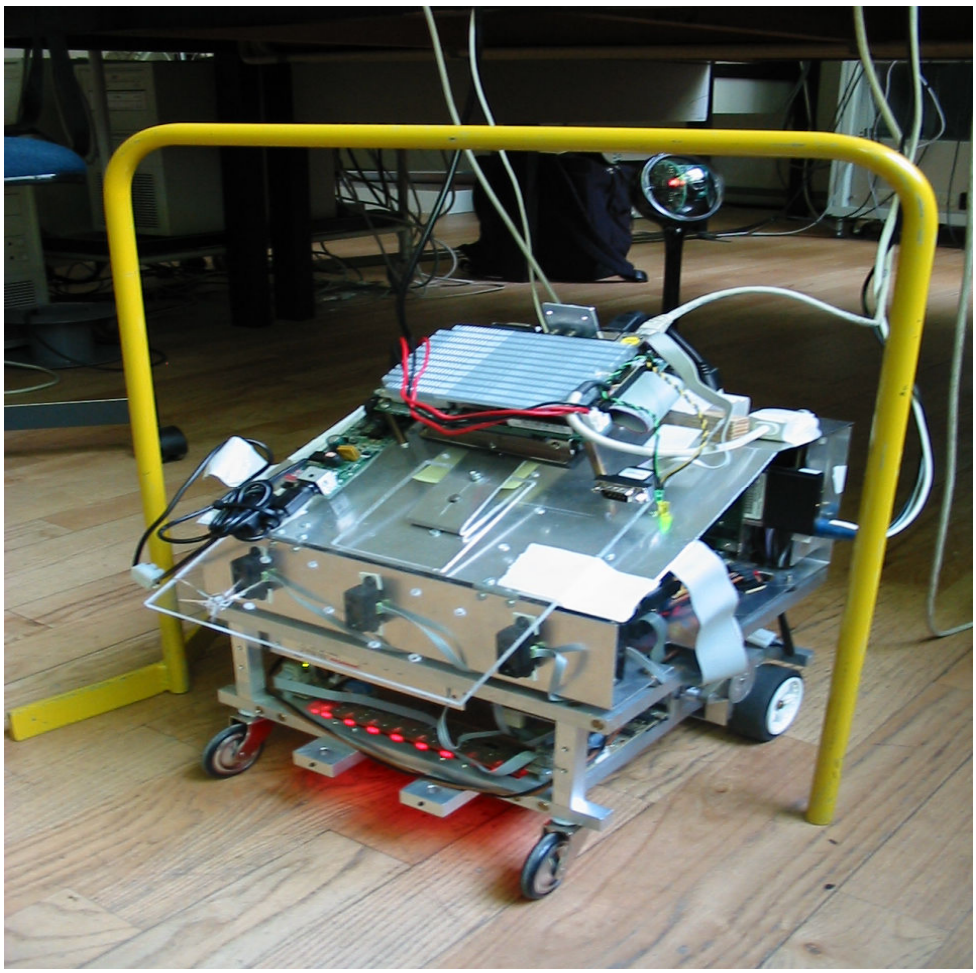
Vision til navigering af RoboCup-robot

13-Ugers Specialkursus Afleveret: 2/6-2004



Udført af:

s973989 Allan Krogh Jensen

Under vejledning af:
Christian Andersen og Nils Andersen
Automation, Ørsted•DTU

Indholdsfortegnelse

Indledning

Jeg har længe tænkt på at bygge en RoboCup-robot til DTU's årlige konkurrence. Efter at have fulgt kurset 02501 Billedanalyse, vision og computergrafik, fik jeg den idè at robotten kunne navigere igennem banen, ved brug af stereovision. Robotten skulle måle dens position i forhold til objekterne på banen, hvorved den upræcise odometri i teorien burde være overflødig. Robotten ville under turen på banen opbygge en simplificeret 3D model af banen, som så gerne skulle kunne bruges til at klare alle forhindringer på banen.

Da et sådan projekt ville være ganske relevant for min uddannelse som civilingeniør i automation, ønskede jeg at oprette et specialkursus. Da projektet omhandlede metoder som aldrig før har været afprøvet på en RoboCup-robot, blev det i første omgang til et indledende 3-ugers kursus til indledende undersøgelser, og herefter dette 13-ugers kursus. Kurset er afgrænset således at jeg kun skal koncentrere mig om softwaren til navigering af robotten, min kammerat sørger for at bygge en fjernstyret bil om til en robot, med det elektronik som er nødvendigt.

Billedanalyse og især stereovision kræver meget computerkraft. Derfor var det nødvendigt at skaffe en tilpas kraftig computer, som samtidig fyldte/vejede minimalt og uden det store strømforbrug, således at den kunne monteres på en RoboCup-robot. Jeg søgte derfor en sponsor som havde det ønskede udstyr, og fandt DataRespons, som har donneret: En NEXCOM EBC365 SBC med VIA CPU på 667 Mhz, En 20 Gb harddisk, To Logitech QuickCam Sphere webcams med pan/tilt funktionalitet, samt et IO-modul til SBC'en med 16 digitale I/O. Betingelsen er at robotten opkaldes efter dem, bliver udstyret med deres logo, og at vi bærer deres reklame T-shirts til konkurrencen.

Problemformulering

Som tidligere nævnt er dette kursus fortsættelsen på det indledende 3-ugers kursus til at undersøge mulighederne for at bruge vision til navigering af en Robocup-robot. 3-ugers kurset viste tydeligt at software udvikling til vision tager meget længere tid end først antaget. Men samtidig gav det også endnu mere ”blod på tanden” til at prøve kræfter med vision software. Der var derfor god grund til at fortsætte med dette kursus.

Kursets mål er derfor nu at udvikle et stereovision system, ttil navigering af en RoboCup-robot igennem roboCup-banen. Nærmere detaljer om systemts opbygning og struktur findes i næste kapitel.

Til kurset blev i første omgang udarbejdet følgende tidplan:

Uge	**Delopgave**
5	Optimering af billedfiltret med MatLab
6	Forbedring af EdgeDetection-laget
7	Styring af Pan/tilt webcams
8	EdgeLine-laget
9	Polygon-laget
10	Polygon-laget
11	Photogtrametry-laget
12	Samling af robotten
13	MovementDetection-laget
14	3DObjectDetection-laget
15	Afsluttende tests af robotten
16	Afsluttende tests af robotten
17	RoboCup-finale
18	Test på SMR
19	Dokumentation
20	Dokumentation
21	Dokumentation og aflevering af rapport

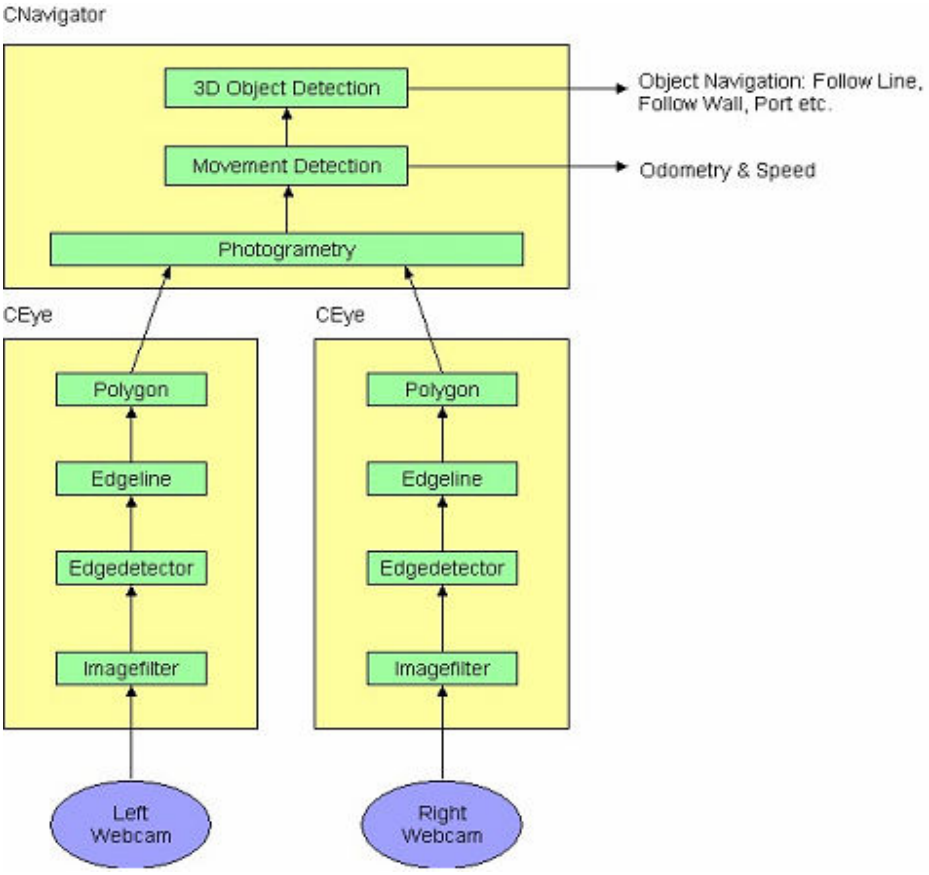
Pga. det enorme tidspres mht. til at få noget praktisk til at virke inden projektafleveringen, blev afleveringsfristen udsat til 2/6-2004.

Eftersom udviklingen stadig tog længere tid end først antaget, og der kun var en mdr. til RoboCup-finalen, blev målet endnu engang afgrænset. Således at systemet blot skal kunne finde midten af den nærmeste RoboCup-port i synsfeltet, og så styrer robotten efter det. På baggrund af denne målsætning er udarbejdet følgende revideredetidsplan:

Uge	Dato	Delopgave
14	29/3 – 2/4	Port detektering og navigering
15	5/4 – 9/4	Software til at køre igennem en port. Færdigsamling af hardware
16	12/4 – 16/4	Kommunikation med hardware . Test af kørsel igennem port.
17	19/4 – 23/4	Afsluttende test og justeringer. Konkurrence
18	26/4 – 30/4	Test på SMR robot
19	3/5 – 7/5	Dokumentation
20	10/5 – 14/5	Dokumentation og rapport aflevering

1. Den grundlæggende idé

Softwaren som skal sørge for vision-navigeringen skal opbygges som en mængde lag oven på hinanden. Nederst indlæses rå billeder fra de to webcams, øverst kan aflæses i hvilken retning og med hvilken hastighed robotten skal bevæge sig i for f.eks. at følge en væg eller køre igennem en port. De er arrangeret således at lagene bliver mere og mere "inteligente" jo højere de ligger. Et diagram for de enkelte lag i softwaren er vist herunder:



Figur 1-1: Overordnet diagram for systemets lagdelte struktur

Modellen består af et overordnet CNavigator objekt som består af to ens CEye objekter, der hver repræsentere et "øje" som bruges til den stereoskopiske opmåling. Det rå billede fra webcamet sendes først ind i et specielt billedfilter hvor det omdannes til et simplere billede uden unødvendige detaljer, som f.eks. højlys og mindre pixelfejl. Nærmere beskrevet bliver det rå billede lavet om til et billede med færre forskellige farver (CCartoonImage).

Ved at have billedet repræsenteret på denne måde, burde det nu være nemmere at finde de rette kontraster/kanter i billedet. Idet mindre textur forskellige i billedet er udglattet. Kanterne skal findes i Edgedetector-laget, og gemmes som et

binært billede. Udfra dette binære-billede af kanterne, kan kanterne simplificeres yderligere, ved istedet at beskrive kanterne med en række kantlinier. Således at en hvid streg i billedet f.eks. beskrives med en kantlinie på hver side, eller evt. flere linier på den ene side, hvis stregen buer. Denne operation foregår i Edgelline-laget.

Beskrivelsen af billedet kan forenkles endnu en gang, ved at samle kantlinierne i 2D polygoner. Således at hver polygon består af nogle kantliner med det tilfælles at de omkranser samme objekt i billedet. Et polygon består således af et sæt kantlinier og en farve som er den mest hyppige på det omkransede objekt. På dette niveau kan der også indbygges et filter således at der kun dannes polygoner med de farver som der er interesse for (f.eks gul (porte) og hvid (streg) osv.).

Efter at begge "øjne" (CEye-objekter), har fundet et sæt polygoner som beskriver hver deres billede, kan selve den stereoskopiske process begynde. Denne process kaldes også fotogrametri og foregår i Photogrametry-laget. Normalt er det store problem ved automatiseret fotogrametri at få computeren til at finde ud af hvilke pixels der hører sammen i de to billeder. Ved at have simplificeret billedet til polygoner fremfor pixels, burde denne process blive en del mere overkommelig og hurtig. Ideen er at polygonerne i de to billeder sammenlignes udfra ca. possitioner, farver, antal kanter, og nabo polygoner. De som så passer bedst sammen pares, og sendes videre til de fotogrametriske beregninger.

Efter de fotogrametriske beregninger er der dannet 3D polygon objekter, til at beskrive det billede som "øjnene" ser tilsammen. 3D polygon objekterne er blot 2D Polygoner med (x,y,z)-koordinater for hvert hjørne i polygonet. Der er ikke tale om en egentlig 3D model af de objekter som ses på billedet.

Ved at beregne foreskellen på to 3D polygon billeder, kan man finde ud af hvordan robotten har bevæget sig imellem de to billeder. Hvis tiden imellem også kendes, som det er meningen her, kan robottens hastighed også beregnes. Disse beregninger skal foregå Movement Detection-laget. Outputtet fra dette lag kan hvis det bliver tilpas præcist bruges som odometri parametre, dvs. robottens absolutte position. Hastigheden kan i teorien bruges som feed-back til styring af motoren, men vi vælger nu nok også at indbygge et tachometer på hjulene for en sikkerhedsskyld.

Oprindelig var det meningen af navigatoren i sidste ende skulle generere en komplet 3D-model af banen, som den kunne bruge som kort til navigering. Men det bliver uden tvivl en alt for stor mundfuld til dette projekt. Derfor er øverste lag "blot" et 3D Object Detectiong-lag til at følge specifikke objekter på RoboCup-banen. Såsom streger, vægge, porte osv.

2. ImageFilter-laget

Softwaren ønskes opbygget så objekt orienteret som muligt for at give den bedste struktur og overblik over systemet. Det er tanken at billedfiltret i første omgang skal omfatte pixelvis ændring af inputbilledet, derfor kan stort set hele billedfiltrets funktionalitet samles i et objekt som repræsentre en output pixel fra billedfiltret (CCartoonPixel). For at disse pixels kan fungere som et helt billede skal de samles i et overordnet output billede (CCartoonImage). Klasserne har fået de navne de har fordi outputtet gerne skulle ligne tegninger fra en tegnefilm, dvs. med en yderst begrænset pallette. Eftersom der findes et billedfilter i hvert CEye-objekt, skal der være et CCartoonImage i CEye-klassen. Et klassediagram for den software som skal konstrueres til billedfiltret er vist herunder. Kildekoden til CCartoonImage og CCartoonPixel findes i appendix.

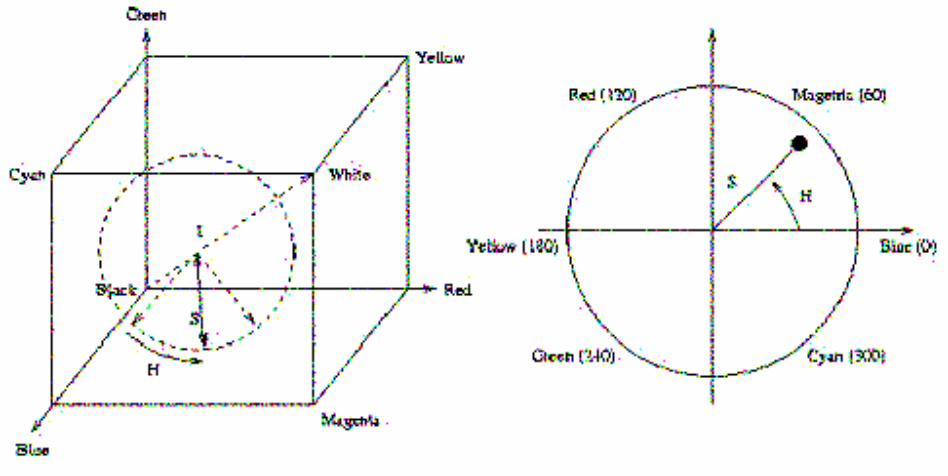
Figur 2-1: Klassedigram for klasserne der indgår i billedfiltret

Softwaren bygges oven på en eksisterende testapplikation til webcam-driveren, udviklet af Christian Andersen. Denne applikation har et tekstbasseret brugerinterface og er istand til at gemme billederne fra webcammet og filtret i BMP-format, så de derefter kan hentes og ses. Webcam-driveren er fundet på http://www.smcc.demon.nl/webcam/ . Som webcam bruges et Logitech QuickCam 3000, indtil dem med pan/tilt-funktionalitet bliver leveret. Testapplikationen giver mulighed for at hente billeder fra kameraet enten som rå billeder i YUV/IUV-format via URawImage klassen, eller som mere tilpassede billeder i RGB-format fra UImage640 klassen. Billederne kan kun gemmes som BMP-format hvis de findes som UImage640-objekter, derfor har jeg i første omgang valgt at bruge sidst nævnte klasse både til inputbilledet og til outputbiledet. Det er muligvis ikke det hurtigste da billedet først skal konverteres fra UrawImage til Uimage640 før det sende ind i filtret, men det kan ændres hvis det bliver nødvendigt. Det vigtigste lige nu er at få et billedfilter som virker.

2.1. Teorien bag billedfiltret

Billedfiltret skal gøre robotten immun overfor ændringer i lysstyrken, samt for mindre textur ændringer. Når billedet kommer ind i filtret er hver pixel beskrevet i RGB-formatet, dvs. mængden af rød, grøn og blå beskrives med hver sin byte-værdi. Ved at konvertere hver pixel til IHS-format, beskrives farven istedet udfra de tre værdier: intensity (lysstyrke), hue (anstrøg, farvegruppe) og saturation (mætning). Dette format er nemmere at arbejde med i et billedfilter, idet de bedre beskriver den menneskelige opfattelse af farver. Lysstyrken svarer til hvor meget hvidt lys der lyses med på den pågældende farve. Anstrøget betegner med en vinkel hvilken farvegruppe der er tale om, f.eks. Rød, gul eller blå. Mætning betegner koncentrationen af pågældende farve, jo mindre denne værdi er jo mere grålig er den. Hvis robotten skal være imun overfor lysstyrke og små farve

ændringer i billedet, burde det altså være muligt ved kun at beskrive hver pixel med dens hue-værdi. En grafisk model for IHS-formatet er vist herunder:



Figur 2-2: IHS farve formatet

Der er dog et problem ved denne teori: robotten skal også kunne se gråtoner for at kunne følge sort og hvide streger på et gråt gulv. I RGB-formatet er gråtoner de steder hvor RGB-værdierne er ens, og lysstyrken af gråtonen bestemmes af denne værdi. I IHS-format giver hue-værdien ingen antydning af hvilken gråtone der er tale om, men er istedet ligesom saturation bare støj, hvis der er tale om en gråtone. Det er kun intensiteten som angiver hvilken værdi gråtonen har. Derfor skal filtret tage højde for hue hvis der er tale om en farve og tage højde for intensiteten hvis der er tale om en gråtone. Det burde heller ikke være noget problem for det burde være nemt at finde ud af om inputtet er en farve eller en gråtone: hvis saturation er 0 er det en gråtone ellers er det en farve. Der vil dog altid være lidt støj på saturation-værdien, selvom det er en gråtone. Det kan bl.a. Skyldes det lys som falder på overfladen. Derfor er det nødvendigt at definere et område for saturation hvor værdien antages at være en gråtone.

Value (byte)								***Color***
7	***6***	***5***	***4***	***3***	***2***	***1***	***0***	
0	0	Intensity Value (0 to 63)						Graylevel (64 levels)
0	1	Hue Value (64 to 255)						Color (192 levels)
1	0							
1	1							

Det er samtidig et mål at outputbiledet fra filteret skal fylde så lidt som muligt, for at lette den videre processering. Derfor ønsker jeg at gemme hver pixel i en byte, organiseret som vist herover. Således bliver der mulighed for at beskrive 64

forskellige gråtoner og 192 forskellige anstrøg (farvergrupper). En outputpixel fra filtret er som sagt implementret i klassen CCartoonPixel, hvor ovennævnte byte-værdi er en privat variabel. Kildekoden findes i appendix.

En pixel indlæses med inputRGB()-funktionen hvis det er i RGB-format, og med inputIUV()-funktionen hvis det er i IUV/YUV-format. Begge funktioner får pixelværdierne ind som et UPixel-objekt, der kan være i enten RGB eller YUV format. Det er så op til brugeren at sørge for at det er det rigtige format som sendes ind. UPixel-klassen er udviklet af Christian Andersen, og allerede en del af testapplikationen. Begge funktioner omdanner deres input-format til IHS-formatet som så sendes til funktionen inputIHS(), som står for selve filtreringen. Når en filtreret pixelværdi skal hentes ud, foregår det på samme måde: getIHS()-funktionen henter output IHS-værdien og returnere den til enten getRGB() eller getIUV() funktionen. Udover disse hovedfunktioner er der en række hjælpe funktioner i klassen, som vil blive beskrevet senere.

Som filtret er implementeret nu er det kun RGB funktionerne der anvendes, eftersom billedet indlæses fra et UImage640 objekt. Input RGB-værdierne ligger som byte værdier i et UPixel objekt, og konverteres til decimalværdier i IHS-formatet. Det foregår ved at gange dem med en transformationsmatrix, som beskrevet på side 39 i bogen “Image analysis, vision and computer graphics” af Jens Michael Carstensen, DTU. Til matrix-operationerne anvendes Umatrix4-klassen udviklet og gennemtestet af Christian Andersen. De fundne værdier sendes så videre til inputIHS()-funktionen.

I inputIHS()-funktionen tages der i første omgang stilling til om inputtet er en gråtone eller en farve. Dette bestemmes udfra saturation-værdien, og den returnerede værdi af getMaxGrayLevelSat()-funktionen. Funktionen returnere grænseværdien imellem en farve og en gråtone, og blev i første omgang sat til en konstant tærskelværdi. Hvis input saturation-værdien er mindre end værdien af getMaxGrayLevelSat(), klassificeres den som en gråtone, ellers som en farve. Hvis det er en gråtone sættes de to MSB i value til “00”, og de resterende bits bruges til at beskrive intensiteten som et niveau fra 0 til 63. Hvor 0 svarer til 0 og 63 svarer til $\sqrt{3}$ (1.732...). Hvis inputtet derimod er blevet klassificeret som en farve, læses kun hue-værdien (vinklen) og den afrundes til en af de 192 mulige niveauer. Hvor 0 svarer til 0 radianer, og 192 svarer til 2π radianer.

På næsten samme måde virker getIHS()-funktionen. Den tester først om value er gråtone eller farve ved at kalde funktionen isGrayLevel(). Funktionen returnere true hvis value er mindre end 64 (to MSB er “00”). Hvis det er en gråtone bestemmer value intensiteten af gråtonen, hue og saturation sættes til nul, hvorved outputtet bliver en perfekt gråtone uden noget “støj”. Hvis der er tale om en farve betegner value-63 en hue værdi fra 0 til 2π. Intensity og Saturation sættes her til nogle konstanter, som er henholdsvis INTENSITY_OUTPUT og SATURATION_OUTPUT. De er ens for alle farver der kommer ud af filtret, og skal blot vælges således at de kan repræsenteres i RGB-kuben.

Den videre omregning fra IHS til RGB foretages af getRGB()-funktionen, som blot foretager den inverse transformation i forhold til inputRGB()-funktionen. getIUV()-funktionen og inputIUV()-funktionen anvendes som sagt ikke pt. Men er ligesom inputRGB() og getRGB() blot transformationsfunktioner til og fra IHS-formatet. Derfor vil disse funktioner ikke blive beskrevet nærmere her, se kildekoden for yderligere detaljer.

For at danne et billede med disse CCartoonPixels, anvendes CCartoonImage-klassen (se kildekoden i appendix). Hvor selve billedet er gemt i et array af CCartoonPixels, kaldet image. I klassen findes funktioner til at indlæse et billede i form af et UImage640-objekt i filtret, og til at hente output-billedet ud i samme format. De er kaldt henholdsvis inputImage() og getImage(). De overfører ganske enkelt billedet et pixel ad gangen ind og ud af filtret.

2.2. Problemerne med billedfiltret

Til at teste billedfiltret har jeg i første omgang lavet et testbillede, som er vist forneden til venstre, set af webcammet. Testbilledet er tegnet i Paint, hvor det er muligt at angive farverne i IHS-værdier, og udskrevet på et A4 ark. Det består af 24 farvede felter og 8 gråtonede felter. De farvede felter er arrangeret således at hue værdien forøges ud af x-aksen, og saturation-værdien forøges opad y-aksen. Hvis filtret virker optimalt skal outputtet ikke tage højde for de forskellige saturation-værdier, men udelukkende hue-værdierne, hvorved de tre felter som ligger i samme kollonne skal få samme farve. Gråtonefelterne har blot forskellig intensitet og har som hovedformål at teste om filtret kan adskille gråtoner og farver.

Figur 2-3: Filter output af testbilledet uden særlig belysning

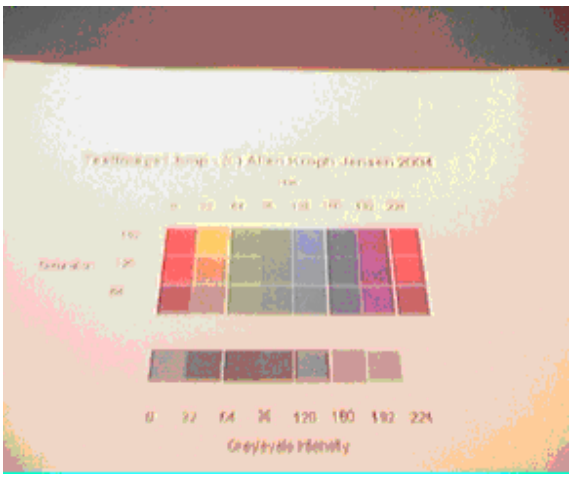
Figur 2-4: Filter output af testbilledet med stærkere kunstig belysning.

Som det ses vises stort set hele farvespektraet, dog er der en del farver som fejlagtigt klassificeres som gråtoner. Funktionen som klassificerer hvorvidt en farve er en farve eller en gråtone, er getMaxGraylevelSat(). Afhængig af hvilken hue- og intensity-værdi der medsendes soim argumenter, returnere den en

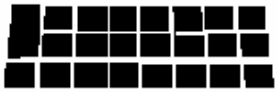
passende saturation-grænseværdi. Hvis pixelen er over denne værdi er det en farve ellers er det en gråtone. Grænseværdierne er tilnærmet efter en lang række justeringer, og er stadig ikke helt perfekte, som det også ses på billederne herover er det stadig meget afhængigt af belysningsstyrken og typen, hvor godt det virker. Når testbilledet belyses kraftigt, opfatter den pludselig højlyset på papiret som orange i stedet for en gråtone. Funktionen kræver altså yderligere forbedringer.

2.3. Optimering af billedfiltret med MatLab

I princippet kan det være nødvendigt med 65536 forskellige grænseværdier, hvis der skal være een for alle kombinationer af hue- og intensity-værdier. Men så burde getGrayLevelSat() også virker i alle situationer. For at afprøve denne idé, ønsker jeg at bruge MatLab til at generere en fil med de 65536 forskellige grænseværdier. Filen skal så herefter kunne indlæses i et et array i billedfiltret, hvorfra getMaxGrayLevelSat() så kan hente den ønskede grænseværdi, udfra de givne hue- og intensity-værdier.



Figur 2-5: Råt billede af testbilledet med kunstifbelysning (P2_Input.bmp).



