DATAGRAFIKK 2019

**Mette Strand Hornnes**

Table of Contents

[1. Intro 3](#_Toc26996039)

[1.1 Model, view, projection 3](#_Toc26996040)

[1.2 Shadere 3](#_Toc26996041)

[1.3 Hjelpeklasser 4](#_Toc26996042)

[2 Implementasjoner 5](#_Toc26996043)

[2.1 First Person Camera 5](#_Toc26996044)

[2.2 Skybox 6](#_Toc26996045)

[2.3 Kube med Parallax Mapping 7](#_Toc26996046)

[2.4 Pyramide med Parallax mapping 9](#_Toc26996047)

[2.5 Hexogonal prisme 9](#_Toc26996048)

[2.6 Felles for objekter 10](#_Toc26996049)

[3. Hjelp og inspirasjon til implementasjon 11](#_Toc26996050)

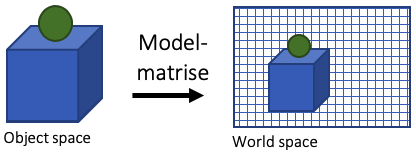
[4. Litteraturliste 12](#_Toc26996051)

# 1. Intro

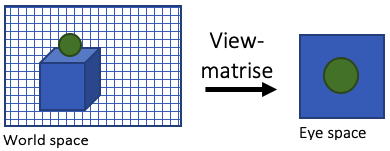
For å enklere kunne forklare implementasjonene ønsker jeg først å forklare generelt om de tre matrisene model, view og projection, samt om shadere. Sist vil jeg beskrive de to hjelpeklassene jeg benytter. Alle illustrasjoner i dokumentet er egenlaget.

## 1.1 Model, view, projection

MODEL - «MODEL TO WORLD»  
Denne matrisen plasserer objektet verdenen. Med denne matrisen kan man bestemme hvor objektet skal plasseres, samt bestemme rotasjon og skalering.

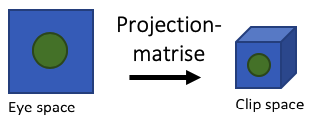


VIEW - «WORLD TO VIEW»   
Denne matrisen bestemmer hvordan verden (med objetet) skal plasseres i forhold til kameraet. Kameraet, som er våre øyne, vil alltid se nedover den negative z-aksen. For at objektene skal syntes, må derfor verden bevege seg.



PROJECTION - VIEW TO PROJECTION»

Projectionmatrisen gjør om objektene som nå er i 3D til 2D, slik at vår hardware kan vise de korrekt. Det er to typer projections. *Perpective* som tar med dybde i betraktningen slik at objekter som er lenger unna vil se mindre ut, og *Ortohraphics* som ikke gjør det. Med orthographics vil objekter som er lenger unna se like ut og man vil derfor ikke få denne 3D-effekten man får med perspective <https://www.youtube.com/watch?v=-tonZsbHty8>.



## 1.2 Shadere

For de ulike objektene har jeg to shadere. En vertex-shader og en fragment-shader. Oppgaven til en vertex-shader er å spesifisere punktene som bygger objektet. Etter input fra andre deler av koden settes disse punktene på korrekte steder i verden. Den vil også sende ulike paramtre videre til fragment-shader for videre håndtering.

Fragment-shaderen håndterer alle delene mellom disse punktene slik at det bygges flater. Det meste av texture og lys håndtering derfor her.

## 1.3 Hjelpeklasser

For å enklere kunne implementere bruk av ulike shadere og lasting av texture er det laget egne klasser for dette, i henholdsvis shader.h og texture.h.

SHADER

Shader.h består av en Shader-klasse, med tilhørende metoder for enklere initialisering og aktivering av shadere. Ved opprettelse av en instans av Shader sendes det med filsti til ønsket vertex- og fragment shader. For begge shaderne vil det hentes inn kildekoden som leses og legges i hver sin variabel. Deretter kompileres disse shaderne og knyttes til et Program til den instansierte Shaderen. Når man ønsker å benytte denne shaderen trenger man nå kun å kalle på funksjonen *Use()* som ligger i klassen*.* Denne sender programmet til shaderen med som parameter til glUseProgram-metoden og sørger derfor for at denne blir brukt ved innhenting av Uniform locations, samt andre utførelser opp mot satte shadere.

