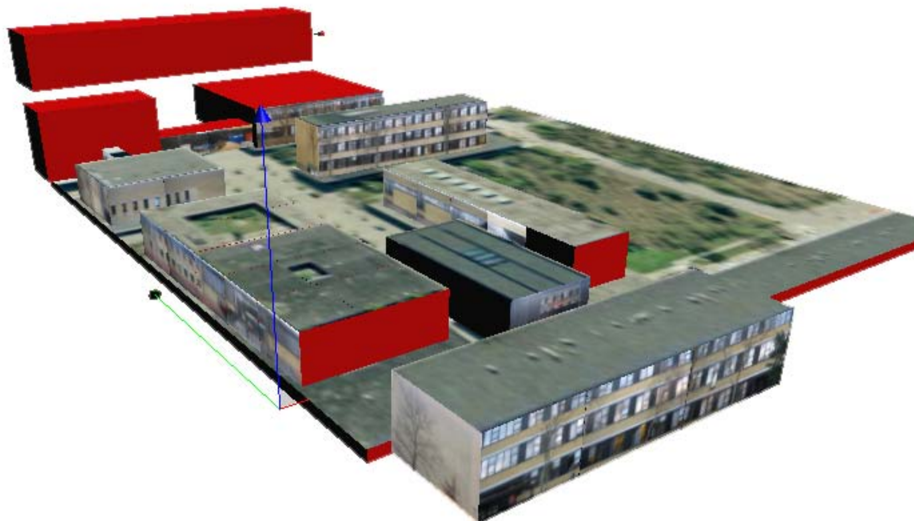
Informatik og Matematik Modellering
Danmarks Tekniske Universtitet

DTU

Brugerguidet konstruktion af arkitektoniske modeller fra fotografier



Christian Højgaard Pedersen (s042314)

5. februar 2008

Vejleder: Jakob Andreas Bærentzen

Abstrakt

I programmer som Google Earth ser vi nu til dags at 3D-repræsentationer af bygninger er indsat sammen med uplant terræn. Der foreligger et stort stykke arbejde ved at pålægge teksturer til disse, så de ikke blot optræder som grå kasser, som det er tilfældet nu. I denne rapport er tekstureringsmetodik til rekonstruktion af bygninger ud fra fotos studeret ved hjælp af oprettelse af et simpelt CAD-værktøj til tegning af kubiske bygninger, som efterfølgende pålægges tekstur projektivt, selvom rapporten også diskuterer almindelig, ortogonal texture mapping. Rapporten indeholder udførlig beskrivelse at implementeringen af selve CAD-værktøjet og den projektive texture mapping, og forskellige metoder til forøgelse af billedkvaliteten samt effektivisering af projektorpositionering diskuteres og belyses.
Det viser sig at texture mapping af geometri til genskabelse af bygninger ikke er så triviel en opgave at forsimple end som så. Blandt projektets konklusioner er, at genoprettelse af geometri fra fotografier stiller store krav til fotografiernes kvalitet og natur, samt at der skal relativt avancerede algoritmer til for henholdsvis at forøge kvaliteten af de restaurerede bygninger og forsimple positioneringsarbejdet med projektoren før man kan tale om en decideret forbedring af arbejdsprocessen. Projektet indeholder ikke en implementering af sådanne, men diskuterer dybdegående mulighederne på området.

Keywords: Fotogrammetri, Vision, Texture Mapping, Geometry Modelling, CAD-software

Indhold

Forord 5
1 Introduktion 6
1.1 Historisk baggrund 6
1.2 Projektets vision 8
2 Kravspecifikation 9
2.1 Kravspecifikation 9
2.2 Tekniske faciliteter 10
3 Implementering
3.1 Introduktion til Navigator klassen 11
3.2 Introduktion til Drawing klassen 11
3.3 Opsætning af virtuelt miljø 12
3.4 Navigation 12
3.5 Picking 16
3.5.1 Z-picking (depth-picking) 16
3.5.2 Ray picking 16
3.6 Mesh klasser 19
3.6.1 Quad 19
3.6.2 RectPar 20
3.6.3 Bestemmelse af bygningers højde 21
3.7 Selection 23
3.8 Lagring og tegning af flere bygninger 24
3.9 Teksturering i projektet 25
3.10 Ortogonal teksturering 25
3.11 Projektiv teksturering 26
3.11.1 OpenGLs egen texture generation facilitet 28
3.11.2 Projektiv teksturering m. GLSL 28
3.12 Automatisk positionering af projektoren 29
3.13 Teksturering, endeligt 30
4 Evaluering 31
4.1 Kort om testene 31
4.2 Grafiske artifacts 32
4.3 Benyttede datastrukturer 34
4.4 Algoritmisk tilgang 35
4.4.1 Zenith Rotation 35

4.4.2 Højdebestemmelse ved Ray-casting 35
4.4.3 Picking – Svagheder ved begge algoritmer 36
4.5 Virker programmet tilfredsstillende? 37

5 Diskussion 38

5.1 Teksturering: Hvad er bedst? 38
5.2 Fremtidige udvidelser: Canoma Pinning teknikken 38
5.3 Fremtidige udvidelser: Mere avanceret geometri 43

6 Konklusion 44

Bibliografi 46
Appendix A – Screenshots 48
Appendix B – Brugermanual 51
Appendix C – Kildekode

C.1 Definitions.h 53
C.2 Drawing.h 54
C.3 Drawing.cpp 55
C.4 FileIO.h 61
C.5 FileIO.cpp 62
C.6 Navigator.h 68
C.7 Navigator.cpp 70
C.8 Projector.h 90
C.9 Projector.cpp 90
C.10 Quad.h 94
C.11 Quad.cpp 95
C.12 Ray.h 98
C.13 Ray.cpp 98
C.14 RectPar.h 101
C.15 RectPar.cpp 101
C.16 TexNav.h 106
C.17 TexNav.cpp 106
C.18 TextureLoader.h 107
C.19 TextureLoader.cpp 107
C.20 main.cpp 109
C.21 projTex.vert 123
C.22 projTex.frag 123

Forord

Denne rapport er produktet af et bachelorprojekt i Softwareteknologi fremstillet ved Informatik for Matematisk Modellering, Danmarks Tekniske Universitet (DTU). Læsere forventes at have basal forståelse for programmeringssproget C++ med tilhørende grafikbibliotek OpenGL samt gængs computergrafisk terminologi.
Projektet forløb fra 1. september 2007 til 5. februar 2008 og er udført af Christian Højgaard Pedersen, s042314.

Kapitel 1

Introduktion

1.1 Historisk baggrund

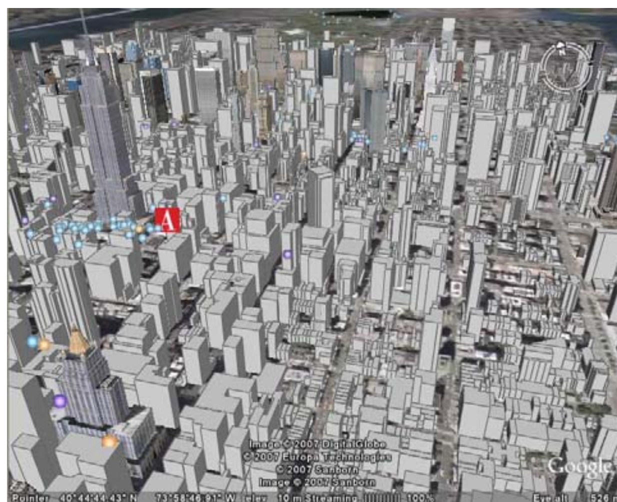
I computergrafik er vi oppe mod udfordringen at omdanne en virtuel verden repræsenteret ved en mængde punkter i et 3-dimensionalt koordinatsystem til en 2-dimensionel flade så den kan vises på skærmen. Den matematiske model der løser dette problem beror, som vi ved, på viewing-transformationer, perspektiviske korrektioner osv.
Disse danner én af grundstenene i moderne computergrafik og er, ikke overraskende, baseret overvejende på principperne bag et almindeligt kamera, eftersom et sådant løser nøjagtig det samme problem for os.

Fra tid til anden udfordres vi imidlertid af den situation, at vi står med den 2-dimensionelle repræsentation, og ønsker os tilbage til den 3-dimensionelle. Denne problemstilling er straks sværere at løse, fordi vi i den oprindelige transformation opgav en afgørende faktor da vi droppede en dimension og tilhørende koordinat – Nemlig dybdeinformation.
Af de utallige konkrete eksempler på netop dette problem kan nævnes genfinding af højdeinformation fra billeder til landkort taget af fly, konstruktion af 3D-modeller af et menneskehoved ud fra et billede af en persons ansigt, etc. At gendanne 3-dimensionel information ud fra 2D-billeder har været brugt stort set siden fotografering selv og kendes som disciplinen ”fotogrammetri”.[1]

I 1999 udgav det amerikanske selskab MetaCreations Corp.[2] et program ved navn ”Canoma”. Programmet kunne genskabe en 3D-scene ud fra billeder af scenen taget fra forskellige vinkler, men led under diverse krav, det første af hvilke netop er nødvendigheden af flere forskellige billeder af samme scene. Ydermere måtte brugeren ”nagle” (eng. ”pin”) punkter af billederne til wireframe objekter indsat I miljøet, så programmet vidste hvor i en 3D-model de enkelte dele af 2D-billedet hørte til. Denne proces resulterer i nydelige miljøer men involverer et forholdsvis ekstensivt stykke manuelt arbejde såfremt større scener ønskes, og øges kun jo mere komplekse strukturer billedet indeholder.

Ligeledes findes problemet, denne gang bogstavelig talt i global størrelsesorden, i programmet Earth View, udviklet af Keyhole Inc og udgivet i 2005, da firmaet blev opkøbt af Google og programmets titel ændret til Google Earth [3]. De fleste kender Google Earth som et underholdende program der lader brugeren bese Jorden fra satellitbilleder mappet til en model af kloden, men programmet er først for nyligt begyndt at gøre forsøg I retning af 3D-repræsentationer af geometri på overfladen, hvor terræn og urbane miljøer igen skal gendannes fra 2D.

Til det formål anskaffede Google sig i 2006 @Last Software's "SketchUp" fordi det havde en plugin der tillod eksport af 3D-modeller til netop Google Earth.
Således kunne brugere verden over nu generere 3D-modeller af bygninger til Google Earth, men problemet består endnu: Bygningerne skal pålægges teksturer så de ikke blot er grå bokse. Denne proces er langsomt påbegyndt men langt fra færdig, jf. billedet nedenfor.



Figur 1: Det sydlige Manhattan med sporadisk teksturerede bygninger. Google Earth, Dec 2007.12.07.

1.2 Projektets vision

Formålet med dette projekt er derfor at foretage en undersøgelse af hvordan 3D-modeller lettest og mest effektivt kan pålægges brugervalgte teksturer. Fremgangsmåden er at programmere et simpelt værktøj til oprettelse af 3D-modeller, der vil tjene det formål at repræsentere kubiske bygninger såsom dem fundet på Danmarks Tekniske Universitet. Disse bygninger forsøges herefter projektivt tekstureret så det kan bestemmes om denne model letter arbejdsbyrden på nogen måde, samt hvorvidt projektiv teksturering er visuelt tilfredsstillende.

Værktøjet vil kræve opsætning af et typisk 3-dimensionelt miljø, grundplanet i hvilket fungerer som kanvas for de tegnede bygninger. Bygninger kræver 2 typer meshes, ét til at repræsentere et simpelt rektangel, samt ét til at repræsentere bygningen selv. Herfra implementeres projektiv teksturering samt en virtuel model af selve projektoren. Det forventes at brugeren gennem programmets interface har kontrol over position af kamera (eget synspunkt), bygninger og projektor. Et hjælpevindue stilles ydermere til rådighed for at skaffe overblik over hvilke teksturer programmet har til rådighed og valget imellem dem.

Eftersom værktøjet vil kræve en objektorienteret tilgang er et high level sprog en nødvendighed. Derudover er applikationen computergrafisk af natur, hvorfor et grafikbibliotek er nødvendigt. Man kunne med lethed vælge C# og Direct3d, men min erfaring ligger hos OpenGL, så programmet kodes i C++/OpenGL.

Kapitel 2

Kravspecifikation

Dette kapitel gennemgår programmets kravspecifikation og diskuterer hvilke tekniske faciliteter der skal tages i brug for at imødekomme specifikationen.

2.1 Kravspecifikation

Gennem projektets vision i afsnit 1.2 berørte vi kort kravspecifikationen for det færdige program. Her følger en uddybelse af hvad programmet mere konkret skal indeholde.

1. Opsætning af 3d-rum
    a. Der opstilles et 3-dimensionelt, kartesisk koordinatsystem i hvilket y-aksen er op, og x og z former et typisk 2-dimensionelt koordinat system med x i sin normale retning, når systemet beses *fra neden***.** Dette er et *venstrehåndet* koordinatsystem.
    b. Akserne skal være synlige og tydeligt markeret for brugeren, så denne kan orientere sig.
    c. Brugeren skal kunne navigere rundt i miljøet ved brug af både mus og keyboard, der kræves altså et velstruktureret bruger input system.
2. Dynamisk tegnefunktionalitet
    a. Det skal være muligt at tegne rektangler i grundplanet ved at ”trække” dem ud med musen som man gør det med et lasso-værktøj omkring ikoner i Windows.
    b. Det skal være muligt at ekstrudere rektangler i grundplanet for at forme bygninger.
    c. Det skal være muligt at flytte rundt på både rektangler og bygninger, samt slette dem igen.
    d. Det skal være muligt at ændre størrelsen af bygninger.
    e. Miljøet skal være belyst, så brugeren har et tydeligt indtryk af bygningers 3-dimensionelle natur.
3. Projektiv teksturering
    a. Programmet skal indeholde en virtuel repræsentation af en projektor, der kan belyse geometri med tekstur som et lysbilledapparat ville gøre det i virkeligheden.
    b. Det skal være tydeligt for brugeren hvor projektoren er og hvor den peger hen. Brugeren må generelt aldrig have følelsen af at have ”tabt” hverken sin egen orientering eller projektorens.
4. Bruger interface
    a. Programmet skal indeholde et vindue udover det primære, i hvilket de tilgængelige teksturer vises. Brugeren skal have lejlighed til at udvælge og bese teksturer heri. Den i dette vindue valgte tekstur skal være den, der projiceres i hovedvinduet.

b. I hovedvinduet skal forefindes adskillige kontroller, der tillader brugeren at navigere rundt på forskellige måder, udvælge tegnet geometri, flytte rundt på tegnet geometri, flytte rundt på projektors location såvel som retning samt naturligvis at binde teksturer til bygninger.

2.2 Tekniske faciliteter

Programmet benytter sig af adskillige sproglige udbygninger og features som letter arbejdet. Denne sektion har til formål at opridse hvilke biblioteker programmet tager brug af.

OpenGL og GLUT
Projektet benytter sig af GLUT til at håndtere window management og OpenGL Extension Wrangler Library (GLEW) til understøttelse af vertex- og pixelshaders.

GLUT (http://www.opengl.org/resources/libraries/glut/) er udviklet af Mark Kilgard, og GLEW (http://glew.sourceforge.net/) af Milan Ikits og Marcelo Magallon.

Computer Graphics Linear Algebra (CGLA)
Applikationen er i svær grad afhængig af vektorer og i mindre grad af 3x3-matricer, hvorfor CGLA inkluderes. CGLA er et effektivt, platform uafhængigt bibliotek med mange nyttige elementer hyppigt brugt i computergrafik. Det er en del af at større framework ved navn GEL, og udviklet af Andreas Bærentzen.
Kildekode samt dokumentation af tilgængeligt på http://www2.imm.dtu.dk/projects/GEL/

GLUI
Som kravspecifikationen bærer tydelig præg af vil programmet få behov for et bibliotek der kan tage sig af opsætning af et grafisk brugerinterface, da det vil være alt for tidskrævende selv at konstruere knapper, drop-down menuer og andet. Kravet er at biblioteket skal holde sig fra window management da det jo håndteres af glut, dvs. der skal altså bruges noget generelt OpenGL-kompatibelt. Jeg foretog i projektets indledende faser en søgning på et sådant, og der er heldigvis en god del til rådighed på nettet.

Der findes en liste over nogle få på én af udviklernes hjemmeside, http://www.bramstein.nl/gui/, selvom siden i skrivende stund er nede. Mange af dem har mere eller mindre farvestrålende widgets tiltænkt spilinterfaces, så valget faldt på et mere neutralt interface ved navn GLUI. Dette er tilgængeligt på http://www.cs.unc.edu/~rademach/glui/ og udviklet af Paul Rademacher. Interfacet inkluderer widgets som translations- og rotations-knapper udover alle tænkeligt sædvanligt (knapper, tekstbokse, listbokse, osv) og egner sig således strålende til dette projekts krav.

Kapitel 3

Implementering

Dette kapitel er essentielt delt i to grunddele. Den ene, kaldet ”CAD-Systemet”, bearbejder opsætning af hele CAD-delen af programmet, det værende sig navigationen rundt i miljøet samt tegnefunktionaliteten. Den anden, kaldet ”Texture Mapping”, omhandler implementeringen af den projektive teksturering samt den virtuelle lysbilledprojektor.

CAD-Systemet

3.1 Introduktion til Navigator klassen

Én af programmets helt centrate klasser er Navigator. Klassen varetager en bred vifte af ansvarsområder som vi løbende støder på gennem rapporten, så den introduceres derfor her.
Oprindeligt oprettedes Navigator som en kameraklasse hvis formål var at holde styr på kameraets position og retning, men som projektet skred frem voksede klassens omfang, og ved afslutningen har Navigator udviklet sig en central for behandling af brugerinput. Behandling af alle feedbacks det grafiske brugerinterface (glui) genererer samt alle de funktioner gluts forskellige callbacks kalder er at finde i Navigator. Klassen håndterer dermed blandt andet keyboard, mus, animation, selection, picking, aktivt vindue og udvalgt geometri, de to først nævnte værende absolut ikke-trivielle funktioner.

3.2 Introduktion til Drawing klassen

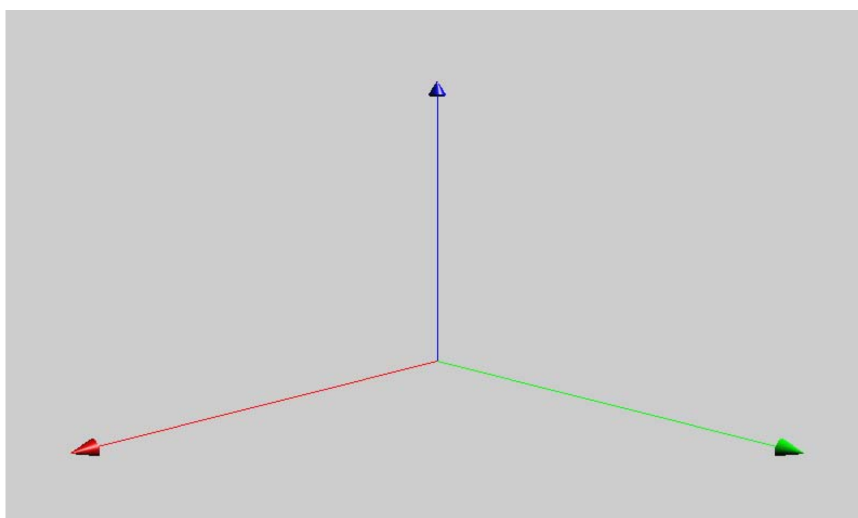
Navigator hænger uløseligt sammen med klassen Drawing, som har til ansvar at foretage alle OpenGL kald der tegner den specificerede geometri samt at oprette datastrukturer til at holde styr på de forskellige objekter. Klassen står således for at tegne både Quads og RectPars, tilføje og fjerne dem fra deres respektive datastrukturer samt at hente referencer til dem. Klassen står også for at tegne koordinatsystemets akser og opsætte simple lysforhold, så uteksturereret geometri fremstår nuanceret (modsat uniform farvet).

Navigator har en pointer til et drawing objekt og kalder klassens funktioner som resultat af diverse bruger input, og sammen med programmets main former Navigator og Drawing programmets absolutte kerne.

3.3 Opsætning af virtuelt 3D-miljø

OpenGLs koordinatsystem er venstrehåndet, så API defaulten tager sig automatisk af dette.

Kameraets start-position er initialiseret til (3.0, 1.5, 3.0) og dets start-retning til (-1.5, -0.5, -1.5), svarende til et at-point (også kaldet ”point of interest”) beliggende i summen af position og retning, altså (1.5, 1.0, 1.5). Planet y = 0 betragtes som ”jorden”, dvs. det kanvas på hvilket alle firkanter tegnes og alle bygninger konstrueres.
For at give brugeren et klart overblik over dennes position i rummet visualiseres koordinatsystemets akser og deres positive retning markeres med en lille kegle. X-aksen er grøn, Z er rød og Y er blå. Med disse banaliteter på plads introduceres brugeren for følgende billede i hovedvinduet når programmet åbnes:



Figur 2: Kameraets start position i 3D-miljøet

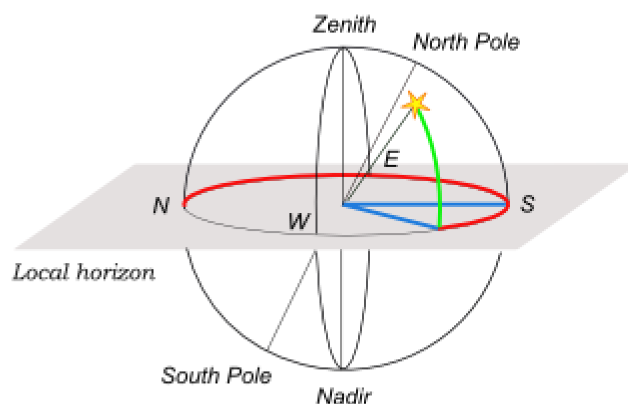
3.4 Navigation

Næste skridt i implementeringen bør være navigation, så kameraet kan flyttes interaktivt rundt i systemet. Translation frem og tilbage er åbenlys – Det er blot et spørgsmål om addition og subtraktion af en konstant gange kameraets synsretning. Konstanten kan bestemmes af en Timer-klasse, som i programmet bruges til måle noget, som varierer afhængig af clockfrekvensen på den computer programmet kører på, for at sikre ensartet navigation uanset processorhastighed. Hvad præcis der måles er op til den enkelte programmør, men ofte, som også her, er det den tid, det tog at rendere forrige frame.

Der er implementeret to måder at navigere på. Den ene (freefly) er vektorbaseret, mens den anden er sfærisk, og således tillader brugeren at rotere rundt om et punkt i forlængelse af synsretningen. I freeflying findes de vektorer, der lægges til for at dreje til højre og venstre som krydsprodukterne af synsretning og kameraets up-vektor. (Det er en typisk brugt teknik).
Under rotation oprettes rotationsmatricer som beskrevet i Edward Angels bog. [7] Før rotationen kunne implementeres måtte det bestemmes hvilket punkt der roteres omkring, hvilket gav lidt tænkearbejde under udarbejdelsen. Man kunne selvfølgelig rotere om koordinatsystemets origo, hvilket ville resultere i en pålidelig og tilregnelig løsning, men det giver ikke altid det bedste resultat. Hvis brugeren for eksempel er i gang med at arbejde et sted langt fra origo, og roterer kameraet fra den position, så vil kameraet rejse en umådelig stor distance, fordi afstanden til origo resulterer i en enorm sfære, det kan bevæge sig over. Det lader til at være en skidt løsning, fordi grunden til overhovedet at rotere kameraet sandsynligvis er at kunne beskue geometri beliggende indenfor kort afstand fra den modsatte side af *geometrien* – Ikke fra den modsatte side af *koordinatsystemet*.

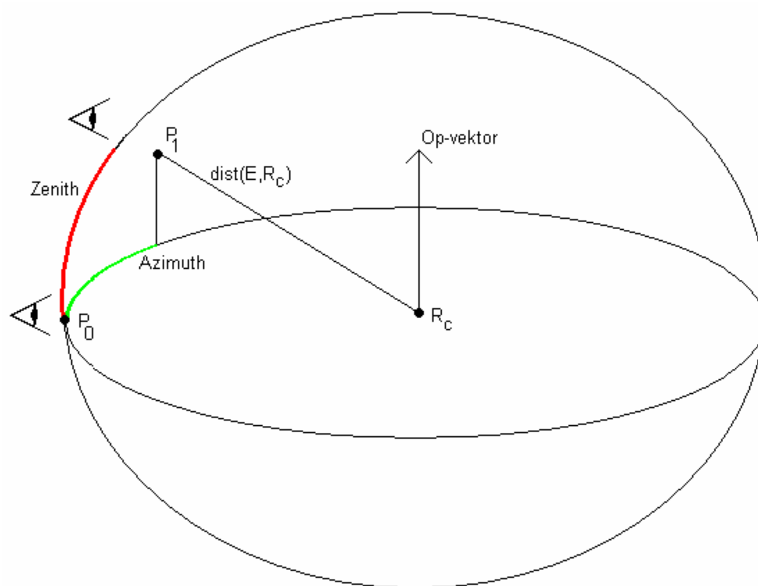
Derfor lader det til at være en bedre idé at basere rotationens centrum på kameraets at-point på en måde. Så vil rotationen finde sted om et punkt omtrent dér, hvor brugeren ønsker den. Der eksperimenteredes med at lægge centrum på 5.0 gange vektoren fra kamerapositionen til at-pointet (altså 5.0 gange kameraretningen). Når den løsning kombineres med frem/tilbage translation med musens hjul virker det ganske udmærket, men det er ikke altid der er en mus med et hjul til rådighed; brugeren kunne sidde ved en bærbar, for eksempel. Derfor er der i det færdige program indsat et spinning tool, der tillader brugeren selv at vælge afstanden efter behov.

Ved rotationen bruger vi terminologi fra det horisontale koordinatsystem. [8] Her betragter vi to begreber, azimuth og zenith. Azimuth er en vinkel mellem et referenceplan og et punkt. Zenith er et punkt beliggende ”over hovedet”, som er begrebets etymologiske betydning. I programmet bruges musens bevægelse over skærmen til at indikere rotationsretninger, så de pixels over hvilke musen rejser markerer derfor vinklerne. Pixels gennemløbet i y-retning tolkes som zenith-vinklen, og i x-retning som azimuth-vinklen. Begreberne bruges således direkte til at beskrive vertikal og horisontal rotation, trods det, at terminologien i forhold til navigation i det horisontale koordinatsystem ikke er helt korrekt.



Figur 3: Illustration, forklarende begreberne azimuth og zenith som traditionelt brugt i det horisontale koordinatsystem.[9]

Da vinkler nu er præcist defineret kan matricer opstilles og rotationen endeligt implementeres. Azimuth rotation foregår altid om en vektor i samme retning som koordinatsystemets op-vektor gennem det valgte punkt. Zenith rotation foregår altid om en vektor i en retning perpendikulær til kameraets synsretning gennem det valgte punkt. Se figuren her:



Figur 4: Rotation med centrum i punktet R_c. Vinklerne azimuth og zenith er markeret på tegningen med henholdsvis grøn og rød. Punkterne P_0 og P_1 indikerer sammen den oprindelige og den nye kamera position.

Den horisontale rotation foregår simpelt nok ved at trække det punkt, der roteres omkring fra kamerapositionen. Nu ligger kameraet i origo, og rotationsmatricen er nu:

$$R_y(azimuth) = \begin{pmatrix} \cos(azimuth) & 0 & \sin(azimuth) \\ 0 & 1 & 0 \\ -\sin(azimuth) & 0 & \cos(azimuth) \end{pmatrix}$$

, altså blot rotation om y med azimuth grader. Efterfølgende tillægges centrum for rotationen igen, så kameraet returneres til sin nye location, relativ til centrum.

Vertikal rotation er noget mere besværlig at udføre, fordi den ikke er en rotation om en enkelt af koordinatsystemets akser, men om en arbitrær vektor defineret af krydsproduktet mellem kameraets synsretning og systemets op-vektor. Matematikken bag denne type rotation involverer for det første individuelle rotationer mellem alle tre af systemets akser, for det andet transformation af vektoren ind i yz-planet før rotation om x kan finde sted, osv. At opsætte hele proceduren manuelt er relativt

vanskeligt og tungt udsat for fejl, så programmet benytter sig af et fiffigt lille trick til at forsimple opgaven.

Det er nemlig heldigvis sådan, at OpenGL indeholder masser af faciliteter, der kan manipulere en matrix på den måde vi ønsker her. For at oprette matricen til zenith rotation tager vi derfor kortvarigt modelviewmatricen til låns. Den sættes til identitet, og nu er det ganske enkelt et spørgsmål om ét enkelt kald til `glRotatef`, som jo netop tager en vinkel og koordinaterne til den vektor vi ønsker at rotere omkring som argumenter. Nu indeholder modelviewmatricen præcis den rotationsmatrix der skal bruges, og den ekstraheres så med `glGetFloatv`. Hvorvidt dette altid er en brugbar metode vil jeg vende tilbage til i Kapitel 4.

3.5 Picking

Ét af programmets visionspunkter er at undersøge hvordan teksturering potentielt set kan lettes, så en vis grad af brugervenlighed må forventes. Derfor implementeres en metode hvorpå brugeren kan specificere de punkter, hun ønsker skal afgrænse geometri. Man kunne naturligvis lade brugeren eksplicit definere punkter via tekstbokse, men det kan dårligt siges at være brugervenligt, så ansvaret for at bestemme punkter i verdenskoordinatsystemet baseret på punkter i skærmens koordinatsystem ligger hos applikationen.
Denne teknik kaldes ”picking” og er et interessant problem som i virkeligheden udgør et, i forhold til projektets emne, ækvivalent subproblem af det vi introducerede i kapitel 1, fordi der er tale om gendannelse af tabt dybdeinformation. (Fra screen space til world space). Applikationen benytter sig af to forskellige algoritmer, der på hver sin måde forsøger at komme problemet til livs. Som sædvanlig har forskellige algoritmer hver sit sæt af fordele og ulemper. Disse er grunden til at programmet bruger begge algoritmer, og vi vil diskutere dem i kapitel 4.

3.5.1 Z-picking (depth-picking)

Z-picking forsøger at afprojicere det pågældende punkt (x,y) i skærmkoordinatsystemet til det tilsvarende punkt (x,y,z) i verdenskoordinatsystemet. Algoritmen henter først to GL tilstandsvariable: Viewport dimensionerne og det interval indenfor hvilket dybdeværdier ligger (`GL_DEPTH_RANGE`). Dernæst læses dybdeværdien i pixelen, på hvilket musemarkøren ligger når funktionen kaldes. Der foretages en undersøgelse af hvorvidt den fundne dybdeinformation matcher den maksimale dybde. Er dette tilfældet har brugeren klikket på det bagerste clipping plan, og funktionen returnerer `false`. Ellers sættes modelview-matricen til identiteten og kameraet opsættes via `gluLookAt`, for at sikre at modelviewmatricen kun indeholder view-matricen, og altså ikke model-transformationer på tidspunktet.