Figur 2-6: Farvemaske til inputbilledet (P2_Mask.bmp)

For at kunne udarbejde disse grænseværdier, skal der indlæses en lang række eksempler på pixels, som skal fortolkes som farver. Til det formål bruges fire rå input-billeder (P1_Input.bmp, P2_Input.bmp, P3_Input.bmp og P4_Input.bmp), af henholdsvis testbilledet og nogle RoboCup-effekter, i forskellig belysning. Disse fire billeder indlæses direkte i MatLab, som BMP billeder, i hver sin matrice.

For at kunne filtrere de områder af billeder fra, hvor der er farver, laves der billed-masker for hver af billederne (P1_Mask.bmp, P2_Mask.bmp. osv.). Farvemaskerne laves manuelt i Paint, ved at overtegene de pixels der skal klassificeres som farver, med sort. Alle grålige neuancer overmales med hvid. Farvemaskerne gemmes tilslut som 1-bit BMP billeder (Sort/Hvid), og indlæses i hver sin logiske matrice (true/false) i MatLab.

Figur 2-7: Råt billede af RoboCup-effekter (P3_Input.bmp). Kun porten skal klassificeres som en farven, resten er gråtoner.

Figur 2-8: Farvemaske til input-billedet til venstre (P3_Mask.bmp)

Med denne store mængde rådata inde i MatLab kan processen begynde. Alle pixels i de fire inputbilleder, skal sortes således at farve-pixels indsættes i en liste (ColPix), og gråtone-pixels i en anden (GLPix). Denne sortering foregår ved at udmaske billedet med henholdsvis farvemasken (til ColPix) og den inverterede farvemaske (til GLPix). Udmaskningen foretages i funktionen mask(). Den medtager et inputbillede-matrix og en maske-matrice som argument og returnere så en liste med alle de pixels, som blev overlappet af masken. Funktionen gennemløber ganske enkelt alle pixels i inputbilledet, hvis pågældende pixel-position så er true i masken, tilføjes pixelen til output listen. Se også appendix.

Efter at have kørt mask() på alle inputbilleder, med både inverteret og ikke-inverteret maske. Skal alle pixels (både i ColPix og GLPix), konveteres til IHS-værdier. Konverteringen sker med den indbyggede MatLab funktion: rgb2hsv().

Grænseværdierne, eller grænsefladen som den også kan betegnes skal selvfølgelig ligge imellem: max. saturation for en gråtone med de pågældende (I, H) værdier, og min saturation for en farve med de samme (I, H)-værdier. Disse to flader ligger henholdsvis i matricerne CPixMin og GPixMax. CPixMin (min. saturation værdi for alle farver i (I, H)-planet) findes ved at køre funktionen: min_sat() på ColPix matricen, som vist herunder. Tilsvarende køres max_sat()-funktionen på GLPix matricen for at finde GPixMax-fladen.

```
CPixMin = min_sat(ColPix);
GPixMax = max_sat(GLPix);
```

Både min_sat og max_sat virker ved at de gennemløber hele input listen, og opdatere output matricen løbende, indtil alle pixels i input-listen er gennemset. Udfra den enkelte pixels I og H værdi indlæses den aktuelle max/min saturation, for pågældende (I, H)-kombination. Hvis pixlens S værdi er størrer end denne (max_sat) eller mindre end denne (min_sat), opdateres output matricen på

pågældende plads. Se også implementeringen i appendix.

Når CPixMin og GPixMax er fundet, kan selve seperationsfladen findes. Det gøres med funktionen GLMSat(). Den tager CPixMin og GPixMax som input. Som output kommer dels seperations fladen (ss-matricen), og en DAT-fil som kan indlæses direkte i billedfiltret.

```
ss = GLMSat(CPixMin, GPixMax);
```

Funktionen gennemgår de to flader (CPixMin og GPixMax) udfra alle (I, H)-kombinationer. For hvert punkt beregnes en difference (d) imellem fladerne. Hvis denne værdi er 1, betyder det at (I, H) kombination ikke var til stede blandt input billederne, der blev hverken fundet min. eller max. værdi for denne kombination. Derfor sættes seperationspunktet også til 1.
Hvis differencen er positiv, dvs. CPixMin er størrer end GPixMax, er alt som det skal være og seperationspunktet lægges lige imellem dem.
Hvis derimod differencen er negativ, kan der ikke opstilles et seperationspunkt imellem fladerne, eftersom de overlapper hinanden på pågældende punkt. Der sættes seperationspunktet også her til 1, hvilket betyder at alle pixels med denne (I, H)-kombination klassificeres som gråtoner. Når hele ss-fladen er fundet gemmes den i filen ”filter1.dat”.

Anvendelse af disse 65.000 grænseværdier i billedfiltret, gav et svagt forbedret resultat. Men dog stadig en del pixels som blev fejl klassificeret. Fejlklaccificering skyldes nok primært at ikke all (I, H)-kombinationer var representeret blandt inputbillederne, og derfor pr. definition klassificeres som gråtoner, selvom de egentlig er farver. En anden grund til fejlklassificeringen kan stadig være den meget varierende belysning. Blandt inputbillederne var der nemlig kun to forskellige belysningstyper representeret, dagslys fra vinduet og kunstig lys fra en glødelampe.

Ulempen ved at anvende en array på 65.000 floating-point værdier, er selvfølgelig at det sluger en hel del hukommelse. Taget i betragtning af den begrænsede forbedring af klassificeringen, valgte jeg derfor at gå tilbage til den tidligere omtalte klassificerings metode. MatLab metoden vil dog uden tvivl kunne forbedre klassificeringen, hvis der blev brugt flere forskellige inputbilleder.

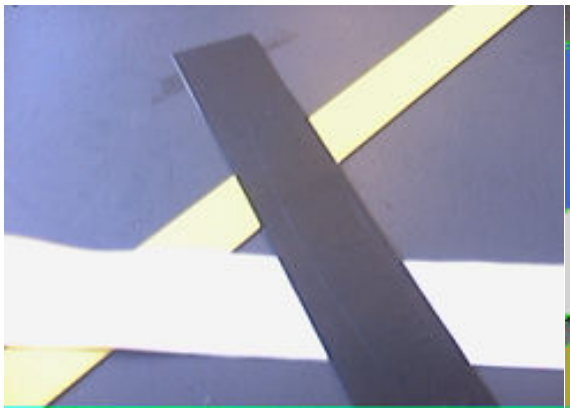
3. EdgeDetection-laget

Idéen med at lave et smart billedfilter, som frasorterede unødvendige detaljer i billedet, var netop at den følgende kantdetekteringsproces skulle blive ganske enkel. Det skulle ganske enkelt være sådan at hvis to nabo-pixels var forskellige (på deres value-værdi) så var der en kant imellem dem. Eftersom jeg vidste at filteret ikke var helt perfekt, ændrede jeg dog denne betingelse til at to nabo-pixels nu skulle være "tilpas" forskellige for at der skulle være en kant imellem dem. Der skulle med andre ord være en tæskelværdi for hvor meget de enkelte pixels skulle variere. Det lød jo stadig meget enkelt men skulle også vise sig langt fra at være tilstrækkeligt, til at opnå en god kantdetektering.

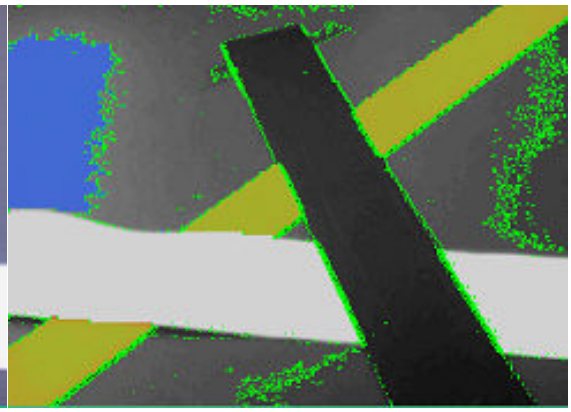
3.1. Den naive kantdetekteringsalgoritme

Kantdetekteringen foretages af findEdges()-funktionen i CCartoonImage-klassen, efter at alle pixels er indlæst i filtret. Funktionen gennemløber alle pixels i output-billedet, og tester om der skulle være en kant imellem den aktuelle pixel og en af de tre ovenliggende eller den forrige pixel. Dette gøres ved at kalde isEdge()-funktionen i pågældende pixel, med den aktuelle nabo-pixel som argument. Hvis der er en kant imellem de pågældende pixels, sættes den aktuelle position til true i kant billedet (edgeImage).

isEdge()-funktionen tester først om den ene pixel er gråtone og den anden farve, hvis det er tilfældet klassificeres området som en kant, og den returnerer derfor true. Hvis de begge er gråtoner, antages det at der er en kant hvis deres værdier afviger med 2 eller mere. Hvis de begge er farver, antages det at der der en kant hvis de afviger med 8 eller mere. Disse tærskelværdier er i første omgang hardcoded i funktionen. For at vise hvor algoritmen finder kanterne, tegnes edgeImage oven på output-billedet fra filtret, således at kanterne bliver grønne.

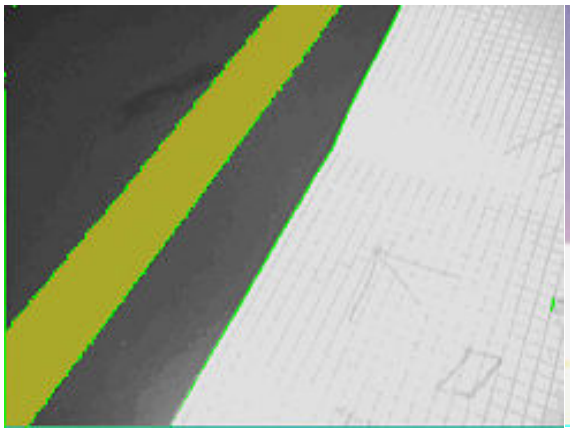


Figur 3-1: Testopstilling med relevante RoboCup-effekter, som det ser ud før det kommer igennem filtret. Billedet er taget med almindeligt dagslys fra vinduet.



Figur 3-2: Testopstillingen efter den har været igennem filtret og kantdetekteringen. Det blå stammer fra højlyset.

Til at teste kantdetekteringen lavede jeg en testopstilling (vist herover til venstre), bestående af en sort og hvid tape af den type som bruges til RoboCup-konkurrencen, samt en gul lineal med nogenlunde samme farve som de porte der findes på RoboCup-banen. Outputtet efter kantdetektering er vist på billedet til højre. Som det ses er den ikke helt så perfekt som man havde håbet på. For det første er den del af bordet, med højlyset, blevet til en blå sky, for det andet er der fundet en hel del kanter, især ude til højre, hvor der ikke burde være nogen. For det tredie er mange af de kanter som er fundet rigtigt, yderst perforeret og næsten manglende. Det gælder især imellem sort og gråt og imellem gul og hvid. Den blå sky kan forklares med at bordet med det blåtte øje syntes at være grå-blåt, hvilket især fremhæves med dagslyset fra vinduet. Dette problem er igen et spørgsmål om forbedring af getMaxGraylevelSat()-funktionen, som tidligere omtalt. Jeg forsøgte også at ændre findEdges()-funktionen således at den istedet tester alle 8 naboer til en pixel for kanter, men det gjorde bare at der kom endnu flere fejlplacerede kanter.



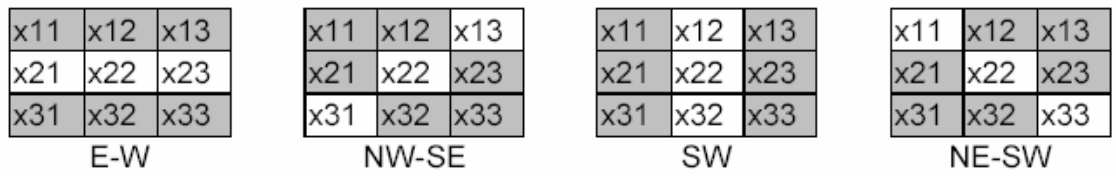
Figur 3-3: Anden testopstilling af gul lineal og et hvidt papir. Her virker kantdetekteringen tilsyneladende perfekt.

Figur 3-4: Testopstillingen fra før, med kunstig belysning fra en arkitektlampe.

Det viste sig dog efter nogle forsøg at det ikke var i alle situationer kantdetekteringen virkede lige dårligt. Jo mere hvidt der findes på billedet jo bedre. På billedet herover til venstre består ca. halvdelen af billedet af et hvidt papir, og som det ses er kantdetekteringen nærmest perfekt. Fænomenet skyldes at webcammet automatisk regulerer lysstyrken i billedet således at ingen pixels går i mætning, dvs. hvis billedet er meget lyst bliver de mørkeste pixels gjort endnu mørkere, hvorved kontrasten forbedres, og det bliver nemmere at finde kanter. Kantdetekteringsalgoritmen kunne derfor i teorien forbedres ved blot at påklistre hvide skyklapper på en del af kameraets udsyn. Det er dog nok ikke holdbart i længden, så det er nok en bedre idè at forbedre algoritmen eller finde en ny, som man ved virker.

3.2. Gamma Ratio Edge Detection-algoritmen (GRED)

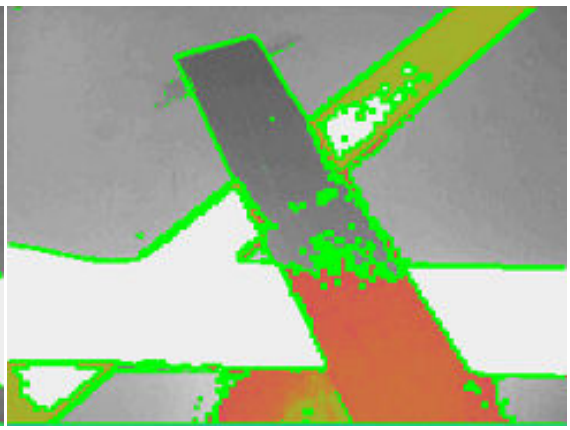
Jeg måtte se i øjnene at kantdetekteringen ikke kunne løses helt så simplet som jeg havde håbet på. Derfor ønskede jeg nu at finde en helt ny algoritme til løsning af problemet. I bogen omtales Gamma Ratio Edge Detector algoritmen, så jeg besluttede mig for at implementere og afprøve den, for at se om den skulle være bedre. Algoritmen er implementeret i funktionen findEdgesGRED() i CCartoonImage-klassen, og kan indsættes som direkte erstatning for den før omtalte findEdges(). Algoritmen gennemløber, ligesom findEdges(), alle pixels i output-billedet fra filtret.



Figur 3-5: De 4 kantorienteringer som anvendes i GRED-algoritmen

For hver pixel beregnes en faktor for hvor meget kant den repræsentere i henhold til hver af fire mulige kantorienteringer: EW, NESW, NS og NWSE (vist herover). Den maximale værdi af disse fire faktore, samt deres reciprokke værdier, findes og betegnes R. Hvis R er størrer end en tærskelværdi (som her er sat til 1.1), antages pågældende pixel at være en kant, og det indskrives i edgeImage arrayet.

Figur 3-6: Outputbillede af testopstillingen i almindeligt dagslys fra vinduet. Kantdetektering må siges at virke markant bedre end før.



Figur 3-7: Outputbillede af testopstillingen i kunstig belysning fra en arkitektlampe. Det rødlige skær skyldes lyset fra glødepæren i lampen.

Billederne herover viser outputtet efter kantdetekteringen med den nye algoritme, hvor testopstillingen er belyst med henholdsvis dagslys og kunstigt lys fra en glødepære. Kantdetekteringen er en del bedre, men dog stadig ikke perfekt. Som set tidligere giver lyset fra vinduet en blå sky på billedet til venstre.

Tilsvarende ses det at lyset fra glødepæren danner en rødlig sky nederst på billedet til højre. Disse problemer er igen noget der skal løses nede i billedfiltret, som tidligere omtalt.

På billedet til højre ses yderligere et specielt fænomen; store dele af den gule lineal er blevet hvid, pga. af den kraftige belysning. Det skyldes tilsyneladende at der er sket en form for mætning et sted i billedfiltret. Bortset fra disse fejlplacerede farveklumper som selvfølgelig giver nogle kanter der ikke burde være der, er der faktisk ingen fejl placerede kanter, udover lidt småfnug hist og her. Disse småfnug kan nemt frasorteres når kanterne skal laves om til kantlinier og polygoner højere oppe i systemet. Det mest problematiske ved Gamma Ratio Edge Detector algoritmen, er nok at den genererer nogle alt for kraftige kanter, som måske kan blive sværere at omdanne til kantlinier højere oppe i systemet.

3.3. SUSAN Edge Detector-algoritmen

En hurtig søgning på nettet natyder at der findes utroligt mange kantdetekterings algoritmer. Hver med deres fordele og ulember. SUSAN Edge Detector-algoritmen er en yderst simpel algoritmen, som skulle kunne optimeres til at danne nogle fine tynde kantelinier, på relativ kort tid. Derfor var den også ganske oplagt at afprøve i denne forbindelse.

SUSAN står for: Smallest Univalue Segment Assimilating Nucleus. Betegnelsen dækker over en række algoritmer til bla. billedstøj filtrering, kantdetektering og hjørnedetektering. Algoritmerne er alle udviklet af Stephen M. Smith fra Oxford University, primært til brug ved hjernescanning. De bygger alle på lavniveau billedbehandling, med det formål at de skal være mindst mulig processorkrævende.

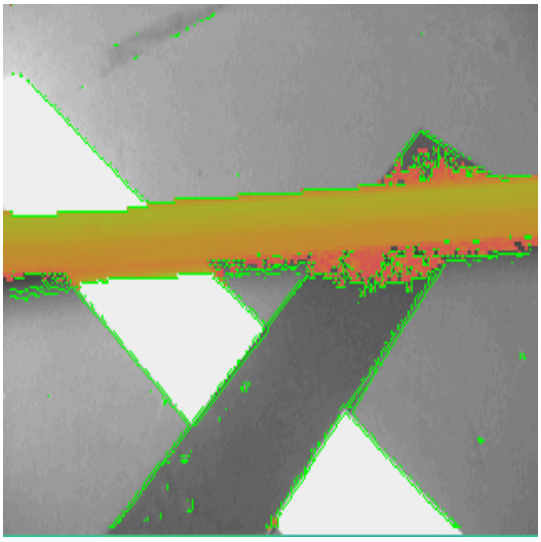
Som de fleste andre kantdetekteringsalgoritmer gennemløber SUSAN hver enkelt pixel i billedet, med et vindue, hvor den enkelte pixel altid er i centrum. Afhængig af nabopixlenes værdi i forhold til center pixlen, tages der der så stilling til om der findes en kant på pågældende position. I modsætning til de firkantede vinduer som de fleste andre algoritmer bruger, anvender SUSAN et cirkulært (tilnærmelsesvis cirkulært) vindue, for at følsomheden overfor kanter skal være den samme ligegyldig hvilken retning kanten vender. I min implementering anvender jeg et vindue med en radius på ca. 3,4 pixels, svarende til et areal på 37 pixels.

Vinduet eller masken som den også kaldes, placeres som sagt over hvert enkelt pixel i billedet, med pixlet i centrum. Antallet af pixels i masken der har samme værdi som pixlen i centrum, inde for en given tærskel værdi (t), kaldes pixlens n-værdi. Tærskelværdien t skal finjusteres således at den passer bedst muligt til applikationen. Hvis t værdien er for lille medtages for meget støj i billedet. Hvis den er for stor er det ikke alle kanter som detekteres.

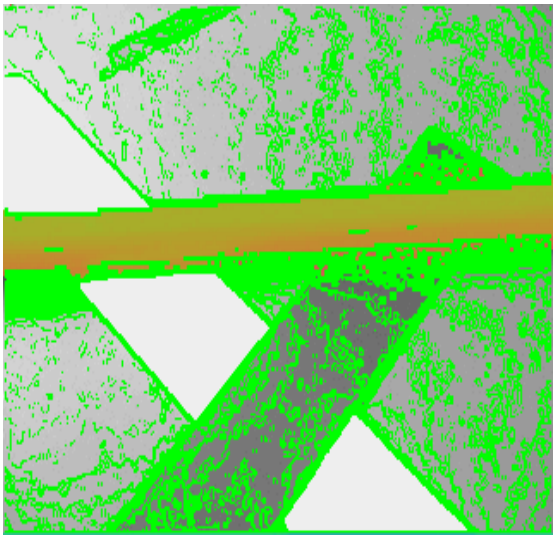
Når n-værdien er fundet, sammenlignes den med en geometrisk tærskelværdi g,

som er direkte afhængig af maskens areal. Typisk defineres g= ¾ *nmax. Hvor nmax er maskens areal excl. centerpixlen (i dette tilfælde 36 pixels). Det vil med andre ord sige at hvis antallet af pixels i masken, med tilnærmelsesvis samme værdi som center- pixlen, er under ¾ af maskens areal så er der en kant.

Jeg har implementeret denne SUSAN algoritme i form af funktionen findEdges_SUSAN() i CCarttonImage-klassen. Først indlæses pixelværdierne for alle pixels i masken, undtaget centerpixlen, i I[] arrayet. Derefter gennem løbes alle disse pixelværdier, imens der tælles hvor mange der har tilnærmelsesvis samme pixelværdi som centerpixlen. Hvis denne fundne n værdi så er mindre end g-værdien, markeres pixlen som en kant i edgeImage- arrayet.



Figur 3-8: SUSAN-algoritmen med for HØJ tærskelværdi (t). Der er ikke meget støj de steder hvor der ikke findes kanter, men til gengæld er der mange huller i de fundne kanter.

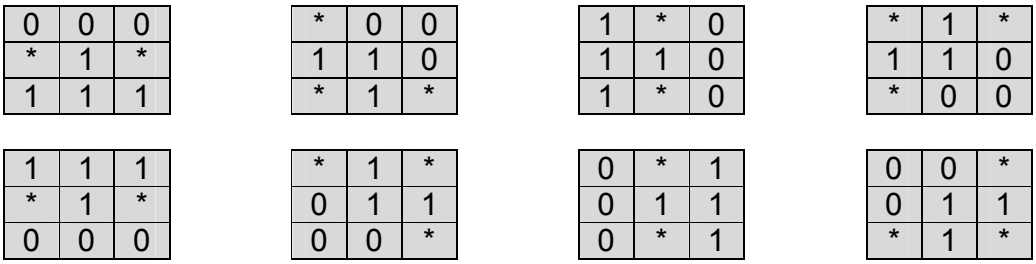


Figur 3-9: SUSAN-algoritmen med for LAV tærskelværdi (t). Alle kanterer massive og uden huller, men til gengæld findes der mange kanter på steder hvor de ikke burde være.

Som det ses på billederne herover er det meget svært at få finjusteret tærskelværdien t, således at algoritmen finder netop de kanter de skal findes. Billedet til venstre ser umiddelbart meget godt ud med fine tynde kanter, og de små ”støjpletter” hist og her burde nemt kunne fjernes. Men kigger man nærmere på billedet vil man opdage at kanterne indeholder en hel del huller, desuden findes der nogle steder flere parallele kanter hvor der kun burde være en. Dette er mindst et lige så stort problem, når der skal findes kantlinier i næste lag, som de tykke kanter GRED-algoritmen finder. Derfor må jeg nok konkludere at SUSAN ikke er bedre egnet end GRED, men trods alt en smule hurtigere.

3.4. Edge thinning

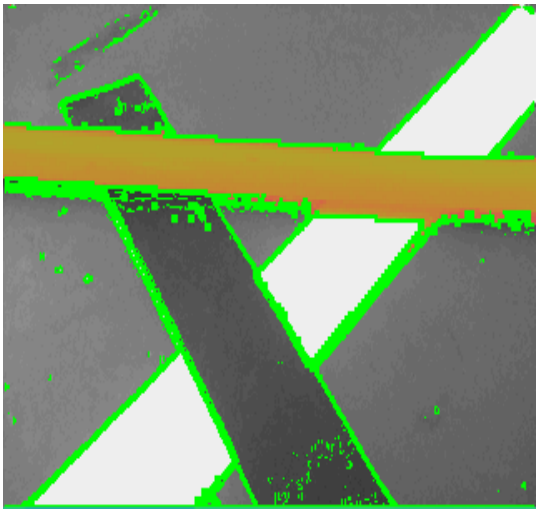
På trods af de massive kanter som GRED-algoritmen generere, så er den til gengæld mere imun over for støj i billedet. Den laver faktisk kun kanter der hvor der skal være kanter. De massive kanter, som er op til 10 pixels bredde, er meget besværlige at lave edge-lines efter. Der vil ganske enkelt være lige så stor sandsynlighed for at edge-lines genereres på tværs af kanterne, som der hvor de egentlig skulle være. For at sikre at edge-lines kan placeres rigtigt oppe I næste lag (EdgeLine-laget), er det nødvendigt med kanter der er 1 pixel brede. Ingen af de afprøvede kantdetekteringsalgoritmer, kunne danne en så fin kant. Derfor må der istedet udføres en udtynding af kanterne, efter de er fundet af GRED-algoritmen. Funktionaliteten indskydes som et ekstra lag I systemet, kaldet EdgeThinning-laget, imellem EdgeDetection-laget og EdgeLine-laget.



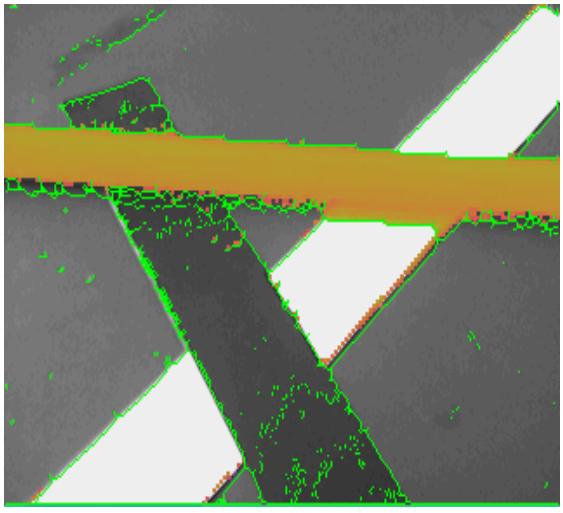
Figur 3-10: De 8 edge-thinning masker der anvendes i L8-udtynding. Kantpixlen fjernes hvis naboværdierne stemmer med en af disse 8 masker.

Udtyndingen (thinning) foregår ved at alle punkter I kantbilledet (edgeImage), gennemløbes med de otte masker som er vist I figuren herover, indtil kanterne ikke kan udtyndes mere. Maskerne i figuren herover kaldes L8. Hvis centerpixelen danner en tyk line sammen med en af sine 8 nabopunkter, bliver den ganske enkelt fjernet fra kantbilledet. “1” betyder at pågældende nabopunkt skal være markeret som et kantpunkt, ”0” at den ikke skal være et kantpunkt, imens “*” betyder “don’t care”. Hvis disse nabopunkt værdier så er op fyldt, fjernes kantpunktet i midten.

Udtyndingen er er implementeret i funktionen improveEdges() fra CCartoonImage-klassen. Den gennemløber alle punkter i edgeImage, indtil der ikke sker flere ændringer (thinned == false), og sender hver enkelt til thin_L8() i samme klasse. Thin_L8() tester om pågældende kantpunkt passer I en af de otte før omtalte masker, hvis det er tilfældes slettes kantpunktet fra edgeImage. Det har selvfølgelig sine omkostninger at indføre sådan et ekstra lag I systemet, rent processerings-mæssigt. EdgeImage skal oftest gennemløbes 5-7 gange før det ikke kan udtyndes mere. Svarende til worst case 320*240*7= 537.600 kald af thin_L8()-funktionen. Til gengæld får man så også et kun kanter på 1 pixels bredde, se høre billede herunder.



Figur 3-11: Output fra GRED-algoritmen FØR edge-thinning. De meget massive kanter er svære at lave korrekte edgelines efter. Derfor skal de udtyndes.