TEXTURE

Texture.h holder på en TextureLoading-klasse som skal gjøre det enklere og mer oversiktlig å hente inn bilder som brukes på elementene. I denne klassen ligger det to statiske funksjoner *LoadTexture* og *LoadSkyBox*.   
*LoadTexture* vil få tilsendt en filsti til ønsket bilde som skal benyttes som skal lastes inn. Et bibliotek kalt SOIL hjelper til med dette. Deretter generes en id til denne texturen som skal benyttes, spesifiser ulike attributter og parametre for det slik at det fremstår som ønsket og binder bildet til texturens plassering. Deretter returneres denne id-en for videre bruk.  
*LoadSkyBox* vil på de fleste måter fungere likt som *LoadTexture,* men da denne skal benyttes til SkyBox mottar denne 6 bilder som alle må lastes inn og bindes. Her vil det istedenfor bruk av GL\_TEXTURE\_2D ved binding som over, benyttes GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X hvor x økes med en gang for hver.

Instansiering av de ulike shaderne, samt innlasting av de ulike texturene gjøres i *initGL()*. Både shaderne og texture-idene er tilgjengelige for bruk alle steder i koden da variablene for disse er initialisert globalt.

# 2 Implementasjoner

## 2.1 First Person Camera

For å kunne bevege seg rundt i verden er det implementert såkalt First Person Camera. Ved å flytte alle objekter i motsatt retning av det man trykker vil det skapes en illusjon av at man beveger seg rundt (<https://learnopengl.com/Getting-started/Camera>).

For bedre oversikt og enklere bruk er ulike Camera-funksjoner lagt i en egen header-fil kalt camera.h. Denne inneholder en Camera-klasse, med tilhørende konstruktør og funksjoner for bruk av instansiert Camera.

I main.cpp vil det opprettes en instanse av Camera. Her settes startposisjonen til 0.0f, 0.0f, 5.0f. Altså skal den settes midt i verdenen, men noe mot deg (langs den positive z-aksen) slik at vi kan se objektene. Ved instansieringen vil det også attributtene for retning, vinkel, hastighet, sensitivitet og zoom settes. Disse er forhåndsdefinerte i klassen.

PRODUSERER VIEW-MATRISE  
Camera-klassen inneholder funksjonen *GetViewMatrix.* Denne vil det ved hjelp av glm∷lookAt bygge en view-matrise basert på attributtene til Camera. Denne brukes for å sette view-matrise til alle objektene.

PROSESSERER MUSEBEVEGELSER  
I main.cpp er det satt opp funksjonen *mouseCallback* som kalles hver gang musen beveger på seg. Denne igjen kalle funksjonen *processMouseMovement* i Camera-klassen og sende med x- g y-posisjonene til mus. Med disse float-verdiene vil attributtene til Camera oppdateres og på den måten vil også view-matrisen man får i *GetViewMatrix*-funksjonen endres.

PROSESSERER TASTATURTRYKK

Main.cpp inneholder også funksjonen *doMovement* som kalles i while-løkken i main-metoden. Altså vil denne funksjonen kalles «konstant» så lenge programmet kjører. Hver gang den kalles vil den sjekke om piltaster eller A,S,W,D,Z,X er trykket ned på tastaturet. Om dette er tilfelle vil korrekt retning ut i fra disse sendes til *processKeyBoard-*funksjoneni Camera-klassen. Deltatime, som hindrer hakkete bevegelser ved å bestemme hvor lenge hvert frame (bilde) skal vises på skjermen, sendes også som parameter. I *processKeyBoard* vil posisjonen til Camera endres og dermed vil også view-matrisen fra *GetViewMatrix*-funksjonen endres. Denn vil også endre zoom-verdier da jeg ønsket det også skal føles som man beveger seg rundt inne i skyboxen.

