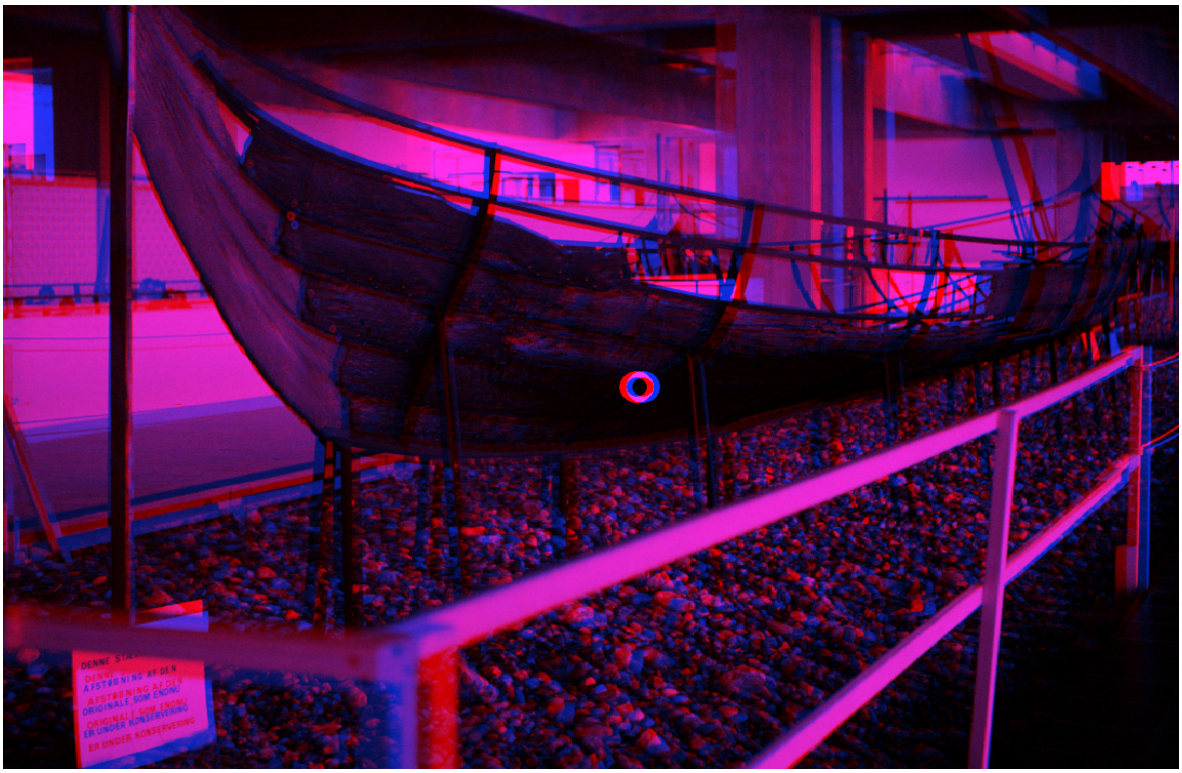
IMPLEMENTERING AF MODULÆRT STEREOINTERFACE TIL 3D MÅLING I DIGITALE BILLEDER.



Eksamensprojekt ved
Institut for Informatik og Matematisk Modellering (IMM)
Afdeling for Geoinformatik
Danmarks Tekniske Universitet (DTU)

Lyngby d. 8/4 2005
Af:

Jesper Kiær Thygesen (s973872)

Dansk titel: Implementering af modulært stereointerface til 3D målinger i digitale billeder.

English Title: Implementation of a modular stereo interface for 3D measurements in digital images.

Programtitel: Stereo Image Display Module (SIDM).

Vejleder:

Assoc. Prof. Keld Dueholm

IMM, Geoinformatik.

Assisterende Vejleder:

Assoc. Prof. Jens Thyge Kristensen

IMM, Computer Science & Engineering

Abstrakt

Denne rapport beskriver udviklingen af et programmodul, der kan vise stereomodeller, dvs. ægte tredimensionale billeder, ud fra normale bitmap billeder. Stereomodeller kan vises i flere vinduer samtidig, og hvert vindue vil desuden indeholde et rumligt målemærke, kaldet et vandrende mærke. Hvis det benyttede kamera er kalibreret, og orienteringen af billederne er kendte, kan der foretages tredimensionale målinger vha. det vandrende mærke.

I projektet fokuseres der på design, implementering og test af programmodulet. Modulet er udviklet med henblik på senere udvidelse, og der er derfor lagt vægt på en velstruktureret, modulær opbygning. Programmodulet skal benyttes fra andre programmer via funktionskald, og en demo af et sådant program er blevet udviklet.

Udviklingen af programmet er foregået på en Windows-platform, men programmet er udviklet til platformuafhængighed. C++ er benyttet som implementeringssprog og OpenGL benyttes til visning af billeder.

Abstract (English)

This report describes the development of a program module, which shows stereo models, i.e. true three dimensional images, created from normal bitmap images. Stereo models can be displayed in multiple windows simultaneously, and each window will also include a three dimensional measuring mark, called a floating point. If the camera has been calibrated, and the orientation of both images is known, three dimensional measurements can be performed using the floating point.

Focus has been placed on the design, implementation and test of the program module. The module has been developed with later expansions in mind, and emphasis has been placed on a modular construction. The module will be called from other programs by function calls, and a demo of such a program has been developed.

The program module has been developed on a Windows platform, but has been prepared for portability to other platforms. The implementation language is C++ and OpenGL has been used to display images.

Indholdsfortegnelse

Forord

Denne rapport beskriver et eksamensprojekt, der er udført under vejledning af Lektor Keld Dueholm på instituttet for Informatik og Matematisk Modellering (IMM), afdelingen for Geoinformatik på Danmarks Tekniske Universitet (DTU), med yderligere assistance fra Lektor Jens-Thyge Kristensen på IMM, afdelingen for Computer Science and Engineering. Projektet har forløbet i tidsrummet 18. september 2004 til 8. april 2005, med indlagt en måneds ferie. Projektet repræsenterer 30 point for den studerende.

Målet med projektet har været at designe og implementere et programmodul, baseret på ønsker af vejleder Keld Dueholm. Det har været en rimelig bunden opgave, hvor programmets funktionaliteter har været fastlagt fra start. Fokus af projektet er derfor lagt på design og implementering og ikke på analysen af problemstillingen.

Jeg vil gerne takke min vejleder Keld Dueholm for at have gjort dette projekt muligt og for stort engagement i projektet, og jeg vil takke begge mine vejledere for upåklagelig hjælp, hver gang jeg har følt det nødvendigt. En stor tak går til Mikkel Gjøl for hans assistance til min forståelse af OpenGL og GLUT. Til sidst vil jeg gerne takke Peter Svendsen, Jens Hvelplund, Christian Hansen og Kristian Thygesen for hjælp med rapporten, og flere personer fra køkken 29 på Kampsax Kollegiet for lån af computere i tide og utide.

RAPPORTSTRUKTUR

I dette afsnit vil rapportens struktur kort blive gennemgået. Ligesom for projektet, er fokus af denne rapport lagt på design og implementering, frem for analyse af problemstillingen og lignende programmer.

1 Problembeskrivelse: Problemstillingen, der ligger til grund for projektet, introduceres i dette kapitel, og den benyttede matematik bag problemet gennemgås. Denne baggrundsviden introducerer begreberne, der benyttes i projektets problemformulering, og denne er derfor placeret sidst i kapitlet.

2 Analyse: Problemet analyseres dybere ved kort at gennemgå lignende programmer. Hovedparten af kapitlet beskriver kravspecifikation for projektet.

3 Design: Dette kapitel gennemgår programmets implementeringsuafhængige design, dvs. det design, der er tegnet uden at tage højde for implementeringssprog og benyttede programbiblioteker.

4 Implementeringsspecifik analyse: Efter implementeringssproget er valgt, følger en analyse og efterfølgende valg af programbiblioteker, som kan anvendes af programmet.

5 Implementering: Dette kapitel beskriver det implementeringsspecifikke design samt selve implementeringsfasen. Beskrivelsen af designet vil fokusere på områder, hvor det afviger fra det implementeringsuafhængige design.

6 Test: Gennem dets udvikling er programmet løbende blevet testet. Dette kapitel gennemgår metoderne valgt til test samt de vigtigste resultater.

7 Konklusion: Projektforløbet og det endelige resultat analyseres. Desuden bliver forslag til videreudvikling af programmet gennemgået.

Appendiks

Appendiks A Ordforklaringer: Dette appendiks opsummerer betydningen af specifikke forkortelser og betegnelser, der benyttes gennem rapporten.

Appendiks B Projektforløb: En gennemgang af projektforløbet med iterationsplan og tidsforbrug.

Appendiks C Vejledning til bruger: En brugervejledning til installation og brugen af programmodulet via det medfølgende demobrugerprogram.

Appendiks D Vejledning til udvikler: Information om SIDM, der har interesse for en videreudvikling af modulet eller benyttelse af modulet fra eget program.

Appendiks E Programdesign: Dette appendiks fungerer som et opslagsværk til designkapitlet. Det indeholder benyttede use cases, klassediagrammer o.a.

Appendiks F Test: Detaljerede resultater over test af programmodulet.

Appendiks G Referencer: Referencer til benyttet litteratur og programbiblioteker. Henvisninger indført i kantede parenteser, dvs. [ ], er henvisninger til disse referencer.

1 Problembeskrivelse

Dette kapitel gennemgår problemstillingen, der ligger til grund for dette projekt. De første afsnit introducerer emnet og gennemgår de benyttede begreber indenfor stereoskopi. Efterfølgende beskrives matematikken, som skal benyttes i programmet. Kapitlet afsluttes med projektets problemformulering.

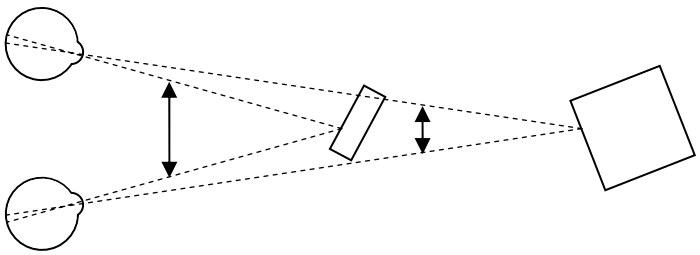
1.1 DYBDESYN

Sidder man i en togkupe og betragter landskabet, man kører forbi, ved man, at stationære objekter, der bevæger sig hurtigt forbi ens synsfelt, ligger tættere på end objekter, der bevæger sig langsomt. Dette fænomen kaldes bevægelsesparallakser (*motion parallax*) og er blot en af mange metoder, vi mennesker benytter til at bestemme dybden i den verden vi ser.

På kortere distancer har mennesker en mere nøjagtig måde at bestemme dybden på. Grundet den indbyrdes afstand mellem vores to øjne vil det venstre øje se verden en anelse forskelligt fra det højre øje. Hjernen sammensætter de to billeder til det tredimensionale billede, de fleste mennesker opfatter. Effekten kaldes på engelsk *binocular disparity*, som kan oversættes til parallakser på dansk. Dybdebestemmelse baseret på parallakser er meget nøjagtig inden for få meter, mens den forringes som afstanden øges.

Tæt sammenhørende med parallakser er øjnenes fokusering (*accommodation*) og konvergens (*convergence*). Når man fokuserer på fjerne objekter vil ens øjne være drejet i den samme retning. Kigger man derimod på nærmere objekter, vil øjnene være drejet en anelse ind mod hinanden. Effekten kaldes konvergens (se Figur 1-1). Hjernen benytter øjnenes konvergens til korrekt at placere de to modtagne billeder i forhold til hinanden. Desuden benyttes øjnenes konvergens til at bestemme deres fokusering [R17].

For at se et objekt tydeligt skal øjets linse tilpasses, så dens brændvidde passer med afstanden til det beskuede objekt. Denne fokusering tilpasses af hjernen ud fra øjnenes konvergens. Hvis øjnene fx har parallelle synslinier, dvs. de ikke konvergerer, vil øjnene fokusere på meget fjerne objekter, og nære objekter vil derfor være uskarpe [R17].



Figur 1-1: Figuren viser effekten af parallakser, og hvordan øjnenes konvergens afhænger af afstanden til det beskuede objekt.

1.2 STEREOSKOPI

Stereoskopi beskriver teknikken, hvor parallakser benyttes til at danne illusionen af dybde i todimensionale billeder. Ideen er at vise ét billede på det venstre øje og et andet billede på det højre øje. På denne måde vil beskueren opfatte de to todimensionale billeder som værende sammenhørende, og resultatet bliver et samlet tredimensionalt billede. De to billeder, der danner 3d-billedet i en stereoskopisk visning, kaldes tilsammen for en stereoskopisk model eller blot stereomodel.

For at en stereomodel skal opfattes korrekt af vores hjerne, er det vigtigt at billederne minder om det, vi selv ville forvente at se med vores egne øjne. Ser vi bort fra konvergens, kigger vores to øjne altid i samme retning. De to billeder i en stereomodel skal derfor også tages med nær parallelle fotograferingsretninger (se dog afsnit 1.6 Billednormalisering).

Når to billeder vises ovenpå hinanden for at danne en stereomodel, vil et punkt i det ene billede ikke nødvendigvis ligge på samme position i det andet billede. Når punktet er forskudt i vandret retning, siges billedet at have horisontale parallakser. Det er disse horisontale parallakser, der skaber illusionen af dybde i modellen. Hvis et punkt derimod er forskudt i lodret retning, har billedet vertikale parallakser. Da menneskets øjne kun er forskudt horisontalt i forhold til hinanden, forventer hjernen ikke at observere vertikale parallakser i de sete billeder. Tilstedeværelsen af vertikale parallakser i en stereomodel vil derfor ødelægge stereoeffekten i modellen. Betegnelsen stereoskopisk billedpar bruges i dette projekt om et par billeder, der har potentiale til at blive en stereoskopisk model, men som ikke nødvendigvis er orienteret korrekt i forhold til hinanden.

Den relativt korte afstand mellem vores øjne (*ocular distance*) bevirker, at effekten af parallakser (dvs. de horisontale parallakser) er forsvindende på større afstande. Ved at have en langt større afstand mellem kamerapositionerne for de to billeder i en stereomodel er det muligt at udføre effektive dybdebestemmelser selv på meget store afstande.

1.3 STEREOSKOPISKE BETRAGTNINGSSYSTEMER

Der findes mange forskellige metoder til at opsplitte billeder, så kun det venstre billede ses af det venstre øje, og kun det højre billede ses af det højre øje. Nogle af de mest anvendte af disse stereoskopiske betragtningssystemer vil kort blive introduceret nedenstående.

HMD: I et *head mounted display* (HMD), også kendt som en virtual reality hjelm, benyttes en simpel metode til at opsplitte billederne. Hjelmen indeholder to små skærme, én skærm til hvert øje, der viser hver sit billede.

Anaglyph: Ved *anaglyph*-metoden benyttes farvefiltre til at adskille de to billeder fra hinanden. Almindelig anaglyph kan kun vise monokrome billeder, så farvebilleder skal først konverteres til monokrome billeder ved at benytte lysstyrken af hver pixel (kaldet *intensity* i IHS formatet). Det ene billede i en stereomodel vises i rene røde farver, mens det andet billede vises i en komplementær farve, dvs. enten cyan, grøn eller blå. Ved at benytte briller med glas i tilsvarende farver vil kun farverne fra det ene billede kunne slippe gennem glasset for

det ene øje. Fx tillader et rent rødt glas kun røde farver at trænge igennem, mens alle andre farve absorberes. Dette er årsagen til, at vi ser glasset som værende rødt.

LCD briller: En anden metode til at vise stereomodeller på en computerskærm er ved brug af LCD *shutter glasses*. Disse elektroniske briller kan blokere lys til skiftevis det ene og det andet øje. Når brillerne synkroniseres med skærmens opdatering, vil hvert øje kun se hver anden frame på skærmen. Efter hver opdatering skifter skærmen mellem at vise højre og venstre billede (kaldes *page flipping*). På denne måde dannes den stereoskopiske effekt, der i modsætning til anaglyph kan vises i farver.

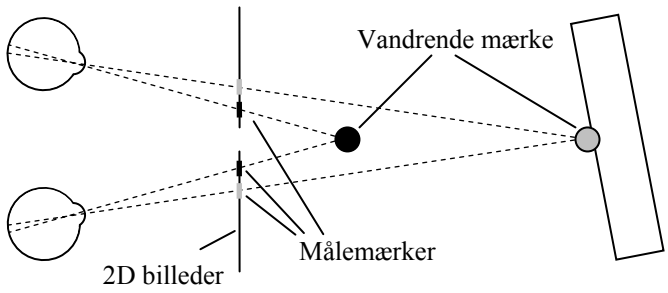
Polarisering: Et polaroid er en tynd, gennemsigtig, syntetisk plade, der pga. dens molekylære opbygning kun tillader lys med en given polarisering at passere. Et polaroid placeret foran en lyskilde vil derfor polarisere lyset, der gennemtrænger det. Da vandret polariseret lys ikke kan gennemtrænge et polaroid, der er gennemsigtigt for lodret polariseret lys - og omvendt - kan polaroider benyttes til at adskille de to billeder i en stereomodel fra hinanden. Polarisering kan benyttes i forbindelse med en computerskærm, ved at placere et elektronisk, polariseret filter foran skærmen. Efter hver opdatering af skærmen vil dette filter vende polariseringen. Ved at benytte page flipping kan de to billeder adskilles, så de polariseres i hver sin retning. Når brugeren benytter polaroid briller, hvor de to glas er modsat polariserede, vil de to billeder vises på hvert sit øje. En anden metode til at polarisere lyset fra digitale billeder er at benytte to projektorer med modsatrettede polaroidfiltre. Den ene projektor viser det venstre billede, og den anden det højre billede, og de to billeder vises oven på hinanden på et lærred.

1.4 STEREOSKOPISK MÅLING

En af anvendelsesmulighederne for stereoskopi er til tredimensionale målinger i stereomodeller også kaldet stereoskopiske målinger. Dvs. målinger hvor brugeren ikke blot kan måle i højden og i bredden af det tredimensionale billede men også i dybden. Til en sådan måling benyttes et tredimensionalt målemærke, kaldet et vandrende mærke. Ligesom stereomodellen, består det vandrende mærke af to billeder; et billede til det venstre øje og et andet billede til det højre øje. Et sådant enkeltbillede vil i denne rapport refereres til som et målemærke. Dvs. et vandrende mærke udgøres af to målemærker, ét på hvert billede.

Da det vandrende mærke vises i stereo ligesom billederne, kan også dybden af det vandrende mærke bestemmes. Hvis de to målemærker, der udgør det vandrende mærke, er placeret på samme punkt på skærmen, vil det vandrende mærke opfattes som liggende i niveau med skærmoverfladen. Hvis målemærket, der ses med det venstre øje, ligger til venstre for det andet målemærke, vil det vandrende mærke opfattes som være placeret dybere inde i skærmen. Ved på denne måde at bevæge det vandrende mærke i dybden kan det placeres, så det ser ud til at ligge oven på et objekt eller flade (se Figur 1-2). Dette er muligt pga. menneskets evne til at tolke effekten af parallakser. På denne måde kan rimelig præcise tredimensionale målinger udføres på modellen. Et alternativt til at ændre dybden af det vandrende mærke, er at forskubbes de to billeder i forhold

til hinanden og derved ændre dybden af objekter i billederne.



Figur 1-2: Dybden af det vandrende mærke kan bestemmes ud fra målemærkernes indbyrdes placering på de to billeder. Det er desuden muligt at fornemme, når det vandrende mærke befinder sig på overfladen af et objekt eller en flade.

Et af de store anvendelsesområder for stereoskopiske målinger er ved opmålinger af landskaber. Dette kunne fx være optegning af højdekurver ud fra flybilleder, dvs. billeder taget fra lavtgående fly. Inden computeren vandt frem, blev sådanne stereoskopiske målinger udført på stereoskoper. De benyttede stereoskoper er ofte avancerede mekaniske instrumenter, der vha. spejle og linser projicerer to fotografier op på hvert sit øje. For at se korrekt i disse instrumenter skal øjnene ikke konvergere, dvs. man fokuserer som på et fjernt objekt. Linser i apparatet sikrer, at fotografierne alligevel træder skarpt frem.

I de senere år er det blevet stadig mere almindeligt at benytte computerprogrammer til at udføre stereoskopiske målinger på. Selvom en computerskærm ikke kan vise nær den samme detaljegrad som den analoge metode, er computeren langt billigere. Desuden er det muligt direkte at læse målingerne ind på computeren og behandle disse målinger digitalt.

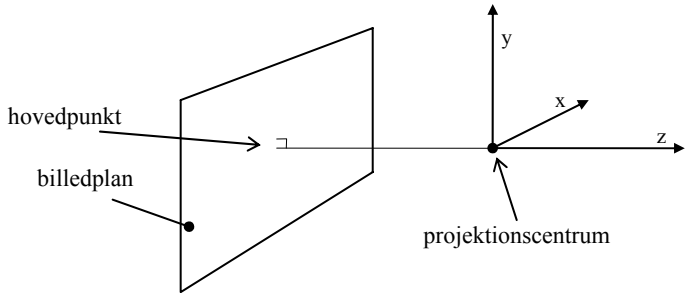
1.5 STRÅLELIGNINGERNE

For at kunne udføre stereoskopiske målinger med det vandrende mærke er man nødt til at vide hvordan et punkt i rummet projiceres ned til et tilsvarende punkt i billedet. Denne projektion beskrives af straaleligningerne, og disse vil blive gennemgået i dette afsnit. Afsnittet benytter stort set samme betegnelser og beregningsmetoder, som bliver benyttet i [R3].

I dette projekt vil der primært arbejdes med tre forskellige koordinatsystemer:

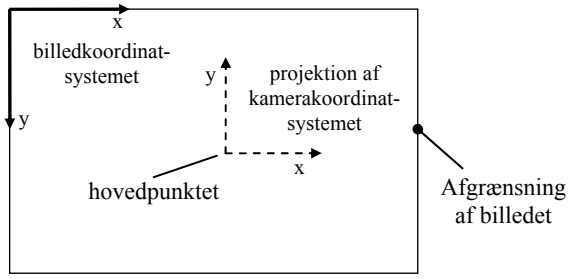
Objektkoordinater er 3D koordinater i den afbildede verden. I en stereomodel bygget over et sæt af flybilleder kan x- og y-koordinaterne fx være længde- og breddegrader, mens z-koordinaten er højden over vandet.

Kamerakoordinater definerer 3D koordinater, der er centreret i kameraets projektionscenter. x- og y-akserne ligger parallelt med billedplanet, dvs. det plan, hvorpå billedet genereres, og den negative z-akse løber gennem hovedpunktet (se Figur 1-3). Hovedpunktet er det punkt i billedplanet, der ligger vinkelret under projektionscenteret. Objektkoordinater kan konverteres til kamerakoordinater alene ved rotation og translation.



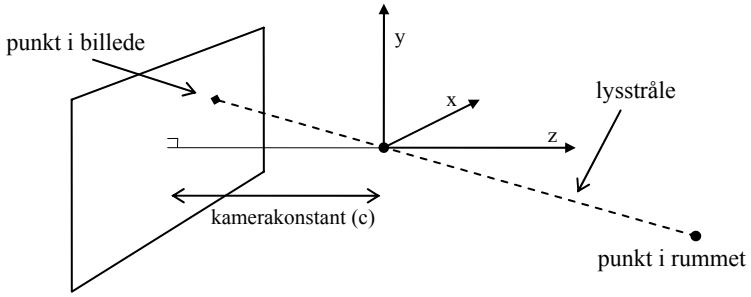
Figur 1-3: Kamerakoordinater centreres i kameraets projektionscenter. En vinkelret linie på billedplanet, der går gennem projektionsceteret, skærer billedplanet i hovedpunktet.

Billedkoordinater er 2D pixelkoordinater i det endelige billede. Centrum af koordinatsystemer ligger i billedets øvre, venstre hjørne. X-aksen er positiv mod højre mens y-aksen er positiv nedad (se Figur 1-4). I det ideelle tilfælde kan kamerakoordinater konverteres til billedkoordinater ved en perspektivisk projektion og en efterfølgende translation af koordinatsystemet fra hovedpunktet til hjørnet af billedet. Ofte vil yderligere operationer dog medtages, for at tage højde for fejl i kameraet.



Figur 1-4: Billedkoordinatsystemet placeres i billedets øverste, venstre hjørne. Y-aksen er modsat rettet i det projicerede kamerakoordinatsystem.

Figur 1-5 viser hvordan en lysstråle fra et punkt i rummet ledes gennem kameraets projektionscenter for efterfølgende at blive projiceret ned på billedet. Denne proces kan simuleres matematisk vha. straaleligningerne. Først vil punktets objektkoordinater konverteres til kamerakoordinater, hvorefter disse koordinater vil projiceres ned på billedplanet og konverteres til billedkoordinater. Disse forløb beskrives i de efterfølgende afsnit.



Figur 1-5: Et punkt i rummet sendes gennem projektionscenteret for at blive projiceret ned på billedplanet.

1.5.1 Homogene koordinater

I de efterfølgende afsnit vil der ofte blive benyttet homogene koordinater i stedet for almindelige fysiske koordinater. Homogene koordinater tilføjer en ekstra dimension til de fysiske koordinater, hvilket gør det muligt at modellere perspektiviske projektioner. De homogene koordinater simplificerer også udførelse af andre operationer som fx translation.

Sammenhængen mellem fysiske og homogene koordinater kan ses nedenfor. Her er w en konstant, der er forskellige fra 0.

$$\underline{X} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \rightarrow \begin{bmatrix} w \cdot X \\ w \cdot Y \\ w \cdot Z \\ w \end{bmatrix} = \underline{\hat{X}}$$

I ligningerne i dette kapitel vil homogene koordinater markeres med en ”hat”, vektorer eller punkter markeres med en enkelt understreg, mens matricer markeres med dobbelt understreg.

1.5.2 Ydre orientering

Den ydre orientering beskriver sammenhængen mellem objektkoordinater $\underline{X}$ = (X, Y, Z) og kamerakoordinater $\underline{x}$ = (x, y, z). Orienteringen beskrives ud fra objektkoordinaterne af projektionscenteret (X_0, Y_0, Z_0), samt rotationen af kameraet (Ω, Φ, K) omkring hver af de tre akser.

Konvertering fra objektkoordinater til kamerakoordinater kan udføres lineært via en translation, efterfulgt af en rotation omkring hver af systemets akser.

$$\underline{\hat{x}} = \underline{\underline{R}} \cdot \underline{\underline{T}} \cdot \underline{\hat{X}}$$

Her beskriver matricen $\underline{\underline{T}}$ translationen af koordinatsystemet:

$$\underline{\underline{T}} = \begin{bmatrix} 1 & 0 & 0 & -X_0 \\ 0 & 1 & 0 & -Y_0 \\ 0 & 0 & 1 & -Z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matricen $\underline{\underline{R}}$ beskriver rotationen af kamerakoordinatsystemet i forhold til objektkoordinatsystemet. Matricen opbygges af de tre matricer $\underline{\underline{R}}_\Omega$, $\underline{\underline{R}}_\Phi$ og $\underline{\underline{R}}_K$, hvor $\underline{\underline{R}}_\Omega$ er rotationen omkring X-aksen, $\underline{\underline{R}}_\Phi$ er rotation omkring Y-aksen, og $\underline{\underline{R}}_K$ er rotationen omkring Z-aksen.

$$\underline{\underline{R}} = \underline{\underline{R}}_K \cdot \underline{\underline{R}}_\Phi \cdot \underline{\underline{R}}_\Omega$$

$$\underline{\underline{R}}_{\Omega} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\Omega) & \sin(\Omega) & 0 \\ 0 & -\sin(\Omega) & \cos(\Omega) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\underline{\underline{R}}_{\Phi} = \begin{bmatrix} \cos(\Phi) & 0 & -\sin(\Phi) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\Phi) & 0 & \cos(\Phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\underline{\underline{R}}_{K} = \begin{bmatrix} \cos(K) & \sin(K) & 0 & 0 \\ -\sin(K) & \cos(K) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

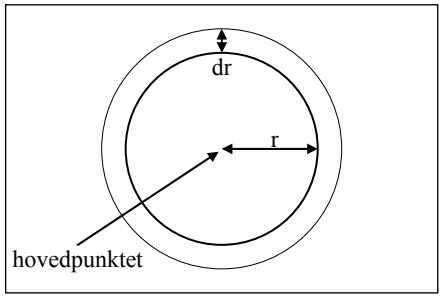
1.5.3 Indre orientering

Den indre orientering beskriver konverteringen af kamerakoordinater til billedkoordinater. Første trin er at projicere kamerakoordinater $\underline{x} = (x, y, z)$ direkte ned på billedplanet, til hvad der i denne rapport kaldes projicerede kamerakoordinater $\underline{x}_c = (x_c, y_c)$. Projektionen forløber ved den lineære operation:

$$\hat{\underline{x}}_c = \underline{\underline{P}} \cdot \hat{\underline{x}} \quad \text{, hvor } \underline{\underline{P}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1/c & 0 \end{bmatrix}$$

Her beskriver c kamerakonstanten, der er afstanden mellem projektionscenteret og hovedpunktet.

I det optimale tilfælde vil lysstråler løbe i en lige linie gennem projektionscenteret til billedplanet. I et almindeligt kamera vil kameraets linse dog forvrænge lysstrålen, så denne ikke forløber som en lige linie. Linsens forvrængning opdeles i tangentiel fortegning og radial fortegning: Den tangentielle fortegning beskriver hvordan den indgående og udgående lysstråle forskydes, så de ikke skærer aksen mellem hovedpunkt og projektionscenter (kaldet den optiske akse) på det samme sted. Den radiale fortegning forårsager, at vinklen på den indgående lysstråle er forskellig fra den udgående stråle. Hovedparten af den radiale fortegning er symmetrisk omkring den optiske akse og kaldes derfor den radialsymmetriske fortegning. Da den radialsymmetriske fortegning er den absolut største fortegning, vil der efterfølgende kun blive taget højde for denne [R3].



Figur 1-6: Den radialsymmetriske afvigelse beskriver en symetrisk forskydning direkte mod eller væk fra hovedpunktet.

Den radialsymmetriske fortegning forårsager en symmetrisk afvigelse omkring hovedpunktet i billedet (se Figur 1-6). Denne fortegning afhænger af afstanden r til hovedpunktet og vil ofte beskrives som et polynomium:

$$dr = a_1 \cdot r + a_3 \cdot r^3 + a_5 \cdot r^5 + a_7 \cdot r^7, \text{ hvor } r = \sqrt{x_c^{\,2} + y_c^{\,2}}$$

Hvis man opdeler beregningen på de to akser, får man følgende formel, der kan konvertere de projicerede kamerakoordinater $\underline{x}_c = (x_c, y_c)$ til tilsvarende koordinater, der er korrigeret mht. den radialsymmetriske fortegning $\underline{x}_r = (x_r, y_r)$.

$$x_r = (1+\Delta) \cdot x_c$$

$$y_r = (1+\Delta) \cdot y_c$$

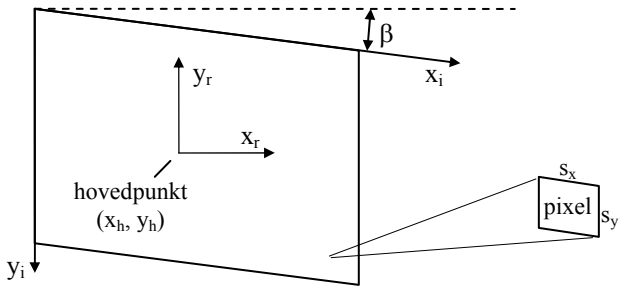
$$\text{hvor } \Delta = a_1 + a_3 \cdot r^2 + a_5 \cdot r^4 + a_7 \cdot r^6, \text{ og } r = \sqrt{x_c^{\,2} + y_c^{\,2}}$$

Det bør bemærkes at denne operation ikke er lineær.

Til sidst skal de projicerede kamerakoordinater, som er korrigeret mht. den radialsymmetriske fortegning $\underline{x}_r = (x_r, y_r)$ konverteres til billedkoordinater $\underline{x}_i = (x_i, y_i)$.

I det ideelle tilfælde ville dette blot kræve at man parallelforskyder koordinatsystemet fra at have centrum i hovedpunktet (x_h, y_h) til at have centrum i øverste, venstre hjørne af billedet. I praksis vil transformationen af billedet til pixels ofte forårsage deformationer. Hovedparten af disse deformationer kan beskrives af en affin transformation.

Den affine transformation beskrives ud fra akseskævheden β og *aspect ratio* a, der beskriver forholdet mellem bredde s_x og højde s_y af en pixel, dvs. $a = s_y/s_x$ (se Figur 1-7).



Figur 1-7: Affine transformation.

De projicerede kamerakoordinater, som er korrigeret mht. den radialsymmetriske fortegning kan konverteres til billedkoordinater ved den lineære operation:

$$\underline{\hat{x}}_i = \underline{\underline{b}} \cdot \underline{\hat{x}}_r \text{, hvor } \underline{\underline{b}} = \begin{bmatrix} a \cdot \cos(\beta) & a \cdot \sin(\beta) & x_h \\ 0 & -1 & y_h \\ 0 & 0 & 1 \end{bmatrix}$$

1.5.4 Stråleligninger og stereoskopi

I de forrige afsnit blev det beskrevet, hvordan objektkoordinater kan konverteres til billedkoordinater. Normalt vil en stereoskopisk beregning starte med et kendt objektpunkt, og ud fra stråleligningerne beregne de tilsvarende billedkoordinater i hvert billede. For at fjerne vertikale parallakser skal de to billeder flyttes i forhold til hinanden, så de to billedpunkter placeres på samme vandrette linie på skærmen.

De indre orienteringsparametre, der er beskrevet i det forrige afsnit, er unikke for hvert kamera, men er ens for alle billeder, der er taget med dette kamera. De ydre orienteringsparametre er unikke for hvert billede. Før et billede korrekt kan benyttes i en stereomodel, skal både de ydre og de indre orienteringsparametre være kendte. Det er dog muligt at skabe stereoskopiske billeder uden at kende orienteringsparametrene. Ved manuelt at flytte de to billeder i forhold til hinanden, kan de placeres således, at de skaber en korrekt stereoeffekt. Et program, der tillader at flytte rundt på de stereoskopiske billeder uden at være bundet af stråleligningerne, siges at være i komparatortilstand.

En sådan komparatortilstand bliver ofte benyttet til at kalibrere billeder eller kameraer, dvs. fastsætte de ydre og/eller indre orienteringsparametre. Ud fra fællespunkter i de to billeder kan billedernes indbyrdes orientering bestemmes. For at opbygge en stereomodel, hvori der kan udføres 3D-målinger, kræves dog at objektkoordinaterne kendes for de valgte punkter. Ved at udmåle et givent antal punkter i et billede, hvortil det tilhørende objektpunkt er kendt, kan samtlige orienteringsparametre beregnes (se fx [R3]).

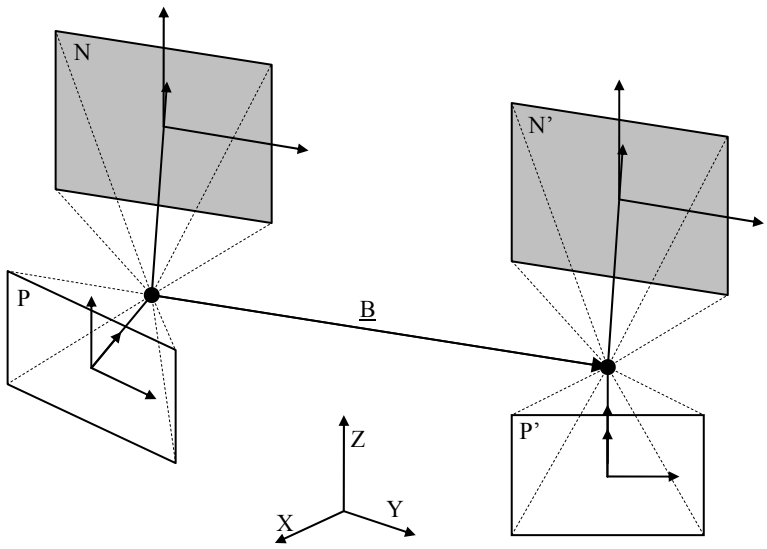
1.6 BILLEDNORMALISERING

En stereomodel vil kun give et perfekt stereobillede, hvis de to billeder, der danner modellen, er taget med et fortegningsfrit kamera, der var orienteret på præcis den samme måde, da de to billeder blev taget. Dvs. fotograferingsretningen skal være ens for de to billeder og vinkelret på basis, som er vektoren, der løber fra det ene projektionscenter til det andet.

Almindeligvis vil kameraet have været orienteret en anelse forskelligt, da de to billeder blev taget, og bl.a. kameraets linse vil give fortegninger i billederne, som beskrevet i afsnit 1.5.3. Sådanne billeder kan give udmærkede stereoeffekter i området omkring det vandrende mærke, hvor de to billeder er orienteret korrekt i forhold til hinanden. Kigger man derimod på andre dele af billedet, vil variationer i rotationen af kameraerne samt fx linseforvrængninger ofte forårsage vertikale parallakser, som ødelægger stereoeffekten.

Linseforvrængninger som fx den radialsymmetriske fortegning kan fjernes ved at opbygge et nyt billede ud fra det gamle, hvor der tages højde for disse forvrængninger. Denne transformering er ikke lineær og vil sjældent udføres af programmet, der viser stereobilledet. Ofte vil man vælge ikke at tage højde for disse forvrængninger. I dette projekt vil billederne ikke blive forbehandlet for at fjerne de radialsymmetriske forvrængninger, men disse vil stadig tages højde for i konverteringen fra objektkoordinater til billedkoordinater.

Hvis to billeder er taget med noget nær parallelle fotograferingsretninger kan de parvist normaliseres. Med normalisering menes, at billederne transformeres, så kameraorienteringerne er ens i de to billeder. Normalisering udføres ved at projicere de to billeder gennem hvert sit projektionscenter og ned på et fælles plan. Dette plan skal være parallelt med basis. Normalen skal have en retning, der er så tæt som muligt på z-aksen i de to kamerakoordinatsystemer (se Figur 1-8).



Figur 1-8: Normalisering af billeder.

De to billeder P og P' kan konverteres til de tilsvarende normaliserede billeder N og N', ved at ændre rotationen af billedernes kamerakoordinatsystem. Dvs. de originale rotationer (Ω, Φ, K) og (Ω', Φ', K') for henholdsvis billede P og P' skal ændres til den fælles rotation (θ_x, θ_y, θ_z) af kamerakoordinatsystemerne for de normaliserede billeder:

Basisvektoren $\underline{B}$ bestemmes ud fra projektionscenteret for de to billeder, dvs. (X_0, Y_0, Z_0) og (X_0', Y_0', Z_0') for henholdsvis billede P og P'.

$$\underline{B} = \left(B_x, B_y, B_z\right) = \left(X_0' - X_0, Y_0' - Y_0, Z_0' - Z_0\right)$$

Rotationen af kamerakoordinatsystemet for de normaliserede billeder kan bestemmes ud fra basisvektoren. Først roteres omkring Z-aksen, så X-aksen ligger på samme vertikale plan som basisvektoren:

$$\theta_z = \tan^{-1}\left(\frac{B_y}{B_x}\right)$$

Dernæst roteres omkring den roterede Y-akse, så X-aksen er parallel med basen:

$$\theta_y = \tan^{-1}\left(\frac{-B_z}{\sqrt{B_x^{\,2} + B_y^{\,2}}}\right)$$

Til sidst roteres omkring den dobbelt-roterede X-akse, så Z-aksen ligger tæt på z-aksen for de originale kamerakoordinatsystemer. I dette projekt vil middelværdien af de originale rotationer af kamerakoordinatsystemerne benyttes, dvs.:

$$\theta_x = \frac{\Omega + \Omega'}{2}$$

Rotationsmatricerne, der svarer til rotationerne θ_x, θ_y og θ_z, kaldes henholdsvis $\underline{\underline{R}}_{\theta x}$, $\underline{\underline{R}}_{\theta y}$ og $\underline{\underline{R}}_{\theta z}$. Disse bliver opbygget som beskrevet i afsnit 1.5.2. Den samlede rotationsmatrice $\underline{\underline{R}}_B$ forbinder objektkoordinatsystemet til et koordinatsystem, der er parallelt med kamerakoordinatsystemet for de normaliserede billeder:

$$\underline{\underline{R}}_B = \underline{\underline{R}}_{\theta x} \cdot \underline{\underline{R}}_{\theta y} \cdot \underline{\underline{R}}_{\theta z}$$

For at konvertere punktet $\underline{x}_i = (x_i, y_i)$ i billede P til det tilsvarende punkt $\underline{x}_n = (x_n, y_n)$ i billede N, skal $\underline{x}_i$ først konverteres til et punkt $\underline{X} = (X, Y, Z)$ i objektkoordinatsystemet. Ser man bort fra den radialsymmetriske fortegning, kan et punkt X i objektkoordinatsystemet konverteres til det tilsvarende punkt $\underline{x}_i$ i billedkoordinatsystemet ved følgende lineære operation (se afsnit 1.5). Bemærk at formlerne benytter homogene koordinater.

$$\hat{\underline{x}}_i = \underline{\underline{b}} \cdot \underline{\underline{P}} \cdot \underline{\underline{R}} \cdot \underline{\underline{T}} \cdot \hat{\underline{X}}$$

Hvis beregningen vendes om, kan $\underline{X}$ bestemmes ud fra $\underline{x}_i$ som ønsket:

$$\hat{\underline{X}} = \underline{\underline{T}}^{-1} \cdot \underline{\underline{R}}^{-1} \cdot \underline{\underline{P}}^{-1} \cdot \underline{\underline{b}}^{-1} \cdot \hat{\underline{x}}_i$$

Dette punkt i objektkoordinatsystemet kan benyttes til at bestemme det tilsvarende punkt i det normaliserede billede, dvs.

$$\hat{\underline{x}}_n = \underline{\underline{b}} \cdot \underline{\underline{P}} \cdot \underline{\underline{R}}_B \cdot \underline{\underline{T}} \cdot \hat{\underline{X}}$$

Hvis de to forrige formler sammensættes, vil translationerne $\underline{\underline{T}} \cdot \underline{\underline{T}}^{-1}$ udgå. Da rotationsmatricen er ortogonal er $\underline{\underline{R}}^{-1} = \underline{\underline{R}}^{t}$. Den samlede lineære operation for at konvertere et billede P til det normaliserede billede N bliver derfor:

$$\hat{\underline{x}}_n = \underline{\underline{b}} \cdot \underline{\underline{P}} \cdot \underline{\underline{R}}_B \cdot \underline{\underline{R}}^{t} \cdot \underline{\underline{P}}^{-1} \cdot \underline{\underline{b}}^{-1} \cdot \hat{\underline{x}}_i$$

I modsætning til fjernelse af den radialsymmetriske fortegning, kan normalisering af billeder udføres lineært. Dette tillader at grafikkort med 3D-acceleratorer kan udføre beregningen.

Konvertering af billede P' til det normaliserede billede N' forløber på tilsvarende måde.

1.7 PROBLEMFORMULERING

Målet med dette projekt er at udvikle et programmodul til stereoskopisk visning af billedpar, med mulighed for at foretage 3D målinger i den stereoskopiske model, samt at vise flere modeller samtidig på skærmen.

1.7.1 Minimumskrav for programmodulet

Programmodulet skal som minimum opfylde følgende krav:

Den stereoskopiske model skal indeholde et vandrende mærke til udpegning af 3D målepunkter. Programmodulet skal omfatte en matematisk kameramodel, der 'realtime' omsætter operatørens input fra f.eks. en mus til 3D bevægelser af dette mærke, enten ved at flytte billederne eller mærket. Mærkets position skal løbende beregnes i såvel objektkoordinater som billedkoordinater. Programmodulet skal desuden håndtere flere stereovinduer simultant, så flere stereomodeller kan vises på skærmen på samme tid. Informationer om billeder, modeller og kameraer skal hentes fra en database.

Det udviklede programmodul skal ikke eksekveres alene, men i stedet benyttes af et andet program, kaldet brugerprogrammet. Der skal udvikles et standardiseret interface med funktioner til kommunikation mellem det udviklede programmodul og dette brugerprogram. Fx skal man via disse kunne sende det vandrende mærke til en bestemt position, kunne bede om billed- eller objektkoordinater til den aktuelle position af det vandrende mærke, overføre orienteringsparametre til stereomodeller m.m. De præcise specifikationer fastsættes i kravspecifikationen.

Der skal desuden programmeres et lille brugerprogram, der i form af en simpel brugerflade kan benyttes til at teste samtlige funktionaliteter i programmodulet.

Der lægges vægt på at programmodulet er velstruktureret, veldokumenteret og gennemtestet med en objektorienteret opbygning, der gør det let at tilføje nye funktionaliteter til programmodulet.

1.7.2 Udvidelser til programmet

Følgende beskriver tiltænkte udvidelser til programmodulet. Det er et ønske, at programmodulet er struktureret med hovedparten af disse udvidelser i tankerne, så implementeringen af de udvidelser, der ikke nås i projektet, simplificeres.

- Programmodulet vil køre på andre platforme end Windows XP. Primært Unix og Linux.
- Stereovinduerne vil anvende forskellige stereoskopiske betragtningssystemer. Fx anaglyph, shutter glasses, polariseret lys, m.m.
- Operatøren vil have mulighed for frit at skifte til samme objektpunkt i flere forskellige stereoskopiske modeller af det samme område, og programmet skal foretage et automatisk skifte til nabomodellen, når det vandrende mærke føres ud over modelafgrænsningen.
- Interfacet til brugerprogrammet vil indeholde funktioner til optegning af 3D kurver og punkter i den stereoskopiske model.

1.8 OPSUMMERING

Dette kapitel introducerer begrebet stereoskopi, som er en metode hvormed to todimensionale billeder vises på hvert sit øje, så de tilsammen danner et tredimensionalt billede kaldet en stereomodel. Der introduceres forskellige metoder, hvormed disse to billeder kan opsplittes, så de ses med hvert sit øje.

Ofte benyttes et såkaldt vandrende mærke til at udføre målinger i stereomodeller. I forbindelse med disse målinger skal et punkt, i den verden stereomodellen viser, konverteres til punkter i hvert af de to billeder. Den matematiske beregning, der ligger bag denne konvertering, bliver gennemgået i detaljer. Der bliver også beskrevet, hvordan to billeder kan normaliseres, så de danner en mere realistisk stereomodel.

Kapitlet afsluttes med problemformuleringen for projektet. Kort beskrevet er formålet med projektet at udvikle et programmodul, der kan vise stereomodeller og udføre målinger i disse.

2 Analyse

Det forrige kapitel analyserer problemstillingen bag dette projekt og beskriver de overordnede mål med det programmodul, der udvikles i projektet. Analysen vil fortsætte i dette kapitel ved kort at kigge nærmere på, hvordan lignende programmer fungerer. Kapitlet afsluttes med kravspecifikationen for projektet.

Dette projekt er en bunden opgave, hvor udvikleren ikke har haft mærkbar indflydelse på, hvilke funktionaliteter det færdige programmodul skal have. Analysen i dette kapitel vil derfor være meget begrænset, og fokus vil i stedet lægges på en gennemgang af kravspecifikationen.

2.1 STEREOSKOPIPROGRAMMER

Stereoskopi er ikke en nyhed indenfor computerverdenen. Der findes mange programmer på markedet i dag, som tilbyder mulighed for at vise stereoskopiske billeder. Udbuddet reduceres dog en hel del, hvis man kun kigger på programmer, der kan vise stereomodeller ud fra bitmap billeder, der orienteres vha. stråleligningerne.

Det har ikke været muligt at finde hverken fulde programmer eller demoer af programmer, der opfylder ovenstående begrænsning og samtidig kan hentes gratis fra nettet. Ser man derimod på lignende kommercielle produkter, findes der mange eksempler på markedet. Et af disse er *ImageStation Stereo Display* (ISSD), som er udviklet af *Z/I Imaging Corporation* [R39]. Dette produkt benyttes i øjeblikket på bl.a. Geoinformatik afdelingen ved IMM på DTU, og det har derfor været muligt at teste programmet.

ISSD er en lille del af en større programpakke kaldet *ImageStation*, som er et fotogrammetrisystem, der tilbyder diverse funktionaliteter til arbejde med primært fly- og satellitbilleder. Ser man på ISSD alene, er programmet en stereoskopisk fremviser, der kan benyttes til at vise højdetaljerede enkeltbilleder eller stereomodeller. ISSD kan indlæse enkeltbilleder direkte fra en fil, mens stereomodeller indlæses fra en specialiseret fil. Denne modelfil kan opbygges i andre af programmerne i ImageStation-programpakken for efterfølgende at blive vist i ISSD. Når programmet viser stereomodeller kan disse blive normaliseret mens de vises.

Programmet tilbyder to primære metoder til navigation i stereotilstand. Standardmetoden er at bevæge det vandrende mærke mens billederne står stille, dette tillader at mærket kan bevæges ud af vinduet, så det forsvinder. Ved den alternative metode, kaldet *roam*, er det vandrende mærke låst til centrum af skærmen, og musebevægelser vil i stedet flytte billederne. I begge tilstande kan brugeren frit zoome ind og ud af billedet. Koordinaterne for positionen af det vandrende mærke kan hele tiden ses på skærmen.

Af indstillingsmuligheder tilbyder programmet bl.a. at indstille hastigheden, hvormed det vandrende mærke bevæges, og konfigurere hvilke knapper, der er bundet til hvilke funktionaliteter. Det er muligt at vælge mellem flere typer af målemærker eller tegne sit eget målemærke og benytte dette. Lysstyrke og kon-

trast af billede og skærm kan modificeres på flere forskellige måder.

Programmet har mange ekstra funktionaliteter. Fx kan brugeren tegne streger i billedet, og disse vil blive placeret på specielle lag oven på billedet. Dette gør, at billedet ikke ændres, og at de ekstra lag kan fjernes, når brugeren ønsker det. De resterende funktionaliteter i ISSD vil ikke blive gennemgået i denne analyse, da de ligger uden for målet med det udviklede programmodul.

ISSD er kun et af mange eksempler på programmer, der arbejder med orienterede stereomodeller. Det har ikke været muligt at få adgang til andre af disse produkter, men producenternes hjemmesider giver i mange tilfælde et godt indtryk af funktionaliteterne i programmet (se [R40], [R41] og [R42]). Det virker til, at de andre lokaliserede programmer ikke adskiller sig meget fra ISSD. Alle er fotogrammetrisystemer, der har udgangspunkt i fly- og satellitbilleder, og de funktionaliteter, der reklameres med, minder om dem, der leveres af ISSD.

2.2 DET UDVIKLEDE PROGRAMMODUL

Selvom, der findes mange stereoskopiprogrammer på markedet i dag, er det ikke lykkedes at finde programmer, som kan erstatte det programmodul, der skal udvikles i dette projekt.

Det udviklede programmodul skal danne kernen i en videre udvikling af specialiserede fotogrammetrimoduler på IMM ved DTU. Dette kræver at modulet er open source, og at andre programmer kan kalde funktioner i modulet. Fælles for de undersøgte programmer er, at de alle er stand-alone programmer. Ingen af programmerne er open source, og der er ingen tegn på at nogen af programmerne leverer mulighed for at kunne kaldes fra andre programmer.

En anden forskel mellem det udviklede programmodul og de analyserede programmer er, at sidstnævnte ser ud til at fokusere på få, detaljerede modeller, hvoraf kun én vises på skærmen af gangen. Det udviklede programmodul skal i stedet være målrettet på at udføre målinger i flere småbilledmodeller, der vises på skærmen samtidig i flere forskellige vinduer.

Selvom der er store forskelle på de to programmer, kan mange af funktionaliteterne fra ISSD stadig benyttes i det udviklede programmodul. Det er Keld Dueholms ønske, at alle de ovenstående nævnte funktionaliteter i ISSD på et tidspunkt skal blive inkorporeret i programmodulet.

Der blev i ISSD observeret enkelte funktionaliteter, der vil forsøges forbedret i det udviklede programmodul. Når man navigerer en stereomodel i standardtilstanden, kan det vandrende mærke bevæges ud af det område af billedet, der vises på skærmen, og på denne måde forsvinde. For at undgå dette vil det udviklede programmodul have en grænsezone i kanten af vinduet. Når det vandrende mærke bevæges til denne grænsezone, vil billeder og mærker flyttes i forhold til vinduet, så det vandrende mærke flyttes nærmere centrum. På denne måde sikres det, at det vandrende mærke forbliver synligt inden i vinduet.

En anden ulempe i ISSD er generering af nye modeller, som skal udføres af andre programmer i programpakken. I det udviklede programmodul vil der stiles efter at simplificere generering af nye modeller. Modulet skal hente modeldata fra en database, der på en simpel og overskuelig måde gemmer data, så data

eventuelt kan indskrives manuelt i databasen. Alternativet er at benytte det funktionsbaserede interface i programmodulet fra et andet program.

Et sidste irritationsmoment, der blev observeret i ISSD, var et hastighedsproblem. ISSD blev testet på en ældre maskine, der havde problemer med især roam-navigationsmetoden. Programmet kunne ikke nå at tegne både billederne og de ekstra lag hurtigt nok op, til at modellen kunne manøvreres flydende. Brugeren tydeligt se, at billederne blev optegnet, og bevægelse i billederne gik meget langsomt. Begge dele var til stor gene for brugeren. Udviklingen af programmodulet skal derfor fokusere meget på en optimal ydeevne, så en lignende situation så vidt muligt kan undgås.

