti budu předpokládat, že $A(y)\leq B(y)$) a rozkladový řádek y nejprve porovnám souřadnice. Je-li $A(y)\leq y\leq B(y)$ a zároveň $A(y)\neq B(y)$, pak má úsečka s rozkladovým řádkem právě jeden průsečík. Souřadnici x dopočítáme lineární interpolací:

$$x=A(x)+(y-A(y))\cdot\frac{A(x)-B(x)}{A(y)-B(y)}$$

Na ostatní případy se pohlíží, jako by úsečka s řádkem neměla průsečík žádný.

2.5 Kružnice a oblouky

Kružnice a jejich části (oblouky) představují jeden z typů implementovaných křivek. Oblouk je definován třemi body P_1, P_2 a P_3, přičemž P_1 a P_2 představují počáteční a koncový bod oblouku a P_3 je referenční bod, kterým oblouk prochází. V případě, že $P_1=P_2$, je oblouk uzavřen do celé kružnice a její průměr je definován úsečkou P_1P_3.

2.5.1 Převod na lepší reprezentaci

Interně je oblouk reprezentován souřadnicemi středu $C=[C(x),C(y)]$, poloměrem r a úhly α_1,α_2, které odpovídají pozicím hraničních bodů oblouku na kružnici. Problém může nastat pokud P_{1-3} leží na jedné přímce. V takovém případě je *r* nekonečno, předchozí datová reprezentace nebude fungovat a křivka se bude reprezentovat úsečkou mezi body (P_1,P_2).

Souřadnice středu se ze tří zadaných bodů spočítají jako průsečík os úseček P_1P_3 a P_2P_3. Obě osy si vyjádřím obecnou rovnicí přímky: $ax+by+c=0$, kde a a b jsou koeficienty normálového vektoru přímky a c je posunutí vůči počátku. Průsečík dvou přímek bude vyjádřen řešením soustavy rovnic těchto přímek, kde x a y jsou souřadnice hledaného průsečíku. Soustava nemusí mít řešení, pokud jsou obě osy rovnoběžné. V takovém případě leží střed zadané kružnice v nekonečnu a všechny tři definující body leží na jedné přímce.

Nejprve spočítám normálový vektor obou os. Protože je osa kolmá na úsečku a normála osy je kolmá na osu, bude normála rovnoběžná s úsečkou. Položím tedy normálu rovnou směrnici úsečky:

$$\vec{v}_1=P_3-P_1 \quad \text{a} \quad \vec{v}_2=P_2-P_3$$

Dále si spočítám pomocnou proměnnou b (kterou budu potřebovat pro další výpočty).

$$b=\vec{v}_2(x)\cdot\vec{v}_1(y)-\vec{v}_1(x)\cdot\vec{v}_2(y)$$

Pokud je $b=0$, pak jsou oba vektory lineárně závislé a zadané tři body leží v jedné přímce. Pomocné koeficienty c_1 a c_2 pak spočítám:

$$c_1=\vec{v}_1(x)\frac{P_1(x)+P_3(x)}{2}+\vec{v}_1(y)\frac{P_1(y)+P_3(y)}{2}$$

$$c_2=\vec{v}_2(x)\frac{P_2(x)+P_3(x)}{2}+\vec{v}_2(y)\frac{P_2(y)+P_3(y)}{2}$$

A ve výsledku vyjádřím souřadnice středu ze soustavy rovnic:

$$C(x)=\frac{\vec{v}_2(y)\cdot c_1-\vec{v}_1(y)\cdot c_2}{b} \quad \text{a} \quad C(y)=\frac{\vec{v}_1(x)\cdot c_2-\vec{v}_2(x)\cdot c_1}{b}$$

Poloměr snadno dopočítám pomocí Pythagorovy věty:

$$r=\sqrt{((C(x)-P_1(x))^2+(C(y)-P_1(y))^2)}$$

Jednoduchým výpočtem přes funkce $\arcsin$ (resp. $\arccos$) se určí úhly α_1, α_2 v radiánech. Na závěr ještě upravím úhly tak, aby počáteční úhel $\alpha_1 \in (0, 2\pi)$ byl vždy menší než koncový úhel $\alpha_2 \in (\alpha_1, \alpha_1+2\pi)$.

2.5.2 Další parametry kružnicového oblouku

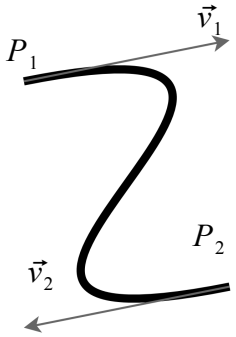
Délka oblouku je rovna obloukové míře (úhlu v radiánech) vynásobené poloměrem: $l=r(\alpha_2-\alpha_1)$

Pro další výpočty je potřeba převést parametr t na úhel α následujícím vztahem $\alpha=\alpha_1+t(\alpha_2-\alpha_1)$. Úhel α reprezentuje pozici na kružnicovém oblouku stejně dobře, jako parametr t, ale bude se s ním lépe pracovat v upravené reprezentaci kružnicového oblouku.

Tečný vektor $\vec{v}$ v bodě definovaném úhlem α se vypočítá jednoduše goniometrickými vzorci: $\vec{v}(x)=-\sin\alpha$, $\vec{v}(y)=-\cos\alpha$.

Bounding-box je v tomto případě minimálním obdélníkem obsahujícím počáteční a koncový bod a případně některý ze čtyř „krajních“ bodů na kružnici (body definované úhly $\{0, \pi/2, \pi, 3\pi/2\}$), pokud ovšem tyto body leží uvnitř oblouku (v intervalu $[\alpha_1, \alpha_2]$).

2.6 Fergusonovy kubiky



Obrázek 1:
Fergusonova kubika

Fergusonovy kubiky jsou jednoduché polynomiální křivky třetího stupně, podobné Beziérovým kubikám. Tyto křivky jsem zvolil, protože se s nimi snadno počítá, jejich vzorce jdou lehce derivovat a uživateli nabídnou dostatečně silné prostředky k tvorbě obrázků.

Fergusonovy kubiky jsou definovány počátečním a koncovým bodem P_1, P_2 a tečnými vektory v těchto bodech $\vec{v}_1, \vec{v}_2$. Z nich se snadno vyjádří parametrická rovnice pro výpočet bodů křivky (viz [1]):

$$Q(t)=P_1F_1(t)+P_2F_2(t)+\vec{v}_1F_3(t)+\vec{v}_2F_4(t)$$

kde $F_{1,2,3,4}(t)$ jsou tzv. *kubické Hermitovské polynomy* a mají tvar:

$$F_1(t)=2t^3-3t^2+1$$

$$F_2(t)=-2t^3+3t^2$$

$$F_3(t)=t^3-2t^2+t$$

$$F_4(t)=t^3-t^2$$

2.6.1 Další vlastnosti Fergusonových kubik

Délku Fergusonovy kubiky lze sice analyticky spočítat, avšak vyjádření příslušného vzorce a jeho rozumná implementace by byla příliš náročná. Pro výpočet délky jsem raději zvolil jednoduchý aproximační algoritmus, který lze použít na libovolnou křivku. Jeho popis se nachází v kapitole 3.1.

Z matematické analýzy víme, že tečný vektor funkce se v konkrétním bodě spočítá jako první derivace této funkce. V tomto případě je situace o trochu komplikovanější, protože výsledkem funkce $Q(t)$ je dvojrozměrná hodnota $[x, y]$. Naštěstí na sobě nejsou jednotlivé složky x a y závislé, a proto můžeme funkci Q bez problémů derivovat (podle t) po složkách. Výsledná funkce $Q'(t)$ pak vrátí hodnoty $[x', y']$, což jsou právě složky hledaného tečného vektoru.

V parametrické funkci $Q(t)$ jsou na proměnné t závislé pouze Hermitovské polynomy, a proto stačí zderivovat pouze je (na ostatní lze z hlediska derivování pohlížet jako na konstanty):

$$Q'(t)=P_1F_1'(t)+P_2F_2'(t)+\vec{v}_1F_3'(t)+\vec{v}_2F_4'(t)$$

$$F_1'(t)=6t^2-6t$$

$$F_2'(t)=-6t^2+6t$$

$$F_3'(t)=3t^2-4t+1$$

$$F_4'(t)=3t^2-2t$$

Kromě výpočtu tečného vektoru mi funkce $Q'(t)$ poslouží k vypočítání bounding-boxu. Na rozdíl od funkce jedné proměnné zde budou podezřelé z extrému všechny body, v nichž má tečný vektor jednu ze složek rovnou nule. Pokud je $\vec{v}(x)=0$, pak se jedná o kandidáta na extrém levé, nebo pravé strany, pokud je $\vec{v}(y)=0$, může se jednat o extrém určující horní, nebo spodní stranu bounding-boxu. Rovnici musím rozepsat postupně pro obě složky. Zde je rovnice pro složku x (druhou složku mohu odvodit analogicky záměnnou y za x).

$$P_1(x)(6t^2-6t)+P_2(x)(6t-6t^2)+\vec{v}_1(x)(3t^2-4t+1)+\vec{v}_2(x)(3t^2-2t)=0$$

$$t^2(6P_1(x)-6P_2(x)+3\vec{v}_1+3\vec{v}_2)+t(-6P_1(x)+6P_2(x)-4\vec{v}_1(x)-2\vec{v}_2(x))+\vec{v}_1=0$$

Pro přehlednost:

$$At^2+Bt+\vec{v}_1=0$$

$$A=6P_1(x)-6P_2(x)+3\vec{v}_1+3\vec{v}_2$$

$$B=6P_2(x)-6P_1(x)-4\vec{v}_1(x)-2\vec{v}_2(x)$$

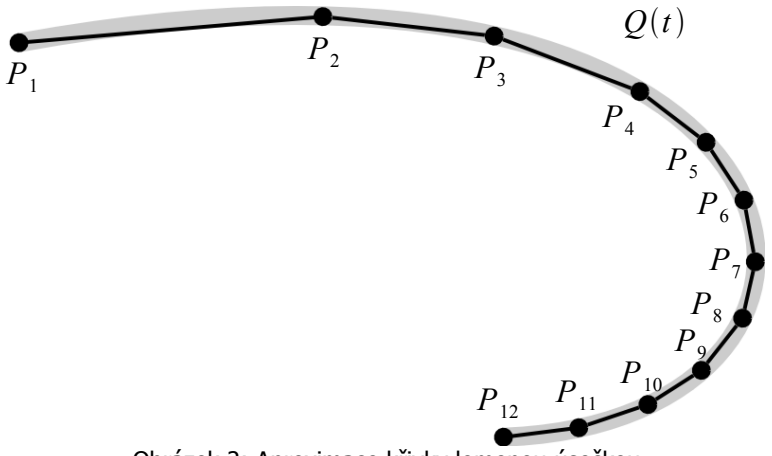
Tím jsem rovnici upravil na kvadratický tvar, který již umím vyřešit (viz kapitola 2.2). Kvadratická rovnice může mít až dva reálné kořeny $t_{1,2}$, samotné body podezřelé z extrému pak dostanu dosazením do funkce $Q(t_1)$ a $Q(t_2)$. Mezi podezřelé body musím také automaticky zahrnout počáteční a koncový bod.

3. Použité algoritmy a datové struktury

3.1 Obecné algoritmy používané u křivek

3.1.1 Aproximace křivky lomenou úsečkou

Algoritmy založené na práci s obecnými křivkami (zadanými pouze parametrickou funkcí $Q(t)$) jsou ve většině případů příliš náročné na výpočetní výkon. Pro tyto účely je vhodnější aproximovat křivku lomenou úsečkou (*polyline*) a použít algoritmy pracující pouze s úsečkami, které jsou výrazně rychlejší.



Obrázek 2: Aproximace křivky lomenou úsečkou

Zde popsaný algoritmus spočítá pro křivku zadanou $Q(t)$ posloupnost bodů P (jednotlivé body budu značit $P_1, P_2, \ldots, P_n$), které tvoří aproximační lomenou úsečku $P_1P_2, P_2P_3, \ldots, P_{n-1}P_n$.

