Cannyev detektor rubova
Tehnička dokumentacija

Verzija <1.4>

Studentski tim: Šime Bašić
Petar-Šime Čepo
Igor Dodolović
Dražen Dostal
Stipe Grbić
Matija Gulić
Ivan Horvatin
Siniša Louč
Ivana Sučić

Nastavnik: Siniša Šegvić

Sadržaj

1. Uvod 3

2. Korištenje programskog okruženja cvsh 4

3. Cannyev detektor rubova 9

3.1 Detekcija rubova 9

3.2 Cannyev detektor rubova 9

3.3 Glađenje slike 9

3.4 Određivanje gradijenta intenziteta slike 12

3.5 Stanjivanje rubova (non-maximum suppression) 13

3.6 Usporedba s pragom 17

4. Rezultati 21

4.1 Primjer obrade slike 21

4.2 Kvaliteta i brzina 23

5. Zaključak 29

6. Literatura 30

1. Uvod

Detekcija rubova je fundamentalni postupak korišten u aplikacijama računalnog vida. Tim postupkom detektiraju se rubovi objekata, odnosno granice između objekata i pozadine na slici. Dobivene informacije koriste se u ekstrakciji i segmentaciji objekata.

Cilj projekta je razviti Cannyev algoritam za detekciju rubova. Kriteriji funkcionalnosti algoritma su dobra detekcija, dobra lokalizacija i minimalan višestruki odziv. Dobra detekcija kaže da bi algoritam trebao obilježiti što je moguće više stvarnih rubova koji zaista postoje na promatranoj slici, dobra lokalizacija je uvjet da su obilježeni rubovi što je moguće manje udaljeni od stvarnih rubova na slici, a minimalan višestruki odziv govori da bi rub trebao biti obilježen najviše jednom sa što manjom prisutnošću šuma. Algoritam se može podijeliti u četiri faze (glađenje slike, određivanje gradijenata intenziteta slike, stanjivanje rubova i usporedba s pragom) koje su objašnjene u trećem poglavlju.

Zbog velike vremenske složenosti algoritma odlučeno je da će se algoritam razvijati u programskom jeziku C++. Prilikom razvijanja i testiranja Cannyevog algoritma za detekciju rubova korišten je razvojni alat Visual C++ 2008 Express Edition [1].

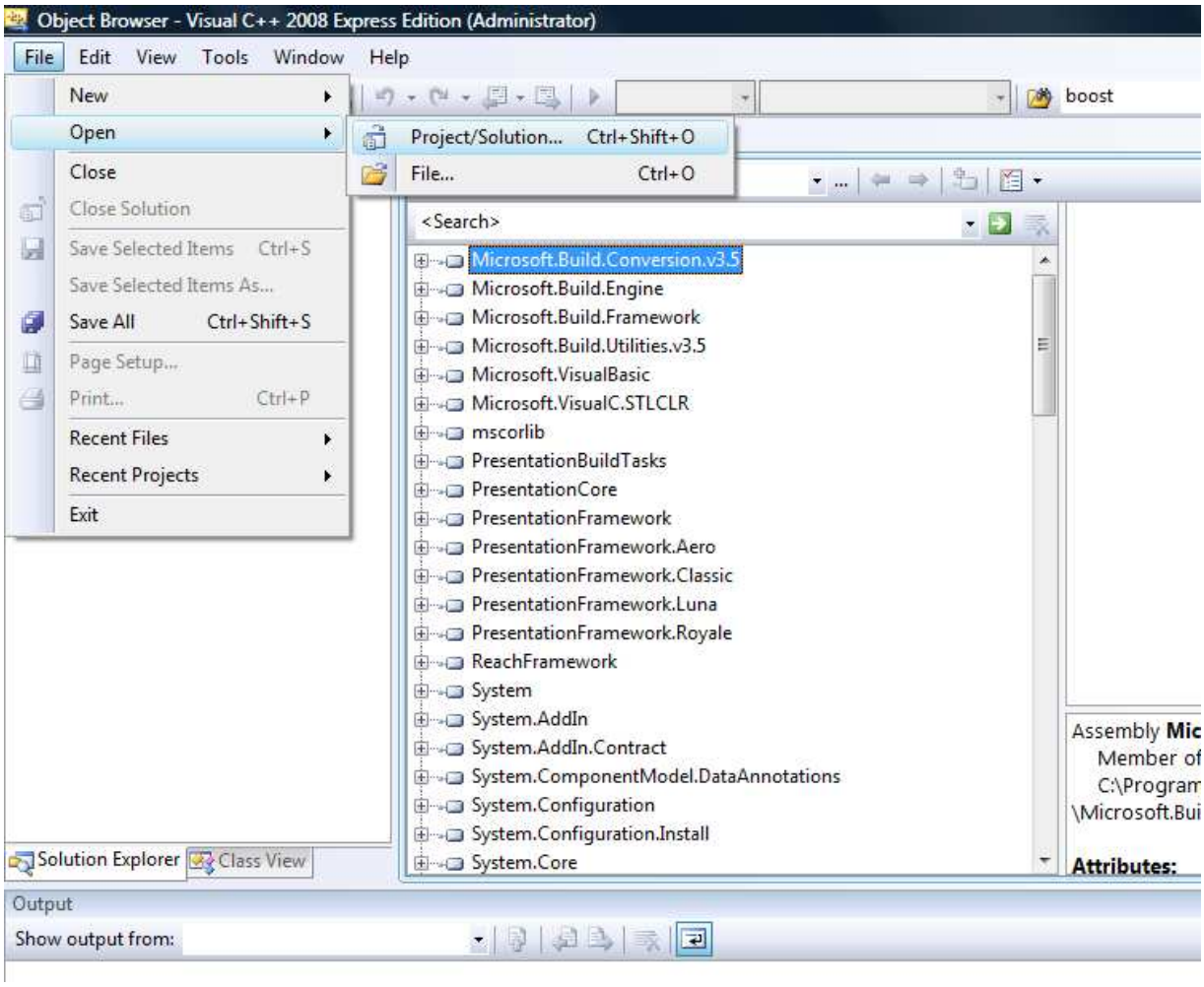
2. Korištenje programskog okruženja cvsh

Za jednostavno ispitivanje ispravnosti algoritma kojeg smo razvili trebalo je uspostaviti optimalnu integraciju sa programskim okruženjem (ljuskom) cvsh (engl. computer vision shell). Ljuska cvsh je razvijana na ZEMRIS-u s ciljem olakšanja eksperimentiranja s algoritmima računalnog vida. U glavnu funkcionalnost ljuske mogu se uvrstiti korisničko sučelje, pribavljanje slike te iscrtavanje slike. Ljuska je zamišljena tako da se korisnički postupci prevedu zajedno s ljuskom u jedinstvenu izvršnu datoteku. Lokacija predviđena za razvoj Cannyevog postupka u stablu izvornog kôda ljuske je direktorij `cvsh_src\ext\canny251`. Naš algoritam je definiran u okviru razreda `alg_canny251`. Povezivanje ljuske s korisničkim postupkom obavlja se preko virtualnog poziva metode `process`(), koja se poziva svaki put kad korisnik inicira obradu slike aktivnim algoritmom. Metoda `alg_canny251::process`() deklarirana je kao:

```
virtual void process (
    const img_vectorAbstract& src,
    const win_event_vectorAbstract& events,
    int msDayUtc);
```

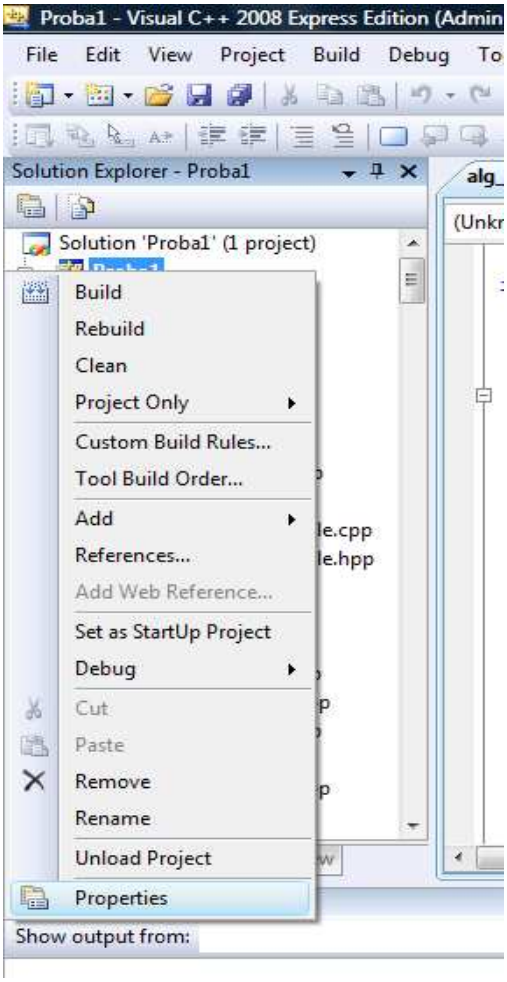
Prvi korak u prevođenju ljuske je kreiranje projekta u koji je potrebno dodati datoteke izvornog kôda ljuske (direktorij `cvsh_src`), ovaj dio je dodatno objašnjen u dokumentaciji prošlogodišnje grupe [13]. Za prevođenje ljuske potrebno je imati određene elemente biblioteke boost [2], kao i Windows Media Format SDK [3]. U izradi smo koristili WMF SDK 11. Biblioteke boost i Windows Media Format SDK je također potrebno spremiti na disk (npr., `C:\boost_1_36_0`, odnosno `C:\WMFSDK11`). U cilju testiranja ispravnosti algoritma nužno je odabrati određen broj slika (koristili smo format .bmp) te ih zatim pohraniti na disk (npr., `C:\slike`).