2.3 KRAVSPECIFIKATION

Dette hovedafsnit beskriver i detaljer de krav, som er stillet til det udviklede programmodul. Programmodulet er navngivet SIDM, hvilket er en forkortelse for *Stereo Image Display Module*. Denne betegnelse vil blive benyttet i resten af rapporten. Gennem den iterative udvikling af SIDM er kravspecifikationen blevet omskrevet gentagne gange. Dette hovedafsnit beskriver den kravspecifikation, der var gældende ved projektets afslutning.

Afsnit 2.3.1 til 2.3.5 beskriver de funktionelle krav. Dvs. hvilke funktionaliteter SIDM skal have, hvilke typer input det skal kunne modtage, og hvilket output det skal give.

Afsnit 2.3.6 beskriver funktionelle krav til eventuelle udvidelser af SIDM.

Afsnit 2.3.7 gennemgår de ikke-funktionelle krav til SIDM.

Afsnit 2.3.8 beskriver krav til det medfølgende brugerprogram, både de funktionelle og de ikke-funktionelle.

2.3.1 Grafikvinduer

Målet med SIDM er at vise og behandle stereoskopiske billedpar. Programmet skal kunne administrere flere åbne grafikvinduer, der hver især viser et sådant billedpar. Virkemåden af – og kommunikationen med et grafikvindue afhænger af hvilken tilstand, vinduet befinder sig i. Et vindues tilstand afhænger af flere forskellige tilstandsvariable, der hver især kan have en ud af to værdier. Nedenfor beskrives de mulige tilstandsvariable, og hvordan disse påvirker funktionaliteten af et grafikvindue. Med mindre andet er angivet, er en given tilstandsvariabel uafhængig af de andre variable. Formålet med at definere disse tilstandsvariable er, at simplificere senere beskrivelser af programmets virkemåde.

Modeltilstand eller komparatortilstand: Modeltilstand benyttes til at vise orienterede stereomodeller. For at et stereoskopisk billedpar kan vises i modeltilstand, skal både de ydre og indre orienteringsparametre være kendte. I denne tilstand vil målemærkerne tilsammen udgøre et vandrende mærke, hvor tilhørende objektkoordinater vil være kendte, og bevægelsesinput vil påvirke disse direkte. Ud fra objektkoordinaterne beregnes målemærkernes billedkoordinater. I komparatortilstand benyttes ingen orienteringsparametre og objektkoordinaterne kendes derfor ikke. Bevægelsesinput påvirker billedkoordinaterne direkte.

Musemarkør 'låst' eller 'fri': I låst tilstand er musemarkøren usynlig, og musebevægelser flytter målemærkerne i stedet for markøren. I fri tilstand fungerer markøren som en almindelig markør i operativsystemet, dvs. den kan bevæges frit over hele skærmen uden at påvirke mærket.

Mærke 'låst' eller 'frit': Hvis mærket er låst, vil de to målemærker være centreret omkring centrum af vinduet. Bevægelsesinput vil flytte rundt på billederne i stedet for målemærkerne. Hvis mærket er frit, kan målemærkerne til en vis grænse bevæges frit indenfor vinduet. Når et målemærke nærmer sig kanten af vinduet skal billeder og målemærker flyttes samlet, så centrum af målemærkerne bevæges mod midten af vinduet. I de fleste tilfælde vil billederne forblive stillestående i forhold til vinduet, mens bevægelsesinput flytter rundt på målemærkerne. Den ene undtagelse til dette er hvis det ene billede er låst (se nedenfor). Den anden undtagelse er under modeltilstand, hvor restfejl i vertikale parallakser korrigeres ved at justere på det ene billedes vertikale placering, således at de to målemærker placeres på samme vandrette linie.

Billede 'låst' eller 'frit': Disse tilstande er kun aktuelle, når vinduet befinder sig i komparatortilstand. I låst tilstand er det aktive billede fastlåst til sit målemærke. Hvis fx det venstre billede er aktivt og låst, vil målemærket i det venstre billede ikke kunne ændre position i forhold til det venstre billede. I 'frit billede tilstand' er ingen af billederne låst til sit målemærke.

Aktive billede 'venstre' eller 'højre': Disse tilstande er kun aktuelle når vinduet befinder sig i komparatortilstand. Det aktive billede benyttes til bestemmelse af hvilket billede, der låses til sit målemærke, når vinduet er i 'låst billede tilstand'.

Stereoskopisk billedpar eller enkeltbillede: Et vindue vil almindeligvis vise et stereoskopisk billedpar. I komparatortilstand er det dog muligt kun at åbne et enkelt billede, i hvilket tilfælde vinduet ikke benytter stereo og kun et enkelt målemærke vises.

2.3.2 Direkte input til grafikvindue

Det skal være muligt for brugeren af SIDM, direkte at påvirke et grafikvindue vha. tastatur og mus.

Bevægelse af musen vil være den primære metode til navigation: Når markøren er låst, skal musebevægelser flytte målemærkerne i forhold til billedet, i et plan parallelt med skærmen.

Herudover skal det være muligt at binde en tastaturtast, en museknap eller en retning på musehjulet til en af følgende kommandoer:

- Skift mellem låst og fri musemarkør.
- Skift mellem låst og frit mærke.
- Skift mellem låst og frit billede (kun aktuelt i komparatortilstand).
- Bevæge det vandrende mærke i forhold til billedet i alle retninger. Dvs. op, ned, højre, venstre, ud af skærmen og ind i skærmen. Hvis vinduet er i komparatortilstand vil bevægelse af det vandrende mærke ind i eller ud af

skærmen, i stedet bevæge de to målemærker mod hinanden eller væk fra hinanden.

- Ændre hastigheden hvormed det vandrende mærke bevæges.
- Zoom ud af eller ind i billedet.
- Forstørre eller formindske mærket.
- Aktivere det såkaldte primære input til brugerprogrammet. Når det primære input aktiveres, dvs. tasten trykkes eller slippes, vil brugerprogrammet informeres om dette, samtidig med at målemærkernes koordinater medsendes. Det primære input er tiltænkt benyttet til indlæsning af koordinater til brugerprogrammet. Der eksisterer kun ét primært input.
- Aktivere et såkaldt sekundært input til brugerprogrammet. Brugerprogrammet vil kun informeres, når en tast bundet til sekundært input trykkes ned (men ikke når tasten slippes igen), og der vil ikke medsendes koordinater. SIDM kan definere et vilkårligt antal sekundære input.

2.3.3 Output i grafikvindue

Et grafikvindue skal vise et eller to billeder, med hvert sit tilhørende målemærke, samt tekst, der beskriver den nuværende situation i billedet. Det skal være muligt at ændre udseende på målemærkerne. Hvis et vindue indeholder to billeder skal både billederne og målemærkerne vises stereoskopisk. Som udgangspunkt benyttes anaglyph-metoden til den stereoskopiske fremvisning.

Programmet skal som minimum kunne indlæse billeder fra følgende formater: BMP, JPEG, PNG, TGA og TIFF.

Et vindue skal kunne vise følgende information til brugeren.

- Det vandrende mærkes objektkoordinater, forudsat vinduet er i modeltilstand.
- Målemærkernes billedkoordinater i hvert af de to billeder. Dvs. pixelpositionen af det venstre målemærke i det venstre billede og pixelpositionen af den højre målemærke i det højre billede.
- Navnet på den indlæste model eller navne på de indlæste billeder hvis billederne ikke er registreret som en model.
- Hvilke tilstande vinduet er i: Model- eller komparatortilstand, frit eller låst mærke, frit eller låst billede, og hvilket billede, der er aktivt.

2.3.4 Database

Information om modeller, billeder og de kameraer, der har taget billederne, skal kunne indskrives i og læses fra en database. På en Windows platform kunne dette fx være en Microsoft Access database.

Efterfølgende beskrives hvilken information, der skal eksistere i denne database. Præcis hvordan denne information struktureres er op til udvikleren.

For hvert billede skal der kendes:

- Billednavn.
- Sti og filnavn på billedfil.
- Hvilket kamera, der har taget billedet.
- Kameraets placering og orientering i objektkoordinater.
- Målemærkets startposition i billedkoordinater hvis dette billede åbnes i komparatortilstand.
- Billedets startzoomfaktor, der benyttes, hvis dette billede åbnes i komparatortilstand.

For hvert kamera skal der kendes:

- Kameranavn.
- Kamerakonstanten.
- Fire parametre til at definere den radialsymmetriske fortegning.
- To parametre til at definere den tangentielle fortegning. Disse parametre vil ikke benyttes i den nuværende version af SIDM, men databasen skal indeholder dem til fremtid benyttelse.
- Placering af hovedpunktet i billedkoordinater.
- To parametre til at definere affine deformationer af billeder, der er taget med dette kamera.

For hver model skal der kendes:

- Modelnavn.
- Hvilke to billeder, der er henholdsvis venstre og højre billede i modellen.
- Det vandrende mærkes startposition i objektkoordinater.
- Billedernes startzoomfaktor.

2.3.5 Funktionsinterface

For at benytte SIDM kræves et andet program, der definerer hvornår et vindue skal åbnes, hvilken model, der skal indlæses osv. Dette eksterne program kaldes i denne rapport for brugerprogrammet.

Behandling af grafikvinduer: Funktioner skal eksistere til at åbne et nyt vindue, lukke et specifikt åbent vindue og modtage en liste over åbne vinduer. Det vindue, brugeren sidst har benyttet, defineres som det aktive vindue. Der skal implementeres funktioner til at læse hvilket vindue, der er aktivt, og skifte det aktive vindue til et andet.

Tilstandsskift: SIDM skal indeholde funktioner til at ændre et vindues tilstand og læse hvilken tilstand, et vindue befinder sig i. Dette inkluderer model/komparatortilstand, låst/fri musemarkør, låst/frit mærke, låst/frit billede og hvilket af de to billeder, der er aktivt.

Billeder og modeller: Det skal via funktionskald være muligt at åbne et til to billeder eller én model i et grafikvindue. Indstillinger for disse billeder og modeller skal SIDM hente fra databasen. Brugerprogrammet skal også kunne læse hvilken model og hvilke billeder, der er åbne i øjeblikket.

Koordinater: Det skal ligeledes via funktionskald være muligt at flytte målemærkernes position til et andet koordinatsæt samt læse målemærkernes nuværende koordinater. I modeltilstand skal det vandrende mærke kunne flyttes til en given objektkoordinatposition, og både objektkoordinater og billedkoordinater skal kunne læses. I komparatortilstand eksisterer objektkoordinaterne ikke, og målemærkerne flyttes ud fra billedkoordinater. Hvis to billeder er åbne, vil målemærkerne flyttes ud fra billedkoordinater i det ene billede. Dvs. det ene målemærke flyttes til den angivne position i det ene billede, og det andet målemærke flyttes den samme afstand i det andet billede.

SIDM kalder brugerprogrammet: Brugerprogrammet skal have mulighed for at definere egne funktioner, der kaldes af SIDM, når en given situation indtræffer:

- Når det aktive billede skifter: Information om hvilket billede, der nu er blevet aktivt medsendes.
- Når målemærkerne skifter position, eller når det primære input til brugerprogrammet aktiveres: Det vandrendes mærkes objekt- og billedkoordinater medsendes. Der medsendes desuden information om tilstanden af den knap eller tast, der er bundet til det primære input. Dvs. hvorvidt knappen lige er blevet trykket ned, holdes nede, lige er blevet sluppet, eller ikke er trykket ned.
- Når sekundært input til brugerprogrammet bliver aktiveret: Information om hvilken sekundære input funktion, der er aktiveret, medsendes.

Database: SIDM skal give mulighed for at brugerprogrammet kan læse fra og skrive til databasen. For både kameraer, billeder og modeller skal funktioner eksistere til at: Læse data om et givent element, modificere data for et givent element og tilføje et nyt element til databasen. Desuden skal der kunne foretages en søgning på relevante parametre for hvert element.

2.3.6 Udvidelser

Det er hensigten med SIDM, at det senere skal udvides med yderligere funktionaliteter. Dette afsnit beskriver nogle af disse tiltænkte udvidelser. Programmet bør designes med disse udvidelser i tankerne, så de vil være simple senere at implementere. Hvis tiden tillader det, kan nogle af disse udvidelser også implementeres som en del af dette projekt.

Normalisering af billeder: Indtil videre er det antaget, at de benyttede billeder vises på skærmen uden nogen forbehandling. Hvis de to billeder der udgør en

model ikke er taget med parallelle fotograferingsretninger, vil det ikke være muligt at se modellen stereoskopisk korrekt. SIDM skal udvides til at normalisere billederne, inden de vises på skærmen, så et acceptabelt stereoskopisk billede kan skabes.

Automatisk skift mellem modeller: Brugeren af SIDM vil ofte arbejde med flere modeller, der viser det samme generelle område af verden. Fx kan flere modeller arbejde sammen til at vise et større område, eller modeller kan være taget med forskellige afstande til målet. En vigtig udvidelse til SIDM er, at programmet automatisk skal kunne skifte til samme objektpunkt i andre modeller af det samme område. Dette skift skal kunne aktiveres på tre måder:

- Hvis brugeren skifter mellem to vinduer, der begge har åbne modeller af det samme generelle område, skal det vandrende mærke om muligt flyttes, så det har samme objektposition i det nye vindue. Hvis objektkoordinaterne af det vandrende mærke i det forladte vindue ikke findes i det nye vindue, skal det vandrende mærke ikke flyttes.
- Brugeren skal via fx en menu, kunne frembringe en liste over de modeller, der indeholder det vandrende mærkes nuværende objektkoordinater. Et valg i denne liste skal indlæse den nye model, og det vandrende mærke skal flyttes til det angivne objektpunkt.
- Når et af målemærkerne bevæges udenfor sit billede, skal programmet undersøge, om en anden model dækker dette område. Hvis dette er tilfældet, skal den nye model indlæses, og det vandrende mærke flyttes til den samme objektkoordinat.

Alternative stereoskopiske betragtningssystemer: Det primære stereoskopiske betragtningssystem i SIDM vil være anaglyph. Programmet bør senere udvides til også at kunne benytte 'page flipping' og evt. andre betragtningssystemer.

3D-streger i modellen: En interessant mulighed for brugeren af SIDM, ville være at kunne tegne 3D punkter og kurver i modellen. Dette kunne fx være højdekurver i en model over et landskab. Disse punkter og kurver bør kunne tegnes i flere forskellige lag, hvor hvert lag kan slås til eller fra efter brugerens ønske. Interfacet til brugerprogrammet skal derfor udvides med funktioner til at tegne disse punkter eller kurver. Følgende funktioner bør implementeres:

- Funktioner til at vælge hvilket eller hvilke lag, der er synlige, samt hvilket lag, der tegnes på i øjeblikket.
- Funktioner til at tegne en streg fra et punkt til et andet, samt funktioner til at vælge stregfarve og stregtykkelse.
- Funktioner til at tegne et symbol i et givent koordinatsæt, samt funktioner til at vælge mellem forskellige symboler (prik, kryds, cirkel, m.v.).

2.3.7 Ikke-funktionelle krav

Hvor de tidligere afsnit har beskrevet SIDM's funktionelle krav, vil dette afsnit beskrive de ikke-funktionelle krav.

Ydeevne: Det er meget svært at fastsætte et relevant og let måleligt krav for programmets ydeevne. Hvor godt programmet kører, vil desuden afhænge af den hardware, der er til rådighed, samt af størrelsen af de indlæste billeder. Kravet til dette programs ydeevne vil derfor blive formuleret meget overfladisk:

SIDM skal på en almindelig, moderne PC kunne arbejde med to billeder på hver 8 megapixel uden mærkbare forsinkelser mellem input og respons. Dvs. at programmet skal reagere på alle brugerinput, uden brugeren registrerer forsinkelser. Ved almindeligt brug skal billedet optegnes uden synlige forsinkelser og med en frekvens på minimum 50 Hz. De fleste funktionskald fra brugerprogrammet skal også forløbe uden mærkbare forsinkelser. Undtagelserne er åbning af nye billeder og modeller, der nødvendigvis vil tage noget tid. Denne tid bør dog også begrænses så meget som muligt. Definitionen for en moderne PC sættes til en 3000 MHz processor, 512 Mb ram og et 3D accelereret grafikkort med 128 Mb ram.

Et mere konkret krav til ydeevnen er, at programmet i stilstand ikke må genere kørslen af andre programmer. Dvs. når programmet ikke indlæser billeder, og brugeren ikke sender ny input til programmet, skal dette ikke bruge processorressourcer.

Hukommelsesforbrug: Grundet antallet og størrelsen af billederne vil SIDM komme til at arbejde med meget store datamængder. Det er et krav, at data i denne størrelse slettes fra hukommelsen, når det ikke længere skal benyttes. Det er desuden et krav, at programmet ikke har nogen former for memory leaks. Ud over dette er der ingen krav til programmets hukommelsesforbrug.

Platform: SIDM skal kunne køre på et moderne Windows-system. Dvs. på en maskine, der kører Windows XP. En tiltænkt udvidelse til programmet er, at det også skal kunne køre på andre operativsystemer; primært i andre versioner af Windows samt på Unix og Linux maskiner.

Programdesign: Programmets struktur skal designes med det formål, at simplificere tilføjelsen af nye - og ændring af eksisterende - funktionaliteter. Programmet bør derfor have en modulær opbygning, hvor kommunikation mellem modulerne skal gå gennem et veldefineret interface.

Kodestruktur: Det skal være så nemt som muligt for en ny programmør at sætte sig ind i programkoden. Det er derfor vigtigt at kildekoden er velstruktureret og veldokumenteret, samt at den indeholder brugbare kommentarer.

Brug af eksterne programbiblioteker: SIDM må ikke være afhængig af kommercielle programmer eller programbiblioteker, der ikke er en del af operativsystemet. Dvs. det skal være tilladt at anvende de benyttede biblioteker uden at betale for det.

2.3.8 Demobrugerprogram

SIDM kan ikke eksekveres alene men skal startes og administreres fra et brugerprogram (se afsnit 2.3.5). Det er derfor en del af dette projekt at udvikle en simpel demonstration af SIDM i form af et simpelt brugerprogram. Dette program kaldes for fremtiden demobrugerprogrammet.

Funktionelle krav til demobrugerprogrammet: Det skal via demobrugerprogrammet være muligt på en simpel måde at afprøve samtlige interfacefunktioner i SIDM. Der stilles ingen yderligere krav til demobrugerprogrammets udseende eller opbygning.

Ikke-funktionelle krav til demobrugerprogrammet: Demobrugerprogrammet skal kunne køre på en Windows XP. Der stilles ingen specielle krav til design og kodestruktur i dette program, men koden skal selvfølgelig stadig være gennemskuelig for en anden programmør. Demobrugerprogrammet må ligesom SIDM ikke være afhængige af kommercielle programmer eller biblioteker, der ikke er en del af Windows XP.

2.4 OPSUMMERING

Dette kapitel indledes med en analyse af programmer, der minder om det programmodul, som skal udvikles i projektet. Da programmodulets funktionaliteter er stort set fastlagte fra start, er denne analyse meget kort.

Det udviklede programmodul navngives SIDM, hvilket er en forkortelse for *Stereo Image Display Module*. Hovedparten af kapitlet gennemgår kravspecifikationen til SIDM, hvoraf hovedtrækkene vil blive opsummeret her:

Programmodulet skal kunne åbne et eller flere vinduer, der hver især kan vise en stereomodel. Brugeren skal via musen og tastaturtryk kunne bevæge et vandrende mærke rundt i modellen, og hele tiden se koordinaterne for dette mærke. Som alternativ til stereomodellen skal et vindue kunne vise billeder i såkaldt komparatortilstand, hvor de to billeder kan bevæges uafhængigt af hinanden. Information om billeder og modeller skal hentes fra en database.

SIDM er ikke et stand-alone program men er et modul, der skal køres fra et andet program, kaldet brugerprogrammet. Interfacet til dette brugerprogram skal indeholde funktioner, der kan påvirke stereomodellerne på forskellig vis og kommunikere med databasen. Som en del af dette projekt skal et simpelt brugerprogram udvikles til at teste samtlige funktionaliteter i SIDM.

3 Design

Opbygningen af SIDM defineres i designfasen, som bliver gennemgået i dette kapitel. Designet er objektorienteret, men holdes på et overordnet, ikke-implementeringsspecifikt niveau. Dette tillader, at designet teoretisk set kan realiseres i et hvilket som helst objektorienteret programmeringssprog.

3.1 UDVIKLINGSPROCES

Udviklingen af SIDM er baseret på *The Unified Software Development Process* (USPD), også kendt som *The Unified Process* (UP) [R2], [R6]. Hvor USPD er meget velegnet til de fleste større projekter, er den på nogle områder lidt omfattende til et enkeltmands-projekt. Udviklingen af SIDM vil derfor på flere områder afvige fra den officielle standard.

De fleste udviklingsprocesser opdeles som et minimum i følgende faser eller arbejdsaktiviteter:

- **Analyse:** Programmets funktionaliteter samt øvrige krav til programmet analyseres og fastsættes i samarbejde mellem udvikler og bruger.
- **Design:** Ud fra kravene tegnes designet af programmet. Dette design afhænger ikke af det sprog, der senere skal benyttes til implementeringen.
- **Implementering:** Designet realiseres i et udviklingssprog.
- **Test:** Programmet testes for fejl, og der undersøges, hvorvidt programmet opfylder de stillede krav.

I udviklingsprocessen kendt som *the simple waterfall process* vil disse eller lignende aktiviteter blive gennemgået en efter en, og når den sidste fase er overstået, er programmet færdigt. I praksis er denne simple udviklingsmetode dog kun anvendelig i de færreste tilfælde. USPD følger i stedet en iterativ og *incremental* (gradvist voksende) udvikling. Dette vil sige, at de fire ovenstående faser gennemkøres gentagne gange, og for hver iteration vokser programmet.

Designet af SIDM vil være meget afhængigt af de benyttede programbiblioteker, og disse vil afhænge af det valgte implementeringssprog. I stedet for at tage højde for implementeringssprog i designfasen, er det valgt at tilføje yderligere to faser mellem design og implementering:

- **Implementeringsspecifik analyse:** Analyse og valg af hvilket programmeringssprog og hvilke programbiblioteker, der skal benyttes. Denne fase behandles i rapporten i kapitel 4.
- **Implementeringsspecifikt design:** Ændring i designet, så det tager højde for de benyttede programpakker. I denne rapport vil det implementeringsspecifikke design behandles sideløbende med selve implementeringen i kapitel 5.

Disse seks faser vil ikke have lige meget fokus i hver iteration gennem udviklingen af projektet. De første iterationer vil næsten kun fokusere på de implementeringsuafhængige analyse- og designfaser, mens de sidste iterationer foku-

serer primært på de sidste faser. Fx har det implementeringsuafhængige design, der beskrives i dette kapitel, stort set være umodificeret i den sidste halvdel af projektet. I appendiks B findes en beskrivelse af alle gennemgåede iterationer.

I dette kapitel benyttes *Unified Modelling Language* (UML). UML er et modelleringssprog, dvs. en definition af hvordan specifikke diagramtyper skal optegnes. Det er ikke et krav, at læseren er bekendt med UML, for at forstå dette kapitel. Første gang en given UML diagramtype bliver brugt i kapitlet, vil diagramtypen blive forklaret i detaljer.

3.2 USE CASES

USPD baseres på *use cases*, der er opbygget ud fra den funktionelle del af kravspecifikationen. En use case er en teknik, der benyttes til at definere de potentielle krav for et system. Hver use case beskriver et eller flere hændelsesforløb, der viser hvordan systemet skal samarbejde med brugeren for at opnå et specifikt mål. En use case indeholder normalt ingen programmeringsmæssige tekniske betegnelser, men skrives i stedet på et sprog, der kan forstås af brugeren.

Inden opbygningen af en use case bliver gennemgået, er det nødvendigt at forstå begrebet *actor*. En actor er i denne sammenhæng en person, en organisation eller et computersystem, der kommunikerer med det udviklede produkt. Ofte vil et produkt have mange actors. En typisk netværksapplikation kan fx have en bruger, en klient, en server og en serveradministrator som actors. SIDM har kun to actors: Brugeren og systemet.

Samtlige use cases til dette projekt kan findes i appendiks E.1. Figur 3-1 viser et eksempel på en af disse use cases. En use case indledes med en overskrift, der viser use casens ID og beskriver hvilken funktionalitet, der implementeres i denne use case. Normalt ville overskriften efterfølges af en definition af *primary actor*, dvs. hvilken actor, der aktiverer funktionaliteten. Da systemet aldrig selv aktiverer en sådan funktionalitet, vil brugeren være primary actor i samtlige use cases, og denne linie er derfor udeladt.

Foruden overskriften er de use cases, der er tegnet i dette projekt, delt op i fem områder:

Stakeholders & Interests: En *stakeholder* er i denne sammenhæng en person eller organisation, der har interesse i den beskrevne funktionalitet. For et program, der administrer et salg af en vare i en butik, kan der fx eksistere følgende stakeholders: Kunden, sælgeren, firmaet og staten (registrering af salg til beregning af skatter). Dette afsnit definerer hvad hver enkelt stakeholder ønsker at opnå ved den beskrevne funktionalitet. SIDM har kun en enkelt stakeholder, nemlig brugeren, og dette afsnit er derfor ganske kort.

Precondition: *Precondition* definerer krav til programmets tilstand, før den beskrevne funktionalitet kan benyttes. Disse krav bliver ikke testet i denne use case, men forventes allerede at være opfyldt.

Postcondition: *Postcondition* beskriver programmets tilstand efter en vellykket gennemførsel af denne use case.

Main Success Scenario: Dette afsnit beskriver trinnene i det almindelige forløb for en vellykket gennemkørsel af denne use case.

Extensions: *Extensions* beskriver alternative forløb ved udvikling af den beskrevne funktionalitet. Dette inkluderer både forløb, der er fejlfrie, og forløb, der afbrydes af fejl. Disse alternative forløb udgrener fra basisforløbet på de specificerede trin. I eksemplet i Figur 3-1 er der fx to udgreninger fra trin 3: Trin 3a og trin 3b. Hver udgrening påbegynder et nyt nummereret forløb, der igen kan grene ud. For at begrænse længden af use cases i denne rapport, tillades at et forløb i extensions henviser til main success scenario som en afslutning på forløbet. Afsluttes et alternativt forløb fx med "Gå til Pkt. 4", menes der, at dette forløb afsluttes som basisforløbet fra trin 4 og frem.

UC14 Indlæsning af billeder, der ikke tilhører en model

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at åbne et eller to billeder uden for en model.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Den eller de ønskede billeder er nu åbne.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at indlæse et eller to billeder. Billedernes id medsendes.
2. Billeder eller den model, der allerede er vist i vinduet fjernes.
3. Systemet åbner den eller de ønskede billeder.
4. Systemet skifter om nødvendigt til komparatortilstand.
5. Systemet flytter målemærkerne i forhold til billederne og billederne i forhold til hinanden, så positionen af hvert målemærke i det tilhørende billede svarer til det, der er defineret i billedernes data.

Extensions

3a. Et eller flere af de ønskede id'er findes ikke i databasen.
   1. Systemet informerer i grafikvinduet brugeren om fejl ved indlæsning af det ene eller begge billeder.

3b. Systemet kan ikke finde et eller flere af de ønskede billeder på den i databasen angivne sti, eller en fejl opstår under indlæsning af en billedfil.
   1. Systemet informerer i grafikvinduet brugeren om fejl ved indlæsning af det ene eller begge billeder.

Figur 3-1: Eksempel på en use case.

3.2.1 Use case model

Use cases sammenfattes ofte i en *use case model*, der foruden use cases indeholder *systemsekvensdiagrammer* og *use case diagrammer*: Et systemsekvensdiagram er et billede, der viser handlingsforløbet for en given use case, samt hvordan de forskellige actors kommunikerer. Et use case diagram viser grafisk sammenhængen mellem use cases og deres primary actors.

I dette projekt afbrydes use case modellen efter udviklingen af use cases. Pga. SIDM's begrænsede mængde af actors, menes systemsekvensdiagrammer og use case diagrammer ikke at give nogen nævneværdig hjælp til udviklingen af designet.

3.3 DESIGNBESLUTNINGER

Inden designet af SIDM kan tegnes, er det nødvendigt at tage visse designbeslutninger. Ved udviklingen af et program som SIDM må der hele tidens tages beslutninger omkring designet, og det ligger uden for denne rapport at argumentere for dem alle. Dette hovedafsnit vil i stedet beskrive nogle få udvalget problemstillinger, som alle antages at have større indflydelse på det endelige produkt.

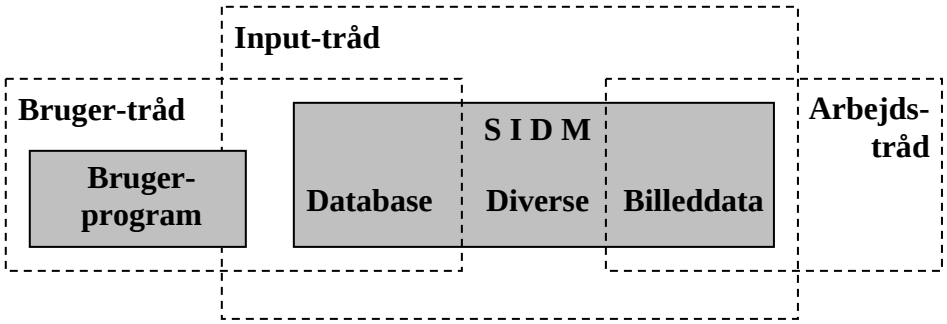
3.3.1 Overordnet valg

Det er hensigten, at SIDM fortsat skal udvikles efter afslutningen af dette projekt. En af de vigtigste designbeslutninger, der er taget i dette projekt, er at alle foreslåede udvidelser på et tidspunkt skal kunne implementeres i SIDM. Alle udvidelser til SIDM skal derfor tages i betragtning under designet, så dette ikke skal omstruktureres for at understøtte senere forbedringer. Den største implikation af dette er, at SIDM ikke må benytte platformspecifik kode. Et af udvidelsespunkterne til SIDM er, at modulet skal kunne køre på diverse platforme som fx Windows, Unix og Linux. Dette valg vil vise sig at have stor indflydelse på hele projektforløbet.

3.3.2 Programtråde

SIDM skal til hver en tid kunne modtage direkte input fra brugeren i et grafikvindue. Modulet er derfor nødt til at eje sin egen programtråd, der venter på input fra brugeren og reagerer, når den modtager en kommando. Det er vigtigt, at denne inputtråd ikke udfører tidskrævende operationer, da den under en sådan operation ikke kan modtage nye brugerinput. Modulet er derfor nødt til også at have en arbejdstråd, der udfører disse tidskrævende operationer. Sideløbende med disse tråde i SIDM vil brugerprogrammet køre i en eller flere tråde, der alle kan kalde SIDM.

Med alle disse tråde, der kan kalde funktioner i SIDM, er det vigtigt, at modulet er synkroniseret korrekt. Dvs. alle kombinationer af kald fra de forskellige tråde skal eksekverer fejlfrit. For at simplificere denne synkronisering bør muligheden for interaktionen mellem tråde begrænses så meget som muligt. Første skridt til dette er at begrænse de områder af programmet, hvor en given tråd kan påvirke data (se Figur 3-2).

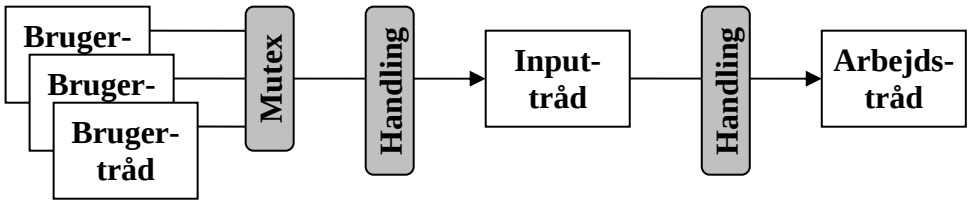


Figur 3-2: Oversigt over hvilke tråde, der kan påvirke data i de forskellige områder af SIDM.

Inputtråden vil være den primære tråd i SIDM, og den har potentiel adgang til alt data i modulet. Desuden kan inputtråden kalde enkelte funktioner i brugerprogrammet. Det er op til designeren af brugerprogrammet at sikre, at disse kald synkroniseres med brugerprogrammets tråde, og at inputtråden frigives med det samme. Alternativt kunne inputtråden opstarte en ny tråd, hver gang en funktion i brugerprogrammet kaldes, og benytte denne tråd til at kalde brugerprogramfunktionerne. Denne løsning vil være rimelig nem at implementere i en kommende iteration.

Arbejdstråden skal som tidligere nævnt udføre tidskrævende operationer i SIDM. For at adskille trådende så meget som muligt er det vigtigt at begrænse arbejdstrådens opgaver mest muligt. Hvis en opgave i et *worst case scenario* kan udføres af inputtråden, så denne igen kan være klar til at modtage input inden for 10ms, skal opgaven ikke overlades til arbejdstråden. Det samme gælder for opgaver, som kræves udført, før inputtråden kan reagere på nye input. I den nuværende version af SIDM er det kun indlæsning og evt. forbehandling af billeder, der bør udføres af arbejdstråden.

Brugerprogramtrådene kalder funktioner til at påvirke grafikvinduer i SIDM. Flere af disse kald vil lede tråden dybt ind i SIDM og derfor kræve synkroniseringer i mange områder af modulet. For at adskille inputtråd og brugertråd så meget som muligt overføres disse kald til inputtråden via handlingskøer (se nedenfor). Brugertråden kan foruden grafikvinduer også påvirke databasedelen i SIDM. Databasedelen vil have et rimeligt begrænset interface, og er derfor nem at sikre mod flere tråde. Det er derfor valgt at give brugertråden adgang til databasedelen.



Figur 3-3: Kommunikation mellem tråde i SIDM. En mutex sikrer, at kun én tråd fra brugerprogrammet kan have adgang til SIDM på et givent tidspunkt. Kommunikation mellem bruger og inputtråd samt mellem inputtråd og arbejdstråd udføres via handlingskøer.

Figur 3-3 viser en oversigt over kommunikation mellem tråde i SIDM. Adgang fra de forskellige tråde i brugerprogrammet begrænses af en *mutual exclusion* (*mutex*). En mutex fungerer som en lås på en ressource, der deles mellem flere tråde. Kun den tråd, der ejer (dvs. har låst) den givne mutex, har adgang til den delte ressource. Ingen anden tråd kan eje mutexen, før den ejende tråd frigiver mutexen igen. Når interfacet til SIDM beskyttes af en mutex, vil det sige, at kun én udefrakommende tråd kan have adgang til SIDM på et givent tidspunkt.

Den aktive brugertråd overfører et kald til inputtråden via et handlingsobjekt, der specificerer den handling, brugerprogrammet ønsker udført. Handlingsobjektet placeres på en kø, som løbende kontrolleres af inputtråden. Når inputtråden finder et objekt i køen, fjernes dette fra køen, og kaldet udføres (se afsnit 3.4.3). Inputtråden kommunikerer på en tilsvarende måde med arbejdstråden. Da den eneste operation, arbejdstråden udfører, er at indlæse og præparere billeder, placeres billedobjekter i stedet for handlingsobjekter på køen.

3.3.3 Fejlbehandling

Programmøren, der designer et programmodul eller -bibliotek kan identificere kørselsfejl i sit modul, men vil generelt ikke vide, hvad der skal gøres ved disse fejl. Personen, der benytter modulet fra sit eget program, er klar over hvilken handling, der skal tages, når en fejl opstår i modulet, men kan ikke selv opdage fejlen. Dette paradoks gør, at det er nødvendigt for modulet, at kunne informere det kaldende brugerprogram, når en fejl opstår. Denne kommunikation kan udføres på flere måder:

Exception handling: I moderne programmeringssprog (C++, Java, C#, osv.) er *exception handling* ofte den valgte metode til at informere om fejl. Når en fejl opstår, kaster modulet en *exception* (throw), og brugerprogrammet skal sørge for at gribe denne fejl. Ulempen ved denne metode er, at ikke alle programmeringssprog understøtter exception handling, og selv hvis metoden understøttes af sproget, kan det være besværligt at konvertere exceptions mellem sprog. Hvis fejlinformation overføres fra modulet til brugerprogrammet i form af exceptions, vil det derfor gøre det sværere eller helt umuligt at kalde modulet fra et andet programmeringssprog.

Signal handling: Den kaldte funktion returnerer en værdi, der repræsenterer en fejl. Denne metode benyttes primært i ældre sprog som fx C. Et problem ved metoden er, at brugerprogrammet er nødt til at teste returværdien af hvert eneste funktionskald i modulet. Disse test er efter udviklerens mening lidt upraktiske: De kræver en hel del ekstra kode og kan nemt overses. Et andet problem ved metoden er, at den begrænser hvilke værdier, der kan returneres fra funktionen, da det er nødvendigt at reservere mindst én returværdi til at repræsentere en fejl.

Fejlkø: Modulet registrerer fejlen som et element på en kø. Brugerprogrammet kan til hver en tid kontrollere, om denne kø er tom, samt trække fejlinformation ud fra køen. Køen bør testes løbende, så den generelt kun holder et fejlelement. Kun hvis samme kald resulterer i flere fejl, bør køen kunne holde flere elementer. Ulempen ved denne metode er, at brugerprogrammet selv skal huske at undersøge for fejl, hvilket kan glemmes.

Fejlfunktion: Modulet kalder en prædefineret funktion i brugerprogrammet når en fejl opstår.

Så længe en fejl i SIDM kan behandles lokalt, uden at brugerprogrammet behøver at bliver informeret eller tage et valg, vil SIDM selv behandle fejlen. Metoden til denne interne fejlbehandling afhænger af programmeringssproget, men vil formentlig være exception handling.

SIDM vil også forsøge at rydde op efter fejl, der ikke fuldt ud kan behandles lokalt. Dvs. SIDM vil prøve at sikre at modulet forsat kører korrekt. Er dette ikke muligt vil SIDM benytte fejlfunktionsmetoden til med det samme at informere brugerprogrammet om problemet. Ligegyldigt om SIDM kan rydde op efter fejlen eller ej, vil modulet informere om fejlen på en fejlkø. Hvis et brugerprogram ønsker at følge nøje med i, hvad der sker i SIDM, skal det løbende kontrollere denne fejlkø. Selv hvis brugerprogrammet aldrig undersøger fejlkøen, vil det stadig blive informeret, når en kritisk fejl opstår via fejlfunktionen.

3.3.4 Programkonfiguration

Det er altid en vigtig beslutning, at bestemme hvordan konfigurationsinformation skal administreres i et program. Ofte skal mange forskellige dele af et program konfigureres på en eller anden måde, så den mest direkte løsning er, at hver klasse gemmer sin egen konfiguration. En anden løsning er at benytte et fælles objekt til at administrere al konfiguration i programmet.

Det ønskes, at SIDM skal indlæse sine konfigurationer fra en konfigurationsfil. Dette vil være nemmest at implementere, hvis konfigurationen bliver behandlet samlet i et enkelt objekt. Dette objekt skal kunne tilgås fra alle dele af programmet, der har brug for konfigurationsinformation.

3.3.5 Databaseforbindelse

Kommunikation med en database kan ofte være en af flaskehalsene i et programs ydeevne. Det er selvfølgelig muligt at lave en hurtig og effektiv forbindelse, men ofte lykkedes dette ikke helt. Især hvis man ønsker en generel løsning, der kan kommunikere med flere forskellige databaser som fx ODBC (se afsnit 4.7), vil den ofte være langsom.

SIDM kommunikerer med databasen, hver gang en ny model eller et billede åbnes. Åbning af et nyt billede vil efter al sandsynlighed tage markant længere tid end kommunikationen med databasen, så en langsom databaseforbindelse vil i denne forbindelse ikke genere ydeevnen mærkbart.

Når et brugerprogram ønsker at kommunikere med databasen, vil dette også administreres af SIDM. Udvikleren af SIDM har ingen chance for at vide, om brugerprogrammets kommunikation med databasen er tidsmæssig kritisk. Dvs. om en optimering af kommunikationshastigheden er ønskværdig. For at være på den sikre side ønskes det om muligt at optimere kommunikationshastigheden med databasen.

En løsning, der kraftigt vil øge kommunikationshastigheden med databasen, er ved programopstart at indlæse hele databasen i computerens hukommelse. Ulempen ved denne løsning er det store forbrug af hukommelse. Selvom kravspecifikationen ikke sætter nogen begrænsninger på hukommelsesforbruget af SIDM, må det alligevel tages med i betragtning, når en sådan løsning overvejes.

Appendiks E.3 beskriver opbygningen af den database, der benyttes i SIDM. Størrelsen og antallet af attributter i hver tabel benyttes i Figur 3-4 til at beregne, hvor meget et billede i gennemsnit vil fylde i hukommelsen:

	Camera	**Images**	**Models**	**Bytes**
Tekststrenge	1	2	1	×32
Integers	1	6	3	×4
Floats (32 bit)	11	6	4	×4
Doubles (64 bit)	0	3	0	×8
Bytes per element	80	136	60	
Elementer per billede	$\times^1/_5$	×1	$\times^2/_3$	
Bytes per billede	16	136	40	=192

Figur 3-4: Beregning af hvor meget databasen vil fylde hvis indlæst i computerens hukommelse. Resultatet siger, at der for hvert billede i databasen benyttes under 200 bytes. Dette svarer til over 5000 billeder per Mb RAM. Et element er en enkelt række i en database, dvs. et kamera, et billede eller en model. Der er gjort følgende antagelser: En gennemsnitlig tekststreng fylder max 32 bytes, i gennemsnit er minimum fem billeder taget med samme kamera, samt at der som minimum går tre billeder på to modeller.

Resultatet af ovenstående beregning viser, at et billede i gennemsnit fylder 192 bytes, hvilket svarer til 5000 billeder per Mb hukommelse. Antallet af billeder i databasen forventes aldrig at komme op i nærheden af 5000, og formentlig heller ikke op på en tiendedel af dette. Pixeldata for to 5-megapixel billeder fylder 30Mb i hukommelsen. Ofte vil man i stereoskopi arbejde med billeder med endnu højere detaljegrad og have flere billeder åbne på samme tid. Hukommelsesforbruget for en indlæst database må derfor siges at være ubetydelig i forhold til forbruget til pixeldata.

Ud fra resultatet af denne analyse vælges det, at SIDM skal indlæse databasen til hukommelsen ved opstart. Når brugerprogrammet skriver til databasen, skal data som udgangspunkt skrives både til hukommelsen og til selve databasen. Dette vil forhindre at data går tabt ved en fejlagtig nedlukning af modulet. I en senere udvidelse af SIDM kan brugeren have større indflydelse på, hvornår data skrives til databasen, og hvornår det kun skrives til computerens hukommelse.

3.4 DESIGNSTRUKTUR

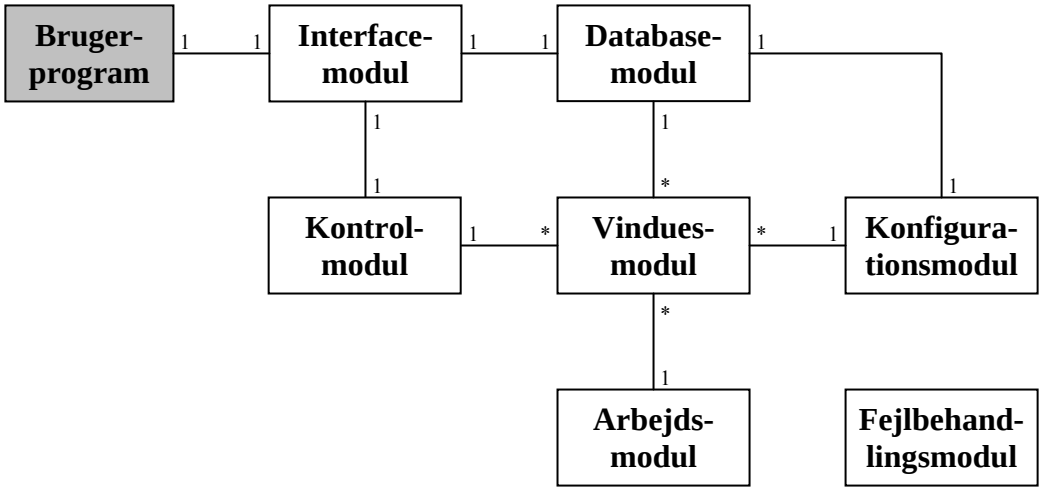
Designet af SIDM bygges op omkring de genererede use cases, mens designbeslutningerne tages med i betragtning. Dette hovedafsnit vil gennemgå det endelige implementeringsuafhængige design af SIDM.

Designet vil i denne rapport kun blive gennemgået overfladisk for de fleste dele af programmet. Enkelte interessante områder i programmet vil dog beskrives i dybere detaljer. Yderligere detaljer om designet kan findes i appendiks E. Det bør bemærkes, at dette design er implementeringsuafhængigt og derfor på mange områder ikke fuldt ud specificeret. For en fyldestgørende beskrivelse af det endelige implementeringsspecifikke design henvises til dokumentationen af programkoden, der kan findes på HTML format på den medfølgende CD.

3.4.1 Modulære design

Det er et ønske, at SIDM skal opbygges modulært, så en given del af programmet er nem at udskifte uden at påvirke resten af programmet. De logiske elementer i en sådan modulær opbygning er programmets klasser. Herudover vælges endnu et abstraktionsniveau, der sammensætter en eller flere klasser til moduler. Opdeling i disse moduler eksisterer ikke på det programmelle niveau men kun i designet som en metode til at simplificere overskueligheden af programmet. Figur 3-5 viser det modulære implementeringsuafhængige design af SIDM.

Stregerne mellem modulerne i dette diagram viser forbindelser (*associations*) mellem modulerne. Tal over stregerne beskriver *multipliciteten*: Fx kan et kontrolmodul forbindes til et vilkårligt antal vinduesmoduler (* betyder 0 eller flere), men et vinduesmodul vil kun forbinde til ét kontrolmodul.



Figur 3-5: Modulært design af SIDM. Brugerprogrammet er ikke en del af SIDM, men medtages her for at illustrere, hvordan SIDM kaldes. Alle moduler kan kommunikere med fejlbehandlingsmodulet, men stregerne er udeladt for at forbedre oversigten.

Hvert modul i SIDM har sit eget afgrænsede arbejdsområde, som kort beskrives nedenstående:

Interfacemodul: Brugerprogrammet tilgår SIDM gennem interfacemodulet, der leder kald videre til enten database-, kontrol- eller fejlbehandlingsmodulet.

Databasemodul: Dette modul administrerer al kommunikation med databasen, og gemmer en version af databasen i hukommelsen.

Kontrolmodul: Kontrolmodulet administrerer de forskellige grafikvinduer i SIDM. Modulet leder desuden kald fra brugerprogrammet til det ønskede vinduesmodul.

Vinduesmodul: Dette primære modul i SIDM administrerer et enkelt vindue.

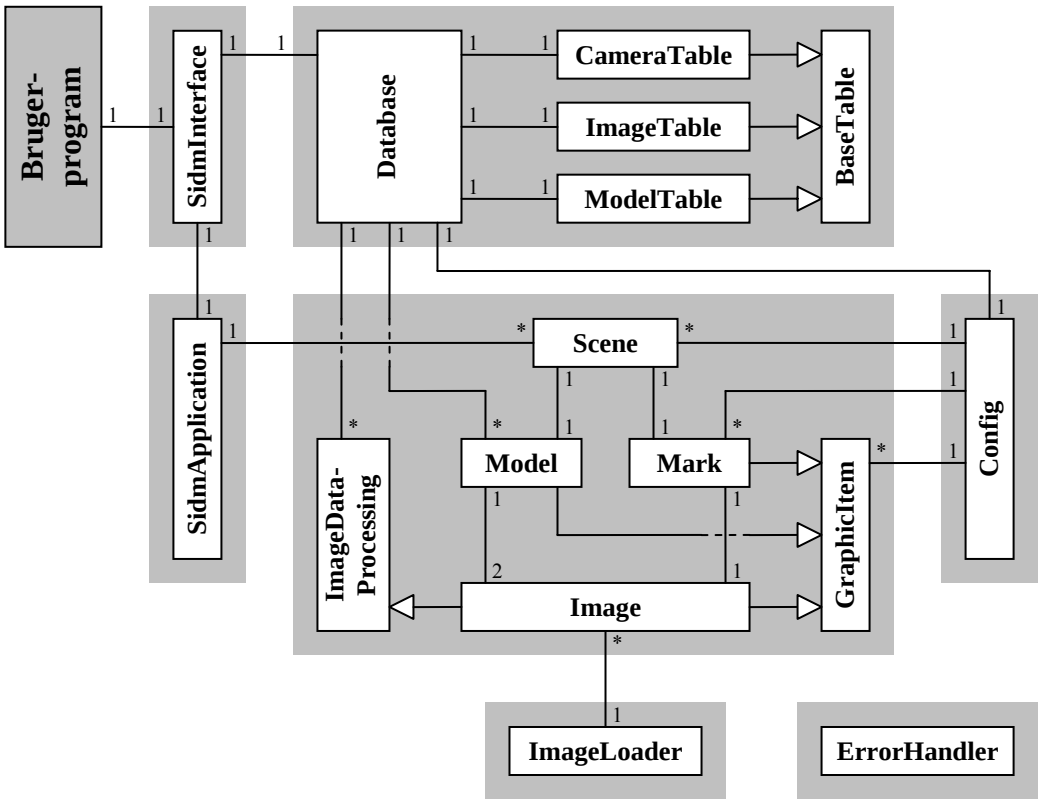
Arbejdsmodul: Arbejdsmodulet udfører tidskrævende opgaver for vinduesmodulet.

Konfigurationsmodul: Dette modul indeholder al konfiguration til SIDM, og det kan indlæse denne konfiguration fra en fil.

Fejlbehandlingsmodul: Fejlbehandlingsmodulet administrerer fejl, der stødes på i de andre moduler i SIDM.

3.4.2 Detaljerede design

Det modulære design er benyttet som basis for det detaljerede design. Figur 3-6 viser klassediagrammet over de primære klasser i SIDM. Yderligere detaljer om alle klasser i SIDM kan findes i Appendiks E.2 Klassestrukturer. I dette appendiks defineres alle member variable, public metoder og flere private metoder i de primære klasser. Herudover identificeres alle andre typer, som implementeres i SIDM.



Figur 3-6: Klassediagram over de primære klasser i SIDM. *ErrorHandler*-klassen kan kaldes fra alle andre klasser i SIDM. Moduler er afgrænset af grå felter, så de kan sammenlignes med det modulære design i Figur 3-5. Modulernes placering i forhold til hinanden svarer til placeringen i det modulære design.

Kun en af klasserne i et modul kan tilgås fra de andre moduler. Denne klasse kaldes modulets interfaceklasse. Der er ingen begrænsning på hvilke klasser i et modul, der kan tilgå andre moduler. Som det ses i figuren, indeholder hovedparten af modulerne kun en enkelt klasse på dette tidspunkt i udviklingen af SIDM. Denne klasse er selvfølgelig modulets interfaceklasse. *Database*-klassen er databasemodulets interface klasse, mens *Scene* er vinduesmodulets interface klasse.

Klassediagrammet og klassernes udseende er udviklet ud fra information beskrevet i de senere afsnit i dette kapitel. Klassediagrammet er medtaget allerede på dette tidspunkt for at introducere klassenavne o.l., der benyttes i efterfølgende afsnit.

3.4.3 Kommunikation mellem moduler

I dette afsnit klargøres hvordan de forskellige moduler i SIDM kommunikerer med interfaceklassen i andre moduler. De fleste af disse kommunikationer vil foregå som almindelige metodekald til en anden klasse og vil derfor ikke bringes op her. I stedet vil dette afsnit gennemgå de få steder, hvor kommunikation ikke følger de almindelige metoder.

Som det ses på Figur 3-6 bliver både *Database*-, *Config*- og *ErrorHandler* klasserne kaldt fra flere forskellige klasser i SIDM. Der må kun eksistere en instans af hver af disse tre klasser, så en pointer til disse instanser skal kendes af flere forskellige klasser spredt rundt omkring i SIDM. I stedet for at overføre disse pointers fra klasse til klasse indtil de når deres mål, implementerer de tre klasser en strukturel opbygning kaldet Singleton design pattern (se afsnit 3.5). Kommunikation med disse klasser er begrænset af en mutex, der sikrer, at to tråde ikke bearbejder de samme data samtidig.

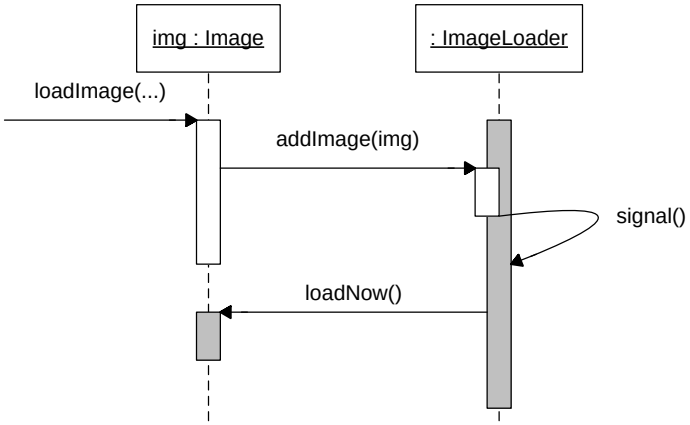
Som beskrevet i afsnit 3.3.2 vil der eksistere tre tråde samtidig i SIDM:

- Brugertråden har adgang til interfacemodulet, databasemodulet, fejlbehandlingsmodulet samt en enkelt metode i kontrolmodulet, der bruges til kommunikation med inputtråden.
- Arbejdstråden har adgang til arbejdsmodulet, fejlbehandlingsmodulet samt en enkelt metode i vinduesmodulet, der bruges til kommunikation med inputtråden.
- Inputtråden har adgang til database-, kontrol-, vindues-, konfigurations- og fejlbehandlingsmodulet, samt en enkelt metode i arbejdsmodulet, der bruges til kommunikation med arbejdstråden.