Hlavní algoritmus:

vstup: funkce Q, *výstup:* seznam P

- připrav seznam P obsahující pouze $Q(0)$
- pomocný algoritmus (Q, $(0,1)$, P)
- přidej $Q(1)$ na konec P

Pomocný algoritmus:

vstup: funkce Q, interval (t_1, t_2), P, *výstup:* modifikovaný seznam P

- spočítej střed intervalu $s=(t_1+t_2)/2$
- spočítej tečný vektor v bodě s : $\vec{v}=Q'(s)$
- spočítej rozdíl δ mezi délkou úsečky $Q(t_1)Q(t_2)$ a součtem délek úseček $Q(t_1)Q(s)$ a $Q(s)Q(t_2)$.

$$\delta=\sqrt{(x_1-x_s)^2+(y_1-y_s)^2}+\sqrt{(x_s-x_2)^2+(y_s-y_2)^2}-\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

- pokud je odchylka δ (a tedy i zakřivení) malá a zároveň tečný vektor $\vec{v}$ je rovnoběžný s vektorem $Q(t_1)-Q(t_2)$, pak je aproximace dostatečná, algoritmus vrátí původní seznam P a skončí
- rekurzivní volání – pomocný algoritmus (Q, (t_1, s), P)
- přidej $Q(s)$ na konec P
- rekurzivní volání – pomocný algoritmus (Q, (s, t_2), P)

Hlavní výhodou tohoto algoritmu je možnost zvolit si požadovanou přesnost aproximace. Následně pak lze zvolit nízkou přesnost pro běžnou práci s obrázkem a naopak vysokou přesnost při ukládání do rastrového formátu. Nízká přesnost bude mít za následek snížení počtu aproximačních bodů a tomu odpovídající zvýšení rychlosti.

3.1.2 Zjištění délky obecné křivky

Délky některých jednoduchých křivek lze spočítat za pomoci matematické analýzy. Bohužel již výpočet délky Fergusonovy kubiky je poměrně náročný jak na sestavení vzorce, tak na jeho rozumnou implementaci, a proto zde použiji jednoduchý algoritmus, který spočítá délku pouze přibližně. Křivku si aproximuji dostatečně jemnou lomenou úsečkou (viz algoritmus 3.1.1) a následně sečtu délky dílčích úseček.

Nechť P je posloupnost bodů vrácených předchozím algoritmem a $N=\|P\|$ je počet prvků P, pak přibližnou délku l spočítám:

$$l=\sum_{i=0}^{N-1}\|P_i-P_{i+1}\|=\sum_{i=0}^{N-1}\sqrt{(x_i-x_{i+1})^2+(y_i-y_{i+1})^2}$$

3.2 Vykreslování křivek

Nejprve nadefinuji použité termíny.

Def: *Vzdálenost bodu* A od křivky Q budu značit $d(Q,A)$ a hodnota $d(Q,A)=min(\|Q(t)-A\|)\ \ t\in[0,1]$, tedy minimum ze vzdáleností A a všech bodů křivky Q.

Def: *Tloušťka křivky* je reálné číslo udávající sílu čáry, kterou bude křivka vykreslena na zobrazovací zařízení.

Def: *Obraz křivky* je množina všech bodů v rovině, jejichž vzdálenost od křivky je menší, nebo rovna polovině tloušťky křivky.

Rastrové obrazové formáty samozřejmě používají diskrétní rastrovou mřížku (složenou z obrazových jednotek – pixelů). Obraz křivky je proto ještě nutné rastrovat do této mřížky. Pro každý pixel se určí jak velká jeho část leží v obrazu křivky a podle toho se pro něj zvolí průhlednost při vykreslování (pixely ležící celé uvnitř budou vyplněny plnou barvou, pixely ležící vně budou naopak plně průsvitné).

K vykreslování křivek používám dva různé algoritmy v závislosti na požadovaném výsledku. Při vykreslování do náhledového okna postačí jen hrubé vykreslení křivky, avšak hlavním kritériem je rychlost, aby bylo možné efektivně v reálném čase obrázek editovat. Naopak při vykreslování obrázku do souboru v rastrovém formátu není rychlost příliš důležitá a důraz je kladen na kvalitu.

3.2.1 Vykreslování štětcem

Přesnější a pomalejší algoritmus používá k vykreslení stejný postup jako malíř, který kreslí štětcem na plátno. Vyplněný obrys štětce se vykreslí

v každém bodě křivky a tím vznikne velmi přesný a hladký obraz. Štětec jsem použil kruhový, protože přesně odpovídá požadovaným vlastnostem obrazu křivky (viz definice výše).

Pochopitelně není technicky možné nakreslit otisk štětce v každém bodě úsečky (neboť jich je nekonečně mnoho), ale je třeba zvolit vhodný vzorkovací krok, aby byl obraz hladký, ale zároveň aby algoritmus nebyl zbytečně pomalý. Nejvýhodnější se mi zdá vzorkovat křivku po půlpixelových krocích.

Algoritmus:

- spočítej délku křivky l (v pixelech)
- pro všechna $t=\frac{k}{2l}$, kde $k \in \{0 \ldots 2l\}$ nakresli kruh se středem v bodě $Q(t)$ a průměrem rovným tloušťce křivky.

3.2.2 Vykreslování křivky jako mnohoúhelníku

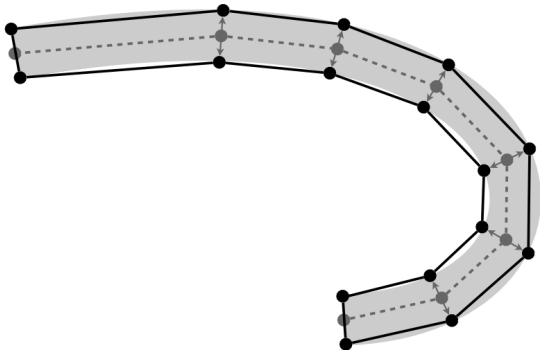
Rychlejší, avšak méně přesný algoritmus je založen na aproximaci obrazu křivky pomocí polygonu. Takto vzniklý polygon se pak snadno vykreslí algoritmem pro vyplňování ploch jednou barvou, který je popsán v kapitole 3.3.

Polygon je definován podobně jako lomená úsečka (tzn. jako posloupnost bodů P), ale navíc se předpokládá, že poslední bod je spojen s prvním.

Algoritmus vytvoření polygonu:

vstup: křivka Q, tloušťka w, *výstup:* seznam bodů polygonu P

- vyber body $t_1, t_2 \ldots t_n$, kde každý bod $t_i \in [0,1]$ a zároveň platí, že $t_1 < t_2 < \ldots < t_n$ a ke každému t_i přiřaď tečný vektor $\vec{v}_i = Q'(t_i)$
- všechny vektory $\vec{v}_i$ otoč o 90° a uprav jejich velikost na $w/2$
- pro všechna $i = 1..n$ vlož do seznamu P body $Q(t_i) + \vec{v}_i$
- pro všechna $i = n..1$ vlož do seznamu P body $Q(t_i) - \vec{v}_i$



Obrázek 3: Aproximace oblasti křivky polygonem

V algoritmu jsem neuvedl, jakým způsobem vybrat body $t_{1,} t_2 \dots t_n$. Jako nejlepší řešení se mi zde jeví použít upravený algoritmus pro aproximaci křivky lomenou čárou. Algoritmus by místo souřadnic bodů ($Q(t)$) ukládal do seznamu pouze parametr t.

Algoritmus rovněž nedodržuje kruhové zakončení křivky (které vyplývá z definice oblasti křivky). Protože se však jedná pouze o hrubý algoritmus, dovolím si tento problém ponechat bez dalšího řešení.

3.3 Vykreslování ploch

Vykreslování části roviny ohraničené křivkami (dále jen plochy) je jeden z klíčových algoritmů celého programu.

Def: *Hraniční křivkou* plochy rozumím jednu, nebo více křivek spojitě napojených v krajních bodech, kde první a poslední krajní bod těchto křivek jsou totožné.

Def: Bod je *vnitřní* vzhledem k nějaké hraniční křivce, pokud libovolná polopřímka s počátkem v tomto bodě má s hraniční křivkou lichý počet průsečíků. Bod je *vnější* vzhledem k nějaké hraniční křivce, pokud není vnitřní.

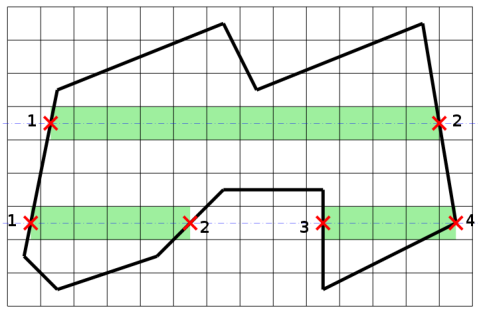
Def: *Plocha* určená hraniční křivkou je podmnožina roviny, jejíž všechny body jsou *vnitřními* vzhledem k dané hraniční křivce.

Výše uvedená definice vyplňované plochy se v literatuře [1] označuje jako metoda „sudá-lichá“.

Pozn: Jako průsečík polopřímky a křivky se nepočítá pouhý dotyk polopřímky. Protíná-li polopřímka hraniční křivku v bodě, kde křivka navíc protíná sama sebe, je započítán průsečík s každou dílčí částí křivky.

3.3.1 Vyplňování mnohoúhelníku (polygonu)

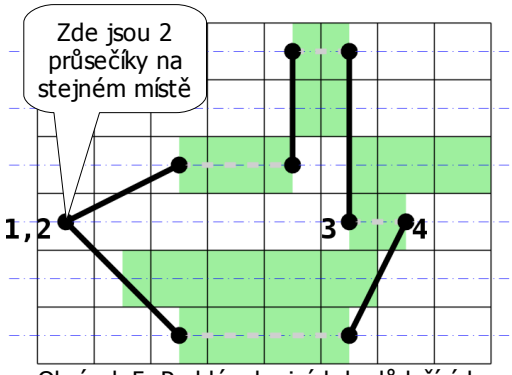
Speciálním případem hraniční křivky je mnohoúhelník (polygon). Vyplňování probíhá po rozkladových řádcích (scan-lines), které odpovídají řádkům pixelů.



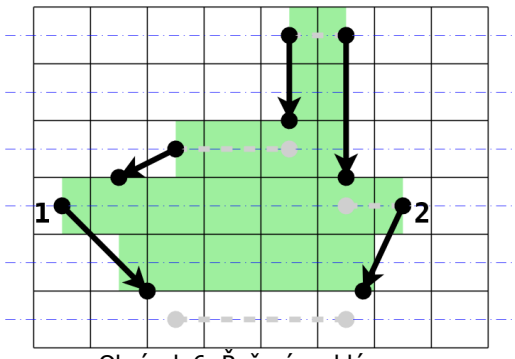
Obrázek 4: Vyplňování pomocí rozkladových řádků

Rozkladový řádek se prochází zleva doprava a zkoumají se průsečíky s polygonem. Ve výsledku je vyplněn vždy souvislý úsek pixelů od lichého průsečíku k následujícímu sudému průsečíku (tzn. $1-2$, $3-4$, …). Úsečky z polygonu, které leží celé na nějakém rozkladovém řádku odstraníme úplně, protože mají s tímto řádkem nekonečně mnoho průsečíků, a tak je nemůžeme rozumně zpracovat.

Dále je potřeba vyřešit problém s krajními body jednotlivých úseček. Pokud takový bod leží přímo na rozkladovém řádku, je započítán jako dva průsečíky (za každou úsečku, ve které leží), viz obrázek 5. Problém vyřešíme tak, že všechny úsečky orientujeme shora dolů a spodní bod posuneme nepatrně výš (viz obrázek 6). Další informace naleznete např. v literatuře [1].



Obrázek 5: Problém krajních bodů ležících na rozkladových řádcích



Obrázek 6: Řešení problému

Algoritmus vyplňování:

vstup: posloupnost bodů polygonu P, *výstup:* obrazová data

- urči boudning-box B polygonu (hodnoty $B.top$ a $B.bottom$ určují rozsah rozkladových řádků, které zasahují do polygonu)
- pro každou úsečku polygonu P zjisti průsečíky s rozkladovými řádky a každý průsečík zatřiď na příslušný řádek podle x-ové souřadnice
- na každém rozkladovém řádku vyplň všechny úseky mezi lichým a následujícím sudým průsečíkem

Průsečíky úsečky AB a rozkladových řádků zjistím takto:

vstup: souřadnice bodů úsečky AB, *výstup:* seznam průsečíků S

- připrav prázdný seznam S
- úsečku AB orientuj tak, aby $A_y < B_y$, pokud je $A_y = B_y$, tak skonči (a vrať prázdný seznam S)
- spočítej hodnoty $dy = B_y - A_y$ a $dx = (B_x - A_x)/dy$
- polož $x = A_x$ a $y = A_y$
- dokud je $dy > 0$, opakuj
  - přidej do seznamu S průsečík $[x, y]$
  - y zvětši o 1, k x přičti dx
  - dy sniž o 1

Algoritmus bylo samozřejmě potřeba ještě trochu přizpůsobit, aby pracoval dobře i s reálnými hodnotami proměnných. Implementace se liší i v několika dalších drobných detailech, avšak idea algoritmu odpovídá výše uvedenému popisu.

