ZPO - Cannyho detektor hran

David Bayer (xbayer09)
Michal Glos (xglosm01)

6. května 2022

1 Společná část

1.1 Zadání

Vytvořte aplikaci realizující Cannyho hranový detektor. Vstupními parametry detektoru jsou: standadrní odchylka Gaussova filtru a horní a dolní hranice hystereze při prahování. Implementujte dvě verze filtru - demonstrační, složenou z jednotlivých kroků detekce zvlášť, funkce pro Gaussovo rozmazání, potlačení nemaxim, apod., a optimální, jedna efektivně pracující funkce.

1.2 Cannyho detektor hran

Teorie

Cannyho detektor hran je široce používáná metoda z oblasti zpracování obrazu pro detekci hran. Je praktickou aproximací matematicky optimálního detektoru hran.

Proces detekce hran Cannyho metodou se skládá celkem z 6 kroků.

1. Převedení obrázku do odstínů šedi.

$$G = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

2. Vyhlazení obrazu pomocí *Gaussova filtru*. Jedná se o filtr, jehož koeficienty jsou jsou spočteny Gaussovou funkcí.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

3. Aplikace Sobelova filtru pro získání gradientů G a jejich úhlů Θ.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * img, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * img$$

$$G = \sqrt{G_x^2 + G_y^2}, \quad \Theta = \tan^{-1} \frac{G_y}{G_x}$$

4. Potlačení ne-maximálních hodnot. Podle úhlů získaných z předešlého kroku určíme ke každé detekované hraně (pro každý pixel) normální vektor. Hodnotu pixelu p_x nastavíme na hodnotu 0, pokud má alespoň jeden z jeho sousedů ve směru svého normálního vektoru hodnotu vyšší, než původní pixel s maximální hodnotou oproti pixelům ve směru normálního vektoru vúči detekované hraně. Tím se detekované hrany zúží na šířku 1 pixelu.

5. Dvojité práhování. Pomocí koeficientů *htr* a *ltr* se určí vysoký a nízký práh pomocí vzorců:

$$treshold_{low} = ltr \cdot \max(image)$$
$$treshold_{high} = htr \cdot \max(image)$$

   Následně se větší z práhu použije k získání pixelů, kde se hrana určitě vyskytuje a menším z prahů se určí polohy pixelů, které jsou kandidátem na hranu (tyto pixely pak budou dále zpracovány pomocí hystereze).

6. Hystereze — proces, který iteruje přes všechny pixely obrázku získaného v minulém kroku a přemění na detekované hrany všechny kandidátní pixely sousedící s pixelem hrany.

Technologie

Pro implementaci aplikace bude využit programovací jazyk *Python 3.9* (v jiných verzích může dojít k problému s kompatibilitou). Pro práci s obrazovými daty bude využita knihovna *openCV* a také knihovna pro efektivní práci s poli *numpy*.

Testování

K testování výsledného detektoru bude vytvořen samostatný skript, který přijme výstupní obrázek detektoru a jeho originál a zjistí, jak moc se dané obrázky liší a vypočte základní statistiky.

1.3 Rozhraní aplikace

Výsledkem práce je aplikace s ovládáním přes rozhraní příkazové řádky. Běh programu lze nakonfigurovat mnoha možnostmi, avšak i ve výchozím nastavení lze dosáhnout dobrých výsledků. Pro správné fungování potřebuje aplikace akorát cestu k souboru, který má zpracovat, zobrazení výsledků je popsáno v sekci *ostatní*.

Gaussovo rozmazní

Gaussovo rozmazání lze ručně nakonfigurovat pomocí dvou argumentů

- `--b-sigma B_SIGMA` — Přímé nastavnění odchylky *Gaussova filtru*
- `--b-sigma-c B_SIGMA_COEF` — Koeficient, kterým bude pronásobena dynamicky vypočítána sigma,

Detekce hran

K detekci hran se zde používá tzv. Sobelův filtr, který zjistí gradient každého pixelu ve směru x i y.