Slutteligt hentes OpenGLs projektionsmatrix og modelviewmatricen, og punktet afprojiceres vha. en GL Utility Library funktion, `gluUnproject`, der sørger for at gemme worldspace ækvivalensen til screenspace-punktet i 3 doubles til brug udenfor funktionen.

3.5.2 Ray picking

Ray picking, ulig Z-picking, forsøger ikke endeligt at bestemme punkter ved at afprojicere det fra skærmen, men benytter sig i stedet af ray casting, hvorfor en klasse til at repræsentere linier i rummet er at finde i programmet. Metoden finder punkter ved at lade en parametrisk linie skyde fra kameraets position i retning af det pixel, på hvilket brugeren klikkede med musen. (Renderingsmetoden raytracing gør i øvrigt det samme for alle pixels i vinduet for at generere rays af første niveau. Raytracing er udenfor rapportens område men nævnes kort fordi teknikken indledningsvis er den samme som her).

Som med Z-picking hentes først viewport-information, hvorefter modelviewmatricen sættes til kun at indeholde viewing transformationer. Ray picking benytter sig også af `gluUnproject`, så vi skal igen bruge modelview- og projektionsmatricerne.
Herefter foretages to separate kald til `gluUnproject`: Først med dybdeværdien sat til 0.0, altså det forreste clipping plan, og dernæst med dybdeværdien sat til 1.0, altså det bagerste clipping plan. De to punkter disse to kald resulterer i gemmes, og trækkes hernæst fra hinanden. Det interessante er, at resultatet af subtraktionen mellem disse to punkter udgør den parametriske linies retning.
Da oprindelsespunktet er kameraets position har vi nu alt hvad vi skal bruge for at konstruere strålen. Funktionen `ray_pick` benytter sig af to hjælpefunktioner der henholdsvis udregner skæring med et arbitrært plan og efterfølgende returnerer parameteren t, der udgør afstanden til det plan, med hvilket man er interesseret i en skæring.
Da programmet udelukkende tegner i grundplanet y = 0 er vi naturligvis interesseret i skæring med xz-planet, og når vi har udregnet t er det blot at indsætte parameteren i liniens ligning. Resultatet er det punkt i grundplanet, som brugeren klikkede på.

Returnering af t er trivielt, men skæringsberegningen har nogle få punkter der er matematisk interessante nok til at nævnes her.
I skæringsalgoritmen udregnes to floating point variable, V_d og V_0:

$$V_d = \vec{N} \bullet \vec{R}_d$$

$$V_0 = -(\vec{N} \bullet \vec{R}_d + D)$$

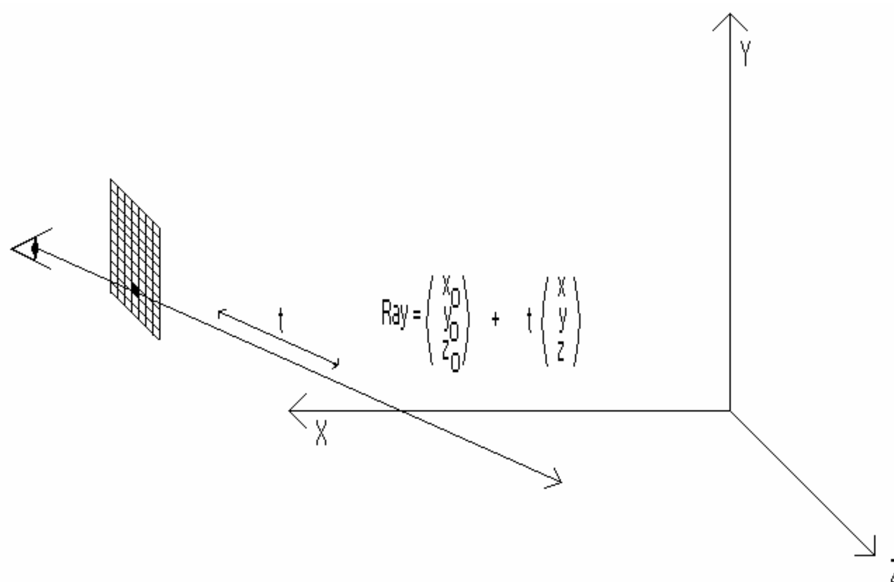
Hvor $\vec{N}$ er planets normalvektor, $\vec{R}_d$ er liniens retning og D er afstanden fra planet til koordinatsystemets origo. Fra vektorregning ved vi at $\vec{N} \bullet \vec{R}_d = \cos(\theta)$, hvor θ er vinklen mellem de to vektorer.

Dette er af interesse, fordi vi kan bruge prikproduktet til at bestemme hvorvidt en skæring overhovedet finder sted, og om vi er interesseret i den pågældende skæring *hvis* en sådan finder sted. Er prikproduktet 0, så er strålen vinkelret på planet, og en skæring kan således ikke finde sted. Hvis prikproduktet er større end 0, så er vinklen mellem planets normal og strålens retning spids. Vi er kun interesserede i stumpe vinkler, fordi alle andre betyder vi har skåret planet fra et sted under det. Programmet forventer ikke at brugeren er interesseret i at tegne på grundplanet fra en kameraposition under det, så skæringer af denne karakter ignoreres simpelthen.

Findes en acceptabel skæring kan t udregnes som kvotienten mellem V_d og V_0, altså:

$$t = \frac{V_0}{V_d}$$

Nedenstående simple figur illustrerer algoritmen.



Figur 5: Ray picking. En parametrisk linie skydes fra øjet (kameraet) gennem en pixel i viewporten og skæring med xz-planet udregnes.

Til slut skal siges at begge algoritmer bruges i programmet, fordi deres respektive måder at virke på er ideel i to forskellige situationer. I første omgang bruges ray-picking, fordi programmet har behov for at bestemme punkter i planen. I starten blev Z-picking også brugt til det, men det kræver at planet indeholder noget geometri at tegne på, ellers vil Z-picking evaluere alle klikkede punkter til det bagerste clipping plan, og det kan vi ikke bruge til noget. For at imødekomme det problem kunne man stille klikbart geometri til rådighed ved indledningsvist at tegne en kæmpe quad, der så fungerer som kanvas, men det medfører yderligere komplikationer for det første i form af en begrænsning af tegnefladen, for det andet i, at OpenGL så får problemer med at finde ud af hvad der skal tegnes øverst – Quads i grundplanet, eller grundplanet selv. Sidstnævnte burde kunne løses ved at tegne quads med `glDepthFunc(GL_LEQUAL)` sat, men når tegning i forvejen kun foregår i planen virker en ray-casting løsning ideel.
Senere i programmet er der til gengæld behov for at bestemme punkter andetsteds, nemlig når der klikkes på sider af bygninger for at teksturere dem. Til dette formål bruges Z-picking, fordi en ray-casting løsning ville kræve oprettelse af et plan, som bygningens side flugter med, og det kan ikke umiddelbart findes.

3.6 Mesh Klasser

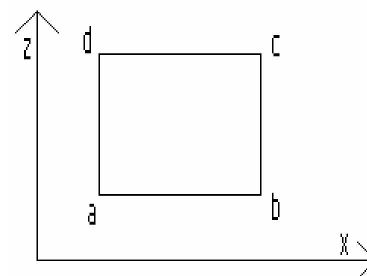
De to klasser der varetager repræsentationen af den understøttede geometri gennemgås nu, og det diskuteres hvordan programmet udregner koordinater og opstiller geometri på baggrund af det input det modtager fra brugeren.

3.6.1 Quad

Quad er en forkortelse for det engelske "quadrilateral" og betyder blot "firkant". Af hensyn til overensstemmelse mellem programmets og rapportens terminologi benævnes enhver firkant herefter ved betegnelsen "Quad".
Quads udgør den fundamentale enhed i alle strukturer i programmet. Enhver bygning starter som en quad i grundplanet og ekstruderes herfra til den kubiske struktur programmet benytter som sin repræsentation af en bygning, så vi starter en gennemgang af programmets tegnefunktionalitet ved denne.

En klasse, Quad, implementerer quads i systemet. En quad tegnes af brugeren ved at klikke på et punkt i grundplanet med venstre museknap og, med knappen i bund, trække quaden ud i den ønskede størrelse. Det giver sig selv at alt man behøver for at tegne en quad (eller en kube, for den sags skyld) er to diametralt modsatte hjørner, her kaldet a og c. Det er derfor constructorens arbejde selv at udregne b og d på baggrund af a og c's x- og z-koordinater. Y sættes lig 0, fordi quaden jo tegnes i grundplanet.

Punkterne b og d udregnes så de ligger som vist på figur 4, fordi OpenGL per default tolker overflader, vis punkters rækkefølge er modsat uret som front faces. (glFrontFace default er CCW).



Figur 6: Quad m. CCW punkter

Simpelt nok er b's x-koordinat lig c's og z-koordinaten lig a's mens en tilsvarende logik bruges til d. Her støder vi til gengæld på et mindre problem, idet vi ikke kan være sikre på hvilken retning brugeren har tegnet i. Der er ingen garanti for at første punkt, a, ligger der hvor det er vist på figuren. Brugeren kan have sat musen så a ligger i et hvilket som helst af de på figuren viste hjørner. Det er altså ikke nok bare at antage at b og d's koordinater altid afhænger af a og c som beskrevet ovenfor, man må først udføre et mindre analytisk arbejde for at finde ud af med hvilken orientering brugeren har tegnet quaden. Denne analyse foregår ligeledes i constructoren til en quad, da hjørnernes koordinater sættes her.

Herudover har klassen også adskillige standard get- & set-funktioner til rådighed samt en funktion, der translaterer quad'en med en vektor.

Slutteligt har hver quad en lille `typedef struct` som indeholder data forbundet med projektoren. Hvordan dette bruges vil vi diskutere i senere i kapitlet, hvor teksturering af scenen behandles.

3.6.2 RectPar

RectPar er en forkortelse for det engelske ”rectangular parallelepiped”, som betyder ”rektangulært parallelepipidum”, og er enhver 3-dimensionel struktur med 6 sider, hvor siderne er parallelogrammer. Skal man være lidt sprogligt pedantisk er en kube (en ”regulær hexahedron”) et eksempel på en sådan, men ethvert rektangulært parallelepipidum er omvendt *ikke* en kube, fordi en kubes sider er kvadrater. Det er derfor i sagens natur terminologisk forkert at henføre til bygninger i programmet som ”kubiske”, selvom rapporten tager sig den frihed at gøre dette.

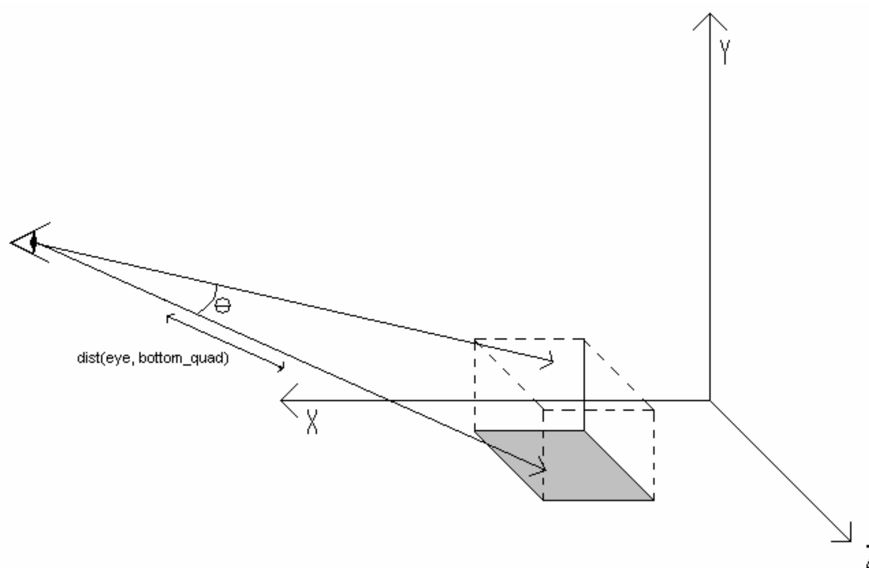
I dette projekt repræsenteres bygninger ved klassen RectPar. Bygningen stykkes sammen af 6 quads, hvorfor hvert RectPar-objekt har 6 quads som member variables; en bund, en top og 4 sider. I constructoren modtages hjørnerne a og c af bunden samt en højde, h. En quad dannes ud fra a og c, og sættes til bygningens bund. Herefter sættes tagets koordinater til bundens, men med y tillagt h. Den resterende kode i constructoren er umiddelbart selvforklarende. Den sætter blot sidernes koordinater ud fra toppens og bundens.

Klassen indeholder tre funktioner mere, to af hvilke er trivielle. (Den ene parallelforskyder (translaterer) bygningen ved at kalde translations-funktionen for hver af bygningens quads mens den anden blot returnerer pointere til bygningens quads i et array). Den sidste funktion i klassen modtager et punkt og analyserer sig frem til hvilken quad punktet tilhører, og returnerer denne. Kan funktionen ikke matche punktet til nogen quad returneres en dummy som indeholder rene nuller.
Fremgangsmåden er at sammenligne det modtagne punkts koordinater med koordinaterne i alle quads, én efter én, indtil en quad findes hvor alle 4 hjørner har en koordinat til fælles med det pågældende punkt. Er dette tilfældet konkluderes det, at punktet ligger i samme plan som den quad der undersøges, og en pointer til denne quad returneres. Funktionen bruges i forbindelse med tekstureringsfaciliteten i programmet, og når denne gennemgås vil vi vende tilbage til den med flere kommentarer, da funktionen har nogle svagheder som er værd at diskutere.

3.6.3 Bestemmelse af bygningers højde

Målet er at bygninger ekstruderes fra grundplanet ved at klikke på en quad og trække bygningen op i den ønskede højde med musen. Derfor må en metode til bestemmelse af bygningens højde ud fra de pixels musen bevæger sig på skærmen findes. Dette lyder umiddelbart simpelt men der er mere kød på opgaven end som så, hvorfor de tanker der er gjort præsenteres her.
For at starte i det simple kunne man selvfølgelig bare mappe pixels fra musens bevægelse lineært direkte til en højde på bygningen via en matematisk funktion, men det skal vise sig at være en absolut horribel løsning, fordi det helt ignorerer kameraets afstand til den ønskede bygning. Resultatet ville blive at en bygning svært ukontrolleret stiger til uønskede højder når kameraet er tæt på mens en acceptabel højde bliver umulig at opnå hvis der tegnes på lang afstand.
Helt afgjort skal afstanden mellem kamera og den udvalgte quad involveres. Den i programmet brugte metode benytter sig igen af ray casting, som vi i det følgende skal se.

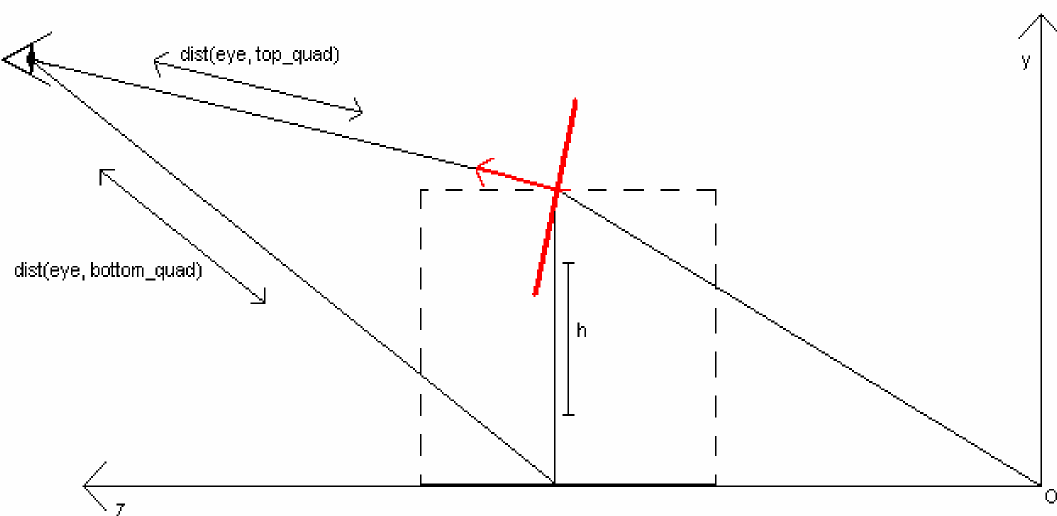
Der kastes denne gang to stråler, i første omgang én fra øjet til bygningens nederste quad, og efterfølgende interaktivt én i retning af musen, når den bevæges væk fra det punkt på den nederste quad brugeren trykkede på for at ekstrudere. Vha. prikproduktet mellem de to strålers retningsvektorer findes vinklen θ mellem dem. På samme måde som ved ray picking findes afstanden fra øjet til nederste quad. Denne påganges vinklen sammen med en float på 0.94 som er et "magisk" tal, om man vil, der er fundet frem til ved at finjustere. Metoden, som illustreret på figur 7, inddrager afstanden fra kameraet og producerer bygninger af tilfredsstillende højder. Det vises imidlertid at denne metode langt fra er den bedste af adskillige årsager som jeg vil tage op i kapitel 4.



Figur 7: Højdebestemmelse ved ray casting

Jeg kom senere i projektets forløb på en bedre metode som burde være mere matematisk korrekt og resultere i en lidt nøjagtigere højde, ser man bort fra floating point afrundinger. Denne præsenteres også her, selvom den ikke er implementeret.

Metoden går ud på at oprette et plan perpendikulært på retningen af den stråle, der går fra øjet til den ønskede bygnings tag, dvs. et plan der har strålens retning, gange -1, som normalvektor. Eneste sidste variabel der er behov for er planets afstand, D, til origo. Denne sættes til afstanden fra det punkt i bygningens gulv, brugeren klikkede på, og udgør dermed algoritmens eneste unøjagtighed, eftersom planets afstand til origo i virkeligheden er hypotenusen i en retvinklet trekant, mens jeg fejlagtigt sætter den til en katete. Se nedenstående figur:



Figur 8: Et forsøg på højdebestemmelse via oprettelse af plan (rødt)

Unøjagtighedens signifikans er afhængig af hvor langt væk fra origo den pågældende bygning forsøges tegnet. Tæt på er forskellen mellem hypotenusen og den katete hvis længde vi bruger større, førende til en større unøjagtighed.
Tanken er nu, at strålen fra kamera til bygningens tag skæres med det oprettede plan, hvorved afstanden fra kameraet til taget findes. Dermed har vi to sider i en trekant, hvor højden er den eneste ubekendte. De nødvendige vinkler mellem de to stråler og mellem strålen fra kamera til grundplan er intet problem at finde, og man burde så kunne regne sig frem til højden vha. enten sinus- eller cosinusrelationen.

3.7 Selection

Udover teksturering er tanken at bygninger og quads skal kunne flyttes rundt af brugeren og størrelsen skal kunne ændres, så brugeren kan lege med den måde teksturingen ændrer sig på når geometrien ændrer sig og bruge det til at fintune det visuelle indtryk.
Programmet har derfor, ikke overraskende, behov for en måde hvorpå allerede tegnede objekter kan identificeres på, så det kan specificeres med musen hvilket objekt, man ønsker af arbejde med. For at imødekomme dette krav introduceres selection.

Traditionelt set er der adskillige foreslåede metoder man kan implementere selection på. OpenGL har en indbygget selection facilitet der involverer initialisering af en name stack. Uden at gå for meget i detaljer tildeles objekter heltalsnavne, som skubbes på en stack under et separat renderings pass. Det involverer at sætte `glRenderMode` til selection og oprette en picking matrix i passende størrelse, alt efter hvor præcis man ønsker sin selecting skal være. Jo større picking matrix, jo mere unøjagtig tillader programmet brugeren at klikke. OpenGL tager sig herefter af resten, og indsætter data genereret som funktion af et hit i en bestemt buffer, programmøren selv definerer, hvor hvert hit resulterer i 4 elementer i bufferen. Først og fremmest hvor mange navne dette hit inkluderer, da objekter kan overlappe. Dernæst min-max dybdeinformation for de vertices, der indgår i hittet, og slutteligt, navnet [4].
OpenGLs selection facilitet har en mindre ulempe, idet objekter der tegnes udenfor skærmen i implementationer der for eksempel indeholder scrollbars eller er uden view frustum culling også tildeles navne og potentielt set kan udvælges ved fejl.

Der findes et interessant alternativ, hvor objekter tegnes i det separate renderings pass med hver sin unikke farve. Under selection læser man farven på det pixel, på hvilket brugeren klikkede, og sammenligner resultatet med farven på alle objekter i scenen. Findes et match, er der tale om et hit. Denne metode eliminerer problemet med objekter tegnet offscreen, men umuliggør til gengæld multiple hits, eftersom man kun kan detektere farven på det objekt, der ligger tegnet øverst, når objekter overlapper.

Projektet bruger OpenGLs indbyggede selection, for det første fordi den var umiddelbart tilgængelig. Jeg kendte til begge metoder da jeg implementerede selection, men mente ikke de nævnte unøjagtigheder retfærdiggjorde at bruge tid på den farve-baserede selection. Det virker usandsynligt at brugeren nogensinde skulle komme ud for det scenarium at der klikkes på offscreen objekter, og programmet bruger ikke scrollbars. Som konsollen af og til giver udtryk for forekommer dobbelthits derfor hos overlappende objekter. I disse tilfælde udvælger implementeringen automatisk det objekt, der blev tegnet sidst, fordi det ligger øverst i den datastruktur, der indeholder objekterne (som under tegning gennemløbes fra start til slut). Man kunne naturligvis have gjort et forsøg på at løse tvetydigheder ved multiple hits ved

at spørge brugeren, men programmer der er for snakkesagelige på den måde generer mere end de hjælper, så det accepteres at programmet træffer valget for én.

Før selection implementeringen kan kaldes helt afsluttet er der en sidste beslutning der fortjener opmærksomhed.
Programmet benytter nemlig samme selection-funktion til både quads og bygninger, men de to typer objekter tilhører hver sin datastruktur, så for at fortælle programmet hvilken datastruktur det skal tilgå som funktion af en selection, var det nødvendigt med en måde at bestemme hvilken slags, brugeren klikkede på.
Problemet løses simpelt ved simpelthen at tilføje 100 til alle navne, der tilhører bygninger, når disse loades på namestacken. Herefter tolker programmet alle selection hits, hvis navn er over 100 som en bygning, og alle derunder som quads. Det betyder selvfølgelig at der opstår alvorlige problemer hvis en quad får tildelt et navn *over* 100, men før det sker skal der i forvejen være tegnet 100 quads i grundplanet.
Fra et design-standpunkt forekommer det helt utænkeligt at nogen nogensinde skulle finde på at tegne over 100 quads før vedkommende ekstruderer blot én af dem til en bygning og på den måde frigør navnet, så på dette grundlag accepteres løsningen uden videre sikkerhedsforanstaltninger.

3.8 Lagring og tegning af flere bygninger

De tidligere nævnte datastrukturer, der i programmet indeholder oprettet geometri kommenteres nu mere udførligt. Nærmere bestemt er der tale om to strukturer af typen `vector`. De er at finde i klassen Drawing, men alle tilføjelser og fjernelser fra beholderne foregår via pointeren i Navigator, fordi de er resultatet af brugerinput, og de behandles som bekendt af Navigator. Så snart museknappen slippes enten under tegning af en quad eller ekstrudering af en bygning, betragtes geometrien som færdigtegnet, og tilføjes. Ligeledes slettes objektet, hvis der efter objektet er udvalgt trykkes delete.

Under hver gentegning (eksekvering af `display`) gennemløbes de to vektorer og alle objekter tegnes. Hvor ansvaret for at tegne de enkelte sider af bygninger ligger, afhænger af hvorvidt siderne er teksturerede eller ej, så den funktionalitet vil vi vende tilbage til efter afsnittene om teksturering.

Texture Mapping

3.9 Teksturering i projektet

Teksturering er en helt essentiel del af projektet, og ligesom tegne funktionaliteten er det vigtigt, at den implementeres med en vis mængde brugervenlighed for tanke. Det er målet at programmet skal kunne teksturere tilfredsstillende med så lidt arbejde fra brugeren som muligt. Denne målsætning kommer til at have kraftig indflydelse på de valg der træffes under implementeringen af programmets tekstureringsfaciliteter. Da jeg startede researcharbejdet forbundet med projektet havde jeg først i tankerne at bruge almindelig ortogonal-teksturering fordi projektiv teksturering forekom mig lidt overmodigt, projektets omfang taget i betragtning. Det skulle heldigvis senere vise sig at ændres, og vi skal i det følgende se hvordan projektiv teksturering med succes kan implementeres og bruges i et projekt som dette.

3.10 Ortogonal teksturering

Med almindelig, ortogonal teksturering påklæbes teksturen ganske simpelt overfladen med hardcodede teksturkoordinater. Man vil kunne implementere det indenfor overskuelig tid og ualmindelig simpelt ved at tegne al geometri med teksturkoordinater og så slå teksturering til og fra vha. OpenGL state, når tegnesystemet informeres om at den pågældende flade er tekstureret.

Der er desværre nogle invaliderende komplikationer man vil være nødt til at tage højde for med denne teknik. For det første er det ikke sandsynligt, at det billede, brugeren forsøger at bruge som tekstur er et perfekt billede af en matchende husflade. Husfacaden kan være en del af et større perspektiv, dvs. der kan være sceneri rundt om huset på billedet, som brugeren ikke ønsker på modellen. Derfor vil man være nødt til at supplere implementeringen med et display, hvor brugeren med et lasso-værktøj kan udvælge rektangulære, akse-orienterede segmenter, og så bruge `glCopyTex2D` til at udvælge den del, af det sekundære vindues framebuffer, brugeren ønsker at bruge som tekstur.

Billedet kan også være taget med en anden perspektivisk projektion end ortogonal. De eneste typer projektioner der vil være brugbare til ortogonal teksturering er faktisk ortografiske, eller måske et såkaldt one-point-perspective, som forklaret i Edward Angels bog, "Interactive Computer Graphics".
Enhver anden projektion i billedet vil resultere i 2D-repræsentationer af husfladen, som ikke flugter med billedets lodrette og vandrette koordinatakser. Hvordan skulle man kunne udskære den del af billedet, der så svarer til husfacaden? Og selv hvis det lykkedes, hvad sker der så med billedsegmentet når det påklæbes modellen? Det vil utænkeligt blive udsat for forvrængninger og skaleringer, der til syvende og sidst resulterer i en visuelt utilfredsstillende teksturering.

Endegyldigt er der alt for mange lammende vanskeligheder behæftet almindelig ortogonalteksturering til at det kan anses for brugbart i dette projekt. Det stiller ganske enkelt for store krav til input-billederne, og må konkluderes at være for simpel en teknik.

3.11 Projektiv teksturering

Projektiv teksturering har været kendt siden 1978, så der er ikke noget revolutionerende ved teknikken. Cass Everitt fra Nvidia har skrevet en rigtig god artikel [5] om emnet som jeg støttede mig meget opad under implementeringen. Teknikken har været brugt flere forskellige steder, eksempelvis til lommelygten i ID Softwares Doom III, hvor en lys, cirkulær tekstur bruges til at belyse scenen fremefter, svarende til billedet nedenfor.

Figur 9: Lommelygten i billedet er et eksempel på projektiv teksturering fra ID Softwares Doom III.

Idéen er matematisk at modellere den måde, et lysbilledapparat fungerer på, så geometri, der er i vejen for projektorens synsvidde bliver ”belyst” af teksturen i stedet for eksplicit at have defineret teksturkoordinater, som det var tilfældet i forrige afsnit. Dette betyder naturligvis at teksturkoordinaterne skal kalkuleres interaktivt, men matematikken bag det er heldigvis forholdsvis simpel. En projektor fungerer nemlig komplet analog til den velkendte model for et kamera. (Den er essentielt et ”omvendt” kamera). Derfor er de ligninger, der modellerer projektoren også næsten de samme, som dem der modellerer et kamera:

$$T_o = \begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} P_p V_p M \qquad [5]$$

$$T_e = \begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} P_p V_p V_e^{-1} \qquad [5]$$

, hvor P_p er projektorens projektionsmatrix, V_p dens viewing matrix, M OpenGLs modelmatrix og V_e^{-1} den inverse af kameraets view matrix, som bruges fordi vi har behov for at transformere vertex koordinater tilbage fra eye-space til world-space. Den skalering og translation der finder sted til sidst skyldes, at teksturkoordinater skal mappes til intervallet [0;1], mens vertex-koordinater (som teksturmatricen T påganges) ligger afbilledet i et interval afhængig af dybdeskalaen, typisk [-1;1].

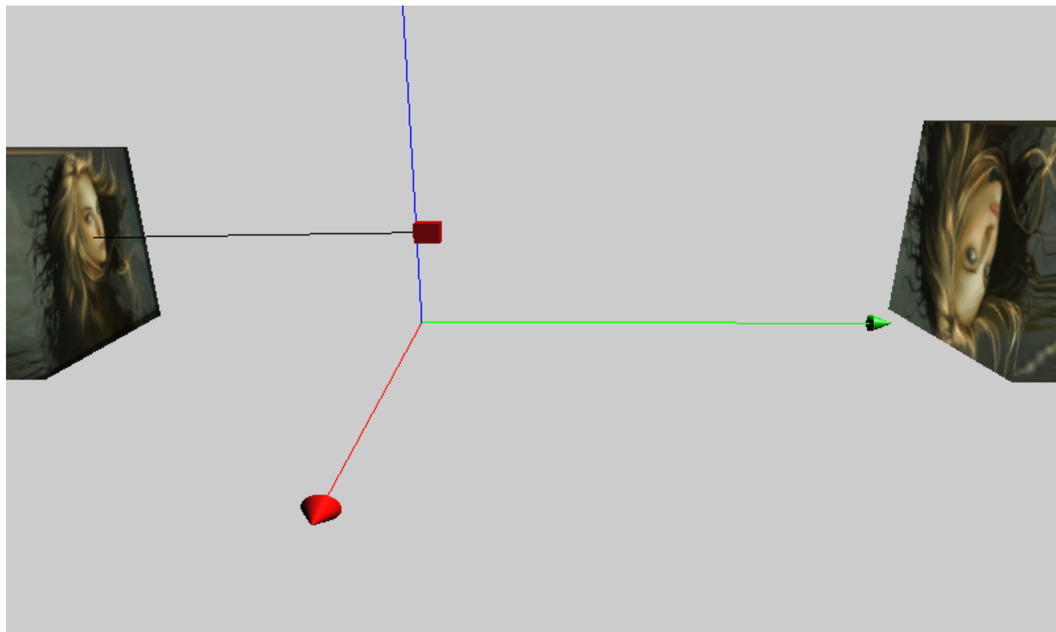
Projektiv teksturering kan gøres enten ”eye-linear” eller ”object-linear”, hvilket er en reference til det koordinat system teksturkoordinaterne udregnes i. Ved eye-linearity er koordinaterne relativ til kameraet, og ændres derfor hvis kameraet på nogen måde ændres. Resultatet er at objekter modtager tekstur på en måde der får dem til at ”svømme” gennem teksturen, hvis de bevæger sig gennem projektorens synsvidde [6].
Ved object-linearity er koordinaterne relativ til objektet, og vil opføre sig som om de var naglet til objektet. Tekstureringen er derfor statisk ved animation af geometrien.
De to ligninger ovenfor er til udregning af tekstur matricer henholdsvis til object-linear (T_0) og eye-linear (T_e) koordinater.

