SAMMENDRAG AV HOVEDPROSJEKT

Tittel:	«Interaktiv visualisering av kaustikkmønstre»	Nr. : 11
		Dato : 18.05.04
Deltaker(e):	Mads Nyborg	
	Lars Petter Madsstuen	
	Maria Wroldsen	
Veileder(e):	Ivar Farup	
	Jan Henrik Wold	
Oppdragsgiver:	Kunsthøgskolen i Oslo	
Kontaktperson:	Harald Anthonsen	
Stikkord (4 stk)	Java, raytracer, lysgeometri, brukervennlighet	

Antall sider: 57	Antall bilag: 10	Tilgjengelighet (åpen/konfidensiell): åpen

Kort beskrivelse av hovedprosjektet:

Oppgaven går ut på å utvikle et program for Kunsthøgskolen i Oslo. Programmet skal brukes til eksperimentering med lys og speilende flater.

Brukeren skal i et «tegneområde» ha mulighet til å plassere speil med ulike geometriske former og lyskilder. Et av hovedpoengene med programmet er at brukeren i sanntid skal se lysgangen fra lyskildene via de speilende flatene. Noen ganger vil dette resultere i lysende kurver på de stedende hvor konsentrasjonen av lysstråler er høyest. Disse lysende kurvene kalles kaustikker.

Programmet gir mulighet for å rendre «tegneområdet» til 3D slik at brukeren får et mer realistisk bilde av kaustikkene.

Programmet skal være brukervennlig. Det skal derfor legges stor vekt på GUI og hjelpefunksjoner.

Forord

«Interaktiv visualisering av kaustikkmønstre» er laget av tre studenter ved Høgskolen i Gjøvik som et hovedprosjekt våren 2004. Oppdragsgiver for dette prosjektet er Kunsthøgskolen i Oslo ved Harald Anthonsen.

Formålet med denne rapporten vil være å dokumentere arbeidet vi har gjort underveis i prosjektperioden. Den vil også være til hjelp dersom noen ved en senere anledning vil fortsette å arbeide med programmet.

Vi vil gjerne takke veilederne våre, dr. Ivar Farup og cand. mag Jan Henrik Wold, for god oppfølging og hjelp underveis i prosjektet. Ellers ønsker vi også å takke dr. Are Strandlie og høgskoleingeniør Øivind Kolloen for at de stilte opp og hjalp oss i den perioden som de andre veilederne ikke var tilstede på skolen.

Gjøvik, 18. mai 2004

_____

Lars Petter Madsstuen

_____

Maria Sunde Wroldsen

_____

Mads Nyborg

Innholdsfortegnelse

Figurer

Liste over vedlegg finnes på side 59.

1 Innledning

1.1 Oppgavedefinisjon

Ved Kunsthøgskolen i Oslo arbeides det blant annet med lys som kunstnerisk uttrykksmiddel. Spesiell oppmerksomhet er viet billedformer med basis i såkalte kaustikker. Kaustikker er lysende kurver som fremtrer i bestemte mønstre når et system av lyskilder og krumme speilende flater stilles opp på et diffuserende underlag, for eksempel papir. Ved å tilpasse lyskildenes og speilenes form, posisjon, orientering og bevegelse kan man frembringe alt fra de enkleste kaustikkmønstre til de mest iøynefallende lysspill.

I den kreative, utprøvende fasen er det imidlertid et hinder at det er svært tidkrevende å lage de ulike fysiske oppsettene. Det er derfor ønskelig å kunne eksperimentere med kaustikkmønstre ved hjelp av en interaktiv grafisk simuleringsprogramvare. Det er selvfølgelig mulig å benytte kommersielt tilgjengelige 3D-programmer, men disse er ofte såpass generelle og dermed tunge å bruke at terskelen for en ikke-datakyndig i praksis blir for høy.

Oppgaven vår går derfor ut på å lage et program som gjør det mulig å prøve ut ulike plassering av speilende flater og lyskilder, ved hjelp av et program som letter arbeidet og tidsforbruket, i eksperimenteringsfasen.

1.2 Målgruppe

Målgruppen for dette programmet er ansatte og studenter ved Kunsthøgskolen. De ønsker å benytte seg av programmet i forbindelse med undervisning og eksperimentering slik at de slipper fysiske installasjoner som er veldig tidkrevende i utprøvingsfasen av lys- og speilinstallasjoner.

Målgruppen for rapporten er hovedsakelig sensor og eventuelt noen som skal utvikle programmet videre ved en senere anledning.

1.3 Formål

Ønsket om et slikt program kom fra Kunsthøgskolen ved oppdragsgiver Harald Anthonsen. Grunnen til dette var et ønske om å spare tid i forbindelse med eksperimenterings-fasen. Et slikt program vil forenkle deres hverdag i forberedelsene til verkstedet, ved at de slipper å tegne opp lysstrålene og refleksjonene som vil forekomme for hånd.

Studentene ved Kunsthøgskolen har ikke all verdens datakunnskaper, og vi har derfor forsøkt å legge vekt på brukervennlighet underveis. For vår egen del har vi ikke tidligere jobbet noe særlig med å lage brukervennlige program av noe slag.

Vi har forsøkt å sette fokus på objektorientering. Dette har vi gjort for at det skal bli lettere å videreutvikle programmet på et senere tidspunkt. Et annet formål med prosjektet har rett og slett vært å gjennomføre et større prosjekt og gjøre erfaringer i forbindelse med det.

1.4 Egen bakgrunn og kompetanse

Alle i prosjektgruppen er 3. års studenter som går bachelor i ingeniørfag, data, på Høgskolen i Gjøvik. Vi har alle gått retning for programmering i 3. klasse. Det passet derfor bra med et programmeringsprosjekt av denne typen. I og med at vi har gått programmering, har alle god kjennskap til Java. Det at vi har jobbet med et velkjent språk, har gitt oss muligheten til å fokusere på det virkelige problemet, og ikke ting som GUI og lignende.

Vi måtte også begynne å sette oss inn i Java3D-verdenen, da vi trodde dette kunne komme til nytte. Men akk og ve, hvor feil kan man ta? Vi gikk derfor over til raytraceren POV-ray i stedet.

Vi har også brukt tid til å sette oss inn i nytt stoff, blant annet Java2D og POV-ray. Ellers kan vi nevne at det er første gang vi gjennomfører et prosjekt av denne størrelsen.

1.5 Utviklingsmodell

Vi har tatt utgangspunkt i fossefallsmodellen. Gruppen var i utgangspunktet negativ til denne modellen, med tanke på at den i mange tilfeller medfører lang tids arbeid med et «usynlig» produkt. I ettertid synes vi at modellen har fungert tilfredsstillende. Det har vært en ryddig måte å jobbe på.

Modellen krever også at alle krav skal avdekkes først i prosjektet. Dette passet oss godt, da det gjorde at vi relativt tidlig fikk god oversikt og forståelse for hvilke krav som var satt, og dermed hva som skulle utvikles.

1.6 Rammer/arbeidsformer

Prosjektet utgjør en arbeidsbelastning på 20 studiepoeng. Vi har jobbet jevnt med prosjektet hele våren. Ved prosjektstart fikk vi tildelt en pc og et rom i kjelleren som skulle deles med en annen gruppe. På grunn av at vi kun fikk tildelt én pc har vi valgt å sitte på A 213-laben slik at alle kunne jobbe

med hver sin pc. I starten jobbet vi en del sammen for å spesifisere krav og bestemme det grunnleggende. Når vi begynte med koding derimot, har vi delt arbeidsoppgavene mellom oss og jobbet mer hver for oss. Vi har stort sett jobbet til samme tidspunkter og på skolen sånn at vi har spurt hverandre dersom det skulle være noe.

Stort sett har vi vært på skolen på dagtid gjennom hele semesteret. Mot slutten har selvsagt arbeidet blitt mer intensivt og kveldene lengre. Logg for arbeidet er vedlagt prosjektrapporten (vedlegg F).

1.7 Organisering av rapporten

Vi har brukt malen for hovedprosjekter på Høgskolen i Gjøvik som grunnlag for oppbyggingen av denne rapporten.

Rapporten er delt inn i sju kapitler:

1. **Innledning**
   Her tar vi for oss oppgavedefinisjon, formål utviklingsmodell, rammer etc.

2. **Kravspesifikasjon**
   Dette kapitlet handler om hvilke krav vi satte til programmet i starten av prosjektet.

3. **Design**
   Dette kapitlet omhandler designet av programmet, både utseende- og systemmessig.

4. **Koding/implementasjon**
   Her tar vi for oss hvordan vi implementerte funksjonaliteten som ble bestemt under kravspesifikasjon- og designfasen.

5. **Diskusjon av resultater**
   Her drøfter vi hvordan prosjektet ble i forhold til hva vi planla i starten.

6. **Konklusjon**
   Kort konklusjon/oppsummering av prosjektet.

7. **Appendiks/vedlegg**

2 Kravspesifisering

2.1 Omgivelser

Systemet skal utvikles i Java og vil dermed være plattformuavhengig. Dette har imidlertid ikke noe å si for Kunsthøgskolen i og med at de kun har Windowsbaserte maskiner.

Med i installasjonen skal det være en rett versjon av JVM (Java Virtual Machine) og Povray slik at brukeren slipper å tenke på det.

Programmet skal brukes av en bruker på en maskin om gangen. Programmet skal kunne kjøres på relativt nye Windows-maskiner.

2.2 Systemets brukere

Programmet skal brukes av både studenter ved Kunsthøgskolen og kunstnere som ønsker å eksperimentere med lysgeometri. De har lite kjennskap til data.

2.3 Funksjon

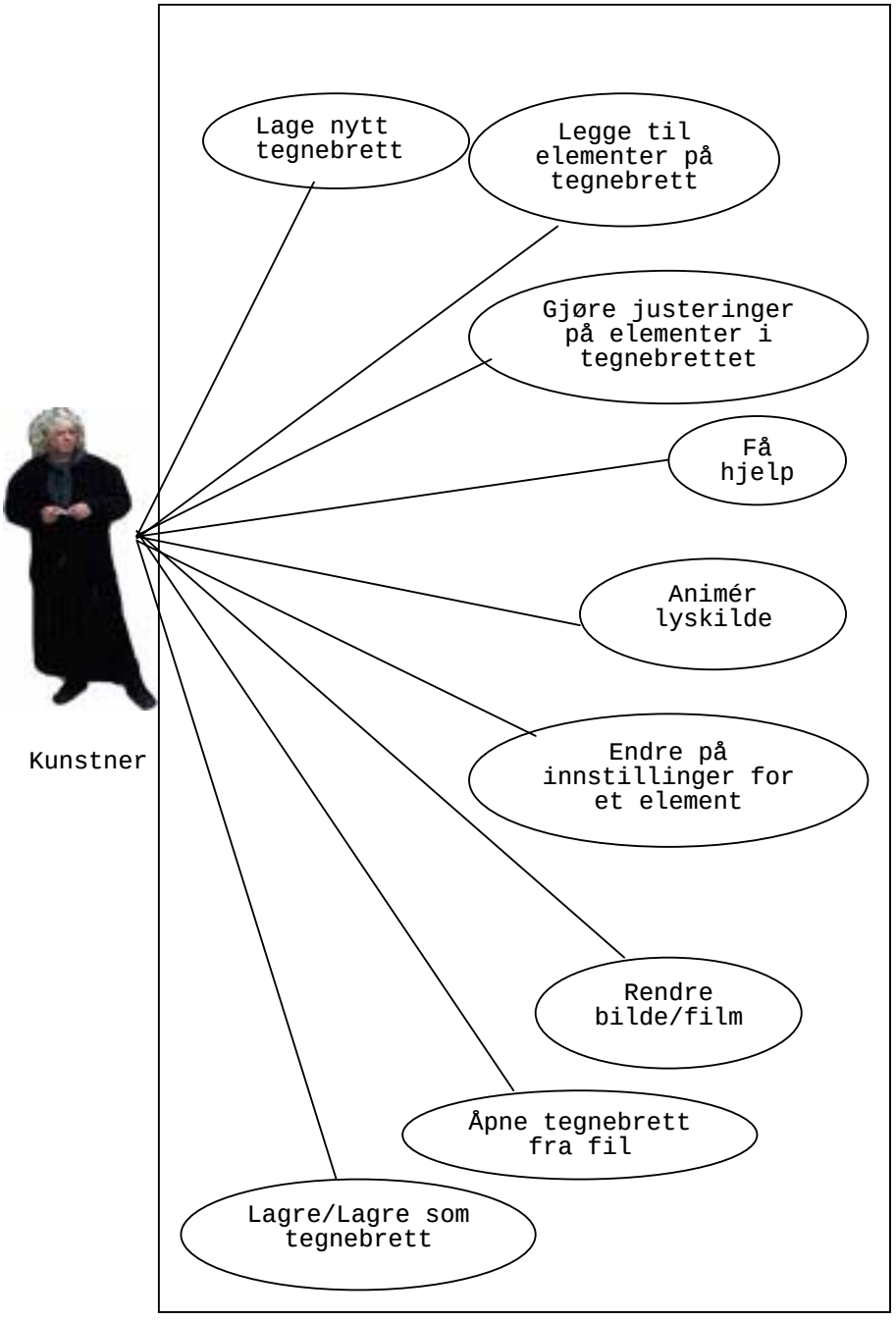
Vi har valgt å dele systemet i moduler, slik:

- Tegnebrett: Det skal lages et tegnebrett som brukeren kan leke med.
- Filhåndtering: Det skal være mulig for bruker å lagre og åpne tidligere lagrede filer.
- Tegneverktøy: Det skal være flere ulike tegneverktøy som brukeren kan legge til elementer på tegnebrettet med.
- Hjelpemodul: Brukeren skal ha mulighet for å spørre systemet om hjelp.
- 2D-visning av lysstråler (og kaustikkurver): Brukeren skal ha mulighet for å se lysgangen i en 2D-forhåndsvisning.
- GUI: Brukergrensesnittet skal være så enkelt som mulig, men likevel inneholde en del kompleks funksjonalitet.
- oDDplaYer: Brukeren skal kunne se animasjoner som film eller bilde ved bruk av denne.
- Rendre-modul: For at tegnebrettet skal vises i 3D må det rendres.
- Animasjon: Det skal finnes enkel funksjonalitet for animasjon av lyskilde.
- Måloversikt (arbeidstegning): Brukeren skal kunne få en oversikt over mål på de forskjellige elementene og deres plassering.
- Hvis tid: Undo/Redo: Brukeren skal ha mulighet for å angre/gjøre om operasjoner på tegnebrettet.
- Hvis tid: Kopier/Klipp ut/Lim inn: Brukeren skal ha mulighet for å kopiere/klippe/lime elementer fra/til tegnebrettet.
- Hvis tid: Et egenskapervindu for å justere på elementene geometrisk ved hjelp av kjente matematiske parametere.

2.4 Operasjon

Vi har valgt å bruke Use Case for å beskrive hvordan systemet forventes å fungere i normal tilstand (beskrivelse). På noen har vi også tatt med alternativer som beskriver hvordan programmet opptrer i feilsituasjoner.

2.4.1 Use Case diagram



Figur 1 Use Case diagram

2.4.2 High-Level Use Case-beskrivelse

I alle Use Case'ene er aktøren bruker.

Use Case:	Lage nytt tegnebrett
Beskrivelse:	Bruker ber systemet opprette et nytt tegnebrett, systemet ber bruker velge størrelse, så blir tegnebrettet opprettet.

Use Case:	Legge til elementer på tegnebrett
Beskrivelse:	Bruker velger et verktøy fra verktøytoolbaren, og kan ved hjelp av museaktiviteter benytte dette verktøyet i tegnebrettet. Når bruker slipper museknappen vil elementet bli lagt til i tegnebrettet.
Alternativer:	Dersom brukeren prøver å legge til hele eller deler av et element utenfor tegnebrettet legges det ikke til. Brukeren vil få en feilmelding nederst i vinduet.

Use Case:	Gjøre justeringer på elementer i tegnebrettet
Beskrivelse:	Bruker velger et element på tegnebrettet ved å klikke på det med rotasjons- eller pekerverktøyet. Hvis rotasjonsverktøyet er valgt, kan bruker rotere elementet ved å klikke og dra i markeringen. Hvis bruker vil skalere elementet, klikkes det på et hjørne, og dras til elementet har oppnådd ønsket størrelse. Flytting skjer ved at brukeren klikker på et element og drar det til ønsket posisjon og slipper museknappen.
Alternativer:	Dersom brukeren gjør forandringer slik at hele eller deler av et element havner utenfor tegnebrettet vil justeringen ikke bli gjort. Brukeren vil få en feilmelding nederst i vinduet.

Use Case:	Endre på høyde/diffusitet/refleksjon for en flate
Beskrivelse:	Når et flateelement er valgt, kan brukeren gjøre innstillinger ved å bruke diverse sliders og inputboks på tab'en «innstillinger» i alternativer-vinduet.
Alternativer:	Dersom brukeren taster inn en høyde som er ulovlig vil endringen ikke bli gjort og brukeren får en feilmelding nederst i vinduet.

Use Case:	Endre på innstillinger på en komponent
Beskrivelse:	Når en komponent er valgt, kan brukeren gjøre innstillinger ved å bruke diverse sliders og inputbokser på tab'en «innstillinger» i alternativervinduet. For en flate gjelder høyde, diffusitet og refleksjon, for en lyskilde: styrke og farge. Begge har høyde.
Alternativer:	Dersom brukeren taster inn ulovlig høyde vil endringen ikke bli gjort og brukeren får en feilmelding nederst i vinduet.

Use Case:	Rendre bilde/film
Beskrivelse:	Bruker ber systemet rendre tegnebrettet ved å trykke på rendreknappen, og vil da få spørsmål om evt. animasjoner skal være med i rendringen og hvilket navn bildet/filmen skal lagres under. Deretter blir bildet/filmen rendret og vist i eget vindu, ved hjelp av Oddplayer 5.0.
Alternativer:	Dersom brukeren ikke får lagret vil det komme en feilmelding.

Use Case:	Lagre/Lagre som tegnebrett
Beskrivelse:	Bruker kan lagre tegnebrettet på fil, ved å trykke på lagreknappen eller velge lagre/lagre som fra filmenyen. Dersom filen eksisterer fra før, vil du ha muligheten til å skrive over denne ved hjelp av lagre. Hvis den ikke eksisterer fra før, vil du få spørsmål om hva du vil kalle filen og hvor du vil plassere denne ved hjelp av en diskutforsker.
Alternativer:	Dersom brukeren ikke får lagret, vil det komme en feilmelding.

Use Case:	Åpne tegnebrett fra fil
Beskrivelse:	Bruker kan åpne et tidligere lagret tegnebrett fra fil ved å bruke åpne-knappen eller åpne på filmenyen. Brukeren vil da få opp en diskutforsker, og kan velge hvilken fil hun vil åpne ved hjelp av museklikk.
Alternativer:	Dersom brukeren skriver et ugyldig filnavn eller en fil som programmet ikke klarer å åpne vil det komme en feilmelding.

Use Case:	Animér lyskilde
Beskrivelse:	Brukeren kan velge en lyskilde, og klikke på animér linje knappen og får da muligheten til å tegne en kurve/linje han/hun vil at lyskilden skal følge. Ved hjelp av animasjon-tab'en i alternativer vinduet kan bruker velge høyden til lyskilden over tid ved å tegne en høydekurve i et høyde/tid diagram.
Alternativer:	Dersom brukeren ikke har valgt en lyskilde vil det komme en feilmelding til brukeren nederst i hjørnet og han får ikke mulighet til å tegne en animasjonsbane. Dersom lyskilden allerede har en animasjon vil brukeren få spørsmål om den allerede eksisterende animasjonen skal slettes.
Use Case:	Få hjelp
Beskrivelse:	Bruker kan ved hjelp av hjelp-menyen få hjelp. Der kan hun/han velge mellom å søke på stikkord, slå opp i stikkordsregister eller lese brukermanual. Bruker kan også høyreklikke på et område i GUIen og velge «hva er dette?» for å få opp et hjelpevindu som forklarer hva man kan gjøre i dette området.
Alternativer:	Dersom brukeren søker på et ord som ikke finnes i brukermanualen vil han få beskjed om «ingen treff»

2.5 Aspekter omkring livssyklus

Det vil ikke bli noe form for vedlikehold av dette programmet ettersom det er et hovedprosjekt som skal avsluttes mai 2004. Når det gjelder flyttbarhet, ser ikke vi dette som et problem, da programmet vil bli levert i en installasjonspakke. Siden det er utviklet i Java™ vil det være plattformuavhengig. Vi kommer sannsynligvis ikke til å erstatte noen av modulene, som tidligere sagt, siden vi ikke kommer til å utføre vedlikehold.

2.6 Krav til ytelse

Kaustikkurvene skal vises i sanntid. Det er ikke satt noe krav til rendretiden, men de skal ha mulighet for å bestemme oppløsning selv og utifra dette få et estimat for hvor lang tid rendringen vil ta.

2.7 Funksjonell spesifikasjon

2.7.1 Operasjonelle systemkrav

2.7.1.1 Hastighet

Vi har valgt å ikke sette noe krav til tiden det skal ta å rendre et bilde, fordi vi ønsker heller at bildet blir av tilstrekkelig kvalitet noe som vi er usikre på hvor lang tid vil ta. Dersom dette tar altfor lang tid, velger vi antagelig å gjøre justeringer med tanke på oppløsningen. Vi har ikke tenkt å legge vekt på hvor lang tid oppdateringen av den eventuelle forhåndsvisningen av 3D-bildet vil ta. Denne forhåndsvisningen vil ha lav prioritet slik at den ikke opptar prosessorkapasitet for resten av programmet. Denne forhåndsvisningen er med andre ord ingen viktig del av applikasjonen vår.

Vi har som tidligere forklart bestemt oss for å programmere i Java. I denne forbindelse må vi ta hensyn til at Java ikke er

spesielt raskt i forhold til andre språk. Noen av grunnene til dette er at Java har innebygd funksjonalitet for behandling av exceptions, noe som er tidkrevende. Java kan også ha trege GUI-komponenter. Den største ulempen med Java med tanke på hastighet er likevel at koden kompileres til bytekode og ikke til god gammeldags maskinkode. Det er jo velkjent for de fleste at et program som blir interpretert ikke har den samme hastigheten som et tilsvarende program i fullkompilert binærkode.

2.7.1.2 Robusthet/pålitelighet

For å forhindre at bruker prøver å rendre en «fysisk» ulovlig 3D verden, sier programmet i fra i det bruker gjør noe galt ved hjelp av feilmeldinger, og ulovlige verdier blir ikke satt. Feilmeldingene vil komme nederst på skjermen, på en statuslinje, eller som eget vindu, hvis feilmeldingen er alvorlig nok.

2.7.1.3 Minnebruk/kapasitet

Vi har ingen formening om antall elementer som kan plasseres på tegnebrettet uten at det vil få konsekvenser for minnebruk, og gjør at programmet henger seg. Dette vil vi imidlertid teste på et senere tidspunkt. Det samme problemet gjelder hvor mange tegnebrett som kan være oppe samtidig.

2.7.2 Funksjonelle krav pr modul i detalj

Tegnebrett

K1	Brukeren skal ha et (eventuelt flere) tegnebrett tilgjengelig.
K1.1	Brukeren skal kunne opprette et nytt tegnebrett.
K1.1.1	Brukeren skal ha mulighet for å velge størrelse på tegnebrettet når det opprettes.
K1.2	Brukeren skal kunne legge til elementer fra tegnebrettet.
K1.3	Brukeren skal kunne velge et element fra tegnebrettet.
K1.3.1	Brukeren skal kunne fjerne valgt element fra tegnebrettet.
K1.3.2	Brukeren skal kunne flytte/rotere/skalere det valgte elementet på tegnebrettet.
K1.4	Brukeren skal ha mulighet for å legge til/fjerne rutenett på tegnebrettet.
K1.4.1	Brukeren skal kunne velge rutestørrelse.
K1.5	Brukeren skal få en oversikt over alle de elementene som finnes på tegnebrettet i en elementliste.
K1.5.1	For hvert av elementene i elementlisten finnes det informasjon om elementet som er tilgjengelig for brukeren.
K1.6	Brukeren skal kunne justere på elementene
K1.6.1	Brukeren skal kunne justere lysstyrke/høyde/farge for lyskilden.
K1.6.2	Brukeren skal kunne justere de ulike sideflatenes egenskaper (diffusitet/absorpsjonsgrad/høyde).
K1.6.3	Brukeren skal kunne bestemme tykkelsen på de ulike flatene.

