TDT4195 Bildeteknikk

Av
Sanjay Manikarnika og Kristian Stormark

Innhold

1 Innledning **1**

2 Behandling av bildet **1**
2.1 Segmentering 1
2.2 Kantsporing 3
2.3 Polygontilpasning 4

3 Grafisk utforming **6**
3.1 Struktur 6
3.2 Tegning 6
3.2.1 Tegning av himmel 7
3.2.2 Tegning av vegger 7
3.2.3 Flytting 7
3.3 Kollisjonsdeteksjon 8
3.4 Konfigurasjon av 3D-miljøet 9

4 Avlsutningsvis **10**

1 Innledning

Utgangspunktet for denne oppgaven er et flyfoto av en større labyrint (se figur 1). Vi vil her behandlet bildet slik at labyrinten enkelt kan gis en grafisk fremstilling på en datamaskin. Labyrinten vil på bakgrunn av dette modelleres i 3D vha. *OpenGL*.

Figur 1: "The cow maze"

2 Behandling av bildet

For denne deloppgaven er implementeringen foretatt i *C*.

2.1 Segmentering

Bildet av labyrinten er et forholdsvis godt egnet til segmentering, da skillet mellom hekkene og stiene stort sett er markant. Vår målsetningen med segmenteringen er her å skille ut de delen av bildet som utgjør *hekkene*, ettersom det er disse som hovedsaklig vil representere strukturene i 3D-modellen.

Under dette arbeidet ble flere ulike fremgangsmåter forsøkt, og da med varierende resultat. Vi ble imidlertid tidlig klar over at det var en del støy på bildet, og vi brukte derfor filtrerte utgaver av bildet som utgangspunkt. Det vi hadde størst suksess med i denne sammenheng var å kjøre en serie av påfølgende medianfiltre med maskestørrelse 3 (se figur 2a).

I begynnelsen hadde vi hovedfokus på *fargebaserte* segmenteringsmetoder, ettersom de grønne hekkene og lysebrune stiene tilsynelatende ville være klart separerte i fargerommet (eng. *colorspace*). Det beste resultatet oppnådde vi her ved å filtrere bort alle fargene som ikke var innenfor et spesifisert rektangel (jfr. figur 2b).

(a) Medianfiltering $\times 6$

(b) $(60, 120) \times (70, 130) \times (60, 110)$ RBG-filter på bildet i a)

Figur 2: Bilder fra eksperimenteringen

For å forsøke å forbedre segmenteringen eksperimenterte vi også med andre metoder. Noe uventet oppnådde vi det beste resultatet ved å filtrere mhp. intensistet. I motsetning til ved fargefiltreringen skilte vi da ut de lysere *stiene*, og ikke de mørkere hekkene. Et problem som da oppstod var at de smale og mørkere delene av stiene så ut til å gå tapt. Dette forsøkte vi å kompensere for ved å trekke ut de mørkeste delen av bildet separat og så sammenfatte de to “komponentene”. Resultat ble etter vår mening tilfredsstillende. Vi utførte deretter en medianfiltrering med maskestørrelse 7 for å fjerne siste rest av fragmenter, og avslutningsvis ble bildet invertert slik at hekk-strukturen utgjorden den intensive delen av bildet.
De ulike operasjonene av den endelige segmenteringsprosessen er for fullstendighetsskyld oppsummert under. Bilder fra de ulike stegene er gjengitt i figur 3.

Ordinære stier: 6 medianfiltreringer m/maskestørrelse 3 til gråtone, v/ gj.sn. komponentverdi tresholdfiltrering med skranke 110
Mørke områder: 3 medianfiltreringer m/maskestørrelse 3 $(0,70) \times (0,70) \times (0,70)$ RBG-filtefilterering tresholdfiltrering med skranke 10 medianfiltrering m/maskestørrelse 5
Endelig segmentering: kombiner **Ordinære stier** og **Mørke områder** medianfiltrering m/maskestørrelse 5 invertér