Inputtråden kommunikerer med arbejdstråden, når et billede skal indlæses fra harddisken og evt. forbehandles, inden det kan vises på skærmen. Denne kommunikation foregår som vist på Figur 3-7.

De lodrette, stiplede streger i figuren beskriver instanser af en klasse. Beskrivelsen i kassen ved toppen af stregen følger formatet: 'Instans : klasse'. Hvis instansen ikke er navngivet, er beskrivelsen blot: ': klasse'. Søjlerne på den lodrette linie kaldes *activation boxes*. De beskriver, hvornår den angivne klasse har fokus, dvs. at en metode i denne klasse er blevet kaldt men endnu ikke har af-

sluttet. Pilene i figuren beskriver metodekald, mens stiplede pile (ikke med i denne figur) viser returnerede værdier.



Figur 3-7: Sekvensdiagram, der viser kommunikation mellem inputtråden (hvide søjler) og arbejdstråden (grå søjler).

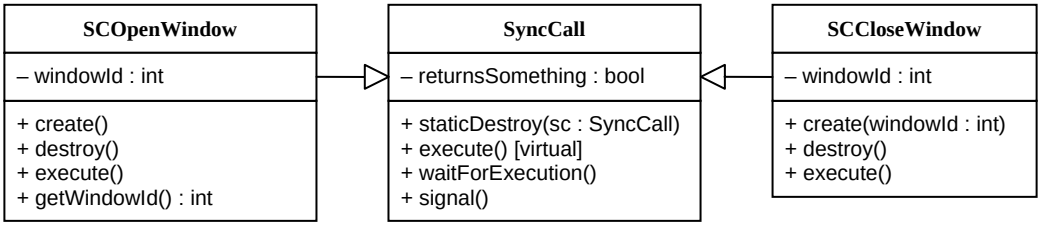
Når en instans af klassen *Image* beordres til at indlæse sit billede (*loadImage* kaldes), kaldes *addImage*() i den aktive instans af *ImageLoader*. Denne metode placerer instansen af *Image* på en kø i *ImageLoader*, signalerer arbejdstråden om, at der er kommet arbejde til den, og afslutter.

Mens arbejdstråden ikke har noget at lave, ligger den i dvale og venter på signal fra inputtråden. Når dette signal modtages, henter arbejdstråden *Image*-objektet fra køen, og kalder metoden *loadNow*() i denne instans. Denne metode indlæser billedet, hvorefter arbejdstråden returnerer til sin dvaletilstand.

En *enumeration* variabel i *Image*-objektet holder styr på i hvilken fase billedet befinder sig. Den kan have følgende værdier: EMPTY, READY_TO_LOAD, ON_QUEUE, LOADING og READY (se Appendiks E.2, *ImageStatus*). Objektet benytter denne variabel til at sikre, at de to tråde ikke tilgår objektet samtidig.

Kommunikationen mellem bruger- og inputtråd er mere avanceret end forbindelsen mellem input- og arbejdstråd. Fællestrækket for de to metoder er, at den kaldende tråd placerer et objekt i en kø, som den ventende tråd udtrækker og behandler. Når bruger- og inputtråd kommunikerer, er det handlingsobjekter, der placeres i køen.

Et handlingsobjekt er en instans af en *syncronized call* klasse. Hver type af kald fra brugertråden til inputtråden implementeres i sin egen syncronized call klasse, og alle disse klasser nedarver fra den abstrakte klasse *SyncCall*. Figur 3-8 viser et klassediagram over to af disse klasser: *SCOpenWindow* og *SCCloseWindow*.



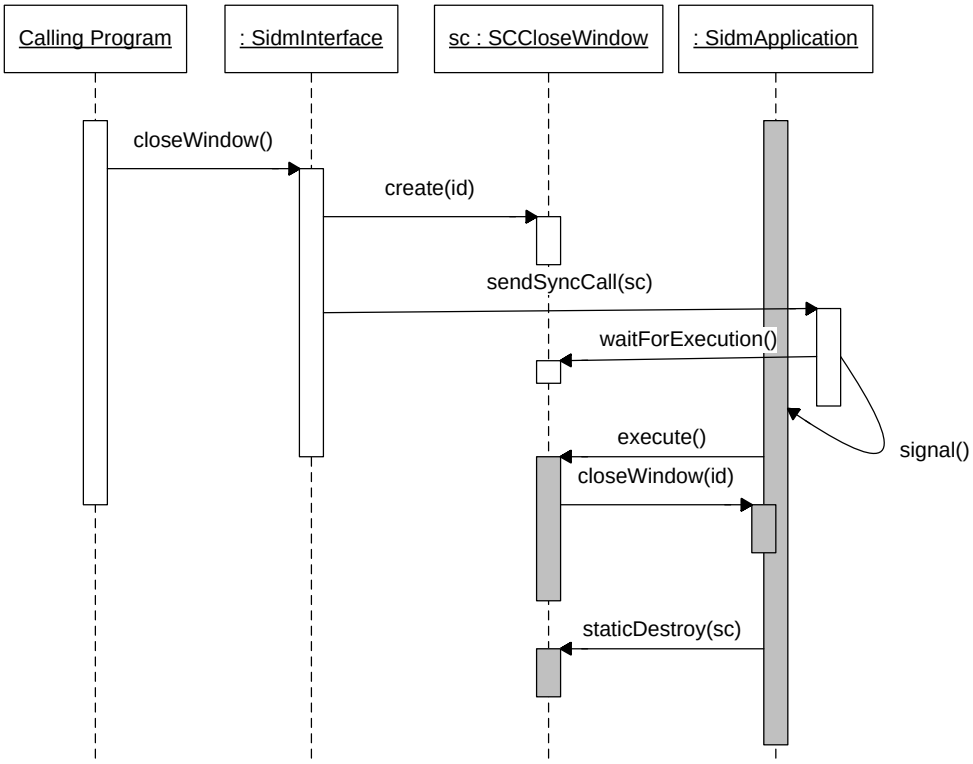
Figur 3-8: Klassediagram over to *syncronized call* (SC) klasser. Begge klasser nedarver fra den abstrakte klasse *SyncCall*.

SCOpenWindow benyttes, når brugerprogrammet kalder *openWindow*(). Kaldet åbner et nyt grafikvindue og returnerer id på dette. *SCCloseWindow* benyttes tilsvarende til kald af *closeWindow*(id). Kaldet lukker vinduet med det angivne id, og returnerer intet.

Kommunikation mellem bruger- og inputtråd kan forløbe på to forskellige måder, alt efter om det ønskede funktionskald returnerer en værdi eller ej. Hvilken metode, der skal benyttes, defineres i variablen *returnsSomething*. *SCOpenWindow* sætter denne variabel til true, mens *SCCloseWindow* sætter den til false. Denne variabel har indflydelse på de fleste metoder i disse klasser:

- *create*(…) genererer en instans af klassen, og gemmer evt. variable i instansen.
- *destroy*() destruerer denne instans af klassen.
- *staticDestroy*(sc) er en statisk metode, der kalder *destroy*() på argumentet, hvis dennes *returnsSomething* variable er false. Ellers gør metoden intet.
- *waitForExecution*() sætter den kaldende tråd i dvale indtil *signal*() kaldes, hvis *returnsSomething* er true. Ellers gør metoden intet.
- *execute*() kalder den tilsvarende metode i *SidmApplication*. Evt. argumenter til denne metode findes som member variable i klassen, tilsvarende gemmes evt. returværdier i klassens member variable. Hvis *returnsSomething* er true, afsluttes med at kalde *signal*().

Sekvensdiagrammet i Figur 3-9 viser hvordan *SCCloseWindow* benyttes. Andre kald, der ikke returnerer en værdi, vil forløbe på tilsvarende måde.

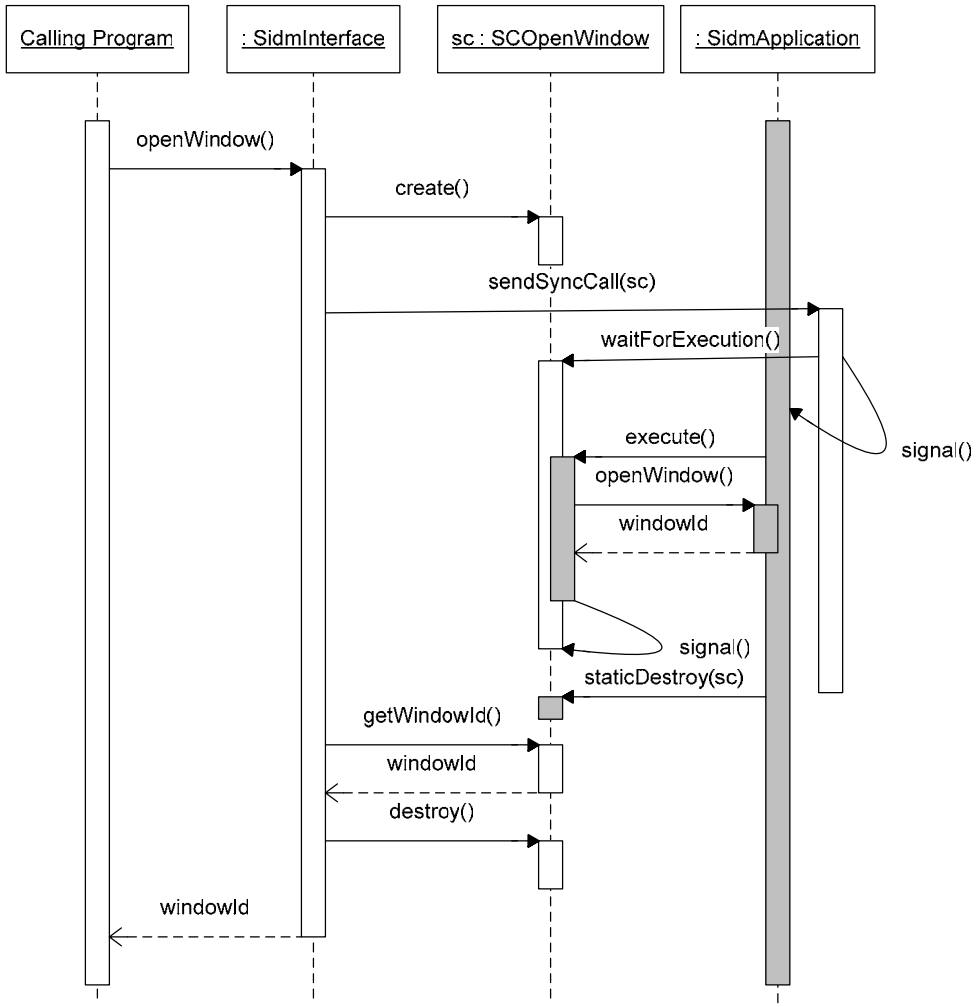


Figur 3-9: Sekvensdiagram der viser et eksempel på kommunikation mellem brugertråden (hvide søjler) og inputtråden (grå søjler). Forløbet illustrerer et funktionskald fra brugerprogrammet, der ikke returnerer en værdi. *CloseWindow* er brugt som eksempel.

Brugertråden laver en instans af *SCCloseWindow* og overfører det aktuelle vindues-Id til denne instans. Herefter kaldes *sendSyncCall* i *SidmApplication*, som placerer objektet på en kø, sender signal til inputtråden og kalder *waitForExecution*() i objektet. Da *returnsSomething* er false afslutter denne metode med det samme og brugertråden forlader SIDM.

Hver gang inputtråden har et ledigt øjeblik, kontrollerer den, om der findes objekter i handlingsobjektkøen. Hvis ikke, går tråden i dvale, mens den venter på brugerinput eller et signal. Når inputtråden finder et objekt i køen, udtrækkes objektet, og *execute*() kaldes. Denne metode kalder *closeWindow* i *SidmApplication*. Til sidst destrueres syncronized call objektet via *staticDestroy*.

Kommunikationen foregår som sagt på en lidt anden måde, når funktionskaldet skal returnerer en værdi. Figur 3-10 viser sekvensdiagrammet for et sådan kald. *SCOpenWindow* er brugt som eksempel.

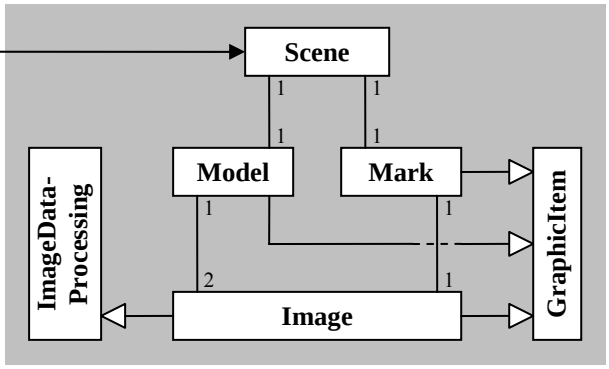


Figur 3-10: Sekvensdiagram, der viser et eksempel på kommunikation mellem brugertråden (hvide søjler) og inputtråden (grå søjler). Forløbet illustrerer et funktionskald fra brugerprogrammet, der returnerer en værdi. *OpenWindow* er brugt som eksempel.

Når et funktionskald skal returnere en værdi, er *returnsSomething* i den tilhørende syncronized call klasse sat til true. Dette resulterer i, at *waitForExecution*() sætter brugertråden i dvale, indtil inputtråden kalder *signal*(). Metoden *staticDestroy*(sc) vil i dette tilfælde ikke have nogen effekt. Når brugertråden vågner fra sin dvale, henter den returinformationen fra *SyncCall*-objektet og destruerer dette objekt.

3.4.4 Vinduesmodul

Hjertet af SIDM ligger i vinduesmodulet, der administrerer grafik i - og input til – et enkelt grafikvindue. Modulet er opbygget af seks klasser, vis indbyrdes relationer kan ses på Figur 3-11.



Figur 3-11: Klassediagram for vinduesmodul (udtræk fra Figur 3-6).

Efterfølgende vil de primære arbejdsopgaver i hver klasse kort beskrives:

Scene: Modulets interfaceklasse, der tilgås fra kontrolmodulet. Kontrolmodulet kalder *Scene*, når det videresender et kald fra brugerprogrammet, og når det modtager direkte brugerinput via mus eller tastatur (se dog afsnit 3.4.8). *Scene* behandler disse kald selv, eller videresender dem til *Model* eller *Mark*. *Scene* står desuden for den overordnede styring af grafikken i vinduet. Dvs. den kalder *draw*() på *Mark* og *Model*.

GraphicItem: Dette er en abstrakt basisklasse for alle elementer, der kan vises i grafikvinduet. Klassens specifikke opgaver vil afhænge meget af implementeringssproget.

Model: Denne klasse administrerer den aktive model i vinduet. Klassen indeholder to instanser af *Image* klassen. Fire af de nævnte indstillingsvariable administreres i denne klasse: 'Model-/komparatortilstand', 'aktive billede venstre/højre', 'låst/frit aktivt billede' og 'et eller to billeder' (se afsnit 2.3.1).

Mark: Målemærkerne styres i denne klasse. Den primære information i denne klasse er målemærkernes (eller det vandrende mærkes) koordinater, som der findes tre versioner af: Et sæt objektkoordinater, to sæt billedkoordinater og to sæt skærmkoordinater. Sidstnævnte er implementeringsspecifikke koordinater, der uafhængigt af billedet definerer, hvor i vinduet de to målemærker skal tegnes. Indstillingsvariablene 'låst/frit mærke' og 'låst/fri markør' administreres i denne klasse (se afsnit 2.3.1).

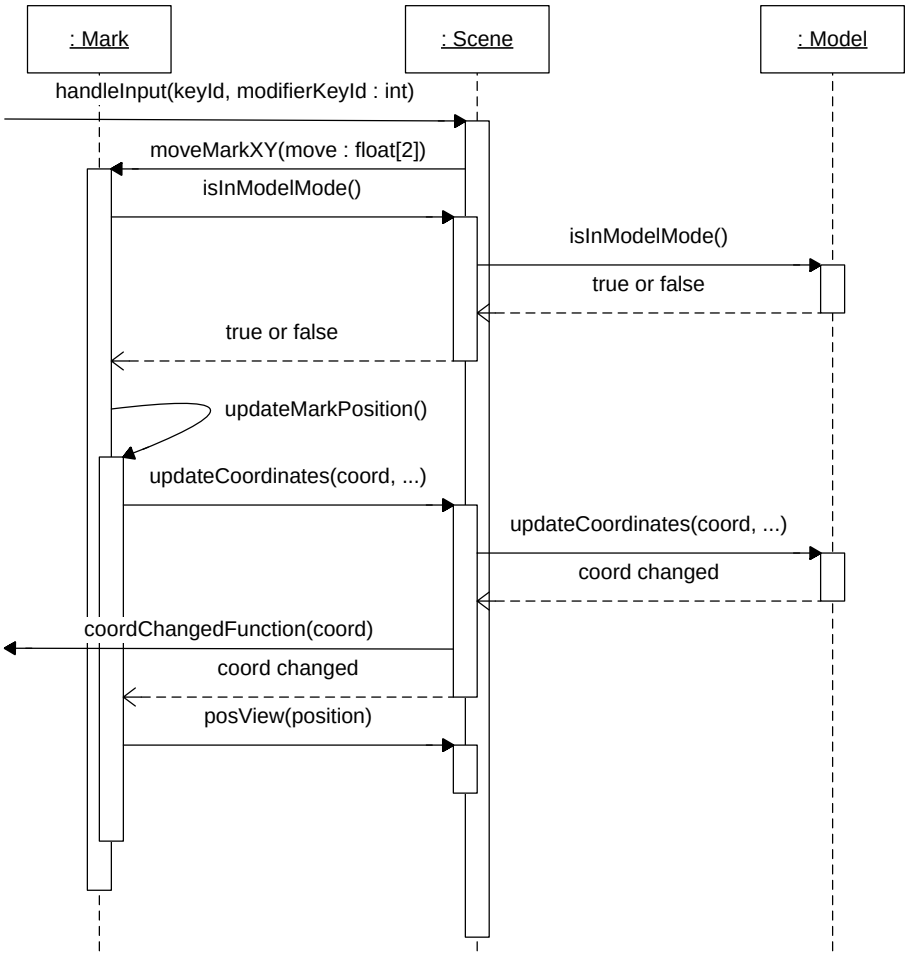
Image: Indeholder pixeldata og rutiner til at tegne vinduet på skærmen.

ImageDataProcessing: Indeholder billedets orienteringsdata, samt funktioner til at regne med disse. Fx en funktion, der kan konvertere objektkoordinater til billedkoordinater.

Det vil være en for stor opgave i denne rapport at gennemgå disse klasser og deres specifikke arbejdsopgaver i detaljer. I stedet vil der blive gennemgået et enkelt eksempel på funktionaliteten i disse klasser:

I dette eksempel antages det, at brugeren lige har trykket på en tast, der er bundet til kommandoen (MOVE_MOUSE_XY, LEFT), hvilket betyder, at måle-

mærkerne skal flyttes et lille stykke til venstre. Kontrolmodulet har registreret dette tastetryk, har registreret id på den trykkede tast og kalder *handleInput*(…) i *Scene*. Se Figur 3-12 for et sekvensdiagram over de resulterende handlinger:



Figur 3-12: Sekvensdiagram over et typisk kald i vinduesmodulet. Det viste eksempel beskriver forløbet for et tastaturinput, der er bundet til at flytte målemærkerne.

Når *handleInput* kaldes i *Scene*, kontrollerer metoden om den angivne tastekombination er bundet til en kommando (se afsnit 3.4.7 for en beskrivelse af inputbindinger). Hvis dette er tilfældet sender metoden kaldet videre til den korrekte funktion. I inputkonfigurationen findes information om hvor langt et skridt, målemærkerne skal flyttes per tastetryk. Denne information benyttes til at danne en bevægelsesvektor, der videresendes til *MoveMarkXY* i Mark-klassen.

MoveMarkXY kontrollerer om modellen er i modeltilstand. Hvis den er, adderes den medsendte vektor til det vandrende mærkes objektkoordinater, ellers adderes vektoren til målemærkernes billedkoordinater.

UpdateCoordinates beregner billedkoordinater ud fra objektkoordinater, og skærmkoordinater ud fra billedkoordinater. Begge disse konverteringer udføres af det respektive *Image*-objekt (billed- til objektkoordinater beregnes af basis-

klassen *ImageDataProcessing*).

Når *Scene* modtager de opdaterede koordinater på vej tilbage til *Mark*, kaldes en funktion i brugerprogrammet, for at informere denne om, at koordinaterne er ændret.

Mark afslutter med at kalde *PosView* (eller *ScrollToViewPos*, hvis mærket ikke er låst) for at flytte fokus af skærmen, så målemærkerne stadig holdes i centrum.

3.4.5 Vinduesmodul – udvidelser

Dette afsnit beskriver i overfladiske træk hvordan de foreslåede udvidelser skal indføres i designet.

Alternative stereoskopiske betragtningssystemer: Der er intet i designet, der begrænser hvilke former for stereoskopiske betragtningssystemer, der kan bruges. Det er ideen med dette design, at *GraphicItem* administrerer hvilket stereoskopisk betragtningssystem, der benyttes, og hvorvidt objektet tegnes til venstre eller højre øje. I nogle implementeringssystemer kan dette være mere passende at implementere i *Scene*-klassen.

Normalisering af billeder: Igen et punkt som er direkte afhængigt af implementeringssystemet. Matricen til normalisering vil beregnes i *ImageDataProcessing*, og denne matrice vil benyttes i *Image*-klassen.

3D-streger i modellen: Use cases UC44 til UC52 beskriver i detaljer, hvilke funktionaliteter der skal tilføjes, for at implementere denne udvidelse. Selve stregerne administreres af en ny klasse, der ligger som en instans i *Model*-klassen. Informationer om stregerne kan gemmes på flere måder, hvoraf den umiddelbare mest simple er blot at gemme brugerens kommandoer. Dvs. at placere objekter, der minder om handlingsobjekterne på en kø i den nye klasse. Hver gang stregerne skal tegnes op, bliver denne kø gennemgået fra ende til anden. *updateCoordinates*-funktionen i *Model*-klassen benyttes til at konvertere objektkoordinaterne til skærmkoordinater.

Automatisk skift mellem modeller: For at denne funktionalitet kan implementeres, må yderligere informationer om billederne kendes. Til *Images*-tabellen i databasen tilføjes fire værdier: *setId*, *groupId*, *width* og *height*. De to sidstnævnte beskriver bredde og højde af den aktuelle del af billedet. Dvs. fraregnet evt. kanter forårsaget af kameraet eller dets ophæng. *setId* og *groupId* beskriver billedernes indbyrdes sammenhæng: Billeder, der viser det samme generelle område, har samme *groupId*. Hvis to billeder også er taget fra omtrent den samme afstand og retning, har de også samme *setId*. Fx hvis et fly tager otte billeder af et landskab fra samme højde, vil disse billeder alle have samme *setId* og *groupId*. Hvis flyet stiger højere op for at tage nogle mindre detaljerede oversigtsbilleder, har disse billeder samme *groupId*, som de første billeder, men ikke samme *setId*.

Skifter brugeren fra ét vindue til et andet vindue, der har en åben model med det samme *groupId*, vil kontrolmodulet først kalde *getCoordinates* i det forladte vindue. Disse koordinater vil benyttes i et efterfølgende kald af *gotoObjectCoordinates* i det nye vindue, for at flytte det vandrende mærke til den samme

position.

Når en ny model læses ind, indlæses samtidig alle andre modeller, der har det samme *setId* og *groupId*. For at begrænse hukommelsesforbruget bør disse ekstra *Model*-instanser ikke som standard indlæse billedernes pixel-data. Når brugeren bevæger det vandrende mærke uden for den primære model, undersøges alle de sekundære modeller, for hvorvidt det vandrende mærke befinder sig inden for deres afgrænsning. Hvis dette er tilfældet, vil SIDM skifte til den angivne model som primærmodel. *isPointInImage*-funktionen i *ImageDataProcessing* kan fortælle hvorvidt et punkt er inden for et billede eller ej. *width* og *height* benyttes til at bestemme hvornår et punkt er inde i et billede eller ej.

Den sidste mulighed er, at brugeren skal kunne åbne en menu, der viser en liste over modeller med samme *groupId* som den eksisterende, og som inkludere det vandrende mærkes objektkoordinat. Dette bør implementeres i den nye klasse *ModelMenu*, der kan kaldes fra *Scene*. *ModelMenu* kan benytte eksisterende funktioner i *Database* og *Model* til at undersøge, hvilke andre modeller der dækker det samme område. Et *Model*-objekt bør indlæses for hver af disse modeller, for at beregne om den angivne model indeholder det vandrende mærkes position eller ej.

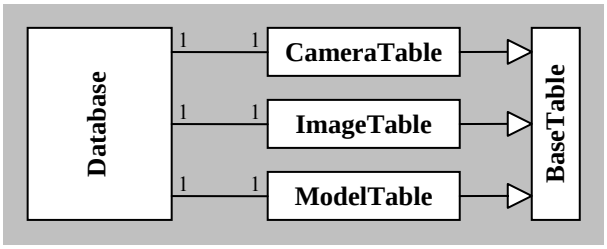
3.4.6 Databasemodul

Databasen indeholder tre tabeller, der alle kan blive tilgået på samme måde. Det skal være muligt at udtrække data, indsætte ny data og lave en søgning på data. SIDM har en klasse til hver tabel i databasen. Disse tre klasser skal udføre mange af de samme operationer og vil derfor nedarve fra den samme basisklasse (se Figur 3-13). De strukturer, der benyttes til at gruppere data for en enkelt *tuple* (element i en tabel), nedarver ligeledes fra en fælles basisstruktur, og det samme gælder de strukturer, der gemmer søgedata. Ved at benytte disse basisstrukturer kan langt de fleste operationer udføres i *BaseTable*-klassen. De specialiserede klasser indeholder i praksis kun tre metoder:

LoadTupleFromDb: Indlæser en enkelt tuple fra databasen.

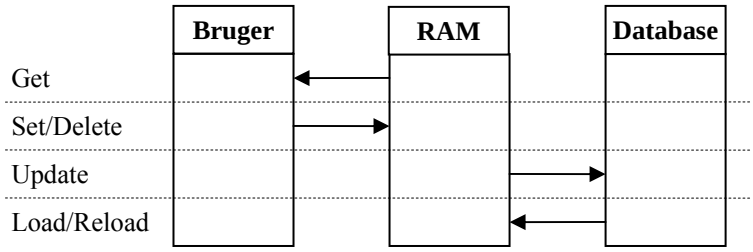
SaveTupleToDb: Gemmer en enkelt tuple i databasen.

DoMatchSearch: Bestemmer om en tuple matcher de angivne søgekriterier.



Figur 3-13: Klassediagram for databasemodul (udtræk fra Figur 3-6).

Det blev i afsnit 3.3.5 bestemt, at databasen skal indlæses i computerens hukommelse, så udtræk og søgninger vil forløbe hurtigere. Dette giver fire typer af dataflow mellem brugerklassen, hukommelse (RAM) og databasen. Figur 3-14 beskriver navngivninger for funktioner, der udfører disse overførsler.



Figur 3-14: Dataflow i en databasetabelklasse.

Foruden data fra databasen vil en datatuple indeholder en enumeration variabel af typen *TupleStatus*. Denne beskriver hvordan den angivne tuple i hukommelsen passer med den tilsvarende tuple i databasen. Variablen kan have værdierne UNCHANGED, MODIFIED, NEW og DELETED. Variablen sættes til UNCHANGED, når en tuple indlæses fra database til hukommelse. En *set*- eller *delete*-operation på den angivne tuple vil ændre *tupleStatus* til en af de andre værdier. Når *update* kaldes, kontrolleres *tupleStatus* for at se hvilken operation, der skal tages: Gør intet (UNCHANGED), opdater tuple (MODIFIED), tilføj tuple (NEW) eller slet tuple (DELETED).

3.4.7 Konfigurationsmodul

Alle konfigurationsinformationer til SIDM administreres af klassen *Config* i konfigurationsmodulet. Informationerne opdeles i fire strukturer. Hver af disse strukturer indeholder konfiguration til en enkelt klasse. Efterfølgende er en oversigt over de primære informationer i hver af disse strukturer. Det præcise indhold af hver struktur kan ses i Figur A - 17 i appendiks E.2.

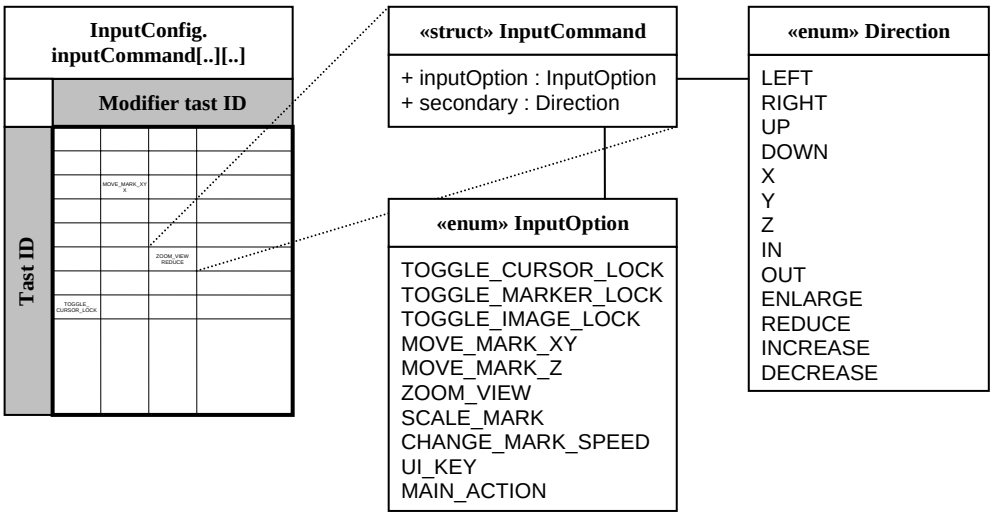
DatabaseConfig: Forbindelsesinformationer til databasen. Bruges af *Database*.

StereoConfig: Aktive stereovisningsmetode og anaglyphfarver. Bruges af *GraphicItem*.

MarkConfig: Målemærkernes udseende. Bruges af *Mark*.

InputConfig: Hvilke taster er bundet til hvilke kommandoer. Bruges af *Scene*.

Efterfølgende vil der kort beskrives, hvordan *InputConfig* gemmer information om bundne taster: *InputConfig* indeholder et todimensionalt array kaldet *inputCommands*. Dette array gemmer information om, hvilke taster der er bundet til hvilke input kommandoer (se Figur 3-15). Den ene dimension i arrayet strækker over de forskellige primære taster og knapper, mens den anden dimension strækker over nedholdte *metataster* (Shift, Ctrl og Alt). Bestemmelse af præcise ID'er på taster og knapper overlades til implementeringssproget.



Figur 3-15: Konfiguration af taster og inputkommandoer.

Hver plads i dette array er enten tom eller indeholder en *InputCommand* struktur. Hvis elementet er tomt, er der ingen kommando bundet til denne tastkombination, dvs. der skal ikke udføres nogen handling af programmet. Hvis pladsen holder en *InputCommand* struktur, definerer denne hvilken handling, der skal tages:

Som det ses på Figur 3-15, indeholder *InputCommand* to enumeration variable: *inputOption* og *secondary*.

inputOption: Denne værdi bestemmer hvilken primære kommando, der er tale om. Dette kunne fx være SCALE_MARK, der ændrer målemærkernes størrelse.

secondary: Flere af de primære kommandoer mangler yderligere information for at definere præcis hvad programmet skal gøre. Fx skal der til kommandoen SCALE_MARK også informeres om, hvorvidt målemærkerne skal forstørres (ENLARGE) eller formindskes (REDUCE). Denne information leveres i sekundærvariablen.

Ved opstart af SIDM indlæses konfigurationsinformationerne fra en fil. Hvordan denne proces skal implementeres vil være afhængigt af implementeringssproget. Nogle sprog vil indeholde værktøjer til dette, mens det skal implementeres af programmøren i andre sprog.

3.4.8 Andre moduler

Dette afsnit vil kort gennemgå de resterende tre moduler i SIDM.

Interfacemodulet: Dette moduls eneste opgave er at videresende kald fra brugerprogrammet til de respektive moduler. Indholdet af *SidmInterface*, som er den eneste klasse i dette modul, kan ses i Figur A - 18 i appendiks E.2.

Kontrolmodul: Det programbibliotek, der benyttes til administration af vinduer og modtagelse af brugerinput, vil have stor indflydelse på indholdet og udseendet af kontrolmodulet. Det er derfor ikke muligt at tage alt for mange designbe-

slutninger mht. dette modul, før implementeringssprog og programbiblioteker er valgt. Det antages, at dette modul skal administrere brugerinput, dvs. bestemme id på de taster, der er trykket, og sende denne information til det aktive vinduesmodul. Afhængigt af det benyttede programbibliotek, er det dog muligt, at en bedre løsning ville være, at lade vinduesmodulet modtage sine egne brugerinput.

Fejlbehandlingsmodul: Dette modul skal administrere kommunikation af fejl med brugerprogrammet. Modulet vil fungere som beskrevet i afsnit 3.3.3.

3.5 DESIGN PATTERNS

Når man som programmør skal designe kode til at løse et givent problem, kan man sidde længe og overveje præcis, hvordan koden skal opbygges. Det er sjældent det store problem at finde en brugbar løsning, men ofte holdes man tilbage med en fornemmelse af, at der må findes en bedre metode til løsning af problemet. Det samme problem har formentlig været overvejet af hundrede vis af programmører før dig, og de er hver især kommet op med en brugbar løsning.

Design patterns er standardiserede løsninger til almindelige problemer i software design. Målet med disse er at simplificere udviklingsprocessen ved at give nær færdige løsninger, der har været brugt tidligere og fundet effektive. Et design pattern er ikke færdiggenereret kode, der direkte kan kopieres ind i dit program. Det er i stedet en efterprøvet løsning til hvilke klasser, der skal arbejde sammen på hvilken måde for at løse et givent problem.

Den første bog om design patterns ”*Design Patterns - Elements of Reusable Software,* af Gamma, Helm, Johnson, and Vlissides” udkom i 1995. Bogen refereres ofte til som ’*the Gang og Four*’ (GoF), hvor navnet henviser til de fire forfattere, og bogen anses stadig som den fundamentale bog om emnet. Bogen gennemgår 23 design patterns, der beskriver løsninger på generelle problemer, der ofte stødes på under programdesign.

Designet af SIDM er baseret på fire forskellige design patterns fra GoF: *Singleton*, *Template Method*, *Façade* og *Command*:

Singleton: *Singleton* pattern sikrer, at én og kun én instans af en klasse eksisterer og giver samtidig et globalt tilgangspunkt til denne klasse. Dette pattern benyttes af *SidmApplication*, *ImageLoader*, *Database*, *Config* og *ErrorHandler*.

Template Method: Dette design pattern formaliserer ideen om at definere en algoritme i en abstrakt klasse, mens detaljer efterlades til at blive implementeret i en underklasse. *Template Method* benyttes af *BaseTable* sammen med underklasserne *ImageTable*, *CameraTable* og *ModelTable*.

Façade: Et *Façade* pattern skjuler et underliggende delsystem og tjener som router til de forskellige klasser og metoder i dette system. Dette giver den ydre verden et enkelt og simpelt interface i stedet for det avancerede delsystem. *SidmInterface* implementerer et simpelt *Façade* pattern, ved at virke som et mellemled mellem brugerprogrammet på den ene side og *SidmApplication*, *Database* og *ErrorHandler* på den anden side.

Command: *Command* pattern placerer en forespørgsel om en given handling inde i et objekt, og giver dette objekt et kendt interface. Dette tillader, at hand-

lingen siden kan ændres uden at påvirke klienten, og at forespørgsler kan placeres på en kø eller logges. I SIDM benyttes ideen fra *Command* pattern til kommunikation mellem *SidmInterface* og *SidmApplication*, dvs. implementering af handlingsobjekterne.

3.6 OPSUMMERING

Dette kapitel beskriver det implementeringsuafhængige design af SIDM, dvs. der ikke er taget højde for implementeringssprog eller benyttede programpakker.

Udviklingen af SIDM baseres på såkaldte use cases, der er bygget op over kravspecifikationen. Disse use cases benyttes som basis for det videre design. SIDM er undergået en iterativ udvikling, hvor programmet er bygget op over adskillige faser, der hver især er delvist afsluttede.

Strukturen af SIDM er opdelt i moduler, som er en gruppering af en eller flere klasser, der arbejder sammen for at udføre et fælles mål. Hvert modul har et begrænset interface til de andre moduler, så kommunikationen mellem moduler er veldefineret.

4 Implementeringsspecifik analyse

De forrige kapitler har gennemgået problemet, analysen og designet på et overordnet niveau, der er holdt åbent for de fleste objektorienterede implementeringssystemer. I dette kapitel øges detaljegraden et niveau, når programmeringssproget og nyttige programpakker analyseres og udvælges.

4.1 PROGRAMMERINGSSPROG

Der findes et stort antal programmeringssprog på markedet i dag. De fleste af disse er specialiserede i at løse en begrænset niche af problemer eller til kun at virke på specifikke platforme. Til dette projekt begrænses mængden af programmeringssprog til dem, der har potentiale til effektivt at kunne løse det benævnte problem. Når mængden yderligere reduceres til de sprog, som udvikleren har et rimeligt kendskab til, efterlader det følgende sprog: C, C++, Java og C# .NET. Af disse fire sprog kan C med det samme elimineres, da det som funktionsbaseret sprog ikke kan opfylde det objektorienterede design, der er tegnet til programmet.

Ved valg af et programmeringssprog til implementering af SIDM skal følgende punkter overvejes:

- Ydeevne af det færdige program.
- Udviklingstid baseret på kodestruktur og udvalget af biblioteker i sproget.
- Udvikling, kompilering og eksekvering af programmet fra forskellige operativsystemer.

4.1.1 C++

C++ er bygget direkte oven på C og tilbyder et objektorienteret sprog med mulighed for en rigtig god ydeevne af det færdige program. C/C++ har længe været de primære sprog til produktion af programdele, der kræver høj ydeevne som fx computerspil.

En stor ulempe ved sproget er det begrænsede standardbibliotek. Der findes ingen direkte metoder til databaseadgang, grafikvisning, synkronisering og andre platformspecifikke operationer. Programmøren er derfor nødt til selv at skrive den platformspecifikke kode til disse operationer eller benytte eksisterende programpakker udviklet af andre programmører. Hvis programmet kun skal køre på en specifik platform, er dette ikke det store problem. Omfattende platformspecifikke programpakker, der indeholder de fleste af de nødvendige funktionaliteter til dette projekt, findes til de fleste platforme. Fx kan nævnes MFC til Microsoft Windows.

Problemet opstår, når programmet skal være platformuafhængigt. Her er programmøren ofte nødt til at finde de ønskede funktionaliteter i små, uafhængige programpakker. Hvis platformuafhængige programpakker benyttes, kan programmet udvikles og kompileres på de fleste platforme. Programkoden kan kun eksekveres på det operativsystem, det er kompileret til. Til gengæld kan det

kompilerede program køres på alle maskiner, der benytter det givne operativsystem, uden først at installere yderligere programmer.

4.1.2 Java

Java regnes for et højere niveau programmeringssprog end C++. Grundet bl.a. den indbyggede *garbage collector* og et stort standardbibliotek er det simpelt og effektivt at skrive programkode i dette sprog. Mange funktionaliteter, der ikke findes i C++ standarden, er implementeret som standard i Java. Prisen, der betales for den hurtige udviklingshastighed, er en generel lavere ydeevne end tilsvarende C++ programmer. Med ordet 'generel' menes, at du godt kan lave et program, der vil eksekvere hurtigere på Java end på C++, men i de fleste ydelseskrævende operationer - som fx grafikbehandling - vil en C++ løsning have bedst ydeevne.

Java kompileres ikke direkte til en eksekverbar fil, som er tilfældet i C++. I stedet genererer Java-kompileren script filer. For at eksekvere disse filer kræves et softwarelag kaldet *Java Virtual Machine* (JVM). Fordelen ved denne metode er, at den samme kompilering kan eksekveres på alle platforme, hvilket fx er vigtigt mht. web-applikationer. Ulempen er, at JVM skal være installeret på en maskine, før et Java program kan køres, samt at dette mellemled er den største årsag til Javas nedsatte ydeevne på flere områder i forhold til C++ [R20].

4.1.3 C#

C# .NET (udtales C-sharp dot net) minder på mange områder om Java, og de fleste af bemærkningerne omkring Java gælder også for C#. Ligesom Java kører C# ovenpå et software lag (et *managed execution environment*). I C#'s tilfælde er dette lag *.NET Framework Common Language Runtime* (CLR). Selvom dette miljø er under udvikling til flere forskellige operativsystemer, er det endnu kun hundrede procent implementeret til de nyere Windows systemer. Det menes derfor at C# endnu ikke egner sig som et multiplatforms programmeringssprog.

4.1.4 Valg af programmeringssprog

Det programmodul, der udvikles i forbindelse med dette projekt, skal udføre krævende operationer med store mængder grafisk data. Udvikleren føler derfor, at ydeevne har højere prioritet end programmeringshastighed. Det er af denne grund blevet besluttet at benytte C++ som implementeringssprog.

4.2 PROGRAMPAKKER

Som tidligere beskrevet indeholder standardbiblioteket i C++ ingen platformspecifikke funktionaliteter. De fleste sådanne funktionaliteter er dog før blevet implementeret af andre programmører. Der findes i dag en stor mængde programpakker - ofte kaldet API'er - der hver især løser et eller flere specifikke problemer.

API er en forkortelse for *Application Programming Interface*. En API er "et sæt af definitioner om hvordan ét stykke computersoftware kommunikerer med et

andet" [R20]. I denne rapport bruges betegnelsen API dog om enhver implementering af software, der kan benyttes af andet software via klasser og funktionskald. Dette inkluderer implementeringer af en API, SDK (*Standard Developer Kit*), *toolkits* og lignende software.

Første skridt til valg af API'er er at definere hvilke funktionaliteter, der er brug for. SIDM mangler følgende funktionaliteter, der ikke dækkes af C++ standardbibliotek:

- **F1** Åbne og lukke vinduer.
- **F2** Vise grafik i vinduer.
- **F3** Modtage brugerinput til vinduer.
- **F4** Opstart, nedlukning og synkronisering mellem tråde
- **F5** Indlæsning af billeder fra diverse billedformater.
- **F6** Kommunikation med en database.
- **F7** Matrixregning. Denne komponent er rimelig simpel selv at implementere, så den er ikke kritisk nødvendig.

De indledende krav til en API, der skal benyttes i dette projekt, er, at den kan køre på både Windows XP/2000/NT, Linux og Unix platforme, samt at den er gratis. Dvs. at den lovligt kan hentes fra Internettet og benyttes i programmet uden nogen monetær omkostning.

De efterfølgende afsnit i dette kapitel indeholder analyse og valg af API'er, der leverer de ovenstående funktionaliteter. Afsnittene er skrevet kronologisk, hvilket vil sige, at valg taget i det første afsnit kan have indflydelse på valgene i de efterfølgende afsnit.

4.3 GRAFIKBEHANDLING

Ser man på de grafiske krav til SIDM, skal det færdige programmodul i et vindue kunne vise to billeder, to målemærker samt koordinater o.a. tekstuel information. Målemærkerne kan enten tegnes oven på billedet i form af streger o.l., eller det kan eksistere i form af et lille billede (ikon), der placeres oven på de andre billeder. I sidstnævnte tilfælde skal den grafiske API kunne tage højde for ikonets alfa farvekomponenter, når målemærkerne placeres oven på billederne. Er dette ikke muligt, vil målemærkerne kun kunne have en rektangulærform, hvilket ikke er tilfredsstillende. Når billederne vises som anaglyph, skal SIDM kunne sammenblande de to billeder, så begge kan ses korrekt på skærmen. Det er ikke et krav, at alle disse funktionaliteter skal understøttes af den grafiske API. Funktionaliteter, der ikke tilbydes af denne, skal dog understøttes på anden måde i SIDM, evt. i en anden API.

Den første beslutning, der skal tages i valget af en API til grafikbehandling, er, hvorvidt der skal arbejdes med 2D eller 3D grafik.

Vælges en 3D løsning er valget af API simpel. I disse år benyttes primært kun to 3D grafik-API'er - *OpenGL* og *Direct3D*. Hvor OpenGL understøttes af de fleste platforme, er Direct3D en del af Microsofts DirectX og understøttes der-

for kun af Windows systemer. Ønsker man et program, der kan køre på flere platforme, er OpenGL derfor det rigtige valg.

SIDM skal i praksis kun vise todimensionale billeder og grafik. Kan et problem løses på to måder er det ofte praktisk at benytte den simple løsning. Umiddelbart kan en 2D grafik-API derfor virke som det bedste valg. Udvalget af ikke-kommercielle 2D grafik-API'er, der kan køre på de krævede platforme, er dog ganske begrænset. Sammenlignes de fundne 2D grafik-API'er med en OpenGL løsning, har sidstnævnte flere fordele:

- OpenGL kan udføre farveblandinger (*blending*) direkte på grafikkortet. Dette tillader en simpel og effektiv implementering af anaglyph stereovisningsmetoden. Ingen af de fundne 2D grafik-API'er indeholder funktionaliteter til at sammenblande billeder. En 2D-løsning ville derfor kræve en ekstern funktionalitet til at sammenblande to billeder til et enkelt billede i computerens processor. Dette vil øge programmets kompleksitet og reducere dets ydeevne.
- De fundne 2D grafik-API'er er alle del af en GUI (*Graphical User Interface*): API'erne administrerer selv det vindue, grafikken vises i. Ingen af disse API'er leverer mulighed for Dual Display eller Page Flipping. OpenGL er afhængig af andre API'er til at administrere grafikvinduet, og forhindrer derfor ikke umiddelbart muligheden for Dual Display og Page Flipping.
- OpenGL har indbyggede matrixoperationer og vil kunne udføre billednormalisering i realtime direkte på grafikkortet.

Da mange operationer i OpenGL udføres på grafikkortet, og da de fleste moderne computere i dag indeholder kraftige 3D grafikacceleratorer, forventes en OpenGL løsning ikke at have ringere ydeevne end en tilsvarende 2D løsning. Valget af grafik-API falder derfor på OpenGL, hvormed funktionaliteterne F2 og F7 er understøttet (se afsnit 4.2).

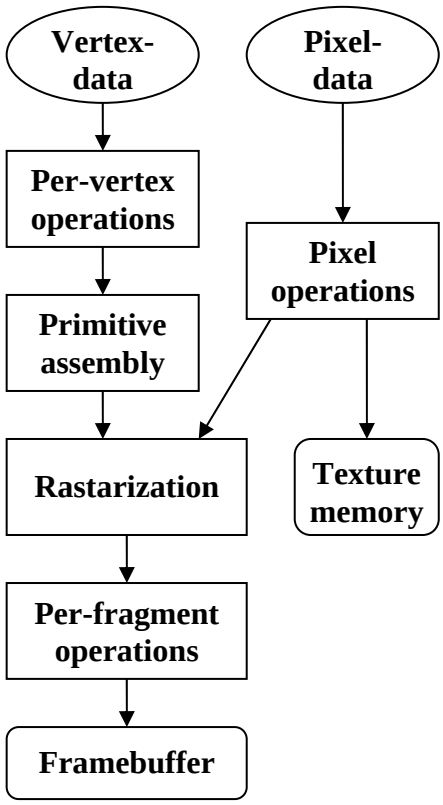
4.3.1 OpenGLs Rendering Pipeline

For at få en forståelse af OpenGL's virkemåde, er det nødvendigt at se på, hvordan OpenGL behandler grafisk data. De fleste implementeringer af OpenGL gennemgår den samme række af operationer for at konvertere den indtastede data til grafik på skærmen. Denne serie af operationer kaldes *OpenGLs rendering pipeline*. En simplificeret oversigt over en mulig rækkefølge af operationer kan ses på Figur 4-1. Operationer, der enten ikke vil blive benyttet af SIDM eller som ikke føles nødvendige for forståelsen af OpenGL's virkemåde, er ikke inkluderet i figuren.

Nedenstående vil hvert operationstrin i den viste rendering pipeline kort blive gennemgået. Denne gennemgang fokuserer på operationer, der vil blive benyttet af SIDM, og er derfor på ingen måde komplet. Informationer til dette afsnit er taget fra [R1], [R12] og [R18].

Vertex-data: Ordet *vertex* bruges i 3D grafik til at definere et punkt, der eksisterer i et tredimensionalt rum. Alle geometriske objekter kan opbygges af en el-

ler flere vertices. Fx kan en ret linie defineres ud fra to vertices. Vertex-data eller geometrisk data, inkluderer alt form for tredimensionalt data. Dvs. punkter, linier og polygoner, der defineres ud fra vertices. Hvis en flade skal have en tekstur, defineres teksturkoordinaterne (se afsnit 5.2.1) på dette stadie, mens selve teksturbilledet sendes som pixeldata.



Figur 4-1: Oversigt over en simplificeret udgave af OpenGL's rendering pipeline.

Per-vertex operations: På dette trin flyttes, formes og roteres den geometriske data, så objekter ligger korrekt placeret i forhold til hinanden og til verdenen. Denne transformering udføres som lineære operationer via matrixmultiplikationer.

Primitive assembly: Den primære opgave i *primitive assembly* er at tilklippe 3D-verdenen, så kun de ønskede objekter eller dele af objekter bliver vist. I SIDM vil dette defineres i form af en retning og position hvorfra verdenen skal ses samt størrelsen af det område, der skal vises. Resultatet af dette trin er geometriske primitiver, dvs. vertices med tilhørende position, farve, evt. teksturkoordinater og lignende værdier, der skal benyttes i *rasterization*.

Pixeldata: Pixeldata inkluderer billeder og andet todimensionalt grafik. Dvs. prikker, streger, tegn og andet, der defineres ud fra 2D raster-koordinater. Billeder, der benyttes som teksturer på 3D-flader, indsendes også til OpenGL som pixeldata.

Pixel operations: Pixeldata bliver først udpakket og konverteret til det rigtige format. Det formaterede data sendes enten til rasterization eller til grafikkortets texture memory.

Texture memory: Moderne grafikkort besidder meget RAM, der primært benyttes til at lagre teksturer. Typisk indeholder et nyt grafikkort i dag 128Mb onboard RAM, hvilket i teorien kan lagre samlet over 40 Mpixel billeder. Så længe der er hukommelse til det, vil OpenGL uploade teksturer til grafikkortet. Når grafikkortets hukommelse er fyldt op, vil resterende teksturer placeres i processorhukommelsen. Funktioner i OpenGL giver programmøren mulighed for at have indflydelse på hvilke teksturer, der placeres i grafikkortets hukommelse, og hvilke, der placeres i processorhukommelsen.

Rasterization: På dette trin konverteres både geometrisk data og pixeldata til fragmenter. Hvert fragment svarer til en enkelt pixel i framebufferen og inkluderer bl.a. information om farve og dybde. Når fx en linie bliver rastarized til fragmenter, undersøges hvilke pixels i vinduet, linien gennemløber. Her tages bl.a. højde for linietykkelse og hvis linien er stiplet.

Per-fragment operations: Før værdier gemmes i framebufferen, gennemgås en serie af operationer, der kan ændre eller bortkaste fragmenter. Den første operation er *texturing*, hvor teksturelementer genereres ud fra texture memory og anbringes på hvert fragment. Efterfølgende udføres bl.a. *blending* inden den gældende pixel placeres i framebufferen. Blending beskriver den proces, hvor farven på den eksisterende pixel i framebufferen sammenlignes med det indkomne fragment for at bestemme den resulterende farve på den angivne pixel. OpenGL tilbyder flere former for *blending functions*, der definerer hvordan de to farver skal sammensættes.

Framebuffer: *Framebufferen* er en del af grafikkortets hukommelse, der indeholder informationer om én frame på skærmen. I de fleste situationer tilbyder OpenGL to framebuffers: En front buffer, som er den grafik, der kan ses på skærmen, og en back buffer, som ikke vises. Normalt vil grafikken blive opbygget på back bufferen, som efterfølgende udskiftes med front bufferen. På denne måde undgår man synlig optegning af objekter. På nogle grafikkortet kan OpenGL i stedet tilbyde fire framebuffers (kaldet stereo buffers), hvor både front og back buffers opdeles i en *left* og en *right* buffer. Disse buffers bruges i forbindelse med stereobilleder, som administreres af grafikkortet. Fx ved brug af shutter glasses.

4.4 STYRING AF VINDUER OG INPUT

OpenGL er afhængig af, at en ekstern API starter den kontekst (*context*), hvori OpenGL skal køre. Hvis OpenGL skal vise grafik, kræves en rendering context, hvilken ofte er forbundet til et vindue. SIDM kræver funktionaliteter til at åbne og administrere et sådan vindue. Følgende funktionaliteter er ønsket:

- Åbne et eller flere vinduer med tilhørende OpenGL rendering context, skifte mellem disse vinduer og lukke dem individuelt igen. Forudsat grafikkortet understøtter dette, skal denne rendering context tilbyde stereo buffers til stereovisning via shutter glasses.
- Vælge hvilken grafikenhed, der skal modtage grafikken. Dette skal benyttes til dual display stereovisning.
- Udskrive tekstuel information i hvert vindue, enten i en statuslinie eller

lignende felt, eller direkte som grafik.

