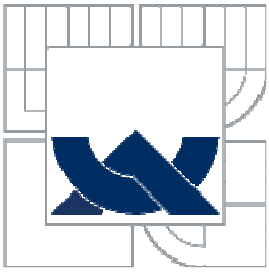
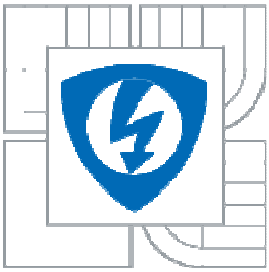


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚRICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

GENEROVÁNÍ TESTOVACÍCH VZORŮ
TEST PATTERN GENERATION

DIPLOMOVÁ PRÁCE
MASTER´S THESIS

AUTOR PRÁCE — Bc. MARTIN HAŠEK
AUTHOR

VEDOUCÍ PRÁCE — ING. MILOSLAV RICHTER, PH.D.
SUPERVISOR

BRNO 2010

VYSOKÉ UČENÍ
TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

Ústav automatizace a měřicí techniky

Diplomová práce

magisterský navazující studijní obor
Kybernetika, automatizace a měření

Student:	Bc. Martin Hašek	***ID:***	89133
Ročník:	2	***Akademický rok:***	2009/2010

NÁZEV TÉMATU:

Generování testovacích vzorů

POKYNY PRO VYPRACOVÁNÍ:

Napište program pro generování testovacích vzorů pro testování algoritmů vyhodnocování obrazu. Součástí návrhu je možnost tvorby testovacích vzorů snímané předlohy a filtrů v obrazové cestě. Navržené a realizované algoritmy by měly umožňovat jednoduché přidávání dalších typů filtrů a vzorů. Vzory by měly umožnit i jednoduché prostorové útvary (s kalibračními vzory). Na jednoduchém příkladu popište práci s programem a přidávání nových filtrů a vzorů.

DOPORUČENÁ LITERATURA:

Hlaváč V., Šonka M.: Počítačová vidění,Grada, Praha 1992, ISBN 80-85424-67-3
Faugeras O.: Three-Dimensional Computer Vision, The MIT Press 1993
Kraus K.: Photogrammetrie 1 und 2, Ummler / Bonn, 1996
Žára J., Beneš B., Sochor J., Felkel P.: Moderní počítačová grafika, Computer Press, 1998, ISBN 80-251-0454-0

Termín zadání:	8.2.2010	***Termín odevzdání:***	24.5.2010

Vedoucí práce: Ing. Miloslav Richter, Ph.D.

prof. Ing. Pavel Jura, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Vysoké učení technické v Brně

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Generování testovacích vzorů

Diplomová práce

Obor: Kybernetika, automatizace a měření
Student: Bc. Martin Hašek
Vedoucí práce: Ing. Miloslav Richter, Ph.D.

Abstrakt:

Tato diplomová práce je zaměřena na vytvoření softwarové aplikace, která dokáže simulovat optické vady čoček objektivů. Také umožňuje sestavit vlastní testovací vzory. Aplikace by měla být co nejvíce modulární, aby bylo snadné přidat nový typ zkreslení nebo vzoru. Vzory je možné natáčet v prostoru. Celý projekt lze uložit do souboru ve formátu XML. Práce je rozdělena do čtyř částí. První část popisuje, jak jsou řešeny jednotlivé moduly reprezentující kameru, vzory a zkreslení. Dále se zabývá jejich tvorbou, vzájemnou komunikací a správou. V další části jsou popsány navržené a upravené třídy. Jsou zde rozebrány členské proměnné a klíčové metody rozhraní. Také je zde popsáno jak vytvořit novou třídu a začlenit do aplikace. Poslední část se věnuje popisu programu pro uživatele.

Klíčová slova:
Optické zkreslení, Vada čočky, Testovací vzor

Brno University of Technology

Faculty of Electrical Engineering and Communication

Department of Control, Measurement and Instrumentation

Test Pattern Generation

Master's Thesis

Secialisation of study: Cybernetics, Kontrol and Measurement
Student: Bc. Martin Hašek
Supervisor: Ing. Miloslav Richter, Ph.D.

Abstract:

The goal of this thesis is to create an application for simulation optical distortion of lenses. It also makes possible create an own test pattern. The application should be modular enough to be easy adding new type of distortion or pattern. Pattern can be rotate in space. The project can be saved in XML file for further use. Whole thesis is dividend to four parts. The first part describes solution of particular modules representing camera, patterns and distortion. Followed by their generating, communication and management. Created and modified classes are described in next part. There are explained members variables and interface method. Creation and implementation of new type of classes is discussed in this part too. Last part describes how to use an application as a user.

Key words:
Optical Distortion, Lens Distortion, Test Pattern

Bibliografická citace

HAŠEK, M. Generování testovacích vzorů. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2010. 63s. Vedoucí práce Ing. Miloslav Richter Ph.D.

Prohlášení

„Prohlašuji, že svou diplomovou práci na téma Generování testovacích vzorů jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.“

V Brně dne: **24. května 2010** ……
podpis autora

Poděkování

Děkuji vedoucímu diplomové práce Ing. Miloslavu Richterovi, Ph.D. za metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

V Brně dne: **24. května 2010** ……
podpis autora

OBSAH

1. ÚVOD

Diplomová práce se zabývá tvorbou aplikace pro simulování optických vad objektivů. Účelem je nahradit nutnost sestavování reálné optické soustavy pro pořízení zkresleného snímku. Výsledný snímek se používá pro testování korekčních algoritmů, které se snaží dané zkreslení odstranit. Při zobrazování reálné scény se na výsledku podílí velké množství parametrů a je obtížné všechny popsat. Proto se vytvoří pomocí aplikace model, u kterého jsou parametry předem známé a na něj se poté se aplikují korekční algoritmy. Další výhoda spočívá v možnosti parametry zkreslení libovolně nastavovat, což by v reálné scéně mohlo být velmi složitě dosažitelné.

1.1 ZADÁNÍ

Napište program pro generování testovacích vzorů pro testování algoritmů vyhodnocování obrazu. Součástí návrhu je možnost tvorby testovacích vzorů snímané předlohy a filtrů v obrazové cestě. Navržené a realizované algoritmy by měly umožňovat jednoduché přidávání dalších typů filtrů a vzorů. Vzory by měly umožnit i jednoduché prostorové útvary (s kalibračními vzory). Na jednoduchém příkladu popište práci s programem a přidávání nových filtrů a vzorů.

1.2 CÍL PRÁCE

Cílem práce je rozšířit aplikaci z bakalářské práce Petra Černého [1] a případně opravit chyby. Zejména se jedná o upravení hospodaření s pamětí během průběhu simulace, tak aby byla alokována paměť jen po nezbytně nutnou dobu. Při zpracovávání velkého množství obrazů s velkým rozlišením by se aplikace stala paměťově velmi náročnou. Dále rozšířit vytváření vzorů o možnost jejich natáčení okolo všech tří os optické soustavy a umožnit tak tvorbu jednoduchých prostorových útvarů.

Umožnit snadné sestavení vzoru a konfiguraci optického prostředí a možnost tuto soustavu vhodným způsobem uložit, editovat a načíst bez nutnosti vše znovu vytvářet.

1.3 ROZBOR ZADÁNÍ

Problematika návrhu modelu je detailně popsána v [1]. Stručně shrnuto se jedná o to, že do simulace zkreslení se nezahrnují všechny možné fyzikální jevy s tím související. To ani není nutné, neboť některé nemají na výsledek velký vliv. Místo toho optickou soustavu tvoří projekční roviny v podobě bitmapy, která zachycuje vlastnosti soustavy v daném umístění. Navzorkování obrazu na konečný počet pixelů způsobí ztrátu informací. Ovšem reálné digitální snímací zařízení (kamera, fotoaparát) zachycuje obraz na CCD čip, který má také konečný počet pixelů (rozlišení).

Dalším požadavkem je aby bylo možné roviny vzoru natáčet v prostoru. Pro zjednodušení bude střed otáčení ležet na optické ose. Tím vyvstává problém jak vyřešit zobrazení objektu v 3D scéně. Pro správné perspektivní zobrazení je zapotřebí brát v úvahu vzdálenost bodu na rovině vzoru od projekční roviny, na které se scéna zobrazí. Proto je nutné zvolit způsob, jakým bude rovina vzoru popsána a jak se určí její natočení vůči osám optické soustavy. Řešením může být využití algoritmů používaných ve 3D grafice.

Nadefinování optického prostředí a tvorba testovacího vzoru by měla být pro uživatele co nejjednodušší. V původním návrhu v sobě filtry obsahovaly datové úložiště pro náhledy zkreslení a také nesly informaci o filtrech předchozích a následujících. To má za následek horší hospodaření s pamětí a komplikované přidávání filtrů kdekoli do výpočetního řetězce. V novém návrhu je vytvořena další třída, která se stará o ukládání mezivýsledků a také zařazování filtrů do výpočtu. Třídy filtrů tak obsahují pouze své specifické vlastnosti a o jejich volání se stará nadřazená třída. Další výhoda spočívá v možnosti vytvoření více typů filtrů s různým nastavením parametrů, přičemž je možné využívat pro simulaci jen vybrané.

Pro snadné znovupoužití vytvořených testovacích vzorů a filtrů by měla být umožněno konfiguraci aplikace uložit a poté kdykoli znovu načíst. Pro samotné

uložení dat připadá v úvahu několik formátů. Nejméně na disku zabere soubor uložený v podobě binárních dat. Takto uložený soubor by umožňoval znovu načíst uloženou konfiguraci, ovšem je nečitelný pro člověka a tím je nemožné jej smysluplně editovat. Nabízí se tedy textový výstup. Prostý text by se ale mohl stát nepřehledný ve velkém počtu parametrů a také by se snadno mohlo chybovat při editování. Proto se jako vhodné řešení jeví využití jazyka XML, který je strukturovaný a pro uživatele přehledný.

2. ŘEŠENÍ

V této kapitole je postupně popsáno, jak jsou jednotlivé zadané problémy řešeny. Kostra aplikace vychází z předchozí bakalářské práce [1]. Navíc je přidána možnost natáčet roviny testovacího vzoru okolo všech tří os optické soustavy. Možnost nadefinovaný projekt ukládat, editovat a načíst. Také jsou upraveny některé části kódu tak, aby aplikace byla více modulární.

2.1 SOUČASNÝ STAV

Základní myšlenkou dle [1] je rozdělit celý program na několik částí tak, aby každá část představovala ucelený problém. Výsledkem je rozdělení aplikace do tří celků:

- snímací zařízení – zachycuje důležité vlastnosti reálného zařízení (digitální kamera nebo fotoaparát) vyjma optických vad objektivu. Jedná se o rozlišení snímacího CCD čipu udávané v pixelech, vzdálenost od počátku souřadného systému optické soustavy.
- filtry – realizují optické vady objektivu. Tyto vady mohou být různých kategorií např. geometrické, jasové, frekvenční.
- kalibrační vzor – představuje obraz, který je předlohou pro zkreslení.

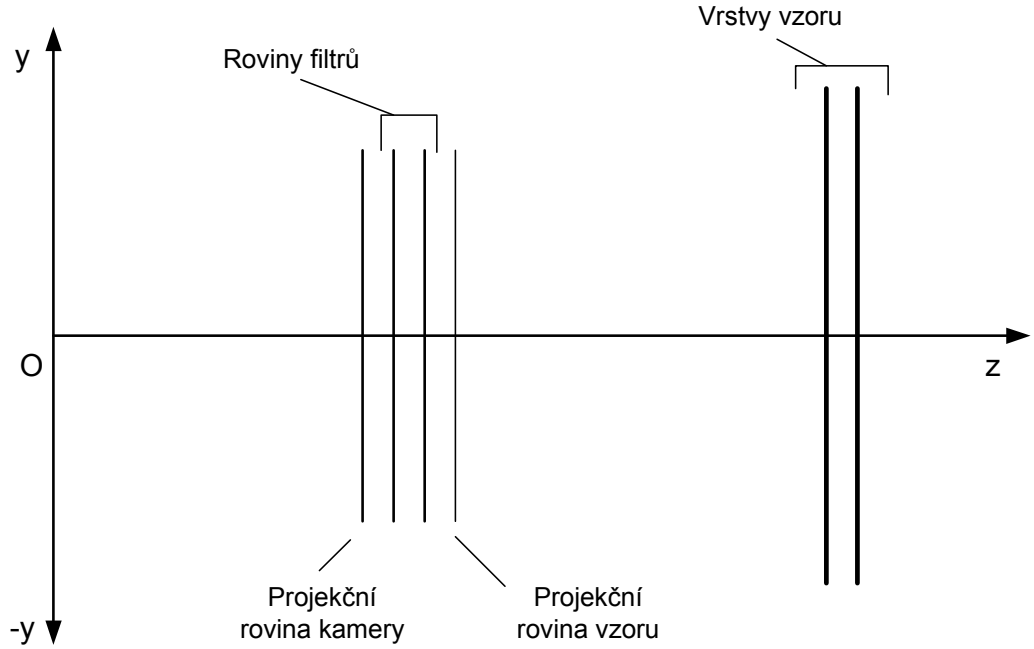
Prvním požadavkem na vylepšení je upravit používání paměti tak, aby nebyla obsazená i v době, kdy již není zapotřebí. V původním návrhu objekty filtrů obsahovaly datové úložiště pro náhledy zkreslení. Jak se filtry postupně zpracovávaly, zůstával v nich obraz, přestože již nebyl zapotřebí. Další nevýhoda spočívá v tom, že objekt filtru i roviny testovacího vzoru nese ukazatel na svého následníka v řadě. Takto není možné snadno přesunovat filtry a roviny v pořadí výpočtu či nějaký dočasně vyřadit z výpočetního řetězce.

Aplikace také neumožňovala posunutí a rotaci roviny vzoru. Nedostatkem také bylo nemožnost jakkoli nadefinované vzory a filtry uložit.

2.2 NÁVRH ŘEŠENÍ

Tato kapitola postupně popisuje, jakým způsobem jsou vyřešeny nové problémy a co bylo upraveno v původním návrhu, tak aby lépe vyhovoval současným požadavkům.

Na obrázku 2-1 je znázorněno jak je definována optická soustava a jak se do ní umisťují jednotlivé části.



Obrázek 2-1 Umístění jednotlivých částí soustavy na optickou osu

Vzdálenost projekční roviny kamery, roviny filtrů a projekční roviny vzoru je dána umístěním kamery na optickou osu *z*. Testovací vzor se skládá z vrstev, které jsou ve zvolené vzdálenosti od počátku také na optické ose *z*.

