

Rapport i faget
SIF 8066 - Datasyn

Segmentering av fargebilder



Rune M. Andersen
Trondheim, 06.05.2002

Oppgavebeskrivelse

Oppgaven går ut på å skrive et program som kjenner igjen og trekker ut segmenter i et fargebilde. Programmet tar inn et fargebilde, itererer oppgitt algoritme over bildet, trekker ut kantene, og fyller de "flate" områdene i bildet med entydige farger for å markere distinkte regioner.

Programmet er skrevet i Java2 og leser bilder i JPEG/PNG/GIF-format. Etter at algoritmen er ferdig kan bildene lagres som PNG-format. PNG er valgt fordi det er en open-source-algoritme og en ferdig PNG-encoder ble funnet på nettet.

Denne rapporten beskriver en del av koden bak programmet, og viser testresultater på fire utvalgte bilder.

Rapporten er del av øvingsopplegg i faget SIF 8066 - Datasyn, ved NTNU, våren 2003.

Gjennomføring

Koding av pseudokode

Jeg begynte tidlig i semesteret å programmere på segmenteringsprogrammet, og fant raskt ut at dette hadde vært en smart beslutning, ettersom det tok en del tid å leke med forskjellige variasjoner av den oppgitte algoritmen.

Pseudokode for segmentering:

```
Normalise colour image:
for each pixel position, P,
BEGIN
    if r**2+g**2+b**2 .gt. thresh
    then scale r, g, b so that max(r,g,b)=1
    else set r, g and b to 0
END
Iterate{
for each pixel position, P,
BEGIN
    v1=variance over a 3*3 patch
    v2=variance over a 5*5 patch
    if v2 .lt. v1+Tol
    then average the [r,g,b] at P with the rest of the 3*3 patch
END
}until no significant change
Trace around (insides of) walls of high variance to obtain regions
```

I starten brukte jeg algoritmen slik den var oppgitt, men dette syntes å gi varierende resultater. Ettersom man skulle beregne variansen i et lite område i bildet og glatte dersom variansen var lav, mente jeg man like gjerne kunne bruke variansen direkte til å finne kantene.

Etter en del prøving og feiling fant jeg de rette parametrene og det viste seg at den modifiserte utgaven fungerte bra.

Modifisert algoritme for kantdeteksjon:

```
public Bilde finnKanter(Bilde bilde, double toleranse, int storrelse) {
    Bilde retur = new Bilde(bilde.hentBredde(), bilde.hentHoyde());
    for (int x = 0; x < bilde.hentBredde(); x++) {
        for (int y = 0; y < bilde.hentHoyde(); y++) {
            double v1 = varianse(bilde, x, y, storrelse);
            if (v1 < toleranse) {
                retur.sett(x, y, Color.black);
            } else {
                retur.sett(x, y, Color.white);
            }
        }
    }
    return retur;
}
```

I denne koden er det brukt en metode som heter varianse(...). Koden for denne ser slik ut:

Javakode for varians(...):

```

public double varians(Bilde bilde, int x, int y, int storrelse) {
    int quad = storrelse / 2;
    double maximum = 0;
    double r=bilde.hent(x,y,'r');
    double g=bilde.hent(x,y,'g');
    double b=bilde.hent(x,y,'b');

    for (int i = -quad; i <= quad; i++) {
        for (int j = -quad; j <= quad; j++) {
            double max2 = Math.sqrt(
                Math.pow((r-bilde.hent(x+i,y+j,'r')),2)+
                Math.pow((g-bilde.hent(x+i,y+j,'g')),2)+
                Math.pow((b-bilde.hent(x+i,y+j,'b')),2));
            maximum = Math.max(maximum, max2);
        }
    }
    return maximum;
}

```

Jeg forstod oppgaven opprinnelig slik at man skulle beregne fargeratioen (dvs. R/G/B) og deretter finne differansen mellom to slike utregninger. Dersom forskjellen var over en viss terskel ville man ha funnet en kant i bildet. I stedet for ratio, beregner jeg avstanden (euklidisk avstand i fargekuben) mellom midtpunktet og alle punktene i maskeområdet. Hele tiden sørger jeg for å ta vare på den største av disse.