OpenGL hjælper heldigvis en hel del med de matricer jeg skal bruge i ligningen, eftersom projektorens projektionsmatrix kan sættes med `gluPerspective` og viewmatricen med `gluLookAt`. Implementeringen undgår helt den slemmeste udregning, nemlig den inverse matrix der er behov for. Hvordan vil jeg komme tilbage til senere.

Der er to metoder jeg umiddelbart kender til som kan udføre den endelige udregning. Jeg har implementeret begge i løbet af forløbet, men endte med at droppe den første, hvorfor den kun gennemgås kort i de følgende to underafsnit.

3.11.1 OpenGLs egen texture generation facilitet

Før i tiden opnåedes automatisk generering af teksturkoordinater i OpenGL med adskillige kald til `glTexGen*` hvor referenceplaner, der udgør rækkerne i OpenGLs texture matrix, oprettes til samtlige 4 teksturkoordinater, og begyndelsesvist sættes til identitet.
Denne metode forlod jeg umiddelbart efter dens implementering. Grunden er primært, at brugen af OpenGLs faciliteter i dette tilfælde afskriver programmøren kontrol over selve udregningen af teksturkoordinater, og det gør det svært at håndtere invers projektionen, som er en ofte uønsket feature ved projektiv teksturering. Se figur.



Figur 10: Eksempel på invers projektion. Projektoren er repræsenteret ved den lille røde kube, og dens retning ved linien pegende mod venstre i biledet. Bygningen til højre modtager fejlagtigt tekstur bag projektoren.

3.11.2 Projektiv teksturering med GLSL

I stedet for at overlade udregningen til `glTexGen*` kan man skrive en mindre shader i GLSL, der løser opgaven for én. GLSL har built-in adgang til alle de matricer, der indgår i ligningen, så hvis man vil, kan man foretage matrixmultiplikationen her også, men jeg har dog valgt at bruge OpenGLs standard matrixmanipulerende kald til at opsætte projektoren først, og tilgår herefter teksturmatricen ved `gl_TextureMatrix[0]` i GLSL. Det er muligt der er lidt (minimalt) at hente

fra et optimeringsstandpunkt ved at lade GPU'en konkatenere matricerne for én i stedet for gøre det i værtsprogrammet, men dette projekt er ikke som sådan en realtidsapplikation, så jeg fandt det ikke nødvendigt at bekymre mig synderligt om det.

Der er en farlig kilde til forvirring i Everitts artikel, der omhandler den inverse matrix, der beskrives som nødvendig. For det første indebærer det, at skaffe en invers en masse uelegante udregninger og konverteringer, og for det andet ville man skulle sikre sig at, den matrix man inverterer kun indeholder kameraets viewing transformationer, og altså ikke også modeltransformationer, som det er tilfældet med OpenGLs modelviewmatrix. Man kan let komme ud for at spilde en masse tid med at fundere over hvordan det problem løses så man stringent kan opstille den ligning Everitt beskriver, men heldigvis er det slet ikke nødvendigt.
Ligningerne fra [5] er nemlig ud fra den antagelse, at de vertex-koordinater, man påganger T befinder sig i eye-space, og skal mappes tilbage til world space. Det er derfor kameraets inverse viewing matrix overhovedet skulle bruges, men den tilstand vertex-shaderen modtager vertex-koordinater i er jo i forvejen i world space, hvilket fuldstændig fritager os besværet med inverteringer.

Som en sidste fordel ved den GLSL-baserede løsning kan nævnes, at den spejlvendte, revers projektion nævnt i forrige afsnit kan fjernes, fordi vi har fuld kontrol over hvordan de udregnede teksturkoordinater bruges i fragment shaderen. Det forholder sig nemlig sådan, at det homogene teksturkoordinat q er negativt omme bagved projektoren. Derfor bruges den valgte tekstur til at sætte farven på fragmenter, hvor q er positivt, og resultatet er en projektor, der kun kan "skyde" fremad.

3.12 Automatisk positionering af projektoren

Det har hele tiden været en del af visionen omkring tekstureringsarbejdet, at det skulle forsøges lettet på en måde, og indtil videre foreligger der stadig en del "klikkeri" med både mus og keyboard for henholdsvis at orientere sig og skubbe projektoren derhen, hvorfra man ønsker tekstureringen skal finde sted. Det ville være ønskeligt med en måde at få programmet til at flytte projektoren automatisk på, så den står, hvis ikke præcist, så i det mindste nogenlunde hvor brugeren kan tænkes at ville have placeret den selv. Med andre ord efterlyses altså en algoritme, der kan foretage et kvalificeret gæt på projiceringspositionen. Så kan det endelige fintunings-arbejde overlades til brugeren uden at bebyrde vedkommende med at flytte projektoren større distancer mellem tekstureringer.
For at imødekomme dette krav er der i programmet implementeret en simpel løsning, der med et enkelt klik flytter projektoren hen foran den side, på hvilken brugeren klikkede. Metoden er banal og diskuteres derfor ikke i detaljer, men kort forklaret positioneres projektoren ortogonalt ud fra en sides centrum, en afstand fra siden afhængig af det tal, der på tidspunktet for museklikket står i den tilhørende spinner i programmets øverste højre hjørne.
Herfra kan brugeren rotere projektoren relativt til bygningens centrum med musen, og foretage finjusteringer med knapperne nederst i interfacet.

3.13 Teksturering, endeligt

Nu da programmets tegnefunktionalitet og tekstureringsfaciliteter er helt på plads kan det endelig diskuteres, hvordan teksturer fastsættes bygninger, så det, der oprindeligt var projektets mål, nemlig det at kunne opbygge et virtuelt urbant miljø, kan nås. Det er nu blevet tid til at vende tilbage til den struct, som er en del af Quad klassen og blev kort berørt tilbage i kapitel 3. Som nævnt indeholder den data forbundet med projektoren. Nærmere fortalt er der tale om to vektorer, som hver repræsenterer en position og en retning for projektoren. Derudover er der også en heltalsværdi som er ID for det texture object, der indeholder den tekstur, som hører til den pågældende quad. Disse bruges til at flytte rundt på projektoren i det splitsekund en tekstureret quad tegnes, så programmet husker altså på denne måde hvor projektoren var henne, da brugeren i sin tid brugte positionen til at teksturere quaden. Slutteligt indeholder structen en boolean, som sættes alt efter hvorvidt quaden overhovedet er tekstureret.
Tillige er det også tid til at diskutere ansvaret for tegning af bygningers enkelte sider nu. Drawing har to funktioner til gennemløb af den af sine vectors, der indeholder bygninger. Funktionerne er kun marginalt forskellige, idet de i bund og grund indeholder kald til samme tegnefunktionalitet, og forskellen består kun i, at den ene tager sig af at tegne sider med tekstur og den anden sider uden tekstur. Begge funktioner kaldes i `mains display`-funktion, hvor de er henført til som hvert sit "pass". Terminologisk indebærer det gentegning af identisk geometri, blot under forskellige omstændigheder når algoritmer benytter sig af multiple "passes". Det er ikke helt tilfældet her, fordi funktionen, der tegner teksturerede sider holder sig fra sider uden tekstur, og omvendt. Derfor spilder programmet ikke ressourcer på dobbelttegning; der er kun et mindre overhead behæftet det dobbelte gennemløb af vectoren.

Den måde at gribe tingene an på valgtes fordi det resulterer i fuld kontrol over hvilke sider, der modtager tekstur på brugerens anmodning, og hvilke der ikke gør. Det løser på en elegant måde et problem med projektiv teksturering som opstår, når geometri foran projektoren og i dens skudvidde modtager tekstur uden de skulle have haft det. Det skyldes, af geometri ikke skygger for hinanden når de projektivt tekstureres. Al geometri, fuldstændig uden hensyntagen til deres dybde set fra projektoren, bliver tekstureret hvis det er indenfor projektorens frustum, og det er bestemt ikke ønskeligt når en bygning er "færdig" og har fået forskellige teksturer fastsat siderne af brugeren. Det ville være en vanskelig udfordring (og sikkert kræve en teknik, der minder om shadow mapping) at bestemme hvorvidt geometri ligger bagved anden geometri når siderne i bygninger fra vectoren skal tegnes. Men det undgår vi altså helt at bekymre os om ved at tegne siderne med dedikerede projektordata på denne måde.

Kapitel 4

Evaluering

Dette kapitel betragter den skrevne kode kvalitativt. Formålet i kapitlet er at verificere programmets funktionalitet og, hvis noget ikke virker, diskutere hvorfor. Ligeledes stilles spørgsmålstegn ved om de dele, der opfylder kravspecifikationen gør det på den mest optimale måde.

4.1 Kort om testene

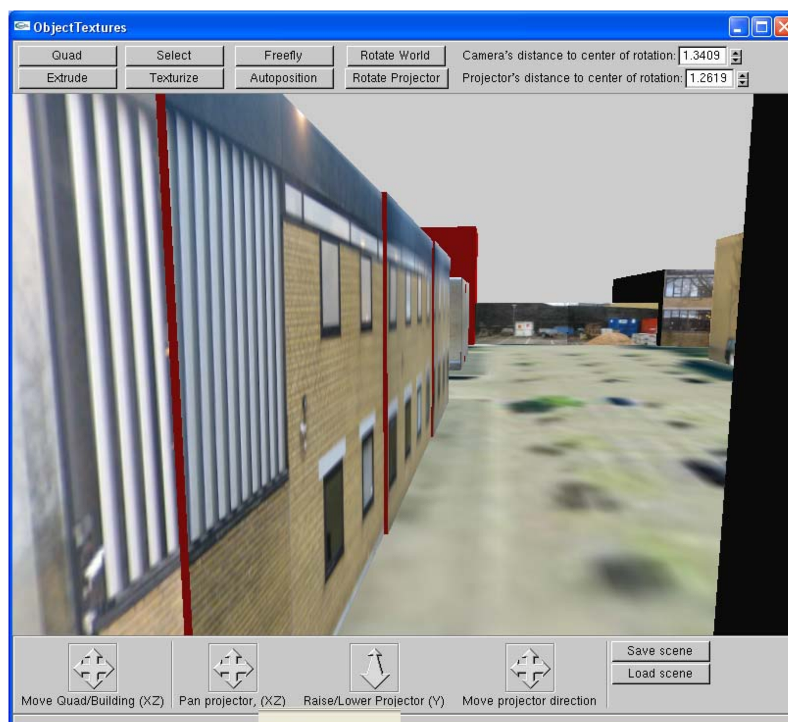
I ethvert foretagende der involverer et programmeringsarbejde er det selvfølgelig nødvendigt at sikre programmets funktionalitet, så projektet skal naturligvis indebære en form for test af den skrevne kode. Fra et softwareengineering-synspunkt taler man typisk om to forskellige typer tests: Strukturel- og funktionel test. Undertiden kaldes de også ”whitebox” og ”blackbox” som henførelse til synligheden af den testede kode under udførelsen. En strukturel/whitebox test er et enormt analytisk arbejde, der kræver kørsel af alle tænkelige tilstande af alle statements. (Altså sand/falsk evaluering af alle if-sætninger, eksekveringer 0, 1 og flere gange af alle for-løkker, osv). Herefter holdes resultatet op imod en tabel af forventede testresultater. Stemmer det faktiske testresultat ikke overens med det forventede betragtes testen som fejlslagen og en omskrivning af den testede kode må overvejes.
Funktionel/blackbox test gennemgår ikke slavisk alle tilstande af programmet, men tester i stedet programmets funktioner fra brugerens perspektiv. Programmet forsøges således brugt på så mange måder som man umiddelbart kan tænke sig til, (sandsynlige såvel som usandsynlige), for at afdække så stor en flade af brugsmåder som muligt.

Testen i dette projekt er fuldstændig funktionel. En strukturel ville ganske enkelt medføre et mere massivt stykke arbejde i forbindelse med oprettelse af tabeller af forventede resultater for ikke at nævne udførelse af selve testen end den afsatte tid tillod. Derfor prioriteredes i stedet blot at bruge programmet som man tænker sig en almindelig bruger ville gøre det, og så observere om programmet opfører sig tilfredsstillende eller ej. Så kan man efterfølgende diskutere hvorfor eller hvorfor ikke, hvilket undertegnede finder mere meningsfyldt end blot at sammenholde rå testdata.

Kapitlet her indeholder derfor en serie diskussioner af forskellige dele af programmets funktionalitet samt et forsvar af de implementeringsmæssige beslutninger. Det belyses hvorvidt der forekommer deciderede grafiske fejl (artifacts), valg af datastrukturer, samt en diskussion omkring programmets algoritmiske tilgang.

4.2 Grafiske Artifacts

Der er nogle konsekvenser ved den brugte metode der ikke umiddelbart kan afhjælpes, som man blot må acceptere. Se for eksempel følgende billede:



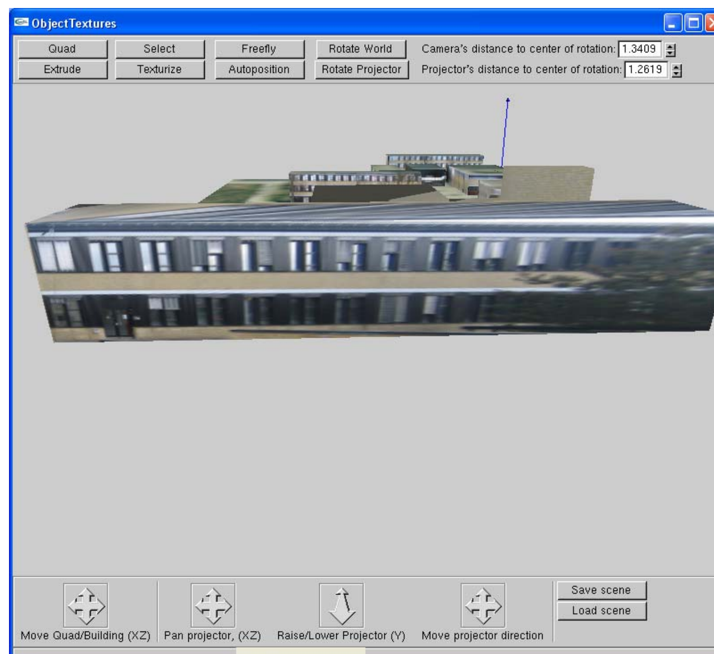
Figur 11: Scene fra gangen mellem bygning 303 og 302, restaureret. De røde linieri muren i bygning 303 til venstre markerer, hvor de enkelte segmenter adskilles.

Det centrale i billedet er de lodrette, røde linier i muren i bygningen til venstre. De forekommer, fordi det var nødvendigt at konstruere bygningen ud fra mange små segmenter, som man så med noget rimelig tungt pillearbejde stykker sammen. Grunden er, at de billeder der er taget i området ikke kan overskue hele husfacaden på én gang, partielt fordi de fleste billeder måtte tages i trange passager hvor den nødvendige afstand til muren ikke kunne opnås, partielt fordi et almindeligt, kommercielt digitalkamera ganske enkelt har sine begrænsninger. (Kameraets objektiv var ikke ligefrem vidvinkel). Resultatet er, at en bygning pålægges tekstur lidt efter lidt, efterhånden som et passende segment kunne tilføjes den overordnede struktur. De røde linier er små uregelmæssigheder i segmenternes dimensioner, der er mest tydelige når scenen beskues langs muren, som det er tilfældet ovenfor. Der er også andre konsekvenser end de røde linier – Læg mærke til forskellen i murstenenes farve mellem segmenterne yderst til venstre. Det ene segment har markant lysere mursten end det andet, fordi samme lysforhold mellem forskellige fotografier er svære at replicere. Slutteligt kommer selvfølgelig det faktum, at forskellige billeder kun tilnærmelsesvist passer sammen. Et ”100% fit” er virtuelt umuligt at opnå.

Når den nødvendige afstand til en husfacade ikke kan opnås kan en løsning på ovenstående problem være at stille sig tættere på muren og bygningens gavl, og tage et billede ned langs bygningens side, så man får et perspektivisk billede som dette:

Figur 12: Billede af bygning 305, taget perspektivisk i stedet for ortogonalt, for at inkludere hele bygningens facade.

Herefter oprettes en bygning i fuld størrelse, og man indstiller projektoren til at belyse bygningen fra den samme position, som billedet ovenfor er taget. Så får man noget, der ligner følgende resultat:



Figur 13: Teksturering af bygning i fuld størrelse med perspektivisk tekstur

Det står hurtigt klart, at den løsning ikke er kvalitativt holdbar, heller. Man får ganske vist en bygning i fuld størrelse som er lettere at håndtere hvis man ønsker at flytte på den, rotere osv. og man slipper også for de afslørende kanter og farveforskelle mellem segmenter. Til gengæld får man en facade, der er skarpt tekstureret i den ene ende, men uklart og udtværet i den anden. Dette sker, ikke overraskende, fordi der er en større og større uoverensstemmelse mellem antallet af texels til rådighed per antal pixels jo længere man kommer væk fra projektoren, så ved bygningens bagende oplever man at mag-filteret smører texels udover uacceptabelt mange pixels. Dette er ækvivalent med ”The deer-in-headlights-effect” beskrevet af Everitt i [5].

En interessant måde at løse problemet på kunne være at tage et billede fra hver side at den ønskede facade, og blende teksturerne sammen i en shader før facaden endeligt tekstureres. Idéen er, at den procentdel af hver tekstur, man bruger i en given situation skal afhænge af kameraets position i forhold til projektoren. Jo tættere på en bestemt projektor kameraet står, jo større procentdel af tekstur fra netop den projektor bruges i blendingen.
Resultatet forestilles at ville være en ensartet bygning uden brug af segmenteret konstruktion, og uden alt for synlig uskarphed i bygningens ender. Midten af bygningen ville stadig lide lidt under problemet, fordi den ville modtage 50% af hver tekstur, altså lige dele uklarhed.

4.3 Benyttede datastrukturer

Som det er blevet nævnt gennem rapporten benytter programmet sig af vectorer til at indeholde bygninger og quads. Dette er gjort hovedsageligt fordi det ikke vides på forhånd hvor mange bygninger/quads programmet kan tænkes at skulle indeholde, så dette, i kombination med behovet for at indsætte og slette elementer, gjorde den dynamisk voksende natur af vectors tillokkende.
Vectors har nogle svagheder som bør tages i betragtning før den vælges. Til projektets formål passer en vector fint fordi den er en rimelig high level datastruktur, og den tilhørende funktionalitet på forhånd implementeret i C++ STL var belejlig at bruge. En vector har rigtig mange ting til fælles med en traditionel stack, afsløret ved tilstedeværelsen af member-funktioner som ”`push_back`” og ”`pop_back`”, og den opfører sig også på mange måder som én, men sammenlignet med statiske arrays indeholder den et mindre overhead. Tillige skal det siges at projektet her benytter sig af member-funktionen `erase`, når bygninger og quads udvalgt af brugeren fjernes ved tryk på delete. Denne funktion er umådeholdent dyr, fordi en sletning medfører at skifte samtlige elementer i vectoren ned et antal pladser svarende til de slettede elementer, så vectorens indicer forbliver sammenhængende, og den til vectoren afsatte hukommelse ikke fragmenteres.

Derfor ville en vector ikke nødvendigvis være fornuftig at bruge i en realtidsgrafisk sammenhæng. Med nutidens computere er vi efterhånden nået frem til at hukommelse er en billigere ressource end rå processorkraft. Så i realtidsgrafik ville man

nok reservere plads med et statisk array, så tilstrækkelig stort at det forekommer utænkeligt at det skulle blive opbrugt.

Til alle andre ting som kræver datastrukturer bruges statiske arrays, fordi antallet af elementer er kendt, eller fordi de kræves af ekstern API. Der er tale om ting som små arrays til GLUIs live-variables, eller en midlertidig beholder til en bygnings 6 sider.

4.4 Algoritmisk tilgang

Det er blevet tid til at vende tilbage til de gennem rapporten lovede forklaringer på diverse fordele og ulemper ved de algoritmer der er blevet brugt.

4.4.1 Zenith Rotation

Den første metode, vi skal diskutere kvaliteten af, er den der bruges til oprettelse af rotationsmatrix til zenith rotation. For at opsummere kort, lånes modelviewmatricen til oprettelsen, fordi det giver anledning til brug af `glRotatef`, som sætter matricen. Dette er der ikke direkte noget galt i, tværtimod er det udmærket at benytte sig af OpenGLs funktionalitet så vidt muligt til at hjælpe med den slags kalkuleringer. Derimod er det farligt og typisk en elendig idé at bruge kald, der anmoder om værdien af OpenGL tilstandsvariable, fordi de kan ligge i videohukommelse, og enhver trafik mellem grafik og main memory bør så vidt muligt begrænses. Faktisk anbefales det slet ikke at bruge kald til `glGetIntegerv/Floatv/Doublev` udenfor debugging kontekst og nødvendige bestemmelser af hardware-begrænsninger. [11]
Det virker imidlertid som en lidt bombastisk sat begrænsning, fordi der uden problemer kan foregå tilstandsanmodninger i Z-picking algoritmen. Dette er tilladeligt fra et realtids-synspunkt fordi Z-picking ikke finder sted i realtime, men en enkelt gang, når der klikkes med musen for at bestemme siden på en bygning.
Rotation, til gengæld, er realtime, og hvis der her havde været tale om en større produktion skulle rotationsmatricen helt klart have været oprettet manuelt og uden nødvendigt kald til `glGetFloatv`. Som det er nu lader kaldene heldigvis ikke til at gå synderligt udover programmets framerate når der roteres, selv i større miljøer med mange bygninger.

4.4.2 Højdebestemmelse ved ray-casting

Algoritmens brug af vinklen mellem de to stråler er grundlæggende baseret på en fornuftig nok tanke, men bruges desværre ikke helt korrekt i programmet. Dette skyldes at algoritmen *direkte* bruger vinklen, uden at tage højde for hvor den øverste stråle skydes hen i forhold til den nederste. Dette betyder at man kan ekstrudere en bygning ved at føre musen ud til siden eller nedad, for den sags skyld - Implementeringen er ligeglad med hvor, bare der opstår en vinkel mellem de to stråler den kan måle på.
Man kunne løse problemet med et par krumspring, for eksempel ved at tildele den øverste stråle samme x-koordinat i skærmen som den nederste, altså ignorere gen-

nemløb af horisontale pixels. Så vil det ingen effekt have at føre musen ud til siden og bygningen vil kun ekstruderes ved at ændre musens y-koordinat. Ligeledes kunne problemet i negativ y-retning løses, ved at gemme det første skærmkoordinat der trykkedes på, og ignorere forsøg på at ekstrudere hvis efterfølgende y-koordinater på der tilbagelægges på skærmen er større end det første. (OpenGL har y = 0 øverst på skærmen).
De nævnte spidsfindigheder resulterer ikke som sådan i deciderede fejl, men kan nærmere siges at være pudsige, uønskede 'features' ved algoritmen, så udover et minutiøst irritationsmoment udgør de ikke noget trussel. Der er ikke foretaget forsøg på at udbedre dem, da andre ting på tiden havde større prioritet.

4.4.3 Picking – Svagheder ved begge algoritmer

De forskellige picking algoritmer brugt i programmet er heller ikke uden deres fejl. Den måde de forskellige algoritmer virker på er, som vi diskuterede i kapitel 3, i første omgang grunden til overhovedet at bruge to forskellige algoritmer, men raycasting er kun ideelt i dette program fordi programmet antager at al tegning foregår i planen. Dette er en kraftig simplificering og et professionelt værktøj ville sandsynligvis understøtte oprettelse af bygninger på et ujævnt terræn. Her kommer raycasting til kort, fordi det plan, der skæres med ikke længere kendes. (Hverken normalvektoren eller afstand til origo). Et punkt vil stadig kunne findes, fordi man blot ville kunne udregne skæring mellem strålen og den geometri terrænet består af. (Ray-triangle intersection[12]). Men det introducerer en hel verden af problemer i forbindelse med optimering af skæringsalgoritmer som slet ikke er nødvendig at besvære sig med når Z-picking løser problemet lettere.

Da arbejdet med implementeringen nåede til at måtte bestemme sider på bygninger blev Z-picking taget i brug igen, men her opdagedes til gengæld én af svaghederne behæftet dén algoritme.
Z-picking bestemmer nemlig slet ikke numerisk korrekte punkter. Fejlen blev opdaget ved at klikke på en bygnings underside. Her vides med sikkerhed, at koordinatets y-værdi er eksakt 0, fordi det eksplicit sættes til 0 af Quads constructor, men Z-picking bestemte kun punktets y-værdi tilnærmelsesvist, typisk med en fejl ude på 3. decimal. Fejlene tænkes at opstå når Z-picking mapper punkter tilbage fra screen- til world-space, hvorved floating point afrundinger forårsager unøjagtighederne. Fejlen er alvorlig fordi den umuliggør succesrig sammenligning med punkterne i den bygning der er klikket på, og så kan den ønskede side ikke findes. For at råde bod sammenlignes punktet med sidernes hjørner, plus/minus et tal, epsilon, på 0,02, for at introducere en gavmild fejlmargin til søgealgoritmen. Herefter virker det tilfredsstillende stabilt.

4.5 Virker programmet tilfredsstillende?

Det korte svar er ja. Som det fremgår af billederne i Appendix A, hvor en stor del af Danmarks Tekniske Universitets 3. kvadrant er restaureret, kan programmet fint bruges til at oprette større miljøer. Brugen af programmet afslørede dog nogle få bugs af større og mindre grad. Da kapitlet gør det ud for et testafsnit virker det passende at afslutte med en gennemgang af de bugs, der stadig er at finde i programmet, samt en kommentar om hvad der er gjort ved dem.

Out of range-exceptions

Programmet lider under out of range-exceptions, som bliver kastet af de vectors programmet bruger til at opbevare bygninger og quads. Problemet opstår på grund af samspil mellem selection-algoritmen, og kald til vectors på baggrund af selections. Selections skulle gerne tilsvare indices i de vectors der indeholder strukturerne, men der findes tilfælde, hvor programmets indtryk af, hvad der er den aktuelle selection er forkert. Dette kan få programmet til at foretage anmodninger på indices, der er større end vectoren, og så går programmet ned.
Fejlen er forsøgt rettet med nogen succes, men optræder desværre stadig lejlighedsvist.

Oprettelse af quads/bygninger

Geometri fejloprettes af og til, i quads tilfælde formodentligt fordi hjørnevariablene a og c ikke gensættes ordentligt. I bygningers tilfælde springer bygninger nogle gang til uønskede højder i stedet for at ekstruderes normalt. Fejlen er sjælden og udgør en bagatel. Den er ikke forsøgt rettet.

Teksturering af bygninger

Når en projektor er korrekt indstillet og brugeren klikker på en side for at fastsætte tekstureringen, slår sætningen af projektor-data til den pågældende side undertiden fejl. Fejlen opstår i særdeleshed når gavle forsøges tekstureret på segmenterede bygninger, fordi selection-rutinen registrerer hits på samtlige segmenter ned gennem bygningen og efterfølgende har svært ved at sætte aktuelle selection til det segment der indeholder gavlen, som brugeren ønsker. Sættes selection forkert, slår bestemmelsen af siden der udgør gavlen fejl, og så kan projektordata ikke sættes.
Fejlen er ikke forsøgt rettet pga. tidspres, men udgør et alvorligt irritationsmoment. Der er pt. ikke andet for end at klikke lidt rundt på siden indtil selection-rutinen rammer det rigtige segment og tekstureringen kan finde sted.

Rotation af bygninger

Bygningerne kan roteres i programmet ved først at udvælger dem og herefter dreje dem enten højre eller venstre om med piletasterne. Desværre er rotation noget man må vente med til man er tilfreds med både bygninges størrelse og de tekstureringer man måtte have ophæftet siderne. Ændring af bygningens dimensioner og teksturering af dens sider er nemlig ikke mulig efter rotation har fundet sted.
Fejlen er absolut alvorlig og bør rettes, men blev opdaget meget sent i forløbet og der er således ikke gjort noget forsøg på at udbedre den.

Kapitel 5

Diskussion

Formålet med dette kapitel er at diskutere hvilken af de foreslåede metoder, der fungerer bedst, hvad der kan gøres bedre, og hvad eventuelle planer for fremtiden er. Kapitlet stiller og besvarer spørgsmålet om, hvordan der kan fortsættes herfra.

5.1 Teksturering: Hvad er bedst?

Der kan ikke være nogen tvivl om at blandt de to metoder der er diskuteret her er projektiv teksturering langt den bedste, overvejende af grunde allerede gennemgået i kapitel 3. Som tilknyttet kommentar kan man sige, at projektiv teksturering også tillader brugeren at se bygningen med tekstur på flere forskellige sider, før der træffes endelig afgørelse om at bruge teksturen til den pågældende bygning. Med ortogonal-teksturering ville man kun kunne lægge tekstur på en enkelt siden ad gangen, og det gør det sværere at skaffe brugeren et visuelt indtryk under udarbejdningen af en scene.

5.2 Fremtidige Udvidelser: Canoma Pinning Teknikken

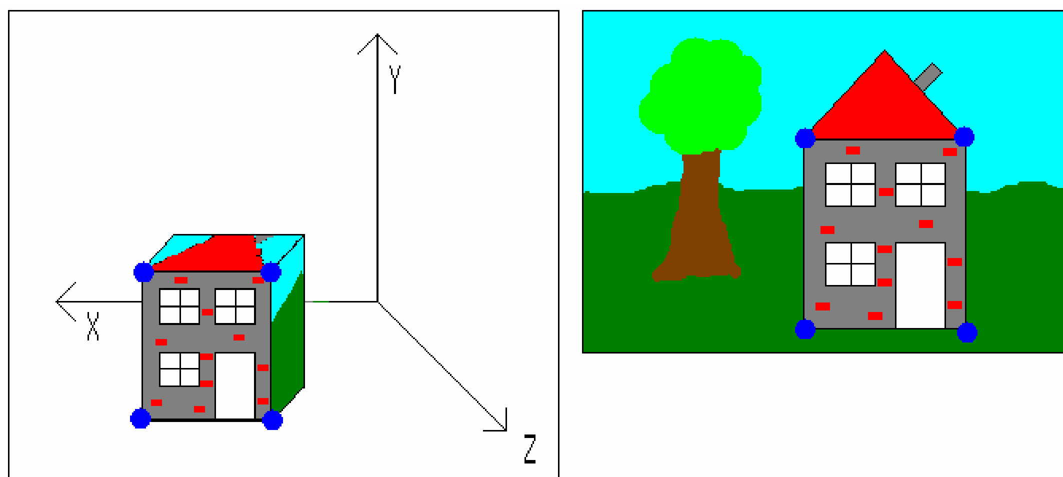
Metoden til automatisk positionering beskrevet i kapitel 3 fungerer ganske vist tilfredsstillende, primært fordi de billeder der i projektet blev brugt er taget ortogonalt, så metodens ortogonale positionering er passende. Men den vil ikke være tilstrækkelig i den generelle case, fordi billeder sagtens kan være taget perspektivisk. (Hen langs hele en bygnings facade). Det implementerede er ganske enkelt den ”billige løsning”.
Hen mod slutningen af projektets forløb blev der stillet er interessant forslag til hvordan positioneringen også kunne finde sted automatisk, så selve flytningen af projektoren fra én bygning til en anden blev udført langt mere præcist af programmet selv. Tanken er baseret på pinning-funktionaliteten i Canoma, som blev omtalt i introduktionen.