- Registrere når en tast på tastaturet eller en museknap trykkes eller slippes i et givent vindue, samt når musens hjul bevæges.
- Registrere hver gang musen bliver bevæget indenfor et åbent vindue. Programmet skal desuden kunne skjule musemarkøren, samt sikre at den skjulte markør forbliver inde i vinduet.
- Tillade åbning af en menu, hvori et givent element kan vælges.

Hvis man begrænser mængden af API'er til dem, der opfylder de indledende krav til dette projekt, benyttes der generelt kun to forskellige API'er til vinduesstyring og administration af OpenGL context: *SDL* og *GLUT*. Der findes dog flere varianter af sidstnævnte.

4.4.1 Simple DirectMedia Layer

Simple DirectMedia Layer (SDL) [R27] er en multimedia API, der understøtter grafik, lyd, brugerinput og flere tråde. Brugerinput gemmes som hændelser (events) i en kø, og det er op til brugeren af API'en at læse og tolke disse events. SDL tilbyder som standard ingen mulighed for tekstuelle felter udenfor grafikområdet og ingen mulighed for menuer. API'en indeholder dog fonte, der kan benyttes til at skrive tekstuel information inden for grafikrammen. SDL kan registrere musebevægelser og skjule musemarkøren. En skjult markør kan låses, så den ikke forlader grafikvinduet.

SDL opfylder de fleste krav, SIDM stiller, men fejler dog på et kritisk område. På nuværende tidspunkt understøtter SDL kun et enkelt grafikvindue. Flere vinduer er et af de væsentlige krav til SIDM, og SDL må derfor opgives.

4.4.2 OpenGL Utility Toolkit

OpenGL Utility Toolkit (GLUT) [R22] er på mange områder mere begrænset end SDL, da den hverken understøtter lyd eller tråde. Behandlingen af brugerinput er også mere begrænset i GLUT. I stedet for at placere hændelser i en kø, hvor programmøren selv kan administrere dem, benytter GLUT callback funktioner. Dvs. når en specifik hændelse indtræffer, kalder GLUT en tilsvarende funktion, der er prædefineret af brugeren. Denne metode giver mindre programmørmæssig frihed end metoden, der benyttes i SDL. GLUT tilbyder dog brugen af flere grafikvinduer og er derfor indtil videre at foretrække frem for SDL.

Ligesom SDL tilbyder GLUT som standard ingen mulighed for tekstuelle felter udenfor grafikområdet, men inkluderer fonte, der kan benyttes til at skrive teksten inden for grafikområdet. GLUT kan også registrere alle musebevægelser, og skjule markøren når det ønskes. En skjult markør i GLUT kan dog ikke låses, og den kan derfor bevæges uden for grafikvinduet. Et funktionskald kan ændre musens position i et vindue, og vha. denne funktion kan programmøren forsøge at sikre, at markøren forbliver indenfor vinduet. I modsætning til SDL understøtter GLUT simple menuer.

GLUT sender ingen informationer, når en tastaturtast slippes. I stedet sendes en "tast trykket" hændelse gentagende gange, hvis en tast holdes nede i længere tid. Musetryk behandles anderledes. Her sendes information når knappen trykkes, og igen når den slippes. Ifølge den nuværende kravspecifikation skal SIDM generelt kun informeres, når en tast eller knap bliver trykket ned. Information om, at knappen slippes igen, er kun nødvendig i forbindelse med knappen bundet som primære input til brugerprogrammet (se 2.3.5). Da denne funktion sandsynligvis altid vil være bundet til en museknap, er det ikke nødvendigt, at input-API'en registrerer, når en tastaturtast slippes.

Selvom GLUT opfylder mange af de gældende krav, er der stadig mangler:

- Når bare et enkelt vindue bliver lukket i GLUT, lukker alle resterende vinder automatisk, og API'en lukker ned. Dette er på ingen måde hensigtsmæssigt, da brugeren gerne skal kunne åbne og lukke individuelle vinduer uafhængigt af resten.
- Selvom GLUT understøtter stereo buffers, kan API'en ikke skifte mellem flere grafikenheder. Det er derfor ikke muligt at bruge Dual Buffer stereovisningsmetoden via GLUT. Dual Buffers er ikke en vigtig del af SIDM, så denne mangel er ikke kritisk.
- GLUT registrerer ikke når musehjulet bevæges. Det er et krav til programmet, at musens hjul kan benyttes til at bevæge det vandrende mærke i dybden, så denne mangel er nødt til at blive udbedret.

4.4.3 Free OpenGL Utility Toolkit

Udviklingen af GLUT stoppede i 1998, hvilket efterhånden gør API'en aldrende. Da API'en ikke er udgivet som open source, kan andre programmører ikke arbejde videre med den. Flittige programmører valgte derfor at starte forfra og udvikle en API, der indeholdte de samme funktionaliteter som GLUT, men som kunne videreudvikles. Resultater blev *FreeGLUT* [R23].

FreeGLUT tilbyder stort set de samme funktionaliteter som GLUT. Enkelte funktioner er endnu ikke implementeret, men disse skal ikke benyttes af SIDM. Den store fordel ved FreeGLUT frem for GLUT er, at førstnævnte understøtter musehjulet og tillader at et vindue lukkes, uden at de resterende vinduer også lukkes ned.

Selvom FreeGLUT opfylder alle de kritiske krav til en vinduesstyrings-API, er der stadig problemer. Udvikleren af dette projekt har over flere dage forgæves forsøgt at kompilere en stabil version af FreeGLUT. Dette lykkedes desværre aldrig, og API'en må derfor opgives.

4.4.4 Open Source GLUT

Løsningen på problemet med at finde en vinduesstyrings-API kom med *OpenGLUT* [R24], der er et helt ny API. OpenGLUT bygger videre på FreeGLUT, med det primære formål at fjerne bugs og øge kompatibilitet. Dette må siges at være lykkedes: Kompileringen af OpenGLUT kører uden problemer, og selv med omfattende test er det ikke lykkedes at genskabe problemerne

fra FreeGLUT. OpenGLUT tilbyder de samme funktionaliteter som FreeGLUT. De få udvidelser der er føjet til OpenGLUT, skal ikke benyttes i SIDM.

Valget af API til vinduesstyring er faldet på OpenGLUT. Hermed er funktionaliteterne F1 og F3 også understøttet (se afsnit 4.2).

4.4.5 Tekstuel information

Selvom OpenGLUT understøtter alle kritiske krav, er der stadig områder, hvor API'en er mangelfuld. Et er disse er i præsentation af tekstuel information. OpenGLUT tilbyder fonte, der kan benyttes til at skrive tekst direkte i grafikområdet. En mere ønskværdig løsning ville dog være, hvis teksten kunne skrives på en præsentabel måde uden for det grafiske område. Fx i en bar i bunden af vinduet. En yderligere bonus ville være, hvis brugeren af SIDM kunne markere og kopiere denne tekst.

I et forsøg på at opfylde ovenstående ønske er der blevet gennemgået flere API'er, der kan arbejde sammen med GLUT til at opbygge en egentlig brugerflade i et grafikvindue. Håbet er, at en sådan API kan opdele grafikvinduet i en statuslinie og det grafiske område. De to API'er, der er undersøgt, er *OpenGL User Interface* (GLUI) [R25] og *A Picoscopic User Interface* (PUI) [R26]. Fællestrækket for de to API'er er, at de begge bliver rendered i OpenGL. Da tal og tegn kun eksisterer som pixels i et grafisk vindue, er det ikke muligt i nogle af disse API'er at markere og/eller kopiere tekst. Begge API'er løser problemet med en pæn præsentation af tekst, og de er begge nemme at implementere. Af de to API'er vælges PUI. Dette skyldes ikke nogen mærkbare fordele i denne API, men blot at udvikleren føler at PUI er struktureret bedre og nemmere at gå til.

The Fast Light Toolkit (FLTK) [R28] er et eksempel på en brugerflade-API, der ikke bliver rendered i OpenGL. Dette tillader, at brugeren kan markere og kopiere tekst fra vinduet. På nuværende tidspunkt mangler denne funktionalitet dog i Windows-versionen. FLTK er en omfattende API, der bl.a. indeholder sin egen GLUT emulator. API'en fokuserer dog på brugerfladen, og det grafiske interface er derfor ikke så nemt at gå til som i GLUT og SDL. Ved analysens afslutning er visse af de krævede funktionaliteter ikke identificeret i FLTK. Dette inkluderer bl.a. sikring af, at musen forbliver inden for vinduet, når den er usynlig. Det antages dog, at disse problemer kan løses med en mere dybdegående analyse. Pga. kompleksiteten i FLTK og det faktum, at FLTK ikke tilbyder yderligere brugbare funktionaliteter i forhold til PUI eller GLUI, vil FLTK ikke benyttes i SIDM.

Rimeligt sent i udviklingsforløbet blev der observeret uoverensstemmelser mellem PUI og database-API'en. Problemet blev ikke løst ved at skifte til GLUI, og det blev besluttet, at et skift til FLTK ikke kunne betale sig på dette tidspunkt. Se kapitel '6 Test' for en dyberegående beskrivelse af dette problem. Da det heller ikke var muligt at udskifte database API'en (se 4.7), er SIDM udviklet uden brug af en API til præsentation af tekstuel information. I stedet er et tekstfelt implementeret af udvikleren ved brug af fonte fra OpenGLUT.

4.5 MULTI-THREADING

Et program er nødt til at have kontrol over sin egen tråd, hvis det skal reagere på direkte input fra brugeren. GLUT sikrer dette ved at tage fuldt kontrol over tråden, når API'en initialiseres. Denne simple løsning er upraktisk, da den tvinger brugerprogrammet til selv at starte ekstra tråde op. Desuden skal SIDM foruden inputtråden også have en arbejdstråd (se afsnit 3.3.2), og er derfor nødt til som minimum at starte én tråd op selv.

Når to eller flere tråde kalder funktioner i samme program, er det vigtigt at programmet er synkroniseret korrekt. SIDM har brug for følgende funktionaliteter til trådbehandling og synkronisering:

- Opstart og nedlukning af tråde.
- Implementering af mutex (se afsnit 3.3.2).
- Implementering af en semafor, monitor eller lignende synkroniserings metode, der tillader, at en tråd sover, indtil den signaleres af en anden tråd.

Det er ikke det store problem at finde API'er der tilbyder ovenstående funktionaliteter. Der findes mange bredt favnende API'er, der enten forsøger at erstatte C++ standard bibliotek eller forbedre det på mange forskellige områder. Flere af disse API'er tilbyder også understøttelse af tråde og synkronisering. Af eksempler kan nævnes *C++ Portable Library* (*PTypes*) [R30] og *The Adaptive Communication Environment* (ACE) [R32]. Da SIDM ikke vil benytte anden end multi-thread delen af disse API'er, er en omfattende bibliotekspakke ikke det bedste valg: En sådan API vil optage unødigt meget plads både som kilde kode og som binær kode, og medmindre SIDM linker statisk til denne API, vil det kompilerede modul også optage mere plads på harddisken.

I enkelte tilfælde er multi-threading også implementeret i API'er, der er specialiserede til at understøtte netop dette. De eneste fundne eksempler, der opfylder de indledende krav, er *pthreads* og *Zthreads* [R31]. Mange moderne operativsystemer indeholder som standard en eller anden form for tråd-bibliotek. Desværre er de fleste af disse biblioteker unikke til det specifikke operativsystem. POSIX er en standard, der definerer en API til at skrive multi-thread applikationer. I de senere år er flere operativsystemer begyndt at adoptere denne standard. Grundet Microsofts markedsføringspolitik, vil Windows-systemer formentlig aldrig implementere denne standard. *POSIX threads for Win32* [R29] løser dette problem ved at tilbyde POSIX standarden til Windows-systemer.

Da pthreads allerede er implementeret på flere systemer, er denne API valgt til SIDM. Gennemgående test af pthreads viser ingen tegn på fejl. Hvis API'en senere skulle give problemer, burde det ikke være noget problem at skifte til en af de andre nævnte API'er.

4.6 INDLÆSNING AF BILLEDFILER

Den primære opgave i SIDM er at vise billeder på skærmen. Disse billeder kan eksisterer i flere forskellige formater, og SIDM skal kunne indlæse dem alle (se 2.3.3). SIDM har ikke behov for yderligere funktionaliteter end korrekt indlæsning af de nævnte formater.

Mange billedformater understøttes af deres egen specialiserede API. Som et alternativ til at inkludere en unik API for hvert billedformat, findes mere avancerede billedbehandlings-API'er, der understøtter flere billedformater samtidig. Eksempler på sådanne API'er er: *ImageMagick* [R35], *Paintlib* [R36], *FreeImage* [R33] og *Developer's Image Library* (DevIL) [R34]. Disse API'er benytter sig alle af de grundlæggende formatspecifikke API'er, men indeholder også yderligere funktionaliteter til billedbehandling.

De ovenstående API'er tilbyder hver især de nødvendige funktionaliteter. Her ud over understøtter API'erne en mængde yderligere billedformater og tilbyder hver især forskellige billedbehandlingsrutiner. Disse yderligere funktionaliteter skal efter al sandsynlighed ikke benyttes i SIDM, og de har derfor ikke den store indflydelse på valget af API. Valget er i stedet baseret på API'ens opbygning og kompleksitet.

DevIL er ikke mindre kompleks end de andre biblioteker, men udmærker sig ved dets modulære opbygning. API'ens funktioner er opdelt i tre underbiblioteker: De simple funktionaliteter - som fx indlæsning af billeder - ligger i grundbiblioteket *IL*, mens mere avancerede funktionaliteter er placeret i de udvidede biblioteker *ILU* og *ILUT*. Grundbiblioteket IL kan benyttes uden at implementere de andre to biblioteker, hvilket minimerer programmets størrelse. DevIL, der tidligere var kendt som *OpenIl*, er opbygget, så det minder meget om OpenGL, hvilket simplificerer samarbejdet mellem de to API'er. Af disse to grunde er DevIL valgt som billede-API til SIDM. Ligesom den valgte API til multithreading kan DevIL nemt blive udskiftet med en af de andre API'er, hvis den viser sig at give problemer.

4.7 DATABASEFORBINDELSE

SIDM skal benytte en standard database til at opbevare data over kameraer, billeder og modeller. Med en standard database menes en database, der kan tilgås fra eksisterende programmer. Eksempler kan være *MySQL* og *Microsoft Access*. I stedet for at fastsætte en specifik database, som SIDM skal benytte, er det i stedet valgt at benytte *ODBC*.

Open Database Connectivity (ODBC) er en standardiseret API, der benyttes som et forbindelsesled til en database. Hverken operativsystem, programmeringssprog eller database er fastlåst i denne API. Microsoft Windows var den første platform, der implementerede ODBC, men i dag eksisterer også versioner til bl.a. Unix og Linux.

Den store fordel ved at benytte ODBC frem for en databasespecifik API er, at brugeren af SIDM selv kan vælge hvilken database vedkommende vil benytte. SIDM vil blive udviklet på en Windows platform, og en *Microsoft Access Database* vil benyttes under udviklingen. Fordelen ved denne database frem for mange andre er, at den allerede er implementeret i styresystemet, hvilket gør programmet simpelt at installere.

ODBC er en rimelig omfattende API og derfor tidskrævende at sætte sig ind i. Der ønskes derfor at finde en mellemliggende API, der kan gøre database forbindelsen simplere, mere intuitiv og frem for alt hurtigere at implementere. Der findes flere kommercielle API'er på markedet til dette formål, men det er kun

lykkedes at finde en enkelt API, der opfylder krav om ikke at være kommerciel og kunne køre på flere platforme. Denne API er *LibODBC++* [R37], der tidligere blev kaldt *FreeODBC*. Af mangel på flere valgmuligheder, benyttes denne API i SIDM.

I kapitel 6 beskrives hvordan LibODBC formentlig har været medskyldig i diverse fejl observeret i SIDM. Den eneste erstatning til LibODBC er at benytte ODBC API'en direkte. Fejlene blev desværre identificeret for sent til, at det tidsmæssigt var muligt at skifte til denne anden løsning.

4.8 DEMOBRUGERPROGRAM

Som beskrevet i kravspecifikationen er det en del af dette projekt at udvikle et simpelt demobrugerprogram til at teste SIDM-modulet. Dette program skal kun køres på en Windows-maskine og kan derfor skrives i et Windows-specifik programmeringssprog. Valget af en API til opbygning af dette demobrugerprogram er derfor meget frit:

Da projektet udvikles i Visual Studio, vil det åbenlyse valg ligge på M*icrosoft Foundation Class (MFC) Library*. MFC er en omfattende C++ API, der bl.a. tilbyder opbygning af detaljerede brugerflader. Visual Studio indeholder en service, der i form at en tegneprogramslignende brugerflade kan benyttes til hurtigt og simpelt at opbygge en brugerflade i MFC. Ulempen ved brug af denne service er, at programkoden bl.a. pga. ressource filer bliver unødvendig kompleks. Blev brugerfladen udviklet uden brug af den førnævnte service, kunne brugerfladen skrives på en mere simpel og direkte måde.

Det er tanken, at SIDM-modulet skal kunne kaldes fra andre programmeringssprog end C++. Ifølge Microsoft vil fremtidens programmeringssprog være baseret på .NET, så det kunne være interessant at vise, at SIDM-modulet også kan køres fra et .NET programmeringssprog. Valget af en API til demobrugerprogrammet er derfor faldet på C#, som udvikleren før har haft gode erfaringer med:

Ligesom til MFC indeholder Visual Studio en service til at opbygge brugerflader i C#. Fordelen ved C# er, at den kode, der generes af servicen, ligner den kode, man selv ville skrive uden brug af servicen. Grundet garbage collector, klasseudvalg og generelt opbygning af koden er C# efter udviklerens mening hurtigere og nemmere at skrive kode i. Den færdige kode vil af samme årsag være nemmere at overskue.

Den store ulempe ved C# er, at almindelig C++ kode skal konverteres til hvad Microsoft kalder *managed code*, før .NET kan benytte koden. Der kræves derfor et interface, der skal ligges oven på SIDM, før modulet kan benyttes i C#. Det forventes at udviklingen af dette interface vil være tidskrævende. Efterfølgende vil det færdige interface og dermed SIDM-modulet dog kunne benyttes af andre sprog i .NET pakken, hvilket forhåbentlig vil opveje for den forbrugte tid.

4.9 OPSUMMERING

Dette kapitel analyserer hvilket programmeringssprog og hvilke programbiblioteker, der skal benyttes til implementering af SIDM.

Primært pga. den gode ydeevne er C++ valgt som programmeringssprog til implementering af SIDM. For at vise SIDM's alsidighed, vil demobrugerprogrammet implementeres i C# .NET.

SIDM vil benytte OpenGLUT til vinduesstyring og modtagelse af input, OpenGL til grafikken, pthreads til synkronisering og administration af tråde, DevIL til at indlæse billeder fra diverse formater, samt LibODBC++ til kommunikation med Database via ODBC.

5 Implementering

Det implementeringsspecifikke design og selve implementeringen bliver samlet gennemgået i dette kapitel. Hvor det implementeringsuafhængige design i kapitel 3 også inkluderer designinformationer om fremtidige udvidelser, vil dette kapitel kun gennemgå de dele, der allerede er implementeret.

Da det grundlæggende design allerede er gennemgået i kapitel 3, vil dette kapitel kun beskrive interessante områder, der afviger eller ikke er blevet præciseret nok i det implementeringsuafhængige design. Ønskes yderligere detaljer om designet og programmets opbygning, henvises til programdokumentationen, der findes på den medfølgende CD.

Primært pga. de benyttede navngivningsregler (se afsnit 5.5.1) har klasser og typer ændret navn fra designet til implementeringen. I alle tilfælde bør sammenhængen mellem disse navne være nem at identificere.

5.1 UDVIKLINGSMILJØ

Til udviklingen af SIDM er der i dette projekt benyttet *Microsoft Visual Studio .NET 2003* [R43]. SIDM er ikke testet i andre udviklingsmiljøer end *Visual Studio*, men modulet burde teoretisk set kunne kompileres på en hvilken som helst C++ kompiler, forudsat at de benyttede API'er også er kompileret til denne kompiler. Disse API'er er prækompileret til dynamiske biblioteker, før de benyttes i SIDM.

Visual Studio opdeler udviklede programmer i projekter og solutions. Et projekt er et enligt program med dertilhørende kodefiler. Hvert projekt kan kompileres enten til en eksekverbar fil (.exe) eller til et programbibliotek (.lib og .dll). SIDM modulet er et eksempel på et sådant projekt, der almindeligvis kompileres til et dynamisk bibliotek, så det kan benyttes fra andre programmer. Foruden dette projekt er der benyttet fem andre projekter til udviklingen af SIDM. To af disse projekter definerer interfaces til SIDM:

- *SIDMwrap* definerer et Managed Code interface til SIDM. Dette interface benyttes af .NET applikationer.
- *SIDMfunctionInterface* definerer et funktionsbaseret interface til SIDM, der bl.a. kan benyttes af Delphi.

De resterende tre projekter definerer demobrugerprogrammer, der benyttes til at teste SIDM's tre interfaces: To meget simple brugerprogrammer skrevet i C++ benyttes til at teste SIDM's basisinterface og det funktionsbaserede interface. Et mere avanceret brugerprogram skrevet i C# benyttes til at teste interfacet til Managed Code.

En solution er en samling af projekter, der kan kompileres samlet eller hver for sig. Udviklingen af SIDM benytter fire solutions. En solution til hvert demobrugerprogram med tilhørende interface og SIDM, samt en solution, der kun indeholder SIDM. Denne sidste solution benytter en simpel *main*-funktion, til at opstarte en en-tråds-version af SIDM.

Visual Studio indeholder en avanceret debugger, der simplificerer fejlfinding i koden. Desværre er denne debugger ikke kompatibel med det benyttede trådbibliotek, hvilket forhindrer debuggeren i at påvirke eller læse fra andre tråde end den primære tråd. Den førnævnte en-tråds-version af SIDM er derfor benyttet til at debugge hovedsageligt vinduesmodulet. Denne version af SIDM kan desværre ikke genskabe alle typer af fejl, der eksisterer i fler-tråds-versionerne af SIDM, og det var derfor nødvendigt at benytte en anden løsning til at finde disse fejl (se afsnit 5.3.4).

5.2 INDLÆSNING OG VISNING AF BILLEDER

En af de mere krævende arbejdsopgaver i dette projekt var at vise billeder korrekt. Dette hovedafsnit beskriver hvordan SIDM genererer og viser billeder i et grafikvindue.

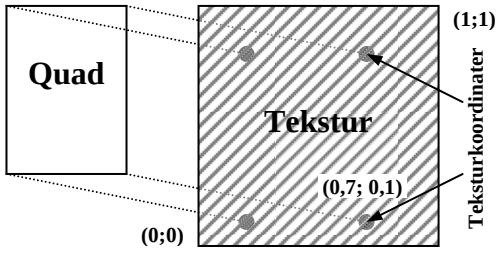
OpenGL tilbyder to forskellige metoder til visning af billeder:

Den direkte metode er at sende billedet umodificeret til 'Per-fragment operations' trinnet. Dvs. så hvert fragment svarer til en pixel i billedet. (se afsnit 4.3.1 *OpenGLs Rendering Pipeline*). Denne metode er simpel at implementere men tilbyder ingen direkte metode til at zoome i billedet eller udføre andre lineære operationer som fx normalisering (se afsnit 1.6). Disse operationer kan dog udføres af computerens processor, inden billedet sendes til OpenGL.

Den anden metode til at vise billeder i OpenGL er, at placere billedet som en tekstur på en geometrisk flade. Som beskrevet i afsnit 4.3.1 bliver teksturkoordinater sendt sammen med de geometriske objekter som vertex-data. Grafikkortet kan derfor udføre lineære operationer på disse data og dermed på billedet. Dette tillader, at zoom og normalisering af billedet beregnes på grafikkortet, hvorfor denne metode er blevet valgt.

5.2.1 Teksturkoordinater

En tekstur er et billede, der placeres ovenpå et geometrisk objekt for at give dette en specifik overflade. I SIDM vil disse geometriske objekter være simple, rektangulære flader kaldet quads. Når OpenGL skal placere en tekstur på en sådan quad, bliver teksturens position bestemt ud fra teksturkoordinater. Disse koordinater beskriver beliggenheden af hvert af fladens hjørner i teksturen (se Figur 5-1).



Figur 5-1: OpenGL holder styr på en teksturs beliggenhed på et objekt vha. teksturkoordinater. Disse koordinater defineres som en fraktion af teksturens størrelse. Dvs. x-koordinaten er 0,0 i teksturens venstre side, 0,5 i centrum og 1,0 i teksturens højre side.

I SIDM skal teksturerne hverken strækkes eller roteres, når de placeres på fladen. Teksturkoordinaterne vil derfor ligge parvis på de samme to horisontale og vertikale linier. Dvs. der skal kun kendes fire værdier til at bestemme alle fire sæt teksturkoordinater: TCx_{start}, TCx_{end}, TCy_{start} og TCy_{end}. Fx vil den øverste venstre teksturkoordinat have koordinatsættet (TCx_{start}, TCy_{end}).

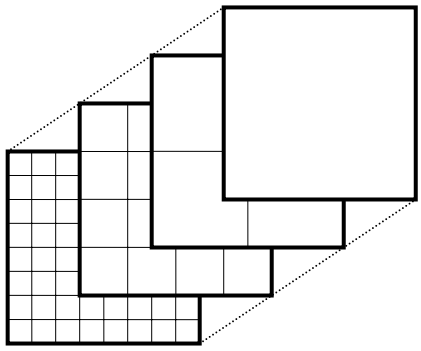
5.2.2 Mipmapping

Teksturen placeres på den geometriske flade i 'Per-fragment operations' trinnet i OpenGL's rendering pipeline (se afsnit 4.3.1). Kun i de færreste tilfælde vil teksturen placeres således, at hver texel (en pixel i teksturen) svarer præcis til en pixel i framebufferen. Almindeligvis vil teksturen enten skulle forstørres eller formindskes. I disse tilfælde skal OpenGL bestemme hvilken farve, hver enkelt pixel skal have ud fra de inkluderede texels. Præcis hvordan OpenGL beregner disse farver kan indstilles i API'en. Den mest naturtro metode er en linier beregning, og det er denne metode, der vil benyttes i SIDM.

Ved en forstørrelse vil én texel blive til flere pixels i framebufferen. Farven på disse pixels bestemmes ud fra et vægtet middel af de nærmeste texels.

Når en tekstur formindskes, vil en pixel i framebufferen beregnes ud fra mere end én texel. Farven på denne pixel bliver et gennemsnit af farven af de inkluderede texels. Denne proces, der kaldes *anti-aliasing*, kan være krævende at udføre, hvis pixelen indeholder for mange texels. OpenGL kan reducere beregningstiden ved kun at benytte enkelte af de inkluderede texels til beregning af pixelfarven, men dette kan forringe kvaliteten af billedet ved at danne synlige 'farvefejl' i billedet.

Mipmapping er en metode, der benyttes til at løse formindskelsesproblemet i OpenGL. Et sæt af mipmaps vil typisk blive genereret, når teksturen præpareres. Et mipmap sæt indeholder et antal bitmap billeder, der hver især viser en kopi af basisteksturen, men med en lavere detaljegrad. Det første mipmap genereres ud fra teksturen ved at sammensætte fire texels i teksturen til én pixel i mipmappet. Det andet mipmap genereres på tilsvarende måde ud fra det første mipmap, osv. Hvis basisteksturen har en sidelængde på 2^n pixels, vil der typisk blive genereret n mipmaps. Fx en tekstur med en basisstørrelse på 64×64 (sidelængde 2^6), kan have seks mipmaps med størrelserne: 32×32, 16×16, 8×8, 4×4, 2×2 og 1×1 (se Figur 5-2).



Figur 5-2: Mipmaps af en 8×8 tekstur.

I stedet for at benytte basisteksturen, når en tekstur skal vises i formindsket udgave, kan programmet benytte de to mipmaps, der svarer bedst til den viste størrelse af teksturen. Hvis en tekstur fx skal vises, så den kun fylder 20×20 pixels på skærmen, vil en interpolation af 32×32 mipmappet og 16×16 mipmappet blive benyttet.

Fordelene ved mipmaps er, at anti-aliasing allerede er udført, hvilket øger renderingshastigheden og reducerer uønskede artefakter. Ulemperne ved metoden er, at det tager længere tid at generere teksturen og teksturen vil forbruge mere RAM. Det samlede forbrug af RAM vil øges med: $¼ + ¼^2 + ¼^3 + \ldots = {}^1/_3$. Dvs. en tekstur på 3 Mb vil fylde en ekstra Mb, hvis der genereres mipmaps over den.

Artefakter fra manglende anti-aliasing er mest markante i kunstige billeder med kraftige farveovergange. Da SIDM typisk vil benyttes til at vise fotografier, vil artefakter formentlig ikke give de store problemer. Desuden vil et billede i SIDM sjældent blive vist ved et meget lavt detaljeniveau, så der vil typisk ikke være brug for kraftige minimeringer. Det antages af disse grunde, at SIDM normalt ikke vil have den store fordel af at benytte mipmaps. Mipmaps kan dog stadig have den fordel, at de kan forbedre ydeevnen af langsommere maskiner. Det vælges derfor at tilføje en indstilling i konfigurationsfilen, der kan slå mipmaps til og fra i SIDM.

5.2.3 Texture Tiling

For at et billede kan benyttes som en tekstur i OpenGL, kræves det at billedet er kvadratisk med sidelængder lig en potens af to. Dvs. bredde = højde = 2^n, hvor n er et helt, positivt tal. Herudover lægger grafikkortet en øvre grænse på størrelsen af en tekstur. For et moderne grafikkort vil denne maksimumsgrænse typisk ligge på 4096×4096 texels, men for ældre grafikkort kan den ligge nede på 1024×1024 texels eller endnu mindre. Da billeder, der skal vises i SIDM, ofte vil overstige disse grænser, kan SIDM blive nødt til at opdele et billede i flere teksturer. Dette kaldes *texture tiling*. Der er til dette formål inkluderet den ekstra klasse *CTexture*, der administrerer en enkelt tekstur, og benyttes af *CGiImage*.

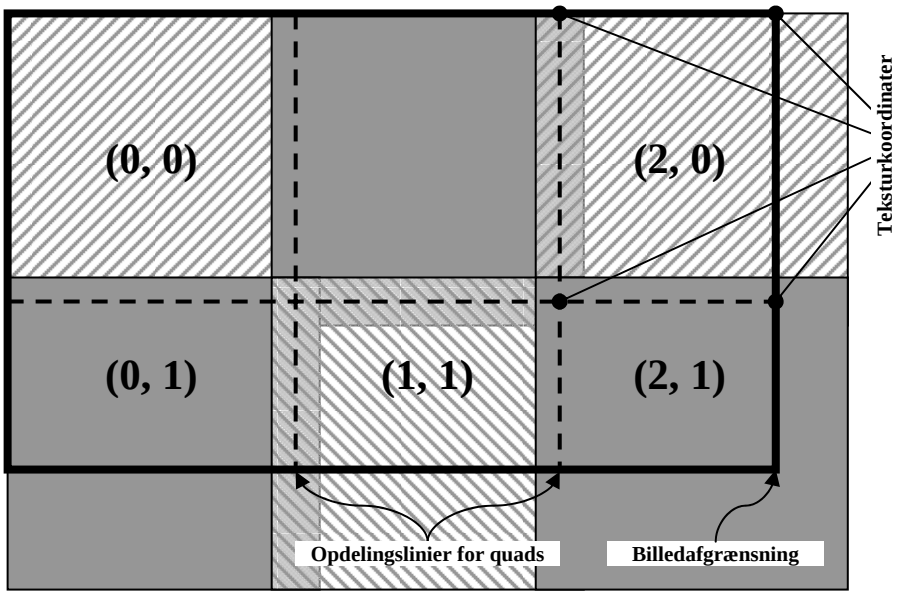
Ved at opdele et billede i flere teksturer kan et billede af en hvilken som helst størrelse teoretisk set vises i OpenGL. Med 'teoretisk set' menes, at tilgængelig hukommelse kan sætte en øvre grænse på, hvor store billeder der kan vises.

Når et billede opsplittes i flere teksturer, som efterfølgende bliver vist ved siden af hinanden, vil der opstå synlige linier, hvor to teksturer mødes. Årsagen til dette vil typisk være, at farven på pixels i kanten af teksturen ikke bliver beregnet korrekt:

Som beskrevet i det forrige afsnit benyttes der generelt flere texels til beregning af farven på en enkelt pixel i framebufferen. Hvis en eller flere af disse texels ligger uden for teksturen, vil OpenGL ikke kende deres korrekte farve. Med den rette konfigurering vil OpenGL antage, at disse manglende texels har samme farve som den nærmeste kendte texel. Dette passer dog ikke altid overens med farven af den samme pixel i naboteksturen, hvilket medfører den synlige linie.

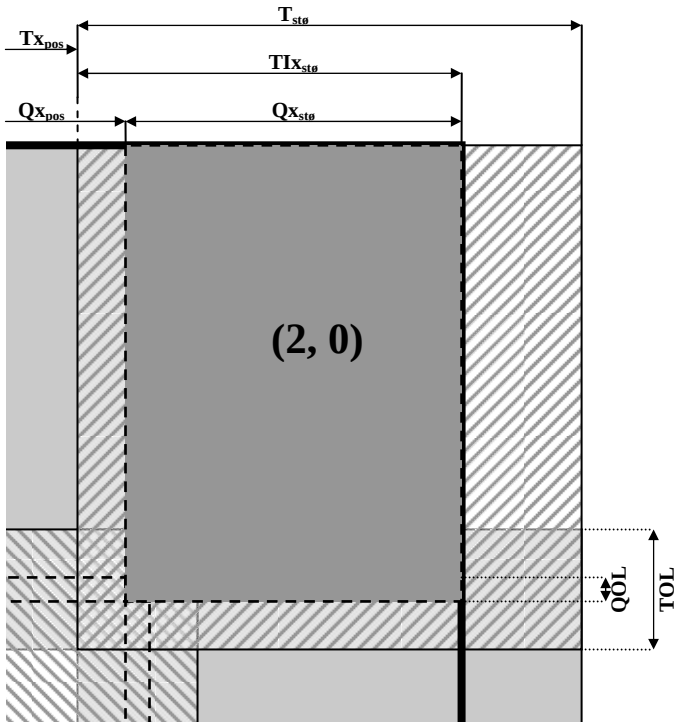
Løsningen på dette problem er at overlappe teksturerne, når de udklippes af bil-

ledet. Teksturkoordinaterne placeres en tilsvarende længde inden i teksturen, så kanten af teksturen ikke er synlig på quad'en (se Figur 5-3).



Figur 5-3: Texture tiling. Et billede opdeles i seks teksturer, der hver især overlapper naboerne en given længde. De stiplede linier viser den del af teksturen, der er synlig på hver enkelt quad.

Hvor mange pixels, to teksturer skal overlappe hinanden med, vil afhænge af billedets indhold og zoomfaktor og om hvorvidt der benyttes mipmaps eller ej.



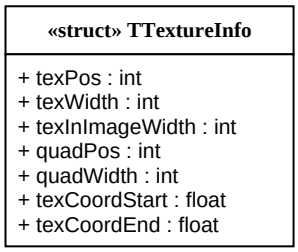
Figur 5-4: Benyttede betegnelser ved texture tiling. Tilsvarende størrelser skal kendes for den lodrette retning (y). Figuren viser det øverste højre hjørne af Figur 5-3.

Selvom teksturerne overlappede hinanden, blev der på nogle grafikkort stadig observeret sorte streger mellem de forskellige quads. Disse streger skyldtes efter al sandsynlighed afrundingsfejl på de float-værdier, der bestemmer quad'ens position. Dette problem blev løst ved, at nabo quads overlapper en enkelt texel.

Figur 5-4 viser de størrelser, der skal være kendte for korrekt at opbygge disse quads. Alle værdier måles i pixels.

- **TOL:** Teksturoverlap definerer det antal pixels, to naboteksturer skal overlappe hinanden med, når de udtrækkes fra billedet. Værdien af denne vil formentlig være 2 eller 4.
- **QOL:** Quad-overlap definerer det antal pixels, to nabo-quads skal overlappe hinanden med. Denne vil generelt kun have værdierne 0 eller 1.
- **Tx_{pos} & Ty_{pos}:** Teksturens startposition, dvs. afstand fra øverste, venstre hjørne af billedet, til øverste venstre hjørne af teksturen.
- **$T_{stø}$:** Teksturens størrelse, dvs. både bredde og højde.
- **$TIx_{stø}$ & $TIy_{stø}$:** Størrelse af den del af teksturen, der ligger inden for billedet. For teksturer, der ikke rækker ud fra billedet, vil $TIx_{stø} = TIy_{stø} = T_{stø}$.
- **Qx_{pos} & Qy_{pos}:** Quad'ens startposition, dvs. afstand fra øverste, venstre hjørne af billedet til øverste venstre hjørne af quad'en.
- **$Qx_{stø}$ & $Qy_{stø}$:** Quad'ens størrelse.

Foruden ovenstående værdier skal teksturkoordinaterne også bestemmes, dvs. TCx_{start}, TCx_{end}, TCy_{start} og TCy_{end}. All disse værdier beregnes i metoden *GenerateTextureInfo* i *CGiImage*-klassen og gemmes i strukturen *TTextureInfo* (se Figur 5-5).



Figur 5-5: Struktur, der indeholder konstruktionsdata for en tekstur.

Hver instans af denne struktur gemmer information om en retning, enten lodret eller vandret. Er et billede fx delt op i 3×2 felter som i eksempelet i Figur 5-3, vil der genereres fem instanser af *TTextureInfo*; tre til den vandrette information og to til den lodrette. Disse informationer benyttes af flere forskellige funktioner ved indlæsning af billeder og opbygning af teksturerne.

5.2.4 Indlæsning af billeder

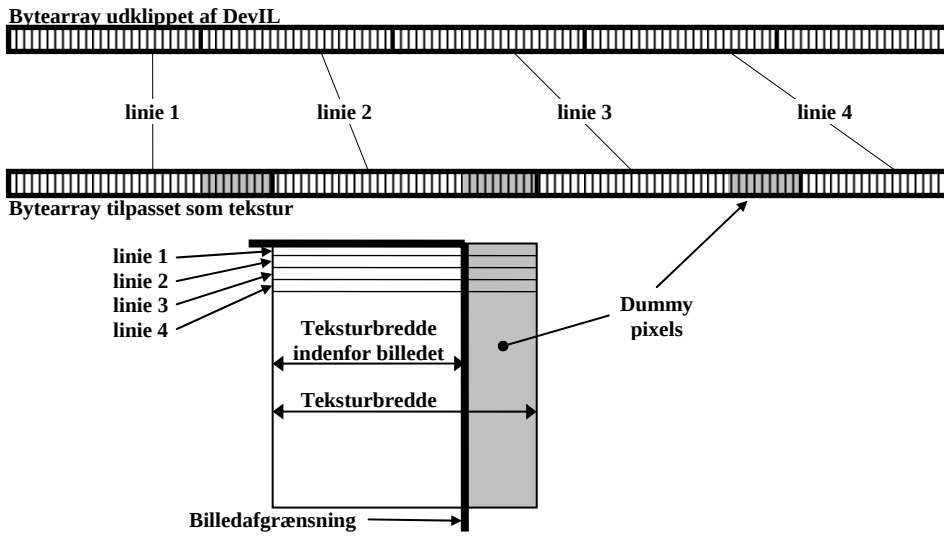
Når et billede skal indlæses og forbehandles i SIDM, gennemgår det følgende trin:

- **Indlæsning:** DevIL indlæser billedet fra billedfilen og gemmer billeddata i hukommelsen.

- **Udklip:** Det indlæste billede deles op i kvadratiske stykker, der kan bruges til teksturer.
- **Teksturgenerering:** OpenGL genererer teksturer ud fra de udklippede billedstykker. Dette kan enten være enkeltstående teksturer eller mipmaps.

Opdeling af et billede i mindre stykker blev til at starte med udført af en indbygget funktion i DevIL. Hvis en ønsket teksturkvadrant stikker uden for billedet, vil denne funktion kun udklippe den del af billedstykket, der befinder sig inden for billedet. Et sådant billedstykke skal efterfølgende udvides, så det bliver kvadratisk, dvs. der skal tilføjes dummy pixels til billedet. Disse ekstra pixels vil ikke blive vist på skærmen, men kan stadig have indflydelse på farven af den yderste viste pixel (se afsnit 5.2.3). Disse pixels sættes af denne grund til farven af den nærmeste kendte pixel.

Pixeldata gemmes række for række i et endimensionalt bytearray. Hvis et sådant billede skal udvides i vandret retning, skal alt pixeldata kopieres endnu engang (se Figur 5-6). Denne ekstra kopiering af pixeldata vil kunne undgås, hvis dummy pixels tilføjes mens data udklippes.



Figur 5-6: Udklipning af billedstykke vha. DevIL. Hvis teksturen stikker uden for billedet, skal billeddata efterbehandles, så teksturen bliver kvadratisk. I det viste tilfælde indsættes dummy pixels efter hver linie med kendt pixeldata.

Et andet ønske til en udklipningsfunktion er, at den tilbyder muligheden for at vende billedet på hovedet. Dvs. at den skal placere nederste linie fra billedet øverst i billedstykket. Hvor almindelige billedformater gemmer pixeldata fra den øverste linie og ned, er der enkelte formater (fx BMP og TIF), der gemmer pixeldata fra den nederste linie og op. Hvis ingen foranstaltninger bliver taget, vil OpenGL vise sådanne billeder på hovedet. En måde at vende billedet på er, at flytte rundt på teksturkoordinaterne. Disse koordinater skal bruges ved hver optegning af billedet, så en pænere løsning vil være at vende billedet om, mens det bliver udklippet.

Det blev besluttet at udvikle en specialiseret udklipningsmetode til SIDM. Den

primære årsag var ikke mangel på de ønskede funktionaliteter, men at DevIL-funktionen havde en overraskende ringe ydeevne.

Handling	Operationstid
Indlæsning af billede fra BMP fil	15 ms/pixel
Indlæsning af billede fra JPG fil	50-70 ms/pixel
Indlæsning af billede fra PNG fil	100-140 ms/pixel
Udklip af billedstykke	48 ms/pixel
Teksturgenerering, normal	8 ms/pixel
Teksturgenerering, mipmaps	93 ms/pixel

Figur 5-7: Operationstider for indlæsning og præparering af billeder. Testen er udført på en *Barton 2500+* processor med et *GeForce Ti4400* grafikkort med 128 mb RAM.

Figur 5-7 viser operationstider for de tidskrævende operationer ved indlæsning af et billede. Det ses, at indlæsning af et pakket billede (JPG/PNG) og generering af mipmaps er klart de mest tidskrævende processer. Normalt forventes almindelige teksturer dog at blive brugt frem for mipmaps. Hvis der herudover benyttes et ikke pakket billedformat som fx BMP, er det pludselig udklipning af billedstykker, der forbruger 60-70 % af tiden. Dette tal virker unødvendigt højt, og det tyder på, at DevIL's udklipningsfunktion ikke er særlig optimeret.

Den udviklede metode til udklip af billedstykker kaldes *ExtractTexture* og findes i *CGiImage* klassen. Det blev på alle områder forsøgt at optimere ydeevnen af denne metode. Udover at tilbyde de to ønskede funktionaliteter viste metoden sig at være over dobbelt så hurtig som funktionen i DevIL. Dvs. på den benyttede computer kunne metoden udklippe et billede på under halvdelen af den tid, det tog udklipningsfunktionen fra DevIL at udklippe det samme billede.

Den tidskrævende del af den udviklede udklipningsmetode er kopiering af en byte (*char*) fra billedets bytearray til det bytearray, der skal gemme billedstykket. De fleste computere nu om dage er 32-bit. Dvs. de kan behandle op til 32-bit data synkront. Kopiering af én byte (8-bit) af gangen udnytter slet ikke denne busbredde, og det blev derfor forsøgt med en optimeret version af udklipningsfunktionen: I stedet for at kopiere en byte ad gangen, bliver fire på hinanden efterfølgende bytes fra arrayet type castet til en integer (32 bit), og denne integer kopieret. Denne modifikation øgede ydeevnen af funktionen med yderligere 200%.

Udklip med DevIL	48 ms/pixel
Udklip med 8-bit version	21 ms/pixel
Udklip med 32-bit version	7 ms/pixel

Figur 5-8: Operationstider for udklipning af billeder.

Figur 5-8 sammenligner operationstider for udklipning ved brug af de tre funktioner. Den endelige 32-bit version af *ExtractTexture* viste sig at være over seks gange hurtigere end DevIL-funktionen. Implementeringen af denne funktion har mere end halveret tiden det tager at indlæse og forbehandle et ikke pakket billede til almindelige teksturer.

Ligegyldigt hvor meget udklipningsfunktionen er blevet optimeret, er det stadig en tidskrævende proces at indlæse og forbehandle billeder, og dette arbejde udføres derfor af arbejdstråden. Desværre er det i den nuværende version af SIDM ikke muligt for arbejdstråden at generere teksturer eller mipmaps. Denne operation udføres af OpenGL og kræver en OpenGL kontekst (se afsnit 4.4). Desværre indeholder OpenGLUT ingen funktionaliteter til at opstarte en OpenGL kontekst, der ikke er bundet til et grafikvindue, og arbejdstråden kan derfor ikke få sin egen kontekst. Indtil videre vil disse operationer udføres af inputtråden, som derfor i disse perioder ikke vil modtage brugerinput.

Der er tre løsninger på dette problem:

1. Arbejdstråden kan åbne sit eget grafikvindue, som efterfølgende minimeres eller placeres uden for skærmen. Brugeren vil i de fleste operativsystemer have mulighed for alligevel at opdage dette vindue, og denne løsning er ikke praktisk.
2. Hvor opbygning af mipmaps er meget tidskrævende, går generering af almindelige teksturer relativt hurtigt og generer derfor ikke brugeren mærkbart. Hvis mipmaps blev opbygget uden for OpenGL, ville dette kunne gøres i arbejdstråden og derfor ikke genere brugeren. De opbyggede mipmaps skal selvfølgelig overføres til OpenGL, men ligesom generering af almindelige teksturer, vil denne operation ikke være særlig tidskrævende.
3. En anden API, der kan åbne en OpenGL kontekst uden et grafikvindue, benyttes i stedet for, eller samtidig med OpenGLUT.

Indtil videre forbliver problemet dog uløst, og brugeren må derfor acceptere de uundgåelige pauser.

5.2.5 Stereoskopisk visning af billeder

Den stereoskopiske visning administreres af basisklassen *CGiBaseItem* (kaldet *GraphicItem* i designet). Denne klasse nedarves af alle grafikvisningsklasser. Klassen administrerer de tre forskellige stereovisningsmetoder på følgende måde:

Anaglyph: Hvis anaglyph-metoden benyttes, vil *CGiBaseItem* sørge for at initialisere den korrekte farveblandingsmetode (*blending*), samt sætte grundfarverne korrekt til fx rød eller grøn.

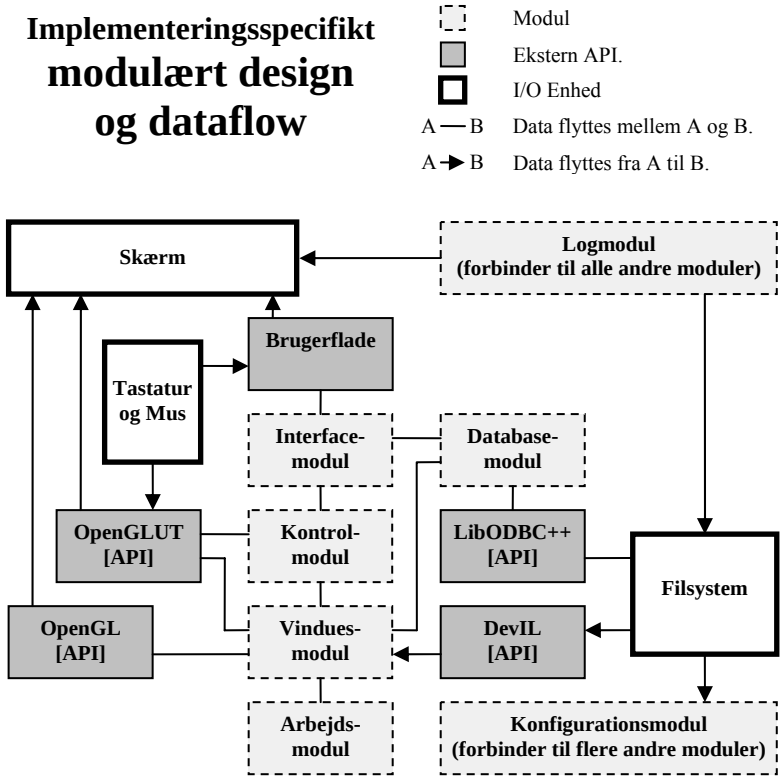
Page flipping: Page flipping benyttes både i forbindelse med shutter glasses og sammen med et polaroidfilter foran skærmen. OpenGL inkluderer såkaldte stereobuffers (se afsnit 4.3.1), dvs. en højre og en venstre buffer. *CGiBaseItem* administrerer skift mellem disse buffers, inden et grafikelement tegnes. Ved at opdele grafik på disse to buffers burde OpenGL i samarbejde med grafikkortets driver automatisk implementere page flipping. Pga. tidsmangel er denne funkti-

onalitet kun testet meget overfladisk. Denne test viste, at page flipping i øjeblikket ikke virker korrekt.

Dual Display: Dual display benyttes både af HMD og ved benyttelse af to projektorer med polariseret lys. Metoden kræver enten to grafikkort eller et kort med to grafikudgange, hvilket desværre ikke understøttes af OpenGLUT. For at implementere dual display, må et alternativ til OpenGLUT benyttes.

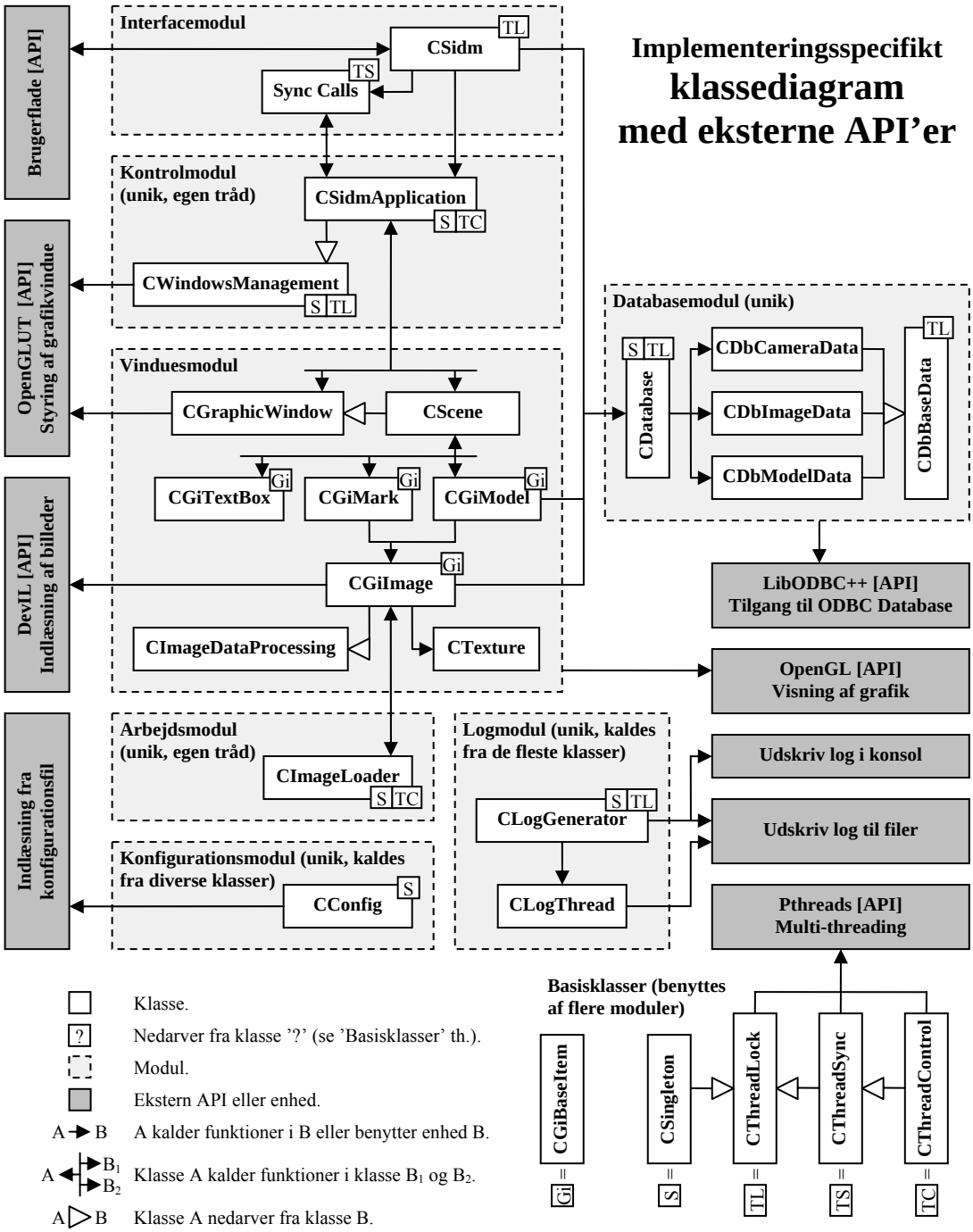
5.3 DESIGNÆNDRINGER OG -SPECIFICERINGER

Det endelige design af SIDM har gennemgået flere ændringer i forhold til det implementeringsuafhængige design. Dette afsnit gennemgår de mest markante af disse ændringer samt enkelte specificeringer på områder, der ikke var præciseret i det originale design.