Ved å gjøre det på denne måten slapp jeg å håndtere svært små tall og jeg mente å kunne se at programmet skilte bedre mellom fargene.

Alternativt til å beregne avstand mellom midtpunkt og de resterende piksler i masken, kunne man ha beregnet totalt største avstand. I en maske på 3x3 piksler ville imidlertid denne varianten gitt $(9 \cdot 8) / 2 = 36$ beregninger i stedet for 9, og beregningstiden ville økt dramatisk.

Første gang jeg forsøkte å bruke euklidisk avstand gikk jeg i en stygg felle. Jeg beregnet $\sqrt{(R^2+G^2+B^2)_{\text{bilde1}} - \sqrt{(R^2+G^2+B^2)_{\text{bilde2}}}$, noe som gikk greit for de fleste bilder, men som absolutt ikke virket i bilder hvor man hadde segmenter som var helt røde, grønne eller blå. Tanken bak denne feilen var å bytte ratio med avstand, men i farten ble det gjort en forglemmelse som ikke ble avdekket før sent ut i arbeidet. Feilen er imidlertid rettet, og jeg oppnådde langt bedre kantfølsomhet etter rettingen. (Faktisk oppnådde jeg så mye bedre følsomhet at metoden med feil i noen tilfeller oppnådde bedre resultater enn den riktige). Algoritmen for normalisering av bildet ble implementert i henhold til oppgitt pseudokode:

Algoritme for normalisering:

```

public Bilde normaliser(Bilde bilde, double terskel) {
    Bilde retur = new Bilde(bilde.hentBredde(), bilde.hentHoyde());
    for (int x = 0; x < bilde.hentBredde(); x++) {
        for (int y = 0; y < bilde.hentHoyde(); y++) {
            int divisor;
            if ((
                Math.pow(bilde.hent(x, y, 'r'), 2) +
                Math.pow(bilde.hent(x, y, 'g'), 2) +
                Math.pow(bilde.hent(x, y, 'b'), 2)) > terskel) {
                divisor = Math.max(bilde.hent(x, y, 'r'),
                    Math.max(bilde.hent(x, y, 'g'),
                        bilde.hent(x, y, 'b')));
                retur.sett(x, y,
                    (bilde.hent(x, y, 'r') * 255) / divisor,
                    (bilde.hent(x, y, 'g') * 255) / divisor,
                    (bilde.hent(x, y, 'b') * 255) / divisor);
            } else {
                retur.sett(x, y, Color.black);
            }
        }
    }
    return retur;
}

```

Annen relevant koding

I tillegg til å implementere pseudokoden fra oppgaveteksten og boka, skrev jeg noen andre algoritmer som skulle hjelpe segmenteringen. En beskrivelse av disse følger.

Medianfilter

Selv om bildene var normaliserte, var de enda ikke helt klare for å segmenteres. For å jevne bildene, tok jeg i bruk et enkelt medianfilter. Dette flytter en maske med valgfri størrelse (typisk 3x3, 5x5 osv.) over bildet, og lager et nytt bilde hvor pikselet i midten av masken blir erstattet av det pikselet som er middelpikselet i masken. På sort/hvitt-bilder er det svært enkelt å beregne middelpikselet, ettersom man bare sorterer på fargeintensitet, og tar det midterste pikselet blant de sorterte. I fargebilder er denne operasjonen mer kompleks.

Jeg forsøkte først å finne middelpikselet for hvert av R-, G- og B-lagene i bildet. Det vil si å bruke sort/hvitt-metoden på hver fargekomponent og konstruere et nytt piksel som består av de tre komponentene. (Nytt piksel konstrueres der $R = \text{median}(\text{RødtBilde})$, $G = \text{median}(\text{GrøntBilde})$ og $B = \text{median}(\text{BlåttBilde})$).

