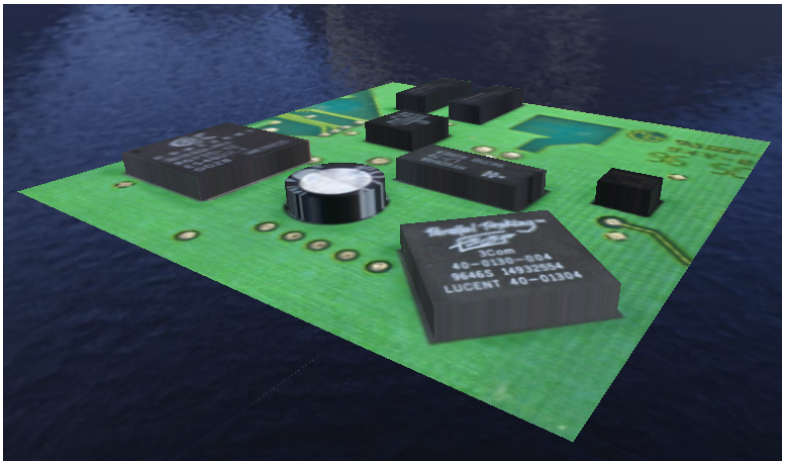
Prosjektrapport for TDT4195 Bildeteknikk



Simon Jonassen og Oddveig Elisabeth Linvåg

Norges teknisk-naturvitenskapelige universitetet
Institutt for datateknikk og informasjonsvitenskap

Innhold

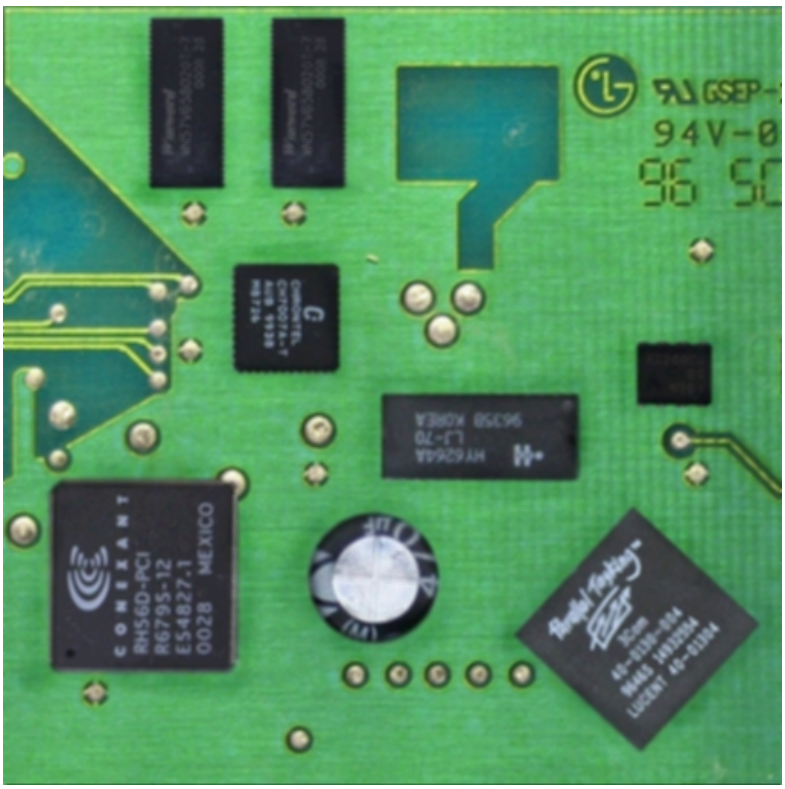
1 Bildebehandling

I bildebehandlingsdelen av oppgaven måtte vi gjenkjenne komponentene på bildet; vi har delt prosesseringen i 6 ulike steg som vi skal forklare under.

1.1 Preprosessering

1.1.1 Metoden

Det gitte bildet inneholder en del støy som trolig ville gitt dårligere resultat ved segmentering. Støyfiltreringen utføres gjennom konvolusjon med en vanlig uniform maske.



