Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra kybernetiky

BAKALÁŘSKÁ PRÁCE

Sledování objektu pomocí snímače Kinect

Plzeň 2014 **Filip Berka**

PROHLÁŠENÍ

Předkládám tímto k posouzení a obhajobě bakalářskou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž seznam je její součástí.

V Plzni dne ....

Poděkování

Rád bych poděkoval Ing. Marku Hrúzovi PhD. za odbornou pomoc, podporu a rady při zpracování bakalářské práce.

Dále bych chtěl poděkovat své rodině a přátelům.

Anotace

V teoretické části této práce jsou nastíněny teorie o počítačovém vidění, obraze a základních metodách jeho zpracování. Dále pak krátká kapitola věnovaná 3D grafice.

V praktické části je popsána naprogramovaná aplikace související s uvedeným teoretickým pozadím. Jde o sledování jednoduchého objektu 3D senzorem, jeho vykreslení ve 3D prostoru a nahrazení reálného objektu vykresleným.

Klíčová slova: Kinect, OpenCV, OpenGL, OpenNI, image processing, tracking, Particle Filter

Abstract

The theoretical part of this thesis concerns with theory of computer vision, image and basic methods of its processing. There is also a short chapter about 3D graphics.

In the practical part there is a description of the programmed application which is connected to mentioned theoretical background. It is about tracking a simple object with 3D sensor, its rendering in 3D space and replacement of the real object with the rendered one.

Key words: Kinect, OpenCV, OpenGL, OpenNI, image processing, tracking, Particle Filter

Obsah

1 Úvod **1**

1.1 Řešený problém 1

1.2 Současný stav ve světě 1

2 Kinect **3**

2.1 Generování hloubkové mapy 4

2.2 Propojení s počítačem 5

3 Zpracování obrazu **7**

3.1 Obraz 7

3.1.1 Definice a digitalizace 7

3.1.2 Vlastnosti obrazu 9

3.1.3 Kvalita obrazu 10

3.1.4 Implementace v OpenCV 11

3.2 Zpracování obrazu 12

3.2.1 Segmentace 12

3.2.2 Matematická morfologie 14

3.2.3 Implementace v OpenCV 19

3.3 Použité metody 20

3.3.1 Algoritmus Suzuki85 20

3.3.2 Particle Filter 23

4 3D grafika **25**

4.1 OpenGL 25

4.1.1 Historie 25

4.1.2 Funkce OpenGL 26

5 Algoritmus programu **30**

5.1 Inicializace OpenGL 30

5.2 Inicializace Kinectu 31

5.3 Prahování 31

5.4 Estimace polohy kostky 33

5.5 Particle Filter 36

6 Problémy při zpracování **38**

6.1 Jas 38

6.2 Posouvání prahu 38
6.3 Odhad rotace 38
6.4 FPS 39

7 Závěr **40**

8 Literatura **41**

9 Příloha **42**

1 Úvod

Kybernetika je věda zabývající se mimo jiné i komunikací člověk - stroj.Ve světě je tato oblast známa jako HCI - Human Computer Interaction. Co je to vlastně komunikace? Podle online oxfordského slovníku je to výměna informací mluvením, psaním nebo skrze jiné médium. Omezíme-li se na komunikaci člověk - stroj, kde stroj představuje počítač, může nás typicky napadnout klávesnice a myš jako vstupní zařízení a monitor či reproduktory jako výstupní zařízení. Dnes již existují i jiné způsoby komunikace mezi člověkem a počítačem. Nejlépe jsou tyto způsoby viditelné na chytrých telefonech. Srovnejme dnešní mobilní telefon s 10 let starým mobilním telefonem. Rozdíl je markantní: nové telefony jsou vybaveny dotykovými displeji, rozpoznáváním hlasových příkazů a ty nejmodernější umí rozeznávat gesta, jako například mávnutí rukou nebo polohu očí a podle toho automaticky scrollují čtený text.

1.1 Řešený problém

Cílem této práce bylo seznámit se se senzorem Kinect, knihovnou OpenCV a knihovnou OpenGL. Všechny tyto komponenty propojit s počítačem a naprogramovat real-time aplikaci, která bude zpracovávat hloubkovou mapu z Kinectu a stanovovat hypotézu o poloze jednoduchého tělesa. Toto těleso pak bude vykreslovat ve 3D prostoru a poté jím co nejpřesněji nahrazovat těleso skutečné. Jako těleso byla zvolena Rubikova kostka.

1.2 Současný stav ve světě

Řešená úloha v této práci se inspirovala prací autorů I. Oikonomidise, N. Kyriazise a A. A. Argyrose: *Efficient model based tracking of the articulated motion of hands* [1], aplikace, která využívá Kinectu ke sledování ruky a její rekonstrukci v OpenGL. Ruka na sobě přitom nemá žádné speciální značky nebo rukavici. Program funguje na základě prvotního pozorování ruky $O = (o_s, o_d)$, kde o_s je segmentovaná barva kůže a o_d k ní korespondující hloubková mapa. Modelovaná ruka v OpenGL se skládá z dlaně (eliptický válec) a pěti prstů (kombinace kuželů a koulí). Celkově se počítá s 26 stupni volnosti. Cílem je těchto 26 parametrů odhadnout tak, aby modelovaná ruka nejlépe odpovídala originálu. Tento odhad nebo hypotéza je nazvána h. Rozdíl mezi hypotézou h a observací O je získán funkcí $E(h,O)$:

$$E(h, O) = D(O, h, C) + \lambda_k \cdot kc(h)$$

kde λ_k je normalizační faktor, C je kalibrační informace, kc je koeficient znevýhodnění nepravděpodobných pozic ruky, funkce $D(O,h,C)$ je:

$$D(O,h,C) = \frac{\sum \min(|o_d - r_d|, d_M)}{\sum(o_s \cup r_m) + \epsilon} + \lambda \left(1 - \frac{2\sum(o_s \cap r_m)}{\sum(o_s \cap r_m) + \sum(o_s \cup r_m)}\right)$$

kde λ je konstantní normalizační faktor, ϵ je malá aditivní konstanta, aby se zabránilo dělení nulou, r_m je binární mapa vytvořená porovnáním o_d s r_d, což je hloubková mapa z OpenGL. Konkrétněji r_m je 1, pokud se obě mapy v týchž bodech liší maximálně o hodnotu d_M a 0 jindy.

Tím je získána hodnota rozdílu modelu a originálu. Aby bylo sledování ruky modelem přesné, musí být tato hodnota optimalizována. To se provádí pomocí technik Particle Filteru a Particle Swarm Optimisation (PSO). Pro podrobnější popis viz [1].

Cílem této práce bylo přiblížit se tomuto konceptu, nikoliv však s rukou, ale pouze s kostkou. V prvních částech bude nastíněna teorie použitá v programu. Kapitola 5 pak popisuje naprogramovanou aplikaci.

2 Kinect

Microsoft Kinect pro XBOX 360 (dále jen Kinect) a jiné pohybové 3D senzory jsou stále poměrně novou záležitostí ve světě. Zejména herní průmysl, pro který byly původně stvořeny, se tak rozrostl o další "joystick"a již nyní existuje mnoho her, které ovládá sám hráč pomocí gest a mluveného slova. Nejen herní průmysl však těží z jejich existencí. Od doby, kdy byl vyvinut ovladač pro připojení k počítači, se nadšenci pro computer vision snaží o rozšiřování svých aplikací o nové rozměry, zejména jde-li o hloubku nebo zvuk.

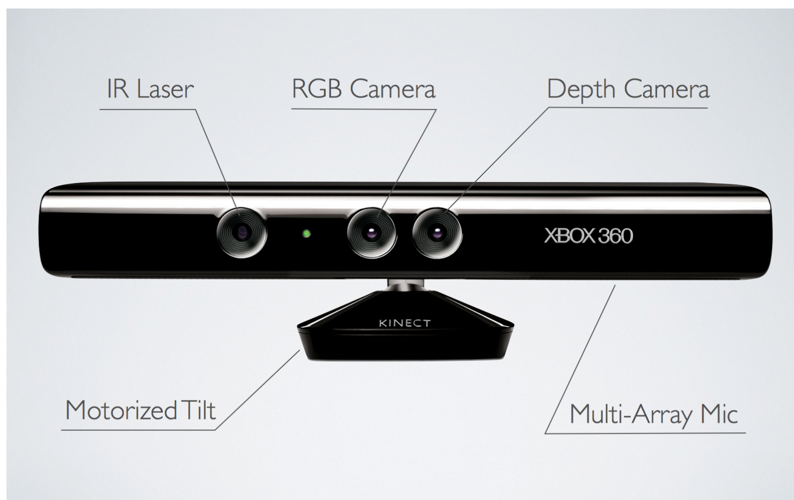
Kinect byl poprvé představen 1. června 2009 na konferenci Electronic Entertainment Expo pod názvem Projekt Natal , což znamená zrození (nové herní generace). Název byl 13. června 2010 změněn na Kinect [2], což je složenina ze slov kinetic (pohybový) a connect (připojit), což vystihuje záměry tohoto zařízení. Prodej v Americe začal 4. listopadu 2010 a v Evropě o týden později, 10. listopadu 2010. Od té doby se jich k datu 12. únor 2013 prodalo 24 milionů. Jedno zařízení přibližně za 2500 Kč [3]. Skládá se z RGB kamery, pole 4 mikrofonů a hloubkového senzoru, jenž dokáže určit vzdálenost objektů od kamery. Ze získaných dat dokáže Kinect identifikovat až 6 lidí. Snímání probíhá nezávisle na informacích o uživatelích nebo prostředí. Člověk přitom nepotřebuje žádné speciální oblečení nebo aby měl na svém těle připevněny značky. Je také doplněn o technologii voice recognition a hráči tak mohou mezi sebou komunikovat přes XBOX bez sluchátek [2].

Kinect vyrábí firma Rare ltd. patřící pod Microsoft Game Studios. Technologie použitá v Kinectu byla vynalezena izraelskou firmou PrimeSense. Jedná se o technologii, která používá infračervený projektor, infračervenou kameru a speciální mikročip, který sleduje objekty a jejich pohyb ve třídimenzionálním prostoru.

Nebude dlouho trvat, než se tyto technologie stanou standardní součástí televizí, počítačů i mobilů a pro ovládání domu budeme potřebovat pouze ruku nebo hlas.

2.1 Generování hloubkové mapy

Na Obrázku 1 jsou vidět hlavní části Kinectu: RGB kamera, infračervený zdroj, hloubkový senzor, pole 4 mikrofonů a motorizovaná základna.



Obrázek 1: Umístění hardwarových komponent v Kinectu, převzato z [4]

RGB kamera zachycuje standardní barevný 2D obraz, který má standardně při 30 Hz rozlišení 640x480 pixelů. Hardwarově je však při nižším *frame rate* schopna až rozlišení 1280x1024 pixelů.

Hloubkový senzor má dvě části: Infračervený zdroj a snímač infračerveného záření. Infračerveným zdrojem je laser, který promítá síť bodů do zorného pole kamery (Obrázek 2).

Obrázek 2: IR zdroj promítá síť bodů, převzato z [4]

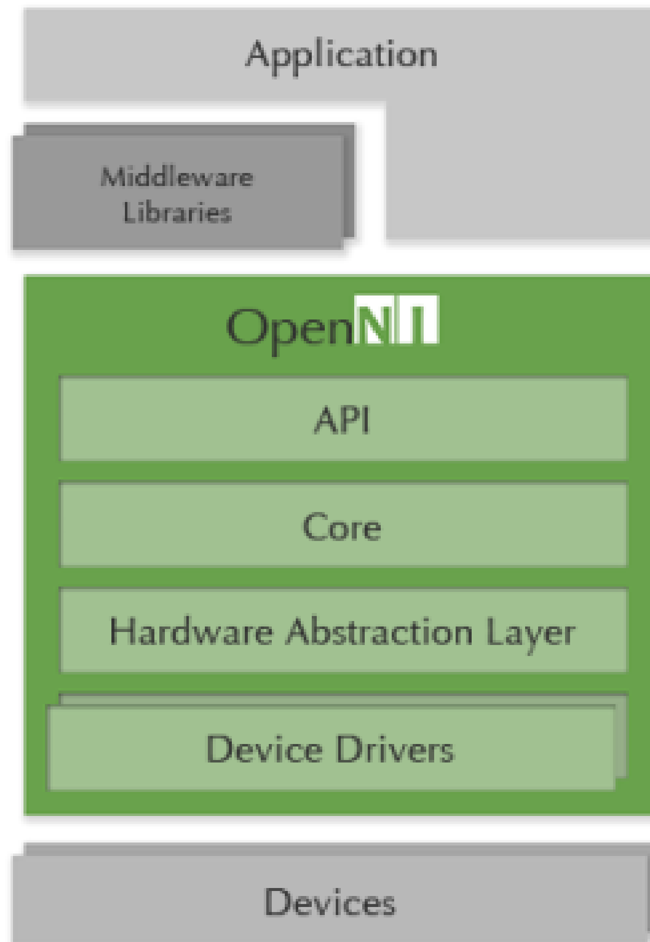
Tuto síť zpětně detekuje CMOS (Complimentary metal-oxide semicoductor) senzor. Podle přečtené sítě se vygeneruje hloubková mapa podle pravidla, že pokud jsou tečky hodně od sebe, objekt se nalézá blízko a pokud jsou tečky nahuštěny blízko sebe, objekt se nalézá dále. Tuto hloubkovou mapu pak posílá do XBOXU/počítače. Přesnost v ose Z je na 1 cm. V osách X a Y je přesnost na milimetr. Obrazové úhly jsou 43 stupňů vertikálně a 57 stupňů horizontálně. Hráči by se měli pohybovat v minimální vzdálenosti 1.2 m do 3.5 m. Od roku 2012 se však na trhu pohybuje vylepšená verze Kinect for Windows, který hloubku rozeznává už od 40 cm, dá se propojit s počítačem přes USB. "FOV"(field of view) se nezměnil. Informace převzaty z [5].