Figur 5-9: Implementeringsspecifikt modulært design af SIDM. Figuren viser dataflow internt i programmet og mellem eksterne enheder. Figuren bruger ikke UML.

Figur 5-9 viser det implementeringsspecifikke modulære design, samt hvordan data bevæges internt i programmet og mellem eksterne API'er og enheder. Den eneste ændring fra det originale design er, at fejlbehandlingsmodulet er erstattet af et logmodul (se afsnit 5.3.3). Desuden bliver konfigurationsmodulet nu tilgået fra så mange klasser, at associationslinier til dette modul er udeladt af hensyn til overskueligheden af figuren.



Figur 5-10: Implementeringsspecifikt klassediagram af SIDM. Figuren beskriver desuden forbindelse til eksterne API'er og enheder. Figuren bruger ikke UML.

Det implementeringsspecifikke klassediagram kan ses på Figur 5-10. Som det ses, har programmet fået nogle nye klasser, hvoraf *CTexture* allerede er introduceret i afsnit 5.2.3. I de efterfølgende afsnit i dette kapitel vil de resterende nye klasser blive beskrevet.

- Afsnit 5.3.1 introducerer *CThreadLock*, *CThreadSync*, *CThreadControl* og *CSingleton*.

- Afsnit 5.3.2 introducerer *CWindowManagement* og *CGraphicWindow*.
- Afsnit 5.3.3 introducerer *CGiTextBox*.
- Afsnit 5.3.4 introducerer logmodulet med klasserne *CLogGenerator* og *CLogThread*.
- Afsnit 5.3.5 giver yderligere informationer om klassen *CConfig*.

5.3.1 Egenskabsklasser

En af forskellene mellem C++ og højere niveau programmeringssprog som C# og Java er, at klasser i C++ kan nedarve fra mere end én basisklasse. Denne mulighed udnyttes i SIDM i form af basisklasser, der kan give specifikke egenskaber til en klasse. SIDM benytter fire af disse klasser:

CThreadLock: Denne klasse tilbyder en mutex til den nedarvende klasse. Dvs. den tilføjer funktionerne *Lock*, *TryLock* og *Unlock*. Denne klasse benyttes af klasser, hvortil der skal være begrænset adgang. *CThreadLock* og de efterfølgende to thread-klasser benytter pthreads API'en. Denne API tilgås som et *Façade* design pattern (se afsnit 3.5) og bliver dermed isoleret fra brugeren af klassen. Dette tillader fx, at pthreads kan udskiftes med en anden API uden at have indflydelse på resten af SIDM.

CThreadSync: En klasse, der nedarver *CThreadSync*, får både adgang til en mutex som ved ovenstående klasse samt synkroniseringsfunktionerne *Wait* og *Signal*. Kun syncronized call klasserne benytter *CThreadSync* direkte. I disse klasser benyttes synkroniseringsfunktionerne i forbindelse med returværdier til interfacet.

CThreadControl: Denne klasse tilbyder, at en nedarvende klasse kan starte sin egen tråd. *CThreadControl* tilbyder også synkroniseringsfunktionerne fra de to ovenstående klasser. De ekstra funktioner, denne klasse tilføjer, er *StartThread*, *IsRunning* (returnerer true hvis tråden kører) og *WaitForTermination* (den kaldende tråd pauses indtil tråden i klassen terminerer). Klassen, der nedarver *CThreadControl*, skal deklarere den rent virtuelle (pure virtual) funktion *ThreadEntrance*. Denne funktion kaldes, når tråden er startet op.

CSingleton: Singleton design pattern er beskrevet i afsnit 3.5. Flere klasser i SIDM skal implementere dette design pattern, og der blev derfor arbejdet på at finde en fælles løsning til denne implementering. Problemet ved at implementere Singleton pattern i en basisklasse er, at denne skal generere en instans af den nedarvende klasse. Problemet blev til sidst løst ved at definere basisklassen som en templateklasse. *CSingleton* tilføjer hermed et trådsikkert singleton pattern til den nedarvende klasse.

5.3.2 Implementering af OpenGLUT

Efter en længere analyse blev OpenGLUT valgt som API til input- og vinduesadministration. Selvom denne API tilbyder alle de vigtigste funktionaliteter, er den stadig mangelfuld på visse områder. Det er af denne grund ikke umuligt, at en senere videreudvikling af SIDM vil udskifte OpenGLUT med en anden API.

Ligesom for alle andre eksterne API'er i SIDM er det derfor vigtigt at holde OpenGLUT så isoleret som muligt fra resten af programmet.

Måden hvormed OpenGLUT administrer input, er meget forskellig fra metoden, der benyttes af andre lignende API'er. Det er derfor ikke nemt at isolere OpenGLUT på en sådan måde, at API'en kan erstattes af en vilkårlig anden input- og vinduesadministrations API.

Den metode, der blev valgt i det implementeringsspecifikke design, er, at implementere OpenGLUT i to klasser i to forskellige moduler. *CWindowManagement* nedarves af *CSidmApplication* i kontrolmodulet, mens *CGraphicWindow* nedarves af *CScene* i vinduesmodulet. Det kan umiddelbart virke som en rodet løsning, at implementere API'en på denne måde, men da OpenGLUT påvirker så mange områder af SIDM, menes dette alligevel at være den pæneste og mest effektive metode. Vigtigst er dog, at denne opdeling også kan benyttes af andre API'er, som benytter en anden metode til indlæsning af brugerinput. Dette tillader, at OpenGLUT kan skiftes ud med en anden API, uden at påvirke resten af koden.

CGraphicWindow implementerer alle vinduesspecifikke funktionaliteter fra OpenGLUT. Den nedarvende klasse (*CScene*) tilbydes hermed metoder til at åbne, lukke, og på anden måde påvirke det tilhørende vindue. Andre funktionaliteter, der tilbydes, er fx udskrift af tekst og flytning af musemarkøren. *CGraphicWindow* behandler også brugerinput til det angivne vindue. Klassen definerer en gruppe rent virtuelle funktioner, der hver især svarer til en type af brugerinput eller anden hændelse. Fx *MouseButton*, *MouseMove*, *ReshapeWindow* osv. *CScene* skal blot deklarerer disse funktioner, og *CGraphicWindow* vil sørge for, at de rigtige funktioner kaldes på de rigtige tidspunkter.

CWindowManagement implementerer på en tilsvarende måde alle de funktionaliteter, der ikke er forbundet til et enkelt vindue. Den vigtigste af disse funktionaliteter er *IdleFunction*, der kaldes hver gang OpenGLUT har et ledigt øjeblik. Dette funktionskald giver *CSidmApplication* adgang til inputtråden (som normalt holdes låst af OpenGLUT) og dermed mulighed for at indlæse og behandle funktionskald fra brugerprogrammet.

5.3.3 Udskrivning af tekst

Udskrivning af tekst i et grafikvindue viste sig at være mere besværligt end først antaget i det implementeringsuafhængige design. Af denne grund er der inkluderet en ny klasse, *CTextBox* til dette formål. Ligesom de andre klasser i SIDM, der repræsenterer objekter, som vises i grafikvinduet, nedarver *CTextBox* også fra *CGiBaseItem* (kaldes *GraphicItem* i det indledende design).

CGiTextBox henter via *CScene* informationer om vinduets nuværende tilstand fra *CGiMark* og *CGiModel*. Denne information udskrives i et tekstfelt, der placeres i det ene hjørne af vinduet. Foruden dette tekstfelt, kan *CGiTextBox* udskrive informationer i tekstbarer (tekstfelter med kun én tekstlinie), der vises over eller under tekstfeltet. Disse tekstbarer benyttes til at informere brugeren om ikke kritiske fejl i vinduet, samt indlæsningsstatus på billederne. Hvis SIDM har information om, hvor langt et givent trin i indlæsningen af et billede er nået,

vil denne information gives til brugeren i form af en tilsvarende udfyldning af den tilhørende tekstbar. Dvs. hvis et trin er halvvejs færdigt, vil tekstbaren være halvvejs udfyldt.

Brugeren af SIDM kan i konfigurationsfilen definere forskellige indstillinger for tekstfeltet og tekstbarerne:

- I hvilket hjørne af vinduet og hvor mange pixels fra dette hjørne tekstfeltet befinder sig. Hvis et af de øvre hjørner vælges, vil tekstbarer placeres under tekstfeltet, ellers vil barerne placeres over tekstfeltet.
- Størrelse af tekstfelt og tekstbarer, samt afstand mellem tekstbarer og mellem tekstlinier i tekstfeltet.
- Farver på tekst, tekstfelt og både fyldte og ikke fyldte tekstbarer. Alphakomponenten på disse farver kan også specificeres, hvilket tillader gennemsigtige tekstfelter.
- Hvilken information, der vises i tekstfeltet. Mulighederne er: objektkoordinater, billedkoordinater, om vinduet er i model- eller komparatortilstand, hvorvidt mærket er låst eller ej samt hvilket billede der er låst til sit målemærke (om noget).

5.3.4 Fejlfinding og fejlbehandling

Den interne fejlbehandling i SIDM benytter som forventet exception handling. Til dette formål er der i SIDM inkluderet en gruppe af små fejlbesked-klasser, der alle nedarver fra basisklassen *EException*. Hver af disse klasser svarer til en given fejltype, og når en fejl opstår, kastes en instans af den tilhørende fejlbeskedklasse. Fordelen ved at benytte mange forskellige exception klasser er, at klassens type kan benyttes til at specificere, hvilken fejl der er tale om.

Den nuværende version af SIDM indeholder endnu ikke funktionaliteter til at informere brugerprogrammet om fejl. I stedet for et fejlbehandlingsmodul, der kan kommunikere med brugerprogrammet, indeholder SIDM i øjeblikket et logmodul, der skriver både fejl- og fejlfindingsinformationer ud til filer og konsol. Under udviklingen af SIDM er dette modul mere anvendelig end fejlbehandlingsmodulet. Når det bliver nødvendigt, er det rimeligt simpelt at omdanne logmodulet til et fejlbehandlingsmodul.

Den primære årsag til implementeringen af logmodulet er hjælp til fejlfinding i programmet. Modulet samler al debug-information på ét sted i programmet og tillader fælles udskrivning til forskellige mål. Så længe brugerprogrammet kører fra en konsol, kunne et alternativ til logmodulet være, at udskrive information direkte fra de forskellige klasser til konsollen. C++ bruger de to streams - *cout* og *cerr* - til udskrivning i konsol. Disse streams udskriver tegn som de modtager dem. Hvis flere tråde kalder en af disse streams samtidig, vil de to outputlinier blandes sammen. Det er derfor nødvendigt med en samlet, trådsikker løsning til output.

En anden fordel ved en samlet løsning er, at det er nemmere at tilbyde flere forskellige muligheder for udskrivning. Den nuværende implementering af logmodulet kan udskrive loggen til flere forskellige mål:

- I konsollen, hvor hver tråd identificeres med et ID i starten af loglinien.
- I en enkelt tekstfil, hvor hver tråd identificeres som ovenstående.
- I en HTML fil, hvor loglinien fra forskellige tråde udskrives i forskellige farver.
- I flere tekstfiler, hvor hver tråd skriver til sin egen fil, og hver linie indledes af et tidsstempel til sammenligning mellem tråde.

Hvilke af disse udskrivningsmål, der benyttes, defineres i konfigurationsfilen. Denne fil definerer også hvilke logniveauer, der skal udskrives. En funktion, der sender en logbesked til logmodulet, definerer hvilket niveau denne besked har. Eksempel på niveauer er ERROR, WARNING, og DEBUG. En binær vektor i konfigurationen definerer hvilke af disse niveauer af logbeskeder, der skal udskrives.

Logmodulet indeholder to klasser: *CLogGenerator* er logmodulets interfaceklasse. Denne klasse indeholder en liste med én instans af *CLogThread* per tråd, der benytter logmodulet. Hver tråd er forbundet til sin egen instans af denne klasse via trådens ID. Første gang *CLogGenerator* kaldes af en given tråd oprettes en *CLogThread*, der passer til den kaldende tråd.

CLogGenerator implementerer Singleton design pattern og kan derfor blive tilgåets fra alle steder i SIDM. Den nemmeste måde at forstå opbygningen af logmodulet på, er ved at se på hvordan en udskrift til dette forløber (se Figur 5-11):

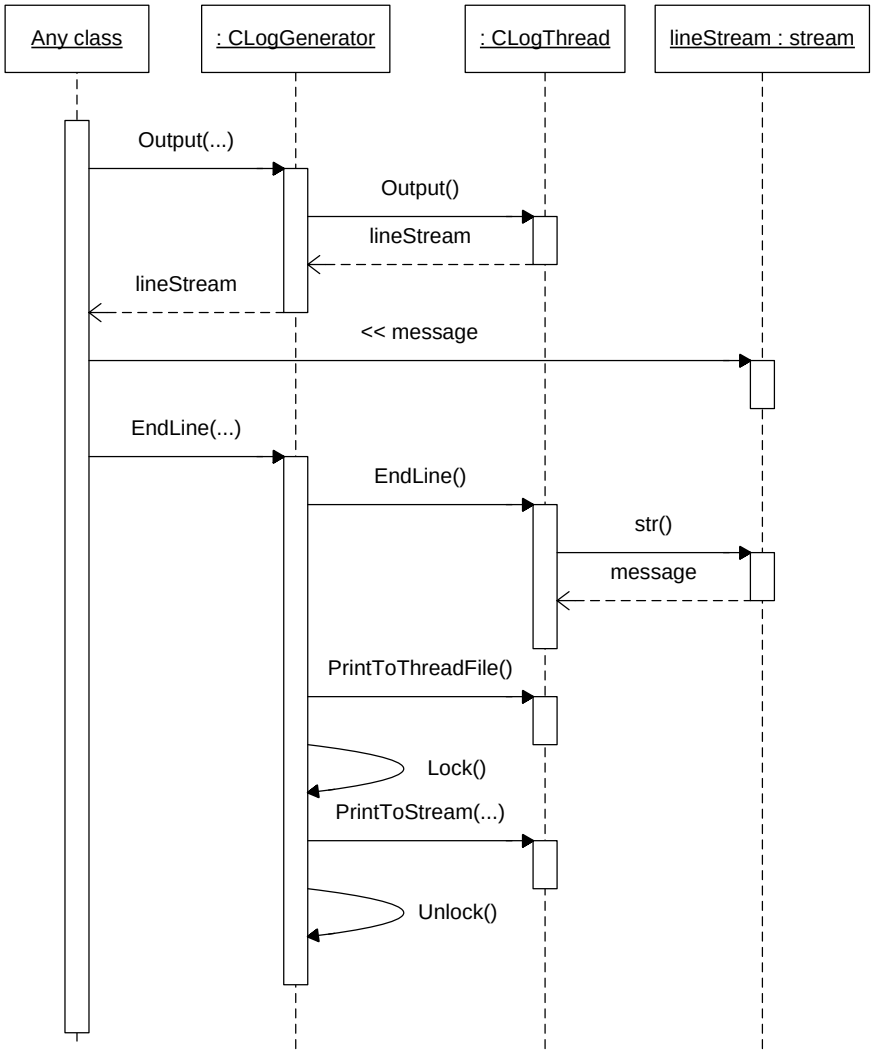
I forbindelse med fejlfinding har programmøren i mange tilfælde brug for at se værdier på forskellige numeriske variable. Disse værdier skal derfor nemt kunne udskrives i loggen sideløbende med almindelig tekst. Det er af denne grund valgt at benytte streams til overførsel af beskeder til logmodulet. For at tillade, at flere tråde benytter loggen samtidig, indeholder hver *CLogThread* sin egen stream (*lineStream*).

Sending af en logbesked forløber i tre trin: Først kaldes *Output(…)* for at få adgang til trådens logstream. Dernæst sendes logbeskeden til denne stream. Til sidst kaldes *EndLine(…)*, hvor *CLogThread* indlæser beskeden fra streamen og efterfølgende udskriver beskeden.

SIDM benytter makroer til at simplificere udskrivning af logbeskeder. Ved at benytte disse makroer, kan en logbesked udskrives på en kort og overskuelig måde. Fx:

```
L_WARN ("The value of the variable size is: " << size );
```

Ovenstående makro udskriver beskeder med niveauet WARNING. Lignende makroer er defineret for de andre niveauer.



Figur 5-11: Sekvensdiagram, der beskriver hvordan et typisk logoutput forløber.

5.3.5 Indlæsning af konfiguration

De ændringer, der er udført på designet efter valg af implementeringssprog og programpakker, har resulteret i, at der er brug for yderligere konfigurationsinformation i *CConfig* klassen. Dette har givet tre nye datastrukturer:

TCfgLog: Denne struktur indeholder konfiguration af logmodulet og bruges af *CLogGenerator*. Informationen inkluderer navne og stier på logfiler, samt hvilke logniveauer, der skal udskrives.

TCfgTextBox: Indeholder indstillinger til *CGiTextBox*.

TCfgImage: Indeholder indstillinger om hvordan billeder, skal indlæses og vises. Benyttes af *CGiImage*.

Konfigurationen til SIDM indlæses fra tekstfilen SIDM.ini. En linie i denne fil har følgende syntaks:

```
[Konfigurationsvariabel] = [første argument] [andet argument] …
```

To eksempler kunne være:

```
MarkScale    =  0.5
InputButton  =  MouseWheelUp  KeyShift  ScaleMark  Reduce
```

CConfig-klassen indlæser disse indstillinger linie for linie. Hver linie indlæses først i en stream, hvorefter nøgleord og værdier læses fra denne linie-stream. Hver type af værdi eller nøgleord indlæses i sin egen funktion. Fx indlæses en integer af funktionen *ReadInt* og en tast eller knap indlæses af *ReadButton*.

Hver gruppe af nøgleord er indlæst i sit eget *map*, der forbinder nøgleordene med tilhørende enumeration værdier. Fx er teksten ”*MouseWheelUp*” forbundet til værdien *SIDM_MOUSE_WHEEL_UP* fra *MMouseButtons* enumerationen.

En indlæsning fra en linie-stream starter med at kalde *ReadSettings*, som identificerer hvilken indstilling, der er tale om. Funktionen benytter det tilhørende map til at konverterer nøgleordet til en værdi i enumerationen *MCfgSettings*. En switch statement benytter denne værdi til at sende fokus til den rigtige kodesektion. Koden i denne sektion indlæser de forventede argumenter vha. *Read*-funktioner. Fx:

```
case SIDM_CFG_STEREO_ANAGLYPH_COLOR:
{
   MDirection dir = ReadDirection(lineStream, 2);  ← indlæs ”Left” eller ”Right”
   Float32    r   = ReadFloat(lineStream);         ← indlæs rød farvekomponent (float)
   Float32    b   = ReadFloat(lineStream);         ← indlæs blå farvekomponent (float)
   Float32    g   = ReadFloat(lineStream);         ← indlæs grøn farvekomponent (float)
   m_stereo.AnaglyphColor( dir, r, b, g );         ← gem værdierne i strukturen.
   break;
}
```

Hvis en fejl opstår under en *Read*-operation, afbrydes indlæsning af den givne linie, og indlæsningen fortsættes med den næste linie. Indlæsning af konfiguration holder sin egen log, der udskrives i en logfil. I denne fil udskrives informationer om alle indstillingslinier, der fejler ved indlæsning.

5.4 BRUGERPROGRAM

Det inkluderede interface i SIDM er ligesom resten af modulet skrevet i C++ og kan derfor benyttes af et brugerprogram, der også er skrevet i C++. For at benytte SIDM fra andre programmeringssprog vil det ofte være nødvendigt at designe et interface, der gør C++ koden forståelig for det pågældende sprog. Dette afsnit beskriver de to ekstra interfaces samt det C# demobrugerprogram, der er udviklet til SIDM.

Det inkluderede interface til SIDM består af *CSidm* klassen. Når et nyt interface skal defineres, skal det kalde metoderne i denne klasse. De problemer, der kan opstå med implementeringen af et sådant interface, vil ofte være forbundet til de typer af argumenter og returværdier, der benyttes i disse metoder. De forskellige typer, der benyttes i *CSidm*’s metoder, vil kort opsummeres efterfølgende:

Primitive typer: Der bliver benyttet fire forskellige primitive typer i interfacet:

Boolean, 32-bit integer og 32 og 64-bit *floating point* værdier. Disse typer vil almindeligvis være simple at konvertere til andre interfaces.

Enumerationer: To enumerationer i SIDM bliver også benyttet i interfacet: *MButtonState* og *MDirection*. Sidstnævnte definerer 16 forskellige værdier, hvoraf kun to benyttes i interfacet: *SIDM_LEFT* og *SIDM_RIGHT*.

Strukturer: Interfacet benytter seks strukturer defineret i SIDM: *TCoordinates*, *TCameraData*, *TImageData*, *TModelData*, *TImageSearch*, *TModelSearch*.

Klasser: To klasser fra C++ standard bibliotek benyttes i interfacet til SIDM: *list<Int32>* definerer et array, der kan holde et variabelt antal 32-bit integeværdier, og *string* gemmer en tekststreng af variabel længde. Sidstnævnte findes kun som variabel i de benyttede strukturer.

Funktions pointers: Noget af det mest besværlige ved definitionen af et nyt interface til SIDM vil være at implementere funktion pointers. Dvs. sende en værdi til SIDM, som modulet kan benytte til at kalde en funktion i brugerprogrammet.

5.4.1 Interface til Delphi

Det funktionsbaserede interface til SIDM er rimeligt simpelt opbygget: *SidmInitialize*() genererer en instans af *CSidm*, gemmer en pointer til denne i en global variabel, og initialiserer SIDM modulet. *SidmTerminate*() afslutter modulet og sletter den globale variabel. Alle andre funktioner kalder blot den tilsvarende funktion i *CSidm* instansen, der er bundet til den globale variabel.

Brugeren af dette interface må selv sørge for at starte med at kalde *SidmInitialize* og afslutte med at kalde *SidmTerminate*. Hvis *SidmInitialize* kaldes mens *CSidm* allerede er initialiseret, eller hvis en af de andre funktioner kaldes mens *CSidm* ikke er initialiseret, vil den pågældende funktion afslutte uden at udføre nogen handling.

Formålet med det funktionsbaserede interface er, at det skal bygges på ren C-kode. I mange programmeringssprog er det mere simpelt at implementere C-kode frem for C++ kode. De to C++ klasser, *list* og *string*, må derfor erstattes af tilsvarende C variable: En *list<int>* kan gemmes i et integerarray, forudsat dette array er stort nok. Arrayet initialiseres af det kaldende program, og en integervariabel, der definerer størrelsen af dette array, føjes til funktionens argumentliste. Kun elementer op til det antal, der specificeres i denne variabel, vil blive overført fra *list* objektet til arrayet. Variable af typen *string* konverteres på tilsvarende måde til et char-array. Da de fleste strukturer i interfacet, benytter *string*, må disse strukturer overføres til tilsvarende strukturer, der kun benytter C-kode.

Det funktionsbaserede interface er endnu ikke fuldt implementeret, og endnu kan kun omkring halvdelen af funktionerne benyttes fra Delphi. Implementeringen af dette interface blev påbegyndt meget sent i projektforløbet og blev yderligere forsinket af udviklerens manglende kendskab til Delphi og mangel på en Delphi-kompiler. De fleste kommunikationsproblemer med Delphi er blevet løst. På nuværende tidspunkt mangler der dog stadig kendskab til, fx hvordan funktionspointere skal implementeres.

5.4.2 Interface til .NET

Der er i forbindelse med dette projekt opbygget et interface til SIDM, der tillader .NET applikationer at kalde modulet. Implementering af dette interface viste sig at være en af de mere komplekse opgaver i projektet. Udviklingen krævede en forståelse af hvordan .NET applikationer bliver eksekveret i forhold til almindelige C++ applikationer, og mange af de nødvendige informationer var svære at finde frem til. Dette afsnit forsøger at give et indtryk af, hvordan det udviklede interface fungerer. Problemstillingen vil kun blive gennemgået meget generelt og er på ingen måde fyldestgørende.

Når almindelig C++ kildekode bliver kompileret, oversættes den til såkaldt *native code*. Native code kan eksekveres af det operativsystem, på hvilket det er udviklet, men kan ikke aktiveres på andre platforme.

Kildekode til et .NET program vil normalt kompileres til såkaldt managed code: Kompileren oversætter kildekoden til *Microsoft Intermediate Language* (MSIL), der er et sæt platformuafhængige instruktioner. For at eksekvere managed code kræves et underliggende software-lag, som kaldes *Common Language Runtime* (CLR). Under eksekvering af programmet vil CLR konvertere MSIL til native code, der kan eksekveres på det angivne system. Kildekode, der ikke kompileres til managed code og derfor ikke køres af CLR, kaldes for *unmanaged code*.

Visual C++ .NET tilbyder mulighed for at sammenblande managed og unmanaged code i den samme eksekverbare fil, der efterfølgende siges at være i *mixed-mode*. Når denne fil eksekveres, vil managed code blive kørt via CLR, mens unmanaged code køres direkte af systemet. Kommunikationen mellem managed og unmanaged code kan udføres på flere forskellige måder. En af metoderne er at benytte en *managed wrapper*.

For at tillade .NET applikationer at benytte SIDM, er der udviklet en managed wrapper til SIDM. En managed wrapper er et interface, der kalder unmanaged code, men som selv er i managed code. Den udviklede wrapper til SIDM implementeres i klassen *WSidm*. Denne klasses constructor genererer en instans af *CSidm*, mens *WSidm*'s destructor sletter denne instans. Funktionerne i *WSidm* kalder de tilsvarende funktioner i denne instans af *CSidm*.

SIDM benytter sig flere steder af statiske variable. For at initialiserer disse variable, kræves at modulet har et *entry point*, dvs. at modulet får eksekveringsfokus ved programopstart. En fil i mixed-mode vil ikke automatisk levere dette entry point, og programmøren er derfor nødt til eksplicit at definere dette.

Den første handling i *WSidm*'s constructor er at kalde funktionen:

```
__crt_dll_initialize()
```

Tilsvarende er den sidste handling i *WSidm*'s destructor at kalde:

```
__crt_dll_terminate()
```

Kort fortalt sørger disse funktioner for at levere et entry point til C runtime og giver dermed C runtime mulighed for at initialisere de statiske variable i SIDM.

For hver funktion, der findes i *CSidm*, er der defineret en tilsvarende funktion i *WSidm*, hvor argumenterne er konverteret til managed code. Konvertering af disse argumenter er i flere tilfælde rimelig omfattende. Fx er alle benyttede strukturer og enumerationer blevet gendefineret i managed code. Strukturerne er konverteret til tilsvarende klasser, der indeholder funktionaliteter til at konvertere til og fra den originale struktur.

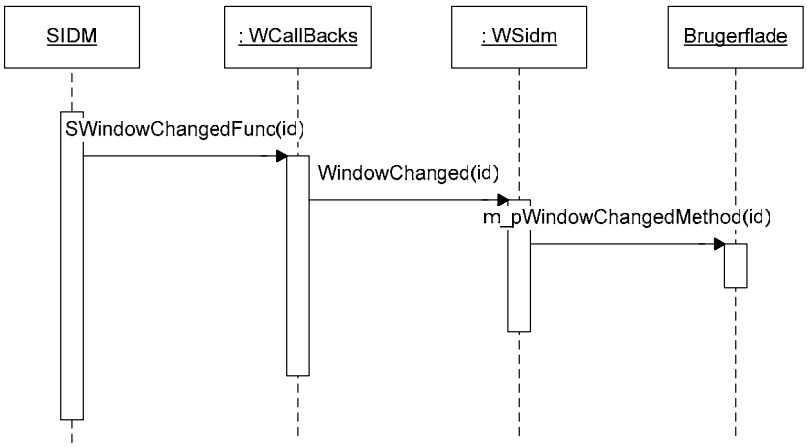
Hvor konvertering fra unmanaged data til managed data i flere tilfælde er rimelig simpel, er det ofte mere besværligt at lokalisere metoder til at konvertere den anden vej. Fx skal en managed String (*myString*) konverteres til et char array på følgende måde:

```
static_cast<const char*> (System::Runtime::InteropServices::Marshal …
::StringToHGlobalAnsi(myString).ToPointer());
```

Funktionskaldet er simpelt nok, men det er svært at finde frem til den rigtige statiske funktion, når den ligger gemt under fire niveauer af namespaces.

En anden funktionalitet, der var besværligt at implementere i dette interface, var callback-funktioner. Dvs. de tre funktioner, som SIDM skal kunne kalde i brugerprogrammet. Disse funktionskald skal i interfacet forløbe over flere trin.

SIDM kan kun kalde statiske eller globale funktioner, der er i unmanaged code. Da *WSidm* er i managed code, må en ekstra klasse *WCallBacks* implementeres i unmanaged code for at muliggøre callbacks. Denne klasse indeholder én statisk funktion per callback funktion i SIDM. Når brugerprogrammet definerer en funktion, der skal kaldes af SIDM, vil en delegate til denne funktion gemmes i *WSidm*, mens en pointer til den tilsvarende statiske funktion i *WCallBacks* sendes til SIDM. En delegate er en form for avanceret funktionspointer, der benyttes af .NET. Når et callback skal udføres i SIDM, kalder modulet en af de statiske funktioner i *WCallBacks*, der igen kalder den tilsvarende managed metode i *WSidm*. Denne metode kalder den ønskede metode i brugerprogrammet, via den gemte delegate (se Figur 5-12).



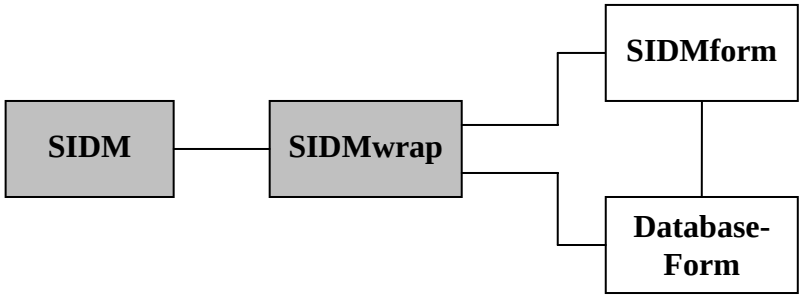
Figur 5-12: Funktionskald fra SIDM til brugerprogrammet. I eksemplet er benyttet *WindowChanged*, der kaldes hver gang et nyt vindue kommer i fokus.

5.4.3 Demobrugerprogrammet

Det medfølgende demobrugerprogram til SIDM er skrevet i C# .NET og benytter .NET interfacet til at kommunikere med SIDM. Som det kan ses på Figur 5-13, indeholder demobrugerprogrammet to klasser: *SIDMform* og *DatabaseForm*. Hver af disse klasser genererer og administrerer deres egen dialogboks:

SIDMform administrerer den primære dialogboks, som giver mulighed for at teste alle funktioner i SIDM, der kommunikerer med kontrol- eller vinduesmodulet.

DatabaseForm administrerer en ekstra dialogboks, der kan åbnes den primære dialogboks. Funktioner, der benyttes til at kommunikation med databasen, kan testes i denne ekstra dialogboks.



Figur 5-13: Klassediagram for C# demobrugerprogram til SIDM. *SIDM* og *SIDMwrap* er ikke en del af demobrugerprogrammet men medtages her for at illustrere sammenhængen mellem programbibliotekerne.

Designet af demobrugerprogrammet er baseret på små kompakte dialogbokse, der tillader, at man kan teste alle funktionaliteter i SIDM, uden at dialogboksene optager for meget plads på skærmen. Dette har resulteret i dialogbokse, der kan virke en smule overfyldte og komplicerede ved første øjekast. Kontrolenhederne er dog grupperet, så de efter kort tid bør være til at finde rundt i. Koden i demobrugerprogrammet er simpelt bygget op og burde være nem at overskue. Denne rapport vil derfor ikke kommentere denne kode yderligere.

5.5 KILDEKODESTRUKTUR

Vedligeholdelse og udvidelse af software er en vigtig del af mange programmørers arbejde. Ofte vil det ikke være den programmør, der udviklede produktet, som senere skal arbejde videre med det. Selv hvis en programmør skal videreudvikle et af sine egne programmer, kan det være en større proces igen at gøre sig bekendt med koden. At skabe sig et overblik over ukendt kode kompliceres alt for ofte af en rodet kildekode, hvor den originale programmør ikke har benyttet klare og konsistente retningslinier.

Dette hovedafsnit beskriver de retningslinier, der er benyttet til navngivning og kodestruktur i SIDM. Det foreslås, at disse retningslinier forsat benyttes i en videre udvikling af modulet.

5.5.1 Navngivning

Navngivningen af variable, metoder og klasser i SIDM er baseret på bl.a. *Hungarian Notation* samt de navngivningsregler, der benyttes i *Microsoft Foundation Classes*. Hungarian Notation er ikke fuldt ud implementeret i koden, da udvikleren føler, at dette reducerer overskueligheden af koden frem for at øge den. Nedenstående er en klargøring af de navngivningsregler, der er benyttet i SIDM. Medmindre andet er angivet, skal hvert ord starte med stort bogstav og de resterende bogstaver være små:

Variable: Det første bogstav i variabelnavnet er altid lille. Member-variable starter med *m_*, statiske variable starter med *s* og pointers starter med *p*. For variable, der dække flere af disse betegnelser vil flere bogstaver benyttes i nævnte rækkefølge. Eks: *m_config* (en member-variabel), *m_spArray* (en member-variabel, der er en statisk pointer), *pPos* (en midlertidig pointer), og *size* (en midlertidig variabel).

Metoder: Statiske metoder starter med *S*. Metodenavnet skal være beskrivende. Almindeligvis bør access-funktioner ikke starte med *Set* eller *Get*, da dette blot forstørrer koden. Eks: *ExtractTexture*, *SInstance* (statisk funktion).

Klasser: Klassenavne starter generelt med C. Klasser, der benyttes til exceptions, starter i stedet med *E*. Database-tabelklasser starter med *CDb*, syncronized call klasser starter med *CSc,* og *graphic item* klasser starter med *CGi*. Eks: *CSidmApplication*, *CScOpenWindow*, *CGiImage, EInvalidArgument*.

Strukturer: Starter med T. *m_* udelades fra navnet på public variable i en struktur. Eks: *TImageData*.

Enumerationer: Enumerationer starter med M. Eks: *MDirection*.

Typer: Typedefinitioner slutter på *Type*, dog ikke primitive typer som *int* og *float*. Eks: *CoordChangedType*, *Int32*.

Konstanter: Makroer og andre definitions samt navne på konstanter i enumerationer skrives med udelukkende store bogstaver. Alle navne starter med *SIDM_* og ord deles af tegnet '_'. Konstanter i samme enumeration starter derudover generelt med det samme ord. Eks: SIDM_LEFT, SIDM_TUPLE_NEW, SIDM_TUPLE_UNCHANGED.

5.5.2 Struktur

Foruden regler for navngivning, følger SIDM nogle retningslinier for kodens generelle struktur.

Filopdeling: Header filer indeholder normalt definitionerne for en enkelt klasse. Enkelte headerfiler vil dog indeholde definitioner for flere typer, enumerationer, strukturer eller små, ensformige klasser. Source filer indeholder deklarationer til metoderne, der er defineret i den tilsvarende header fil. Funktioner er inddelt på samme måde og ligger i samme rækkefølge i header fil såvel som i source filen. Inline-funktioner deklareres i header filen i stedet for i source filen.

Includes: Inkludering af andre header filer i en given header fil begrænses så meget som muligt. Dette gør, at en ændring i en enkelt header fil ikke resulterer

i, at alle klasser skal genkompileres. Ingen header filer fra eksterne API'er må inkluderes i en header fil. Dette sikrer at brugerprogrammet ikke behøver at have adgang til header filer fra de eksterne API'er. Includes opdeles i tre grupper: *System includes* henviser til C++ standardbiblioteket, *extern includes* henviser til eksterne API'er, og *local includes* indeholder header-filer det fra samme projekt (dvs. SIDM).

Klassestruktur: Alle member variable i en klasse er private, mens member variable i strukturer er public. Funktioner er generelt public eller protected. Foruden disse tilgangsspecifikationer, opdeles funktioner i tre grupper: *Lifecycle* indeholder constructors, destructors, initializers og lignende funktioner. *Access and Inquiry* indeholder funktioner der direkte tilgår member variable eller giver information om disse. *Operations* inkluderer de resterende funktioner i klassen.

Andre retningslinier: Ud over det ovenstående, følge strukturen i SIDM følgende regler:

- Der må ikke benyttes nogle "*Using namespace*" i header filer.
- Metoder, der ikke påvirker member variable i klassen, skal erklæres const. Det samme skal argumenter og returværdier, der ikke må modificeres.
- Brug af inline-funktioner skal begrænses til de steder, hvor det kan have mærkbar indflydelse på programmets ydelse.

5.5.3 Dokumentation og kommentarer

Kildekoden til SIDM er skrevet med engelske dokumenterer og engelsk navngivning af variable, typer osv.

Alle deklarationer eller definitioner i SIDM indledes med et dokumenterende felt, der beskriver det givne element. Dette felt benytter direktiver og er opbygget på en sådan måde, at det kan indlæses i programmet *CcDoc* [R38]. Dette program kan udtrække dokumentationen fra kodefilerne og opbyge HTML dokumenter, der indeholder denne dokumentation. Foruden denne indledende forklaring, indeholder funktionerne i SIDM løbende kommentarer om alle interessante eller ikke åbenlyse operationer.

Linieskift og indrykning følger de samme retningslinier gennem hele koden, med fokus på overskuelighed af koden. Alle linier i både kildetekst og dokumentation er begrænset til at være højest 78 tegn bredde. Dette sikrer at kildekoden nemt kan printes ud, og stadig bibeholde strukturen. Figur 5-14 viser et eksempel på strukturen for en funktionsdeklaration.

```
//
//============================================================================
//@{
// Call back method called when the mouse is moved inside the window
// linked to this instance.
// Calls CursorMoved in the CGiMark.
//
// @param x     Horizontal position of the cursor in window
//              coordinates.
// @param y     Vertical position of the cursor in window coordinates.
// @param state Whether a mouse button is held down or not.
//@}
//
//============================================================================
void CScene::
MouseMove(Int32 x,
                     Int32 y,
          MButtonState state)
{
    ...
}
```

Figur 5-14: Eksempel på strukturering af kildekode i SIDM. Funktion-headers skifter til ny linie inden funktionsnavnet, så dette er nemt at finde. Hvis funktionen modtager flere argumenter, stilles hvert ekstra argument på sin egen linie.

5.6 OPSUMMERING

Dette kapitel gennemgår både det implementeringsspecifikke design og selve implementeringen. Beskrivelsen gennemgår ikke alle områder af designet til bunds, men fokuserer i stedet på begrænsede, interessante områder.

Fokus er lagt på beskrivelsen af, hvordan billeder bliver indlæst og vist på skærmen, men der er også lagt vægt på beskrivelsen af bl.a. fejlbehandling, konfigurationsindlæsning, og beskrivelse af interfacet til demobrugerprogrammet.

Kapitlet afsluttes med en beskrivelse af kildekodestrukturen i programmet, inklusiv den benyttede navngivningsmetode og hvordan koden er dokumenteret.

6 Test

Dette kapitel beskriver de test, der er udført på SIDM i løbet af projektet, samt resultaterne af disse test. Først i kapitlet beskrives den benyttede fremgangsmetode til at teste SIDM (afsnit 6.1). Efterfølgende gennemgås interessante fejl, som disse test har identificeret (afsnit 6.2). I slutningen af kapitlet beskrives den afsluttende test af SIDM (afsnit 6.3).

6.1 STRUKTURERING AF TEST

For at fange fejl så tidligt som muligt er SIDM løbende blevet testet gennem hele udviklingsforløbet. De test, der er blevet udført på SIDM, kan opdeles i fire hovedtyper:

Funktionel test: En funktionel test er en black-box-test, hvor programmets funktionaliteter testes fra brugersiden. Testen indbefatter en undersøgelse af hvorvidt funktionaliteterne i SIDM lever op til de funktionelle krav, der er stillet i kravspecifikationen. Desuden sikres, at ingen fejl opstår ved benyttelse af disse funktionaliteter.

Ikke-funktionel test: Den ikke-funktionelle test undersøger om SIDM overholder de ikke-funktionelle krav stillet i kravspecifikationen.

Stresstest: Begrebet stresstest benyttes i denne rapport om test, hvor man forsøger at fremprovokere fejl i programmet, uden at benytte en fastlagt fremgangsmetode. Stresstesten er et godt supplement til de øvrige test, da den kan finde fejl, der kun opstår ved længere tids arbejde med programmet. Dette vil ofte dreje sig om fejl i forbindelse med hukommelsesforbrug eller kommunikation mellem tråde.

Modulær test: Den modulære test kan tolkes som en funktionel test af et enkelt modul i SIDM. I denne test vil interfacet til et givent modul gennemtestes, ved at samtlige public funktioner i dette interface kaldes, med forskellige typer af argumenter.

Det er valgt ikke at udføre en fuld strukturel test på SIDM, dvs. en test hvor samtlige mulige grene i programmet gennemkøres. Årsagen til dette valg er, at SIDM stadig er på udviklingsstadiet. Programmodulet ændres i øjeblikket så hurtigt, at en strukturel test fra en tidligere version hurtigt mister sin værdi. Det er selvfølgelig en fordel for en programmør præcis at kunne se, hvordan SIDM opførte sig i tidligere versioner. Når denne fordel lægges op mod, hvor stor en arbejdsbyrde det er at udføre en sådan test, anses fordelen for at være for lille.

Hver iterationsfase fra og med IT7 ’SIDM v0.1 – Vinduesbehandling’ (se appendiks B.1) er blevet afsluttet med en modulær test, en stresstest og en begrænset funktionel test af SIDM. I den funktionelle test blev alle fuldt ud implementerede funktionaliteter testet. Resultaterne af disse test har haft indflydelse på den næste iteration, og enkelte af de fundne fejl beskrives i afsnit 6.2.

Den afsluttende test af SIDM har været mere gennemarbejdet og har inkluderet alle fire typer af test. Den afsluttende testfase beskrives nærmere i afsnit 6.3.

6.2 FEJLBESKRIVELSER

Under implementeringen af et program som SIDM vil man hele tiden støde på nye fejl, dvs. funktionaliteter i programmet, der ikke opfører sig som de burde. Langt hovedparten af disse fejl vil blive lokaliseret og rettet med det samme, men fra tid til anden vil en fejl undvige lokaliseringen. Dette er generelt tilfælde med fejl, der ikke kan genskabes, eller fejl, der opstår på, hvad der kan virke som tilfældig tidspunkter under eksekvering af programmet. Under implementeringen af SIDM blev der stødt på flere af disse fejl, hvor langt hovedparten siden er blevet lokaliseret og elimineret. I dette afsnit beskrives to af disse fejl. Den ene fejl er medtaget som ren kuriositet, mens den anden er medtaget, fordi den endnu ikke er blevet løst.

6.2.1 Uheldigt valgt funktionsnavn

LoadImage metoden i *CGiImage* klassen bliver kaldt, når et nyt billede skal indlæses fra en fil. Da funktionen skal kunne kaldes fra brugerprogrammet, benyttes dette funktionsnavn i flere forskellige klasser, der leder fra brugerprogrammet til *CGiImage* klassen. Dette funktionsnavn viste sig i løbet af implementeringen ikke at være et godt valg, da Visual Studio fra tid til anden beklagede sig over, at den ikke kunne finde funktionen *LoadImageA*. Fejlen opstod kun i visse solutions, mens andre solutions kørte fint.

Det viste sig, at fejlen skyldtes headerfilen *WinUser.h*. Denne fil definerer makroen:

```
#define LoadImage  LoadImageA
```

Denne makro fortæller kompileren, at den skal ændre alle *LoadImage* navne i koden til *LoadImageA*. *WinUser.h* bliver aldrig inkluderet direkte af udvikleren, men i stedet som sekundær inkludering i andre Windows header-filer. Da ikke alle filer indirekte inkluderer denne headerfil, vil *LoadImage* kun ændre navn i nogle af filerne, hvorved problemet opstår.

Problemet er siden blevet løst ved at omdøbe alle berørte funktioner til *LoadImageA*. Det er dog stadig uopklaret hvorfor Microsoft tillader en så destruktiv makro at blive defineret i en almindeligt brugt headerfil, uden at informere om, at *LoadImage* er et reserveret ord.

6.2.2 Tekststrenge og databaser

Den nuværende version af SIDM indeholder desværre en fejl, som endnu ikke er blevet sikkert lokaliseret. Fejlen antages at have spøgt i SIDM fra en tidlig version, men pga. af fejlens undvigende karakter, kan det ikke garanteres, at der hele tiden har været tale om den samme fejl. Dette afsnit beskriver tidsforløbet for denne fejl, fra første gang den formodentligt opstod, til den nuværende version af SIDM.

I dette afsnit benyttes betegnelserne debug version og release version af SIDM. I Visual Studio vil en debug version indeholde debug informationer, udføre flere test på data, og derfor køre langsommere. En debugversion kan kun køres på systemer, der har Visual Studio installeret. En release version i Visual Studio er

optimeret så den kører hurtigt og effektivt, og den kan kører på all Windows-maskiner.

Fejlen formodes første gang at have opstået, da det færdige databasemodul blev sammensat med resten af SIDM under iterationen IT10 (se appendiks B.1). På dette tidspunkt blev API'en PUI benyttet til visning af tekst i grafikvinduerne, og indtil da havde API'en fungeret fejlfrit (se afsnit 4.4.5). Efter inkluderingen af databasemodulet begyndte PUI fra tid til anden at give fejl under udskrivning af tekst. En nærmere undersøgelse viste, at fejlene kun opstod hvis database-API'en LibODBC blev kald, og det selvom de to API'er intet havde med hinanden at gøre. En udskiftning af PUI til den lignende API GLUI hjalp ikke på problemet. På dette tidspunkt var der ingen direkte alternativer til LibODBC, så det blev i stedet valgt at opgive at benytte PUI. Håbet var at LibODBC ikke havde egentlige fejl, men at problemet 'blot' skyldes inkompatibilitet mellem de to moduler. Det viste sig senere at dette håb var for optimistisk.

Efter demobrugerprogrammet blev implementeret i iteration IT11 kunne stresstest fra tid til anden resultere i fejl i interfacet mellem SIDM og demobrugerprogrammet. Fejlen opstod aldrig i debug versioner af SIDM, kun i release versioner, og kun når variable af typen string skulle overføres. Klassen string er en del af C++ standardbibliotek. Fejlen kunne aldrig genskabes og derfor ikke præcis lokaliseres.

I den sidste implementeringsrelaterede iteration af SIDM (IT12) blev konfigurationsmodulet udvidet til at kunne indlæse konfigurationsinformation fra en fil. Under implementeringen af denne funktionalitet begyndte debuggeren i Visual Studio at opføre sig underligt. Når man via debuggeren aflæste værdier på string variable, fik man forkerte og ofte meget ulogiske værdier oplyst. I andre tilfælde mistede string-objekter deres værdier, efter de blev returneret fra en funktion. Dette problem opstod ligegyldigt om databasemodulet blev initialiseret eller ej, så det havde umiddelbart intet at gøre med LibODBC. Desværre blev det ikke undersøgt om problemet forsvandt ved helt at fjerne LibODBC fra systemet, dvs. undlade at indlæse dertilhørende headerfiler og dll-filer. Efter en udførlig gennemgang af opsætningen i samtlige API'er og SIDM, blev der observeret uoverensstemmelser mellem bl.a. benyttede standardprogrambiblioteker. Ved at rette disse uoverensstemmelser forsvandt fejlen ved konfigurationsindlæsning.

I den nuværende version af SIDM træder fejlen frem i forbindelse med databasedialogboksen i demobrugerprogrammet. Hvis man trykker [*Refresh*] gentagende gange for kameratabellen, vil programmet før eller siden lukke ned med en fejlbesked eller alternativt låse helt, så demobrugerprogrammet ikke længere modtager input. Fejlen kan også opstå ved andre handlinger i databasevinduet. Fejlen er igen observeret til at være string-relateret, men det er ikke muligt at lokalisere, præcis hvad der forårsager fejlen. Fejlen opstår kun i release-versionen af SIDM, og det har endnu ikke været muligt at fremprovokere en lignende fejl i debug-versionen.

Hvis alle informationer omkring fejlen og lokalisering af denne skulle medtages her, ville beskrivelsen fylde adskillige sider. Der er derfor udeladt en masse detaljer i den ovenstående beskrivelse.

Alt tyder på at fejlen altid opstår i forbindelse med strings. De fleste gange op-

står fejlen direkte som følge af kommunikation med *LibODBC*, og i alle tilfælde er biblioteket inkluderet i SIDM når fejlen opstår. Udvikleren antager kraftigt, at fejlen følger af et problem med *LibODBC*, formentlig et kompatibilitetsproblem med den benyttede version af Visual Studio. Den eneste umiddelbare løsning er at finde et alternativt til *LibODBC*, evt. at kalde *ODBC* direkte fra SIDM uden brug af mellemliggende API.

6.3 AFSLUTTENDE TEST

Den endelige version af SIDM har gennemgået både en modulær test, en funktionel test, en ikke-funktionel test, og en stresstest.

6.3.1 Modulær test

I den modulære test er samtlige funktionaliteter i logmodulet og i databasemodulet blevet testet. I konfigurationsmodulet er indlæsningsfunktionaliteten testet ved brug af forskellige linier i konfigurationsfilen. Arbejdsmodulet er testet ved at sikre, at det kun kan indlæse billeder, der eksisterer i den korrekte tilstand. De resterende tre moduler er ikke blevet testet for sig selv i en modulær test. Interfacemodulet er ikke blevet testet alene, fordi det ikke udfører andre handlinger end at videresende funktionskald, og vindues- og konfigurationsmodulerne er ikke testet alene fordi disse er for tæt forbundne til hinanden. Der blev ikke observeret nogen fejl i nogle af disse test. De modulære test er valgt ikke at blive formaliseret i denne rapport. Dette skyldes dels mangel på fejl i disse test og dels at den funktionelle test dækker stort set de samme områder.

6.3.2 Funktionel test

Den funktionelle test af SIDM er baseret på de tegnede use cases. Hvis testen viser at SIDM opfylder alle use cases, betyder dette, at programmets funktionaliteter kører som de skal. Begrundelsen er, at hele den funktionelle del af kravspecifikationen er implementeret i use cases. Testen mht. disse use cases kan ses i detaljer i appendiks F. Resultatet af denne test er, at alle implementerede funktionaliteter i SIDM virker som forventet.

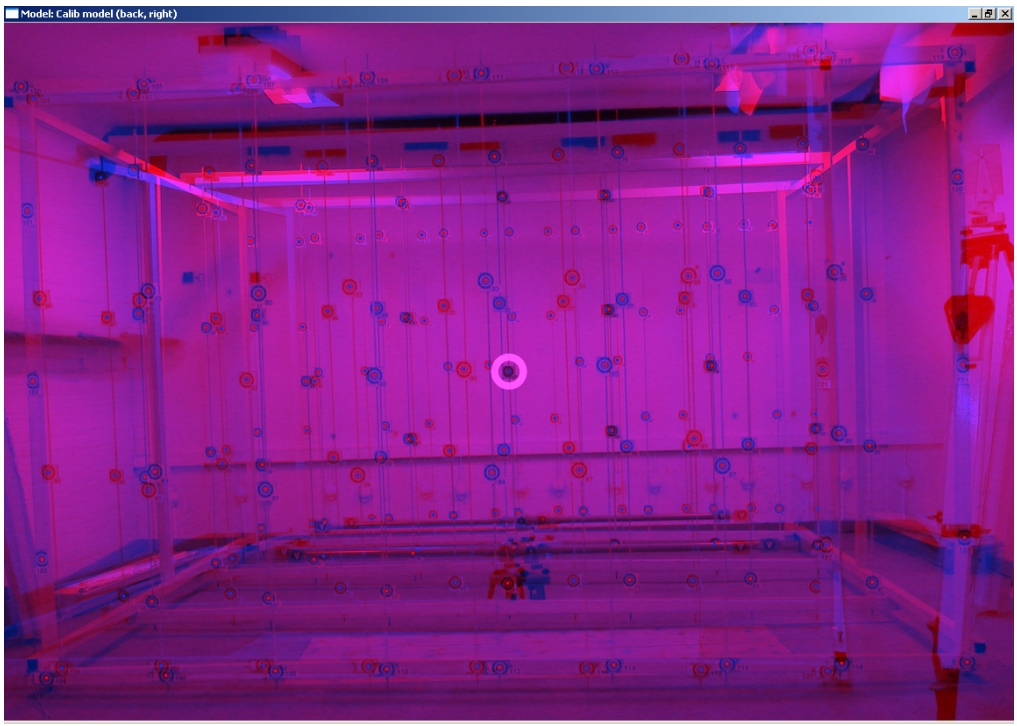
Fejlinformation fra SIDM skrives i øjeblikket til log filer. Selvom det ikke er specificeret i kravspecifikationen, bør brugerprogrammet kunne hente disse fejlinformationer fra SIDM via funktionskald. Denne funktionalitet er endnu ikke implementeret.

I forbindelse med den funktionelle og de modulære test har der været flere informationer, der ikke kunne læses ved udelukkende at se på output på skærmen og returværdier fra funktionskald. Et eksempel på dette er sikring af, at konverteringen af objektkoordinater til billedkoordinater forløber korrekt. Nedenstående beskrives kort hvordan dette blev testet.