Postojeći projekt otvaramo pozivom File → Open → Project/Solution koristeći putanju do projekta (npr., `Desktop\New Folder\New Folder\Canny251\Canny251.sin`) (slika 2.1).



Slika 2.1 - Otvaranje projekta u okruženju Visual C++ 2008 Express Edition

Određene postavke bitne za ispravan rad ljuske se ne prenose kopiranjem projekta pa ih je potrebno ručno postaviti. Desnim klikom miša na projekt u Solution Explorer-u otvaramo padajući izbornik u kojem odabiremo opciju Properties (slika 2.2).

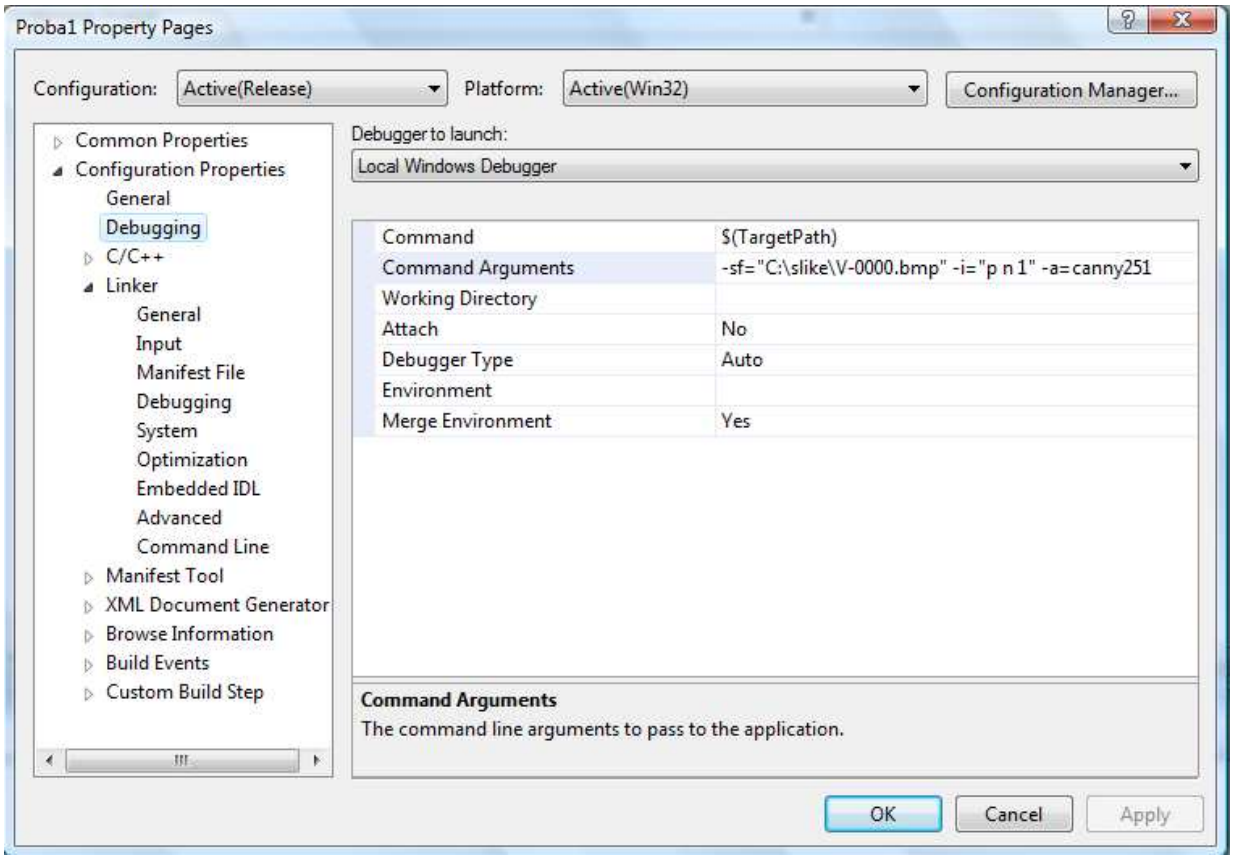


Slika 2.2 - Otvaranje kartice Properties

Za upisivanje parametara naredbenog retka u razvojno okruženje potrebno je pozicionirati se na Configuration Properties → Debugging → Command Arguments (slika 2.3) te tamo upisati:

-sf=”C:\slike\naziv_slike.bmp” -i=”p n 1” -a=canny251

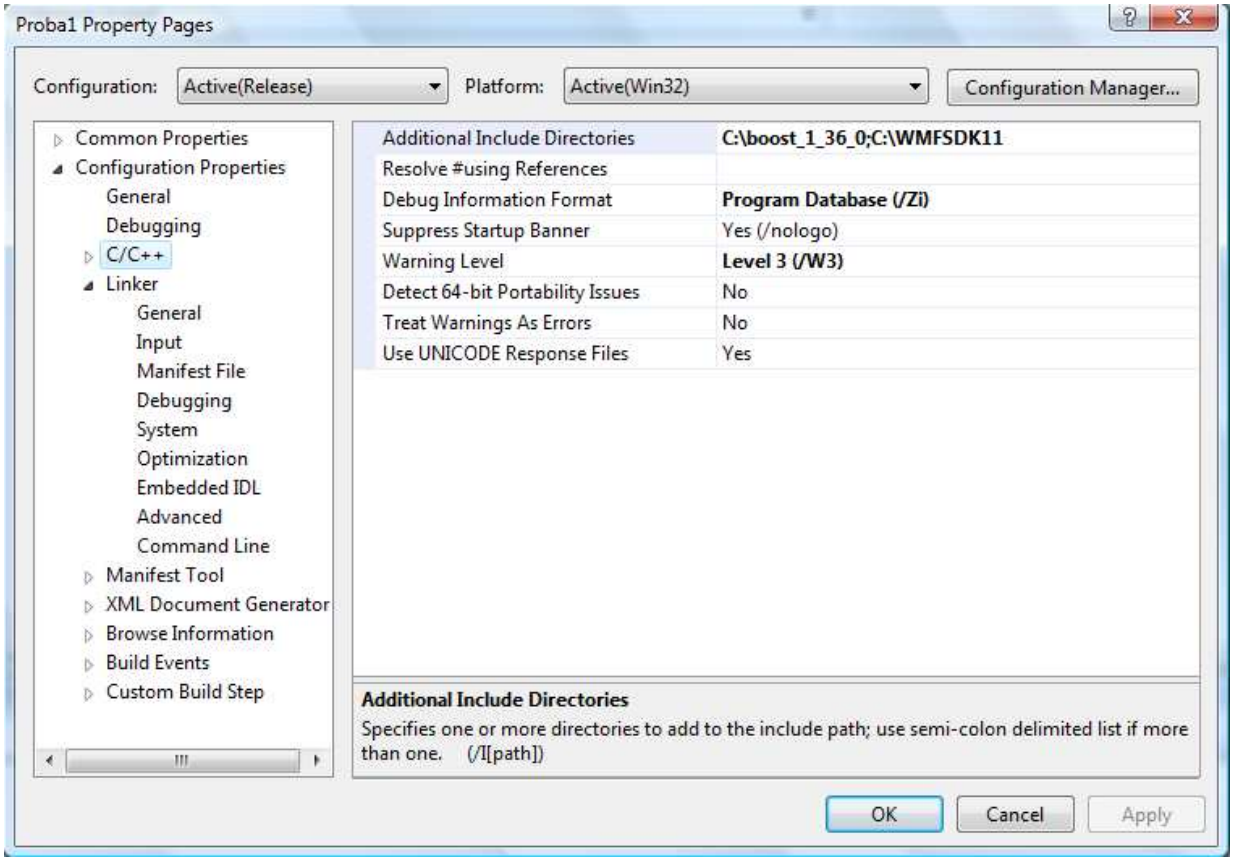
Naredba se sastoji od putanje (Path) izvorne datoteke slike (-sf), prve naredbe koja se prosljeđuje korisničkom sučelju (-i) te naziva algoritma koji se koristi (-a). Naziv algoritma povezuje se sa klasom `alg_base`, preko metoda `s_name()` i `name()`. Pomoću metode `alg_base::create`, ljuska stvara instancu pripadajućeg algoritma koju dalje koristi za obradu slike. Dio putanje `naziv_slike` treba promijeniti u naziv one slike koju ćemo koristiti za testiranje.