(a) Stier

(b) Mørke områder

(c) De to delene lagt sammen

(d) Filtrert og inverter

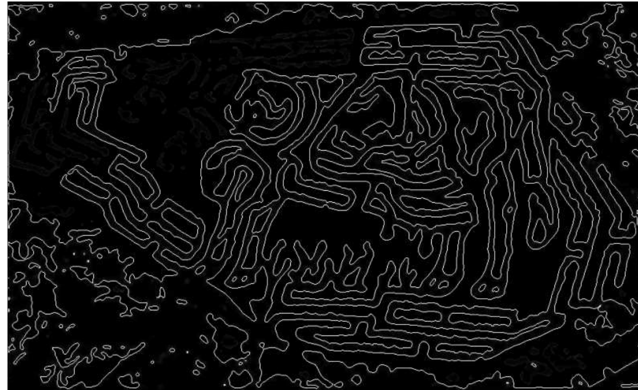
Figur 3: Stegene i segmenteringen

2.2 Kantsporing

For å modellere labyrinten som planlagt var det nødvendig å danne en beskrivelse av hvordan de ulike hekkene" var plassert. I henhold til dette ønsket vi å spore kantenlinje til blokkene i det segmenterte bildet.
Fremgangsmåten som ble valgt kan i pseudokode uttrykkes på følgende form:

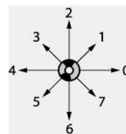
```
for hver linje fra toppen{
  for hver punkt fra venstre{
    if (punkt klassifiseres som struktur){
      spor strukturen som punktet tilhører;
      lagre sporet i minnet;
      slett sporet og punktene innenfor fra bildet;
    }
  }
}
```

Figur 4 viser resultatet av en slik sporing. Merk for øvrig at de store, sorte områdene i denne ikke direkte skyldes tap pga. segmentering, men at disse utgjør *innsiden* av strukturer og derfor er blitt slettet under sporingensprosessen.



Figur 4: Kantlinjer

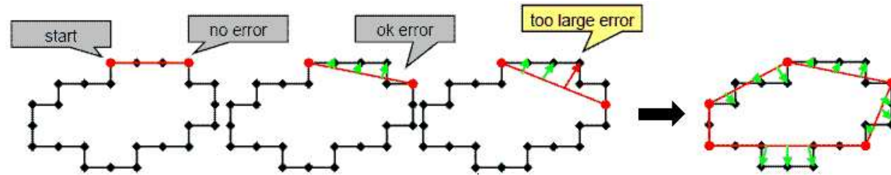
For å spore hver struktur ble det anvendt såkalte *chain codes*, og hver kantlinje ble gjennomløpt ved å konsekvent holde strukturen på høyre side. Dette ble gjort ved å registrere langs hvilken retning et punkt ankommes og så velge retning i henhold til dette og de omliggende punktene. De mulige forflytningsretningene ble angitt i henhold til fremstillingen i figur 5.
Ankommes f.eks. et punkt langs langs retning -4 ($=0$) er den prioriterte rekkefølgen av avgangsretninger gitt ved $3, 2, 1, 0, 7, 6, 5, 4$. Tilsvarende er rekkefølgen dersom det ankommes langs retning -6 gitt ved $5, 4, 3, 2, 1, 0, 7, 6$. Den første av disse avgangsretningene som tilhører strukturen velges, og den ønskede sporingen oppnås ved å registrere alle besøkte punkter. Strukturer som består av isolerte punkt ble behandlet spesielt, nærmere bestemt ved at disse forkastes.



Figur 5: Ulike forflytningsretninger

2.3 Polygontilpasning

For å mer hensiktsmessig kunne representere hekkene ble hver struktur tilpasset ved bruk av polygoner. Dette ble gjort ved å finne et polygon som beskriver strukturen med tilfredsstillende nøyaktighet, og det ble brukt feilmål som skissert i oppgaveteksten (jfr. figur 6).



Figur 6: Polygontilpasnings metode

Ved å anvende denne fremgangsmåten på de allerede sporede kantlinjene ble labyrintstrukturen representert som vist i figur 7. De hvite punktene angir da hjørnene i hvert polygon. Se for øvrig skjermbildet bakerst i dette dokumentet.