Figur 1: Glattet bilde

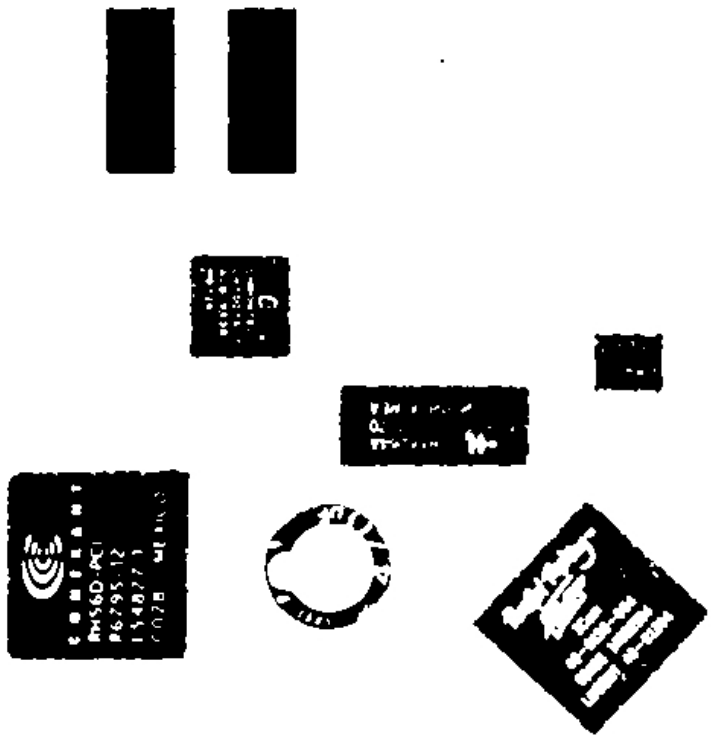
1.1.2 Alternative forslag

Her kunne vi også bruke en eller annen stokastisk filtrering, som f.eks. medianfiltrering, men siden det ikke ble observert noe særlig «salt og pepper»-støy fant vi en slik filtrering unødvendig.

1.2 Segmentering

1.2.1 Metoden

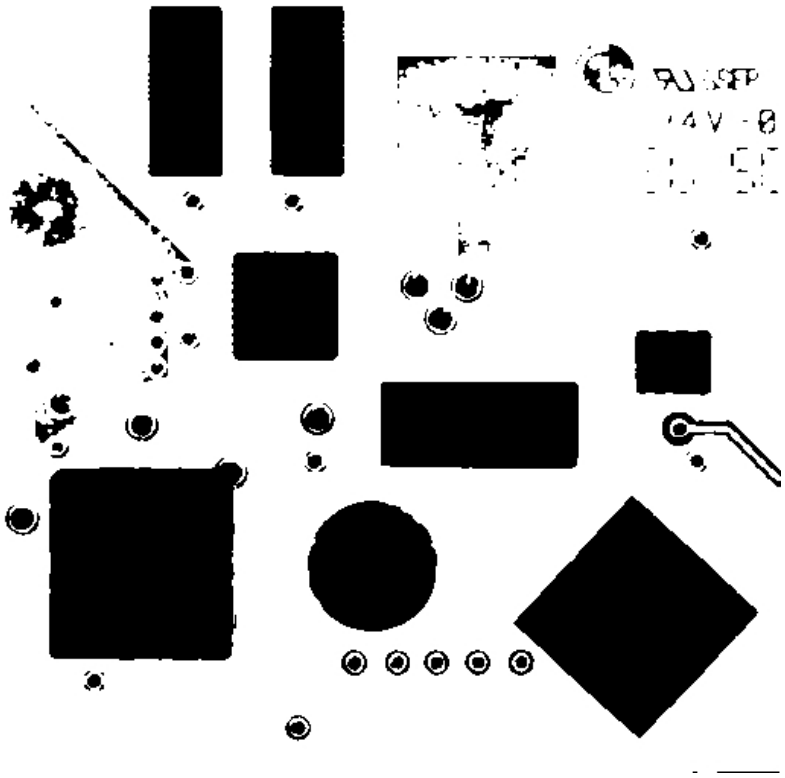
Det vi bruker er en enkel segmenteringmetode basert på andel av grønn farge per piksel - andel av grønt skal enten dominere over rødt og blått, overstige en terskelverdi som vi har satt til 100 etter en del forsøk, eller begge deler. Ulempen med denne metoden er at kondensatoren i midten av bildet blir splittet opp i flere segmenter - dette er noe som bør fikses senere i prosesseringen. Bortsett fra dette så gir segmenteringen et godt resultat (se figur 2).