Figur 3-12: Output fra GRED-algoritmen EFTER edge-thinning med L8-maske. En bi-effekt ved denne udtynding er at der dannes små "spikes" vinkelret på de rigtige kanter.

Udover at udtyndings processen tager en del ekstra tid, er der faktisk kun en ulempe ved den. Som det ses på billedet herover til højre, dannes der af og til små ”spikes” vinkelret på den egentlige kant. Det skyldes at den massive kant på det punkt var lidt tykkere end ellers. Det tolkes af L8 maskerne som en forgrening, og fjernes derfor ikke. Problemet kan minimeres ved at bruge størrer masker under udtyndingen, men det er der umiddelbart ingen grund til at bruge ekstra tid på. De små spikes kan fjenes rimelig nemt i EdgeLine-laget.

4. EdgeLine-laget

For at de fundne kantpunkter, i edgeImage-arrayet, fra forrige lag skal kunne omdannes til polygoner, skal de først forenkles så meget som muligt. Det gøres i EdgeLine laget, og som navnet antyder, betyder det at kanterne ”samples” til en mængde kantlinie segmenter (edgelines). Således at en hvid streg i billedet f.eks. beskrives med en kantlinie på hver side, eller evt. flere linier på den ene side, hvis stregen buer. Efter dette lag beskrives kanterne derfor i form en mængde CEdgeLine-objekter. Denne ”forenkling” af kanterne viste sig en del mere omfattende end først antaget.

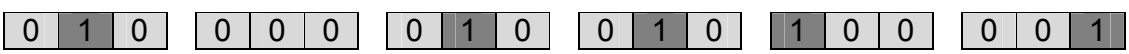
4.1. Indramning af kanterne

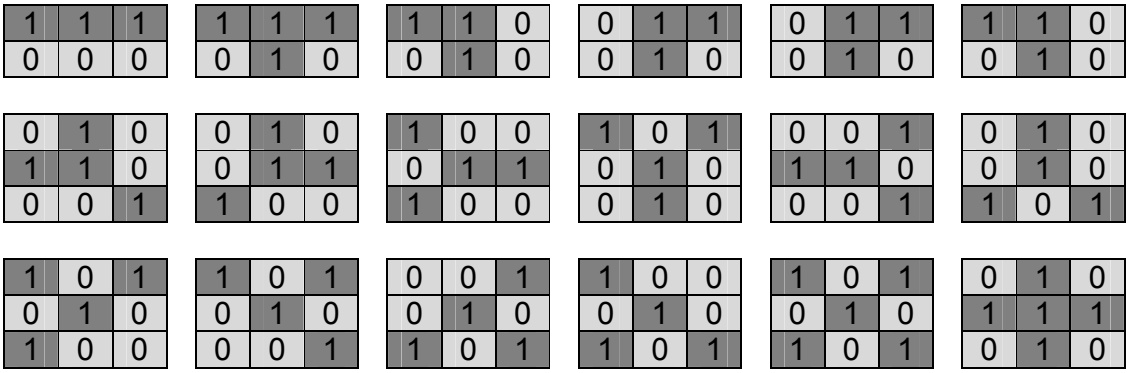
Når der senere skal genereres polygoner af de edge-lines som findes i dette lag, skal de helst værre lukkede inde for billedets udsyn. Da det kun er lukkede polygoner der kan bruges til de videre beregninger. Derfor skal der selvfølgelig også være edge-lines langs billedets ydre kanter. Som det måske blev bemærket tidligere så scanner kantdetekteringsalgoritmerne ikke efter kanter blandt de pixels som ligger nærmest billedets ydrekanter (1 pixel afstand for GRED, og 3 pixels afstand for SUSAN). Det skyldes ganske enkelt at deres masker/vinduer ikke skal komme uden for billedets ydrekanter. Der kommer altså under ingen omstændigheder kantpunkter til på billedets ydrekanter.

For alligevel at kunne frembringe edge-lines ved billedets ydrekanter, indsættes en kunstig ramme af kantpunkter i edgeImage. Således at den omkranser alle øvrige kantpunkter. Rammen er 2 pixels mindre end billedet i både højde og bredde. Derved skærer den netop de kantpunkter som evt. findes af GRED-algoritmen i dens ydrepunkter. Rammen tegnes af fire rette linier, således at de skærer hinanden i de fire hjørner. Derved bliver disse fire hjørner detekteret som kantknudepunkter (se næste afsnit). Indramningen af kanterne er implementeret i funktionen initEdgeImage() fra CCartoonImage-klassen.

4.2. Kantknudepunkter

For at de genererede edge-lines skal kunne bruges højere oppe i systemet til generering af polygoner, og senere 3D beregninger, er det ikke nok at linierne bare forenkler de fundne kanter. Det er også nødvendigt at en edge-line stoppes hvis den løber ind i en forgrening, således at den forbindes hertil, når linierne senere skal samles til polygoner. For at kunne vide om edge-linien er løbet ind i en forgrening, er det nødvendigt med en funktion til at teste om et givent punkt i edgeImage-arrayet er et kantknudepunkt eller ej. Denne funktionalitet er implementeret i isEdgeNode() funktionen fra CCartoonImage-klassen.





Figur 4-1: Maskerne der anvendes i isEdgeNode()-funktionen til detektering af kantknudepunkter i edgeImage

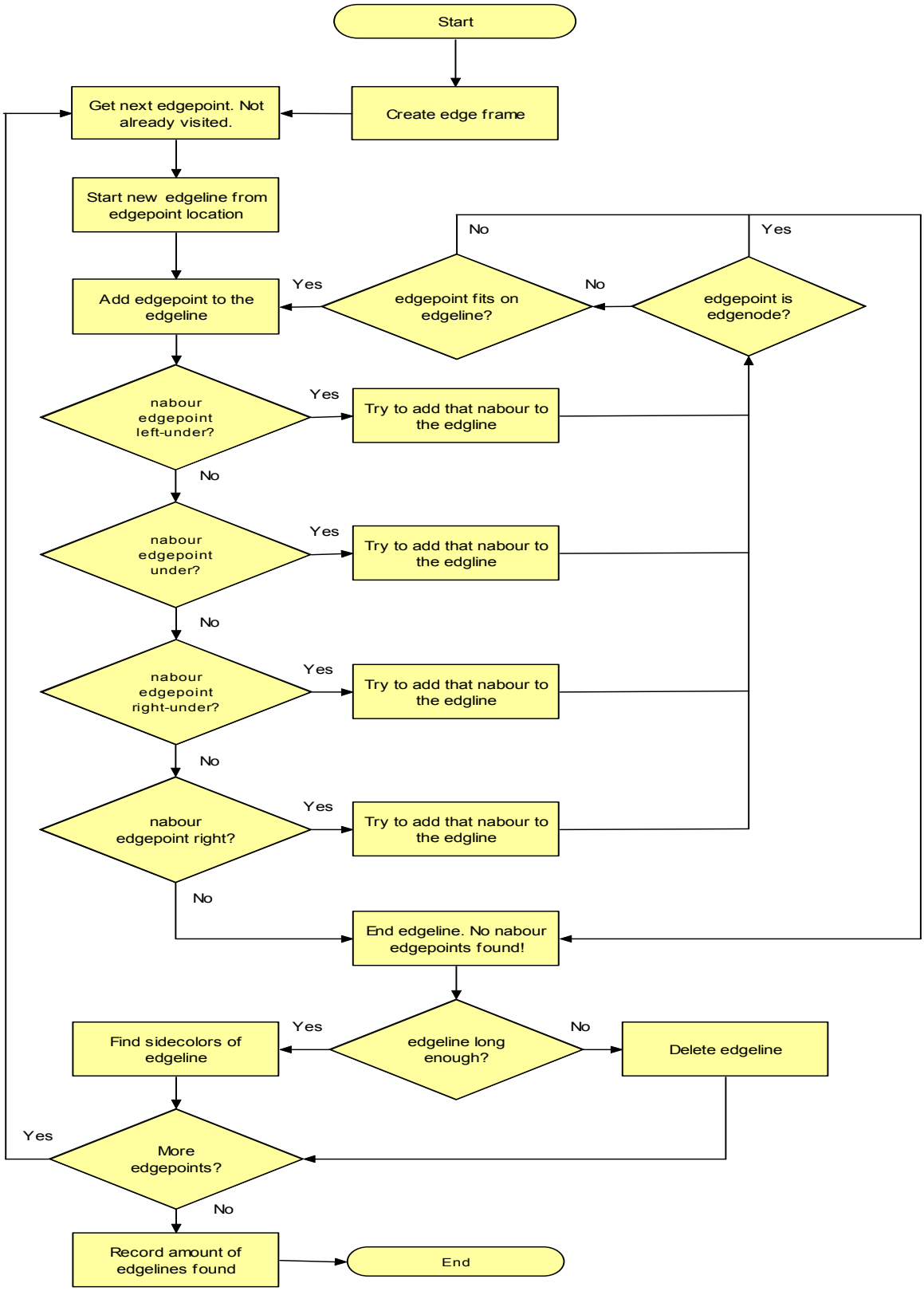
Funktionen anvender de 18 masker som er vist herover. Hvis en af dem passer, på det pågældende punkt i edgeImage, returneres true. Trods sin simple konstruktion virker isEdgeNode() faktisk ganske godt. At funktionenen så også finder knuder der hvor de før omtalte ”spikes” sidder på en rigtig kantlinie, er jo et problem som bør løses et andet sted i systemet.

4.3. Generering af kantlinier

Selve processen med at omdanne kantpunkter i edgeImage til kantlinier (edgelines), foregår i generateEdgeLines()-funktione. Den benytter sig så af en række hjælpe funktioner, som bla. de før omtale initEdgeImage() og isEdgeNode. generateEdgeLines() står med andre ord for den overordnede processering i EdgeLine-laget. Flowdiagrammet på næste skulle gerne gøre algoritmen bag den mere overskuelig. Den kaldes kaldes derfor direkte fra inputImage()-funktionen, efter findEdges_GRED().

Først indsættes en ramme af kantpunkter i edgeImage. Det gøres ved kald af initEdgeImage()-funktionen, som beskrevet i et tidligere afsnit. Når det er klaret gennemløbes edgeImage-arrayet punkt for punkt. Når der findes et kantpunkt, (EdgeImage = true på pågældende possition), og kantpunktet ikke allerede er besøgt, startes en kantlinie fra pågældende punkt. Til at holde styr på om et kantpunkt har været besøgt før, dvs. indkluderet i en anden kantlinie, anvendes visited-arrayet med samme størelse som edgeImage. Det bliver initialiseret til false på alle positioner (foregår i under kantdetekteringen i findEdges_GRED()-funktionen).

Det første kantpunkt i en kantlinie kan altid tilføjes kvit og frit, eftersom der endnu ikke er nogen kantlinie det skal ligge på. Men sådan er det selvfølgelig ikke med de resterende punkter. Når et kantpunkt er blevet tilføjet til kantlinien, scannes der efter nye kantpunkt kandidater blandt kantpunktets nabo-punkter. Eftersom punkterne gennemløbes oppefra, fra venstre til højre, ledes der kun blandt de naboer, som ikke allerede er blevet gennemløbet.



Figur 4-2: Flowdiagram for generateEdgeLines()-funktionen.

Dvs. den nederst til venstre, den lige under, den nederst til højre, samt den lige til højre for kantpunktet. Hvis der findes et kantpunkt blandt disse naboer, antages at det skal tilføjes til den igangværende kantlinie. Men før den kan blive tilføjet skal den lige bestå et par tests.

Som det også blev beskrevet i et et tidligere afsnit så ønskes kantlinien stoppet hvis den stødder ind i en kantforgrening, således at der kan tages højde for denne. Derfor testes først som det nye kantpunkt er et kantknudepunkt, med kald af den før omtalte isEdgeNode()-funktion. Kun hvis kantpunktet IKKE er et knudepunkt går det videre til næste test. Ellers stoppes kantlinien (omtales senere).

Næste test er den vigtigste af alle: passer kantpunktet på den igangværende kantlinie. Til det formål anvendes funktionen addPoint() fra CEdgeLine-klassen. Udover at teste om kantpunktet kan tilføjes til pågældende kantlinie (CEdgeLine-objekt), sørger den også for at tilføje punktet til kantlinien hvis det er muligt. Funktionen returnere true hvis det er tilfældet. En nærmere beskrivelse af denne funktion findes i næste afsnit.

Hvis kantpunktet blev tilføjet til kantlinien, markeres det som tilføjet (visited = true), og der fortsættes med at finde nabo-punkter til det osv. som tidligere beskrevet. Hvis kantpunktet derimod dumpede en af de to tests, afsluttes kantlinien. Det samme sker hvis der ikke kunde findes nogle nabo-punkter til et kantpunkt. Når en kantlinie afsluttes testes for om kantlinien er lang nok, til at den anvendes. Det gøres for at frasortere de kantlinier som er genereret udfra ”støj”-kanter i edgeImage-arrayet. Til at beslutte om hvorvidt en kantlinie er ”lang nok” anvendes en tærkselværdi: minELLength. Den har jeg justeret til 5 pixels, efter nogle eksperimenter.

Kantliniens længde beregnes tilnærmelsesvis af getAppLenght()-funktionen, fra pågældende CEdgeLine-objekt. Funktionen returnere ikke den nøjagtige euklidiske afstand, men der imod længste afstand af henholdsvis x og y retningen. Det giver en udemærket vurdering af hvor lang kantlien er, og så er det meget hurtigere beregningsmæssigt, i forhold til den Euklidiske afstandsformel.

Hvis kantlinen vurderes til at være lang nok, gemmes den og dens to sidefarver findes. Sidefarverne er de to forskellige farver, som billedet har på hver sin side af kantlinien. Sidefarverne bruges i næste lag til at samle kantlinierne i polygoner. Se nærmere beskrivelse af denne process i næste kapitel. Sidefarvene findes ved at kalde funktionen setELSColors() fra CCartoonImage-klassen (se nærmere beskrivelse senere).

Hvis kantlinien viste sig at være for kort, skal den slettes og alle de kantpunkter som indgik i den skal markeres fri på markedet (visited = false). Det gøres ved at kalde funktionen unvisitEdgeLine(). Proceduren gentages indtil alle kantpunkter i edgeImage har været igennem. Når alle kantlinier til slut er fundet, gemmes antallet af kantlinier i klassevariablen totalEdgeLines, til senere brug i Polygon-

laget. Dette var en overordnet beskrivelse af hvordan kantlinie genereringen foregår. En mere detaljeret forklaring af addPoint() og sideELSColors(), findes i de to følgende afsnit.

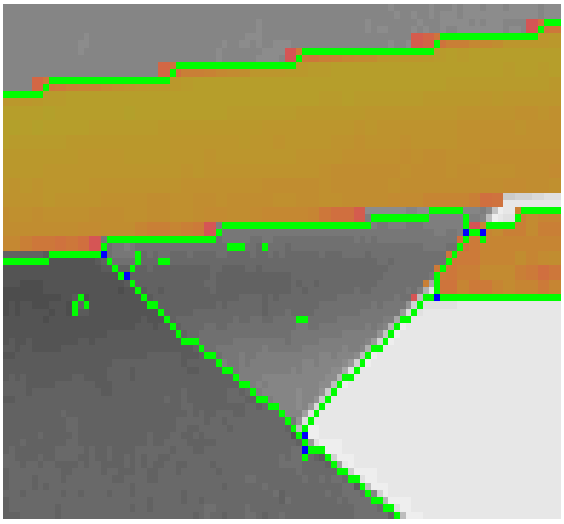
4.4. Tilføjning af et kantpunkt til en kantlinie

Selve vurderingen af om et kantpunkt passer på en kantlinie og den evt. tilføjningen af punktet til kantlinien, foregår som sagt i addPoint()-funktionen fra CEdgeLine-klassen. Funktionen stiller kort fotalt det simple krav at kantpunktet skal flugte med kantlinien, dvs. hvis kantlinien gøres længere i begge retninger, skal kantpunktet ligge herpå. Alt information omkring den pågældende kantlinie ligger i det CEdgeLine-objekt som repræsentere kantlinien. Disse data omfatter bla. kantliniens start- (xStart, yStart) og slutpunkter (xEnd, yEnd), samt dens sidefarver (s1Color og s2Color).

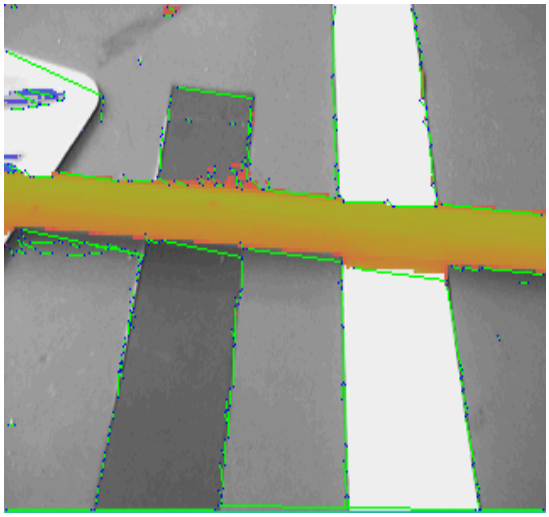
Som tidligere beskrevet kaldes addPoint()-funktionen hver gang et kantpunkt ønskes tilføjet til pågældende kantlinie. Kantpunktets position medsendes som argumenterne x og y. Hvis det er det første punkt i kantlinien, sættes både start og slutpunkt til denne position, og der returnes true. Hvis det ikke er det første punkt på kantlinien testes om det nye kantpunkt er nabo til enten kantliniens start eller slutpunkt. Denne test udføres af funktionen isNabours(). Den returnere ganske enkelt true hvis punkterne ligger i en afstand af <=1 pixel af hinanden. Hvis detviser sig at kantpunktet er nabo til enten start eller slutpunktet, kan den egentlige test begynde.

Først beregnes kantliniens udspænding i henholdsvis x og y retningen (dX og dY), hvis det antages at punktet tilføjes. Da der er tale om en ”digital” linie betående af pixels, og ikke en ”analog” som den der tegnes med en blyant på et stykke papir, kan vi ikke bare bruge liniens ligning direkte. Ikke nok med at lodrette linier ikke kan tegnes pga. hældningskoeficienten a bliver uendelig, men der kommer også huller i linien hvis den har en hældning over 45 grader. For at kompensere for det, testes derfor først om hældningen er over eller under 45 grader. Hvis hældningen er under 45 grader, anvendes den traditionelle linie-ligning, hvor y-værdien er en funktion af x-værdien. Hvis den er under 45 anvendes den omvendte linie ligning hvor x-værdien er en funktion af y-værdien.

Når det er besluttet hvilken af de to muligheder der skal vælges, beregnes de to konstanter til liniens ligning (testA og testB,) hvis det antages at det nye kantpunkt tilføjes til linien. Herefter testes om liniens ligning stadig er opfyldt for kantliniens: nuværende endepunkt (xEnd, yEnd), forrige endepunkt (xPreEnd, yPreEnd) og et punkt som ligger halvvejs på linien (halfX, halfY). Kun hvis alle disse punkter stadig ligger på linien indenfor en fastsat nøjagtighed, efter det nye kantpunkt er forsøgt tilføjet, bliver kantpunktet godkendt og tilføjet til kantlinien. Kantliniens endepunkt bliver dens forrige endepunkt og det nye kantpunkt bliver liniens nye endpunkt. Nøjagtigheden er angivet i form af t, som jeg efter nogle forsøg har fastsat til 1.0 pixel.



Figur 4-3: isEdgeNode()-funktionens evne til at finde knudepunkter i kanterne. De blå punkter er knudepunkter.



Figur 4-4: Fundne kantlinier. Som det ses dækker kantlinierne ikke alle de steder hvor der burde være kanter. Disse manglede steder skyldes at der på disse steder kun kunne dannes for korte kantlinier, som derfor er blevet fjernet igen.

Algoritmen til at finde kantlinier er absolut ikke perfekt. Blandt dens ulemper kan nævnes at langtfra alle kantpunkter ender med at være en del af en kantlinie. Det medfører at der tit opstår større afstande imellem kantlinierne, og det er et problem når der skal dannes polygoner af dem. Problemet skyldes at kantpunkterne på visse steder ikke ligger på en linie der er lang nok, kantlinien bliver derfor slettet igen. En anden ulempe ved algoritmen er at kantlinierne kan blive placeret meget upræcist, i forhold til hvor der egentlig var en kant. Det skyldes dels at det ikke er alle punkterne i linien der testes når et nyt kantpunkt skal tilføjes, men kun to endepunkter og et halvvejs. En anden grund er værdien af nøjagtighedsvariablen t, jo højere den er jo mere upræcist ligger kantlinien i forhold til den oprindelige kant. Men jo mindre t bliver desto kortere bliver kantlinierne, hvilket så medfører at flere kasseres. Så det er igen er hård balancegang.

4.5. Kantliniens sidefarver

For at gøre det muligt for polygon-laget at sammenkæde kantlinierne til polygoner, tildeles hver kantlinie to sidefarver. Som hver især er farven af området på pågældende side af kantlinien. Polygon-laget bruger så disse sidefarver til at bestemme hvilke kantlinier der passer i et polygon, dvs. omkranser et område med samme farve, men mere om det i næste kapitel. Sidefarverne s1Color og s2Color i et CEdgeLine-objekt er CCartoonPixel værdier. Sidefarverne findes ved at kalde funktionen setELSColors() fra CCartoonImage-klassen, hvilket det som sagt klares af generateEdgeLines(). Funktionen får den kantlinie, som der ønskes fundet sidefarver til, medsendt som argument. Farven på samme side af kantlinien kan godt variere en lille smule

langs linien, men der ønskes kun fundet en sidefarve til hver side. Dette problem kunne løses ved at midle alle farverne på pågældende side hele vejen hen ad kantlinien. Det er dog både meget resource krævende, og så kan man risikere at man ender op med en farve som faktisk ikke findes på pågældende side, men blot ligger imellem (på farveskalaen) to som findes der.

Jeg har istedet valgt en meget mere simpel metode, som faktisk giver et udemærket resultat i de fleste tilfælde. Først findes punktet halvvejs på linien (halfX og halfY). Fra dette punkt går man så et antal pixels (colSamDis) ud til begge sider af kantlinien og læser pixelværdierne. Disse to pixelværdier gemmes så som kantlinien to sidefarver s1Color og s2Color. ColSamDis er en justerbar variable. Det viser sig nemlig at de pixels som ligger umiddelbart ved siden af kantlinien, næsten altid er utilregnelige. Det skyldes dels at kantlinien næsten altid ligger lidt forskudt i forhold til der hvor der faktisk er en kant, dels at pixelværdierne ved en kant ikke skifter momentant, men langsomt fader over i en anden farve. ColSamDis skal derfor ikke vælges alt for lille, men på den anden side heller ikke for stor. For så kan den jo riskiere at lande bag en helt anden kantlinie med en helt tredie farve. Så her kan igen tales om en hårdfin balancegang. Jeg kom frem til at en colSamDis på 4 pixels passede udemærket til dette formål, men det afhænger bl.a. hvor store objekter der skal skelnes imellem.

4.6. Foreslag til yderligere forbedringer

Algoritmen virker som tidligere omtalt ikke helt optimalt. Den ville uden tvivl undgå de mange huller blandt kantlinierne, hvis den ikke skulle frasortere de kantlinier, som er genereret udfra ”støj”-kanter fra EdgeDetektion-laget. Her kommer ordsproget ”garbage-in-garbage-out” virkelig til sin ret, for det ville uden tvivl hjælpe hvis de underliggende lag blev forbedret. En anden måde algoritmen måske kunne forbedres på, kunne være ved ikke absolut at tegne kantlinier henover hver enkelt punkt. I stedet skulle der tegnes kantlinier imellem de kantknudepunkter og skarpe-kanthjørner, som som var direkte forbundet med kantpunkter. Kantlinierne ville så ikke altid komme til at ligge lige præcist oven på de faktiske kanter, men det ville så blot kræve at der tages højde for det når sidefarverne skal findes.

En tredie måde til at løse problemet, var måske ved helt at undvære EdgeLine-laget. Det jo faktisk ikke nødvendigt for at kunne danne polygoner. Kantpunkterne kunne istedet bruges direkte som sider i polygonerne, så ville der ikke være nogen problemer med at de ikke lå der hvor de blev fundet, eller nogle huller imellem dem. Men det ville selvfølgelig gøre at polygonerne fik en hel del flere data at tygge på, og det var jo netop det EdgeLine-laget skulle rode bod på.

5. Polygon-laget

Kantlinierne skal nu sammensættes til egentlige objekter i billedet. Sådanne objekter kunne f.eks. være en RoboCup-port eller en hvid streg. Ved at have billedet defineret som en mængde objekter, fremfor bare pixels, skulle det gerne blive nemmere at udføre de fotogrametriske beregninger. Eftersom det så blot er to polygoner som sammenlignes, fremfor nogle pixelarealer. Objekterne i billedet kan betegnes som polygoner, eftersom de består af et antal kantlinier. For at en samling kantlinier kan kaldes et polygon kræves det at de har nogle ting til fælles, heriblandt en fælles sidefarve og at de tilsammen danner en lukket polygon. Det er netop denne process med at danne polygoner udfra kantlinier, der foregår i polygon-laget. Et polygon er repræsenteret som et objekt af C2DPolygon-klassen (se implementering i appendix). Den hedder 2D fordi polygonet endnu kun har x og y koordinater i billedet. Det er meningen at der også skal komme et C3DPolygon, efter de fotogrametriske beregninger, som istedet indeholder x, y og z koordinater for polygonets placering i forhold til robotten.

5.1. Udstrækning af kantlinierne

Som tidligere beskrevet er der oftest mindre huller imellem de kantlinier som egentlig passer sammen. Det gør det lidt sværere når kantlinierne skal samles til lukkede polygoner, dvs. en mængde kantlinier hvor endepunktet af sidste kantlinien er det samme som startpunktet af første linie. For at forsøge chanserne for at generere sådanne perfekte polygoner, må disse huller lukkes.

Processen udføres i funktionen improveEdgeLines(), umiddelbart efter at kantlinierne er lavet. Hullet imellem to kantlinier som burde være forbundet, således at slutpunktet for den første er lig start punktet på den næste, kan fjernes ved at strække/forlænge den første kantlinie en smule således at den netop når til den anden kantlinies startpunkt. Det gøres på følgende måde:

Alle de fundne kantlinier gennemløbes en efter en. Algoritmen scanner så systematisk efter andre kantliniers startpunkter, som ligger tæt på kantliniens endepunkt. Først med en lille radius, så størrer og størrer indtil den enten finder en kantlinie eller er nået max. søgeradius (maxReachDist). Radius starter med at være 2 pixels, denne afstand gør at alle kantlinier som i forvejen er forbundet (uden huller) findes, og derfor ikke behøver at blive strukket. Hvis der ikke findes en kantlinie i denne søgeradius, forøges radiusen med 4, osv. Indtil maxReachDist er nået. Jeg har efter nogle forsøg fundet frem til at en maxReachdist på12 passede godt til mit formål. Men det er igen et spørgsmål om en balancegang, hvis maxReachDist er for lille er det ikke alle huller der lukkes, er den for stor ødelægges kantlinierne troværdighed. De kommer simpelthen ikke til at ligge der hvor kanterne i virkeligheden er.

Hvis der findes en anden kantlinie indenfor rækkevidde fra kantliniens

endepunkt, udføres et stræk af kantlinien. Dvs. endepunktet af kantlinien sættes til startpunktet af den fundne nabo-kantlinie. Der tages her ikke højde for at de har samme sidefarver, og derfor hører til samme polygon. Det formodes bare eftersom de ligger ”tilpas” tæt på hinanden. Udstrækning af kantlinierne er selvfølgelig en lappeløsning på problemet fra forrige lag, med at der dannes huller imellem kantlinierne, og der er ingen tvivl om at det ville være bedst hvis denne udstrækning helt kunne undgås. Problemet med udstrækning af kantlinierne er også at de kommer til at ligge en smule forkert i forhold til hvor kanten i virkeligheden er. Dette problem kunne evt. løses ved at i stedet at indskyde en ekstra kantlinie i hullerne imellem kantlinierne. Derved ville de blive der hvor de oprindelig blev fundet.

