MI-DZO
Hloubka ostrosti, rozmazání pohybem, hybridní obraz

Ondřej Červenka

17. května 2017

1 Úvod

V této práci jsem implementoval několik algoritmů pro zpracování obrazu: rozmazání obrazu podle jeho hloubokvé mapy, simlulaci efektu pohybové neostrosti, a tvorbu hybridních obrazů.

2 Analýza

2.1 Hloubka ostrosti

Cílem této části práce bylo vytvořit program, který provede rozmazání obrázku na základě dodané hloubkové mapy. Základní vstup programu tedy představuje dvojice *obrázek, hloubková mapa* (viz **obrázek 1**).

Dále si uživatel zvolí místo, které bude odpovídat nejvyšší hladině při rozmazávání. Míra rozmazání každého pixelu tedy není určena absolutně hloubkou pixelu podle hloubkové mapy, ale rozdílem hloubek mezi právě zpracovávaným pixelem a místem, které zvolil uživatel.

Díky tomu může uživatel například rozmazat předměty v popředí obrázku a „zaostřit“ na předměty v pozadí, nebo naopak.

Při implementaci jsem pro rozmazávání obrázku použil bilaterální a box filtr.

2.1.1 Box filtr

Při filtrování obrazu pomocí box filtru je každý pixel výstupního obrázku tvořen průměrem okolních pixelů vstupního obrázku. Jádro velikosti $n = 3$ tohoto filtru tedy vypadá následnovně:

$$K_{n,n} = \frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

(a) Původní obrázek (b) Hloubková mapa

Obrázek 1: Obrázek a jeho hloubková mapa z datasetu Middlebury (`http://vision.middlebury.edu/stereo/`). U hloubkové mapy platí, že čím vyšší hodnota, tím blíže je část obrázku ke kameře. Nejvzdálenější části jsou tedy černě

Velikost jádra je přímo úměrná rozmazání obrázku. Při rozmazávání bude tedy každý pixel výstupního obrázku průměrem $n * n$ pixelů vstupního obrázku, kde n je určeno rozdílem hloubky právě filtrovaného pixelu a hloubky zvolené uživatelem (konkrétní mapování hloubek na velikost jádra je popsáno v kapitole 3.1).

2.1.2 Bilaterální filtr

Biletarální filtr je nelinární filtr definovaný následujícím vzorcem[1]:

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(||p - q||) G_{\sigma_r}(I_p - I_q) I_q$$

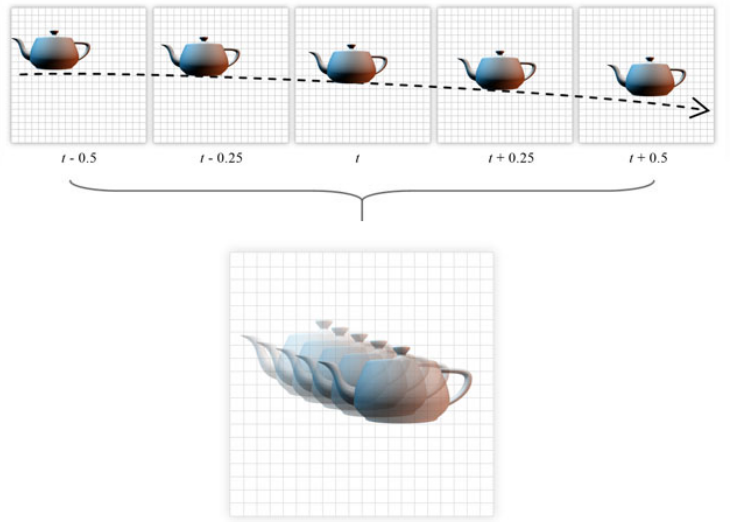
kde:

- p jsou souřadnice právě filtrovaného pixelu
- I_p je intenzita právě filtrovaného pixelu
- S je okénko pixelů se středem v p
- $G_\sigma(x) = e^{\frac{-x^2}{2*\sigma^2}}$
- $BF[I]_p$ je intenzita výsledného pixelu ve výstupním obrázku

a W_p je normalizační faktor pro pixel p, daný vzorcem:

$$W_p = \sum_{q \in S} G_{\sigma_s}(||p - q||) G_{\sigma_r}(I_p - I_q)$$

Jak je vidět, je zde kromě intenzity brána v potaz také vzdálenost pixelu od právě filtrovaného pixelu.