Filhåndtering

K2	Det skal finnes funksjonalitet for filhåndtering.
K2.1	Brukeren skal ha mulighet for å lagre tegnebrettet på fil. Det skal gis mulighet for å velge filnavn og plassering.
K2.1.1	Brukeren skal ha mulighet for å overskrive en allerede eksisterende fil.
K2.2	Brukeren skal ha mulighet for å hente fram en fil på disk.
K2.3	Brukeren skal kunne lagre ferdig rendret film/bilde.

Tegneverktøy

K3	Brukeren skal ha mulighet for å legge til lyskilder og ulike speilende flater på tegnebrettet.
K3.1	Brukeren skal ha mulighet for å velge ulike tegneverktøy.
K3.1.1	Brukeren skal kunne legge til parabelformede speilende flater.
K3.1.2	Brukeren skal kunne legge til ellipseformede speilende flater.
K3.1.2.1	Brukeren skal kunne taste inn høyde, bredde og eventuell åpningsvinkel.
K3.1.3	Brukeren skal kunne legge til rettlinjeformede speilende flater.
K3.1.4	Brukeren skal kunne legge til kurvelinjeformede speilende flater.
K3.1.5	Hvis tid: Brukeren skal kunne legge til hyperbelformede flater.
K3.2	Brukeren skal ha mulighet for å velge lyskilde.
K3.2.1	Brukeren skal kunne legge til en punktlyskilde.
K3.2.2	Brukeren skal kunne legge til andre typer lyskilder. (Hvis tid)

Hjelpemodul

K4	Det skal finnes en hjelpefunksjon.
K4.1	Brukeren skal ha mulighet for å lese en brukermanual.
K4.1.1	Brukeren skal kunne søke etter stikkord i denne brukermanualen som hun/han trenger hjelp til.
K4.1.2	Brukeren skal kunne slå opp i stikkordsregister.

2D-visning av kaustikkurver

K5	Det skal genereres «kaustikkurver» som blir lagt til på tegnebrettet.
K5.1	Brukeren skal ha mulighet for å vise/skjule kaustikkurvene på tegnebrettet.
K5.2	«Kaustikkurvene» skal oppdateres hver gang en lyskilde/speilende flate blir lagt til/fjernet/endret.

GUI

K6	Det skal være et vindusbasert brukergrensesnitt.
K6.1	GUIen skal være brukervennlig.
K6.1.1	Knappeikoner skal være intuitive.
K6.1.2	Alle knapper skal ha ToolTip-tekst.
K6.1.3	Brukeren skal ved høyreklikk få opp et søk i brukermanualen som hjelp.
K6.2	Brukergrensesnittet skal være et MDI (Multi Document Interface).
K6.2.1	De delene som logisk hører sammen skal være gruppert i samme vindu.

oDDplaYer

K7	oDDplaYer skal vise film/bilde generert av programmet.

Rendre-modul

K9	Brukeren skal ha mulighet for å rendre en «3D-versjon» av tegnebrettet.
K9.1	Brukeren skal ha mulighet for å rendere til en .mpg dersom brukeren har laget en/flere animasjon(er).
K9.1.1	Brukeren skal kunne slå av animasjon før rendering uten at animasjonen fjernes fra tegnebrettet.
K9.1.2	Brukeren skal ha mulighet for å rendre til en .jpg.

Animasjon

K10	Brukeren skal kunne lage en bane for et speil/lyskilde.
K10.1	Banen skal genereres og legges til ved hjelp en strek med tydelig farge på tegnebrettet.
K10.2	Hvis tid: Det skal være mulig å justere høyden over tid for lyskilden.
K10.3	Hvis tid: Brukeren skal kunne trykke på en play-knapp for å se denne animasjonen i 2D.

Måloversikt (arbeidstegning)

K11	Brukeren skal kunne få en oversikt over elementene og deres plassering, størrelse og innstillinger.
K11.1	Brukeren skal kunne få en utskrift av tegnebrettet hvor elementene er navngitt.
K11.2	Brukeren skal kunne få en fil som inneholder informasjon om størrelse, innstillinger, plassering av de navngitte elementene.

Hvis tid: Undo/Redo

K13	Det skal være mulighet for å angre/gjøre om en eller flere operasjoner på tegnebrettet.
K13.1	Maksimalt antall operasjoner man kan angre/gjøre om er ?

Hvis tid: Kopier/Klipp ut/Lim inn

K14	Det skal være mulighet for å kopiere/klippe ut/lime inn et element fra/til tegnebrettet.

2.8 Begrensninger (HW/SW)

Maskinene bør ha mulighet til å kjøre Java uten større problemer.

2.9 Dokumentasjon

All dokumentasjon av kode vil være JavaDoc. I tillegg vil rapporten dokumentere gruppas arbeid og det endelige programmet.

2.10 Konfigurasjons- og versjonsstyring

Arbeidet med å holde orden på de ulike versjonene underveis har vi tenkt til å løse ved hjelp av noen selvlagde regler for konfigurasjonsstyring. Det er viktig at alle som foretar endringer i koden/legger til noe nytt følger disse reglene nøye. Reglene som vi har kommet fram til er som følger:

- Øverst i hver fil skal datoen for siste endring fremkomme, samt en kort forklaring på hva som er endret.
- Det opprettes en ftp hvor alle filer skal ligge.

- På starten av en arbeidsdag opprettes en ny mappe med navn lik dagens dato, og alt fra "forrige" mappe blir overført til denne. Alle endringer som blir gjort denne dagen legges til i denne mappen.
- I hver mappe skal det ligge en .txt-fil som inneholder hvilke filer som er i bruk og av hvem. Når man er ferdig med å endre på en fil, eller det er lenge til man skal endre den igjen, skal man oppdatere filen i mappen med dagens dato, og fjerne filen fra .txt-filen.
- Backup skal foretas minst en gang i uken, gjerne oftere hvis man har gjort større endringer. Backup skal lagres på hjemmeområdet til backupansvarlig, evt. sendes på mail til backupansvarlig. Backup skjer på fredag etter endt arbeidsdag.

2.11 Krav til utvidelser

Det skal legges til rette for senere utvidelser dersom noen ønsker å gjøre det. Dette skal gjøres i form av objektorientert kode og god dokumentasjon. I kravspesifikasjonen har vi satt noen krav merket «hvis tid». Dette kan være aktuelle utvidelser dersom vi ikke rekker det. I tillegg kan det være aktuelt med flere tegneverktøy, animasjon av flater og større frihet i animasjonene. Ytelsen vil antagelig også ha et stort forbedringspotensial.

2.12 Aspekter omkring installasjon

2.12.1 Opplæring

Vi satser på en god brukermanual og et intuitivt utformet program, og håper at det holder i første omgang.

2.13 Utgivelser underveis

Det vil ikke foreligge noen utgivelser underveis både på grunn av at systemutviklingsmodellen fossefall nekter oss dette og på grunn av prosjektets korte levetid.

3 Design

Formålet med designdokumentasjonen er å gi en oversikt over hvordan kravene som er satt i kravspesifiserings-prosessen skal realiseres og implementeres. Vi brukte designfasen til å designe GUI og diskusjon omkring klasser, funksjoner og hvordan de viktigste klassene skulle henge sammen og kommunisere. Forslag til klassediagram over den store helheten ble også utarbeidet.

3.1 Det grafiske brukergrensesnittet

Ut i fra prosjektbeskrivelsen, kravspesifikasjonen og samtaler veileder og oppdragsgiver kom vi fram til et GUI-forslag. Vi har prøvd å lage GUI som brukerne kan kjenne seg igjen i fra blant annet Adobe Illustrator og andre lignende programmer. Studentene ved Kunsthøgskolen har en viss kjennskap til Illustrator. Dette var avgjørende for valget om grafisk brukergrensesnitt. Ellers har vi forsøkt å legge vekt på at skjermbildene skal være intuitive slik at brukerne uten store problemer klarer å bruke programmet på en fornuftig måte.

Programmet består av følgende vinduer:

• Hovedvindu
• Elementvindu med tittelen «Infosenteret»
• Innstillingsvindu for flater og lyskilder
• Verktøyvindu
• Tegnebrett

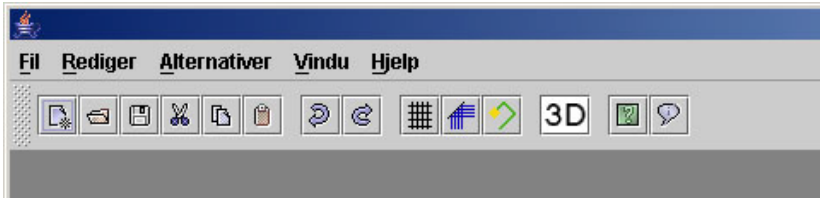
3.1.1 Hovedvindu

Dette vinduet inneholder en knappelinje med intuitive knapper. På knappene som har de mest velkjente funksjonene har vi brukt ikoner fra Sun som passer bra sammen med layout ellers i brukergrensesnittet. Noen av ikonene på knappene er designet selv, da det ikke finnes allerede eksisterende knapper som symboliserer funksjonalitet som «Kaustikker på/av» og «Rutenett på/av». Knappene har også en forklarende tekst, såkalt ToolTip-tekst, slik at brukeren får et hint om funksjonaliteten dersom musa holdes over den aktuelle knappen.

Knappene er:

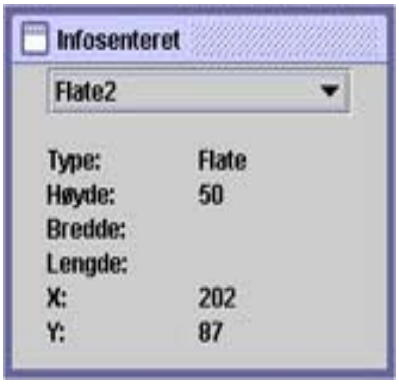
• Nytt dokument
• Åpne dokument
• Lagre dokument
• Klipp ut element
• Kopier element
• Lim inn element
• Toggle rutenett av/på
• Toggle kaustikkurver av/på
• Toggle animasjonslinjer av/på
• Rendre bilde
• Hjelp
• Om

Hovedvinduet har også en menylinje hvor med den samme funksjonaliteten som knappene. Menylinjen kan nås ved å bruke tastaturet også. Eksempelvis kan Fil nås ved Alt + F slik understrekingen av F viser.



Figur 2 Hovedvinduets meny- og knappelinje

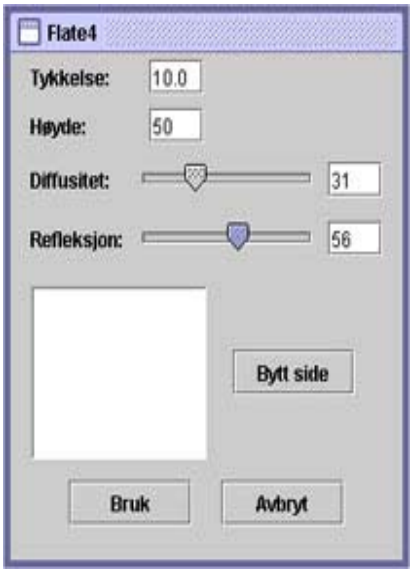
3.1.2 Elementvindu



Figur 3 Elementvindu

I elementvinduet har brukeren tilgang til en komponent-rullgardin, hvor han lett kan markere et element ved å velge det i fra listen. Høyden til flaten eller lyskilden blir vist, og dessuten om det er en flate eller en lyskilde som er markert. Elementvinduet viser også til en hver tid hvor på det gjeldende tegnebrettet musepekeren er, ved hjelp av en x- og y-koordinat. Brukeren kan ikke endre innstillinger ved hjelp av elmenentvinduet. Dette er kun et informasjonssted for de ulike komponentene på tegnebrettet.

3.1.3 Innstillingsvinduer



Figur 4 Flateinnstillinger

I innstillingsvinduene får brukeren informasjon om den markerte flaten eller lyskilden, og også mulighet til å forandre på en del av parametrene.