5.2. Generering af polygoner

Efter kantlinierne er forbedret en smule, kan selve polygon dannelsen eller samlingen begynde. Det klares af funktionen generatePolygons() fra CCartoonImage-klassen. Der tildeles et antal polygoner (maxPolygons) som funktionen må generere udaf kantlinierne. Hver kantlinie kan indgå i polygoner, eftersom den har to sider, og derfor adskiller to objekter i billedet. Et polygon forsøges færdiggjort (lukket) ad gangen. Hvis det så enten bliver lukket, eller må afbrydes fordi det ikke kan lukkes, startes et nyt polygon.

Algoritmen er ikke specielt hurtig, men tilgengæld finder den med 100 % sikkerhed alle polygoner, hvis der vel og mærket er kantlinier til det. Hver gang et nyt polygon startes prøves alle kantlinier af for at se om en af dem skulle passe i polygonet, og det gøres udfra samme princip som i improveEdgeLines(). Først med en lille søgeradius, derefter størrer og størrer.

Polygonet dannes som perler på en snor, ved at tilføje kantlinierne en efter en indtil polygonet gerne skulle ”bidde sig selv i halen”. Selve tilføjningen af en kantlinie til et polygon, foretages af funktionen addEdge(), fra pågældende polygons C2DPolygon-objekt. Den beskrives nærmere i næste afsnit. Når en kantlinie er blevet tilføjet til et polygon markeres det med at s1Incl eller s2Incl, fra pågældende CedgeLine-objekt, sættes true. Hvilken en af de de to variabler der sættes true afhænger af hvilken side der vender ind imod polygonet. En kantlinie kan derfor indkluderes i to polygoner.

5.3. Tilføjning af en kantlinie til et polygon

Et C2DPolygon-objekt repræsentere som sagt et polygon, og alle de data som det indeholder. Disse date er bl.a. en mængde kantlinier (CEdgeLine-objekter) som polygonet består, et array (innerside) som fortæller hvilken side af hver enkel kantlinie der hører til pågældende polygon, og et flag som fortæller om polygonet er lukket eller ej (isClosed).Tilføjning af en kantlinie til polygonet, foregår med addEdge()-funktionen, der kun returnere true hvis kantlinien blev tilføjet til polygonet. Funktionen fungerer på følgende måde:

Kun hvis polygonet IKKE allerede er færdigt/lukket, kan der blive tale om at tilføje en kantlinie til det. Hvis den nye kantlinie er den første i polygonet, bestemmer denne kantlinies sidefarver, hvilken farve det objekt er som polygonet skal omkrandse. Det kræver selvfølgelig at min. en af kantliniens sider ikke allerede er inkluderet i et andet polygon. Den side af kantlinien som viser sig at være fri, bliver indersiden af polygonet. Hvis begge sider er fri, vælges side 1 pr. definition. Når et polygon på denne måde er startet, initialiseres det, hvilket blandt andet vil sige at den får tildelt en farve, med et kald af updateColor()-funktionen, og addEdge() returnere true. Mere om updateColor() i næste afsnit.

Inden kantlinien forsøges tilføjet, testes lige om polygonets mængde af kantlinier kan ”bide sig selv i halen”, og derved lukke polygonet, med den søgeradius som anvendes (reachDist). reachDist er den før omtalte søgeradius fra generatePolygons(), der medsendes som argument. For en sikkerhedsskyld tjekkes også om polygonet min. indeholder tre kantlinier. Så der ikke sker fejl pga. en kantlinie som er kortere end reachDist. Hvis polygonet opfylder disse krav kan det lukkes, og funktionen checkVisibility() kaldes. checkVisibility() tester om hele det fundne objekt kan ses på billedet, eller det kun er en del af det der er synligt. Det er bl.a. en vigtig oplysning når der skal findes RoboCup-porte, men mere om det i næste kapitel.

Nu kan selve testen, om kantlinien kan tilføjes eller ej, begynde. Det er ganske enkelt kun tilfældet hvis den nye kantliniens startpunkt ligger inde for rækkvidde (reachDist) fra sidste kantlinies endepunkt. Ved den sidste kantlinie forstås den kantlinie som sidst blev tilføjet til polygonet. Hvis kantlinien opfylder dette krav, testes om en af kantliniens sidefarver passer tilpas godt med polygonets, samt selvfølgelig om den side der evt. passer ikke allerede er indkluderet i et andet polygon. Til at afgøre om en sidefarve passer tilpas godt til polygonets, anvendes tærskelværdien t. Det er igen en værdi som kan justeres i det uendelige. Efter som jeg mest er interesseret i at kigge på gule-objekter (RoboCup-porte), har jeg sat den til 16, hvilket gør at alle gule værdier dækkes. Bieffekten ved en så høj t-værdi er at nogle røde og grønne objekter også kan risikere at blive godtaget, men dem er der heldigvis ikke mange af på en RoboCup-bane!

Hvis også en af sidefarverne på kantlinien passer til polygonet, tilføjes kantlinien til polygonet. Samtidig registreres hvilken side af kantlinien som vender ind imod polygonet. Herefter opdateres polygonets farve, ved kald af updateColor(), og der returneres true. Hvis kantlinien ikke klarede alle disse tests returneres false som indikation af at den ikke blev tilføjet.

5.4. Farvning af et polygon

Polygonets farve opdateres dynamisk hver gang en kantlinie tilføjes. Polygonets

farve skal gerne repræsentere en fælles farve for alle de kantlinier som indgår i den. Dvs. den side af kantlinierne som vender indad i polygonet. Man kunne her selvfølgelig vælge at tage middelværdien af alle kantliniernes inderste sidefarve. Det kunne dog have den bivirkning at polygonet fik en farve som slet ikke var blandt kantliniernes inderfarver. F.eks. hvis en enkelt kantlinies inderfarve afviger meget fra de restende (pga. en fejl), så vil den trække middelværdien helt ud i skoven. En langt bedre metode er derimod at lade polygonets farver være den der hyppigst forekommer blandt kantliniernes inderfarver.

Grunden til at polygonets farve skal opdateres kontinuerligt, fremfor bare at lade den første kantlinie bestemme farven, er at det jo ikke nødvendigvis er den første kantlinie som har den mest korrekte farve for polygonet. Ved at opdatere farven kontinuerligt, får alle kantlinier lige meget ret til at ændre farven på polygonet. Som tidligere omtalt foretages farveopdateringen af funktionen updateColor() i C2DPolygon-klassen. Den virker på følgende måde:

Afhængig af hvilken side af den pågældende kantlinie der vender indad, tilføjes dennes sidefarve til et histogram array (colorHistogram) i C2DPolygon-objektet. Histogrammet angiver hyppigheden er alle farver blandt kantliniernes inderfarver. Det er indexeret efter farveværdien (CCarttoonPixel-værdien), som kan være fra 0-255. Hver gang histogrammet er blevet opdateret, tjekkes samtidig om den tilføjede farves hyppighed overstiger max. hyppigheden. Dvs. om hyppigheden for farven er størrer end hyppigheden for den polygonet har pt. Hvis det er tilfældet, er der fundet end ny max. hyppighed og en ny farve til polygonet. Polygonets farve gemmes i member-variablen: color.

5.5. Beregning af hjørnepunkter

For at kunne beregne 3D koordinater for et polygon, skal det samme polygon findes i både højre og venstre billede. Det er oftest et problem med stereovision, at få pared de to billeder, således at der kan udføres fotogrametriske beregninger. Det er netop en af grundene til at jeg ønskede at opdele billederne i polygoner fremfor pixels. Det er en del nemmere at sammenligne en lille mængde store objekter, fremfor en stor mængde små objekter. For at kunne pare to polygoner, skal man kunne sammenligne dem, og vurdere hvor meget de ligner hinanden. Kategorierne som polygonerne skal sammenligneds efter er bl.a. farve, form, størrelse og en ca. placering i billedet. For at kunne frem til disse parametre, skal det være muligt at beregne ydrepunkter og hjørner for polygonet. Ydrehjørnerne, dvs. de punkter som er tættes på billedets hjørner, beregnes af funktionen findExtremeCorners() fra C2DPolygon-objektet. Den virker på følgende måde:

Funktionen kan kun køres hvis polygonet er lukket. Det antages først at alle ydrehjørnerne ligger i centrum af billedet, således at der er noget at sammenligne med når de skal findes, gennem iteration. Herefter gennemløbes alle kantlinier i polygonet en efter en. Hver kantlinie testes for om den er et nyt ydrehjørne, i alle fire hjørner: TopLeftMost, TopRightMost, BottomLeftMost og

BottomRightMost.

Selve testen foregår ved at ”tegne” en linie fra kantliniens endepunkt, med en hældning på 45 grader, ud imod det hjørne der ønskes testet for. Derved findes et skæringspunkt b, for henholdsvis aksen x=0 (venstre hjørner) og aksen x=320 (højre hjørner). Denne b-værdi sammenlignes så med de tidligere fundne b-værdier. Hvis den så er nærmere imod 0 (for øvre hjørner), eller nærmere imod 240 (for nedre hjørner), er der fundet et nyt ydrehjørne. Værdierne 320 og 240, kommer af at billedet har dimensionerne 320*240. Når ydrehjørnerne er fundet kan det nederste midtpunkt også findes. Dette punkt er bl.a. interessant, når midten af en RoboCup-port skal findes (se næste kapitel).

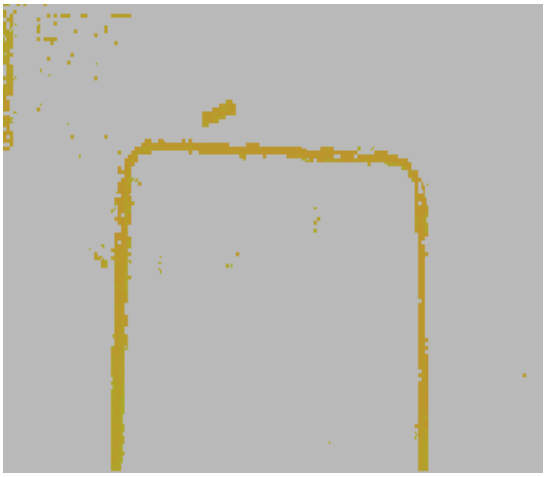
6. Styring efter RoboCup-port

På grund af at udviklingen af systemet tog meget mere tid end først antaget, og der pt. nu kun var en mdr. til RoboCup-finalen, besluttede jeg at begrænse projektet. Således skulle det nu ikke længere omfate stereovision (dvs. to webcam), men ”blot” dreje sig om at finde en Robocup-port i et billede, og styre efter midten af den.

6.1. Detektering af gule objekter

På grund at den reviderede reviderede problemformulering, kan det nu godt tillades at systemet ikke længere virke ligeså generelt overfor alt hvad der findes på billederne. Der kan med andre koncentreres om RoboCup-porte. Som det ses på billedet herunder skiller Robocup-portene sig en del udfra omgivelserne, eftersom de er malet i en meget gul farve. Denne tydelighed er endnu kraftigere på RoboCup-banen, hvor de fleste andre farver som forekommer er hvide, sorte eller grålige.

Figur 6-1: Råbilledet af RoboCup- porten.



Figur 6-2: Efter frafiltrering af alle andre farver end gul, i ImageFilter-laget. Porten går ganske fint igennem, men der kommer samtidig en begrænset mængde støj med.

Erfaringer har vist at det er bedst at fjerne uønskede data på så lavt et niveau som muligt, ellers bliver de oftest bare forstærket på vejen op igennem lagene. Den første filtrering af porte fra andre objekter, skal derfor nu foregå allerede i ImageFilter-laget. Det sættes ganske enkelt kun til at gemme pixels som er gule, de resterende pixels farves grå. Se billedet herover til højre.

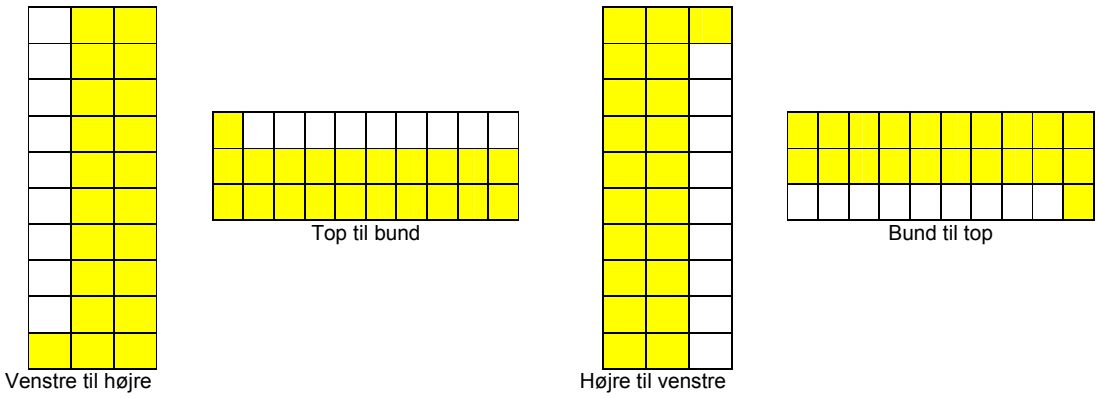
Funktionaliteten er implementeret i funktionen inputImage() fra CCartoonImage-klassen. Efterhånden som hver pixel gennemløbes, testes først om outputværdien (CCartoonPixel-værdien) for pågældende pixel er gul, før den gemmes i image-arrayet. Den funktion der bruges til at teste om en pixel er gul,

er isYellow() fra CCartoonPixel-klassen. I IHS-formatet har gul en hue-værdi på 180 grader (pi i radianer). Så gul har derfor en CCartoonPixel-værdi på på 64+192/2 = 160. isYellow() returnere derfor true hvis værdien er 160 inden for en margin af 10 dvs. fra 150-170, for at være på den sikre side. Det er igen et spørgsmål om finjustering. Hvis margin er for lille er det kun brudstykker af porten som kommer med, er margin for stor, kommer der for meget støj med.

6.2. Generering af port-polygon

Med et billede som antags udelukkende at indeholde den evt. RoboCup-port, som måtte være indenfor synsvidde. Kan den videre processering også forenkles en helt. Det er faktisk kun interessant at finde portens placering i billedet, således at robotten kan styres efter portens midterpunkt. Eftersom der kun antages at befinde sig et objekt i billedet (porten), er der ingen grund til at finde kantlinier, og dele billedet op i polygon-objekter. Det er kun nødvendigt at finde et polygon som repræsentere porten. Denne forenklede process foretages af funktionen findPortPolygon() fra CCartoonImage-klassen.

Funktionen arbejder slet ikke efter samme metode som generatePolygons(). Istedet finder den port-polygonet ved at skubbe kantlinier ind fra alle fire sider, indtil de stødder ind i et tilpas stort gult område. Kantlinierne er forbundet med hinanden som en elastik, der spændes rundt om det gul objekt. Når polygonet først er fundet på denne måde, kan de resterende bergninger af ydrehjørnerne fortages som normalt. Funktionen fungerer på følgende måde:



Figur 6-3: Scannemasker til detektering af en port i billedet.

Først scannes fra venstre, med start for neden. Ved hver linie scannes imod højre side af billedet. Maskerne der bruges til scanningen er vist herover. Hvis masken stødder ind i et område, hvor de viste pixels alle er gule, indsættes en kantlinie. Kantlinien tegnes fra forrige kantlinie til den nye lokation direkte. Denne alternative måde at tilføje kantlinier til et polygon, udføres af funktionen lineTo(), som ganske enkelt tegner en kant i polygonet.

Efter hele venstreside af porten er scannet, fortsættes med scanninger fra oven

og nedefter, fra højre imodv venstre og fra bund til top. Denne metode giver selvfølgelig en del flere kantlinier i polygonet, end med den normale metode, eftersom hver kantlinie tit kun er 1 pixel lang.

Grunden til at der anvendes så store masker til at teste om scanningen har fundet porten, er for at undgå at få alt for mange små ”støj”-elementer med i portpolygonet. Der vil dog alligevel komme lidt støj med, især på mange af de forsøg som blev fortaget med porten stående på laboratioriets halvgule trægulv. For at sikre at portpolygonet virkelig er en port, bruges funktionen isPort() fra C2DPolygon-klassen. Den tester ganske enkelt om der er tilpas mange kantlinier i polygonet. Eksperimenter har vist at en port faktisk altid får tildelt min. 400 kantlinier. Imens billeder uden en port, oftest kun giver mindre end 100 kantlinier.

6.3. Fokusering på portens åbning

Med den fundne portpolygon kan der nu foretages de tidligere omtalte beregninger af ydrehjørnerne, samt portens midterpunkt. Fidusen er nu at dette midterpunkt skal omsættes til et styre signal for robotten. Her kan drages nytte af en af de Logitech Sphere webcams, med pan/tilt funktionalitet, som jeg har fået tildelt. Planen er at kameraet skal styres således at det altid fokussere på portens midterpunkt. Derved skulle hele porten helst altid være synlig på billedet, når den selvfølgelig eri tilpas afstand. Styresignalet til webcamet kan så også bruges til at styrerobotten imod midten af porten.

Denne automatiske styring er implementeret i funktionen automaticControl() i CEye-klassen. Se appendix. Denne funktion består af en reguleringssløjfe. Først tages et billede igennem billedfiltret, herefter kaldes funktionen getPortCenter(), for at beregne portens midterpunkt (omtalt tidligere). Når den er fundet justeres webcammet så det fokussere på det, det gøres med funktionen adjustCenter() fra CEye-klassen. adjustCenter() omdanner blot midtpunktets koordinat i billedet til en relativ bevægelse som sendes videre til webcam driveren. Se detaljer i kildekoden i appendix. På denne måde sørger webcammet for at portens centrum altid er så meget som muligt i midten af billedet.

6.4. Test på SMR

Styringen efter porte endte alligevel med at blive frakoblet på konkurrencedagen, eftersom det virkede alt for usikkert. Istedet blev det den traditionelle liniesensor som overtog kontrollen af robotten. For alligevel at få testet vision systemet i praksis, monterede jeg computeren og webcam på en af instituttets SMR-robotter. Fordelen ved at anvende en SMR til sådanne tests, er at den dels består af gennemtestet hardware, desuden kan den styres med det såkaldte SMR Command Language.

Grunden til at jeg ikke anvender SMRens egen computer til at køre min software på, er at den ikke har installeret den nyeste webcam driver, som understøtter

Logitech Sphere webcams. Det er derfor nemmere blot at montere min SBC oven på SMRen, og så kommunikere med SMRen via en netværksforbindelse. Ved hjælp af SMRCL kommandoer.

De før anvendte parametre til justering af kameraet, kan derfor anvendes direkte til styring af robotten. panAngle indeholder kameraets panoreringsvinkel, hvor negative grader er imod venstre og positive imod højre. tiltAngle er ikke interessant til dette formål, eftersom robotten jo ikke kan bevæge sig op og ned. Eftersom robotten kun skal styre efter porte, foregår udsendingen af kommandoer kun hvis isPort() = true. Styringen foregår så ved at sende følgende kommandoer, hver gang en port er opdaget i billedet.

```
smr.connectSMRCL("smr1", 31001);
snprintf(cmd, 100, "turn %f", -panAngle);
smr.smrMove(cmd, 20.0, &cond);
snprintf(cmd, 100, "fwd 0.1 @v0.2 @a0.5");
smr.smrMove(cmd, 20.0, &cond);
```

Første linie opretter en netværksforbindelse til SMR-robotten. De næste par linier sørger for at dreje robotten med den vinkel som kameraet blev drejet med. Efter at have ændret robottens kurs, sørger de to sidste linier for at robotten kører 10 cm frem ad den nye kurs.

Denne simple styringmetode virker absolut ikke hver gang den prøves. Robotten finder oftest en port, og begynder så at styre efter den. Men efter nogle reguleringer mister den oftest orientering og drejer enten ind i portens sidestolpe eller helt væk fra porten. Men jeg har dog også oplevet nogle gange hvor den har kørt igennem porten som den skulle. Grunden til de fejl der opstår, tror jeg bunder i enten at porten i billedet pludselig fylder så meget af billedet at den ikke ligner en port (fordi robotten er tæt på porten). En anden grund kunne være at at det hele køre temmelig asynkront. Her tænker jeg især på at: indlæsning af et billede, drejning af webcammet og styring af SMRen måske burde være mere synkronisseret. Således at der f.eks. ikke tages billeder imens webcammet eller SMRen er igang med at dreje. Men der er desværre ikke mere tid til at optimmere styringen i denne omgang.

Konklusion

Jeg har i det sidste års tid gået en del og tænkt over hvordan man kunne lave et vision system som kun interesserede sig for de vigtige ting i et billede, og ignorerede detaljer som lysstyrke og små textur foreskelle. Det har derfor været en fornøjelse endelig at få afprøvet min teori i praksis, selvom der er fremkommet en del problemstillinger jeg ikke havde regnet med. Der er utroligt mange ting der skal tages højde for når man arbejder med computer vision, og det er nok også derfor at jeg syntes netop det område er spændende.

Det er utroligt at noget der er så nemt for en menneske hjerne, er så svært at beskrive som algoritme til en computer! Det er pirrerende at vide inputtet til systemet (billedet) indeholder alle de informationer man skal bruge for at kunne styre robotten igennem banen, man skal ”bare” dekode dem!

En af de oprindelige idéer med at opbygget et lagdelt system var, udover at simplificere dataene efterhånden som de kom op igennem lagene, også at nogle fejl fra lavere lag nemmere kunne fjernes på højere lag. Den sidste idé har vist sig at være meget dårlig. For som med alle andre computer programmer, gælder reglen om ”garbage-in-garbage-out” stadig, og endnu værre i en lagdelt struktur. Her forstærkes fejlene faktisk efterhånden som de kommer op igennem lagene.

De fleste af de algoritmer som anvendes inden for vision, er et spørgsmål om en hardfin balancegang. De er stort set alle afhængige er af en finjusteret tærskelværdi. Hvis man endelig kan få fundet en som virker perfekt overalt på billedet, så virker den som regel ikke ligeså godt på et andet billede! Noget der virkelig har vagt min interesse er derfor udvikling af algoritmer uden ”magiske” konstante tærskelværdier!

Som det ses nåede jeg ikke helt så langt som jeg havde regnet med i første omgang. Selvom der i månederne op til RoboCup-konkurrencen blev arbejdet 3 dage om ugen, og tit aftenen med, hvilket nok må siges at være end 10 points tid, så var det altså ikke nok til at vision systemet kom til at styre RoboCup-robotten til finalen. Det er også altid svært at vurdere hvor lang tid de enkelte delopgaver tager, især når der er tale om et specialkursus. Men vi skal uden tvivl gøre et forsøg igen næste år!

Dette kursus har givet mig endnu mere blod på tanden til at arbejde videre med projekter inden for computer vision området. Selvom systemet ikke nåede at blive færdig, er jeg stadig overbevist om at det må være muligt at styre en RoboCup-robot blot ved hjælp af vision. Om ikke på den mest effektive måde, så under alle omstændigheder på en ny og anderledes måde, som ikke før er set i RoboCup-konkurrencen.

