

RAPPORT
SIE4070 Konstruksjon av digitalkamera
i CMOS-krets

Øyvind Tanum

Frank Johannessen

Rune M. Andersen

23. april 2003

Abstrakt

Vi er tre 4.klasse studenter fra linjen datateknikk ved Norges teknisk-naturvitenskapelige universitet (NTNU). Vi tar faget SIE 4070 Konstruksjon av digitale kamerakretser i CMOS-teknologi ved fysikalsk elektronikk ¹. I motsetning til de fleste andre studentene som tar kurset, tilhører vi ikke fysikalsk institutt. Vi har mindre elektronikk-kunnskap og har derfor valgt å fokusere oppgaven vår mer mot datateknikk.

Slik utviklingen går idag, er det stadig behov for videre automatisering. Spesielt innen industri, men også andre deler av samfunnet. Et felt som åpner store muligheter for økt automatisering i disse dager er datasyn. Ved å kunne gi en maskin syn og evnen til å tolke det den ser, kan den arbeide under mye større autonome forhold. Maskinen kan for eksempel kalibrere seg selv etter det den ser. Det å gi en maskin syn er en enkel sak, men å få den til å forstå det den ser er mye vanskeligere. En metode for å forenkle tolkningen av et bilde er å utføre en kantdeteksjon av det. Kantdeteksjon er en forholdsvis enkel bildebehandlingsmetode samtidig som det er en av de aller viktigste. Grunnen er ganske enkelt den enorme reduksjonen i datamengden som maskinen må tolke. Et forholdsvis normalt digitalt bilde kan inneholde for eksempel 3 MegaByte med data, mens kantene etter deteksjon (og noe videre behandling) kan beskrives som vektorer på noen få KiloByte lagerplass.

Trondheim 23. april 2003

Frank Johannessen

Rune Martin Andersen

Øyvind Tanum

¹<http://www.fysel.ntnu.no/Courses/SIE4070/>

Innhold

1	Sammendrag	1
2	Bakgrunnstoff	3
2.1	Pikselet	3
2.1.1	Photogate vs. photodiode	4
2.2	Analog til digital omformer (ADC)	4
2.2.1	Nyquist ADC	8
2.2.2	Oversampling ADC	12
2.3	Bildebehandling i software	13
2.3.1	Automatisk hvit-balanse	13
2.3.2	Fargeinterpolasjon	13
2.3.3	Gammakorrigering	14
2.3.4	Kantdeteksjon	15
3	Designvalg	17
3.1	Piksel	17
3.2	A/D omformer	17
4	Implementasjon av filtre	19
4.1	Lavpassfilter	20
4.2	Høypassfilter	20
4.3	Andre bildebehandlingsteknikker	21
4.3.1	Gråtoning av kanter	22
4.3.2	Terskling (eng. Thresholding)	22
5	Resultater og konklusjon	25
5.1	Resultat av implementering av kantdeteksjon	25
5.1.1	Bilde 1: Lett	25
5.1.2	Bilde 2: Middels	25
5.1.3	Bilde 3: Vanskelig	27
5.2	Konklusjon	27
6	Kildereferanser	29

Kapittel 1

Sammendrag

Vi har i denne rapporten studert et kamera og implementert tilhørende programvare som er ment for deteksjon av kanter i en innendørs scene. Vi har studert de forskjellige aspektene som ligger til grunn for et digitalt kamera. Kameraet vi har valgt å se nærmere på er basert på CMOS-teknologien.

Måten vi har skaffet informasjon om de ulike delene er ved søk på Internett, samt bøker og tidligere rapporter som vi ble henvist til av undervisningsassistent Kristian Vonbun. Ut i fra denne informasjonen har vi valgt de kamerakomponentene som vi mener passer best til vår applikasjon.

Etttersom ingen av gruppe medlemmene har bakgrunn innen elektronikk, hadde vi innledningsvis store problemer med å finne ut hvor vi skulle begynne. Vi har derfor vektlagt bakgrunns materialet, slik at vi bedret vår basisforståelse av teknologien bak digitalkameraer. Vår bakgrunn fra datateknologi har også ført til at vi har fokuserert mer på programvare delen og mindre på det elektroniske. Resultatene av kameraet kan vi egentlig ikke si noe om, ettersom vi ikke har implementert det. Vi har derimot implementert programvare som detekterer kantene i et bildet. Programvaren burde integreres i hardware hvis den skal brukes i et praktisk bildebehandlingsystem, men vi ville vise funksjonaliteten fremfor integrerbarheten.

Kapittel 2

Bakgrunnstoff

I de siste 20-30 årene har CCD (Charge Coupled Device) blitt den nesten enerådende bildesensoren brukt i digitale kameraer. Gjennom årene har denne teknologien utviklet seg til bruk i mange applikasjoner, som f.eks. scannere og videokameraer. På midten av 90-tallet ble det tatt i bruk en ny teknologi innen bildesensorer. Dette var APS-CMOS (Active Pixel - Complementary Metal-Oxide Semiconductor) sensor. "Active Pixel"-sensor betyr at hvert piksel har en innebygget forsterker som medfører et bedre signal til støy forhold. Fordelene med CMOS er lavere strømforbruk, pris og areal. Samtidig slipper bildesensorbransjen å bekoste all utvikling selv, da dette er en generell teknologi. Dette medfører også at fabrikker som produserer bildesensorer kan produsere andre CMOS-produkter.

Noen likheter og ulikheter:

- Produksjon av CCD krever spesialiserte/dyre prosesser
- CMOS kutter faste kostnadene, fordi kostnadene blir spredt over et mye større antall typer produkter.
- Begge er produsert av silicon = lik følsomhet over hele spekteret synlig lys til IR
- Lik fotokonverteringsprosess
- Begge støtter photogate og photodiode

I dette kapittelet vil teoribakgrunnen for de forskjellige hoveddelene i digitalkameraet gjennomgås. Disse er pikselet, analog til digital omformeren og bildebehandlingen.

2.1 Pikselet

Pikselet er kretsen som genererer et punkt i bildet. Hjertet i pikselet er en fototransformator. Denne konverterer innkommende lys til en strøm av elektroner. Innen CMOS-fototeknologi er det tre fototransformatorer som vanligvis brukes, nemlig fotodiode, fotogate og fototransistor. I denne

rapporten ser vi bort fra fototransistorer, da fotodiode og fotogate ansees som bedre på de fleste områder.

2.1.1 Photogate vs. photodiode

Photogate og photodiode har mange likhetstrekk, men avhengig av applikasjon kan ofte en teknologi foretrekkes fremfor en annen. En slik avveining kan baseres på de små ulikhetene mellom photodiode og photogate.

Følgende punkter kan settes opp:

- Antall transistorer: En aktiv photodiode trenger tre transistorer, mens en photogate trenger 5. Dette påvirker den fysiske størrelsen på pikselet, som igjen påvirker størrelsen på CMOS-brikken, samt oppløsningen til kameraet. Dersom høy oppløsning eller liten størrelse er ønsket, foretrekker man photodiode fremfor photogate. Fill factor blir også påvirket av antall transistorer på brikken, noe som igjen påvirker lysfølsomheten.
- Kvantevirkningsgrad: Photodiode har høyere kvantevirkningsgrad enn photogate fordi sistnevnte har en polygate som ligger over depleksjonsområdet på pikselet. Dette reduserer lysfølsomheten.
- Støy: Photodioden har 10-20 ganger mer støy enn photogaten, noe som er gunstig hvis man ønsker best mulig bildekvalitet, til for eksempel høyoppløselige og svært detaljerte bilder.
- Conversion gain: En photogate har liten utgangsnødd, noe som fører til stor conversion gain.
- Lysfølsomhet: Fotodioden er mer lysfølsom, spesielt i den blå enden av spekteret.

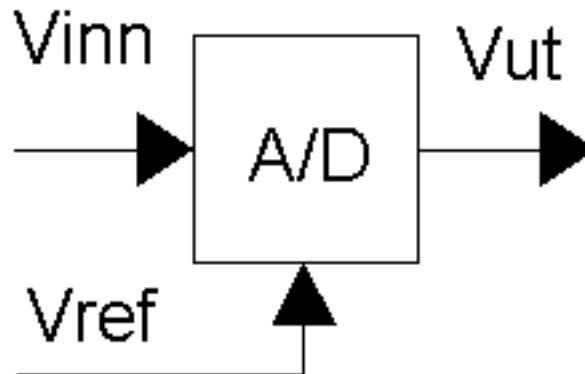
Først når man har gjennomgått disse punktene og funnet ut hva man ønsker, kan man velge den pikseltypen man synes passer best.

2.2 Analog til digital omformer (ADC)

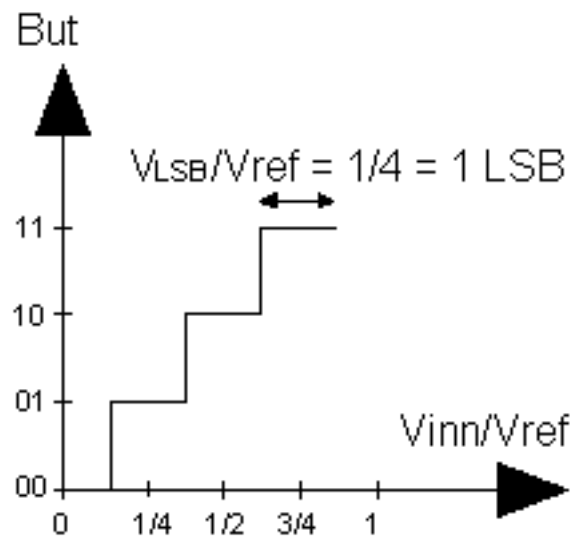
Utsignalet fra sensoren i digitale kameraer (CCD/CMOS) er et analogt spennings-signal. Dette signalet blir forsterket og digitalisert av en Analog til digital konverter (ADC). Den digitale utverdien fra ADC'en vil representere mengden av lys som pikselet fanget opp under integrasjonstiden. De fleste ADC'er i vanlige digitale kameraer har 8 bit. Det vil si at den kan konvertere et inn signal til 256 ulike utverdier for lysstyrke. Mer avanserte kameraer kan ha 10 eller 12 bit konvertering som gir henholdsvis 1024 og 4096 ulike lysintensitets nivåer.