Teknikken går ud på at lade brugeren klikke på koordinater i det vindue, den valgte tekstur vises i. Disse koordinater udgør teksturkoordinater. Herefter klikker brugeren på positioner i hovedvinduet, som udgør de tilsvarende verdens-koordinater. Fra afsnit 3.10 kender vi ligningen for teksturkoordinater baseret på verdenskoordinater:

$$\begin{pmatrix} s \\ t \\ r \\ q \end{pmatrix} = T_e \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}_e$$

, hvor $T_e = \begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} P_p V_p$, som beskrevet i 3.10, men med den inverse kamera-viewing matrix fjernet, da den ikke er nødvendig i denne implementering.

De koordinater brugeren specificerer, udgør henholdsvis (s, t, r, q) og (x, y, z, w), og tanken er at man regner sig frem til teksturmatricen T_e, som så kan bruges til at positionere og indstille projektoren. Præcisionen af algoritmen er derefter kun begrænset af, hvor præcist brugeren kan klikke på hjørner af bygninger i de to vinduer:



Figur 14: De blå prikker i ovenstående markerer de af brugeren valgte teksturkoordinater til højre, og de tilsvarende verdenskoordinater til venstre.

Uheldigvis er den bagvedliggende matematik på ingen måde triviel. Der er tale om et image warp, som beskrevet i [13], der behandler matematikken bag warping af billeder mellem rum af samme dimension. Følgende matematiske gennemgang er baseret på [13] og indeholder først en forklaring på hvordan den kan lade sig gøre i 2 dimensioner, dernæst et forsøg på konvertering til vores 3-dimensionelle situation.

I to dimensioner kan warpet opstilles på følgende måde, hvor koordinaterne $\begin{pmatrix} x \\ y \end{pmatrix}$ betegner koordinater i hovedvinduet, hvis ækvivalens i homogene koordinater er $\begin{pmatrix} x' \\ y' \\ w \end{pmatrix}$, mens de tilsvarende koordinater i teksturvinduet er $\begin{pmatrix} s \\ t \end{pmatrix}$ og $\begin{pmatrix} s' \\ t' \\ q \end{pmatrix}$. Som sædvanlig fås de reelle koordinater fra de homogene ved at dividere igennem med koordinatsættets homogene koordinat.
Warpet foretages nu med en 3x3-matrix, hvis elementer betegnes med bogstaverne a-i:

$P_d = M_{sd} P_s \Leftrightarrow \begin{pmatrix} x' \\ y' \\ w \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} s' \\ t' \\ q \end{pmatrix}$, hvor indices s og d står for "source" og "destination".

Hvis man udfører matrixmultiplikationen og dividerer igennem med det resulterende udtryk for q for at vende tilbage til formler for de reelle koordinater x og y får man, at:

$$x = \frac{as + bt + c}{gs + ht + i}, \text{ og } y = \frac{ds + et + f}{gs + ht + i}$$

Forsimpler man vores opgave til 2D står vi nu i den situation at have x og y samt s og t, og ønsker at finde frem til mapping-matricen, M_{sd}. Elementet i kan forventes at være 1, så vi behøver reelt kun opstille 8 ligninger for at finde de 8 ubekendte. Dette lader sig gøre hvis brugeren har specificeret 4 sæt af koordinater (x_k, y_k), (s_k og t_k), hvor $k \in \{0,1,2,3\}$, svarende til hjørnerne på en husfacade i teksturvinduet og i hovedvinduet jf. figur 14. Ganger man igennem med nævnerne i ovenstående to ligninger for x og y får man under antagelse af i = 1:

$$x_k = s_k a + t_k b + c - s_k x_k g - t_k x_k h$$
$$y_k = s_k d + t_k e + f - s_k y_k g - t_k y_k h$$

Med disse udtryk kan følgende 8x8-matrix nu opstilles og løses ved Gauss-elimination for elementerne i M_{sd}:

$$\begin{pmatrix} s_0 & t_0 & 1 & 0 & 0 & 0 & -sx_0 & -tx_0 \\ s_1 & t_1 & 1 & 0 & 0 & 0 & -sx_1 & -tx_1 \\ s_2 & t_2 & 1 & 0 & 0 & 0 & -sx_2 & -tx_2 \\ s_3 & t_3 & 1 & 0 & 0 & 0 & -sx_3 & -tx_3 \\ 0 & 0 & 0 & s & t & 1 & -sy_0 & -ty_0 \\ 0 & 0 & 0 & s & t & 1 & -sy_1 & -ty_1 \\ 0 & 0 & 0 & s & t & 1 & -sy_2 & -ty_2 \\ 0 & 0 & 0 & s & t & 1 & -sy_3 & -ty_4 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Så vidt så godt. Ovenstående udgør et kort resumé over artiklens udmærkede forklaring på udregning af mappingmatricen i 2D-2D warps, men vores situation er noget mere kompliceret end det, fordi vi er nødt til at regne i 3-dimensioner. Matematikken er naturligvis ækvivalent men vokser betragteligt under tilføjelse af den 3. dimension. Ovenstående koefficientmatrix er en 8x8 i 2 dimensioner, men under 3 dimensioner bliver det en 12x12-matrix. For at forsimple matematikken lidt foretages nogle indledende antagelser. Vi starter med først at erindres, at den ønskede mappingmatrix er teksturmatricen, som er en konkatenation:

$$T_e = \begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} P_p V_p$$

Hvis vi antager at projektionsmatricen P_p kan fastsættes af brugeren via interfacet, så skal vi kun finde projektorens viewingmatrix V_p, og så er problemet indsnævret til den gængse opfattelse af OpenGLs modelviewmatrix, uden modeltransformationer; det er jo det V_p i sagens natur er når den sættes med `gluLookAt`. Om modelviewmatricen ved vi, at den nederste række består af tallene 0, 0, 0 og 1. Benyttes en gennemgang ligesom for 2 dimensioner resulterer det i følgende ønskede V_p:

$$V_p = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Udføres mappingen på baggrund af kun denne matrix, altså opstilles transformationen uden projektionsmatricen og skaling/translation af teksturkoordinater, så vil man få følgende udtryk for x_k, y_k og z_k som funktion af elementerne i V_p:

$$x_k = as_k + bt_k + cr_k + d$$
$$y_k = es_k + ft_k + gr_k + h$$
$$z_k = is_k + jt_k + kr_k + l$$

Med 4 punkter kan følgende 12 ligninger nu dannes og opstilles på matrixform:

$$\begin{pmatrix} s_0 & t_0 & r_0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ s_1 & t_1 & r_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ s_2 & t_2 & r_2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ s_3 & t_3 & r_3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & s_0 & t_0 & r_0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & s_1 & t_1 & r_1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & s_2 & t_2 & r_2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & s_3 & t_3 & r_3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & s_0 & t_0 & r_0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & s_1 & t_1 & r_1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & s_2 & t_2 & r_2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & s_3 & t_3 & r_3 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ j \\ k \\ l \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \\ z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix}$$

Under normale omstændigheder ville det være løsningen på opgaven, og man ville kunne løse analytisk for V_p's elementer og implementere løsningen. Desværre løber vi her ind i et yderst alvorligt problem.

Vi kender ikke $r_{0\text{-}3}$. Disse værdier indikerer dybdeinformationen i teksturkoordinaterne, og de er jo udvalgt af brugeren på en 2D-flade i det separate vindue, som ingen dybdeinformation indeholder. Dette resulterer sammen med V_p's elementer i i alt 12+4 = 16 ubekendte og dermed 15 frihedsgrader, og det er ganske enkelt for mange når man kun kan danne de 12 ligninger ovenfor. Den skuffende konklusion herpå er, at opgaven ikke umiddelbart kan løses analytisk.

I samarbejde med projektets vejleder diskuteredes forskellige teoretiske løsninger på problemet der involverer yderligere 2 punkter og muligvis en numerisk løsning via. mindste kvadraters metode, men disse blev fremlagt tæt på projektforløbets afslutning, og en endelig matematisk løsning er således endnu ikke fundet.

Teknikken er derfor ikke implementeret men udgør den absolut mest interessante tanke for fremtidige udvidelser, og vil med sikkerhed have høj prioritet givet mere tid. En undersøgelse af præcist hvordan problemet er tacklet i Canoma ville være et oplagt sted at starte sin research.

5.3 Fremtidige Udvidelser: Mere Avanceret Geometri

En anden åbenlys udvidelse af programmet er tillæg til den type geometri programmets tegnefunktionalitet pt. er i stand til at oprette. Da projektets vision var under udarbejdning, var det med bygninger beliggende på Danmarks Tekniske Universitets arealer for tanke, fordi de stort set uden undtagelse er kubiske og derfor utrolig simple at modellere. Der ville være en god chance for at billeder taget på DTU passede forholdsvis fornuftigt til den geometri man indenfor kort tid kunne implementere understøttelse for i et projekt som dette. Som vi berørte i afsnit 1.2 er visionen bag projektet en udforskning af tekstureringsmetodik. Ikke en øvelse i implementering af forskelligartet geometri. Derfor var kubiske bygninger tilstrækkelige at implementere.

Det sagt, så er kun at tegne rektangulære parallelepipidæ selvsagt en uacceptabel begrænsning i et videre forløb. Det er ingen hemmelighed at programmets tegnefunktionalitet, altså hele paradigmet med at ekstrudere kasser fra grundplanet, er baseret på Google Sketchup, og den optimale situation er klart at kunne understøtte samme mangfoldighed af forskelligt geometri så programmet kan bruges til at skabe rigere miljøer. Det ville være et spændende eksperiment at prøve at belyse avanceret geometri som et kirkespir, eller et slot for den sags skyld, med et luftfoto af den tilsvarende virkelige bygning, blot for at prøve kræfter med den projektive teksturering. Så kunne man studere fra hvor mange synsvinkler, udover projektorens egen, tekstureringen virker tilfredsstillende. Eventuelt kunne man tilføje understøttelse af indlæsning af Wavefront OBJ-filer, og lade CAD-programmet skalere, translatere osv.

Kapitel 6

Konklusion

I projektets konklusion undersøges hvad der gennem perioden er opnået og absolut ligeså vigtigt, hvorvidt det opnåede stemmer overens med den målsætning vi startede med at formulere.

I rapporten er studeret tekstureringsmetodik forbundet med oprettelse af bygninger på baggrund af fotos. Undersøgelsen er foretaget via programmeringen af et mindre CAD-program, der muliggør oprettelse af simpel, kubisk geometri. Heri kan geometrien belyses projektivt af forskellige teksturer brugeren vælger i et separat vindue.
I forhold til programmets kravspecifikation kan målene så absolut siges at være nået. Programmet indeholder et interaktivt 3D-miljø, i hvilket brugeren har fuld kontrol over kameraets position og retning og kan navigere rundt på forskellige måder, så en scene hurtigt kan beses fra forskellige vinkler.
Heri kan kubiske bygninger oprettes ved at udtrække rektangler i planen, og ekstrudere dem til kasser. Kasserne kan efterfølgende translateres i planen, roteres og omformes som det passer brugeren.
Når brugeren er tilfreds med den oprettede model kan en virtuel repræsentation af en lysbilledprojektor teksturere geometrien projektivt. Ligesom både navigation og den oprettede geometri har brugeren gennem programmets interface fuld kontrol over projektorens position og retning. Programmet indeholder ydermere funktionalitet, der forsøger at hjælpe brugeren ved at foretage et indledningsvist gæt på projektorens ønskede position. Herfra kan projektoren roteres om den valgte geometri, relativ til geometriens centrum.
Sammen med programmets grafiske brugerinterface og det assisterende teksturvindue danner ovenstående rammen for opfyldelsen af den i afsnit 2.1 optegnede kravspecifikation, og målet betegnes derfor formelt som nået.

Hvad er så fundet? Nu da selve programmets krav betragtes nået, hvad konkluderes så om den undersøgte tekstureringsmetodik? Hvad er svaret på det, projektet oprindeligt satte sig for at undersøge?
Det korte svar vil umiddelbart være, at teksturering af bygninger til brug i programmer som Google Earth forbliver en besværlig affære. Selvom jeg føler jeg er kommet godt omkring selve processen at teksturere bygninger og har opnået en god forståelse af hvordan i særdeleshed projektiv teksturering kan bruges til formålet, vil jeg ikke gå så langt som at sige det er lykkes mig at forsimple opgaven. Restaureringen af 3. kvadrant som jeg foretog gav et rimelig godt resultat, og trods de grafiske artifacts kan projektiv teksturering godt bruges til at gendanne geometri fra billeder i 3D, men jeg må indrømme at arbejdet forbliver tungt. Til mit forsvar vil jeg tilføje at den visuelle kvalitet af restaureringen mere havde noget at gøre med naturen af de brugte billeder end med det program, jeg har skrevet.

Konklusionen er derfor, at hvis arbejdet med at teksturere bygninger skal lettes, så stiller det store krav til kvaliteten af de billeder, der skal påhæftes bygningerne. De skal være taget fra bestemte vinkler der ikke altid er mulige, og uønskede objekter som træer, parkerede biler osv. må ikke være i vejen når billederne tages. Dette er ofte uoverkommelige krav af åbenlyse årsager, og i professionelle applikationer (Google Earth) ser vi også at de teksturer bygningerne har typisk er mørke og homogene, som om bygningen kun var ambient belyst. De stammer næppe fra faktiske billeder af bygningen men lader til at være kunstigt fremstillet.

Slutteligt vil jeg sige at projektet har været en yderst lærerig affære at beskæftige sig med og resulteret i en signifikant tilføjelse til min erfaring med computergrafisk arbejde generelt.

Bibliografi

[1] Wikipedia artikel om fotogrammetri:
http://en.wikipedia.org/wiki/Photogrammetry

[2] Hjemmeside for programmet Canoma, Metacreations Corp, 1999.
http://www.canoma.com/

[3] Wikipedia artikel om Google Earth, sektionen ”Overview”:
http://en.wikipedia.org/wiki/Google_earth

[4] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider & Tom Davis. The Hit Record. I *OpenGL Programming Guide*, siderne 363-364. Addison Wesley Publishing Company, 2005. (The Red Book)

[5] Cass Everitt. *Projective Texture Mapping*. Ukendt udgiver. Tilgængelig på http://www.nps.navy.mil/cs/sullivan/MV4470/resources/projective_texture_mapping.pdf

[6] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider & Tom Davis. The Hit Record. I *OpenGL Programming Guide*, plate 18, siderne side 481. Addison Wesley Publishing Company, 2005. (The Red Book)

[7] Edward Angel. Rotation. I *Interactive Computer Graphics – A top down approach using OpenGL*, siderne 202-212. Pearson Education Inc. 2006.

[8] Wikipedia artikel om horisontale koordinatsystemer:
http://en.wikipedia.org/wiki/Horizontal_coordinate_system

[9] Illustration lavet af Francisco Javier Blanco González til brug på Wikipedia. Vises i artiklen referet i [8]. Billedet kan frit modificeres og distribueres.

[10] Paul Rademacher. *GLUI – A GLUT-Based User Interface Library*. Brugermanual til programmets grafiske brugerflade.

[11] www.OpenGL.org. Mark Kilgaard. Remember Your Matrix Mode. I *”Avoiding 16 Common OpenGL Pitfalls”*, 2000.
http://www.opengl.org/resources/features/KilgardTechniques/oglpitfall/

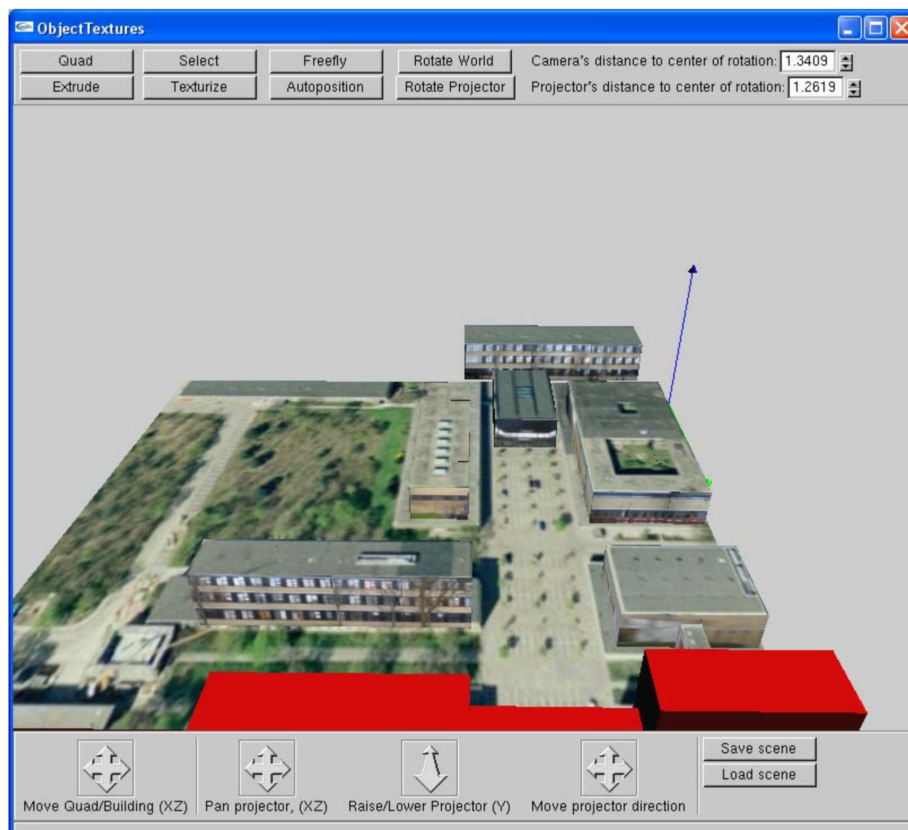
[12] Tomas Akenine-Möller. Eric Haines. Ray/Triangle Intersection. I *“Real-time Rendering”*, siderne 578-582. A. K. Peters, Ltd. 2002.

[13] Paul Heckbert. Projective Mappings for Image Warping. I *“Fundamentals of Texture Mapping and Image Warping”*, Master’s Thesis, siderne 17-21, CS Division, U.C. Berkeley, Juni 1989.

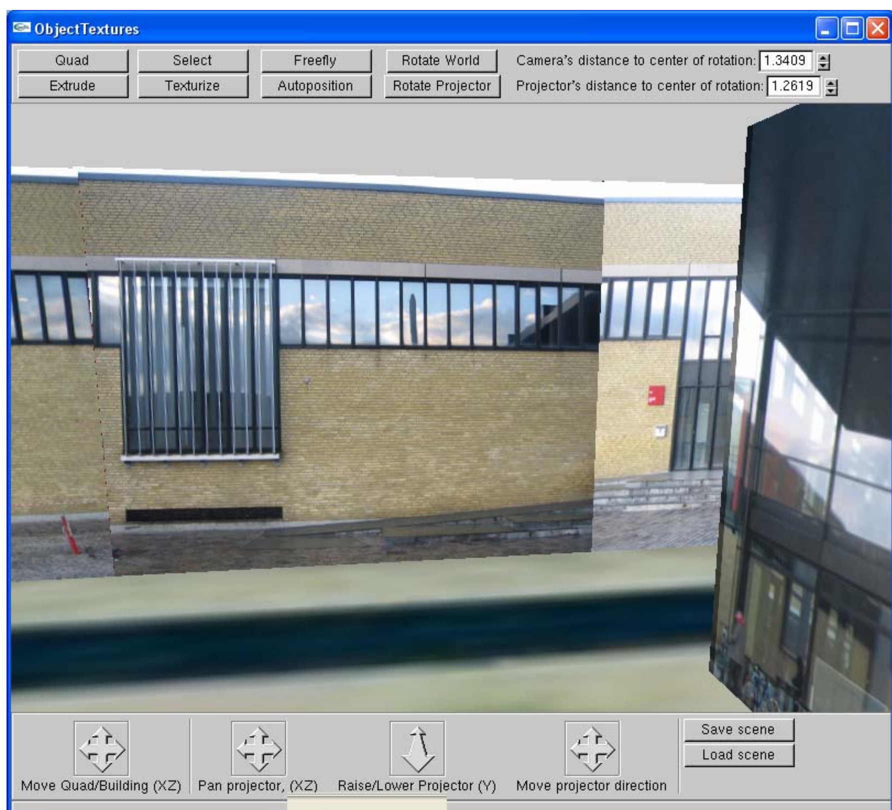
[14] Paul E. Debevec. Camillo J. Taylor. Jitendra Malik. *”Modelling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach”*. University of California at Berkeley. Ukendt udgivelsesår.

[15] Jacobo Rodriguez Villar. OpenGL Shading Language Course. Kapitlerne 1-5. Fra Shader Designers hjemmeside på:
www.TyphoonLabs.com

Appendix A – Screenshots

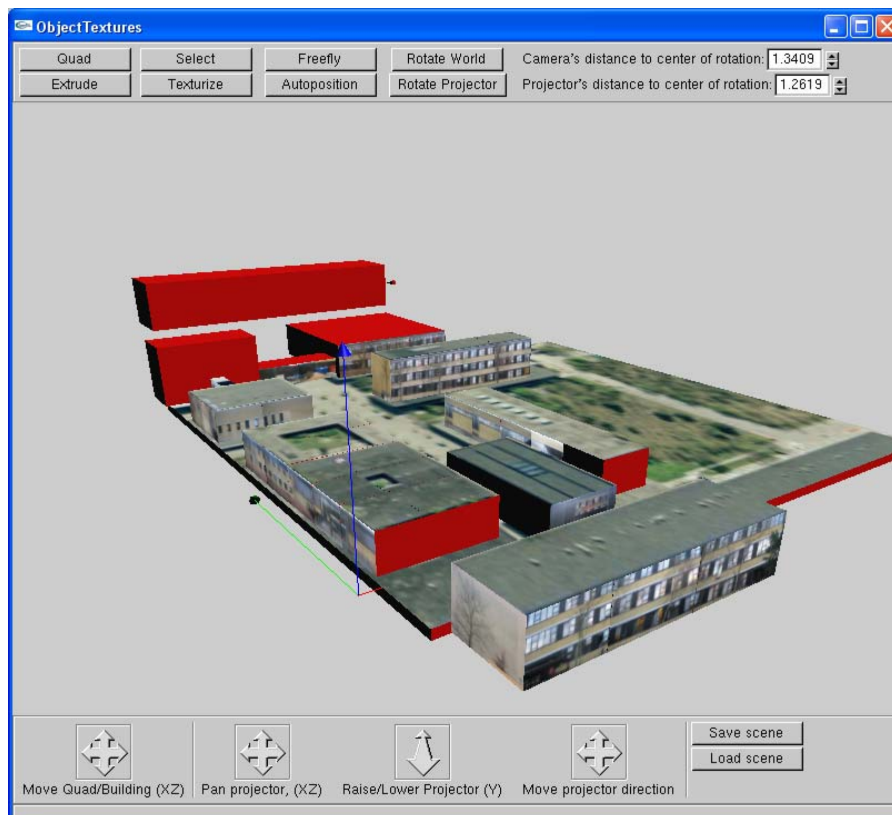


Øverst til højre: Fugleperspektiv af 3. Kvadrant på DTU, set fra syd mod nord. De røde bygninger er henholdsvis bygning 305 og 308, som der ikke er taget billeder af, hvorfor de ingen teksturer har.
Nederst til højre: Bygning 307 i forgrunden, skyggende for Matematiktorvet.



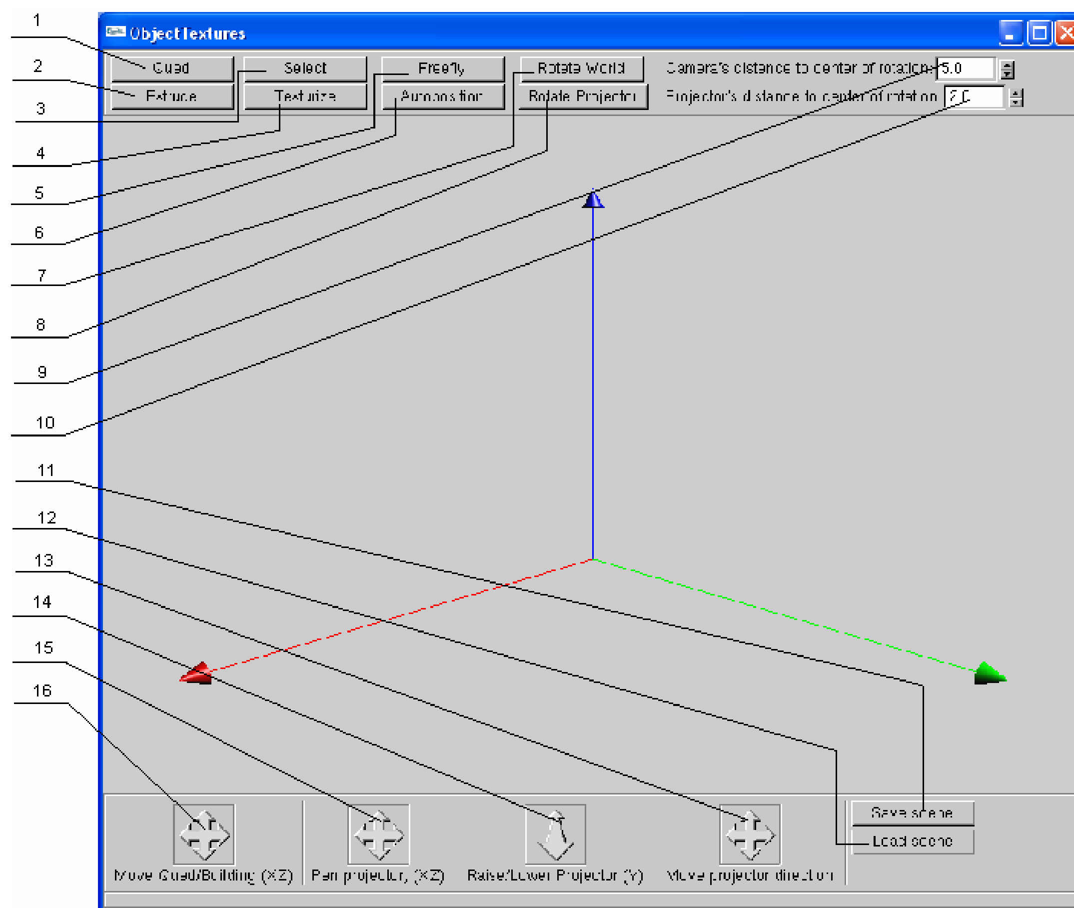
Øverst til højre: Kamera placeret i øjenhøjde på Matematiktorvet, kiggende mod bygning 302 i midten.
Nederst til højre: Kamera beliggende foran bygning 302, kiggende mod bygning 306. Screenshot taget for at understrege grafiske artifacts forbundet segmenteret opbyggelse af geometri.

Til højre: Fugleperspektiv af 3. kvadrants nordlige ende, kamera kiggende mod syd-vest. Lang bygning allerbagest eksisterer ikke i virkeligheden, den er oprettet til test af projicering af perspektivisk tekstur.



Appendix B – Brugermanual

Hovedvinduet



1. Knappen “Quad”. Tegner rektangler i grundplanen.
2. Knappen ”Extrude”. Ekstruderer quads fra grundplanen til bygninger.
3. Knappen ”Select”. Udvælger geometri.
4. Knappen ”Texturize”. Fastsætter tekstur, projektorposition og retning til en side.
5. Knappen ”Freefly”. Tillader frit flyvning rundt i miljøet med musen og a,s,d,w.
6. Knappen ”Autoposition”. Sætter projektoren, så den valgte tekstur skydes ortogonalt på den næste side, der klikkes på.
7. Knappen ”Rotate World”. Roterer kameraet rundt om et punkt bestemt af nuværende retning, og det tal, der står i feltet under 9.
8. Knappen ”Rotate Projektor”. Roterer projektoren rundt om udvalgt bygning relativ til bygningens centrum efter den er auto-positioneret.
9. Spinneren ”Camera’s distance to center of rotation”. Sætter afstanden til punktet der roteres om ved tryk på 7.

10. Spinneren ”Projektor’s distance to center of rotation”. Sætter afstanden til bygningens centrum, når kameraet autopositioneres og når det roteres om udvalgt geometri.
11. Knappen ”Save scene”. Gemmer al data forbundet med bygninger i en txt-fil ved navn ”BuildingData.txt”.
12. Knappen ”Load scene”. Indlæser gemt data fra BuildingData.txt og opdaterer scenen.
13. Translationsknappen ”Move Projector Direction”. Flytter på projektorens retning med piletasterne.
14. Translationsknappen ”Raise/Lower Projector (Y)”. Flytter projektoren op og ned i y-retning.
15. Translationsknappen ”Pan Projector, (XZ)”. Flytter projektoren rundt i x- og z-retning.
16. Translationsknappen ”Move Quad/Building”. Flytter udvalgt geometri rundt i XZ-planet.

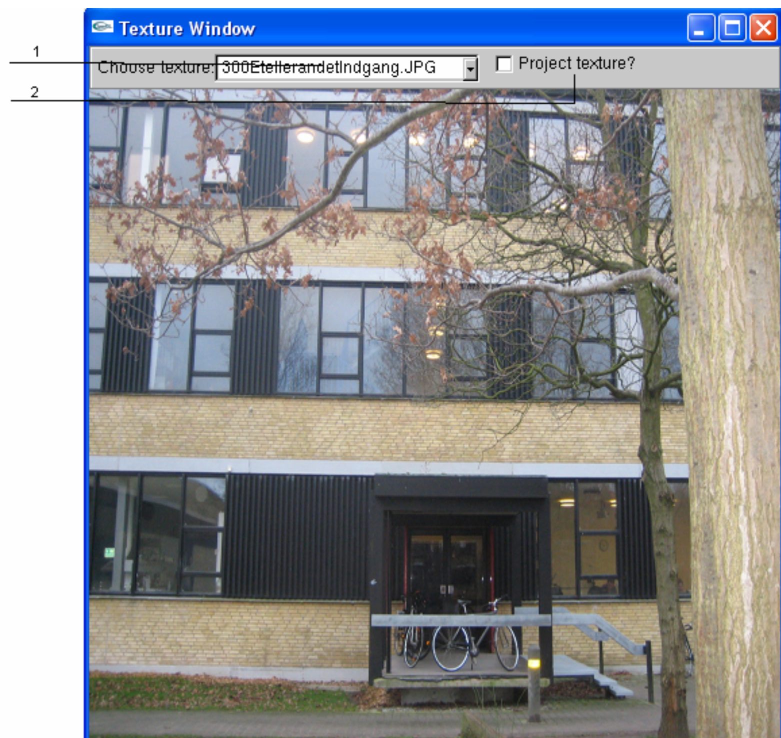
Alle translationsknapper samt spinners kan ændre følsomhed hvis man holder henholdsvis CTRL eller SHIFT i bund. CTRL sænker følsomheden, mens SHIFT øger den.