2.2.1 Snímací zařízení

V aplikaci je realizováno jedinou jednoduchou třídou *CCamera*. Má za úkol reprezentovat vlastnosti reálného snímacího zařízení. Jedná se o rozlišení snímacího CCD čipu. Vzdálenost umístění na ose z od počátku optické soustavy. Tím je dáno z jaké vzdálenosti se scéna snímá

2.2.2 Kalibrační vzory

Kalibrační vzor představuje vstupní obraz, který je následně zpracován filtry dle definovaného zkreslení. Na výsledný obraz se pak aplikují korekční algoritmy, které se snaží eliminovat způsobené zkreslení. Testovací vzor by měl být zvolen tak, aby na něm vybrané zkreslení bylo dobře vidět. V aplikaci je testovací vzor složen z vrstev, přičemž každá vrstva reprezentuje geometrické primitivum jako je přímka či kruh.

2.2.2.1 Geometrická primitiva

Za základní grafické prvky v rovině uvažujeme úsečky, lomené čáry, kružnice, elipsy a křivky. Pokud mají základní prvky plošný charakter, je u nich rozlišena barva vnitřní části a pozadí. Výsledný obraz je posloupností pixelů, tzv. rastrový obraz. Při jeho tvorbě je potřeba nalézt všechny pixely, reprezentující daný grafický prvek, a přiřadit jim barvu daného prvku.

V aplikaci jsou realizované dva základní grafické prvky – úsečka a kruh. Jejich základní vlastnosti jsou popsány níže.

Úsečka [1]

Úsečka je dána koncovými body $[x_1, y_1]$ a $[x_2, y_2]$. Její popis je doplněn o další atributy jako počet opakování v x-ovém, y-ovém směru a její tloušťka. Nejtenčí možná čára má hodnotu jedna (jeden pixel).

Běžně se úsečka rasterizuje celá najednou pomocí různých algoritmů (DDA, Bresenhamův [2]), což v tomto způsobu řešení není zcela vhodné. Protože pokud bude na sebe poskládáno více vrstev, budou muset existovat bitmapy pro všechny jednotlivé vrstvy a tím pádem vzroste nárok na paměť. Proto se získává jasová hodnota jen pro aktuálně potřebný pixel ve všech vrstvách, výsledná hodnota je pak závislá na průhlednosti jednotlivých vrstev. Při výpočtu se nejprve ze zadaných koncových bodů určí parametry přímky, jejíž analytický popis je:

$$ax + by + c = 0\,, \qquad (2.1)$$

takto

$$a = y_2 - y_1, \quad b = x_1 - x_2, \quad c = -ax - by\,. \qquad (2.2)$$

Je-li šířka úsečky větší než jedna, je dopočítávána symetricky na obě strany, ale pouze na dominantní ose prvku. Dominantnost osy je určena úhlem, který svírá prvek s osou y. Je-li tento úhel menší než 45° je dominantní osa y, v opačném případě je dominantní osa x. Výpočet úhlu α je dán vztahem

$$l_a = y_2 - y_1$$
$$l_b = x_1 - x_2 \qquad (2.3)$$
$$\alpha = \arccos \frac{(l_a \bullet a + l_b \bullet b)}{\sqrt{(l_a^2 + l_b^2)} \bullet \sqrt{(a^2 + b^2)}}$$

kde l_a *a* l_b představují složky směrového vektoru osy y,

a, b jsou parametry přímky, jejíž úhel počítáme,

α je úhel přímky s osou y [rad]

Barevná hodnota pro konkrétní pixel se získá tak, že se určí vzdálenost tohoto pixelu od středu úsečky s přihlédnutím na její šířku.

$$d = \frac{ax + by + c}{\sqrt{a^2 + b^2}} \qquad (2.4)$$

kde *a, b, c* jsou koeficienty přímky.

Na základě této vzdálenosti se určí, zdali pixel náleží úsečce nebo okolí a přiřadí se mu příslušná barevná hodnota.

Kruh [1]

Kruh patří mezi další základní grafické primitiva. Je definována svým středem [x_c, y_c] a poloměrem *r*. Výpočet jasové hodnoty aktuálního pixelu se provádí podobně jako u úsečky, zjištěním vzdálenosti od středu kruhu, podle následujícího vztahu

$$d = \sqrt{x_c^2 + y_c^2} \qquad (2.5)$$

Pokud je vzdálenost menší nebo rovna poloměru, pixel leží vně a je mu přiřazena Barevná hodnota výplně. V opačném případě je pixelu přiřazena barevná hodnota pozadí.

2.2.2.2 Geometrická transformace kalibračního vzoru[2]

Aplikace vyžaduje, aby bylo možné každou rovinu vzoru natáčet v prostoru okolo jejího středu, jímž prochází optická osa. V ploše roviny musí být rovněž umožněno grafické primitivum otočit a posunout vůči své původní poloze.
S výhodou lze využít toho, že základní grafický prvek je popsán parametry, které ho jednoznačně určují. Není tedy nutné provádět transformaci pro každý pixel celé roviny, ale s výhodou lze transformovat pouze tyto charakteristické parametry. U kruhu je to souřadnice střed a u úsečky koncový a počáteční bod. Úloha rotace v prostoru se tak rozdělí na dva problémy. Na natočení grafického prvku ve své rovině, na což můžeme použít pouze 2D transformaci. A na následné natočení celé roviny se vzorem.

Rovinná transformace vzoru se provádí pomocí násobení matic. Vždy se transformují pouze body definující dané grafické primitivum a to následujícím způsobem. Posunutí bodu [x, y] do bodu [x', y'] o d_x a $d_{y,}$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \end{bmatrix}. \qquad (2.6)$$

Kde d_x a d_y představuje posunutí v ose x a y.

Pro rotaci střed otáčení leží na optické ose, takže souřadnice jsou S = [0, 0]. Rotace bodu [x, y] do bodu [x', y'] o úhel γ,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\gamma & -\sin\gamma \\ \sin\gamma & \cos\gamma \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \qquad (2.7)$$

Tímto je vyřešeno natáčení a posunutí v rámci roviny. Samotnou polohu roviny pak určuje její normálový vektor v_n. Ve výchozí pozici je rovina kolmá na optickou osu a normálový vektor má tedy hodnotu $v_n = (0,0,1)$. Natočení roviny se převede na natočení vektoru a při vykreslování scény se pracuje s tímto vektorem. Rotace vektoru kolem osy x o úhel α je dána vztahem,

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad (2.8).$$

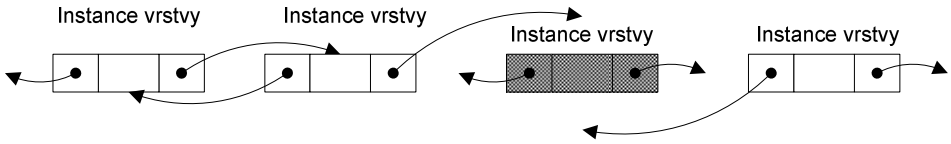
A rotace kolem osy y o úhel β

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad (2.9).$$

Před každým novým natočením je normálový vektor nastaven na výchozí hodnotu, takže rovina je na počátku vždy kolmá na optickou osu.

2.2.2.3 Sestavení testovacího vzoru

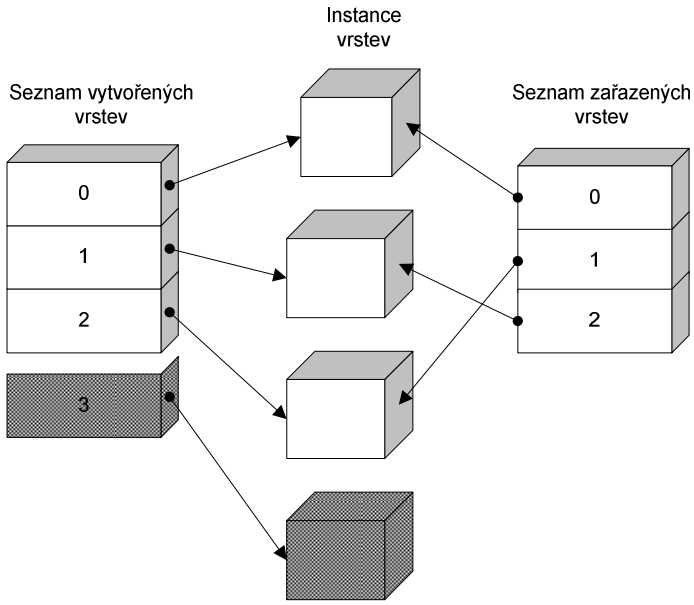
Testovací vzor je sestaven z vrstev, jejichž vlastnosti jsou popsány výše. V původním návrhu řešení podle **Chyba! Nenalezen zdroj odkazů.** každá vytvořená instance obsahuje ukazatel na následující a předchozí vzor v řadě, jak je naznačeno na obrázku 2-2.



Obrázek 2-2 Původní struktura seznamu vzorů

To přináší potíže, pokud chceme zařadit kamkoli v pořadí další vrstvu. Musí se upravit hodnoty čtyř ukazatelů, aby byl seznam opět správný. Chybí také možnost jednoduše vyřadit jakoukoli vrstvu z celkového vzoru.

Novým řešením je použít seznam ukazatelů na vytvořené vrstvy [4], viz obrázek 2-3.

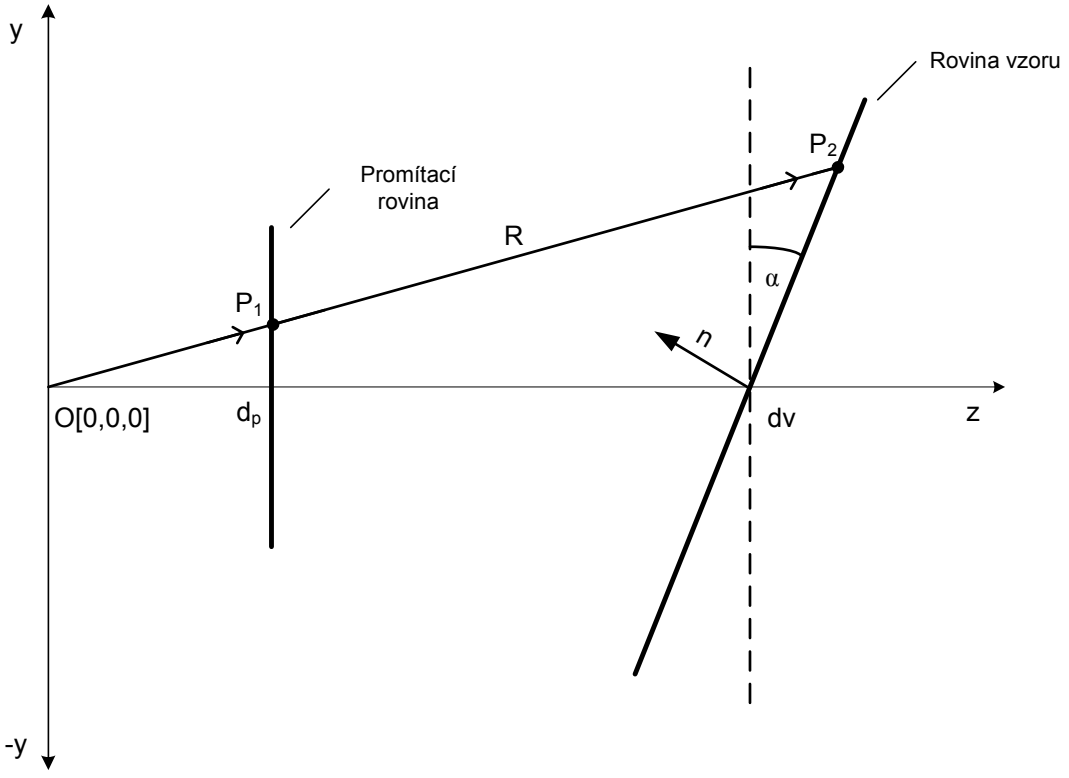


Obrázek 2-3 Nová struktura seznamu vrstev. V levé části seznam pro všechny vytvořené vrstvy. Vpravo seznam pouze pro vrstvy vybrané pro testovací vzor.

Tento seznam je znázorněn v levé části obrázku. Jakmile je vytvořena nová instance vrstvy, je zařazena na konec seznamu (znázorněno šrafováním). Takto lze přidat libovolný počet vrstev. Každá vrstva obsahuje příznak, zda se má pro výpočet použít či nikoli. Pokud ano, je ukazatel na ní zařazen do druhého seznamu, který je znázorněn v pravé části obrázku. Zde záleží na pořadí vkládání. Seznam umožňuje snadné vložení ukazatele na jakoukoli pozici v něm, nebo odstranění ukazatele ze seznamu. Tím je zajištěno, že lze vybrat jakoukoli vytvořenou vrstvu, zařadit ji na libovolnou pozici. Přitom stále zůstávají k dispozici pro další použití všechny vytvořené vrstvy. Pro zapouzdření těchto seznamů a práci s nimi je vhodné vytvořit novou třídu, která přebere zodpovědnost za vytváření a řazení vrstev. Třídy reprezentující grafická primitiva tak charakterizují jen své specifické vlastnosti a o jejich používání se stará nově vytvořená třída pojmenovaná *CPatternManager*.

Úkolem této třídy je tedy spravovat seznam vrstev a také je sestavit do výsledného testovacího vzoru, který bude vstupem pro filtry. Odpadá tan nutnost používat pro vytvoření vzoru filtr třídy *CFilterMasterImage*, který musel být zařazen jako první ve výpočtu (nejdále od kamery). Lze tak použít aplikaci pouze na generování vzorů bez nutnosti zařazovat jakýkoliv filtr.

Jak už bylo řečeno, tato třída se také stará o sestavení testovacího vzoru. Jelikož vrstvy vzoru mohou být umístěny (a natočeny) různě daleko od počátku a taktéž kamera, je zapotřebí zvolit vhodné promítání. Pro snímání 3D scény se zachováním proporcí se nejvíce hodí perspektivní promítání. Které zobrazí vzdálenější objekty menší a bližší větší. Střed promítání umístíme do počátku souřadného systému optického prostředí. Do zvolené vzdálenosti od tohoto počátku umístíme snímací zařízení, které je reprezentováno projekční rovinou s daným rozlišením. Zbývá vyřešit jak pixelům této roviny správně přiřadit barvu ze scény. Pokud budeme postupně brát všechny pixely z rovin scény a promítat na projekční rovinu, vlivem zaokrouhlování se některé pixely mohou překrývat a některé nemusí být naopak nastaveny vůbec. Navíc není potřeba zobrazovat celou definovanou scénu, neboť pohled na scénu je omezen zorným úhlem kamery. Proto je použita metoda sledování paprsku [8], viz obrázek 2-4.