Deretter forsøkte jeg å sortere pikslene etter gjennomsnittverdien av R-, G- og B-komponentene, altså å gjøre bildet til gråtoner og sortere etter intensitet.

Til sist forsøkte jeg å legge fargekuben ut lineært ved å sortere etter $R*100+G*10+B$. Hvilken faktor man bruker er uvesentlig. Det som betyr noe her er at det sorteres først etter rødt, deretter etter grønt og til sist etter blått.

Konklusjonen ble at medianen av hver enkelt komponent fungerte best. Dette ga minst støy i bildet og produserte de beste totalresultatene.

Algoritme for medianfilter:

```
public Bilde median(Bilde bilde, int storrelse) {
    Bilde retur = new Bilde(bilde.hentBredde(), bilde.hentHoyde());
    for (int x = 0; x < bilde.hentBredde(); x++) {
        for (int y = 0; y < bilde.hentHoyde(); y++) {
            retur.sett(x, y,
                pixelMedian(bilde, x, y, storrelse, 'r'),
                pixelMedian(bilde, x, y, storrelse, 'g'),
                pixelMedian(bilde, x, y, storrelse, 'b'));
        }
    }
    return retur;
}
```

Hjelpealgoritme for medianfilter:

```

public int pixelMedian(Bilde bilde, int x, int y, int storrelse, char farge) {
    int quad = storrelse / 2;
    int[] liste1 = new int[storrelse * storrelse];
    // Leser alle fargeverdiene i området
    int teller = 0;
    for (int i = -quad; i <= quad; i++) {
        for (int j = -quad; j <= quad; j++) {
            int a = bilde.hent(x + i, y + j, farge);
            liste1[teller++] = a;
        }
    }
    // Lager en ny liste som er sortert
    int[] liste2 = new int[liste1.length];
    for (int i = 0; i < liste1.length; i++) {
        int minste = liste1[0];
        for (int j = 0; j < liste1.length; j++) {
            if (liste1[j] < minste) {
                minste = liste1[j];
                index = j;
            }
        }
        liste2[i] = minste;
        liste1[index] = 9999;
    }
    // Returnerer det midterste elementet
    return (liste2[(liste2.length) / 2]);
}

```

Flood fill

Også til flood fill forsøkte jeg forskjellige algoritmer.

Den første baserte seg på rekursive kall til dem av de fire naboene som enda ikke var markert. Hver nabo ville så kalle sine umarkerte naboer og det hele ville spre seg på samme måte som bakterievekst. Den rekursive måten ble imidlertid for tung og ressurskrevende, og fikk etter noen sekunders kjøring stackfeil fordi den brukte opp alt minnet.

Etterpå forsøkte jeg å lage en veggfølgende robot som malte gulvet der den gikk. Denne fungerte greit, men jeg fikk vanskeligheter med at den malte seg inne dersom en vegg hadde et en piksel bredt hull. Hvis dette skulle unngås måtte roboten tillates å kunne male over et allerede malt område, men den ville da bare lage en enkelt stripe langs "veggen" i bildet, og ikke trekke lengre innover i bildet, slik jeg hadde håpet. For at den skulle fylle hele rommet måtte den settes i gang igjen og igjen, og hver gang ville den lage en ny stripe, helt til området var fylt. Dette tok svært lang tid og ble oppgitt.

Neste forsøk gikk ut på å benytte firenaboprinsippet ved å skanne over bildet mange ganger. Hver gang skannpunktet kom til et umerket piksel som hadde en markert nabo, ble det umerkede pikselet markert med naboens farge. Metoden var ganske enkel, og fungerte hvert gang. Eneste ulempen var at den brukte lang tid på å fullføre et bilde, særlig hvis bildet bestod av smale ganger mellom de markerte områdene.

Lenge etter at jeg hadde slått meg til ro med den trege, men dog fungerende, metoden fikk jeg en ny idé. I stedet for at jeg kalte naboer rekursivt, la jeg dem nå i en kø. Metoden gikk i en løkke hvor den plukket det fremste elementet, fargela det og puttet dets ufargelagte naboer bakerst i køen. Til slutt ville man ikke finne flere piksler å legge til i køen, og det angitte området ville da vært fylt.