’c’-knappen på keyboardet ændrer den værdi, med hvilken bygningers dimensioner ændres mellem fin og grov. Defaultsetting er grov.

Teksturvinduet

1. Drop-down menu til udvælgelse af aktuel tekstur.
2. Tickbox til at slå generel projicering af den aktuelle tekstur til og fra.

Begge valg har øjeblikke effekt i hovedvinduet.



Appendix C – Kildekode

C.1 Definitions.h

```
#ifndef _DEFINITIONS_GUARD_
#define _DEFINITIONS_GUARD_

/// This header is no class declaration but merely a collection of
definitions used throughout the code.
/// Gathering definitions in one file was deemed better than ad hoc
defining where needed for several reasons,
/// though in particular because those that are numerically ordered
become difficult to keep track of as the code
/// grows in size, and mistakingly assigning identical integers to
two different modes could prove fatal.
/// This technique, for the record, is similar to Java's tendency
to gather public static final variables
/// in one file.

/// Numerically consecutive defines:

#define SQUARE_MODE 1
#define CUBE_MODE 2
#define FREE_FLY 3
#define ROTATE_CAM 4
#define PROJECTOR_TRANSLATION_MODE 5
#define NUM_SIDES 6
#define TEX_CHOOSER 7
#define GEOMETRY_TRANSLATION_MODE 8
#define PROJECTION_CHECKBOX 9
#define TEXTURIZE_WORLD 10
#define PROJECTOR_DIRECTION_MODE 11
#define CHANGE_CAM_ROT_DIST_MODE 12
#define AUTOPOSITION_PROJECTOR_MODE 13
#define ROTATE_PROJECTOR 14
#define CHANGE_PROJ_ROT_DIST_MODE 15
#define SAVE_SCENE 16
#define LOAD_SCENE 17


/// Non-consecutives:

#define MIN_TEX_WINDOW_WIDTH 256
#define WINDOW_X 800
#define WINDOW_Y 700
#define PI 3.14159265358979323846
#define NO_HITS 1000000

#endif
```

C.2 Drawing.h

```
#ifndef _DRAWING_GUARD_
#define _DRAWING_GUARD_

#include <iostream>
#include <GL/glut.h>
#include <vector>
#include "Quad.h"
#include "RectPar.h"
#include "CGLA/Vec3f.h"
#include "Projector.h"

class Drawing {

    TexNav * texture_navigator;

public:

    Drawing(){}

      void drawSquaref(CGLA::Vec3f, CGLA::Vec3f);

      void drawSquaref(Quad);

      void drawBottomQuad(Quad);

      void drawAllQuads(GLenum);

      void drawAllRectPars(GLenum, Projector*, TexNav*, GLuint *);

    void drawAllRectPars(GLenum);

      void drawRectPar(RectPar, Projector*, TexNav*, GLuint *);

    void drawRectPar(RectPar);

      void drawAxes();

      void storeQuad(CGLA::Vec3f, CGLA::Vec3f);

      void removeQuad(int);

    void remove_building(int);

      Quad * get_selected_quad(int);

    RectPar * get_selected_building(int);

      void storeRectPar(CGLA::Vec3f, CGLA::Vec3f, float);

      void light();

      bool isQuadsEmpty();

    bool isBuildingsEmpty();
```

```
    bool selectionOkay(int);

    std::vector<RectPar>* get_buildings();

};

#endif
```

C.3 Drawing.cpp

```
#include <gl/glew.h>
#include "Drawing.h"
#include <vector>

using namespace CGLA;
using namespace std;

//Storage containers for ground level quads and for buildings.
vector<Quad> quads;
vector<RectPar> buildings;

// Draws a rectangular parallelepiped with no static textures
void Drawing::drawRectPar(RectPar building) {

    // Initialize data structure for building's sides
    Quad * box_quads[NUM_SIDES];
    building.returnAllQuads(box_quads);

    if(!box_quads[0]->is_textured())
        drawBottomQuad(*box_quads[0]);
    if(!box_quads[1]->is_textured())
        drawSquaref(*box_quads[1]);
    if(!box_quads[2]->is_textured())
        drawSquaref(*box_quads[2]);
    if(!box_quads[3]->is_textured())
        drawSquaref(*box_quads[3]);
    if(!box_quads[4]->is_textured())
        drawSquaref(*box_quads[4]);
    if(!box_quads[5]->is_textured())
        drawSquaref(*box_quads[5]);
}

// Draws those sides of a rectangular parallelepiped that have
textures set by user
void Drawing::drawRectPar(RectPar building, Projector * proj,
TexNav * texture_navigator, GLuint * proj_prog) {

      // Initialize data structure for building's sides
      Quad * box_quads[NUM_SIDES];
      building.returnAllQuads(box_quads);

      // Draw individual sides, texture mapping where applicable.
This is a copy paste approach used for all sides
    // of the building.
```

```
    // Has this particular quad been textured by the user?
    if(box_quads[0]->is_textured()) {

        // Moving projector, setting the texture to the one
assigned to this quad, and drawing quad with texture mapping
        proj->set_temp_projector_posndir(box_quads[0]-
>get_projector_position(), box_quads[0]-
>get_projector_direction());
        glBindTexture(GL_TEXTURE_2D, box_quads[0]-
>get_texture_object_id());
        glUseProgram(*proj_prog);
        drawBottomQuad(*box_quads[0]);
        glUseProgram(0);
        // We need to reset the projector to its original position
and have it project whatever's in the secondary
        // window once again
        proj->set_up_projector();
        glBindTexture(GL_TEXTURE_2D, *texture_navigator-
>get_live_var());
    }

    if(box_quads[1]->is_textured()) {

        proj->set_temp_projector_posndir(box_quads[1]-
>get_projector_position(), box_quads[1]-
>get_projector_direction());
        glBindTexture(GL_TEXTURE_2D, box_quads[1]-
>get_texture_object_id());
        glUseProgram(*proj_prog);
        drawSquaref(*box_quads[1]);
        glUseProgram(0);
        proj->set_up_projector();
        glBindTexture(GL_TEXTURE_2D, *texture_navigator-
>get_live_var());
    }

    if(box_quads[2]->is_textured()) {

        proj->set_temp_projector_posndir(box_quads[2]-
>get_projector_position(), box_quads[2]-
>get_projector_direction());
        glBindTexture(GL_TEXTURE_2D, box_quads[2]-
>get_texture_object_id());
        glUseProgram(*proj_prog);
        drawSquaref(*box_quads[2]);
        glUseProgram(0);

        proj->set_up_projector();
        glBindTexture(GL_TEXTURE_2D, *texture_navigator-
>get_live_var());
    }

    if(box_quads[3]->is_textured()) {

        proj->set_temp_projector_posndir(box_quads[3]-
>get_projector_position(), box_quads[3]-
>get_projector_direction());
```

```
        glBindTexture(GL_TEXTURE_2D, box_quads[3]-
>get_texture_object_id());
        glUseProgram(*proj_prog);
        drawSquaref(*box_quads[3]);
        glUseProgram(0);

        proj->set_up_projector();
        glBindTexture(GL_TEXTURE_2D, *texture_navigator-
>get_live_var());
    }

    if(box_quads[4]->is_textured()) {

        proj->set_temp_projector_posndir(box_quads[4]-
>get_projector_position(), box_quads[4]-
>get_projector_direction());
        glBindTexture(GL_TEXTURE_2D, box_quads[4]-
>get_texture_object_id());
        glUseProgram(*proj_prog);
        drawSquaref(*box_quads[4]);
        glUseProgram(0);

        proj->set_up_projector();
        glBindTexture(GL_TEXTURE_2D, *texture_navigator-
>get_live_var());
    }

    if(box_quads[5]->is_textured()) {

        proj->set_temp_projector_posndir(box_quads[5]-
>get_projector_position(), box_quads[5]-
>get_projector_direction());
        glBindTexture(GL_TEXTURE_2D, box_quads[5]-
>get_texture_object_id());
        glUseProgram(*proj_prog);
        drawSquaref(*box_quads[5]);
        glUseProgram(0);

        proj->set_up_projector();
        glBindTexture(GL_TEXTURE_2D, *texture_navigator-
>get_live_var());
    }

}

void Drawing::storeQuad(Vec3f a, Vec3f c) {
      quads.push_back(Quad(a, c));
}

void Drawing::storeRectPar(Vec3f a, Vec3f c, float height) {
      buildings.push_back(RectPar(a,c, height));
}

void Drawing::drawSquaref(Vec3f a, Vec3f c) {
      Quad quad_points = Quad(a,c);
      glPushAttrib(GL_ALL_ATTRIB_BITS);
      Vec3f edge_1 = quad_points.get_b() - a;
```

```
	Vec3f edge_2 = c - a;
	Vec3f surface_normal = normalize(cross(edge_1, edge_2));

	glBegin(GL_QUADS);
		glNormal3f(surface_normal[0], surface_normal[1],
surface_normal[2]);
		glVertex3f(quad_points.get_a()[0],
quad_points.get_a()[1], quad_points.get_a()[2]);
		glVertex3f(quad_points.get_b()[0],
quad_points.get_b()[1], quad_points.get_b()[2]);
		glVertex3f(quad_points.get_c()[0],
quad_points.get_c()[1], quad_points.get_c()[2]);
		glVertex3f(quad_points.get_d()[0],
quad_points.get_d()[1], quad_points.get_d()[2]);
	glEnd();
	glPopAttrib();
}

void Drawing::drawSquaref(Quad quad_points) {
	glPushAttrib(GL_ALL_ATTRIB_BITS);

	// Define normal
	Vec3f edge_1 = quad_points.get_b() - quad_points.get_a();
	Vec3f edge_2 = quad_points.get_c() - quad_points.get_a();
	Vec3f surface_normal = normalize(cross(edge_1, edge_2));

	glBegin(GL_QUADS);

	glNormal3f(surface_normal[0],surface_normal[1],surface_normal
[2]);
		glVertex3f(quad_points.get_a()[0],
quad_points.get_a()[1], quad_points.get_a()[2]);
		glVertex3f(quad_points.get_b()[0],
quad_points.get_b()[1], quad_points.get_b()[2]);
		glVertex3f(quad_points.get_c()[0],
quad_points.get_c()[1], quad_points.get_c()[2]);
		glVertex3f(quad_points.get_d()[0],
quad_points.get_d()[1], quad_points.get_d()[2]);
	glEnd();
	glPopAttrib();
}

void Drawing::drawBottomQuad(Quad quad_points) {
	glPushAttrib(GL_ALL_ATTRIB_BITS);

	// Define normal
	Vec3f edge_1 = quad_points.get_b() - quad_points.get_a();
	Vec3f edge_2 = quad_points.get_c() - quad_points.get_a();
	Vec3f surface_normal = -1*normalize(cross(edge_1, edge_2));

	glBegin(GL_QUADS);

	glNormal3f(surface_normal[0],surface_normal[1],surface_normal
[2]);
		glVertex3f(quad_points.get_a()[0],
quad_points.get_a()[1], quad_points.get_a()[2]);
```

```
            glVertex3f(quad_points.get_b()[0],
quad_points.get_b()[1], quad_points.get_b()[2]);
            glVertex3f(quad_points.get_c()[0],
quad_points.get_c()[1], quad_points.get_c()[2]);
            glVertex3f(quad_points.get_d()[0],
quad_points.get_d()[1], quad_points.get_d()[2]);
      glEnd();
      glPopAttrib();
}

void Drawing::drawAllQuads(GLenum mode) {
      for(int i = 0; i < quads.size(); i++) {
            if (mode == GL_SELECT)
                  glLoadName(i);
        drawSquaref(quads.at(i));
      }
}

void Drawing::drawAllRectPars(GLenum mode, Projector * proj, TexNav
* texture_navigator, GLuint * proj_prog) {
      for(int i = 0; i < buildings.size(); i++) {
        if (mode == GL_SELECT)
            glLoadName(100+i);
            drawRectPar(buildings.at(i), proj, texture_navigator,
proj_prog);
      }
}

void Drawing::drawAllRectPars(GLenum mode) {
    for(int i = 0; i < buildings.size(); i++) {
        if (mode == GL_SELECT)
            glLoadName(100+i);
        drawRectPar(buildings.at(i));
    }
}

// A function for drawing the axes of the world. It is intented as
a navigational aid to the user.

void Drawing::drawAxes() {

            glDisable(GL_LIGHTING);
            GLfloat v0[] = {0., 0., 0.};
        GLfloat vx[]= {2., 0., 0.};
        GLfloat vy[]= {0, 2., 0.};
        GLfloat vz[]= {0., 0., 2.};

        glPushAttrib(GL_CURRENT_BIT);
        glColor3f (0, 1.0, 0.);
        glBegin (GL_LINES);
                glVertex3fv (v0);
                glVertex3fv (vx);
        glEnd ();

            glColor3f(0.0, 0.0, 1.0);
        glBegin (GL_LINES);
                glVertex3fv (v0);
```

```
                glVertex3fv (vy);
        glEnd ();

            glColor3f(1.0, 0.0, 0.0);
        glBegin (GL_LINES);
                glVertex3fv (v0);
                glVertex3fv (vz);
        glEnd ();

            glPopAttrib();
            glEnable(GL_LIGHTING);

            // Draw cones to further indicate positive axis
directions

            // Z-cone
            glPushMatrix();
            glTranslatef(vz[0], vz[1], vz[2]);
            glutSolidCone(0.05, 0.1, 50, 50);
            glPopMatrix();

            // X-cone
            glPushAttrib(GL_ALL_ATTRIB_BITS);
            glPushMatrix();
            glTranslatef(vx[0], vx[1], vx[2]);
            glRotatef(90,0,1,0);
            GLfloat mat_green_diffuse[] = { 0.0, 1.0, 0.0, 0.0 };
            glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_green_diffuse);
            glutSolidCone(0.05, 0.1, 50, 50);
            glPopMatrix();
            glPopAttrib();

        // Y-cone
            glPushAttrib(GL_ALL_ATTRIB_BITS);
            glPushMatrix();
            glTranslatef(vy[0], vy[1], vy[2]);
            glRotatef(-90,1,0,0);
            GLfloat mat_blue_diffuse[] = { 0.0, 0.0, 1.0, 0.0 };
            glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_blue_diffuse);
            glutSolidCone(0.05, 0.1, 50, 50);
            glPopMatrix();
            glPopAttrib();
}

// Setting up light parameters

void Drawing::light() {

      GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0};
      GLfloat mat_diffuse[] = { 1.0, 0.0, 0.0, 0.0 };
      GLfloat mat_shininess[] = { 50.0 };
      GLfloat light_position[] = { 4.0, 4.0, 4.0, 0.0 };
      glClearColor( 0.0, 0.0, 0.0, 0.0);

      glShadeModel(GL_SMOOTH);

      glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
```

```
        glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
        glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
        glLightfv(GL_LIGHT0, GL_POSITION, light_position);

        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
}

void Drawing::removeQuad(int index) {
    //cout << "Index of quad to be removed is: " << index << " and
the size of the container is: " << quads.size() << endl;

    quads.erase(quads.begin() + index);
}

void Drawing::remove_building(int index) {
    buildings.erase(buildings.begin() + index);
}

Quad * Drawing::get_selected_quad(int index) {
    return &quads.at(index);
}

RectPar * Drawing::get_selected_building(int index) {
    return &buildings.at(index);
}

bool Drawing::isQuadsEmpty() {
        return quads.empty();
}

bool Drawing::isBuildingsEmpty() {
    return buildings.empty();
}

bool Drawing::selectionOkay(int index) {
    if (index >= quads.size()) {
        return false;
    }
    else {
        return true;
    }
}

vector<RectPar>* Drawing::get_buildings(){
    return &buildings;
}
```

C.4 FileIO.h

```
#ifndef _FILEIO_GUARD_
#define _FILEIO_GUARD_

#include <iostream>
#include <fstream>
```

```
#include <sstream>
#include <string>
#include <vector>
#include <stdio.h>
#include <stdlib.h>
#include "RectPar.h"

    void saveTable(std::vector<RectPar>);
    void Tokenize(const std::string&, std::vector<std::string>&,
const std::string&);
    void readFile(std::vector<RectPar>*);

#endif
```

C.5 FileIO.cpp

```
#include "FileIO.h"

using namespace std;
using namespace CGLA;

void saveTable(vector<RectPar> buildings) {

    Quad * sides[NUM_SIDES];

    // Delete existing file. Done, because we do not wish to append
entire program state to last save.
    // Instead, we wish to delete last save, and create a new save
using current program state. This
    // corresponds to updating since last save, i.e. appending
changes.
    remove("BuildingData.txt");

    // Create new file, and commence saving procedure unless vector
is empty
    ofstream file;
    if(!buildings.empty()) {
    file.open("BuildingData.txt");

        for (int i = 0; i < buildings.size(); i++) {

        buildings.at(i).returnAllQuads(sides);

        // The actual writing to file here is laboriously long and
silly, really.
        // It happens like the following because CGLA hasn't
defined the operator "<<" for file
        // output streams, so we must extract the individual
coordinates and save those.
        // A building is defined by the corner vertices of its
bottom quad and its height, so these data are
        // saved as the first 7 parameters on the line.
        // After that, the sides of the building may have projector
data associated with them. These are
        // saved on subsequent spots.
```

```
        file << sides[0]->get_a()[0] << " " << sides[0]->get_a()[1]
<< " " << sides[0]->get_a()[2] << " "
             << sides[0]->get_c()[0] << " " << sides[0]->get_c()[1]
<< " " << sides[0]->get_c()[2] << " "
             << sides[5]->get_c()[1] - sides[0]->get_c()[1] << " ";

        for (int j = 0; j < NUM_SIDES; j++) {
            file << sides[j]->is_textured() << " "
                 << sides[j]->get_projector_direction()[0] << " "
<< sides[j]->get_projector_direction()[1] << " " << sides[j]-
>get_projector_direction()[2] << " "
                 << sides[j]->get_projector_position()[0] << " " <<
sides[j]->get_projector_position()[1] << " " << sides[j]-
>get_projector_position()[2] << " "
                 << sides[j]->get_texture_object_id() << " ";

        }
        file << endl;
    }
    }
    file.close();
}

void Tokenize(const string& str, vector<string>& tokens, const
string& delimiters = " ") {

    // Skip delimiters at beginning.
    string::size_type lastPos = str.find_first_not_of(delimiters,
0);
    // Find first "non-delimiter".
    string::size_type pos = str.find_first_of(delimiters, lastPos);

    while (string::npos != pos || string::npos != lastPos)
    {
        // Found a token, add it to the vector.
        tokens.push_back(str.substr(lastPos, pos - lastPos));
        // Skip delimiters.  Note the "not_of"
        lastPos = str.find_first_not_of(delimiters, pos);
        // Find next "non-delimiter"
        pos = str.find_first_of(delimiters, lastPos);
    }
}

void readFile (vector<RectPar>* buildings) {

    // We need to clear the vector of buildings, because we'll be
reloading whatever's in the text file,
    // not appending what's in the text file to current contents
    buildings->clear();

    // Variables associated with file reading and string
tokenization
    string line, current_token;
    vector<string> line_tokens;
    ifstream myfile;
```

```
    // Data associated with a building:
    Vec3f a, c,
        bottom_projector_direction, bottom_projector_position,
        side1_projector_direction, side1_projector_position,
        side2_projector_direction, side2_projector_position,
        side3_projector_direction, side3_projector_position,
        side4_projector_direction, side4_projector_position,
        top_projector_direction, top_projector_position;
    float height;
    int bottom_texture_id, side1_texture_id, side2_texture_id,
side3_texture_id, side4_texture_id, top_texture_id;
    bool is_bottom_textured = false, is_side1_textured = false,
is_side2_textured = false, is_side3_textured = false,
is_side4_textured = false, is_top_textured = false;

    myfile.open("BuildingData.txt", fstream::in | fstream::out);
    if(myfile.is_open()){
        while (!myfile.eof()) {
            getline (myfile, line);
            line_tokens.clear();
            Tokenize(line, line_tokens, " ");
            for (int i = 0; i < line_tokens.size(); i++) {
             current_token = line_tokens.at(i);
                istringstream buffer(current_token);

                // Time for the big reload. >_<
                // This ugly piece of code loads all the data
associated with the building and, in particular,
                // the information for the projector-struct that
each Quad has.

                switch (i) {
                        case 0:
                            buffer >> a[0];
                            break;
                        case 1:
                            buffer >> a[1];
                            break;
                        case 2:
                            buffer >> a[2];
                            break;
                        case 3:
                            buffer >> c[0];
                            break;
                        case 4:
                            buffer >> c[1];
                            break;
                        case 5:
                            buffer >> c[2];
                            break;
                        case 6:
                            buffer >> height;
                            break;
                        case 7:
                            buffer >> is_bottom_textured;
                            break;
                        case 8:
```

```
                        buffer >>
bottom_projector_direction[0];
                        break;
                    case 9:
                        buffer >>
bottom_projector_direction[1];
                        break;
                    case 10:
                        buffer >>
bottom_projector_direction[2];
                        break;
                    case 11:
                        buffer >> bottom_projector_position[0];
                        break;
                    case 12:
                        buffer >> bottom_projector_position[1];
                        break;
                    case 13:
                        buffer >> bottom_projector_position[2];
                        break;
                    case 14:
                        buffer >> bottom_texture_id;
                        break;
                    case 15:
                        buffer >> is_side1_textured;
                        break;
                    case 16:
                        buffer >> side1_projector_direction[0];
                        break;
                    case 17:
                        buffer >> side1_projector_direction[1];
                        break;
                    case 18:
                        buffer >> side1_projector_direction[2];
                        break;
                    case 19:
                        buffer >> side1_projector_position[0];
                        break;
                    case 20:
                        buffer >> side1_projector_position[1];
                        break;
                    case 21:
                        buffer >> side1_projector_position[2];
                        break;
                    case 22:
                        buffer >> side1_texture_id;
                        break;
                    case 23:
                        buffer >> is_side2_textured;
                        break;
                    case 24:
                        buffer >> side2_projector_direction[0];
                        break;
                    case 25:
                        buffer >> side2_projector_direction[1];
                        break;
                    case 26:
```

```
        buffer >> side2_projector_direction[2];
        break;
    case 27:
        buffer >> side2_projector_position[0];
        break;
    case 28:
        buffer >> side2_projector_position[1];
        break;
    case 29:
        buffer >> side2_projector_position[2];
        break;
    case 30:
        buffer >> side2_texture_id;
        break;
    case 31:
        buffer >> is_side3_textured;
        break;
    case 32:
        buffer >> side3_projector_direction[0];
        break;
    case 33:
        buffer >> side3_projector_direction[1];
        break;
    case 34:
        buffer >> side3_projector_direction[2];
        break;
    case 35:
        buffer >> side3_projector_position[0];
        break;
    case 36:
        buffer >> side3_projector_position[1];
        break;
    case 37:
        buffer >> side3_projector_position[2];
        break;
    case 38:
        buffer >> side3_texture_id;
        break;
    case 39:
        buffer >> is_side4_textured;
        break;
    case 40:
        buffer >> side4_projector_direction[0];
        break;
    case 41:
        buffer >> side4_projector_direction[1];
        break;
    case 42:
        buffer >> side4_projector_direction[2];
        break;
    case 43:
        buffer >> side4_projector_position[0];
        break;
    case 44:
        buffer >> side4_projector_position[1];
        break;
    case 45:
```

```
                        buffer >> side4_projector_position[2];
                        break;
                    case 46:
                        buffer >> side4_texture_id;
                        break;
                    case 47:
                        buffer >> is_top_textured;
                        break;
                    case 48:
                        buffer >> top_projector_direction[0];
                        break;
                    case 49:
                        buffer >> top_projector_direction[1];
                        break;
                    case 50:
                        buffer >> top_projector_direction[2];
                        break;
                    case 51:
                        buffer >> top_projector_position[0];
                        break;
                    case 52:
                        buffer >> top_projector_position[1];
                        break;
                    case 53:
                        buffer >> top_projector_position[2];
                        break;
                    case 54:
                        buffer >> top_texture_id;
                        break;
                    default:
                        break;
                }
            }
            RectPar loaded_building = RectPar(a,c,height);
            Quad * sides[NUM_SIDES];
            loaded_building.returnAllQuads(sides);
            if(is_bottom_textured && line_tokens.size() != 0) {
                sides[0]->set_textured(is_bottom_textured);
                sides[0]-
>set_projector_direction(bottom_projector_direction);
                sides[0]-
>set_projector_position(bottom_projector_position);
                sides[0]->set_texture_object_id(bottom_texture_id);
            }
            if(is_side1_textured && line_tokens.size() != 0) {
                sides[1]->set_textured(is_side1_textured);
                sides[1]-
>set_projector_direction(side1_projector_direction);
                sides[1]-
>set_projector_position(side1_projector_position);
                sides[1]->set_texture_object_id(side1_texture_id);
            }
            if(is_side2_textured && line_tokens.size() != 0) {
                sides[2]->set_textured(is_side2_textured);
                sides[2]-
>set_projector_direction(side2_projector_direction);
```

```
                sides[2]-
>set_projector_position(side2_projector_position);
                sides[2]->set_texture_object_id(side2_texture_id);
            }
            if(is_side3_textured && line_tokens.size() != 0) {
                sides[3]->set_textured(is_side3_textured);
                sides[3]-
>set_projector_direction(side3_projector_direction);
                sides[3]-
>set_projector_position(side3_projector_position);
                sides[3]->set_texture_object_id(side3_texture_id);
            }
            if(is_side4_textured && line_tokens.size() != 0) {
                sides[4]->set_textured(is_side4_textured);
                sides[4]-
>set_projector_direction(side4_projector_direction);
                sides[4]-
>set_projector_position(side4_projector_position);
                sides[4]->set_texture_object_id(side4_texture_id);
            }
            if(is_top_textured && line_tokens.size() != 0) {
                sides[5]->set_textured(is_top_textured);
                sides[5]-
>set_projector_direction(top_projector_direction);
                sides[5]-
>set_projector_position(top_projector_position);
                sides[5]->set_texture_object_id(top_texture_id);
            }
            if (line_tokens.size() != 0) {
                buildings->push_back(loaded_building);
            }

        }
    }
    else cout << "Unable to open file" << endl;

}
```

C.6 Navigator.h

```
#ifndef _NAVIGATOR_GUARD_
#define _NAVIGATOR_GUARD_
#include <iostream>
#include <GL/glut.h>
#include "CGLA/Vec3f.h"
#include "TexNav.h"
#include "Definitions.h"

class Drawing;
class Projector;

class Navigator {

      CGLA::Vec3f cam_dir, cam_pos;
```

```
      bool up_pressed, down_pressed, left_pressed, right_pressed,
wheel_down, wheel_up;
      int mode, modifier_key;
    float offset;

    Projector * proj;
    TexNav * texture_navigator;
    GLuint * proj_prog;

public:

    Navigator(){};

    Navigator(Projector*, TexNav*, GLuint*);

      void special_keys (int, int, int);

      void keyboard(unsigned char, int, int);

      void animate(float);

      void mouse_motion(int, int);

      void mouse(int, int, int, int);

      void select(int, int);

      void processHits(GLint, GLuint[]);

      bool depth_pick(int, int);

      void ray_pick(int, int);

      CGLA::Vec3f getDirection();

      CGLA::Vec3f getPosition();

      CGLA::Vec3f get_a();

      CGLA::Vec3f get_c();

      float get_height();

      void set_main_window(int);

      void set_tex_window(int);

      int get_main_window();

      int get_tex_window();

      void set_mode(int);

      int get_mode();

      int get_current_selection();
```

```
    void set_cam_dist_to_rot_center(float);

    void set_proj_dist_to_rot_center(float);

};
#endif
```