3.3.2 Vyplňování složitějších ploch

Vyplňování složitějších ploch je o mnoho náročnější na výpočetní sílu i na samotnou implementaci. Nejjednodušší a nejefektivnější řešení, které se nabízí, je aproximace hraniční křivky polygonem a převedení situace na předchozí případ.

Polygon se jednoduše poskládá z lomených čar, kterými postupně aproximuji jednotlivé křivky tvořící hranici oblasti. Algoritmus na aproximaci křivky lomenou úsečkou je popsán v kapitole 3.1.1.

3.4 Zpětná identifikace objektů z obrázku

Uživatelské grafické rozhraní potřebuje ke své plnohodnotné práci umět manipulovat s vektorovými objekty myší. Je tedy nezbytné, aby existoval zpětný mechanismus identifikace vektorových objektů z rastrových dat. Jinými slovy pro každý pixel je třeba umět určit, který vektorový objekt jej vygeneroval.

Nejjednodušší by bylo použít datovou reprezentaci, kde by si každý pixel pamatoval ukazatel na objekt, ze kterého byl vykreslen. Tento přístup by nám zajistil konstantní časovou složitost při zpětném vyhledávání objektů. Ukazatele uložené společně s pixely by se snadno aktualizovaly i slévaly z více vrstev, avšak paměťové nároky aplikace by se efektivně zdvojnásobily (za předpokladu, že ukazatel zabere stejně paměti jako informace o barvě pixelu).

Bude-li objektů řádově méně než pixelů na obrázku, stane se tato reprezentace velice neúsporná. Přesto, že časová složitost má zpravidla přednost před paměťovou, dovolím si zde udělat výjimku a navrhnu paměťově úspornější datovou strukturu, která bude mít nepatrně horší časovou složitost.

3.4.1 Popis datové struktury

Základem datové struktury je jednorozměrné pole řádků (jak název napovídá pole je indexované souřadnicí y). Jeden řádek představuje posloupnost n_y záznamů, kde každý záznam je složen z hodnoty souřadnice x a ukazatele na vektorový objekt. Záznamy na řádku jsou setříděny vzestupně podle souřadnice x a hodnoty x jsou v rámci řádku unikátní.

Pro přehlednost si očísluji záznamy na řádku $1,\dots,n_y$. Symbolem x_i budu značit x-ovou souřadnici a $\vec{u}_i$ ukazatel na objekt i-tého záznamu. Pixely na daném řádku v intervalu $[x_i, x_{i+1}-1]$ byly vygenerovány objektem $\vec{u}_i$. Pokud je $\vec{u}_i$ `NULL`, pak daný interval pixelů nepatří žádnému objektu (pixely jsou průhledné, resp. přebírají barvu pozadí). Je-li $x_1>0$, pixely v intervalu $[0,x_1-1]$ se považují za prázdné (stejně, jako v případě `NULL` ukazatele). Je-li $x_n \le \textit{šířka řádku}$, pixely v intervalu $[x_n, \textit{šířka řádku}]$ patří objektu $\vec{u}_n$.

Posloupnost záznamů lze reprezentovat několika způsoby. Abych ušetřil co nejvíce paměti, budu řádek ukládat jako pole dvojic (souřadnice x, ukazatel na objekt $\vec{u}$) setříděné podle x. Ostatní možné reprezentace (např. setříděný seznam, BVS) by zbytečně zabraly další paměť na pomocné ukazatele. Navíc půjde nad polem dobře realizovat binární vyhledávání.

3.4.2 Algoritmus pro vyhledávání podle souřadnic

Algoritmus vyhledání ukazatele na vektorový objekt ležící na daných souřadnicích již vyplývá z definice datové struktury.

vstup: souřadnice $[x, y]$, *výstup:* ukazatel na hledaný objekt $\vec{u}$

- podle souřadnice y vyber řádek
- binárním vyhledáváním nalezni na zvoleném řádku záznam i, jehož hodnota x_i je největší možná splňující podmínku $x_i \leq x$
    - pokud takový záznam i existuje, vrať $\vec{u}_i$
    - jinak vrať `NULL`

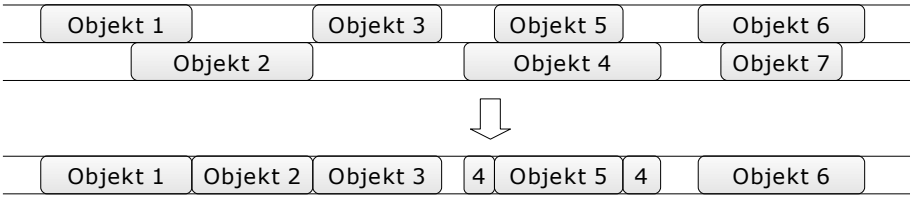
Hledaný řádek y naleznu v poli řádků v konstantním čase. Vyhledání záznamu na řádku, který je menší, nebo roven x, zabere algoritmem binárního vyhledávání maximálně $O(\log_2 n_y)$, kde n_y je počet záznamů na řádku.

Číslo n_y je shora omezeno šířkou řádku. Pokud navíc omezíme šířku řádku číslem 2^{32}, bude horní odhad na počet operací $\log_2 2^{32} = 32$, takže tento algoritmus nebude v reálné implementaci o mnoho pomalejší, než přímé vyhledávání ukazatelů v dvourozměrném poli.

3.4.3 Naplnění zvolené struktury daty

Data se vkládají společně s pixely při renderování obrázku a struktura si je sama zatřiďuje. Každá vrstva si uchovává společně s vykresleným rastrovým obrázkem také jednu datovou strukturu pro zpětné vyhledávání objektů. Data a obrázky z jednotlivých vrstev se následně slévají. Díky tomuto postupu se znovu překreslují pouze vrstvy, které se změní (podobně jako samotné obrázky).

Algoritmy pro vkládání a slévání jsou si velice podobné. Nejdůležitějším článkem obou algoritmů je operace slévání dvou řádků, kterou nyní popíšu.



Obrázek 7: Algoritmus slévání dvou řádků

Algoritmus slévání dvou řádků postupně prochází první řádek zleva do prava. Je-li na prvním řádku objekt, je zkopírován do výsledného řádku a všechny objekty ležící ve stejném rozsahu na druhém řádku jsou přeskočeny. Pokud je na prvním řádku prázdno, algoritmus kopíruje do výstupu obsah druhého řádku (ať už objekty, nebo prázdné místo). Při implementaci je třeba ošetřit některé speciální případy, jako např. objekty 4 a 5 na obrázku výše. Objekt 5 z prvního řádku překryje pouze část objektu 4, takže ve výsledku se objekt 4 objeví dvakrát.

Slévání dvou řádků funguje přirozeně v lineárním čase. Kompletní slití dvou struktur zabere nejvýše $O(w \cdot h)$, kde w je šířka a h je výška obrázku v pixelech (což je stejný čas, jaký potřebují ke slití pixely z odpovídajících vrstev).

4. Struktura aplikace a detaily implementace

4.1 Rozbor a návrh řešení

Při návrhu samotné aplikace bylo třeba učinit několik zásadních rozhodnutí a zohlednit některé požadavky:

- vybrat vhodné knihovny pro grafické uživatelské rozhraní
- rozhodnout, které části kódu budou dynamicky rozšiřitelné a navrhnout rozhraní pro zásuvné moduly
- navrhnout mechanismy pro urychlení vykreslování
- navrhnout datovou reprezentaci vektorových objektů tak, aby umožňovala animace
- vhodné rozdělení kódu aplikace

4.1.1 Výběr grafických knihoven

Výběr grafických knihoven je naprosto zásadní, protože se od něho odvíjí návrh velké části aplikace. Automaticky jsem z výběru vyloučil knihovny, jejichž licence by bránila volnému užívání této aplikace. Rovněž jsem vyloučil knihovny zaměřené čistě na jednu platformu, neboť mé ambice směřují k multiplatformnímu projektu.

Tím jsem výběr efektivně zúžil na knihovny GTK+ [4] a wxWidgets [8]. Pro obě knihovny existuje aplikační rozhraní v C++ a obě knihovny jsou použity v řadě úspěšných projektů. Nakonec padla volba na GTK+, protože se jedná o knihovny vyvíjené přímo pro grafickou aplikaci GIMP a nabízí tedy širokou podporu práce s rastrovou grafikou.

4.1.2 Zásuvné moduly

Většina moderních aplikací používá zásuvné moduly. Je to jednoduchý a poměrně efektivní mechanismus, kterým lze dosáhnout rozšiřitelnosti

hotové aplikace a poskytnout uživateli volbu právě těch modulů, které potřebuje. Na druhou stranu je tato krásná myšlenka vykoupena podstatně složitějším návrhem aplikace, neboť každý typ zásuvného modulu musí mít vlastní aplikační rozhraní a v řadě případů musí být moduly schopny komunikovat i mezi sebou.

Při návrhu aplikace jsem se rozhodl ponechat nejsložitější věci, jako jsou vektorové objekty a třídy pro práci s obrázky, uvnitř aplikace. Naopak části kódu, které jsou relativně nezávislé (nástroje uživatelského rozhraní, rastrové filtry a třídy pro ukládání souborů), jsem umístil do zásuvných modulů.

Zásuvné moduly jsou řešeny standardním způsobem pomocí dynamicky linkovaných knihoven. Přesný popis rozhraní a další informace jsou uvedeny v kapitole 4.7.

4.1.3 Mechanismy pro zrychlení vykreslování

Rychlost odezvy patří mezi nejdůležitější parametry interaktivních aplikací. Vykreslení velkého množství vektorových objektů může být časově poměrně náročné a nebylo by vhodné, kdyby se provádělo i v případech, kdy není nezbytně třeba, jako např. při posunu náhledového okna, nebo změně přiblížení. Bylo tedy nezbytné zavést mechanismy, které sníží počet rastrování vektorových objektů.

Aktuálně zobrazovaný obrázek a každá vrstva obsahují vyrovnávací paměť, do které se vektorový obraz renderuje. Při vykreslování do okna aplikace se pak berou již rastrová data z vyrovnávací paměti. Díky tomu jsou operace jako posun náhledu nebo změna přiblížení mnohem rychlejší.

Při editaci objektů, které se nachází pouze v jedné vrstvě, je překreslen pouze rastrový obrázek ve vyrovnávací paměti dané vrstvy. Rastrové obrazy ostatních vrstev zůstanou zachovány a celkový obraz se vytvoří pouhým složením jednotlivých vrstev. Podrobněji je práce s vyrovnávací pamětí popsána v kapitole 4.7.

4.1.4 Realizace animací

Při návrhu datových struktur bylo potřeba zohlednit způsob vytváření animací. Z nepřeberného množství možností jsem následně vybral tu, která se mi zdála implementačně nejjednodušší a zároveň funkční. Ze všech vektorových objektů se animují pouze body. Křivky jsou definované body a překreslují se v závislosti na pohybu bodů. Analogicky plochy jsou závislé na křivkách a vykreslují se podle nich.

Tato reprezentace s sebou nese některá úskalí, která je třeba řešit. Největším problémem jsou křivky, které jsou definovány i jinými parametry, než body (např. Fergusonovy kubiky jsou definovány ještě tečnými vektory v koncových bodech). Tento problém lze vyřešit např. tak, že bych zakončení tečných vektorů reprezentoval rovněž body. Body se mohou animovat a tím by bylo možné plynule měnit i tečné vektory v koncových bodech (Fergusonovy kubiky se budou vlastně reprezentovat Beziérovými kubikami).

Dalším úskalím jsou animace skalárních parametrů vektorových objektů (tloušťka čar, barva apod.). Současná implementace povoluje pouze konstantní hodnoty těchto parametrů (tzn. např. tloušťka čáry je v celém animačním rámci konstantní). Tento problém by se dal hladce vyřešit například tak, že by hodnota každého skalárního parametru byla definována funkcí o jedné proměnné (v implementaci představované funktorem). Technická realizace by byla velice jednoduchá, avšak příslušné grafické uživatelské rozhraní by si vyžádalo nemalé množství práce.