Obrázek 2: Jednoduchá implementace pohybové nerovnosti. Zdroj obrázku: `http://john-chapman-graphics.blogspot.cz/2013/01/what-is-motion-blur-motion-pictures-are.html`

My ale chceme do filtrování zapojit hloubkovou mapu, a proto intenzitu pixelů $(I_p - I_q)$ nahradíme rozdílem hloubek. Díky tomu bude při filtrování každého pixelu potlačen vliv pixelů v příliš rozdílných hloubkách. Upravený vzorec pro bilaterální filtr bude vypadat takto:

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_d}(D_p - D_q) G_{\sigma_s}(||p - q||) I_q$$

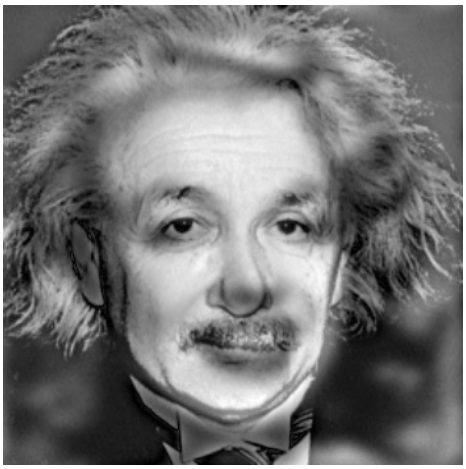
$$W_p = \sum_{q \in S} G_{\sigma_d}(D_p - D_q) G_{\sigma_s}(||p - q||)$$

kde D_p, D_q jsou hloubky pixelů dány hloubkovou mapou.

2.2 Pohybová neostrost

Nejjednodušší způsob, jak vytvořit efekt pohybové neostrosti, je nasnímat obrázek v různých posunutích a z těchto vzorků vytvořit vážený součet. Tento postup je zobrazen na **obrázku 2**.

Nevýhoda tohoto postupu je, že se nehodí pro většinu realitme aplikací kvůli jeho rychlosti[2]. Pro naši ukázkovou implementaci však bude dostačující.



Obrázek 3: Hybridní obrázek sestavený z portrétů Alberta Einsteina a Marilyn Monroe. Z dálky se obrázek zdá jako portrét Marilyn Monroe, při pozorování z blízka však vynikne Einstein.

2.3 Hybridní obrazy

Hybridní obraz je složený z dvou různých obrazů za účelem poskytnutí dvou různých interpretací v závislosti na pozorovací vzdálenosti. Obraz se tedy jeví jako něco jiného při pozorování z blízka a z dálky.

Obrázek 3 byl vytvořen aplikací *low pass* a *high pass* filtru na jednotlivé portréty (viz **obrázek 4**) a jejich následným sečtením.

Obě tyto části hybridního obrázku můžeme získat vyfiltrováním vysokofrekvenčních, respektive nízkofrekvenčních komponent z Fourierova spektra portrétů pomocí gaussovské funkce[3]:

$$I_l = \mathcal{F}^{-1}(\mathcal{F}(I_1) * G)$$

$$I_h = \mathcal{F}^{-1}(\mathcal{F}(I_2) * (1 - (G))$$

kde operace $*$ představuje násobení jednotlivých prvků dvou stejně velkých matic a G je matice, pro jejíž prvky platí: $g_{x,y} = e^{\frac{(x-c_x)^2+(y-c_y)^2}{2\sigma^2}}$, c_x, c_y jsou souřadnice středu matice[4].