Figur 2: Segmentert bilde

1.2.2 Variasjoner

Det ble observert at hvis konjunksjon ble brukt i stedet for disjunksjon så kunne det runde elementet (kondensatoren) avbildes litt bedre, men på den andre siden hadde vi da måtte håndtere loddepunktene og kretsstier senere. Vi valgte metoden over siden resultatet er mye enklere å forbedre etterpå.

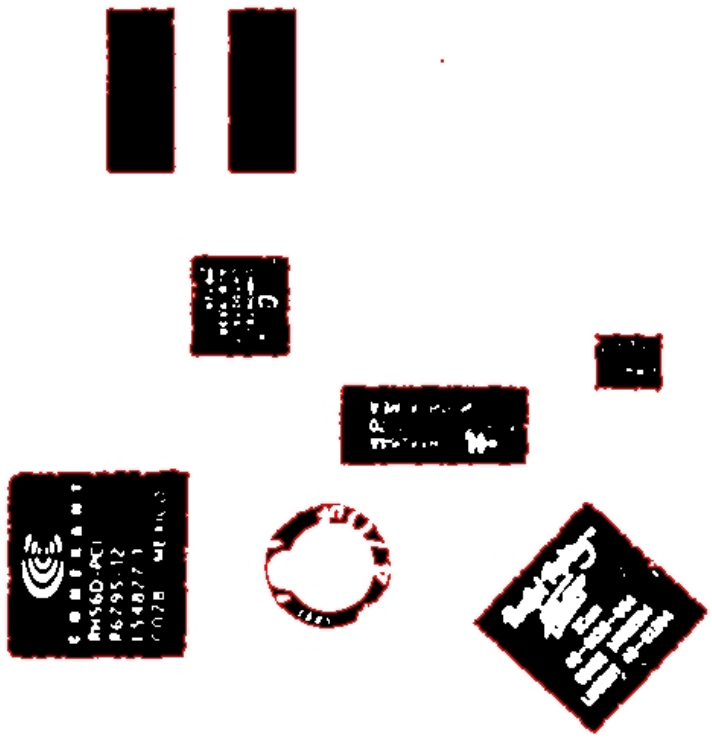


Figur 3: Bilde segmentert med AND i sted for OR

1.3 Kantdeteksjon og polygondanning

1.3.1 Metoden

Vi velger å skanne bildet fra topp til bunn, fra venstre til høyre. Alle kanter som blir funnet og som ikke ble sporet fra før vil følges på innsiden i klokkeretningen inntil det kommer et punkt som ble besøkt tidligere. Alle punkter som blir besøkt underveis lagres inn i en polygonstruktur. Rotasjonstelleren brukes til å forkaste alle interne polygoner (altså polygoner som ligger inn i et annet polygon).



Figur 4: Segmentert bilde med ferdigprosesserte kanter

Siden vi bruker tre fargekanaler (noe som kan være ueffektivt med tanke på at segmenteringen gir et binært bilde) brukes den røde kanalen til å indikere hvilke kanter som er besøkt fra før. På den måte unngår vi å spore samme kant flere ganger. Fargekanalen brukes også til å bestemme om et punkt befinnes inne i et polygon eller utenfor (men dette har vist seg å ha flere unntak som vil fikses av metoden over).