Figuroversigt

Appendix A – Kildekode

Appendix A1 – CCartoonPixel

```
/***************************************************************************
    ccartoonpixel.h  -   class representing a simplified pixel, having 64
                         gray level or 192 colors

    begin                : Mon Jan 12 2004
    copyright            : (C) 2004 by Allan Krogh Jensen
    email                : s973989@student.dtu.dk
 ***************************************************************************/

#ifndef CCARTOONPIXEL_H
#define CCARTOONPIXEL_H

#include <math.h>
#include "ucommon.h"
#include "umatrix.h"

//hardcoded filter parameters
#define INTENSITY_OUTPUT 128        // output intensity, for colors only
#define SATURATION_OUTPUT 128       // output saturation, for colors only

/** Look-up table holding the color/graylevel seperation surface used for the image filter
*/
//extern unsigned char GLMaxSat[255][255];

/**   *@author Allan Krogh Jensen   */
/**Class representing a simplified pixel, having 64 gray level or 192 colors*/
class CCartoonPixel
  {
  private:
    /** Holding the byte value of the pixel. Values from 0 to 63 are graylevels.
    Values from 64 to 255 are colors */
    unsigned char value;

    /**Returns pointers to floating IHS parameters of the pixel. All values are
    from 0 to 1 */
    void getIHS(float* i, float* h, float* s);

    /**Calculates and returns the maximum graylevel saturation, from a given hue.
    Value is 0..255 and intensity*/
    unsigned char getMaxGraylevelSat(float i, float h);

    /**Returns the output intensity of the pixel if it is a graylevel. */
    float getIntensity();

  public:
    /** Constructor */
    CCartoonPixel();

    /** Destructor */
    ~CCartoonPixel();

    /** Returns true if pixel is a graylevel */
    bool isGrayLevel();

    /** Input pixel values in IHS-format, into the filter. I should be between
    0 and sgrt(3). H should be between 0 and 2*PI. S should be between 0 and 1 */
    void inputIHS(float i, float h, float s);

    /** Input pixel values in IUV/YUV-format, into the filter. OBS! Input UPixel is
    assumed to be in IUV/YUV-format */
    void inputIUV(UPixel pix);

    /** Input pixel values in RGB-format, into the filter. OBS! Input UPixel is assumed
    to be in RGB-format */
    void inputRGB(UPixel pix);

    /** input a square pixel area of 2*2 into one cartoon pixel */
    void inputRGB_4to1(UPixel pix1, UPixel pix2, UPixel pix3, UPixel pix4);
```

```
    /** Returns output value from filter as an UPixel in IUV/YUV-format */
    UPixel getIUV();

    /** Returns output value from filter as an UPixel in RGB-format */
    UPixel getRGB();

    /** Improves the contrast if the pixel is a graylevel, to compensate
    for the dependence of the light condition. Must be called in each pixel
    in an image, after all pixels in picture has been given a value */
    void improveGrayLevel(float minGLIntensity, float maxGLIntensity);

    /** Updates the min and max intensity parameters given as arguments according the the
    actual intensity of the pixel iff it is a graylevel */
    void updateGLIExtremes(float* minGLIntensity, float* maxGLIntensity);

    /** Set the pixel value directly */
    void setValue(unsigned char val);

    /** Return the pixel value directly */
    unsigned char getValue();

    /** Returns true if there is an edge between the pixel and the pixel
    given as an argument */
    bool isEdge(CCartoonPixel pix);

    /**Loads the seperation surface from file into the GLMSat matrix */
    void initGLMSat();

    /** returns true if the pixel is yelleow (port colored!) */
    bool isYellow();

  };

#endif


/***************************************************************************
    ccartoonpixel.cpp  - class representing a simplified pixel, having 64
                         gray level or 192 colors

    begin                : Mon Jan 12 2004
    copyright            : (C) 2004 by Allan Krogh Jensen
    email                : s973989@student.dtu.dk
 ***************************************************************************/

#include "ccartoonpixel.h"

//unsigned char GLMaxSat[255][255];

//default constructor--------------------------------------------------------
CCartoonPixel::CCartoonPixel()
  {
  }

//Destructor-----------------------------------------------------------------
CCartoonPixel::~CCartoonPixel()
  {
  }

//isGrayLevel ----------------------------------------------------------------
bool CCartoonPixel::isGrayLevel()
  {
  return (value < 64);   // if 2 MSB of value is 0 the pixel
  }                      // is a graylevel

//inputIHS-------------------------------------------------------------------
void CCartoonPixel::inputIHS(float i, float h, float s)
  {  // i: intensity: 0..sqrt(3) h: hue: 0..2*PI s: saturation: 0..1
  if((s * 255.0) <= getMaxGraylevelSat(i, h))     //if the input is a graylevel..
    {      //convert the intensity into a graylevel from 0 to 63
    value = (unsigned char)((i / 1.732) * 63.0);                    //sqrt(3) = 1.732
    }
  else    //if the input is a color
    {       //convert the hue into a colorlevel from 64 to 255
    value = (unsigned char)(63 + ((h / (2.0 * PI)) * 192.0));
```

```
    }
  }

//inputIUV-------------------------------------------------------------------
void CCartoonPixel::inputIUV(UPixel pix)
  {  // filters an input pixel, assumes it is in IUV format
  float i, h, s;  // local hue and saturation values
  float u, v;     // local floating u and v values: 0..1

  if(pix.u >= 128 && pix.v >= 128)       // if pixel is in first quadrant
    {
    u = (pix.u - 128) / 128.0; //calculate the coresponding floating u and v
    v = (pix.v - 128) / 128.0; //range 0..1 from origo of SH-plane
    h = atan(v / u);           //calc. hue angle
    }
  else if(pix.u < 128 && pix.v >= 128)   // if pixel is in sec. quadrant
    {
    u = (127 - pix.u) / 127.0; //calculate the coresponding floating u and v
    v = (pix.v - 128) / 128.0; //range 0..1 from origo of SH-plane
    h = atan(v / u) + 0.5 * PI;//calc. hue angle
    }
  else if(pix.u < 128 && pix.v < 128)    // if pixel is in third quadrant
    {
    u = (127 - pix.u) / 127.0; //calculate the coresponding floating u and v
    v = (127 - pix.v) / 127.0; //range 0..1 from origo of SH-plane
    h = atan(v / u) + PI;      //calc. hue angle
    }
  else if(pix.u >= 128 && pix.v < 128)   // if pixel is in fourth quadrant
    {
    u = (pix.u - 128) / 128.0; // calculate the coresponding floating u and v
    v = (127 - pix.v) / 127.0; // range 0..1 from origo of SH-plane
    h = atan(v / u) + 1.5 * PI;// calc. hue angle
    }

  i = (pix.y / 255.0) * sqrt(3.0);
  s = sqrt(u * u + v * v);         // calc. saturation using good old Pythagoras
  inputIHS(i, h, s); // use this function for the further filtering
  }

//inputRGB-------------------------------------------------------------------
void CCartoonPixel::inputRGB(UPixel pix)
  {   //filters an input pixel, assumes it is in RGB format
  float i, h, s;          //local values of intensity, hue and saturation
  UMatrix4 res, tra, rgb; // matrices used for transformation to IHS

  rgb.init(3, 1);         // RGB input matrix
  tra.init(3, 3);         // transformation matrix
  res.init(3, 1);         // resulting matrix

  rgb.set(0, 0, pix.r / 255.0);    // insert red value in matrix
  rgb.set(1, 0, pix.g / 255.0);    // insert green value in matrix
  rgb.set(2, 0, pix.b / 255.0);    // insert red value in matrix

  tra.set(0, 0, 0.333);            // generate transformation matrix
  tra.set(0, 1, 0.333);            // used to transform RGB to IHS
  tra.set(0, 2, 0.333);
  tra.set(1, 0, -0.5);
  tra.set(1, 1, -0.5);
  tra.set(1, 2, 1.0);
  tra.set(2, 0, 0.866);
  tra.set(2, 1, -0.866);
  tra.set(2, 2, 0.0);

  res = tra * rgb;        // multiply RGB matrix with transformation matrix
  i = res.get(0, 0);      // get the intensity from resulting matrix

  h = atan2(res.get(2, 0), res.get(1, 0)); //calc. hue using special atan2()
  if(h < 0.0) h = h + 2.0 * PI;            //convert to non-negative angle
  if(h > PI) h = fmod(h, 2.0 * PI);

  s = sqrt(res.get(1, 0) * res.get(1, 0) +
    res.get(2, 0) * res.get(2, 0));        //calc. saturation

  inputIHS(i, h, s);  //use this function for the further filtering
  }
```

```
//getIHS-----------------------------------------------------------------------
void CCartoonPixel::getIHS(float* i, float* h, float* s)
  { // returns pointers to the IHS output value of the pixel
  if(isGrayLevel())      // if pixel is a greylevel
    {
    *i = getIntensity();   // calc. intensity only, h and s is set to 0.0
    *h = *s = 0.0;         // making output a pure greylevel
    }
  else     // if pixel is a colorlevel
    {
    *i = INTENSITY_OUTPUT / 255.0;          // intesity and saturation are
    *s = SATURATION_OUTPUT / 255.0;         // defined as constants for colors
    *h = ((value - 63) / 192.0) * 2 * PI;   // hue is in radians
    }
  }

//getIUV-----------------------------------------------------------------------------
UPixel CCartoonPixel::getIUV()
  {  // returns IUV values of the pixel
  float i, h, s;             // i,h,s values
  UPixel pix;                // local pixel to be returned

  getIHS(&i, &h, &s);                  // receive IHS values
  pix.y = (unsigned char)((i/sqrt(3.0)) * 255.0); // calc. byte value of intensity

  if(h <= 0.5 * PI)           //if pixel is in first quadrant
    {
    pix.u = (unsigned char)(128.0 + s * cos(h) * 128.0);
    pix.v = (unsigned char)(128.0 + s * sin(h) * 128.0);
    }
  else if(h <= PI)            //if pixel is in sec. quadrant
    {
    pix.u = (unsigned char)(128.0 - s * cos(h) * 128.0);
    pix.v = (unsigned char)(128.0 + s * sin(h) * 128.0);
    }
  else if(h <= 1.5 * PI)      //if pixel is in third quadrant
    {
    pix.u = (unsigned char)(128.0 - s * cos(h) * 128.0);
    pix.v = (unsigned char)(128.0 - s * sin(h) * 128.0);
    }
  else if(h < 2.0 * PI)       //if pixel is in fourth quadrant
    {
    pix.u = (unsigned char)(128.0 + s * cos(h) * 128.0);
    pix.v = (unsigned char)(128.0 - s * sin(h) * 128.0);
    }

  return pix;  //return IUV values in a UPixel
  }

//getRGB-----------------------------------------------------------------------------
UPixel CCartoonPixel::getRGB()
  {   // returns RGB values of the pixel
  float i, h, s;             // i,h,s values
  UPixel pix;                // local pixel to be returned
  UMatrix4 res, tra, inp;    // matrices used for transformation to RGB
  float v1, v2;              // values used for the transformation

  getIHS(&i, &h, &s);        // receive IHS values

  inp.init(3, 1);            // i,v1,v2 input matrix
  tra.init(3, 3);            // transformation matrix
  res.init(3, 1);            // resulting matrix

  v1 = s * cos(h);
  v2 = s * sin(h);
  inp.set(0, 0, i);            // insert i value in matrix
  inp.set(1, 0, v1);         // insert v1 value in matrix
  inp.set(2, 0, v2);         // insert v2 value in matrix

  tra.set(0, 0, 1.0);        // generate transformation matrix
  tra.set(0, 1, -0.333);     // used to transform IHS to RGB
  tra.set(0, 2, 0.577);
  tra.set(1, 0, 1.0);
  tra.set(1, 1, -0.333);
  tra.set(1, 2, -0.577);
  tra.set(2, 0, 1.0);
```

```
  tra.set(2, 1, 0.666);
  tra.set(2, 2, 0.0);

  res = tra * inp;    //multiply input matrix with transformation matrix
  pix.r = (unsigned char)(res.get(0,0) * 255.0);      //red
  pix.g = (unsigned char)(res.get(1,0) * 255.0);      //green
  pix.b = (unsigned char)(res.get(2,0) * 255.0);      //blue

  return pix;         //return RGB values in a UPixel
  }

//getMaxGraylevelSat-----------------------------------------------------------------
unsigned char CCartoonPixel::getMaxGraylevelSat(float i, float h)
  {   //returns the max. saturation af a grayscale for the given hue
  //int ii, ih;
  if(h <= (1.0/6.0) * PI) return (unsigned char)(45 / i);   //blue
  if(h <= (2.0/6.0) * PI) return (unsigned char)(40 / i);   //blue-magenta
  if(h <= (3.0/6.0) * PI) return (unsigned char)(40 / i);   //magenta
  if(h <= (4.0/6.0) * PI) return (unsigned char)(28 / i);   //magenta-red
  if(h <= (5.0/6.0) * PI) return (unsigned char)(10 * i);   //red
  if(h <= (6.0/6.0) * PI) return (unsigned char)(10 * i);   //orange
  if(h <= (7.0/6.0) * PI) return (unsigned char)(8 / i);    //yellow
  if(h <= (8.0/6.0) * PI) return (unsigned char)(10 / i);   //yellow-green
  if(h <= (9.0/6.0) * PI) return (unsigned char)(20 / i);   //green
  if(h <= (10.0/6.0) * PI) return (unsigned char)(28 / i);  //green-cyan
  if(h <= (11.0/6.0) * PI) return (unsigned char)(40 / i);  //cyan
  if(h <= (12.0/6.0) * PI) return (unsigned char)(40 / i);  //cyan-blue
  return (unsigned char)(40 / i);     //default should not be reachable!
  /*ii = (int)((i / sqrt(3.0)) * 255.0);
  ih = (int)((h / (2.0 * PI)) * 255.0);
  return GLMaxSat[ii][ih] * 255.0;*/
  }

//improveGrayLevels------------------------------------------------------------------
void CCartoonPixel::improveGrayLevel(float minGLIntensity, float maxGLIntensity)
  {
  float oldInten, newInten;

  if(isGrayLevel())   //improves the contrast if the pixel is a graylevel
    {
    oldInten = getIntensity();      //calculate old intensity
    newInten = (oldInten - minGLIntensity) / (maxGLIntensity - minGLIntensity);
    value = (unsigned char)(newInten * 63.0);
    }
  }

//resetStatics-----------------------------------------------------------------------
void CCartoonPixel::updateGLIExtremes(float* minGLIntensity, float* maxGLIntensity)
  {
  float intensity;

  if(isGrayLevel())                   //only if pixel is a graylevel
    {
    intensity = getIntensity();
    if(intensity < *minGLIntensity)       //if intensity is less min. intensity..
      *minGLIntensity = intensity;        //..update min. intensity
    else if(intensity > *maxGLIntensity)  //if intensity is higher than max. intensity..
      *maxGLIntensity = intensity;        //..update max. intensity
    }
  }

//getIntensity-----------------------------------------------------------------------
float CCartoonPixel::getIntensity()
  {
  return value * (sqrt(3.0) / 63.0);    //intensity: 0....sqrt(3)
  }

//setValue---------------------------------------------------------------------------
void CCartoonPixel::setValue(unsigned char val)
  {
  value = val;
  }

//getValue---------------------------------------------------------------------------
unsigned char CCartoonPixel::getValue()
  {
```

```
  return value;
  }

//isEdge-------------------------------------------------------------------------------
bool CCartoonPixel::isEdge(CCartoonPixel pix)
  {
  if(isGrayLevel() != pix.isGrayLevel())  //if one is color and other is graylevel
    {
    return true;       //there is an edge
    }
  else if(isGrayLevel())    //if they are both graylevels
    {
    if((value - pix.getValue() > 2) || (pix.getValue() - value > 2))
      return true;     //if they differs more than the threeshold its an edge
    else
      return false;
    }
  else                      //if they are both colors
    {
    if((value - pix.getValue() > 8) || (pix.getValue() - value > 8))
      return true;     //if they differs more than the threeshold its an edge
    else
      return false;
    }
  }

//initGLMSat---------------------------------------------------------------------------
void CCartoonPixel::initGLMSat()
  {
  /*unsigned int i, h;
  FILE* fp;

  fp = fopen( "filter1.dat", "r");

  for(i = 0; i < 256; i++)
    for(h = 0; h < 256; h++)
      fscanf(fp, "%u/n", &GLMaxSat[i][h]);
  fclose(fp);*/
  }

//inputRGB_4to1------------------------------------------------------------------------
void CCartoonPixel::inputRGB_4to1(UPixel pix1, UPixel pix2, UPixel pix3, UPixel pix4)
  {
  CCartoonPixel cp[4];
  unsigned int i, j;
  unsigned int average = 0;
  unsigned char maxDif = 0;
  unsigned char mostOutside = 0;
  unsigned char t = 8;

  cp[0].inputRGB(pix1);
  cp[1].inputRGB(pix2);
  cp[2].inputRGB(pix3);
  cp[3].inputRGB(pix4);

  //find most used color value
  for(i = 0; i < 4; i++)
    for(j = 0; j < 4; j++)
      if(cp[i].getValue() == cp[j].getValue())
        value = cp[i].getValue();     //use the most common value for the merged pixel

  //find average color value
  for(i = 0; i < 4; i++)
    average = average + cp[i].getValue();
  average = average / 4;

  //find most outside color value
  for(i = 0; i < 4; i++)
    if(average > cp[i].getValue())
      if(average - cp[i].getValue() > maxDif)
        {
        maxDif = average - cp[i].getValue();
        if(maxDif >= t)
          value = cp[i].getValue();
        }
    else
```

```
      if(cp[i].getValue() - average > maxDif)
        {
        maxDif = cp[i].getValue() - average;
        if(maxDif >= t)
          value = cp[i].getValue();
        }
  }

//isYellow-------------------------------------------------------------------------
bool CCartoonPixel::isYellow()
  {
  return ((value > (160 - 10)) && (value < (160 + 10)));
   }
```