Denne metoden fungerte svært bra og kunne fargelegge bildene i løpet av 0,5 sekunder, i motsetning til ca. 40 sekunder som de tregeste metodene brukte.

Algoritme for flood fill:

```
public Bilde fyllRegioner(Bilde bilde) {
    Bilde retur = new Bilde(bilde.hentBredde(), bilde.hentHoyde());

    // Kopierer bildet over til returbildet
    for (int x = 0; x < bilde.hentBredde(); x++) {
        for (int y = 0; y < bilde.hentHoyde(); y++) {
            retur.sett(x, y, bilde.hent(x, y));
        }
    }

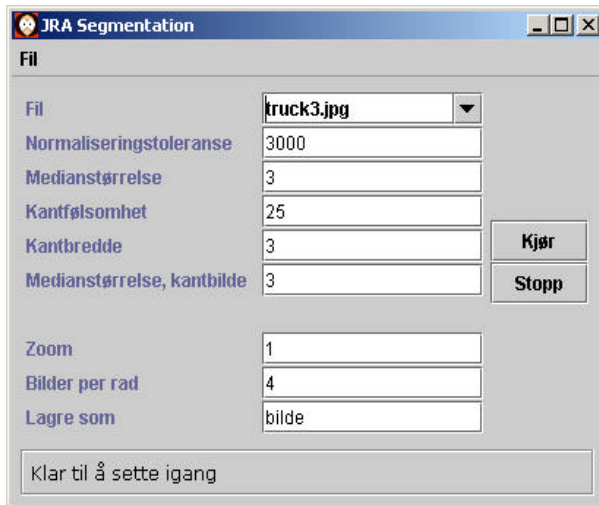
    int teller = 0;
    while (true) { //teller gir antall regioner
        Color farge = new Color(
            random.nextInt(254),
            random.nextInt(254),
            random.nextInt(254)); // Ny tilfeldig farge
    }
    // Finner neste sorte (dvs. ufargelagte) punkt
    Posisjon neste = nesteSorte(retur);
    if (neste == null) {
        break;
    }
    ko.add(neste);
    // Itererer over køen for den gitte fargen
    while (ko.size() > 0) {
        fyll(retur, farge);
    }
    teller++;
    return retur;
}
```

Hjelpealgoritme for flood fill:

```
public void fyll(Bilde bilde, Color farge) {
    int x = ((Posisjon) ko.get(0)).x;
    int y = ((Posisjon) ko.get(0)).y;
    if (bilde.hent(x, y - 1).equals(Color.black)) {
        bilde.sett(x, y - 1, farge);
        ko.add(new Posisjon(x, y - 1));
    }
    if (bilde.hent(x + 1, y).equals(Color.black)) {
        bilde.sett(x + 1, y, farge);
        ko.add(new Posisjon(x + 1, y));
    }
    if (bilde.hent(x, y + 1).equals(Color.black)) {
        bilde.sett(x, y + 1, farge);
        ko.add(new Posisjon(x, y + 1));
    }
    if (bilde.hent(x - 1, y).equals(Color.black)) {
        bilde.sett(x - 1, y, farge);
        ko.add(new Posisjon(x - 1, y));
    }
    ko.remove(0);
}
```

Grensesnitt

Etter at algoritmene var på plass begynte jeg å finpusse brukergrensesnittet. Jeg lagde et enkelte kontrollpanel hvor man kunne laste inn en del forhåndsdefinerte bilder, stille parameterverdier og starte/stoppe algoritmene. Etterhvert la jeg også til mulighet for å hente inn egne bilder.



Figur 1: Brukergransesnittet

Ved å høyreklikke i de genererte bildene vil programmet automatisk lagre dem i pathen programmet ble kjørt fra.