C.7 Navigator.cpp

```
#include "Navigator.h"
#include "Drawing.h"
#include "Ray.h"
#include "CGLA/Mat3x3f.h"

using namespace CGLA;
using namespace std;

Vec3f point, a, c;
bool firsttime = true;
Drawing * drawer;
Quad dummy = Quad(Vec3f(0.0,0.0,0.0), Vec3f(0.0,0.0,0.0));
Quad * selected_side = &dummy;
int selection = 0, main_window, tex_window;
float height_end, dist2bottom, angle, dist_cam_rot_center = 5.0,
dist_proj_rot_center = 2.0, // Variables associated with drawing
    azimuth_begin, azimuth_end, azimuth, // Azimuth calculations
for trackball movement
        zenith_begin, zenith_end, zenith; // Zenith calculations
for trackball movement
Ray eye2top;
Ray eye2bottom;

// Size of selection buffer.
#define BUF_SIZE 1024

Navigator::Navigator(Projector * proj, TexNav * tex_nav, GLuint*
proj_prog)
 {
            cam_dir = Vec3f(-1.5, -0.5, -1.5);
            cam_pos = Vec3f(3.0, 1.5, 3.0);

            down_pressed = false;
            up_pressed = false;
            left_pressed = false;
            right_pressed = false;
            wheel_down = false;
            wheel_up = false;
            mode = 0;
        this->proj = proj;
        this->texture_navigator = tex_nav;
        this->proj_prog = proj_prog;
        this->offset = 0.02;
      }

void Navigator::processHits(GLint hits, GLuint buffer[]) {
```

```
      unsigned int i, j;
      GLuint names, *ptr;

      printf("hits = %d\n", hits);
      ptr = (GLuint *) buffer;
      for (i = 0; i < hits; i++) {  /* for each hit  */
            names = *ptr;
            ptr = ptr+3;
        selection = *ptr;
        ptr++;
      }
    cout << "Name of selection: " << selection << endl;
    if (hits == 0)
        selection = NO_HITS;
}

void Navigator::select(int x, int y) {
      GLuint selectBuffer[BUF_SIZE];
      GLint hits;
      GLint viewport[4];

      // Preparatory steps. Extraction of viewport, assigning
target for hit data, and switching to selection render mode.
      glGetIntegerv(GL_VIEWPORT, viewport);
      glSelectBuffer(BUF_SIZE, selectBuffer);
      glRenderMode(GL_SELECT);

      // Initialize namestack and push 0, so that calls to
glLoadName don't cause stack underflows.
      glInitNames();
      glPushName(0);

      // Defining the pick matrix to be used during the selection
rendering pass
      glMatrixMode(GL_PROJECTION);
      glPushMatrix();
      glLoadIdentity();
      gluPickMatrix(x, viewport[3] - y, 5, 5, viewport);
      gluPerspective(60, 1, 0.01, 100);

      // Draw quads in selection mode. The routine loads their
names elsewhere.
      glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    drawer->drawAllQuads(GL_SELECT);
    drawer->drawAllRectPars(GL_SELECT);
    drawer->drawAllRectPars(GL_SELECT, proj, texture_navigator,
proj_prog);
      glPopMatrix();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();

      // Resetting matrix mode
      glMatrixMode(GL_MODELVIEW);

      // Resetting render mode and processing hits
```

```
        hits = glRenderMode(GL_RENDER);
        processHits(hits, selectBuffer);
}

void Navigator::keyboard(unsigned char key, int x, int y) {

    float epsilon = 0.001;
    switch (key) {

        case 97:
            left_pressed = true;
            break;
        case 99:
            if (offset <= 0.02 + epsilon && offset >= 0.02 -
epsilon)
                offset = 0.002;
            else
                offset = 0.02;
            break;
        case 115:
            down_pressed = true;
            break;
        case 100:
            right_pressed = true;
            break;
        case 119:
            up_pressed = true;
            break;
            case 113:
            case 27:
                  exit(0);
                  break;
            case 127:
                  if (mode == GL_SELECT) {
                if (selection < 100 && selection != NO_HITS) {
                    if (!drawer->isQuadsEmpty())
                                drawer->removeQuad(selection);
                }
                else {
                    if (!drawer->isBuildingsEmpty() && selection !=
NO_HITS)
                        drawer->remove_building(selection-100);
                }
                        glutPostRedisplay();
                  }
            break;
        case 43:
            // Has a side been clicked? (i.e. is the pointer
pointing at something non-dummy?)
            if (selected_side->get_a() != selected_side->get_c() &&
mode == GL_SELECT && selection != NO_HITS) {

                // Extract pointers to all sides of the selected
building
                Quad * sides[NUM_SIDES];
                drawer->get_selected_building(selection-100)-
>returnAllQuads(sides);
```

```
                // Direction of scaling. We want the "+"-key to
enlarge the building, so we translate vertices
                // in direction of its normal vector.
                Vec3f edge_1 = selected_side->get_b() -
selected_side->get_a();
                Vec3f edge_2 = selected_side->get_c() -
selected_side->get_a();
                Vec3f surface_normal = normalize(cross(edge_1,
edge_2));

                bool a_checked_out = false, b_checked_out = false,
c_checked_out = false, d_checked_out = false;
                int iterator_of_selected_side;

                // Traverse all points in order to determine, if a
point coincides with one on the selected side.
                // If so, these points need to be moved alongside
the selected quad.
                for (int i = 0; i < 6; i++) {

                    if (sides[i]->get_a() == selected_side->get_a()
|| sides[i]->get_a() == selected_side->get_b() || sides[i]->get_a()
== selected_side->get_c() || sides[i]->get_a() == selected_side-
>get_d()) {
                        a_checked_out = true;
                    }
                    if (sides[i]->get_b() == selected_side->get_a()
|| sides[i]->get_b() == selected_side->get_b() || sides[i]->get_b()
== selected_side->get_c() || sides[i]->get_b() == selected_side-
>get_d()) {
                        b_checked_out = true;
                    }
                    if (sides[i]->get_c() == selected_side->get_a()
|| sides[i]->get_c() == selected_side->get_b() || sides[i]->get_c()
== selected_side->get_c() || sides[i]->get_c() == selected_side-
>get_d()) {
                        c_checked_out = true;
                    }
                    if (sides[i]->get_d() == selected_side->get_a()
|| sides[i]->get_d() == selected_side->get_b() || sides[i]->get_d()
== selected_side->get_c() || sides[i]->get_d() == selected_side-
>get_d()) {
                        d_checked_out = true;
                    }

                    // Did all points check out? If so, it's
because this side is the selected one.
                    // We'd like to not move its vertices until
after the for-loop, because
                    // we're operating on a pointer-basis here. If
we move it now, it's very likely to
                    // screw up the upcoming comparison with the
points of the remaining sides.
                    if (a_checked_out && b_checked_out &&
c_checked_out && d_checked_out) {
                        iterator_of_selected_side = i;
```

```
                    a_checked_out = false;
                    b_checked_out = false;
                    c_checked_out = false;
                    d_checked_out = false;
                }

                // If this isn't the case, we move those of the
points that did check out, i.e. those that
                // coincided with the side we selected for
reshaping.
                else {
                    if(a_checked_out) {
                        sides[i]->set_a(sides[i]->get_a() +
offset * surface_normal);
                        a_checked_out = false;
                    }
                    if(b_checked_out) {
                        sides[i]->set_b(sides[i]->get_b() +
offset * surface_normal);
                        b_checked_out = false;
                    }
                    if(c_checked_out) {
                        sides[i]->set_c(sides[i]->get_c() +
offset * surface_normal);
                        c_checked_out = false;
                    }
                    if(d_checked_out) {
                        sides[i]->set_d(sides[i]->get_d() +
offset * surface_normal);
                        d_checked_out = false;
                    }
                }
            }
            // Finally, we move the vertices that belong to the
selected side:
            sides[iterator_of_selected_side]-
>set_a(sides[iterator_of_selected_side]->get_a() + offset *
surface_normal);
            sides[iterator_of_selected_side]-
>set_b(sides[iterator_of_selected_side]->get_b() + offset *
surface_normal);
            sides[iterator_of_selected_side]-
>set_c(sides[iterator_of_selected_side]->get_c() + offset *
surface_normal);
            sides[iterator_of_selected_side]-
>set_d(sides[iterator_of_selected_side]->get_d() + offset *
surface_normal);
            glutPostRedisplay();
        }
        break;
    case 45:
        // Has a side been clicked? (i.e. is the pointer
pointing at something non-dummy?)
        if (selected_side->get_a() != selected_side->get_c() &&
mode == GL_SELECT && selection != NO_HITS) {
```

```
                // Extract pointers to all sides of the selected
building
                Quad * sides[NUM_SIDES];
                drawer->get_selected_building(selection-100)-
>returnAllQuads(sides);

                // Direction of scaling. We want the "-"-key to
shrink the building, so we translate vertices
                // in the opposite direction of their normal
vector.
                Vec3f edge_1 = selected_side->get_b() -
selected_side->get_a();
                Vec3f edge_2 = selected_side->get_c() -
selected_side->get_a();
                Vec3f surface_normal = normalize(cross(edge_1,
edge_2));

                bool a_checked_out = false, b_checked_out = false,
c_checked_out = false, d_checked_out = false;
                int iterator_of_selected_side;

                // Traverse all points in order to determine, if a
point coincides with one on the selected side.
                // If so, these points need to be moved alongside
the selected quad.
                for (int i = 0; i < 6; i++) {

                    if (sides[i]->get_a() == selected_side->get_a()
|| sides[i]->get_a() == selected_side->get_b() || sides[i]->get_a()
== selected_side->get_c() || sides[i]->get_a() == selected_side-
>get_d()) {
                        a_checked_out = true;
                    }
                    if (sides[i]->get_b() == selected_side->get_a()
|| sides[i]->get_b() == selected_side->get_b() || sides[i]->get_b()
== selected_side->get_c() || sides[i]->get_b() == selected_side-
>get_d()) {
                        b_checked_out = true;
                    }
                    if (sides[i]->get_c() == selected_side->get_a()
|| sides[i]->get_c() == selected_side->get_b() || sides[i]->get_c()
== selected_side->get_c() || sides[i]->get_c() == selected_side-
>get_d()) {
                        c_checked_out = true;
                    }
                    if (sides[i]->get_d() == selected_side->get_a()
|| sides[i]->get_d() == selected_side->get_b() || sides[i]->get_d()
== selected_side->get_c() || sides[i]->get_d() == selected_side-
>get_d()) {
                        d_checked_out = true;
                    }

                    // Did all points check out? If so, it's
because this side is the selected one.
                    // We'd like to not move its vertices until
after the for-loop, because
```

```
                    // we're operating on a pointer-basis here. If
we move it now, it's very likely to
                    // screw up the upcoming comparison with the
points of the remaining sides.

                    if (a_checked_out && b_checked_out &&
c_checked_out && d_checked_out) {
                        iterator_of_selected_side = i;
                        a_checked_out = false;
                        b_checked_out = false;
                        c_checked_out = false;
                        d_checked_out = false;
                    }
                    // If this isn't the case, we move those of the
points that did check out, i.e. those that
                    // coincided with the side we selected for
reshaping.
                    else {
                        if(a_checked_out) {
                            sides[i]->set_a(sides[i]->get_a() -
offset * surface_normal);
                            a_checked_out = false;
                        }
                        if(b_checked_out) {
                            sides[i]->set_b(sides[i]->get_b() -
offset * surface_normal);
                            b_checked_out = false;
                        }
                        if(c_checked_out) {
                            sides[i]->set_c(sides[i]->get_c() -
offset * surface_normal);
                            c_checked_out = false;
                        }
                        if(d_checked_out) {
                            sides[i]->set_d(sides[i]->get_d() -
offset * surface_normal);
                            d_checked_out = false;
                        }
                    }
                }
                // Finally, we move the vertices that belong to the
selected side:
                sides[iterator_of_selected_side]-
>set_a(sides[iterator_of_selected_side]->get_a() - offset *
surface_normal);
                sides[iterator_of_selected_side]-
>set_b(sides[iterator_of_selected_side]->get_b() - offset *
surface_normal);
                sides[iterator_of_selected_side]-
>set_c(sides[iterator_of_selected_side]->get_c() - offset *
surface_normal);
                sides[iterator_of_selected_side]-
>set_d(sides[iterator_of_selected_side]->get_d() - offset *
surface_normal);
                glutPostRedisplay();
            }
            break;
```

```
		default:
			break;
	}
}

void Navigator::animate(float secs) {
	if(up_pressed == true && mode == FREE_FLY) {
		cam_pos += 20*secs*cam_dir;
		up_pressed = false;
		glutSetWindow(main_window);
		glutPostRedisplay();
	}
	else if(down_pressed == true && mode == FREE_FLY) {
		cam_pos -= 20*secs*cam_dir;
		down_pressed = false;
		glutSetWindow(main_window);
		glutPostRedisplay();
	}
	else if(left_pressed == true && mode == FREE_FLY) {
		Vec3f up_vector = Vec3f(0.0, 1.0, 0.0);
		Vec3f left_addition_vector = CGLA::cross(up_vector,
cam_dir);
		cam_pos += 15*secs*left_addition_vector;
		left_pressed = false;
		glutSetWindow(main_window);
		glutPostRedisplay();
	}
	else if(right_pressed == true && mode == FREE_FLY) {
		Vec3f down_vector = Vec3f(0.0, -1.0, 0.0);
		Vec3f right_addition_vector = CGLA::cross(down_vector,
cam_dir);
		cam_pos += 15*secs*right_addition_vector;
		right_pressed = false;
		glutSetWindow(main_window);
		glutPostRedisplay();
	}
	else if(wheel_down == true && mode == ROTATE_CAM){
		cam_pos -= 100*secs*cam_dir;
		wheel_down = false;
		glutSetWindow(main_window);
		glutPostRedisplay();
	}
	else if(wheel_up == true && mode == ROTATE_CAM){
		cam_pos += 100*secs*cam_dir;
		wheel_up = false;
		glutSetWindow(main_window);
		glutPostRedisplay();
	}
}

void Navigator::special_keys(int key, int x, int y) {

    if (mode == GL_SELECT) {
		switch(key) {
			case GLUT_KEY_UP:

				break;
```

```
                    case GLUT_KEY_DOWN:

                          break;
                    case GLUT_KEY_LEFT:
                  if(selection >= 100 && selection != NO_HITS) {
                      // Retrieving sides of selected building:
                      Quad * sides[NUM_SIDES];
                      drawer->get_selected_building(selection-100)-
>returnAllQuads(sides);

                      // Setting up vector around which to rotate as
well as center of rotation
                      Vec3f building_center = drawer-
>get_selected_building(selection-100)->get_center();
                      Vec3f up_vector = Vec3f(0.0, 1.0, 0.0);
                      float azimuth = -2*PI/360;

                      //// Azimuth rotation matrix ////
                      Mat3x3f azimuth_rotation =
Mat3x3f(Vec3f(cos(azimuth), 0, sin(azimuth)), Vec3f(0,1,0), Vec3f(-
sin(azimuth), 0, cos(azimuth)));

                      // Traverse all points of all sides, rotating
with the matrix

                      Vec3f rotated_point; // Auxiliary, temporary
point used inside for-loop
                      for (int i = 0; i < 6; i++) {
                          // Retrieve a
                          rotated_point = sides[i]->get_a();

                          // Subtract center of building, rotate, and
re-add center of building
                          rotated_point -= building_center;
                          rotated_point = azimuth_rotation *
rotated_point;
                          rotated_point += building_center;

                          // Point is now rotated relative to center
of building, so we set the new point
                          sides[i]->set_a(rotated_point);

                          // Retrieve b
                          rotated_point = sides[i]->get_b();

                          // Subtract center of building, rotate, and
re-add center of building
                          rotated_point -= building_center;
                          rotated_point = azimuth_rotation *
rotated_point;
                          rotated_point += building_center;

                          // Point is now rotated relative to center
of building, so we set the new point
                          sides[i]->set_b(rotated_point);

                          // Retrieve c
```

```
                        rotated_point = sides[i]->get_c();

                        // Subtract center of building, rotate, and
re-add center of building
                        rotated_point -= building_center;
                        rotated_point = azimuth_rotation *
rotated_point;
                        rotated_point += building_center;

                        // Point is now rotated relative to center
of building, so we set the new point
                        sides[i]->set_c(rotated_point);

                        // Retrieve a
                        rotated_point = sides[i]->get_d();

                        // Subtract center of building, rotate, and
re-add center of building
                        rotated_point -= building_center;
                        rotated_point = azimuth_rotation *
rotated_point;
                        rotated_point += building_center;

                        // Point is now rotated relative to center
of building, so we set the new point
                        sides[i]->set_d(rotated_point);

                        // All points of this quad have now been
rotated. If the quad is textured, however,
                        // we will also need to deal with its
projector location and direction
                        if(sides[i]->is_textured()) {
                            rotated_point = sides[i]-
>get_projector_position();

                            rotated_point -= building_center;
                            rotated_point = azimuth_rotation *
rotated_point;
                            rotated_point += building_center;

                            sides[i]-
>set_projector_position(rotated_point);

                            rotated_point = sides[i]-
>get_projector_direction();
                            rotated_point = azimuth_rotation *
rotated_point;
                            sides[i]-
>set_projector_direction(rotated_point);
                        }

                    }

                glutPostRedisplay();
                }
                        break;
                  case GLUT_KEY_RIGHT:
```

```
                    if(selection >= 100 && selection != NO_HITS) {
                        // Retrieving sides of selected building:
                        Quad * sides[NUM_SIDES];
                        drawer->get_selected_building(selection-100)-
>returnAllQuads(sides);

                        // Setting up vector around which to rotate as
well as center of rotation
                        Vec3f building_center = drawer-
>get_selected_building(selection-100)->get_center();
                        Vec3f up_vector = Vec3f(0.0, 1.0, 0.0);
                        float azimuth = 2*PI/360;

                        //// Azimuth rotation matrix ////
                        Mat3x3f azimuth_rotation =
Mat3x3f(Vec3f(cos(azimuth), 0, sin(azimuth)), Vec3f(0,1,0), Vec3f(-
sin(azimuth), 0, cos(azimuth)));

                        // Traverse all points of all sides, rotating
with the matrix

                        Vec3f rotated_point; // Auxiliary, temporary
point used inside for-loop
                        for (int i = 0; i < 6; i++) {
                            // Retrieve a
                            rotated_point = sides[i]->get_a();

                            // Subtract center of building, rotate, and
re-add center of building
                            rotated_point -= building_center;
                            rotated_point = azimuth_rotation *
rotated_point;
                            rotated_point += building_center;

                            // Point is now rotated relative to center
of building, so we set the new point
                            sides[i]->set_a(rotated_point);

                            // Retrieve b
                            rotated_point = sides[i]->get_b();

                            // Subtract center of building, rotate, and
re-add center of building
                            rotated_point -= building_center;
                            rotated_point = azimuth_rotation *
rotated_point;
                            rotated_point += building_center;

                            // Point is now rotated relative to center
of building, so we set the new point
                            sides[i]->set_b(rotated_point);

                            // Retrieve c
                            rotated_point = sides[i]->get_c();

                            // Subtract center of building, rotate, and
re-add center of building
```

```
                        rotated_point -= building_center;
                        rotated_point = azimuth_rotation *
rotated_point;
                        rotated_point += building_center;

                        // Point is now rotated relative to center
of building, so we set the new point
                        sides[i]->set_c(rotated_point);

                        // Retrieve a
                        rotated_point = sides[i]->get_d();

                        // Subtract center of building, rotate, and
re-add center of building
                        rotated_point -= building_center;
                        rotated_point = azimuth_rotation *
rotated_point;
                        rotated_point += building_center;

                        // Point is now rotated relative to center
of building, so we set the new point
                        sides[i]->set_d(rotated_point);

                        // All points of this quad have now been
rotated. If the quad is textured, however,
                        // we will also need to deal with its
projector location and direction
                        if(sides[i]->is_textured()) {
                            rotated_point = sides[i]-
>get_projector_position();

                            rotated_point -= building_center;
                            rotated_point = azimuth_rotation *
rotated_point;
                            rotated_point += building_center;

                            sides[i]-
>set_projector_position(rotated_point);

                            rotated_point = sides[i]-
>get_projector_direction();
                            rotated_point = azimuth_rotation *
rotated_point;
                            sides[i]-
>set_projector_direction(rotated_point);
                        }

                    }

                glutPostRedisplay();
                }
                        break;
            default:
                break;
        }
      }
}
```

```
void Navigator::mouse_motion(int x, int y) {

	// Extruding quad to form a building
	if(mode == CUBE_MODE) {
		if(firsttime) {
			select(x,y);
			eye2bottom.set_direction(x,y,cam_pos, cam_dir);
			eye2bottom.set_origin(cam_pos);
			eye2bottom.intersect_with_plane(Vec3f(0,1,0), 0);
			dist2bottom = eye2bottom.get_parameter();
			firsttime = false;
		}
		else {
			eye2top.set_direction(x, y, cam_pos, cam_dir);
			eye2top.set_origin(cam_pos);
			angle =
eye2top.compute_angle_between_rays(eye2bottom.get_direction(),
eye2top.get_direction());
			height_end = 0.94*dist2bottom*angle;
			glutSetWindow(main_window);
			glutPostRedisplay();
		}
	}

    else if (mode == FREE_FLY) {

        if (firsttime) {
            azimuth_begin = x;
            azimuth_end = x;
            zenith_begin = y;
            zenith_end = y;
            firsttime = false;
        }

        else {
            azimuth_end = x;
            zenith_end = y;
            azimuth = azimuth_begin - azimuth_end;
            zenith = zenith_begin - zenith_end;
            firsttime = true;
        }
        if (azimuth > 0 && azimuth < 10) {
            Vec3f up_vector = Vec3f(0.0, 1.0, 0.0);
            Vec3f left_addition_vector = CGLA::cross(up_vector,
cam_dir);
            cam_dir += 0.01*azimuth*left_addition_vector;
            cam_dir.normalize();
            glutSetWindow(main_window);
            glutPostRedisplay();
        }
        else if (azimuth < 0 && azimuth > -10) {
            Vec3f down_vector = Vec3f(0.0, -1.0, 0.0);
            Vec3f right_addition_vector = CGLA::cross(down_vector,
cam_dir);
            cam_dir -= 0.01*azimuth*right_addition_vector;
            cam_dir.normalize();
```

```
            glutSetWindow(main_window);
            glutPostRedisplay();
        }
        if (zenith > 0 && zenith < 10) {
            Vec3f up_vector = Vec3f(0.0, 1.0, 0.0);
            cam_dir += 0.01*zenith*up_vector;
            cam_dir.normalize();
            glutSetWindow(main_window);
            glutPostRedisplay();
        }
        else if (zenith < 0 && zenith > -10) {
            Vec3f down_vector = Vec3f(0.0, -1.0, 0.0);
            cam_dir -= 0.01*zenith*down_vector;
            cam_dir.normalize();
            glutSetWindow(main_window);
            glutPostRedisplay();
        }
    }

      // Drawing a quad in the xz-plane
      else if(mode == SQUARE_MODE) {
            if(firsttime) {
                  ray_pick(x,y);
                  a = point;
                  c = point;
                  firsttime = false;
            }
            else {
                  ray_pick(x,y);
                  c = point;
            glutSetWindow(main_window);
                  glutPostRedisplay();
            }
    }
    else if (mode == ROTATE_CAM) {
        Vec3f at_point =
cam_pos+dist_cam_rot_center*normalize(cam_dir);
        Vec3f up_vector = Vec3f(0.0, 1.0, 0.0);
        Vec3f zenith_axis_of_rotation = normalize(cross(cam_dir,
up_vector));

        if (firsttime) {
            azimuth_begin = x;
            azimuth_end = x;
            zenith_begin = y;
            zenith_end = y;
            firsttime = false;
        }

        else {
            azimuth_end = x;
            zenith_end = y;
            azimuth = (2*PI/WINDOW_X)*(azimuth_begin -
azimuth_end);
            zenith = 30*(2*PI/WINDOW_Y)*(zenith_begin -
zenith_end);
            firsttime = true;
```

```
        }

        //// Azimuth rotation ////
        Mat3x3f azimuth_rotation = Mat3x3f(Vec3f(cos(azimuth), 0,
sin(azimuth)), Vec3f(0,1,0), Vec3f(-sin(azimuth), 0,
cos(azimuth)));

        //// Zenith rotation ////
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glLoadIdentity();
        glRotatef(zenith, zenith_axis_of_rotation[0],
zenith_axis_of_rotation[1], zenith_axis_of_rotation[2]);
        GLfloat zenith_rotation4x4[16];
        glGetFloatv(GL_MODELVIEW_MATRIX, zenith_rotation4x4);
        glPopMatrix();

        Mat3x3f zenith_rotation =
Mat3x3f(Vec3f(zenith_rotation4x4[0], zenith_rotation4x4[1],
zenith_rotation4x4[2]),

Vec3f(zenith_rotation4x4[4], zenith_rotation4x4[5],
zenith_rotation4x4[6]),

Vec3f(zenith_rotation4x4[8], zenith_rotation4x4[9],
zenith_rotation4x4[10]));
        cam_pos -= at_point;
        cam_pos = azimuth_rotation * zenith_rotation * cam_pos;
        cam_pos += at_point;
        cam_dir = azimuth_rotation * zenith_rotation * cam_dir;
        glutSetWindow(main_window);
        glutPostRedisplay();
    }
    else if(mode == ROTATE_PROJECTOR && selection != NO_HITS) {
        Vec3f center_of_rotation = drawer-
>get_selected_building(selection-100)->get_center();

        if (firsttime) {
            azimuth_begin = x;
            azimuth_end = x;
            zenith_begin = y;
            zenith_end = y;
            firsttime = false;
        }

        else {
            azimuth_end = x;
            zenith_end = y;
            azimuth = (2*PI/WINDOW_X)*(azimuth_begin -
azimuth_end);
            zenith = 30*(2*PI/WINDOW_Y)*(zenith_begin -
zenith_end);
            firsttime = true;
        }
        if (zenith > -3.0 && zenith < 3.0 && azimuth > -3.0 &&
azimuth < 3.0)
```

```
            proj->rotate_projector(center_of_rotation, -azimuth,
zenith);
        glutSetWindow(main_window);
        glutPostRedisplay();
    }
}


void Navigator::mouse(int button, int state, int x, int y) {

    // Select whatever
      if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN && mode
== GL_SELECT) {
        select(x,y);
        depth_pick(x,y);
        if (selection >= 100 && selection != NO_HITS) {
            selected_side = drawer-
>get_selected_building(selection-100)->getQuad(point);

        }
    }
    // User is releasing button after having drawn quad on
groundlevel, so we store it
      else if(button == GLUT_LEFT_BUTTON && state == GLUT_UP &&
mode == SQUARE_MODE) {
            drawer->storeQuad(a,c);
            firsttime = true;
      }
    // User is releasing button after extruded building from
groundlevel, so we store it and delete the
    // original quad, resetting drawing data
      else if(button == GLUT_LEFT_BUTTON && state == GLUT_UP &&
mode == CUBE_MODE) {
            if (!drawer->isQuadsEmpty() && selection != NO_HITS) {
                  drawer->storeRectPar(drawer-
>get_selected_quad(selection)->get_a(), drawer-
>get_selected_quad(selection)->get_c() , height_end);
                  drawer->removeQuad(selection);
            }
            height_end = 0;
            firsttime = true;
      }
    // User releases button after rotating the world, so we reset
the associated data
    else if(button == GLUT_LEFT_BUTTON && state == GLUT_UP && mode
== ROTATE_CAM) {
        firsttime = true;
    }
    // User rolls the wheel.
      else if (state == GLUT_UP) {
            if(button == GLUT_WHEEL_DOWN && mode == ROTATE_CAM)
                  wheel_down = true;
            if(button == GLUT_WHEEL_UP && mode == ROTATE_CAM)
                  wheel_up = true;

      }
    // User has clicked on the texturize button.
```

```
    else if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN &&
mode == TEXTURIZE_WORLD) {

        // Did the user hit a building with the click?
        select(x,y);

        // If so, we process the request
        if (selection >= 100 && selection != NO_HITS) {

            // Extract point clicked on by user
            depth_pick(x,y);

            // Extract building clicked on and attempt to identify
clicked side
            Quad * pClicked_side = drawer-
>get_selected_building(selection-100)->getQuad(point);

            // Now, was the identification succesful? If not, our
pointer will point to a null-quad, i.e. a dummy
            // quad with zeroes for all coordinates:
            if (pClicked_side->get_a() != pClicked_side->get_c()) {
                // Coordinates a and c are inequal - We've got a
hit, so we associate current projector data with
                // quad's projector_data struct, and set the quad
to be textured:
                pClicked_side->set_projector_direction(proj-
>get_direction());
                pClicked_side->set_projector_position(proj-
>get_position());
                pClicked_side-
>set_texture_object_id(*texture_navigator->get_live_var());
                pClicked_side->set_textured(true);
            }
        }
    }
    else if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN &&
mode == AUTOPOSITION_PROJECTOR_MODE) {

        // Did the user hit a building with the click?
        select(x,y);

        // If so, we process the request
        if (selection >= 100 && selection != NO_HITS) {

            // Extract point clicked on by user
            depth_pick(x,y);

            // Extract building clicked on and attempt to identify
clicked side
            Quad * pClicked_side = drawer-
>get_selected_building(selection-100)->getQuad(point);

            if (pClicked_side->get_a() != pClicked_side->get_c()) {
                Vec3f quad_midpoint =  pClicked_side->get_c() + 0.5
* (pClicked_side->get_a() - pClicked_side->get_c());
```

```
                Vec3f surface_normal =
normalize(cross(pClicked_side->get_b() - pClicked_side->get_a(),
pClicked_side->get_c() - pClicked_side->get_a()));

                proj-
>set_position(quad_midpoint+dist_proj_rot_center*surface_normal);
                proj->set_direction(-surface_normal);
                glutPostRedisplay();
            }
        }
    }
}

// Picking function using ray casting. Intersects with xz-plane to
produce quad-drawing coordinates:

void Navigator::ray_pick(int x, int y) {

      // Extract viewport:
      GLint viewport[4];
      glGetIntegerv(GL_VIEWPORT, viewport);

      // Extract modelview and projection matrices:
      glLoadIdentity();

      // gluLookAt is used to specify viewing parameters.
      gluLookAt(  cam_pos[0], cam_pos[1], cam_pos[2], // Camera-
position
                  cam_pos[0]+cam_dir[0], cam_pos[1]+cam_dir[1],
cam_pos[2]+cam_dir[2], // At-point
                              0.0, 1.0, 0.0); // Up-vector (Camera
orientation)

      double mvmat[16];
      glGetDoublev(GL_MODELVIEW_MATRIX, mvmat);

      // Copy the projection matrix. We assume it is unchanged.
      double prjmat[16];
      glGetDoublev(GL_PROJECTION_MATRIX, prjmat);

      // Now, use two different calls to gluUnproject to obtain ray
direction:
      double z_near_x, z_near_y, z_near_z;
      gluUnProject(x, viewport[3]-y, 0.0, mvmat, prjmat, viewport,
&z_near_x, &z_near_y, &z_near_z);

      double z_far_x, z_far_y, z_far_z;
      gluUnProject(x, viewport[3]-y, 1.0, mvmat, prjmat, viewport,
&z_far_x, &z_far_y, &z_far_z);

      Vec3f z_near = Vec3f(z_near_x, z_near_y, z_near_z);
      Vec3f z_far = Vec3f(z_far_x, z_far_y, z_far_z);

      // Pick ray's direction is now z_far - z_near. Its origin is
the view location:
      Vec3f ray_dir = normalize(z_far - z_near);
      Ray pick_ray = Ray(cam_pos, ray_dir);
```

```
        pick_ray.intersect_with_plane(Vec3f(0,1,0), 0);
        point = pick_ray.get_point_of_intersection();
}

// Depth picking function. Function is directly copied with minor
modifications from the framework handout from
// 02563, Virtual Reality Systems, courtesy of JAB.

bool Navigator::depth_pick(int x, int y) {

        // Enquire about the viewport dimensions
        GLint viewport[4];
        glGetIntegerv(GL_VIEWPORT, viewport);

        // Get the minimum and maximum depth values.
        float minmax_depth[2];
        glGetFloatv(GL_DEPTH_RANGE, minmax_depth);

        // Read a single pixel at the position of the mouse cursor.
        float depth;
        glReadPixels(x, viewport[3]-y, 1,1, GL_DEPTH_COMPONENT,
                                                     GL_FLOAT, (void*)
&depth);

        // If the depth corresponds to the far plane, we clicked on
the
        // background.
        if(depth == minmax_depth[1])
        return false;

        // The lines below copy the viewing transformation from
OpenGL
        // to local variables. The call to gluLookAt must have
exactly
        // the same parameters as when the scene is drawn.
        glLoadIdentity();

        // gluLookAt is used to specify viewing parameters.
        gluLookAt(   cam_pos[0], cam_pos[1], cam_pos[2], // Camera-
position
                     cam_pos[0]+cam_dir[0], cam_pos[1]+cam_dir[1],
cam_pos[2]+cam_dir[2], // At-point
                                 0.0, 1.0, 0.0); // Up-vector (Camera
orientation)

        double mvmat[16];
        glGetDoublev(GL_MODELVIEW_MATRIX, mvmat);

        // Copy the projection matrix. We assume it is unchanged.
        double prjmat[16];
        glGetDoublev(GL_PROJECTION_MATRIX, prjmat);

        // Now unproject the point from screen to world coordinates.
        double ox, oy, oz;
        gluUnProject(x,viewport[3]-y,depth,
                                                     mvmat,prjmat,viewport,
                                                     &ox, &oy, &oz);
```

```
        point = Vec3f(ox,oy,oz);
        return true;
}

Vec3f Navigator::getDirection() {
        return cam_dir;
}

Vec3f Navigator::getPosition() {
        return cam_pos;
}

Vec3f Navigator::get_a() {
        return a;
}

Vec3f Navigator::get_c() {
        return c;
}

float Navigator::get_height() {
        return height_end;
}

void Navigator::set_main_window(int window_id) {
        main_window = window_id;
}

void Navigator::set_tex_window(int window_id) {
        tex_window = window_id;
}

int Navigator::get_main_window() {
        return main_window;
}

int Navigator::get_tex_window() {
        return tex_window;
}

void Navigator::set_mode(int new_mode) {
        mode = new_mode;
}

int Navigator::get_mode() {
        return mode;
}

int Navigator::get_current_selection() {
        return selection;
}

void Navigator::set_cam_dist_to_rot_center(float newdist) {
    dist_cam_rot_center = newdist;
}
```

```
void Navigator::set_proj_dist_to_rot_center(float newdist) {
    dist_proj_rot_center = newdist;
}
```