Po vyfiltrování vysokofrekvenčních komponent a zpětné transformaci z Fourierova spektra tedy vznikne efekt rozmazání, což bude obrázek, který převáží při pozorování z dálky.

Naopak po odstranění nízkofrekvenčních komponent zůstanou pouze výrazné hrany obrázku, které vyniknou jen při pozorování zblízka. Zde je pro dosažení požadovaného efektu důležité, aby obrázek filtrovaný *low pass* filtrem vhodně doplnil hrany získané *high pass* filtrem.

(a) Portrét Alberta Ensteina filtrovaný *high pass* filtrem

(b) Portrét Marilyn Monroe filtrovaný *low pass* filtrem

Obrázek 4: Portréty tvořící hybridní obrázek po aplikaci *low pass* a *high pass* filtru

Výsledný hybridní obrázek, vzniklý součtem *low pass* a *high pass* filtrů, je tedy definován takto:

$$H = I_l + I_h = \mathcal{F}^{-1}(\mathcal{F}(I_1) * G) + \mathcal{F}^{-1}(\mathcal{F}(I_2) * (1 - G))$$

3 Implmentace

První dvě úlohy (hloubka ostrosti a pohybová neostrost) jsem implementoval v jazyce C++, a to převážně kvůli rychlosti. K implementaci jsem použil knihovnu OpenCV[1] (verze 3.2.0), především pro načítání a zápis obrázků (knihovna poskytuje třídu `cv::Mat` pro práci s maticí obrázku).

Třetí úlohu jsem implementoval v jazyce Python s použitím matematické knihovny Numpy[2]. Tato knihovna obashuje třídu `ndarray`, která se hodí pro uložení obrázku, a také implementace rychlé Fourierovy transformace.

K načítání a zápisu obrázku jsem i zde použil OpenCV (šlo by však využít jakoukoliv jinou Python knihovnu poskytující funkce pro načtení/zápis obrázku, například SciPy).

Hlavní důvod, proč jsem pro třetí úlohu zvolil Python a Numpy je ten, že implementace v tomto prostředí je velmi snadná a v podstatě přesně kopíruje matematický popis ze sekce 2.3.

Jelikož zde budou prakticky pouze volány funkce z knihovny Numpy, které jsou efektivně implementovány v jazyce C, neměla by tato volba mít výraznější vliv na dobu běhu programu.

[1] `http://opencv.org/`
[2] `http://www.numpy.org/`

3.1 Hloubka ostrosti

Implementace tohoto algoritmu je poměrně přímočará. Program po načtení vstupů spočítá postupně každý pixel výsledného obrázku podle zvoleného filtru.

Jak bylo zmíněno v sekci 2.1, velikost jádra (počet okolních pixelů, ze kterých je počítán výsledný pixel) závisí na rozdílu hloubek mezi právě filtrovaným pixelem a uživatelem zvoleným bodem v hloubkové mapě. Přesná implementace je v ukázce 1.