Bildene lagres i PNG-format, og encoderen har jeg hentet på nettet. Java2 har delvis støtte for JPEG, men PNG er et fritt format, og mange gratisencodere for forskjellige språk er tilgjengelig. Modulen jeg brukte består av kun to kildekodefiler som jeg kompilerer sammen med resten av prosjektet.

Encode bilder i PNG-format:

```
PngEncoderB pb = new PngEncoderB(bilde.bi); // bi er et BufferedImage i bildeklassen min
pb.setCompressionLevel(1);
byte[] pngbytes;

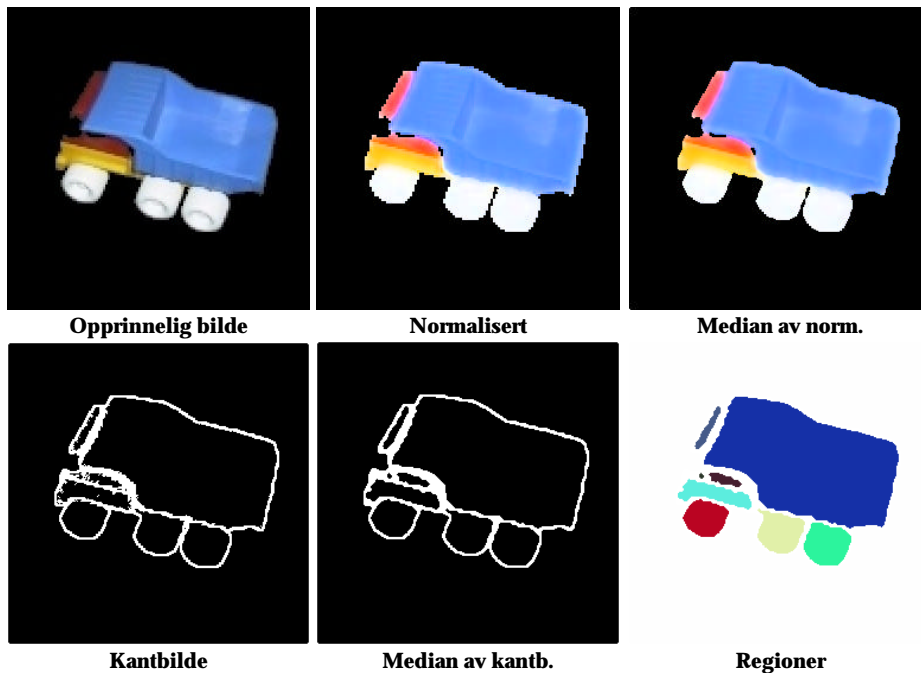
try {
    FileOutputStream outfile = new FileOutputStream("bilde.png");
    pngbytes = pb.pngEncode();
    if (pngbytes == null) {
        System.out.println("Null image");
    } else {
        outfile.write(pngbytes);
    }
    outfile.flush();
    outfile.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

Testing og resultater

En del bilder ble forsøkt segmentert med varierende parametre. Dette avsnittet vil vise inputbilder, hvert steg i prosessen og resultatbildet. Parametrene vil også bli listet opp, men med mindre noe annet er oppgitt vil normaliseringstoleransen være 3000. Denne trenger kun forandres når bildene blir svært mørke. 3000 er forøvrig ingen magisk verdi, den ligger bare et godt stykke over hva man kan regne med å få bruk for.

truck3.jpg

truck3.jpg er bildet som er vist som eksempel på fagets øvelsessider. Ettersom bildet var svært lite i utgangspunktet har jeg doblet størrelsen ved hjelp av "nærmeste nabo utvidelse", det vil si at hvert enkelt piksel ganske enkelt er doblet i størrelse.



Som man ser av bildene er segmenteringen ganske bra. Det har oppstått en liten feil ved førerhuset på lastebilden, fordi et vindu bryter med regelmessigheten i bildet. Kantbildet er preget av litt støy, men det meste av denne er blitt borte etter å ha kjørt medianfilteret over bildet.