2.2 Propojení s počítačem

Microsoft zpočátku nevydal žádné ovladače ani SDK (software development kit), které by umožnily počítačové komunitě Kinect nějak využít. Společnost Adafruit Industries na toto reagovala tak, že vyhlásila soutěž o 2000 dolarů pro toho, kdo jako první naprogramuje open source ovladač. Poté, co Microsoft tuto soutěž odsoudil, vzrostla odměna soutěže až na 3000 dolarů. Po 6 dnech od vydání Kinectu na trh, byl vydán první ovladač. Po několika dalších open source ovladačích Microsoft prohlásil, že soutěž nijak neodsuzoval a že byl zvědavý, s čím komunita přijde. A tak bylo legálně povoleno nadšencům programovat ovladače a vývojová prostředí pro práci s Kinectem na počítači.

Jednou ze společností zabývajících se touto problematikou je *OpenNI* (Open Natural Interaction - otevřená přirozená interakce), kterou založili mimo jiné i členové původní firmy PrimeSense, která se zasloužila o vynalezení technologie použité v Kinectu. Tato společnost vytvořila multiplatformní open source framework, který definuje API (Application Programming Interface) používající přirozenou interakci se zařízeními.

Na obrázku 3 je popsána architektura komunikace mezi zařízeními (devices, např. Kinect), OpenNI a aplikací s různými rozšířeními, tzv. middlewary. Tyto middlewary jsou například sledování skeletu, rozpoznávání gest a především zmíněné sledování ruky a spousta dalších od komunity z celého světa.



Obrázek 3: OpenNI architektura, převzato z [6]

Společnost PrimeSense byla v listopadu roku 2013 koupena společností Apple za 350 milionů dolarů a 23. dubna byla zrušena stránka openni.org, kde se nalézaly všechny open source middlewary a knihovny.
Informace převzaty z [6].

3 Zpracování obrazu

Computer vision (dále jen CV) neboli počítačové vidění je relativně nový obor. Zabývá se vytvářením hw/sw schopných zpracovávat obrazové informace a na základě tohoto zpracování provádět různé další operace při znalosti nějaké informace z obrazu. Dnes se CV používá například v oblastech:

- detekce jevů
- interakce člověk-počítač
- analýza medicínských zobrazovacích technik
- automatické kontroly produktů
- rekonstrukce prostředí
- robotika
- bezpilotní letouny
- ovládání procesů

Přestože úloha počítačového vidění není obecně přesně definována, protože její cíle mohou být různé, byla za přibližně 40 let vývoje metod CV vymyšlena spousta algoritmů řešení jednoduchých úloh.

V této části bude nastíněna teorie o základních strukturách a funkcích použitých v aplikační části.

3.1 Obraz

Abychom se mohli bavit o obraze, je třeba definovat, co to je a jak ho popsat.

3.1.1 Definice a digitalizace

Obraz můžeme považovat za signál o více než jednom rozměru. Je chápán například jako obraz na sítnici oka nebo jako obraz sejmutý TV kamerou. Matematickým modelem obrazu je tzv. *obrazová funkce*. *Obrazovou funkcí* se rozumí spojitá funkce obrazu dvou proměnných $f(x,y)$, kde (x,y) označují souřadnice bodu ve 2D prostoru, tedy rovině. Nebo můžeme ještě uvažovat $f(x,y,t)$, kde t je čas. Hodnoty této funkce odpovídají fyzikální veličině, podle zařízení, které má obraz zachytávat. Například se může jednat o jas, teplotu, obecně záření. Chceme-li s obrazem pracovat dále v počítači, je potřeba ho digitalizovat. Digitalizace obrazu má dvě části: *vzorkování* a *kvantování*.

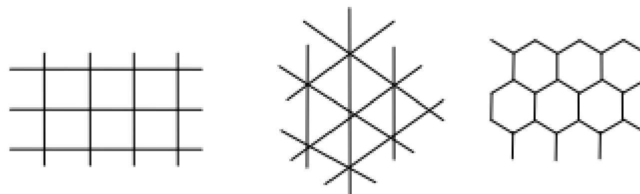
Vzorkování obrazu znamená rozdělení obrazové funkce *f(x,y)* na matici M řádků a N sloupců a *kvantování* každému vzorku přiřadí celočíselnou hodnotu z celého funkčního oboru funkce *f(x,y)*, jež byla rozdělena na K dílů. Poté tedy funkce nabývá hodnot, které reprezentují barvu daného maticového elementu neboli pixelu (z anglického jazyka - picture element). Existuje více reprezentací barvy, jsou to kupříkladu: RGB (Red, Green, Blue), HSV (Hue, Saturation, Value), HLS (Hue, Lightness, Saturation) nebo RGBA (Red, Green, Blue, Alpha). Samozřejmě se můžeme omezit jen na jeden tzv. kanál a mít tedy každý pixel definovaný pouze jednou hodnotou, například hloubkou.

Je nasnadě, že čím více pixelů v matici a čím více kvantizačních úrovní, tím bude obraz přesnější. Digitalizace je jen diskrétní aproximace spojité funkce *f(x,y)*.

Vzorkování

Jak bylo řečeno, vzorkování je rozdělení obrazové funkce na matici pixelů. Každý pixel tedy zaujímá nějakou plochu ve finálním digitálním obraze. Jeho plocha je všude stejně barevná podle informace, tomu se říká bodové vzorkování (point sampling). Existují i další metody vzorkování, kde se využívá například průměru barvy z okolí, tím se dají odstranit jisté vady, které se mohou vyskytnout.

Člověk si pod rozdělením do matice představí zejména normální strukturu matice, totiž obdélníkovou mřížku. V praxi se však můžeme setkat i s hexagonální nebo trojúhelníkovou mřížkou:



Obrázek 4: Typy vzorkování, převzato z [7]

Pixel je dále nedělitelným prvkem obrazu.

Kvantování

Kvantování znamená rozdělení oboru hodnot obrazové funkce na K intervalů, kterým je přidělena jedna hodnota. Tím se tedy obrazová funkce diskretizuje. Tyto hodnoty od sebe mohou být vzdáleny ekvidistantně i neekvidistantně. Neekvidistantní rozdělení může vyřešit problémy s nedostatečnou přesností v okolí nějaké specifické hladiny obrazové funkce, kde například můžeme potřebovat rozlišit byť jen drobné detaily.

Pokud použijeme b bitů k rozlišení hladin, pak intervalů bude:

$$K = 2^b$$

Tedy každému pixelu a každému jeho kanálu se přiřadí hodnota od 0 do K-1.

3.1.2 Vlastnosti obrazu

Vzdálenost pixelů

Jak již bylo uvedeno několikrát obraz je po digitalizaci rozdělen do matice pixelů. Je vhodné zavést nějaký způsob měření vzdálenosti dvou pixelů. Funkci D nazveme vzdáleností, pokud platí:

$$D(p,q) \geq 0,\ \ D(p,q) = 0\ pokud\ p = q$$

$$D(p,q) = D(q,p)$$

$$D(p,r) \leq D(p,q) + D(q,r)$$

Vzdálenost mezi dvěma body v matici na indexech *(i,j)* a *(h,k)* poté můžeme vypočítat například standardně jako euklidovskou vzdálenost:

$$D_E((i,j),(h,k)) = \sqrt{(i-h)^2 + (j-k)^2}$$

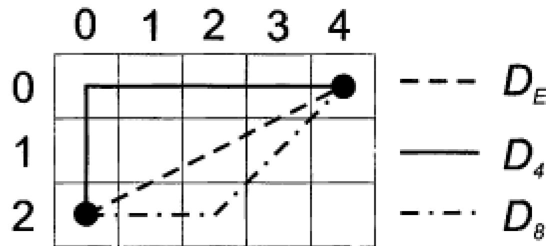
Je nasnadě, že tato metoda vrací neceločíselné výsledky a vypočítat odmocninu se může jevit jako zbytečně obtížné, zvláště pak v real-time aplikacích, kde jde o každou milisekundu. Proto se ke vzdálenosti mezi dvěma body přistupuje i jinak. Minimum elementárních kroků v mřížce matice od startovního ke koncovému bodu ve 4 základních směrech (nahoru, doleva, dolů, doprava) se označuje D_4, tzv. 'Manhattan distance'. Matematický vzorec pro tuto vzdálenost je:

$$D_4((i,j),(h,k)) = |i-h| + |j-k|$$

Pokud dovolíme i pohyb diagonální, dostáváme vzdálenost D_8, tzv. 'chessboard distance'. Matematický vzorec pro tuto vzdálenost je:

$$D_8((i,j),(h,k)) = \max\{|i-h|, |j-k|\}$$

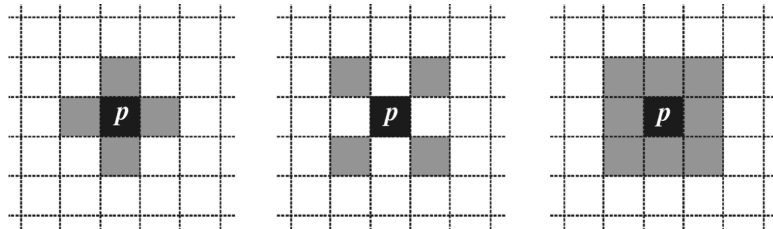
Obrázek 5 popisuje příklad všech tří možností vzdáleností.



Obrázek 5: Vzdálenosti pixelů, převzato z [8]

Sousednost pixelů

Sousedností pixelů rozumíme vztah dvou pixelů, které mezi sebou mají vzdálenost $D_4(p,q) = 1$. Takovým pixelům se pak říká, že tvoří 4-sousednost (4-neighbourhood). Analogicky pak pro 8-sousednost (8-neighbourhood). Na obrázku 6 jsou zobrazeny dvě verze 4-sousednosti a 8-sousednosti:



Obrázek 6: Ukázky sousednosti, převzato z [8]

3.1.3 Kvalita obrazu

Při zachycení, přenosu nebo při digitalizaci dochází k vadám způsobených z různých důvodů. Tyto problémy se souhrnně nazývají šum. Šum může ovlivnit kvalitu obrazu. Dobře známým nešvarem je přílišný jas, který nahrávací zařízení není schopné správně zpracovat. Je jasné, že kvalita obrazu je proměnlivá též v závislosti na úloze, kterou má daná aplikace řešit, zda se má kamera pohybovat či ne, co má sledovat atd.
Informace převzaty z [8].

3.1.4 Implementace v OpenCV

OpenCV je otevřená multiplatformní knihovna pro práci s obrazem. Vznikla v roce 1999 jako projekt firmy Intel. První alfa verze byla představena v roce 2000 na konferenci IEEE Conference on Computer Vision and Pattern Recognition. V roce 2006 pak byla vydána verze 1.0. K datu 25. dubna 2014 se rozšířila na verzi 2.4.9.

OpenCV je napsána v jazyce C++ a její primární interface je taky v C++. Existují již i interface například v Pythonu, Javě a Matlabu.
Obraz je v knihovně OpenCV reprezentován strukturou *cv::Mat* (maticí) s typy:

CV_8UC1	*CV_8UC2*	*CV_8UC3*	*CV_8UC4*
CV_16UC1	*CV_16UC2*	*CV_16UC3*	*CV_16UC4*
CV_32FC1	*CV_32FC2*	*CV_32FC3*	*CV_32FC4*
CV_64FC1	*CV_64FC2*	*CV_64FC3*	*CV_64FC4*

Kde vždy číslo za podtržítkem znamená počet bitů kvantizace, další písmeno typ proměnné a další číslo počet kanálů. Ke všem unsigned typům existují i jejich signed verze.