- `--rgb` — Aplikovat sobelův filtr na všechny barevné kanály, ne pouze na jeden kanál v odstínech šedi
- `--g-ksize G_KSIZE` — velikost jádra pro *Sobelův filtr*

Dvojité práhování

Pro dovjité práhovnání jsou důležité následující 2 parametry určující koeficienty k výpočtu obou prahů pomocí vztahů

- `--htr HTR` — Koeficient většího z prahů, samotný práh se pak vypočítá podle vztahu v sekci teorie - dvojité práhování
- `--ltr LTR` — Koeficient nižšího z prahů, samotný práh se pak vypočítá podle vztahu v sekci teorie - dvojité práhování

Ostatní

Ostatní možnosti pro nastavení typu a granularity výstupu, pojmenování souborů a optimálního zobrazování výsledků na obrazovku

- `--max-resolution WIDTH HEIGHT` — maximální rozlišení zobrazovaného obrázku
- `--output FILENAME`, `-o FILENAME` — jméno souboru s výstupem (bez přípon)
- `--save` — uložit výstup(y) do souboru
- `--show` — zobrazit výstup(y) na obrazovce
- `--step` — vytvořit výstup pro všechny fáze Cannyho detektoru hran,
- `--grid` — uložit výstup jednotlivých kroků Cannyho detektoru hran jako jeden výstup seřazen do mřížky
- `input` — soubor s obrázkem ke zpracování.

Rozdělení práce

- David Bayer — konfigurace programu, načítání argumentů, kroky 1–3 výpočtu.
- Michal Glos — efektnivní implementace kroků výpočtu 4, 5 a 6. Algoritmus a řizení testování.

2 David Bayer

2.1 Zdroje

V předchozí kapitole byla popsána Cannyho detekce hran. Tato kapitola se zabývá vlastní částí autora, tedy prvními třemi kroky výpočtu. Obecné informace o Cannyho detektoru hran byly čerpány z dokumentace ke knihovně *openCV* a z Wikipedie. Znalosti pro převod z RGB na šedotónový obraz byly čerpány z článku 1, o Gaussově filtru z článku 2 a k detekci hran z článku 3.

2.2 Konfigurace a argumenty programu

Pro parsování argumentů bude použit knihovna *argparse*. S danými parametry pak bude vytvořen objekt třídy `Canny`, který provede požadované operace nad vstupním souborem metodou `apply_canny`.

2.3 Převod z RGB na šedotónový obraz

Obraz v RGB lze na šedotónový obraz převést snadno průměrovou metodou. Hodnoty kanálů jsou jednoduše zprůměrovány (je třeba dávat pozor na přetečení datového typu při sčítání kanálů).

$$G = \frac{R + G + B}{3}$$

Nicméně tento přístup se nepoužívá, protože lidské oko není citlivé na všechny barvy stejně. Proto se používá metoda zohledňující váhy jednotlivých barev.

$$G = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Tyto hodnoty jsou pak implicitně normalizovány do intervalu $\langle 0, 1 \rangle$. V implementaci bude použita pro konverzi funkce `cvtColor` z knihovny *openCV*.

2.4 Gaussův filtr

Gaussův filtr je používán k rozmazání obrazu. Jeho koeficienty jsou počítány z rovnice normálního rozdělení $\mathcal{N}(\mu, \sigma^2)$. Velikost konvolučního jádra lze odvodit z vlastností normálního rozdělení. Abychom počítali s 99 % významných hodnot, potřebujeme velikost kernelu $6 \cdot \sigma$. V praxi se ale používají menší kernely, protože s většími odchylkami značně narůstá množství výpočtů pro spočtení konvoluce. Běžně se tak setkáme s kernely o velikosti 3x3, 5x5 nebo 7x7.

Matici konvolučního jádra lze získat pomocí funkce `getGaussianKernel` a konvoluci spočítat pomocí funkce `filter2D` z knihovny *openCV*.

2.5 Výpočet intenzit a směrů gradientů

K výpočtu intenzit a směrů gradientů je třeba nejprve aproximovat horizontální derivace G_x a vertikální derivace G_y. Toho je dosaženo konvolucí obrazu rozmazaného Gaussovým filtrem s některým z jader pro detekci hran. V této implementaci bude použit *Sobelův* operátor implementovaný v knihovně *openCV* funkcí `Sobel`.