Parametrene:

Medianstørrelse: 3

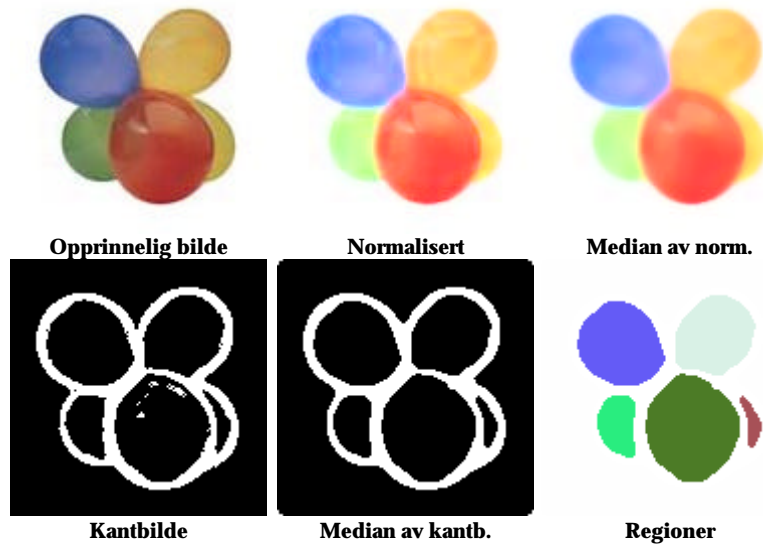
Kantfølsomhet: 28

Kantbredde: 3

Medianstørrelse, kantbilde: 3

balloon.jpg

Dette bildet viser en knute med ballonger, og bildet inneholder en del refleksjoner fra ballongenes glatte overflate. Resultatet blir likevel godt hvis man bruker de rette parametrene.



Etter at kantene er markert er det kommet frem en del falske kanter som følge av refleksjonene. Medianfilteret greier imidlertid å fjerne disse på en tilfredsstillende måte. En alternativ metode for å fjerne de minste falske kantene er å sjekke om de er del av en omsluttende figur. Dersom de ikke omslutter et sort område kan man regne med at det ikke er en virkelig kant og slette dem fra bildet. En slik algoritme er imidlertid ikke implementert i denne oppgaven.

Parametrene:

Medianstørrelse: 7

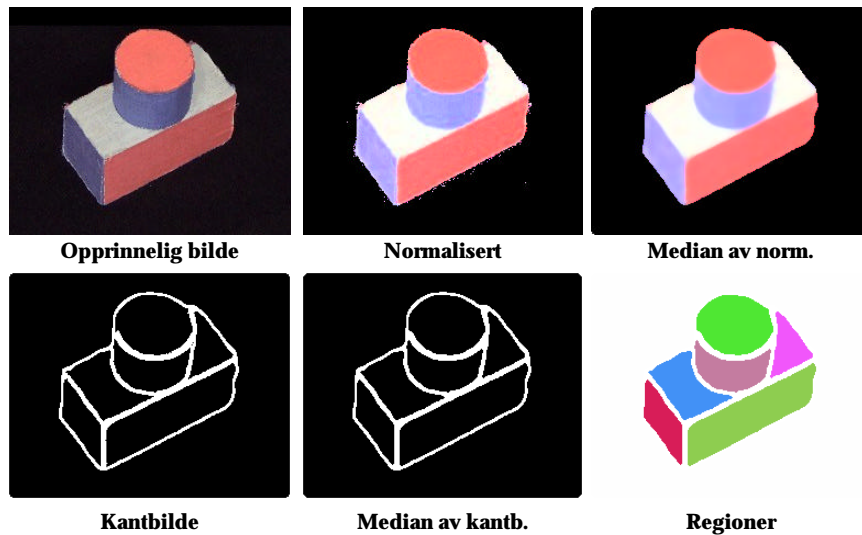
Kantfølsomhet: 20

Kantbredde: 3

Medianstørrelse, kantbilde: 5

block.jpg

Bildet er en del av testbildepakken som er satt sammen for faget. Figuren er enkel og man forventer et godt resultat.



Som resultatet viser er bildet godt segmentert og vi har oppnådd jevne, fine kanter. Til disse bildene ble det brukt et medianfilter med størrelse 7 for å glatte det normaliserte bildet, men det holder strengt tatt med 3. Bildet blir like godt segmentert, men kantene blir noe mer hakkete.

Parametrene:

Medianstørrelse: 7

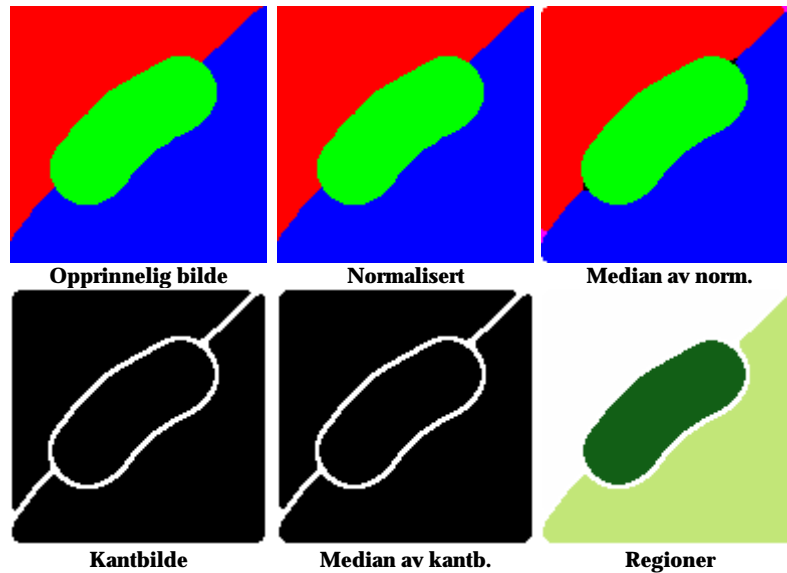
Kantfølsomhet: 20

Kantbredde: 3

Medianstørrelse, kantbilde: 3

rgb.png