Výše zmíněné problémy jsem se rozhodl prozatím neimplementovat. Animace samotné zapouzdřují tzv. rámce, jejichž popis naleznete v kapitole 4.2.2.

4.1.5 Kód aplikace

Aplikace je implementována v jazyce C++ s použitím standardních knihoven STL [6] a grafických knihoven Gtk+ [4] (s aplikačním rozhraním

Gtkmm [5]). Celý kód (kromě několika málo globálních funkcí) je zabalen do tříd. Každá entita vyskytující se v aplikaci má korespondující třídu. Aplikace je rozdělena na čtyři logické celky:

- datová reprezentace dokumentu (nosná konstrukce)
- vektorové objekty a výpočetní modely (generátory)
- grafické uživatelské rozhraní
- zásuvné moduly

Vektorové objekty jsou základním stavebním kamenem celé aplikace. Zapouzdřují vzorce a algoritmy z kapitol 2. a 3. do tříd jazyka C++, aby bylo možné je jednoduše používat ve všech částech aplikace. Objekty jsou celkem tří typů: body, křivky a plochy a jejich bližší popis naleznete v kapitole 4.3.

S vektorovými objekty je úzce provázán systém pro rastrování a správu vrstev, rámců a obrázků. Celý obrázek je zapouzdřen do jedné třídy, ve které jsou uloženy základní údaje. Obrázek obsahuje jeden, nebo i více rámců. Rámce představují buď jednotlivé snímky animace, nebo jednu animační sekvenci. Každý rámec může být navíc rozdělen na více vrstev. Vrstvy mají stejný význam jako v běžných grafických aplikacích, jako je Photoshop nebo GIMP. Jednotlivé vrstvy teprve obsahují samotné vektorové objekty. Celý aparát je poměrně složitý a je podrobněji popsán v kapitole 4.2.

Grafické uživatelské rozhraní má na starosti vykreslování obrázku a odchytávání událostí myši a klávesnice, které přeposílá aktivnímu nástroji. Nástroje jsou do aplikace vkládány pomocí zásuvných modulů, což umožňuje jejich snadné rozšiřování. Knihovny GTK+ neumožňují pohodlnou implementaci MDI a problém otevírání více dokumentů současně je řešen stejně, jako v aplikaci GIMP tj. každý obrázek je otevřen v samostatném okně. Detailní popis GUI naleznete v kapitole 4.5.

4.2 Datová reprezentace dokumentu (obrázku)

Obrázek je rozložen na několik logických částí. Hlavní třídou, která reprezentuje obrázek je **`CImage`**. Obrázek se skládá z rámců a animačních sekvencí, které zapouzdřuje třída **`CFrame`**. A konečně každý rámec může obsahovat několik vrstev v podobě tříd **`CPlane`**.

Výše uvedené rozdělení tříd přináší řadu výhod. Obrázek se může skládat z více scén (rámců), které obsahují naprosto oddělené vektorové objekty. Obdobně jeden rámec může obsahovat několik vrstev a tím docílit správného překrývání objektů, případně některé vrstvy částečně zprůhlednit atd.

Nyní k jednotlivým třídám podrobně:

4.2.1 Obrázky (třída CImage)

Hlavní třída reprezentující celý obrázek. V členských proměnných uchovává jen nejdůležitější informace platné pro všechny rámce (rozměry obrázku v pixelech, název obrázku a uživatelský komentář). Dále obsahuje seznam rámců (STL-list ukazatelů na **`CFrame`**) a seznam použitých rastrových filtrů (STL-list ukazatelů na **`CFilter`**). Každý obrázek musí obsahovat alespoň jeden rámec. Rastrové filtry jsou speciální třídy, které provádí operace nad obrázkem po jeho vykreslení do rastrového formátu (podrobněji jsou popsány v kapitole 4.6.3).

4.2.2 Rámce (třída CFrame)

Třída **`CFrame`** představuje jeden rámec (snímek animace), případně multi-rámec (několik snímků s jednoduchou animací).

Jednoduchý rámec (*single-frame*) se nijak neliší od jednoho vektorového obrázku. S ostatními rámci má společnou pouze velikost (a případně rastrové filtry), avšak všechny vektorové objekty žijí pouze v tomto rámci.

Multi rámec (*multi-frame*) je téměř totožný s jednoduchým rámcem, avšak do výsledné animace se vykreslí jako dva a více snímků. V multi-

rámci se mohou používat animované body (potomci třídy **`CMovingPoint`**). Na prvním a posledním snímku multi-rámce jsou pohyblivé body vykresleny v krajních bodech. Na vnitřních snímcích jsou body umístěny do pozic, které odpovídají pozici v poměrném čase na trajektorii tohoto bodu. Výpočet mohou ještě ovlivnit časové funktory (viz dále).

Rámec obsahuje seznam globálních bodů (STL-list ukazatelů na **`CPoint`**), seznam jednotlivých vrstev (STL-list ukazatelů na **`CPlane`**), obrazový buffer (**`Gdk::Pixbuf`**), ve kterém se uchovává poslední vykreslení obrázku, a **`CResolver`**, což je třída, která slouží pro zpětnou identifikaci vektorových objektů v obrázku.

Je-li **`CFrame`** multi-rámcem, obsahuje navíc informace o počtu vnitřních rámců (snímků), index aktivního snímku (který je právě vykreslen a editován) a pole náhledových obrázků na jednotlivé snímky (*thumbnails*), které používá GUI k zobrazování náhledů.

Při vykreslování rámce metoda `renderPreview` nejprve nastaví všem objektům na všech vrstvách „správný čas“ metodou `CObject::setTime`. Čas se určí jako podíl pořadí aktivního snímku rámce a počtu snímků v rámci (tzn. první snímek má čas $t=0$, atd. až poslední snímek má čas $t=1$). Navíc může rámec obsahovat ukazatel na časový funktor (potomek třídy **`CTimeFunctor`** – viz kapitola 4.6.5), který slouží jako modifikátor průběhu času. Z matematického hlediska si lze časový funktor představit jako funkci $f:[0,1]\rightarrow[0,1]$. Tato funkce může změnit lineární běh času v multi-rámci na jakýkoli jiný (např. na kvadratický a simulovat tak zrychlení při pohybu bodů).

Po nastavení času vnitřním objektům metoda `renderPreview` překreslí všechny vrstvy a jejich obrazy (včetně příslušných objektů **`CResolver`**) poskládá do výsledného obrazu (a výsledného resolveru).

4.2.3 Vrstvy (třída CPlane)

Tato třída představuje jednu vrstvu (*layer*) rámce. Každý rámec obsahuje alespoň jednu vrstvu. Třída **`CPlane`** zároveň funguje jako kontejner pro všechny vektorové objekty a lokální body.

V členských proměnných uchovává informace o vrstvě (viditelnost, průhlednost a název) a také seznamy bodů, křivek a oblastí (STL-listy ukazatelů na **`CPoint`**, **`CCurve`** a **`CArea`**), které leží v této vrstvě.

Při vykreslování metodou `renderPreview` používá vrstva vyrovnávací paměť v podobě vlastního obrazového bufferu (**`Gdk::Pixbuf`**). Do tohoto bufferu se obraz vrstvy vykreslí a následně se rastrová data již jen kopírují (není třeba neustále vykreslovat vektorové objekty). Dojde-li ke změně libovolného objektu, je tento objekt povinen zavolat metodu `setRedraw` vrstvy, ve které leží, a při dalším požadavku na vykreslení si vrstva zaktualizuje obrázek ve vyrovnávací paměti.

Vektorové objekty jsou vykreslovány v opačném pořadí, než jsou uloženy v seznamech (tj. objekty uložené poslední budou vykresleny jako první a z hlediska uživatele budou nejdále v pozadí). Nejprve se vykreslují oblasti a následně jsou přes ně překresleny křivky. Body nejsou vykreslovány, neboť se nejedná o viditelné objekty (body může vykreslovat pouze GUI a to navíc ve vlastní režii).

Při smazání vrstvy zajistí destruktor uvolnění všech objektů, které ve vrstvě leží.

4.3 Vektorové objekty (model, hierarchie, vztahy)

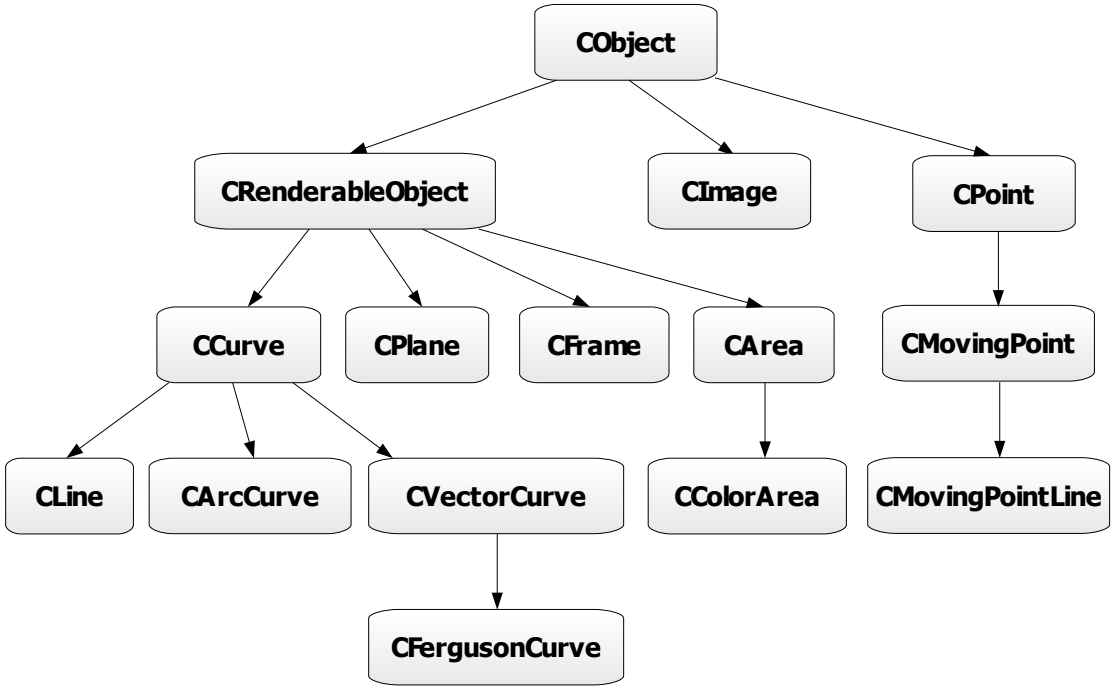
Každý vektorový objekt, který může uživatel v aplikaci používat, je zabalen do vlastní třídy. Vzhledem k povaze těchto objektů mají třídy hierarchické uspořádání, které je zobrazeno na obrázku 8.

Třída **`CPoint`** a třídy, které jsou na obrázku zakresleny jako listy stromu, jsou finální a jejich instance se používají. Ostatní třídy jsou

abstraktní a to jak z hlediska logického návrhu, tak z hlediska implementace (mají čistě virtuální metody).

Kromě vektorových objektů jsou do tříd zabaleny i obrázky, rámce a vrstvy, neboť mají s vektorovými objekty některé společné vlastnosti a metody (např. název objektu).

Návaznosti a vlastnosti jednotlivých objektů jsou podrobně popsány v kapitole 4.3.1.



Obrázek 8: Hierarchie vektorových a datových objektů

4.3.1 Přehled vektorových objektů

CObject

Definuje společnou abstraktní třídu pro všechny objekty na obrázku 8. Jako členskou proměnnou uchovává *jméno objektu* (řetězec, kterým uživatel může rozlišovat instance jednotlivých objektů).

Dále definuje virtuální metodu `getTyp`, která vrací typ objektu. Typ objektu určuje jeho třídu a všechny třídy odvozené od **`CObject`** mají vlastní typ. Tuto techniku používám jako náhradu za běhové informace (RTTI) a kontrolu přetypování (`dynamic_cast`). Uvnitř

třídy **CObject** je také definován výčet známých konstant, které `getTyp` rozlišuje.