Obrázek 2-4 Metoda sledování paprsku

Ze středu promítání se vyšle paprsek postupně přes všechny pixely promítací roviny. Paprsek R dopadne na aktuální rovinu vzoru a místo dopadu určuje barvu pixelu promítací roviny. Souřadnice, kam dopadne paprsek, se určí z následujícího odvození. Paprsek R je možné popsat parametrickou rovnicí [8]

$$\Re : P(t) = P_0 + t(P_1 - P_0) \qquad (2.10),$$

kde P_1 a P_0 jsou body ležící ne této přímce a t je parametr.

Dále je potřeba vyřešit průnik paprsku s plochou. Rovina ρ je určena bodem T_0 a normálovým vektorem $\vec{n}$. Pro libovolný bod $X \in \rho(X \neq T_0)$ platí, že vektory $\overrightarrow{XT_0}$ a $\vec{n}$ jsou kolmé. Platí tedy

$$(X - T_0) \bullet \vec{n} = 0 \qquad (2.11)$$

Dosazením parametrické rovnice (2.10) do rovnice (2.11) dostaneme

$$(P_0 + t(P_1 - P_0) - T_0) \bullet \vec{n} = 0, \qquad (2.12)$$

odtud pak

$$t = \frac{(T_0 - P_0) \bullet \vec{n}}{(P_1 - P_0) \bullet \vec{n}}. \qquad (2.13)$$

Dosazením získaného parametru *t* do rovnice (2.10), dostaneme souřadnice pixelu na rovině vzoru.

S výhodou můžeme zvolit bod $T_0 = [0, 0, d_v]$, bod $P_0 = [0, 0, 0]$ a bod $P_1 = [x, y, d_p]$. Rovnice pro parametr *t* pak přejde do tvaru

$$t = \frac{(T_0 - \emptyset) \bullet \vec{n}}{(P_1 - \emptyset) \bullet \vec{n}}. \qquad (2.14)$$

Takto určíme hodnoty pixelů pro všechny body projekční roviny. Tato projekční rovina je součástí třídy *CPatternManager* a je předána jako vstup pro filtry.

2.2.3 Filtry

Filtry simulují optické vady objektivu. Představují rovinu, v které se promítne zvolené zkreslení. Lze poskládat několik filtrů za sebe a tak simulovat více vad objektivu najednou. V této kapitole je popsán také postup, jak se filtry vytváří a probíhá samotný výpočet.

2.2.3.1 Základní vlastnosti filtrů

Každý objekt filtru je charakterizován několika vlastnostmi, které jsou pro všechny typy filtrů shodné. Jsou to především rozměry filtru, které udávají jeho rozlišení v pixelech. To se určuje na základě rozlišení objektu kamery a parametrů filtru. Po stanovení rozměrů filtru je vypočítán jeho střed, který leží na optické ose soustavy.

Nově lze každý filtr pojmenovat a tím ho snadno identifikovat v seznamu vytvořených a používaných filtrů. Pojmenování lze uskutečnit během vytváření filtru, nebo později dodatečně přejmenovat.

Další změnou oproti původnímu návrhu je odstranění datového úložiště pro obraz mezivýsledku a filtry již neobsahují ukazatele na svého následovníka a předchůdce. O volání filtrů a ukládání jejich mezivýsledků se stará nová třída, stejně jako je tomu u vzorů.

2.2.3.2Vytvořené filtry

Pro tuto aplikaci byly navrženy filtry realizující nejčastější optické vady čoček a to zkreslení geometrické a jasové [1]

Jasové zkreslení – viňetace [1]

Vinětace je druh zkreslení projevující se na obrazech jejich postupným tmavnutím směrem ke krajům. Matematický vztah vinětace je dán zjednodušeným vztahem

$$L = L' \cdot \cos^4(r). \qquad (2.15)$$

V návrhu se však vychází z tohoto vztahu

$$L = L' \cdot \cos(r), \qquad (2.16)$$

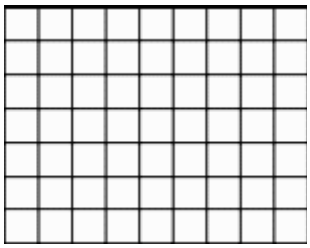
který je méně náročný na výpočet díky absenci čtvrté mocniny.

Geometrické zkreslení – soudkovitost [1]

Toto zkreslení je zapříčiněno rozdílným příčným zvětšením bodů ležících na optické ose a mimo ni. Body mimo osu mají jinou ohniskovou vzdálenost, která ovlivňuje výsledné zvětšení. Zjednodušeně lze zkreslení popsat rovnicí

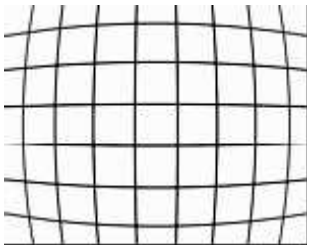
$$\Delta M = \frac{M_i - M}{M} \qquad (2.17)$$

kde M je příčné osové zvětšení a M_i příčné zvětšení bodu mimo optickou osu. Je-li zvětšení stejné v celém rozsahu zorného pole, pak hovoříme o tzv. ortoskopické soustavě obrázek 2-5.



Obrázek 2-5 Obraz bez zkreslení

Když se zvětšení pro body mimo osu se vzdáleností od centra čočky zmenšuje, jedná se o záporné zvětšení, tzv. soudkové, viz obrázek 2-6.

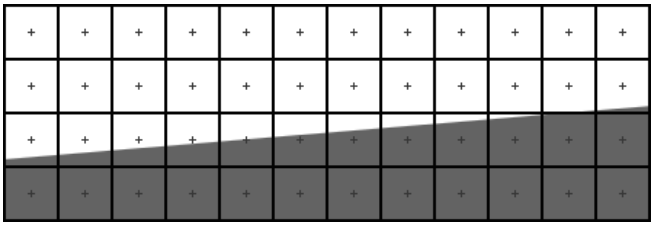


Obrázek 2-6 Soudkovité zkreslení

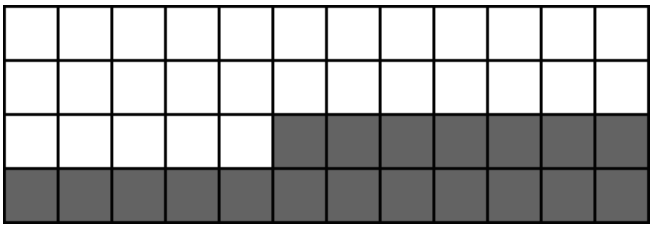
Frekvenční vlastnosti obrazu – antialiasing [9]

Aliasing je jev, ke kterému může docházet při převodu spojité informace na diskrétní. V tomto případě převod spojitého obrazu na diskrétní obraz, který je reprezentován mřížkou pixelů. Vzhledem k tomu, že aktivní plocha jednoho pixelu není nekonečně malá, není dodržen vzorkovací teorém a v obrazu se projevuje nežádoucí alias v podobě zubatých hran. Na obrázku 2-7 je znázorněn objekt, který

je potřeba vykreslit. Barevné vzorky jsou odebrány pouze ve středu pixelu a touto barvou je poté vyplněna celá jeho plocha. Výsledkem je zubatý schod, viz obrázek 2-8.

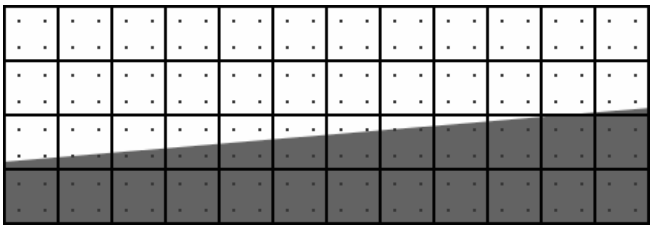


Obrázek 2-7 Rasterizace objektu. Zdroj [9]

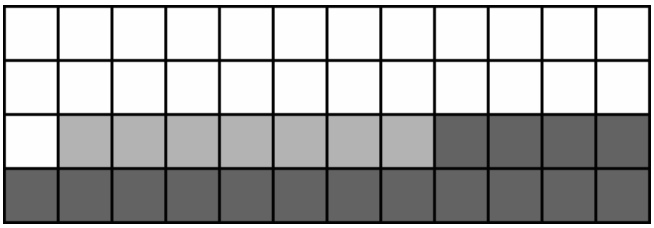


Obrázek 2-8 Výsledek rasterizace. Zdroj [9]

Možností jak potlačit tuto zubatost je rozdělit pixel na několik menších subpixelů, viz obrázek 2-9. To v praxi znamená vykreslit scénu ve větším rozlišení. Poté se obraz zmenší na původní rozlišení tak, že se zprůměrují hodnoty subpixelů náležící velkému pixelu. Nedochází pak k ostrému přechodu, ale jeví se mírně rozmazaný, viz obrázek 2-10.



Obrázek 2-9 Zvýšení rozlišení při rasterizaci. Zdroj [9]



Obrázek 2-10 Výsledek po zprůměrování subpixelů. Zdroj [9]

Čím více subpixelů použijeme, tím více budou přechody jemnější. Zvyšuje se však výpočetní náročnost, neboť scéna se musí vykreslit ve větším rozlišení. Pro čtyři subpixley vzroste rozlišení 4x, pro devět subpixelů 9x.

2.2.3.3Správa filtrů

Ze stejných důvodů jako u vzorů i filtrů je navržena nová třída *CFilterManager*, která se stará o vytváření a volání filtrů. Přebírá také datové úložiště pro obraz, které bylo přímo ve filtrech. Je tak soustředěno na jedno specifičtější místo. Třídy filtrů tak pouze zajišťují obrazovou transformaci, vstupní data a místo pro jejich uložení je jim dodáno touto řídící třídou.

Další výhodou je hospodárnost s pamětí a inicializace filtrů před výpočtem bez nutnosti spustit celý výpočet naráz. V původním řešení se výpočet odstartoval zavoláním funkce pro naplnění dat filtru, který byl nejblíže kameře. Pro něj se alokovala potřebná paměť a poté se pokusil vzít data z následujícího filtru. Pokud nebyl připraven, opět se pro něj alokovala paměť a požadoval data z následujícího filtru. Takto se prošly všechny filtry v řadě. Jakmile se toto provedlo u všech, začaly se od zadu plnit daty.

V novém návrhu jsou tyto dvě fáze odděleny. Třída stejně jako u vzorů obsahuje sezam filtrů, které se použijí pro výpočet. Nejprve se tedy celý seznam projde ve směru od kamery k předloze a pro každý filtr se určí jeho rozměr. Nealokuje se ale žádná paměť. Tím je umožněno snadno vložit kamkoli do řady nový filtr a jednoduše přepočítat rozměry ostatních filtrů. Není tedy nutné v této fázi vůbec pracovat s pamětí.

V druhé fázi se spustí samotný výpočet. Seznam zařazených filtrů se prochází od nejvzdálenějšího od kamery. Získá se jeho požadovaný rozměr, na jehož základě si třída *CFilterManager* alokuje paměť pro výsledek. Zavolá funkci filtru, která provede transformaci obrazu a ten je předán jako jeden parametr. Druhým parametrem je ukazatel na paměť pro výsledek. Ten se může zobrazit nebo uložit (závisí na nastavení příznaků). Paměť vyhrazená pro zdrojový obraz už není potřeba a může se uvolnit. Výsledek transformace je nyní novým zdrojem pro další filtr. S pamětí tak pracuje pouze aktuální zpracovávaný filtr.

2.2.4 Ukládání projektu

Pro uložení projektu byl zvolen formát XML. Jedná se o značkovací jazyk, který je vhodný díky své kompatibilitě a přehledné struktuře. Zde je příklad uloženého projektu.

```
<?xml version="1.0" ?>
- <Project>
    - <Settings>
        <SavePath>C:\Projects\</SavePath>
        <File type=".jpg" />
      </Settings>
      <Camera width="800" height="600" distanceO="545" />
    - <Pattern name="Vzor1.jpg" display="1" save="1">
        - <layer type="1" title="linka" use="1" atPosition="1">
            <distance d="845" />
            <begin x="0" y="-50" />
            <end x="0" y="50" />
            <thickness t="40" />
            <angle alfa="0" beta="0" gama="45" />
          - <color>
              <object A="255" R="255" G="255" B="255" />
              <env A="255" R="0" G="0" B="0" />
              </color>
          - <period>
              <repeat x="140" y="160" />
              <count x="-1" y="-1" />
            </period>
        </layer>
      + <layer type="0" title="Kruh" use="0" atPosition="-1">
      + <layer type="1" title="Linie" use="0" atPosition="-1">
      </Pattern>
    - <Filters>
        <filter type="0" title="Kamera.jpg" param="0" save="1" display="1" use="1"
          atPosition="-1" />
        <filter type="1" title="Soudek.jpg" param="0.001" save="1" display="1" use="1"
          atPosition="-1" />
        <filter type="1" title="Barrel.jpg" param="0.0057" save="1" display="0" use="0"
          atPosition="-1" />
        <filter type="2" title="Vinetace.jpg" param="0.96" save="1" display="1" use="1"
          atPosition="-1" />
      </Filters>
  </Project>
```

Takovýto soubor lze snadno upravovat pomocí jakéhokoli textového editoru a měnit tak nastavení optické soustavy. Umožňuje nadefinovat několik různých typů vrstev testovacího vzoru a filtrů pro výpočet použít jen některé. Obměna optické soustavy je tak velmi variabilní.

Detailní popis struktury souboru je popsán v kapitole 4.1.