Ukázka 1: Výpočet velikosti jádra

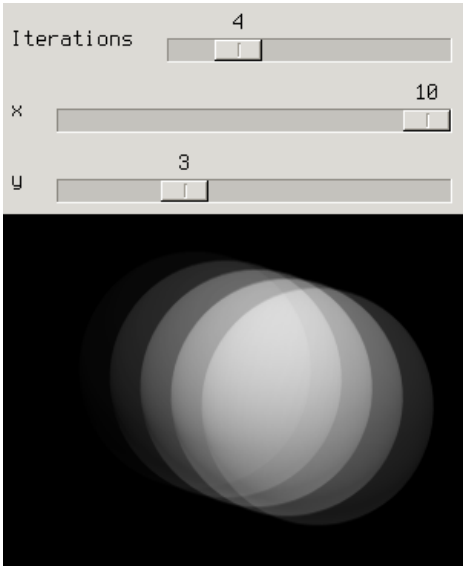
```
/*
depth -- hloubka filtrovaneho pixelu
target_depth -- hloubka zvolena uzivatelem
*/
int get_kernel_by_depth(uchar depth,
                        uchar target_depth,
                        double min_depth,
                        double max_depth)
{
  double maxdiff = max_depth - min_depth;
  double depthn = ((double) depth - min_depth) /
     maxdiff;
  double target_depthn = ((double) target_depth -
     min_depth) / maxdiff;

  double diff = abs(depthn - target_depthn);

  if (diff < 0.2) return 1;
  if (diff < 0.5) return 5;
  if (diff < 0.7) return 11;
  return 13;
}
```

V případě box filtru je pak výsledný pixel tvořen průměry RGB složek okolních pixelů.

U bilaterálního filtru je jako vzdálenost pixelů $||p - q||$ (parametr funkce G_{σ_r}) zvolena euklidovská vzdálenost. Místo intenzit je použit rozdíl hloubek filtrovaného pixelu a okolních pixelů, jak je zmíněno v sekci 2.1.2.

Samotná knihovna OpenCV neumožňuje tvořit propracovanější grafická rozhraní, dovoluje však zobrazení okna s obrázkem a odchytávání stisknutých kláves.

Vytvořil jsem tedy velmi jednoduché rozhraní, kde si uživatel pomocí směrových šipek a tečky vykreslené do obrázku může zvolit hloubku, podle které se bude rozmazávat. Poté stisknutím mezerníku aplikuje box filtr, stisknutím klávesy `Enter` aplikuje bilaterální filtr.



Obrázek 5: Aplikace pro demonstraci vytváření efektu pohybové neostrosti.

Klávesou `Alt` je možno přepínat mezi zobrazením původního obrázku a hloubkové mapy při volbě hloubky, klávesa `s` uloží obrázek po aplikaci filtru do souboru `res.png`. Program se ukončí stiskem klávesy `Esc`.

3.2 Pohybová neostrost

Při implementaci efektu pohybové neostrosti jsem použil postup popsaný v sekci 2.2.Vytvořil jsem tedy několik posunutí vstupního obrázku, a tyto posunutí jsem sečetl pomocí funkce `addWeighted()` z knihovny OpenCV. Váženým součtem vznikl efekt překryvu jednotlivých fázi posunu.

Posun jednotlivých fází jsem implementoval pomocí affinní transformace (funkce `warpAffine()` z OpenCV). Díky tomu si může uživatel zvolit simulovaný směr pohybu předmětu. Kromě toho uživatel také volí počet iterací (posunů).

V demo aplikaci (**obrázek 5**)je možné směrový vektor a počet iterací nastavovat pomocí posuvníku[3]. Po ukončení aplikace (klávesa `Esc`) je výsledek uložen do souboru `res.png`.

3.3 Hybridní obrazy

Implementace prakticky kopíruje matematický popis ze sekce 2.3. Jednotlivé portréty se tedy převedou do Fourierova spektra, poté jsou pomocí Gaussova

[3]Posuvník je jeden z mála GUI prvků, které OpenCV defaultně podporuje. Pokud bychom chtěli vytvářet složitější rozhraní, je možné OpenCV zkompilovat s podporou frameworku Qt.

filtru vyfiltrovány nízkofrekvenční a vysokofrekvenčí komponenty a výsledek je převeden zpět pomocí inverzní Fourierovy transformace.

Funkce pro Fourierovu transformaci a její inverzi poskytne knihovna Numpy. Kromě těchto funkcí je zapotřebí ještě funkce `fftshift()`, která posune komponenty s nulovou frekvencí do středu matice spektra (po transformaci se tyto komponenty nacházejí v rozích matice).