Typ cv::Mat je tedy do matice uložený digitalizovaný obraz s vlastnostmi:

- Size, která představuje velikost matice (rows - počet řádků, cols - počet sloupců)
- Type - typ matice
- Dims - dimenze matice
- Data - ukazatel na uložená data v matici

Práce s maticemi se podobají práci s maticemi v prostředí Matlab. Fungují zde stejné operátory ke sčítání, odčítání a násobení matic. Další podobností s Matlabem jsou pak maticové konstruktory jako *eye()*, *zeros()* nebo *ones()*. Standardní kontruktor je *Mat(int rows, int cols, int type)*, kde rows je počet řádků, cols počet sloupců a type je typ matice (viz Tabulka). Důležitým konstruktorem je *Mat(int rows, int cols, int type, void* data)*, tento vytvoří matici a naplní ji daty z jednorozměrného pole. Důležitou funkcí v této práci je také vybrání konkrétního obdélníku a uložení ho do matice: *Mat::operator()(const Rect& roi)*, kde Rect je typ proměnné, který tento obdélník představuje.

cv::Rect je obdélník, reprezentovaný svou šířkou, výškou a pozicí svého levého horního rohu v matici, z níž jsme ho vyřízli. Známe-li konturu(viz podkapitola 3.3.1)), již chceme z matice vyříznout a dále zkoumat jen tento prostor, použijeme k tomu funkci *boundingRect()*, která vrátí nejmenší obdélník, který s osou x svírá 0 stupňů.

cv::RotatedRect je otočený obdélník, reprezentovaný svou šířkou, výškou a pozicí středu vůči matici, z níž jsme ho vyřízli. Známe-li konturu (viz podkapitola 3.3.1), již chceme z matice

vyříznout a dále zkoumat jen tento prostor, použijeme k tomu funkci *minAreaRect()*, která vrátí nejmenší obdélník, který s osou x svírá úhel *angle*. Informace převzaty z [9].

3.2 Zpracování obrazu

V této kapitole budou popsány některé základní techniky zpracování obrazu.

3.2.1 Segmentace

Segmentací se rozumí rozdělení digitálního obrazu na oblasti, které nás zajímají. Segmentačních technik je několik a záleží na úloze, kterou z nich je ideální použít. Jsou to například:

- Prahování
- Detekce hran
- Srovnávání se vzorem

Prahování

Prahování (thresholding) je nejstarší, nejpoužívanější a nejjednodušší metodou. Je nenáročná na hardware a proto je nejrychlejší a lze ji provádět v reálném čase. Používá se pro rozlišení objektů v obraze s různou hodnotou obrazové funkce. Rozhodující otázkou v prahování je nastavení prahů. Určit je automaticky není nikdy triviální záležitostí, protože dosti záleží na poloze kamery, ozáření kamery, jejích parametrech, odrazivosti snímaného povrchu, atd. Často se tedy určuje nejprve ručně. Obecná rovnice prahování:

$$dst(x, y) = A \; pokud \; src(x, y) > threshold$$

$$dst(x, y) = B \; pokud \; src(x, y) \leq threshold$$

kde *dst* označuje výsledný obraz, *src* zdrojový obraz a *threshold* práh. Existuje několik druhů prahování:

Binární prahování přiřadí každému pixelu s větší hodnotou než práh hodnotu prahu a všem ostatním nulu, tedy $A = threshold$ a $B = 0$

Binární prahování inverzní přiřadí naopak každému pixelu s větší hodnotou než práh nulu a všem ostatním hodnotu prahu, tedy $A = 0$ a $B = threshold$

Osekávání (truncate) přiřadí každému pixelu s větší hodnotou než práh hodnotu prahu a všechny ostatní ponechá, jak byly, tedy $A = threshold$ a $B = src(x,y)$

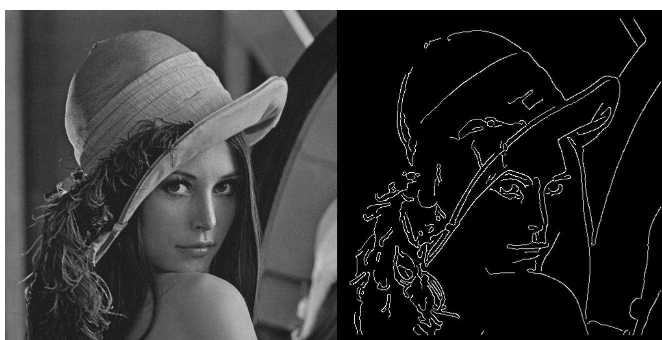
Prahování na nulu ponechá každému pixelu s větší hodnotou než práh hodnotu původní a všem ostatním nastaví hodnotu nula, tedy $A = src(x,y)$ a $B = 0$

Prahování na nulu inverzní přiřadí každému pixelu s větší hodnotou než práh nulu a všechny ostatní ponechá, jak byly, tedy $A = 0$ a $B = src(x,y)$
Prahování s horním a dolním prahem přiřadí pixelům, jejichž hodnota je v intervalu {lb,ub} jedničku a všem ostatním nulu.

Obrázek 7: Ukázka prahování

Detekce hran

Hranou se rozumí určitá nespojitost obrazu například v jasu, barvě nebo textuře. Hrany se tedy vyskytují zejména v oblastech s výraznou změnou obrazu. V místě hrany bude velká derivace obrazové funkce. Aby byl výpočet jednodušší, hrany se detekují jen ve 4 směrech. Aproximace těchto derivací se provádí pomocí konvoluce s vhodným jádrem. Těmto jádrům se také říká operátory. Jsou to například: *Robertsův, Prewittův, Sobelův, Robinsonův a Kirschův.* Nejvíce používanou metodou se stala *Cannyho* metoda hledání hran, která používá *Sobelův* operátor.

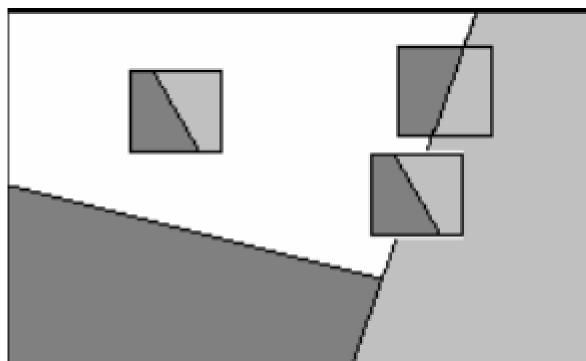


Obrázek 8: Ukázka detekce hran

Srovnávání se vzorem

Takzvaný template matching hledá známé objekty v obraze. Pokud by byl nulový šum, úloha by byla velmi snadná, protože by se v obraze našla přesná kopie vzoru. Tak tomu ale není a vždy se mezi obrazem a vzorem vyskytne chyba. Mírou této chyby může být například rozdíl v obrazových

funkcích. Tato metoda je velmi náročná, protože musí porovnat vzor s každým bodem v obraze. Zvlášť pokud je obraz například zvětšený, pootočený nebo geometricky zkreslený.



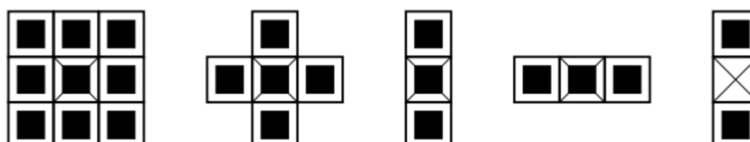
Obrázek 9: Ukázka srovnávání

Informace převzaty z [9].

3.2.2 Matematická morfologie

Po segmentaci se mohou ke slovu dostat morfologické operace, které mají za úkol získaný obraz upravit. Tyto operace pomáhají k odstranění šumu, zjednodušení tvaru, rozdělení tvaru na více jednotlivých tvarů, zvýraznění struktury (kostra), zaplnění děr v nalezených objektech atd. Tyto metody byly vyvinuty především pro binární obrazy.

Metody matematické morfologie pracují s binárním obrazem a tzv. strukturním elementem, kde tyto prvky bere jako bodové množiny. Bodová množina celého binárního obrazu se označuje X a jsou v ní všechny body objektů, jejím doplňkem je množina X^c, kde jsou všechny body pozadí a děr v objektech. Strukturní element se označuje B a popisuje, co se má s binárním obrazem stát. Bod obrazu, který se shoduje s počátkem strukturního elementu, se nazývá *okamžitý bod.* Transformace pak probíhá tak, že strukturním elementem projíždíme celý binární obraz a výsledek relace zapisujeme do okamžitého bodu.



Obrázek 10: Ukázka strukturních elementů, převzato z [10]

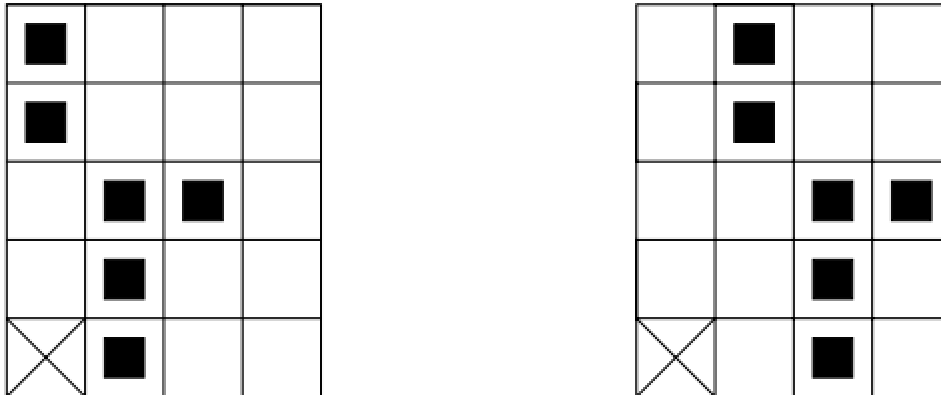
Základními morfologickými operacemi jsou:

- Translace
- Dilatace
- Eroze
- Uzavření
- Otevření

Translace

Translace X_h bodové množiny X znamená, jak název napovídá, posunutí jejích bodů o nějaký vektor h.

$$X_h = \{p \in \mathbb{E}^{\nvDash},\ p = x + h\}$$



Obrázek 11: Ukázka translace, převzato z [10]

Dilatace

Dilatace je Minkowského součet dvou bodových množin:

$$X \oplus B = \bigcup_{b \in B} X_b$$

ekvivalentní zapis:

$$X \oplus B = \{p \in \mathbb{E}^2 : p = x + b,\ x \in X,\ b \in B\}$$

Nejčastěji používaným strukturním elementem je plná matice 3x3 s počátkem veprostřed, tedy celá 8-sousednost (viz podkapitola 3.1.2). Pokud je tento strukturní element použit, objekty se rozšíří o jednu úroveň do prostoru a díry o tloušťce 2 v nich se zaplní, totéž platí o tzv. zálivech, tj. jedna dílčí kompenzace poruch.

Vlastnosti dilatace:

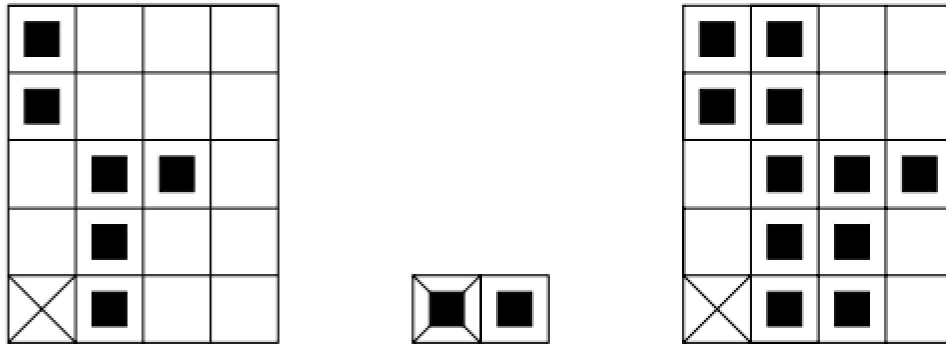
- Komutativnost: $X \oplus B = B \oplus X$
- Asociativnost: $X \oplus (B \oplus D) = (X \oplus B) \oplus D$
- Ivariance vůči posunu: $X_h \oplus B = (X \oplus B)_h$

Příklad dilatace:

$X = \{(1,0),(1,1),(1,2),(2,2),(0,3),(0,4)\}$

$B = \{(0,0),(1,0)\}$

$X \oplus B = \{(1,0),(1,1),(1,2),(2,2),(0,3),(0,4),(2,0),(2,1),(2,2),(3,2),(1,3),(1,4)\}$



Obrázek 12: Ukázka dilatace, převzato z [10]

Eroze

Eroze je duální metodou k dilataci. Není však inverzní. Je to Minkowského rozdíl:

$$X \ominus B = \bigcap_{b \in B} X_{-b}$$

ekvivalentní zápis:

$$X \ominus B = \{p \in \mathbb{E}^2 : p = x + b \in X \ \forall b \in B\}$$

Vlastnosti eroze:

- Antiextenzivnost: Je-li $(0,0) \in B$, potom $X \ominus B \subseteq X$
- Invariance vůči posunu: $X_h \ominus B = (X \ominus B)_h$, $X \ominus B_h = (X \ominus B)_{-h}$
- Nekomutativnost: $X \ominus B \neq B \ominus X$

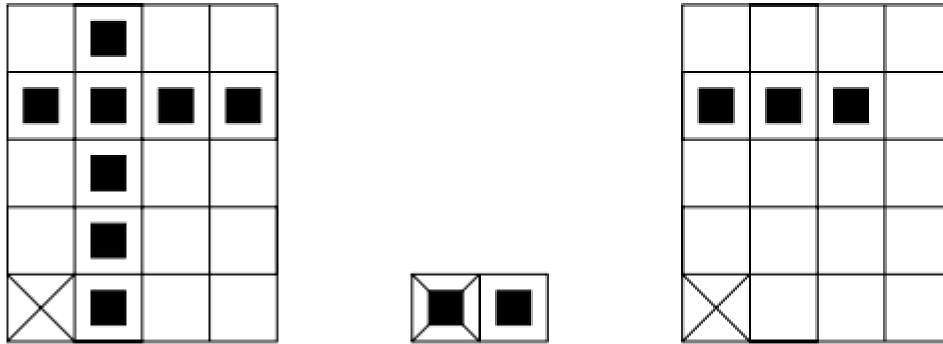
Nejčastěji používaným strukturním elementem je opět plná 8-sousednost. Pokud je tento element použit, objekty se zmenší o jednu úroveň. Také zmizí samotné body a malé chybičky. Jestliže odečteme od původního obrazu jeho erozi, získáme obrysy objektu.