3. REALIZOVANÉ TŘÍDY

Tato kapitola popisuje klíčové prvky navržených tříd a úpravy oproti původnímu návrhu **Chyba! Nenalezen zdroj odkazů.**. Hlavním úkolem je zachovat modularitu aplikace, tak aby bylo možné snadné naprogramování nových tříd pro vzory tak pro filtry

3.1 DIAGNOSTIKA

Prvním krokem pro zlepšení přehlednosti a urychlení ladění kódu byl přidán diagnostický dumping, který poskytuje knihovna MFC [4]. Použitím makra *TRACE* je možné vypsat do výstupu ladícího okna jakoukoli zprávu a sledovat tak správnost chodu programu. Příklad použití makra *TRACE* v programu. Při alokování paměti se vypíše její velikost. Po úspěšné alokaci se vypíše hlášení o správném provedení funkce.

```
float*** CPixelArray::AllocMem(int colmns,int rows,int depth)
{
  TRACE("Allocating memory %d x %d x %d\n", colmns, rows, depth);
                          .
  TRACE("Allocating done\n");
}
```

Další alternativou k makru *TRACE* je využití kontextu dumpu, který je více kompatibilnější s jazykem C++. Výhodou je, že lze vypsat vnitřní stav obejektu a jméno třídy. Aby to bylo možné, je zapotřebí ve třídě předefinovat virtuální funkci *Dump* a v ní vypsat potřebné parametry, které se poté zobrazí ve výstupní okně po zavolání funkce *Dump*.

```
void CLine::Dump(CDumpContext& dc) const
{
      CPrimitiveBase::Dump(dc);
      dc << "Begin point X = " << m_begin.X <<
            "\n            Y = " << m_begin.Y <<
            "\nEnd point X= " << m_end.X <<
            "\n          Y= " << m_end.Y <<
            "\nThick: " << m_thick <<
            "\nKoeficients a = " << m_a <<
            "\n            b = " << m_b <<
            "\n            c = " << m_c <<
            "\nAngle to Y: " << m_angle << "\n\n";
}
```

Po skončení programu se také vypisují všechny objekty, které nejsou zrušeny.

3.2 POMOCNÁ TŘÍDA STRUKTUR

Tato třída pojmenovaná *CStructures* zahrnuje několik často používaných struktur, které usnadňují předávání parametrů mezi objekty [1]. K původním strukturám, které jsou

Point – souřadnice bodu

CenterPoint – souřadnice středu

ARGBPixel – hodnota pixelu vyjádřená v RGB složkách a průhlednosti A,

byly přidány nové a to

Angle – hodnoty úhlů určující natočení roviny testovacího vzoru

```
typedef struct{
      float alfa; ///< úhel natočení okolo osy x
      float beta; ///< úhel natočení okolo osy y
      float gama; ///< úhel natočení okolo osy z
}Angle;
```

Supersample – poměr převzorkování testovacího vzoru pro vyšší rozlišení

```
typedef struct{
      int X;      ///< poměr převzorkování na x-ové souřadnici
      int Y;      ///< poměr převzorkování na y-ové souřadnici
}Supersample;
```

3.3 TŘÍDA SNÍMACÍHO ZAŘÍZENÍ

Tato třída nazvaná *CCamera* reprezentuje některé důležité vlastnosti reálného snímacího zařízení [1]. Jedná se konkrétně o rozlišení a vzdálenost kamery od počátku optické soustavy. Privátní část deklarace třídy

```
private :
      int m_xSize;            ///< rozlišení v ose x
      int m_ySize;            ///< rozlišení v ose y
      int m_distanceToO;      ///< vzdálenost od počátku soustavy
      CStructures::CenterPoint m_imageCenter; ///< střed obrazu
      bool m_isReady;         ///<příznak     určující     kompletní
                              nastavení snímacího zařízení
```

V novém návrhu byla přidána členská proměnná m_isReady, která zachycuje stav připravenosti objektu. Pokud nejsou nastaveny všechny parametry, není čím snímat scénu a tím pádem nelze spustit simulaci.

Původní konstruktor byl nahrazen novým, kde místo ohniskové vzdálenosti se nastavuje umístění snímacího zařízení.

```
CCamera::CCamera(int xSize, int ySize, int m_distanceToO )
{
	TRACE("Entering CCamera parameter constructor\n");

	m_distanceToO = m_distanceToO;
	m_xSize = xSize;
	m_ySize = ySize;
	m_imageCenter.X = (float)xSize / 2;
	m_imageCenter.Y = (float)ySize / 2;
	m_isReady = true;
}
```

Pro zjištění stavu připravenosti kamery se používá veřejná metoda

```
bool IsReady() {return m_isReady;} ///<Zjistí připravenost snímacího
                                    zařízení
```

Pokud vrátí hodnotu *true*, kamera je připravená a výpočet může být zahájen, v případě *false* není nastaven některý z parametrů.

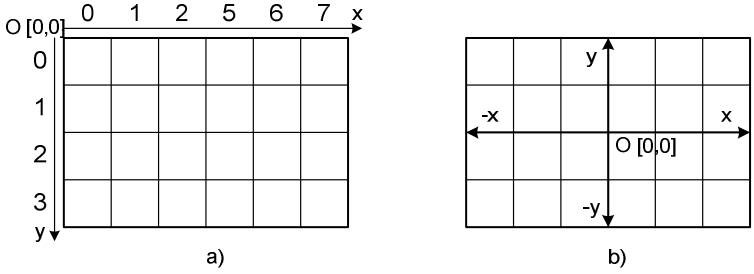
3.4 TŘÍDY KALIBRAČNÍHO VZORU

Následující kapitola popisuje třídy, které jsou nezbytné pro sestavení kalibračního vzoru.

3.4.1 CPixelArray

Tato třída slouží jako úložiště pro obrazová data. Obraz je složen z pixelů a každý pixel reprezentuje barvu v ARGB modelu. Pro složky R, G, B je rozsah hodnot od nejtmavší (0) po nejsvětlejší (255). Kanál A určuje průhlednost pixelu, kde 0 znamená úplnou průhlednost a 255 neprůhlednost. Základní návrh a chování třídy je popsáno v [1].

Indexování datového pole obrazu začíná v levém horním rohu, ovšem při práci se vzory se uvažuje střed na optické ose a tím dochází k rozdílné indexaci pole dat. Viz obrázek 3-1.



Obrázek 3-1 Indexace a) datového pole, b) obrazové plochy

Proto je nutné souřadnice obrazové plochy vzoru přepočítat na odpovídající souřadnice v datovém poli. To se v původním návrhu provádělo v třídách filtrů při výpočtu zkreslení. Tzn., že v případě zapomenutí volání této funkce se přistupovalo na špatné indexy datového pole a mohlo dojít k pádu aplikace. Lepším řešením je přesunutí funkce zodpovědné za přepočet indexů právě do třídy *CPixelArray*. Tím je navenek skryto, jakým skutečným způsobem je indexováno datové úložiště pro obraz. Není tedy dále při programování přemýšlet, zda aktuální souřadnice převést či ne. Definice této funkce je následující

```
CStructures::Point CPixelArray::GetBasePosition(CStructures::Point
virtualPosition)
{
	CStructures::Point basePoint;

	basePoint.X = m_center.X + virtualPosition.X;
	basePoint.Y = m_center.Y - virtualPosition.Y;
	if(basePoint.Y == m_height)
		basePoint.Y--;

	return basePoint;
}
```

Vstupem je bod *virtualPosition* na obrazové ploše, k němu se přičtou souřadnice středu datového pole a výsledkem je bod v datovém poli *basePoint*.

Další změnou je možnost pracovat s barevnými obrázky. Prvním krokem je právě upravení funkcí, které ukládají a získávají hodnotu pixelu. Jedná se o `SetGrayPixel(CStructures::Point point, float value)` a `GetGrayPixel (CStructures::Point point, float value)`.

Tyto funkce měli jako vstupní barvu pixelu jedinou hodnotu, která se nastavila pro všechny barevné kanály stejně, obraz tak byl ve stupních šedi. Aby bylo možné pracovat s barevným obrazem, je potřeba ukládat (načítat) vlastní hodnotu pro každý barevný kanál. Třída byla doplněna o nové funkce, jejichž definice vypadají takto:

a) funkce pro nastavení RGB hodnot pixelu uložených v proměnné *pixel*.

```
void CPixelArray::SetRGBPixel(CStructures::Point point,
CStructures::ARGBPixel pixel)
{
	CStructures::Point basePoint = GetBasePosition(point);

	if(basePoint.X >= m_width || basePoint.Y >= m_height)
	{
		TRACE("Chyba! Prekrocen rozsah pole\n");
	}
	else
```

```
	{
		m_data[0][basePoint.Y][basePoint.X] = pixel.B;
		m_data[1][basePoint.Y][basePoint.X] = pixel.G;
		m_data[2][basePoint.Y][basePoint.X] = pixel.R;
	}
}
```

b) funkce pro získání RGB hodnoty pixelu vrácené v proměnné *pixel*.

```
CStructures::ARGBPixel CPixelArray::GetRGBPixel(CStructures::Point
point)
{
	CStructures::ARGBPixel pixel = {255, 0, 0, 0};
	CStructures::Point basePoint = GetBasePosition(point);

if(basePoint.X >= 0 && basePoint.X < m_width && basePoint.Y >= 0 &&
basePoint.Y < m_height )
	{
		pixel.B = m_data[0][basePoint.Y][basePoint.X];
		pixel.G = m_data[1][basePoint.Y][basePoint.X];
		pixel.R = m_data[2][basePoint.Y][basePoint.X];
	}
	return pixel;
}
```

Tyto funkce pak nahradí původní. Pokud bude potřeba pracovat s obrazem ve stupních šedi, RGB složky pixelu se nastaví na stejnou hodnotu.

3.4.2 CPrimitiveBase

Jde o základní třídu, která zahrnuje vlastnosti společné pro všechny základní geometrické objekty, které tvoří dílčí rovinné vrstvy výsledného vzoru. Detailnější popis je v [1]. Oproti původnímu řešení došlo ke změně umístění členských proměnných ze sekce *protected* do sekce *private*. To má za následek, že tyto proměnné jsou přímo dostupné pouze uvnitř této základní třídy. Okolním třídám, dokonce i zděděným, jsou pak přístupné přes veřejné metody. Tím je dosaženo toho, že ve zděděných třídách se volají pouze metody rozhraní základní třídy a ta tedy plně zodpovídá za platné hodnoty svých členských proměnných. Ve zděděných třídách se tedy nemusíme starat o správnost nastavovaných parametrů. A v případě změny v základní třídě nemusíme dohledávat změny s tím související. Zde je výpis původních členských proměnných:

```
private:
	int m_distance;   ///< Vzdálenost objektu od počátku
	int m_width;      ///< Šířka objektu
	int m_height;     ///< Výška objektu
	CStructures::CenterPoint  m_center;          ///< Střed objektu
```

Hodnota barvy pixelu ve stupni šedi byla nahrazena RGB hodnotou, jak pro barvu výplně objektu, tak pro barvu pozadí:

```
CStructures::ARGBPixel m_ObjectColor; ///< Barva geometrického
                                          objetku
CStructures::ARGBPixel m_EnvColor;    ///<Barva  okolí  gemetrického
                                      objektu
```

Dále byly přidány následující členské proměnné rozšiřující třídu o další funkčnost:

```
char* m_LayerName;      ///< Název vrstvy vzoru
CStructures::Angle m_angle;   ///< Úhly natočení objektu okolo os
                                  X,Y,Z ve stupních
bool m_bUse;      ///< vrstva \c true = zařazena, \c false =
                      nezařazena do výpočtu
double m_vn[3]; ///< Normálový vektor plochy vrstvy.\ Určuje
                    natočení.
```

Proměnná *m_LayerName* představuje pojmenování vrstvy pro větší přehlednost, *m_angle* uchovává informaci o natočení roviny vrstvy pro případné uložení do souboru, *m_bUse* určuje, zda se vytvořená vrstva zařadí do výsledného testovacího vzoru či nikoli a poslední proměnná *m_vn* je normálový vektor roviny, určující její natočení.

Rotaci roviny zajišťují dvě metody, jedna je virtuální a musí být přepsána ve zděděné třídě, která bude představovat konkrétní geometrický objekt. Tato funkce, jejíž deklarace je:

```
virtual void Rotate(double gama);   ///< Otočí objekt okolo osy
                                        Z o úhel \a gama
```

má za úkol rotaci geometrického objektu v rovině se středem otáčení na optické ose.

Další virtuální funkcí, která musí být také přepsána pro konkrétní třídu, je:

```
virtual void Translate(int dx, int dy);  ///< Posune objekt v
                                            rovině o \a dx a \a dy
```

Ta má na starost posunutí geometrického objektu ve své rovině.

Rotaci této roviny v prostoru provádí funkce s deklarací:

```
void RotXY(double angleX, double angleY);
```

kde vstupem jsou úhly ve stupních a určují velikost natočení okolo os x a y.

Funkce provádí násobení matic podle rovnice (2.8) a (2.9). Před každou novou rotací je nastaven normálový vektor na hodnotu (0, 0, 1) což znamená, že rovina je kolmá

na optickou osu. Po provedení rotace jsou úhly uloženy do členské proměnné *m_angle*.

3.4.3 CCircle

Tato třída reprezentuje jedno ze základních grafických primitiv – kruh [1]. Je odvozena ze základní třídy *CPrimitiveBase* veřejným děděním. Má tedy všechny vlastnosti jako bázová třída a definuje nové, konkrétní parametry související s geometrickým popisem kruhu. Jsou to [1]:

```
private:
      int m_radius;     ///< Poloměr kruhu
      CStructures::CenterPoint m_circleCenter; ///< Střed v kruhu na
                                                  rovině vrstvy
```

Nově je přidána členská proměnná,

```
      int m_type;       ///< ID geometrickéo objektu
```

jejíž hodnota určuje, o jaký typ geometrického objektu se jedná. Pro kruh nabývá hodnoty 0. Toto rozlišení je důležité pro správné ukládání a načítání projektu ze souboru.

Pozměněn je i konstruktor, který nyní vypadá takto:

```
CCircle::CCircle(int distance, CStructures::CenterPoint
circleCenter, int radius, CStructures::ARGBPixel ObjectColor,
CStructures::ARGBPixel EnvColor, const char* title)
      :CPrimitiveBase(distance, ObjectColor, EnvColor, title)
{
      TRACE("-- Entering CCircle RGB constructor --\n");

      m_radius = radius;
      m_circleCenter = circleCenter;

      CPrimitiveBase::SetLayerCenter(m_circleCenter);
      CPrimitiveBase::SetSize(2*m_radius, 2*m_radius);
      CPrimitiveBase::SetXRepetition(2*m_radius, -1);
      CPrimitiveBase::SetYRepetition(2*m_radius, -1);

      m_type = 0;
}
```

Parametry, které jsou součástí základní třídy, inicializuje její konstruktor, který je volán před konstruktorem třídy zděděné. Tím je zajištěno, že každá třída je zodpovědná za svoje data.