Figur 7: Polygontilpasning

Kildekode er gjengitt under. For en beskrivelse av datastrukturer henviser vi til den vedlagte, fulle kode.

```
polypoints *trimPoints (polypoints *old_points, double maxDist){
  polypoints *new_points, *retObj, *a, *b, *c, *prev;
  new_points = retObj = createPolypoints(old_points->posX, old_points->posY, NULL);
  c = a = old_points;   prev =  b = c->next;
  while (  !((b->posX==retObj->posX)&&(b->posY==retObj->posY))  ){
    if (measureDist(a,b,c)>=maxDist){
      new_points->next = createPolypoints(prev->posX, prev->posY, NULL);
      new_points = new_points->next;
      c = a = prev;
      b = c->next;
    }
    else if (c==b){
      prev = b;
      b = b->next;
      c = a;
    }
    else{  c = c->next;
    }
  }
  new_points->next = createPolypoints(b->posX, b->posY, NULL);
  return(retObj);
}
```

3 Grafisk utforming

Denne deloppgaven er implementert i *C++* og er bygget opp mot windows-bibliotekene. Koden kompilerer under Visual Studio. Utførelsen er inspirert av NeHe "lesson01"[1] mhp. initsiering av *OpenGL*.

3.1 Struktur

Laberynten er definert av polygoner som vi hentet ut av bildet. Disse polygonene er representert av en struktur. Denne holder verdier som xmin, xmax, zmin, zmax, antall vegger i strukturen (numofQuads) og en peker til disse veggene (qdata). Dataene xmin, xmax, zmin og zmax benyttes i kollisjonstesting som blir beskrevet senere.

```
typedef struct sectionDEF {
   float xMax;
   float xMin;
   float zMax;
   float zMin;
   int numofQuads;
   QuadWall *qdata;
} Section;
```

Veggene som er i hver seksjon er av datatype QuadWall. Dette er en ny struktur som definerer en vegg. Den har altså 4 punkter som er definert av typen *Vertex*.

```
typedef struct quadWallDEF{
    Vertex UpperRight;
    Vertex UpperLeft;
    Vertex LowerLeft;
    Vertex LowerRight;
} QuadWall;
```

Vertex er punkter og holder informasjon om posisjon, teksture kordinater og normalvektorer. Siden vi opererer i 3D er posision og normalvektor definert vha. en egen struktur; *Vector3d* som inneholder 3 flyttallsverdier til hver av de respektive dimensjonne. Normalene blir brukt til å la *OpenGL* beregnye lysitensitet dersom vi velger å skru på lyseffekter.

```
typedef struct Vector3dDEF {
        float x;
        float y;
        float z;
} Vector3d;

typedef struct vertexDEF {
        Vector3d position;
        Vector3d normal;
        float u;
        float v;
} Vertex;
```

Programmet laster først inn en del innstillingsparametre fra filen *config.ini*. Deretter leses data for labyrinten fra fil, samtidig som nødvendige utregninger (normalvektorer, teksturekoordinater, xmin,xmax,zmin,zmax etc) foretas. Programmet kjører så en hoved-løkke hvor tegning, flytting og kollisjonstesting utføres.

3.2 Tegning

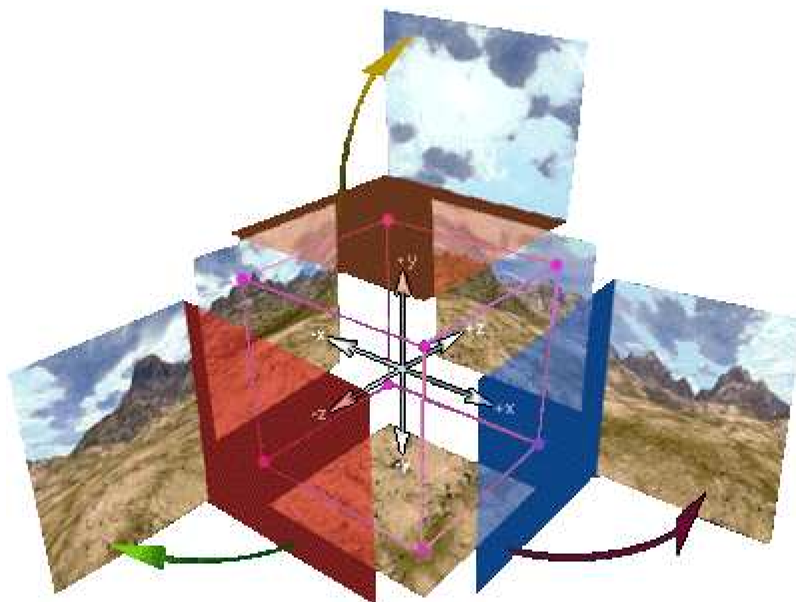
Alle tegningene av vegger, gulv og himmel blir utført med *GL_QUAD*. Det brukes *GL_LINE* og *GL_LINE_STRIP* for tegning av linjer på kartet. Teksturene er bmp-bilder og leses via