Pak už je potřeba jen funkce, která vytvoří matici Gaussova filtru odpovídající velikosti. Tato funkce je uvedena v **ukázce 2**.

Ukázka 2: Vytvoření matice gaussiánu

```
#rows, cols jsou rozmery vstupnich obrazku
def get_gaussian(rows, cols, sigma):
    c_y = rows // 2 + (rows % 2)
    c_x = cols // 2 + (cols % 2)

    def gaussian(x, y):
        exponent = ((x - c_x)**2 + (y - c_y)**2)/
            (-2*sigma**2)
        return np.exp(exponent)

    return np.fromfunction(
      lambda x, y : gaussian(x,y),
      (rows, cols))
```

Z uživatelského hlediska je program velmi jednoduchý. Prostě spočítá výše popsanou metodou hybridní obrázek ze dvou vstupních obrázků, které jsou zadávány jako argumenty při spouštění příkazové řádky. Třetí (volitelný) argument je σ Gaussova filtru. Vstupní obrázky musí mít stejnou velikost, výsledný obrázek je uložen do souboru `res.png`.

Bohužel také není zaručeno, že výsledný obrázek bude dosahovat požadovaného efektu. Důležité je především zvolit vhodně se doplňující vstupní obrázky.

4 Výsledky

4.1 Hloubka ostrosti

Na **obrázcích 6** je vidět výsledek obou filtrů v případě, že uživatel zvolil pro „zaostření“ nejvzdálenější bod (v tomto případě stěna za dřevěnou mříží, viz hloubková mapa 1).

V tomto případě není vidět znatelný rozdíl mezi bilaterálním filtrem a box filtrem. To je pravděpodobě způsobeno tím, že mezi nejvíce rozostřenými předměty (kornouty v popředí) nejsou až tak výrazné změny v hloubce.

Závisí tedy na volbě parametru σ_d. Nižší hodnoty více potlačí váhu pixelů i v méně rozdílných hloubkách, díky čemuž vyniknou hrany předmětů i v případě

(a) Bilaterální filtr při zaostření na nejvzdálenější předměty, $\sigma_d = 100, \sigma_s = 60$

(b) Box filtr při zaostření na nejvzdálenější předměty

Obrázek 6: Porovnání bilaterálního a box filtru při zaostření na nejvzdálenější předměty

výrazného rozmazání. Záleží tedy jakého efektu chceme dosáhnout.

Na **obrázcích 7** je naopak zaostřeno na předměty v popředí. Zde je vliv bilaterálního filtru výraznější, především na dřevěné mřížce v zadu na obrázku. Zde byly náhlé změny hloubky výraznější, a proto více vynikly hrany mřížky.

(a) Bilaterální filtr při zaostření na blízké předměty, $\sigma_d = 100, \sigma_s = 60$

(b) Box filtr při zaostření na blízké předměty

Obrázek 7: Porovnání bilaterálního a box filtru při zaostření na blízké předměty

4.2 Pohybová neostrost

Efekt rozostření pohybem nejlépe funguje na jednoduchých objektech, ideálně na jednobarevném nebo průhledném pozadí, jako je například míček na **obrázku 8a**.