Ideelt sett vil en A/D-omformer få inn en signalspenning (V_{in}) og en referansespenning (V_{ref}), og sende ut et digitalt signal som tilsvarer den analoge verdien. Det digitale signalet er diskret, i motsetning til den analoge spenningen som er kontinuert. Referansespenningen tilsvarer den største signalspenningen omformeren kan konvertere addert med verdi på det minste bittet (LSB). Det finnes

to hovedtyper A/D-omformere, nemlig Nyquist og Oversampling omformere. Det er nødvendig å benytte en A/D-omformer, dersom man skal drive med Digital Signal Prosessering (DSP).



Figur 2.1: 'Black box' A/D-konverter



Figur 2.2: A/D IO

Digital signalprosessering tilbyr:

- Et ekstremt bra signal/støy forhold. Dette forholdet er nær uendelig.
- Lav system kostnad
- Gjentakende system

En A/D-konverter har følgende flaskehalser:

- dynamisk område
- omformings-hastighet
- strømforbruk

Krav og ønsker til en A/D-omformer:

- Innsignalet må være på spenningsbølgeform.
- Det dynamiske området til inngangssignalet bør nær det dynamiske området til data-anskaffelse-systemet, som vanligvis er V_{ref} eller $2 * V_{ref}$.
- Det er viktig å maksimere oppløsningen til A/D-omformeren.
- Kildeimpedansen til inngangssignalet bør være lavt nok, slik at endringer i impedansen til data-anskaffelse-systemets inngangssignal ikke har noen effekt på inngangssignalet.
- Båndbredden til inngangssignalet må være begrenset til mindre en halvparten av samplingraten til A/D-omforminger.

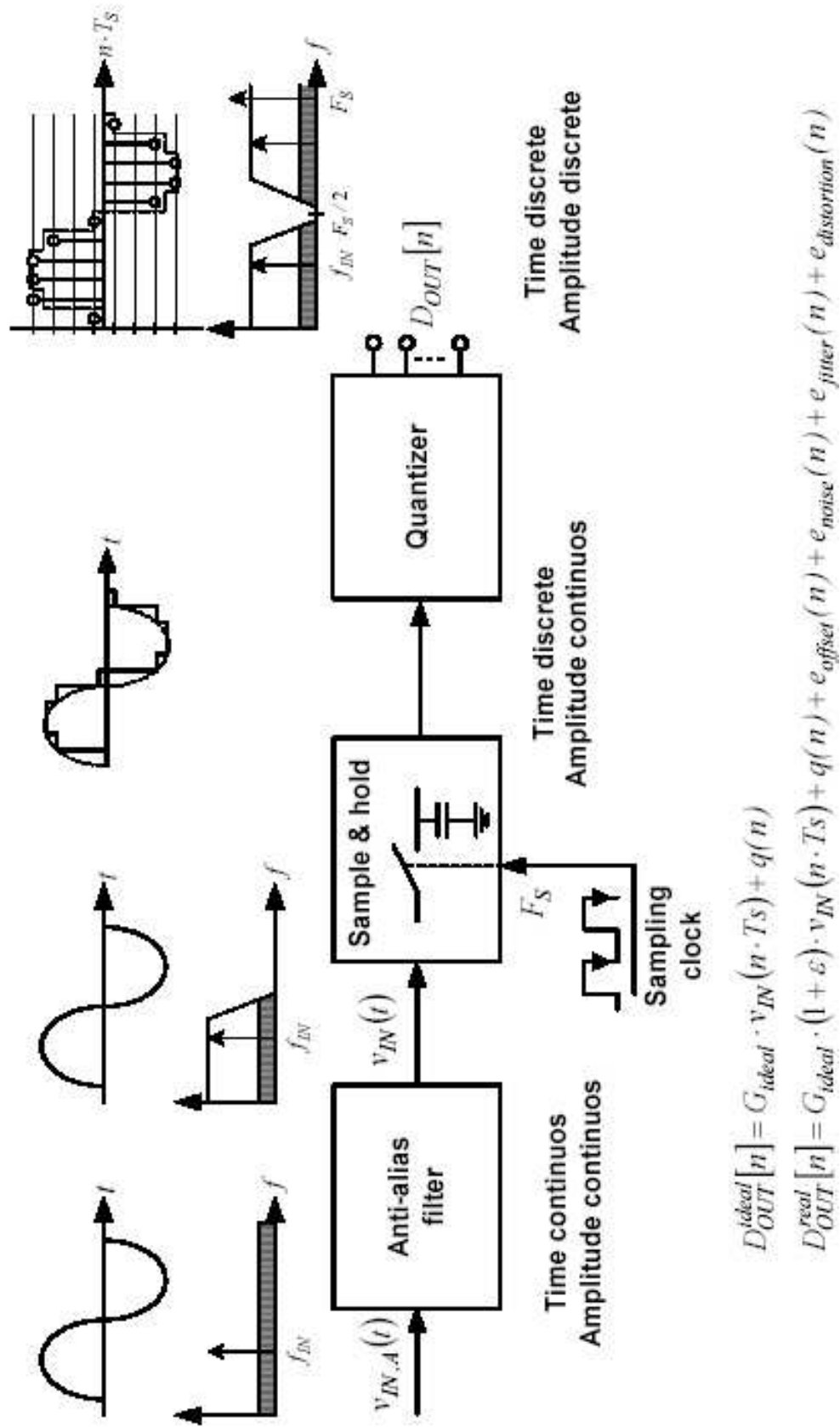
Anti-aliasing filter

Nyquist kriteriet krever at alle signaler må begrenses til mindre enn halvparten av samplingraten til samplingsystemet. Dette fører til at man ikke får aliasing i bildet. Mange signaler har allerede et begrenset spektrum, så da er dette ikke noe problem. For 'brede' signalet behøves en analogt lavpass filter før data-anskaffelse-systemet. Den minste dempningen til dette filteret ved aliasing-frekvensen bør minimum være ($A_{min} = 20 * \log(\sqrt{3} * 2^B)$), hvor B er antall bit i A/D-omformeren. Denne formelen kommer av at det er et minimum støy nivå som stammer fra samplingprosessen, så det er ingen vits å dempe sensorsignalet til et lavere nivå enn støynivået.

Problemer med anti-aliasing filter:

- Tidsrespons
Når man designer et anti-aliasing filter er det fristende å redusere signal-attenueringstiden så mye som mulig slik at tidsresponsen blir best mulig. Et filter med ekstremt liten attenueringstid vil kunne gi ringe-effekter ved store endringer i inn-signalet. Ringe-effekt er et påfølgende sinus ut-signal hvor amplituden reduseres over tid. Ringe-effekten vil øke inverst proporsjonalt med attenueringstiden.
- Tids-forsinkelse p.g.a. inn-signal frekvens
I de fleste analoge kretser vil høyfrekvente og lavfrekvente signaler forplante seg i ulik hastighet. Høyfrekvente signaler beveger seg langsommere enn lavfrekvente. Dette blir spesielt et stort problem når man konverterer flere signaler samtidig (f.eks. ved musikk).
- Amplitude forvrengning
Anti-aliasing filteret kommer til å endre på signalets frekvens-struktur. Dette vil ofte skape komplikasjoner.

A/D-converter basics



$$D_{OUT}^{ideal}[n] = G_{ideal} \cdot v_{IN}(n \cdot T_s) + q(n)$$

$$D_{OUT}^{real}[n] = G_{ideal} \cdot (1 + \epsilon) \cdot v_{IN}(n \cdot T_s) + q(n) + e_{offset}(n) + e_{noise}(n) + e_{jitter}(n) + e_{distortion}(n)$$

Figur 2.3: Oversikt over A/D-konverter

Alle disse ulempene kan reduseres ved å øke samplingsraten. Lineære fasefiltre vil også selvfølgelig kunne redusere problemet med ulik tidsforsinkelse avhengig av signal-frekvens.

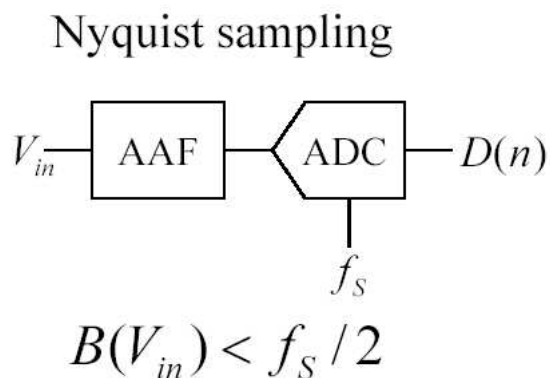
Støy

Siden A/D-omformeren gir ut 2^B nivåer stammer det støy fra det kvantiserte ut-signalet. Forholdet mellom signalet og det kvantiserte signalet kaller SNR (Signal to Noise Ratio). SNR er i dB og er tilnærmet seks multiplisert med antall bit i A/D-omformeren: $20 * \log(SNR) = 6 * Bits$. Dette betyr at signal/støy forholdet for en 16-bit A/D-omformer 96dB. Det er også andre kilder som forstyrrer utgangssignalet til en konverter, som for eksempel støy fra sensoren, signal-behandling kretsene og andre digitale kretser. Nøkkelen til reduksjon av støyen er å maksimere inngangssignalet. For å oppnå dette må forsterkningen (gain) i signal-behandling kretsene økes inntil maksimum sensor ut-signal er lik referansespenningen i A/D-omformeren. Det er også mulig å redusere referansespenningen til sensornivået (maks). Problemet med dette er at støy vil ødelegge de små signalene. Tommelregel: Hold referansespenningen minst like stor som maksimalt digitalt signal.