Et testfelt er en konstruktion, der bliver benyttet til kalibrering af kameraer. Det består af et stort antal målemærker fordelt i rummet, hvor den eksakte rumlige position af hvert målemærke er kendt. I forbindelse med dette projekt er der taget en serie af billeder af et sådant testfelt fra forskellige positioner. Orienteringsparametre for disse billeder og for det benyttede kamera er fastsat i et eks-

ternt program og indskrevet i SIDM's database.

På Figur 6-1 ses to af disse billeder i anaglyph farver. Ud fra de kendte positioner af målemærkerne (i objektkoordinater), er der beregnet de tilsvarende billedkoordinater. Disse beregnede punkter plottes ind på billedet i form af prikker - røde prikker i det røde billede og blå prikker i det blå. Som det ses på figuren ligger næsten alle prikker i det præcise centrum af det tilhørende målemærke. De få synlige afvigelser skyldes afrunding i OpenGL, hvor prikker forskubbes til den nærmeste pixelposition i framebufferen.



Figur 6-1: Stereomodel over kalibreringsfelt. De røde og blå prikker markerer den beregnede position af hvert målemærke. Grundet manglende normalisering af billederne samt for stor afstand mellem kamerapositionerne kan modellen ikke ses korrekt i 3D.

6.3.3 Stresstest

Selv om alle funktionaliteter umiddelbart ser ud til at virke i den modulære og funktionelle test, opstår der, som beskrevet i afsnit 6.2.2, problemer ved stresstest af systemet. Primært ved brug af databasedialogboksen i demobrugerprogrammet vil der hurtigt opstå fejl efter gentagende operationer. Stresstesten resulterede også i fejl ved udelukkende brug af den primære dialogboks i demobrugerprogrammet. Det var dog langt mere besværligt at fremprovokere fejl ved denne metode, og ofte opstod der først fejl efter mange minutters stresstest. Det er ikke muligt at finde et system i disse fejl, så fejlene kan genskabes. Ovenstående beskriver stresstest af release versionen. Det lykkedes aldrig at skabe fejl i stresstest af debug versionen. Den forventede årsag til denne fejl, samt hvad der skal gøres for at fjerne fejlen, er beskrevet i afsnit 6.2.2.

6.3.4 Ikke-funktionel test

Den ikke-funktionelle test undersøger om programmodulet lever op til de ikke-funktionelle krav i kravspecifikationen. De enkelte krav vil blive behandlet individuelt i samme rækkefølge som de fremstår i kravspecifikationen i afsnit 2.3.7.

Ydeevne: Ydeevnen af SIDM er testet på en *Barton 2500+* processor med 1024 Mb ram og et *GeForce Ti4400* grafikkort med 128 Mb ram. Denne computer må tolkes som værende langsommere end den 'moderne computer', der defineres i kravspecifikationen. Under arbejde med to 8-megapixel billeder, viste SIDM en opdateringsfrekvens på 100 gange i sekundet, hvilket svarer til skærmens opdateringsfrekvens på 100 Hz. SIDM optegner kun billedet, når der er brug for det, så denne test blev udført ved konstant at bevæge det vandrende mærke rundt på skærmen. Kravet i kravspecifikationen var en opdateringsfrekvens på 50Hz, så dette krav må siges at være opfyldt.

Det er også et krav, at de fleste funktionskald fra brugerprogrammet forløber uden forsinkelser. Da ingen af de implementerede funktionaliteter, der leveres af interfacet, er tidskritiske, blev dette testet fra en brugers synspunkt. Dvs. at de forskellige funktionaliteter i demobrugerprogrammet blev afprøvet, mens brugeren holdte øje med forsinkelser. Ingen af funktionaliteterne viste synlige forsinkelser, hverken fra brugerprogrammet til SIDM eller fra SIDM til brugerprogrammet. Da disse kald bliver yderligere sløvet ned af demobrugerprogrammet og det mellemliggende interface, må SIDM siges at opfylde det undersøgte krav. Det var heller ikke muligt at observere forsinkelser, fra en funktion til indlæsning af billeder blev kaldt, til vinduet viste at indlæsningen var påbegyndt. Der går dog noget tid før billedet er fuldt indlæst og vises i vinduet.

Det sidste stillede krav til programmodulets ydeevne er, at SIDM ikke må bruge processorressourcer ved stilstand. Så længe et grafikvindue er åbent er dette krav desværre ikke opfyldt. Årsagen er, at så længe *OpenGLUT* er startet, vil inputtråden administreres af denne API. Hver gang *OpenGLUT* har et ledigt øjeblik, vil API'en kalde SIDM, så programmodulet kan undersøge, om brugerprogrammet har sendt nye funktionskald, der skal behandles. Ved stilstand forløber denne undersøgelse uafbrudt, og den forbruger derfor processorressourcer. Det er desværre ikke muligt at undgå denne situation. Hvis SIDM vælger at pause tråden, når modulet får fokus, vil *OpenGLUT* ikke længere registrere brugerinput. Modsat, hvis *OpenGLUT* ikke kalder SIDM løbende, vil modulet kun have mulighed for at eksekvere funktionskald, når et brugerinput bliver registreret af *OpenGLUT*. Den eneste løsning på problemet er, at finde en alternativt vinduesstyrings API. Problemet er dog ikke så stort, da tråden har lav prioritet i Windows. Selvom *Windows Jobliste* (*Windows Task Manager*) fortæller at SIDM forbruger 99% at processorressourcerne, kunne det ved en brugstest ikke mærkes på ydeevnen af andre programmer, der blev kørt samtidigt.

Hukommelsesforbrug: Der er i implementeringen af SIDM lagt vægt på, at alt allokeret hukommelse bliver dealokeret efter brug, så der ikke opstår memory leaks. Samtidig er det forsøgt at slette alle større datamængder så snart de ikke skal bruges mere. I *Windows Jobliste* kan man observere hukommelsesforbruget af den proces, der kører SIDM. Hukommelsesforbruget fluktuerer meget pga.

forbrug i OpenGL (og C# hvis demobrugerprogrammet benyttes). Det er dog stadig muligt at fastsætte at hukommelsesforbruget ikke stiger efter at have arbejdet i længere tid med SIDM. Dette må betyde, at SIDM ikke har nogle memory leaks.

Platform: SIDM opfylder kravet om, at kunne køre på et moderne Windowssystem. Teoretisk set kan programmet også kompileres på Linux eller Unix maskiner, men dette er aldrig blevet testet.

Programdesign: SIDM er som krævet opdelt i moduler, der hver især har et begrænset interface i form af en interfaceklasse (se afsnit 3.4.2). Pga. den modulære opbygning og brugen af flere tråde i SIDM er implementering af nye funktioner til interfacet dog ikke simpel. Hvis en ny funktion fx skal kommunikere med model-klassen (*CGiModel*), skal følgende programdele oprettes:

1. Hvis funktionen skal kunne tilgås fra demobrugerprogrammet, skal den implementeres i interfacet meddel C++ og C#.
2. Funktionen skal føjes til interfacemodulet i klassen *CSidm*.
3. Der skal oprettes en syncronized call klasse, der symboliserer funktionskaldet, så funktionen kan kommunikerer på tværs af trådene.
4. Funktionen skal oprettes i kontrolmodulet (*CSidmApplication*), der skal dirigere kaldet til det rigtige vinduesmodul.
5. En tilsvarende funktion skal oprettes i *CScene*, der er interfaceklassen til vinduesmodulet.
6. Til sidst skal funktionen tilføjes i *CGiModel*.

Punkt 1, 2, 3 og 6 kan ikke undlades, i den nuværende strukturering af SIDM. Med en mindre omstrukturering af kontrol- og vinduesmodulet vil det dog være muligt i mange tilfælde at springe punkt 4 og 5 over. Ideen er at tilføje yderligere information til syncronized call klasserne, der fortæller hvortil klassen skal sendes. En enkelt funktion i *CSidmApplication* kan behandle alle syncronized call klasser og hvis krævet, sende objektet videre til *CScene*.

Kodestruktur: Det er udviklerens overbevisning at kildekoden som krævet er velstruktureret og veldokumenteret (se afsnit 5.5). Dette krav gælder kun for SIDM og ikke for det udviklede demobrugerprogram eller de udviklede interfaces. Disse ekstra programdele er derfor ikke dokumenteret ligeså detaljeret som SIDM.

Brug af eksterne programbiblioteker: SIDM benytter ingen kommercielle API'er, der ikke er en del af operativsystemet, og opfylder derfor også dette krav.

6.4 OPSUMMERING

Dette kapitel beskriver de test SIDM har gennemgået og hvorvidt den endelige version af SIDM opfylder de stillede krav.

Alle funktionelle krav til SIDM er opfyldt. Desværre indeholder SIDM i øjeblikket en undvigende fejl, der får modulet til at lukke ned fra tid til anden. Ka-

pitlet foreslår hvordan dette problem løses.

De fleste ikke-funktionelle krav til modulet er også opfyldt. Målet om en simpel implementering af nye funktionaliteter er kun delvist opfyldt, da kommunikationsvejen fra brugerprogrammet til det endelige mål i vinduesmodulet kan gøres kortere. Tilsvarende må det siges, at kravet om, at SIDM ikke skal forbruge processorressourcer ved stilstand, ikke er opfyldt. Dette problem er dog ikke så stort, da programmet ved en brugstest ikke generede kørsel af andre programmer.

7 Konklusion

I dette projekt er der blevet udviklet et programmodul, der kan vise stereomodeller ud fra bitmap billeder. Programmodulet er navngivet *Stereo Image Display Module* (SIDM) og skal danne kernen i en videre udvikling af specifikke fotogrammetriske programmer ved IMM på DTU. SIDM kan ikke eksekveres alene, men skal kaldes fra et brugerprogram, via et veldefineret funktionsinterface. Et simpelt eksempel på et sådant brugerprogram er blevet udviklet i dette projekt.

SIDM har fulgt en iterativ udvikling, hvor hver iteration har defineret et afgrænset miniprojekt. Opdelingen i disse delprojekter er defineret i appendiks men har ikke haft indflydelse på struktureringen af rapporten.

Hovedparten af kravene til SIDM har været bundet fra starten af projektet. Use cases er opstillet ud fra de funktionelle krav, og disse use cases danner baggrund for designet af programmodulet. Designet af SIDM har gennemgået to faser. I første fase er det overordnede design tegnet uden at tage højde for programmeringssprog. Efter programmeringssprog og programpakker var blevet valgt, har SIDM gennemgået sit andet designforløb, hvor designet blev færdiggjort. Gennem hele forløbet har der været fokus på et velstruktureret, modulært design. Dette er til dels opnået ved at strukturere efter velkendte design patterns, hvor dette var muligt. I kildekoden er der ligeledes lagt vægt på en veldefineret, gennemgående strukturering, kommentering og navngivning. En fyldestgørende dokumentation er opbygget til at dokumentere hele kildekoden.

Valg af programmeringssprog og programpakker har sammen med indlæringsperioden for disse programpakker optaget en væsentlig del af projektperioden. Denne proces blev kompliceret af et valg om ikke at benytte platformspecifikke programbiblioteker. Efter en dybdegående analyse blev det valgt at benytte C++ til implementeringssprog for SIDM, mens C# benyttes til at implementere det simple brugerprogram. Billeder indlæses fra filer via programpakken *DevIL* [R34] og vises på skærmen ved brug af *OpenGLUT* [R24] og *OpenGL* [R21]. SIDM's database tilgås via *ODBC* og *LibODBC* [R37], mens tråde styres via *pthreads* [R29].

Den nuværende version af SIDM opfylder stort set alle stillede krav til modulet. De eneste mangler er to ikke-funktionelle krav, der ikke fuldt ud er opfyldt. Begge mangler anses dog for at være ganske ubetydelige, og har ingen indflydelse på kørsel af modulet. Mere kritisk er, at den nuværende version af SIDM indeholder en flygtig fejl, der opstår fra tid til anden ved brug af SIDM fra demobrugerprogrammet. Fejlen opstår ved brug af string klassen, som er en standard klasse i C++. Det er udviklerens overbevisning, at fejlen skyldes et kompatibilitetsproblem med *LibODBC*-API'en. Fejlen opstår ikke på udviklerens egen computer under almindeligt arbejde med SIDM, og den blev derfor opdaget for sent i projektforløbet til, at et alternativ til *LibODBC* kunne findes og implementeres.

7.1 FREMTIDIGE UDVIDELSER

Selv om det udviklede programmodul på nuværende tidspunkt opfylder alle de grundlæggende krav, er der stadig mulighed for forbedringer. I kravspecifikationen (afsnit 2.3) er der introduceret eksempler på fremtidige udvidelser til SIDM, og andre ideer er kommet frem gennem projektforløbet. Dette afsnit vil kort beskrive udviklerens forslag til fremtidige udvidelser og modifikationer af SIDM.

Det første mål må være at lokalisere og fjerne string fejlen (se afsnit 6.2.2). Dette vil formentligt kræve, at der benyttes en alternativ metode til kommunikation med databasen. Det umiddelbare forslag er, at kalde *ODBC* direkte uden brug af mellemliggende API. Hvis string fejlen fjernes, bør alle funktionaliteter i programmet køre fejlfrit.

Efter fejlen er elimineret, kan man begynde at føje ekstra funktionaliteter til programmodulet. Nedenfor følger en kort beskrivelse af alle foreslåede udvidelser til SIDM:

3D-streger: Implementering af funktionaliteten ’3d-streger i modellen’ (se afsnit 2.3.6 og afsnit 3.4.5).

Autobestemmelse af grundhastighed: I øjeblikket kan hastigheden, hvormed det vandrende mærke bevæges, ændres af brugeren med tastetryk. Bevægelse af det vandrende mærke udføres via objektkoordinater (eller kamerakoordinater). I begge tilfælde kan målestoksforholdet mellem disse koordinater og billedkoordinaterne være meget forskellig fra model til model, hvormed bevægelseshastigheden også vil variere meget. En foreslået udvidelse er at basere grundhastigheden på orienteringsparametrene af modellen, for på denne måde at have mere ens bevægelseshastigheder i de forskellige modeller.

Autoskift af modeller: Implementering af funktionaliteten ’Automatisk skift mellem modeller’ (se afsnit 2.3.6 og afsnit 3.4.5).

Billednormalisering: Afsnit 1.6 beskriver den matematiske beregning for normalisering af billeder. Da denne beregning består udelukkende af matrixoperationer, vil den være simpel at implementere i OpenGL. Eneste problem ligger i, at OpenGL ikke indeholder funktionaliteter til invertering af matricer, hvilket skal benyttes i beregningen.

Fejlbehandlingsmodul: Selvom alle fejl kan læses i de udskrevne logs fra det eksisterende logmodul, vil det være interessant for brugerprogrammet også at kunne hente disse fejl. Fejlbehandlingsmodulet er beskrevet i kapitel 3, primært i afsnit 3.3.3.

Implementer page flipping: I teorien burde page flipping allerede være implementeret i SIDM. Indledende test viste dog at page flipping i øjeblikket ikke virker korrekt. Det antages at problemet rimeligt nemt kan løses, hvorved denne funktionalitet er implementeret.

Interface til Delphi: Vejleder Keld Dueholm har planer om i nær fremtid at udvikle et brugerprogram til SIDM skrevet i Delphi. Det er derfor vigtigt at få færdigimplementeret det funktionsbaserede interface mellem C++ og Delphi.

Kompilering på andre operativsystemer: SIDM kan i teorien kompileres på både Windows, Unix og Linux systemer, når blot de benyttede API'er også er kompileret til disse systemer. Programmodulet er endnu ikke blevet testet på andre systemer end Windows, så det er en passende opgave at sikre, at programmodulet nu også kan kører på de andre operativsystemer.

Opsætningsprogram: Det ville hjælpe mange brugere, hvis ændringer til konfigurationsfilen kunne udføres via en brugerflade. Fx bindinger af taster og knapper kunne simplificeres kraftigt med et brugervenligt interface. En anden mulighed i et sådant program er en farveviser, der hjælper brugeren til at bestemme hvilke farver, der passer bedst til den benyttede skærm og de benyttede anaglyph-briller. Opsætningsdelen behøver ikke at blive implementeret i SIDM men kan være et stand-alone program.

Simplificeret rute af funktionskald: Som beskrevet i afsnit 6.3.4, er det i den nuværende version af SIDM rimeligt besværligt at implementere nye funktionskald til SIDM's interface. Samme afsnit beskriver hvordan dette evt. kan simplificeres ved en mindre omstrukturering af programmet.

Skærmorienteret bevægelse i modeltilstand: I den nuværende version af SIDM bliver bevægelse i modeltilstand udført parallelt med objektkoordinatsystemet. Fx bevæger musebevægelser det vandrende mærke parallelt med objektkoordinatsystemets xy-plan. Hvis kamerakoordinatsystemet er roteret mærkbart i forhold til objektkoordinatsystemet, vil bevægelsesinput ikke længe passe logisk med de bevægelser, det vandrende mærke udfører på skærmen. Løsningen på dette problem er, at musebevægelser påvirker kamerakoordinaterne i stedet for objektkoordinaterne. Da kamerakoordinaterne ligger parallelt med skærmen, vil sådanne bevægelser virke mere logisk for brugeren. Denne udvidelse bør ikke implementeres før billednormaliseringen er implementeret, da dette vil sørge for, at de to kamerakoordinatsystemer ligger parallelt med hinanden.

Udvidede indstillinger: Der er flere muligheder for udvidelse af indstillinger i SIDM. Fx kunne SIDM gemme tidligere indstillinger i programmet, så åbne vinduer, med deres størrelse, placering, åbnet model, osv. blev gemt i en fil. Indstillingsmulighederne for tekstboksen kunne også finpudses, så SIDM selv beregner mange af størrelserne i stedet for at overlade det til konfigurationen.

Appendiks

Denne rapport indeholder følgende information i appendiks:

Appendiks A Ordforklaringer: Dette appendiks opsummerer betydningen af specifikke forkortelser og betegnelser, der benyttes gennem rapporten.

Appendiks B Projektforløb: Programudviklingen har været opdelt i flere iterationer. Dette appendiks beskriver disse iterationer samt tidsforbrug generelt i projektet..

Appendiks C Vejledning til bruger: Dette appendiks indeholder en installationsvejledning og en brugervejledning til SIDM og det medfølgende demobrugerprogram. Her beskrives også hvordan programmodulet konfigureres samt beliggenheden af den elektroniske version af denne rapport.

Appendiks D Vejledning til udvikler: Med programmører som målgruppe beskriver dette appendiks de forskellige interfaces til SIDM. Her beskrives også placering af udviklerrelaterede filer på CD'en, inklusiv en fuld dokumentation af hele kildekoden.

Appendiks E Programdesign: Dette appendiks kan benyttes som et opslagsværk til designkapitlet. Det indeholder benyttede use cases, klassediagrammer, og databasens opbygning.

Appendiks F Test: Dette appendiks indeholder resultater af den funktionelle test af det endelige programmodul.

Appendiks G Referencer: Referencer til benyttet litteratur og programbiblioteker.

Figurer i appendiks er nummereret fortløbende over alle appendiks og er navngivet 'Figur A - ?'. Bogstavet 'A' står for Appendiks og ikke for appendiks A.

A Ordforklaringer

Betydningen af forkortelser og fagspecifikke ord, der benyttes i denne rapport, forklares første gang betegnelsen benyttes. Nedenstående opsummeres betydningen af betegnelser, der benyttes flere gange i løbet af rapporten.

Anaglyph: En metode til at vise en stereomodel ved at benytte farvefiltre til at adskille de to billeder.

Billedkoordinater: Et sæt billedkoordinater definerer en pixelposition i et billede. Nulpunktet for billedkoordinater befinder sig i øverste, venstre hjørne af billedet. Positive x-koordinater måles mod venstre, mens positive y-koordinater måles nedad. Fx vil punktet (7,5; 3) befinde sig syv og en halv pixels fra den venstre kant og 3 pixels fra den øvre kant af billedet.

Brugerprogram: Dette er en betegnelse for et program, der benytter SIDM.

Demobrugerprogram: Denne betegnelse benyttes om det brugerprogram, der er udviklet som en del af dette projekt til at teste funktionaliteterne i SIDM.

Dual Display: En metode til at vise en stereomodel ved at vise de to billeder på hver sin skærm eller via hver sin projektor.

Komparatortilstand: Et grafikvindue i SIDM kan befinde sig i enten model- eller komparatortilstand. I komparatortilstand kan billederne flyttes frit i forhold til hinanden. Denne tilstand vil ofte benyttes til at registrere koordinater for tilsvarende punkter i de to billeder. Disse koordinater kan benyttes til at beregne orienteringsparametre for billederne, så de senere kan vises i modeltilstand.

Markør: Ordet musemarkør eller blot markør henviser til den almindelige musemarkør, der benyttes i operativsystemet; ofte en pil. Markøren skal ikke forveksles med et målemærke.

Modeltilstand: Et grafikvindue i SIDM kan befinde sig i enten model- eller komparatortilstand. I modeltilstand vil billederne være orienteret i forhold til hinanden, så de altid ses i korrekt 3D. Målemærkerne vil i denne tilstand danne det vandrende mærke, der kan måle 3D koordinater.

Målemærke: Et målemærke er et lille ikon, der flyttes rundt i et billede, og benyttes til at udføre 2D-målinger i dette. Målemærket adskiller sig fra musemarkøren i og med, at målemærket aldrig forlader sit vindue. I en stereomodel vil to målemærker til sammen danne et vandrende mærke, der kan måle i 3D.

Objektkoordinater: Objektkoordinater definerer en 3D position i en stereomodel. Koordinaterne vil defineres ud fra billedernes orienteringsparametre og kan ud fra disse konverteres til billedkoordinater i de to billeder.

Page Flipping: En metode til at vise en stereomodel ved skiftevis at vise det højre og det venstre billede på skærmen.

SIDM: *Stereo Image Display Module* (SIDM) er navnet på det programmodul, der udvikles i dette projekt.

Stereoskopisk billedpar: To billeder der vha. stereoskopi har potentiale til at danne et 3D billede. En stereoskopisk model indeholder et stereoskopisk billedpar, men et stereoskopisk billedpar danner ikke nødvendigvis en stereoskopisk model.

Stereoskopisk model: En stereoskopisk model, også kaldet en stereomodel eller blot en model, er et sæt af to billeder, der vha. stereoskopi danner et 3D billede.

Stereoskopi: Stereoskopi er en teknik, der benyttes til at danne illusionen af dybde i todimensionale billeder ved at vise en anelse forskellige billeder på højre og venstre øje.

Vandrende mærke: Et vandrende mærke er et tredimensionalt målemærke der benyttes til 3D målinger i stereomodeller. Et vandrende mærke udgøres af to målemærker, et målemærke vises til hvert sit øje.

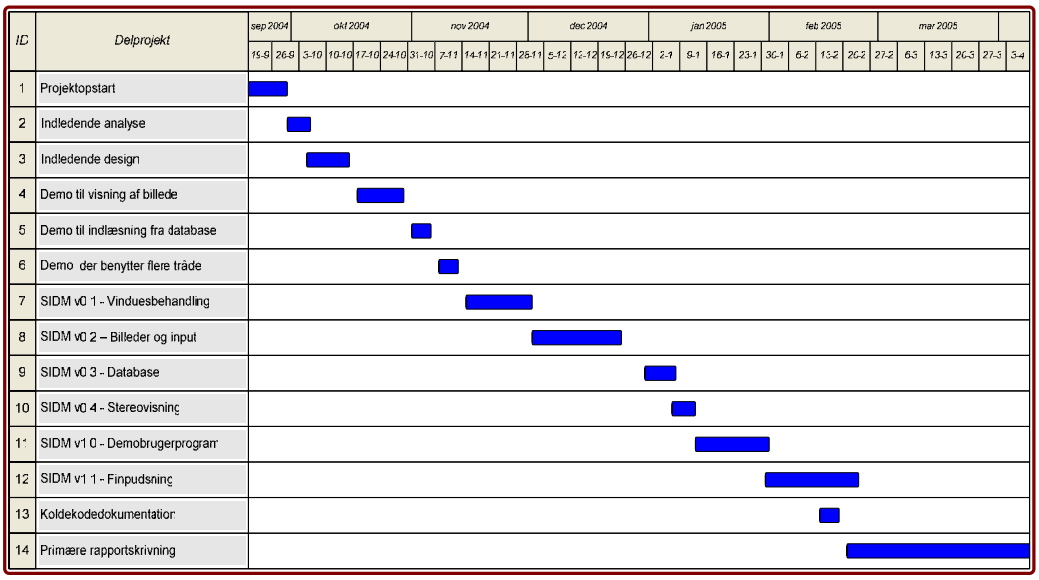
Vertikale parallakser: En stereoskopisk model indeholder vertikale parallakser, hvis det samme punkt i de to billeder ikke befinder sig på samme vandrette linie. En sådan model vil ikke ses korrekt i 3D.

B Projektforløb

I dette appendiks beskrives forløbet og tidsforbruget for dette projekt.

B.1 ITERATIONSPLAN

Dette projekt er udviklet som en længere række af delprojekter, der hver især har været mere eller mindre afsluttede. Denne opdeling i delprojekter er formaliseret i form af en iterationsplan. Figur A - 1 viser en tidsmæssig oversigt over denne iterationsplan. Samtlige delprojekter vil blive gennemgået i slutningen af dette afsnit.



Figur A - 1: Iterationsplan for SIDM. Figuren viser hvornår og hvor længe hvert delprojekt i SIDM har forløbet.

Detaljegraden i beskrivelserne af delprojekterne i dette appendiks er minimeret, så oversigten er lettere at overskue. De originale beskrivelser af delprojekterne indeholder langt flere detaljer, men disse har ikke den store relevans for læseren af denne rapport.

IT1 Projektopstart

Formål: Indledende analyse af problemstilling og definition af problemformulering.

Bemærkninger: Problemformuleringen blev udarbejdet i tæt samarbejde med vejleder Keld Dueholm.

Påbegyndelsesdato: 20/9-2004

Projekteret afslutningsdato: 29/9-2004

Realiseret afslutningsdato: 29/9-2004

IT2 Indledende analyse

Formål: Analyse af lignende programmer, projektdimensionering, første udkast til kravspecifikationen.

Bemærkninger: Kravspecifikationen og dimensioneringen blev udarbejdet i tæt samarbejde med vejleder Keld Dueholm.

Påbegyndelsesdato: 30/9-2004

Projekteret afslutningsdato: 5/10-2004

Realiseret afslutningsdato: 5/10-2004

IT3 Indledende design

Formål: Opbygning af use cases over første udkast til kravspecifikationen. Fastlæggelse af det overordnede implementeringsuafhængige design. Klarlægning af krævede programværktøjer.

Bemærkninger: Det blev fastlagt at SIDM ville have brug for værktøjer til at kommunikere med database, administrere flere tråde, indlæse billeder, vise grafik, administrere vinduer samt modtage input fra mus og tastatur.

Påbegyndelsesdato: 5/10-2004

Projekteret afslutningsdato: 15/10-2004

Realiseret afslutningsdato: 15/10-2004

IT4 Demo til visning af billede

Formål: Bestemmelse af programmeringssprog. Valg af primære programpakker til visning af grafik. Opbygning af et demoprogram, der kan vise et billede samt noget tekst i et grafikvindue og modtage simpelt input.

Bemærkninger: C++ blev valgt som programmeringssprog, DevIL, OpenGL, OpenGLUT og PUI blev valgt som API'er. Demoprogrammet opfylder kravene, men kan kun vise billeder, der kan eksistere som en enkelt tekstur. Dvs. billedets dimensioner skal være lig 2^n, fx 1024×1024.

Påbegyndelsesdato: 18/10-2004

Projekteret afslutningsdato: 25/10-2004

Realiseret afslutningsdato: 29/10-2004

IT5 Demo til indlæsning fra database

Formål: Valg af API til indlæsning fra database. Opbygning af et demoprogram, der kan indlæse fra en database.

Bemærkninger: LibODBC++ blev valgt som API. Udviklingen af dette demoprogram gav ingen større problemer.

Påbegyndelsesdato: 1/11-2004

Projekteret afslutningsdato: 5/11-2004

Realiseret afslutningsdato: 4/11-2004

IT6 Demo der benytter flere tråde

Formål: Valg af API til administrering af tråde og synkronisering. Opbygning af et demoprogram, der kan starte og lukke en tråd, samt synkronisere mellem de to tråde via *mutex* og *signal*/*wait*..

Bemærkninger: Pthreads blev valgt som API. Udviklingen af dette demoprogram gav ingen større problemer.

Påbegyndelsesdato: 8/11-2004

Projekteret afslutningsdato: 12/11-2004

Realiseret afslutningsdato: 12/11-2004

IT7 SIDM v0.1 – Vinduesbehandling

Formål: Udvikling (dvs. analyse, design, implementering og test) af første version af SIDM. Opbygning af et simpelt konsol-baseret C++ brugerprogram, der kan teste funktionaliteterne. Kravene til SIDM på dette stadie er, at samtlige funktionaliteter til vinduesbehandling skal implementeres.

Bemærkninger: Det viste sig at være ganske besværligt at holde hoved og hale på alle de benyttede tråde. Desuden skulle udvikles et interface til udskrivning af debuginformation, da output klassen i C++ ikke er *thread safe*. Dette interface vil også benyttes som en midlertidig løsning til information om fejl i SIDM. Exception handling benyttes internt i SIDM.

Påbegyndelsesdato: 15/11-2004

Projekteret afslutningsdato: 26/11-2004

Realiseret afslutningsdato: 1/12-2004

IT8 SIDM v0.2 – Billeder og input

Formål: Videreudvikling af SIDM og demobrugerprogrammet. De nye krav er, at SIDM korrekt skal vise et billede af en hvilken som helst størrelse, samt modtage brugerinput, der let kan ændres i konfigurationen.

Bemærkninger: Der blev udviklet et alternativt brugerprogram til SIDM, der kun benyttede en enkelt tråd. Det var et større projekt at finde frem til hvordan billeder korrekt skal vises i OpenGL, dvs. hvordan tiling fungerer.

Påbegyndelsesdato: 2/12-2004

Projekteret afslutningsdato: 22/12-2004

Realiseret afslutningsdato: 23/12-2004

IT9 SIDM v0.3 – Database

Formål: Videreudvikling af SIDM og demobrugerprogrammet. Design af databasen, udvikling af den del af SIDM, der skal kommunikere med databasen, og implementering af denne del i resten af SIDM.

Bemærkninger: Der opstod ingen problemer under udviklingen af databasedelen. Ved sammenkoblingen af databasedelen med resten af SIDM opstod komplikationer mellem PUI og LibODBC++. Det blev besluttet at opgive PUI.

Påbegyndelsesdato: 28/12-2004

Projekteret afslutningsdato: 4/1-2005

Realiseret afslutningsdato: 7/1-2005

IT10 SIDM v0.4 – Stereovisning

Formål: Videreudvikling af SIDM og demobrugerprogrammet. Implementering af anaglyph stereovisningsmetoden i SIDM og konvertering af objektkoordinater til billedkoordinater.

Bemærkninger: Det viste sig senere at hverken den benyttede anaglyph metode eller konverteringen fra objektkoordinater til billedkoordinater fungerede korrekt. Testen af dette delprojekt blev ikke udført grundigt nok.

Påbegyndelsesdato: 7/1-2005

Projekteret afslutningsdato: 12/1-2005

Realiseret afslutningsdato: 12/1-2005

IT11 SIDM v1.0 – Demobrugerprogram

Formål: Implementering af et grafisk demobrugerprogram til SIDM, der understøtter alle funktionaliteter i SIDM.

Bemærkninger: Det blev valgt at udvikle demobrugerprogrammet i C#. Forbindelsen mellem C++ og C# viste sig at være krævende at lukke.

Påbegyndelsesdato: 13/1-2005

Projekteret afslutningsdato: 21/1-2005

Realiseret afslutningsdato: 28/1-2005

IT12 SIDM v1.1 – Finpudsning

Formål: Finpudsning af programmet, implementering af mindre ændringer og rettelse af fejl i SIDM. Afsluttende test af programmodulet.

Bemærkninger: Dette delprojekt består i praksis af flere mindre iterationer. Flere gange i denne fase er kravspecifikationen blevet udvidet, og de nye funktionaliteter føjet til design og implementering for til sidst at blive testet. Af tilføjede funktionaliteter kan fx nævnes indlæsning af konfigurationsdata fra fil.

Påbegyndelsesdato: 31/1-2005

Projekteret afslutningsdato: 25/2-2005

Realiseret afslutningsdato: 23/2-2005

IT13 Kildekodedokumentation

Formål: Færdiggørelse af dokumentationen af kildekoden.

Bemærkninger: Kommentarer er løbende blevet skrevet i kildekoden, mens denne blev udarbejdet. Opgaven i dette delprojekt var at færdiggøre denne dokumentering af koden, og udtrække dokumentering til et eksternt dokument.

Påbegyndelsesdato: 14/2-2005

Projekteret afslutningsdato: 21/2-2005

Realiseret afslutningsdato: 21/2-2005

IT14 Primær rapportskrivning

Formål: Rapporten skrives.

Bemærkninger: Der er løbende nedskrevet informationer til rapporten gennem hele projektforløbet. Denne information kan være helt fra enkelte stikord til færdig tekst. I dette sidste delprojekt samles alle disse information i den endelige rapport.

Påbegyndelsesdato: 21/2-2005

Projekteret afslutningsdato: 8/4-2005

Realiseret afslutningsdato: 8/4-2005

B.2 TIDSFORBRUG

Figur A - 2 viser både det projekterede og det reelle tidsforbrug i dette projekt, fordelt på otte arbejdsopgaver:

For-analyse: Den indledende analyse af problemstillingen og udarbejdelse af problemformulering.

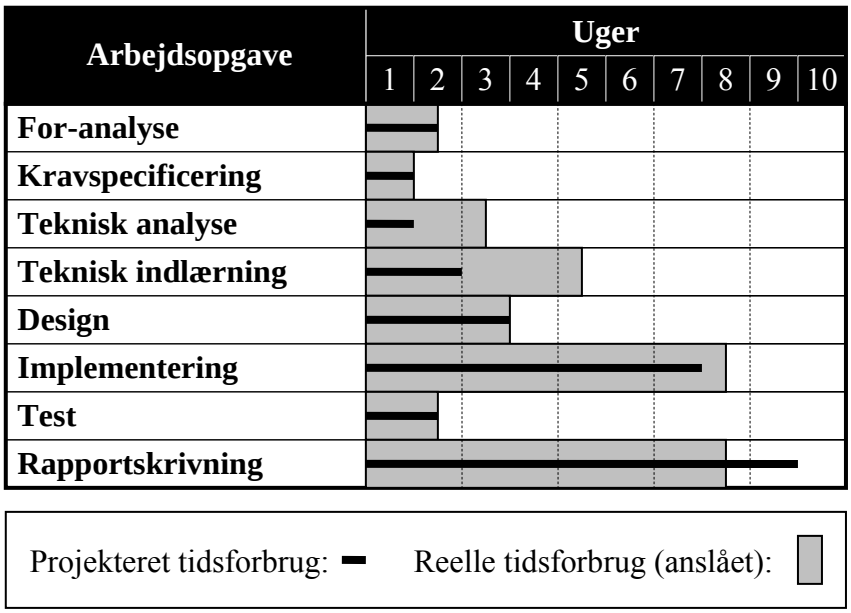
Kravspecificeringen: Dimensionering og fastsættelse af krav til programmet.

Teknisk analyse: Analyse og valg af programmeringssprog og programpakker.

Teknisk indlæring: Omfatter indlæring af programpakker og programmeringssprog.

Design: Inkluderer både det implementeringsuafhængige og det implementeringsspecifikke design.

Implementering, test og rapportskrivning: Titlerne er selvforklarende.



Figur A - 2: Projekteret og reelt tidforbug på de forskellige arbejdsopgaver under projektet. På grund af den iterative udvikling af projektet er det ikke muligt at give en præcis bestemmelse af det reelle tidsforbrug. De ovenstående reelle tidsforbrug er derfor anslåede værdier.

Som det ses af figuren, afviger det reelle tidsforbrug meget fra det projekterede for de to arbejdsområder; teknisk analyse og teknisk indlæring. Ved udarbejdelsen af den oprindelige plan for tidsforbrug, blev det antaget, at udvikleren allerede besad det meste af den fornødne tekniske viden til dette projekt. Pga. af de valgte programmeringssprog og programpakker viste denne antagelse sig tydeligt at være forkert. Det antages at omkring en fjerdedel af hele projektperioden er brugt på tilegnelse af ny viden om sprog og API'er. Grundet dette uventede tidsforbrug blev afleveringsfristen for projektet udskudt tre uger.

C Vejledning til bruger

Dette appendiks indeholder en installations- og brugervejledning af SIDM og demobrugerprogrammet. Afsnit C.1 beskriver de filer på den medfølgende CD, der har interesse for læseren af denne rapport. Afsnit C.2 indeholder en installationsvejledning af SIDM og demobrugerprogrammet. Afsnit C.3 og C.4 giver en brugervejledning til SIDM mens afsnit C.5 og C.6 beskriver brug af demobrugerprogrammet.

C.1 INDHOLD AF CD

To af de mapper, der findes på den medfølgende CD, har interesse for den almindelige læser af denne rapport. De tre resterende hovedmapper har kun relevans for en programmør. Disse beskrives i appendiks D i afsnit D.1.

- **Report:** Denne hovedmappe indeholder en elektronisk version af denne rapport på pdf format. Både indholdsfortegnelsen og alle henvisninger er implementeret som links, så læseren nemt kan bevæge sig rundt i rapporten.
- **SIDM:** Her findes en prækompileret version af SIDM og demobrugerprogrammet. Mappen indeholder følgende undermapper:
  - *BinRelease*: Indeholder en eksekverbar version af SIDM med demobrugerprogrammet.
  - *Database*: SIDM's database.
  - *images*: Indeholder de billeder, der benyttes af programmet.
  - *temp*: Denne mappe er i øjeblikket tom. SIDM vil placere logfilerne i denne mappe, hvis den eksisterer på kørselstidspunktet.

C.2 INSTALLATIONSVEJLEDNING

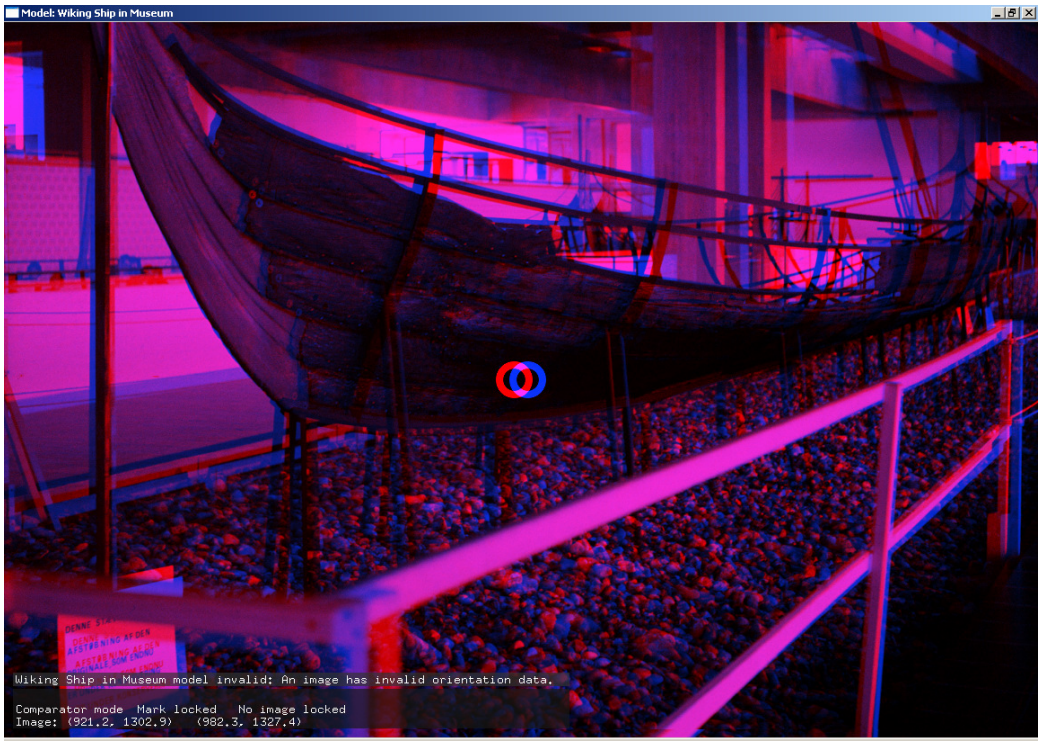
Før du kan eksekvere SIDM fra det medfølgende demobrugerprogram skal du kopiere det til harddisken og oprette en ODBC-forbindelse til databasen. Nedenstående beskrives denne procedure trin for trin:

- Kopier mappen *SIDM* fra CD'en over på harddisken. Mappen fylder omkring 230 Mb grundet de medfølgende billeder.
- Gå ind i Windows [Kontrolpanel] vælg [Administration] og dernæst [Datakilder (ODBC)].
- Tryk [Tilføj] og vælg [Microsoft Access Driver (*.mdb)] og dernæst [Udfør]
- Under [Datakildenavn:] skriver du "SidmDatabase". Tryk [Vælg] og find frem til det bibliotek, du lige har kopieret over på din harddisk. Gå ind under SIDM/Database og vælg "SidmDatabase.mdb". Tryk [Ok] et par gange til du er ude.
- Programmet startes fra filen "*SIDM/BinRelease/SIDMtestDialog.exe*".

C.3 GRAFIKVINDUE

SIDM kan via sit brugerprogram åbne et eller flere grafikvinduer. Dette afsnit beskriver et sådant grafikvindue, og hvordan brugeren direkte kan kommunikere med dette.

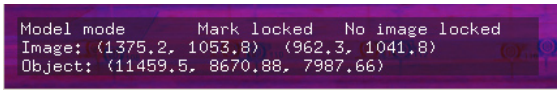
På Figur A - 3 ses et eksempel på et grafikvindue, der viser et stereobillede opbygget af to enkeltbilleder – et rødt og et blåt. Det røde billede skal ses med det venstre øje mens det blå billede skal ses med det højre. Til hvert billede følger et målemærke i en tilsvarende farve (ringene i midten af vinduet). Disse to målemærker danner tilsammen det vandrende mærke, når de ses stereoskopisk. De medfølgende anaglyph-briller kan benyttes til at opsplitte billederne på de to øjne, så de kan ses i stereo. Det røde brilleglas skal på det venstre øje.



Figur A - 3: Et grafikvindue i SIDM.

Tekstboks og tekstbarer

I nederste venstre hjørne af grafikvinduet i Figur A - 3 ses tekstboksen og en enkelt tekstbar (lige over tekstboksen). Tekstboksen informerer om den nuværende tilstand i grafikvinduet. Den øverste tekstlinie beskriver hvorvidt vinduet er i model- eller komparatortilstand, om mærket er låst eller frit, og om et af de to billeder er låst til mærket. Hvis vinduet viser mindst et billede, vil anden linie i tekstboksen beskrive målemærkernes (eller målemærkets) billedkoordinater. Den tredje og sidste linie i tekstboksen er synlig, hvis vinduet er i modeltilstand. Denne viser objektkoordinaterne af det vandrende mærke (se Figur A - 4).



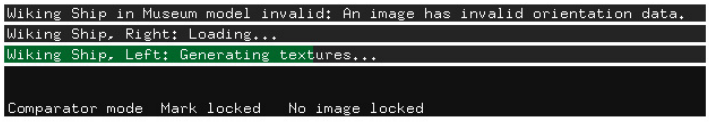
Figur A - 4: Tekstboks.

Indlæsningsstatus af billeder og model bliver skrevet i tekstbarer over tekstboksen. Et billede gennemløber fire trin under indlæsning:

- *"On queue to be loaded..."*: Billedet venter på at bliver indlæst.
- *"Loading..."*: Billedet indlæses fra sin billedfil.
- *"Splitting..."*: Billedet opsplittes i kvadrater, der kan benyttes til teksturer.
- *"Generating textures..."*: OpenGL genererer teksturerne.

Under *Splitting* og *Generating textures* vil tekstbaren udfyldes for at illustrere, hvor langt SIDM er med dette trin. Et eksempel på dette kan ses på Figur A - 5.

Hvis en fejl opstår under indlæsning af et billede eller en model, vil der ligeledes blive informeret om dette i en tekstbar. Fx vises beskeden *"An image has invalid orientation data"*, hvis mindst et af billederne i den indlæste model mangler orienteringsparametre. Dette resulterer i, at billederne kun kan vises i komparatortilstand.



Figur A - 5: Tekstbarer.

Tilstande

De forskellige tilstande i et grafikvindue beskrives i rapporten og bliver kort opsummeret her:

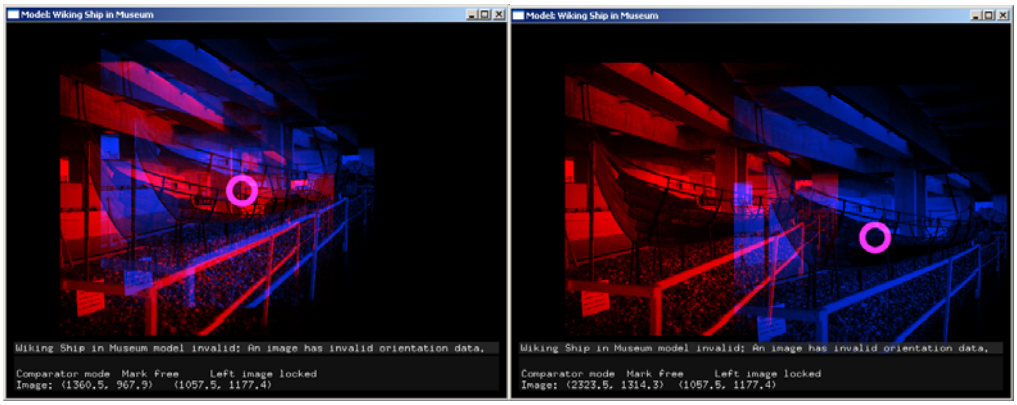
Model- eller komparatortilstand: I modeltilstand benyttes stråleligningerne til at beregne objektkoordinater og placere billeder og mærker korrekt i vinduet. Et stereobillede kan kun eksistere i modeltilstand, hvis alle orienteringsparametre er kendt. I komparatortilstand kan billeder bevæges frit i forhold til hinanden eller billeder kan vises enkeltvis.

Låst markør: Før målemærkerne kan bevæges, skal markøren låses til mærket.

Låst mærke: Hvis mærket er låst, vil billederne bevæges mens målemærkerne er stationære i vinduet. Hvis mærket er frit er billederne stationære, mens målemærkerne bevæges. Når et frit mærke bevæges mod kanten af vinduet, vil billeder og målemærker bevæges, så mærkerne flyttes nærmere centrum af vinduet. Afhængig af konfigurationen af SIDM kan dette enten ske flydende eller i spring. Konfigurationen bestemmer også størrelsen af denne rullegrænse (*scroll boarder*).

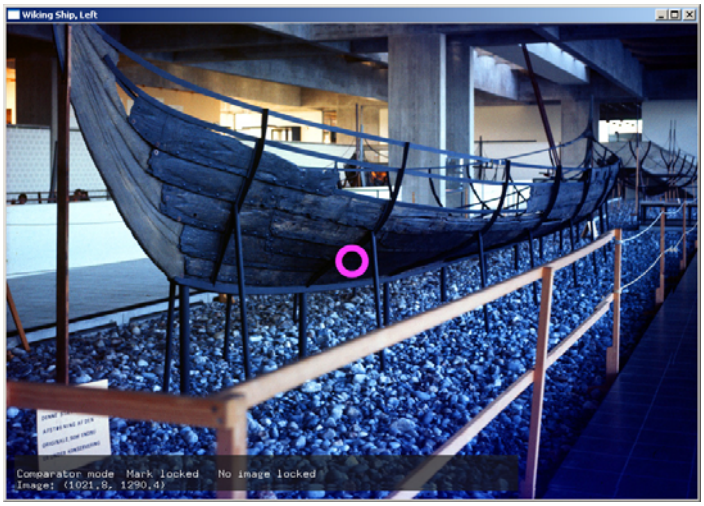
Låst billede: Hvis det aktive billede er låst, og vinduet er i komparatortilstand, vil det aktive billede være fastlåst til sit målemærke. På denne måde kan det ene billede trækkes rundt i forhold til det andet billede (se Figur A - 6).

Aktive billede: Enten det højre eller det venstre billede vil være det aktive billede.



Figur A - 6: Eksempel på låst aktivt billede i komparatortilstand.

Stereobillede eller enkeltbillede: Alle tidligere viste vinduer indeholder to billeder. Et vindue kan også åbne et enkelt billede, i hvilket tilfælde billedet ikke vises i anaglyph-farver (Figur A - 7).



Figur A - 7: Et enkelt billede vises uden brug af stereovisning.

Brugerinput

Forudsat at der ikke er ændret i konfigurationsfilen (se afsnit C.4), kan grafikvinduet påvirkes af brugeren på følgende måde:

Bevægelse af målemærker: Musebevægelser og piletaster benyttes til at bevæge målemærkerne i xy-planet, mens musehjulet og [PageUp]/[PageDown] bevæger det vandrende mærke i dybden.

Ændring af hastighed: Hastigheden, hvormed målemærkerne bevæges, kan ændres med [Ctrl] + [Pil op]/[Pil ned].

Størrelse af målemærker: Størrelsen af målemærkerne kan ændres med [Shift]+musehjul eller [Home]/[End].

Zoom i billeder: Billederne kan forstørres eller formindskes med [Ctrl]+musehjul eller [Insert]/[Delete].

Tilstandsændring: [Højre museknap] skifte mellem låst og fri markør, [Tab] skifter mellem låst og frit mærke, mens [Mellemrum] skifter mellem låst og frit aktive billede. De resterende tilstandsskift kan kun udføres via brugerprogrammet.

Input til brugefladen: [Venstre museknap] er bundet til primære input, mens tasterne [1] til [5] er sekundære input.

C.4 KONFIGURATION

Konfigurationen af SIDM bliver defineres i filen *SIDM/BinRelease/SIDM.ini*. Et godt forslag er at benytte Windows *Notesblok* (*Notepad*) med *Tekstombrydning* (*Word Wrap*) slået fra, til at editere i denne fil.

Filen er opbygget af to slags linier: Kommentarer (starter med '#') og indstillinger. Hver indstillingslinie starter med en indstillingsvariabel efterfulgt af værdien eller værdierne for denne variabel. Dvs.

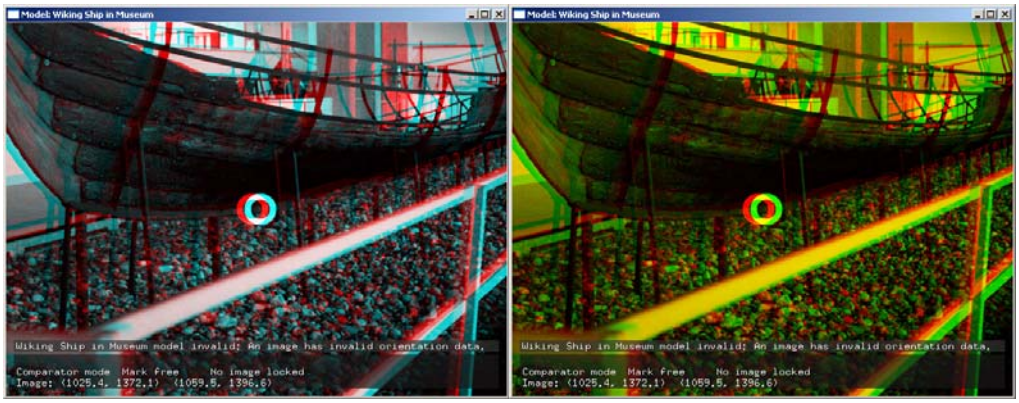
```
Indstillingsvariabel = værdi1 værdi2 værdi3 ...
```

Den nuværende version af SIDM benytter 43 forskellige indstillingsvariable, og de vil kun overfladisk blive gennemgået her. For yderligere information henvises til *SIDM.ini*, der i kommentarerne udførligt beskriver hvordan filen skal benyttes, og hvilke indstillingsmuligheder den indeholder. Det anbefales at disse kommentarer læses, inden man begynder at ændre på indstillinger.

Indstillingsvariablene er opdelt i seks grupper, hvor alle indstillingsvariable i en gruppe starter med det samme ord. Efterfølgende beskrives de forskellige grupper af indstillinger indledt med dette startord:

Log: Indstillingerne til loggen bestemmer, hvortil loggen skal skrives, og hvilke typer af information der skal skrives ud.

Stereo: SIDM er i øjeblikket kun testet til at virke med anaglyph-stereo. De benyttede farver til anaglyph kan fastsættes i stereo-indstillingerne. De tidligere billeder i denne rapport benytter rød/blå anaglyphfarver. Figur A - 8 viser to eksempler på alternative anaglyphfarver: Rød/cyan og rød/grøn.



Figur A - 8: Eksempler på andre anaglyph farver. Billedet tv. er rød/cyan mens billedet th. er rød/grøn.

Database: I databaseindstillingerne vælges navn på ODBC-forbindelse samt brugernavn og kodeord til databasen.

Input: Tastaturtaster og museknapper kan bindes til diverse kommandoer i inputindstillingerne. Her defineres også hastigheden, hvormed de forskellige input forløber, og placering af rullegrænsen.