BRUK:

Piltaster eller A,W,S,D = Venstre, frem, bak, høyre

Z = Ned

X = Opp

Musebevegelser = Endrer vinkel

## 2.2 Skybox

Skybox er en enkel måte å få et inntrykk av et landskap eller verden rundt kameraet. Skyboxen vil bestå av 6 bilder som hvert for seg representerer en side inne i en kube. Disse 6 2D-bildene vil brettes rundt kameraet og på den måten vil det oppleves som en verden man kan se seg rundt i. <https://learnopengl.com/Advanced-OpenGL/Cubemaps>

GENERERER PUNKTER OG SETTER DE TIL VALGT BUFFER – *generateSkyBoxVerticesAndSetArraysAndBuffers()*For opprettelse av denne kuben som skal utgjøre verdenen setter jeg først de ulike punktene som til sammen utgjør disse*.* I utgangspunktet var alle punktene til kuben til alle sidene definert manuelt, men etter hvert som jeg fikk mer forståelse og så at flere av punktene ble gjenbrukt for ulike sider valgte jeg å opprette en løkke som produserer kuben. For hver runde i løkken plukkes det ut de punktene som trengs for å bygge nåværende side og legger denne til i vektoreren *skyBoxVertices* som skal holde på alle sidene.

Nå løkken er ferdigkjørt (totalt 6 ganger) oppretter jeg en vertex-array og buffer for skyBoxen. Deretter fyller jeg bufferen med dataen produsert i løkkene, altså punktene for kuben. Til slutt spesifiseres attributtene. I dette tilfelle har jeg kun posisjons-koordinater.

TEGNER TRIANGLENE – *drawSkybox()*

I funksjonen *drawSkybox* spesifiseres det først hvilkenshader som skal benyttes. Deretter bindes ønsket texture, som er lastet inn i *initGL-*funksjonen, og sender dette til tilhørende fragment-shader. Videre skal ulike matriser for skybox settes. I likhet med de andre objektene vil de ulike uniformlokasjonene som gjør det mulig å sende matrisene til shadere hentes inn i *initGL*. Variabelene som holder på disse lokasjonene initaliseres globalt, slik at de kan nås fra alle steder i koden.

View-matrisen settes som alle de andre etter det camera-klassen produserer via *GetViewMatrix()-*funksjonen. Da skybox skal wrappes rundt og ikke sees som er kube foran kamera slik som de andre objektene må eventuelle «translation»-komponenter. Dette gjøres ved å konverteres matrisen til en 3x3-matrise, of deretter tilbake igjen <https://learnopengl.com/Advanced-OpenGL/Cubemaps>.

For Skybox er det ikke spesifisert noe model-matrise da denne ikke skal plasseres et bestemt sted i verden, men skal «wrappes» rundt oss.

For Skybox settes også projection-matrisen i denne funksjonen. Denne henter inn verdi fra getZoom() fra Camera-klassen som blir oppdatert ved bruk av piltaster. På den måten vil man få illusjonen av at man kommer nærmere en side, selv om man i utgangspunktet kun zoomer inn.

Etter at valgt vertex-array aktiveres, tegnes trekantene som sammen skal utgjøre skybox-kuben med *glDrawArrays*. Denne er satt til 36 da det skal tegnes 12 trekanter som alle har 3 punkter.

SETTER PROJECTION-MATRISE – *resizeGL()*

Projection-matrisen blir, i likhet med for alle de andre objektene, satt i *resizeGL-*funksjonen. Denne metoden kalles hver gang størrelsen på vinduet endres, da jeg har satt at dette skal være mulig. Jeg benytter samme view-matrise for alle objektene, inkludert skybox.

SHADERE – *skybox.vert & skybox.frag*

Vertex-shader får inn projection- og view-matrise satt i main.cpp. Sammen med vektoren med posisjonene opprettet i *generateSkyBoxVerticesAndSetArraysAndBuffers* plasseres punktene i verden. Posisjonene vil så bli sendt til fragment-shaderen hvor de sammen med texturen setter fargen til de ulike pikslene (fragment) mellom punktene (vertex).

## 2.3 Kube med Parallax Mapping

PARALLAX MAPPING  
Med Parallex Maping vil vi kunne skape en illuasjon av at flatene på objektet har en dybde. I tillegg til en vanlig texture, en såkalt diffuse texture, vil vi legge på to andre.