[1] http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=01

auxDIBImageLoad-funksjonen.

3.2.1 Tegning av himmel

Vi benytter oss av en teknikk kalt skybox. Denne går ut på å tegne en boks rundt kamerats posisjon der sidene peker innover og har teksture av himmel. Det er vanlig å skru av dybde testing når man tegner himmelen ettersom man ikke ønsker at lys eller tåke skal påvirke himmelen. Skisse over skyboks:



Siden man skrur av dybde testing er det samme hvaslags kordinater man bruker så lenge de definerer en kube. Vi har brukt 5 flater istedenfor 6 som er vist på tegningen. Flaten vi ikke har tatt med er den som definerer bakken langs -y.

3.2.2 Tegning av vegger

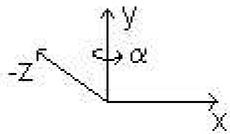
Tegningen av vegger gjøres gjennom en løkke som først går gjennom alle seksjonene og for hver seksjon går gjennom veggene. Teksure kordinatene blir skalert idet vi leser fra datafilen. Dette gjør at teksturene repeteres flere ganger langs store vegger. Se drawSector(int index) i kildekode for tegning og readFileData() for fillesning.

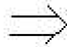
3.2.3 Flytting

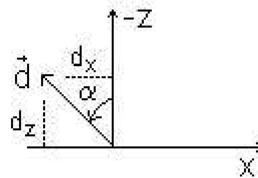
For å forenkle flytting og rotering av kamerat er det laget en egen struktur;

```
typedef struct cameraDEF {
   Vector3d position;
   Vector3d direction;
   Vector3d speed;
} CAMERA;
CAMERA camera;
```

Flytting og rotering skjer ved å oppdatere variablene i *camera*. For å bevege seg i riktig rettning er det nødvendig med et par geometriske betraktninger. Rett frem vil for eksempel ikke være langs z-aksen dersom vi har rotert kameraet. Følgende skisse over geometri og trigometriske forhold:







forover flytt:
x=a*sin(α)
z=a*cos(α)

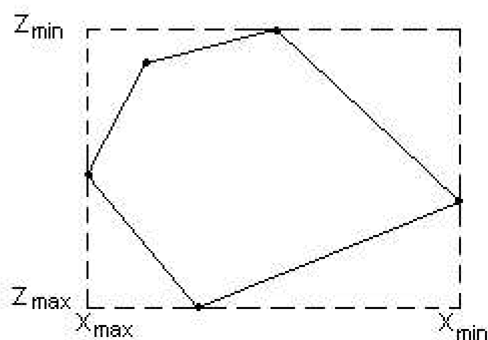
Dette gir oss de relasjonene vi behøver. Et lite konverteringsproblem oppstår da *glRotatef* opererer med grader, mens vi vil bruke radianer i argumentene til cos og sin.

```
if (keys[VK_UP]){
 //convert from degrees to radians and set speed for x,z components.
 camera.speed.z +=(float)  (0.1f*cos((camera.direction.y/360)*2*3.146));
 camera.speed.x +=(float) (-0.1f*sin((camera.direction.y/360)*2*3.146));
}
```

Posisjonen til kameraet vil bli oppdatert senere i løkken, så sant vi ikke kolliderer i en vegg.
Rotering av kamera skjer ved å oppdatere *camera.direction* etter input fra tastaturet eller mus.