2.6 Testovací skript

K testování výsledného programu byl vytvořen skript *compare.py*, který přijímá na vstupu výstupní obrázek detektoru hran a obrázek, na kterém byla detekce hran prováděna. Nejprve je porovnávaný obrázek s hranami převeden do šedotónového barevného prostoru a poté je zkonvertován na pole *bool* hodnot značících, zda se na dané pozici vyskytuje hrana či nikoliv. Ze zdrojového obrázku je třeba získat obrázek s referenčními hranami. Tento obrázek je generován podle algoritmu 1 tak, že nad zdrojovým obrázkem je spuštěno několik detektorů hran, z jejichž výstupů jsou dále spočteny obrázky reprezentující úrovně shody mezi jednotlivými detektory hran. Jako referenční obrázek je pak vybrána úroveň specifikována příznakem `-c, --compare-level`.

```
for y ← 0 to height do
    for x ← 0 to width do
        count ← 0

        for out ← in edge_detector_outputs do
            if out[y][x] > THRESHOLD then
                count ← count + 1
            end if
        end for

        ref_img[y][x]← count + compare_level ≥ len(edge_detector_outputs)
    end for
end for
```

Algorithm 1: Algoritmus pro generování referenčního obrázku.

3 Michal Glos

3.1 Zdroje

Průběh detekce hran pomocí cannyho algoritmu jsme již popsali v první sekci, zda však přikládám i články použity k jeho studiu - článek 1. a článek 2.

3.2 Potlačení ne-maximálních hodnot

Podle úhlů získaných z předešlého kroku určíme ke každé detekované hraně (pro každý pixel) normální vektor $v_{x,y}$, jehož prvky mohou nabývat pouze hodnot z intervalu $\langle -1, 1 \rangle$. Hodnotu každého z pixelů $p_{x,y}$ pak nastavíme na hodnotu 0, pokud $p_{x,y} < max(p_{[x,y]+v_{x,y}}, p_{[x,y]-v_{x,y}})$, kdy jsou celočíselné pozice pixelů $p_{[x,y]+v_{x,y}}$ a$p_{[x,y]-v_{x,y}}$ získány zaokrouhlením k nejbližšímu celému čílsu. Tím se detekované hrany zúží na šířku 1 pixelu.

3.3 Dvojité práhování

Pomocí koeficientů ltr a htr se získají absolutní hodnoty obou prahů. Následuje práhování obrázku z fáze 4, kdy pixelům s hodnotou vyšší, než je menší práh je přiřazena šedá barva a pixelům, jejichž hodnota přesahuje hodnotu vyššího prahu je přiřazena bílá barva.

3.4 Hystereze

Hystereze je implementována tak, že program iteruje přes všechny pixely obrázku a pokud narazí na šedivý pixel (viz dvojité práhování), podívá se na své sousedy. Pokud alespoň jeden ze sousedů má bílou barvu, pixel je také přebarven na bílo a je součástí finálních detekovaných hran. Protože se sporné pixely zabarvují ve směru iterace přes obrázek, může nastat případ, kdy bude nějaká kandidátní hrana v jiném směru, než ve kterém je procházen obrázek. Proto je tento proces spuštěn celkem čtyřikrát tak, aby prošel obrázkem ve všech možných směrech.

3.5 Efektivní implementace funkce pro výpočet Hausdorffovy vzdálenosti

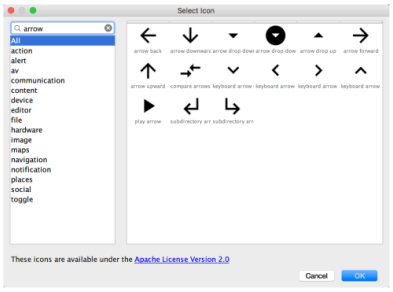
Testování probíhá ve funkci `distance2` , která vrací metriky používané pro ohodnocení úspěšností hranových detektorů. Obrázek s detekovanými hranami je testován vůči referenčnímu obrázku, kdy je mezi jejími hranami vypočítána tzv. Hausdorffova vzdálenost - maximální vzdálenost ze všech minimálních vzdáleností (pro každý pixel hrany) mezi hranami referenčními a testovanými. Výpočetní složitost tohoto algoritmu je relativně vysoká, proto byl kladen velký důraz na efektivní implementaci pomocí knihovny *numpy*.
Jako metriku korektnosti detekce hran jsme však nepoužili pouze Hausdorffovu vzdálenost, ale také nejbližší vzdálenost mezi chybně detekovanými pixely hran a referenčními hranami. Výstupem funkce pak je slovník, který obsahuje hodnoty pro Hausdorffovu vzdálenost, průměr nejkratší vzdálenosti chybných pixelů od pixelů referenčních, počet pixelů a směrodatná odchylka vzdálenosti. Tyto hodnoty jsou vypočítány v celkem třech variantách, a to pro všechny chybně detekované pixely, pro falešně detekované pixely a pro nedetekované pixely. Výsledky testů lze vidět v tabulce 4.