Slika 2.3 - Upisivanje parametara naredbenog retka (command line arguments)

Nakon toga se potrebno pozicionirati na Configuration Properties → C/C++ → General → Additional Include Directories (slika 2.4) i upisati putanje do sučeljnih datoteka (header-a, #include) biblioteka boost i WMFSDK:

C:\boost_1_36_0; C:\WMFSDK11



Slika 2.4 - Uključivanje Windows Media Format SDK i boost biblioteke u projekt

I na kraju još samo na Configuration Properties → Linker → Input → Additional Dependencies treba upisati datoteke s izvršnim kôdom korištenih biblioteka:

“C:\WMFSDK11\lib\wmvcore.lib” vfw32.lib

Biblioteka WMF koristi se za čitanje datoteka tipa .wmv, dok vfw32 služi za čitanje starijeg formata .avi.

3. Cannyev detektor rubova

3.1 Detekcija rubova

Detekcija rubova je jedan od fundamentalnih problema koji se javljaju pri računalnoj obradi slike. Rubovi su definirani kao 'nagle' promjene u intenzitetu susjednih piksela i predstavljaju granice objekata na slici, odnosno koriste se za detekciju položaja ili orijentacije tih objekata.

Za složenije slike, 'naglih' promjena može biti mnogo od kojih neke niti ne moraju biti stvarni rubovi. Zato detektori rubova moraju biti primjenljivi za različite razrede slika, moraju imati dobru preciznost (većina pravih rubova je i detektirana) i lokalizaciju (detektirani rubovi moraju biti što bliže stvarnim rubovima na slici) te minimalan višestruki odziv (svaki rub je detektiran samo jednom, nema tzv. 'lažnih rubova'). U povijesti je bilo mnogo pokušaja rješenja ovih oprečnih zahtjeva pa postoji mnogo metoda detekcije rubova. Jedna od češće korištenih metoda je Cannyev detektor rubova.

3.2 Cannyev detektor rubova

Cannyev detektor rubova jedan je od najkorištenijih i najuspješnijih algoritama detekcije rubova. Detektor je naziv dobio prema Johnu F. Cannyu koji je algoritam za detekciju razvio, a potom i predstavio u djelu *'A Computational Approach To Edge Detection'* 1986. godine [4].

Cannyev algoritam možemo razmatrati kao filtar koji ulaznu crno-bijelu sliku obrađuje na način da na izlaznoj slici postoje samo rubovi ili ne-rubovi. Detekcija rubova ovim algoritmom odvija se u nekoliko koraka pa je on podijeljen na četiri funkcionalne cjeline. To su glađenje slike, određivanje gradijenta intenziteta slike, stanjivanje rubova (non-maximum suppression) te usporedba s pragom.

3.3 Glađenje slike

Glađenje je postupak filtriranja slike koji kao rezultat, u odnosu na originalnu sliku, daje zamagljenu sliku smanjene detaljnosti. Glađenjem slike [5] rješavamo se štetnog šuma koji je sastavni dio slike što je bitno za kasniju detekciju rubova (eliminacija lažnih rubova). Šum se uglavnom sastoji od visokih frekvencija pa ga možemo ukloniti nekom vrstom linearnog

niskopropusnog filtra. Ovdje koristimo diskretno filtriranje, pri čemu na ulaznu funkciju (`y`) primjenjujemo filtar (`h`) te dobivamo novu izlaznu funkciju (`g`):

$$g(n) = \sum_{N=-\infty}^{+\infty} f(n-N)h(N)$$

Takav filtar je i Gaussov filtar kojeg smo koristili u ovom algoritmu. Pošto je Gaussova funkcija definirana na beskonačnom intervalu, u našem slučaju on je aproksimiran dvodimenzionalnom matricom (uz konačni interval i maleni gubitak preciznosti radi aproksimacije) popunjenom vrijednostima prema formuli (1). Gaussova 2D funkcija će osigurati određivanje vrijednosti svakog piksela na temelju njegovog susjedstva. Drugim riječima, svaki piksel koji svojim intenzitetom odskače od susjednih biti će „harmoniziran" s okolinom.

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

Gaussova 1D funkcija:
$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \quad (2)$$

gdje je: σ - standardna devijacija Gaussove funkcije, tj. njena rasprostranjenost
x - udaljenost od centralnog piksela po x osi
y - udaljenost od centralnog piksela po y osi

Budući da je gaussova 2D funkcija zapravo umnožak dviju Gaussovih 1D funkcija, glađenje slike može se, uz iste rezultate, izvršiti i Gausovim vektorom. Time se vrijeme izvođenja znatno skraćuje. Prolaz slike Gaussovim 2D filtrom je vremenske složenosti **O(n^2*p)**, dok za dvostruki prolaz Gaussovim 1D filtarom ona iznosi **O(2*n*p)**, gdje je **n** dimezija filtra, a **p** ukupan broj piksela slike. Daljni koraci glađenja slike biti će zato opisani korištenjem Gaussovog vektora koji je i implementiran u konačnoj verziji progama.

Gaussov vektor popunjen je vrijednostima prema Gaussovoj 1D funkciji (2). Duljinu vektora potrebno je odrediti tako da ona bude optimalna jer direktno utječe na kvalitetu i brzinu izvođenja glađenja slike. Drugi uvjet za duljinu vektora je da ona bude neparan broj zbog jednoznačnog određivanja centralnog piksela. Gaussova razdioba je definirana na beskonačnom intervalu, ali mi taj interval moramo ograničiti zbog postizanja veće brzine izvođenja. Zato koristimo „preciznost" od 3σ (3 sigma) koja nam osigurava točnost Gaussove razdiobe od 99.73% [6]. Pošto uzimamo obostrani interval, „zaokruženih" 3σ množimo sa dva i dodajemo

jedan što nam osigurava da je konačna duljina vektora neparan broj. Konačna formula izgleda ovako:

duljina vektora = 1 + 2*round(3*sigma) (3)

Dodatni uvjet je da zbroj svih vrijednosti unutar vektora iznosi jedan. To je nužni uvjet jer u protivnom originalna slika gubi (ili dobiva) na svjetlini zbog smanjenog (povećanog) ukupnog doprinosa izvorne slike. Da bi ukupan zbroj koeficijenata iznosio jedan, svaki element vektora se dijeli s ukupnim zbrojem starog vektora. Takav novi vektor se zove normalizirani vektor i koristi se u sljedećem koraku glađenja.
Funkcija koja stvara Gaussov vektor prema opisanom algoritmu je:

```
void makeGaussKernel (
    int szKernel,
    double sigma,
    std::vector<double>& vektor);
```

Funkcija kao parametre prima duljinu vektora (szKernel) prethodno izračunatu po formuli (3), standardnu devijaciju Gaussove funkcije σ te referencu na prazan vektor.

Sada sliku gladimo Gaussovim vektorom. Glađenje se vrši konvolucijom vektora sa slikom prolaskom vektora duž cijele slike vodoravno redak po redak počevši od donjeg lijevog kuta pa sve do gornjeg desnog. Konvoluciju je potrebno obaviti dva puta, jednom retčanim vektorom, a zatim stupčanim vektorom koristeći vodoravno zaglađenu sliku kao ulaz. U obzir moramo uzeti da Gaussov vektor nikada ne smije niti jednim svojim dijelom izaći van okvira slike. To postižemo uvođenjem margine koja označava dimenziju rubnog dijela slike unutar kojeg izlazne vrijednosti nisu definirane, pa ni ne trebaju biti izračunate. Veličina margina se računa po formuli:

margina = veličina vektora / 2 (4)

Funkcija koja vrši glađenje prema opisanom algoritmu je:

```
int filterGaussSeparatedDouble (
    double sigma,
    const img_wrap& imgGray,
    img_wrap& imgFirstPassSmooth,
    img_wrap& imgSmooth);
```

Funkcija vraća iznos margine prethodno izračunate unutar funkcije po formuli (4). Iznos margine nam je potreban za slijedeći korak Cannyevog algoritma detekcije rubova - "određivanje gradijenta" gdje se koristi margina, jednaka staroj vrijednosti uvećanoj za jedan. Funkcija preko parametara prima standardnu devijaciju σ, ulaznu crno-bijelu sliku koju funkcija izglađuje, privremenu sliku koja je rezultat prvog prolaza (konvolucije) Gaussovim vektorom te potpuno izglađenu sliku nakon drugog prolaza Gaussovim vektorom.

Bitno je naglasiti da se u ovom dijelu Cannyevog algoritma za detekciju rubova definiraju važni parametri za kvalitetu zaglađivanja i brzinu izvođenja. O tim parametrima kasnije ovisi i uspješnost Cannyevog algoritma. Iz komandne linije prilikom pokretanja Cannyevog algoritma, među ostalim, zadaje se i standardna devijacija σ. Na temelju tog parametra kasnije će biti izračunata veličina Gaussovog vektora prema formuli (3). Gaussov vektor s većim parametrom sigma smanjuje brzinu izvođenja algoritma te uzrokuje veću zamućenost izlazne slike, povećavajući tako grešku lokalizacije, ali i smanjujući utjecaj štetnog šuma. Za Gaussov vektor s manjim parametrom sigma vrijedi obratno. Veće su maske neprikladne za izvođenje u stvarnom vremenu (*real-time processing*). One su korisne jedino pri detektiranju većih, a glađih rubova, kao što su rubovi duge i sl.