2.2.1 Nyquist ADC

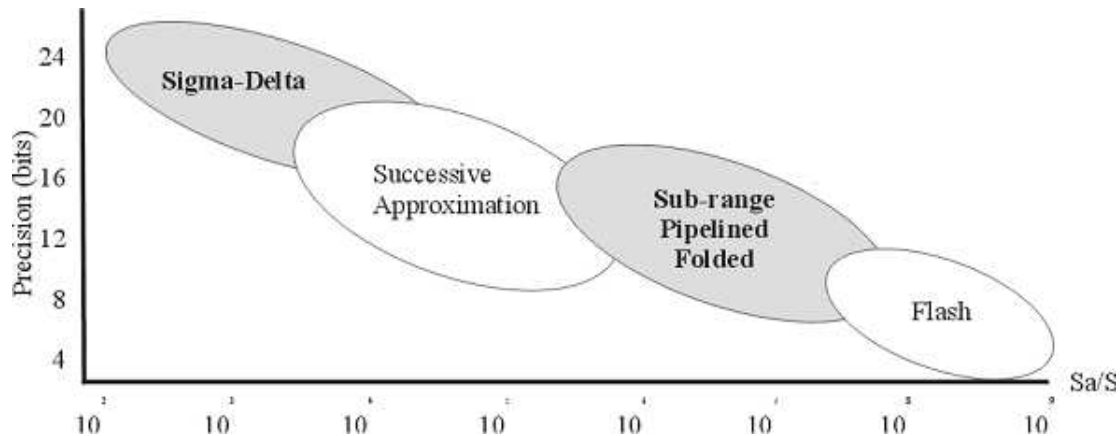
Nyquist omformerne er de konverterne som genererer en serie ut-verdier, der hver verdi har et en-til-en samsvar til en enkel innverdi. Nyquist rate konvertere har praktiske problemer med anti-aliasing- og gjenskapningsfiltre.

Nyquist rate eller intervall er den teoretisk minste samplingraten som fullstendig beskriver signalet. Den egentlige sampling raten som behøves for at gjenskape det originale signalet, er noe større en Nyquist raten, fordi sampling prosessen lager kvantiseringsfeil. Nyquist omformere opererer sjelden ved Nyquist raten, og det er typisk å ha en hastighet på 1,5-10 ganger Nyquist raten. Nyquist konverterne stiller større krav til anti alias filter (AAF) enn oversampling konvertere.



Figur 2.4: Nyquist A/D-konverter skjema

Forskjellige A/D-konverter Arkitekturer		
Lav/Middels hastighet Høy nøyaktighet	Medium hastighet Medium Nøyaktighet	Høy hastighet Lav/Middels nøyaktighet
Integrerende Oversampling	Etterfølgende Approksimering Algoritmisk	Flash To-steg Interpolerende Sammenleggende Samlebånd (pipeline) Tidssammenfletting



Figur 2.5: Nyquist A/D-konverter karakteristikk

Sample-and-hold kretser

Analog til digital omformningen begynner med en sample-and-hold krets (SogH) eller en track-and-hold krets (TogH). Disse forventes å øyeblikkelig ta opp et sample fra inngangssignalet og holde denne verdien til den behøves. Fra sampling teoremet er det en velkjent kobling mellom det analoge systemets båndbredde og den minste samplingsraten (Shannons Sampling Teorem) som kreves, hvis vi skal sikres en feilfri signal-gjenoppbygning. SogH elektronikken oppretter både kvalitativ og kvantitativ grenser på samplingsprosessen. SogH kretsene kan bli sett på som ideelle brytere som blir dårligere grunnet jitter etc.

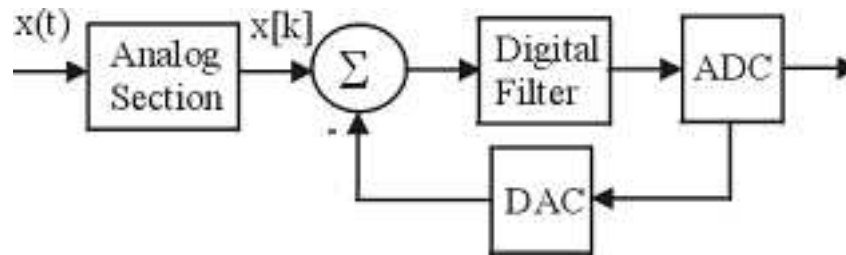
Typer Nyquist omformere

- **Integrerende ADC**

Integrasjonskonverterere gir veldig god oppløsning og tar liten plass. Men de bruker desverre ekstremt lang tid.

- **Delta-Sigma ADC**

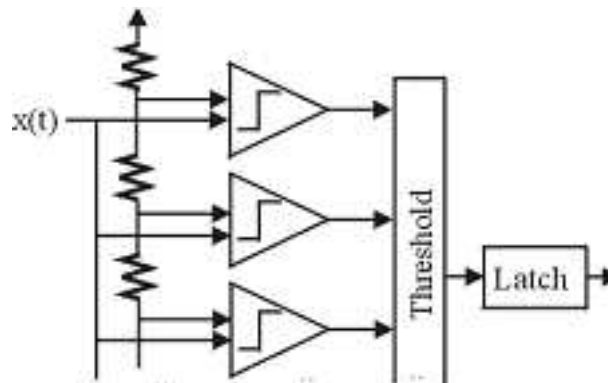
Delta Sigma omformeren kombinerer oversampling med feedback. Den analoge fronten inneholder et antall upresise komponenter. Kompleksiteten er flyttet til sluttprosessering, som krever en Digital til Analog konverter (DAC) og digitale filtre. Delta Sigma omformerne opererer ved relativt lave samplingsrater.



Figur 2.6: Delta Sigma ADC skjema

- **Flash ADC**

Flash konvertere består av en samling dedikerte utjevnere (komparatorer) og et flerlags terskel detektorer. Falsh omformerne er den raskeste av alle ADC arkitekturene, men bruker mye energi og er stor av støyrelse.



Figur 2.7: Flash ADC skjema

- **Samlebånd ADC (pipeline)**

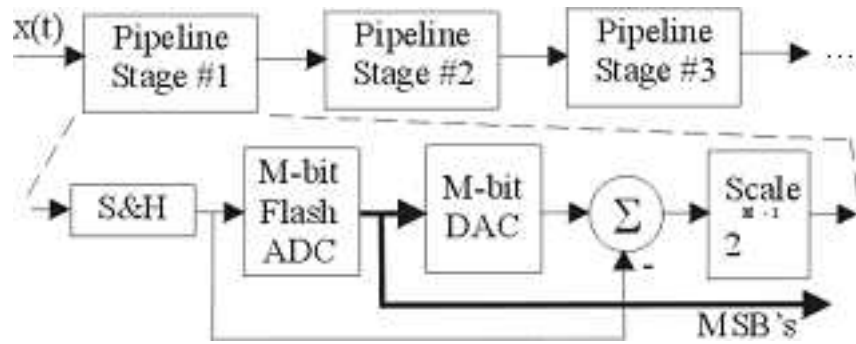
Samlebånd omformere fordeler konverteringsprosessen over flere steg ved å bruke en samling av ADC og DAC med kort ordlengde. Hvert steg består av en AD omformer som opererer i sekvens med den etterfølgende ADC. Samlebånd ADC krever en track and hold krets, en kvantiserer og en DAC for alle stegene unntatt det siste steget.

- **Folded ADC**

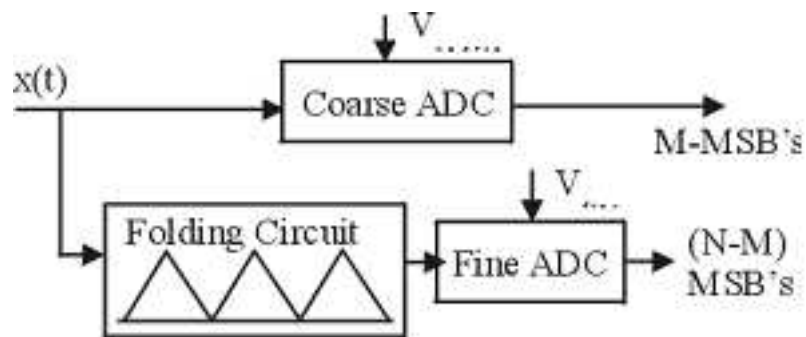
Foldning, også kalt interpolasjon av ADC benytter seg av en rekke foldnings-blokkersom vist i figuren under. Interpolasjon blir brukt her til å redusere antall blokker man trenger for en gitt oppløsning. En foldnings-blokk inneholder N sammenkoblede differensierte par og former $\log(2N)$ kvantiserere. Frekvensen til foldnings-blokkens minste byte er N ganger så stor som frekvensen til det analoge signalet. En foldnings-ADC uten interpolasjon er det samme som en Flash-ADC.

- **Etterfølgende Approksimasjoner ADC**

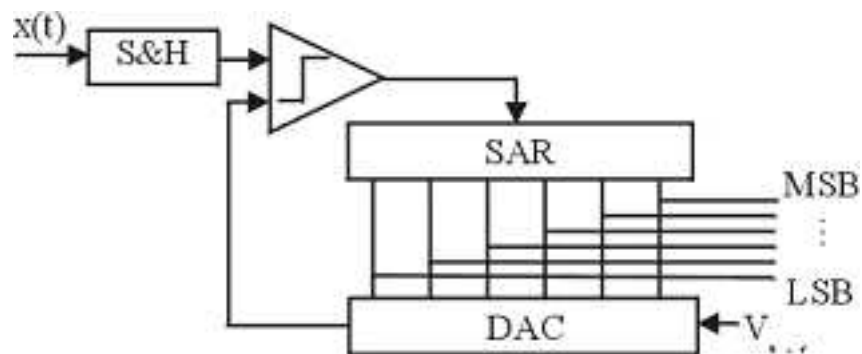
Etterfølgende approksimasjonskonvertere er populære for mellomklasse applikasjoner. En slik ADC er illustrert i figuren under. Hastigheten av den er avhengig av ordbredden.