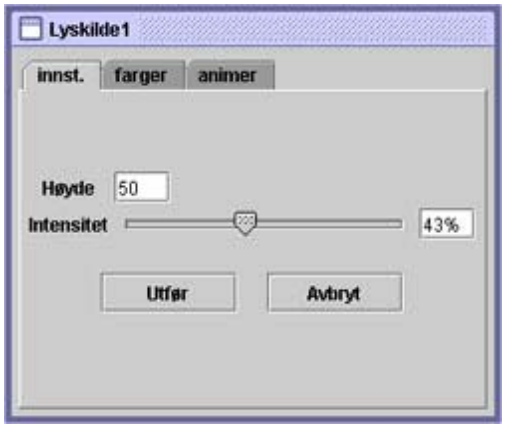
Flateinnstillinger

Når brukeren markerer en komponent som er en flate vil innstillingsvinduet vise

innstillinger som kan gjøres på den nevnte flaten. Disse innstillingene er: tykkelse, høyde, diffusitet og refleksjon. Diffusitet og refleksjon kan være forskjellig for de ulike sidene. Disse parameterne vil først og fremst gi utslag når tegnebrettet rendres til tre dimensjoner.

Lysinnstillinger

Når brukeren markerer en komponent som også er en lyskilde, vil innstillingsvinduet vise diverse innstillinger for denne lyskilden. Vinduet for en lyskilde skiller seg fra vinduet for flater. For en lyskilde vil andre parametere være interessante. Vinduet vil nå ha tre tabs (valg). Den første tab'en gir mulighet for å stille høyde og intensitet. Nummer to gir mulighet for å skifte lysets farge. Den siste gjør at brukeren kan bestemme høyde per tid for en animert lyskilde.

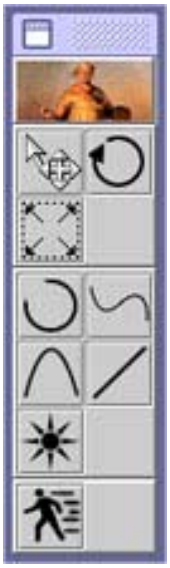


Figur 5 Lysinnstillinger

3.1.4 Verktøyvindu

Dette vinduet fungerer som et verktøyskrin for brukeren. Her kan brukeren velge og vrake mellom disse verktøyene:

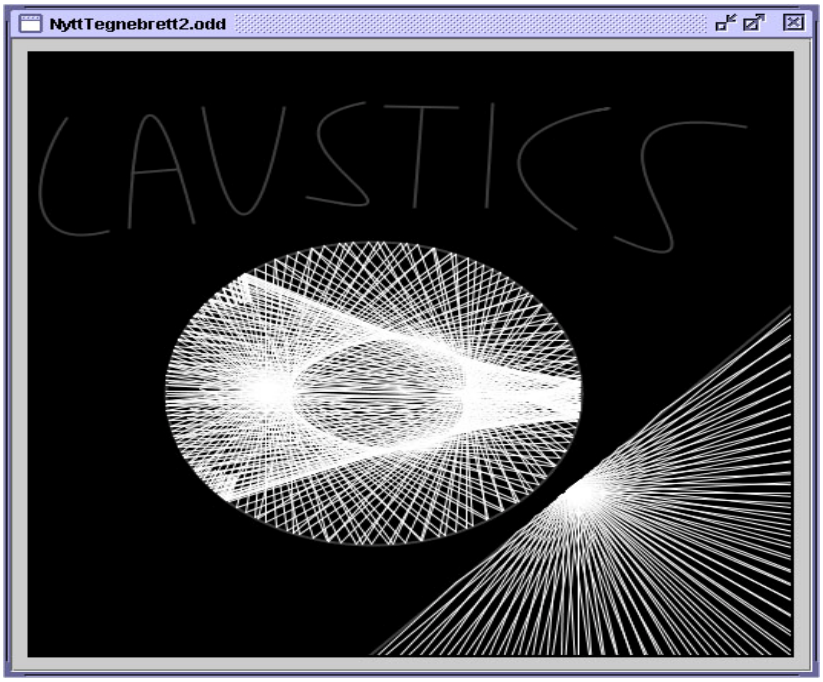
- Marker/flytt – verktøy
- Roteringsverktøy
- Skaleringsverktøy
- Ellipseverktøy
- Bezierkurveverktøy
- Parabelverktøy
- Linjeverktøy
- Punktlyskildeverktøy
- Animér lyskilde-verktøy



Figur 6 Verktøyvindu

3.1.5 Tegnebrett

Tegnebrettet inneholder et tegneområde hvor brukeren kan plassere de ulike figurene og lyskildene. I tegneområdet kan brukeren velge om lysstrålene ut fra en lyskilde skal vises slik som i eksempelet under. Brukeren har også valgmulighet for å legge rutenett på tegneområdet og animasjon av lyskilde. Tegnebrettet kan også skrives ut sammen med et tilhørende ark med informasjon over alle elementene.



Figur 7 Tegnebrett

3.1.6 Java «look and feel»

Java har støtte for flere forskjellige såkalte «look and feel» biblioteker, som for eksempel Windows, Motif og MacOS. Men selv om Windows sin «look and feel» hadde vært mest velkjent for våre brukere, så valgte vi standard Java «look and feel». Det er to grunner til at vi gjorde dette; den første er at når vi bygger opp GUIen til et vindu, benytter vi oss av *GridBagLayout*. Det som er tingen med *GridBagLayout* er at den ikke liker alle typer «look and feel» helt uten videre, men

er nødvendig for å kunne være såpass frie som vi har vært i utformingen av vinduene. Den andre grunnen er at vi ikke fant noen gode standardikoner i Windows-stil, og en kræsj mellom layout ellers og ikoner kan vi ikke gå med på.

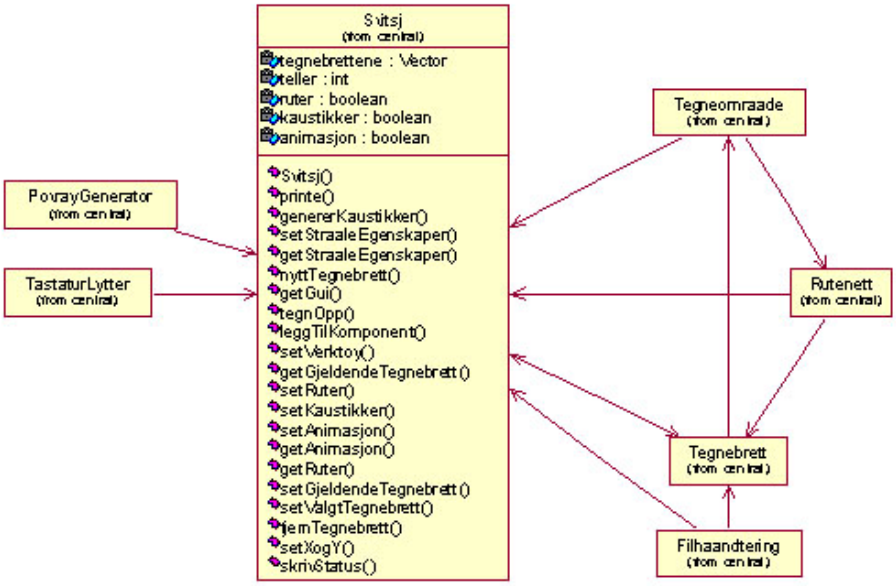
3.2 Systemdesign

Vi lagde tidlig i designfasen klassediagrammer over systemet slik vi mente det skulle se ut. Dette har forandret seg noe underveis, men helheten er stort sett den samme. Klassediagrammene som følger i dette avsnittet beskriver den endelige løsningen. Før vi startet kodingen ble vi enige om at det var greit å ha en oversikt over de viktigste klassene og deres funksjoner. Denne er å finne i vedlegg I til rapporten. Kommentarer og diskusjon omkring endringer og nye avgjørelser som ble tatt underveis finnes i delen med drøftinger og konklusjoner. Avsnitt 3.2.1 tar for seg hvordan systemdesignet ble til slutt.

3.2.1 Klassediagrammer med forklaringer

Generelt sett er det en metode eller en klasse som har ansvaret for en funksjon i programmet vårt. Denne klassen eller metoden delegerer deretter ansvaret ned til komponentene. Som for eksempel ved å skrive til fil, hvor klassen *Filhaandtering* går igjennom komponentvectoren og ber hver enkelt av komponentene å skrive seg selv til fil, ved hjelp av *Komponents tilFil(BufferedWriter bw)*.

Pakken koding.central

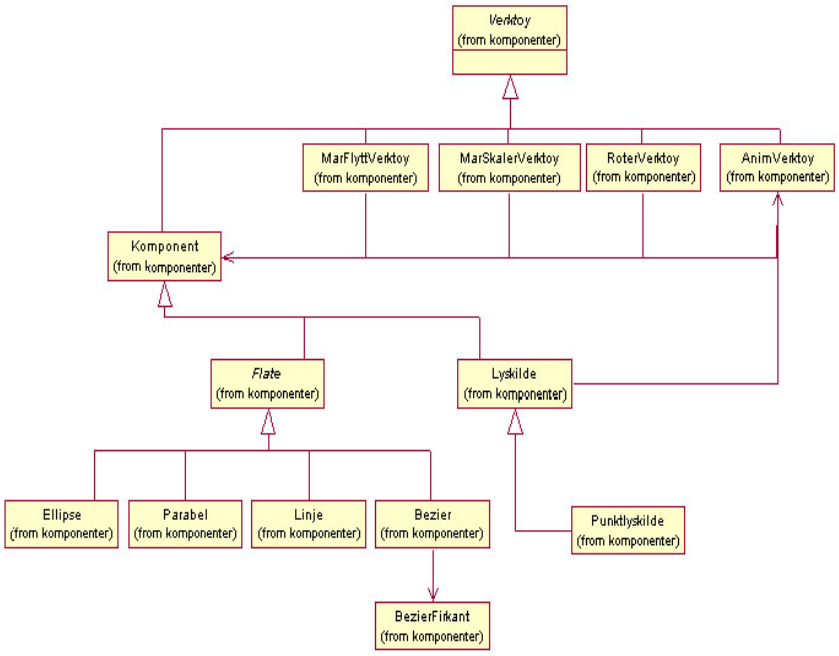


Figur 8 Klassediagram over pakken central

Dette klassediagrammet viser strukturen for pakken *koding.central*. Den viktigste klassen her er *Svitsj*. Denne klassen skal være selve kjernen og fungere som et sentrum i programstrukturen. Svitsjen skal holde orden på de ulike tegnebrettene som er oppe til enhver tid ved hjelp av en *Vector* med alle tegnebrettene. Oppdatering av denne vectoren ved fjerning eller dersom et nytt tegnebrett legges til skal også administreres av svitsjen. Den skal til enhver tid

sørge for å oppdatere hvilket tegnebrett som er det gjeldende. Svitsjen skal også ha ansvaret for å oppdatere tegnebrettet ved hjelp av en *tegnOpp()*-metoden som skal kalle alle komponentene på tegnebrettet sin egen *tegnOpp()*. I samme metode skal også rutenett, lysstråler og animasjon eventuelt tegnes. Om disse skal tegnes skal sjekkes ved hjelp av boolean for ruter, kaustikker og animasjon. Disse verdiene skal settes ved hjelp av knapper i knappelinje eller ved hjelp av menylinjen. På samme måte skal svitsjen også sørge for riktig utskrift til printer.

Pakken koding.komponenter



Figur 9 Klassediagram over pakken komponenter

Ut fra klassediagrammet for pakken *koding.komponenter* ser vi at *Verktoy* er baseklassen. *Verktoy*-klassen er superklassen for alle de andre klassene og implementerer interfacene *MouseListener* og *MouseMotionListener*. Vi har valgt å bygge opp

strukturen slik at hver komponent er et tegneverktøy i seg selv. *Komponent*-klassen har to subklasser, *Flate* og *Lyskilde*. Disse klassene har igjen subklasser; *Flate* har subklassene *Ellipse*, *Parabel*, *Linje* og *Bezier*. *Lyskilde* har subklassen *Punktlyskilde*. I klassediagrammet over har vi utelatt variable og funksjoner av den grunn at det ser mer oversiktlig ut og at dette vil bli omtalt nærmere i delen om koding og implementasjon.