C.8 Projector.h

```
#ifndef _PROJECTOR_GUARD_
#define _PROJECTOR_GUARD_
#include "CGLA/Vec3f.h"
#include "Navigator.h"
#include "Definitions.h"

class Projector {

    CGLA::Vec3f ProjDir, ProjPos;

    Navigator * navigator;

public:

    Projector(CGLA::Vec3f, CGLA::Vec3f, Navigator*);

    void move_projector_pos(CGLA::Vec3f);

    void move_projector_dir(CGLA::Vec3f);

    CGLA::Vec3f get_position();

    CGLA::Vec3f get_direction();

    void set_up_projector();

    void set_temp_projector_posndir(CGLA::Vec3f, CGLA::Vec3f);

    void drawDirection();

    void set_direction(CGLA::Vec3f);

    void set_position(CGLA::Vec3f);

    void rotate_projector(CGLA::Vec3f, float, float);

};

#endif
```

C.9 Projector.cpp

```
#include "Projector.h"
#include "GL/glut.h"
#include "cgla/Mat3x3f.h"

using namespace CGLA;
```

```
Projector::Projector(Vec3f init_pos, Vec3f init_dir, Navigator *
nav) {
    this->ProjPos = init_pos;
    this->ProjDir = init_dir;
    this->navigator = nav;
}

using namespace std;

void Projector::drawDirection() {

    glPushMatrix();
    glTranslatef(ProjPos[0],ProjPos[1],ProjPos[2]);
    glutSolidCube(0.1);

    glDisable(GL_LIGHTING);
    glColor3f(0.0, 0.0, 0.0);
    glBegin(GL_LINES);
    glVertex3f(0.0,0.0,0.0);
    glVertex3f(ProjDir[0]*0.3, ProjDir[1]*0.3, ProjDir[2]*0.3);
    glEnd();
    glEnable(GL_LIGHTING);

    glPopMatrix();

}

void Projector::move_projector_dir(Vec3f dir_addition) {

    Vec3f projector_up_vector = Vec3f(0.0, -1.0, 0.0);
    Vec3f zenith_axis_of_rotation = normalize(cross(ProjDir,
projector_up_vector));

    float zenith = dir_addition.length();
    float azimuth = dir_addition.length();

    // Analyzing azimuth/zenith sign
    if (dir_addition[0] > 0) {
        azimuth *= -1.0;
    }
    else if(dir_addition[2] < 0) {
        zenith *= -1.0;
    }

    // Generating projector zenith rotation matrix
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
    glRotatef(zenith, zenith_axis_of_rotation[0],
zenith_axis_of_rotation[1], zenith_axis_of_rotation[2]);
    GLfloat zenith_rotation4x4[16];
    glGetFloatv(GL_MODELVIEW_MATRIX, zenith_rotation4x4);
    glPopMatrix();

    Mat3x3f zenith_rotation = Mat3x3f(Vec3f(zenith_rotation4x4[0],
zenith_rotation4x4[1], zenith_rotation4x4[2]),
```

```
        Vec3f(zenith_rotation4x4[4], zenith_rotation4x4[5],
zenith_rotation4x4[6]),
        Vec3f(zenith_rotation4x4[8], zenith_rotation4x4[9],
zenith_rotation4x4[10]));

    // Generating projector azimuth rotation matrix
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
    glRotatef(azimuth, projector_up_vector[0],
projector_up_vector[1], projector_up_vector[2]);
    GLfloat azimuth_rotation4x4[16];
    glGetFloatv(GL_MODELVIEW_MATRIX, azimuth_rotation4x4);
    glPopMatrix();

    Mat3x3f azimuth_rotation =
Mat3x3f(Vec3f(azimuth_rotation4x4[0], azimuth_rotation4x4[1],
azimuth_rotation4x4[2]),
        Vec3f(azimuth_rotation4x4[4], azimuth_rotation4x4[5],
azimuth_rotation4x4[6]),
        Vec3f(azimuth_rotation4x4[8], azimuth_rotation4x4[9],
azimuth_rotation4x4[10]));

    // Rotation matrices complete. Analyzing input and updating
direction as applicable:

    if (dir_addition[0] != 0)
        ProjDir = azimuth_rotation * ProjDir;
    else
        ProjDir = zenith_rotation * ProjDir;

}

void Projector::move_projector_pos(Vec3f pos_addition) {

    Vec3f reverse_dir;

    reverse_dir[0] = -navigator->getDirection()[0];
    reverse_dir[1] = 0;
    reverse_dir[2] = -navigator->getDirection()[2];
    normalize(reverse_dir);

    Vec3f up = Vec3f(0.0, -1.0, 0.0);

    Vec3f left_vec = normalize(cross(-reverse_dir, up));

    if ( pos_addition[2] != 0) {
        ProjPos -= 0.01*pos_addition[2]*reverse_dir;
    }
    else if ( pos_addition[0] != 0) {
        ProjPos -= 0.01*pos_addition[0]*left_vec;
    }
    else if (pos_addition[1] != 0) {
        ProjPos += 0.01*pos_addition;
    }
}
```

```
Vec3f Projector::get_direction() {
    return ProjDir;
}

Vec3f Projector::get_position() {
    return ProjPos;
}

void Projector::set_up_projector() {

    glMatrixMode(GL_TEXTURE);
    glLoadIdentity();

    // Set scale and bias:
      glTranslatef(0.5, 0.5, 0.5);
    glScalef(0.5, 0.5, 0.5);
      glRotatef(180, 0.0, 1.0, 0.0);
    gluPerspective(45.0, 1.0, 0.5, 50.0);
    gluLookAt(ProjPos[0], ProjPos[1], ProjPos[2], // Projector
position

ProjPos[0]+ProjDir[0],ProjPos[1]+ProjDir[1],ProjPos[2]+ProjDir[2],
// Projector at-point
        0.0, -1.0, 0.0); // Projector up-vector
    glMatrixMode(GL_MODELVIEW);

}

void Projector::set_temp_projector_posndir(Vec3f temppos, Vec3f
tempdir) {

    glMatrixMode(GL_TEXTURE);
    glLoadIdentity();

    // Set scale and bias:
    glTranslatef(0.5, 0.5, 0.5);
    glScalef(0.5, 0.5, 0.5);
    glRotatef(180, 0.0, 1.0, 0.0);
    gluPerspective(45.0, 1.0, 0.5, 50.0);
    gluLookAt(temppos[0], temppos[1], temppos[2], // Projector
position

temppos[0]+tempdir[0],temppos[1]+tempdir[1],temppos[2]+tempdir[2],
// Projector at-point
        0.0, -1.0, 0.0); // Projector up-vector

    glMatrixMode(GL_MODELVIEW);
}

void Projector::set_direction(Vec3f newdir) {
    ProjDir = newdir;
}

void Projector::set_position(Vec3f newpos) {
    ProjPos = newpos;
}
```

```
void Projector::rotate_projector(Vec3f center_of_rotation, float
azimuth, float zenith) {
    Vec3f up_vector = Vec3f(0.0, 1.0, 0.0);
    Vec3f zenith_axis_of_rotation = normalize(cross(ProjDir,
up_vector));

    //// Azimuth rotation ////
    Mat3x3f azimuth_rotation = Mat3x3f(Vec3f(cos(azimuth), 0,
sin(azimuth)), up_vector, Vec3f(-sin(azimuth), 0, cos(azimuth)));

    //// Zenith rotation ////
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
    glRotatef(zenith, zenith_axis_of_rotation[0],
zenith_axis_of_rotation[1], zenith_axis_of_rotation[2]);
    GLfloat zenith_rotation4x4[16];
    glGetFloatv(GL_MODELVIEW_MATRIX, zenith_rotation4x4);
    glPopMatrix();

    Mat3x3f zenith_rotation = Mat3x3f(Vec3f(zenith_rotation4x4[0],
zenith_rotation4x4[1], zenith_rotation4x4[2]),
        Vec3f(zenith_rotation4x4[4], zenith_rotation4x4[5],
zenith_rotation4x4[6]),
        Vec3f(zenith_rotation4x4[8], zenith_rotation4x4[9],
zenith_rotation4x4[10]));

    ProjPos -= center_of_rotation;
    ProjPos = azimuth_rotation * zenith_rotation * ProjPos;
    ProjPos += center_of_rotation;
    ProjDir = azimuth_rotation * zenith_rotation * ProjDir;
}
```

C.10 Quad.h

```
#ifndef _QUAD_GUARD_
#define _QUAD_GUARD_

#include "CGLA/Vec3f.h"
#include "Definitions.h"

typedef struct {
    CGLA::Vec3f proj_position;
    CGLA::Vec3f proj_direction;
    bool textured;
    int texture_object_id;
} projector_data;

class Quad {

      CGLA::Vec3f a, b, c, d;
    projector_data projector_info;

public:
```

```
      Quad(){}

      Quad(CGLA::Vec3f, CGLA::Vec3f);

      CGLA::Vec3f get_a();

      CGLA::Vec3f get_b();

      CGLA::Vec3f get_c();

      CGLA::Vec3f get_d();

      void set_a(CGLA::Vec3f);

      void set_b(CGLA::Vec3f);

      void set_c(CGLA::Vec3f);

      void set_d(CGLA::Vec3f);

    void translate_quad(CGLA::Vec3f);

    void set_projector_direction(CGLA::Vec3f);

    void set_projector_position(CGLA::Vec3f);

    CGLA::Vec3f get_projector_direction();

    CGLA::Vec3f get_projector_position();

    void set_textured(bool);

    bool is_textured();

    void set_texture_object_id(int);

    int get_texture_object_id();

};

#endif
```

C.11 Quad.cpp

```
#include "Quad.h"

using namespace CGLA;

Quad::Quad(Vec3f a, Vec3f c) {

      this->a = a;
      this->c = c;

      if (a[0]<c[0]) {
            if(a[2] > c[2]) {
```

```
				this->b[0] = c[0];
				this->b[1] = 0;
				this->b[2] = a[2];

				this->d[0] = a[0];
				this->d[1] = 0;
				this->d[2] = c[2];
			}
			else {
				this->b[0] = a[0];
				this->b[1] = 0;
				this->b[2] = c[2];

				this->d[0] = c[0];
				this->d[1] = 0;
				this->d[2] = a[2];

			}
		}
		else {
			if(a[2] > c[2]) {
				this->b[0] = a[0];
				this->b[1] = 0;
				this->b[2] = c[2];

				this->d[0] = c[0];
				this->d[1] = 0;
				this->d[2] = a[2];

			}
			else {
				this->b[0] = c[0];
				this->b[1] = 0;
				this->b[2] = a[2];

				this->d[0] = a[0];
				this->d[1] = 0;
				this->d[2] = c[2];
			}


		}
	projector_info.texture_object_id = 0;
	projector_info.textured = false;
}

Vec3f Quad::get_a() {
	return a;
}

Vec3f Quad::get_b() {
	return b;
}

Vec3f Quad::get_c() {
	return c;
}
```

```
Vec3f Quad::get_d() {
      return d;
}

void Quad::set_a(Vec3f a) {
      this->a = a;
}

void Quad::set_b(Vec3f b) {
      this->b = b;
}

void Quad::set_c(Vec3f c) {
      this->c = c;
}

void Quad::set_d(Vec3f d) {
      this->d = d;
}

void Quad::translate_quad(Vec3f offset) {

    a += offset;
    b += offset;
    c += offset;
    d += offset;
    projector_info.proj_position += offset;
}

Vec3f Quad::get_projector_position() {
    return projector_info.proj_position;
}

Vec3f Quad::get_projector_direction() {
    return projector_info.proj_direction;
}

void Quad::set_projector_position(Vec3f new_pos) {
    projector_info.proj_position = new_pos;
}

void Quad::set_projector_direction(Vec3f new_dir) {
    projector_info.proj_direction = new_dir;
}

bool Quad::is_textured() {
    return projector_info.textured;
}

void Quad::set_textured(bool textured) {
    projector_info.textured = textured;
}

void Quad::set_texture_object_id(int id) {
    projector_info.texture_object_id = id;
}
```

```
int Quad::get_texture_object_id() {
    return projector_info.texture_object_id;
}
```

C.12 Ray.h

```
#ifndef _RAY_GUARD_
#define _RAY_GUARD_

#include "CGLA/Vec3f.h"

class Ray {

      CGLA::Vec3f origin, direction;

      float t;

public:

      Ray(){};

      Ray(CGLA::Vec3f, CGLA::Vec3f);

      bool intersect_with_plane(CGLA::Vec3f, float);

      CGLA::Vec3f get_point_of_intersection();

      float compute_angle_between_rays(CGLA::Vec3f, CGLA::Vec3f);

      CGLA::Vec3f get_direction();

      float compute_incident_angle(CGLA::Vec3f);

      float compute_height_of_building(float, float, float);

      void set_direction(int, int, CGLA::Vec3f, CGLA::Vec3f);

      void set_origin(CGLA::Vec3f);

      float get_parameter();
};

#endif
```

C.13 Ray.cpp

```
#include "Ray.h"
#include <GL/glut.h>

using namespace std;
using namespace CGLA;

Ray::Ray(Vec3f origin, Vec3f direction) {
```

```
	this->origin = origin;
	this->direction = direction;
	this->t = 0;
}


/// Determines whether ray intersects given plane satisfactorily
and sets t if so
bool Ray::intersect_with_plane(Vec3f normal, float D) {
	// First, compute dot product between plane normal and ray
direction
	float V_d = dot(normal, direction);

	// Second, compute dot product between normal and origin and
add plane distance from origin, D:
	float V_0 = -(dot(normal, origin) + D);

	// If V_d = 0, the ray is parallel to the plane and no
intersection takes place.
	// We take similar action if V_d > 0, because that means the
normal of the plane is pointing away from the ray, i.e.
	// we are pointing at the plane from below it. This program
does not assume the user draws on the plane from beneath,
	// so the function returns false in this case as well.

	if (V_d == 0 || V_d > 0)
		return false;

	// We have intersection
	else {
		t = V_0 / V_d;

		// Do we intersect a plane behind ray origin? If so, we
are not interested:
		if (t < 0)
			return false;
		else {
			// Valid intersection
			return true;
		}

	}

}

/// Returns point of intersection between this ray and the plane
most recently intersected
Vec3f Ray::get_point_of_intersection() {
	return origin + t*direction;
}

/// Computes the angle between this ray and one given as parameter
float Ray::compute_angle_between_rays(Vec3f ray1_dir, Vec3f
ray2_dir) {
	return acos(dot(ray1_dir, ray2_dir));
}
```

```
/// Returns the direction vector of the ray
Vec3f Ray::get_direction() {
      return direction;
}

/// Computes incident angle between this ray and a surface, the
normal of which is given as parameter
float Ray::compute_incident_angle(Vec3f normal) {
      return acos(dot(-direction, normal));
}

float Ray::compute_height_of_building(float A, float C, float t) {
      return 0.5*sin(A)*t/sin(C);
}

void Ray::set_direction(int x, int y, Vec3f cur_pos, Vec3f cam_dir)
{
      // Extract viewport:
      GLint viewport[4];
      glGetIntegerv(GL_VIEWPORT, viewport);

      // Extract modelview and projection matrices:
      glLoadIdentity();

      // gluLookAt is used to specify viewing parameters.
      gluLookAt(  cur_pos[0], cur_pos[1], cur_pos[2], // Camera-
position
            cur_pos[0]+cam_dir[0], cur_pos[1]+cam_dir[1],
cur_pos[2]+cam_dir[2], // At-point
                              0.0, 1.0, 0.0); // Up-vector (Camera
orientation)

      double mvmat[16];
      glGetDoublev(GL_MODELVIEW_MATRIX, mvmat);

      // Copy the projection matrix. We assume it is unchanged.
      double prjmat[16];
      glGetDoublev(GL_PROJECTION_MATRIX, prjmat);

      // Now, use two different calls to gluUnproject to obtain ray
direction:
      double z_near_x, z_near_y, z_near_z;
      gluUnProject(x, viewport[3]-y, 0.0, mvmat, prjmat, viewport,
&z_near_x, &z_near_y, &z_near_z);

      double z_far_x, z_far_y, z_far_z;
      gluUnProject(x, viewport[3]-y, 1.0, mvmat, prjmat, viewport,
&z_far_x, &z_far_y, &z_far_z);

      Vec3f z_near = Vec3f(z_near_x, z_near_y, z_near_z);
      Vec3f z_far = Vec3f(z_far_x, z_far_y, z_far_z);

      // Pick ray's direction is now z_far - z_near. Its origin is
the view location:
      direction = normalize(z_far - z_near);
}
```

```
void Ray::set_origin(Vec3f origin) {
      this->origin = origin;
}

float Ray::get_parameter() {
      return t;
}
```

C.14 RectPar.h

```
#ifndef _RECTPAR_GUARD_
#define _RECTPAR_GUARD_

#include "Quad.h"
#include "CGLA/Vec3f.h"
#include "Definitions.h"

class RectPar {

      Quad bottom_quad, top_quad, side_1, side_2, side_3, side_4;

public:

      RectPar(){};

      RectPar(CGLA::Vec3f, CGLA::Vec3f, float);

      Quad * getQuad(CGLA::Vec3f);

      void returnAllQuads(Quad*[]);

    void translate_building(CGLA::Vec3f);

    CGLA::Vec3f get_center();
};

#endif
```

C.15 RectPar.cpp

```
#include "RectPar.h"

using namespace CGLA;

Quad dummy(Vec3f(0.0,0.0,0.0), Vec3f(0.0,0.0,0.0));

RectPar::RectPar(Vec3f a, Vec3f c, float height) {

      // Set bottom quad of the box (provided as parameters a and
c):
      this->bottom_quad = Quad(a,c);

      // Set top quad of the box. (Calculated as the bottom quad's
vertices, elevated along y-axis by height):
```

```
      this->top_quad.set_a(bottom_quad.get_a() + Vec3f(0.0, height,
0.0));
      this->top_quad.set_b(bottom_quad.get_b() + Vec3f(0.0, height,
0.0));
      this->top_quad.set_c(bottom_quad.get_c() + Vec3f(0.0, height,
0.0));
      this->top_quad.set_d(bottom_quad.get_d() + Vec3f(0.0, height,
0.0));
    this->top_quad.set_textured(false);

      // Set the first side of the box. Its vertices are the a and
b vertices of top and bottom, respectively:
      this->side_1.set_a(bottom_quad.get_a());
      this->side_1.set_b(bottom_quad.get_b());
      this->side_1.set_c(top_quad.get_b());
      this->side_1.set_d(top_quad.get_a());
    this->side_1.set_textured(false);

      // Set the second side of the box. Its vertices are the b and
c vertices of top and bottom, respectively:
      this->side_2.set_a(bottom_quad.get_b());
      this->side_2.set_b(bottom_quad.get_c());
      this->side_2.set_c(top_quad.get_c());
      this->side_2.set_d(top_quad.get_b());
    this->side_2.set_textured(false);

      // Set the third side of the box. Its vertices are the c and
d vertices of top and bottom, respectively:
      this->side_3.set_a(bottom_quad.get_c());
      this->side_3.set_b(bottom_quad.get_d());
      this->side_3.set_c(top_quad.get_d());
      this->side_3.set_d(top_quad.get_c());
    this->side_3.set_textured(false);

      // Set the fourth side of the box. Its vertices are the d and
a vertices of top and bottom, respectively:
      this->side_4.set_a(bottom_quad.get_d());
      this->side_4.set_b(bottom_quad.get_a());
      this->side_4.set_c(top_quad.get_a());
      this->side_4.set_d(top_quad.get_d());
    this->side_4.set_textured(false);
}

/// Returns a quad based on a point, assumed to be on the
building's surface.
Quad * RectPar::getQuad(Vec3f point) {
    int counter = 0;
    int coordinate = 0;
    float epsilon = 0.020;

    /// Check if point is in bottom quad:
    for (int i = 0; i < 3; i++) {
        if (bottom_quad.get_a()[i] < point[i] + epsilon &&
bottom_quad.get_a()[i] > point[i] - epsilon) {
            counter++;
            coordinate = i;
        }
```

```
    }

    if ((bottom_quad.get_b()[coordinate] < point[coordinate] +
epsilon && bottom_quad.get_b()[coordinate] > point[coordinate] -
epsilon)
        && (bottom_quad.get_c()[coordinate] < point[coordinate] +
epsilon && bottom_quad.get_c()[coordinate] > point[coordinate] -
epsilon)
        && (bottom_quad.get_d()[coordinate] < point[coordinate] +
epsilon && bottom_quad.get_d()[coordinate] > point[coordinate] -
epsilon)) {
        counter += 3;
    }

    /// Was that it? If count is 4, then all four corners of the
quad have a coordinate in common with the supplied
    /// point. I.e. they are in the same plane. We may therefore
assume we've found the side that was clicked on.
    if (counter == 4) {
        return &bottom_quad;
    }
    else
        counter = 0;

    /// Control continues if bottom quad was not clicked on. Top
quad is examined:

    for (int i = 0; i < 3; i++) {
        if (top_quad.get_a()[i] < point[i] + epsilon &&
top_quad.get_a()[i] > point[i] - epsilon) {
            counter++;
            coordinate = i;
        }
    }

    if ((top_quad.get_b()[coordinate] < point[coordinate] + epsilon
&& top_quad.get_b()[coordinate] > point[coordinate] - epsilon)
        && (top_quad.get_c()[coordinate] < point[coordinate] +
epsilon && top_quad.get_c()[coordinate] > point[coordinate] -
epsilon)
        && (top_quad.get_d()[coordinate] < point[coordinate] +
epsilon && top_quad.get_d()[coordinate] > point[coordinate] -
epsilon)) {
        counter += 3;
    }

    if (counter == 4) {
        return &top_quad;
    }
    else
        counter = 0;

    // Still no hit, so examination continues to the sides,
checking them one by one:

    for (int i = 0; i < 3; i++) {
```

```
        if (side_1.get_a()[i] < point[i] + epsilon &&
side_1.get_a()[i] > point[i] - epsilon) {
            counter++;
            coordinate = i;
        }
    }

    if ((side_1.get_b()[coordinate] < point[coordinate] + epsilon
&& side_1.get_b()[coordinate] > point[coordinate] - epsilon)
        && (side_1.get_c()[coordinate] < point[coordinate] +
epsilon && side_1.get_c()[coordinate] > point[coordinate] -
epsilon)
        && (side_1.get_d()[coordinate] < point[coordinate] +
epsilon && side_1.get_d()[coordinate] > point[coordinate] -
epsilon)) {
        counter += 3;
    }

    if (counter == 4) {
        return &side_1;
    }
    else
        counter = 0;

    for (int i = 0; i < 3; i++) {
        if (side_2.get_a()[i] < point[i] + epsilon &&
side_2.get_a()[i] > point[i] - epsilon) {
            counter++;
            coordinate = i;
        }
    }

    if ((side_2.get_b()[coordinate] < point[coordinate] + epsilon
&& side_2.get_b()[coordinate] > point[coordinate] - epsilon)
        && (side_2.get_c()[coordinate] < point[coordinate] +
epsilon && side_2.get_c()[coordinate] > point[coordinate] -
epsilon)
        && (side_2.get_d()[coordinate] < point[coordinate] +
epsilon && side_2.get_d()[coordinate] > point[coordinate] -
epsilon)) {
        counter += 3;
    }

    if (counter == 4) {
        return &side_2;
    }
    else
        counter = 0;

    for (int i = 0; i < 3; i++) {
        if (side_3.get_a()[i] < point[i] + epsilon &&
side_3.get_a()[i] > point[i] - epsilon) {
            counter++;
            coordinate = i;
        }
    }
```

```
    if ((side_3.get_b()[coordinate] < point[coordinate] + epsilon
&& side_3.get_b()[coordinate] > point[coordinate] - epsilon)
        && (side_3.get_c()[coordinate] < point[coordinate] +
epsilon && side_3.get_c()[coordinate] > point[coordinate] -
epsilon)
        && (side_3.get_d()[coordinate] < point[coordinate] +
epsilon && side_3.get_d()[coordinate] > point[coordinate] -
epsilon)) {
        counter += 3;
    }

    if (counter == 4) {
        return &side_3;
    }
    else
        counter = 0;

    for (int i = 0; i < 3; i++) {
        if (side_4.get_a()[i] < point[i] + epsilon &&
side_4.get_a()[i] > point[i] - epsilon) {
            counter++;
            coordinate = i;
        }
    }

    if ((side_4.get_b()[coordinate] < point[coordinate] + epsilon
&& side_4.get_b()[coordinate] > point[coordinate] - epsilon)
        && (side_4.get_c()[coordinate] < point[coordinate] +
epsilon && side_4.get_c()[coordinate] > point[coordinate] -
epsilon)
        && (side_4.get_d()[coordinate] < point[coordinate] +
epsilon && side_4.get_d()[coordinate] > point[coordinate] -
epsilon)) {
        counter += 3;
    }

    /// Was that it? If count is 4, then all four corners of the
quad have a coordinate in common with the supplied
    /// point. I.e. they are in the same plane. We may therefore
assume we've found the side that was clicked on.
    if (counter == 4) {
        return &side_4;
    }
    else {
        // At this point, we've been through all sides
unsuccesfully, so we have to assume the point isn't on the
        // surface of the building at all. Thus, we're returning
pointer to global dummy. (Null-quad)
        return &dummy;
    }
}

/// Return pointers to all quads of the box (Pass by reference)
void RectPar::returnAllQuads(Quad * result[]) {

      // Setting indices of supplied array
      result[0] = &bottom_quad;
```

```
        result[1] = &side_1;
        result[2] = &side_2;
        result[3] = &side_3;
        result[4] = &side_4;
        result[5] = &top_quad;

}

/// Translate building by supplied vector
void RectPar::translate_building(Vec3f offset) {

    // Translating individual quads of building
    bottom_quad.translate_quad(offset);
    side_1.translate_quad(offset);
    side_2.translate_quad(offset);
    side_3.translate_quad(offset);
    side_4.translate_quad(offset);
    top_quad.translate_quad(offset);

}

Vec3f RectPar::get_center() {
    return bottom_quad.get_a() + 0.5*(top_quad.get_c() -
bottom_quad.get_a());
}
```

C.16 TexNav.h

```
#ifndef _TEXNAV_GUARD_
#define _TEXNAV_GUARD_

#include "Definitions.h"
#include "Quad.h"

class TexNav {

    int listbox_live_var;

public:

    TexNav();

    void mouse();

    int* get_live_var();

};

#endif
```

C.17 TexNav.cpp

```
#include "TexNav.h"
```

```
using namespace CGLA;

TexNav::TexNav() {
    this->listbox_live_var = 1;
}

void TexNav::mouse() {

}

int *TexNav::get_live_var() {
    return &listbox_live_var;
}
```

C.18 TextureLoader.h

```
#ifndef _TEXTURE_LOADER_GUARD_
#define _TEXTURE_LOADER_GUARD_

#include <iostream>
#include <IL/ilu.h>
#include <GL/glut.h>
#include <vector>

class TextureLoader {

      std::vector<GLuint> textures;

public:

      TextureLoader();

      void load_texture(char*);

      GLuint get_texture(int);

    void print_texture_IDs();

      void init_devIL();

};

#endif
```

C.19 TextureLoader.cpp

```
#include "TextureLoader.h"

using namespace std;

TextureLoader::TextureLoader() {
}

void TextureLoader::load_texture(char* filename) {
```

```
        // Create and bind an image.
        ILuint ImgId;
        GLuint tex;
        ilGenImages(1, &ImgId);
        ilBindImage(ImgId);

        // Load the image - abort program if it fails.
        if(!ilLoadImage(ILstring(filename))) {
                    cout << "Could not load " << filename << endl;
                    exit(0);
              }
        ilConvertImage(IL_RGB, IL_UNSIGNED_BYTE);

        // Get dimensions of image.
        int size_x = ilGetInteger(IL_IMAGE_WIDTH);
        int size_y = ilGetInteger(IL_IMAGE_HEIGHT);
        int bytes_per_pixel = ilGetInteger(IL_IMAGE_BYTES_PER_PIXEL);

        //Set a pointer to point to the first byte of the image.
        const unsigned char* image = ilGetData();

        // ---------------------------------------
        // Use the image as an OpenGL texture
        // ---------------------------------------

        //Create and bind a texture name
        glGenTextures(1, &tex);
        glBindTexture(GL_TEXTURE_2D, tex);

        // Setup the texture interpolation
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);

        // Specify how we deal with wrapping.
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

        // Make sure that the texture combines with light settings
properly
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

        // Build a pyramid of texture images.
        gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, size_x, size_y,
GL_RGB, GL_UNSIGNED_BYTE, image);
        textures.push_back(tex);
}

GLuint TextureLoader::get_texture(int index) {
    return textures.at(index);
}

void TextureLoader::init_devIL() {

        // Initialize image loading library (DevIL)
```

```
        ilInit();
        iluInit();

        // Enable conversion from palette to RGB
        ilEnable(IL_CONV_PAL);

}

void TextureLoader::print_texture_IDs() {
    for (int i = 0; i < textures.size(); i++) {
        cout << textures.at(i) << endl;
    }
}
```