En normal-texture som er av en blåaktig farge og vil få frem små ujevnheter, og displacement-texture som er svart og hvit som vil «løfte» deler av texturen opp <https://www.pluralsight.com/blog/film-games/bump-normal-and-displacement-maps>

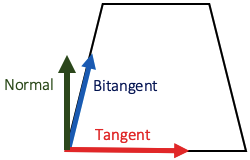
. Sammen vil de bestemme hvor mye lys ulike fragment skal ha og hvilken vinkel det skal reflekteres tilbake. Dette vil skape en illusjon av at noe er nærmere eller lenger unna, og vi får med det en dybde i teksturen.   
<https://learnopengl.com/Advanced-Lighting/Normal-Mapping>

GENERERER PUNKTER OG SETTER DE TIL VALGT BUFFER - *generateCubeVerticesAndSetArraysAndBuffers()*

På samme måte som ved skybox bygger jeg denne kuben ved hjelp av en løkke, men da jeg skal implementere parallex mapping trenger denne en del mer. I tillegg til posisjonen til de ulike punktene behøves det også UV-, normal-, tangent- og bitangent-koordinater

UV-verdier   
Disse koordinatene forteller hvordan texturen skal plasseres. Disse er definert manuelt og er lik for alle sider,

Normal-verdier  
Disse koordinater forteller hvordan normal-texturen skal plasseres og hvordan lyset skal treffe. Disse var i utgangspunktet satt direkte på samme måte som posisjonene. Etter tips fra foreleser og veileder Lars Vidar Magnusson, valgte jeg å endre koden slik at disse kalkuleres for hver side. Ved å ta kryssproduktet av sider av trekantene som bygger en side, og deretter legge sammen disse to normaliserte resultatene, vil jeg få normal-verdier for ønsket side.

Tangent- og Bitangent-verdier  
Disse koordinatene gjør at vi kan sette lys i et annet koordinat-space, slik at vi kan benytte samme normal-texture uavhengig av plasseringen. Tangent blir, for hver side kalkulert, ut i fra posisjons- og uv-koordinatene <https://learnopengl.com/Advanced-Lighting/Normal-Mapping>. Opprinnelig ble også Bitangen-verdiene beregnet her, men da denne verdien kan finnes ved å ta kryssproduktet av normal- og tangent-verdiene er dette noe jeg heller har valgt å beregne i vertex-shaderen.

Når koordinatene for en side er ferdig utregnet kan det, på samme måte som ved med skybox, settes sammen til en side. Disse vil deretter legges til i vektoren *cubeVertices* som etter at løkken har kjørt 6 ganger vil holde på alle koordintene til kuben. Disse kan nå settes til ønsket buffer og attributtene for de ulike koordinatene kan spesifiseres, slik at disse enkelt kan hentes ut i shaderen.

TEGNER TRIANGLENE - *drawCube()*

En annen shader skal nå brukes. Denne funksjonen starter derfor med å definere dette. Deretter hentes alle tre texturene som trengs for å implementere parralax mapping inn og bindes slik at disse nå kan brukes i den tilhørende fragment-shaderen.

View-matrise til kuben blir satt ved hjelp av Camera sin *getViewMatrix* forklart tidligere. Denne hentes inn for hver runde i while-løkken, slik at man får en illusjon av at man beveveger seg rundt objektet.

Til model-matrisen benytter jeg nå glm∷translate. Da alle objektene er plassert med ut i fra verden sin origo må jeg flytte kuben slik at den ikke ligge på samme sted som de andre objektene. Ved å benytte translate, flytter jeg kubens origo 4 punkter til venstre.

SETTER LYS-MATRISER – setLightMatricesForCubeAndPyramide()

Da denne kuben skal ha to lys mot seg sendes også verdier for lysenes posisjon og farge, samt en view-matrise som brukes for begge lyskildene til shader. Lys en har en hvit farge og vil sirkulere, men lys to har en rødlig farge og står stille.   
Da disse lysene skal være helt like for både kube og pyramide er koden for dette ekstrahert ut i en egen metode, slik at denne kan gjenbrukes for begge de to.

Etter at valgt vertex-array aktiveres, tegnes trekantene som sammen skal utgjøre kuben med *glDrawArrays*. Denne er satt, likt som skybox, til 36 da det skal tegnes 12 trekanter som alle har 3 punkter.

SHADERE – *cubeAndPyramid.vert & cubeAndPyramid.frag*

Vertex-shader model-, view, og projection -matriser, koordinater fra *cubeVertices*. Ut i fra disse vil kuben plasseres i verdenen. Sammen med posisjon og view til lysene vil det også beregnes tangent-verdier som sendes videre til fragment-shaderen.