Appendix A2 – CCartoonImage

```
/***************************************************************
    ccartoonimage.h  -  class representing an image of simplified pixels
                        used for filtering

    begin                : Mon Jan 12 2004
    copyright            : (C) 2004 by Allan Krogh Jensen
    email                : s973989@student.dtu.dk
 ***************************************************************/

#ifndef CCARTOONIMAGE_H
#define CCARTOONIMAGE_H

#include "cstopwatch.h"
#include "ccartoonpixel.h"
#include "urawimage.h"
#include "cedgeline.h"
#include "c2dpolygon.h"

#define IMAGE_WIDTH 640           //max. image dimensions
#define IMAGE_HEIGHT 480

//Output Picture Options
#define SHOW_CARTOON_PIXELS
//#define SHOW_EDGE_IMAGE
//#define SHOW_EDGE_NODES
//#define SHOW_EDGE_LINES
//#define SHOW_POLYGONS
#define SHOW_PORT_POLYGON

/** *@author Allan Krogh Jensen   */
/** Class representing an image of simplified pixels used for filtering */
class CCartoonImage
  {
  private:
    /** Holding all the CCartoonPixels of the image */
    CCartoonPixel image[IMAGE_WIDTH][IMAGE_HEIGHT];

    /** true if an edge were found on specified location */
    bool edgeImage[IMAGE_WIDTH][IMAGE_HEIGHT];

    /** true if the point has been included in a edge-line */
    bool visited[IMAGE_WIDTH][IMAGE_HEIGHT];

    /** Edgelines in the image */
    CEdgeLine edgeLine[2000];

    /** number of total edge-lines */
    int totalEdgeLines;

    /** polygons int the image */
    C2DPolygon polygon[100];

    /** number of total polygons */
    int totalPolygons;

    float minGLIntensity;  //min. graylevel intensity of all pixels in the image
    float maxGLIntensity;  //max. graylevel intensity of all pixels in the image
```

```
    /** actual image dimensions */
    int height;
    int width;

    /** Finding max of the given arguments */
    float max(float a, float b);

    /** Generating Edge Lines from the edge points in the edge image */
    void generateEdgeLines();

    /** Returns true if an edge-point is also an edge-node */
    bool isEdgeNode(int x, int y);

    /** thin edge around specified edge-point */
    void thin(int x, int y);

    /** thin edge around specified edge-point using L8-thinning */
    bool thin_L8(int x, int y);

    /** Finds edges in the image. Edges are
    stored in the edgeImage */
    void findEdges();

    /** Find edges in the image using the Gamma Ratio Edge Detector algorithm. Edges are
    stored in the edgeImage */
    void findEdges_GRED();

    /** Find edges in the image using the Gamma Ratio Edge Detector algorithm, with
thinning.
    Edges are stored in the edgeImage */
    void findEdges_ThinGRED();

    /** find edges in the image using the SUSAN edge detection algorithm. Edges are
    stored in the edgeImage */
    void findEdges_SUSAN();

    /** set edge-line side colors */
    void setELSColors(CEdgeLine* el);

    /** unvisit specified edge-line in the visit image */
    void unvisitEdgeLine(CEdgeLine el);

    /** improves found edges by thinning and filling gabs */
    void improveEdges();

    /** stretches edge-lines so that the small spaces between them are removed */
    void improveEdgeLines();

    /** expand edge to the image border */
    void edgeToBorder(int x, int y);

    /** init edgeImage with a border of edge-points */
    void initEdgeImage();

    /** Generate polygons from the edge-lines in the image */
    void generatePolygons();

    /** Find a specific port-polygon in the image */
    void findPortPolygon();

  public:
    /** port-polygon (largest yellow polygon) */
    C2DPolygon portPolygon;

    /** line-polygon (largest white polygon in image) */
    C2DPolygon linePolygon;

    /** Constructor */
    CCartoonImage();

    /** Destructor */
    ~CCartoonImage();

    /** Input an image into the filter for processing */
    void inputImage(UImage* inpImage, bool oneToOne);
```

```
    /** Get the filtered output image */
    void getImage(UImage* retImage);

    /** getPortCenter */
    void getPortCenter(int* centerX, int* centerY);

  };

#endif

/***************************************************************************
    ccartoonimage.cpp  -  class representing an image of simplified pixels
                          used for filtering

    begin                : Mon Jan 12 2004
    copyright            : (C) 2004 by Allan Krogh Jensen
    email                : s973989@student.dtu.dk
 ***************************************************************************/

#include "ccartoonimage.h"

//default constructor--------------------------------------------------------
CCartoonImage::CCartoonImage()
  {
  //image[0][0].initGLMSat();

  minGLIntensity = sqrt(3) / 2;   //init both extremes to half of max. intensity
  maxGLIntensity = sqrt(3) / 2;   //in the IHS model: sqrt(3)
  }

//default destructor---------------------------------------------------------
CCartoonImage::~CCartoonImage()
  {
  }

//max------------------------------------------------------------------------
float CCartoonImage::max(float a, float b)
  {
  if(a >= b)
    return a;
  else
    return b;
  }

//input----------------------------------------------------------------------------
void CCartoonImage::inputImage(UImage* inpImage, bool oneToOne)
  {         //assumes image is in RGB format
  int row, column;     //pixel position counters
  int x ,y;
  UPixel newInpPix, preInpPix;  //new input pixel and previous input pixel
  CCartoonPixel preCarPix;      //previous cartoon pixel
  CStopWatch sw;
  int centerX = 0, centerY = 0;

  printf("Filtering input image...");
  sw.start();

  if(oneToOne)                 //filter image in ratio 1 to 1
    {
    height = inpImage->height;
    width = inpImage->width;

    for(row = 0; row < height; row++)   // filter image pixel by pixel
      for(column = 0; column < width; column++)
        {
        newInpPix = inpImage->GetPix(row, column);    //get new pixel from input image
        if(newInpPix == preInpPix)      //if new input pixel is the same as the previous..
          {
          image[column][row].setValue(preCarPix.getValue());   //..use the same value
          }
        else                     //if it is different from the previous pixel..
          {
          preCarPix.inputRGB(newInpPix);        //load RGB values into pixel
          if(!preCarPix.isYellow())     //only use pixel if it is yellow
             preCarPix.setValue(63);       //all others are white
```

```
          image[column][row] = preCarPix;
          image[column][row].updateGLIExtremes(&minGLIntensity, &maxGLIntensity);
          preInpPix = newInpPix;
          }
        }
      }
    else        //filter image in ratio 4 to 1
      {
      for(row = 0; row < height; row=row+2)   // filter image
        for(column = 0; column < width; column=column+2)
          {
          x = column / 2;
          y = row / 2;
          image[x][y].inputRGB_4to1(inpImage->GetPix(row, column),
                                    inpImage->GetPix(row, column+1),
                                    inpImage->GetPix(row+1, column),
                                    inpImage->GetPix(row+1, column+1));
          }
      height = inpImage->height / 2;
      width = inpImage->width / 2;
      }

  printf("\t\tDone! in %f s out size (%ux%u)\n", sw.stop(), width, height);

  //findEdges_SUSAN();                //Edge Detection Layer
  //findEdges_GRED();
  //findEdges_ThinGRED();
  //improveEdges();
  //generateEdgeLines();                //Edge Line Layer
  //improveEdgeLines();
  //generatePolygons();                 //Polygon Layer
  portPolygon.reset();
  findPortPolygon();
  portPolygon.getCenter(&centerX, &centerY);
  printf("Focus Point of Port: (%u, %u)\n", centerX, centerY);
  }

//input-------------------------------------------------------------------------
void CCartoonImage::getImage(UImage* retImage)
  {
  int row, column;    //pixel position counters
  UPixel edgePix;     //pixel value used to display an edge
  CEdgeLine el;       //testing
  CStopWatch sw;      //stop watch

  printf("Extracting output to BMP-image...");
  sw.start();

  retImage->width = width;
  retImage->height = height;

  #ifdef SHOW_CARTOON_PIXELS
  for(row = 0; row < height; row++)   //get cartoon image pixel by pixel
    for(column = 0; column < width; column++)
      //if(image[column][row].isYellow())   //only show yellew pixels in output
        retImage->SetPix(row, column, image[column][row].getRGB());
  #endif

  #ifdef SHOW_EDGE_IMAGE
  edgePix.r = 0;
  edgePix.g = 128;
  edgePix.b = 0;

  //display found edges on output image
  for(row = 0; row < height; row++)
    for(column = 0; column < width; column++)
      if(edgeImage[column][row])    //if edge was found on location
        retImage->SetPix(row, column, edgePix);  //..draw an edge pixel
  #endif

  #ifdef SHOW_EDGE_NODES
  //display edge-nodes on the output image
  edgePix.r = 0;
  edgePix.g = 0;
  edgePix.b = 255;
```

```
  for(row = 0; row < height; row++)
    for(column = 0; column < width; column++)
      if(isEdgeNode(column, row))    //if point is an edge-node
        retImage->SetPix(row, column, edgePix);   //..draw an node pixel
  #endif

  #ifdef SHOW_EDGE_LINES
  edgePix.r = 0;
  edgePix.g = 255;
  edgePix.b = 0;

  for(row = 0; row < totalEdgeLines; row++)    //draw all edge-lines
    edgeLine[row].draw(retImage, edgePix, true, true);  //with color spots
  #endif

  #ifdef SHOW_POLYGONS
  edgePix.r = 0;
  edgePix.g = 255;
  edgePix.b = 0;

  for(row = 0; row < totalPolygons; row++)    //draw all polygons
    if(polygon[row].isClosed)       //only draw closed polygons
      polygon[row].paint(retImage);
  #endif

  #ifdef SHOW_PORT_POLYGON
  portPolygon.paint(retImage);
  portPolygon.drawCorners(retImage);
  #endif

  printf("\tDone! in %f s picture size (%ux%u)\n", sw.stop(), width, height);
  }

//findEdges-------------------------------------------------------------------------
void CCartoonImage::findEdges()
  {
  int row, column;     //pixel position counters

  for(row = 0; row < height; row++)   //loop image pixel by pixel
    for(column = 0; column < width; column++)
      {
      image[column][row].improveGrayLevel(minGLIntensity, maxGLIntensity);

      //use the 8 surounding pixels to find edges
      //if((row - 1) >= 0 && (column - 1) >= 0)             //top left
        if(image[column][row].isEdge(image[column-1][row-1]))
          edgeImage[column][row] = true;
        else
          edgeImage[column][row] = false;
      //if((row - 1) >= 0 )                                 //top center
        if(image[column][row].isEdge(image[column][row-1]))
          edgeImage[column][row] = true;
        else
          edgeImage[column][row] = false;
      //if((row - 1) >= 0 && (column + 1) < width)          //top right
        if(image[column][row].isEdge(image[column+1][row-1]))
          edgeImage[column][row] = true;
        else
          edgeImage[column][row] = false;
      //if((column - 1) >= 0)                               //previous
        if(image[column][row].isEdge(image[column-1][row]))
          edgeImage[column][row] = true;
        else
          edgeImage[column][row] = false;
      /*if((column + 1) < width)                            //next
        if(image[column][row].isEdge(image[column+1][row]))
          edgeImage[column][row] = true;
        else
          edgeImage[column][row] = false;
      if((row + 1) < height && (column - 1) >= 0)         //bottom left
        if(image[column][row].isEdge(image[column-1][row+1]))
          edgeImage[column][row] = true;
        else
          edgeImage[column][row] = false;
      if((row + 1) < height)                              //bottom center
        if(image[column][row].isEdge(image[column][row+1]))
```

```
          edgeImage[column][row] = true;
        else
          edgeImage[column][row] = false;
      if((row + 1) < height && (column + 1) < width)      //bottom right
        if(image[column][row].isEdge(image[column+1][row+1]))
          edgeImage[column][row] = true;
        else
          edgeImage[column][row] = false; */
      }

  }

//findEdgesGRED-------------------------------------------------------------------------
void CCartoonImage::findEdges_GRED()
  {    //find edges using the Gamma Ratio Edge Detector algorithm
  register int row, column;                //pixel position counters
  float rEW, rNESW, rNS, rNWSE;   //ratio for the 4 edge directions
  float x11, x12, x13;            //pixel values in the search window
  float x21, x22, x23;
  float x31, x32, x33;
  float r;                        //max. ratio
  float t = 1.1;                  //threeshold
  CStopWatch sw;

  printf("Finding edges using GRED...");
  sw.start();

  for(row = 1; row < height - 1; row++)   //loop image pixel by pixel
    for(column = 1; column < width - 1; column++)
      {
      x11 = image[column-1][row-1].getValue();
      x12 = image[column][row-1].getValue();
      x13 = image[column+1][row-1].getValue();
      x21 = image[column-1][row].getValue();
      x22 = image[column][row].getValue();
      x23 = image[column+1][row].getValue();
      x31 = image[column-1][row+1].getValue();
      x32 = image[column][row+1].getValue();
      x33 = image[column+1][row+1].getValue();
      rEW = (x11 + x12 + x13) / (x31 + x32 + x33);
      rNESW = (x21 + x11 + x12) / (x32 + x33 + x23);
      rNS = (x11 + x21 + x31) / (x13 + x23 + x33);
      rNWSE = (x21 + x31 + x32) / (x12 + x13 + x23);
      r = max(rEW, max(rNESW, max(rNS, max(rNWSE, max(1/rEW, max(1/rNESW, max(1/rNS,
1/rNWSE)))))));
      if(r > t)           //if ratio is greater than an threshold there is an edge
        {
        edgeImage[column][row] = true;
        //edgeToBorder(column, row);    //if edge-point is close to border, expand to
border
        }
      else
        edgeImage[column][row] = false;
      visited[column][row] = false;
      }
  printf("\t\tDone! in %f s\n", sw.stop());
  }

//thin----------------------------------------------------------------------------------
--
void CCartoonImage::thin(int x, int y)
  {   //thin an edge structure (5*5) around specified point
  if(edgeImage[x-2][y] && edgeImage[x-1][y] && edgeImage[x+1][y] && edgeImage[x+2][y])
    {
    edgeImage[x-2][y-1] = false; //delete edge-points thickening this line
    edgeImage[x-1][y-1] = false; //XXXXX
    edgeImage[x][y-1] = false;   //00000
    edgeImage[x+1][y-1] = false; //11111
    edgeImage[x+2][y-1] = false; //XXXXX
    }                            //XXXXX
  else if(edgeImage[x-2][y-1] && edgeImage[x-1][y-1] && edgeImage[x+1][y+1] &&
edgeImage[x+2][y+1])
    {
    edgeImage[x-2][y-2] = false; //delete edge-points thickening this line
    edgeImage[x-1][y-2] = false;
    edgeImage[x][y-1] = false;   //000XX
```

```
    edgeImage[x+1][y] = false;   //1100X
    edgeImage[x+2][y] = false;   //XX100
    edgeImage[x][y-2] = false;   //XXX11
    edgeImage[x+1][y-1] = false; //XXXXX
    }
  else if(edgeImage[x-2][y-2] && edgeImage[x-1][y-1] && edgeImage[x+1][y+1] &&
edgeImage[x+2][y+2])
    {
    edgeImage[x-1][y-2] = false; //delete edge-points thickening this line
    edgeImage[x][y-1] = false;
    edgeImage[x+1][y] = false;   //100XX
    edgeImage[x+2][y+1] = false; //X100X
    edgeImage[x][y-2] = false;   //XX100
    edgeImage[x+1][y-1] = false; //XXX10
    edgeImage[x+2][y] = false;   //XXXX1
    }
  else if(edgeImage[x-1][y-2] && edgeImage[x-1][y-1] && edgeImage[x+1][y+1] &&
edgeImage[x+1][y+2])
    {
    edgeImage[x-2][y-2] = false;  //delete edge-points thickening this line
    edgeImage[x-2][y-1] = false;
    edgeImage[x-1][y] = false;    //01XXX
    edgeImage[x][y+1] = false;    //01XXX
    edgeImage[x][y+1] = false;    //001XX
    edgeImage[x-2][y] = false;    //XX01X
    edgeImage[x-1][y+1] = false;  //XX01X
    }
  else if(edgeImage[x][y-2] && edgeImage[x][y-1] && edgeImage[x][y+1] &&
edgeImage[x][y+2])
    {
    edgeImage[x-1][y-2] = false;  //delete edge-points thickening this line
    edgeImage[x-1][y-1] = false;  //X01XX
    edgeImage[x-1][y] = false;    //X01XX
    edgeImage[x-1][y+1] = false;  //X01XX
    edgeImage[x-1][y+2] = false;  //X01XX
    }                             //X01XX
  else if(edgeImage[x+1][y-2] && edgeImage[x+1][y-1] && edgeImage[x-1][y+1] &&
edgeImage[x-1][y+2])
    {
    edgeImage[x][y-2] = false;    //delete edge-points thickening this line
    edgeImage[x][y-1] = false;
    edgeImage[x-1][y] = false;    //XX01X
    edgeImage[x-2][y+1] = false;  //X001X
    edgeImage[x-2][y+2] = false;  //001XX
    edgeImage[x-1][y-1] = false;  //01XXX
    edgeImage[x-2][y] = false;    //01XXX
    }
  else if(edgeImage[x+2][y-2] && edgeImage[x+1][y-1] && edgeImage[x-1][y+1] &&
edgeImage[x-2][y+2])
    {
    edgeImage[x+1][y-2] = false;  //delete edge-points thickening this line
    edgeImage[x][y-1] = false;
    edgeImage[x-1][y] = false;    //XX001
    edgeImage[x-2][y+1] = false;  //X001X
    edgeImage[x-2][y] = false;    //001XX
    edgeImage[x-1][y-1] = false;  //01XXX
    edgeImage[x][y-2] = false;    //1XXXX
    }
  else if(edgeImage[x+2][y-1] && edgeImage[x+1][y-1] && edgeImage[x-1][y+1] &&
edgeImage[x-2][y+1])
    {
    edgeImage[x-2][y] = false;    //delete edge-points thickening this line
    edgeImage[x-1][y] = false;
    edgeImage[x-1][y-1] = false;  //XX000
    edgeImage[x][y-2] = false;    //X0011
    edgeImage[x][y-1] = false;    //001XX
    edgeImage[x+1][y-2] = false;  //11XXX
    edgeImage[x+2][y-2] = false;  //XXXXX
    }
  else if(edgeImage[x-2][y+1] && edgeImage[x-1][y] && edgeImage[x+1][y] &&
edgeImage[x+2][y-1])
    {
    edgeImage[x-2][y] = false;    //delete edge-points thickening this line
    edgeImage[x-2][y-1] = false;
    edgeImage[x-1][y-1] = false;  //XXX00
    edgeImage[x][y-1] = false;    //00001
```

```
    edgeImage[x+1][y-1] = false;  //0111X
    edgeImage[x+1][y-2] = false;  //1XXXX
    edgeImage[x+2][y-2] = false;  //XXXXX
    }
  else if(edgeImage[x-2][y-1] && edgeImage[x-1][y] && edgeImage[x+1][y] &&
edgeImage[x+2][y+1])
    {
    edgeImage[x-2][y-2] = false;  //delete edge-points thickening this line
    edgeImage[x-1][y-2] = false;
    edgeImage[x-1][y-1] = false;  //00XXX
    edgeImage[x][y-1] = false;    //10000
    edgeImage[x+1][y-1] = false;  //X1110
    edgeImage[x+2][y-1] = false;  //XXXX1
    edgeImage[x+2][y] = false;    //XXXXX
    }
  else if(edgeImage[x-1][y+2] && edgeImage[x][y+1] && edgeImage[x][y-1] &&
edgeImage[x+1][y-2])
    {
    edgeImage[x-2][y+2] = false;  //delete edge-points thickening this line
    edgeImage[x-2][y+1] = false;
    edgeImage[x-1][y+1] = false;  //X001X
    edgeImage[x-1][y] = false;    //X01XX
    edgeImage[x-1][y-1] = false;  //X01XX
    edgeImage[x-1][y-2] = false;  //001XX
    edgeImage[x][y-2] = false;    //01XXX
    }
  else if(edgeImage[x-1][y-2] && edgeImage[x][y-1] && edgeImage[x][y+1] &&
edgeImage[x+1][y+2])
    {
    edgeImage[x-2][y-2] = false;  //delete edge-points thickening this line
    edgeImage[x-2][y-1] = false;
    edgeImage[x-1][y-1] = false;  //01XXX
    edgeImage[x-1][y] = false;    //001XX
    edgeImage[x-1][y+1] = false;  //X01XX
    edgeImage[x-1][y+2] = false;  //X01XX
    edgeImage[x][y+2] = false;    //X001X
    }
  }

//findEdges_ThinGRED-----------------------------------------------------------------------
void CCartoonImage::findEdges_ThinGRED()
  {    //find edges using the Gamma Ratio Edge Detector algorithm and thinning
  int row, column;               //pixel position counters
  float rEW, rNESW, rNS, rNWSE;   //ratio for the 4 edge directions
  float x11, x12, x13;            //pixel values in the search window
  float x21, x22, x23;
  float x31, x32, x33;
  float r;                        //max. ratio
  float t = 1.1;                  //threeshold

  //OBS! does not image border pixels yet
  for(row = 4; row < height - 4; row++)   //loop image pixel by pixel
    for(column = 4; column < width - 4; column++)
      {
      x11 = image[column-1][row-1].getValue();
      x12 = image[column][row-1].getValue();
      x13 = image[column+1][row-1].getValue();
      x21 = image[column-1][row].getValue();
      x22 = image[column][row].getValue();
      x23 = image[column+1][row].getValue();
      x31 = image[column-1][row+1].getValue();
      x32 = image[column][row+1].getValue();
      x33 = image[column+1][row+1].getValue();
      rEW = (x11 + x12 + x13) / (x31 + x32 + x33);
      rNESW = (x21 + x11 + x12) / (x32 + x33 + x23);
      rNS = (x11 + x21 + x31) / (x13 + x23 + x33);
      rNWSE = (x21 + x31 + x32) / (x12 + x13 + x23);
      r = max(rEW, max(rNESW, max(rNS, max(rNWSE, max(1/rEW, max(1/rNESW, max(1/rNS,
1/rNWSE)))))));
      if(r > t)           //if ratio is greater than an threshold there is an edge
        edgeImage[column][row] = true;
      else
        edgeImage[column][row] = false;
      visited[column][row] = false;
      //if(edgeImage[column-2][row-2]) thin(column-2, row-2);   //thin completed square
(5*5)
```

```
      if(edgeImage[column-2][row-1]) thin_L8(column-2, row-1);  //thin completed square
(3*3) using L8
      if(edgeImage[column-1][row-1]) thin_L8(column-1, row-1);
      }
  }

//findEdgesSUSAN-------------------------------------------------------------------------
void CCartoonImage::findEdges_SUSAN()
  {
  register int row, column;
  unsigned char I[40];          //values of the other pixels in the scan window circle
  int r;                        //index of a pixel in the window circle
  int n;                        //number of pixels in the USAN-area (pixels of same value)
  unsigned char t=1;            //difference threeshold
  int n_max=36;                 //number of pixels in the circle window
  int g = (3.0/4.0) * n_max;    //geometric threeshold

  //OBS! does not image border pixels yet
  for(row = 3; row < height - 3; row++)           //scan image pixel by pixel
    for(column = 3; column < width - 3; column++)
      {
      I[0] = image[column-1][row-1].getValue();   //inner circle
      I[1] = image[column][row-1].getValue();
      I[2] = image[column+1][row-1].getValue();
      I[3] = image[column-1][row].getValue();
      I[4] = image[column+1][row].getValue();
      I[5] = image[column-1][row+1].getValue();
      I[6] = image[column][row+1].getValue();
      I[7] = image[column+1][row+1].getValue();
      I[8] = image[column-2][row-2].getValue();   //2nd inner circle
      I[9] = image[column-1][row-2].getValue();
      I[10] = image[column][row-2].getValue();
      I[11] = image[column+1][row-2].getValue();
      I[12] = image[column+2][row-2].getValue();
      I[13] = image[column-2][row+2].getValue();
      I[14] = image[column-1][row+2].getValue();
      I[15] = image[column][row+2].getValue();
      I[16] = image[column+1][row+2].getValue();
      I[17] = image[column+2][row+2].getValue();
      I[18] = image[column-2][row-1].getValue();
      I[19] = image[column-2][row].getValue();
      I[20] = image[column-2][row+1].getValue();
      I[21] = image[column+2][row-1].getValue();
      I[22] = image[column+2][row].getValue();
      I[23] = image[column+2][row+1].getValue();
      I[24] = image[column-1][row-3].getValue();  //outer circle
      I[25] = image[column][row-3].getValue();
      I[26] = image[column+1][row-3].getValue();
      I[27] = image[column+3][row-1].getValue();
      I[28] = image[column+3][row].getValue();
      I[29] = image[column+3][row+1].getValue();
      I[30] = image[column-1][row+3].getValue();
      I[31] = image[column][row+3].getValue();
      I[32] = image[column+1][row+3].getValue();
      I[33] = image[column-3][row-1].getValue();
      I[34] = image[column-3][row].getValue();
      I[35] = image[column-3][row+1].getValue();

      n=0;
      for(r = 0; r < n_max; r++)
        if(abs(I[r] - image[column][row].getValue()) <= t)
          n++;      //calc. amount of pixels with same value in window

      //store edge response in edge map
      if(n < g)
        edgeImage[column][row] = true;
      else
        edgeImage[column][row] = false;

      visited[column][row] = false;
      }
  }

//generateEdgeLines----------------------------------------------------------------------
void CCartoonImage::generateEdgeLines()
  {
```

```
  int edgeLineNo = 0;      //actual edgeline
  register int newX, newY;          //next edge-point
  register int column, row;         //start point of edge-line
  int minELLength = 5;     //min. accepted lenght of an edge-line
  newX = newY = 0;
  CStopWatch sw;

  printf("Generating edge-lines...");
  sw.start();

  initEdgeImage();        //generate edge-point border

  for(row = 1; row < height - 1; row++)           //loop edge-image point by point
    for(column = 1; column < width - 1; column++) //..only inside the edge-point border
frame
      if(edgeImage[column][row] && !visited[column][row])   //take an unvisited edge-
point..
        {
        newX = column;
        newY = row;

        while(edgeLine[edgeLineNo].addPoint(newX, newY) && !isEdgeNode(newX, newY))
          {   //as long as a new edge-point can be added to edge-line, and it is not an
edge node
          visited[newX][newY] = true;

          //find first nabour edge-point
          if(edgeImage[newX-1][newY+1] && !visited[newX-1][newY+1])
            {
            newX = newX - 1;              //* 1 *
            newY = newY + 1;              //1 * *
            }
          else if(edgeImage[newX][newY+1] && !visited[newX][newY+1])
            {
            newX = newX;                  //* 1 *
            newY = newY + 1;              //0 1 *
            }
          else if(edgeImage[newX+1][newY+1] && !visited[newX+1][newY+1])
            {
            newX = newX + 1;              //* 1 *
            newY = newY + 1;              //0 0 1
            }
          else if(edgeImage[newX+1][newY] && !visited[newX+1][newY])
            {
            newX = newX + 1;              //* 1 1
            newY = newY;                  //0 0 0
            }
          else                            //* * 0
            break;                        //0 0 0
          }
        //end edge-line
        if((newX != column || newY != row) && edgeLine[edgeLineNo].getAppLenght() >=
minELLength)
          {    //if usable edge-line found...
          setELSColors(&edgeLine[edgeLineNo]);   //set side colors of the edge-line
          edgeLineNo++;                         //save edge-line
          }
        else                              //else discard edge-line
          {
          unvisitEdgeLine(edgeLine[edgeLineNo]);  //mark edge-points in it as un-visited
          edgeLine[edgeLineNo].reset();    //reset edge-line and overwrite with next edge-
line
          }
        }
  totalEdgeLines = edgeLineNo;      //update total number of edge-lines

  printf("\t\tDone! in %f s making in %u edge-lines\n", sw.stop(), totalEdgeLines);
  }

//isEdgeNode-------------------------------------------------------------------------
bool CCartoonImage::isEdgeNode(int x, int y)
  {
  if(!edgeImage[x-1][y-1] &&  edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&
      edgeImage[x-1][y]   &&  edgeImage[x][y]   &&  edgeImage[x+1][y]   &&
     !edgeImage[x-1][y+1] && !edgeImage[x][y+1] && !edgeImage[x+1][y+1])
     return true;
```

```
  if(!edgeImage[x-1][y-1] && !edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&
      edgeImage[x-1][y]   &&  edgeImage[x][y]   &&  edgeImage[x+1][y]   &&
     !edgeImage[x-1][y+1] &&  edgeImage[x][y+1] && !edgeImage[x+1][y+1])
     return true;
  if(!edgeImage[x-1][y-1] &&  edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&
      edgeImage[x-1][y]   &&  edgeImage[x][y]   && !edgeImage[x+1][y]   &&
     !edgeImage[x-1][y+1] && !edgeImage[x][y+1] &&  edgeImage[x+1][y+1])
     return true;
  if(!edgeImage[x-1][y-1] && !edgeImage[x][y-1] &&  edgeImage[x+1][y-1] &&
      edgeImage[x-1][y]   &&  edgeImage[x][y]   && !edgeImage[x+1][y]   &&
     !edgeImage[x-1][y+1] &&  edgeImage[x][y+1] && !edgeImage[x+1][y+1])
     return true;
  if(!edgeImage[x-1][y-1] &&  edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&
      edgeImage[x-1][y]   &&  edgeImage[x][y]   && !edgeImage[x+1][y]   &&
     !edgeImage[x-1][y+1] &&  edgeImage[x][y+1] && !edgeImage[x+1][y+1])
     return true;
  if(!edgeImage[x-1][y-1] &&  edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&
     !edgeImage[x-1][y]   &&  edgeImage[x][y]   &&  edgeImage[x+1][y]   &&
     !edgeImage[x-1][y+1] &&  edgeImage[x][y+1] && !edgeImage[x+1][y+1])
     return true;
  if( edgeImage[x-1][y-1] && !edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&
     !edgeImage[x-1][y]   &&  edgeImage[x][y]   &&  edgeImage[x+1][y]   &&
     !edgeImage[x-1][y+1] &&  edgeImage[x][y+1] && !edgeImage[x+1][y+1])
     return true;
  if(!edgeImage[x-1][y-1] &&  edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&
     !edgeImage[x-1][y]   &&  edgeImage[x][y]   &&  edgeImage[x+1][y]   &&
      edgeImage[x-1][y+1] && !edgeImage[x][y+1] && !edgeImage[x+1][y+1])
     return true;
  if(!edgeImage[x-1][y-1] &&  edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&
     !edgeImage[x-1][y]   &&  edgeImage[x][y]   && !edgeImage[x+1][y]   &&
      edgeImage[x-1][y+1] && !edgeImage[x][y+1] &&  edgeImage[x+1][y+1])
     return true;
  if( edgeImage[x-1][y-1] && !edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&
     !edgeImage[x-1][y]   &&  edgeImage[x][y]   &&  edgeImage[x+1][y]   &&
      edgeImage[x-1][y+1] && !edgeImage[x][y+1] && !edgeImage[x+1][y+1])
     return true;
  if( edgeImage[x-1][y-1] && !edgeImage[x][y-1] &&  edgeImage[x+1][y-1] &&
     !edgeImage[x-1][y]   &&  edgeImage[x][y]   && !edgeImage[x+1][y]   &&
     !edgeImage[x-1][y+1] &&  edgeImage[x][y+1] && !edgeImage[x+1][y+1])
     return true;
  if(!edgeImage[x-1][y-1] && !edgeImage[x][y-1] &&  edgeImage[x+1][y-1] &&
      edgeImage[x-1][y]   &&  edgeImage[x][y]   && !edgeImage[x+1][y]   &&
     !edgeImage[x-1][y+1] && !edgeImage[x][y+1] &&  edgeImage[x+1][y+1])
     return true;
  if(!edgeImage[x-1][y-1] &&  edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&
      edgeImage[x-1][y]   &&  edgeImage[x][y]   &&  edgeImage[x+1][y]   &&
     !edgeImage[x-1][y+1] &&  edgeImage[x][y+1] && !edgeImage[x+1][y+1])
     return true;
  if( edgeImage[x-1][y-1] && !edgeImage[x][y-1] &&  edgeImage[x+1][y-1] &&
     !edgeImage[x-1][y]   &&  edgeImage[x][y]   && !edgeImage[x+1][y]   &&
      edgeImage[x-1][y+1] && !edgeImage[x][y+1] &&  edgeImage[x+1][y+1])
     return true;
  if( edgeImage[x-1][y-1] && !edgeImage[x][y-1] &&  edgeImage[x+1][y-1] &&
     !edgeImage[x-1][y]   &&  edgeImage[x][y]   && !edgeImage[x+1][y]   &&
      edgeImage[x-1][y+1] && !edgeImage[x][y+1] &&  edgeImage[x+1][y+1])
     return true;
  if(!edgeImage[x-1][y-1] && !edgeImage[x][y-1] &&  edgeImage[x+1][y-1] &&
     !edgeImage[x-1][y]   &&  edgeImage[x][y]   && !edgeImage[x+1][y]   &&
      edgeImage[x-1][y+1] && !edgeImage[x][y+1] &&  edgeImage[x+1][y+1])
     return true;
  return false;
  }

//thin_L8-------------------------------------------------------------------------------
bool CCartoonImage::thin_L8(int x, int y)
  { //returns true if the pixel were thinned
  //it is assumed that there is an edge-point in (x, y) if this function were called
  if(!edgeImage[x-1][y-1] && !edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&      //0 0 0
```

```
      edgeImage[x-1][y+1] && edgeImage[x][y+1] && edgeImage[x+1][y+1])          //* 1 *
    edgeImage[x][y] = false;                                                    //1 1 1

  else if(!edgeImage[x][y-1] && !edgeImage[x+1][y-1] && edgeImage[x-1][y] &&    //* 0 0
          !edgeImage[x+1][y] && edgeImage[x][y+1])                            //1 1 0
    edgeImage[x][y] = false;                                                    //* 1 *

  else if(edgeImage[x-1][y-1] && !edgeImage[x+1][y-1] && edgeImage[x-1][y] &&   //1 * 0
          !edgeImage[x+1][y] && edgeImage[x-1][y+1] && !edgeImage[x+1][y+1])    //1 1 0
    edgeImage[x][y] = false;                                                    //1 * 0

  else if(edgeImage[x][y-1] && edgeImage[x-1][y] && !edgeImage[x+1][y] &&       //* 1 *
          !edgeImage[x][y+1] && !edgeImage[x+1][y+1])                         //1 1 0
    edgeImage[x][y] = false;                                                    //* 0 0

  else if(edgeImage[x-1][y-1] && edgeImage[x][y-1] && edgeImage[x+1][y-1] &&    //1 1 1
          !edgeImage[x-1][y+1] && !edgeImage[x][y+1] && !edgeImage[x+1][y+1])   //* 1 *
    edgeImage[x][y] = false;                                                    //0 0 0

  else if(edgeImage[x][y-1] && !edgeImage[x-1][y] && edgeImage[x+1][y] &&       //* 1 *
          !edgeImage[x-1][y+1] && !edgeImage[x][y+1])                         //0 1 1
    edgeImage[x][y] = false;                                                    //0 0 *

  else if(!edgeImage[x-1][y-1] && edgeImage[x+1][y-1] && !edgeImage[x-1][y] &&  //0 * 1
          edgeImage[x+1][y] && !edgeImage[x-1][y+1] && edgeImage[x+1][y+1])     //0 1 1
    edgeImage[x][y] = false;                                                    //0 * 1

  else if(!edgeImage[x-1][y-1] && !edgeImage[x][y-1] && !edgeImage[x-1][y] &&   //0 0 *
          edgeImage[x+1][y] && edgeImage[x][y+1])                             //0 1 1
    edgeImage[x][y] = false;                                                    //* 1 *

  else if(!edgeImage[x-1][y-1] && !edgeImage[x][y-1] && !edgeImage[x+1][y-1] &&   //0 0 0
          !edgeImage[x-1][y]   &&                       !edgeImage[x+1][y]   &&   //0 1 0
          !edgeImage[x-1][y+1] && !edgeImage[x][y+1] && !edgeImage[x+1][y+1])     //0 0 0
//0 0 0
    edgeImage[x][y] = false;

  else
    return false;   //edge-point were not thinned
  return true;      //if execution were in one of the if/else if there were a thinning
  }

//setELSColors-------------------------------------------------------------------------------
-
void CCartoonImage::setELSColors(CEdgeLine* el)
  {
  short dX, dY;
  int halfX, halfY;               //point on the middel of the line
  unsigned char colSamDis = 4;    //color sample distance perpendicular from edge-line
  UPixel pix;

  pix.r = 255;
  pix.g = 0;
  pix.b = 0;

  dX = el->xEnd - el->xStart;               //calc. derivatives of the line
  dY = el->yEnd - el->yStart;

  halfX = el->xStart + dX;   //calc. coordinates for middle point
  halfY = el->yStart + dY;

  if(abs(dX) > abs(dY))     //if edge-line is mostly horizontal...
    {
    if(el->xEnd > el->xStart)     //edge-line going to the right
      {                                                             //s1
      el->s1Color.setValue(image[halfX][halfY-colSamDis].getValue());   //-->
      el->s2Color.setValue(image[halfX][halfY+colSamDis].getValue());   //s2
      }
    else                          //edge-line going to the left
      {                                                             //s2
      el->s1Color.setValue(image[halfX][halfY+colSamDis].getValue());   //<--
      el->s2Color.setValue(image[halfX][halfY-colSamDis].getValue());   //s1
      }
    }
  else                      //if edge-line is mostly vertical...
    {
```

```
    if(el->yEnd < el->yStart)     //edge-line going up
      {                                                                    //  ^
      el->s1Color.setValue(image[halfX-colSamDis][halfY].getValue());   //s1|s2
      el->s2Color.setValue(image[halfX+colSamDis][halfY].getValue());   //  |
      }
    else                          //edge-line going down
      {                                                                    //  |
      el->s1Color.setValue(image[halfX+colSamDis][halfY].getValue());   //s2|s1
      el->s2Color.setValue(image[halfX-colSamDis][halfY].getValue());   //  v
      }
    }
  }

//unvisitLine-----------------------------------------------------------------------------
void CCartoonImage::unvisitEdgeLine(CEdgeLine el)
  {
  float testA, testB;
  float dX, dY;
  int x, y;

  dX = (float)el.xEnd - (float)el.xStart;
  dY = (float)el.yEnd - (float)el.yStart;

  //unvisit all edge-points in the edge-line
  if(el.getAppLenght() == 1 || (el.yEnd - el.yStart) == 0)        //if line only consists
of one edge-point..
    {
    visited[el.xStart][el.yStart] = false;    //unvisit it
    }
  else if(absf(dX) > absf(dY))     //if edge-line is mostly horizontal... step along x-
axis
    {
    testA = dY / dX;
    testB = el.yStart - testA * el.xStart;

    if(el.xEnd > el.xStart)                              //step direction: xStart-xEnd
      for(x = el.xStart; x <= el.xEnd; x++)
        visited[x][(int)(testA * x + testB)] = false;   //unvisit point
    else                                                 //step direction: xEnd-xStart
      for(x = el.xEnd; x <= el.xStart; x++)
        visited[x][(int)(testA * x + testB)] = false;
    }
  else                                  //if edge-line is mostly vertical... step along y-
axis
    {
    testA = dX / dY;
    testB = el.xStart - testA * el.yStart;

    if(el.yEnd > el.yStart)                              //step direction: yStart-yEnd
      for(y = el.yStart; y <= el.yEnd; y++)
        visited[(int)(testA * y + testB)][y] = false;
    else                                                 //step direction: yEnd-yStart
      for(y = el.yEnd; y <= el.yStart; y++)
        visited[(int)(testA * y + testB)][y] = false;
    }
  }

//improveEdges----------------------------------------------------------------------------
void CCartoonImage::improveEdges()
  {
  register int x, y, i;
  bool thinned = true;
  int runs = 0;
  CStopWatch sw;

  printf("Thinning edges using L8-pattern...");
  sw.start();

  while(thinned)   //as long as the edges can be thinned more..
    {
    thinned = false;    //only run once more if this loop improves the edges
    runs++;
    for(y = 0; y < height; y++)       //loop edge-image point by point
      for(x = 0; x < width; x++)
        if(edgeImage[x][y])                     //if edge-point found...
          if(thin_L8(x, y)) thinned = true;;    //...thin if nessesary
```

```
    }

  printf("\tDone! in %f s using %u runs\n", sw.stop(), runs);
  }

//improveEdgeLines-------------------------------------------------------------------------
void CCartoonImage::improveEdgeLines()
  {
  int s;                    //actual edge-line to stretch
  int d;                    //edge-line trying to reach with a stretch
  bool stretched;           //flag marking if actual edge-line has been stretched
  int maxReachDist = 12;    //max. space between edge-lines to stretch
  int reachDist;            //actual reach dist. for stretching edge-lines

  //fill spaces between edge-lines by stretching
  for(s = 0; s < totalEdgeLines; s++)     //for each edge-line...
    {
    stretched = false;  //init that line is not stretched yet
    d = 0;
    reachDist = 2;      //start by finding already connected edge-lines
    while(!stretched && (reachDist <= maxReachDist))
      {      //first try to reach short then longer and longer...
      while(!stretched && (d < totalEdgeLines))   //try to stretch actual edge-line..
        {   //..until it has been stretched or all other edge-lines has been tested for
reachability
        if((abs(edgeLine[s].xEnd - edgeLine[d].xStart) <= reachDist) &&
           (abs(edgeLine[s].yEnd - edgeLine[d].yStart) <= reachDist) && (s != d))
          {
          edgeLine[s].xEnd = edgeLine[d].xStart;  //stretch edge-line to start point
          edgeLine[s].yEnd = edgeLine[d].yStart;  //of the other edge-line
          stretched = true;                    //mark that edge-line has been stretched
          }
        d++;      //try next edge-line
        }
      reachDist = reachDist + 4;    //try to reach longer if nessesary
      }
    }
  }

//edgeToBorder-----------------------------------------------------------------------------
void CCartoonImage::edgeToBorder(int x, int y)
  {
  int borderDist = 5;    //max. border distance for edge expantion
  int i;

  if(x < borderDist)                //left border
    for(i = 0; i < x; i++)
      edgeImage[i][y] = true;
  else if(x > width - borderDist)   //rigth border
    for(i = x; i < width; i++)
      edgeImage[i][y] = true;
  else if(y < borderDist)           //top border
    for(i = 0; i < y; i++)
      edgeImage[x][i] = true;
  else if(y > height - borderDist)  //bottom border
    for(i = y; i < height; i++)
      edgeImage[x][i] = true;
  }

//initEdgeImage----------------------------------------------------------------------------
void CCartoonImage::initEdgeImage()
  {
  int x, y;
  for(x = 0; x < width; x++)            //draw edge-point border frame
    {
    edgeImage[x][1] = true;             //top border
    visited[x][1] = false;
    edgeImage[x][height -  2] = true;   //bottom border
    visited[x][height -  2] = false;
    }
  for(y = 0; y < height; y++)
    {
    edgeImage[1][y] = true;             //left border
    visited[1][y] = false;
    edgeImage[width - 2][y] = true;     //bottom border
    visited[width - 2][y] = false;
```

```
    }
  }

//generatePolygons------------------------------------------------------------------------
void CCartoonImage::generatePolygons()
  {
  int polygonNum;           //polygon number
  int maxPolygons = 100;    //max. number of polygons
  int reachDist;            //reach distance when trying to add edge-lines
  int reachDistInc;         //reach distance incrementation
  int maxReachDist = 32;    //max. reach distance
  int edgeLineNum;          //edge-line number
  CEdgeLine el;             //temp. test edge-line
  UPixel pix;               //temp. test pix
  CStopWatch sw;

  //counter til addded polygons!!!
  printf("Generating polygons...");
  sw.start();

  for(polygonNum = 0; polygonNum < maxPolygons; polygonNum++)
    {
    reachDist = 1;
    reachDistInc = 1;
    while(reachDist <= maxReachDist)
      {
      for(edgeLineNum = 0; edgeLineNum < totalEdgeLines; edgeLineNum++)
        if(!edgeLine[edgeLineNum].s1Incl || !edgeLine[edgeLineNum].s2Incl)
          polygon[polygonNum].addEdge(&edgeLine[edgeLineNum], reachDist);

      reachDist = reachDist + reachDistInc;         //inc. reachDist by 2^n
      reachDistInc = reachDistInc + reachDistInc;
      }
    }

  //test polygon
  /*polygon[0].reset();

  el.s1Incl = true;
  el.s2Incl = false;
  el.set(50, 50, 200, 50);
  el.s1Color.setValue(220);
  el.s2Color.setValue(160);
  polygon[0].addEdge(&el, 10);

  el.s1Incl = false;
  el.s2Incl = false;
  el.set(205, 55, 205, 150);
  el.s1Color.setValue(220);
  el.s2Color.setValue(160);
  polygon[0].addEdge(&el, 10);

  el.s1Incl = false;
  el.s2Incl = false;
  el.set(205, 150, 180, 150);
  el.s1Color.setValue(220);
  el.s2Color.setValue(160);
  polygon[0].addEdge(&el, 10);

  el.s1Incl = false;
  el.s2Incl = false;
  el.set(180, 80, 180, 150);
  el.s1Color.setValue(160);
  el.s2Color.setValue(220);
  polygon[0].addEdge(&el, 10);

  el.s1Incl = false;
  el.s2Incl = false;
  el.set(180, 80, 80, 80);
  el.s1Color.setValue(220);
  el.s2Color.setValue(160);
  polygon[0].addEdge(&el, 10);

  el.s1Incl = false;                //her
  el.s2Incl = false;
  el.set(80, 150, 80, 85);
```

```
  el.s1Color.setValue(160);
  el.s2Color.setValue(220);
  polygon[0].addEdge(&el, 10);

  el.s1Incl = false;
  el.s2Incl = false;
  el.set(80, 150, 45, 150);
  el.s1Color.setValue(220);
  el.s2Color.setValue(160);
  polygon[0].addEdge(&el, 10);

  el.s1Incl = false;
  el.s2Incl = false;
  el.set(45, 150, 45, 55);
  el.s1Color.setValue(220);
  el.s2Color.setValue(160);
  polygon[0].addEdge(&el, 10);

  el.s1Incl = false;               //dummy edge
  el.s2Incl = false;
  el.set(200, 200, 250, 255);
  el.s1Color.setValue(220);
  el.s2Color.setValue(160);
  polygon[0].addEdge(&el, 10);
  */

  totalPolygons = maxPolygons;    //temp. always 100 polygons
  printf("\t\t\tDone! in %f s making %u polygons\n", sw.stop(), totalPolygons);
  }

//findPortPolygon-------------------------------------------------------------------------
void CCartoonImage::findPortPolygon()
  {
  int x, y;
  int px, py;
  int pl = 10, ps = 2;
  CStopWatch sw;

  printf("Finding port polygon...");
  sw.start();

  //reset visited image
  for(x = 0; x < width; x++)
    for(y = 0; y < height; y++)
      visited[x][y] = false;

  //scan from bottom to top, left - right
  for(y = height - 2; y >= pl; y--)
    for(x = 0; x < width-ps-1; x++)
      if(!visited[x][y] && image[x][y].isYellow() &&
         image[x+1][y].isYellow() && image[x+1][y-1].isYellow() && image[x+1][y-
2].isYellow() &&
         image[x+1][y-3].isYellow() && image[x+1][y-4].isYellow() && image[x+1][y-
5].isYellow() && image[x+1][y-6].isYellow() && image[x+1][y-7].isYellow() && image[x+1][y-
8].isYellow() && image[x+1][y-9].isYellow() &&
         image[x+2][y].isYellow() && image[x+2][y-1].isYellow() && image[x+2][y-
2].isYellow() &&
         image[x+2][y-3].isYellow() && image[x+2][y-4].isYellow() && image[x+2][y-
5].isYellow() && image[x+2][y-6].isYellow() && image[x+2][y-7].isYellow() && image[x+2][y-
8].isYellow() && image[x+2][y-9].isYellow())
        {   //if yellow pixel reached..
        portPolygon.lineTo(x, y);
        visited[x][y] = true;
        break;
        }

  //scan from left to right, top - down
  for(x = 1; x < width-pl; x++)
    for(y = 0; y < height-ps; y++)
      if(!visited[x][y] && image[x][y].isYellow() &&
         image[x][y+1].isYellow() && image[x+1][y+1].isYellow() &&
image[x+2][y+1].isYellow() &&
         image[x+3][y+1].isYellow() && image[x+4][y+1].isYellow() &&
image[x+5][y+1].isYellow() && image[x+6][y+1].isYellow() && image[x+7][y+1].isYellow() &&
image[x+8][y+1].isYellow() && image[x+9][y+1].isYellow() &&
```

```
         image[x][y+2].isYellow() && image[x+1][y+2].isYellow() &&
image[x+2][y+2].isYellow() &&
         image[x+3][y+2].isYellow() && image[x+4][y+2].isYellow() &&
image[x+5][y+2].isYellow() && image[x+6][y+2].isYellow() && image[x+7][y+2].isYellow() &&
image[x+8][y+2].isYellow() && image[x+9][y+2].isYellow())
        {  //if yellow pixel reached..
        portPolygon.lineTo(x, y);
        visited[x][y] = true;
        break;
        }

  //scan from top to bottom, right - left
  for(y = 1; y < height-pl-1; y++)
    for(x = width - 1; x > ps; x--)
      if(!visited[x][y] && image[x][y].isYellow() &&
         image[x-1][y].isYellow() && image[x-1][y+1].isYellow() && image[x-
1][y+2].isYellow() &&
         image[x-1][y+3].isYellow() && image[x-1][y+4].isYellow() && image[x-
1][y+5].isYellow() && image[x-1][y+6].isYellow() && image[x-1][y+7].isYellow() && image[x-
1][y+8].isYellow() && image[x-1][y+9].isYellow() &&
         image[x-2][y].isYellow() && image[x-2][y+1].isYellow() && image[x-
2][y+2].isYellow() &&
         image[x-2][y+3].isYellow() && image[x-2][y+4].isYellow() && image[x-
2][y+5].isYellow() && image[x-2][y+6].isYellow() && image[x-2][y+7].isYellow() && image[x-
2][y+8].isYellow() && image[x-2][y+9].isYellow())
        {  //if yellow pixel reached..
        portPolygon.lineTo(x, y);
        visited[x][y] = true;
        break;
        }

  //scan from right to left, bottom - up
  for(x = width - 2; x >= pl; x--)
    for(y = height - 1; y >= ps; y--)
      if(!visited[x][y] && image[x][y].isYellow() &&
         image[x][y-1].isYellow() && image[x-1][y-1].isYellow() && image[x-2][y-
1].isYellow() &&
         image[x-3][y-1].isYellow() && image[x-4][y-1].isYellow() && image[x-5][y-
1].isYellow() && image[x-6][y-1].isYellow() && image[x-7][y-1].isYellow() && image[x-8][y-
1].isYellow() && image[x-9][y-1].isYellow() &&
         image[x][y-2].isYellow() && image[x-1][y-2].isYellow() && image[x-2][y-
2].isYellow() &&
         image[x-3][y-2].isYellow() && image[x-4][y-2].isYellow() && image[x-5][y-
2].isYellow() && image[x-6][y-2].isYellow() && image[x-7][y-2].isYellow() && image[x-8][y-
2].isYellow() && image[x-9][y-2].isYellow())
        {  //if yellow pixel reached..
        portPolygon.lineTo(x, y);
        visited[x][y] = true;
        break;
        }

  printf("\t\t\tDone! in %f s using %u edge-lines\n", sw.stop(),
portPolygon.getNoOfEdges());
  }

//getPortCenter-------------------------------------------------------------------------
void CCartoonImage::getPortCenter(int* centerX, int* centerY)
  {
  portPolygon.getCenter(centerX, centerY);
   }
```