Dále je zapotřebí předefinovat virtuální metody základní třídy, tak aby vyhovovali této třídě. Jedná se o metody:

```
void CCircle::Translate(int dx, int dy)
{
	TRACE("-- Entering CCircle::Translate --\n");

	m_circleCenter.X += dx;
	m_circleCenter.Y += dy;

}
```

K současnému středu kruhu se přičte velikost posunutí v ose x a y.

Metoda pro rotaci v rovině:

```
void CCircle::Rotate(double gama)
{
	//TRACE("-- Entering CCircle::Rotate --\n");

	CStructures::Angle angle;
	int rotX, rotY;
	double gama_rad;
	const double pi = 3.14159;

	gama_rad = pi/180*gama;

	rotX = cos(gama_rad)*m_circleCenter.X -
		sin(gama_rad)*m_circleCenter.Y;

	rotY = sin(gama_rad)*m_circleCenter.X +
		cos(gama_rad)*m_circleCenter.Y;

	m_circleCenter.X = rotX;
	m_circleCenter.Y = rotY;

	angle = GetAngle();
	angle.gama = gama;
	SetAngle(angle);
}
```

Souřadnice středu se zrotují podle vztahu (2.7). Tyto souřadnice se nastaví jako nový střed a uloží se velikost úhlu, o který objekt natočil.

S možností navzorkovat scénu ve větším rozlišení je přepsána i tato virtuální funkce ze základní třídy:

```
void CCircle::Resample(CStructures::Supersample supersample)
{
	TRACE("Entering CCircle::Resample\n");

	m_circleCenter.X = m_circleCenter.X * supersample.X;
	m_circleCenter.Y = m_circleCenter.Y * supersample.Y;

	CPrimitiveBase::SetLayerCenter(m_circleCenter);

	if(supersample.X >= supersample.Y)
	{
		m_radius = m_radius * supersample.X;
```

```
		}
		else
		{
			m_radius = m_radius * supersample.Y;
		}

		m_XRepeatPeriod = m_XRepeatPeriod * supersample.X;
		m_YRepeatPeriod = m_YRepeatPeriod * supersample.Y;

	}
```

Vstupní proměnná *supersample* určuje poměr nevzorkování v ose x a y. Podle tohoto poměru jsou přepočítány rozměry geometrického objektu tak, aby odpovídali novému rozlišení.

3.4.4 CLine

Tato třída je reprezentací dalšího geometrického primitiva – přímky (úsečky). Je odvozena stejným způsobem jako třída *CCircle* z předchozí kapitoly. Původní členské parametry z [1] jsou:

```
	private:
		int m_thick;		///<Tloušťka čáry
		double m_angle;		///<Úhel úsečky s osou Y
		double m_a, m_b, m_c;	///<Koeficienty úsečky  ax+by+c=0
		CStructures::Point m_begin;	///< Počáteční bod úsečky
		CStructures::Point m_end;	///< Koncový bod úsečky
```

Nově přidaná proměnná určující typ geometrického objektu:

```
	int m_type; ///< ID geometrickéo objektu
```

Pro úsečku je roven 1.

Posunutí úsečky v rovině zajišťuje předefinovaná funkce ze základní třídy:

```
void CLine::Translate(int dx, int dy)
{
	TRACE("-- Entering CLine::Translate --\n");

	m_begin.X += dx;
	m_begin.Y += dy;

	m_end.X += dx;
	m_end.Y += dy;

	//prepocet nove polohy
	CreateLineImage();
}
```

Koncový a počáteční bod jsou posunuty o danou velikost v ose x a y.

Nakonec se zavolá metoda *CreateLineImage(),* která přepočítá nové parametry úsečky.

Předefinovaná metoda pro rotaci v rovině:

```
void CLine::Rotate(double gama)
{
	//TRACE("-- Entering CLine::Rotate --\n");

	CStructures::Angle angle;
	int rotX, rotY;
	double gama_rad;
	const double pi = 3.14159;

	gama_rad = pi/180*gama;

	//rotace pocatecni bodu
	rotX = cos(gama_rad)*m_begin.X –
		sin(gama_rad)*m_begin.Y;

	rotY = sin(gama_rad)*m_begin.X +
		cos(gama_rad)*m_begin.Y;

	m_begin.X = rotX;
	m_begin.Y = rotY;

	//rotace koncoveho bodu
	rotX = cos(gama_rad)*m_end.X - sin(gama_rad)*m_end.Y;
	rotY = sin(gama_rad)*m_end.X + cos(gama_rad)*m_end.Y;

	m_end.X = rotX;
	m_end.Y = rotY;

	//prepocet nove plohy
	CreateLineImage();

	angle = GetAngle();
	angle.gama = gama;
	SetAngle(angle);

}
```

Koncový a počáteční bod se zrotují o velikost zadaných úhlů a opět se nakonec zavolá metoda *CreateLineImage().*

Tato metoda je privátní a má následující definici:

```
void CLine::CreateLineImage()
{
	TRACE("--- Entering CLine::CreateLineImage\n");

	//Osetreni platnosti zadani - 2 stejné body netvoří přímku
	if(!(m_begin.X == m_end.X && m_begin.Y == m_end.Y))
	{
		m_a = float(m_end.Y - m_begin.Y);
		m_b = m_begin.X - m_end.X;
```

```
		//smerovy vektor, dosadime jeden z bodu
		m_c = -m_a * m_begin.X - m_b * m_begin.Y;

	}
	else
	{
		TRACE("Points are identical!\n");

		m_a = 0;
		m_b = 0;
		m_c = 0;
	}

	m_angle = GetAngleToYAxe();

	int difX, difY;
	int width, heigth;
	CStructures::CenterPoint center;

	difX = abs(m_begin.X - m_end.X);
	difY = abs(m_begin.Y - m_end.Y);

	if (m_angle <= 45)	//dominantni osa Y
	{
		//vypocet rozmeru vzoru
		width = abs(m_end.X - m_begin.X) + m_thick;
		heigth = abs(m_end.Y - m_begin.Y);
		SetSize(width, heigth);
	}
	else			//dominantni osa X
	{
		//vypocet rozmeru vzoru
		width = abs(m_end.X - m_begin.X);
		heigth = abs(m_end.Y - m_begin.Y) + m_thick;
		SetSize(width, heigth);
	}

	//určení středu
	center.X = (m_begin.X + m_end.X)/2;
	center.Y = (m_begin.Y + m_end.Y)/2;
	CPrimitiveBase::SetLayerCenter(center);

}
```

Tato privátní funkce je volána vždy, když dojde ke změně pozice polohy úsečky. Při vykreslování je nutné znát charakteristické parametry přímky a ty jsou touto metodou vždy přepočítány tak, aby odpovídali aktuálnímu umístění přímky.

Poslední novu důležitou metodou je opět *Resample()* pro případ vykreslování ve větším rozlišení.

```
	void CLine::Resample(CStructures::Supersample supersample)
	{
		TRACE("Entering CLine::Resample\n");
```

```
        m_begin.X = m_begin.X * supersample.X;
        m_begin.Y = m_begin.Y * supersample.Y;

        m_end.X = m_end.X * supersample.X;
        m_end.Y = m_end.Y * supersample.Y;

        m_thick = m_thick * supersample.X;

        m_XRepeatPeriod = m_XRepeatPeriod * supersample.X;
        m_YRepeatPeriod = m_YRepeatPeriod * supersample.Y;

        CreateLineImage();
    }
```

3.4.5 CPatternManager

Tato třída je nově navrhnutá a má za úkol vytváření nových vrstev, zařazování do seznamů a sestavování testovacího vzoru tak jak je popsáno v kapitole 2.2.2.3.

Zde jsou popsány důležité členské proměnné a metody této třídy. Privátní část deklarace obsahuje tyto členy:

```
private:
    CPixelArray* m_outPattern; ///< Uložiště pro výsledný obraz
                                   testovacího vzoru
    CObList m_listLayerCreated;   ///< Seznam všech vytvořených
                                   vrstev (indexován od 0)
    CObList m_listLayerRegistred; ///< Seznam zařazených vrstev pro
                                   testovací vzor (indexován od 0)
    POSITION m_pos;          ///< Ukazatel na pozici v seznamech
    char* m_patternName;     ///< Název výsledného testovacího
                                   vzoru
    bool m_bDisplay;         ///< Příznak určující zobrazení vzoru.
    bool m_bSave;            ///< Příznak určující uložení vzoru.

    int m_projDistance;      ///< Vzdálenost proječní roviny od
                                   počátku optické osy
    int m_xSize, m_ySize;    ///< Rozlišení předlohy
    CStructures::Supersample m_supersample;   ///< Koeficient
                                                  převzorkování
```

Proměnné *m_listLayerCreated* a *m_listLayerRegistred* jsou seznamy pro ukazatele vytvořených a zařazených vrstev testovacího vzoru. Proměnná *m_outPattern* ukazuje na datové úložiště pro výsledný testovací obraz.

Pro přidávání vrstev do seznamu jsou určeny tyto metody:

```
    void AddLayer(CPrimitiveBase* pPrimitive);
```

Tato funkce přidá nově vytvořenou vrstvu, na níž ukazuje *pPrimitive*, na konec seznamu všech vytvořených vrstev.

```
void RegisterLayer(CPrimitiveBase* pPrimitive, int index);
```

Zařadí vrstvu, na níž ukazuje *pPrimitive* do seznamu vrstev použitých pro testovací vzor. Vrstva je zařazena na zvolenou pozici danou proměnnou *index.*

Toto je sada funkcí pro práci se seznamem:

```
void RegisterAll();     ///< Zařadí všechny vytvořené vrstvy do
                              testovacího vzoru
void UnregisterAll();   ///< Odstrani všechny vrstvy z testovacího
                              vzoru
void Empty();           ///< Vyprázdní oba seznamy vrstev
void MoveUp(int index); ///< Přesune vrstvu na daném indexu v pořadí
                              blíže ke kameře
void MoveDown(int index);    ///< Přesune vrstvu na daném indexu v
                                   pořadí dále od kamery
```

Jejich úlohy jsou patrné z komentářů.

Za sestavení výsledného vzoru je zodpovědná funkce

```
int BuildMasterImage(bool bPreview, bool useFile = false, const
                     char* fileName = "");
```

První parametr určuje, zda chceme zobrazit náhled (*true*) testovacího vzoru nebo zda se půjde o vytvoření vzoru v plném rozlišení (*false*). Druhý určuje, zda se jako testovací vzor použije soubor (*true*) nebo vlastní vzor (*false*). V případě použití vzoru ze souboru je cesta uvedena v posledním parametru.

3.4.6 Vytvoření nové třídy geometrického primitiva

Při vytváření nového grafického primitiva je nejprve nutno určit jak bude vypadat jeho analytický popis. Z toho se určí parametry, které bude nový vzor mít. Tyto parametry budou privátní proměnné nově navržené třídy. Tato třída reprezentující nový vzor dědí z bázové třídy *CPrimitiveBase* a tudíž má přístup k jejím proměnným přes veřejné metody rozhraní.

Dalším krokem je vytvoření konstruktoru pro novou třídu, který se postará o inicializaci svých proměnných. Voláním konstruktoru základní třídy se pak nastaví zbytek.

Následuje přepsání virtuálních funkcí bázové třídy, tak aby plnily funkčnost v nové třídě. Jde konkrétně o tyto metody:

CStructures::ARGBPixel GetPixelValue(CStructures::Point position)

Tato metoda vrátí hodnotu pixelu na dané pozici. Je v ní nutno definovat algoritmus, podle kterého se zjistí, zda pixel je součástí vzoru nebo pozadí a podle toho vrátí jeho barvu.

void Translate(int dx, int dy)

Funkce zajišťující posunutí vzoru v jeho rovinně. Zde je potřeba přepočítat nové souřadnice bodů, které popisují daný vzor, podle rovnice (2.6).

void Rotate(double alfa)

Funkce zajišťující otočení vzoru v jeho rovinně. Opět je potřeba přepočítat nové souřadnice bodů, popisující vzor, podle rovnice (2.7)

V hlavičkovém souboru *CPrimitiveBase.h* je zapotřebí definovat hodnotu, která bude tento nový vzor jednoznačně určovat.

```
#define CIRCLE 0  ///< ID vrstvy circle
#define LINE 1    ///< ID vrstvy line
#define NEW 2     //Nová vrstva
```

Aby bylo možné novou třídu ukládat a načítat, musí se přidat do funkcí *Load* a *Save* z třídy *CAppMain* kód pro uložení, načtení parametrů. Posledním krokem je přetížení funkce *CreateLayer*, která zavolá správný konstruktor a přidá vrstvu do seznamu vytvořených.

```
CPrimitiveBase* CAppMain::CreateLayer(const char* title, int
distance, typ param1, typ param2, CStructures::ARGBPixel
Object, CStructures::ARGBPixel Env)
{
      TRACE("CNew\n");

      CPrimitiveBase* pPB;

      pPB = new CNew(distance, center, radius, Object, Env,
      title);
      m_patternManager.AddLayer(pPB);

      return pPB;
}
```

3.5 TŘÍDY FILTRŮ

Tato kapitola postupně popisuje všechny třídy, které jsou nutné pro simulaci optických vad čoček. Základ těchto tříd vychází z [1] a jsou zde hlavně popsány zásadní změny oproti původní verzi.

3.5.1 CFilterBase

Základní třída pro filtry zahrnující společné jejich společné vlastnosti. Tak jako u základní třídy geometrických objektů i zde se všechny členské proměnné nyní nachází v privátní části deklarace třídy. Současný stav vypadá takto:

```
private:
      static int num_filters; ///< Počet vytvořených instancí třídy

      int m_len;              ///< Délka řetězc
      int   m_Width;          ///< Šířka filtrtu
      int   m_Height;         ///< Výška filtru
      CStructures::CenterPoint m_Center;  ///< Souřadnice středu
                                          filtru
```

Následující proměnné jsou nově přidány

```
      char* m_FilterName;                          ///< Název filtru
      CStructures::Supersample m_supersample; ///< Poměr
                                              převzorkování
      bool m_bSave; ///< \c true  = vysledek transformace chceme
                    ulozit, \c false = nechceme
      bool m_bDisplay;  ///< \c true = vysledek chceme zobrazit, \c
                        false = nechceme
      bool m_bUse;      ///< filtr \c true = zařazen, \c false =
                        nezařazen do výpočtu
```

Proměnná *m_FilterName* představuje název filtru a pod tímto se bude i ukládat výsledek transformace konkrétního filtru. Zbylé proměnné *m_bSave*, *m_Display* a *m_bUse* mohou nabývat hodnot *true* nebo *false* a určují, zda se mezivýsledek uloží na disk, zobrazí během výpočtu a zda se daný filtr použije ve výpočetním řetězci. Tyto příznaky se nastavují pomocí veřejných metod rozhraní.