På grunn av klassehierarkiet har vi også benyttet oss av polymorfisme. Polymorfisme brukes når vi har flere klasser i et klassehierarki som alle skal ha samme metode, men med ulik implementasjon av den samme metoden. Når programmet vårt jobber med objekter fra dette klassehierarkiet vil det benytte seg av den riktige metoden ut fra hva slags type objekt det er. Eksempler på slike metoder for komponenthierarkiet i vårt program vil bli metoder for å tegne seg opp, tegne seg til utskrift, skrive seg til fil etc. Alle komponenter skal ha disse metodene, men implementasjonen for de ulike komponentene skal være forskjellig. Polymorfisme gjør det også enklere å utvide systemet med flere klasser. Eksempelvis skal det bli greit å legge til flere typer komponenter i programmet.

De fire klassene *MarSkalVerktoy*, *MarFlyttVerktoy*, *RoterVerktoy* og *AnimVerktoy* skiller seg fra de andre verktøyene ved at disse utfører operasjoner på allerede eksisterende komponenter. På tegnebrettet vil brukeren ha mulighet for å markere en komponent ved hjelp av et verktøy og deretter skalere det, flytte det, rotere det eller lage en animasjon.

3.3 Verktøy i designprosessen

For å lage skisser av brukergrensesnittet lagde vi aller først helt enkle skisser ved hjelp av penn og papir. Deretter brukte vi Adobe Photoshop for å lage noen GUI-forslag slik at oppdragsgiver Harald Anthonsen kunne se på disse og gi en kommentar. Vinduene under avsnittet om det grafiske brukergrensesnittet er hentet fra programmet og identiske med vinduene slik de er ved prosjektslutt.

Til systemdesign brukte vi også for det meste penn og papir. Etter at vi følte oss fornøyd med oppbyggingen av systemet,

skrev vi ned et forslag ved hjelp av Word og tabeller, dette er som tidligere nevnt vedlagt etter rapporten (KlasserogFunksjoner.doc, vedlegg I). For å lage klassediagrammer har vi brukt Rational Rose Enterprise Edition.

4 Implementering/koding

4.1 Utviklingsmiljøet

4.1.1 Java

Vi valgte å utvikle programmet vårt i Java som tidligere nevnt. En av hovedgrunnene til at vi valgte Java i stedet for for eksempel C++, er at dette er et språk vi har blitt godt kjent med igjennom faget programutvikling, og delvis også kompilatorteknikk. Noen av fordelene med Java er for eksempel at programmering av GUI går som en lek, og vi vet hva vi får; layouten på vinduene, knappene, rullgardinene etc. ser proff ut (i motsetning til de mange forskjellige GUI-løsningene i C++). Vi er også bedre kjent med objektstruktur og hvordan objektorientering fungerer i Java, siden det er det eneste programmeringsspråket vi har produsert noe med en viss størrelse i. I tillegg har Java mange ferdige klasser som vi er kjent med funksjonaliteten i, og som er til god hjelp, slik som for eksempel *java.util.Vector* og *java.awt.geom.AffineTransform*. Java har også store fordeler når det gjelder dokumentasjon, både når det gjelder vår dokumentasjon av kode (JavaDoc), og Suns API dokumentasjon. Garbage Collection systemet til Java er også en fordel for oss som er litt late av oss; ingen svevende objekter her i gården. Selv om vi måtte sette oss inn i

Java2D, var ikke dette noe stort problem – takket være den nevnte API- dokumentasjonen. Vi tror at hvis vi skulle sette oss inn i en tilsvarende funksjonalitet i C++ ville det kostet oss mange timer mer med lærebrillene på.

Men fikk vi alle disse fordelene helt uten at vi gikk glipp av noe? Vi tror den største ulempen ligger i en hastighetsreduksjon, siden Java absolutt vil være plattformuavhengig og har løst dette ved å la koden kompileres til bytekode som interpreteres av en Java Virtual Machine. Det sier seg selv at jo flere ledd instruksjonene må igjennom før den når prosessoren, jo tregere går det. Det at Java er plattformuavhengig er ingen fordel for oss på noen måte, da både vi og studentene ved Kunsthøgskolen i Oslo sverger til gode gamle Windows. Ikke at det er en stor ulempe, men det bør nevnes for å få litt balanse i Java fordel/ulempe-oversikten vår; for å få kjørt programmer utviklet i Java, må brukeren ha installert den nevnte JVM (og det er jo litt mer jobb enn å ikke installere det).

4.1.2 POV-ray

Til å skape bilder som simulerer vår virkelige tredimensjonale verden, bruker vi en raytracer som går under navnet POV-ray (Presistence of Vision Raytracer). Egentlig hadde vi planer om å benytte oss av Java3D til å rendre slike bilder, men fant til slutt ut at bildene da ville hatt altfor lav kvalitet, da Java3D ikke er veldig avansert når det gjelder refleksjoner av lys – og da er mye av poenget borte med 3D-rendringen. Det at vi valgte akkurat POV-ray var mer eller mindre helt tilfeldig, kan nesten skylde på google.com, som til stadig returnerte linker til sider som hadde noe med POV-ray å gjøre da vi søkte etter informasjon om raytracing (det tyder jo egentlig på at den er velkjent i alle fall).

POV-ray er en freeware raytracer, og vi fant fort ut at det var veldig mye ressurser på nettet som omhandlet akkurat den. I tillegg til å ha en ganske bra dokumentasjon, så er den grei å sette seg inn i for personer som er relativt blanke når det gjelder data og tre dimensjoner. POV-ray har også en veldig flittig brukerskare som via nyhetsgruppen news.povray.org kunne hjelpe med diverse problemer som oppstod underveis.

Raytracing går ut på, som navnet insinuerer, å følge en lysstråle. La oss tenke oss at vi har en lyskilde, noen objekter som mer eller mindre reflekterer lyset og et øye som skal oppfatte lyset. En raytracer vil da følge strålene som kommer ut i fra lyskilden, hver og en etter tur, og se om de treffer et av objektene, hvis så, skal strålen reflekteres? Hvis objektet har en perfekt speilende overflate vil lysstrålen da fortsette fra objektet med en annen retning før den kanskje treffer øyet (merk kanskje). Det kan hende den i stedet treffer det andre objektet, som er delvis gjennomsiktig, altså vil noe av lysstrålen gå igjennom objektet, mens andre deler av lysstrålen reflekteres. Så må det igjen sjekkes om noe av dette lyset treffer øyet. Denne måten kalles "forward raytracing" (de lærde strides om hva som er "forward raytracing" og hva som er "backwards raytracing", men så lenge vi er konsistente så er det god stemning). Ulempen med denne måten er at over 99% av strålene og refleksjonene aldri treffer øyet, noe som betyr masse prosessering helt uten grunn. En mer effektiv måte da er det som kalles "backwards raytracing"; det vil si at raytraceren starter i øyet og følger lysstrålene mot fartsretningen.

4.2 Koding

4.2.1 Kodelayout

Vi gikk over all koden med emacs og trykket på tabulator for hver linje for å få en standard i innrykk, ellers er alle startkrøllparanteser på samme linje som sin tilhørende if, else, while, class etc. og alle sluttkrøllparanteser er på egen linje(unntak for catch). Kommentarer følger JavaDoc-standard.

Eksempel på kode:

```
package koding.GUI;

import koding.central.Tegnebrett;
import javax.swing.JMenuItem;

/**
 *Klassen brukes for aa legge til menyvalg under menyen Vindu.
 *Alle tegnebrettene som er opprettet vil finnes under Vindu-menyen.
 *@since 1.9
 *@author <a href="mailto:kaustikker@hotmail.com">Maria, Mads & Lars Petter</a>
 */
public class Menyting extends JMenuItem{

    private Tegnebrett tegnebrett;
    /**
     *Tom constructor.
     */
    public Menyting(){}

    /**
     *Constructor som lager et menyvalg (JMenuItem) som er navnet paa tegnebrettet.
     *Tar vare paa referansen til tegnebrettet.
     *@since 1.9
     */
    public Menyting(Tegnebrett tb){
                super(tb.toString());
                tegnebrett = tb;
    }

    /**
     *Metoden returnerer et tegnebrett.
     *@return tegnebrettet.
     *@since 1.9
     */
    public Tegnebrett getTegnebrett(){
                return tegnebrett;
    }
}
```

Figur 10 Eksempel på kode

4.2.2 Komponenter, tegnebrett og tegneverktøy.

Et tegnebrett er en instans av klassen *Tegnebrett*, og inneholder en *java.util.Vector* med referanser til alle komponenter som tilhører tegnebrettet. I tillegg inneholder den et tegneområde, hvor komponentene grafisk blir vist til brukeren, samt funksjoner for å legge til å fjerne komponenter fra komponentvectoren. Tegneområdet viser komponentene ved å gå igjennom komponentvectoren og si til hver av komponentene at de skal tegne seg opp på meg.

Alle komponenter er sine egne tegneverktøy, dette ble implementert ved at klassen *Komponent* som er baseklassen til alle flater og lyskilder arver fra klassen *Verktoy*, som egentlig er en *MouseMotionAdapter* og *MouseAdapter* i ett. Det vil si at klasser som arver fra *Verktoy* har tilgang til alle typer musehendelser, sånn som for eksempel når musen blir flyttet, en museknapp blir trykket, eller om knappen er nede

når musen flyttes. Når bruker velger et tegneverktøy ved å klikke på en av knappene i verktøyvinduet, blir dermed en muselytter lagt til på tegnebrettets tegneområde. Nå kan bruker forme en figur ved å trykke ned venstre mustast og bevege på musen. Komponenten vil bli lagt til i komponentvectoren i det bruker trykker mustasten.

Alle komponenter har en referanse *figur* av formen *java.geom.Shape,* som er en superklasse til alle Java2D-figurer, og det er figur som blir tegnet når *Komponents tegnOpp()* blir kalt. I tillegg har Komponent en *AffineTransform* ved navn *samleMatrise*, som er en 3x3 matrise som tar vare på alle transformasjoner som er skjedd med en komponent siden den ble lagt til tegnebrettet.

4.2.3 Markering, rotasjon, skalering og flytting av komponenter

En komponent som blir lagt til på et tegnebrett, blir straks markert. Et tegnebrett kan bare ha én eller ingen komponenter markert, dette er løst ved at hvert tegnebrett har en referanse til en instans av klassen *Komponent* som kalles *markert*. Det er to måter å markere en komponent på: enten ved å velge denne komponenten i rullgardinen i elementvinduet, eller å velge enten marker/skaler-verktøyet eller marker/flytt-verktøyet. Når en komponent blir valgt i elementvinduet, vil *markert* bli satt til referansen til denne komponenten. Når bruker benytter seg av en av de to verktøyene som har muligheten til å markere derimot, vil det være en muselytter på tegnebrettet som tar vare på x og y koordinaten til det punktet hvor musepekeren er i det brukeren trykker på venstre musetast. Verktøyet benytter seg så av *Tegnebretts finnTruffet(int x, int y)*. Denne funksjonen oppretter et *BufferedImage*, som fungerer på mange måter som et bitmap-bilde (altså en kjempearray med fargeverdier for hver piksel i bildet), og tegner opp en og en komponent på dette. For hver komponent som er tegnet på bildet, sjekkes det om det er en fargeverdi i et omkringliggende område rundt den pikselen som har innkommende x og y koordinat. Hvis den har en verdi, så har funksjonen funnet en komponent som skal markeres. Hvis funksjonen går igjennom hele komponentvectoren og ikke finner noen som tegner på den pikselen, blir *markert* satt til *null*. Det vil med andre ord si at man kan avmarkere komponenter ved å

trykke et sted på tegnebrettet hvor det ikke er noen komponenter.

Etter at en komponent er lagt til tegnebrettet kan den transformeres på tre forskjellige måter: den kan roteres, skaleres og flyttes. For å gjøre dette brukes *RoterVerktoy*, *MarSkalVerktoy* og *MarFlyttVerktoy*. For å transformere en komponent må den være markert. For hvert av verktøyene lages det en transformasjonsmatrise. Denne matrisen sendes med til komponentens *transformer(AffineTransform at)*-metode. Her venstremultipliseres transformasjonen til samlematrisen for komponenten og komponentens *figur* oppdateres med samlematrisen. Det gjøres ved hjelp av *AffineTransform* sin *createTransformedShape(Shape s)*.