Figur 2.8: Samlebånd ADC skjema (pipeline)



Figur 2.9: Folded ADC skjema



Figur 2.10: Etterfølgende approksimasjoner ADC skjema

- **Algoritmisk ADC**

Fungerer forholdsvis likt som en Etterfølgende approksimasjons ADC. Forskjellen er at denne typen dobler feil-spenningen istedenfor å halvere referanse-spenningen.

- **To Skritt ADC**

Denne konverteren er et godt valg for applikasjoner som ikke krever altfor høy oppløsning. Fordelene er blant annet lite plassbehov og lavt strømforbruk samtidig som konverterings-hastigheten er svært høy. Propageringstiden begynner også å nærme seg propageringstiden på en flash ADC.

- **Tids-flettet ADC**

For konvertere som krever ekstremt høy oppløsning, må man bruke flere konvertere i parallell. Et problem her kan bli feil på grunn av forskjellige offset på del-konverterne.

2.2.2 Oversampling ADC

Oversampling omformerne er de konverterne som opererer mye raskere enn inn-signalets Nyquist rate. Disse omformerne bedrer også ut-signalets signal/støy nivå (SNR) ved å filtrere bort kvantiserings-støyen som ikke er innenfor signals båndbredde. Kvantiserings-støy er støy som er forårsaket av approksimeringsfeil i kvantiseringsprosessen som blir brukt og det kvantiserte signalets statistiske særpreg.

Filtreringen i en A/D-omformer blir utført digitalt. Sammenlignet med Nyquist rate omformere trenger vi et mye enklere og billigere anti-alias-filter (AAF). Mens Nyquist rate omformere opererer ved 1,5-10 ganger Nyquistraten, opererer oversamplingkonverterne ved 20-512 ganger NR. Oversamplingomformere må ha bred innbåndbredde. Ved hver dobling av oversamplingforholdet økes signal-støy-forholdet med 3dB. Dette kalles prosessering økning (PG - processing gain), som også økes ved støy forming (noise shaping).

Hvordan oversampling og midling forbedres ytelsen

Oversampling og midling (gjennomsnitt) blir gjort for å forbedre SNR og øke det effektive oppløsningen (f.eks. øke antall bit). Begge disse er det samme. For eksempel hvis vi har en 12-bit omformer og ønsker å generere signaler med 16-bit oppløsning, da kan vi bruke oversampling og midling for å oppnå samme SNR som en 16-bit ADC. Dette vil øke antallet effektive bits (ENOB) for de målte data, som er et annet mål på SNR. Fordi oversampling og midling produserer et lavere støy gulv i signalbåndet, tillater det oss å realisere 16-bit utord.

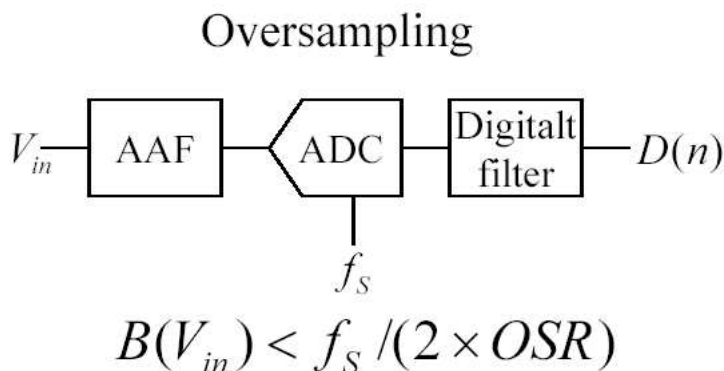
Hvordan oversampling påvirker inn-bånd støy

En sampling frekvens vil tillate aktuelle signaler å bli gjenskapt på halvparten av samplingfrekvensen (Nyquist teorem). For eksempel hvis samplingraten er 100kHz, så kan vi gjenskape og analysere signaler under 50kHz. Sammen med inngangssignalet vil det være et støysignal (hvit støy til stede

ved alle frekvenser) som vil folde. Gulv støyen av samplet støy vil synke i signalbåndet hvis samplingfrekvensen blir øket.

Relasjonen mellom oversampling og økt oppløsning

Hvis vi har den faste støykraften fra kvantiseringsstøy, kan vi kalkulere mengden oversampling som behøves for å øke den effektive oppløsningen. For eksempel hvis vi vil øke antallet effektive bit (ENOB) for en målt parameter med en 12-bit A/D omformer til en 16-bit måling. Da vil vi etablere relasjonen som tillater oss å kalkulere oversamplingkravet, ved å først definere støykarakteristikken.



Figur 2.11: Oversampling A/D-konverterer skjema

2.3 Bildebehandling i software

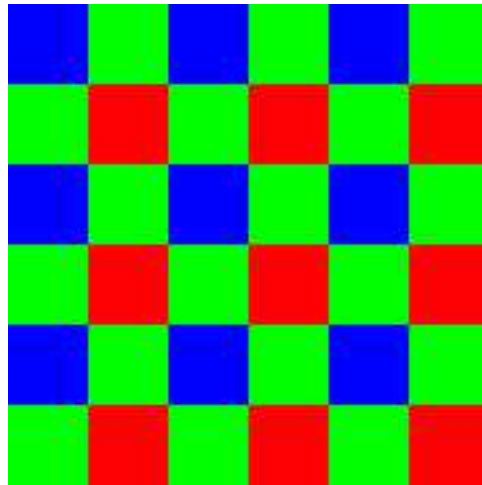
2.3.1 Automatisk hvit-balanse

Ved varierende lysforhold vil hvitbalansen i bildet forandre seg. Mange har sikkert sett at bilder tatt innendørs kan få blåskjær eller bli gule, mens bildene man tar utendørs ser penere ut. Hvitbalansen i bildet forteller om det hvite i bildet virkelig er hvitt, eller om det drar i retning av en annen farge. Digitale fotoapparater har ofte automatisk hvitbalanse for at brukeren skal slippe å forandre instillingene på apparatet avhengig av hvor bildet blir tatt. Å justere hvitbalansen kan være en utfordring, og riktig algoritmevalg er en forutsetning.

2.3.2 Fargeinterpolasjon

Som regel kan hvert piksel i en sensor bare fange opp én farge. Det vil si at vi må blande minst tre forskjellige typer piksler for å greie å representere grunnfargene rød, grønn og blå (RGB). Dersom vi hadde hatt to farger ville vi ganske enkelt kunne laget et sjakkemønster, men siden vi har tre farger er det ingen måte å fordele pikslene helt jevnt uten å få streker av forskjellige farger i bildet. For å løse

dette problemet må ha kunnskap om øyets fysikk. Siden øyet er mest følsomt for grønt lys, kan vi tillate oss å bruke grønne piksler mer enn de to andre fargene. En god fordeling av pikslene er vist i figur 2.12. Denne konfigurasjonen kalles *Bayer pattern* og er en av de mest brukte.



Figur 2.12: Pikslenes farger fordelt i et rutenett

Når bildet skal leses ut, ønsker vi imidlertid ikke et rutemønster med forskjellige farger, men et bilde hvor hvert pixel har korrekt bidrag fra alle tre fargene. For å få til dette må vi interpolere pikslene fra sensoren, noe som kan vise seg å være vanskeligere enn man skulle tro.

Den intuitive måten å gjøre fargeinterpolering er å midle over fargene. For et gitt piksel i posisjon X , kalkulerer vi gjennomsnittet fra alle piksele rundt X og fortsetter prosessen med piksel $X+1$. Problemet med denne metoden er at vi mister mye av skarpheten i bildet, ettersom man har midlet over nabopiksler. Det som skjer med bildet er akkurat det samme som skjer når man blurrer (glatter) et bilde. Hvis man vil ha tilbake skarpheten må man i tillegg benytte seg av høypassfilter eller en annen bildebehandlingsteknikk.

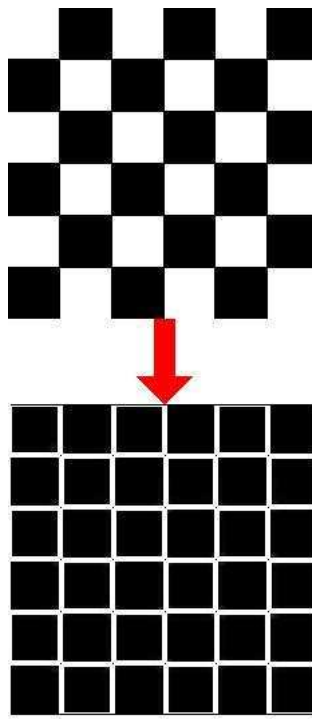
Forbedrede interpoleringsalgoritmer finnes også, men tas ikke med i denne rapporten, siden vi i denne omgang kun ønsker å gi en kort innføring i hva interpolering er, ikke hvordan den kan optimaliseres. Disse teknikkene kalles ofte aperturekorrigering, siden de bringer tilbake skarpheten i bildet.

2.3.3 Gammakorrigering

Fordi lysfølsomheten i pikselene og øynene våre er forskjellig, vil det bli behov for å strekke eller flytte på fargeskalaen for å bestemme hvor mørkt sort skal være og hvor lyst hvitt skal være. Denne skalaforandringen kalles gammakorrigering og kan utføres automatisk ved forhåndsbestemte verdier, eller den kan instilles manuelt ved at brukeren ser på testbilder og bedømmer hvilke som er best.