Všichni potomci této třídy musí mít definované rozhraní pro ukládání a načítání z/do persistentní paměti (souboru). Za tímto účelem definuje **CObject** virtuální metodu `serialize`, která zajišťuje obousměrnou práci (tzn. ukládání i načítání) se souborem. Každý odvozený objekt může tuto metodu předefinovat (pokud je to potřeba), v jejím těle nejprve zavolat metodu předka a následně uložit/načíst vlastní členské proměnné. Tento princip velmi zjednodušuje návrh nestrukturovaného (sekvenčního) ukládání a načítání (každý objekt manipuluje pouze se svými daty). Metoda `serialize` očekává jako parametr referenci na třídu **CArchive**, která zapouzdřuje práci se soubory (viz kapitola 4.6.1).

Pro účely animací **CObject** definuje virtuální metody `setTime` a `moveBy`, které neprovádí žádnou činnost, ale odvozené objekty je mohou předefinovat. Metoda `setTime` je určena pro nastavení časového údaje pro animaci. Jako parametr obdrží reálné číslo $t \in [0,1]$, které představuje relativní čas v multi-rámci, do něhož objekt patří. Hodnota $t = 0$ odpovídá prvnímu snímku a hodnota $t = 1$ poslednímu. Animovaný objekt tak může nastavit své parametry, aby odpovídaly dané fázi animace (např. animovaný bod přepočítá svou polohu tak, že dosadí hodnotu t do funkce $Q(t)$ svojí trajektorie).

Metodu `moveBy` mohou předefinovat objekty, které lze posouvat (tj. **CPoint**, **CCurve** a **CArea**). Kromě předpokládaných parametrů posunu (hodnoty dx a dy) obdrží metoda také parametr t, který má stejný význam jako u `setTime`. Animovaný objekt se tak může rozhodnout, zda má editovat počátek, konec, nebo jiný úsek své trajektorie.

CObject má virtuální destruktor, aby bylo možné bez obav likvidovat i odvozené objekty.

CRenderableObject

Představuje třídu objektů, které lze vykreslit do obrazové paměti. Všechny vektorové objekty kromě bodů jsou potomky této třídy. Rovněž vrstvy a rámce se z hlediska aplikace považují za vykreslitelné objekty.

Třída definuje dvě abstraktní metody, které musí implementovat všechny odvozené třídy: `renderPreview` a `render`. Tyto metody vykreslí objekt na připravené plátno (objekt **`Gdk::Pixbuf`** z knihoven GDK) .

První metoda (`renderPreview`) vykresluje objekt pouze do náhledového okna. Metoda nemusí být přesná, ale měla by pracovat velmi rychle, aby bylo možné obrázek editovat v reálném čase. Tato metoda má také za úkol naplnit data do objektu **`CResolver`**, který GUI využívá pro zpětnou identifikaci objektů z obrázku (viz kapitola 4.4).

Druhá metoda (`render`) se naopak použije při finálním vykreslení objektu (např. při ukládání do souboru v rastrovém formátu). Metoda může být pomalejší, avšak měla by objekt vykreslit s maximální přesností.

Při vykreslování náhledu (`renderPreview`) je povoleno, aby odvozený objekt rastrová data načítal z vyrovnávací paměti (tj. vytvořil si vlastní instanci třídy **`Gdk::Pixbuf`**, do které objekt jednou vykreslí a následně jej bude jen kopírovat z této mezipaměti). Objekt musí umět rozhodnout, zda je možné použít data z vyrovnávací paměti, nebo je třeba objekt znovu vykreslit (pokud se data změnila). Za tímto účelem definuje **`CRenderableObject`** metody `setRedraw` a `redrawNeeded`, které nastaví (resp. zjistí příznak), zda je potřeba překreslit data ve vyrovnávací paměti. Odvozené objekty musí povinně volat metodu `setRedraw`, pokud jakkoli změnily vnitřní data objektu. Tato metoda je navíc virtuální,

takže ji může odvozený objekt předefinovat, avšak odvozené varianty by měly ve svém těle volat originální metodu `CRenderableObject::setRedraw`.

CPoint

Tato třída reprezentuje objekty bodů. **`CPoint`** není odvozen od **`CRenderableObject`**, z čehož také vyplývá, že se automaticky nevykresluje. Body zobrazuje pouze GUI ve vlastní režii, pokud je to potřeba.

Ve členských proměnných uchovává bod své souřadnice x, y (jako reálné hodnoty) a také seznam všech křivek (STL-list ukazatelů na **`CCurve`**), se kterými bod inciduje. V případě jakékoli změny souřadnic zavolá bod pro každou křivku v seznamu její metodu `setRedraw` a informuje tak křivky, že došlo ke změně. Pokud je **`CPoint`** zničen, jeho destruktor zničí také všechny křivky s ním incidující (jinak by nebyly korektně definovány).

V aplikaci se rozlišují body dvou druhů: *lokální* a *globální*. Lokální body se vyskytují pouze v jedné vrstvě a jejich vlastníkem je třída **`CPlane`**. Globální body prochází všemi vrstvami daného rámce a mohou incidovat s křivkami na všech vrstvách. Jejich vlastníkem je třída **`CFrame`**.

CMovingPoint

Definuje abstraktní rozhraní pro animované (pohyblivé) body. Odvozené body se mohou pohybovat po různých trajektoriích a metody třídy **`CMovingPoint`** nabízí obecný přístup k prvkům definujícím tyto trajektorie (počáteční a koncový bod, referenční body, vektory atd.). K výpočtům trajektorií se používají generátory (podobně jako u křivek - viz třída **`CCurve`** níže).

CMovingPointLine

Konkrétní typ pohyblivého bodu, jehož trajektorií je úsečka.

Pro výpočty spojené s pohybem se používá třída **CLineGenerator**.

Třída předefinovává metodu `setTime` a při každém jejím volání nastaví bod do příslušné pozice na trajektorii. Parametr t použije pro určení pozice (jednoduše jej dosadí do funkce $Q(t)$ generátoru).

Stejně tak předefinovává metodu `moveBy`, která slouží k relativnímu posouvání objektu. Je-li hodnota t rovna 0 (resp. 1), posouvá se příslušný krajní bod trajektorie. V ostatních případech se posouvá trajektorie celá (tj. oba koncové body).

CCurve

Definuje abstraktní třídu pro všechny křivky používané v aplikaci. V členských proměnných ukládá základní informace o křivce: ukazatele na počáteční a koncový bod (**CPoint**), tloušťku a barvu.

Každá křivka obsahuje také ukazatel na tzv. generátor. Generátor je potomek třídy **CGenerator** (další informace naleznete v kapitole 4.3.2). Generátor si křivka zaalokuje sama, případně si může vybrat z více generátorů Např. kružnicový oblouk **CArcCurve** se pokusí použít **CArcGenerator**, a teprve v případě, že tento generátor selže (všechny tři body leží v přímce), použije **CLineGenerator**. Většina metod křivky je pak implementována obecně, pouze pomocí metod generátoru.

Křivka dále obsahuje seznam všech oblastí (STL-list ukazatelů na **CArea**), se kterými inciduje coby hraniční křivka. Pokud je změněna (např. protože se posunul jeden z koncových bodů), zavolá pro každou oblast na seznamu její funkci `setRedraw`. Stejně tak pokud je **CCurve** zničena, destruktor zničí i všechny přilehlé oblasti.

CVectorCurve

Abstraktní třída definující speciální typ křivek. Rozšiřuje definici standardní křivky (**`CCurve`**) přidáním dvou vektorů, z nichž jeden je přidružen k počátečnímu a druhý ke koncovému bodu.

CLine

Tato třída reprezentuje obyčejnou úsečku. Neobsahuje žádné vlastní členské proměnné (pouze dědí proměnné od **`CCurve`**). Jako generátor používá třídu **`CLineGenerator`**.

CArcCurve

Tato třída reprezentuje kružnicový oblouk. Oblouk je definován třemi body, a proto **`CArcCurve`** obsahuje navíc ukazatel na třetí bod (**`CPoint`**). Jako generátor používá **`CArcGenerator`**. V případě, že všechny tři body leží v jedné přímce a **`CArcGenerator`** selže, použije se místo něho **`CLineGenerator`**.

CFergusonCurve

Fergusonova kubika je (na rozdíl od předchozích dvou křivek) odvozena od třídy **`CVecorCurve`**. Sama nepřidává žádné další členské proměnné a k vykreslování používá **`CFergusonGenerator`**.

CArea

Definuje oblast ohraničenou dvěma, nebo více křivkami. V členských proměnných uchovává seznam křivek (STL-list ukazatelů na **`CCurve`**), které oblast ohraničují, přičemž dvě sousední křivky (stejně jako první a poslední křivka) v seznamu na sebe navazují (mají společný krajní bod). Navíc si oblast uchovává seznam všech bodů (STL-list ukazatelů na **`CPoint`**), které incidují s hraničními křivkami, aby bylo možné manipulovat přímo s body (např. při posunu celé oblasti se neposouvají jednotlivé křivky, ale přímo hraniční body).

Na rozdíl od křivek a bodů, smazání oblasti nijak neovlivní ostatní objekty.

CColorArea

Konkrétní typ oblasti, který celou plochu, definovanou hraniční křivkou, vyplnění jednou barvou.

4.3.2 Generátory

Generátory jsou výpočetní modely pro jednotlivé křivky. Tyto modely byly umístěny do samostatných tříd, aby mohly být využity i k jiným účelům než vykreslování křivek (např. k výpočtu trajektorií pohybu bodů při animacích).

Každý generátor je odvozen od abstraktní třídy **`CGenerator`**, která neobsahuje žádné členské proměnné a definuje několik čistě virtuálních metod. Odvozená třída pak musí implementovat tyto funkce:

- výpočet libovolného bodu křivky (hodnoty funkce $Q(t)$)
- výpočet tečny v libovolném bodě (hodnoty funkce $Q'(t)$)
- výpočet délky křivky
- výpočet nejmenšího ohraničujícího obdélníku (bounding-box)
- aproximace křivky lomenou úsečkou s nastavitelnou přesností

V aplikaci jsou implementovány tyto generátory křivek:

- **`CLineGenerator`** – jednoduchá úsečka
- **`CArcGenerator`** – kružnicový oblouk
- **`CFergusonGenerator`** – Fergusonova kubika

4.3.3 Další pomocné třídy

Kromě základních tříd a generátorů se v aplikaci vyskytují další entity, které jsou používány celou řadou tříd, avšak nedají se zařadit do žádného samostatného logického celku.

CPixel

Pomocná třída která zabaluje kód pro práci s jedním pixelem v obrazové paměti (v modelu RGBA) při barevné hloubce 32 bitů (8 bitů na kanál). Tato pracuje pouze s pixely uloženými v obrazové paměti reprezentované třídou **`Gdk::Pixbuf`** na x86 kompatibilní architektuře (tj. s čísly uloženými ve formátu big-endian).

CVector

Jak již název napovídá tato třída představuje implementaci dvojrozměrného vektoru. Do členských proměnných ukládá jednotlivé složky x a y, definující vektor v ortogonálním systému souřadnic. Dále nabízí celou řadu základních metod pro práci s vektorem (otočení, změna velikosti, apod.) a implementuje základní aritmetiku nad vektory (sčítání, odčítání a násobení skalárem).

CRectangle

Tato třída reprezentuje obdélník. Uvnitř jsou uloženy celočíselné souřadnice levého horního a pravého dolního rohu. Rovněž nabízí základní metody pro práci s obdélníky, které jsou užitečné zejména pro manipulaci s bounding-boxy (zvětšení/zmenšení, sjednocení a průnik).

CVectorFunctor

Abstraktní třída funktoru určeného pro hromadné operace nad vektory obrázku. Operátor `()` funktoru dostane jako parametr referenci na **`CVector`**, nad kterým provede příslušnou vektorovou operaci.

Každý vektorový objekt (**`CPoint`**, **`CCurve`**, **`CArea`** …) má implementovanou metodu `vectorTransformation`, která dostane jako parametr ukazatel na třídu odvozenou od **`CVectorFunctor`** a aplikuje jej na všechna vlastní vektorová data (např. **`CPoint`** považuje své koordináty za vektor od počátku souřadnic).

Obdobně mají i kontejnery (**CImage**, **CFrame** a **CPlane**) implementovanou metodu `vectorTransformation`, která se postará o vektorovou transformaci všech objektů v kontejneru (případně ve všech vnitřních kontejnerech).

CScalarFunctor

Abstraktní třída funktoru určeného pro hromadné operace nad skalárními daty obrázku. Za skalární se považují číselné atributy objektů, které souvisí s vykreslováním (např. šířka čáry u křivek).