Translasjonsdelen av *MarFlyttVerktoy* fungerer på den måten at det tas vare på punktet der brukeren klikker på musa og tar vare på avstand i x- og y-retning til der brukeren slapp museknappen. Dette gjøres om til en translasjonsmatrise.

Dersom brukeren markerer en komponent ved hjelp av *MarSkalVerktoy* vil komponenten få en ramme med åtte små bokser, en boks i hvert hjørne og en på hver side. Brukeren må treffe en av disse skaleringsboksene for å kunne skalere komponenten. Skaleringsdelen av *MarSkalVerktoy* fungerer på den måten at det sjekkes først om brukeren har truffet en skaleringsboks og hvis treff sjekkes så hvilken av de åtte skaleringsboksene som er truffet. Dersom brukeren treffer en skaleringsboks i et hjørne, skal figuren skaleres både i x- og y-retning. Punktet i motsatt hjørne holdes fast (komponenten skaleres om dette punktet). Deretter beregnes den relative avstanden som musa er flyttet og en skaleringsmatrise blir laget. Dersom brukeren treffer en av boksene midt på en side skal figuren derimot kun skaleres i en retning og om det motsatte punktet for skaleringsboksen. Når *MarSkalVerktoy* blir brukt på en komponent av klassen *Bezier*, vil ikke brukeren ha mulighet til å skalere, men *Bezier*-objektet selv vil bli satt som verktøy, og man kan flytte på kontrollpunktene til kurven.

RoterVerktoy brukes kun til å rotere en komponent. Komponenten må være markert for å bruke dette verktøyet. Rotasjon av lyskilde er ikke mulig. Først finnes midtpunktet til en komponent ved hjelp av *finnSentrum()*-metoden i *Komponent*. Rotasjonen skal skje om dette midtpunktet. Når

brukeren drar musepekeren i tegnebrettet regnes rotasjonsvinkelen kontinuerlig ut og figuren roteres.

4.2.4 Lysstråler og kaustikkurver

Klassen som administrerer utregning og opptegning av lysstråler og strålegang er *Kaustikker*. Nærmere bestemt er det metoden *genererKaustikker(Graphics g)* som tar seg av dette. Først går den igjennom komponentvectoren og ber hver lyskilde returnere lysstråler som skal legges i en lysstrålekø. Dette gjøres ved å kalle *Lyskilde*s *generer*S*traaler()* som returnerer en *Vector* med lysstråler i form av *ParamLinje*-objekter.

Deretter går metoden igjennom denne lysstrålekøen og sjekker for hver lysstråle om den treffer en komponent, og finner ut hvilken av komponentene som har treffpunktet som er nærmest lyskilden (starten på lysstrålen egentlig). *Komponent*s *treff(ParamLinje pl)*, tar seg av utregning av eventuelt treffpunkt, og returnerer dette i form av et *java.awt.geom.Point2D*-objekt. Når det nærmeste treffpunktet er funnet, tegnes det opp en linje mellom starten på lysstrålen og treffpunktet og den komponenten som ble truffet sin *duTraff(ParamLinje pl, Point2D treffpunkt, double avstand)*, blir kalt. Denne metoden returnerer en *Vector* med en ny ParamLinje som tilsvarer en reflektert lysstråle, og eventuelt en ParamLinje som tilsvarer lysstrålen som fortsetter etter slagskyggen. Den *Vector*en som blir returnert legges til i lysstrålekøen igjen. Hvis det ikke returneres noe treffpunkt for en lysstråle skal lysstrålen bare tegnes opp. Slik fortsetter funksjonen helt til lysstrålekøen er tom.

4.2.5 Filhåndtering

Brukeren har mulighet for å lagre det gjeldende tegnebrettet til fil eller åpne et gammelt. Ved ønske om å lagre til fil opprettes et nytt *Filhaandtering*-objekt og dets *skrivTilFil(Tegnebrett tb)*-metode vil bli kalt. En *java.awt.FileDialog* vil vises og brukeren kan velge hvor filen skal lagres og med hvilket filnavn. Så opprettes en *java.io.BufferedWriter*, og metoden går igjennom komponentvectoren og kaller hver komponents

tilFil(BufferedWriter bw). Komponentene har dermed selv ansvaret for å skrive seg riktig til fil, hver komponent bruker en linje i den lagrede filen. Filene er rene tekstfiler.

Hvis bruker ønsker å åpne et allerede lagret tegnebrett, opprettes et nytt *Filhaandtering*-objekt, og dets *lesFraFil()*-metode vil bli kalt. En og en linje leses inn og legges i en *java.util.StringTokenizer*. Det første tokenet vil være en *String* som entydig bestemmer hvilken subklasse av *Komponent* som skal opprettes. De variable i den nye komponenten blir satt ved at man sender med den tidligere nevnte *StringTokenizer*en til constructoren. Komponenten får dermed ansvaret for å lese seg fra fil på tilsvarende måte som den skrev seg til fil.

4.2.6 3D-rendring og POV-ray-kodegenerator

Tegneområdet rendres til 3D ved hjelp av klassen *PovrayGenerator*. Denne klassen sørger for å lage en fil, *Temp.pov*, med POV-ray-kode. Dette gjøres på lignende måte som når tegnebrettet skrives til en fil ved hjelp av et *BufferedWriter*-objekt. I starten av POV-ray-filen skrives informasjon om kameraplassering, globale innstillinger og kode for tegnebrettet i seg selv, i form av et plan. Deretter skrives alle komponentene til fil ved hjelp av komponentenes *tilPovray(BufferedWriter bw)*-metode. Komponentene har dermed selv ansvaret for å skrive seg riktig til POV-ray-filen.

Start av POV-ray skjer ved at det opprettes en ny prosess, ved hjelp av klassen *Runtime*. POV-ray viser selv det ferdig rendrede bildet.

4.3 Installasjon

Vi benyttet oss Nullsoft scriptable install system, en freeware applikasjon som man enkelt kan lage installasjonsprogram med. Installasjonsprogrammet har ansvaret for å installere både POV-ray, Java Runtime Environment og programmet vårt.

5 Diskusjon av resultater

5.1 Diskusjon rundt kravspek

Vi har nådd de fleste målene som vi satte oss med dette prosjektet. Noen avvik fra kravspesifikasjonen har vi. Med tanke på hva som mangler i programmet sett ut fra kravspesifikasjonen gir vi en kommentar til hver modul slik de er beskrevet i kapitlet om kravspesifikasjonsprosessen (avsnitt 2.7.2).

Tegnebrettmodulen tilfredsstiller i all helhet kravspesifikasjonen. Filhåndteringsmodulen mangler funksjonalitet for å lagre ferdig rendret bilde/film. Brukeren har mulighet for å skrive et tegnebrett til fil og etterpå hente det fram igjen. Bildet må i så fall rendres på nytt. Funksjonalitet for film er ikke lagt inn. Dette er kommentert i forbindelse med animasjonsmodulen.

Alle kravene for tegneverktøy er også tilfredsstilt, bortsett fra to krav som vi satte betingelsen «hvis tid» på i starten fordi vi mente at dette ikke var så viktig i første omgang. Oppdragsgiver hadde mange ønsker så vi valgte å føre opp noen av disse som «hvis tid». Kravene som ikke er oppfylt i tegneverktøymodulen er henholdsvis kravet om at brukeren

skal kunne legge til hyperbelformede flater og andre typer lyskilder enn punktlyskilden.

Når det gjelder funksjonalitet for hjelp så er den noe mangelfull i forhold til de kravene som ble satt. Hvis brukeren trykker på knappen for hjelp i programmet vil han få et vindu med alfabetiserte stikkord som linker. Disse kan han trykke på og få mer informasjon om det aktuelle stikkordet. Brukerne har altså ikke mulighet for å søke blant disse stikkordene, men ettersom de står oppført alfabetisk vil det antagelig fungere helt greit uten.

Den neste modulen, 2D-visning av kaustikkurver, fungerer også som beskrevet i kravspesifikasjonen.

GUI-modulen mangler funksjonalitet for hjelpfunksjon ved høyreklikk. Det er egen knapp for hjelp på knappelinjen som alltid vil være synlig i vinduet og også mulighet for å benytte seg av menylinjen dersom man ønsker hjelp.

OddPlayer-modulen ble det ikke noe av, siden animasjonsdelen av programmet ikke er ferdig implementert. Vi så dermed ingen grunn til å ha en egen modul bare for å vise et bilde.

Arbeidstegningsmodulen er implementert. Brukerne vil få en oversikt over tegnebrettet pluss ekstra ark med informasjon om komponentenes plassering. Avhengig av om lysstråler og rutenett er «slått på» i det brukeren velger å skrive ut tegnebrettet vil det også være med i utskriften. For at informasjonsarket skal gi mening bør i rutenettet være på, i og med at dette også viser koordinater.

Hovedgrunnen til at kravene fra kravspesifikasjonen ikke ble oppfylt må nok sies å være tidsmangel. Vi prioriterte i stedet det å få 3D-rendringen av bilder i havn. Mesteparten av avvikene fra kravspeken har faktisk med animasjon å gjøre. Det viste seg også at animasjon var vanskeligere å få til en tidligere antatt, da det ikke er trivielt å få til i POV-ray. I starten fikk vi veldig mange ønsker fra oppdragsgiver når det gjelder akkurat animasjon, som for eksempel roterende og bevegelige flater, lyskilder og bevegelig kamera. Vi begrenset dette til å animere en lyskilde langs en vektor i rommet, men som nevnt, dette fikk vi ikke tid til.

Hjelpfunksjonen ble heller ikke helt som antydet i kravspeken. Dette grunnes mest i at vi trodde Java hadde et ferdiglaget system á la Windows' hjelp, hvor vi bare trengte å legge til artikler, så ordnet systemet resten selv. Der tok vi feil, og derfor må brukerne nøye seg med et html-basert hjelpesystem, da dette var lettere å utvikle enn et fullt og helt hjelp-system. Programmet har heller ikke så veldig mye funksjonalitet, eller er så lite brukervennlig, at det er lett å rote seg bort. Vi har ikke testet programmet ovenfor studentene ved Kunsthøgskolen, men vi regner med at etter kort tids bruk vil de være fortrolig med hvordan programmet fungerer.

5.2 Diskusjon rundt KlasserOgFunksjoner.doc

Når det gjelder hvordan vi hadde tenkt at klassen *Tegnebrett* skulle bli, er det ikke store avvik. Det eneste er at metodene *tilPovray()* og *tilFil()* har falt bort til fordel for *getKomponenter()*. Vi fant ut at dette var mest hensiktsmessig da det var flere klasser og metoder som trengte komponentvectoren. Klassen *xyLytter* har falt bort og blitt en anonym, indre klasse i *Tegnebrett*. Tegneområdet i et tegnebrett skulle også opprinnelig kun være et *JPanel*, men det viste seg at vi måtte overstyre metoden *paintComponent()* for at opptegningen skulle gå glatt. Ansvaret for grafikken på tegneområdet ble forflyttet fra *Tegnebrett* til *Tegneomraade*.

Metoden *forhaandsvisning(AffineTransform at)* i *Komponent*, har falt bort. Opprinnelig skulle denne metoden transformere komponenten midlertidig og tegne den på tegnebrettet. Etter at vi la til komponentene med en gang brukeren begynte å forme den, ble denne metoden overflødig. Nå blir metoden *transformer(AffineTransform at)* brukt i stedet.

Vi hadde planlagt veldig mye bruk av *BufferedImage*-objekter, men dette var før vi i det hele tatt visste så mye om hva denne klassen innebar. Det viste seg at et objekt av *BufferedImage*-klassen var noe som tilsvarte et bmp bilde, og tok dermed veldig mye minne. Siden disse objektene var såpass store, var de også veldig trege å ha med å gjøre, og de ble uaktuelle til denne bruken.

Animasjonsverktøyet har blitt forandret slik at den kun lage en rett lyskildebane, og ikke en bane bestående av flere rette linjer.

Metoden *er_jeg_truffet(int x, int y)* i *Komponent*, har falt bort i og med at *finnTruffet(int x, int y)* i *Tegnebrett* ble som beskrevet i 4.2.3. Metoden *lysstraaletreff()* har blitt delt opp i *treff()* og *duTraff()*, som forklart i 4.2.4.