Příklad eroze:

$X = \{(1,0),(1,1),(1,2),(0,3),(1,3),(2,3),(3,3),(1,4)\}$

$B = \{(0,0),(1,0)\}$

$X \ominus B = \{(0,3),(1,3),(2,3)\}$



Obrázek 13: Ukázka eroze, převzato z [10]

Otevření a uzavření

Otevření je eroze následovaná dilatací:

$$X \circ B = (X \ominus B) \oplus B$$

Uzavření je dilatace následovaná erozí:

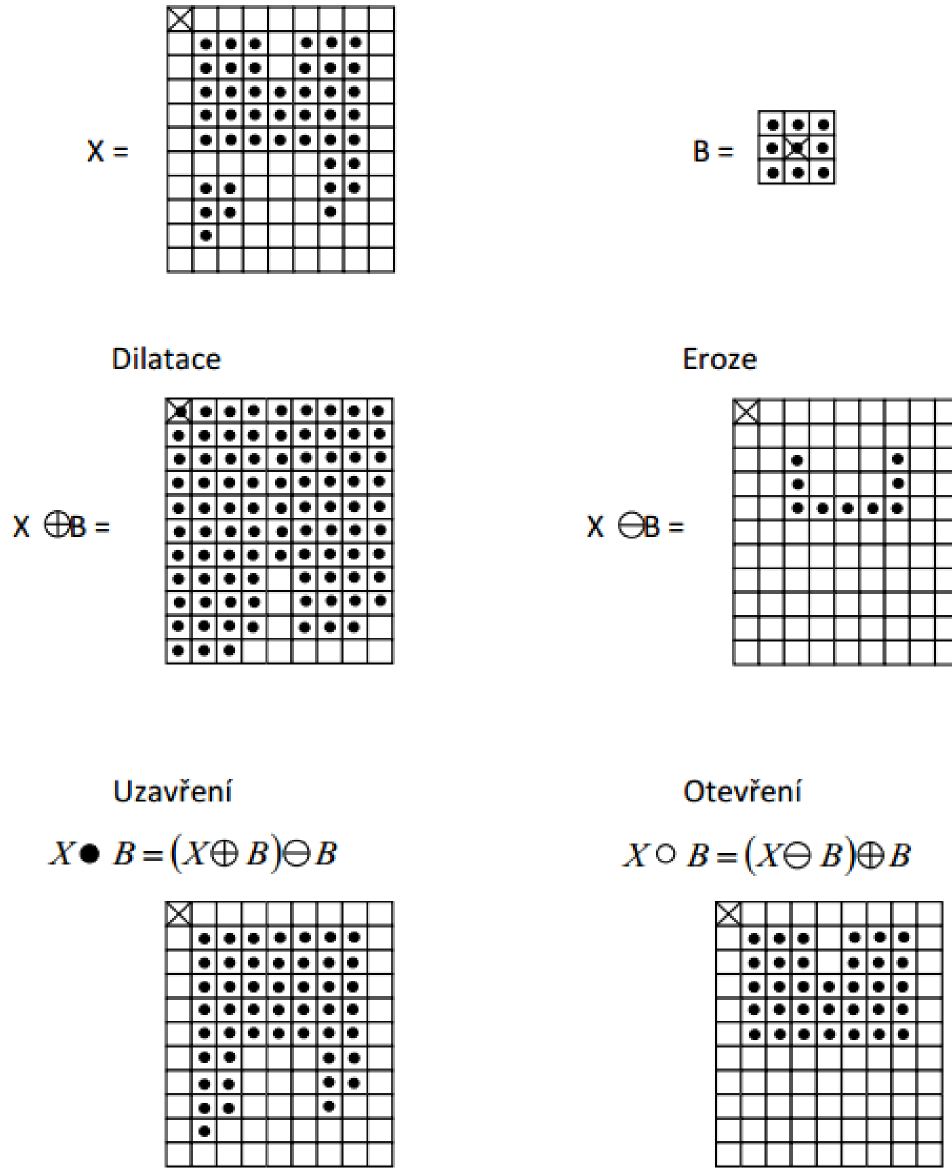
$$X \bullet B = (X \oplus B) \ominus B$$

Výsledný obraz je ten samý ořezaný o detaily. Otevření například může oddělit dva objekty, které jsou blízko u sebe. Uzavření dva blízké objekty spojí a zaplní malé díry a zálivy.

Vlastnosti uzavření a otevření:

- Otevření je antiextenzivní
- Uzavření je extenzivní

- Idempotence otevření: opakované použití této operace nezmění výsledek: $(X \circ B) \circ B = X \circ B$
- Idempotence uzavření: opakované použití této operace nezmění výsledek:$(X \bullet B) \bullet B = X \bullet B$



Obrázek 14: Ukázka otevření a uzavření, převzato z [7]

Informace převzaty z [10].

3.2.3 Implementace v OpenCV

V knihovně OpenCV jsou tyto funkce implementovány v modulu *imgproc*:

Prahování

Prahování se provádí funkcí *threshold()* s parametry: *src* - vstupní obraz, který bude zpracován, *dst* - výstupní obraz, do něhož bude uložen výsledek, *thresh* - práh, *maxval* - hodnota, jež se přiřadí pixelům v případě binárního nebo inverzně binárního prahování, *type* - prahovací typ
Další prahovací funkcí je *inRange()* s parametry *src* - vstupní obraz, který bude zpracován, *lowerb* - spodní hranice intervalu, *upperb* - horní hranice intervalu, *dst* - výstupní obraz, do něhož bude uložen výsledek.

Detekce hran

Pro detekci hran je implementována Cannyho metoda *Canny()* s parametry: *image* - vstupní obraz, který bude zpracován, *edges* - výstupní obraz, do něhož bude uložen výsledek, *threshold1* spodní práh pro hysterezní proceduru, *threshold2* horní práh pro hysterezní proceduru, *apertureSize* číslo pro použitý Sobelův operátor, *L2gradient* - volí přenost počítaného gradientu.

Srovnávání se vzorem

Srovávání se vzorem existuje například metoda *matchTemplate()* s parametry: *image* - vstupní obraz, který bude zpracován, *templ* - hledaný vzor, *result* výsledný obraz, *method* - súecifikace metody porovnávání obrazu se vzorem.

Dilatace

Pro dilataci obrazu se volá metoda *dilate()* s parametry *src* - vstupní obraz, který bude zpracován, *dst* - výstupní obraz stejného typu jako src, do kterého bude uložen výsledek, *element* - strukturní element, *anchor* - bod "středu"strukturního elementu, *iterations* - počet kolikrát bude dilatace provedena.

Eroze

Pro erozi obrazu se volá metoda *erode()* s parametry: *src* - vstupní obraz, který bude zpracován, *dst* - výstupní obraz stejného typu jako src, do kterého bude uložen výsledek, *element* - strukturní element, *anchor* - bod "středu"strukturního elementu, *iterations* - počet kolikrát bude eroze provedena.

Otevření a uzavření

Pro otevření existuje metoda: *open()*, která zavolá *dilate(erode())*, tedy nejdříve se zavolá *erode()* a na její výsledek je použita *dilate()*. Pro uzavření existuje metoda: *close()*, která zavolá *erode(dilate())*, tedy nejdříve se zavolá *dilate()* a na její výsledek je použita *erode()*.
Informace převzaty z [9].

3.3 Použité metody

Obsah této kapitoly pojednává o algoritmech funkcí, jež byly použity.

3.3.1 Algoritmus Suzuki85

Tento algoritmus hledá obrysy neboli *kontury* kolem oblastí v binárních obrazech. Jeho hlavní myšlenkou je sledování hranice, což je jedna z fundamentálních technik ve zpracování binárních obrazů. Odvozuje sekvenci souřadnic z hranic mezi spojenými bílými pixely a spojenými černými pixely (pozadí a díry).

Je zde potřeba zavést několik pojmů:

Definice 1 Hraniční bod (Border Point): Mějme pixel o hodnotě 1 na pozici (i,j), který obsahuje pixel s hodnotou nula na pozici (p,q) ve své 4-/8-sousednosti. Takový pixel nazveme hraničním bodem.

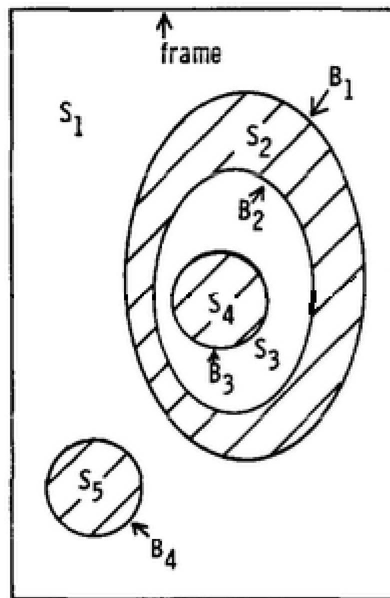
Definice 2 Obklopení spojených oblastí: Mějme dvě spojené oblasti S_1 a S_2 v binárním obraze. Pokud existuje pixel z S_2 pro nějakou cestu (nahoru, doleva, dolů, doprava) od pixelu z S_1 do pixelu na rámečku, říkáme, že S_2 je obklopen S_1. Pokud je S_2 obklopen S_1 a existuje mezi nimi hraniční bod, pak říkáme, že S_2 je obklopen S_1 přímo.

Definice 3 Vnější hranice a hranice děr: vnější hranice je definována jako množina hraničních bodů mezi libovolnou bílou oblastí a černou oblastí, jež ji obklopuje přímo. Podobně mluvíme i o množině hraničních bodů mezi dírou a bílou oblastí, jež ji obklopuje přímo, a to jako o hranici díry. Užíváme však jen termín hranice pro oba případy.

Vlastnost 1 Pro libovolnou bílou oblast binárního obrazu je jeho vnější hranice pouze jedna. Pro libovolnou díru je její hranice díry s oblastí, jež ji obklopuje přímo, také pouze jedna.

Definice 4 Rodičovská hranice: je další hranicí v řadě mezi aktuální hranicí a rámečkem.

Definice 5 Obklopení hranice hranicí: Hranice, jež má rodičovskou hranici, je svou rodičovskou hranicí obklopena.



Obrázek 15: Ukázka oblastí a děr, převzato z [11]

Algoritmus

1) Algoritmus scanuje vstupní binární obraz $F = \{f_{ij}\}$ po řádcích. Když dojede k bodu *(i,j)*, takovému, že se v něm změnila hodnota buď z nuly na jedničku (tj. nalezení vnější hranice) nebo z jedničky na nulu (tj. nalezení hranice díry), prohledávání se zastaví. Tento hraniční bod se považuje za počátek hranice. Této hranici se přiřadí unikátní identifikační číslo NBD. NBD je na začátku procesu nastaveno na 1. Pokud se žádný takový bod nenašel, algoritmus přeskočí ke kroku 7.

2) Dále je třeba rozhodnout úroveň této nově nalezené hranice a její rodičovskou hranici. Během minulého prohledávání jsme taktéž sledovali číslo LNBD, tedy identifikační číslo posledně nalezené hranice. Tato hranice by pak měla být buď rodičovskou hranicí nově nalezené, nebo ta, s kterou nově nalezená hranice rodičovskou hranici sdílí.

3) Sledování hranice probíhá následovně. Prohledává se 8-sousednost pixelu *(i,j)* se začátkem v (i_2, j_2) = *(i,j-1)*, pokud se jedná o vnější hranici, nebo v (i_2, j_2) = *(i,j+1)*, pokud se jedná o hranici díry, po směru hodinových ručiček, aby se nalezl nenulový pixel. První nenulový nalezený pixel označme (i_1, j_1). Pokud se žádný nenalezl, do NBD uložíme hodnotu f_{ij} a algoritmus přeskočí na 7. Do bodu (i_2, j_2) uložíme bod (i_1, j_1) a do nového bodu (i_3, j_3) uložíme bod *(i,j)*, jehož 8-sousednost jsme prohledávali.