Všechny třídy obsahující skalární informace a třídy kontejnerů implementují metodu `scalarTransformation`, která má analogický význam jako `vectorTransformation` u vektorových funktorů.

4.4 Zpětná identifikace objektů z obrázku

V kapitole 3.4 jsem se zmínil o zpětné identifikaci vektorových objektů z rastrových dat. Uživatelské rozhraní potřebuje o každém vykresleném pixelu umět zjistit, který vektorový objekt jej vykreslil, aby bylo možné s objekty dále pracovat (např. při kliknutí myší na obrázek označit objekt ležící pod kurzorem).

Datovou reprezentaci včetně algoritmů popsaných v kapitole 3.4 zapouzdřuje třída **CResolver** a pomocná třída **CSimpleResolver**. Data ve třídě **CResolver** jsou logicky svázána s obrázkem v nějaké vyrovnávací paměti. Instanci třídy **CResolver** obsahují pouze třídy **CFrame** a **CPlane**, přičemž data resolveru v **CFrame** jsou poskládána z dat resolverů v jednotlivých vrstvách.

4.4.1 Třída CResolver

Ukládá informace potřebné pro zpětnou identifikaci objektů jako pole řádků, kde každý řádek je dynamické pole záznamů (STL-vector), ve kterém jsou uloženy struktury **CResolveItem**. Tato

struktura obsahuje hodnotu souřadnice x (podle níž jsou také struktury na řádku vzestupně setříděny) a ukazatel na **`CObject`**. Ukazatel může být i `NULL`, pokud se na daných souřadnicích žádný objekt nenachází. Každý záznam na řádku uchovává informaci o tom, jaký objekt lze nalézt na daném řádku od příslušné souřadnice x dál směrem vpravo (viz kapitola 3.4).

Při vyhledávání objektu podle souřadnic se postupuje jak je popsáno v kapitole 3.4.2. Souřadnice y nám přímo určí řádek a na tomto řádku se pak algoritmem binárního vyhledávání v setříděném poli (podle souřadnice x) nalezne struktura **`CResolveItem`**. Ukazatel z této struktury nám dává hledaný objekt.

Třída **`CResolver`** obsahuje metodu `mergeOver`, která sloučí data s obsahem jiného resolveru. Díky této technice slévání je možné vypočítat a uchovávat pro každou vrstvu vlastní resolver.

4.4.2 CSimpleResolver

Pomocná třída, která slouží k přidávání záznamů o objektech do třídy `CResolver`. Od hlavního resolveru se liší především tím, že obsahuje data týkající se vždy pouze jednoho objektu. Datová reprezentace je postavena na stejném principu. Řádky však nejsou uloženy jako pole, ale jako binární vyhledávací stromy, aby bylo možné je snáze aktualizovat. Uzly těchto stromů tvoří dvojice čísel $[x_1, x_2]$ určující interval, ve kterém se objekt na daném řádku vyskytuje. Binárním strom je uspořádán pouze podle souřadnice x_1 a jednotlivé intervaly se nepřekrývají (pokud dojde při vkládání dat do **`CSimpleResolver`**u k překrytí, intervaly se sloučí).

Do této struktury je možné vkládat horizontální a vertikální oblasti široké jeden pixel (v praxi se tedy vkládá několik pixelů na jeden řádek, nebo jeden pixel na více řádků. Všechna data z **`CSimpleResolveru`** lze pak najednou přidat do hlavního resolveru.

4.5 Grafické uživatelské rozhraní (GUI)

Grafické rozhraní je postaveno na knihovnách Gtk+ [4] verze 2.8, přesněji na jejich rozšíření pro C++, které bývá označováno jako Gtk-- (Gtkmm) [5].

4.5.1 Okna a dialogy

Okna a grafické prvky na nich jsem navrhl pomocí aplikace Glade, která je součástí distribuce knihoven Gtk+. Nadstavba Glademm (součást Gtkmm) mi také pomohla vygenerovat většinu kódu, který souvisí s vytvářením oken a vazbou signálů (událostí) grafických prvků.

Každé okno (ať už hlavní nebo dialogové) je reprezentováno dvěma třídami. Rodičovská třída byla vygenerována aplikací Glade a její název nese postfix `_glade`. Od ní je odvozená konečná třída, jejíž instance je použita v aplikaci.

Aplikace má čtyři standardní a devět dialogových oken. Dialogová okna jsou jednoúčelová a jejich popis naleznete přímo ve zdrojovém kódu. Většina z nich slouží pouze k úpravě vlastností a parametrů vektorových (a jiných) objektů. Standardní okna (kromě okna obrázku) existují vždy právě v jedné instanci a jejich vlastnosti si dovolím popsat na následujících řádcích.

Hlavní okno aplikace (`main_window`)

Je vytvořeno jako první a na jeho existenci je vázán běh hlavního cyklu zpráv (tzn. jeho zavření způsobí ukončení aplikace). Obsahem okna je hlavní menu, tlačítka pro výběr aktivního nástroje a oblast, ve které může aktivní nástroj zobrazit své ovládací prvky s doplňujícím nastavením.

Okno rámců (`frames_window`)

Pomocné okno zobrazující informace o rámcích aktivního obrázku. Okno rovněž obsahuje tlačítka a nástroje pro úpravu rámců

(přidávání, mazání, změna pořadí a vlastností, …). Toto okno může být skryto a opět obnoveno (pomocí tlačítka v menu hlavního okna).

Okno vrstev (`planes_window`)

Pomocné okno zobrazující informace o vrstvách aktivního rámce z aktivního obrázku. Okno rovněž obsahuje tlačítka a nástroje pro úpravu vrstev (přidávání, mazání, změna pořadí a vlastností, …). Toto okno může být skryto a opět obnoveno (pomocí tlačítka v menu hlavního okna).

Okno obrázku (`Document`)

Pro každý otevřený obrázek se vytvoří samostatná instance třídy **`Document`** (a tedy i samostatné okno). Dokument obsahuje odkaz na **`CImage`** tohoto obrázku a sám uchovává pouze informace související se zobrazením a ukládáním obrázku (ukazatel na aktivní rámec a vrstvu, zoom, posunutí levého horního rohu při scrollování, cestu k přidruženému souboru, apod.).

Okno obrázku je odpovědné za odchytávání základních událostí (pohyb myši, stisknutí klávesy apod.) nad obrázkem a jejich přeposílání aktivnímu nástroji. Nástroj má možnost na tyto události reagovat (např. úpravou obrázku) a vynutit si pak překreslení, změnu zoomu apod. Nástroj nemá přímou kontrolu nad dokumentem a všechny akce může provádět pouze zprostředkovaně (jako reakce na události).

Okno obrázku má rovněž přístup k oknu rámců a oknu vrstev, aby mohlo v případě potřeby změnit vybranou vrstvu, nebo překreslit náhledové obrázky rámců.

4.5.2 Uživatelské nástroje pro práci s obrázkem

Nástroje slouží k úpravě obrázků a práci s náhledem. Všechny nástroje musí být odvozeny od abstraktní třídy **`CTool`**, která definuje aplikační rozhraní mezi dokumentem a nástrojem. **`CTool`** obsahuje několik

virtuálních metod představujících obsluhy události. Odvozená třída (tj. nástroj) je může předefinovat a umístit do nich kód reakce na danou událost. Dokument si uchovává ukazatel na aktivní nástroj a při zpracování událostí týkajících se obrázku zavolá příslušnou metodu nástroje. Konkrétně jsou dokumentem přeposílány tyto události:

- pohyb, stisknutí a uvolnění tlačítka myši
- klik, dvojklik a trojklik myši
- stisk a uvolnění klávesy
- požadavek na překreslení (resp. dokreslení informačních značek nástroje do obrázku)
- vybrání, zrušení výběru a reset nástroje
- změna aktivní vrstvy

Každá obslužná metoda události dostane kromě parametrů specifických pro danou událost ještě odkaz na strukturu **`CDocumentStatus`**, kterou připraví dokument těsně před jejím voláním. Struktura slouží jako rozhraní pro předávání dat mezi dokumentem a nástrojem. Nástroj dostane v **`CDocumentStatus`** všechny potřebné informace: ukazatel na vektorová data (tj. ukazatel na **`CFrame`**), ukazatel na aktivní vrstvu (**`CPlane`**), zoom, offset levého horního rohu vůči obrazovce atd. Změny vektorových objektů provádí nástroj přímo (přes ukazatel na rámec, nebo vrstvu). Změnu parametrů týkajících se zobrazení dokumentu (např. zoom) uloží nástroj zpět do struktury **`CDocumentStatus`**, ze které je dokument pohodlně získá po návratu z volané metody nástroje.

Třídy nástrojů jsou umístěny v zásuvných modulech. Podrobnější popis zásuvných modulů naleznete v kapitole 4.7.

Každý nástroj existuje po celou dobu běhu aplikace v jediné instanci (jedná se o singleton). Po načtení vytvoří aplikace pro každý nástroj korespondující tlačítko v nabídce hlavního okna. Toto tlačítko si pamatuje

ukazatel na svůj nástroj. Vybranému oknu obrázku (dokumentu) se zkopíruje ukazatel stisknutého tlačítka (tj. vybraného nástroje). Vzhledem k tomu, že některé nástroje mohou být stavové (uchovávají si informace vázané na aktivní dokument, rámec apod.) dojde vždy při změně aktivního dokumentu nebo při změně aktivního rámce k resetování aktivního nástroje (metodou `reset`).

GUI pro nastavení parametrů nástroje

Nástroje mohou také obsahovat některá vnitřní nastavení (např. nástroj pro vytváření čar obsahuje nastavení tloušťky čáry a její barvu). Aby nebylo nutné navrhovat speciální uživatelské rozhraní vhodné pro všechny nástroje, je tato činnost ponechána na samotných nástrojích. Každá třída nástroje může předefinovat volání funkce `getSettingsWidget` a vrátit při jejím volání ukazatel na libovolnou Gtk komponentu (**`Gtk::Widget`**). Pokud je ukazatel platný (není `NULL`), vloží se tato komponenta do připraveného kontejneru v hlavním okně pod tlačítky nástrojů. Vytváření, obsluha událostí a destrukce grafických komponent patřících nástroji je plně v jeho režii (typicky komponenty Gtk vznikají a zanikají současně s objektem nástroje).

Akce a řešení problému „Undo" a „Redo"

Každý moderní editor (ať již grafický nebo textový) nabízí uživatelům možnost vrátit poslední akci (případně několik akcí). Z časových důvodů nebyla tato vlastnost implementována a zde si dovolím pouze nastínit možný způsob řešení.

Momentálně fungují nástroje tak, že editují vektorová data přímo a nezaznamenávají žádné údaje pro zpětnou rekonstrukci. Nově by se zavedla třída objektů *akce*, která by definovala dvě základní metody `do` a `undo`. Každý nástroj by si pak od této třídy odvodil vlastní třídu a při editaci vektorových dat, by pouze vytvořil její instanci (se správnými parametry požadované akce) a zavolal na ni metodu

`do`. Následně by zařadil takto vytvořený objekt do historie akcí. Uživatel by pak při stisku tlačítka *zpět* pouze aktivoval rutinu, která vybere z historie naposledy vložený objekt a zavolá na něho metodu `undo`. Obdobně by se pak řešilo tlačítko vpřed (opakování odebrané akce).

Výchozí implementace by zálohovala celý rámec (nebo vrstvu) a v případě operace *zpět* by ji celou obnovila. Jednotlivé nástroje by si mohly nadefinovat vlastní a úspornější mechanismus zálohování a vracení změn.

4.6 Ostatní třídy a rozhraní

4.6.1 Práce se soubory (třída CArchive)

Pro načítání a ukládání dat používá aplikace vlastní binární formát. Tento formát je použit pro ukládání a načítání obrázku v nativním formátu aplikace (soubory `*.vgf`) a pro ukládání aktuálního nastavení aplikace (viz třída **`CSettings`** popsaná v kapitole 4.6.4).

Alternativou při výběru byly textové formáty (např. XML). Binární formát byl vybrán z důvodu úspory místa a také pro svou jednodušší implementaci.

Třída **`CArchive`** zapouzdřuje práci se soubory v binárním formátu. Po vytvoření její instance není objekt inicializován a očekává od uživatele třídy zavolání metody `openForLoading`, nebo `openForStoring`. Tyto metody inicializují třídu a otevřou soubor pro čtení (*loading*), nebo zápis (*storing*). Soubor zůstane otevřený a jednoznačně svázaný s objektem, až do jeho destrukce.