Appendix A3 – CEdgeLine

```
/***************************************************************************
    cedgeline.h  -  class representing linear edge part of a CCarttonImage

    begin                : Wed Jan 28 2004
    copyright            : (C) 2004 by Allan Krogh Jensen
    email                : s973989@student.dtu.dk
 ***************************************************************************/

#ifndef CEDGELINE_H
#define CEDGELINE_H
```

```
#include "urawimage.h"
#include "ccartoonpixel.h"
//#include "ccartoonimage.h"

/**
  *@author Allan Krogh Jensen
  */

/** class representing linear edge part of a CCarttonImage */
class CEdgeLine
  {
  private:
    /**previous end pixel-coordinates in the image */
    int xPreEnd, yPreEnd;

    /**returns true if the two points are nabours */
    bool isNabours(unsigned int x1, unsigned int y1, unsigned int x2, unsigned int y2);

    /**returns true only if the edge-line is on the image */
    bool isValid(UImage* img);

  public:
    /**start pixel-coordinates in the image */
    int xStart, yStart;

    /**end pixel-coordinates in the image */
    int xEnd, yEnd;

    /**colors on each side of the edge-line */
    CCartoonPixel s1Color, s2Color;

    /** flag that is true if the side has been included in a polygon */
    unsigned char s1Incl, s2Incl;

    /**default constructor */
    CEdgeLine();

    /** default destructor */
    ~CEdgeLine();

    /** set start and end points directly */
    void set(int xS, int yS, int xE, int yE);

    /** draw the edge line on specified image, with side color spots if wanted */
    void draw(UImage* img, UPixel ec, bool withSpots, bool withDir);

    /** draw the edge line on specified image, in specified color */
    void draw(UImage* img, CCartoonPixel col);

    /** checks if an edge point can be accepted as a part of line */
    bool addPoint(int x, int y);

    /** reset edge-line, unvisit all edge-points in the line */
    void reset();

    /** returns an approximate length of the edge-line */
    int getAppLenght();

    /** get twin edge-line */
    CEdgeLine getTwin(unsigned char side);

    /** copy operator for edge-lines */
    CEdgeLine operator= (CEdgeLine s);

    /** swap direction of the edge-line, swap start and end points */
    void swapDirection();

  };

#endif

/***************************************************************************
    cedgeline.cpp  -  class representing linear edge part of a CCarttonImage

    begin                : Wed Jan 28 2004
```

```
    copyright            : (C) 2004 by Allan Krogh Jensen
    email                : s973989@student.dtu.dk
 ***************************************************************************/

#include "cedgeline.h"

//default constructor-------------------------------------------------------
CEdgeLine::CEdgeLine()
  {
  reset();
  }

//default destructor--------------------------------------------------------
CEdgeLine::~CEdgeLine()
  {
  }

//isNabours----------------------------------------------------------------
bool CEdgeLine::isNabours(unsigned int x1, unsigned int y1, unsigned int x2, unsigned int
y2)
  {     //returns true if the two points (x1, y1) and (y1, y2) are nabours
  return ((abs(x2 - x1) <= 1) && (abs(y2 - y1) <= 1));
  }

//set----------------------------------------------------------------------
void CEdgeLine::set(int xS, int yS, int xE, int yE)
  {
  xStart = xS;
  yStart = yS;
  xEnd = xE;
  yEnd = yE;
  }

//draw---------------------------------------------------------------------
void CEdgeLine::draw(UImage* img, UPixel ec, bool withSpots, bool withDir)
  {
  UPixel pix;
  int dX, dY;
  float halfX, halfY;    //point on the middel of the line
  int spotDist = 4;      //distance from line to side color spots
  int xS1Spot, yS1Spot, xS2Spot, yS2Spot;    //coor. of paint spots for side colors

  img->PaintLine((int)xStart, (int)yStart, (int)xEnd, (int)yEnd, &ec);

  pix.r = 0;            //color of edgeline endpoints (blue)
  pix.g = 0;
  pix.b = 255;
  img->SetPix(yStart, xStart, pix);       //draw end points
  img->SetPix(yEnd, xEnd, pix);

  if(withSpots)         //if side color spots should also be drawn
    {
    dX = xEnd - xStart;         //calc. derivatives of the line
    dY = yEnd - yStart;
    halfX = xStart + (dX / 2);        //calc. coordinates for middle point
    halfY = yStart + (dY / 2);

    if(absf(dX) > absf(dY))     //if edge-line is mostly horizontal...
      {
      if(xEnd > xStart)     //edge-line going to the right
        {                               //s1
        yS1Spot = (int)(halfY - spotDist);   //-->
        yS2Spot = (int)(halfY + spotDist);   //s2
        }
      else                          //edge-line going to the left
        {                             //s2
        yS1Spot = (int)(halfY + spotDist);   //<--
        yS2Spot = (int)(halfY - spotDist);   //s1
        }
      xS1Spot = (int)halfX;
      xS2Spot = (int)halfX;
      }
    else                      //if edge-line is mostly vertical...
      {
      if(yEnd < yStart)     //edge-line going up
        {                                   //  ^
```

```
        xS1Spot = (int)(halfX - spotDist);   //s1|s2
        xS2Spot = (int)(halfX + spotDist);   //  |
        }
      else                          //edge-line going down
        {                                   //  |
        xS1Spot = (int)(halfX + spotDist);   //s2|s1
        xS2Spot = (int)(halfX - spotDist);   //  v
        }
      yS1Spot = (int)halfY;
      yS2Spot = (int)halfY;
      }

    pix.r = 255;            //color of spot border (red)
    pix.g = 0;
    pix.b = 0;

    img->PaintFilledRect(xS1Spot-2, yS1Spot-2, xS1Spot+1, yS1Spot+1, &pix);
    img->SetPix(yS1Spot, xS1Spot, s1Color.getRGB());   //paint side 1 color in a spot
    img->SetPix(yS1Spot, xS1Spot-1, s1Color.getRGB());
    img->SetPix(yS1Spot-1, xS1Spot, s1Color.getRGB());
    img->SetPix(yS1Spot-1, xS1Spot-1, s1Color.getRGB());

    img->PaintFilledRect(xS2Spot-2, yS2Spot-2, xS2Spot+1, yS2Spot+1, &pix);
    img->SetPix(yS2Spot, xS2Spot, s2Color.getRGB());   //paint side 2 color in a spot
    img->SetPix(yS2Spot, xS2Spot-1, s2Color.getRGB());
    img->SetPix(yS2Spot-1, xS2Spot, s2Color.getRGB());
    img->SetPix(yS2Spot-1, xS2Spot-1, s2Color.getRGB());
    }

  if(withDir)     //if direction arrows should be drawn
    {
    dX = xEnd - xStart;         //calc. derivatives of the line
    dY = yEnd - yStart;
    halfX = (int)(xStart + (3.0 / 4.0) * dX);        //calc. coordinates for arrow point
    halfY = (int)(yStart + (3.0 / 4.0) * dY);
    pix.r = 0;            //color of edge-line arrow (green)
    pix.g = 255;
    pix.b = 0;

    if(absf(dX) > absf(dY))     //if edge-line is mostly horizontal...
      img->PaintLine(halfX, halfY - 2, halfX, halfY + 2, &pix);
    else
      img->PaintLine(halfX + 2, halfY, halfX + 2, halfY, &pix);
    }
  }

//addPoint--------------------------------------------------------------------------
bool CEdgeLine::addPoint(int x, int y)
  {
  float dX, dY;
  float testA, testB;
  float t = 1.0;        //precision tolerance
  float halfX, halfY;   //coordinate of the half-way point of the line

  if(xStart == 0 && yStart == 0 && xEnd == 0 && yEnd == 0)    //if first point on line
    {
    xStart = xEnd = xPreEnd = x;      //edge-line starts and ends in the same point
    yStart = yEnd = yPreEnd = y;
    return true;
    }

  if(isNabours(xEnd, yEnd, x, y))    //if point is nabour to end point of line
    {
    dX = (float)x - (float)xStart;
    dY = (float)y - (float)yStart;
    halfX = xStart + (dX / 2);        //calc. coordinates for middle point
    halfY = yStart + (dY / 2);

    if(absf(dX) > absf(dY))     //if edge-line is mostly horizontal... step along x-axis
      {
      testA = dY / dX;
      testB = y - testA * x;
      if((yEnd >= (int)(testA * xEnd + testB - t)) && (yEnd <= (int)(testA * xEnd + testB
+ t))
        && (yPreEnd >= (int)(testA * xPreEnd + testB - t)) && (yPreEnd <= (int)(testA *
xPreEnd + testB + t))
```

```
        && (halfY >= (int)(testA * halfX + testB - t)) && (halfY <= (int)(testA * halfX +
testB + t)))
        {
        xPreEnd = xEnd;
        yPreEnd = yEnd;
        xEnd = x;
        yEnd = y;
        return true;
        }
      else
        return false;
      }
    else                        //if edge-line is mostly vertical... step along y-axis
      {
      testA = dX / dY;
      testB = x - testA * y;
      if((xEnd >= (int)(testA * yEnd + testB - t)) && (xEnd <= (int)(testA * yEnd + testB
+ t))
        && (xPreEnd >= (int)(testA * yPreEnd + testB - t)) && (xPreEnd <= (int)(testA *
yPreEnd + testB + t))
        && (halfX >= (int)(testA * halfY + testB - t)) && (halfX <= (int)(testA * halfY +
testB + t)))
        {
        xPreEnd = xEnd;
        yPreEnd = yEnd;
        xEnd = x;
        yEnd = y;
        return true;
        }
      else
        return false;
      }
    }
  else
    return false;
  }

//reset-------------------------------------------------------------------------
void CEdgeLine::reset()
  {
  xStart = yStart = 0;
  xEnd = yEnd = 0;
  xPreEnd = yPreEnd = 0;
  s1Color.setValue(0);
  s2Color.setValue(0);
  s1Incl = s2Incl = false;
  }

//isValid-----------------------------------------------------------------------
bool CEdgeLine::isValid(UImage* img)
  {
  return ((yEnd < (int)img->height) && (xEnd < (int)img->width));
  }

//getAppLenght------------------------------------------------------------------
int CEdgeLine::getAppLenght()
  {
  return abs(xEnd - xStart) + abs(yEnd - yStart) - 1;    //return approx. lenght
  }

//operator= ---------------------------------------------------------------------
CEdgeLine CEdgeLine::operator= (CEdgeLine s)
  {
  this->xStart = s.xStart;
  this->yStart = s.yStart;
  this->xEnd = s.xEnd;
  this->yEnd = s.yEnd;
  this->s1Color.setValue(s.s1Color.getValue());
  this->s2Color.setValue(s.s2Color.getValue());
  return *this;
  }

//getTwin-----------------------------------------------------------------------
CEdgeLine CEdgeLine::getTwin(unsigned char side)
  {          //OBS! der er stadig bugs i denne funktion, ved parallel forskydning nedad
  float testA, testB;
```

```
  float dX, dY;
  CEdgeLine retEL;
  int newXStart, newYStart, newXEnd, newYEnd;    //start and end-points of the twin line

  dX = (float)xEnd - (float)xStart;
  dY = (float)yEnd - (float)yStart;

  if(absf(dX) > absf(dY))     //if edge-line is mostly horizontal... step along x-axis
    {
    testA = dY / dX;
    testB = yStart - testA * xStart;
    if(side == 1)             //parallel forskyd opad
      testB--;
    else                      //parallel forskyd nedad
      testB++;
    newYStart = (int)(testA * xStart + testB);
    newYEnd = (int)(testA * xEnd + testB);
    retEL.set(xStart, newYStart, xEnd, newYEnd);
    }
  else                        //if edge-line is mostly vertical... step along y-axis
    {
    testA = dX / dY;
    testB = xStart - testA * yStart;
    if(side == 1)             //parallel forskyd til venstre
      testB--;
    else                      //parallel forskyd til right
      testB++;
    newXStart = (int)(testA * yStart + testB);
    newXEnd = (int)(testA * yEnd + testB);
    retEL.set(newXStart, yStart, newXEnd, yEnd);
    }
  return retEL;               //return the twin (parallel forskudte) edge-line
  }

//draw specified color-------------------------------------------------------------------------
void CEdgeLine::draw(UImage* img, CCartoonPixel col)
  {
  UPixel pix;
  pix = col.getRGB();
  img->PaintLine((int)xStart, (int)yStart, (int)xEnd, (int)yEnd, &pix);
  }

//swapDirection--------------------------------------------------------------------------------
void CEdgeLine::swapDirection()
  {
  unsigned int tempX, tempY;    //temp. coordinate
  CCartoonPixel col;            //temp. side color
  bool incl;

  tempX = xStart;
  tempY = yStart;
  xStart = xEnd;
  yStart = yEnd;
  xEnd = tempX;
  yEnd = tempY;

  col = s1Color;                //the sides are also swapped
  s1Color = s2Color;
  s2Color = col;

  incl = s1Incl;
  s1Incl = s2Incl;
  s2Incl = incl;
  }
```