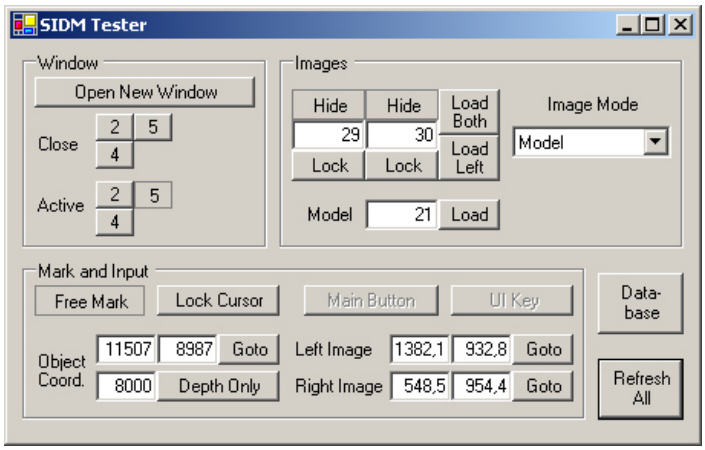
TextBox: I tekstboksindstillingerne vælges størrelse, placering og farve af tekstboksen og tekstbarer. Desuden vælges hvilken information, der skal vises i tekstboksen.

Mark: Indstillingerne til mærket bestemmer mærkets udseende samt hastighed af mærkets bevægelse ved brug af mus i forhold til taster.

Image: I billedindstillingerne defineres forskellige indstillinger for hvordan teksturerne opbygges.

C.5 TESTVINDUE

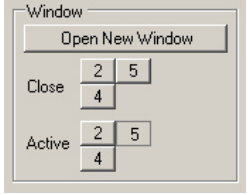
Når demobrugerprogrammet startes op, vil dialogboksen "*SIDM Tester*" åbne. Dette testvindue er inddelt i tre grupperinger af inputfelter og har yderligere to knapper uden for gruppering (se Figur A - 9). [*Database*]-knappen åbner databasevinduet (se afsnit C.6) mens [*Refresh All*] kontakter SIDM for at opdatere alt information i testvinduet. De tre grupperinger vil beskrives i efterfølgende afsnit:



Figur A - 9: Demobrugerprogrammets primære dialogboks.

Window

Vinduesgrupperingen (se Figur A - 10) indeholder knapper til administration af grafikvinduer i SIDM: [*Open New Window*] åbner et nyt vindue og genererer nye knapper ud for *Close* og *Active* med id på dette vindue.



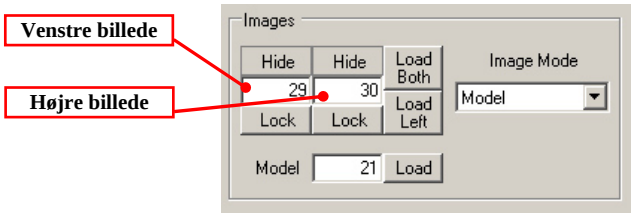
Figur A - 10: Vinduesgruppering.

Knapperne ud for *Close* trykkes for at lukke vinduet med det tilhørende id. [*Refresh All*] bør benyttes, efter et vindue er blevet lukket, for at opdatere knapperne i vinduesfeltet. Alternativt kan et grafikvindue lukkes på almindelig Windows-maner ved at trykke på krydset i hjørnet eller ved at taste [Alt]+[F4].

Knapperne ud fra *Active* fortæller hvilket vindue, der er aktivt i øjeblikket (den knap, der er trykket ned). Det aktive vindue kan skiftet enten ved at trykke på en af disse knapper eller ved at trykke en tast eller knap, mens musemarkøren er inde i det ønskede vindue.

Images

Billeder og modeller administreres i kontrolfelterne i billedgrupperingen Figur A - 11).



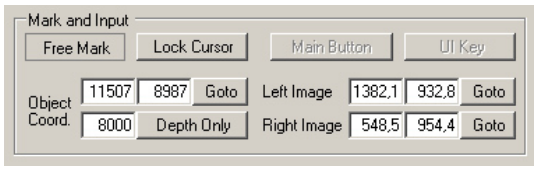
Figur A - 11: Billedgruppering.

[*Load Both*] åbner to billeder i komparator tilstand. Id på de to billeder skal forinden skrives i tekstfelterne til venstre for knappen. [*Load Left*] åbner kun det venstre billede og viser dette billede uden brug af stereovisning. De to [*Hide*]-knapper bestemmer om hvert af de to indlæste billeder skal skjules eller ej. [*Lock*] låser et af de to billeder.

[*Load*] knappen ud for *Model* kan benyttes til at indlæse en model i modeltilstand, mens drop-down menuen *Image Mode* til højre benyttes til at skifte mellem model- og komparatortilstand.

Mark and Input

Mærket administreres i mærkegrupperingen (Figur A - 12). Denne indeholder knapper til at låse og frigøre mærket (*Mark*) og markøren (*Cursor*). [*Goto*] knapperne flytter målemærkerne til specifikke objekt- eller billedkoordinater.



Figur A - 12: Mærkegruppering.

De to sidste knapper kan ikke aktiveres direkte, men benyttes i stedet som indikatorer, når brugeren i et grafikvindue aktiverer primær eller sekundær input til brugerprogrammet. Når det primære input til brugerprogrammet, dvs. [Højre

museknap], trykkes ned, vil [*Main Button*] blive trykkes ned, og den vil løfte sig igen, når knappen slippes. Hvis mærket ikke har skiftet position siden knappen blev trykket eller sluppet, vil [*Main Button*] have navnet [**Main Button**].

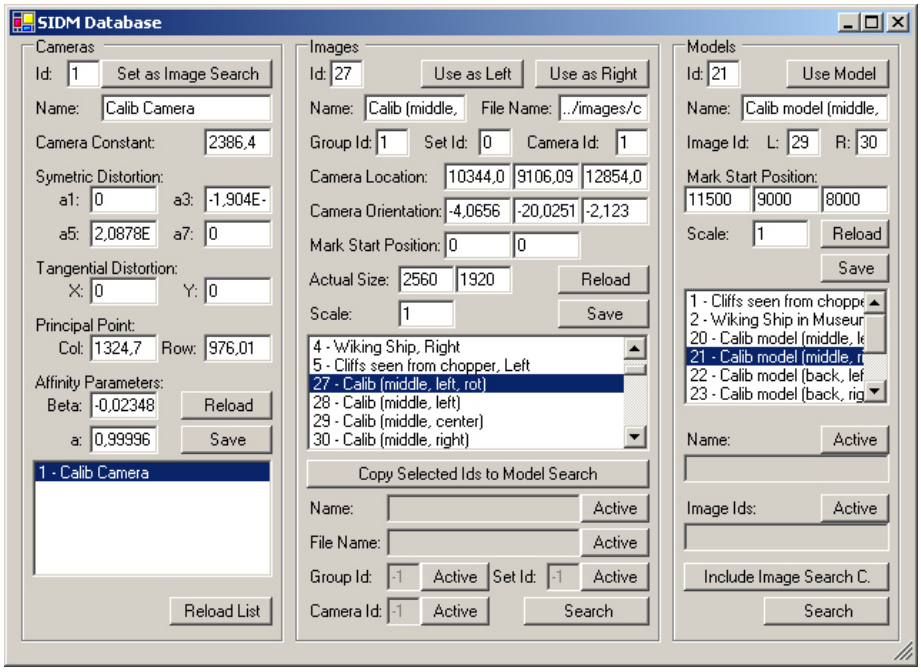
Når et sekundært input aktiveres i et grafikvindue, vil [UI Key] skifte tilstand mellem oppe og nede. Samtidig vil ID på det sekundære input skrives i knappen. Figur A - 13 viser et eksempel, hvor primære input holdes nede uden at flytte mærket, mens det sidste modtagne sekundære input har ID 3.

Figur A - 13: Brugerprogrammet har modtaget input.

Formålet med demobrugerprogrammet er at teste samtlige funktionaliteter i SIDM. Det er derfor nødvendigt at vise når primær og sekundær input til brugerprogrammet modtages, også selvom disse informationer ikke benyttes til noget.

C.6 DATABASEVINDUE

Databasevinduet åbnes ved at trykke [*Database*]-knappen i testvinduet. Vinduet er delt op i tre grupperinger, der svarer til hver sin tabel i databasen (se Figur A - 14).



Figur A - 14: Demobrugerprogrammets databasedialogboks.

Indholdet i hver af de tre grupperinger er opdelt på samme måde:

Øverst i grupperingen vises data for det markerede element. Disse informationer kan ændres af brugeren, og knappen [*Save*] gemmer informationen i databasen. [*Reload*]-knappen benyttes til at genindlæse data for det markerede element.

I midten af grupperingen vises en liste over elementer i tabellen. Data for det element, der markeres i denne liste, vil vises i de ovenstående felter.

Nederst i grupperingen findes felter til søgning i tabellen. For at benytte et givent søgekriterium skal den tilhørende [*Active*]-knap trykke ned. Søgningen udføres ved at trykke [*Search*]. Resultaterne fra søgningen vises i listen ovenover.

Cameras

Foruden kameraets id og navn indeholder kameratabellen de indre orienteringsparametre for det givne kamera. Dette inkluderer kamerakonstanten, konstanter til bestemmelse af den radialsymmetriske fortegning, konstanter til bestemmelse af den tangentielle fortegning (benyttes ikke endnu i SIDM), hovedpunktets koordinater i billedet, samt konstanter til bestemmelse af den affine transformation.

Knappen [*Set as Image Search*] kopierer kameraets id til det tilhørende søgekriteriefelt i billedgrupperingen.

Der kan ikke udføres søgninger i kameratabellen. [*Search*]-knappen er derfor udskiftet med en [*Reload*]-knap, som genindlæser ovenstående liste fra databasen.

Images

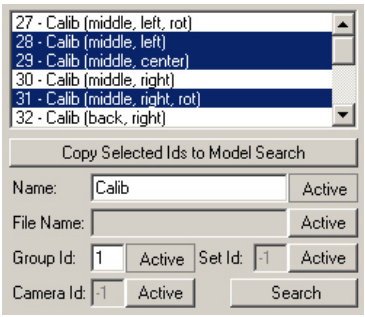
Billedtabellen indeholder følgende informationer for hvert billede:

- Id, navn og filnavn på det angivne billede.
- Id, position og rotation af det kamera, der tog billedet (de ydre orienteringsparametre).
- Startpositionen af målemærket og zoomfaktor, hvis billedet åbnes i komparatortilstand.
- Gruppe-Id, sæt-Id og den aktuelle størrelse af billedet benyttes endnu ikke af SIDM, men skal benyttes ved udvidelsen ”automatisk skift af modeller”.

Knapperne [*Use as Left*] og [*Use as Right*] kopierer det aktive Id til det tilsvarende felt i testvinduet (se Figur A - 11).

Specielt i billedlisten er det muligt at markere flere billeder (se Figur A - 15). Kun data for det billede, der sidst blev markeret, vil vises i datafelterne. Knappen [*Copy Selected Ids to Model Search*] kopierer id på alle valgte billeder til det tilhørende søgekriteriefelt i modelgrupperingen.

Søgekriteriefelterne i billedgrupperingen er navngivet efter de tilhørende datafelter. Figur A - 15 viser et eksempel på hvordan søgekriteriefelterne aktiveres, når [*Active*] er trykket ned. Den viste søgning har udvalgt alle billeder med ordet ”Calib” i sit navn og med gruppe-Id 1. De udvalgte billeder i denne søgning er alle billeder i listen, ikke kun dem, der er markeret af brugeren.



Figur A - 15: Søgningsfelter for billeder.

Models

Foruden id og navn indeholder modeltabellen id på de to billeder i modellen, samt startposition af mærket og startzoomfaktor.

[Use Model] kopierer det aktive Id til *model id* feltet i testvinduet.

Hvis knappen [*Include Image Search C.*] er aktiv, vil begge billeder i en model undersøges ud fra billedsøgekriterierne. Minimum et af disse billeder skal bestå denne test, før en model kan opfylde søgekriteriet. Hvis søgekriteriefeltet *Image Ids* er aktivt, vil billeder i denne liste automatisk bestå billedsøgekriterierne. Billederne opskrives i denne liste ved deres id adskilt af komma eller semikolon.

D Vejledning til udvikler

Hvor det forrige appendiks fokuserer på en bruger af SIDM, fokuserer dette appendiks på en programmør, der ønsker at benytte eller videreudvikle SIDM. Her beskrives de resterende filer på CD'en, samt benyttelse af samtlige interfaces til SIDM.

D.1 INDHOLD AF CD

To af hovedmapperne på den medfølgende CD bliver gennemgået i appendiks afsnit C.1. De resterende tre mapper vil blive gennemgået i dette afsnit.

SourceCode

Dette hovedbibliotek indeholder alt kildekode udviklet i forbindelse med projektet, samt filer der benyttes ved kompilering af programdelene. De fire mapper i hovedbiblioteket SIDM er kopieret fra dette bibliotek.

Fire underbiblioteker indeholder filer fra de benyttede API'er: *BinDebug* og *BinRelease* indeholder kompilerede, binære, dynamiske biblioteker for de benyttede API'er. Binære filer fra SIDM og tilhørende programmoduler vil placeres i disse mapper efter kompilering. *BinDebug* indeholder filer kompileret ved brug af *Multithreaded debug Dll Runtime Library*, mens *BinRelease* indeholder filer kompileret ved brug af *Multithreaded Dll Runtime Library*. Mapperne *includes* og *lib* indeholder henholdsvis *headerfiler* og *import libraries* for de benyttede API'er.

Kildekode til de forskellige programmoduler (*projects*), der er udviklet i forbindelse med SIDM, findes i seks forskellige mapper. Disse mapper indeholder også de tilhørende Visual Studio projekt-filer.

- *SIDM*: Selve SIDM.
- *SIDMfunctionInterface*: Det funktionsbaserede (*C-style*) interface til SIDM. Dette interface er ikke fuldt ud implementeret.
- *SIDMtest*: Primitivt brugerprogram, der kan benyttes til at teste det originale C++ interface til SIDM.
- *SIDMtestDialog*: Demobrugerprogrammet skrevet i C#, der kan benyttes til at teste samtlige funktionaliteter i SIDM via Managed C++ interfacet (*SIDMwrap*).
- *SIDMtestFuncInt*: Primitivt brugerprogram, der kan benyttes til at teste det funktionsbaserede interface til SIDM.
- *SIDMwrap*: Managed C++ interface til SIDM.

Projekter bliver i Visual Studio grupperet i såkaldte *solutions*. Hver solution i SIDM er placeret i sit eget bibliotek og indeholder et til tre programmoduler:

- *SIDM_Solution*: Indeholder kun programmodulet SIDM. I denne solution kompileres SIDM til en exe-fil i stedet for den sædvanlige dll-fil. Exe-

filen kører i kun én tråd og åbner et enkelt vindue med tilhørende model eller billeder. Grundet mangel på flere tråde, tillader denne solution debugging i vinduesmodulet.

- *SIDM_SolutionConsole*: Indeholder *SIDM* og *SIDMtest* og benyttes til at teste hele SIDM uden at være generet af evt. fejlbehæftede interfaces.
- *SIDM_SolutionDialog*: Indeholder *SIDM*, *SIDMwrap* og *SIDMtestDialog*. Denne solution genererer demobrugerprogrammet med alle krævede filer. Mappen *SIDM/BinRelease* indeholder en kompilering af denne solution.
- *SIDM_SolutionFunctionInterface*: Indeholder *SIDM*, *SIDMfunction-Interface* og *SIDMtestFuncInt*. Denne solution benyttes til at teste det funktionsbaserede interface.

De resterende underbiblioteker *Database*, *images* og *temp* er forklaret i afsnit C.1. Biblioteket *temp* benyttes desuden til at gemme midlertidige filer i forbindelse med kompilering.

SourceDocumentation

Denne mappe indeholder kildekodedokumentationen af SIDM. Dokumentationen udklippes fra kildekodefilerne vha. programmet ccdoc, som findes i undermappen af samme navn.

To versioner af dokumentationen er genereret: *Usage_Doc* indeholder kun dokumentation for interfacet til SIDM, samt benyttede typer. *Full_Doc* indeholder hele dokumentationen af SIDM. Mappen indeholder to batch filer, der kan eksekveres for at gendanne dokumentationen af SIDM. For at disse filer kan eksekvere korrekt kræves, at biblioteksstrukturen på CD opretholdes når filerne overføres til harddisken.

SourceDocumentation.html skal køres for at læse dokumentationen af SIDM.

UsedAPIs

I denne mappe findes de fulde versioner af de API'er, der benyttes af SIDM. Før SIDM kan kompileres på andre platforme end Windows kræves at disse API'er genkompileres til den nye platform. Alle filer er pakket med zip.

D.2 C++ INTERFACE

Dette afsnit beskriver overordnet, hvordan SIDM kaldes fra brugerprogrammet uden brug af et eksternt interface. Interfacet til SIDM er implementeret i klassen *CSidm* og består af omkring 40 funktioner. Efterfølgende vil disse funktioner meget kort beskrives. Der henvises til dokumentationen på den medfølgende CD, hvis flere detaljer ønskes:

Opstart og nedlukning: SIDM startes op ved at oprette en instans af klassen *CSidm* og kalde funktionen *Initialize*. Modulet lukkes ned ved at kalde *Terminate*.

Vinduesadministration: Grafikvinduer åbnes og lukkes med funktionerne: *OpenWindow* og *CloseWindow*. *GetActiveWindow* og *SwitchToWindow* infor-

merer om eller vælger, hvilket vindue der er aktivt. *SetWindowChangedFunc* benyttes til at definere en funktion i brugerprogrammet, der skal kaldes, når det aktive vindue skifter. *GetAllWindows* giver id på alle åbne vinduer.

Administration af billeder og modeller: *SetModelMode* definerer om det aktive vindue er i model- eller komparatortilstand, mens *IsInModelMode* informerer om den angivne tilstand. Modeller og billeder åbnes med funktionerne *LoadModel* og *LoadImages*, mens *GetModelId* og *GetImageIds* informerer om hvilke af disse, der er åbne. *SetShownImages* vælger, om hvert af de to billeder skal vises eller skjules. Skift og låsning af det aktive billede administreres af funktionerne *GetActiveImage*, *SetActiveImage*, *IsImageLocked* og *SetImageLock*.

Administration af mærke og input: *IsMarkLocked*, *SetMarkLock*, *IsCursorLocked* og *SetCursorLock* administrerer låsning af mærket og markøren. *GetCoordinates* returnerer mærkets (både målemærkerne og det vandrende mærke) nuværende koordinater, mens *GotoImageCoord* og *GotoObjectCoord* flytter mærket til nye koordinater. Funktionerne *SetUiButtonPressedFunc* og *SetCoordChangedFunc* definerer hvilke funktioner i brugerprogrammet, der skal kaldes, når mærket flyttes, og brugerinput aktiveres.

Kommunikation med database: Funktionerne *GetImageData*, *SetImageData* og *FindImages* benyttes til at hente data om et givent billede, indlæse nyt data om et billede eller udføre en søgning billedtabellen. Tilsvarende benyttes funktionerne *GetCameraData*, *SetCameraData* og *FindCameras* til kameraer og *GetModelData*, *SetModelData* og *FindModels* til modeller.

D.3 MANAGED C++ INTERFACE

Hvis SIDM skal benyttes fra en .NET applikation, skal managed C++ interfacet benyttes. Dette interface implementeres i klassen *WSidm*. *WSidm* benyttes på samme måde og benytter de samme navne som *CSidm*. Den eneste forskel foruden klassenavnet er de typer, der benyttes til argumenter og returværdier.

Alle enumerationer og strukturer, der er defineret i SIDM, bliver gendefineret i managed C++ interfacet. Navnet ændres ved at tilføje et 'W' i starten, fx *MDirection* kaldes *WMDirection* i dette interface. De resterende argumenttyper i SIDM konverteres på følgende måde.

Unmanaged C++ type		**Managed C++ type**
Int32	→	int
Float32	→	float
Float64	→	double
WindowChangedType	→	WindowChangedDelegate
UiButtonPressedType	→	UiButtonPressedDelegate
CoordChangedType	→	CoordChangedDelegate
list<int>	→	System::Collections::ArrayList
(int&, int&)	→	System::Collections::ArrayList

Dokumentationen af SIDM indeholder ingen informationer om dette interface. Da de fleste .NET applikationer udvikles i Visual Studio, er dette ikke noget problem. Visual Studio vil give de nødvendige informationer under opbygning af koden i form af *drop-down* menuer.

D.4 C-STYLE INTERFACE

I forhold til C++ interfacet, vil det funktionsbaserede (*C-style*) interface ofte være lettere at implementere i andre programmeringssprog. Dette interfacet er endnu ikke fuldt udviklet, men er alligevel medsendt på CD'en.

Interfacet indeholder kun globale funktioner og ingen klasser. Funktionsnavnene passer med de tilsvarende funktioner i basisinterfacet, hvor 'Sidm' er tilføjet til starten af navnet. Fx kaldes *CSidm*-funktionen *Terminate* for *SidmTerminate* i det funktionsbaserede interface. Argumenterne er indtil videre de samme som benyttes i basisinterfacet, men dette vil ændres når resten af funktionerne tilføjes.

Filen *SIDMfuncInt.h* i mappen *SourceCode\SIDMfunctionInterface* indeholder information om, hvordan det funktionsbaserede interface implementeres i Delphi.

E Programdesign

Nedenstående findes detaljer om designet af SIDM, som er for omfattende til at inkludere i rapporten. Dette appendiks er ment som et opslagsværk, der kan benyttes under gennemlæsning af kapitel ’3 Design’ i rapporten. Det foreslås derfor ikke at læse dette appendiks uafhængigt af rapporten.

E.1 USE CASES

Nedenstående er nedskrevet samtlige use cases over den nuværende kravspecifikation. *Primary actor* er altid brugeren, derfor er denne linie udeladt.

Use case UC1 til og med UC41 beskriver de obligatoriske funktionaliteter, mens UC42 og frem beskriver de udvidede funktionaliteter.

UC1 Opstart af programmodul

Stakeholders & Interests

Bruger: Ønsker at starte programmodulet op.

Precondition

-

Postcondition

Programmodulet er startet op.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at starte programmodulet op.
2. Systemet starter programmodulet op.

Extensions

2a. Programmodulet er allerede startet.
    1. Intet sker.

UC2 Nedlukning af programmodul

Stakeholders & Interests

Bruger: Ønsker at nedlukke programmodulet op.

Precondition

-

Postcondition

Programmodulet er lukket ned.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at lukke programmodulet ned.
2. Systemet lukker programmodulet ned.

Extensions

2a. Programmodulet er allerede lukket ned.
    1. Intet sker.

UC3 Åbning af vindue

Stakeholders & Interests

Bruger: Ønsker at åbne et nyt grafikvindue.

Precondition

Programmet er startet op.

Postcondition

Et nyt vindue er blevet åbent.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at åbne et nyt vindue.
2. Systemet åbner et nyt vindue og sender brugerprogrammet et id på det nye vindue.

UC4 Lukning af vindue

Stakeholders & Interests

Bruger: Ønsker at lukke et grafikvindue.

Precondition

Programmet er startet op.

Postcondition

Vinduet er blevet lukket.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at lukke et givent vindue identificeret ved sit id.
2. Et vindue med det angivne id findes og er åbent.
3. Systemet lukker vinduet.

Extensions

1a. Brugeren lukker et vindue via operativsystemet. I Windows kunne dette være at trykke på krydset i hjørnet af vinduet.
    1. Systemet lukker vinduet.

2a. Der findes intet åbent vindue med det angivne id.
    1. Intet sker.

UC5 Tilsending af information om vinduesskift

Stakeholders & Interests

Bruger: Ønsker at konfigurere systemet til, hvorvidt det skal sende information om skift af vinduer til brugerprogrammet eller ej.

Precondition

Systemet er startet op.

Postcondition

Systemet er konfigureret som ønsket.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at konfigurere systemet til at sende information om skift af vinduer til brugerprogrammet. Evt. information om hvordan

systemet kontakter brugerprogrammet medsendes.
2. Systemet vil i fremtiden informere brugeren om skift af vinduer.

Extensions

1a. Et funktionskald aktiveres fra brugerprogrammet for at konfigurere systemet til ikke at sende information om skift af vinduer til brugerprogrammet.
    1. Systemet vil i fremtiden ikke informere brugeren om skift af vinduer.

UC6 Skift af aktive vindue

Stakeholders & Interests

Bruger: Ønsker at ændre hvilket vindue der er det aktive vindue.

Precondition

Et grafikvindue er åbent.

Postcondition

Det ønskede grafikvindue er nu det aktive vindue.

Main Success Scenario

1. Musemarkøren befinder sig inden for rammen af det vindue, der ønskes at gøre aktivt.
2. Brugeren trykker en tast eller knap.
3. Det ønskede vindue er ikke allerede det aktive vindue.
4. Systemet skifter til at det ønskede vindue nu er det aktive vindue.
5. Systemet sender information til brugerprogrammet om hvilket vindue der nu er blevet aktivt.

Extensions

1a. Et funktionskald aktiveres fra brugerprogrammet for at sætte et givent vindue identificeret ved sit id til at være det aktive vindue.
    1. Et vindue med det ønskede id eksisterer.
        1a. Der eksisterer intet vindue med det angivne id.
            1. Intet sker.
    2. Gå til pkt. 3.

3a. Det ønskede vindue er allerede det aktive vindue.
    1. Intet sker.

5a. Brugerprogrammet ønsker ikke at få tilsendt info om hvilket billede, der er aktivt.
    1. Intet sker.

UC7 Info om hvilket vindue der er aktivt

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at få at vide hvilket vindue der er aktivt i øjeblikket.

Precondition

Programmet er startet op.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet.

2. Systemet svarer tilbage med id på det aktive vindue.

Extensions

2a. Intet vindue er aktivt.
    1. Systemet svarer tilbage med information om, at intet vindue er aktivt.

UC8 Info om hvilke vinduer der er åbne

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at få at vide id'er på samtlige åbne vinduer.

Precondition

Programmet er startet op.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet.
2. Systemet svarer tilbage med en liste af id'er på de vinduer, der er åbne.

UC9 Info om model- eller komparatortilstand er aktiv

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at få at vide hvorvidt modeltilstand eller komparatortilstand benyttes.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet.
2. Systemet svarer tilbage med hvilken tilstand vinduet er i (model eller komparator).

UC10 Skift mellem model og komparatortilstand

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at vælge om vinduet skal være i model- eller komparatortilstand.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Vinduet er nu i den ønskede tilstand.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at sætte vinduet i modeltilstand.
2. Systemet er i øjeblikket i komparatortilstand.

3. En model er åben.
4. Systemet skifter til modeltilstand.

Extensions

1a. Et funktionskald aktiveres fra brugerprogrammet for at sætte vinduet i komparator-tilstand.
    1. Systemet er i øjeblikket i modeltilstand.
        1a. Systemet er i øjeblikket i komparatortilstand.
            Intet sker.
    2. Systemet skifter til komparatortilstand.
2a. Systemet er i øjeblikket i modeltilstand.
    1. Intet sker.
3a. Ingen model er åben.
    1. Intet sker.

UC11 Vælg hvorvidt et billede er synligt eller skjult

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at vælge om hvert af billederne er synlig eller skjult.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Billedernes synlighed er nu som ønsket.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at vælge om hvert af billederne skal være synlig eller skjult.
2. Systemet ændrer synlighed på hvert af de to billeder så de efterfølgende passer til ønsket. Hvis et billede er skjult, skjules også det tilhørende målemærke. Hvis et billede ikke er åbnet, vil det ikke blive vist, selvom det er defineret synligt.

UC12 Indlæsning af en model

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at åbne en model.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Den ønskede model er nu åben.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at indlæse en ny model. Modellens id medsendes.
2. Billeder eller den model, der allerede er vist i vinduet, fjernes.
3. Systemet åbner den ønskede model.
4. Systemet skifter til modeltilstand.
5. Systemet flytter det vandrende mærke til en position, der er defineret i modellens data.

Extensions

3a. Det ønskede model id findes ikke i databasen.
    1. Systemet informerer i grafikvinduet brugeren om fejl ved indlæsning af model.

3b. Systemet kan ikke finde et eller flere af de ønskede billeder på den i databasen angiven sti, eller en fejl opstår under indlæsning af en billedfil.
    1. Systemet informerer i grafikvinduet brugeren om fejl ved indlæsning af det ene eller begge billeder.

4a. Billederne i -, eller kameraet tilhørende den ønskede model har ikke de fornødne orienterings-parametre.
    1. Systemet skifter til komparatortilstand.
    2. Systemet informerer i grafikvinduet brugeren om fejl ved indlæsning af model.

UC13 Info om hvilken model der vises

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at få at vide hvilken model, der vises.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet.
2. Systemet svarer tilbage med id på den aktive model.

Extensions

2a. Ingen model vises i øjeblikket.
    1. Systemet svarer tilbage med information om at ingen model vises.

UC14 Indlæsning af billeder, der ikke tilhører en model

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at åbne et eller to billeder uden for en model.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Den eller de ønskede billeder er nu åbne.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at indlæse et eller to billeder. Billedernes id medsendes.
2. Billeder eller den model, der allerede er vist i vinduet fjernes.
3. Systemet åbner den eller de ønskede billeder.
4. Systemet skifter til komparatortilstand.
5. Systemet flytter målemærkerne i forhold til billederne og billederne i forhold til hinanden, så positionen af hvert målemærke i det tilhørende billede svarer til det, der er defineret i billedernes data.

Extensions

3a. Et eller flere af de ønskede id'er findes ikke i databasen.

1. Systemet informerer i grafikvinduet brugeren om fejl ved indlæsning af det ene eller begge billeder.

3b. Systemet kan ikke finde et eller flere af de ønskede billeder på den i databasen angiven sti, eller en fejl opstår under indlæsning af en billedfil.
1. Systemet informerer i grafikvinduet brugeren om fejl ved indlæsning af det ene eller begge billeder.

UC15 Info om hvilke billeder, der vises

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at få at vide hvilke billeder, der vises.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet.
2. Systemet svarer tilbage med id på de aktive billeder.

Extensions

2a. Kun et billede vises i øjeblikket.
1. Systemet svarer tilbage med information om at kun et billede vises, samt id på dette billede.

2b. Ingen billeder vises i øjeblikket.
1. Systemet svarer tilbage med information om at ingen billeder vises i øjeblikket.

UC16 Info om hvilket billede er det aktive

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at få at vide hvilket billede, der er det aktive.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet.
2. Systemet svarer tilbage med hvorvidt venstre eller højre billede er aktivt.

UC17 Skift mellem hvilket billede der er det aktive

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at vælge hvilket billede, der er det aktive.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

De ønskede billede er nu aktivt.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at definere hvilket af billederne, der ønskes aktivt.
2. Systemet skifter om nødvendigt så det ønskede billede bliver aktivt.

UC18 Info om et billede er låst eller frit

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at få at vide om det aktive billede er låst eller frit.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet.
2. Systemet svarer tilbage med hvorvidt det aktive billede er låst til sit målemærke eller er frit.

UC19 Skift mellem låst og frit billede

Stakeholders & Interests

Bruger: Ønsker at skifte mellem at det aktive billede er frit eller låst til sit målemærke.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Det aktive billede har skiftet tilstand mellem låst og frit.

Main Success Scenario

1. Brugeren trykker tasten, der er bundet til at skifte mellem låst og frit billede. Fx mellemrum.
2. Billedet er frit.
3. Systemet låser billedet til sit målemærke.

Extensions

1a. Et funktionskald aktiveres fra brugerprogrammet for at sætte det aktive billede til at være låst.
    1. Billedet er frit.
        1a. Billedet er låst.
            1 Intet sker.
    2. Systemet låser billedet til sit målemærke.

1b. Et funktionskald aktiveres fra brugerprogrammet for at sætte det aktive billede til at være frit.
    1. Billedet er låst.
        1a. Billedet er frit.

1 Intet sker.

2. Systemet frigører billedet fra sit målemærke.

2a. Billedet er låst.

1. Systemet frigører billedet fra sit målemærke.

UC20 Info om markøren er låst eller fri

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at få at vide om markøren er låst eller fri.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet.
2. Systemet svarer tilbage med hvorvidt markøren er låst til mærket eller fri.

UC21 Skift mellem låst og fri musemarkør

Stakeholders & Interests

Bruger: Ønsker at skifte mellem fri musemarkør og skjult markør låst til bevægelse af målemærkerne.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Markøren har skiftet tilstand mellem låst og fri.

Main Success Scenario

1. Brugeren trykker tasten, der er bundet til at skifte mellem låst og fri musemarkør. Fx højre museknap.
2. Markøren er fri og inden for rammen af vinduet.
3. Systemet skjuler markøren og låser den til mærket.

Extensions

1a. Et funktionskald aktiveres fra brugerprogrammet for at sætte markøren til at være låst.

1. Markøren er fri.

1a. Markøren er låst.

1 Intet sker.

2. Gå til pkt. 3.

1b. Et funktionskald aktiveres fra brugerprogrammet for at sætte markøren til at være fri.

1. Markøren er låst.

1a. Markøren er fri.

1 Intet sker.

2. Systemet frigører markøren fra mærket og viser denne hvis den før var skjult.

2a. Markøren er låst.

1. Systemet viser markøren halvvejs mellem de to målemærket, og frigører den fra mærket.

UC22 Info om mærket er låst eller frit

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at få at vide om mærket er låst til centrum af vinduet eller frit.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet.
2. Systemet svarer tilbage med hvorvidt mærket er låst til centrum eller frit.

UC23 Skift mellem låst og frit mærke

Stakeholders & Interests

Bruger: Ønsker at skifte mellem frit mærke eller mærke låst til centrum af vinduet.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Mærket har skiftet tilstand mellem låst og frit.

Main Success Scenario

1. Brugeren trykker tasten, der er bundet til at skifte mellem låst og frit målemærke. Fx Tab.
2. Mærket er frit.
3. Systemet flytter billeder og målemærker samlet, så målemærkerne er centreret omkring centrum af vinduet. Systemet låser mærket.

Extensions

1a. Et funktionskald aktiveres fra brugerprogrammet for at sætte mærket til at være låst.
    1. Mærket er frit.
        1a. Mærket er låst.
            1 Intet sker.
    2. Gå til pkt. 3.

1b. Et funktionskald aktiveres fra brugerprogrammet for at sætte mærket til at være frit.
    1. Mærket er låst.
        1a. Mærket er frit.
            1 Intet sker.
    2. Systemet sætter mærket til at være frit.

2a. Mærket er låst.
    1. Systemet sætter mærket til at være frit.

UC24 Tilsending af koordinater og primære input

Stakeholders & Interests

Bruger: Ønsker at konfigurere systemet til, hvorvidt det skal sende information til bruger-

programmet, når det primære input aktiveres, og når koordinaterne ændres.

Precondition

Systemet er startet op.

Postcondition

Systemet er konfigureret som ønsket.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at konfigurere systemet til at sende information til brugerprogrammet, når det primære input aktiveres, og når koordinaterne ændres. Evt. information om hvordan systemet kontakter brugerprogrammet medsendes.
2. Systemet vil i fremtiden informere brugeren om aktivering af det primære input og om skift af koordinater.

Extensions

1a. Et funktionskald aktiveres fra brugerprogrammet for at konfigurere systemet til ikke at sende information til brugerprogrammet, når det primære input aktiveres, og når koordinaterne ændres.
    1. Systemet vil i fremtiden ikke informere brugeren om aktivering af det primære input og om skift af koordinater.

UC25 Bevægelse af målemærkerne

Stakeholders & Interests

Bruger: Ønsker at bevæge målemærkerne.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Målemærkerne er flyttet.

Main Success Scenario

1. Grafikvinduet viser en model i modeltilstand.
2. Brugeren bevæger musen.
3. Musemarkøren er låst til det vandrende mærke.
4. Systemet ændrer det vandrende mærkes objektkoordinater baseret på musens bevægelse. Dvs. øger x-koordinaten hvis musen bevæges til højre.
5. Systemet beregner de tilsvarende billedkoordinater ud fra orienteringsparametre for kamera og billeder.
6. Systemet sender målemærkernes nye koordinater til brugerprogrammet.
7. Systemet flytter målemærkerne til deres nye billedkoordinater og viser de nye koordinater i grafikvinduet.
8. Systemet flytter billeder med tilhørende målemærker i forhold til hinanden, så de to målemærker ikke har ændret position i vinduet.

Extensions

1a. Grafikvinduet viser billede(er) i komparatortilstand.
    1. Intet billede er låst til sit målemærke.
        1a. Det ene billede er låst til sit målemærke.
            1. Gå til pkt. 1a.2, men ignoreres alle forsøg på at ændre billedkoordinater af målemærket i det låste billede. Billedkoordinater på målemærket i det frie

billede ændres som beskrevet.

2. Brugeren bevæger musen.
   1a. Brugeren trykker en tast eller knap, der er bundet til bevægelse af mærket i x-y planet. Fx piletasterne.
      1. Systemet ændrer målemærkernes billedkoordinater baseret på hvilken tast der blev trykket. Hvis tasten fx er bundet til at bevæge mærket til højre, vil x-koordinaten af begge målemærkers billedkoordinater øges med samme værdi. Størrelsen af denne værdi afhænger af hastigheden hvormed mærket bevæges.
      2. Gå til pkt. 6.
   1b. Brugeren trykker en tast eller knap, der er bundet til bevægelse af mærket i z-aksens retning. Fx musens hjul.
      1. Systemet ændrer x-koordinaten på de to sæt billedkoordinater i modsat retning. Hvis tasten fx er bundet til bevægelse ind i skærmen, reduceres x-koordinaten af målemærket i det venstre billede, mens x-koordinaten i det højre billede øges med en tilsvarende værdi. Størrelsen af denne værdi afhænger af hastigheden hvormed mærket bevæges.
      2. Gå til pkt. 6.
   1c. Et funktionskald aktiveres fra brugerprogrammet for at flytte det vandrende-mærke til nye objektkoordinater.
      1. Intet sker.
   1d. Et funktionskald aktiveres fra brugerprogrammet for at flytte målemærkerne til nye billedkoordinater baseret på det venstre billede.
      1. Systemet ændrer billedkoordinaterne af målemærket i det venstre billede til det ønskede. Billedkoordinaterne af målemærket i det højre billede ændres tilsvarende, så forskellen mellem de to sæt billedkoordinater forbliver uændret.
      2. Gå til pkt. 6.
   1e. Et funktionskald aktiveres fra brugerprogrammet for at flytte målemærkerne til nye billedkoordinater baseret på det højre billede.
      1. Systemet ændrer billedkoordinaterne af målemærket i det højre billede til det ønskede. Billedkoordinaterne af målemærket i det venstre billede ændres tilsvarende, så forskellen mellem de to sæt billedkoordinater forbliver uændret.
      2. Gå til pkt. 6.
3. Musemarkøren er låst til mærket.
   2a. Musemarkøren er fri, dvs. ikke låst til mærket.
      1. Musemarkøren flyttes, intet andet sker.
4. Systemet ændrer målemærkernes billedkoordinater baseret på musens bevægelse. Dvs. øger x-koordinaten for begge målemærker, hvis musen bevæges til højre.
5. Gå til pkt. 6.

2a. Brugeren trykker en tast eller knap, der er bundet til bevægelse af mærket. Fx piletasterne for bevægelse i x-y planet eller musens hjul for bevægelse i z-aksens retning.
   1. Systemet ændrer det vandrende mærkes objektkoordinater baseret på hvilken tast der blev trykket. Hvis tasten fx er bundet til at bevæge mærket til højre, vil x-koordinaten øges med en given værdi. Størrelsen af denne værdi afhænger af hastigheden hvormed mærket bevæges.
   2. Gå til pkt. 5.

2b. Et funktionskald aktiveres fra brugerprogrammet for at flytte det vandrende mærke til nye objektkoordinater.
   1. Systemet ændrer det vandrende mærkes objektkoordinater til positionen angivet i funktionskaldet.
   2. Gå til pkt. 5.

2c. Et funktionskald aktiveres fra brugerprogrammet for at flytte målemærkerne til nye billedkoordinater.
    1. Intet sker.
3a. Musemarkøren er fri, dvs. ikke låst til det vandrende mærke.
    1. Musemarkøren flyttes, intet andet sker.
6a. Brugerprogrammet ønsker ikke at få tilsendt koordinater.
    1. Gå til pkt. 7.
8a. Mærke er frit, dvs. ikke låst til centrum af skærmen.
    1. Systemet flytter billederne og de tilhørende målemærker lodret i forhold til hinanden, så de to målemærker ligger på fælles vandrette linie.
    2. Hvis et af målemærkerne er for tæt på kanten af vinduet flyttes billeder og målemærker samlet, så centrum af målemærkerne flyttes nærmere midten af vinduet.

UC26 Ændre hastigheden hvormed mærket bevæges

Stakeholders & Interests

Bruger: Ønsker at ændre den hastighed hvormed mærket bevæges.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Hastigheden af mærket er ændret.

Main Success Scenario

1. Brugeren trykker en tast, der er bundet til enten at øge eller sænke hastigheden mærket bevæges med.
2. Systemet ændrer den hastighed hvormed mærket bevæges.

UC27 Info om mærkets nuværende koordinater

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at modtage mærkets nuværende koordinater.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet.
2. Vinduet er i modeltilstand.
3. Systemet svarer tilbage med det vandrende mærkes nuværende objektkoordinater samt billedkoordinater på de to målemærker.

Extensions

2a. Vinduet er i komparatortilstand, to billeder vises.
    1. Systemet svarer tilbage med billedkoordinater på de to målemærker. Værdier for evt. medfølgende objektkoordinater er udefinerede.
2b. Vinduet er i komparatortilstand, et billede vises.
    1. Systemet svarer tilbage med målemærkets billedkoordinater i det viste billede.

Værdier for evt. medfølgende objektkoordinater og billedkoordinater i det andet billede er udefinerede.

2c. Ingen billeder vises.
    1. Systemet svarer tilbage med udefinerede koordinater.

UC28 Aktivere primære input til brugerprogrammet

Stakeholders & Interests

Bruger: Ønsker at sende information til brugerprogrammet om at det primære input er aktiveret.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugerprogrammet har modtaget information om, at det primære input er blevet aktiveret.

Main Success Scenario

1. Brugeren trykker eller slipper den tast, der er bundet som primære input. Fx venstre museknap.
2. Systemet sender information til brugerprogrammet om at det primære input enten er trykket eller sluppet.

Extensions

2a. Markøren er ikke låst til mærket.
    1. Intet sker.

2b. Brugerprogrammet ønsker ikke at få tilsendt information om hvorvidt det primære input er trykket eller sluppet.
    1. Intet sker.

UC29 Tilsending af sekundært input

Stakeholders & Interests

Bruger: Ønsker at konfigurere systemet til, hvorvidt det skal sende information til brugerprogrammet, når et sekundært input aktiveres.

Precondition

Systemet er startet op.

Postcondition

Systemet er konfigureret som ønsket.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at konfigurere systemet til at sende information til brugerprogrammet, når et sekundært input aktiveres. Evt. information om hvordan systemet kontakter brugerprogrammet medsendes.
2. Systemet vil i fremtiden informere brugeren om aktivering af sekundært input.

Extensions

1a. Et funktionskald aktiveres fra brugerprogrammet for at konfigurere systemet til ikke at sende information til brugerprogrammet, når et sekundært input aktiveres.
    1. Systemet vil i fremtiden ikke informere brugeren om aktivering af sekundært input.

UC30 Aktivere sekundære input til brugerprogrammet

Stakeholders & Interests

Bruger: Ønsker at sende information til brugerprogrammet om at et givent sekundært input er blevet aktiveret.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugerprogrammet har modtaget information om at et sekundært input er aktiveret, og hvilket input der er tale om.

Main Success Scenario

1. Brugeren trykker en tast, der er bundet som sekundært input. Fx ’1’.
2. Systemet sender information til brugerprogrammet om at et sekundært input er blevet trykket samt id på det sekundære input.

Extensions

2a. Brugerprogrammet ønsker ikke at få tilsendt information om et sekundært input er blevet aktiveret.
    1. Intet sker.

UC31 Forstørre eller formindske billeder

Stakeholders & Interests

Bruger: Ønsker at forstørre eller formindske de viste billeder eller model.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue. Et eller to billeder eller en model er åben.

Postcondition

Billederne er blevet forstørret eller formindsket.

Main Success Scenario

1. Brugeren trykker en tast, der er bundet til at forstørre eller formindske de viste billeder. Fx control+musehjul.
2. Systemet forstørrer eller formindsker de viste billeder.

UC32 Forstørre eller formindske målemærkerne

Stakeholders & Interests

Bruger: Ønsker at forstørre eller målemærkerne.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Målemærkerne er blevet forstørret eller formindsket.

Main Success Scenario

1. Brugeren trykker en tast, der er bundet til at forstørre eller formindske målemærkerne. Fx shift+musehjul.

2. Systemet forstørrer eller formindsker målemærkerne.

UC33 Info om et kamera

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at modtage de indre parameterindstillinger for et givent kamera.

Precondition

Programmet er startet op.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet. Id på det ønskede kamera medsendes.
2. Systemet svarer tilbage med information om det angivne kamera.

Extensions

2a. Kameraet med det angivne id findes ikke i databasen.
    1. Systemet svarer tilbage med information om at det ønskede kamera ikke findes.

UC34 Info om et billede

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at modtage informationer om et givent billede.

Precondition

Programmet er startet op.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet. Id på det ønskede billede medsendes.
2. Systemet svarer tilbage med information om det angivne billede.

Extensions

2a. Billedet med det angivne id findes ikke i databasen.
    1. Systemet svarer tilbage med information om at det ønskede billede ikke findes.

UC35 Info om en model

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at modtage informationer om en given model.

Precondition

Programmet er startet op.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet for at . Id på den ønskede model medsendes.
2. Systemet svarer tilbage med information om den angivne model.

Extensions

2a. Modellen med det angivne id findes ikke i databasen.
    1. Systemet svarer tilbage med information om at den ønskede model ikke findes.

UC36 Ændre data for et kamera

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at ændre de indre parameterindstillinger for et givent kamera, eller tilføje et nyt kamera til databasen.

Precondition

Programmet er startet op.

Postcondition

Databasen er blevet opdateret med den ønskede information.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet, med det nye data for det angivne kamera.
2. Kameraet med det angivne id findes i databasen.
3. Systemet ændrer data for det angivne kamera.

Extensions

2a. Kameraet med det angivne id findes ikke i databasen.
    1. Systemet tilføjer et nyt kamera til databasen.

UC37 Ændre data for et billede

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at ændre indstillingerne for et givent billede, eller tilføje et nyt billede til databasen.

Precondition

Programmet er startet op.

Postcondition

Databasen er blevet opdateret med den ønskede information.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet, med det nye data for det angivne billede.
2. Billedet med det angivne id findes i databasen.
3. Systemet ændrer data for det angivne billede.

Extensions

2a. Billedet med det angivne id findes ikke i databasen.
    1. Systemet tilføjer et nyt billede til databasen.

UC38 Ændre data for en model

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at ændre indstillingerne for en given model, eller tilføje en ny model til databasen.

Precondition

Programmet er startet op.

Postcondition

Databasen er blevet opdateret med den ønskede information.

Main Success Scenario

1. Et funktionskald aktiveres fra brugerprogrammet, med det nye data for den angivne model.
2. Modellen med det angivne id findes i databasen.
3. Systemet ændrer data for den angivne model.

Extensions

2a. Modellen med det angivne id findes ikke i databasen.
    1. Systemet tilføjer en ny model til databasen.

UC39 Søgning på kameraer

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at udføre en søgning i databasen over kameraer, og modtage id på alle kameraer, der opfylder søgningskriterierne.

Precondition

Programmet er startet op.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet. Søgningskriterierne medsendes.
2. Systemet svarer tilbage med en liste, der indeholder id på alle kameraer, der opfylder søgningskriterierne. Denne liste kan være tom.

UC40 Søgning på billeder

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at udføre en søgning i databasen over billeder, og modtage id på alle billeder, der opfylder søgningskriterierne.

Precondition

Programmet er startet op.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogram-

met. Søgningskriterierne medsendes.
2. Systemet svarer tilbage med en liste, der indeholder id på alle billeder, der opfylder søgningskriterierne. Denne liste kan være tom.

UC41 Søgning på modeller

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at udføre en søgning i databasen over modeller, og modtage id på alle modeller, der opfylder søgningskriterierne.

Precondition

Programmet er startet op.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet. Søgningskriterierne medsendes.
2. Systemet svarer tilbage med en liste, der indeholder id på alle modeller, der opfylder søgningskriterierne. Denne liste kan være tom.

UC42 Udvidelse – Ændring til UC6

Denne use case benyttes hvis udvidelsen 'Automatisk skift mellem modeller' implementeres (se afsnit 2.3.6). Dette er en udvidelse til 'UC6 - Skift af aktive vindue'.

Main Success Scenario

1-5 Fra UC6.
6. Intet yderligere sker.

Extensions

6a. Både det forladte og det nye vindue har åbne modeller i modeltilstand og disse to modeller viser det samme generelle område.
    1. Det vandrende mærkes position (objektkoordinater) i det forladte vindue læses, og det vandrende mærket flyttes til den samme position i det nye aktive vindue.
        1a. Den læste position ligger uden for modellen i det nye vindue.
            1. Intet yderligere sker.

UC43 Udvidelse – Bibeholde mærkeposition i anden model

Denne use case benyttes hvis udvidelsen ' Automatisk skift mellem modeller' implementeres (se afsnit 2.3.6).

Stakeholders & Interests

Bruger: Ønsker at skifte fra en model til en anden, der viser det samme generelle område, men ønsker at bibeholde det vandrende mærkes position i objektkoordinater.

Precondition

Det aktive vindue viser en model i modeltilstand.

Postcondition

Programmet har skiftet til en ny model og det vandrende mærkes position forbliver den samme.

Main Success Scenario

1. Brugeren vælger et andet vindue som aktivt (som beskrevet i UC6).
2. Det nye aktive vindue indeholder en model i modeltilstand, der viser det samme generelle område som den forladte model, og det vandrende mærkes position i den forladte model ligger inden for området af den nye model.
3. Det vandrende mærke i den nye model flyttes til samme position (objektkoordinater), som det vandrende mærke var på i den forladte model.

Extensions

1a. Brugeren trykker tasten, der er bundet til at åbne menu over modeller. Fx venstre museknap.
    1. Programmet gennemgår databasen for modeller, der viser det samme generelle område som den aktive model og som indeholder det vandrende mærkes nuværende position.
    2. Programmet åbner en menu, der viser en liste over de fundne modeller.
        2a. Ingen modeller blev fundet, der opfyldte de ovenstående krav.
            1. Brugeren informeres om, at ingen andre modeller dækker det nuværende område.
    3. Brugeren vælger en af modellerne i menuen.
        3a. Brugeren lukker menuen uden at vælge en model.
            1. Intet sker.
    4. Den gamle model lukkes og den nye model åbnes.
    5. Gå til pkt. 3.

1b. Brugeren flytter et af målemærkerne ud over kanten af sit billede.
    1. Programmet finde i databasen en anden model, der viser det samme generelle område som den aktive model, som er fotograferet fra omtrent den samme retning og afstand og som indeholder det vandrende mærkes nuværende position.
        1a. Programmet kan i databasen ikke finde en model, der opfylder de ovenstående krav.
            1. Intet sker.
    2. Den gamle model lukkes og den nye model åbnes.
    3. Gå til pkt. 3.

2a. Det nye aktive vindue indeholder ikke en model i modeltilstand, modellen viser ikke det samme generelle område som den forladte model, eller det vandrende mærkes tidligere position ligger ikke inden for den nye model.
    1. Det aktive vindue skiftes som beskrevet i UC6, men det vandrende mærke flyttes ikke yderligere.

UC44 Udvidelse – Valg af stregniveau

Denne use case benyttes hvis udvidelsen ’3D-streger i modellen’ implementeres (se afsnit 2.3.6).

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at vælge hvilket stregniveau skal være det aktive. Dvs. hvilket niveau efterfølgende streger skal tegnes på.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Det ønskede stregniveau er aktiveret.

Main Success Scenario

1. Et funktionskald, aktiveres fra brugerprogrammet, med info om hvilket stregniveau der

skal aktiveres.
2. Programmet sætter det ønskede niveau som aktivt.

Extensions

2a. Det ønskede stregniveau er ugyldigt.
1. Intet sker.

UC45 Udvidelse – Info om et stregniveau er synligt eller skjult

Denne use case benyttes hvis udvidelsen ’3D-streger i modellen’ implementeres (se afsnit 2.3.6).

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at få at vide om et givent stregniveau er synligt eller skjult.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Brugeren har modtaget den ønskede information.

Main Success Scenario

1. Et funktionskald, der beder om den ønskede information, aktiveres fra brugerprogrammet, med information om hvilket stregniveau, der er tale om.
2. Systemet svarer tilbage med hvorvidt det angivne stregniveau er synligt eller skjult.

Extensions

2a. Det angivne stregniveau er ugyldigt.
1. Systemet svarer tilbage med information om at det angivne stregniveau ikke findes.

UC46 Udvidelse – Vælge om et stregniveau er synligt eller skjult

Denne use case benyttes hvis udvidelsen ’3D-streger i modellen’ implementeres (se afsnit 2.3.6).

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at vælge om et givent stregniveau er synligt eller skjult.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Det ønskede stregniveau er synligt eller skjult.

Main Success Scenario

1. Et funktionskald, aktiveres fra brugerprogrammet, med info om hvilket stregniveau der er tale om og hvorvidt dette niveau skal gøres synligt eller skal skjules.
2. Programmet gør det ønskede niveau synligt eller skjult efter ønske.

Extensions

2a. Det ønskede stregniveau er ugyldigt.
1. Intet sker.