4 Vyhodnocení

Testování naší implementace Cannyho hranového detektoru probíhalo na 3 obrázcích 1 pomocí skriptu *compare.py*. Skript byl spouštěn s hodnotou compare level 2, s výjimkou vyhodnocení obrázku `windows.png` , který byl vyhodnocován oproti compare level 4 z důvodu velmi jednoduchých hran.

(a) lena.png, 512x512 px

(b) roots.png, 184x123 px



(c) windows.png, 492x361 px

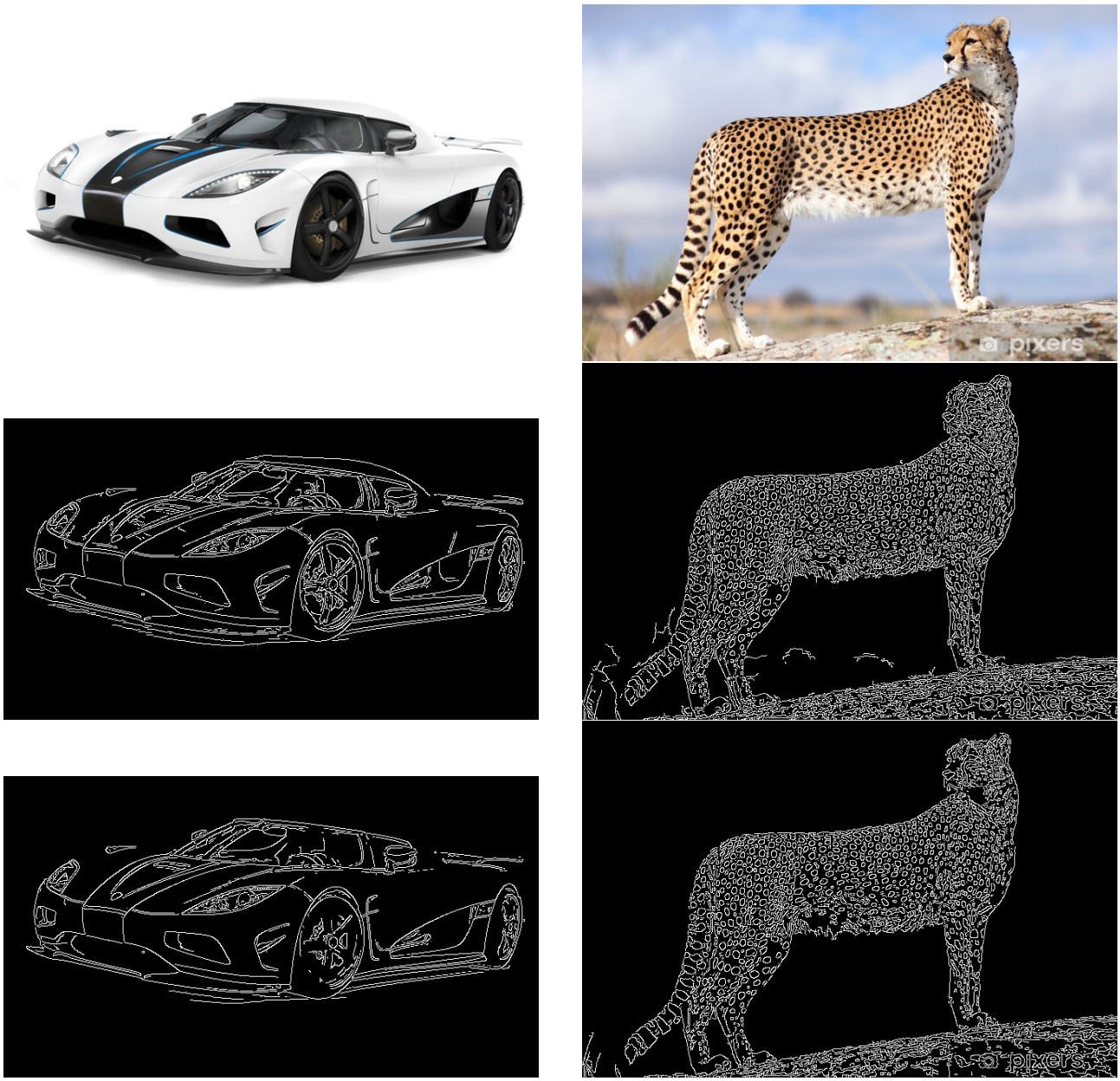
Obrázek 1: Testovací obrázky.