1.3.2 Variasjoner

Vi hadde forsøkt å implementere algoritmen til Wall-Following-Robot fra «Artificial Intelligence: A New Synthesis» av Nils J. Nilsson. Med en gang hadde vi avslørt at algoritmen

kunne gi uendelige løkker i noen spesielle tilfeller hvor en vegg dannet en flaskehals med en annen vegg.

1.3.3 Alternative forslag

Vi kunne hatt implementert Zero Crossing for kant deteksjon og så linke kantene sammen rekursivt: for alle nabopiksler til en piksel som blir valgt tar vi for oss alle naboer rekursivt; den grenen i rekursjonstreet som avslutter med samme piksel som rotnoden har vil returnere et polygon ved tilbakesporing av alle noder opp til rotnoden.

1.4 Forenkling av polygoner

1.4.1 Metoden

Metoden her er ganske enkel - vi følger kanten, og for tre punkter a, b, c beregnes kryssproduktet mellom vektorene fra a til b og fra b til c. Hvis kryssproduktet blir lik eller mindre enn null skal punktet i midten slettes, siden dette danner vinkel mindre enn 2π. Hvis det er mulig skal det forrige punktet, a, sjekkes en gang til. Når metoden har gått rundt en gang vil vi kun sitte igjen med de konvekse hjørnene. Deretter interpolerer vi alle hjørnene med en enkel linjeestimator, $y = b + ax$ eller $x = b + ay$, avhengig av forholdet mellom abs(dx) og abs(dy). Deretter vil vi forkaste alle polygoner som er for små.

1.5 Gjenkjenning av objekter

1.5.1 Metoden

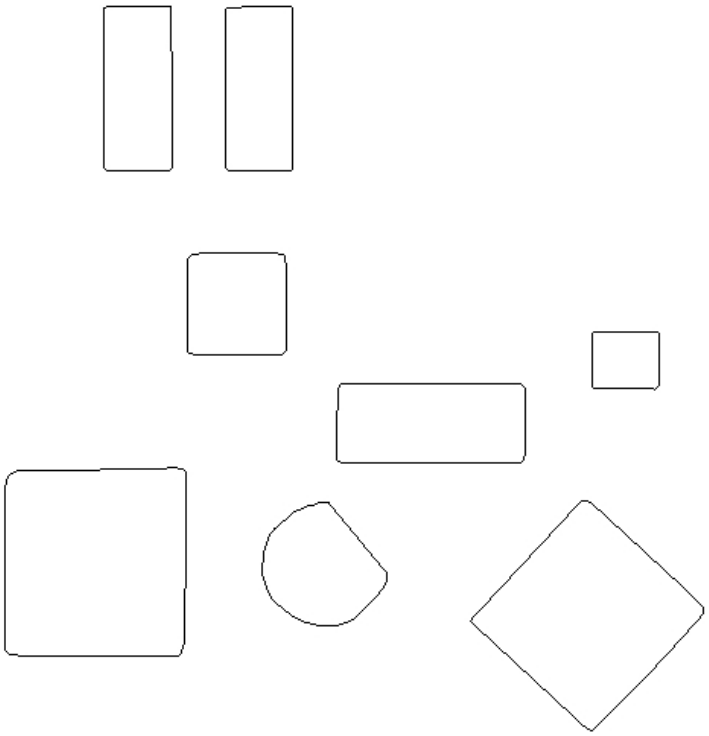
Her vil vi bare følge polygonet langs kanten og måle avstand til det estimerte centroid-punkt (summen av min og maks punkter delt på to), og dersom avstanden ligger mellom 0.9 og 1.1 fra et punkt til polygonets centroid-punkt sier vi at punktet «treffer». Hvis polygonet har over 75 prosent av punktene treffer sier vi at dette er en sirkel. Fordeler med denne metoden er at den tillater å ha defekter i sirkler.