UC47 Udvidelse – Valg af stregtykkelse

Denne use case benyttes hvis udvidelsen ’3D-streger i modellen’ implementeres (se afsnit 2.3.6).

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at vælge hvor tyk en streg, der skal tegnes.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Pennen, der tegner stregerne har nu den ønskede tykkelse.

Main Success Scenario

1. Et funktionskald, aktiveres fra brugerprogrammet, med info om hvor tyk en streg, der skal tegnes.
2. Programmet ændrer pennens tykkelse til det ønskede niveau.

Extensions

2a. Den ønskede stregtykkelse er ugyldig.
    1. Intet sker.

UC48 Udvidelse – Valg af stregfarve

Denne use case benyttes hvis udvidelsen ’3D-streger i modellen’ implementeres (se afsnit 2.3.6).

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at vælge hvilken farve, der skal tegnes med.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Pennen, der tegner stregerne benytter nu den ønskede farve.

Main Success Scenario

1. Et funktionskald, aktiveres fra brugerprogrammet, med info om hvilken farve, der skal benyttes.
2. Programmet ændrer pennens farve til den ønskede.

Extensions

2a. Den ønskede farve er ugyldig.
    1. Intet sker.

UC49 Udvidelse – Flytning af pennen

Denne use case benyttes hvis udvidelsen ’3D-streger i modellen’ implementeres (se afsnit 2.3.6).

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at flytte pennen til en anden position, uden at tegne en streg.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Pennen er flyttet til den ønskede position.

Main Success Scenario

1. Et funktionskald, aktiveres fra brugerprogrammet, med info om den nye position (i objektkoordinater) pennen skal have.
2. Programmet flytter pennen til den ønskede position.

Extensions

2a. Det aktive vindue viser ikke en model i modeltilstand.
    1. Intet sker.

UC50 Udvidelse – Tegning af en streg

Denne use case benyttes hvis udvidelsen ’3D-streger i modellen’ implementeres (se afsnit 2.3.6).

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at tegne en streg på modellen.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue. Det stregniveau som stregen ønskes tegnet på er det aktive.

Postcondition

Pennen flyttes til den nye position mens en streg tegnes.

Main Success Scenario

1. Et funktionskald, aktiveres fra brugerprogrammet, med info om den position (i objektkoordinater) hvortil stregen skal tegnes.
2. Programmet flytter pennen til den ønskede position, mens en streg tegnes fra den gamle position til den nye. Stregen har den nuværende valgte tykkelse og farve. Hvis det aktive stregniveau er skjult er stregen usynlig indtil niveauet gøres synligt.

Extensions

2a. Det aktive vindue viser ikke en model i modeltilstand.
    1. Intet sker.

UC51 Udvidelse – Valg af symbol

Denne use case benyttes hvis udvidelsen ’3D-streger i modellen’ implementeres (se afsnit 2.3.6).

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at vælge hvilket symbol, der skal benyttes som punkttegn.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue.

Postcondition

Det ønskede symbol er nu det aktive, og fremtidige punktplaceringer vil tegne dette symbol.

Main Success Scenario

1. Et funktionskald, aktiveres fra brugerprogrammet, med info om hvilket symbol, der skal benyttes.

2. Programmet ændrer det aktive symbol til det ønskede.

Extensions

2a. Det ønskede symbol er ugyldig.
    1. Intet sker.

UC52 Udvidelse – Tegning af punkttegn

Denne use case benyttes hvis udvidelsen ’3D-streger i modellen’ implementeres (se afsnit 2.3.6).

Stakeholders & Interests

Bruger: Ønsker via brugerprogrammet at tegne et punkt på modellen.

Precondition

Det vindue som handlingen ønskes udført på er åbent og er det aktive vindue. Det stregniveau som punktet ønskes tegnet på er det aktive.

Postcondition

Det aktive symbol tegnes som punkt på pennens nuværende position.

Main Success Scenario

1. Et funktionskald, aktiveres fra brugerprogrammet.
2. Programmet tegner det aktive symbol som punkttegn på pennens nuværende position. Symbolet har den valgte farve.

Extensions

2a. Det aktive vindue viser ikke en model i modeltilstand.
    1. Intet sker.

E.2 KLASSESTRUKTURER

Dette appendiks viser den implementeringsuafhængige designstruktur over klasserne i SIDM.

Nedenstående er en liste over de typer, der benyttes som member variable, argumenter eller returtyper i de beskrevne klasser.

- **int:** 32 bit heltal.
- **float:** 32 bit decimaltal (floating-point).
- **double:** 64 bit decimaltal.
- **boolean:** Kan antage værdierne true (1) eller false (0).
- **xList:** En liste af typen x. Fx intList er en liste af int. En liste er et array, der kan have et vilkårligt antal elementer.
- **xRef:** En reference til typen x. Fx intRef er en reference til en int. En reference som funktionsargument benyttes til at returnere informationer til den kaldende funktion.
- **x[n]:** Et array af typen x, der har plads til n elementer. Fx int[2] er et int array med plads til to elementer.
- **String:** En tekststreng af vilkårlig længde.
- **Color:** En variabel, der gemmer en given farve. Kan evt. være en float[3].
- **Delegate:** En funktionspointer. En delegate peger på en funktion, og denne funktion kan eksekveres gennem delegate-variablen.
- **DatabaseIdentifier:** En variabel, der unikt identificerer en database. Det kan fx var et id eller navn.
- **DatabaseConnection:** En åben forbindelse til en database.
- **«struct»:** En gruppering af flere forskellige typer. Kan tolkes som en simpel klasse uden funktioner, hvor alle member variable er public.
- **«enum»:** En enumeration er en type, der kun kan have et begrænset antal forskellige navngivne værdier. En boolean kan tolkes som en enum, der kan have værdierne 'true' og 'false'.

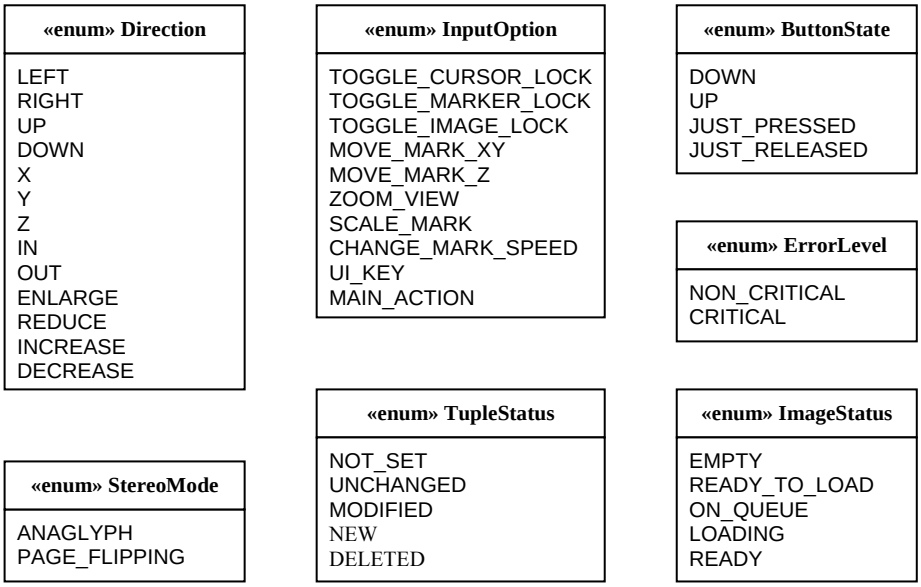
På de efterfølgende sidder ses strukturen af klasser, strukturer (*structs*) og enumerationer i SIDM. Variabler og funktioner i disse klasser vil ikke beskrives nærmere i dette appendiks. Nogle af disse er gennemgået i selve rapporten og andre følger logisk ud fra variabel- eller funktionsnavnet.

Generelt vil alle use cases, der indeholder en linie, som starter med "*Et funktions kald aktiveres fra brugerprogrammet…*" repræsenteres i form af en enkelt funktion i *SidmInterface*. Fx bliver 'UC14 - *Indlæsning af billeder, der ikke tilhører en model*' repræsenteret af funktionen *loadImages*. Hvis funktionaliteten påvirker databasen, vil en tilsvarende funktion findes i *Database*-klassen,

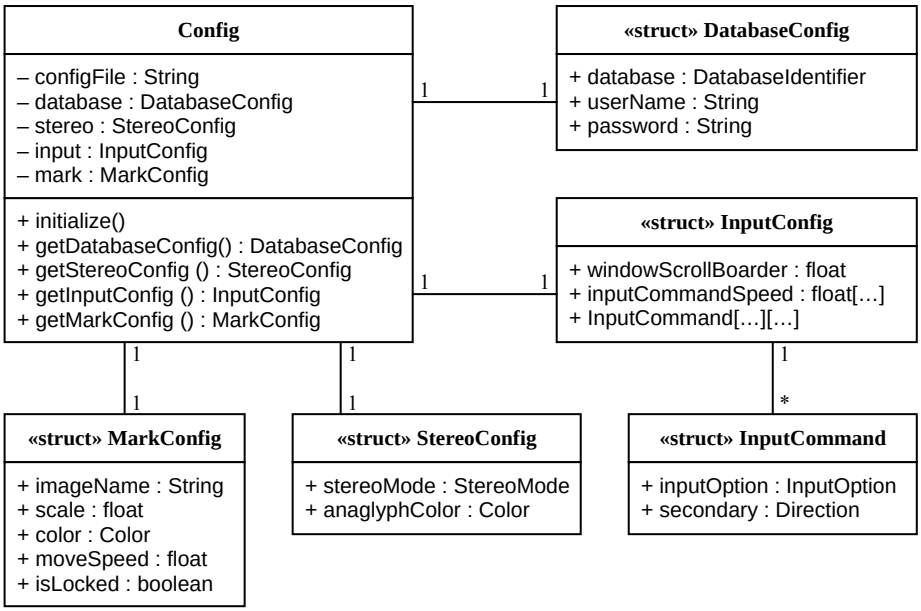
ellers vil den samme funktion findes i *SidmApplication*. Hvis funktionaliteten påvirker elementer i et grafikvindue, vil funktionen også eksistere i *Scene*, og evt. også i *Mark* eller *Model*, afhængigt af hvad funktionaliteten skal påvirke.

Ønskes en fyldestgørende beskrivelse af samtlige klasser, funktioner, variable og typer i SIDM henvises til dokumentationen over implementeringen, som kan findes på den medfølgende CD (se afsnit D.1).

Figurteksterne på de efterfølgende sider, beskriver hvor de illustrerede klasser og typer hører til i SIDM.

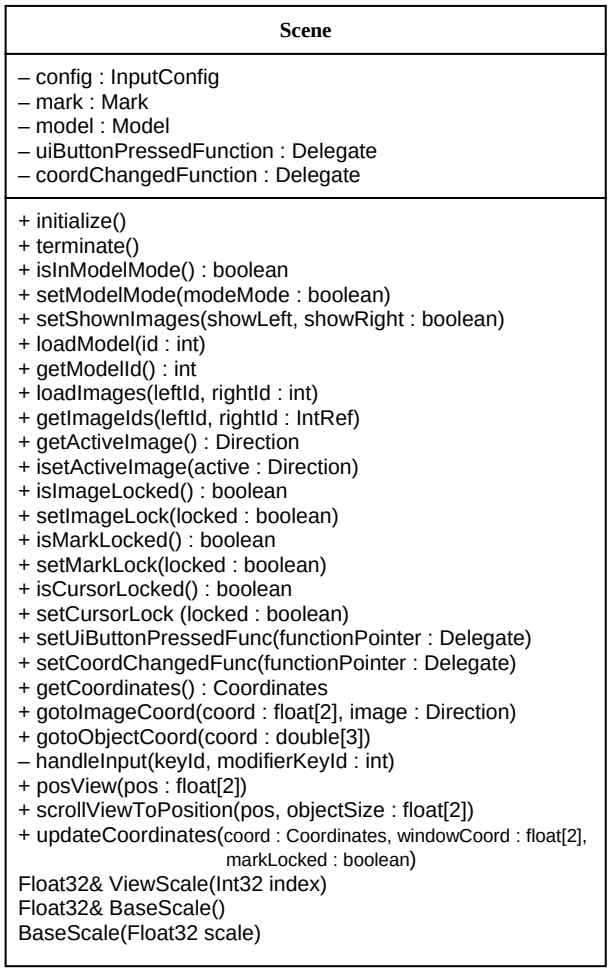


Figur A - 16: Oversigt over enumerationer i SIDM.

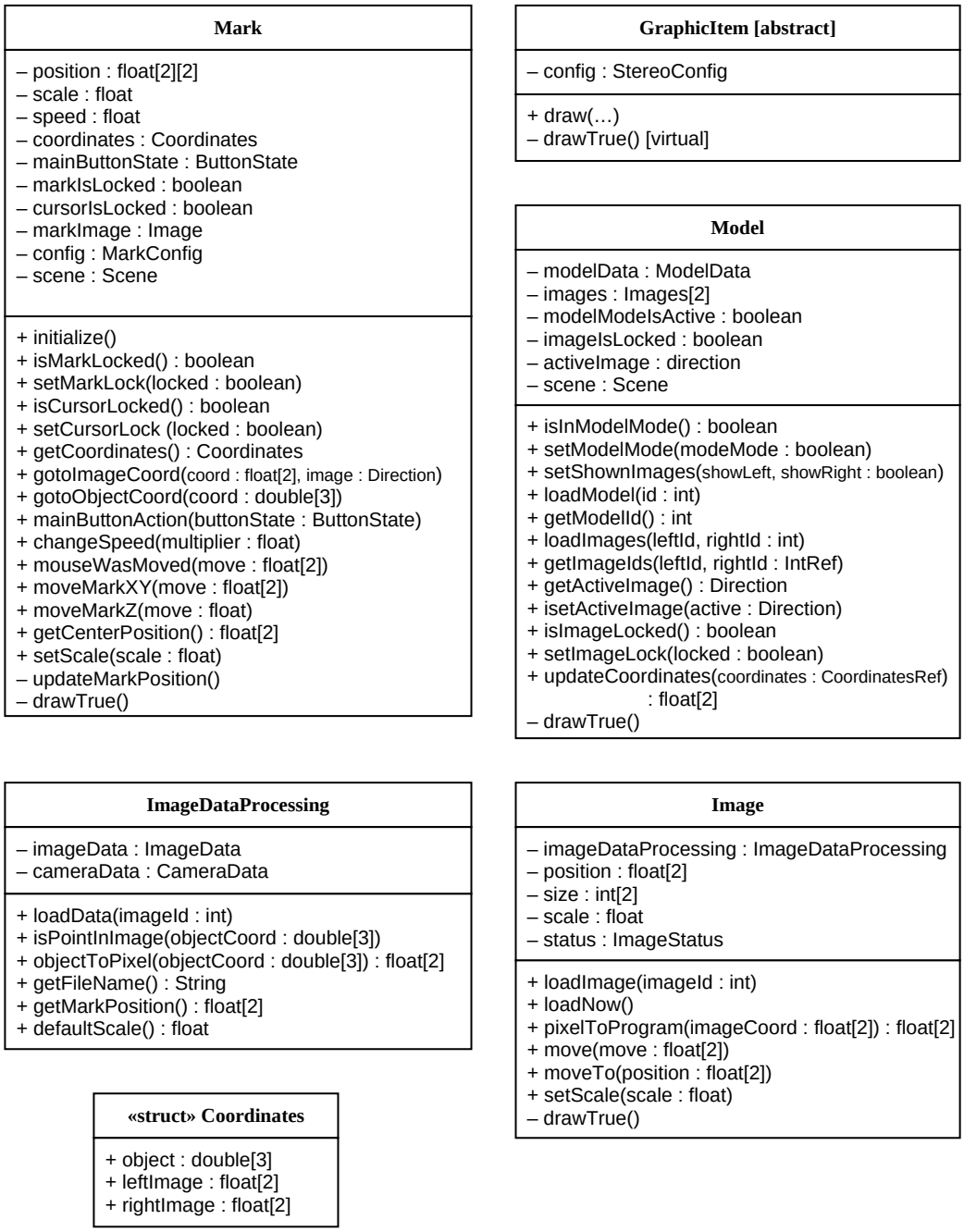


Figur A - 17: Struktur af *Config* klassen og tilhørende structs i konfigurationsmodulet.

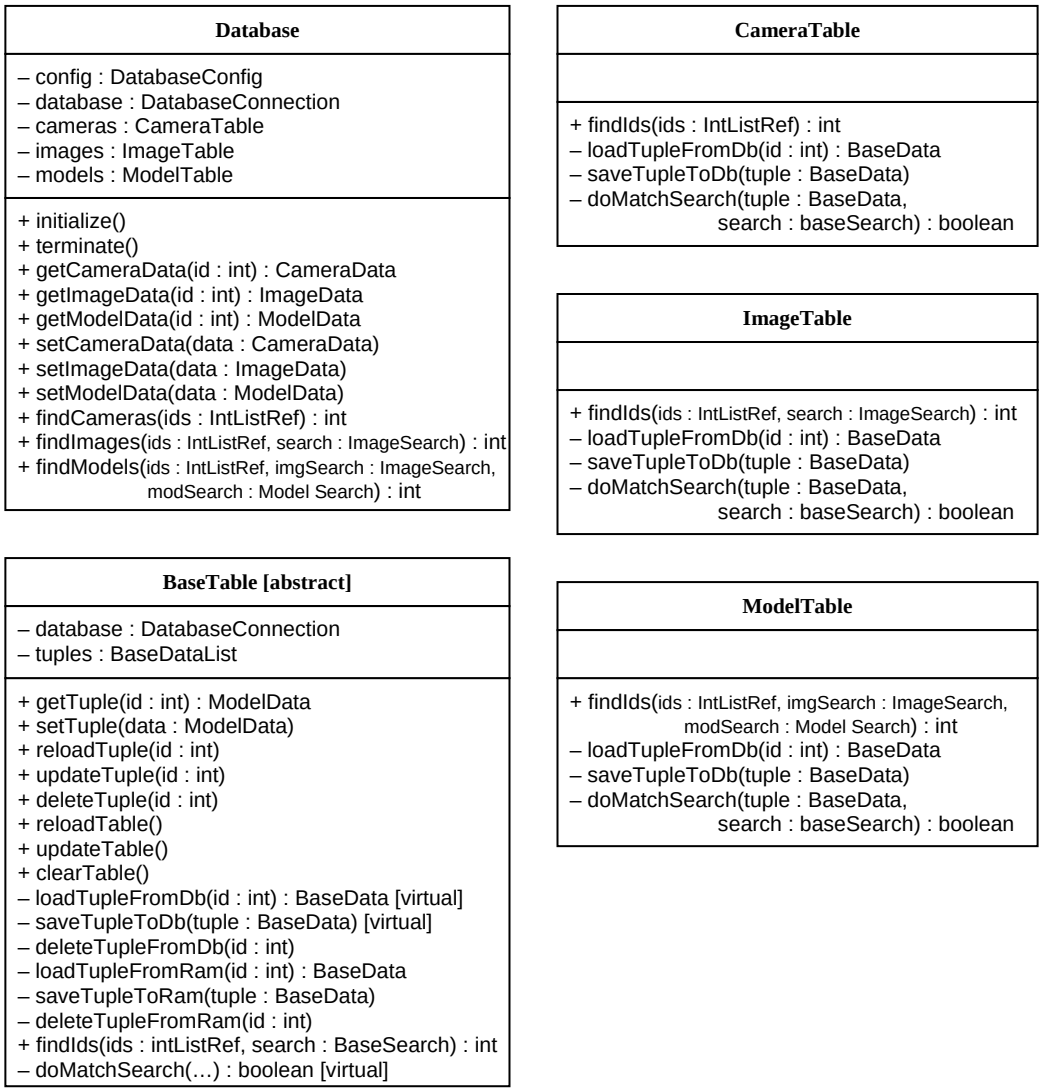
Figur A - 18: Klassestruktur over klasser i interface-, kontrol-, arbejds- og fejlbehandlingsmodulet.



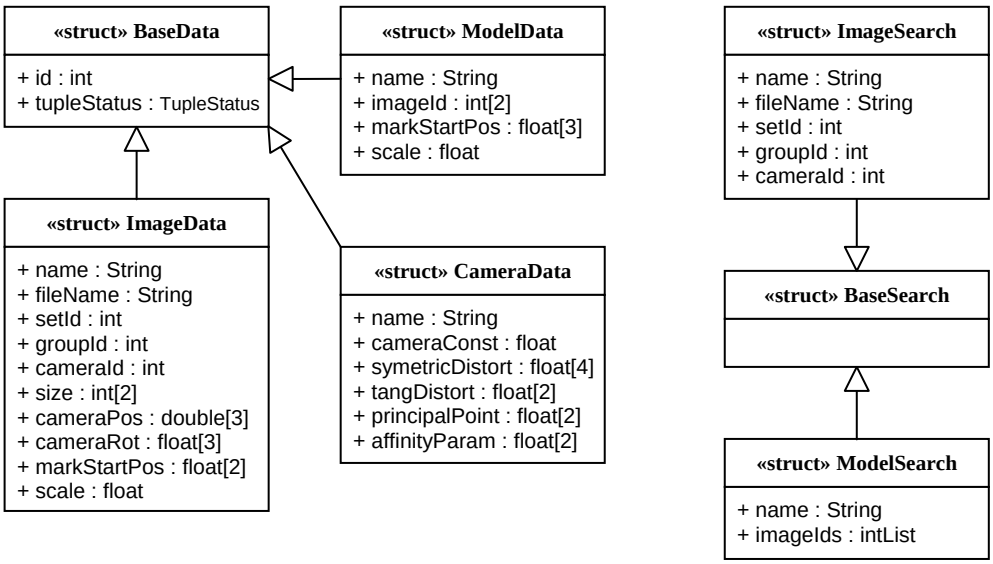
Figur A - 19: Oversigt over interfaceklassen til vinduesmodulet.



Figur A - 20: Oversigt over de resterende klasser i vinduesmodulet.



Figur A - 21: Oversigt over samtlige klasser i databasemodulet.



Figur A - 22: Oversigt over strukturer, der primært benyttes af databasemodulet.

E.3 DATABASESTRUKTUR

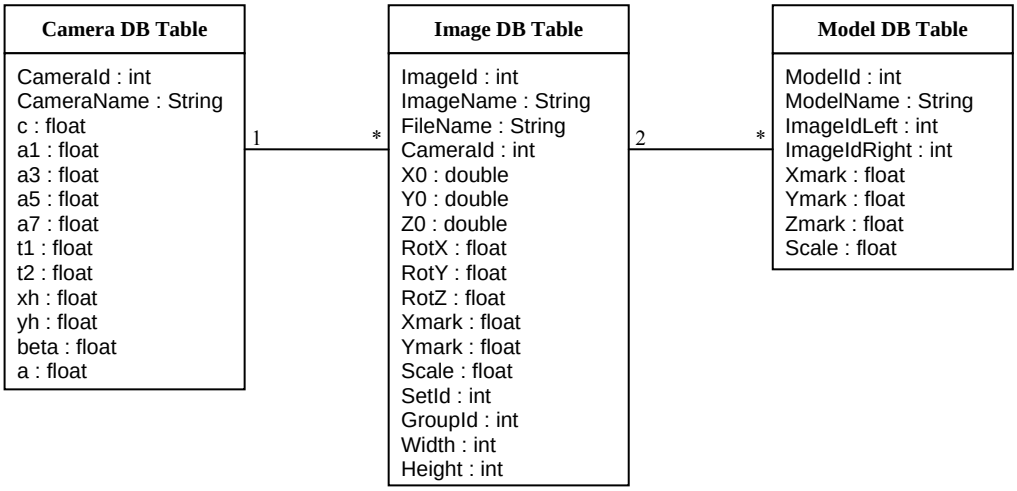
Figur A - 23 viser en oversigt over den database, der benyttes af SIDM. Tabellerne er bygget op omkring de krævede dataværdier specificeret i kravspecifikationen.

Alle tre tabeller indeholder id og navn på det angivne element. Ud over dette indeholder tabellerne følgende data:

Kameraer (*Cameras*): Kamerakonstant (c), radialsymmetriske fortegnelseskonstanter (*a1*, *a3*, *a5*, *a7*), tangentielle fortegnelseskonstanter (*t1*, *t2*), hovedpunktets position (*xh*, *yh*) og affine transformationskonstanter (*beta*, *a*).

Billeder (*Images*): Navn på billedfil (*FileName*), kameraets id (*CameraId*), kameraets projektionscenter (*X0*, *Y0*, *Z0*), kameraets rotation (*RotX*, *RotY*, *RotZ*), målemærkets startposition (*Xmark*, *Ymark*), billedes startzoomfaktor (*Scale*), id'er til at definerer billedets forbindelse til andre billeder (*SetId*, *GroupId*) og størrelse af den viste del af billedet (*Width*, *Height*). De fire sidstnævnte variable er ikke bestemt ud fra kravspecifikationen, men er krævet for at implementere udvidelsen "automatisk skift mellem modeller".

Modeller (*Models*): Id på de to billeder, der udgør modellen (*ImageIdLeft*, *ImageIdRight*), det vandrende mærkes startposition (*Xmark*, *Ymark, Zmark*) og billedernes startzoomfaktor (*Scale*).



Figur A - 23: Oversigt over databasen brugt i SIDM.

F Test

Den funktionelle test af det endelige programmodul er baseret på use cases. I dette appendiks beskrives samtlige test over samtlige use cases. Én test vil blive udført per use case, plus en ekstra test for hver alternativ gren (*Extension*) i use casen. I enkelte tilfælde vil flere eller færre test blive udført.

For at begrænse teksten i nedenstående beskrivelser, vil nogle linier starte med teksten ”Som første, men …”. Dette betyder, at testen udføres som den første test i use casen, men med den noterede ændring. På tilsvarende måde kan forventet resultat henvende til første test i use casen.

UC1 - Opstart af programmodul	**Forventet resultat**	**Reelt resultat**
Initialize() kaldes, SIDM er ikke startet op.	SIDM starter op.	Som forventet.
Som første, men SIDM er startet op.	Intet sker.	Som forventet.

UC2 - Nedlukning af programmodul	**Forventet resultat**	**Reelt resultat**
Terminate() kaldes, SIDM er startet op.	SIDM lukker begge tråde korrekt ned.	Som forventet.
Som første, men SIDM er ikke startet op.	Intet sker.	Som forventet.

UC3 - Åbning af vindue	**Forventet resultat**	**Reelt resultat**
OpenWindow() kaldes.	Et nyt vindue åbnes, id på dette vindue returneres.	Som forventet.

UC4 - Lukning af vindue	**Forventet resultat**	**Reelt resultat**
CloseWindow(1) kaldes. Vinduet med id 1 er åbent.	Vindue med id 1 lukkes.	Som forventet.
Som første, men vinduet med id 1 er ikke åbent.	Intet sker	Som forventet.
Brugeren trykker på krydset i hjørnet af et vinduet.	Vinduet lukkes	Som forventet.
Brugeren trykker ALT-F4 mens vinduet er aktivt.	Vinduet lukkes	Som forventet.

UC5 - Tilsending af information om vinduesskift	**Forventet resultat**	**Reelt resultat**
SetWindowChangedFunc(pFunc) kaldes, hvor pFunc er en pointer til en statisk eller global funktion.	SIDM vil for fremtiden sende information om vinduesskift til denne funktion.	Som forventet.
SetWindowChangedFunc(null) kaldes.	SIDM sender ikke mere information om vinduesskift.	Som forventet.

UC6 - Skift af aktive vindue	**Forventet resultat**	**Reelt resultat**
Brugeren trykker en tast eller knap mens musemarkøren befinder sig inden for et ikke aktivt vindue. WindowChangedFunc er sat (se UC5).	Den angivne funktion i brugerprogrammet kaldes. Id på det aktiverede vindue medsendes.	Som forventet.
Som første, men vinduet er allerede aktivt.	Intet sker.	Som forventet.
Som første, men WindowChangedFunc er ikke sat.	Intet sker.	Som forventet.
SwitchToWindow(1) kaldes. Vinduet med id 1 er åbent men ikke aktivt og WindowChangedFunc er sat.	Som første.	Som forventet.
SwitchToWindow(1) kaldes. Vinduet med id 1 er enten ikke åbent eller er allerede aktivt, eller WindowChangedFunc er ikke sat.	Intet sker.	Som forventet.

UC7 - Info om hvilket vindue der er aktivt	**Forventet resultat**	**Reelt resultat**
GetActiveWindow() kaldes. Et vindue er aktivt.	Id på det aktive vindue returneres.	Som forventet.

Som første, men intet vindue er aktivt.	-1 returneres.	Som forventet.

UC8 - Info om hvilke vinduer der er åbne	**Forventet resultat**	**Reelt resultat**
GetAllWindows(idList) kaldes, hvor ids er en tom liste.	Listen fyldes med id på alle åbne vinduer.	Som forventet.

UC9 - Info om model- eller komparatortilstand er aktiv	**Forventet resultat**	**Reelt resultat**
IsInModelMode () kaldes. Det aktive vindue er i modeltilstand.	Funktionen returnerer true.	Som forventet.
Som første, men det aktive vindue er i komparatortilstand.	Funktionen returnerer false.	Som forventet.

UC10 - Skift mellem model og komparatortilstand	**Forventet resultat**	**Reelt resultat**
SetModelMode(true) kaldes. Det aktive vindue er i komparatortilstand men har en model med kendte orienteringsparametre åben.	Det aktive vindue skifter til modeltilstand.	Som forventet.
Som første, men det aktive vindue har ikke en model med kendte orienteringsparametre åben.	Intet sker.	Som forventet.
Som første, men det aktive vindue er i modeltilstand.	Intet sker.	Som forventet.
SetModelMode(false) kaldes. Det aktive vindue er i modeltilstand.	Det aktive vindue skifter til komparatortilstand.	Som forventet.
SetModelMode(false) kaldes. Det aktive vindue er i komparatortilstand.	Intet sker.	Som forventet.

UC11 - Vælg hvorvidt et billede er synligt eller skjult	**Forventet resultat**	**Reelt resultat**
SetShownImages(true, false) kaldes. To billeder er åbne i det aktive vindue.	Det venstre billede vises mens det højre billede skjules.	Som forventet.

UC12 - Indlæsning af en model	**Forventet resultat**	**Reelt resultat**
LoadModel(1) kaldes. Et vindue er aktivt. Modellen med id 1 eksisterer i databasen og alle orienteringsparametre for de to billeder og det benyttede kamera er kendte. Ingen problemer opstår under indlæsning af billeder.	Eksisterende billeder fjernes fra vinduet. Vinduet åbner de to billeder tilhørende den ønskede model, skifter til modeltilstand og flytter det vandrende mærke til positionen angivet i modeldata.	Som forventet.
Som første, men der opstår problemer under indlæsning.	Eksisterende billeder fjernes fra vinduet, og det opståede problem beskrives i vinduet.	Som forventet.
Som første, men modellen findes ikke i databasen.	Eksisterende billeder fjernes fra vinduet. information om at modeldata mangler skrives i vinduet.	Som forventet.
Som første, men orienteringsparametre mangler for et billede eller kamera.	Eksisterende billeder fjernes fra vinduet. Vinduet åbner de to billeder tilhørende den ønskede model, skifter til komparatortilstand, flytter de to målemærket til positionen angivet i billeddata, og beskriver den opståede fejl i vinduet.	Som forventet.

UC13 - Info om hvilken model der vises	**Forventet resultat**	**Reelt resultat**
GetModelId() kaldes. Det aktive vindue har en åben model.	Id på den åbne model returneres.	Som forventet.
Som første, men det aktive vindue har ingen åben model.	-1 returneres.	Som forventet.

UC14 - Indlæsning af billeder, der ikke tilhører en model	**Forventet resultat**	**Reelt resultat**
LoadImages(1, 2) kaldes. Et vindue er aktivt. Billeder med id 1 og id 2 eksisterer i databasen, og ingen problemer opstår	Eksisterende billeder fjernes fra vinduet. Vinduet åbner de to billeder, skifter til komparator-tilstand og flytter de to målemærker til	Som forventet.

under indlæsning af disse.	positionen angivet i data for billederne.	
Som første, men et af de to billeder kan ikke indlæses, enten pga. manglende data i databasen eller pga. problemer ved indlæsning.	Eksisterende billeder fjernes fra vinduet. Vinduet åbner det billede der fandtes i databasen og kunne indlæses, skifter til komparator-tilstand, flytter det tilhørende målemærke til positionen angivet i billeddata og informerer brugeren om manglende billeddata eller problemer ved indlæsning af det andet billede.	Som forventet.
Som første, men ingen af de to billeder kan indlæses korrekt, enten pga. manglende data i databasen eller pga. problemer ved indlæsning.	Eksisterende billeder fjernes fra vinduet. Vinduet skifter til komparator-tilstand og informerer brugeren om manglende billeddata eller problemer ved indlæsning af billeder.	Som forventet.

UC15 - Info om hvilke billeder, der vises	**Forventet resultat**	**Reelt resultat**
GetImageIds(leftId, rightId) kaldes, hvor leftId og rightId er referencer til variable. To billeder er indlæst i det aktive vindue.	Id på de to billeder indlæses i referencerne.	Som forventet.
Som første, men kun et billede er indlæst.	Id på det aktive billede indlæses i den tilhørende reference. Den anden reference får tildelt værdien -1.	Som forventet.
Som første, men ingen billeder er indlæst.	De to referencer får tildelt værdien -1.	Som forventet.

UC16 - Info om hvilket billede er det aktive	**Forventet resultat**	**Reelt resultat**
GetActiveImage() kaldes. Det højre billede er det aktive billede.	Funktionen returnerer SIDM_RIGHT.	Som forventet.
Som første, men det venstre billede er det aktive billede.	Funktionen returnerer SIDM_LEFT.	Som forventet.

UC17 - Skift mellem hvilket billede der er det aktive	**Forventet resultat**	**Reelt resultat**
SetActiveImage(SIDM_LEFT) kaldes. Det højre billede er det aktive billede.	Det venstre billede sættes til at være det aktive billede.	Som forventet.
Som første, men det venstre billede er det aktive billede.	Intet sker.	Som forventet.

UC18 - Info om et billede er låst eller frit	**Forventet resultat**	**Reelt resultat**
IsImageLocked() kaldes. Det aktive billede er låst.	Funktionen returnerer true.	Som forventet.
Som første, men det aktive billede er frit.	Funktionen returnerer false.	Som forventet.

UC19 - Skift mellem låst og frit billede	**Forventet resultat**	**Reelt resultat**
Brugeren trykker en tast, der er bundet til at skifte mellem låst og frit billede. Det aktive billede er frit.	Det aktive billede låses.	Som forventet.
Som første, men det aktive billede er låst.	Det aktive billede sættes frit.	Som forventet.
SetImageLock(true) kaldes. Den aktive billede er frit.	Det aktive billede låses.	Som forventet.
SetImageLock(true) kaldes. Den aktive billede er låst.	Intet sker.	Som forventet.
SetImageLock(false) kaldes. Den aktive billede er låst.	Det aktive billede sættes frit.	Som forventet.

UC20 - Info om markøren er låst eller fri	**Forventet resultat**	**Reelt resultat**
IsCursorLocked() kaldes. Markøren i det aktive vindue er låst.	Funktionen returnerer true.	Som forventet.
Som første, men markøren er fri.	Funktionen returnerer false.	Som forventet.

UC21 - Skift mellem låst og fri musemarkør	**Forventet resultat**	**Reelt resultat**
Brugeren trykker en tast, der er bundet til at skifte mellem låst og fri markør. Markøren er fri.	Markøren låses.	Som forventet.
Som første, men markøren er fri.	Markøren sættes fri.	Som forventet.

SetCursorLock(true) kaldes. Markøren er fri.	Markøren låses.	Som forventet.
SetCursorLock(true) kaldes. Markøren er låst.	Intet sker.	Som forventet.
SetCursorLock(false) kaldes. Markøren er låst.	Markøren sættes fri.	Som forventet.

UC22 - Info om mærket er låst eller frit	**Forventet resultat**	**Reelt resultat**
IsMarkLocked() kaldes. Mærket i det aktive vindue er låst.	Funktionen returnerer true.	Som forventet.
Som første, men mærket er frit.	Funktionen returnerer false.	Som forventet.

UC23 - Skift mellem låst og frit mærke	**Forventet resultat**	**Reelt resultat**
Brugeren trykker en tast, der er bundet til at skifte mellem låst og frit mærke. Markøren er fri.	Mærket låses.	Som forventet.
Som første, men mærket er frit.	Mærket sættes frit.	Som forventet.
SetMarkLock(true) kaldes. Mærket er frit.	Mærket låses.	Som forventet.
SetMarkLock(true) kaldes. Mærket er låst.	Intet sker.	Som forventet.
SetMarkLock(false) kaldes. Mærket er låst.	Mærket sættes frit.	Som forventet.

UC24 - Tilsending af koordinater og primære input	**Forventet resultat**	**Reelt resultat**
SetCoordChangedFunc(pFunc) kaldes, hvor pFunc er en pointer til en statisk eller global funktion.	SIDM vil for fremtiden sende information til denne funktion, når koordinater skifter eller primær input aktiveres.	Som forventet.
SetCoordChangedFunc (null) kaldes.	SIDM vil for fremtiden ikke sende information, når koordinater skifter eller primær input aktiveres.	Som forventet.

UC25 - Bevægelse af målemærkerne	**Forventet resultat**	**Reelt resultat**
Brugeren bevæger musen til højre. Det aktive vindue er i modeltilstand, markøren er låst, mærket er låst og CoordChangedFunc er sat (se UC24)	SIDM øger x-objektkoordinaten af det vandrende mærke, beregner nye billedkoordinater ud fra denne nye position, skriver de nye koordinater i tekstboksen og lytter de to billeder i forhold til skærm og målemærker, så målemærkerne står rigtigt. Den angivne funktion i brugerprogrammet kaldes og de nye koordinater medsendes.	Som forventet.
Som øverste, men musen bevæges nedad.	Som øverste, men y-objektkoordinaten af det vandrende mærke sænkes og x-objektkoordinaten forbliver uændret.	Som forventet.
Som øverste, men markøren er fri.	Intet sker.	Som forventet.
Søm øverste, men mærket er frit.	Som øverste, men målemærkerne flyttes i stedet for billederne. Det højre billede kan flyttes en smule op eller ned for at fjerne vertikale parallakser. Hvis den nye position placerer et at målemærkerne uden for den definerede kantzone i billedet, flyttes begge billeder og målemærker nærmere centrum.	Som forventet.
Som øverste, men CoordChangedFunc er ikke sat.	Som øverste, men ingen information sendes til brugerprogrammet.	Som forventet.
Som øverste, men det aktive vindue er i komparatortilstand, og det aktive billede er frit.	Som øverste, men objektkoordinaterne modificeres ikke. I stedet øges x-billedkoordinaten i begge billeder.	Som forventet.
Som øverste, men det aktive vindue er i komparatortilstand, det venstre billede er det aktive billede og det aktive billede er låst.	Som øverste, men objektkoordinaterne modificeres ikke. I stedet øges x-billedkoordinaten i det højre billede.	Som forventet.
Som øverste, men musen bevæges ikke og brugeren trykker en tast, der er bundet til at bevæge mærket til højre.	Som øverste.	Som forventet.
Som øverste, men musen bevæges ikke og brugeren trykker en tast, der er bundet til at bevæge mærket ind i skærmen.	Som øverste, men z-objektkoordinaten af det vandrende mærke øges og x-objektkoordinaten forbliver uændret.	Som forventet.
Som øverste, men musen bevæges ikke, brugeren trykker en tast, der er bundet til at bevæge mærket ind i skærmen, det aktive	Som øverste, men objektkoordinaterne modificeres ikke. I stedet øges x-billedkoordinaten af det venstre billede og y-	Som forventet.

vindue er i komparatortilstand, og det aktive billede er frit.	koordinaten af det højre billede sænkes tilsvarende.	
Som øverste, men musen bevæges ikke, og GotoObjectCoord(100, 100, 100) kaldes.	Som øverste, men objektkoordinaterne ændres til det angivne (100, 100, 100).	Som forventet.
Som øverste, men musen bevæges ikke, GotoObjectCoord(100, 100, 100) kaldes, og det aktive vindue er i komparatortisltand.	Intet sker.	Som forventet.
Som øverste, men musen bevæges ikke og GotoImageCoord(100, 100, SIDM_LEFT) kaldes.	Intet sker.	Som forventet.
Som øverste, men musen bevæges ikke, GotoImageCoord(100, 100, SIDM_LEFT) kaldes, det aktive vindue er i komparatortisltand og det aktive billede er låst.	Som øverste, men objektkoordinaterne modificeres ikke. I stedet sættes billedkoordinaterne i det venstre billede til (100, 100). Billedkoordinaterne i det højre billede ændres tilsvarende.	Som forventet.
Som øverste, men musen bevæges ikke, GotoImageCoord(100, 100, SIDM_LEFT) kaldes, det aktive vindue er i komparatortisltand, det aktive billede er det venstre og det aktive billede er låst.	Intet sker.	Som forventet.

UC26 - Ændre hastigheden hvormed mærket bevæges	**Forventet resultat**	**Reelt resultat**
Brugeren trykker en tast, der er bundet til at sænke den hastighed, hvormed mærket bevæges.	Mærkets hastighed sænkes.	Som forventet.
Brugeren trykker en tast, der er bundet til at øge den hastighed, hvormed mærket bevæges.	Mærkets hastighed øges.	Som forventet.

UC27 - Info om mærkets nuværende koordinater	**Forventet resultat**	**Reelt resultat**
GetCoordinates() kaldes. Det aktive vindue er i modeltilstand.	Det vandrende mærkes nuværende objektkoordinater samt billed-koordinater for de to målemærker returneres i en struct.	Som forventet.
Som første, men det aktive vindue er i komparatortilstand med to billeder åbne.	Billedkoordinater på det to målemærker returneres i en struct. De resterende værdier i denne er udefinerede.	Som forventet. De udefinerede variable i structen er sat til de sidste kendte koordinater.
Som første, men det aktive vindue er i komparatortilstand med kun et billede åbent.	Billedkoordinater på målemærket i det åbne billede returneres i en struct. De resterende værdier i denne er udefinerede.	Som forventet. De udefinerede variable i structen er sat til de sidste kendte koordinater.
Som første, men det aktive vindue har ingen åbne billeder.	En struct med udefinerede værdier returneres.	Som forventet. De udefinerede variable i structen er sat til de sidste kendte koordinater.

UC28 - Aktivere primære input til brugerprogrammet	**Forventet resultat**	**Reelt resultat**
Tasten, der er bundet til primære input, trykkes ned mens markøren er låst og CoordChangedFunc (se UC24) er sat.	Den angivne funktion i brugerprogrammet kaldes med information om, at primær input er trykket ned.	Som forventet, men mærkets koordinater medsendes.
Som første, men tasten slippes.	Den angivne funktion i brugerprogrammet kaldes med information om, at primær input er trykket op.	Som forventet, men mærkets koordinater medsendes.
Som første, men markøren er fri.	Intet sker.	Som forventet.
Som første, men CoordChangedFunc er ikke sat	Intet sker.	Som forventet.

UC29 - Tilsending af sekundært input	**Forventet resultat**	**Reelt resultat**
SetUiButtonPressedFunc (pFunc) kaldes, hvor	SIDM vil for fremtiden sende	Som forventet.

pFunc er en pointer til en statisk eller global funktion.	information til denne funktion, når et sekundært input aktiveres.	
SetUiButtonPressedFunc(null) kaldes.	SIDM vil for fremtiden ikke sende information, når et sekundært input aktiveres.	Som forventet.

UC30 - Aktivere sekundære input til brugerprogrammet	**Forventet resultat**	**Reelt resultat**
Brugeren trykker en tast, der er bundet til sekundær input. UiButtonPressedFunc (se UC29) er sat.	Den angivne funktion i brugerprogrammet kaldes. Id på det aktiverede sekundære input medsendes.	Som forventet.
Som første, men UiButtonPressedFunc er ikke sat	Intet sker	Som forventet.

UC31 - Forstørre eller formindske billeder	**Forventet resultat**	**Reelt resultat**
Brugeren trykker en tast, der er bundet til at formindske billederne (dvs. zoome ud).	Billederne formindskes.	Som forventet.
Brugeren trykker en tast, der er bundet til at forstørre billederne (dvs. zoome ind).	Billederne forstørres.	Som forventet.

UC32 - Forstørre eller formindske målemærkerne	**Forventet resultat**	**Reelt resultat**
Brugeren trykker en tast, der er bundet til at formindske målemærkerne.	Målemærkerne gøres mindre.	Som forventet.
Brugeren trykker en tast, der er bundet til at forstørre målemærkerne.	Målemærkerne gøres større.	Som forventet.

UC33 - Info om et kamera	**Forventet resultat**	**Reelt resultat**
GetCameraData(1) kaldes. Et kamera med id 1 eksisterer i databasen.	SIDM returnerer data for det angivne kamera.	Som forventet.
Som første, men et kamera med id 1 eksisterer ikke i databasen.	SIDM returnerer null.	Som forventet.

UC34 - Info om et billede	**Forventet resultat**	**Reelt resultat**
GetImageData(1) kaldes. Et billede med id 1 eksisterer i databasen.	SIDM returnerer data for det angivne billede.	Som forventet.
Som første, men et billede med id 1 eksisterer ikke i databasen.	SIDM returnerer null.	Som forventet.

UC35 - Info om en model	**Forventet resultat**	**Reelt resultat**
GetModelData(1) kaldes. En model med id 1 eksisterer i databasen.	SIDM returnerer data for den angivne model.	Som forventet.
Som første, men en model med id 1 eksisterer ikke i databasen.	SIDM returnerer null.	Som forventet.

UC36 - Ændre data for et kamera	**Forventet resultat**	**Reelt resultat**
SetCameraData(pData) kaldes, hvor pData indeholder data på et kamera, hvis ID findes i databasen.	Informationerne i databasen om det angivne kamera ændres til det medsendte.	Som forventet.
Som første, men kameraets id findes ikke i databasen.	Et nyt kamera oprettes i databasen.	Som forventet.

UC37 - Ændre data for et billede	**Forventet resultat**	**Reelt resultat**
SetImageData(pData) kaldes, hvor pData indeholder data på et billede, hvis ID findes i databasen.	Informationerne i databasen om det angivne billede ændres til det medsendte.	Som forventet.
Som første, men billedets id findes ikke i databasen.	Et nyt billede oprettes i databasen.	Som forventet.

UC38 - Ændre data for en model	Forventet resultat	Reelt resultat
SetModelData(pData) kaldes, hvor pData indeholder data på en model, hvis ID findes i databasen.	Informationerne i databasen om den angivne model ændres til det medsendte.	Som forventet.
Som første, men modellens id findes ikke i databasen.	En ny model oprettes i databasen.	Som forventet.

UC39 - Søgning på kameraer	Forventet resultat	Reelt resultat
FindCameras(ids) kaldes, hvor ids er en tom liste.	Listen fyldes med id på alle kameraer i databasen.	Som forventet.

UC40 - Søgning på billeder	Forventet resultat	Reelt resultat
FindImages(ids, pImageSearch) kaldes, hvor ids er en tom liste og pImageSearch indeholder søge kriterier på billeder.	Listen fyldes med id på alle billeder i databasen, der opfylder søgekriterierne.	Som forventet.

UC41 - Søgning på modeller	Forventet resultat	Reelt resultat
FindModels(ids, pImageSearch, pModelSearch) kaldes, hvor ids er en tom liste, pImageSearch indeholder søge kriterier på billeder og pModelSearch indeholder søgekriterier på modeller.	Listen fyldes med id på alle modeller i databasen, der opfylder modelsøgekriterierne, og indeholder mindst et billede, der opfylder billedsøgekriterierne.	Som forventet.

G Referencer

G.1 BØGER OG ARTIKLER

[R1] Edward Angel, Interactive Compute Graphics, A Top-Down Approach Using OpenGL, third edition, Addison Wesley, Pearson Education, USA, 2003.

[R2] S. Bennett, S. McRobb, and R. Farmer, Object-Oriented Systems Analysis and Design Using UML, second edition, McGraw-Hill Education, Maidenhead, Berkshire, 2002.

[R3] J. M. Carstensen, Image Analysis Vision and Computer Graphics, first edition, IMM, Technical University of Denmark, 2001.

[R4] E. M. Mikhail, J. S. Bethel, and J. C. McGlone, Introduction to Modern Photogrammetry, John Wiley and sons, Inc. 2001

[R5] J. C. McGlone, Manual of Photogrammetry, fifth edition, American Society for photogrammetrics and Remote Sensing (asprs), 2004

[R6] Craig Larman, Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and the Unified Process, second edition, Prentice-Hall, Upper Saddle River, NJ, 2001.

[R7] M. Stiefel and R. J. Oberg, Application Development Using C# and .NET, The Integrated .NET Series from Object Innovations, Prentice-Hall, Upper Saddle River, NJ, 2002.

[R8] B. Stroustrup, The C++ Programming Language, third edition, AT&T Labs, Murray Hull, New Jersey, 1998.

[R9] R. Sharp and J. T. Kristensen, Software Development Projects, Department of Information Technology, DTU, 2000

[R10] James W. Cooper, Introduction to Design Patterns in C#, IBM T. J. Watson Research Center, 2002

G.2 ONLINE ARTIKLER O.L.

[R11] Steve Baker, The Omnivorous Bipeds FAQs, How to Tile Textures without Seams, http://www.sjbaker.org/steve/omniv/tiling_textures.html.

[R12] The OpenGL Programming Guide “The Redbook”, The Official Guide to Learning OpenGL, Silicon Graphics, http://www.glprogramming.com/red/.

[R13] Sam Gentile, Introduction to Managed C++, OnDotnet.com, O’Reilly, http://www.ondotnet.com/pub/a/dotnet/2003/01/13/intromcpp.html.

[R14] Mats Henricson and Erik Nyquist, Programming in C++, Rules and Recommendations, http://www.doc.ic.ac.uk/lab/cplus, Ellemtel Telecom-munication Systems Laboratories.

[R15] Todd Hoff, Possibility Outpost, Programmers Corner, C++ Coding

Standard, http://www.possibility.com/Cpp/CppCodingStandard.html, 2005

[R16] Dr. Thierry Toutin and Corinna Vester, Radar and Stereoscopy, Remote Sensing Torturials, Canada Centre for Remote Sensing, http://www.ccrs.nrcan.gc.ca/ccrs/learn/tutorials/stereosc/stereo_e.html

[R17] Mark Bernatchez, Stereoscopy, Principles, and Applications, Virtual Reality Resource, http://vresources.jump-gate.com/articles/vre_articles/stereo/stereo_article.htm, 1996

[R18] Microsoft Developer Network (MSDN) Library, Microsoft Corporation, http://msdn.microsoft.com, 2004-2005

[R19] Usenet posts through Google Groups, http://groups.google.com.

[R20] Wikipedia, The Free Encyclopedia, Wikimedia Foundation, http://en.wikipedia.org/wiki, 2001 to 2005.

G.3 PROGRAMBIBLIOTEKER

[R21] Open Graphic Language (OpenGL), Silicon Graphics (sgi), http://www.sgi.com/products/software/opengl.

[R22] Nate Robins, OpenGL Utility Toolkit (GLUT) for Win32, http://www.xmission.com/~nate/glut.html.

[R23] Steve Baker, Free OpenGL Utility Toolkit (freeglut), http://freeglut.sourceforge.net.

[R24] Richard Rauch, Open Source OpenGL Utility Toolkit (OpenGLUT), http://openglut.sourceforge.net.

[R25] Paul Rademacher, OpenGL User Interface (GLUI) Library, http://www.cs.unc.edu/~rademach/glui.

[R26] Steve Baker, A Picoscopic User Interface (PUI), Steve's Portable Game Library (PLIB), http://plib.sourceforge.net/pui.

[R27] Sam Lantinga, Simple DirectMedia Layer (SDL), http://www.libsdl.org/index.php.

[R28] Bill Spitzak, The Fast Light Toolkit (FLTK), http://www.fltk.org.

[R29] Ross Johnson, Pthreads Win32, POSIX threads for Win32, http://sources.redhat.com/pthreads-win32.

[R30] Hovik Melikyan, C++ Portable Library (PTypes), http://www.melikyan.com/ptypes/.

[R31] Eric Crahen, Zthreads library, http://zthread.sourceforge.net/.

[R32] Douglas C. Schmidt, The Adaptive Communication Environment (ACE), http://www.cs.wustl.edu/~schmidt/ACE.html.

[R33] Hervé Drolon, FreeImage, http://freeimage.sourceforge.net/index.html.

[R34] Denton Woods, Developer's Image Library (DevIL), http://openil.sourceforge.net.

[R35] ImageMagick, ImageMagick Studio LLC, http://www.imagemagick.org.

[R36] Ulrich von Zadow, Paintlib, http://www.paintlib.de/paintlib.

[R37] Manush Dodunekov, LibODBC++, http://LibODBCxx.sourceforge.net/.

G.4 PROGRAMMER

[R38] Joe Linoff, CcDoc, http://ccdoc.sourceforge.net/.

[R39] ImageStation Stereo Display (ISSD), by Z/I Imaging Corporation of Intergraph Corporation, http://imgs.intergraph.com/issd/default.asp.

[R40] SmartTech uSMART Softcopy Orthophoto http://www.smarttech.co.za/.

[R41] KLT Associates, ATLAS, http://www.kltassoc.com/atlas.html.

[R42] Supresoft Inc. VirtuoZo http://www.supresoft.com.cn/english/products/virtuozo/virtuzo.htm.

[R43] Visual C++ .NET and C# .NET, Visual Studio 2003, Microsoft Developer Environment 2003, Microsoft Corporation, http://msdn.microsoft.com/vstudio/.