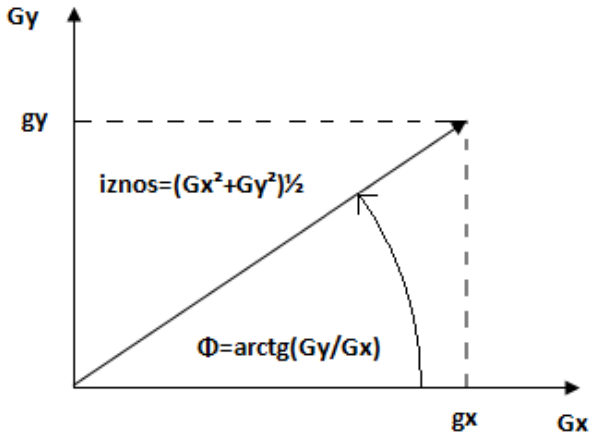
3.4 Određivanje gradijenta intenziteta slike

U ovoj fazi Cannyevog algoritma za svaki piksel određuje se iznos (amplituda) vektora gradijenta i pripadajući smjer pružanja (slika 3.1). Njoj je obvezno prethodila faza glađenja, koja je uklonila štetni šum koji bi u suprotnom još više došao do izražaja (zbog deriviranja). Pri izračunu iznosa gradijenta [7] koristimo podatke njegovih susjeda neposredno lijevo i desno, odnosno neposredno iznad i ispod promatranog piksela. Vodoravnu komponentu iznosa gradijenta (Gx) svakog piksela dobivamo oduzimanjem vrijednosti piksela njegovog desnog i lijevog susjeda. Okomitu komponentu iznosa gradijenta (Gy) dobijemo oduzimanjem vrijednosti piksela iznad i ispod promatranog piksela. Nakon izračuna obiju vrijednosti (Gx i Gy) izračunavamo iznos vektora gradijenta prema formuli:

$$G = [(Gx*Gx) + (Gy*Gy)] \hat{}(1/2)$$

Nakon toga potrebno je izračunati kut pružanja gradijenta za svaki piksel. Budući da su nam poznate obje komponente iznosa gradijenta, za izračun kuta u radijanima koristimo funkciju arkus tangens:

$$\Phi = atan2 (Gy,Gx)$$



Slika 3.1 - Iznos i smjer pružanja gradijenta

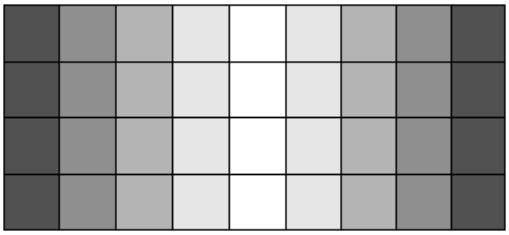
Funkcija koja određuje iznos i smjer pružanja gradijenta je:

```
int gradientDouble (
     int margin,
     const img_wrap& imgSmooth,
     img_wrap& imgGrad,
     img_wrap& imgPhi);
```

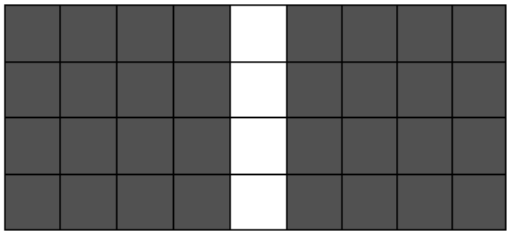
Funkcija vraća iznos margine koja je odmah na početku uvećana za jedan zbog toga što ćemo morati raditi sa neposrednim susjedima promatranih piksela pa ne smijemo uzimati u obzir piksele u kojima rezultat glađenja nije izračunat. Funkcija kao argumente prima 'stari' iznos margine koji se koristio u prethodnoj fazi, zaglađenu sliku iz prve faze te slike iznosa i kuta gradijenta u koje će funkcija zapisati izračunate iznose amplituda, odnosno kutova gradijenata svakog piksela na slici.

3.5 Stanjivanje rubova (non-maximum suppression)

Cilj ove faze Cannyevog algoritma je stanjiti rubove do veličine jednoga piksela (slika 3.3).



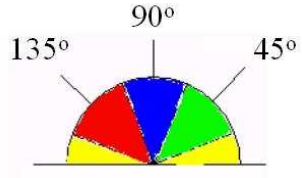
Slika 3.2 – Rub prije stanjivanja



Slika 3.3 - Stanjeni rub

Kako to postići? Prethodna faza računa iznos gradijenta i kut gradijenta u svakom pojedinom pikselu. Ta dva parametra predaje fazi stanjivanja, koja na temelju njih za svaki piksel određuje treba li ga poništiti ili ostaviti.

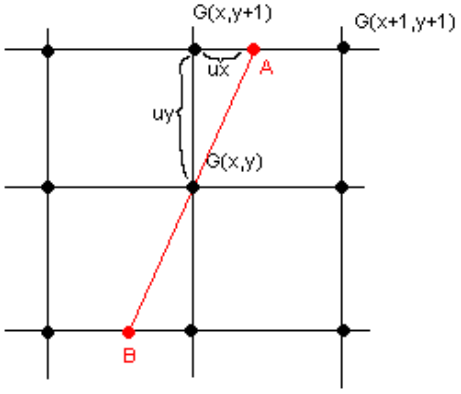
Jednostavan način kojim to postižemo je slijedeći: Ako je kut gradijenta promatranog piksela npr. 0º, to znači da se rub rasprostire pod kutom od 90º kao na slikama gore. Za svaki piksel tog ruba moramo provjeriti je li njegov gradijent veći od susjeda s lijeve i desne strane. Ako je veći od oba, radi se o maksimumu, a ako je barem jedan susjed veći, promatrani piksel moramo poništiti (staviti u nulu). Drugim riječima, svaki gradijent koji nije lokalni maksimum poništavamo, tj. postavljamo u nulu. Naravno da za drukčije kutove gledamo druge susjede, pa tako za kut gradijenta od 45º promatramo gornjeg desnog i donjeg lijevog susjeda, za 90º promatramo susjede iznad i ispod promatranog piksela, te za 135º promatramo gornjeg lijevog i donjeg desnog susjeda. Ova metoda zahtijeva da kutovi gradijenata budu jedan od 4 kuta: 0, 45, 90 ili 135. To se postiže tako da se svaki kut zaokruži na jedan od ova četiri (slika 3.4), i to onaj kojem je najbliži. Tako bismo kutove između 0 º i 22.5º zaokružili na 0º, između 22.5º i 67.5º na 45º, itd. (vidi sliku 3.4).



Slika 3.4 – Mogući kutovi gradijenta

Nešto složenija, ali i efikasnija metoda koju ćemo i koristiti jest interpolacija. Ovu metodu je koristio i sam J. Canny [8], a ta je metoda konačno i implementirana.

Uzmimo primjer sa slike 3.5. Promatramo središnji piksel. Jednostavnom metodom gledali bismo piksele iznad i ispod promatranog, dakle G(x,y+1) i G(x,y-1) (crvena linija označava pružanje gradijenta, ne ruba) i uspoređivali ih sa centralnim, i ako bi centralni bio veći od obojice ostavili bismo ga kakav je, inače ga poništavamo u nulu.



Slika 3.5 – Primjer korištenja interpolacije

Interpolacija neće taj kut zaokružiti na 90º i gledati iznad i ispod, nego će procijeniti iznose gradijenata u točkama A i B te će uspoređivati s njima. Radi lakšeg računanja možemo uzeti da je udaljenost između točaka 1, nije bitno koliko uzimamo jer je bitan samo omjer u_x i u_y. U primjeru gore u_y iznosi 1, a u_x poprima vrijednosti između 0 i 1. To nam daje četiri moguća slučaja:

1. phi < 45 **u_x =1, u_y računamo,**

2. 45 < phi < 90 **u_y =1, u_x računamo,**

3. 90 < phi < 135 **u_y =1, u_x računamo,**

4. 135 < phi **u_x =1, u_y računamo.**

Formula za računanje G_A u gornjem primjeru jest:

$$G_A = u_x * G(x+1,y+1) + (1- u_x) * G(x,y+1)$$

G_B nalazimo na dijametralno suprotnoj strani:

$$G_B = u_x * G(x-1,y-1) + (1- u_x) * G(x,y-1)$$

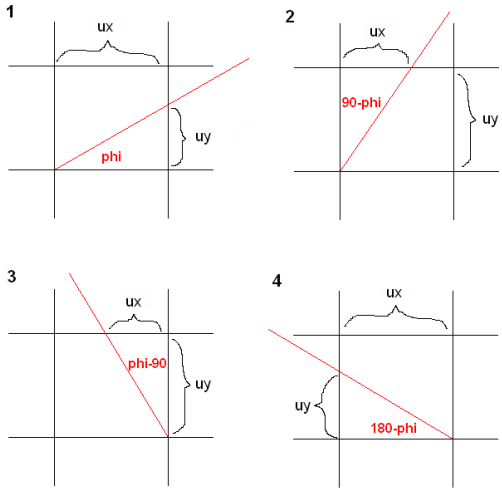
Kako naći u_x, odnosno u_y? Pomoću kuta gradijenta. U prvom slučaju, dakle kad je kut manji od 45º, tangens kuta računamo kao u_y / u_x. Pošto je u_x =1, možemo pisati u_y=tan(phi). Sa slike 3.6 lako odredimo formule za preostala tri slučaja. Formule glase:

1. phi < 45 **u_y =tan(phi)**

2. 45 < phi < 90 **u_x =tan(90-phi)**

3. 90 < phi < 135 **u_x =tan(phi-90)**

4. 135 < phi **u_y =tan(180-phi)**



Slika 3.6 – Primjer mogućih kutova gradijenta

Još samo treba pripaziti na koordinate točaka između kojih interpoliramo. U prvom slučaju to su G(x+1,y) i G(x+1,y+1), u drugom G(x+1,y+1) i G(x,y+1), u trećem G(x,y+1) i G(x-1,y+1) i u četvrtom G(x-1,y+1) i G(x-1,y).