2.3.4 Kantdeteksjon

Kantdeteksjon går ut på å oppdage alle skarpe farge-/lysoverganger i et bilde. Grunnen til at man kaller det for kanter er at man som regel har endringer i lysintensitet over en virkelig kant. Rent matematisk er resultatet man ønsker absoluttverdien av den deriverte av lysintensiteten for hvert punkt i bildet. Man må imidlertid legge merke til at ikke alle lysintensitetsendringer er en følge av kanter i scenen. De kan for eksempel oppstå som følge av lyskilder, flate mønstre eller støy fra kameraet. Det kan også være at scenen er overfylt med små insignifikante kanter som man ikke har regnekraft til å tolke. Så en ren derivasjon av bildet vil nesten aldri være tilstrekkelig (Med unntak for ideelle dataskapte bilder). I praktiske bildebehandlingssystemer vil man ha en førprosessering, som gjør bildet klar for kantdeteksjon, og en etterprosessering, som fjerner helt klare feil i resultatbildet. Metoden er svært ofte den første store behandlingen et bilde går igjennom ved bildegjennkjenning/tolkning.



Figur 2.13: Kantdeteksjon

Kapittel 3

Designvalg

3.1 Piksel

Til vårt bruk falt valget på photodiode. Den viktigste grunner for dette er at vårt applikasjonsområde er å gjenkjenne kanter, og ved kantgjenkjenning er det viktig å ha høy oppløsning tilgjengelig. Hvis oppløsningen er for lav kan to kanter som ligger nært hverandre oppdages som bare en kant (smelte sammen), eller i verste fall ignoreres av algoritmene. Siden vi ønsker at kameraet skal være så lite som mulig, må vi derfor utnytte muligheten for maksimal oppløsning best mulig.

Ettersom vi regner med at bildene er preget av støy i utgangspunktet, antar vi at en photodiode er like grei å bruke som en photogate. Støyfjerning vil uansett bli foretatt for eksempel ved å glatte eller midle bildene. Den høyere oppløsningen er ment å kompensere for eventuelle detaljtap ved glatting.

Høy lysfølsomhet er også viktig, siden man ikke kan regne med å ha ekstra lyskilder ved real-time-bruk. Ved fargebildesegmentering er det hendig å ha god belysning, ellers vil mørke flater oppfattes som grå, og man taper fordelene av å ha fargeinformasjon tilgjengelig.

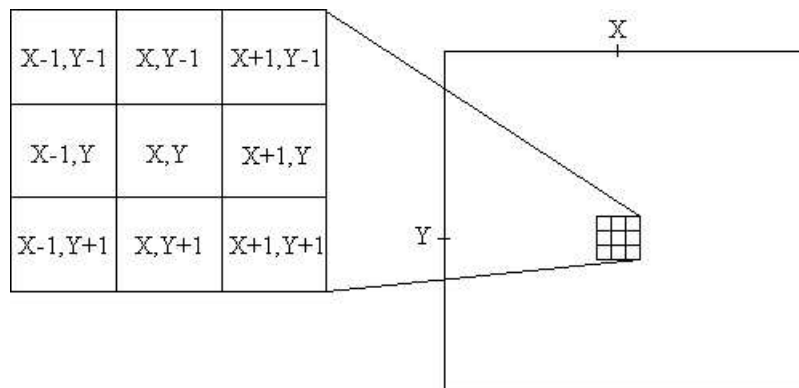
3.2 A/D omformer

Siden vi har et analogt spenningssignal fra bildesensorer, behøver kameraet muligheten til å konvertere dette signalet om til et digitalt signal. Slik det er tenkt vil vårt kamera benytte seg av en 8-bits A/D omformer. Dette gir oss en fargedypde på 24-bit eller 16777216 farger, som er mer enn tilstrekkelig for vår kantdeteksjon-applikasjon. Hvis vi øker antall bit på omformeren vil det gi ekstra kostnad uten å bedre oppløsningen markant. Videre har vi valgt å benytte oss av en 2-stegs arkitektur. Vi har valgt denne fordi den har lite energiforbruk, krever lite plass og har lav integrasjonstid. Denne arkitekturen er mer utdypet i bakgrunnskapitlet. Vi vurderte også å bruke en konverter per piksel, men vi var ikke sikre på om fordelene veide opp for ulempene dette medfører med tanke på kostnad og plass etc. Vi valgte derfor å sende hele bildematriksen til kun en A/D omformer.

Kapittel 4

Implementasjon av filtre

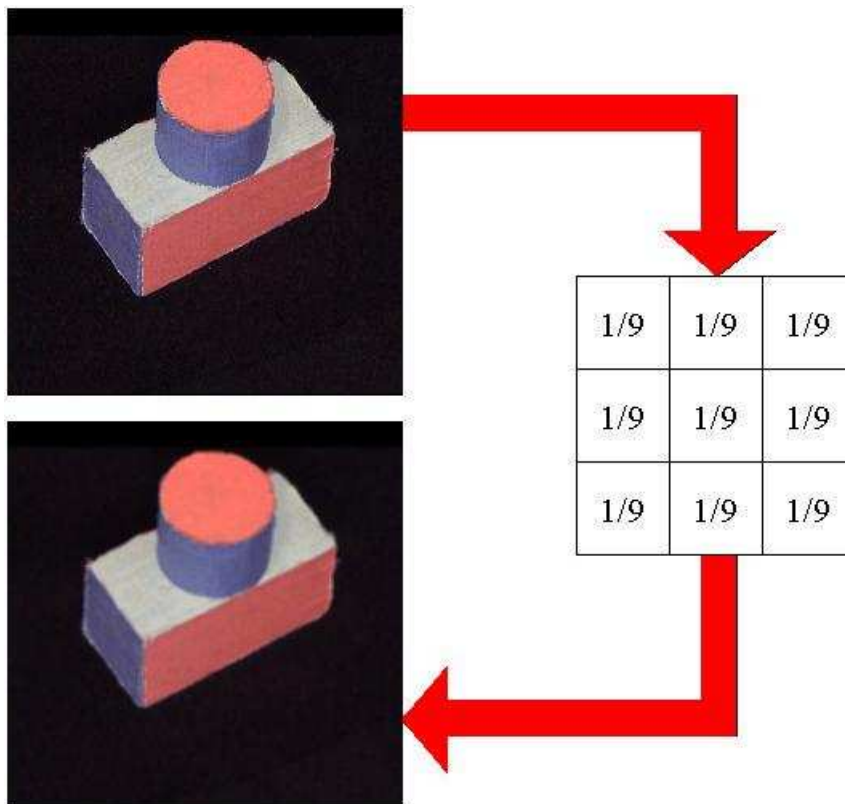
For å kunne hente ut kantene fra et bilde har vi brukt et sett med filtre. Et filter er en vektmatrise man legger over bildet, med det pikselet man skal manipulere i sentrum (se figur 4.1, side 19). Verdien til samme piksel i det manipulerede bildet vil da bli summen av produktet av vekten og tilhørende piksel. Etter at man har brukt vektmatrisen på alle pikselene i det originale bildet har man konstruert et nytt manipulert bilde, som man igjen kan bruke et annet filter på for å komme videre i bildebehandlingen. Disse pikselmaskene er vanligvis kvadratiske og har et oddetall med pikseler på lengden og bredden (for eksempel 3*3 eller 7*7). Hovedgrunnen til dette er forenklingen det fører til ved beregninger. pikselene i ytterkanten rundt bildet kan være noe problematisk ettersom deler av vektmatrisen vil ende opp utenfor bildet. Dette løses ofte med at man setter pikselene utenfor bildet til helt svarte eller at man bare ignorerer disse pikselene. Begge metoder fører ofte til at man får en ramme rundt de manipulerede bildene som ikke helt stemmer overens med resten av bildet. Vi går videre gjennom hvilke filtre som har brukt.



Figur 4.1: Eksempel på hvordan et filter brukes

4.1 Lavpassfilter

Et lavpassfilter er et filter som reduserer raske endringer i bildet og lar langsomme endringer bli. Det kan virke noe paradoksalt at vi fjerner raske endringer som nettop representerer kantene i bildet. Hensikten er at man kan redusere kun de mindre markante kantene og beholde de viktige ved å bruke et svakt lavpassfilter. Et eksempel kan være hvis man tar bilde av et vegghjørne med ruglette tapet. Uten lavpassfiltrering før kanteekstraksjon vil man få et bilde fylt med kanter, men hvis man lavpassfiltrerer først vil veggen virke helt jevn og ved kanteekstraksjon vil man bare ha den kanten som representerer hjørnet av veggen. Artifakte kanter skapt av støy forårsaket av kameraet vil også bli redusert. En annen fordel med å lavpassfiltrere først er at skarpe kanter blir fjernet. Ulempen med slike kanter er at de kan bli ekstraktert som en dobbeltkant. Det filteret vi bruker er en 3×3 matrise med vekten $1/9$ i hvert felt. Dette betyr at hvert piksel i det manipulerede bildet er gjennomsnittsverdien av seg selv og sine 8 naboer.