Dette siste bildet er med for å illustrere at avstandsberegningen (mellom to punkter i fargekuben) er korrekt, samt at bildet kan laste PNG-bilder. Bildet inneholder tre distinkte regioner, hvor hver av disse består av kun én grunnfarge.



Av regionbildet ser man at delen øverst til venstre er hvit, mens de to andre delene er fargelagte. Grunnen til dette er at programmet antar at den første regionen er bakgrunnen, og bakgrunnen farges alltid hvit.

Dette bildet er så godt i utgangspunktet at man egentlig ikke trenger å sette til parametrene til noe mer enn minimumsverdiene for at matematikken i algoritmen skal gå opp. (Unngå divisjon med 0).

Parametrene:

Medianstørrelse: 1

Kantfølsomhet: 1

Kantbredde: 3

Medianstørrelse, kantbilde: 1

Konklusjon

Etter litt tuning og testing virket programmet slik det skulle. De største utfordringene har ikke vært selve segmenteringsalgoritmen, men alle de små tingene som fulgte med det å skulle lage et så fleksibelt program som mulig.

Enkle bilder segmenteres selvsagt best. Naturlige bilder med mange detaljer og fargenivåer blir som regel ikke segmentert særlig bra. Jeg ser imidlertid for meg at teknologien slik den er i dag uansett vil bli mest brukt i industrisammenheng, for eksempel for å hjelpe en robot å gripe objekter langs et samlebånd.

Jeg har lært mye i løpet av denne oppgaven, og siden jeg liker å programmere har det vært både moro og utfordrende.

Kanskje fortsetter jeg med bildehandling som høstprosjekt og diplomoppgave, og siden jeg egentlig tilhører datamaskingruppa tenkte jeg det kunne ha vært spennende å implementere fargebildesegmentering i hardware, og kjøre det i sanntid ved bruk av webcam.

Man får bare vente og se hva som byr seg av anledninger for valg av oppgave.