Konačno, formule glase:

1. (phi < 45)

u_y =tan(phi)
$G_A = u_y * G(x+1,y+1) + (1- u_y) * G(x+1,y)$
$G_B = u_y * G(x-1,y-1) + (1- u_y) * G(x-1,y)$

2. (45 < phi < 90)

u_x =tan(90-phi)
$G_A = u_x * G(x+1,y+1) + (1- u_x) * G(x,y+1)$
$G_B = u_x * G(x-1,y-1) + (1- u_x) * G(x,y-1)$

3. (90 < phi < 135)

u_x =tan(phi-90)
$G_A = u_x * G(x-1,y+1) + (1- u_x) * G(x,y+1)$
$G_B = u_x * G(x+1,y-1) + (1- u_x) * G(x,y-1)$

4. (135 < phi)

u_y =tan(180-phi)
$G_A = u_y * G(x-1,y+1) + (1- u_y) * G(x-1,y)$
$G_B = u_y * G(x+1,y-1) + (1- u_y) * G(x+1,y)$

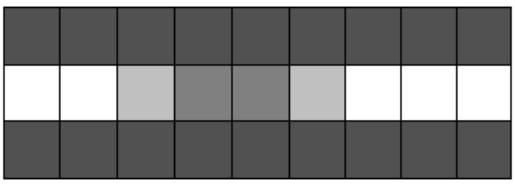
Ako G(x,y) nije veći i od G_A i od G_B onda se poništava u 0, a ako je veći od obojice, ostaje kakav je.

Kôd algoritma stanjivanja rubova je prilično jednostavan ako se razumije logika algoritma. Sastoji se od jedne klase `stanjivanjeDouble()` koja prima marginu, polje iznosa gradijenata i polje kutova gradijenata iz 2. faze. Margina je potrebna zbog ranije opisanih razloga, a polja iznosa i kutova gradijenata nam trebaju iz razloga navedenih u opisu algoritma. Marginu spremamo u varijablu margina, polje iznosa gradijenata spremamo u varijablu grad, a polje kutova gradijenata u varijablu *phi_grad*. Kutovi su izraženi u radijanima jer ih takve prima funkcija `tan()`. Polje *pBitsStanjeno* je pokazivač na polje piksela, a varijabla *width* sadrži širinu slike u pikselima.

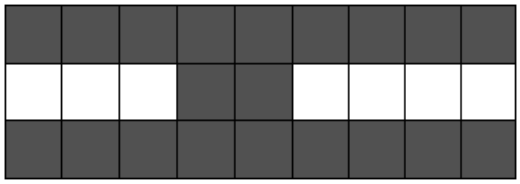
U dvostrukoj petlji prolazimo piksel po piksel i računamo G_A i G_B prema gorenavedenim formulama 1-4. Ako kut ne odgovara granicama potrebnim za formulu iz 1. slučaja, provjeravamo odgovara li granicama za formulu iz 2. slučaja, i tako dok ne ispitamo i posljednji slučaj. Funkciju `assert()` koristimo za provjeru da su u_x i u_y zaista u granicama (0,1). Kad smo izračunali G_A i G_B potrebno je još samo provjeriti da li je piksel koji promatramo maksimum ili ga je potrebno poništiti, i zatim se petlja vraća na početak i prima idući piksel.

3.6 Usporedba s pragom

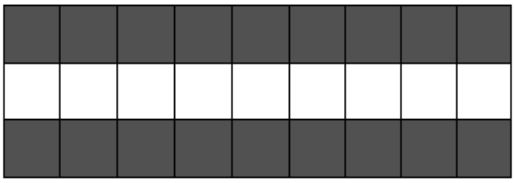
Završna faza u implementaciji Cannyevog algoritma je usporedba s dvostrukim (histereznim) pragom [9]. Ona služi za uklanjanje pruga. Pruge su prekidi u konturi ruba (slika 3.7.b), uzrokovani fluktuiranjem izlaza iznad i ispod praga. Ako se na sliku primijeni samo jedan prag vrijednosti jednake prosječnoj snazi ruba, tada će se zbog šuma pojaviti potonuća ruba ispod praga, jednako kao i proširenja iznad praga pa rub izgleda kao crtkana linija. Kako bi se to izbjeglo, Cannyev detektor koristi dva praga, gornji prag i donji prag.



Slika 3.7.a – Rub prije usporedbe s pragom



Slika 3.7.b – Mogući rezultat usporedbe s jednim pragom



Slika 3.7.c – Očekivani rezultat

Točke s vrijednostima većim od većeg praga su valjane rubne točke. Točke s vrijednostima manjim od manjeg praga nisu valjane rubne točke. Točke čija je vrijednost između dva praga idu u dodatno razmatranje, odnosno gledamo jesu li spojene s valjanim rubnim točkama i ukoliko jesu tada su i one valjane rubne točke.

Korištenje algoritma usporedbe s dvostrukim pragom je slijedeće:

- dva praga označimo sa dPrag i gPrag, gdje je dPrag < gPrag
- ako je vrijednost potencijalne točke ruba veća od gornjeg praga (gPrag) točka se odmah prihvaća kao rub
- ako je vrijednost potencijalne točke ruba manja od donjeg praga točka se odmah odbacuje kao potencijalni rub
- vrijednosti između postavljenih vrijednosti pragova se prihvaćaju ukoliko se dodiruju s pikselima koji su već prihvaćeni kao rub

Realizacija ovog algoritma sastoji se od četiri funkcije.

Prva funkcija deklarirana kao:

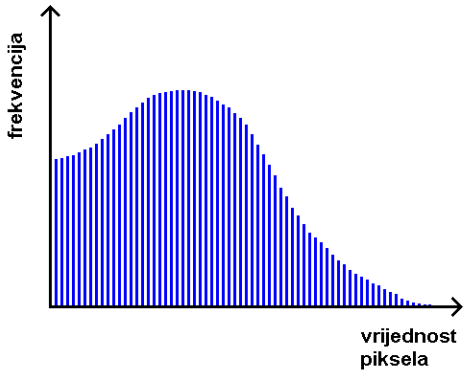
```
void usporedbaSaPragom(
  double th_Hi,
  double th_Lo,
  int margina,
  const img_wrap& imgSuppressed,
  img_wrap& imgThreshold);
```

za argumente prima stanjenu sliku i marginu iz prošle faze, postotak za gornji i donji prag te izlaznu sliku na kojoj će biti iscrtani samo pikseli koji zadovoljavaju uvjete pragova. Prvi korak u daljnjem radu je izračun pragova pa se u tu svrhu unutar funkcije `usporedbaSaPragom()` poziva funkcija za izračun pragova deklarirana kao

```
void izracunGranica(
  double th_Hi,
  double th_Lo,
  int margina,
  const img_wrap& imgSuppressed,
  double& thAbsHi,
  double& thAbsLo);
```

koja na temelju ulaznih postotaka gornjeg i donjeg praga izračunava apsolutne vrijednosti pragova.

Funkcija `izracunGranica()` za argument prima postotke donjeg i gornjeg praga, marginu iz faze stanjivanja, stanjenu sliku i pokazivače na dvije varijable u koje će spremiti apsolutne vrijednosti pragova. Ti apsolutni pragovi se računaju na temelju relativnih (zadanih) pragova, pri čemu zadani postotak govori koliko piksela od njihovog ukupnog broja ne uzimamo na obradu (npr. ako je zadana granica 90%, to znači da uzimamo samo 10% piksela s najvećim iznosom gradijenta). Zadatak funkcije je ispitati koje se sve vrijednosti piksela pojavljuju na slici i koliko ima piksela svake od tih vrijednosti, odnosno treba napraviti razdiobu frekvencije pojavljivanja piksela određenih vrijednosti na slici, tj. histogram:



Slika 3.8 – Primjer histograma

Koristeći te podatke preslikavamo postotke o pragovima u apsolutne vrijednosti pragova. Funkcija prvo prolazi po svim pikselima stanjene slike poštujući margine i istovremeno tražeći najveću vrijednost piksela. Nakon toga definira se polje cjelobrojnih podataka duljine jednake najvećoj vrijednosti piksela (histogram). Zatim se kroz petlju prolazi čitavom slikom te se gledaju vrijednosti svakog piksela. Polje (histogram) se puni na temelju učitane vrijednosti piksela tako da indeks polja koji odgovara toj vrijednosti uvećamo za jedan. Npr. kad učitamo vrijednost piksela 56, polje određeno indeksom 56 povećavamo za jedan. Tim postupkom dobivamo broj pojavljivanja svake vrijednosti piksela. Razlog ograničavanja duljine cjelobrojnog polja na najveću vrijednost piksela na slici jest izbjegavanje adresiranja nepostojećih piksela kad dođemo do piksela čija vrijednost nije u pretpostavljenim granicama. Sada znamo koje se vrijednosti pojavljuju na slici i koliko puta te se poziva funkcija `findThValFromThSum()` deklarirana kao:

```
int findThValFromThSum(
  std::vector<int>& histogram,
  double thSum,
  double g_max,
  double th_Rel);
```

Prvi put se poziva za izračun gornjeg, a drugi put za izračun donjeg apsolutnog praga. Funkcija prima histogram, ukupan broj piksela na slici umanjen za piksele izvan granica margine, maksimalnu vrijednost svih piksela na slici i odgovarajući postotak za granicu.