(a) Efekt rozmazání pohybem aplikovaný na míček, 6 iterací

(b) Efekt rozmazání pohybem aplikovaný na auto, 13 iterací

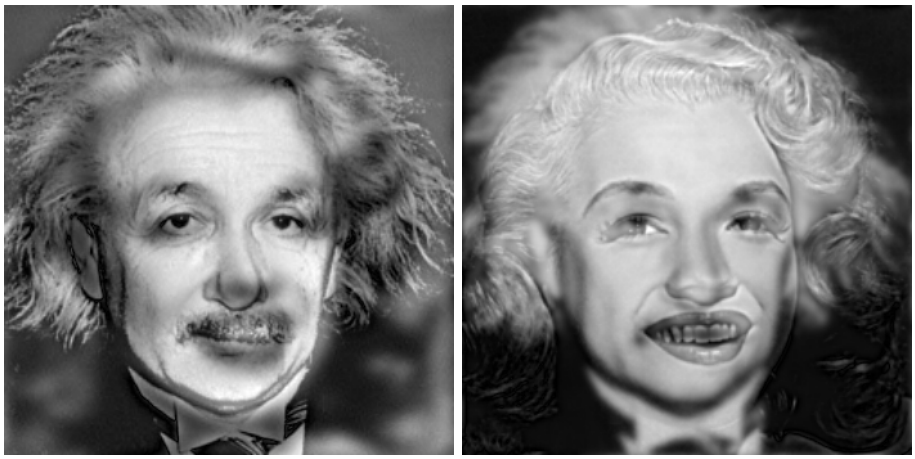
Rozmazání složitějšího objektu (byť stále s jednoduchým pozadím), je na **obrázku 8b**. Zde se efekt také poměrně vydařil, bylo však třeba vhodně nastavit počet iterací a směrový vektor.

Zajímavá by byla možnost doplnění takto rozmazaného obrázku na (ostré) pozadí. Tuto funkci jsem však bohužel už nestihl do programu doplnit.

4.3 Hybridní obrazy

Program pro tvorbu hybridních obrázků jsem ladil na kombinaci portrétů Alberta Einsteina a Marilyn Monroe. Výstup mého programu při těchto vstupních obrázcích byl ukázán v sekci 2.3 (obrázek 3).

Dále jsem vyzkoušel, zda bude tento obrázek fungovat i opačně, tedy tak, aby se z dálky zdál jako portrét Einsteina a z blízka jako portrét Monroe. Porovnání výsledků je na **obrázku 9**.



(a) *High pass* Einstein, *low pass* Monroe, $\sigma = 10$

(b) *Low pass* Einstein, *high pass* Monroe, $\sigma = 20$

Obrázek 9: Srovnání původního hybridního obrázku a jeho obrácené varianty

Ukázalo se, že obrázek se sice z dálky jeví jako Einstein (a první část efektu je tedy splněna), z blízka nedokáže portrét Monroe tolik potlačit Einsteinovy rysy.

5 Závěr

V této práci jsem implementoval algoritmy pro rozostření obrázku podle jeho hloubkové mapy, simulaci rozostření pohybem a tvorbu hybridních obrázků. K implementaci jsem použil jazyky C++ a Python a knihovnu OpenCV.

Vytvořené programy plní sou základní funkci, je zde však stále prostor pro rozšiřování. Bylo by například vhodné doplnit propracovanější grafická rozhraní. Algoritmy jsou také implementovány velmi přímočarým způsobem, a je zde tedy prostor pro další optimalizace.

5.1 Zdrojové kódy

Všechny zdrojové kódy vytvořené v rámci této práce jsou k dispozici v repozitáři na `https://github.com/ggljzr/mi-dzo`.

Reference

[1] Paris, S.; Kornprobst, P.; Tumblin, J.; aj.: A Gentle Introduction to Bilateral Filtering and its Applications. 2007, [online, cit. 2017-05-14]. Dostupné z: https://people.csail.mit.edu/sparis/bf_course/course_notes.pdf

[2] Chapman, J.: Motion Blur Tutorial. 2011, [online, cit. 2017-05-014]. Dostupné z: http://john-chapman-graphics.blogspot.cz/2013/01/what-is-motion-blur-motion-pictures-are.html

[3] Oliva, A.; Torralba, A.; Schyns, P. G.: Hybrid images. 2006, [online, cit. 2017-05-11]. Dostupné z: http://cvcl.mit.edu/publications/OlivaTorralb_Hybrid_Siggraph06.pdf

[4] Kun, J.: Making Hybrid Images. 2014, [online, cit. 2017-05-11]. Dostupné z: https://jeremykun.com/2014/09/29/hybrid-images/