4) Začínaje z dalšího elementu pixelu (i_2, j_2) proti směru hodinových ručiček, se prohledává proti směru hodinových ručiček 8-sousednost aktuálního bodu (i_3, j_3), aby se na nalezl nenulový pixel. První nenulový nalezený pixel označme (i_4, j_4).

5) Změníme hodnotu $f_{i_3j_3}$ bodu (i_3, j_3) podle následujících pravidel:

a) Pokud pixel $(i_3, j_3 + 1)$ je nulový pixel zkoumaný v kroku 4, pak $f_{i_3j_3}$ = -NBD

b) Pokud pixel $(i_3, j_3 + 1)$ není nulový pixel zkoumaný v kroku 4 a $f_{i_3j_3} = 1$, pak $f_{i_3j_3}$ = NBD

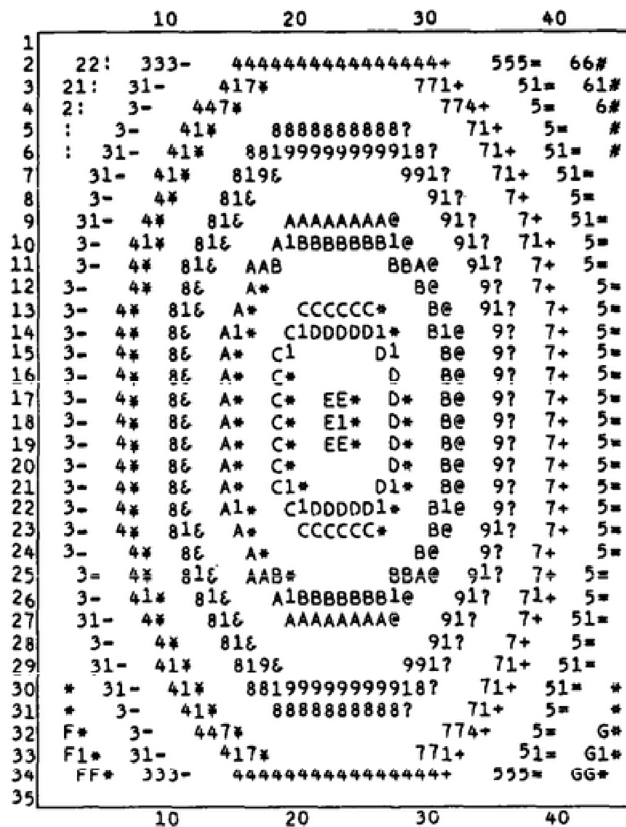
c) Jinak neměníme hodnotu $f_{i_3j_3}$

6) Jestliže platí $(i_4, j_4$ = *(i,j)* a $(i_3, j_3) = (i_1, j_1)$, pokračuje algoritmus krokem 7. Jinak se do bodu (i_2, j_2) uloží bod (i_3, j_3) a do bodu (i_3, j_3) uloží bod (i_4, j_4) a algoritmus se vrátí ke kroku 4.

7) Jestliže platí, že $f_{ij} \neq 1$, pak se do LNBD uloží f_{ij} a scanner se posune na bod *(i,j+1)* a jde se na krok 2. Algoritmus končí, když scanner dojde k pravému spodnímu rohu obrazu. Tento algoritmus může být upraven tak, aby hledal pouze nejvnějšnější kontury.

Implementace v OpenCV

V knihovně OpenCV se tento algoritmus uplatňuje v metodě *findcontours()*, jejímiž parametry jsou *binární obraz*, v kterém se mají kontury nalézt, *pole*, do kterého se budou ukládat nalezené kontury, *hierarchie* jednotlivých úrovní kontur (NBD), *mód*, který určuje které specifické kontury se mají vyhledat, metoda, jež určuje aproximační metodu hledání kontur.



Obrázek 16: Ukázka funkce algoritmu suzuki1985, převzato z [11]

Informace převzaty z [11].

3.3.2 Particle Filter

Particle Filtery neboli Sekvenční Monte Carlo metody jsou algoritmy, které v každém kroku odhadují hustoty pravděpodobnosti rozdělení stavového prostoru. K tomu využívají množinu tzv. particlů, které reprezentují stav zkoumaného objektu. Těm je přiřazována váha reprezentující pravděpodobnost jako kdyby byla generována z hledané hustoty. Particly jsou generovány ze znalosti odhadnuté hustoty z minulého kroku. Tyto metody nepotřebují ke své funkci žádnou znalost o dynamice stavového prostoru zkoumaného objektu. Nejsou však vhodné pro prostory o vysoké dimenzi.

První zmínky o Particle Filteru se datují do 50. let minulého století v Poor Man's Monte Carlo od Hammersleyho [12], který naznačoval Sekvenční Monte Carlo metodu, jaká se používá dnes. Ale teprve v roce 1993 byla tato metoda skutečně implementována. Její autoři pojmenovali svůj algoritmus 'the bootstrap filter', což znamená soběstačný proces, který nepotřebuje žádná vstupní data.

Cílem filteru je odhadnout hustotu pravděpodobnosti stavových proměnných, pokud mu jsou k dispozici stavové proměnné získané na základě pozorování. Particle Filter byl vyvinut jako Skrytý Markovský Model, kde systém obsahuje skryté a pozorovatelné stavové proměnné. Pozorovatelné proměnné jsou v nějakém určitém vztahu k těm skrytým. Úkolem Particle Filteru je odhadnout hodnoty skrytých stavových proměnných x, když má k dispozici pozorované hodnoty y. Přesněji řečeno sekvenci skrytých proměnných x_k pro $k = 0, 1, 2\ldots$, pomocí pozorovaných dat y_k pro $k = 0, 1, 2\ldots$ Všechny Bayesovské odhady x_k vyplývají z aposteriorního rozdělení $p(x_k|y_0, y_1, ..., y_k)$

Algoritmus

Vygenerování jednoho vzorku x v k-tém kroku z rozdělení $p_{x_k|y_{1:k}}(x|y_{1:k})$:

1) Nastav $n = 0$, toto číslo bude počítat zatím vygenerovaný počet particlů.

2) S rovnoměrným rozdělením vyber index L z množiny $\{1,2,\ldots P\}$, kde P označuje maximální počet particlů.

3) Vygeneruj test $\hat{x}$ z rozdělení $p_{x_k|x_{k-1}}(x|x^{(L)}_{k-1|k-1})$

4) Vygeneruj pravděpodobnost $\hat{y}$ pomocí $\hat{x}$ z $p_{y|x}(y_k|\hat{x})$, kde y_k je změřená hodnota.

5) S rovnoměrným rozdělením vygeneruj u z intervalu $[0, m_k]$, kde $m_k = \sup_x p_{y|x}(y_k, x)$

6) Srovnej u a $p(\hat{y})$

6a) Pokud je u větší, opakuj krok 2

6b) Pokud je u menší, ulož $\hat{x}$ jako $x^{(p)}_{k|k}$ a inkrementuj n

7) Pokud se n rovná P, ukonči

Cílem je vygenerovat P particlů v k-tém kroku pouze pomocí particlů z (k-1). kroku. $x^{(L)}_k$ je L. particle vygenerovaný v k-tém kroku. V kroku 3 algoritmu se vygeneruje potenciální x_k podle ná-

hodně vybraného particlu $x_{k-1}^{(L)}$ a tento je přijat nebo odmítnut v kroku 6.

Mezi každým krokem (k-1) a k je proveden tzv. *resampling*, ten smaže particly, kterým se přiřadila příliš malá váha a jsou tedy "zbytečné". Tyto smazané particly se pak zašuměné vygenerují kolem particlů s váhami většími a celý soubor particlů se znormalizuje, aby suma všech vah byla jedna (integrál hustoty je roven jedné). Informace převzaty z [13].

4 3D grafika

3D grafika je speciální oblast počítačové grafiky, která pracuje s trojrozměrnými objekty. Vykreslování 3D grafiky na 2D plochu se nazývá renderování. 3D grafika je hojně využívanou záležitostí zejména například v oblasti filmu, počítačových her, počítačovém modelování, atd. Renderování kromě samotného výpočtu změny souřadnic z 3D světa na obrazovou rovinu může řešit i další věci, jako například efekty:

- Stínování - změna jasu povrchu vykreslovaných objektů v závislosti na osvícení
- Texturování - pokreslení povrchu vykreslovaných objektů
- Mlha - tlumení světla
- Stíny - zakrytí zdroje světla
- Odraz světla - zrcadla
- Průhlednost/Průsvitnost - šíření světla skrze objekty

4.1 OpenGL

4.1.1 Historie

Mezi nejznámější grafické rozhraní patří OpenGL.

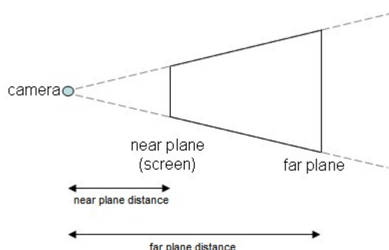
OpenGL (Open source Graphics Library) je multiplatformní, jazykově nezávislé API pro vykreslování 2D nebo 3D grafiky. Používá se při tvorbě počítačových her, CAD programů nebo vykreslování virtuální reality. API interaguje s grafickým procesorem (GPU), aby se dosáhlo hardwarově akcelerovaného vykreslování, tj. rychleji než na CPU. OpenGL standard byl vyvinut společnosté Silicon Graphics Inc. roku 1991 a zveřejněn v lednu roku 1992. Je spravován neziskovým konsorciem Khronos Group.

V 80. letech minulého století byla firma SGI jedničkou v 3D grafice. Jejich IrisGL byla považována za nejmodernější a stala se průmyslovým standardem. Konkurenční firmy Sun, HP, IBM a další se proto snažily přivést na trh vlastní 3D hardwary. To přinutilo SGI změnit IrisGL API na OpenGL, aby trh také ovlivnili. V roce 1992 vytvořila SGI konsorcium, skupiny společností, které by OpenGL udržovali a rozšiřovali v budoucnu. V roce 1994 firma uvažovala o zavedení rozšíření OpenGL++, ale nikdy tuto myšlenku nezrealizovala. V roce 1995 zveřejnil Microsoft svůj Direct3D, který se pak stal hlavním konkurentem OpenGL. Dva roky na to zahájil Microsoft s SGI projekt Fahrenheit, jehož cílem bylo spojení OpenGL a Direct3D v jedno interface a vzít z obou to nejlepší, později se přidala i firma HP. Tento projekt však byl zrušen v roce 1999, kvůli finančnímu omezení firmy SGI a strategickým důvodům Microsoftu. V roce 2006 se odhlasovalo předání správy OpenGL skupině Khronos Group.

4.1.2 Funkce OpenGL

Na začátku každého programu používajícího OpenGL se musí OpenGL incializovat. Jsou dvě fáze incializace OpenGL: vytvoření OpenGL contextu a nahrání všech potřebných funkcí OpenGL.

Vytvoření contextu OpenGL znamená vytvoření okna a do něj vložení 3D prostoru, kam se bude později vykreslovat. Každé okno v MS Windows má tzv. *Device Context*, tento objekt v sobě nese informaci *Pixel Format*. *Pixel Format* popisuje vlastnosti framebufferu, který chceme aby náš OpenGL context měl. *Pixel Format* definuje vykreslování do okna, podporu OpenGL, podporu double bufferingu, barevný formát pixelu (například RGBA), počet bitů rozlišení barev, počet bitů Z-bufferu, Alpha bufferu, Stencil bufferu, různé další věci, více v [14]. Dalšími kroky je nastavení barvy pozadí, perspektivy (funkce *gluPerspective()*), near plane, far plane (hloubkové omezení zorného pole, Obrázek 17), defaultní hloubky depth bufferu, povolení funkce depth bufferu a typ hloubkového testu.

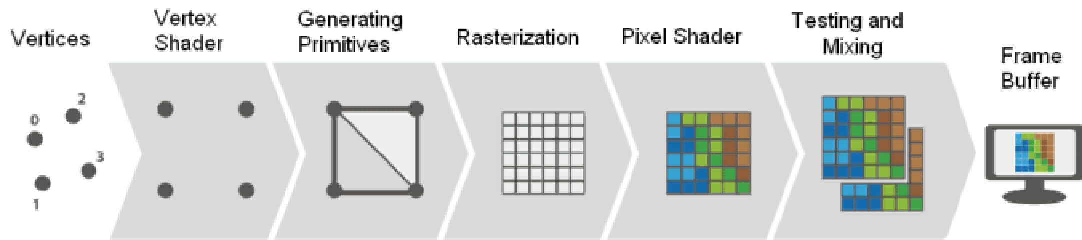


Obrázek 17: Příklad znázorňující near plane a far plane

Dalším krokem je zavolání funkce *wglCreateContext()*. Po vytvoření OpenGL contextu se ještě tento musí "přihlásit o slovo", aby byl aktuálně používaným contextem. O to se postará funkce *wglMakeCurrent()*.