	Total				False negatives				False positives			
	%	avg	stdev	max	%	avg	stdev	max	%	avg	stdev	max
lena.png	12.4	2.30	4.30	100.12	8.2	1.21	0.47	16.12	4.2	4.47	6.91	100.12
roots.png	40.8	1.22	0.65	7.81	32.5	1.07	0.21	2.83	8.3	1.81	1.20	7.81
windows.png	3.4	10.05	17.53	115.43	0.9	1.01	0.08	2.24	3.4	12.45	19.02	115.43

Tabulka 1: Tabulka s výsledky porovnání

Z výsledků lze vyčíst, že referenční obrázek neobsahuje optimálně detekované hrany, dokonce ani jeho detekované hrany nemají šířku 1 pixel. Z toho důvodu si můžeme všimnout, že chybovost v procentech je relativně vysoká, především u obrázku s kořeny stromů, kde se nachází velké množství detekovaných hran.
Zajímavější metrikou jsou však Hausdorffovy vzdálenosti. Její maximální hodnoty jsou relativně vysoké, a to hlavně proto, že námi naimplementovaný detektor dokázal nalézt hrany, které se v referenčním obrázku vůbec nevyskytují — falešně pozitivní sloupec hodnot. Naopak referenční obrázek neobsahuje příliš vzdálené hrany — sloupec falešně negativní.

Ještě zajímavější metrikou je však rozložení této vzdálenosti, kde si můžeme všimnout, že se u všech obrázků velice liší. Obrázek s **kořeny** dosahuje v průměru nejlepších hodnot, a to především proto, že se v obrázku nachází velké množství hran, které nejsou 1 pixel široké. Tato skutečnost vede k tomu, že takové hrany pak mají velice blízko ke stejné hraně detekované našim *cannyho* detektorem, která je však pouze 1 pixel široká — *Hausdorffova* vzdálenost je minimální. U obrázků, který byl použit na úrovní porovnání 4, se však nevyskytují příliš široké hrany, proto je procento špatně detekovaných pixelů mnohem nižší. Rozložení vzdálenosti však ukazuje, že si náš detektor vedl v detekci hran opět lépe, maximální *Hausdorffova* vzdálenost dosahuje až 115 pixelů, což indikuje, že náš *cannyho* detektor dokázal nalézt velice vzdálené hrany, které se ani v referenčním obrázku nevyskytují. Obrázky **oken** a **kořenů** můžeme považovat za dva extrémní případy pro detekci hran, kdy obrázek s **okny** obsahuje velmi čisté a řídké hrany, které jsou pro detekci velice jednoduché, na rozdíl od obrázku s **kořeny**, který obsahuje velké množství neostrých hran, který je naopak pro detekci hran nevhodný. Jako poslední obrázek je známá fotka **Lena**, kterou můžeme vizuálně i podle výsledných hodnot považovat za obrázek zhruba průměrně náročný pro detekci hran. Na tomto obrázku se však také vyskytuje hrana, která nebyla detekována naší implementací *cannyho* detektoru (falešně negativní detekce) s nejvyšší *Hausdorffovou* vzdáleností.



Obrázek 2: Porovnání výsledků našeho detektoru hran (2. řádek) oproti již existujícímu detektoru z *OpenCV2* (3. řádek.)