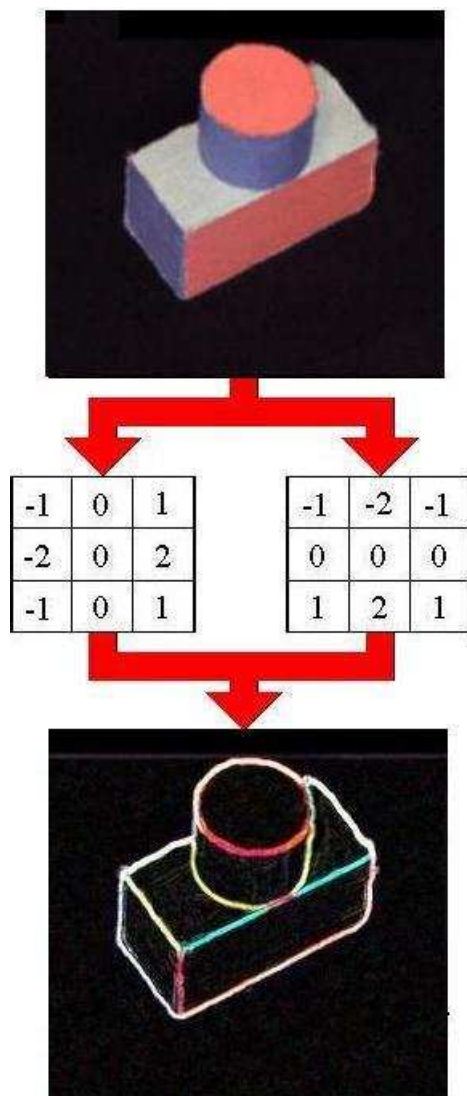


Figur 4.2: Lavpassfilter

4.2 Høypassfilter

Et høypassfilter er en tilnærmet utregning av lengden av gradientvektoren i hvert piksel i bildet. Store endringer i et område av det originale bildet vil representeres med et lyssterkt område i resultatbildet.

Resultatet av et høypassfilter vil altså bli kantene i bildet. Den matrisa vi har brukt blir kalt en 3*3 sobelmatrise (se figur 4.3, side 21). Denne matrisa har en kolonne med 1,2,1 på venstresida og -1,-2,-1 på høyresida. Verdien til pikselet blir nå absoluttverdien av utregningen ettersom resultatet kan bli negativ. Denne vil kun registrere vertikale kanter i bildet. Derfor har vi også brukt en matrise med verdiene 1,2,1 på den øverste raden og -1,-2,-1 på den nederste raden. Denne vil da registrere de horisontale kantene og ved å addere sammen resultatene vil man få med alle kantene.



Figur 4.3: Sobel

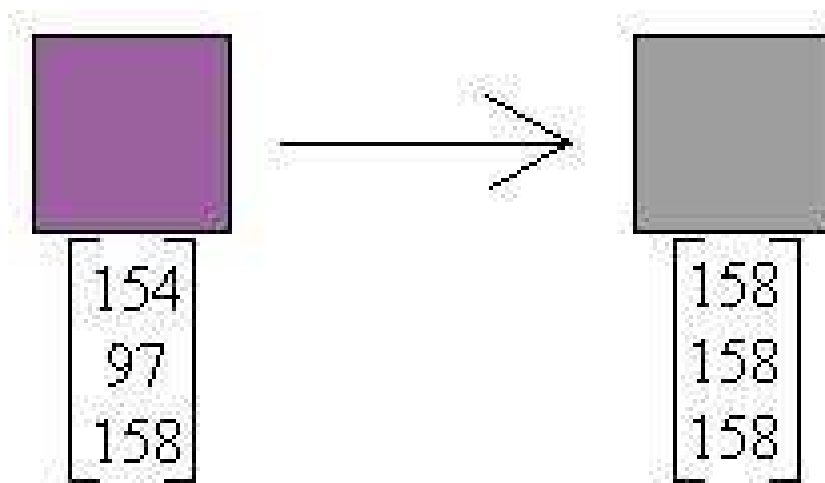
4.3 Andre bildebehandlingsteknikker

Kun bruk av filtre som beskrevet over vil ikke kunne gi et tilstrekkelig godt resultat. Derfor benytter vi oss også av et par andre teknikker for å forbedre resultatet. Disse går hovedsakelig ut på å behandle

ett og ett piksel hver for seg.

4.3.1 Gråtoning av kanter

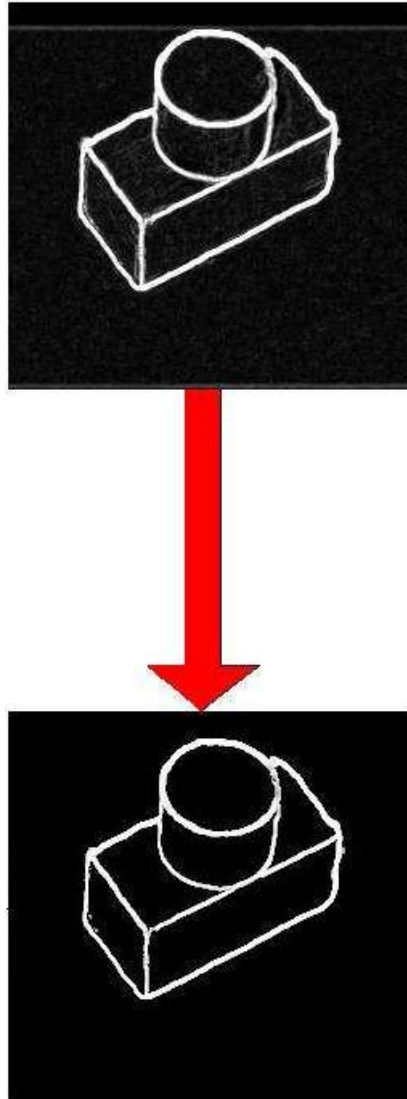
Når filtrene har blitt kjørt vil vi stå igjen med et kantbilde. Kantene her vil være farget i de(n) fargen(e) som endret seg over kanten. Dette vil for det første skape unødvendig mye data, ettersom det kun er kantene vi skal detektere. En annen problem med bildet er ved endringer på flere farger. Dette kan medføre at man får to eller tre svake parallelle kanter i forskjellige farger istedetfor en sterk. Ved å gråtone bildet vil man bli kvitt disse problemene. Måten vi gjør dette på er ved å gå igjennom hvert piksel i bildet og sette verdien på de to svakeste farge-elementene lik den sterkeste. (Se figur 4.4, side 22).



Figur 4.4: Gråtoning av kanter

4.3.2 Terskling (eng. Thresholding)

Etter lavpassfilteret vil det sannsynligvis fremdeles være en del insignifikante kanter i bildet som vi ønsker å bli kvitt. Disse kantene vil være representert med en middels lysstyrke. En enkel måte å kvitte seg med disse er å finne en grense for lysstyrken som de aller fleste uønskede kantene befinner seg under og de aller fleste ønskede kantene befinner seg over. Deretter setter man alle piksler med verdi under grenseverdien til helt svarte og alle over grenseverdien til helt hvite. Måten vi finner denne grenseverdien på er ved å anta at kantene i et bildet nå er omtrent halvparten ønskede kanter og de resterende uønskede kanter. (Dette er ikke annet en en gjetning som viser seg å stemme bra i de fleste bilder). Algoritmen går ut på at man først ignorerer pikselene som er så mørke at det umulig kan være en kant. (Vi har satt dette til lysstyrke- området fra 0-15 prosent). Deretter setter man grenseverdien til gjennomsnitts lysstyrken av de gjennværende pikselene. Resultatet vises av figur 4.5, side 23.



Figur 4.5: Terskling

Kapittel 5

Resultater og konklusjon

Dette kapittelet vil ta for seg resultatene av prosjektet, og i hvilken grad de er i overensstemmelse med tidligere antagelser. Ettersom det kun har blitt tatt overordnede arkitekturvalg og ikke noen konkret design av kameraet, vil vi derfor her kun se på resultatene av programvaredelen av prosjektet.

5.1 Resultat av implementering av kantdeteksjon

Vi vil evaluere funksjonaliteten av algoritmene ut i fra hvor godt de kan detektere kantene i tre forskjellige bilder. Det kan se ut som om det er en del støy rundt kantene i resultatbildene. Dette er ikke et resultat av kantdeteksjonen, men heller på grunn av konverteringen fra bmp-format til jpeg-format.

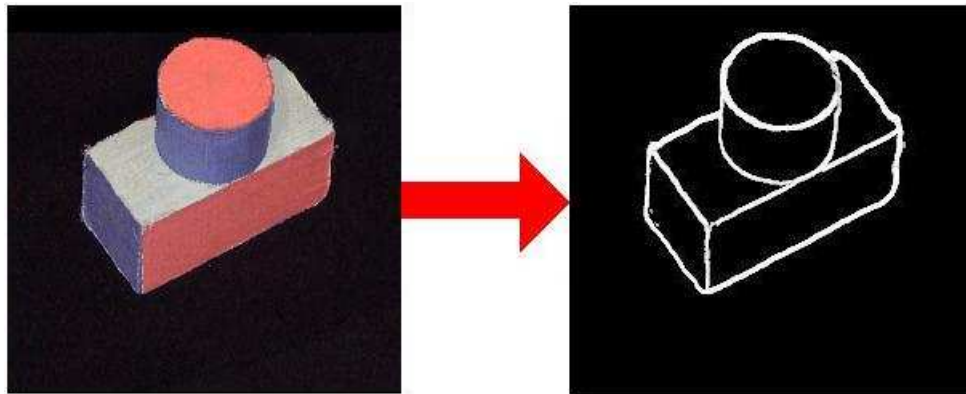
5.1.1 Bilde 1: Lett

Dette er det letteste bildet å detektere kantene på. Bildet består av en blokk hvor alle sidene med felles kant har forskjellige farger. Bakgrunnen er nesten helt uniformt svart (noe uregelmessigheter som følge av støy).

