DATAGRAFIKK 2019

**Mette Strand Hornnes**

Innholdsfortegnelse

[1. Intro 3](#_Toc27083221)

[1.1 Model, view, projection 3](#_Toc27083222)

[1.2 Shadere 3](#_Toc27083223)

[1.3 Hjelpeklasser 4](#_Toc27083224)

[2 Implementasjoner 5](#_Toc27083225)

[2.1 First Person Camera 5](#_Toc27083226)

[2.2 Skybox 6](#_Toc27083227)

[2.3 Kube med Parallax Mapping 7](#_Toc27083228)

[2.4 Pyramide med Parallax mapping 8](#_Toc27083229)

[2.5 Hexogonal prisme 9](#_Toc27083230)

[2.6 Felles for objekter 9](#_Toc27083231)

[3. Kildekode, biblioteker og kompilering 10](#_Toc27083232)

[4. Hjelp og inspirasjon til implementasjon 10](#_Toc27083233)

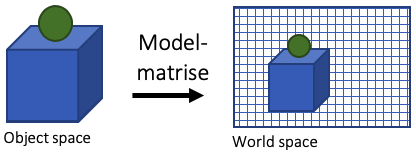
[5. Litteraturliste 11](#_Toc27083234)

# 1. Intro

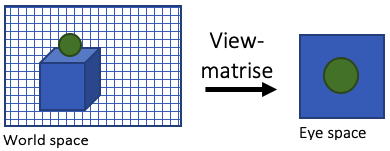
For å enklere kunne forklare implementasjonene ønsker jeg først å forklare generelt om de tre matrisene model, view og projection, samt om shadere. Sist vil jeg beskrive de to hjelpeklassene jeg benytter. Alle illustrasjoner i dokumentet er egenlaget.

## 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 sette eventuell rotasjon og skalering.

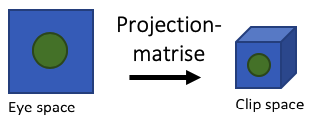


VIEW - «WORLD TO VIEW»   
Denne matrisen bestemmer hvordan verden (med objektet) skal plasseres i forhold til kameraet. Kameraet, som er våre øyne, vil alltid se nedover den negative z-aksen. For at objektene skal syntes, må derfor verden flytte seg.



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 (King, 2013).



## 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 som håndterer alle delene mellom disse punktene slik at det bygges flater. Det meste av texture og lys håndteres 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.

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 en filsti til vertex- og fragment shader. For begge shaderne vil kildekoden hentes inn og leses for å så 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.

TEXTURE

Texture.h holder på en TextureLoading-klasse som gjør 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 bilde skal lastes inn. Et bibliotek kalt SOIL hjelper til med dette. Deretter generes en id til texturen som skal benyttes, ulike attributter spesifiseres slik at det fremstår som ønsket og deretter bindes det til texturens plassering. Id-en returneres 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, benyttes GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X hvor x økes med en gang for hvert bilde.

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 i verdenen (Learn OpenGL, u.å).

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 forhåndsdefineres 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 vil igjen kalle funksjonen *processMouseMovement* i Camera-klassen og sende med x- g y-posisjonene til musepeker. 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,M,L 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 også view-matrisen fra *GetViewMatrix*-funksjonen. Denne vil også endre zoom-verdier da jeg ønsket en illusjon av at man beveger seg inne i skyboxen, samt dybde på parallax mappingen om disse tastene er trykket.

BRUK:

|  |  |
| --- | --- |
| **Piltaster eller A,W,S,D** = Venstre, frem, bak, høyre  **Z/X** = Ned / Opp | **Musebevegelser** = Endrer vinkel  **M/L** = Øker/Minker dybde på Parralax Mapping |

## 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 (Learn OpenGL, u.å).

GENERERER PUNKTER OG SETTER DE TIL VALGT BUFFER – *generateSkyBoxVerticesAndSetArraysAndBuffers()*For opprettelse 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.

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 fjernes. Dette gjøres ved å konverteres matrisen til en 3x3-matrise, og deretter tilbake igjen til en 4x4-matrise (Learn OpenGL, u.å). 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 eller lenger unna en side om man navigerer rundt.

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.

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 illusjon av at flatene på objektet har en dybde. I tillegg til en vanlig såkalt «diffuse texture», vil vi legge på to andre. En «normal-texture» som har en blåaktig farge og vil få frem små ujevnheter, og «displacement-texture» som er svart og hvit som vil «løfte» deler av texturen opp (Pluralsight, 2014).