U funkciji petljom prolazimo po polju vrijednosti sadržanih na slici počevši od elementa sa indeksom 1 (ne od indeksa 0, jer je piksela sa iznosom gradijenta 0 jako puno pa oni „narušavaju statistiku“) postepeno dodajući vrijednosti elemenata polja novoj varijabli *zbroj* u svakoj iteraciji petlje. Petlja se nesmetano odvija sve dok varijabla *zbroj* nije veća od ukupnog broja svih piksela na slici umanjenog za piksele izvan margina i piksele sa vrijednošću nula te pomnoženog ulaznim parametrom za gornji, odnosno donji prag (*th_Rel*). Vrijednost varijable koja određuje redni broj iteracije petlje u trenutku uspješnog prekida petlje jednaka je apsolutnoj vrijednosti gornjeg, odnosno donjeg praga.

Nakon što su poznate apsolutne vrijednosti gornjeg i donjeg praga ulazi se u petlju kojom se prolazi po slici u granicama margine te se na temelju usporedbe vrijednosti piksela s pragovima stvara nova binarna slika. Uspoređuje se vrijednost piksela s većim pragom: ukoliko je vrijednost piksela veća od vrijednosti većeg praga te ukoliko piksel nema vrijednost 255 piksel na istoj poziciji u novoj slici (threshold) se postavlja na vrijednost 255.

Zatim se poziva četvrta, rekurzivna funkcija `rek()` deklarirana kao

```
void rek(
  int margina,
  int pozicija,
  double gGranica,
  double dGranica,
  const img_wrap& imgSuppressed,
  img_wrap& imgThreshold);
```

koja provjerava susjedstvo obrađenog piksela. Funkcija `rek()` za argument prima marginu, lokaciju prethodno obrađenog piksela, apsolutnu vrijednost gornjeg i donjeg praga, stanjenu sliku i izlaznu sliku. Funkcija `rek()` promatra susjedstvo obrađenog piksela te ukoliko je u susjedstvu pronađen piksel sa vrijednošću većom od donjeg praga tada on postaje dio ruba, vrijednost mu se postavlja na 255, a funkcija se rekurzivno poziva koristeći lokaciju tog istog piksela kao argument.

Kod rekurzivnog poziva funkcije `rek()` dolazilo je do problema beskonačne petlje. To se događalo kad imamo dva piksela čije vrijednosti su između donjeg i gornjeg praga, te diraju piksel koji je određen kao rub. Jedan od tih piksela će zvati funkciju `rek()` za susjedni piksel, a nakon toga će taj susjedni piksel zvati funkciju `rek()` za isti piksel koji je za njega pozivao funkciju `rek()`. Ovo se događa zato što ne mijenjamo izvornu sliku. Ovaj problem smo riješili na način da prije svakog poziva funkcije `rek()` provjerimo novonastalu sliku da li na njoj već postoji rub na poziciji za koju pozivamo funkciju `rek()`.

4. Rezultati

4.1 Primjer obrade slike

Ovo je primjer obrade slike algoritmom uz parametar sigma postavljen na jedan te granice postavljene na 95%, odnosno 70%. Nakon obrade iscrtava se 8 slika.

Prva slika (slika 4.1) je ulazna slika, a iscrtavamo ju kako bismo mogli usporediti ulaz i rezultat. Iduća slika (slika 4.2) je kopija ulazne slike, ali je crno-bijela (*grayscale*), jer tako olakšava postupke daljnje obrade.

Slika 4.1 – Ulazna slika

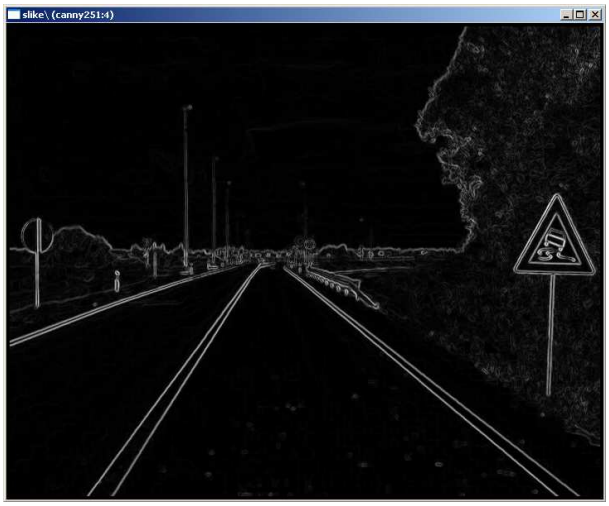
Slika 4.2 – Crno-bijela slika

Slijedeće dvije slike (slika 4.3 i slika 4.4) odnose se na fazu glađenja. Treća slika je rezultat prvog prolaza prethodne slike vertikalno orijentiranim Gaussovim vektorom, dok je četvrta slika rezultat drugog prolaza (preko treće slike) ali ovaj put horizontalno orijentiranim Gaussovim vektorom. Vidljive crne linije uz rub slike ukazuju na marginu.

Slika 4.3 – Prvi prolaz vektorom

Slika 4.4 – Drugi prolaz vektorom

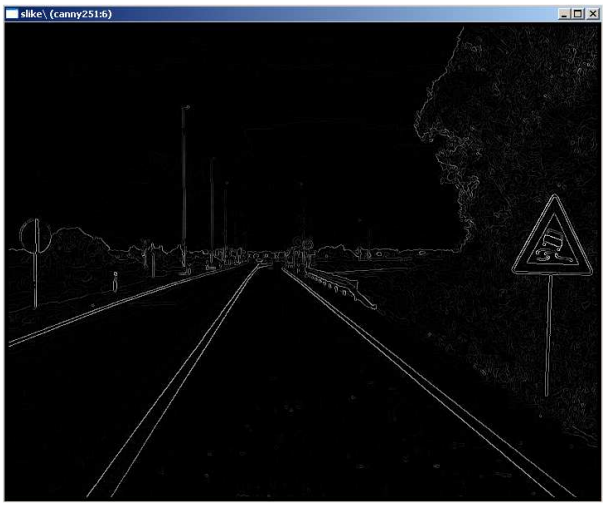
Iduće dvije slike (slika 4.5 i slika 4.6) su rezultat faze određivanja gradijenta intenziteta slike pri čemu peta slika predstavlja iznose gradijenata pojedinih piksela, a šesta grafički prikaz kutova gradijenata pojedinih piksela.



Slika 4.5 – Prikaz gradijenata slike

Slika 4.6 – Prikaz kutova gradijenata slike

Predzadnja slika (slika 4.7) je rezultat faze stanjivanja rubova, a dobiva se na temelju prethodne dvije slike. Tu su rubovi stanjeni na debljinu jednog piksela. Zadnja slika (slika 4.8) je završni rezultat algoritma. Ovdje pikseli poprimaju isključivo jednu od dvije vrijednosti, 0 ili 255, te se tako simulira binarna slika. Na temelju zadanih granica su „odabrani“ pravi rubovi.



Slika 4.7 – Stanjeni rubovi

Slika 4.8 – Konačni rezultat

Slanje sljedeće slike na obradu radi se upisom naredbe *process next* („p n“) u komandnu liniju ljuske. Ta naredba može primiti i broj slika koje želimo obraditi (npr. „p n 3“, će obraditi 3 slike). Također se mogu koristiti i *process previous* („p p“) i *process this* („p t“) naredbe koje na obradu šalju prethodnu sliku ili trenutnu ponovno. Moguća je izmjena parametara algoritma za vrijeme izvršavanja, za što koristimo naredbu *configure* („c“), nakon koje je potrebno upisati nove vrijednosti parametra sigme i vrijednosti granica.