5.3 Diskusjon rundt tidsforbruk/gantt-skjema

Noe av det aller første vi gjorde i forbindelse med dette prosjektet var å lage en plan for hvordan prosjektet skulle gjennomføres ved hjelp av Gantt-skjema med milepæler. Det var en rimelig håpløs oppgave å vurdere hva som egentlig skulle gjøres og hvor lang tid de ulike delene ville ta. Det første skjemaet vi satte opp ble ganske grovt og vi prøvde å følge dette skjemaet fram til kodefasen. Etter både perioder med kravspesifikasjon og design, følte vi at oversikten over programmet var mer klargjort og vi bestemte oss derfor i forbindelse med statusmøte nummer tre å utarbeide et noe mer detaljert Gantt-skjema for den resterende perioden, det vil si fra kodefasen og til levering av prosjektet. Begge Gantt-skjemaene er vedlagt rapporten (vedlegg G og vedlegg H). Ettersom ukene gikk har vi ofte vært på etterskudd i forhold til Gantt-skjemaet på grunn av feil estimering av tidsforbruk.

Første forsinkelse i forhold til det originale Gantt-skjemaet merket vi da vi avsluttet designfasen. Da var vi allerede to uker på etterskudd. På denne tiden var vi veldig usikre på hvordan vi skulle representere de parametriserte funksjonene og hvordan vi skulle regne ut treff/refleksjon. En annen ting som gjorde oss forsinket var at vi trodde veldig lenge at vi kunne bruke Java3D til å rendre 3D-bildene. Da vi fant ut at Java3D ikke kunne produsere bilder med høy nok kvalitet, måtte vi legge om 3D modulen. Det var en stund i tankene våre at vi kunne ha et konstant oppdatert preview av 3D-bildet, men da vi gikk bort fra Java3D viste dette seg som veldig vanskelig, og dette ble fjernet fra kravspeken. Ved hjelp av googling fant vi en freeware raytracer: POV-ray. Veileder tok kontakt med en raytracerkyndig kjenning som bekreftet at denne raytraceren var bra nok til vårt formål. Vi måtte deretter sette oss inn i POV-ray, men etter

oppfordring fra veileder gikk vi videre til kodingfasen og lot POV-ray opplæring være en sideaktivitet.

I og med at vi allerede var to uker på etterskudd og vi i løpet av designfasen hadde funnet ut at det originale Gantt-skjemaet var altfor lite detaljert når det gjelder kodefasen, bestemte vi oss for å lage en ny arbeidsplan som skulle gjelde fram til prosjektslutt. Først laget vi GUIen for å bli litt mer motiverte. Deretter laget vi en oversikt over de viktigste klassene og tilhørende metoder. Dette ble nesten som pseudokode og gjorde starten av programmeringen litt lettere. Vi klarte å følge kodefasen med tanke på hvor lang tid vi skulle bruke til sammen, men da uten støtte for animasjon. Det eneste som ikke ble ferdig i tide var parabel og bezier i forbindelse med generering av kode til POV-ray. Det er lite samsvar mellom hvordan man bygger opp objekter i Java2D og i POV-ray, noe som har gjort denne delen tidkrevende.

Testfasen på slutten av prosjektperioden har falt bort. Det er fordi vi ikke så noen grunn til å ha en fase for testing i og med at vi har testet hver nye modul/komponent vi har implementert til systemet underveis. Vi har altså forsikret oss om at all ny kode kan testes nesten umiddelbart. Dette er tilfelle i inkrementell systemutvikling.

5.4 Forslag til videre arbeid/forbedringer

5.4.1 Animasjonsdel

Animasjonsdelen er vel den største delen som vi hadde planer om som ikke ble ferdig. Det er lagt til verktøy for animasjon og mulighet for å tegne animasjonsbane for en lyskilde i tegnebrettet. I lyskildens tilhørende innstillingsvindu er det mulig å angi lyskildens høyde per tid for animasjonen.

Animasjonsdelen er langt fra fullført. For det første er det ikke mulig å rendre med en animasjon. Det er flere måter å løse dette på, for eksempel å la POV-ray generere ett og ett bilde, for deretter å la programmet sette sammen disse bildene til en film. Så vidt vi vet skal det være støtte for dette i Java Media Framework. En annen løsning ville vært å bruke

POV-rays innebygde animasjonsfunksjon, men denne kan til tider være litt innviklet, og også med variert resultat. Det finnes også såkalte uoffisielle «patches» som støtter animasjon som andre POV-ray brukere har laget, som støtter, men ei heller her er vi sikre på resultatet.

For det andre er heller ikke grensesnittet til brukeren bra nok. Brukeren får ingen informasjon om hvor lang animasjonen er i tid, og det burde stått enheter på tidsaksen. Det er lagt til en slider slik at brukeren skal kunne ha muligheten til å stille målestokken på høydeaksen, men funksjonaliteten er ikke lagt inn. Et punkt til er at høyde per tid-funksjonen er nå kun en rett linje, bruker burde i alle fall ha mulighet til å dele den inn i mindre linjestykker, hvis ikke bedre; en kontinuerlig kurve. Det tilsvarende gjelder lyskildens bane i xy-planet. Bruker burde også ha muligheten til å stille inn hvor lang tid animasjonen skal vare, eller med andre ord; hastigheten til lyskilden.

Det er heller ikke lagt til funksjonalitet for å fjerne bare animasjonen av en lyskilde, eneste mulighet for dette er å fjerne lyskilden, og legge den til igjen.

En litt snedig funksjon ville vært om det hadde vært en play/stop/rewind/ffwd funksjon på 2D-tegnebrettet som bevegde lyskilden etter animasjonsbanen bestemt av brukeren. Da kunne han/hun sett en slags forhåndsvisning av animasjonen før rendring. Dette kunne vært spesielt fiffig hvis flere lyskilder var animert samtidig.

5.4.2 Ytelse

Et ganske stort problem med programmet vårt er hastigheten. Dette merkes spesielt om det er mange lysstråler på tegnebrettet, enten ved at det er mange lyskilder, eller at lyskildene har mange lysstråler hver. Dette kommer av at det er veldig mye utregning for hver lysstråle; den må sjekke for hver enkelt komponent om den treffer. Dette kunne kanskje blitt løst ved å ha en bedre algoritme for å finne ut hvilken komponent en lysstråle treffer. Hvis for eksempel tegnebrettet hadde blitt delt opp i kvadranter rundt lyskilden, og en lysstråle som går igjennom en kvadrant bare hadde sjekket de komponentene som er i denne kvadranten. Dette er bare et eksempel, og ville nok ikke fungert like lett i

praksis, for hvis det kommer flere lyskilder til, vil det bli vanskelige å sette grensen mellom kvadrantene. En lignende idé kunne likevel gitt resultater.

Så vidt vi vet så benytter heller ikke JVM seg direkte av matteprosessoren når den regner med desimaltall. Dette kan delvis løses ved at programmet bruker såkalt «strict floatingpointoperation», og ved å bruke metoder fra klassen *StrictMath* i stedet for *Math*. Dette har vi dessverre ikke fått tid til å sette oss ordentlig inn i.

Unødvendige metodekall kan fjernes for også å øke hastigheten. Hvis for eksempel en metode som kun returnerer en verdi blir kalt flere ganger etter hverandre, ville det vært bedre å ta vare på returverdien i en midlertidig variabel. I de forskjellige komponentenes *treff(ParamLinje pl)* forekommer en del slike unødvendige kall. Vi følte at fortjenesten ved å gjøre om på koden var for liten i forhold til tiden det tar å gjøre det.

Vi var lenge inne på tanken med å dele opp systemet i flere tråder for å gjøre det mer effektivt, men det ville nok hatt sine ulemper også, i alle fall slik systemet er designet nå. En mulighet ville vært å ha alt som hadde med lysstrålene å gjøre i egen tråd, altså både utregning og opptegning. Vi prøvde dette ved å la klassen *Kaustikker* arve fra *Thread*, men da protesterte systemet. Dette har noe med at Java lar skjermbildet tegne seg opp i en egen tråd. Det benyttes også et dobbelbuffer for at opptegningen skal gå glattere. Dette bufferet liker ikke at to forskjellige tråder fikler med det samtidig. Se for øvrig avsnitt 2.7.1.1.

5.4.3 Objektorientering

Vi prøvde i utgangspunktet å lage et program som satte objektorientering i høysetet, men designfasen ble noe amputert, da det gikk litt i stå med tanke på planlegging av lysgeometri og 3D-rendring. Slik vi ser det er programmet nå om ikke veldig effektivt bygget opp, desto mer intuitivt og lettforståelig. Dette kan sees på som en fordel om noen andre enn oss skal fullføre det.

Noen av tingene som ikke ble helt optimale er for eksempel objektet av klassen *Svitsj*, som til slutt fikk for mye

oppmerksomhet av de andre klassene (det er kun et fåtall klasser som ikke har en referanse til dette objektet). Dette kunne blitt løst ved å sitte enda lengre med designfasen, men dette ville nok gått utover kodefasen, og vi kan ikke skryte på oss god tid til å programmere. *Svitsj* gjorde det greit for oss, uansett hvor vi var i systemet, å få tak i metoder i *Tegnebrett*, *GUI*, *ElementVindu* etc.

Som tidligere nevnt har vi også prøvd å dytte mest mulig av ansvaret for ting som skjer i programmet til de enkelte komponentene. Dette hadde sin ulempe i at *Komponent* og dens subklasser nå ble veldig tunge med veldig mye kode. Dette gjør at hvis man skal utvide programmet med en ny komponent, må det en del kode til. Vi kunne løst dette ved å standardisere en del av funksjonaliteten, som for eksempel det å skrive til fil, men dette er ingen veldig lett oppgave da mange av komponentene er såpass forskjellige som det de er. Det er også første gangen vi benytter oss av polymorfisme på et såpass avansert plan som dette.

5.4.4 Rendring av 3D-bilde

Helt i begynnelsen, før vi i det hele tatt tenkte på Java3D eller POV-ray, lekte vi litt med tanken på å lage vår egen raytracer. Da kunne vi utviklet en som var spesiallaget for vårt behov. Muligens ville det kunne blitt mye lettere å gå fra 2D tegnebrett til 3D bilde. I ettertid skjønner vi at dette ville blitt et håpløst prosjekt i så tilfelle, siden vi da ville fått enda mindre tid på å kode den "todimensjonale" delen av programmet.

Til slutt ble vi i alle fall sittende igjen med POV-ray, som faktisk er en relativt god raytracer, men problemet ligger i grensesnittet mellom vårt program og POV-ray. Den eneste måten å gjøre dette på uten å sette oss inn i koden til raytraceren er å generere en fil med kode som POV-ray kan generere et bilde ut i fra. Dette betyr mye tung og stygg kode i hver komponent.

POV-ray har heller ingen direkte støtte for formene våre, så mange må tilnærmes ved hjelp av finurligheter. Dette var ingen lett sak, i og med at det ekstremt viktig at alle overflatene er glatte, siden lys tross alt skal reflekteres på dem. Etter mye om og men har vi kommet frem til noen

gode og noen tilfredsstillende tilnærmelser. Hvis man for eksempel ser godt etter på en 3D-versjon av parabelen vil man legge merke til at den ikke er like tykk hele veien. Vi har ingen forslag til hvordan dette kan løses i POV-ray. 3D-rendring av bezierkurver ble det heller ingen løsning på. Men kanskje det å lage en raytracer tilpasset programmet vårt kunne vært et hovedprosjekt til neste år?

5.4.5 Filhåndtering

Komponentene skriver seg selv til en linje på filen. Hver variabel skilles med '_'. Dette gjorde vi for å skille de fra hverandre ved bruk av *StringTokenizer* når komponentene skal leses inn. Grunnen til at vi har gjort det slik, er at dette er den enkleste måten for oss programmeringsmessig. Dersom vi hadde hatt litt mer tid kunne vi sett på en mer fancy måte å gjøre det på. Det finnes ingen forklaring på fila om hva de ulike variablene er, noe som burde vært med dersom filen skulle hatt mening for brukeren. I og med at brukeren får oversikt over de forskjellige komponentene både i programmet i form av innstillingsvinduer eller ved utskrift, så tenkte vi ikke at det var nødvendig. På en annen side burde vi kanskje gjort filen enda mer uleselig for bruker ved å ikke bruke tekstfiler, men vi regner med at kunsthøgskolestudenter ikke er så nysgjerrige av seg.