1.5.2 Variasjoner

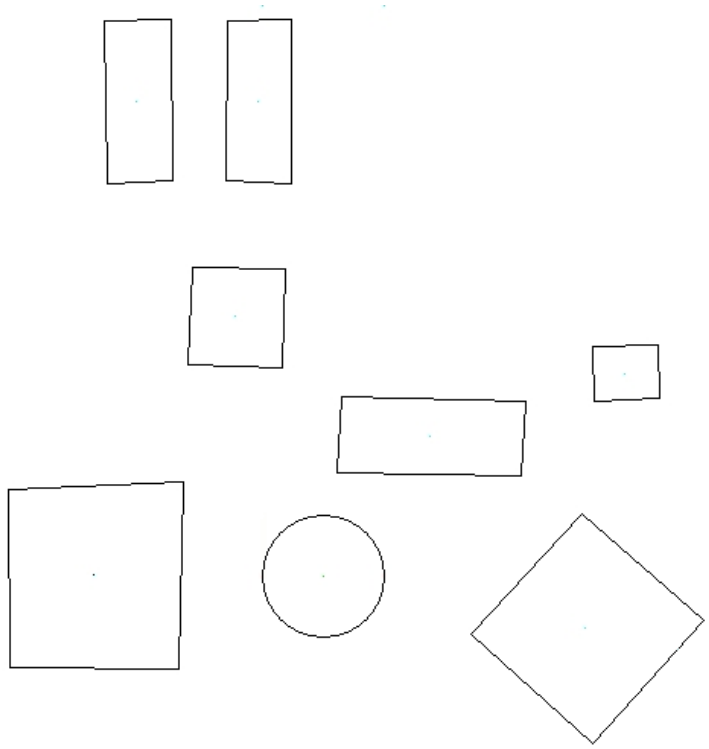
Strengere parametere (f.eks. 0.95 - 1.05 og 90 prosent) vil resultere i høyere sikkerhet og lavere gjenkjenning.

1.5.3 Alternative forslag

Vi har implementert en løsning som fulgte kanten og sjekket om maksimal avstand fra den estimerte origo til et punkt var mindre enn 1.44 (tilnærming av $\sqrt{2}$). Dersom dette stemte skulle polygonet være en firkant (et kvadrat i verste tilfelle). Metoden har vist seg å være riktig i teori, men i praksis ville defekte sirkler kunne bli gjenkjent som firkanter på grunn av eventuelle defekter.



Figur 5: Resultatat av forenkling



Figur 6: Resultatfigurer

1.6 Parameterdeteksjon

1.6.1 Metoden

For sirkler brukes centroidestimat og gjennomsnittsavstanden fra den som randkoordinater. For firkanter blir det litt mer komplisert: Først finner vi et punkt som ligger lengst fra sentrum, og så finner vi det punktet som ligger lengst fra dette første punktet. Til slutt finner vi to punkter hvor summen av avstandene til begge punkter nevnt over er størst, og som ligger over og under linjen mellom disse to punktene tilsvarende.

1.6.2 Alternative forslag

Som det viser seg vil resultatprimitiven for en firkant ikke nødvendigvis være en rettvinklet firkant. Det som kunne gjøres for å fikse dette er å følge kanten mellom to hjørner og finne gjennomsnittsverdi (enten x eller y) for alle punkter - deretter sette en av verdiene til hjørnene til estimatet. Dette vil gi en rettvinklet firkant, men figuren vil stemme med den virkelige figuren ganske lite på grunn av bieffekter med glattings- og segmenteringsmetodene vi har valgt.

Radius til en sirkel kunne estimeres med gjennomsnittsavstand fra sentrum til alle punkter, men metoden ville gi et dårlig estimat på grunn av den ødelagte siden vi har i vår sirkel.

2 Grafikk

I forbindelse med grafikkdelen av oppgaven har vi flere punkter vi vil forklare. Disse er teksturering, oppsett av lyset og tåke, styring og bevegelse, scenestruktur, og kollisjonsdeteksjon.