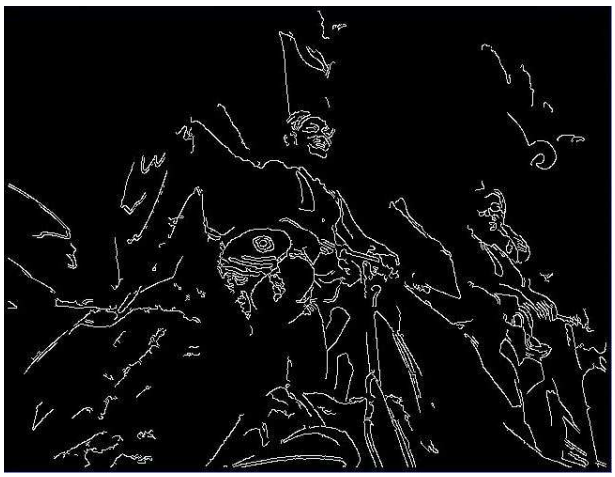
4.2 Kvaliteta i brzina

Dodatno testiranje je obavljeno na 3 ispitne slike (slika 4.9). Svrha testiranja jest pokazati ovisnost brzine obrade slike i same kvalitete prepoznavanja rubova o promjeni ulaznih parametara (sigma, gornja granica i donja granica). Odlučili smo koristiti zahtjevnije slike od onih na kojima smo testirali za vrijeme rada. Testirane su tri moguće vrijednosti parametra sigma (1, 2 i 3) u kombinaciji sa dvije moguće vrijednosti gornje granice (0.95, 0.80) i dvije moguće vrijednosti donje granice (0.70 i 0.50).

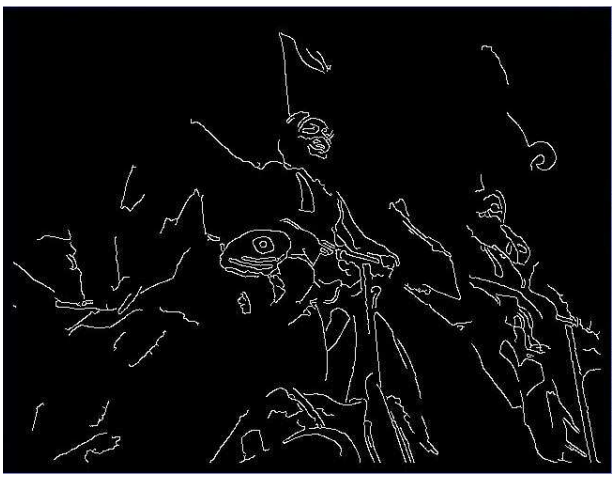
Tri ispitne slike su odabrane tako da budu što je moguće specifičnije. Prva slika je detaljna zbog velikog broja boja i šarenila. Druga je detaljna, ali nije šarena, rubovi su većinom naglašeni kontrastom crno-bijelo. Treća slika je zanimljiva zbog prevladavanja jedne te iste boje.

Slika 4.9 – Ispitne slike

Prva slika sa postavkama ulaznih parametara (1, 0.95, 0.70) izgleda ovako (slika 4.10.a):



Slika 4.10.a – Rezultati za parametre "1, 0.95, 0.70"



Slika 4.10.b - Rezultati za parametre "3, 0.95, 0.70"

Vidljivo je da poprilična količina rubova ostaje nedetektirana. Naravno, povećanjem parametra sigma sve više rubova ostaje izostavljeno (slika 4.10.b prikazuje za sigma=3).

Ovo se može ispraviti snižavanjem granica, tako za (1, 0.80, 0.50) dobivamo (slika 4.11):

Slika 4.11 - Rezultati za parametre "1, 0.80, 0.50"

Ovdje je vidljiva prevelika količina rubova na nekim dijelovima na kojima uopće ne bismo htjeli imati rubove (npr. krilo papige). Ovo je teško izbjeći jer čovjek subjektivno prosuđuje sliku i ne očekuje rubove na žutoj površini krila, a rubovi su detektirani zbog teksture. Odabir idealnih parametara ovisi od slike do slike, a ovisi i u koju svrhu koristimo algoritam (koliku detaljnost želimo). Vrijeme potrebno za obradu slike također ovisi o ulaznim parametrima. Za (1, 0.95, 0.70) vremena obrade pojedine faze iznose (obavljena su 3 mjerenja, rezultati su odvojeni zarezom):

glađenje: 63, 63, 62 ms
izračun gradijenta: 93, 93, 94 ms
stanjivanje: 15, 16, 16 ms
usporedba s pragom: 78, 78, 78 ms

Tri provedena mjerenja daju slijedeći prosjek trajanja pojedinih faza:

glađenje: 62.66 ms
izračun gradijenta: 93.33 ms
stanjivanje: 15.66 ms
usporedba s pragom: 78.00 ms

Za (1, 0.80, 0.50) vremena iznose (jedno mjerenje):

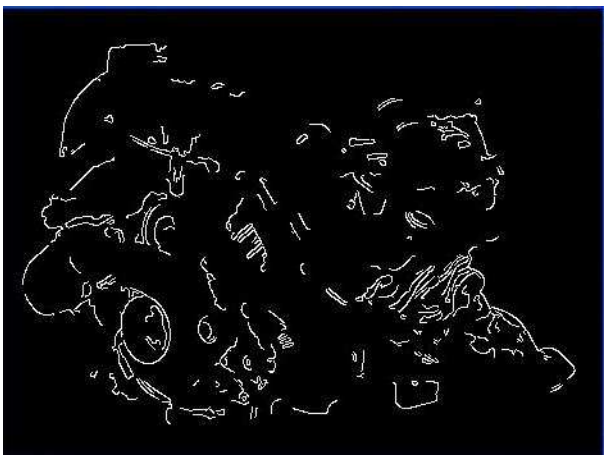
glađenje: 63 ms
izračun gradijenta: 94 ms
stanjivanje: 31 ms
usporedba s pragom: 78 ms

Prikazana vremena nisu fiksna. Ponovnim pokretanjem programa mogu se dobiti i drugačije vrijednosti. Vremena su ovdje dana samo radi predodžbe o tome koliko otprilike traju pojedine faze jedna u odnosu na drugu.

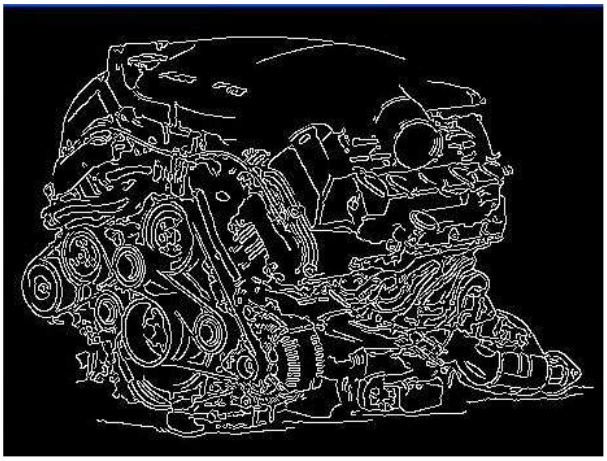
Promjenom vrijednosti gornje i donje granice ne mijenja se vrijeme trajanja usporedbe s pragom jer petlja ostaje ista, razlika je samo u tome što neki pikseli neće biti postavljeni u 0 nego u 255 i obratno.

Promjenom vrijednosti parametra sigma mijenja se vrijeme trajanja glađenja. Što on veći, duže traje obrada. To je zbog toga što veći parametar sigma rezultira većim vektorom Gaussovog glađenja (za detaljnije objašnjenje pogledati poglavlje 3.3)

Druga slika je zahtijevala drastično smanjenje granica. Za (1, 0.95, 0.70) rezultat je vrlo loš (slika 4.12).



4.12 - Rezultati za parametre "1, 0.80, 0.50"



Slika 4.13 Slika - Rezultati za parametre "1, 0.55, 0.30"

Ni sa (1, 0.80, 0.50) rezultat nije bio mnogo bolji. Tek sa (1, 0.55, 0.30) dobili smo prihvatljiv rezultat (slika 4.13).

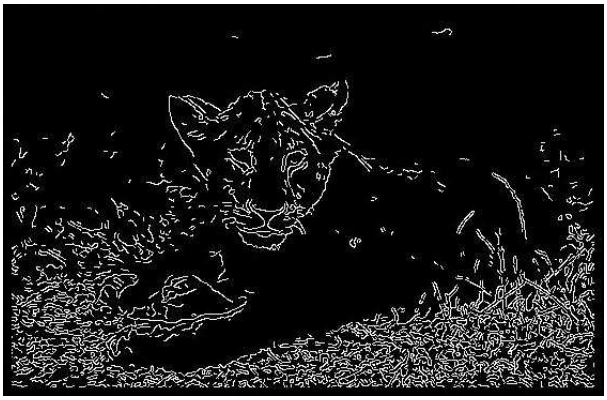
Za iste vrijednosti granica ostale dvije slike izgledaju iznimno loše. Kao što je rečeno ranije, svaka slika je priča za sebe i teško je naći univerzalne postavke.
Vremena su iznosila:

glađenje: 16 ms
izračun gradijenta: 46 ms
stanjivanje: 16 ms
usporedba s pragom: 16 ms

Nešto su manja nego kod prve slike jer je prva bila veće rezolucije.

Za očekivati je bilo da će kod treće slike biti najviše problema. Za (1, 0.95, 0.70) dobivamo vrlo mnogo detalja u travi, a rub tijela lava ostaje nedetektiran (slika 4.14).