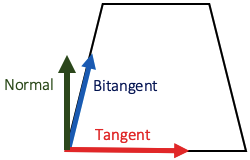
Sammen vil de bestemme hvor mye lys ulike fragment skal ha og hvilken vinkel det skal reflekteres tilbake. Dette vil skape en illusjon av at noe er nærmere eller lenger unna, og vi får med det en dybde i teksturen (Learn OpenGL, u.å).  
Dybden på Parallax Mappingen kan styres med tastene M (more) og L (less). Her kan man virkelig se den ekstra effekten Parallax Mapping gir ovenfor 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 posisjoner 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 (Learn OpenGL, u.å). 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.

TEGNER TRIANGLENE - *drawCube()*

En annen shader skal nå brukes. Denne funksjonen starter derfor med å definere dette, samt sende alle tre texturene, som trengs for å implementere parralax mapping, til den tilhørende fragment-shaderen. View-matrise til kuben blir satt ved hjelp av Camera sin *getViewMatrix* forklart tidligere. Til model-matrisen benytter jeg nå glm∷translate.   
Da alle objektene er plassert likt utifra 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 (langs x-aksen).

SETTER LYS-MATRISER – setLightMatricesForCubeAndPyramide()

Likt som alle objektene skal kuben ha to lys mot seg. Verdier for lysenes posisjon og farge, samt en view-matrise som brukes for begge lyskildene sendes til shaderne. Lys en har en hvit farge og vil sirkulere, mens 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.

SHADERE – *cubeAndPyramid.vert & cubeAndPyramid.frag*

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

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 for kuben. Da de begge skal ha parallax mapping implementert benyttes samme shadere.

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 forskjell er at den ene UV-vektoren for de to siste trekantene justeres. Årsaken til dette er at disse to til sammen skal utgjøre bunnen av pyramiden og jeg ønsket derfor å plasseres texturen noe annerledes.

TEGNER TRIANGLENE - *drawPyramid()*

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 rundt 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 bundet 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 objektene. 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 av 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-case for denne sjekken.

TEGNER TRIANGLENE - *drawHexPrism ()*

I likhet med de andre 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. Dette gjøres likt som for kube og pyramide, men da denne benytter andre shadere, kan ikke metode gjenbrukes.

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 disse objektene. For å få et inntrykk av at objekter lenger unna er mindre, bruker jeg glm∷perspective.

HENTER UNIFORM LOCATIONS – *initGL()*

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

# 3. Kildekode, biblioteker og kompilering

I tillegg til main.cpp består består prosjektet av følgende kildekode-filer:

Shadere: cubeAndPyramide.frag, cubeAndPyramide.vert, hexPrisme.frag, hexPrism.vert, skybox.frag og skybox.vert

Headerfiler: camera.h, shader.h og texture.h

For å kompilere prosjektet trengs følgende biblioteker:

Glew - OpenGL Extension Wrangler Library

SOIL2 - Simple OpenGL Image Loader

Glm - OpenGL Mathematics

Glfw - Graphics Library Framework.

# 4. Hjelp og inspirasjon til implementasjon

Dette prosjektet er 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. Så langt det har latt seg gjøre har jeg også forsøkt å unngå duplikat kode. 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. Jeg har også videreutviklet kode slik at bruk av piltaster fører til endring av projection-matrisen til skyboxen, samt mulighet til å påvirke dybden parallax mappingen.

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>.

BILDER

Bilder benyttet for de tre objektene er hentet Learn OpenGL sin guid om Parallax Mapping.  
Bilder til skybox er hentet fra Custom map makers. <http://www.custommapmakers.org/skyboxes.php>

# 5. Litteraturliste

King J. (2013, 21. oktober). *Model View Projection Matrices* [YouTube]. Hentet fra

<https://www.youtube.com/watch?v=-tonZsbHty8>

Learn OpenGL (u.å). *Camera.* Hentet fra <https://learnopengl.com/Getting-started/Camera>

Learn OpenGL (u.å). *Cubemaps.* Hentet fra <https://learnopengl.com/Advanced-OpenGL/Cubemaps>

Learn OpenGL (u.å). *Normal Mapping.* Hentet fra <https://learnopengl.com/Advanced-Lighting/Normal-Mapping>

Pluralsight (2014, 12. august). *Elliminate Texture Confusion: Bump, Normal and Displacement Maps.* Hentet fra <https://www.pluralsight.com/blog/film-games/bump-normal-and-displacement-maps>