2.1 Teksturering

Et bilde legges på overflaten til en enkel form som er generert i scenen. Vi har lagt tekstur på følgende:

- Veggene i skyboxen.
- Flaten som representerer grafikkortet.
- Toppflaten og sideflatene på komponentene i scenen.

Tekstureringen vi har valgt å bruke kalles 'Mip-mapping'. Med 'mip-mapping' lagres flere bitmap-kopier av en tekstur, hvor hver kopi har en fjerdedel av oppløsningen til den forrige. Selv om hovedteksturen vil brukes når visningen er stor nok til å gjengi den i full detalj, vil et mer passende mipmap-bilde brukes når teksturen sees på fra avstand eller når den brukes på en mindre flate. Vi har valgt denne løsningen fordi den øker hastigheten på tekstureringen av små polygon slik de vi ønsker å teksturere. Vi bruker

for øvrig 2D teksturer og lineære mipmap for vanlige teksturer, men spenner teksturen til kantene på skyboxen.

For å teksturere sidene til komponentene i scenen bruker vi en spesiell løsning: Vi velger her å bruke piksel langs kanten til toppflatene og 'strekke' disse nedover sideflatene.

Når vi binder teksturer til flater er det viktig å holde orden på teksturkoordinatene, slik at vi er sikre på at vi binder rett hjørne i teksturen til rett koordinat/hjørne på flaten. Vi deler alle punktkoordinater på kortets dimensjon slik at vi får en normalisert teksturposisjon.

For å lese inn bildene bruker vi jpeg-biblioteket fra øvingsopplegget i faget.

2.2 Belysning og tåke

Vi bruker standard oppsett av belysningen og tåken. Det som er viktig å legge merke til her er normalene; i OpenGL må man oppgi normalene til flater for å få lyset riktig. Vi har lagt normalene til komponenter ut fra normalvektorene vi har regnet på forhånd, og til runde komponenter ut fra vinkel under tegningen. Alle normalvektorene er normalisert. Allikevel ble det observert at belysningen er litt feil, nemlig at den skjer relativt til kamera og ikke relativt til objektene.

2.3 Oppsett av kamera, styring, bevegelse

Vi bruker to vektorer for å lagre posisjon, v, til kulen og posisjon til kamera, p. Kamera ser alltid mot kulens posisjon. Når vi skal akselerere oss fram så blir det bare å gjøre det i retningen av projeksjon av vt-vektoren (som er en differanse mellom to overnevnte vektorer) i xz-planen. Rotasjon av kamera skjer rundt p-vektoren, vi flytter p-vektor til v-vektorens x og z verdi økt med inverse vt-vektoren rotert med den ønsket hastighetsvinkel i xz-planen. Kameraet kan også reises opp og ned og flyttes fra og mot objektet. Ved oppreising øker vi kameraets y-posisjon. Ved flytting fra og til objektet flytter vi kamera i retningen med projeksjon av vt-vektoren i xz-planen.

For tastatur og mus definerer vi fire callback-funksjoner. De første to brukes for å håndtere mushendelsene. Den ene vil reagere på klikk og lagre koordinater til muspekeren. Den andre finner differansen mellom siste posisjon og den aktuelle posisjonen, reiser/senker og roterer kamera ut fra data og lagrer posisjonen til muspekeren. De to siste funksjonene reagerer på tastaturhendelsene; vi bruker et heltall for å lagre tastaturstatus: når en tast trykkes ned settes en bit i flagget til 1, når tasten slippes settes biten til 0; i idle slås keyflag og den inverse av bitmasken til hver kondisjon ved hjelp av and-operasjon, hvis resultatet ikke blir 0 skjer det den forhåndsdefinerte hendelsen. (Forslag for å bruke bitfeltene kom fra undervisningsassistentene, de sier at det er ganske 1337 å ha noe sånt :))

For å gjøre bevegelsen til kulen og kamera litt finere bruker vi akselerasjon og friksjon. Akseleasjonen påvirker kulens og kameraets hastighet i vt-retningen. Selve hastighet vil påvirke bevegelsen kun i x og z vektorene, og hastighet avtar med 5 prosent etter hver flytting. Poenget med dette er at kulen og kamera ikke stopper med en gang en slipper tastatur og at vi får en fin sidelengskjøringseffekt (Need For Speed «Motherboard

Edition»?). Sistnevnte effekten blir litt hakkete hvis en bruker mus.