Ostatním objektům nabízí **CArchive** metody pro zápis a čtení ze/do souboru a ošetřuje případné chybové stavy. Číst a zapisovat je možné pouze základní datové typy (`int`, `double`, …) a řetězce (`string`). Spolu se zapsanými daty se ukládají i informace o jejich typech, takže při pokusu přečíst nesprávný datový typ dojde k chybě.

Metoda `serialize` (existující u všech potomků třídy **CObject**) očekává referenci na **CArchive**, pomocí kterého provádí práci se souborem. Metoda se používá pro načtení i zápis dat. O prováděné akci rozhoduje stav předané třídy `CArchive` (pokud je archív otevřený pro čtení, data se načítají a naopak).

4.6.2 Export obrázku

Export obrázku do rastrových grafických formátů je jednou z nejdůležitějších funkcí aplikace. Uživateli jsou nabízeny tři druhy exportu:

- uložení aktuálního rámce (snímku) do rastrového formátu
- uložení všech rámců (snímků) do rastrového formátu (každý snímek je uložen do samostatného souboru s jednotným prefixem a číslem snímku)
- uložení celého obrázku jako animace (do jednoho souboru – např. GIF, AVI …)

Ukládání obrázku a animací je zapouzdřeno do metod speciálního objektu – exportéru. Každý grafický formát má vlastní exportér. Rozlišujeme dva druhy exportérů:

- exportéry statických obrázků
- exportéry animací

Exportéry statických obrázků jsou odvozeny od třídy **CImageExporter**, která definuje uživatelské rozhraní pomocí virtuálních metod. Každý exportér obsahuje základní informace o grafickém formátu, do kterého ukládá (popis a standardní koncovku) a vlastní také instanci

třídy **`Gtk::FileFilter`**, kterou Gtk+ používá v dialogovém okně ukládání souboru (pro výběr koncovky souboru, do kterého se bude ukládat).

Všechny exportéry obrázků musí povinně předefinovat virtuální metodu `save`, která obdrží jako parametry obrazová data (ve struktuře **`Gdk::Pixbuf`**) a název souboru, kam se mají uložit. Nepovinně může exportér předefinovat také metodu `showSettingsDlg`. Tato metoda je volána těsně před uložením souboru a exportér zde může zobrazit vlastní dialog s doplňujícími nastaveními pro ukládání (např. zvolená kvalita, barevná hloubka apod.)

Exportéry animací jsou odvozené od třídy **`CAnimationExporter`**. Tato třída obsahuje stejné informace jako předchozí **`CImageExporter`**, avšak abstraktní rozhraní pro ukládání obrázků se liší. Místo metody `save` musí exportér animací předefinovat metody `startSaving`, `addFrame` a `finishSaving`. Jak již názvy napovídají, metoda `startSaving` slouží k otevření souboru (případně zápisu hlavičky apod.). Metoda `addFrame` přidá další snímek do výsledné animace a nakonec metoda `finishSaving` zapíše poslední data (je-li to potřeba) a uzavře soubor obrázku.

Všechny objekty exportérů jsou načteny při startu aplikace ze zásuvných modulů (viz kapitola 4.7.2) a každý se vyskytuje v právě jedné instanci až do ukončení aplikace.

4.6.3 Rastrové filtry

Při exportu do rastrového formátu může být vhodné provést některé úpravy obrazových dat až nad jejich rastrovou reprezentací (úpravy barev, přidání obrázku na pozadí, rozostření apod.). Za tímto účelem byly do obrázku přidány tzv. rastrové filtry. Tyto filtry jsou aplikovány těsně po rasterizaci obrázku, tzn. ještě před jeho uložením do souboru. Při práci se filtry v náhledovém okně nezobrazují, protože by mohly ztěžovat editaci obrázku (např. otočení, nebo zrcadlení by naprosto znemožnilo práci myší).

Každý filtr je reprezentován samostatnou třídou odvozenou od abstraktní třídy **`CFilter`**. Ta definuje základní virtuální metody filtru, z nichž nejdůležitější je `apply`, která dostane obrázek (jako **`Gdk::Pixbuf`**) a aplikuje na něho algoritmus filtru.

Každý filtr má jednoznačné identifikační číslo (ID). Identifikace je nezbytná, aby bylo možné při ukládání obrázků (do nativního formátu VGF) uložit i seznam použitých filtrů. Tento model sice umožňuje načítání filtrů ze zásuvných modulů, avšak identifikační čísla filtrů musí být přidělována centralizovaně. Momentálně jsou implementovány pouze tři ukázkové filtry, které mají přiděleny čísla 1 - 3.

Filtry mohou obsahovat vnitřní nastavení (parametry ovlivňující algoritmus filtru). Metoda `hasSettings` vrací boolovský příznak, zda objekt má, či nemá dodatečné parametry. Pokud objekt vlastní nějaké parametry, je možné na něho zavolat metodu `showSettingsDlg`, která zobrazí dialog s nastaveními těchto parametrů.

Při spuštění aplikace se filtry načtou ze zásuvných modulů (viz kapitola 4.7.3) a po celou dobu existuje od každé třídy filtru alespoň jedna instance. Tyto instance se nazývají referenční a obsahují výchozí nastavení jednotlivých filtrů. Při přidávání filtru do obrázku se vybraný referenční objekt filtru naklonuje (zavolá se na něho metoda `clone`). Při klonování filtru bez parametrů se pouze zkopíruje ukazatel a objekt filtru zůstane stále v jedné instanci. Naopak klonování filtrů s parametry má za následek vytvoření kopie objektu filtru, aby bylo možné měnit jeho parametry pouze pro daný obrázek (tj. na nové kopii).

Při destrukci obrázku (objektu **`CImage`**) se musí zlikvidovat všechny objekty filtrů s parametry (avšak nesmí se likvidovat objekty bez parametrů), na které si obrázek uchovává odkazy.

4.6.4 Ukládání nastavení a informací o aplikaci

O uchovávání informací a nastavení aplikace se stará třída **`CSettings`**. Po celou dobu běhu aplikace existuje v jedné instanci globální proměnné `Settings`. Uživatelské rozhraní, které by umožňovalo měnit jednotlivá nastavení není zatím připraveno a počítá se s ním v budoucích verzích aplikace. Momentálně se zde ukládají pouze informace, které uživatel nastavuje nepřímo (např. pozice a viditelnost oken, poslední otevřený adresář apod.)

Pro ukládání nastavení je využita třída **`CArchive`** (viz kapitola 4.6.1). Data jsou po spuštění aplikace načtena ze souboru `vector_guru.cfg` (z pracovního adresáře). Pokud soubor není přítomen, nebo je poškozen, načtou se výchozí hodnoty nastavení. Data se neukládají průběžně, ale pouze při ukončování aplikace (opuštěním funkce `main`), takže v případě nestandardního ukončení programu jsou změny nastavení ztraceny.

4.6.5 Časové funktory

V kapitole 4.2.2 jsem popsal způsob vytváření jednoduchých animací pomocí multi-rámců. Tyto animace fungovaly tak, že uživatel mohl nastavit krajní pozice pohybujících se bodů a trajektorii, po které se body pohybují, a mutli-rámec sám dopočítal pozice bodů na mezi-snímcích. Body se dopočítávají lineární interpolací, takže z fyzikálního hlediska se každý bod pohybuje konstantní rychlostí. To nemusí být výhodné, pokud chceme simulovat zrychlení, nebo dokonce složitější změny rychlosti.

Tento problém lze částečně vyřešit tím, že zavedeme mechanismus, který změní průběh času (tzn. výpočet interpolace). Proto byly zavedeny tzv. časové funktory. Z matematického hlediska si je lze představit jako funkci $f:[0,1]\rightarrow[0,1]$, avšak implementovány jsou jako objekty.

Každý multi-rámec může obsahovat ukazatel na časový funktor. Tento funktor se použije ke změně průběhu času t, který se nastavuje všem vnitřním objektům rámce (viz metoda `CObject::setTime`, kapitola 4.3.1)

vždy před vykreslením snímku. Časové funktory mohou sloužit k simulaci zrychlení, případně k oscilaci pohybu (tzn. změnit plynulý přechod bodů z $0 \rightarrow 1$ na $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$) apod. Pokud rámec nemá časový funktor, použije se standardní lineární průběh.

Všechny časové funktory jsou odvozené od třídy **`CTimeFunctor`**. Každý funktor je povinen předefinovat operátor `()`, který obdrží jako parametr desetinné číslo t (lineární čas) a vrátí opět desetinné číslo (modifikovaný čas). Krom toho má funktor metody `clone` a `hasSettings`, jejichž význam je totožný se stejnojmennými metodami třídy **`CFilter`** (viz kapitola 4.6.3). Poslední metodou, kterou může funktor předefinovat, je `getMinFrameCount`. Tato metoda vrací minimální počet snímků, které musí multi-rámec obsahovat, aby nedošlo ke ztrátě informací způsobené malým počtem vzorkovacích snímků (vytvoření aliasu). Např. při výše zmíněné oscilaci (přechodu $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$) jsou potřeba alespoň 4 snímky, aby si uživatel všiml, že body oscilují tam a zpět mezi krajními pozicemi.

Časové funktory mají implementované aplikační rozhraní, ale současné uživatelské rozhraní je neumí využít. Rovněž zatím není možné do aplikace funktory přidávat (ze zásuvných modulů). V budoucích verzích budou tyto nedostatky odstraněny.

4.7 Zásuvné moduly

Zásuvné moduly jsou realizovány pomocí dynamicky linkovaných knihoven. Momentálně se využívají k načítání grafických nástrojů, exportérů a rastrových filtrů. Jedna dynamicky linkovaná knihovna může obsahovat i všechny tři typy tříd (tzn. může přidávat nástroje, exportéry i filtry).

Při spouštění aplikace se zavádí všechny dynamické knihovny uložené v adresáři `plugins`, které mají příponu `*.dll`. Potřebuje-li modul ke své činnosti nějaká externí data (obrázky apod.), musí být tato data uložena

v podadresáři `plugins/nazev_modulu`. Např. modul `tools.dll` potřebuje ke své činnosti ikonky nástrojů. Tyto obrázky jsou uloženy v podadreáři `plugins/tools`.

Knihovny musí být napsány v jazyce C++ a musí exportovat alespoň jednu z níže popsaných globálních funkcí `load_plugin_`*`xxx`*, které zajišťují načtení příslušného typu modulů (např. funkce `load_plugin_tools` je odpovědná za načtení uživatelských nástrojů).

Vzhledem k tomu, že kompilátor C++ obaluje symboly identifikátorů vlastními značkami, je nezbytné umístit exportované funkce do bloku **`extern`** `"C" { … }`. Zásuvné moduly mohou (a ve většině případů také potřebují) používat některé třídy a funkce používané v aplikaci. Za tímto účelem byla část kódu aplikace přesunuta do sdílené knihovny, kterou mohou využívat i moduly (viz kapitola 5.2.2).

4.7.1 Moduly s uživatelskými nástroji

Uživatelské nástroje jsou podrobně popsány v kapitole 4.5.2. Modul rozšiřující paletu nástrojů aplikace musí obsahovat globální funkci `load_plugin_tools`. Tato funkce obdrží referenci na seznam nástrojů (STL-list ukazatelů na **`CTool`**), do kterého přidá instance importovaných nástrojů. Kromě toho obdrží také cestu k adresáři, ve kterém se nachází data modulu. Načítání dat probíhá při vytváření objektů nástrojů (tzn. v rámci volání funkce `load_plugin_tools`).

Součástí přiložené aplikace je celkem 10 nástrojů pro editaci obrázku a práci s náhledem, které pokrývají většinu dosavadních možností aplikace. Jejich implementaci si můžete prohlédnout ve zdrojovém kódu (adresář `gui/tools` soubory `tools_basic.h` a `tools_basic_gui.h`, viz kapitola 5.1.2).

4.7.2 Moduly s exportéry obrázků

Exportéry jsou podrobně popsány v kapitole 4.6.2. Zásuvný modul importující nové třídy exportérů musí definovat globální funkci `load_plugin_exporters`, která obdrží reference na seznam exportérů obrázků (STL-list ukazatelů na **`CImageExporter`**) a seznam exportérů animací (STL-list ukazatelů na **`CAnimationExporter`**). Do těchto seznamů přidá odkazy na nové objekty exportérů. Exportéry nenačítají žádná externí data ze souborů.