Jelikož nestačí na začátku uvést #include a header pro OpenGL, musíme funkce nahrát do programu buď přes nahrávací knihovnu Glew, nebo zavolat funkci *wglGetProcAddress()*.

OpenGL je knihovna pro renderování. Nepamatuje si však nic o tom, co vykresluje. Objekty, které jsou zadávány v podobě bodů, tzv. *vertexů*, vnímá pouze jako síť trojúhelníků a stav v jakém tyto trojúhelníky má vykreslit. Kvůli tomu OpenGL funguje tak, že vykresluje pořád dokola všechno znovu a znovu. Pokud je tedy potřeba i malá změna scény, OpenGL přijme znova všechny informace o scéně a vykreslí ji celou znovu, přestože změna byla nepatrná. Stejné je to i s velkými animacemi, každý obraz je vykreslen celý zvlášť. Na celou věc vykreslování se pohlíží jako na pipeline (potrubí). Je to opakující se sekvence kroků, které OpenGL dělá, když vykresluje objekty:



Obrázek 18: OpenGL pipeline, převzato z [14]

1) Příprava pole vertexů

2) Zpracování vertexů:

a) Každý vertex projde tzv. Vertex Shaderem, programem, který zpracovává vertex po vertexu celou OpenGL scénu. Mezi operace, které provádí, patří zejména geometrická transformace vrcholu (přenásobení pohledovou a world maticí). Přičemž je zachován počet vertexů, není možné je přidávat ani odebírat.

b) Teselace - transformace sítě vertexů na tzv. primitiva jako jsou přímky, trojúhelníky nebo čtverce,

c) Geometry Shader je nepovinná fáze. Řídí zpracování vzniklých primitiv. Může též narozdíl od Vertex Shaderu vytvořit nová primitiva z těch, které má již k dispozici.

3) Post-processing - vertexy/primitiva projdou dalšími operacemi, které je připraví na rasterizaci.

4) Rasterizace - proces, který zpracuje primitiva. Jeho výsledkem je sekvence fragmentů. Fragment je stav, který je poté použit pro výpočet finálních dat pixelů. Fragment obsahuje informace o své pozici na obrazovce a další, které byly výstupem Geometry nebo Vertex Shaderů.

5) Fragment/Pixel Shader - pracuje už s rasterizovaným 2D obrazem. Dochází zde k obarvení pixelů nebo aplikace textur.q

6) Testování

a) Scissor Test - tato operace "vystřihne"pixely, které se nevešly do nějaké zvolené obdélníkové oblasti obrazovky.

b) Stencil Test - vyřazuje některé pixely za podmínky, která vychází z porovnání hodnoty ve Stencil bufferu a nějaké referenční hodnoty.

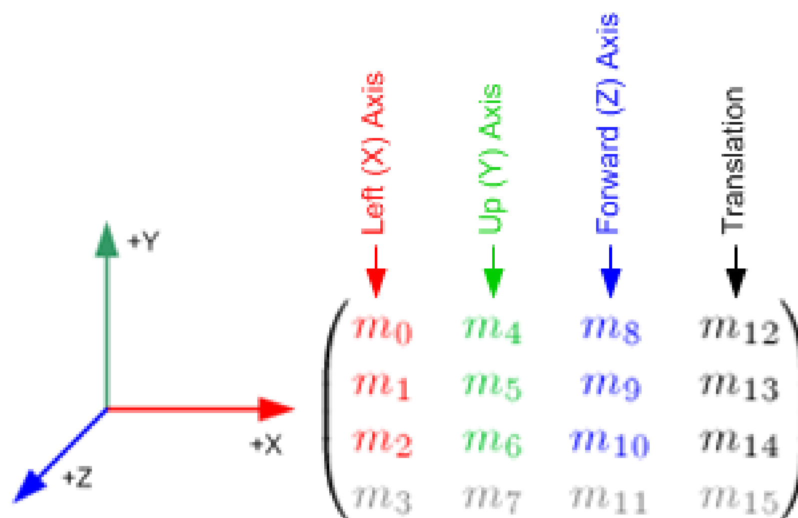
c) Depth Test - vyřazuje některé pixely za podmínky, která vychází z porovnání hloubek pixelů na stejném mistě. Pokud je hloubková kontrola nastavena na LEQUAL, do Depth

bufferu se zapíší jen ty nejbližší pixely.

d) Blending - Smíchá barevné výstupy pixelů z Pixel Shaderu s barvami v color bufferech. Využívá se například k průhlednosti objektů.

e) Zapsání do Framebufferu - všechna data o pixelech, o kterých se rozhodlo, že se nechají vykreslit, se zapíší do Framebufferu. Defaultně jsou 2 Framebuffery, ty se mezi sebou po každém vykreslení střídají.

Co se vlastně děje s polem vertexů během jejich průběhu Vertex Shaderem? Vertex Shader, jak bylo řečeno, vypočítává geometrii vertexů vzhledem k obrazovce, tj. výšce, šířce a hloubce OpenGL "světa". Vertex Shader tedy řeší seskupení všech vertexů, jejich pozici vzhledem k počátku tohoto světa a rotaci objektů, které tyto vertexy tvoří. To se v OpenGL řeší vše přenásobováním transformační maticí.



Obrázek 19: Matice GL_MODELVIEW, převzato z [14]

Čísla (m_{12}, m_{13}, m_{14}) určují posunutí po osách x, y, z, nastavitelné pomocí funkce *glTranslatef()* Číslo m_{15} je tzv. homogenní souřadnice, která je důležitá pro projektivní transformaci. Sloupce s prvky (m_0, m_1, m_2), (m_4, m_5, m_6) a (m_7, m_8, m_9) určují rotaci a scaling, nastavitelné pomocí funkc*glRotatef()* a *glScalef()* Defaultně jsou tyto hodnoty nastaveny na reprezentaci ortogonálnáho systému dimenze 3: vektor $X = (1, 0, 0)$, vektor Y $= (0, 1, 0)$ a vektor $Z = (0, 0, 1)$

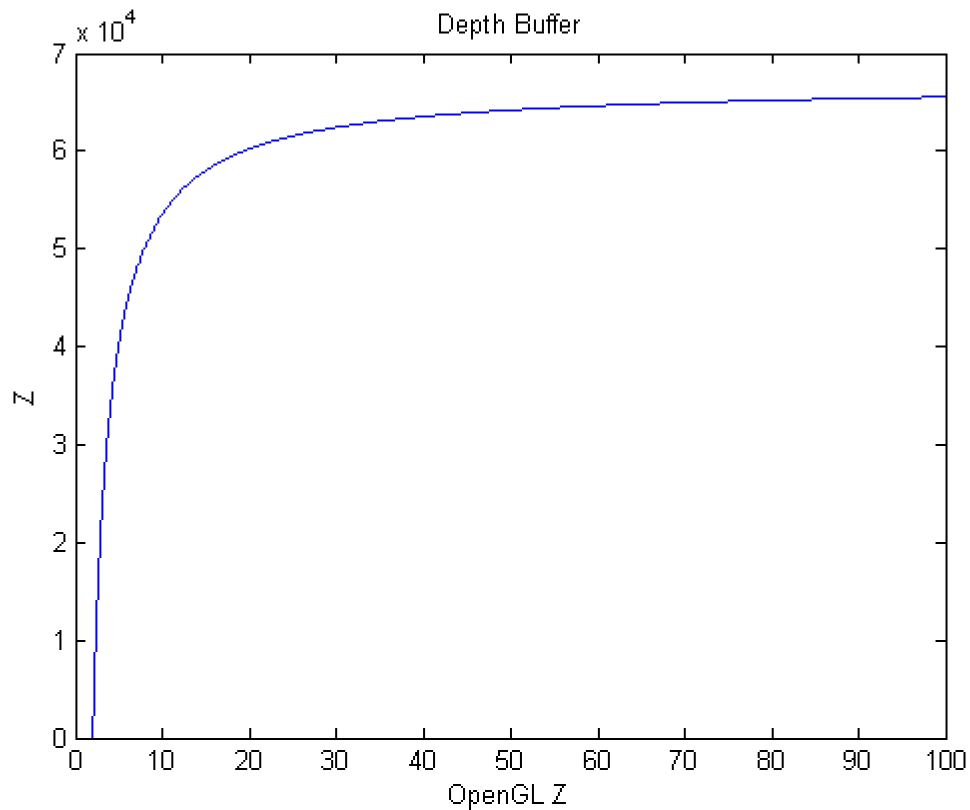
Velmi důležitou věcí je si uvědomit, jak depth buffer generuje hloubku. Je zřejmé, že nějaká konstantní změna hloubky se nám bude jevit metr od nás zjevnější než kilometr od nás. Toto depth

buffer "simuluje"podle následující funkce:

$$Z = S \cdot \frac{FP \cdot NP}{z \cdot (FP - NP) - FP \cdot S}$$

kde Z značí z-ovou souřadnici ve světě OpenGL, S je rozlišení depth bufferu, NP značí near plane, FP značí far plane a z je hloubka před transformací

Graf znázorňující tuto závislost:

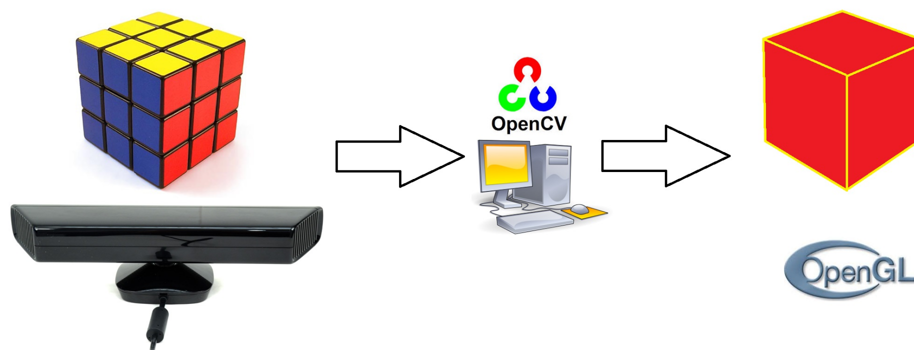


Obrázek 20: Depth buffer

Po vykreslení OpenGL scény se tato dá přečíst a uložit do jednorozměrného pole. Toto dovoluje funkce *glReadPixels()*, která může přečíst libovolný obdélníkový výřez z OpenGL okna z libovolného bufferu (tedy barvu RGBA (nebo samostatně po kanálech), hloubku a svítivost).
S těmito daty pak můžeme dále pracovat dle libosti.
Informace převzaty z [14].

5 Algoritmus programu

Úlohou práce bylo sledování objektu pomocí snímače Kinect, rekonstrukce tohoto objektu ve 3D a vykreslení do barevného obrazu z Kinectu. Pro programování této úlohy bylo použito prostředí Microsoft Visual Studio 2010 s použitím externích knihoven OpenCV, OpenGL a OpenNI. Jako programovací jazyk byl zvolen jazyk C++, viz Příloha C. Pro drobnější výpočty byly použity programy Matlab a MS Office Excel. Jako objekt byla zvolena Rubikova kostka. Sledování probíhá tak, že Kinect sejme reálnou kostku v nějakém stavu $S_R = \{x, y, z, rotx, roty, rotz\}$ a počítač ji převede do matice s počátkem v levém horním rohu. Úkolem programu je pak nalezení tohoto stavu a jeho transformace do OpenGL světa s počátkem uprostřed obrazové plochy. To znamená nalezení vztahů, které danou transformaci provedou a vrátí přibližný stav kostky $S_{Ap} = \{X, Y, Z, rotX, rotY, rotZ\}$. Tento stav se pak doladí particle filterem na $S_{GL} = \{X_{GL}, Y_{GL}, Z_{GL}, rotX_{GL}, rotY_{GL}, rotZ_{GL}\}$ a kostka je vykreslena v OpenGL světě a do barevného obrazu, který je výsledkem.



Obrázek 21: Schéma programu

Prvním krokem celého programu je vytvoření okna a inicializace OpenGL contextu. V tomto okně se dále nachází tlačítko Start a Textové pole. Po stisku tlačítka se inicializuje Kinect a program dále pracuje ve třech fázích. První fáze je inicializační. V této fázi se musí obraz prahovat tak, aby kontura kostky byla největší ze všech kontur právě získaných a nalezených objektů v jednom konkrétním hloubkovém pásmu. Po stisku tlačítka F4 se toto pásmo začne měnit spolu s tím, jak s kostkou uživatel pohybuje ve směru osy Z. Po stisku tlačítka F7 se zapne Particle Filter.