Tastefunskjoner:

- w/s - akselerasjon fram og tilbake
- a/d - snu kamera
- q/e - endre høyde og posisjon til kamera,
- +/- - endre avstand til kulen
- r - reset
- f - fog av/på
- l - lys av/på,
- x- avslutt
- Ved oppstarten brukes w til å bytte bilder.

2.4 Scene og objekter

Vi har forsøkt å implementere noe som ligner på en scenegraf, en begrenset utgave. Først og fremst bruker vi polymorfi på sceneobjekter (bortsett fra kulen som er en del av kamera): det er en klasse som vi kaller for Object3D. Klassen har tre parametrer - x,y,z som kan brukes til å oppgi translasjon relativt til origo eller til forelder objektet. Det er også to viktige funksjoner, nemlig collide(tVector& pos,tVector* mov), som håndterer kollisjoner ved å endre på mov-vektoren hvis det er nødvendig, og render() ,som rendrer objektet ved behov. Deretter defineres det fire klasser - Cubical, Cylinder, Card og Skybox. Den første brukes til å håndtere alle runde objekter, den andre til alle firkantede. De to siste, Skybox- og en Card, har en vektor med barn-objekter; render- og collide-kall på et objekt sendes til barneobjekter og, av den grunn, kan propagere ned i hierarkiet rekursivt. Det er viktig å legge merke til at matrisen legges på stakken og translasjon med x, y og z utføres før innholdet rendres, mens trasnlasjonsvektoren trekkes fra den oppgitte pos-vektoren for å få en lokal pos-vektor før kollisjoner håndteres. Dette skjer nettopp fordi begge prosedyrene skal skje i lokale koordinater.

2.4.1 Begrensniner

Vi har ikke implementert rotasjon siden vi finner dette litt unødvendig for oppgaven våres og alt vi kan få ut av dette er mye overhead under kjøringen. Av samme grunn translerer vi ikke ved funksjonskall på Cubical og Cylinder. Vi finner dette unødvendig å bruke lokale koordinater til å oppgi hjørnene til firkanter eller sentrum til sylinderen og så translere hver gang vi har behov for det, derfor gjør vi alt i koordinater relativt til kortet, og bryter litt på scenegraf-konseptet av den grunn. Altså ikke fordi vi har misforstått dette, men fordi vi finner det mer effektivt å gjøre det på en feil måte.

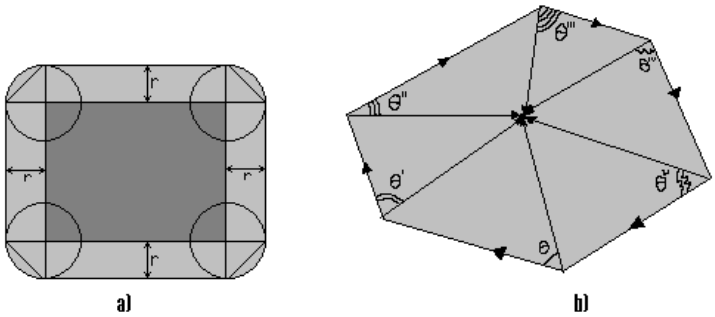
2.5 Kollisjonsdeteksjon

Det er tre objekter kulen kan kollidere med. Disse er: Card, Cylinder og Cubical. Det tillates å falle utenfor Skybox.

Kollisjon med kortet er enkel. Det sjekkes om kulens posisjon er utenfor kortet og rett over overflaten - hvis dette skjer vil y-komponenten til mov-vektoren settes til 0. Poenget med dette er nemlig å få kulen å falle ned hvis den blir utenfor kortet (vi utvider kant med radius til kulen for å tillatte kulen å falle ned kun når den blir helt utforbi kanten).