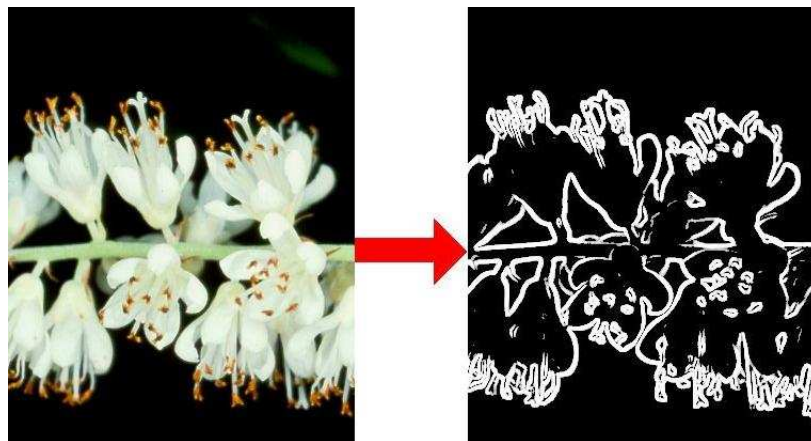
Som man kan se av bildet (se figur 5.1, side 26) får algoritmen med alle de signifikante kantene. Men det har også kommet med to små områder som ikke skulle være der (helt til venstre i figuren). Disse består derimot av så få pixeler, at de kan forkastes med en forholdsvis enkel algoritme forutsatt at man ikke forventer svært små kanter i bildet.

5.1.2 Bilde 2: Middels

Det neste bildet er betraktelig vanskeligere å behandle. Det består av mange små men kontrastfylte kanter. Bakgrunnen er hovedsaklig svart her også, men den inneholder også et grønt blad, som algoritmen bør filtrere bort.



Figur 5.1: Kloss, vanskelighetsgrad: lett

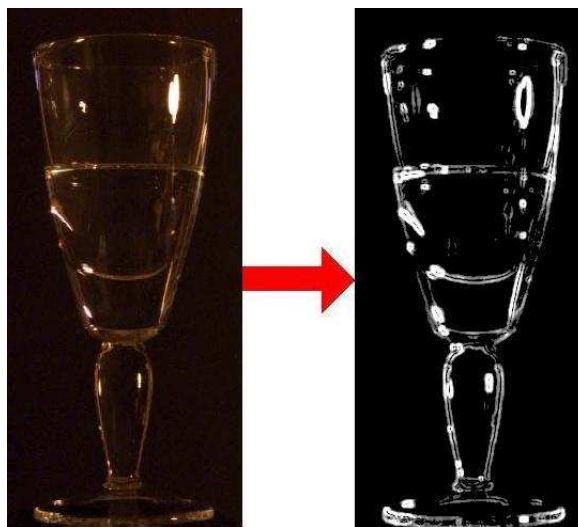


Figur 5.2: Blomst, vanskelighetsgrad: middels

Resultatet blir ikke så bra som for bilde nr. 1. De tynne kantene av pollenstilkene glir inn i hverandre som følge av at kantene i resultatet er for tykke. De aller tynneste kantene blir representert med doble kanter (et vanlig problem for slike algoritmer). Det er også en del områder på noen få pixeler, som ikke skal være der. Til tross for disse manglene har algoritmen også her utført oppgaven tilstrekkelig godt. Alle de viktige kantene i bildet har blitt bevart samtidig som det aller meste andre er blitt filtrert bort. (Se figur 5.2, side 26).

5.1.3 Bilde 3: Vanskelig

Det siste bildet er svært vanskelig å behandle. Objektet i bildet er både gjennomsiktig og reflekterende. Samtidig er det svakt belyst med enkelte områder som reflekterer sterkt lys.



Figur 5.3: Glass, vanskelighetsgrad: vanskelig

Som man kan se (5.3, side 27) blir resultatet temmelig svakt. Det mangler deler av signifikante kanter (øverst på randen og foten av glasset). Områder hvor lys blir reflektert på glasset blir også feilaktig detektert som kanter. Resultatet vil ha begrenset verdi i videre bildebehandling ettersom noe informasjon har gått tapt samtidig som det er blitt skapt en del desinformasjon.

5.2 Konklusjon

Resultatet av de to første bildene viser at algoritmen fungerer for enkle objekter. Den er også tilstrekkelig for mer komplekse objekter så lenge forholdene har blitt tilpasset. Det viktigste er jevn og tilstrekkelig belysning.

Det siste bildet viser derimot begrensningene av algoritmevalget vårt. For å få et godt resultat av et bilde som dette, kreves det kunnskap om bildet. Man må vite hva man holder på å detektere. Dette krever bruk av bildegjenkjenning. Dette igjen medfører et behov for tilgang til en bildedatabase og

mye prosesseringskraft. Men algoritmen vår er valgt for å kunne implementeres som maskinvare i et kamera (og brukes i sanntid), noe som gjør bildegjenkjenning i praksis umulig.

Kapittel 6

Kildereferanser

[1] <http://ccrma-www.stanford.edu/CCRMA/Courses/252/sensors/sensors.html>

[2] Eric Fossum, *CMOS Comes of Pushing the Envelope*, 2001

[3] Boyd Fowler, *CMOS Area Image Sensors With Pixel Level A/D Conversion*, 1995

[4] Keith Michael Findlater, *A CMOS Camera employing a Double Junction Active Pixel*, 2001

[5] S. K. Mendis, S. E. Kemeny, E. Fossum *CMOS Active Pixel Image Sensor for Highly Integrated Imaging Systems*, 1997

[6] R. H. Nixon, S. E. Kemeny, E. Fossum *256x256 CMOS Active Pixel Sensor Camera-On-a-Chip*, 1996

[7] David Johns, Ken Martin, *Analog Integrated Circuit Design*, 1997

Appendiks A - Java Kildekode

```
1
2 import java.util.*;
3 import java.awt.*;
4 import java.awt.event.*;
5
6 /**
7  * Hovedklassen: Tar seg av all styringen
8  */
9 public class Coordinator
10 {
11     //vinduet som viser bildene
12     Frame frame;
13     Frame frame2;
14
15     //selve bildene
16     Imager canvas = new Imager(this, Coordinator.ORGINAL);
17     Imager canvas2 = new Imager(this, Coordinator.EDGE);
18     int height;
19     int width;
20     //tabell som holder åp kummulativpixelintensitetsverider
21     int[] commulative = new int[256];
22     //holder åp pixelverdien hvor bildet skal terskles
23     int threshPoint;
24     final static int ORIGINAL = 1;
25     final static int EDGE = 2;
26     //bildefilen som skal behandles
27     String fileName = "chess.jpg";
28
29
30     public void findThresh()
31     {
32         //antas som rene bekgrunnspixeler
33         int back = commulative[40];
34
35         //Finner ås medianen av de ægjenvrende pixelene
36         int total = commulative[commulative.length-1];
37         int median = (int)Math.ceil( (total - back)/2 );
38         median = median + back;
39         int mark = 0;
40         int counter = 30;
41         boolean found = false;
42
43         while(!found)
44         {
```

```
45         if(commulative[counter]>median)
46         {
47             mark = counter;
48             found = true;
49         }
50         else
51         {
52             counter++;
53         }
54     }
55     System.out.println("Threshpoint_found_on_value:_" + counter);
56     threshPoint = counter;
57 }
58
59
60 public static void main(String[] args)
61 {
62     Coordinator imgFrame = new Coordinator();
63
64     //leser inn filnavn fra brukeren, hvis ingen blir oppgitt
65     //brukes "block.gif"
66     if(args.length > 0)
67         imgFrame.fileName = args[0];
68
69     //Lager en ramme som skal inneholde det orginale bildet
70     imgFrame.frame = new Frame("Original");
71     imgFrame.frame.setSize(1000,1000);
72     imgFrame.frame.setLayout(new BorderLayout());
73     imgFrame.frame.add("Center", imgFrame.canvas);
74     imgFrame.frame.move(0,0);
75     imgFrame.frame.setResizable(true);
76     imgFrame.frame.show();
77
78     //Lager en ramme som skal inneholde det behandlede bildet.
79     imgFrame.frame2 = new Frame("Edge_Picture");
80     imgFrame.frame2.setSize(900,900);
81     imgFrame.frame2.setLayout(new BorderLayout());
82     imgFrame.frame2.add("Center", imgFrame.canvas2);
83     imgFrame.frame2.move(0,350);
84     imgFrame.frame2.setResizable(true);
85     imgFrame.frame2.show();
86     imgFrame.canvas2.paint(imgFrame.canvas2.getGraphics());
87     imgFrame.findThresh();
88     imgFrame.canvas2.thresPicture(imgFrame.threshPoint);
89 }
90 }
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import com.sun.image.codec.jpeg.*;
4 import java.awt.image.BufferedImage;
5 import java.awt.image.DataBuffer;
```

```
7 import java.awt.geom.*;
8 import java.awt.geom.Rectangle2D.*;
9 import java.io.*;
10 import java.awt.image.*;
11
12 /**
13  * Klassen som tar seg av opptegning av bilder
14  */
15 public class Imager extends Canvas implements BufferedImageOp
16 {
17
18     Image img;
19     Coordinator master;
20     int type;
21     int w = 0;
22     int h = 0;
23     WritableRaster raster;
24     int[][][] pixArray;
25     int[] commulativePixelCount = new int[256];
26     boolean done = false;
27
28     public Imager(Coordinator master, int type)
29     {
30         this.master = master;
31         this.type = type;
32     }
33
34     //leser en JPEG eller GIF bildefil
35     public void readImage(String filename)
36     {
37         img = getToolkit().getImage(filename);
38         try
39         {
40             MediaTracker tracker = new MediaTracker(this);
41             tracker.addImage(img, 0);
42             tracker.waitForID(0);
43         }
44         catch (Exception e)
45         {
46             System.out.println("error_in_image-reading:");
47             e.printStackTrace();
48         }
49     }
50
51     //metoden som øhypassfiltrerer bildet. Bruker to sobel filtre.
52     //Et horisontalt og et vertikalt.
53     public void sobel()
54     {
55         int height = img.getHeight(this);
56         int width = img.getWidth(this);
57         int[][][] pixels = new int[width][height][3];
58
59         for(int x=1;x<width-1;x++)
60         {
61             for(int y=1;y<height-1;y++)
62             {
63                 pixels[x][y][0] = Math.abs(pixArray[x-1][y-1][0]
```

```

64         + 2*pixArray[x-1][y][0] + pixArray[x-1][y+1][0]
65         - pixArray[x+1][y-1][0] - 2*pixArray[x+1][y][0]
66         - pixArray[x+1][y+1][0]);
67
68         pixels[x][y][1] = Math.abs(pixArray[x-1][y-1][1]
69         + 2*pixArray[x-1][y][1] + pixArray[x-1][y+1][1]
70         - pixArray[x+1][y-1][1] - 2*pixArray[x+1][y][1]
71         - pixArray[x+1][y+1][1]);
72
73         pixels[x][y][2] = Math.abs(pixArray[x-1][y-1][2]
74         + 2*pixArray[x-1][y][2] + pixArray[x-1][y+1][2]
75         - pixArray[x+1][y-1][2] - 2*pixArray[x+1][y][2]
76         - pixArray[x+1][y+1][2]);
77
78         pixels[x][y][0] = pixels[x][y][0]
79         + Math.abs(pixArray[x-1][y-1][0]
80         + 2*pixArray[x][y-1][0] + pixArray[x+1][y-1][0]
81         - pixArray[x-1][y+1][0] - 2*pixArray[x][y+1][0]
82         - pixArray[x+1][y+1][0]);
83
84         pixels[x][y][1] = pixels[x][y][1]
85         + Math.abs(pixArray[x-1][y-1][1]
86         + 2*pixArray[x][y-1][1] + pixArray[x+1][y-1][1]
87         - pixArray[x-1][y+1][1] - 2*pixArray[x][y+1][1]
88         - pixArray[x+1][y+1][1]);
89
90         pixels[x][y][2] = pixels[x][y][2]
91         + Math.abs(pixArray[x-1][y-1][2]
92         + 2*pixArray[x][y-1][2] + pixArray[x+1][y-1][2]
93         - pixArray[x-1][y+1][2] - 2*pixArray[x][y+1][2]
94         - pixArray[x+1][y+1][2]);
95
96         if(pixels[x][y][0] > 255)
97             pixels[x][y][0] = 255;
98
99         if(pixels[x][y][1] > 255)
100             pixels[x][y][1] = 255;
101
102         if(pixels[x][y][2] > 255)
103             pixels[x][y][2] = 255;
104     }
105 }
106     pixArray = pixels;
107     setPixels();
108 }
109
110 //metoden som lavpassfiltrerer bildet.
111 public void blur()
112 {
113     int height = img.getHeight(this);
114     int width = img.getWidth(this);
115     int[][][] pixels = new int[width][height][3];
116
117     for(int x=1;x<width-1;x++)
118     {
119         for(int y=1;y<height-1;y++)
120         {