5.4.6 Printing

Når bruker vil printe ut tegnebrettet nå, vil han først få et ark med grafikk lik det som er på tegnebrettet. Så vil han få en utskrift av data om komponentene, slik som for eksempel kontrollpunkter for bezierkurve og start og sluttpunkt for linje. Vi stilte imidlertid spørsmålstegn ved dette til oppdragsgiver, men fikk inntrykk av at det var av interesse. Det som antagelig hadde vært lettere å forstå for den som eventuelt skal lage en installasjon av dette, ville vel vært at målene hadde stått på tegningen. Dette hadde i alle fall vært en fordel når det gjelder bezierkurven, da vi ikke skjønner at det kan være lett å forme en figur ut i fra mange start-, slutt- og kontrollpunkter.

5.4.7 Brukergrensesnitt

For at kunsthøgskolestudentene skulle kjenne seg bedre igjen i GUIen, burde vi kanskje brukt Windows «look and feel». Som diskutert i avsnitt 3.1.6, viste dette seg å være vanskeligere enn forventet. Hadde vi brukt lengre tid på dette, ville det nok ha løst seg.

En ting som kan være litt irriterende med GUIen er at småvinduene (elementvindu, innstillingsvindu og verktøyvindu), ikke er såkalt «always on top», med andre ord: at de alltid synes. Dette er ikke så lett å ordne som det virker som, da det ikke bare er en boolean som kan settes eller noe lignende. Vi har ingen forslag til hvordan dette kan løses.

5.4.8 Kopier/klipp ut/lim inn

Vi luftet litt tanken på å legge til funksjonalitet for dette allerede i kravspesifiseringsfasen, men vurderte det som en lavprioritetspost og la den til under «hvis tid» i kravspeken.

Problemet med å implementere denne modulen ligger i å kopiere et objekt. Først antok vi at ved å implementere java.lang.*Cloneable* i komponentene og legge til en funksjon *public Object clone()* som igjen kalte moderklassens *clone()* som igjen, om nødvendig kalte sin moderklasses *clone()*, helt til den nådde *Object.clone()*. Etter nærmere undersøkelser viste det seg at selv om et nytt objekt ville blitt opprettet, ville det blitt helt likt den det ble kopiert ifra. Det vil også si referanser(!). Da måtte vi eventuelt klonet alle objektene som det objektet som skulle klones inneholder også. Og hva om objektene inni objektene inneholder objekter? Klippe ut ville vært å kopiere og fjerne på en gang, og lime inn er bare å legge til den kopierte komponenten på tegnebrettet.

En løsning kunne vært å benytte oss av en teknikk lignende den vi benytter oss av når det gjelder skrive til fil og lese fra fil nå. Hvis det for eksempel hadde vært en metode som var identisk *tilFil()* metoden, med det unntak at den returnerte en *String* i stedet for å skrive det til fil ved hjelp av en *BufferedWriter*. En *StringTokenizer* kunne blitt opprettet rundt

denne *String*en, for så å sende den med til constructoren som ellers brukes til å opprette komponenter som blir lest fra fil.

5.4.9 Undo/redo

Dette er en funksjonalitet som vi regner med brukeren kan komme til å savne. Spesielt vanskelig å implementere er det heller ikke, men en del arbeid er det nok. Vi kan tenke oss en løsning der det er en egen klasse, *AngreGjorOm*, som tar seg av alt som har med undo/redo å gjøre, denne klassen har to instanser av *Vector*, en for undo og en for redo. Denne klassen bør ha en del *final static int* som representerer hver sin operasjon som kan gjøres i programmet. For eksempel en for å rotere en komponent, en for å flytte en komponent, en for å legge til en komponent osv. I tillegg bør det finnes en klasse, *HandlingElement*, som egentlig bare tar vare på en *int* som representerer hvilken type operasjon som har skjedd, to referanser til en instans av klassen *Object* og en referanse til hvilket *Tegnebrett* operasjonen skjedde på. *AngreGjorOm* bør ha en metode for å legge til *HandlingElement*er i undovectoren, en metode *undo()* og en metode *redo()*. *undo()* bør hente et *HandlingElement* fra undovectoren, kjøre dette igjennom en *switch* som vet hva den skal gjøre med dette elementet ut i fra den identifiserende *int*en. Deretter bør metoden opprette et inverst *HandlingElement* og legge dette i redovectoren. Tilsvarende, men motsatt funksjonalitet bør finnes i metoden *redo()*.

For hver funksjonalitet i programmet som bør kunne angres, må det dermed legges til et slikt *HandlingElement* i undovectoren. Hvis brukeren for eksempel har flyttet en komponent, bør det opprettes et *HandlingElement*-objekt hvor den ene *Object*-referansen peker til komponenten, og den andre peker til en *AffineTransform* som hvis kjørt på komponenten vil flytte den tilbake til der den var.

5.4.10 Komponentforbedringer

En del forbedringer kan også gjøres når det gjelder komponentene og interaksjon med bruker. En ting som kunne vært hendig for brukeren var om han/hun kunne sette navnet på flatene og lyskildene selv, i stedet for at de får navnet flate# og lyskilde#, hvor # er et nummer. Dette

kunne skjedd for eksempel i rullgardinen i elementvinduet, eller ved at brukeren høyreklikket på komponenten. Denne funksjonaliteten var egentlig planlagt, men ble nedprioritert.

I elementvinduet finnes det en oversikt over x/y-koordinat, høyde og bredde/lengde. Bredde og lengde blir ikke oppdatert uansett hva bruker gjør. Disse skulle egentlig vise til bredden og lengden(høyden) til en tenkt firkant rundt en komponent. Vi tenkte den kunne være til hjelp når bruker legger til et nytt element.

Når bruker oppretter en bezierkurve, eller klikker på en bezierkurve med marker/skaler-verktøyet har han/hun muligheten til å flytte på kontrollpunktene til kurvene. Det kunne kanskje være ønskelig å også ha muligheten til å flytte på start og sluttpunktene også. Dette er egentlig ikke noen vanskelig oppgave, men det skal litt fikling til. Det som kan gjøres er å legge til start og sluttpunktet i klassen *BezierFirkant,* og laget tilsvarende funksjonalitet med disse punktene som det finnes med kontrollpunktene.

En ting som kanskje bare er smakssak er at når man bruker marker/skaler-verktøyet på en linje, får man opp en skaleringsboks som på alle andre komponenter. Det ville kanskje passet bedre å la brukeren kunne flytte start- og sluttpunktet.

Når brukeren har opprettet en ellipse, får han opp en dialogboks hvor han får spørsmål om han vil ha en åpning i denne komponenten og hvor stor denne eventuelt skal være. Når den er satt, så er den satt en gang for alle. En idé ville vært å la brukeren ha mulighet til å få opp dialogboksen igjen, for eksempel ved hjelp av en meny som dukker opp ved høyreklikk.

Hvis brukeren skal skifte fargen på lyset til en lyskilde, må han gjøre dette ved hjelp av en standard fargevelger som følger med Java. Det kunne kanskje vært bedre om han/hun kunne velge mellom et ferdig antall valgte farger. Brukeren kunne heller forandret på de ferdig valgte fargene ved hjelp av denne fargevelgeren. Fargevelgeren kan også tilpasses bedre, men dette har vi ikke hatt tid til å sette oss inn i.

5.4.11 Andre lyskilder

Å kunne legge til andre lyskilder enn punktlyskilde har aldri vært en del av planen vår. Vi var innom det i kravspesifikasjonsfasen, men regnet med at dette kom til å ta tid og markerte det med «hvis tid». En del problemer er forbundet med dette; det finnes ingen spesiell støtte for dette i POV-ray, hvordan skal dette representeres grafisk i 2D for brukeren og hvordan skal treff/refleksjons-delen av programmet håndtere dette. Vi har egentlig ikke tenkt så mye på det, så vi har ikke noe umiddelbart forslag til løsning av dette, bare at den nye lyskilden må arve fra klassen *Lyskilde*.

5.5 Evaluering av gruppas arbeid

5.5.1 Organisering

Under forprosjektperioden (se vedlegg J) måtte vi velge en prosjektleder. Det var egentlig ingen av oss som hadde lyst til å ta på seg dette ansvaret. Til slutt valgte vi Maria til å være gruppeleder. Hovedansvaret til prosjektleder har vært å holde orden på det praktiske. Utover dette synes vi ikke det var nødvendig å ha en sjef i gruppa. Dersom det skulle oppstå uenigheter som vi ikke klarte å løse innad i gruppa, var planen å ta det opp med veileder. Vi har imidlertid sluppet det.

5.5.2 Fordeling av arbeid

I kodefasen og designfasen jobbet vi for det meste sammen, og diskuterte alle tre for å få best mulig resultat. Når vi kom til kodefasen, jobbet vi mer hver for oss. Hva slags oppgaver hver av oss skulle ha, valgte vi mer eller mindre selv. Hvis en av oss fikk problemer med å løse en oppgave, var de andre ikke sene med å hjelpe til.

5.5.3 Prosjekt som arbeidsform

Prosjektarbeid blir mer og mer populært i arbeidslivet og i den sammenhengen har hovedprosjektet vært en nyttig erfaring for oss. Det er også det første store prosjektet vi har vært igjennom på Høgskolen i Gjøvik eller andre steder. Vi har erfart at det er vanskelig å lage en detaljert plan for kodedelen av et prosjekt før man har kommet i gang med kravspesifikasjon og design.

5.5.4 Subjektiv opplevelse av hovedprosjektet

Vi føler dette prosjektet har gitt oss et innblikk i hvordan det kan komme til å bli å jobbe som dataingeniør. Prosjektet vårt var et relativt programmeringstungt prosjekt, men vi fikk med oss alle fasene av det å utvikle et prosjekt. Vi merket hvor vanskelig det kan være å planlegge tidsfrister og arbeidsoppgaver langt frem i tid, og hvor viktig det er å jobbe jevnt og trutt i et slikt prosjekt. Dette er ting som vi regner med kommer med erfaring – vi har i alle fall blitt én erfaring rikere.

Når det gjelder måten hovedprosjekt er organisert fra skolens side har vi lite å utsette. Vi er spesielt fornøyd med at vi kunne ha et så fritt forhold til veileder. Det var også lett å få hjelp fra andre lærere om vi trengte hjelp; det var faktisk bare å banke på og spørre. Dette tror vi hadde vært vanskelig på et større lærested.

Hvis vi skal klage på noe må det være PC-ene som vi hadde mulighet til å låne i forbindelse med prosjektet. For det første var dette PC-er som har gått ut på dato for lenge siden, og for det andre hadde det passet bedre om hver fikk en PC til disposisjon.

Hovedprosjektsidene på Internet er også litt uoversiktlige, men det samme kan egentlig sies om hele hig-weben. Man fikk den informasjonen man trengte om man lette lenge nok, eller enda lettere: spurte noen som visste det.

6 Konklusjon

Etter å ha gjennomført det avsluttende hovedprosjektet ved HiG er vi rimelig fornøyd med resultatet ut fra den tiden vi har hatt til rådighet. Mange nye tanker og muligheter for å løse problemene har dukket opp mot slutten av prosjektet og det er jo litt synd i og med at vi nå er ferdige.

Vi har klart å tilfredsstille de fleste av de kravene som ble satt i starten av prosjektet, men programmet slik det er nå har også mange muligheter for videreutvikling. Vi tror ikke det vil være veldig vanskelig å fortsette utviklingen på et senere tidspunkt dersom det skulle være ønskelig, i og med at vi har forsøkt å designe systemet på en intuitiv, objektorientert måte. Oppdragsgiver hadde veldig mange ønsker da vi startet prosjektet, og de som vi ikke har oppfylt er helt sikkert fortsatt aktuelle.

Vi er forøvrig veldig fornøyde med veileder. Særlig har han vært hjelpsom når det gjelder lysstrålegeometridelen av programmet, men det har heller ikke vært noe problem å stikke hodet innom om det var noe vi lurte på. Vi har også hatt ukentlige møter med veileder, hvor han ble oppdatert om status på prosjektet, og kom med noen motiverende ord.

Bibliografi

[1] H. Deitel og P. Deitel, Java How to program. 5. utgave (2003)

[2] Sun Microsystems, hjemmeside `java.sun.com`

[3] POV-ray, hjemmeside `www.povray.org`, `news.povray.org`

[4] F. Koch Jensen docJava, hjemmeside `www.docjava.dk`

Vedlegg