Momentálně jsou implementovány exportéry pro rastrové formáty BMP, PNG a JPG. Podpora dalších formátů, především pak animací (GIF, AVI) a vektorových formátů (zejména SVG), bude doplněna ve vyšších verzích aplikace. Třídy si můžete prohlédnout ve zdrojovém kódu v souboru `misc/exporters_basic.h` (viz kapitola 5.1.3).

4.7.3 Moduly s rastrovými filtry

Rastrové filtry jsou podrobně popsány v kapitole 4.6.3. Každý modul, který chce přidat filtry do aplikace, musí exportovat globální funkci `load_plugin_filters`. Tato funkce dostane jako parametr referenci na seznam filtrů indexovaný identifikačním číslem (STL-map, kde klíč je celé číslo a data jsou ukazatel na **`CFilter`**), do kterého přidá instance nových filtrů. V případě kolize identifikačních čísel není nový modul přidán a uživateli se zobrazí varovné hlášení. Ke kolizím může dojít pouze v případě, že uživatel použije zásuvný modul s filtry, které nejsou součástí aplikace a jejich autor nerespektuje potřebu centralizovaného přidělování identifikačních čísel.

Součástí aplikace jsou ukázky tří jednoduchých rastrových filtrů, jejichž implementace je realizována přímo knihovnami Gtk (respektive Gdk). Naleznete je v souboru `misc/fiters_basic.h` (viz kapitola 5.1.3).

5. Kompilace a přehled zdrojového kódu

Na závěr bych se ještě rád zmínil o tom, jak jsou jednotlivé třídy rozděleny do souborů se zdrojovým kódem a jak se po kompilaci linkují.

5.1 Přehled zdrojových souborů

Zdrojový kód je rozdělen poměrně jemně do souborů a související části jsou navíc rozděleny do podadresářů. Zásuvné moduly jsou součástí základního zdrojového kódu, aby bylo možné je linkovat i přímo do aplikace (podrobnosti o způsobech kompilace naleznete v kapitole 5.2). Soubory, které nejsou v této kapitole popsány nejsou významné pro pochopení fungování celé aplikace. Další informace týkající se zdrojového kódu můžete nalézt přímo v jeho komentářích.

5.1.1 Vektorové objekty

Všechny třídy vektorových objektů a kontejnerů se nacházejí v podadresáři `vectors`. Zde je přehled jednotlivých souborů:

`common.h` – pomocné třídy a globální funkce

`generators.h` – generátory (výpočetní modely) křivek

`objects.h` – základní (abstraktní) objekty (**`CObject`**, **`CPoint`**, **`CRenderableObject`**, **`CCurve`**, **`CVectorCurve`** a **`CArea`**)

`motion.h` – pohyblivé body (**`CMovingPoint`**, **`CMovingPointLine`**)

`curves.h` – objekty křivek (**`CLine`**, **`CArcCurve`**, **`CFergusonCurve`**)

`areas.h` – objekty oblastí (**`CColorArea`**)

`framework.h` – kontejnerové objekty (**`CPlane`**, **`CFrame`**, **`CImage`**)

`resolver.h` – zpětná identifikace objektů (**`CResolver`**)

`time_functors.h` – abstraktní třída funktorů (**`CTimeFunctor`**)

5.1.2 Grafické uživatelské rozhraní

Třídy grafického uživatelského rozhraní jsou uloženy v podadresáři `gui`. Soubory se třídami hlavních oken jsou přímo v `gui`, dialogová okna se nachází v `gui/dialogs` a uživatelské nástroje jsou umístěny do `gui/tools`.

Okna GUI jsou tvořena dvěma třídami. Základní třída je generovaná aplikací Glade a za svým názvem má postfix _glade. Odvozená (finální) třída je psaná ručně a obsahuje kód zpracování událostí. Každá třída je uložena v samostatném souboru (resp. ve dvou souborech – hlavička a kód), který má stejné jméno jako třída sama. Třídy všech oken jsou popsány v kapitole 4.5.1.

Příklad: třída **`Document`** má generovanou rodičovskou třídu **`Document_glade`** a jejich kód je uložen v souborech `document.h`, `document.cpp`, `document_glade.h` a `document_glade.cpp`.

Třídy související s nástroji se nachází v `gui/tools`. Zde je uložen jednak kód abstraktních tříd, definující aplikační rozhraní, ale také vestavěné třídy, které se za normálních okolností překládají do zásuvného modulu.

`tools_abstract.h` – definice rozhraní, konstant a pomocných funkcí

`tools_basic.h` – základní sada vestavěných nástrojů (linkuje se do zásuvného modulu)

`tools_basic_gui.h` – pomocné třídy GUI, které zobrazují nastavení vestavěných nástrojů (linkují se spolu nástroji v `tools_basic.h`)

5.1.3 Ostatní třídy

Ostatní třídy jsou umístěny v adresáři `misc` a v kořenovém adresáři projektu. V adresáři misc se nachází třídy, které nebylo vhodné zařadit do samostatného logického celku.

`archives.h` – rozhraní pro práci se soubory (třída **`CArchive`**)

`exporters.h` – abstraktní rozhraní tříd exportérů

`exporters_basic.h` – implementace základních exportérů (linkuje se jako zásuvný modul)

`filters.h` – abstraktní rozhraní tříd rastrových filtrů

`filters_basic.h` – ukázková implementace několika rastrových filtrů (linkuje se jako zásuvný modul)

V kořenovém adresáři zdrojového kódu se nachází zbývající soubory, které jsou typicky používány ve zbylých částech aplikace.

`data.h` – sdílená data, která aplikace používá (obrázky, ikonky, …)

`settings.h` – nastavení (třída **`CSettings`** a proměnná `Settings`)

`plugins.h` – funkce pro načítání zásuvných modulů (pluginů)

5.1.4 Hlavní soubor aplikace a funkce main

Hlavní soubor aplikace je uložen v kořenovém adresáři v souboru `main.cpp` a obsahuje pouze funkci `main` (vstupní bod aplikace). Tato funkce provede inicializaci (načte data a zásuvné moduly), vytvoří základní okno aplikace a spustí hlavní smyčku programu, která odchytává a zpracovává zprávy systému. Po ukončení hlavní smyčky uloží nastavení (ve třídě **`CSettings`**) a ukončí běh programu.

5.2 Přehled kompilace a linkování

Kompilace a linkování je řízeno Makefile skriptem a používá populární GNU překladač GCC (resp. G++) a k němu přidružené nástroje. Makefile byl automaticky vygenerován vývojovým prostředím Code::Blocks [7].

Celý projekt může být zkompilován a slinkován dvěma způsoby:

- monoliticky
- se zásuvnými moduly

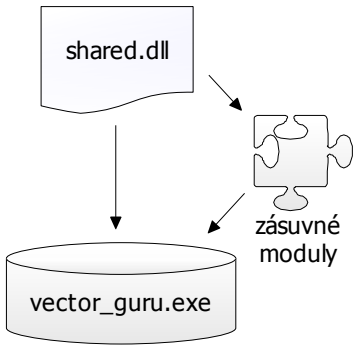
5.2.1 Monolitická kompilace

Při monolitickém kompilovaní je veškerý zdrojový kód slinkován do jediného spustitelného souboru a aplikace nepodporuje přidávání zásuvných modulů. K dipozici jsou pouze zásuvné moduly, které byly součástí kódu v době kompilace. Monolitickou kompilaci lze zapnout přidáním direktivy preprocesoru `BUILD_MONOLITHIC`. Tato metoda se hodí například pro ladění aplikace.

5.2.2 Kompilace se zásuvnými moduly

Kompilování se zásuvnými moduly vytvoří kromě spustitelného souboru aplikace také základní zásuvné moduly. Na tomto místě vzniká malý problém. Zásuvné moduly potřebují využívat některé části zdrojového kódu aplikace a aplikace potřebuje využívat zdrojový kód zásuvných modulů. To efektivně znemožňuje, aby byly zásuvné moduly kompilovány samostatně.

Řešení spočívá ve vytěsnění sdílených částí kódu aplikace do dynamicky linkované knihovny, kterou mohou kromě aplikace používat i zásuvné moduly. Tato sdílená knihovna byla pojmenována `shared.dll` a princip jejího použití je znázorněn na obrázku 9.



Obrázek 9: dynamické linkování

Za sdílené se považují všechny třídy vektorových objektů (obsah adresáře `vectors`), definice abstraktních tříd a rozhraní pro zásuvné moduly (část adresáře `misc` a `gui/tools`), sdílená data a nastavení (`data.h`, `settings.h`). Zásuvné moduly a samotná aplikace si linkují knihovnu `shared.dll` automaticky při spuštění (resp. zavedení do paměti).

Při linkování jsou vytvořeny tři zásuvné moduly (pro každý typ modulů jeden).

- `tools.dll` – nástroje grafického uživatelského rozhraní (soubory `tools_basic.cpp` a `tools_basic_gui.cpp`)
- `exporters.dll` – třídy exportérů (`exporters_basic.cpp`)
- `filters.dll` – rastrové filtry (`filters_basic.cpp`)

Hlavní aplikace pak načítá zásuvné moduly pomocí třídy **`Glib::Module`**, která zapouzdřuje přístup k dynamicky linkovaným knihovnám na různých platformách.

6. Závěr

Mým cílem bylo vytvořit základ vektorového editoru, který by uspokojil mé specifické požadavky. Tento úkol jsem do značné míry splnil vytvořením aplikace Vector Guru, která je součástí této práce.

Momentálně je Vector Guru plně funkční na platformě Windows a s velmi malým úsilím by mělo být možné ji portovat na operační systém Linux. Zároveň je aplikace připravena načítat zásuvné moduly, kterými může náročnější uživatel rozšířit její funkce.

Aplikace je celá navržena tak, aby bylo velice snadné ji rozšiřovat. Objektový návrh navíc zpřehledňuje zdrojový kód a usnadňuje jeho opětovné využití. Všechny části, které mohly být odsunuty z jádra aplikace byly umístěny do zásuvných modulů, což ještě zvýšilo flexibilitu tohoto projektu. Uživatelské rozhraní je rovněž celé postaveno na zásuvných modulech, aby si uživatel mohl vybrat, které nástroje chce mít k dispozici, a které ne.

Celkově je aplikace připravena k použití, avšak některé části zatím nejsou pro uživatele příliš pohodlné a nabídka uživatelských nástrojů, grafických formátů a vektorových objektů by zasloužila rozšířit.

6.1 Další vývoj aplikace

V budoucnu předpokládám, že bude aplikace rozšířena o další uživatelské nástroje a především další formáty souborů, do kterých bude možné exportovat obrázky. Dále by měla být rozšířena množina podporovaných křivek, oblastí a trajektorií pro animaci bodů.

Pro pohodlnější práci bude nutné doplnit a vylepšit některé funkce uživatelského rozhraní, především pak dodělat možnost kopírování objektů přes schránku a také velice důležitou funkci „Zpět“, která vrátí poslední operaci (případně několik operací).

Ve více vzdálené budoucnosti pak předpokládám přidání podpory nelineárních animací. Tzn. jednoduché sekvence obrázků by byly nahrazeny sofistikovanějším modelem, který by umožnil vytváření komplexnějších animací (např. obdoba MS agentů, kteří fungují jako nápověda v MS Office).

Literatura

Následující zdroje jsem používal pro vytvoření této práce a jejich autorům tímto děkuji.

[1] *Jiří Žára, Bedřích Beneš, Petr Felkel*: Moderní počítačová grafika (Computer Press 1998)

[2] *Bruce Eckel*: Myslíme v jazyku C++ (Grada 2000)

[3] *Bruce Eckel, Chuck Allison*: Myslíme v jazyku C++ 2.díl (Grada 2006)

[4] Knihovny GTK+ (http://www.gtk.org/)

[5] Aplikační rozhraní knihoven GTK+ pro C++ (http://www.gtkmm.org/) a jeho dokumentace (http://www.gtkmm.org/docs/gtkmm-2.4/docs/)

[6] Základní knihovny STL (http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/index.html)

[7] Vývojové prostředí Code::Blocks (http://www.codeblocks.org/)

[8] Knihovny wxWidgets (http://www.wxwidgets.org/)

Přílohy

Součástí této práce je přiložené datové DVD s následujícím obsahem:

- kopie této práce ve formátu PDF
- instalátor aplikace Vector Guru a potřebných knihoven pro platformu Windows
- zdrojové kódy aplikace
- uživatelská dokumentace ve formátu XHTML