Appendix A4 – C2DPolygon

```
/***************************************************************************
    c2dpolygon.h  -  class representing a 2D polygon in an image

    begin                : Thu Mar 4 2004
    copyright            : (C) 2004 by Allan Krogh Jensen
    email                : s973989@student.dtu.dk
```

```
 ***************************************************************************/

#ifndef C2DPOLYGON_H
#define C2DPOLYGON_H

#include "cedgeline.h"

/**
  *@author Allan Krogh Jensen
  */

class C2DPolygon
  {
  private:
    /** the edge-lines that together forms the polygon */
    CEdgeLine edges[3000];

    /** the innner sides of each edge in the polygon (1 or 2) */
    unsigned char innerSide[3000];

    /** color histogram of all edge-lines in the polygon, using their inner side colors */
    unsigned char colorHistogram[256];

    /** number of edges in the polygon */
    unsigned int noOfEdges;

    /** flag marking if all of the polygon is visible in the image */
    bool allVisible;

    /** extreme corner points of the polygon */
    unsigned int xTopLeftMost, yTopLeftMost;
    unsigned int xTopRightMost, yTopRightMost;
    unsigned int xBottomLeftMost, yBottomLeftMost;
    unsigned int xBottomRightMost, yBottomRightMost;

    unsigned int xBottomCenter, yBottomCenter;
    unsigned int xCenter, yCenter;

    /** update color of polygon, dependent of the most represented color */
    void updateColor();

    /** returns min of a and b */
    int min(int a, int b);

  public:
    /** flag true if polygon is closed, no more edge-lines can be added */
    bool isClosed;

    /** default constructor */
    C2DPolygon();

    /** default destructor */
    ~C2DPolygon();

     /** checks if the polygon is 100% visible on the image and flags allVisible if so */
    bool checkVisibility();

    /** add an new edge to the polygon if possible*/
    bool addEdge(CEdgeLine* newEdge, int reachDist);

    /** paint the polygons edges and fill it with its solid color */
    void paint(UImage* img);

    /** reset all data in polygon */
    void reset();

    /** generates the polygon line by line, like drawing from one point to another */
    void lineTo(int desX, int desY);

    /** get number of edges */
    int getNoOfEdges();

    /** draw extreme corners */
    void drawCorners(UImage* img);

    /** find extreme corner points */
```

```
    void findExtremeCorners();

    /** get coordinates of the center of gravity */
    void getCenter(int* pX, int* pY);

    /** returns true if polygon is a RoboCup port */
    bool isPort();

     /** solid color of the polygon */
    CCartoonPixel color;


  };

#endif

/***************************************************************************
    c2dpolygon.h  -  class representing a 2D polygon in an image

    begin                : Thu Mar 4 2004
    copyright            : (C) 2004 by Allan Krogh Jensen
    email                : s973989@student.dtu.dk
 ***************************************************************************/

#include "c2dpolygon.h"

//default constructor--------------------------------------------------------
C2DPolygon::C2DPolygon()
  {
  reset();
  }

//default destructor---------------------------------------------------------
C2DPolygon::~C2DPolygon()
  {
  }

//min------------------------------------------------------------------------
int C2DPolygon::min(int a, int b)
  {
  if(a < b)
    return a;
  else
    return b;
  }

//addEdge--------------------------------------------------------------------
bool C2DPolygon::addEdge(CEdgeLine* newEdge, int reachDist)
  {
  int t = 16;     //thresshold color matching value

  if(isClosed)    //if polygon is closed....no more edge-lines can be added
    return false;

  if(noOfEdges == 0)      //if first edge in the polygon...
    {
    if(!newEdge->s1Incl)      //if side 1 of edge-line is not already in another polygon
      {
      innerSide[0] = 1;         //use side 1 in this new polygon
      newEdge->s1Incl = true;
      }
    else if(!newEdge->s2Incl) //else if side 2 of edge-line is not already in another
polygon
      {
      innerSide[0] = 2;         //use side 1 in this new polygon
      newEdge->s2Incl = true;
      }
    else                      //if all sides of the edge-line were already in other
polygons..
      return false;           //..refuse to add it to this

    edges[0] = *newEdge;
    isClosed = false;
    noOfEdges = 1;
    updateColor();
    return true;
```

```
    }

  //first check if polygon can reach its own tail (start-point) from its end-point
  if(abs(edges[0].xStart - edges[noOfEdges - 1].xEnd) <= reachDist &&
     abs(edges[0].yStart - edges[noOfEdges - 1].yEnd) <= reachDist && noOfEdges >= 3)
    {        //polygon must at least be a triangle
    edges[noOfEdges - 1].xEnd = edges[0].xStart;
    edges[noOfEdges - 1].yEnd = edges[0].yStart;
    isClosed = true;
    checkVisibility();
    }

  //if start-point of the edge-line is reachable from the polygon end-point
  if(abs(newEdge->xStart - edges[noOfEdges - 1].xEnd) <= reachDist &&
     abs(newEdge->yStart - edges[noOfEdges - 1].yEnd) <= reachDist)
    {
    //if side 1 of edge matches polygon, and side is not already in a another polygon
    if((abs(color.getValue() - newEdge->s1Color.getValue()) <= t) &&
      !newEdge->s1Incl && (innerSide[noOfEdges - 1] == 1))
      {
      innerSide[noOfEdges] = 1;

      edges[noOfEdges - 1].xEnd = newEdge->xStart; //stretch last edge-line to the new
edge-line
      edges[noOfEdges - 1].yEnd = newEdge->yStart;
      newEdge->s1Incl = true;
      edges[noOfEdges] = *newEdge;                //insert edge-line

      noOfEdges++;
      updateColor();
      return true;
      }

    //if side 2 of edge matches polygon, and side is not already in a another polygon
    else if((abs(color.getValue() - newEdge->s2Color.getValue()) <= t) &&
      !newEdge->s2Incl && (innerSide[noOfEdges - 1] == 2))
      {
      innerSide[noOfEdges] = 2;

      edges[noOfEdges - 1].xEnd = newEdge->xStart;  //stretch last edge-line to the new
edge-line
      edges[noOfEdges - 1].yEnd = newEdge->yStart;
      newEdge->s2Incl = true;
      edges[noOfEdges] = *newEdge;                 //insert edge-line

      noOfEdges++;
      updateColor();
      return true;
      }

    else
      return false;     //...edge-line did not match the polygon color
    }

  //if end-point of the edge-line is reachable from the polygon end-point
  /*else if(abs(newEdge->xEnd - edges[noOfEdges - 1].xEnd) <= reachDist &&
    abs(newEdge->yEnd - edges[noOfEdges - 1].yEnd) <= reachDist)
    {

    //if side 1 of edge matches polygon, and side is not already in a another polygon
    if((abs(color.getValue() - newEdge->s1Color.getValue()) <= t) &&
      !newEdge->s1Incl && (innerSide[noOfEdges - 1] == 2))
      {
      edges[noOfEdges - 1].xEnd = newEdge->xEnd; //stretch last edge-line to the new edge-
line
      edges[noOfEdges - 1].yEnd = newEdge->yEnd;
      newEdge->s1Incl = true;
      edges[noOfEdges] = *newEdge;                //insert edge-line
      edges[noOfEdges].swapDirection();  //make end-point of edge-line be the last in
polygon
      innerSide[noOfEdges] = 2;

      noOfEdges++;
      updateColor();
      return true;
      }
```

```
    //if side 2 of edge matches polygon, and side is not already in a another polygon
    else if((abs(color.getValue() - newEdge->s2Color.getValue()) <= t) &&
      !newEdge->s2Incl && (innerSide[noOfEdges - 1] == 1))
      {
      edges[noOfEdges - 1].xEnd = newEdge->xEnd;  //stretch last edge-line to the new
edge-line
      edges[noOfEdges - 1].yEnd = newEdge->yEnd;
      newEdge->s2Incl = true;
      edges[noOfEdges] = *newEdge;                  //insert edge-line
      edges[noOfEdges].swapDirection();  //make end-point of edge-line be the last in
polygon
      innerSide[noOfEdges] = 1;

      noOfEdges++;
      updateColor();
      return true;
      }

    else
      return false;     //...edge-line did not match the polygon color
    }    */
  else
    return false;       //...edge-line were not reachable by end-point of polygon
  }

//paint-------------------------------------------------------------------------
void C2DPolygon::paint(UImage* img)
  {
  UPixel pix;
  unsigned short int edgeNo;

  pix.r = 0;
  pix.g = 255;
  pix.b = 0;

  for(edgeNo = 0; edgeNo < noOfEdges; edgeNo++)
    {
    edges[edgeNo].draw(img, pix, false, false);  //draw edge
    }
  }

//updateColor-------------------------------------------------------------------
void C2DPolygon::updateColor()
  {    //updates the polygon color by the most represented inner color of the edge-lines
  int i;

  //find the color of the inner side of each edge-line...
  if(innerSide[noOfEdges - 1] == 1)     //update the color histogram with s1 color
    {
    colorHistogram[edges[noOfEdges - 1].s1Color.getValue()]++;
    if(colorHistogram[edges[noOfEdges - 1].s1Color.getValue()] >
colorHistogram[color.getValue()])
      color.setValue(edges[noOfEdges - 1].s1Color.getValue());  //update max. if new found
    }
  else                                  //update the color histogram with s2 color
    {
    colorHistogram[edges[noOfEdges - 1].s2Color.getValue()]++;
    if(colorHistogram[edges[noOfEdges - 1].s2Color.getValue()] >
colorHistogram[color.getValue()])
      color.setValue(edges[noOfEdges - 1].s2Color.getValue());  //update max. if new found
    }
  }

//reset-------------------------------------------------------------------------
void C2DPolygon::reset()
  {
  int i;

  for(i = 0; i < noOfEdges; i++)
    {
    edges[i].reset();       //clear all edges in polygon
    innerSide[i] = 0;
    }

  noOfEdges = 0;
```

```
  isClosed = false;
  allVisible = false;

  for(i = 0; i < 256; i++)  //reset color histogram
    colorHistogram[i] = 0;
  }

//checkVisibility-------------------------------------------------------------------------
bool C2DPolygon::checkVisibility()
  {
  int edgeNo = 0;
  int imgH = 240;
  int imgW = 320;

  allVisible = true;    //polygon is assumed 100 % visible until other approved!
  while(allVisible)     //check if any of the corners reaches the border frame
    {
    if((edges[edgeNo].xStart <= 1) || (edges[edgeNo].xStart >= imgW - 1) ||
       (edges[edgeNo].yStart <= 1) || (edges[edgeNo].yStart >= imgH - 1))
       allVisible = false;
    edgeNo++;
    }
  return allVisible;
  }

//lineTo----------------------------------------------------------------------------------
void C2DPolygon::lineTo(int desX, int desY)
  {
  if(noOfEdges == 0)   //if no edge-line in the polygon
    {
    edges[0].set(desX, desY, desX, desY);
    noOfEdges = 1;
    }
  else                 //if polygon already have one or more edge-lines
    {
    edges[noOfEdges].set(edges[noOfEdges-1].xEnd, edges[noOfEdges-1].yEnd, desX, desY);
    noOfEdges++;
    }
  }

//getNoOfEdges----------------------------------------------------------------------------
int C2DPolygon::getNoOfEdges()
  {
  return noOfEdges;
  }

//findExtremeCorners----------------------------------------------------------------------
void C2DPolygon::findExtremeCorners()
  {
  unsigned int edgeNo;
  int dX, dY;
  int b;
  int bBottomLeftMost = 0;
  int bBottomRightMost = 240;
  int bTopLeftMost = 240;
  int bTopRightMost =  0;

  if(isClosed)
    {
    xTopLeftMost = 160;
    yTopLeftMost = 120;
    xTopRightMost = 160;
    yTopRightMost = 120;
    xBottomLeftMost = 160;
    yBottomLeftMost = 120;
    xBottomRightMost = 160;
    yBottomRightMost = 120;

    for(edgeNo = 0; edgeNo < noOfEdges; edgeNo++)
      {
      //top left-most corner
      b = edges[edgeNo].yEnd - (-1) * edges[edgeNo].xEnd;
      //if((edges[edgeNo].xEnd <= xTopLeftMost) && (edges[edgeNo].yEnd <= yTopLeftMost))
      if(b < bTopLeftMost)
        {
        bTopLeftMost = b;
```

```
        xTopLeftMost = edges[edgeNo].xEnd;
        yTopLeftMost = edges[edgeNo].yEnd;
        }

      //top right-most corner
      b = edges[edgeNo].yEnd - (1) * edges[edgeNo].xEnd;
      //if((edges[edgeNo].xEnd >= xTopRightMost) && (edges[edgeNo].yEnd <= yTopRightMost))
      if(b < bTopRightMost)
        {
        bTopRightMost = b;
        xTopRightMost = edges[edgeNo].xEnd;
        yTopRightMost = edges[edgeNo].yEnd;
        }

      //bottom left-most corner
      b = edges[edgeNo].yEnd - (1) * edges[edgeNo].xEnd;
      //if((edges[edgeNo].xEnd <= xBottomLeftMost) && (edges[edgeNo].yEnd >=
yBottomLeftMost))
      if(b > bBottomLeftMost)
        {
        bBottomLeftMost = b;
        xBottomLeftMost = edges[edgeNo].xEnd;
        yBottomLeftMost = edges[edgeNo].yEnd;
        }

      //bottom right-most corner
      b = edges[edgeNo].yEnd - (-1) * edges[edgeNo].xEnd;
      //if((edges[edgeNo].xEnd >= xBottomRightMost) && (edges[edgeNo].yEnd >=
yBottomRightMost))
      if(b > bBottomRightMost)
        {
        bBottomRightMost = b;
        xBottomRightMost = edges[edgeNo].xEnd;
        yBottomRightMost = edges[edgeNo].yEnd;
        }
      }

    //find bottom center
    xBottomCenter = xBottomLeftMost + ((xBottomRightMost - xBottomLeftMost) / 2);
//calc. coordinates for middle point
    yBottomCenter = min(yBottomLeftMost, yBottomRightMost) + abs(yBottomLeftMost -
yBottomRightMost) / 2;

    printf("BottomCenter: (%u, %u)\n", xBottomCenter, yBottomCenter);

    }
  }

//drawCorners-------------------------------------------------------------------------------
void C2DPolygon::drawCorners(UImage* img)
  {
  UPixel pix;
  int c = 2;    //cross dimensions

  pix.r = 255;
  pix.g = 0;
  pix.b = 0;

  img->PaintLine(xTopLeftMost-c, yTopLeftMost-c, xTopLeftMost+c, yTopLeftMost+c, &pix);
  img->PaintLine(xTopLeftMost-c, yTopLeftMost+c, xTopLeftMost+c, yTopLeftMost-c, &pix);

  img->PaintLine(xTopRightMost-c, yTopRightMost-c, xTopRightMost+c, yTopRightMost+c,
&pix);
  img->PaintLine(xTopRightMost-c, yTopRightMost+c, xTopRightMost+c, yTopRightMost-c,
&pix);

  img->PaintLine(xBottomLeftMost-c, yBottomLeftMost-c, xBottomLeftMost+c,
yBottomLeftMost+c, &pix);
  img->PaintLine(xBottomLeftMost-c, yBottomLeftMost+c, xBottomLeftMost+c, yBottomLeftMost-
c, &pix);

  img->PaintLine(xBottomRightMost-c, yBottomRightMost-c, xBottomRightMost+c,
yBottomRightMost+c, &pix);
  img->PaintLine(xBottomRightMost-c, yBottomRightMost+c, xBottomRightMost+c,
yBottomRightMost-c, &pix);
```

```
  //four-angle holding the port
  img->PaintLine(xTopLeftMost, yTopLeftMost, xTopRightMost, yTopRightMost, &pix);
  img->PaintLine(xTopRightMost, yTopRightMost, xBottomRightMost, yBottomRightMost, &pix);
  img->PaintLine(xBottomRightMost, yBottomRightMost, xBottomLeftMost, yBottomLeftMost,
&pix);
  img->PaintLine(xBottomLeftMost, yBottomLeftMost, xTopLeftMost, yTopLeftMost, &pix);

  //target point (bottom center of port)
  img->PaintLine(xBottomCenter, yBottomCenter-10, xBottomCenter, yBottomCenter+10, &pix);
  img->PaintLine(xBottomCenter-10, yBottomCenter, xBottomCenter+10, yBottomCenter, &pix);

  //center of gravity (used for focus point by webcam)
  img->PaintLine(xCenter, 0, xCenter, 239, &pix);
  img->PaintLine(xCenter-10, yCenter, xCenter+10, yCenter, &pix);
  }

//getCenter-------------------------------------------------------------------------------
void C2DPolygon::getCenter(int* pX, int* pY)
  {
  findExtremeCorners();

  xCenter = xBottomCenter;     //approximation of the center of gravity
  yCenter = min(yTopLeftMost, yTopRightMost) + ((yBottomLeftMost - yTopLeftMost) +
(yBottomRightMost - yTopRightMost)) / 4;
  //yCenter is the top most y value + half of the average vertical side lenght in the
four-angle

  *pX = xCenter;
  *pY = yCenter;
  }

//isPort----------------------------------------------------------------------------------
bool C2DPolygon::isPort()
  {
  return (noOfEdges > 400);
  }
```

Appendix A4 – CEye

```
/***************************************************************************
    ceye.h  -  class representing one eye, including webcam, pan/tilt
               and outer orientation. For use in a stereovision system

    begin                : Mon Jan 12 2004
    copyright            : (C) 2004 by Allan Krogh Jensen
    email                : s973989@student.dtu.dk
 ***************************************************************************/

#ifndef CEYE_H
#define CEYE_H

#include <string.h>
#include <stdio.h>
#include <linux/serial.h>
#include "ucam.h"
#include "crs232port.h"
#include "usmrcl.h"

#define S_0 0
#define S_1 1
#define S_2 2
#define S_3 3
#define S_4 4
#define S_5 5
#define S_6 6
#define S_7 7
#define S_8 8

/**   *@author Allan Krogh Jensen   */

class CEye : public URobCam
  {
  private:
```

```
    float panAngle;
    float tiltAngle;

    /** serial port used for communication with the rest of the robot */
    CRS232Port comPort;

    /** connection to the SMR */
    USmrCl smr;

  public:
    /**default constructor */
    CEye();

    /**default destructor */
    ~CEye();

    bool run(int device);

    void testCameraImage();

    /** sets the absolute orientation (pan and tilt angle) of the webcam
      returns true only if the webcam could be turned into desired pos.*/
    bool setOrientation(int pAngle, int tAngle);

    void resetOrientation();

    /** tilt cam in specified direction by specified degrees */
    void tilt(int degrees);

    /** adjust center of picture to a specific point */
    void adjustCenter(int newCenterX, int newCenterY);

    /** automatic control */
    void automaticControl();

    /** send steering command via RS232 */
    void sendSteeringCommand(int turning);

  };

#endif

/***************************************************************************
    ceye.cpp  -  description

    begin                : Mon Jan 12 2004
    copyright            : (C) 2004 by Allan Krogh Jensen
    email                : s973989@student.dtu.dk
 ***************************************************************************/

#include "ceye.h"

CEye::CEye()
  {

  }

CEye::~CEye()
  {
  }

void CEye::resetOrientation()
  {
  camera.pantiltToHomePosition();
  }

bool CEye::setOrientation(int pAngle, int tAngle)
  {
  //printf("setOrientation res: %u", camera.pantiltSetPosition(false, 200, 200));
  }

void CEye::tilt(int degrees)
  {
  //camera.pantiltSetPosition(true, 0, degrees*100);
  }
```

```
bool CEye::run(int device)
  {
  const int MAX_CMD_LENGTH = 100;
  bool result = true;
  bool quit = false;
  char cmd[MAX_CMD_LENGTH];
  char s[MAX_CMD_LENGTH];
  int err;
  int n;
  int cond;
  bool c;

  // initialize
  snprintf(cmd, MAX_CMD_LENGTH,
           "\n-- RobCam say welcome (compiled %s %s)\n",
           __DATE__, __TIME__);
  sVinfo(cmd, einfo, 0);
  // load default camera settings
  camera.setDeviceNumber(device);
  camera.getCameraName();
  if (camera.isInfoValid())
    camera.loadSettings(NULL);
  // settings may include another device, so revert
  camera.setDeviceNumber(device);
  // start socket server
  //camSockServ.setPort(iPort);
  //camSockServ.startServer();
  //wait a second for socket server to get started
  //Wait(0.1);
  //

  testAvailableCameras(3);

  camera.pantiltToHomePosition();

  Wait(1);

  if (true)
    testCameraImage();


  //
  // print few informations to console (takes time)
  // setInfoToPrint(info, warning, error, debug)
  setInfoToPrint(false, false, true, false);
  // wait for quit

  camera.pantiltSetPosition(true, 0, 0);    //look down 0 degrees
  while (not quit)
  {

    printf("\nType 'q'=quit, 'm'=menu>");
    // wait for signal
    fgets(cmd, MAX_CMD_LENGTH, stdin);
    if (strlen(cmd) > 0)
    {
      switch(cmd[0])
      {
        case 'q': // quit
          closeLogFile();
          quit = true;
          break;
        case 'v': // verbose
          setInfoToPrint(true, true, true, true);
          break;
        case 's': // silent
          setInfoToPrint(false, false, true, false);
          break;
        case 't': // test
          testCameraImage();
          break;
        case 'p': //pan/tilt test
          camera.pantiltToHomePosition();
          break;
        case '8':   //turn
          c = smr.connectSMRCL("smr1", 31001);
```

```
          printf("Return from connect: %u\n", c);
          snprintf(cmd, MAX_INFO_SIZE, "turn 10");
          smr.smrMove(cmd, 20.0, &cond);
          snprintf(cmd, MAX_INFO_SIZE, "fwd 0.1 @v0.2 @a0.5");
          smr.smrMove(cmd, 20.0, &cond);


        break;
        case '2':   //tilt down
          tilt(-2);
        break;
        case 'a':
          {
          automaticControl();
          }
        break;
        case 'l':
          n = sscanf(cmd, "l%s", s);
          if (n == 1)
          {
            err = openLogFile(s);
            if (err != 0)
              printf("Could not open logfile\n");
          }
          else
          {
            closeLogFile();
            printf("Logfile closed\n");
          }
          break;
        case 'm': // print menu
          printf(" ---------------------------------\n");
          printf(" UCamD single camera socket server\n");
          printf(" (Compiled " __DATE__ " " __TIME__ ")\n");
          printf(" 'v': print verbose messages\n");
          printf(" 't': Test\n");
          printf(" 's': silent - print errors only\n");
          printf(" 'l filename': start log to filename\n");
          printf(" 'l': stop logging\n");
          printf(" 'q': quit - terminate server\n");
          printf(" 'p': pan/tilt test\n\n");
          printf(" '8': SMR CL Test\n");
          printf(" '2': tilt down\n");
          printf(" 'a': Automatic Control\n");
          break;
        default:
          break;
      }
    }
    else
      // stop running messages
      setInfoToPrint(false, false, false, false);
  }
  setInfoToPrint(true, true, true, true);
  // save default settings
  if (camera.isInfoValid())
    camera.saveSettings(NULL);
  //
  sVinfo("Stopped", einfo, 0);
  return result;
  }

void CEye::testCameraImage()
  {
  char * name;
  UImage640 imRes;
  char s[MAX_CAM_INFO_SIZE];
  char s2[MAX_CAM_INFO_SIZE];
  CStopWatch sw, tt_sw;

  tt_sw.start();
  //
  //
  if (true)
    {
    // do some initial testing
```

```
    if (true)
      {
      name = camera.getCameraName();
      if (name == NULL)
        sVinfo("No camera available just yet", einfo, 0);
      else
        {
        snprintf(s, MAX_CAM_INFO_SIZE, "Camera: '%s'", name);
        sVinfo(s, einfo, 0);
        }


      //
      // do some image testing
      if ((name != NULL) and (image != NULL))
        {
        image->clear();
        image->isRGB = true;
        image->width = 320;
        image->height = 240;

        imageOut->clear();      /* prepare output image */
        imageOut->isRGB = true;
        imageOut->width = 320;
        imageOut->height = 240;

        printf("Grabbing image...");
        sw.start();
        camera.openAndSetDevice(320, 240, 5);
        camera.getNewImage(rawImage);


        //camera.getNewImage(&image, true);
        if (rawImage->valid)
          { // transfer to full sized image
          rawImage->moveToRGB(image);       /** get input image */

          printf("\t\t\tDone! in %f s raw size(%ux%u)\n", sw.stop(), rawImage->width,
rawImage->height);
          //rawImage->moveToRGB(imageOut);  /** get output image */

          if(image == imageOut)
            {
            printf("/nsame pointers/n");
            }

          ci->inputImage(image, true);     //input ratio 1 to 1
          //ci->getImage(imageOut);

          snprintf(s,MAX_CAM_INFO_SIZE,"%s/input_img%d.bmp",
                      imagePath, rawImage->imageNumber);
          snprintf(s2,MAX_CAM_INFO_SIZE,"%s/output_img%d.bmp",
                      imagePath, rawImage->imageNumber);

          sw.reset();

          /*printf("Saving input image...");
          sw.start();
          image->SaveImageToFile(s, NULL);      /** save input image to file */
         // printf("\t\t\tDone! in %f s size (%ux%u)\n", sw.stop(), image->width, image-
>height);

          sw.reset();

          /*printf("Saving output image...");
          sw.start();
          imageOut->SaveImageToFile(s2, NULL);  /** save output image to file */
          //printf("\t\t\tDone! in %f s size (%ux%u)\n", sw.stop(), imageOut->width,
imageOut->height);


          //printf("Total running time: %f s\n", tt_sw.stop());
          //printf("Output image saved as: ");
          //printf(s2);
          }
        }
```

```
      }
    }
  //
  }

//adjustCenter-----------------------------------------------------------------------
void CEye::adjustCenter(int newCenterX, int newCenterY)
  {
  int panChange = 0;
  int tiltChange = 0;

  printf("\nWanted Focus point: (%i, %i)\n", newCenterX, newCenterY);

  panChange = 2000 * ((newCenterX - 160) / 160.0);
  tiltChange = 2000 * -((newCenterY - 120) / 120.0);

  printf("Pan change: %i, tilt change: %i\n", panChange, tiltChange);

  camera.pantiltSetPosition(true, panChange, tiltChange);

  camera.pantiltGetPosition();
  panAngle = camera.ptPanPos * 0.01;
  tiltAngle = camera.ptTiltPos * 0.01;
  //panAngle = panAngle + (panChange * 0.01);
  //tiltAngle = tiltAngle + (tiltChange * 0.01);

  printf("Absolute orientation: pan angle: %f tilt angle: %f...\n", panAngle, tiltAngle);
  }

//automaticControl-------------------------------------------------------------------
void CEye::automaticControl()
  {
  int cX, cY;
  char s[MAX_CAM_INFO_SIZE];
  char s2[MAX_CAM_INFO_SIZE];
  int loop = 1;
  int state = S_0;            //start in state S_0
  char buf;                    //RS232 commmunication buffer
  char rep;                   //reply buffer
  bool portJustFound;       //true if a port was found in the previous captured image
  int searchPanX;               //x value to focus by if no port is found;
  char cmd[100];
  bool c;
  int cond;

  printf("Starting Automatic Control... Press any key to abort!\n");
  camera.pantiltSetPosition(true, 0,-1500);

  while(loop < 50)
    {
    //input data-----------------------------------------------------------------------
    testCameraImage();               //grab image from cam and process it

    ci->getPortCenter(&cX, &cY);
    //process data---------------------------------------------------------------------
    switch(state)
      {
      case S_0:    //ready to start
        {
        printf("Entering state S_0...\n");

        //if(isStarted()) state=S_1;
        }
      break;
      case S_1:   //follow center of line, high speed
        {
        printf("Entering state S_1...\n");

        //if(branchDetected()) state=S_2;
        }
      break;
      case S_2:   //follow right edge of line, skip wall
        {
        printf("Entering state S_2...\n");

        //if(branchDetected()) state=S_3;
```

```
      }
    break;
    case S_3:   //follow center of line
      {
      printf("Entering state S_3...\n");

      //if(branchDetected() && skipWall) state=S_4;
      }
    break;
    case S_4:   //follow right edge of line, high speed
      {
      printf("Entering state S_4...\n");

      //if(branchDetected() && skipTunnel) state=S_5;
      }
    break;
    case S_5:   //follow right edge of line, skip tunnel
      {
      printf("Entering state S_5...\n");

      //if(branchDetected()) state=S_6;
      }
    break;
    case S_6:   //follow center of line, prepare for outdoor light
      {
      printf("Entering state S_6...\n");

      //if(branchDetected() && !skipStairs) state=S_7;
      }
    break;
    case S_7:   //follow left edge of line, down the stairs
      {
      printf("Entering state S_7...\n");

      //if(crossDetected() && skipFreePorts) state=S_8;
      }
    break;
    case S_8:  //follow left edge of line, to the goal!
      {
      printf("Entering state S_8...\n");

      //final state should be stopped manually!!
      }
    break;
    }

  adjustCenter(cX, cY);                  //adjust camera focus point to center of the
port

  //output data-----------------------------------------------------------------------
  if(ci->portPolygon.isPort())       //if a port is found in the image...
    {
    portJustFound = true;
    printf("PORT DETECTED! \n");

    c = smr.connectSMRCL("smr1", 31001);
    printf("Return from connect: %u\n", c);
    snprintf(cmd, 100, "turn %f", -panAngle);
    smr.smrMove(cmd, 20.0, &cond);
    snprintf(cmd, 100, "fwd 0.1 @v0.2 @a0.5");
    smr.smrMove(cmd, 20.0, &cond);

    /*if(panAngle < 0)
      comPort.sendCommand(true, 1, true, (15.0 * (fabs(panAngle) / 30.0)));
//turn to the left
    else
      comPort.sendCommand(true, 1, false,
       (15.0 * (fabs(panAngle) / 30.0)));         //turn to the right */
    }
  else                                              //if no port found...
    {                    //dont take control if port not found
    comPort.sendCommand(true, 0, true, 0);
    }
    /*
    if(portJustFound)    //if first image were port not found
      {
```

```
        searchPanX = 300;       //initial search x
        portJustFound = false;
        }
      else
        {
        searchPanX = searchPanX - 50;
        if(searchPanX == -50)
           searchPanX = 300;
        }
      adjustCenter(searchPanX, 120);
      }  */

    loop++;
    }

  printf("Automatic control aborted");
  }

//sendSteeringCommand-------------------------------------------------------------------------
void CEye::sendSteeringCommand(int turning)
  {
  printf("Steering Command: %i \n", turning);
  }
```

Appendix B – MatLab filer

Appendix B1 – mask.m

```
function out = mask(inp, mas)
i=1;
for r=1:240
    for c=1:320
        if mas(r,c)
            out(i,:) = [inp(r,c,1), inp(r,c,2), inp(r,c,3)];
            i = i+ 1;
        end
    end
end
```

Appendix B2 – min_sat.m

```
function mi = min_sat(inp)
n=255;
for h=1:n+1
    for i=1:n+1
        mi(h,i) = 1;
    end
end

for p=1:length(inp)
    if inp(p, 2) < mi(round(inp(p, 1)*n)+1, round(inp(p, 3)*n+1))
        mi(round(inp(p, 1)*n)+1, round(inp(p, 3)*n)+1) = inp(p, 2);
    end
end
```

Appendix B3 – max_sat.m

```
function ma = max_sat(inp)
n=255;
for h=1:n+1
    for i=1:n+1
        ma(h,i) = 0;
    end
end

for p=1:length(inp)
    i = round(inp(p, 1)*n)+1;
    h = round(inp(p, 3)*n)+1;
    if inp(p, 2) > ma(i, h)
        ma(i, h) = inp(p, 2);
    end
end
```

Appendix B4 – GLMSat.m

```
%Function to generate a grylevel saturation surface to seperate
%colors from graylevels, dependent of IHS-values. Used for the image
%in the RoboCup-project
%(C) Allan Krogh Jensen 2004
```

```
function ss = GLMSat(CPixMin, GPixMax)
    DEF_GLSM = 1.0;     %default graylevel saturation max
    n = 256;
    file_id = fopen('filter1.dat', 'w');

    %generate seperation surface
    for i = 1:n
        for h = 1:n
            d = CPixMin(i, h) - GPixMax(i, h);
            if d==1      %if value were not represented in input
                ss(i, h) = DEF_GLSM;    %use a constant default value
            elseif d>0  %if d is positive
                ss(i, h) = GPixMax(i, h) + (d / 2); %lay the surface
right between
            else        %if d is negative
                ss(i, h) = DEF_GLSM;
            end
        end
    end

    %saving the seperator surface in file
    for i = 1:n
        for h = 1:n
            fprintf(file_id, '%f\n', ss(i, h));
        end
    end
```

Appendix C – CD-ROM

Vedlagte CD-ROM indeholder kildekode, billeder, MatLab-filer og denne rapport.