```

```

121
122         pixels[x][y][0] = (pixArray[x-1][y-1][0]
123         + pixArray[x-1][y][0] +
124         pixArray[x-1][y+1][0] + pixArray[x+1][y-1][0]
125         + pixArray[x+1][y][0] + pixArray[x+1][y+1][0]
126         + pixArray[x][y-1][0] + pixArray[x][y][0]
127         + pixArray[x][y+1][0]);
128
129         pixels[x][y][0] = (int)Math.ceil(pixels[x][y][0]/9);
130
131         pixels[x][y][1] = (pixArray[x-1][y-1][1]
132         + pixArray[x-1][y][1] + pixArray[x-1][y+1][1]
133         + pixArray[x+1][y-1][1] + pixArray[x+1][y][1]
134         + pixArray[x+1][y+1][1] + pixArray[x][y-1][1]
135         + pixArray[x][y][1] + pixArray[x][y+1][1]);
136
137         pixels[x][y][1] = (int)Math.ceil(pixels[x][y][1]/9);
138
139         pixels[x][y][2] = (pixArray[x-1][y-1][2]
140         + pixArray[x-1][y][2] + pixArray[x-1][y+1][2]
141         + pixArray[x+1][y-1][2] + pixArray[x+1][y][2]
142         + pixArray[x+1][y+1][2] + pixArray[x][y-1][2]
143         + pixArray[x][y][2] + pixArray[x][y+1][2]);
144
145         pixels[x][y][2] = (int)Math.ceil(pixels[x][y][2]/9);
146
147     }
148 }
149 pixArray = pixels;
150 setPixels();
151 }
152
153 //henter ut fargeverdier for pixlene i bildet i form av en matrise
154 public int[][][] getPixels()
155 {
156     int height = img.getHeight(this);
157     int width = img.getWidth(this);
158     int[] maaVaereMed = new int[3];
159     int[] pixel = new int[3];
160     int[][][] pixels = new int[width][height][3];
161
162     for(int x=0;x<width;x++)
163     {
164         for(int y=0;y<height;y++)
165         {
166             pixel = raster.getPixel(x,y,maaVaereMed);
167             pixels[x][y][0] = pixel[0];
168             pixels[x][y][1] = pixel[1];
169             pixels[x][y][2] = pixel[2];
170         }
171     }
172     pixArray = pixels;
173     return pixels;
174 }
175
176 //setter inn fargeverdier for pixlene i bildet ved hjelp av en matrise
177 public void setPixels()

```

```

178     {
179         int height = img.getHeight(this);
180         int width = img.getWidth(this);
181         int[] pixel = new int[3];
182
183         for(int x=0;x<width;x++)
184         {
185             for(int y=0;y<height;y++)
186             {
187                 pixel[0] = pixArray[x][y][0];
188                 pixel[1] = pixArray[x][y][1];
189                 pixel[2] = pixArray[x][y][2];
190                 raster.setPixel(x,y,pixel);
191             }
192         }
193     }
194
195     //tersklerer bildet ut i fra en gitt lys-verdi
196     public void thresPicture(int value)
197     {
198         int height = img.getHeight(this);
199         int width = img.getWidth(this);
200         int[] pixel = new int[3];
201
202         for(int x=0;x<width;x++)
203         {
204             for(int y=0;y<height;y++)
205             {
206                 if(pixArray[x][y][0] < value ||
207                 pixArray[x][y][1] < value ||
208                 pixArray[x][y][2] < value)
209                 {
210                     pixArray[x][y][0] = 0;
211                     pixArray[x][y][1] = 0;
212                     pixArray[x][y][2] = 0;
213                 }
214             }
215         }
216         setPixels();
217     }
218
219     //genererer en vektor som holder åp den kumulative
220     //pixel-intensitetsverdiene
221     public int[] commulativePixelCount()
222     {
223         int height = img.getHeight(this);
224         int width = img.getWidth(this);
225         int maxPCV;
226
227         int[] pixelCount = new int[256];
228         commulativePixelCount = new int[256];
229
230         for(int x=0;x<width;x++)
231         {
232             for(int y=0;y<height;y++)
233             {
234                 maxPCV = 0;

```

```

235         for(int p = 0; p<3; p++)
236         {
237             if(pixArray[x][y][p]> maxPCV )
238                 maxPCV = pixArray[x][y][p];
239         }
240         for(int p = 0; p<3; p++)
241         {
242             pixArray[x][y][p] = maxPCV;
243         }
244         pixelCount[maxPCV]++;
245     }
246 }
247 setPixels();
248 for (int i = 0; i < pixelCount.length; i++ )
249 {
250     if (i == 0)
251         commulativePixelCount[0] = pixelCount[0];
252     else
253         commulativePixelCount[i] =
254             pixelCount[i] + commulativePixelCount[i-1];
255 }
256 return commulativePixelCount;
257 }
258
259
260 //metoden som tegner opp bildene...åbde for orginal og kant.
261 public void paint(Graphics g)
262 {
263     Graphics2D g2 = (Graphics2D) g;
264     Dimension d = getSize();
265     g2.setBackground(getBackground());
266     g2.clearRect(0, 0, d.width, d.height);
267     $g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
268         RenderingHints.VALUE_ANTIALIAS_ON);
269     g2.setRenderingHint(RenderingHints.KEY_RENDERING,
270         RenderingHints.VALUE_RENDER_QUALITY);$
271     w = d.width;
272     h = d.height;
273     readImage(master.fileName);
274     master.height = img.getHeight(this);
275     master.width =img.getWidth(this);
276     BufferedImage bi = (BufferedImage) createImage(w, h);
277     raster = bi.getRaster();
278
279     Graphics2D big = bi.createGraphics();
280
281     big.drawImage(img, 0, 0, null);
282     getPixels();
283
284     if(type == Coordinator.ORGINAL)
285     {
286         master.frame.setSize(master.width + 5, master.height + 30);
287         g2.drawImage(bi, 0, 0, this);
288     }
289     else if(type == Coordinator.EDGE)
290     {
291         blur();

```

```
292         sobel();
293         int[] temp = commulativePixelCount();
294         if(temp[255] > 1000)
295         {
296             master.commulative = temp;
297         }
298         if(master.threshPoint>0)
299         {
300             thresPicture(master.threshPoint);
301         }
302         master.frame2.setSize(master.width + 5, master.height + 30);
303         g2.drawImage(bi,0,0,this);
304     }
305 }
306
307 //Bare tull som åm ævre med under denne linja...
308 public Rectangle2D getBounds2D(BufferedImage img)
309 {
310     Rectangle2D.Double rect = new Rectangle2D.Double(0, 0, (double)w, (double)h);
311     return rect;
312 }
313 public BufferedImage createCompatibleDestImage(BufferedImage img, ColorModel col)
314 {
315     return img;
316 }
317 public Point2D getPoint2D(Point2D src,Point2D dest)
318 {
319     return dest;
320 }
321 public RenderingHints getRenderingHints()
322 {
323     return null;
324 }
325 public BufferedImage filter(BufferedImage src, BufferedImage dest)
326 {
327     dest = src;
328     return dest;
329 }
330 }
```