Celkový postup programu tedy je: *inicializace OpenGL contentu v okně aplikace Win32* $\rightarrow$ *inicializace Kinectu* $\rightarrow$ *1. fáze* $\rightarrow$ *2. fáze* $\rightarrow$ *3. fáze*

5.1 Inicializace OpenGL

Kostrou celého programu je hlavní okno, vytvořené Win32 aplikací. Do tohoto okna je nutno přivést OpenGL context. Pro napsání této části byly využity NeHe tutoriály [15]. Důležitými

částmi jsou nastavení šířky a výšky okna (640×480) near plane (dále NP, nastaveno na 2) a far plane (dále FP, nastaveno na 100) a perspektivy. FOV (Field of view), tedy "zorné pole", nastaveno podle parametrů kinectu na 43 stupňů vertikální úhel a 57 stupňů horizontální úhel. Dále černé pozadí, defaultní hloubku depth bufferu po jeho vymazání na 1, povolení hloubkového testování s typem testu LEQUAL, tedy upřednostňované jsou pixely blíže počátku FOV. Pixel Format byl nastaven na RGBA (kvůli možnosti vykreslení průhledného objektu) s 16 bitovým zobrazením barev, Stencil buffer vypnut a 16 bitový Depth Buffer (tj. celkově 2^{16} hloubkových hodnot).

5.2 Inicializace Kinectu

Dalším krokem je inicializace Kinectu. To znamená inicializace knihovny OpenNI a využití jejích typů k vyhledání *openni::device* Kinectu. Z tohoto *device* bereme dva *openni::VideoStreamy*, do kterých proudí hloubková a barevná data z Kinectu. Z *VideoStreamů* tato data jsou konvertována do OpenCV struktury *cv::Mat*, hloubková do typu *CV_16UC1* a barevná do *CV_8UC3* (viz podkapitola 3.1.4)

5.3 Prahování

Ve všech fázích se nejprve z Kinectu získá hloubková mapa. Obrázek 22 znázorňuje tuto matici převedenou do formátu *CV_8UC1*

Obrázek 22: Osmibitová matice hloubkové mapy

Šestnáctibitová hloubková matice je funkcí *inRange()* odprahována, jak se popisuje v podkapitole 3.3.2. Tím je získán binární obraz (obrázek 23), kde 0 znamená černou barvu a 1 bílou barvu. Je zřejmé, že to, co se nalézalo v aktuálně filtrovaném hloubkovém pásmu, je bílé. Na tento obrázek je použita kombinace morfologických operací (viz 3.2.2) s účelem vyplnění děr, které

pravděpodobně vznikly chybami snímání Kinectem. Na obrázku 23 vidíme, že kostka se sice odprahovala, ale jsou v ní díry a zejména v pravé straně velké zálivy. Na základě experimentů bylo zjištěno, že pětinásobná dilatace se standardním strukturním elementem celé 8-sousednosti a poté zpětná pětinásobná eroze s týmž strukturním elementem tyto vady skoro zcela odstraní (obrázek 24)

Obrázek 23: Binární obrázek před morfologií   Obrázek 24: Binární obrázek po morfologii

Dalším krokem je nalezení kontur. O to se postará funkce *findContours()*, jak bylo popsáno v podkapitole 3.3.1. Jelikož kontur samozřejmě může být více (v daném hloubkovém pásmu se nacházelo více objektů), musí se vybrat jen jedna, o níž předpokládáme, že je ta pravá. V první fázi programu se hledá jednoduše ta s největší plochou.

V druhé fázi se hledá nikoliv největší kontura, ale ta nejvíce obsahově podobná té z minulé iterace nekonečného cyklu. Samozřejmě se může stát i to, že v obrázku bude více takto obsahově podobných kontur (Obrázek 25). Program je však připraven i na tyto situace. Pokud je euklidovská vzdálenost (viz podkapitola 3.1.2) od nejlepší nalezené kontury a minulé nejlepší nalezené kontury více než 50, je vyřazena a tímto kritériem se pokusí projít další nejlepší kontura ve frontě. Tím se zajistí jakási stabilita sledování té správné polohy skutečné kostky. V příloze A na videu bp013.avi je to samozřejmě demonstrováno lépe než jedním obrázkem.

V druhé fázi se rovněž pohybuje střední hodnota a rozptyly prahovaného intervalu podle aktuálně vracené hodnoty hloubky kostky z Kinectu. Můžeme si tedy dovolit pohybovat se s kostkou nějakou rozumnou rychlostí ve směru osy Z.

Obrázek 25: Binární obrázek

Obrázek 26: Barevný obrázek

Po nalezení nejlepší kontury je tato použita v parametrech funkcí *boundingRect()* a *minAreaRect()* (viz podkapitola 3.1.4) a jsou tak získány dva obdélníky, s nimiž se pracuje v další podkapitole.

5.4 Estimace polohy kostky

Po prahování byly získány dva obdélníky (typ *Rect* a *RotatedRect*) . Z těchto obdélníků budeme získávat další informace kam umístit modelovanou kostku ve světě OpenGL.

Kostka má 6 stupňů volnosti pohybu. Ty jsou ve světě OpenGL reprezentovány systémem souřadnic: X, Y, Z, rotace kolem osy X, rotace kolem osy Y, rotace kolem osy Z, nazvěme odhad těchto stavových proměnných stejně jako na začátku této kapitoly: $\{X, Y, Z, rotX, rotY, rotZ\}$. Je potřeba také řešit velikost kostky, která se samozřejmě zmenšuje se zvětšující se hloubkou v níž se kostka nachází a samotný svět OpenGL se o to nepostará při zvoleném nastavení ideálně, nazvěme ji a.

Souřadnice X a Y jsou vyřešeny znalostí polohy obdélníků zjištěných prahováním. Proměnná typu *RotatedRect* v sobě, jak bylo zmíněno v podkapitole 3.3.2, nese informace o poloze svého středu, výšce a šírce. Souřadnicový systém v typu Mat má standardně počátek v levém horním rohu. V OpenGL je tomu však jinak. Počátek souřadnicového systému je veprostřed v hloubce 0. Vzhledem k tomu, že zorné pole v OpenGL světě je v podstatě jehlan, můžeme na základě dvou hodnot šířky nebo výšky vypočítat přímkovou závislost těchto vlastností okna na aktuální hloubce. Experimentálně byla tedy naměřena celková šířka a výška okna v hloubce rovné near plane a následně bylo provedeno to samé v hloubce *near plane + 1*. Z těchto informací se pak interpolací vygenerovala potřebná lineární funkce.

Jako první je řešena transformace z-ové souřadnice podle rovnice Z-bufferu, který je popsán

v podkapitole 4.1.2. V tomto konkrétním případě tedy:

$$Z = S \cdot \frac{FP \cdot NP}{z \cdot (FP - NP) - FP \cdot S}$$

kde Z značí z-ovou souřadnici ve světě OpenGL, $S = (2^{16} - 1)$, *NP* značí near plane, *FP* značí far plane a z je aktuální hloubka, v níž se kostka právě nachází.

Transformace x-ové souřadnice v závislosti na aktuálně nastavené hloubce ve světě OpenGL:

$$X = \frac{x - 320}{\frac{640}{2.08 - (Z + NP) \cdot 1.2}}$$

kde X značí x-ovou souřadnici ve světě OpenGL, x značí transformovanou souřadnici, Z značí již nastavenou hloubku ve světě OpenGL, *NP* značí near plane a *FP* značí far plane. Číselné hodnoty ve jmenovateli jmenovatele jsou koeficienty přímkové závislosti změny šířky zorného pole na hloubce.

Transformace y-ové souřadnice v závislosti na aktuálně nastavené hloubce ve světě OpenGL:

$$Y = \frac{y - 240}{\frac{640}{1.56 - (Z + NP) \cdot 0.92}}$$

kde Y značí y-ovou souřadnici ve světě OpenGL, y značí transformovanou souřadnici, Z značí již nastavenou hloubku ve světě OpenGL, *NP* značí near plane a *FP* značí far plane. Číselné hodnoty ve jmenovateli jmenovatele jsou koeficienty přímkové závislosti změny výšky zorného pole na hloubce.

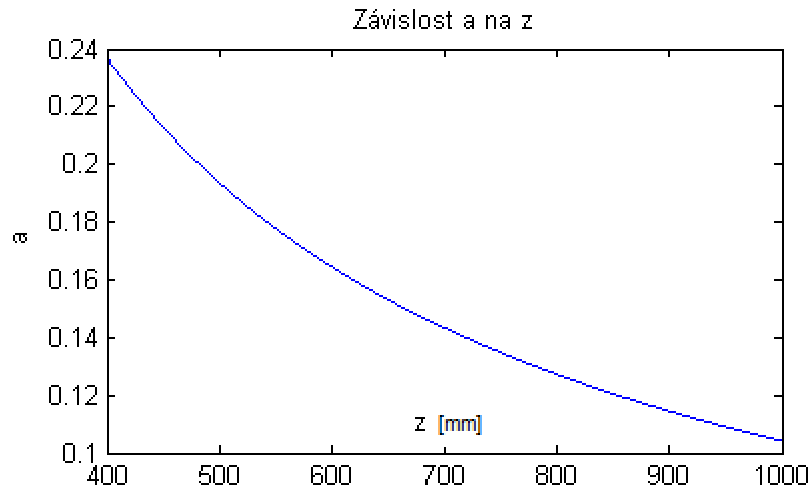
Rotace kolem osy Z je vyřešena informací, kterou v sobě rovněž nese *RotatedRect*, nazvěme ji *angle*. Hodnota se nastavuje podle jeho úhlu natočení. Tedy:

$$rotZ = angle$$

Další v pořadí je řešena velikost vykreslované kostky. Jak bylo řečeno posouvání kostky do hloubky ji samozřejmě samo také zmenšuje, ale při zvoleném nastavení OpenGL světa ji to nezmenšuje dostatečně rychle a proto musela být nalezena funkce, která by toto nastavení dostatečně přesně kompenzovala a velikost kostky byla správná. Toho bylo dosaženo naměřením několika dat a programem MS Office Excel, který z těchto dat interpoloval hyperbolu:

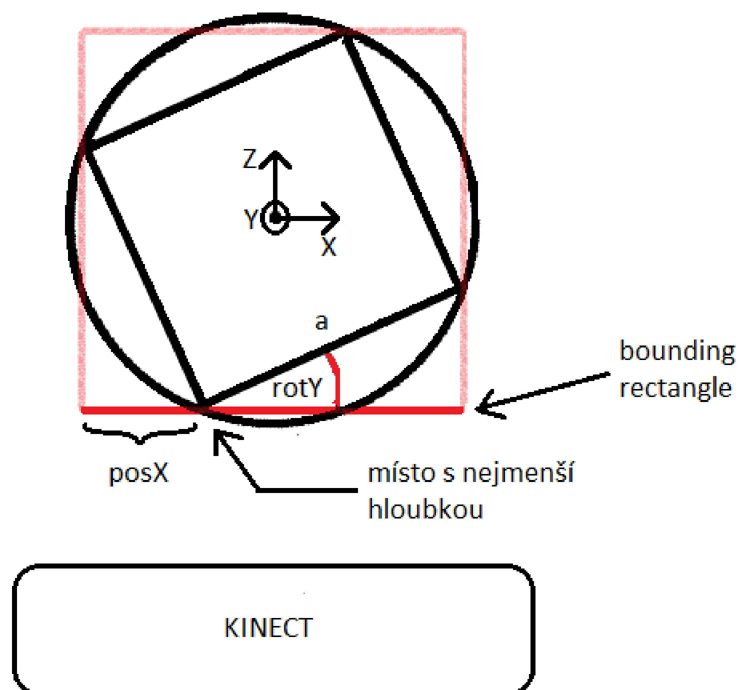
$$a = 50.816 \cdot z^{-0.896}$$

kde z je aktuální hloubka, v níž se kostka právě nachází a číselné hodnoty jsou koeficienty nalezené hyperboly. Na Obrázku 27 je znázorněn vývoj velikosti kostky v intervalu od 40 centimetrů do 1 metru od Kinectu.



Obrázek 27: Závislost velikosti a na hloubce z

Rotace kolem osy Y je řešena tak, že se v prahováním získaném *Rectu* nalezne pixel s nejmenší hloubkovou hodnotou, respektive jeho souřadnice, nazvěme je $posX$ a $posY$. Obrázek 28 je pohled na tuto situaci shora.



Obrázek 28: Řešení rotace kolem osy Y

Díky těmto informacím známe vzdálenost od levé hrany *Rectu* do dané souřadnice a při zcela ideálně provedeném prahování pak můžeme spočítat úhel natočení podle vzorce:

$$rotY = \arcsin\left(\frac{posX}{a}\right)$$

Totéž se dá tvrdit i o rotaci kolem osy X:

$$rotX = \arcsin\left(\frac{posY}{a}\right)$$

Kde a je vypočítaná velikost kostky při dané aktuální hloubce z.

Nyní jsou vypočtena všechna potřebná data k vykreslení kostky v OpenGL světě. Použitím OpenGL funkce (viz podkapitola 4.1.2) *glTranslatef()* se změní translační prvky transformační matice na vypočtené $\{X,\ Y,\ Z\}$. Pomocí funkce *glRotatef()* se změní rotační prvky transformační matice na vypočtené $\{rotX,\ rotY,\ rotZ\}$. Víme, že chceme kreslit kostku, tudíž je potřeba zadat její vrcholy (vertexy) a to pomocí funkce *glVertex3f()*, do které zadáme souřadnice OpenGL světa. V OpenGL se dají tvořit rovnou čtverce, takto tedy vytvoříme 6 čtverců tvořící krychli. Tuto krychli je potřeba ještě zmenšit nebo zvětšit, aby měla správnou velikost. To provedeme přenásobením velikostí stran kostky požadovanou velikostí a.