I fragment-shaderen vil man sammen med disse verdiene få inn de tre ulike texturene man trenger for å få til parallax mapping. Da beregning av lys må utføres to ganger, da jeg har to ulike lys, har jeg valgt å legge denne koden i en egen metode; *getFragColor.* Denne tar i mot lysetsposisjon, farge, styrke på ambient- og specular-lys, samt styrke på hvor mye specular-lyset skal skinne. På den måten kan dette settes ulikt for lysene. Jeg får da returnert en fragment-farge for begge lysene. Ved å finne avstanden fra fragmentet (punktet på objektet) og lysene kalkulerer jeg dempingen av lysene. Disse kan deretter legges sammen, slik at jeg får et endelig fargeresultat for fragmentet.

## 2.4 Pyramide med Parallax mapping

Denne pyramiden gjøres på mange måter likt som som kuben. Da de begge skal ha parallax mapping implemenetert benyttes shadere benyttes.

GENERERER PUNKTER OG SETTER DE TIL VALGT BUFFER - *generatePyramidVerticesAndSetArraysAndBuffers ()*

Jeg har her valgt å produsere en trekant for hver runde av løkken, som nå sammen bygger en pyramide. Utregningene for de ulike koordinatene for en side, vil ellers gjøres likt. En liten endring jeg endte opp å gjøre var å justere den ene UV-vektoren for de to siste trekantene som produseres. Årsaken til dette er at disse to skal til sammen utgjøre bunnen av pyramiden og ønsket derfor texturen til å plasseres noe annerledes.

TEGNER TRIANGLENE - *drawPyramid()*

Ønsket shader aktiveres og settes og sendes til shaderen. View-matrise til kuben blir satt ved hjelp av Camera sin *getViewMatrix*-funksjon, forklart tidligere. Da jeg ønsker at pyramiden skal være noe større en de andre objektene endrer jeg derimot denne matrisen med glm∷scale og setter den til å være dobbelt så stor.

Jeg gjør også noen endringer for pyramidens model-matrise. Pyramiden skal stå i midten og jeg benytter meg derfor ikke av glm∷translate for denne når model-matrise, slik jeg gjorde med kuben. Jeg benytter i midlertid glm∷rotate, da jeg ønsker at den skal rotere y-aksen i gitt hastighet. For lysmatriser kaller jeg så på *setLightMatricesForCubeAndPyramide-*funksjonen forklart tidligere.

## 2.5 Hexogonal prisme

Denne prismen er laget både for å få utfordret meg med andre geometriske former, samt å få frem effekten av parallax mapping ekstra godt. Pyramiden og denne prismen har eksakt samme bilde bint til seg, men det sistnevnte har altså ingen spesiell lyssetting.

GENERERER PUNKTER OG SETTER DE TIL VALGT BUFFER - *generatehexPrismVerticesAndSetArraysAndBuffers()*

Denne formen krever noe fler punkter enn for de tidligere formene. Løkken for å få produsert prismen må derfor kjøre 14 ganger. Da den ikke skal ha parallax mapping består *hexPrismVertices* som bygger den ferdige prismen kun posisjons- normal- og uv-koordinater. Da jeg ønsket en inntrykk av at det er ett bilde som dekker bunn og topp har jeg i midlertid jusert på uv-verdiene etter hvilke sider som bygges. Jeg har valgt en switch for denne sjekken

TEGNER TRIANGLENE - *drawHexPrism ()*

Denne modellen benytter egne shadere. Etter at denne er aktivert, bindes og sendes en texture til den tilhørende fragment-shaderen.

I likhet med de andte objektene settes view-matrise ved hjelp av Camera sin funksjon, forklart tidligere. For model-matrise benytter jeg i likhet med kuben glm∷translate da jeg ønsker at prismen skal stå til høyre for pyramiden. Med translate flytter jeg derfor denne 4 punkter i positiv x-retning. Deretter settes matriser for lysene, som er like som de satt for tidligere objekter, men da denne benytter andre shadere, kan ikke *setLightMatricesForCubeAndPyramide*-metoden brukt tidligere benyttes for denne.