Další změnou je definování virtuální funkce

```
virtual void Transform(CPixelArray* pSource, CPixelArray* pTarget);
```

Ta provede transformaci vstupních obrazových dat *pSource* podle předpisu zkreslení, který daný filtr představuje. Výsledek je uložen do cílového úložiště dat *pTarget*, které může být zobrazeno a uloženo.

V původním návrhu základní třída obsahovala metodu s názvem *FillFilterData()*, která volala další metody pro nastavení filtrů. Definováním virtuální

funkce *Transform* v bázové třídě došlo k separaci úkolů, které nyní mohou být prováděny samostatně a tím lépe řídit běh výpočtu. S tím souvisí nadefinování nové funkce:

```
virtual CStructures::Point GetNextSize() const = 0;
```

která vrací rozměry pro další filtr v řadě.

3.5.2 CFilterCamera

Tato třída představuje pouze spojovací článek mezi programovým objektem kamery a řadou filtrů. Neprovádí žádnou obrazovou transformaci. Jejím výstupem je obraz odpovídající umístění kamery. Provedení třídy se je na podobném principu. Musí být předefinovány virtuální funkce z nového návrhu, tj.

Transform – metoda pouze zkopíruje vstupní data do svého výstupu

GetNextSize – rozlišení následujícího filtru bude stejné jako má kamera.

Změna se týká i konstruktoru

```
CFilterCamera::CFilterCamera(float CameraFocus, float
DistanceToCamera, const char* name)
	: CFilterBase(CameraFocus, DistanceToCamera, name)
{
	TRACE("-- Entering CFilterCamera constructor --\n");
	m_type = 0;
}
```

Ten nejprve zavolá konstruktor základní třídy, který inicializuje své parametry a poté se provede inicializace parametrů třídy zděděné. Proměnná m_type určuje typ filtru, který slouží pro identifikaci při ukládání a načítání projektu. Pro tento filtr má hodnotu 0.

3.5.3 CFilterBarrel

Třída představující geometrickou vadu čočky – soudkovitost. Její nové členské proměnné jsou

```
private:
	double m_paramBarrel;	///< parametr zkreslení
	int m_type;		///< ID filtru
```

Kde hodnota *m_paramBarrel* určuje velikost zkreslení.

Hodnota proměnné *m_type* identifikuje typ zkreslení, pro tento filtr nabývá hodnoty 1.

Konstruktor je deklarován takto:

```
CFilterBarrel(float CameraFocus = 0, float DistanceToCamera = 0,
const char* name = "Untitled", double param = 0);
```

Slouží zároveň jako defaultní konstruktor v případě vynechání parametrů.

Tělo konstruktoru vypadá následovně:

```
CFilterBarrel::CFilterBarrel(float CameraFocus, float
DistanceToCamera, const char* name, double param)
	: CFilterBase(CameraFocus, DistanceToCamera, name)
{
	TRACE("-- Entering CFilterBarrel constructor --\n");

	m_paramBarrel = param;
	m_type = 1;
}
```

Opět je vidět, že každá třída se stará o inicializaci pouze svých proměnných.

Předefinované virtuální funkce bázové třídy:

Transform – provede transformaci vstupních obrazových dat na základě rovnice ()

GetNextSize – vrátí rozlišení potřebné pro následující filtr v řadě v závislosti na velikosti m_paramBarrel

GetParam – vrátí hodnotu parametru m_paramBarrel

3.5.4 CFilterLightness

Třída představující jasovou vadu čočky – vinětace. Stejně jako u předchozí třídy má navíc členské proměnné

```
private:
	float m_paramLightness; ///< parametr zklreslení
	int m_type;             ///< ID filtru
```

Hodnota *m_paramLightness* určuje velikost jasového zkreslení a *m_type* identifikuje typ zkreslení, pro tento filtr nabývá hodnoty 2.

Princip volání konstruktorů je stejný jako u předchozí třídy, stejně tak předefinované virtuální funkce.

3.5.5 CFilterAlias

Třída reprezentující frekvenční vlastnosti obrazu. Tento filtr umožňuje zvětšit rozlišení vzoru na svém vstupu a na výstupu je obraz opět v původním rozlišení avšak s potlačeným alias jevem. Třída má stejné programové vlastnosti jako předchozí. Klíčová funkce je *Transform.* Ta provádí průměrování jasové hodnoty

subpixelů a ukládá jí na pozici velkého pixelu původního rozlišení. Postup výpočtu je takový, že se prochází pole pixelů se zvětšeným rozlišením a to tak, že se indexace pole postupně přesouvá o krok rovný počtu subpixelů. Tím je vždy ukázáno na jeho začátek. Od tohoto začátku se vezme daný počet subpixelů, sečte se jejich hodnota a vydělí se jejich počtem. Tato hodnota se uloží na odpovídající index v poli původního rozlišení. Takto se z obrazu odsraní ostré přechody.

3.5.6 CFilterManager

Tato třída stejně jako *CPatternManager* zahrnuje vytváření filtrů, jejich řazení v seznamech a zajišťuje řízení výpočtu zkreslení. Proměnné v privátní části jsou tyto:

```
private:
      CObList m_listCreated;        ///< Seznam vytvořených filtrů
      CObList m_listRegistred;      ///< Seznam zařazených filtrů do
                                          výpočtu
      POSITION m_pos;               ///< Ukazatel pozice v
                                          seznamech
      CPixelArray* m_pSourceArray;  ///< Datové uložiště pro vstupní
                                          obraz
      CPixelArray* m_pTargetArray;  ///< Datové uložiště pro
                                          výstupní obraz
```

Jak už vyplývá z komentářů, proměnné *m_listCreated* a *m_listRegistred* představují seznamy pro vytvořené a zařazené filtry. Dále pak `m_pSourceArray` a *m_pTargetArray* jsou datová pole pro vstupní a výstupní obrazy.

Funkce pro přidávání filtrů do seznamů a práci s nimi jsou stejné jako u třídy *CPatternManager*.

Důležitou metodou je

```
bool PrepareFilters(CCamera* pCamera, CPatternManager*
                                                pPatternManager);
```

která nejdříve zjistí, zdali je v seznamu pro výpočet zařazen nějaký filtr, pokud ano prochází seznam směrem od kamery k testovacímu vzoru. Pro každý filtr nastavuje jeho potřebnou velikost. Počáteční rozlišení je bráno z objektu kamery, což je první předávaný parametr. Na závěr je nastaveno rozlišení testovacího vzoru, které je stejně velké jako u posledního filtru.

Samotný výpočet se spustí voláním funkce

```
bool StartProcess();
```

která prochází seznam zařazených filtrů zpět ke kameře a postupně pro každý filtr volá funkci *Transform.* Té předává ukazatel na zdroj obrazu, který má filtr zkreslit a ukazatel kam má výsledek uložit. Takto se postupně projdou všechny zařazené filtry.

3.5.7 Vytvoření nové třídy filtru

V první řadě je potřeba stanovit matematický popis nového zkreslení. Vztah by neměl být příliš složitý, proto je zapotřebí provést vhodnou aproximaci, tak aby popis byl co nejjednodušší a přitom stále poskytoval věrohodné výsledky. Z tohoto matematického popisu vyplývají parametry, které bude nová třída mít. Tyto parametry budou privátní proměnné. Při samotné implementaci je postup následující. Nová třída je odvozená od bázové třídy *CFilterBase* a tím pádem dědí všechny její vlastnosti. Což jsou parametry a metody společné všem filtrům. Dále je nutno napsat konstruktor pro novou třídu, který nastaví její parametry. Poté je zavolán konstruktor třídy bázové, který se postará o své parametry. Příklad takového konstruktoru:

```
CFilterNew::CFilterNew(float CameraFocus, float
DistanceToCamera,const char* name, double param)
      : CFilterBase(CameraFocus, DistanceToCamera, name)
{
TRACE("-- Entering CFilterNew constructor --\n");
m_paramNew = param;
}
```

Následuje přepsání virtuálních funkcí základní třídy tak, aby plnily funkčnost danou typem zkreslení. Jedná se o tyto metody:

void SetParam(typ parametr1,…)

Tato metoda přistupuje k privátním proměnným nové třídy a nastavuje jejich hodnoty dle požadavků.

typ GetParam() const

Metoda vrací parametr(y) nově navržené třídy.

CStructures::Point GetNextSize() const

V této metodě dochází k výpočtu velikosti následujícího filtru, který může být zařazen do výpočetního řetězce. Tato velikost slouží pro alokování potřebného

paměťového místa pro následující filtr v řadě. Velikost závisí na parametrech tohoto filtru.

void Transform(CPixelArray pSource, CPixelArray* pTarget)*

V této metodě se musí implementovat matematický popis zkreslení. Postupně se prochází všechny pixely ze zdrojového obrazu a transformují se na základě daného vztahu.

V hlavičkovém souboru *CFilterBase.h* je nutno nadefinovat konstantu pro identifikaci nové třídy

```
#define CAMERA 0  ///< ID filtru camera
#define BARREL 1  ///< ID filtru barrel
#define LIGHTNESS 2     ///< ID filtru lightness
#define ALIAS 3         ///< ID filtru alias
#define NEW 4           //Nový filtr
```

V třídě *CAppMain* je zapotřebí dopsat kód do funkcí Load, Save pro správné uložení parametrů nového filtru a do funkce *CreateFilter* kde se na základě identifikační hodnoty volá příslušný konstruktor třídy.

```
CFilterBase* CAppMain::CreateFilter(int type, const char* title,
double param, int bSave, int bDisplay)
{
      CFilterBase* pFB;

      switch (type)
      {
            case CAMERA:
                  TRACE("Camera\n");
                  pFB = new CFilterCamera(0, 0, title);
                  pFB->Save(bSave);
                  pFB->Display(bDisplay);
                  m_filterManager.AddFilter(pFB);
            break;

            case BARREL:
                  TRACE("Barrel\n");
                  pFB = new CFilterBarrel(0, 0, title, param);
                  pFB->Save(bSave);
                  pFB->Display(bDisplay);
                  m_filterManager.AddFilter(pFB);
            break;

            case LIGHTNESS:
                  TRACE("Lightness\n");
                  pFB = new CFilterLightness(0, 0, title, param);
                  pFB->Save(bSave);
                  pFB->Display(bDisplay);
                  m_filterManager.AddFilter(pFB);
```

```
            break;

            case NEW:
                  TRACE("New\n");
                  pFB = new CNew(0, 0, title, param);
                  pFB->Save(bSave);
                  pFB->Display(bDisplay);
                  m_filterManager.AddFilter(pFB);
            break;

            default:
                  pFB = NULL;
                  TRACE("Neni vybran typ zkresleni\n");

      }

      return pFB;
}
```

3.6 TŘÍDY PRO UKLÁDÁNÍ

Pro práci s XML byly využity již naprogramované třídy zahrnující funkční XML parser. Jedná se o balíček TinyXML, který je dostupný na [7]. Tyto třídy mají v projektu na začátku názvu předponu TiXML.

3.7 TŘÍDA APLIKACE

Tato třída je určena pro komunikaci uživatele s aplikací. Zapouzdřuje všechny doposud popsané třídy tak, aby uživatel byl schopen skrze tuto třídu používat všechny důležité funkce. Pokud by se vytvářelo grafické uživatelské rozhraní, bude komunikovat jen s touto třídou.

Dále jsou popsány důležité vlastnosti této třídy. Privátní část deklarace obsahuje tyto proměnné:

```
private:
      CCamera m_camera; ///< Objekt snímacího zařízení
      CPatternManager m_patternManager;   ///< Správce testovacího
                                              vzoru
      CFilterManager m_filterManager;     ///< Správce filtrů
```

Jedná se o instance tříd kamery, a správce vzorů a filtrů. Pomocí nich je dále řízen běh aplikace.

Následuje popis metod rozhraní k jednotlivým částem. Pro objekt m_camera je zde jediná metoda

```
void SetCamera(int width, int height, int distanceO);
```

která nastaví zadané parametry kamery a tím ji uvede do připraveného stavu.

Metody rozhraní pro správce testovacího vzoru jsou

```
CPrimitiveBase* CreateLayer(const char* title, int distance,
CStructures::CenterPoint center, int radius, CStructures::ARGBPixel
Object, CStructures::ARGBPixel Env);
```

a

```
CPrimitiveBase* CreateLayer(const char* title, int distance,
CStructures::Point begin, CStructures::Point end, int thick,
CStructures::ARGBPixel Object, CStructures::ARGBPixel Env);
```

Podle zadaných parametrů se zavolá příslušná funkce pro vytvoření vrstvy s obrazcem kruhu nebo úsečky a následně se zařadí do seznamu vytvořených vrstev. Je-li vrstva úspěšně vytvořena, funkce na ni vrátí ukazatel. Ten může být využit pro zaregistrování této vrstvy do seznamu pro testovací vzor pomocí funkce

```
void RegisterLayer(CPrimitiveBase* pLayer, int index);
```

kde index určuje pozici vrstvy ve vzoru. Zaregistrovat lze i vrstvu, která je už v seznamu vytvořených a to pomocí této funkce

```
void RegisterLayer(int posFrom, int posTo);
```

Kde první parametr udává index vrstvy v seznamu vytvořených a druhý, na kterou pozici chceme vybranou vrstvu uložit. Zavoláním této metody bez parametrů se automaticky zařadí do vzoru všechny vytvořené vrstvy.

Pro odstranění vrstvy ze vzoru, na zvolené pozici, stačí zavolat funkci

```
void RemoveRegistredLayer(int index);
```

Pro posunutí či natočení vzoru slouží funkce

```
void RotateLayer(double angleX, double angleY, double angleZ, int
                                                              layerPos);
void TranslateLayer(int dx, int dy, int layerPos);
```

které se postarají o správné zavolání všech metod pro natočení a posunutí.

Následující metoda slouží pro rychlý náhled sestaveného vzoru před samotnou simulací.

```
int LayerPreview(int width, int height, bool useFile, const char*
fileName = "");
```

Pokud je umístěna kamera, nastaví se rozlišení vzoru podle prvních dvou parametrů a je sestaven testovací vzor, který se zobrazí bez ovlivnění filtrů, pokud jsou nějaké zařazeny.