Slika 4.14 - Rezultati za parametre "1, 0.95, 0.70"

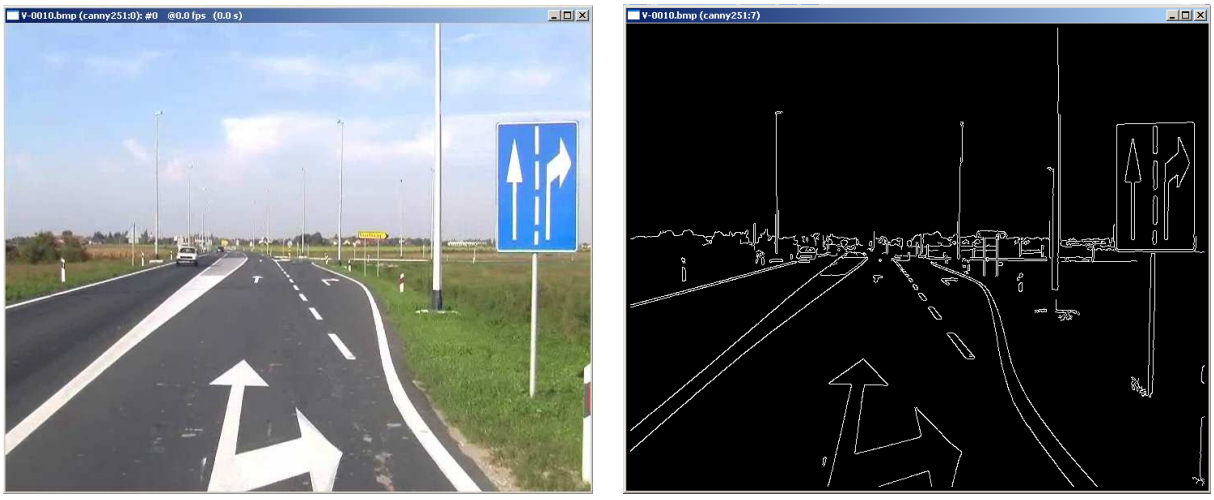


Slika 4.15 - Rezultati za parametre "3, 0.80, 0.70"

Za ovu sliku jednostavno nismo uspjeli pronaći kombinaciju parametara koja bi dala prihvatljiv rezultat. Propuštanjem sve većeg broja rubova (što se postiže smanjenjem parametra sigma i smanjivanjem granica) počinjemo dobivati rubove gdje ih ne želimo, dok tijelo i dalje nije pravilno obrubljeno. Slika 4.15 prikazuje obrađenu sliku za (3, 0.80, 0.70).

Cannyev algoritam ne funkcionira zadovoljavajuće za područja slike na kojima prevladava jedna boja kao što su npr. ljudska lica. Naravno, svaka slika je drugačija i za neke će postojati kombinacija parametara koja daje dobar rezultat, ali u općenitom slučaju ovakve slike nije poželjno obrađivati Cannyevim algoritmom.

Treba napomenuti i činjenicu da smo za testiranje odabrali tri prilično zahtjevne slike kako bi uočili određene mane i slabosti algoritma. Postoje i slučajevi kada radi gotovo savršeno, kao npr. slika 4.16.



Slika 4.16 – Primjer dobrih rezultata

5. Zaključak

Cannyev algoritam u praksi se koristi za detekciju rubova na jednostavnim slikama, kao na primjer za detekciju prometnih znakova na cestama. Na složenijim slikama gdje se rubovi subjektivno određuju (nekim ljudima je nešto rub, a drugima to isto nije), gdje je slika mutna, gdje prevladava jedna boja ili gdje je svjetlina jednakog intenziteta teško je jednoznačno odrediti rubove te donijeti objektivnu ocjenu uspješnosti algoritma.

Kao što je već objašnjeno, sav posao se obavlja u četiri faze. U fazi glađenja je u početku bio implementiran Gaussov filtar u obliku matrice ali je prilikom optimizacije zamijenjen vektorom radi veće brzine obrade slika. Izračun gradijenta se gotovo nije ni mijenjao od prve verzije. Faza stanjivanja rubova je bila tehnički najzahtjevnija. U prvoj verziji su se svi kutovi gradijenata zaokruživali na četiri vrijednosti (0°, 45°, 90° i 135°), ali to je u slijedećoj verziji zamijenjeno s već spomenutom interpolacijom radi kvalitetnijih rezultata. Usporedba s pragom je također bila optimizirana. Uklonjene su bespotrebne provjere i obilazak već obrađenih piksela u rekurzivnim pozivima.

Algoritam se temelji na tri parametra: standardnoj devijaciji, gornjem i donjem pragu. Ovisno o njihovim vrijednostima i tipu slike, algoritam će davati odgovarajuće rezultate. Povećanjem standardne devijacije uklonit će se više šuma, ali se isto tako mogu zanemariti neki manje izraženi rubovi. Gornja i donja granica definiraju koliko rub na izvornoj slici mora biti izražen da bi u konačnici postao rub. Najzahtjevniji dijelovi algoritma za izraditi (programiranje i logika iza njega) su bili stanjivanje rubova i usporedba s pragom, a njihova optimizacija za neke konkretne slučajeve opisana je u [17] i [18]. Kao što je već navedeno, ne postoji metoda koja bi donijela objektivnu ocjenu kvalitete algoritma. Jedino što smo mogli učiniti jest iznijeti svoju subjektivnu procjenu. Što se tiče kvalitete detekcije rubova, uz ispravne parametre (određene na temelju vrste slika koje se obrađuju, npr. slike s puno detalja, s malim razlikama intenziteta piksela itd.) iscrtavaju se svi željeni rubovi. Brzina algoritma ovisi o veličini slika. Ona se može poboljšati ako se umjesto Gaussovog filtra za glađenje koristi adaptivni filtar medijana [19]. Uz slike srednje veličine (npr. 720x576) algoritam prosječno obradi četiri slike u sekundi tako da bi za obradu u stvarnom vremenu trebalo ili smanjiti rezoluciju slika ili primijeniti druge metode. Metoda optimizacije ovisi o tipu slika koje se obrađuju i o tome što želimo dobiti kao rezultat obrade tih slika ovim algoritmom. Osim u prometu, Cannyev algoritam koristi se za detektiranje horizonta u detekciji projektila pomoću infracrvenog svjetla [14], određivanje granice obale i nadgledanje njezine promjene [15], detektiranje kozmičkih zraka [16].

6. Literatura

[1] http://www.microsoft.com/express/download/, 26.10.2008.

[2] boost_1_36_0.zip, *Boost C++ Libraries*, http://sourceforge.net/project/showfiles.php?group_id=7586&package_id..., 26.10.2008.

[3] Windows Media Format 11 SDK, *Windows Media*, http://msdn.microsoft.com/en-us/windowsmedia/bb190309.aspx, 26.10.2008.

[4] Canny, J., 1986., A Computational Approach to Edge Detection , *IEEE Trans. Pattern Analysis and Machine Intelligence*, 8:679-714

[5] Gaussian blur, *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Gaussian_blur, 20.10.2008.

[6] Standard deviation, *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Standard_deviation, 20.10.2008.

[7] Canny edge detector, *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Canny_edge_detector, 23.10.2008.

[8] J. F. Canny, 1983., *Finding edges and lines in images*, 04.11.2008.

[9] V.Papić, 2005.,*Obrada slika i računalni vid*, 23.10.2008.

[10] Agile software development, *Wikipedia, the free encyclopedia,* http://en.wikipedia.org/wiki/Agile_software_development, 22.10.2008.

[11] Pair programming, *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Pair_programming, 22.10.2008.

[12] Andrew Zisserman, Image enhancement and 2D Fourier transforms, 2003., *Two-dimensional Signal Analysis*, http://www.robots.ox.ac.uk/~az/lectures/sa/lect12.pdf, 23.10.2008.

[13] T.Babic, T.Lukinic, D.Kovac, K.Popovic, D.Rojkovic, M.Sverko, Tehnička dokumentacija prošlogodišnjeg projektnog tima, 15.1.2008., *Prepoznavanje znakova*, http://www.zemris.fer.hr/~ssegvic/project/0708/grupa56/dokument13.pdf, 24.10.2008.

[14] A Modified Canny Algorithm for Detecting Sky-Sea Line in Infrared Images, http://portal.acm.org/citation.cfm?id=1172965.1173537&coll=&dl=,30.12.2008

[15] Canny algorithm for modeling shoreline change, http://www.gisdevelopment.net/technology/sar/mm06_015abs.htm

[16] Detecting cosmic strings in the CMB with the Canny algorithm, http://www.iop.org/EJ/abstract/1475-7516/2008/04/015

[17] Canny optimization technique for electron microscope image colourization, http://www.ingentaconnect.com/content/bsc/jms/2008/00000232/00000002/art00015

[18] Statistical optimization of Canny edge detector for measurement of fine line patterns in SEM image, http://www.iop.org/EJ/abstract/0957-0233/16/2/021

[19] A method of edge detection based on improved canny algorithm for the lidar depth image, http://adsabs.harvard.edu/abs/2006SPIE.6419E..24X