SHADERE – *hexPrism.vert & hexPrism.frag*

I vertex-shaderen plasseres nå, i likhet med tidligere, objektet ut i fra model, view, projection-matrise og koordinater fra *hexPrismVertices.* Texture, fragment-posisjon og normal-koordinater sendes videre til fragmentshaderen.   
I likhet med tidligere ønsker jeg å beregne fragment-farge utifra to ulike lyskilder. Dette gjøres derfor med samme metode for begge lysene, likt som i fragment-shaderen for kube og pyramide.

## 2.6 Felles for objekter

SETTER PROJECTION-MATRISE – *resizeGL()*

Projection-matrisen for kube, triangel og prisme blir satt i *resizeGL-*funksjonen. Denne metoden kalles hver gang størrelsen på vinduet endres, da jeg har satt at dette skal være mulig. Jeg benytter samme view-matrise for alle objektene, utenom skybox. For å få et inntrykk av at objekter lenger unna er mindre, bruker jeg glm∷perspective. På den måte får jeg den 3D-effekten jeg ønsker på objektene.

HENTER UNIFORM LOCATIONS – *initGL()*

Innhenting av lokasjonen til alle Uniform-variablene skjer i initGL-funksjonen. Variablene de settes til er initialeres globalt slik at de kan nås alle steder i koden senere.

# 3. Hjelp og inspirasjon til implementasjon

Dette prosjektet består av en god blanding av egen og inspirasjon fra andres kode. Jeg har brukt diverse nettsteder for hjelp, men har samtidig strukturert koden min slik jeg føler gjør den mest mulig oversiktlig og lettlest. Jeg har også forsøkt å gjenbruke lik kode der det har latt seg gjøre, selv om jeg ser noe forbedringspotensial når det gjelder dette. Under kommer en liste med nettsteder jeg har fått hjelp og inspirasjon fra.

GRUNNLEGGENDE OPPSETT

Selve oppsettet med diverse feilhåndtering og initialisering er gjort med hjelp av simple lighting-eksempel gitt av foreleser.

<https://github.com/lavima/itf21219_examples/tree/master/glfw/simple_lighting>

SKYBOX

Implementasjonen av Skybox er gjort med hjelp av Sonar Systems sitt videoklipp: <https://www.youtube.com/watch?v=_EjsL3B9CRY>. Denne baserer seg på learnopengl.com sin guide. Kode for å generere punktene er modifisert og implementert selv, da jeg ønsket å produsere skybox-kuben med hjelp av løkke.

LYS

Til starthjelp for implementasjon av lys er det brukt både Simple Lighting-eksempel gitt av foreleser, samt learnopengl.com og chai3d.org sine guider.

<https://github.com/lavima/itf21219_examples/tree/master/glfw/simple_lighting>  
 <https://learnopengl.com/Lighting/Lighting-maps>

<http://www.chai3d.org/download/doc/html/chapter16-lighting.html>

PARALLAX MAPPING

Parallax-mapping er implementert med hjelp av learnopengl.com sine guider om normal- og parallax mapping. Da disse veiledningene kun viser oppsett av en side er koden for å utvide dette til en kube produsert selv. Det er også gjort noen endringer ved utregning av normalverdier beskrevet over, samt flyttet utregning av bitangent til shaderen.

<https://learnopengl.com/Advanced-Lighting/Normal-Mapping>

<https://learnopengl.com/Advanced-Lighting/Parallax-Mapping>

FIRST PERSON CAMER

Camera-funksjonen er implementert med hjelp av Sonar Systems sin tutorial videoklipp. <https://www.youtube.com/watch?v=ns9eVfHCYdg>. Denne baserer seg på learnopengl.com sin guide. Mulighet for å også kunne bevege seg opp og ned er implementert selv. Det er også gjort noen endringer slik bruk av piltaster fører til endring av projection-matrisen til skyboxen, og dermed får en følelse av at man beveger seg inne i verdenen.

HJELPEKLASSER – SHADER OG TEXTURE

Hjelpeklasser for shader og texture, beskrevet på side 2 er implementert etter forklaring fra Sonans Systems sine youtube-video. Utenom endring av et par variabelnavn har jeg ikke gjort noe annerledes enn denne guiden. <https://www.youtube.com/watch?v=_EjsL3B9CRY>.

FARGER

Hjelp til å finne korrekte RGB-verdier for farge til lys 2

<http://doc.instantreality.org/tools/color_calculator/>

# 4. Litteraturliste