Kollisjon med sylinder er ikke så vanskelig heller - vi sjekker avstand mellom den nye posisjonen og sylinderens midtpunkt. Hvis avstanden blir mindre enn summen av radiusavstander + feilmargin på 0.001 skjer kollisjonen. I så fall regner vi ut normalvektor til kolisjonsstedet, deretter regner vi ut projeksjon av retningsvektoren på normalvektoren (som en kjenner det fra Matematikk 3 er $proj_{\mathbb{A}}(\mathbb{B}) = \frac{\mathbb{A}\cdot\mathbb{B}}{\mathbb{A}\cdot\mathbb{A}}\mathbb{A}$, ved å trekke retningsvektorens projeksjon på normalvektoren fra den får vi projeksjon av retningsvektoren på linjen som tangerer sylinder i det estimerte kollisjonspunktet.

Det blir mer komplisert for firkantede objekter. Men her er løsningen vi kom til: Først finner vi normalene til kantene; dette gjøres ved å ta kryssproduktet mellom kanten og up-vektoren (0,1,0). Deretter dupliseres polygonhjørner og utvides med normalene, f.eks. punkt a blir om til punkt a_ab og a_da (utvidet med normalvektoren til ab ganget med radius til kulen og delt på absoluttverdi til denne vektoren). Utvidelsen danner et polygon med 8 hjørner, i tillegg lager vi 4 sirkler med radius 0 i hver av hjørnene. Union av figurene (se figur 7) har en kant på avstand r fra firkantens kant, videre sjekker vi bare om kulens nye sentrumsposisjon faller innenfor det nye polygonet eller kolliderer med hjørnene. Kollisjon med hjørnene (sirklene) skjer på samme måte som med det sirkelformede objektet over.



Figur 7: Kollisjonskoneptet

Det vi har igjen er å kollisjon mellom kulens sentrum og det nye polygonet,- for alle kanter finner vi kryssproduktet mellom kantvektoren (i klokkeretning) og vektoren fra sjekkepunktet til den nye posisjonen. Dersom alle kryssproduktvektorer peker opp er punktet innenfor polygonet. Denne metoden kan brukes for å sjekke om et punkt ligger innenfor et hvilket som helst konveks polygon, så lenge alle hjørner ligger i samme plan.

Videre sjekker vi med hvilken side av polygonet kollisjonen skjedde. Dette gjøres ved hjelp av et kryssprodukt av vektoren langs den nye kanten og avstandsvektoren fra starthjørnet til den originale posisjonen. Hvis vi ikke klarer å bestemme oss her resetter vi mov-vektoren, dette vil gi at objektet henger seg i sted for å kjøre seg inn i et polygon, ellers så finner vi skli-vektoren ved hjelp av projeksjon av retningsvektoren på normalvektoren, på samme måte som i tilfelle over.

I alle tilfeller erstatter vi mov-vektoren med skli-vektoren til slutt.

Kollisjonsdeteksjonen våres er langt fra å bli perfekt. Og vi har brukt et par dager på Rose på å optimalisere metoden så mye som mulig. Det vi har nå er ikke perfekt og det er minst to andre måter å gjøre kollisjonsdeteksjon på: Den første er å bruke vanlig bitmap og sjekke kollisjoner ved direkte opplag mot bildet. Vi fant denne metoden lite anvendbar og lite matematisk begrunnet, selv om den ikke er så beregningskrevende. En annen måte er å estimere ligningen for polygonkant ($x = ay + b$ eller $y = ax + b$) og lage et ligningssystem av den med vektor for kulen ($(x - a')^2 + (y - b')^2 = r^2$). Det vi får i så fall er en andregradsligning som har enten en, to eller ingen løsninger. Hvis kollisjonen skjer har ligningen minst en løsning, dermed skal determinanten være større eller lik null.