Pro přesouvání vrstev ve vzoru slouží tyto dvě metody

```
void MoveUpLayer(int index);  ///< Posun vrstvy v pořadí vpřed
void MoveDownLayer(int index);     ///< Posun vrstvy v pořadí vzad
```

Metody pro práci s filtry jsou podobné a chovají se stejně, pouze pracují s objektem m_filterManager. Proto je zde pro úplnost pouze jejich výpis

```
CFilterBase* CreateFilter(int type, const char* title, double param,
int bSave, int bDisplay);
void RegisterFilter(int posFrom, int posTo);
void RegisterFilter();        ///< Zařadí do výpočtuy všechny filtry
void RemoveRegistredFilter(int index);///< Odstraní filtr ze seznamu
                                        pro výpočet na dané pozici
void RemoveRegistredFilter(); ///< Odstraní všechny filtry v seznamu
                                 pro výpočet
void MoveUpFilter(int index); ///< Posun filtru v pořadí vpřed
void MoveDownFilter(int index);    ///< Posun filtru v pořadí vzad
```

Do této části také patří metoda

```
int RunSimulation(bool useFile, const char* fileName = "");
```

která spustí celý řetězec výpočtu. Ukázka těla

```
int CAppMain::RunSimulation(bool useFile, const char* fileName)
{
      //musi byt nastavena kamera
      if(m_camera.IsReady())
      {//pokracovani simulace

            //vypocet rozmeru vsech filtru
            cout << "Preparing filters...\n";
            if(!m_filterManager.PrepareFilters(&m_camera,
                                          &m_patternManager))
                  TRACE("\nNejsou vybrany filtry pro vypocet\n");

            //vyber predlohy
            {
      m_patternManager.SetProjDistance(m_camera.GetDistanceToO());
      //vzdalenost projekcni roviny = vzdalenost kamery od pocatku
      m_patternManager.BuildMasterImage(false, useFile,
                                    fileName); //vytvoreni predlohy

      m_filterManager.LoadPattern(m_patternManager.GetPattern());
      //predani predlohy filtrum
            }
            //spusteni vypoctu
            cout << "Processing...\n";
            if(m_filterManager.StartProcess())
                  return 0;
```

```
		else
			return 1;
	}
	else
	{
		TRACE("\nNejsou nastaveny parametry kamery!\n");
		return 3;
	}
```

Nejprve se zkontroluje, zda je připravena kamera. Pokud ne, funkce se ukončí a vrátí chybový kód určující příčinu neúspěchu. Je-li připravena, pokračuje se dál a přes správce filtrů se zavolá metoda pro výpočet rozměrů všech filtrů zařazených do výpočtu. Pokud vše proběhlo v pořádku, tak se přes správce vzoru nastaví vzdálenost projekční roviny a sestaví se testovací vzor. Ten je poté předán správci filtrů, který ho nastaví jako vstupní obraz pro první filtr v řadě výpočtu. Poté je zahájen samotný výpočet, kdy se prochází všechny filtry postupně od testovacího vzoru ke kameře a provádí se definovaná zkreslení. Výsledek je poté uložen v posledním filtru v řadě (nejblíže kameře).

Poslední část třídy tvoří metody pro práci s projektem, tj. ukládání a načítání. Ukázka části kódu, která se stará o uložení projektu. Je zde pouze část starající se o uložení nastavení kamery:

```
void CAppMain::Save(const char *pFileName)
{
TRACE("\n------------------ Saving project ------------------\n");

	TiXmlDocument doc;
	TiXmlDeclaration* decl = new TiXmlDeclaration("1.0","","");
	doc.LinkEndChild(decl);

	//Nazev muze byt nahrazen jmenem ukladaneho projektu
	TiXmlElement* root = new TiXmlElement("Project");
	doc.LinkEndChild(root);

	//blok: Camera
	{
		CStructures::Point size;
		size = m_camera.GetSize();

		TiXmlElement* camera = new TiXmlElement("Camera");
		root->LinkEndChild(camera);
		camera->SetAttribute("width", size.X);
		camera->SetAttribute("height", size.Y);
		camera->SetAttribute("distanceO",
		m_camera.GetDistanceToO());
	}
```

Vždy je zapotřebí vytvořit element, který je spojen s nadřazeným elementem a poté nastavit jeho atributy. Hodnoty jsou brány z příslušných objektů kamery, filtrů a vzorů.

Právě opačně funguje metoda pro načtení projektu. Zde ukázka načtení stejného bloku, tj. kamery:

```
int CAppMain::Load(const char *pFileName)
{
TRACE("\n------------------- Loading project ------------------
\n");
	TiXmlDocument doc(pFileName);
	if(!doc.LoadFile()) return 1;

	//vyprazdneni vsech seznamu pro nove nacteni
	m_filterManager.Empty();
	m_patternManager.Empty();

	TiXmlHandle hDoc(&doc);
	TiXmlElement* pElem;
	TiXmlHandle hRoot(0);

	pElem = hDoc.FirstChildElement().Element();
	if (!pElem) return 2;

	hRoot = TiXmlHandle(pElem);

	//blok: Camera
	{
		cout << "\tCamera...\n";
		CStructures::Point size;
		int distanceO;

		pElem = hRoot.FirstChild("Camera").Element();
		if (!pElem) return 2;

			pElem->QueryIntAttribute("width", &size.X);
			pElem->QueryIntAttribute("height", &size.Y);
			m_camera.SetSize(size.X, size.Y);

			pElem->QueryIntAttribute("distanceO", &distanceO);
			m_camera.SetDistanceO(distanceO);
			m_camera.SetReady(true);

		TRACE("Camera...\n");
		TRACE("width: %d\nheight: %d\ndistanceToOrigin: %d\n",
size.X, size.Y, distanceO);

	}
```

Postupným procházením struktury XML najdeme element s požadovaným pojmenováním, získáme hodnoty jeho atributů a tyto hodnoty se použijí pro nastavení příslušného objektu, v tomto případě kamery.

4. UŽIVATELSKÝ MANUÁL

Tato kapitola se zaměřuje na popis používání aplikace pro uživatele. Nejprve je vysvětleno jak správně vytvořit XML soubor, pomocí kterého lze nadefinovat požadovaný vzor a filtry zkreslení. Následuje část popisující práci se samotnou aplikací.

4.1 VYTVOŘENÍ XML SOUBORU

V následujícím textu je popsáno, jak napsat správně XML soubor, podle kterého se sestaví optické prostředí pro simulaci. Struktura souboru je dělena na elementy, které mají svoji hierarchii.

Cela konfigurace projektu je uzavřena v hlavním kořenovém elementu <Project></Project>. Ten může obsahovat další čtyři elementy

<Settings></Settings> - element je povinný, obsahuje nastavení projektu

<Camera></Camera> - element je povinný, obsahuje nastavení kamery

- atributy elementu
  - width – šířka v pixelech
  - height – výška v pixelech
  - distanceO – vzdálenost od počátku optické soustavy v pixelech

<Pattern></Pattern> - nepovinný element, obsahuje seznam vrstev vzoru

- atributy elementu
  - name – pojmenování výsledného testovacího vzoru. Musí být uvedena přípona určující formát ukládaného obrazu
  - display – 1 = vzor se při výpočtu zobrazí
    0 = vzor se nezobrazí
  - save – 1 = vzor se uloží do souboru
    0 = vzor se nebude ukládat

<Filters></Filters> - nepovinný element, obsahuje seznam filtrů

Základní struktura dokumentu pak vypadá následovně

```
- <Project>
    + <Settings>
      <Camera width="800" height="600" distanceO="545" />
    + <Pattern name="Vzor1.jpg" display="1" save="1">
    + <Filters>
</Project>
```

V elementu <Settings> jsou dva vnořené elementy

<SavePath></SavePath> - element obsahuje cestu k ukládání projektu
- není zcela implementováno, hodnota je ignorována a projekt se ukládá do aktuálního umístění

<File type = ““/> - atribut elementu určuje příponu ukládaných obrázků
- chybí implementace, přípona musí být určena v konkrétních názvech

Příklad obsahu elementu <Settings>

```
- <Settings>
    <SavePath>C:\Projects\</SavePath>
    <File type=".jpg" />
</Settings>
```

Element <Pattern> obsahuje seznam vrstev vzoru v podobě libovolného množství vnořených elementů

<layer></layer> - vrstva vzoru reprezentující grafické primitivum
- atributy elementu
  - type – typ grafického objektu.
    0 = kruh, 1 = úsečka
  - title – pojmenování vrstvy
  - use – 1 = vrstva se použije ve vzoru
    0 = vrstva se nepoužije ve vzoru
  - atPosition – určuje pořadí vrstvy ve vzoru. Nejprve se zpracovává vrstva s indexem 0. Pokud je hodnota -1, je vrstva zařazena na konec seznamu.

Ukázka obsahu elementu <Pattern> se čtyřmi vrstvami

```
- <Pattern name="Vzor1.jpg" display="1" save="1">
    + <layer type="1" title="linka" use="1" atPosition="1">
    + <layer type="0" title="Kruh" use="0" atPosition="-1">
    + <layer type="1" title="Linie" use="0" atPosition="-1">
    + <layer type="0" title="Kolecko" use="1" atPosition="0">
</Pattern>
```

Každý element <layer> navíc obsahuje další elementy popisující charakteristické vlastnosti každého grafického primitiva. Zde se počet a názvy elementů mohou lišit v závislosti na typu grafického objektu. Aplikace umožňuje vytvářet dva typy objektů. Pro kruh jsou to elementy

 - souřadnice středu kruhu v pixelech

- atributy
  - x – souřadnice na ose x
  - y – souřadnice na ose y

<radius></radius> - poloměr kruhu v pixelech

- atributy
  - r – poloměr kruhu

Pro úsečku jsou to elementy

<begin></begin> - souřadnice počátečního bodu úsečky

- atributy
  - x – souřadnice na ose x
  - y – souřadnice na ose y

<end></end> - souřadnice koncového bodu úsečky

- atributy
  - x – souřadnice na ose x
  - y – souřadnice na ose y

<thickness></thickness> - tloušťka úsečky

- atributy
  - t – tloušťka úsečky

Společné elementy pro všechny vrstvy

<distance></distance> - vzdálenost vrstvy od počátku soustavy

- atributy
  - d – vzdálenost vrstvy od počátku soustavy

<angle><angle> - úhly natočení vrstvy

- atributy
  - alfa – úhel natočení okolo osy x [°]
  - beta – úhel natočení okolo osy y [°]
  - gama – úhel natočení okolo osy z [°]

<color><color> - definice barev vrstvy

- elementy
  - <object> – barva výplně objektu
    - atributy
      - A – průhlednost. Hodnoty 0 – 255.
        0 = průhledný
        255 = neprůhledný
      - R – červená barevná složka (0 - 255)
      - G – zelená barevná složka (0 - 255)
      - B – modrá barevná složka (0 - 255)
  - <env> – barva pozadí objektu
    - atributy stejné jako element <object>

<period><period> - opakování vzoru v rovině

- elementy
  - <repeat> - velikost periody opakování vzoru
    - atributy
      - x – velikost periody v ose x
      - y – velikost periody v ose y
  - <count> - počet opakování vzoru
    - atributy
      - x – počet opakování v ose x
      - y – počet opakování v ose y

Ukázka struktury elementu <layer>

```
- <layer type="1" title="linka" use="1" atPosition="1">
    <distance d="845" />
    <begin x="0" y="-50" />
    <end x="0" y="50" />
    <thickness t="40" />
    <angle alfa="0" beta="0" gama="45" />
   - <color>
      <object A="255" R="255" G="255" B="255" />
      <env A="255" R="0" G="0" B="0" />
     </color>
   - <period>
      <repeat x="140" y="160" />
      <count x="-1" y="-1" />
     </period>
  </layer>
```

Element <Filters> obsahuje seznam filtrů v podobě libovolného množství vnořených elementů

<filter> - popisuje filtr realizující optickou vadu

- atributy
  - type – identifikuje typ filtru,
    0 – kamera, 1 – barrel, 2- lightness, 3 - alias
  - title – název pro výstupní obraz filtru. Přípona je povinná a určuje formát pro uložení.
  - param – hodnota parametru zkreslení
  - save – uložení výsledného obrazu filtru 0=Ne, 1=Ano
  - display – zobrazení výsledného obrazu 0=Ne, 1=Ano
  - use – použití filtru pro výpočet 0=Ne, 1=Ano
  - atPosition – pozice filtru ve výpočetním řetězci, index 0 je nejblíže ke kameře. Hodnota -1 zařadí filtr na konec seznamu.

Ukázka obsahu elementu <Filters>

```
- <Filters>
    <filter type="0" title="Kamera.jpg" param="0" save="1" display="1"
      use="1" atPosition="-1" />
    <filter type="1" title="Soudek.jpg" param="0.001" save="1"
      display="1" use="1" atPosition="-1" />
    <filter type="1" title="Barrel.jpg" param="0.0057" save="1"
      display="0" use="0" atPosition="-1" />
    <filter type="2" title="Vinetace.jpg" param="0.96" save="1"
      display="1" use="1" atPosition="-1" />
  </Filters>
```

4.2 OVLÁDÁNÍ PROGRAMU

Aplikace se spouští pomocí **calibrs.exe.** Po spuštění se objeví konzole a uživatel je vyzván k zadání souboru pro načtení projektu.

```
Project name to open:
```

Je potřeba zadat platný název souboru XML, ve kterém je uložena konfigurace projektu, např. **project.xml.**

Pokud je soubor nalezen, proběhne načtení projektu. O průběhu načítání je uživatel informován na obrazovce. Pokud vše proběhne bez chyby, je uživatel vyzván k výběru, zda chce jako testovací vzor použít obraz se souboru nebo vlastní.

```
Use (F)ile or (P)attern?:
```

Zadáním **f** se použije soubor, na jehož umístění je uživatel následně dotázán. V případě zvolení **p** se použije vlastní nadefinovaný vzor.

Pak je uživatel dotázán, zda chce zobrazit náhled vzoru

```
Preview pattern? Y/N:
```

Y – testovací vzor se zobrazí jako náhled, tedy bez žádného zkreslení, **N** – nezobrazí

Další dotaz je na spuštění simulace.

```
Run simulation? Y/N:
```

Po zadání **Y** se spustí simulace a uživatel je o jejím průběhu informován. Nejprve se sestaví testovací vzor v plném rozlišení. Je-li nastaven příznak pro zobrazení, zobrazí se a pro pokračování je nutné mít aktivní okno obrazu a stisknout jakoukoli klávesu. Je-li nastaven příznak pro uložení, výsledek se zároveň uloží a začnou se zpracovávat postupně všechny filtry. Ty opět mohou zobrazit (uložit) své mezivýsledky. Pro pokračování je vždy nutné stisknout libovolnou klávesu v aktivním okně obrazu.