3.3 Kollisjonsdeteksjon

Testing av kollisjon skjer i flere steg. Først sjekkes kameraposisjon opp mot hver seksjon. Vi bruker her en bokstest for å sjekke om kameraet er i nærheten av gjeldende seksjon. Boksene er definert slik at de inneholder hele seksjonen, og vi bruker da xmin, xmax, zmin, xmax for oppretting av boks.



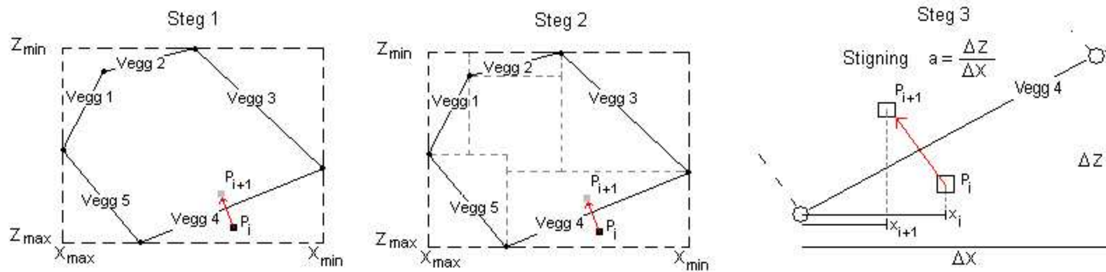
Dersom kameraet vil flytte seg til punktet (x,z) er det innenfor den stiplede firkanten dersom

$$X \geq X_{min} \text{ og } z \geq Z_{min}$$

$$X \leq X_{max} \text{ og } z \leq Z_{max}$$

er oppfylt.
Denne betingelsen blir sjekket for hver seksjon.
Dersom ingen av seksjonene oppfyller betingelsen har vi ingen kollisjon. I motsatt tilfelle må vi teste nærmere på hver vegg som bygger opp seksjonen. Dette fører oss til steg 2 av prosessen.
Hver vegg i seksjonen testes nå for kollisjon. Vi oppretter igjen bokser, men denne gangen inneholder de hver sin vegg. Illustrasjonene under viser hvert steg. Punktet P_i er posisjonen

kameraet befinner seg i før flytt og P_{i+1} er posisjonen etter flytt.
Vi har da $P_{i+1} = P_i + d$, der d er kameraets hastighetsvektor. Vi legger merke til at dersom P_i er innenfor boksen i steg 1 vil den også være innenfor en av veggene i steg 2.



I siste steg avvgjør vi hvor vidt det nye punktet P_{i+1} krysser en vegg. Det første vi er interessert i er stigningstallet på linjen veggen danner. Dersom $\Delta Z = 0$ eller $\Delta X = 0$ vet vi at linjen er rett. Vi vet da at vi har en kollisjon (vi ville i dette tilfelle hatt en boks som var uendelig tynn, men har satt at boksene må ha større bredde enn $|d|$).
Når $\Delta Z > 0$ og $\Delta X > 0$ er vi nødt til å ta hensyn til den skrå linjen. Stigningstalle blir beregnet som $a = \Delta Z/\Delta X$. Vi benytter oss av linjerepresentasjonen $f(x) = ax$ for videre beregninger.

$Z_1 = f(x_i) - z_i$, der punktet z_i er z-kordinaten til P_i.
$Z_2 = f(x_{i+1}) - z_{i+1}$ der punktet zi_{+1} er z-kordinaten til P_{i+1}.

Dersom Z_1 og Z_2 har forskjellig fortegn vet vi at vi prøver å krysse veggen. På denne måten vil vi fange opp kollisjon uavhengig av retning man prøver å krysse linjen.
Som nevnt ovenfor vil eksakt kollisjonstesting kunne gi bokser som er uendelig tynne. Man vil også se en gjennom veggen dersom man kan bevege seg veldig nær den. Programmet opererer derfor med en global parameter *COL_MARGIN* som er ment som et mål på maksimal avstand til en vegg. Kort sagt adderer denne seg med hastighetsvektoren. Ved store hastigheter kan denne settes lavt, og ved små hastigheter bør den økes.