5.5 Particle Filter

Třetí fáze programu nabízí možné doladění chyb, kterých se odhady dopustily. Třetí fáze programu začíná spuštěním funkce particle filteru (viz 3.3.2). V první řadě je potřeba particle filter inicializovat. To znamená definovat, kterých a kolika stavových proměnných se má particle filter dotýkat. A v jakých intervalech se proměnné v těchto particlech mohou pohybovat. Dále pak také jak moc má nově generované particly zašumět při resamplingu. Použitý particle filter vezme stavové proměnné *rotX, rotY* (ostatní stavové proměnné zůstávají stejné) a vytvoří P particlů, kde jsou tyto dvě proměnné rovnoměrně rozděleny do kombinací, kde se obě pohybují v intervalech $< rotI - \delta; rotI + \delta >$, kde I je X nebo Y, a δ nějaký zvolený rozptyl. Těmto P particlům je přidělena váha w_i funkcí *observe()*, pak je provedena normalizace, aby součet vah byl jedna, a *resampling*. Filter dále už generuje nové particly kolem těch s největší pravděpodobností a opakuje celý proces a tak pomalu přichází na hledanou hustotu pravděpodobnosti rozdělení všech particlů. Na konci tohoto procesu se vybere nejpravděpodobnější particle, který je vykreslen jako finální řešení polohy kostky.

Je evidentní, že celý algoritmus stojí především na funkci *observe()*, která přiřazuje particlům jejich váhu. V tomto programu byla zvolena nejjednodušší metoda a to metoda srovnávání návrhu (OpenGL) s originálem (Kinect - hloubková mapa). To znamená vykreslení všech particlů v OpenGL světě, přečtení z OpenGL světa funkcí *glReadPixels()* a uložení těchto dat do struktury

Mat stejného typu jako je skutečná hloubková mapa, z těchto dvou matic vyříznout prahováním nalezený *Rect* a v absolutní hodnotě tyto od sebe odečíst, sečíst všechny prvky ve vzniklé matici a celý výsledek podělit velikostí *Rectu*:

$$Score = 65535 - \frac{\sum_{i=0}^{rw} \sum_{j=0}^{rh} |f_{GL}(i,j) - f_K(i,j)|}{rw * rh}$$

kde rw je šířka *Rectu*, rh výška *Rectu*, $f_{GL}(i,j)$ je obrazová funkce (viz 3.1.1) hloubkového obrazu přečteného ze světa OpenGL na pozici *(i,j)* a $f_K(i,j)$ je obrazová funkce hloubkového obrazu z Kinectu. Score je pak invertované, tím pádem čím vyšší výsledek, tím lepší shoda modelu s originálem.

Po vybrání nejlepšího particlu se odhadnutý stav aktualizuje a kostka se v tomto stavu vykreslí v OpenGL světě. Po vykreslení v OpenGL světě použijeme znovu metodu *glReadPixels()* k přečtení barevného obrazu, který poté spojíme s barevným obrazem z Kinectu. Výsledkem by měla být ruka svírající modelovanou kostku například jako v Obrázku 29:

Obrázek 29: Barevný obraz s modelovanou kostkou

6 Problémy při zpracování

Při běhu programu se mohou vyskytnout jisté problémy, které se buď dají řešit dodržováním jistých předpokladů, nebo jsou řešitelné jen velmi těžko.

6.1 Jas

Prvním problémem, který se dotýká snad všech úloh počítačového vidění, byl možný výskyt přílišného jasu. Ten se podepsal pod špatným čtením hloubkové mapy z Kinectu, pokud byla snímaná oblast příliš osvícena sluncem. To poté vedlo buď k nekvalitně provedenému prahování, vzniku děr a šumů, nebo absolutní tmě (nic nenalezeno). Jednou možností vyřešení tohoto problému se tak jeví neprovozovat program u zdrojů světla jako oken, lamp atd. Další možností je zmatnění povrchu snímaného objektu.

6.2 Posouvání prahu

Posouvání prahu jako takového není až takovým problémem jako posouvání rozptylu. Podle křivky znázorňující funkci depth bufferu (viz obrázek 20) vidíme, že v oblasti nízkých hloubek, ve kterých se sledovaná kostka pohybuje, je vyžadován velký interval k prahování, aby měla šanci odprahovat se celá kostka. Čím je kostka dále od Kinectu, tím menší by tento interval měl být, protože pak vzniká problém s tím, že se odprahují také prsty, které kostku drží. To vede ke zvětšení výsledné kontury a špatným výsledkům, co se týče obdélníků *Rect* a *RotatedRect*, které jsou esenciální pro ideální chod programu. To se dá řešit buď "správným"držením kostky, kde se uživatel snaží držet ji tak, aby se v takových případech prsty shovaly za kostku, nebo nedržet kostku v ruce, ale například na saturnách, které by nebyly Kinectem zpozorovány, jako loutku.

6.3 Odhad rotace

Nejtěžší úlohou na celém programu bylo řešení rotace kolem x-ové a y-ové osy. Důvodem nutnosti tohoto odhadu bylo ulehčení práce particle filteru. Je vhodné na vstup particle filteru totiž vložit nějaká data, od kterých by mohl pracovat, než aby musel projíždět kompletně celý stavový prostor, protože čím více particlů, tím pomalejší funkce. Při odhadování rotací byla vymyšlena spousta možných postupů, z nichž nakonec fungoval alespoň přibližně tento jeden implementovaný. Rotace fungují především samostatně. Pokud by však uživatel natočil kostku jejím vrcholem proti kameře, program by to pravděpodobně nezvládl. Dalším problémem při řešení rotací bylo to, že se kostka v OpenGL světě dostala nějakou svou částí před near plane a tudíž se nevykreslila správně. To bylo vyřešeno umělým posunutím kostky o trochu dál, než měla být podle rovnice depth bufferu (viz vztah pro Z v 5.4). To však potom dělá problém ve funkci *observe()* v particle

filteru, kde rovnice pro srovnávání nefunguje správně, protože z OpenGL se přečte o něco rozdílná hloubka, než z Kinectu.

6.4 FPS

FPS, neboli frames per second znamená v podstatě rychlost programu. Při vypnutém particle filteru v Release verzi programu se FPS pohybuje v intervalu přibližně od 25 do 30, což je velmi slušná rychlost. Jestliže však spustíme particle filter, konkrétně s 30 particly o dimenzi 2 s maximálně 5 kroky odhadování hustoty pravděpodobnosti, FPS klesne až pod 1, což zcela rozhodně nestačí, pokud má jít o real-time aplikaci. Problémem je, že během výpočtu se musí všechny particly vykreslit ve světě OpenGL a porovnat s originálem. To znamená, že na jedno vykreslení do skutečného obrazu vychází až 150 vykreslení pokusných. Možným řešením tohoto problému by byla implemetace particle filteru na GPU.

7 Závěr

Úlohou této práce bylo seznámení se s 3D senzorem Kinect, knihovnou OpenCV a vykreslováním 3D grafiky v OpenGL. Vzhledem k rozsáhlým dokumentacím přístupným na internetu, jakož i množstvím již zodpovězených otázek na různých fórech, to nebyl těžký úkol. Vstupními daty byla hloubková mapa z Kinectu, tato mapa byla převedena pomocí OpenNI do struktury Mat z OpenCV. Aplikací segmentačních technik byla zpracována a byla odhadnuta poloha a stav pozorovaného objektu. Obejktem byla Rubikova kostka. Tato kostka byla v rámci možností co nejvěrněji napodobena vykreslením do OpenGL světa. Modelovaná kostka poté byla doplněna do RGB obrazu z Kinectu.

Během práce se vyskytlo mnoho problémů (viz kapitola 6). Hlavním cílem bylo pokusit se přiblížit algoritmu popsanému v *Efficient model based tracking of the articulated motion of hands* [1]. Co se týče odhadu polohy kostky za použití různých matematických nástrojů, dopadl slušně, viz Příloha B. Kostka je doopravdy sledována za předpokladu, že s ní uživatel před kamerou nepohybuje rychlostí, která není v souladu s rychlostí snímání Kinectu. Můžeme tvrdit, že 4 z 6 stupňů volnosti jsou vyřešeny celkem přesně, hlavním problémem zůstávají netriviální rotace kolem x-ové a y-ové osy, které byly sice vyřešeny, ale toto řešení je často nepřesné, protože prahování je nestabilní. Může zde proto docházet k nesmyslným chybám. Snaha zabránit těmto chybám Particle Filterem byla neefektivní.

Jistým vylepšením v budoucnosti může být naprogramování Particle Filteru na grafický procesor (GPU), který by snázeji stíhal vykreslovat všechny možné particly a počítat funkci skore porovnáváním s originálem z Kinectu. Další věcí, která by měla být vyřešena je podmíněnost správného držení kostky před Kinectem, to by znamenalo nějak odlišit kostku od ruky, například prahováním barvy lidské kůže. Buď tak, nebo nesledovat vůbec kostku, ale zaměřit se pouze na ruku jako v [1].

8 Literatura

[1] I. Oikonomidis, N. Kyriazis, and A. A. Argyros, "Efficient model-based 3d tracking of hand articulations using kinect." in *BMVC*, 2011, pp. 1–11.

[2] M. Vinkler, "Využití pohybového snímače kinect ve virtuální realitě," *Masarykova univerzita, Brno*, 2012.

[3] D. Nitescu, D. Lalanne, and M. Schwaller, "Evaluation of pointing strategies for microsoft kinect sensor device," *Master Thesis, Universidade de Bern, Universidade de Neuchatel, Univesidade de Fribourg*, 2012.

[4] M. Fisher, "Matt's webcorner: Kinect," 2014. [Online]. Available: http://graphics.stanford.edu / mdfisher/Kinect.html

[5] Microsoft, "Kinect for Windows Sensor Components and Specifications." [Online]. Available: http://msdn.microsoft.com/en-us/library/jj131033.aspx

[6] DotNetNuke Corporation, "Open Natural Interaction," 2011. [Online]. Available: http://openni.org

[7] M. Železný, "Zpracování digitálního obrazu," přednášky, *ZČU FAV KKY*, 2005.

[8] M. Sonka, V. Hlavac, R. Boyle *et al.*, *Image processing, analysis, and machine vision.* Thomson Toronto, 2008, vol. 3.

[9] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[10] V. Hlaváč, "Matematická Morfologie," přednášky, *ČVUT Fakulta elektrotechnická, katedra kybernetiky.*

[11] S. Suzuki *et al.*, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32–46, 1985.

[12] J. M. Hammersley and K. W. Morton, "Poor man's monte carlo," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 23–38, 1954.

[13] F. Desbouvries and B. Ait-El-Fquih, "Direct versus prediction-based particle filter algorithms," in *Machine Learning for Signal Processing, 2008. MLSP 2008. IEEE Workshop on.* IEEE, 2008, pp. 303–308.

[14] OpenGL Working Group *et al.*, "Opengl documentation," 2008.

[15] J. Molofee, "Opengl tutorials," *Nehe Productions. http://nehe. gamedev. net.*

9 Příloha

Přílohy se nacházejí na CD uloženém v zadní kapse desek práce.

Příloha A

První přílohou je video, které zobrazuje změnu v hledání správné kontury v první a druhé fázi programu (viz Kapitola 5). V první polovině videa je vidět, že se hledá pouze největší možná kontura a program se tudíž chová nesmyslně a vykresluje kostku špatně. Je také vidět, že pokud do stejného hloubkového pásma vložíme stejnou kostku, program se může soustředit na obě střídavě. V druhé polovině videa je znázorněna funkce druhé fáze. Program již může prahovat kostku v jiných hloubkových hladinách a rovněž si udrží soustředění na jedné, té správné. Jakkoliv kolem ní máváme s čímkoliv podobným, program se ve většině případů udrží. Toto video je nazváno bp013.avi.

Příloha B

Druhou přílohou je video, které znázorňuje celou funkci programu. Video by mělo ukázat odhad postupně všech stavových veličin a to v pořadí: Z, Y, X, rotZ, rotY, rotX. Vidíme, že jednotlivé stavové veličiny řešeny samostatně nejsou odhadovány zcela nepřesně. V posledních několika sekundách videa je vidět funkce Particle Filteru, která nevypadá příliš přesvědčivě kvůli zmíněnému problému s FPS (viz Kapitola 6). Toto video je nazváno bp012.avi.

Příloha C

Třetí přílohou je pak samotný kód napsaný v C++. Kód sestává z hlavního souboru *main.cpp*, který definuje okno, OpenGL prostředí a zpracovává obraz z Kinectu až do stádia odhadnutí stavu, v kterém se kostka má vykreslit. Je v něm rovněž vytvořena instance objektu ParticleFilter, který je definován v souborech *ParticleFilter.h* a *ParticleFilter.cpp*. Metody tohoto objektu se pak starají o zpřesnění stavu kostky.