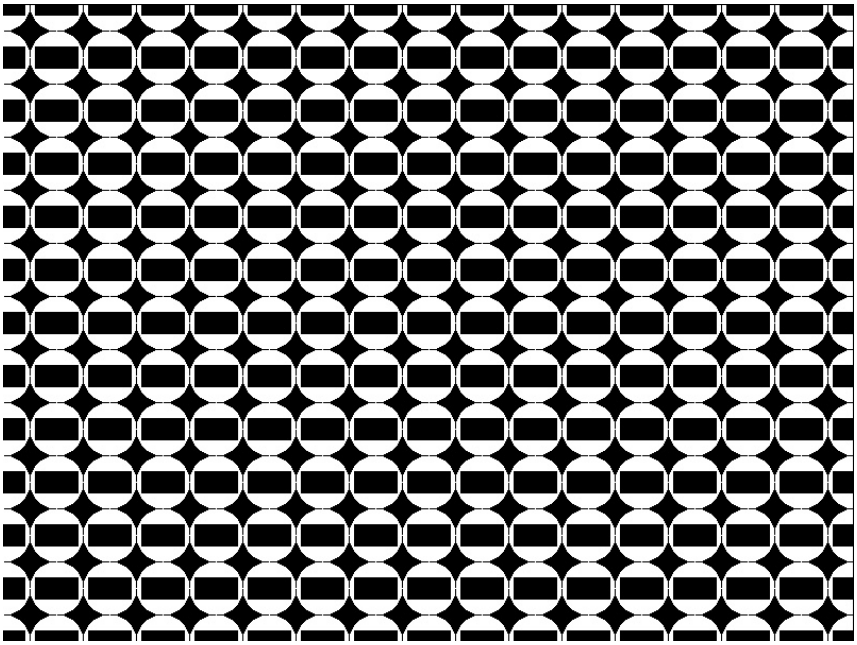
Po skončení simulace jsou případné obrázky uloženy v aktuálním adresáři aplikace. Nakonec je uživatel dotázán na pokračování.

```
Continue? Y/N?
```

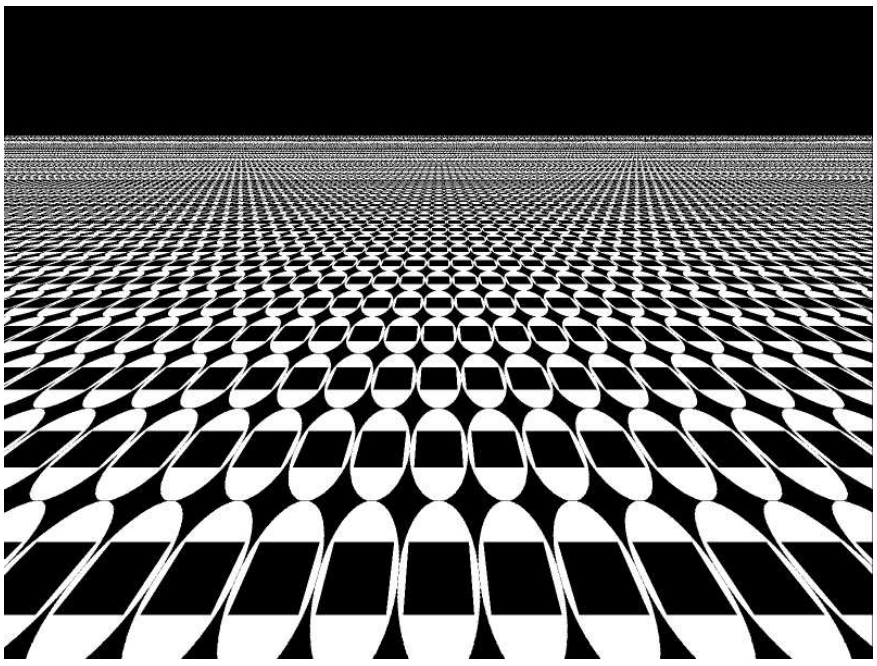
Zadáním **Y** se program opět dotáže na název souboru projektu a celý postup se opakuje. Program se ukončí zadáním **N**.

4.3 UKÁZKY VZORŮ A ZKRESLENÍ

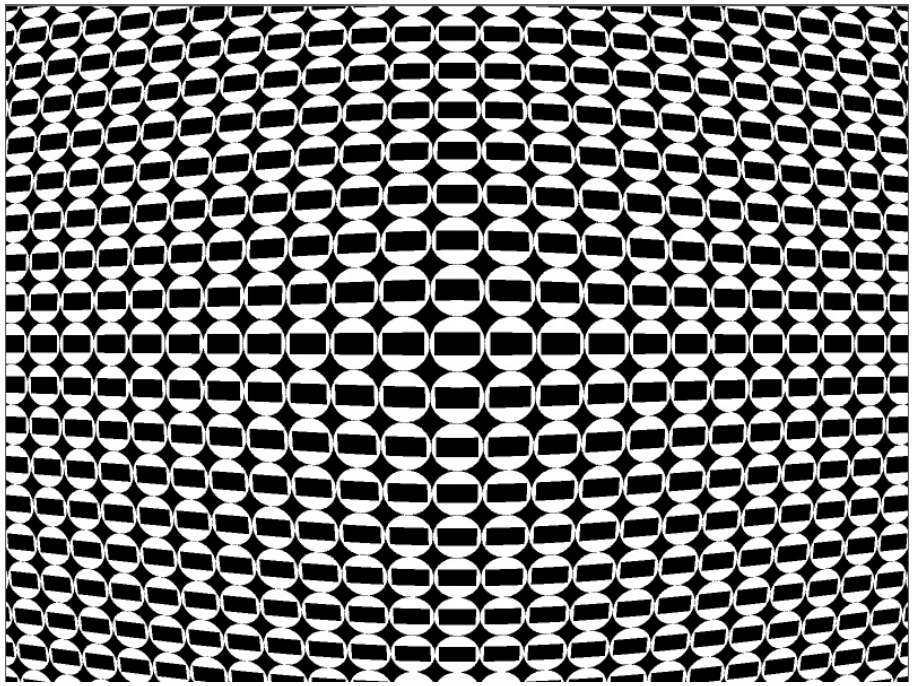
V této kapitole jsou ukázány výsledky práce v podobě výstupních obrázků.



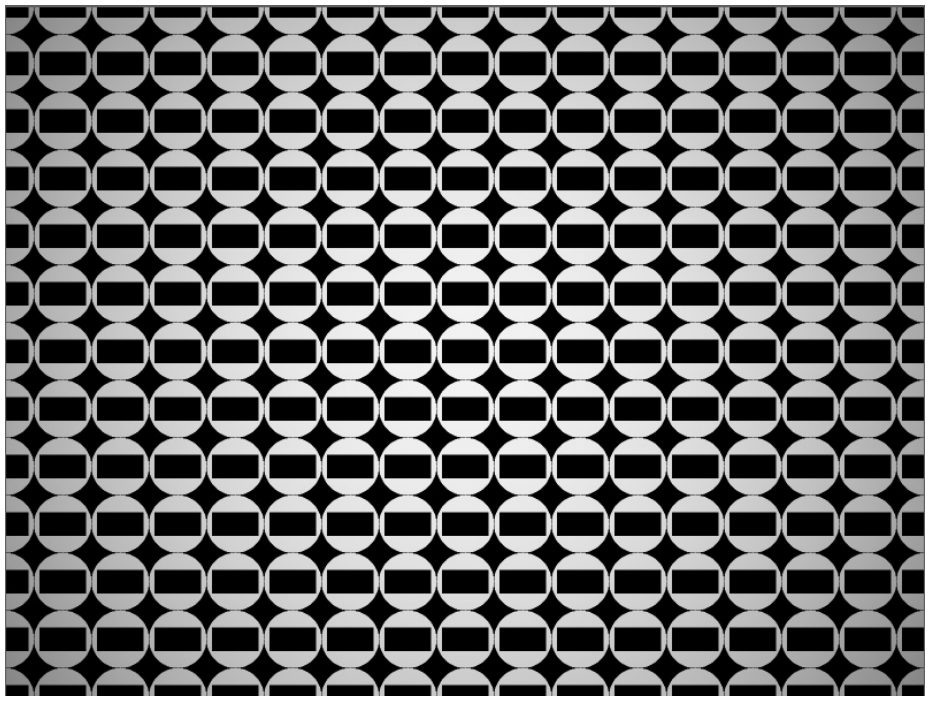
Obrázek 4-1 Testovací vzor složený s bílých kruhů a černých úseček.



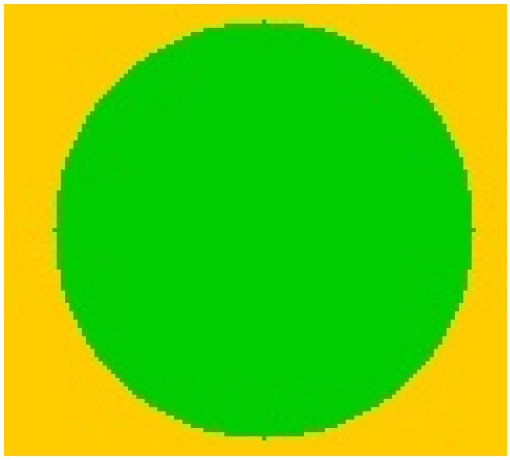
Obrázek 4-2 Vzor z obrázku 4-1 sklopený o 70° okolo osy x s použitím 4x antialiasingu



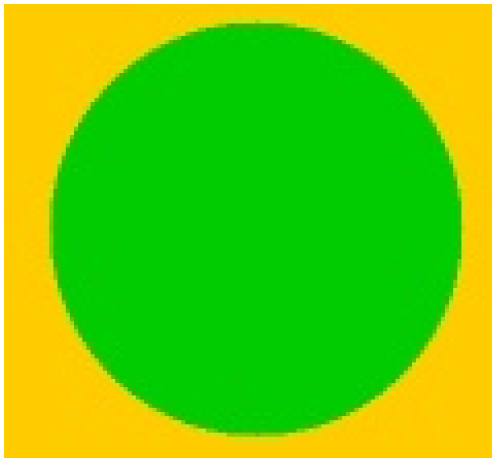
Obrázek 4-3 Vzor se soudkovitým zkreslením. Parametr r=0,0005



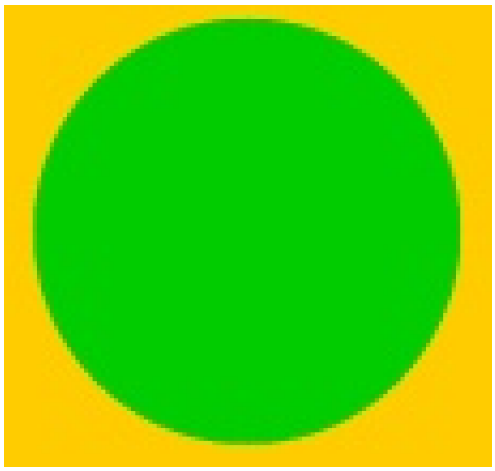
Obrázek 4-4 Vzor s jasovým zkreslením, koeficient c=0,95



Obrázek 4-5 Vykreslení kruhu bez použití antialiasingu



Obrázek 4-6 Vykreslení kruhu s použitím 4x antialiasing



Obrázek 4-7 Vykreslení kruhu s použitím 9x antialiasing

5. ZÁVĚR

V rámci této diplomové práce byla rozšířena aplikace pro vytváření testovacích vzorů a simulaci optických vad čoček.

Testovací vzor lze poskládat z jednotlivých vrstev základních grafických objektů. V aplikaci jsou k dispozici kruh a úsečka. S těmito útvary je možno posouvat a rotovat v rámci roviny. Celou rovinu pak lze natočit v prostoru. O sestavení výsledného vzoru z jednotlivých vrstev, se stará nově navržená třída *CPatternManager*. Ta má také za úkol tyto vrstvy vytvářet a dle potřeby zařazovat do testovacího vzoru. Tím je umožněno jeho snadné a variabilní sestavování.

Optické zkreslení je realizováno pomocí tříd filtrů. O jejich vytváření a řízení během výpočtu se stará nová třída *CFilterManager*. Ta má na starosti vkládání a rušení nových filtrů. Také přidělování paměti při výpočtu, tak aby byla alokována paměť, která je v daný okamžik skutečně potřeba. Během simulace se alokuje paměť jen pro zdrojový a výstupní obraz a nikoli pro všechny filtry najednou. Byl přidán nový filtr reprezentující frekvenční vlastnosti obrazu. Jedná se o potlačení alias efektu v obraze. Tento filtr umožňuje navzorkovat testovací vzor ve vyšším rozlišení a průměrováním hodnot subpixelů zobrazit výsledek v původním rozlišení. Dojde tak k odstranění zubatých hran v obraze.

Pro snadné definování vlastností optické soustavy je možné projekt načítat z XML souboru, ve kterém jsou zapsány potřebné informace. Tento soubor je možné upravovat v běžném textovém editoru. Nahrazuje tak absenci grafického uživatelského rozhraní.

6. LITERATURA

[1] ČERNÝ P., *Generování testovacích vzorů.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií. 45s. Vedoucí bakalářské práce Ing. Miloslav Richter Ph.D.

[2] J. Žára, B. Beneš, J. Sochor, P. Felkel: Moderní počítačová grafika, Computer Press Brno 2004, ISBN 80-251-0454-0

[3] D. Hearn, M. P.Baker: Computer Graphics with OpenGL, USA 2004, ISBN 0-13-120238-3

[4] David J. Kruglinski, Georgie Sheperd, Scot Wingo: Programujeme v Microsoft Visual C++, Computer Press Praha 2000, ISBN 80-7226-262-5

[5] *Knihovna MSDN* [on-line]. Dostupné na http://msdn.com.

[6] *OpenCV dokumentace* [on-line]. Dostupné na http://opencv.willowgarage.com/documentation/index.html

[7] *TinyXML source code* [on-line]. Dostupné na http://www.grinninglizard.com/tinyxml/

[8] *Geometrie, RayTracing* [on-line]. Dostupné na http://cs.wikibooks.org/wiki/Geometrie/Raytracing

[9] *Anti-aliasing* [on-line]. Dostupné na http://extrahardware.cnews.cz/anti-aliasing-dil-ii-bojujeme-proti-aliasu