3.4 Konfigurasjon av 3D-miljøet

I *config.ini* er det altså mulighet for å stille på en del parametere for å forandre utseende og oppførsel til 3D-miljøet. Listen av parametere som er implementert i siste version:

Parameter	**Beskrivelse**
gl_fog:	Angir om programmet skal bruke tåke-effekter. 0 = av, 1 = på.
gl_lightning:	Angir om programmet skal bruke lys-effekter. 0 = av, 1 = på.
wall_height:	Angir høyden på veggene I labyrinten. Vanlige verdier: ¡1..5¿
world_scale:	Faktor kordinatene vil bli skalert med: 1 = ingen skallering, høyere = mindre
collision_margin:	Angir minste avstand til vegg før kollisjon. Vanlige verdier ¡0..4¿
texture_repeat_x:	Angir hyppigheten av repitisjon på veggteksture langs x. Vanlige verdier ¡1..5¿
texture_repeat_y:	Angir hyppigheten av repitisjon på veggteksture langs y. Vanlige verdier ¡1..5¿
map_data:	Filnavnet til datafilen vi ønsker å benytte. Default: Maze.txt

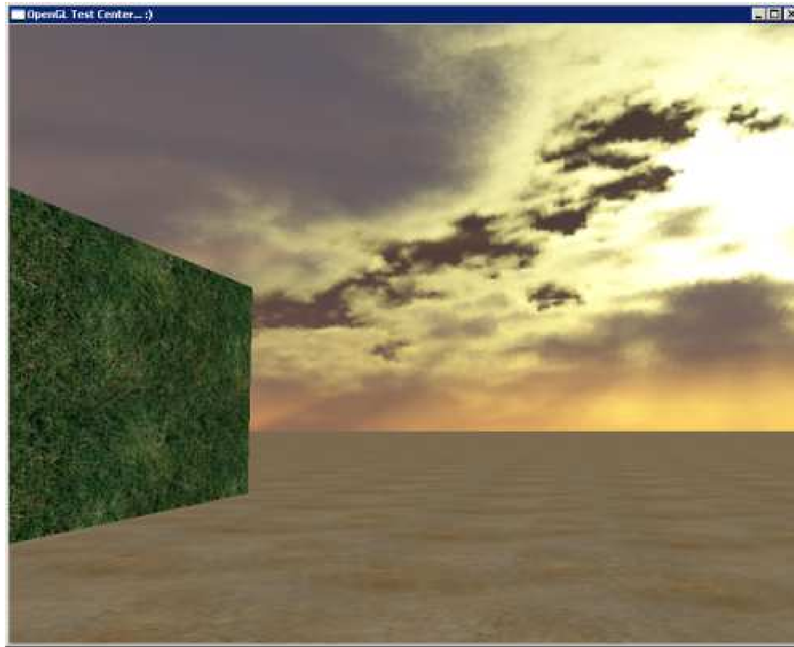
Det må brukes mellomrom mellom parameterene og verdien. Etter verdien skal det være nøyaktig ett linjeskift. I tillegg må rekkefølgen på parameterene være den samme som spesifisert ovenfor.
Feil i filen vil føre til programavbrudd og eventuelt eller andre ubekvemmeligheter.

4 Avlsutningsvis

Vi ser selv at det er en del ting som kunne bli gjort bedre. Eksempelvis kunne segmenteringen med fordel ha blitt finere, slik at vi hadde fått større nøyaktighet øverst i venstre hjørne. Vider kunne polygonene ha blitt klippet i mindre enheter for å sørge for at kollisjonsdeteksjonen blir så effektiv som mulig. I tillegg kunne grafikk-delene helt sikkert blitt optimalisert mer. Tross alt dette er vi likevel fornøyde med resultatet, og noen "screenshots" fra programmet følger:

Figur 8: Med lyssetting

Figur 9: Uten lyssetting



Figur 10: Solnedgang!

Figur 11: Kart