C.20 main.cpp

```
#include "CGLA/Mat4x4f.h"
#include "CGLA/Vec4f.h"
#include <gl/glew.h>
#include "Navigator.h"
#include "Drawing.h"
#include "Glui.h"
#include "TextureLoader.h"
#define WIN32
#include "Timer.h"
#include "Projector.h"
#include "TexNav.h"
#include "FileIO.h"

using namespace std;
using namespace CGLA;

// Utility objects
Navigator navigator;
TextureLoader tex_loader;
TexNav tex_navigator;
CMP::Timer timer;
Projector projector = Projector(Vec3f(3.0, 1.5, 3.0), Vec3f(-3.0,
0, -3.0), &navigator);
Drawing artist;

// Time taken to complete drawing of a scene, plus current texture
dimensions
float frametime, tex_width = 0, tex_height = 0;

// Shader program ID
GLuint proj_prog;

// GLUI pointers:
GLUI * glui_top;
GLUI * glui_bottom;
GLUI * glui_status;
GLUI * glui_tex_top;
GLUI_Translation * quad_building_move;
GLUI_Translation * projector_pan;
```

```
GLUI_Translation * projector_vertical;
GLUI_Translation * projector_direction;
GLUI_Listbox * tex_listbox;

// GLUI live variables
float live_var_obj[2] = {0.0,0.0};
float live_var_proj_xz[2] = {0.0,0.0};
float live_var_proj_y[1] = {0.0};
float live_var_proj_dir[2] = {0.0,0.0};
int live_var_project[1] = {0};
float cam_spinner_live_var[1] = {5.0};
float proj_spinner_live_var[1] = {2.0};


const char* load_file(const char* filename) {
	FILE* f = fopen(filename, "rb");
	fseek(f, 0, SEEK_END);
	long size = ftell(f);
	fseek(f, 0, SEEK_SET);
	char* d = (char*)malloc(size+1);
	fread(d, 1, size, f);
	d[size] = '\0';
	return d;
}

GLuint create_shader(GLenum type, const char* fn) {

	const char* src = load_file(fn);
	GLuint handle = glCreateShader(type);
	glShaderSource(handle, 1, &src, NULL);
	free((char*)src);
	glCompileShader(handle);
	int compiled;
	glGetShaderiv(handle, GL_COMPILE_STATUS, &compiled);
	if (compiled == GL_FALSE){
		int len = 0;
		glGetShaderiv(handle, GL_INFO_LOG_LENGTH, &len);
		char* log = (char*)malloc(len);
		glGetShaderInfoLog(handle, len, NULL, log);
		printf(log); free(log); exit(-1);
		}
	return handle;
}

GLuint create_program(const char* vs, const char* fs){

	GLuint prg = glCreateProgram();

	if (vs)     {
		GLuint vh = create_shader(GL_VERTEX_SHADER, vs);
		glAttachShader(prg, vh);
	}

	if (fs) {
		GLuint fh = create_shader(GL_FRAGMENT_SHADER, fs);
		glAttachShader(prg, fh);
	}
```

```
    glLinkProgram(prg);
    int linked;
    glGetProgramiv(prg, GL_LINK_STATUS, &linked);

    if (linked == GL_FALSE) {
        int len;
        glGetProgramiv(prg, GL_INFO_LOG_LENGTH, &len);
        char* log = (char*)malloc(len);
        glGetProgramInfoLog(prg, len, NULL, log);
        printf(log); free(log); exit(-1);
    }
    return prg;
}

/// Function to be called by glutMouseMotion callback
void mouse_motion(int x, int y) {
    navigator.mouse_motion(x,y);
}

/// Function to be called by glutMouseFunc callback
void mouse(int button, int state, int x, int y) {
    navigator.mouse(button, state, x, y);
}

/// Function to be called by glutSpecialFunc callback
void spec_keyfun(int key, int x, int y) {
    navigator.special_keys(key, x, y);
}

/// Function to be called by glutKeyFunc callback
void keyboard(unsigned char key, int x, int y) {
    navigator.keyboard(key, x, y);
}

/// Function to be called by glutIdleFunc callback
void animate() {
    navigator.animate(frametime);
}

/// Function to be called by glutDisplayFunc callback. This is the
display for the primary window.
void display() {

      // Start the timer to measure frametime
        timer.start();

      glClearColor(0.8, 0.8, 0.8, 0.0);

        /// Clear buffers
        glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // Set up camera
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
```

```
            gluLookAt(navigator.getPosition()[0],
navigator.getPosition()[1], navigator.getPosition()[2], // Camera-
position

      navigator.getPosition()[0]+navigator.getDirection()[0],
navigator.getPosition()[1]+navigator.getDirection()[1],
navigator.getPosition()[2]+navigator.getDirection()[2], // At-point
                              0.0, 1.0, 0.0); // Up-vector (Camera
orientation)
        // Draw auxiliary axes
            artist.drawAxes();

            // Set up projector
        projector.set_up_projector();

        projector.drawDirection();

        if (*tex_navigator.get_live_var() == 0) {
            glBindTexture(GL_TEXTURE_2D,
tex_loader.get_texture(0));
        }
        else
            glBindTexture(GL_TEXTURE_2D,
*tex_navigator.get_live_var());

        if (live_var_project[0])
            glUseProgram(proj_prog);
            glActiveTexture(GL_TEXTURE0);
        glUniform1i(glGetUniformLocation(proj_prog, "tex"), 0);
        if (navigator.get_mode() == CUBE_MODE &&
!artist.isQuadsEmpty() && navigator.get_current_selection() !=
NO_HITS && artist.selectionOkay(navigator.get_current_selection())
&& navigator.get_current_selection() < 100) {

      artist.drawRectPar(RectPar(artist.get_selected_quad(navigator
.get_current_selection())-
>get_a(),artist.get_selected_quad(navigator.get_current_selection()
)->get_c(), navigator.get_height()));
            }

            if (navigator.get_mode() == SQUARE_MODE) {
                  glEnable(GL_DEPTH_TEST);
                  glColor3f(0.5,0.5,0.5);
                  artist.drawSquaref(navigator.get_a(),
navigator.get_c());
            }

            artist.drawAllQuads(GL_RENDER);
            // Drawing buildings - first pass - dynamic texture
        artist.drawAllRectPars(GL_RENDER);

        if(live_var_project[0])
            glUseProgram(0);

        // Drawing buildings - second pass - static textures
        artist.drawAllRectPars(GL_RENDER, &projector,
&tex_navigator, &proj_prog);
```

```
            frametime = timer.get_secs();
            glutSwapBuffers();
}

/// Displayfunc for secondary texture window

void display_texwindow() {

    glutSetWindow(navigator.get_tex_window());
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(0.7, 0.7, 0.7);
    glEnable(GL_TEXTURE_2D);
    if(*tex_navigator.get_live_var() == 0) {
        glBindTexture(GL_TEXTURE_2D, tex_loader.get_texture(0));
    }
    else
        glBindTexture(GL_TEXTURE_2D,
*tex_navigator.get_live_var());
    glBegin(GL_QUADS);
        glTexCoord2f(0.0, 1.0); glVertex2f(0.0, 0.0);
        glTexCoord2f(1.0, 1.0); glVertex2f(tex_width, 0.0);
        glTexCoord2f(1.0, 0.0); glVertex2f(tex_width, tex_height);
        glTexCoord2f(0.0, 0.0); glVertex2f(0.0, tex_height);
    glEnd();
    glDisable(GL_TEXTURE_2D);
    glFlush();
    glutSetWindow(navigator.get_main_window());
}

/// Initializes various states, light parameters, sets up
projection matrix, loads all textures and all shader source
void initgl() {

            //Set preliminary states and define viewing frustum
        glClearColor (0.8, 0.8, 0.8, 0.);
        glMatrixMode (GL_PROJECTION);
        glLoadIdentity ();
        gluPerspective(60, 1, 0.01, 100);
        glMatrixMode (GL_MODELVIEW);
            glLoadIdentity ();
            glEnable(GL_DEPTH_TEST);
            glDepthFunc(GL_LESS);

            glewInit();
            // Initialize image library and texture coordinate
generation
            tex_loader.init_devIL();

      tex_loader.load_texture("Images/300EtellerandetIndgang.JPG");
        tex_loader.load_texture("Images/300NogetGavl.JPG");
        tex_loader.load_texture("Images/300NogetGavlTrappe.JPG");
        tex_loader.load_texture("Images/301Gavl.JPG");
        tex_loader.load_texture("Images/301Side1.JPG");
        tex_loader.load_texture("Images/301Side2.JPG");
        tex_loader.load_texture("Images/301Side3.JPG");
```

```
        tex_loader.load_texture("Images/302Bagside.JPG");
        tex_loader.load_texture("Images/302Front2.JPG");
        tex_loader.load_texture("Images/302Front.JPG");
        tex_loader.load_texture("Images/303Bagside1.JPG");
        tex_loader.load_texture("Images/303Bagside2.JPG");
        tex_loader.load_texture("Images/303Bagside3.JPG");
        tex_loader.load_texture("Images/303Forside1.JPG");
        tex_loader.load_texture("Images/303Forside2.JPG");
        tex_loader.load_texture("Images/303Forside3.JPG");
        tex_loader.load_texture("Images/303Forside4Indgang.JPG");
        tex_loader.load_texture("Images/303ForsideIndgang2.JPG");
        tex_loader.load_texture("Images/303Gavl1.JPG");
        tex_loader.load_texture("Images/303Gavl2.JPG");
        tex_loader.load_texture("Images/303Gavl3.JPG");
        tex_loader.load_texture("Images/303IndgangMatTorv.JPG");
        tex_loader.load_texture("Images/303Side1.JPG");
        tex_loader.load_texture("Images/303Side2.JPG");
        tex_loader.load_texture("Images/303Side3.JPG");
        tex_loader.load_texture("Images/304Side1.JPG");
        tex_loader.load_texture("Images/304Side.JPG");
        tex_loader.load_texture("Images/305Gavl1.JPG");
        tex_loader.load_texture("Images/305Gavl2.JPG");
        tex_loader.load_texture("Images/305Gavl3.JPG");
        tex_loader.load_texture("Images/305Gavl.JPG");
        tex_loader.load_texture("Images/305Niveau2Gavl.JPG");
        tex_loader.load_texture("Images/305Perspektiv1.JPG");
        tex_loader.load_texture("Images/305UNICBagindgang.JPG");
        tex_loader.load_texture("Images/305UNICIndgang.JPG");
        tex_loader.load_texture("Images/306Indgang.JPG");
        tex_loader.load_texture("Images/306Side1.JPG");
        tex_loader.load_texture("Images/306Side2.JPG");
        tex_loader.load_texture("Images/306Side3.JPG");
        tex_loader.load_texture("Images/307Side1.JPG");
        tex_loader.load_texture("Images/307Side2.JPG");
        tex_loader.load_texture("Images/307Side3.JPG");
        tex_loader.load_texture("Images/307Side4.JPG");
        tex_loader.load_texture("Images/308Gavl.JPG");
        tex_loader.load_texture("Images/308IndgangNord.JPG");
        tex_loader.load_texture("Images/308Niveau2Gavl.JPG");
        tex_loader.load_texture("Images/308Side1.JPG");
        tex_loader.load_texture("Images/308Side2.JPG");
        tex_loader.load_texture("Images/308Side3.JPG");
        tex_loader.load_texture("Images/308Side4.JPG");
        tex_loader.load_texture("Images/321Perspektiv1.JPG");
        tex_loader.load_texture("Images/321Perspektiv2.JPG");
        tex_loader.load_texture("Images/322Niveau2Gavl.JPG");
        tex_loader.load_texture("Images/322Perspektiv1.JPG");
        tex_loader.load_texture("Images/325PerspektivGavl.JPG");
        tex_loader.load_texture("Images/DTUHjørneStålbygning.JPG");
        tex_loader.load_texture("Images/DTUSideGeneric2.JPG");
        tex_loader.load_texture("Images/DTUSideGeneric3.JPG");
        tex_loader.load_texture("Images/DTUSideGeneric.JPG");

tex_loader.load_texture("Images/Niveau2GenericPerspektivGavl.JPG");
        tex_loader.load_texture("Images/StenmurFacade2.JPG");
        tex_loader.load_texture("Images/StenmurFacade.JPG");
        tex_loader.load_texture("Images/StenmurMatTorvEnde2.JPG");
```

```
        tex_loader.load_texture("Images/StenmurMatTorvEnde.JPG");
        tex_loader.load_texture("Images/3KvadrantFraOven.jpg");

        proj_prog = create_program("projTex.vert", "projTex.frag");
        navigator = Navigator(&projector, &tex_navigator,
&proj_prog);

        // Setup default lighting
        artist.light();

}

void initgl_tex_window() {
    glClearColor(0.8, 0.8, 0.8, 0.0);

    tex_loader.load_texture("Images/300EtellerandetIndgang.JPG");
    tex_loader.load_texture("Images/300NogetGavl.JPG");
    tex_loader.load_texture("Images/300NogetGavlTrappe.JPG");
    tex_loader.load_texture("Images/301Gavl.JPG");
    tex_loader.load_texture("Images/301Side1.JPG");
    tex_loader.load_texture("Images/301Side2.JPG");
    tex_loader.load_texture("Images/301Side3.JPG");
    tex_loader.load_texture("Images/302Bagside.JPG");
    tex_loader.load_texture("Images/302Front2.JPG");
    tex_loader.load_texture("Images/302Front.JPG");
    tex_loader.load_texture("Images/303Bagside1.JPG");
    tex_loader.load_texture("Images/303Bagside2.JPG");
    tex_loader.load_texture("Images/303Bagside3.JPG");
    tex_loader.load_texture("Images/303Forside1.JPG");
    tex_loader.load_texture("Images/303Forside2.JPG");
    tex_loader.load_texture("Images/303Forside3.JPG");
    tex_loader.load_texture("Images/303Forside4Indgang.JPG");
    tex_loader.load_texture("Images/303ForsideIndgang2.JPG");
    tex_loader.load_texture("Images/303Gavl1.JPG");
    tex_loader.load_texture("Images/303Gavl2.JPG");
    tex_loader.load_texture("Images/303Gavl3.JPG");
    tex_loader.load_texture("Images/303IndgangMatTorv.JPG");
    tex_loader.load_texture("Images/303Side1.JPG");
    tex_loader.load_texture("Images/303Side2.JPG");
    tex_loader.load_texture("Images/303Side3.JPG");
    tex_loader.load_texture("Images/304Side1.JPG");
    tex_loader.load_texture("Images/304Side.JPG");
    tex_loader.load_texture("Images/305Gavl1.JPG");
    tex_loader.load_texture("Images/305Gavl2.JPG");
    tex_loader.load_texture("Images/305Gavl3.JPG");
    tex_loader.load_texture("Images/305Gavl.JPG");
    tex_loader.load_texture("Images/305Niveau2Gavl.JPG");
    tex_loader.load_texture("Images/305Perspektiv1.JPG");
    tex_loader.load_texture("Images/305UNICBagindgang.JPG");
    tex_loader.load_texture("Images/305UNICIndgang.JPG");
    tex_loader.load_texture("Images/306Indgang.JPG");
    tex_loader.load_texture("Images/306Side1.JPG");
    tex_loader.load_texture("Images/306Side2.JPG");
    tex_loader.load_texture("Images/306Side3.JPG");
    tex_loader.load_texture("Images/307Side1.JPG");
    tex_loader.load_texture("Images/307Side2.JPG");
    tex_loader.load_texture("Images/307Side3.JPG");
```

```
    tex_loader.load_texture("Images/307Side4.JPG");
    tex_loader.load_texture("Images/308Gavl.JPG");
    tex_loader.load_texture("Images/308IndgangNord.JPG");
    tex_loader.load_texture("Images/308Niveau2Gavl.JPG");
    tex_loader.load_texture("Images/308Side1.JPG");
    tex_loader.load_texture("Images/308Side2.JPG");
    tex_loader.load_texture("Images/308Side3.JPG");
    tex_loader.load_texture("Images/308Side4.JPG");
    tex_loader.load_texture("Images/321Perspektiv1.JPG");
    tex_loader.load_texture("Images/321Perspektiv2.JPG");
    tex_loader.load_texture("Images/322Niveau2Gavl.JPG");
    tex_loader.load_texture("Images/322Perspektiv1.JPG");
    tex_loader.load_texture("Images/325PerspektivGavl.JPG");
    tex_loader.load_texture("Images/DTUHjørneStålbygning.JPG");
    tex_loader.load_texture("Images/DTUSideGeneric2.JPG");
    tex_loader.load_texture("Images/DTUSideGeneric3.JPG");
    tex_loader.load_texture("Images/DTUSideGeneric.JPG");

tex_loader.load_texture("Images/Niveau2GenericPerspektivGavl.JPG");
    tex_loader.load_texture("Images/StenmurFacade2.JPG");
    tex_loader.load_texture("Images/StenmurFacade.JPG");
    tex_loader.load_texture("Images/StenmurMatTorvEnde2.JPG");
    tex_loader.load_texture("Images/StenmurMatTorvEnde.JPG");
    tex_loader.load_texture("Images/3KvadrantFraOven.jpg");

}

/// Function to be called by glutReshapeFunc callback
void reshape (int w, int h) {
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
      gluPerspective(60, 1, 0.1, 50);
      glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
}

void reshape_texwindow(int w, int h) {

    glutSetWindow(navigator.get_tex_window());

    glBindTexture(GL_TEXTURE_2D, *tex_navigator.get_live_var());
    glGetTexLevelParameterfv(GL_TEXTURE_2D, 0, GL_TEXTURE_WIDTH,
&tex_width);
    glGetTexLevelParameterfv(GL_TEXTURE_2D, 0, GL_TEXTURE_HEIGHT,
&tex_height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if (tex_width < MIN_TEX_WINDOW_WIDTH) {
        gluOrtho2D(0.0, MIN_TEX_WINDOW_WIDTH, 0.0, tex_height+50);
        glViewport(0, 0, MIN_TEX_WINDOW_WIDTH, tex_height);
        glutReshapeWindow(MIN_TEX_WINDOW_WIDTH, tex_height);
    }
    else {
        gluOrtho2D(0.0, tex_width, 0.0, tex_height);
```

```
        glViewport(0, 0, tex_width, tex_height);
        glutReshapeWindow(tex_width, tex_height);
    }
    glMatrixMode(GL_MODELVIEW);
}

/// Callback function to be called by buttons created through glui
void control_cb(int input) {

    if(input == PROJECTOR_TRANSLATION_MODE) {
        projector.move_projector_pos(Vec3f(projector_pan->get_x(),
projector_vertical->get_z(), projector_pan->get_y()));
        glutSetWindow(navigator.get_main_window());
        glutPostRedisplay();
        // Resetting control's live variables
        projector_pan->set_x(0);
        projector_pan->set_y(0);
        projector_vertical->set_z(0);
    }
    else if(input == GEOMETRY_TRANSLATION_MODE &&
navigator.get_mode() == GL_SELECT) {

        Vec3f reverse_dir;

        reverse_dir[0] = -navigator.getDirection()[0];
        reverse_dir[1] = 0;
        reverse_dir[2] = -navigator.getDirection()[2];
        normalize(reverse_dir);

        Vec3f up = Vec3f(0.0, 1.0, 0.0);
        Vec3f left_vec = normalize(cross(-reverse_dir, up));

        Vec3f translation_vector;

        if(live_var_obj[1] > 0)
            translation_vector = -0.1*live_var_obj[1]*reverse_dir;
        else if(live_var_obj[1] < 0)
            translation_vector = -0.1*live_var_obj[1]*reverse_dir;
        else if(live_var_obj[0] > 0)
            translation_vector = 0.1*live_var_obj[0]*left_vec;
        else if(live_var_obj[0] < 0)
            translation_vector = 0.1*live_var_obj[0]*left_vec;

        int selection = navigator.get_current_selection();

        if (selection < 100) {
            artist.get_selected_quad(selection)-
>translate_quad(translation_vector);
            quad_building_move->set_x(0);
            quad_building_move->set_y(0);

        }
        else {
            artist.get_selected_building(selection-100)-
>translate_building(translation_vector);
            quad_building_move->set_x(0);
            quad_building_move->set_y(0);
```

```
        }
        glutSetWindow(navigator.get_main_window());
        glutPostRedisplay();

    }
    else if(input == TEX_CHOOSER) {
        reshape_texwindow(0,0);
        glutPostRedisplay();
        glutSetWindow(navigator.get_main_window());
        glutPostRedisplay();
    }
    else if(input == PROJECTION_CHECKBOX) {
        glutSetWindow(navigator.get_main_window());
        glutPostRedisplay();
    }
    else if(input == PROJECTOR_DIRECTION_MODE) {
        projector.move_projector_dir(Vec3f(live_var_proj_dir[0],
0.0, live_var_proj_dir[1]));
        glutSetWindow(navigator.get_main_window());
        glutPostRedisplay();

        // Resetting live variables
        projector_direction->set_x(0.0);
        projector_direction->set_y(0.0);
    }
    else if(input == CHANGE_CAM_ROT_DIST_MODE){

navigator.set_cam_dist_to_rot_center(cam_spinner_live_var[0]);
    }
    else if(input == CHANGE_PROJ_ROT_DIST_MODE) {

navigator.set_proj_dist_to_rot_center(proj_spinner_live_var[0]);
    }
    else if(input == SAVE_SCENE) {
        saveTable(*artist.get_buildings());
    }
    else if(input == LOAD_SCENE) {
        glutSetWindow(navigator.get_main_window());
        readFile(artist.get_buildings());
        glutPostRedisplay();
    }
    else {
            navigator.set_mode(input);
    }
}

int main (int argc, char ** argv) {

      // Initialize standard OpenGL and GLUT routines
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
    glutInitWindowSize (WINDOW_X, WINDOW_Y);
      glutInitWindowPosition(20,20);

      // Create window and save window ID
      navigator.set_main_window(glutCreateWindow("ObjectTextures"))
;
```

```
      initgl();

      // Create GUI top button bar object and add widgets
      glui_top =
GLUI_Master.create_glui_subwindow(navigator.get_main_window(),
GLUI_SUBWINDOW_TOP);
      glui_top->add_button("Quad", SQUARE_MODE, control_cb);
      glui_top->add_button("Extrude", CUBE_MODE, control_cb);
    glui_top->add_column(false);
      glui_top->add_button("Select", GL_SELECT, control_cb);
    glui_top->add_button("Texturize", TEXTURIZE_WORLD, control_cb);
    glui_top->add_column(false);
      glui_top->add_button("Freefly", FREE_FLY, control_cb);
    glui_top->add_button("Autoposition",
AUTOPOSITION_PROJECTOR_MODE, control_cb);
      glui_top->add_column(false);
      glui_top->add_button("Rotate World", ROTATE_CAM, control_cb);
    glui_top->add_button("Rotate Projector", ROTATE_PROJECTOR,
control_cb);
    glui_top->add_column(false);
    glui_top->add_spinner("Camera's distance to center of
rotation:", GLUI_SPINNER_FLOAT, cam_spinner_live_var,
CHANGE_CAM_ROT_DIST_MODE, control_cb);
    glui_top->add_spinner("Projector's distance to center of
rotation:", GLUI_SPINNER_FLOAT, proj_spinner_live_var,
CHANGE_PROJ_ROT_DIST_MODE, control_cb);

    // Create GUI bottom translation control and add widgets
    glui_status =
GLUI_Master.create_glui_subwindow(navigator.get_main_window(),
GLUI_SUBWINDOW_BOTTOM);

    glui_bottom =
GLUI_Master.create_glui_subwindow(navigator.get_main_window(),
GLUI_SUBWINDOW_BOTTOM);
    quad_building_move = new GLUI_Translation(glui_bottom, "Move
Quad/Building (XZ)", GLUI_TRANSLATION_XY, live_var_obj,
GEOMETRY_TRANSLATION_MODE, control_cb);
    glui_bottom->add_column(true);
    projector_pan = new GLUI_Translation(glui_bottom, "Pan
projector, (XZ)", GLUI_TRANSLATION_XY, live_var_proj_xz,
PROJECTOR_TRANSLATION_MODE, control_cb);
    glui_bottom->add_column(false);
    projector_vertical = new GLUI_Translation(glui_bottom,
"Raise/Lower Projector (Y)", GLUI_TRANSLATION_Z, live_var_proj_y,
PROJECTOR_TRANSLATION_MODE, control_cb);
    glui_bottom->add_column(false);
    projector_direction = new GLUI_Translation(glui_bottom, "Move
projector direction", GLUI_TRANSLATION_XY, live_var_proj_dir,
PROJECTOR_DIRECTION_MODE, control_cb);
    glui_bottom->add_column(true);
    glui_bottom->add_button("Save scene", SAVE_SCENE, control_cb);
    glui_bottom->add_button("Load scene", LOAD_SCENE, control_cb);

    // Callbacks
```

```
      // Setting the usual glut callbacks
      glutDisplayFunc(display);
      glutMotionFunc(mouse_motion);

      // Set glut callbacks that have been replaced by the GUI
library
      GLUI_Master.set_glutMouseFunc(mouse);
      GLUI_Master.set_glutIdleFunc(animate);
      GLUI_Master.set_glutSpecialFunc(spec_keyfun);
      GLUI_Master.set_glutKeyboardFunc(keyboard);
      GLUI_Master.set_glutReshapeFunc(reshape);

    ///// SETTING UP SECONDARY WINDOW FOR DISPLAYING TEXTURES HERE
/////
      glutInitDisplayMode(GLUT_RGBA | GLUT_SINGLE);
    glutInitWindowSize (320, 200);
      glutInitWindowPosition(200,20);

    navigator.set_tex_window(glutCreateWindow("Texture Window"));

    initgl_tex_window();

    glui_tex_top =
GLUI_Master.create_glui_subwindow(navigator.get_tex_window(),
GLUI_SUBWINDOW_TOP);
    tex_listbox = new GLUI_Listbox(glui_tex_top, "Choose texture:",
tex_navigator.get_live_var(), TEX_CHOOSER, control_cb);

    tex_listbox->add_item(tex_loader.get_texture(0),
"300EtellerandetIndgang.JPG");
    tex_listbox->add_item(tex_loader.get_texture(1),
"300NogetGavl.JPG");
    tex_listbox->add_item(tex_loader.get_texture(2),
"300NogetGavlTrappe.JPG");
    tex_listbox->add_item(tex_loader.get_texture(3),
"301Gavl.JPG");
    tex_listbox->add_item(tex_loader.get_texture(4),
"301Side1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(5),
"301Side2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(6),
"301Side3.JPG");
    tex_listbox->add_item(tex_loader.get_texture(7),
"302Bagside.JPG");
    tex_listbox->add_item(tex_loader.get_texture(8),
"302Front2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(9),
"302Front.JPG");
    tex_listbox->add_item(tex_loader.get_texture(10),
"303Bagside1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(11),
"303Bagside2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(12),
"303Bagside3.JPG");
    tex_listbox->add_item(tex_loader.get_texture(13),
"303Forside1.JPG");
```

```
    tex_listbox->add_item(tex_loader.get_texture(14),
"303Forside2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(15),
"303Forside3.JPG");
    tex_listbox->add_item(tex_loader.get_texture(16),
"303Forside4Indgang.JPG");
    tex_listbox->add_item(tex_loader.get_texture(17),
"303ForsideIndgang2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(18),
"303Gavl1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(19),
"303Gavl2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(20),
"303Gavl3.JPG");
    tex_listbox->add_item(tex_loader.get_texture(21),
"303IndgangMatTorv.JPG");
    tex_listbox->add_item(tex_loader.get_texture(22),
"303Side1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(23),
"303Side2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(24),
"303Side3.JPG");
    tex_listbox->add_item(tex_loader.get_texture(25),
"304Side1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(26),
"304Side.JPG");
    tex_listbox->add_item(tex_loader.get_texture(27),
"305Gavl1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(28),
"305Gavl2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(29),
"305Gavl3.JPG");
    tex_listbox->add_item(tex_loader.get_texture(30),
"305Gavl.JPG");
    tex_listbox->add_item(tex_loader.get_texture(31),
"305Niveau2Gavl.JPG");
    tex_listbox->add_item(tex_loader.get_texture(32),
"305Perspektiv1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(33),
"305UNICBagindgang.JPG");
    tex_listbox->add_item(tex_loader.get_texture(34),
"305UNICIndgang.JPG");
    tex_listbox->add_item(tex_loader.get_texture(35),
"306Indgang.JPG");
    tex_listbox->add_item(tex_loader.get_texture(36),
"306Side1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(37),
"306Side2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(38),
"306Side3.JPG");
    tex_listbox->add_item(tex_loader.get_texture(39),
"307Side1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(40),
"307Side2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(41),
"307Side3.JPG");
```

```
    tex_listbox->add_item(tex_loader.get_texture(42),
"307Side4.JPG");
    tex_listbox->add_item(tex_loader.get_texture(43),
"308Gavl.JPG");
    tex_listbox->add_item(tex_loader.get_texture(44),
"308IndgangNord.JPG");
    tex_listbox->add_item(tex_loader.get_texture(45),
"308Niveau2Gavl.JPG");
    tex_listbox->add_item(tex_loader.get_texture(46),
"308Side1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(47),
"308Side2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(48),
"308Side3.JPG");
    tex_listbox->add_item(tex_loader.get_texture(49),
"308Side4.JPG");
    tex_listbox->add_item(tex_loader.get_texture(50),
"321Perspektiv1.JPG");
    tex_listbox->add_item(tex_loader.get_texture(51),
"321Perspektiv2.JPG");
    tex_listbox->add_item(tex_loader.get_texture(52),
"322Niveau2Gavl.JPG");
    tex_listbox->add_item(tex_loader.get_texture(53),
"322Perspektiv1");
    tex_listbox->add_item(tex_loader.get_texture(54),
"325PerspektivGavl");
    tex_listbox->add_item(tex_loader.get_texture(55),
"DTUHjørneStålbygning");
    tex_listbox->add_item(tex_loader.get_texture(56),
"DTUSideGeneric2");
    tex_listbox->add_item(tex_loader.get_texture(57),
"DTUSideGeneric3");
    tex_listbox->add_item(tex_loader.get_texture(58),
"DTUSideGeneric");
    tex_listbox->add_item(tex_loader.get_texture(59),
"Niveau2GenericPerspektivGavl");
    tex_listbox->add_item(tex_loader.get_texture(60),
"StenmurFacade2");
    tex_listbox->add_item(tex_loader.get_texture(61),
"StenmurFacade");
    tex_listbox->add_item(tex_loader.get_texture(62),
"StenmurMatTorvEnde2");
    tex_listbox->add_item(tex_loader.get_texture(63),
"StenmurMatTorvEnde");
    tex_listbox->add_item(tex_loader.get_texture(64),
"3KvadrantFraOven.jpg");

    glui_tex_top->add_column(false);
    glui_tex_top->add_checkbox("Project texture?",
live_var_project, PROJECTION_CHECKBOX, control_cb);

    GLUI_Master.set_glutDisplayFunc(display_texwindow);
    GLUI_Master.set_glutReshapeFunc(reshape_texwindow);

      glutMainLoop();
      return 0;
}
```

C.21 projTex.vert

```
varying vec3 normal, eye_vec, point2light;

void main(void)
{
	normal = gl_NormalMatrix * gl_Normal;
	vec4 eye_position = gl_ModelViewMatrix * gl_Vertex;

	gl_Position = ftransform();

	gl_TexCoord[0] = gl_TextureMatrix[0] * gl_Vertex;

	point2light = gl_LightSource[0].position.xyz - 
eye_position.xyz;
	eye_vec = -eye_position.xyz;
}
```

C.22 projTex.frag

```
uniform sampler2D tex;
varying vec3 normal, eye_vec, point2light;

void main(void)
{
	vec4 final_color;

	vec3 N = normalize(normal);
	vec3 L = normalize(gl_LightSource[0].position.xyz + eye_vec);
	vec3 E = normalize(eye_vec);
	vec3 R = normalize(reflect(-L,N));

	// Ambient term:
	vec4 Iamb = gl_FrontLightProduct[0].ambient;

	// Diffuse term:
	vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 
0.0);
	Idiff = clamp(Idiff, 0.0, 1.0);

	// Specular term
	vec4 Ispec = gl_FrontLightProduct[0].specular * 
pow(max(dot(R,E),0.0),0.3*gl_FrontMaterial.shininess);
    Ispec = clamp(Ispec, 0.0, 1.0);

	final_color = gl_FrontLightModelProduct.sceneColor + Iamb + 
Idiff + Ispec;

	// Suppress the reverse projection.
	if( gl_TexCoord[0].q>0.0 )
	{
		vec4 ProjTexColor = texture2DProj(tex, gl_TexCoord[0]);
```

```
                final_color = ProjTexColor;
        }


        gl_FragColor = final_color;
}
```