DATAGRAFIKK OG VIRTUELLE MILJØER: PROSJEKTRAPPORT

Thomas Angeland, Vetle Deilkås, Vegard Strand

Høgskolen i Østfold, ITF21215 Datagrafikk og Virtuelle Miljøer

08.12.2017

Innholdsfortegnelse

Sammendrag

Rapporten tar for seg produksjonen av virtuelle omgivelser med OpenGL og C++. Her avdekkes og forklares de ulike teknologier og metoder som blir benyttet for å oppnå flere grafiske effekter. Rapporten vil ikke ta for seg hvert eneste steg og valg som ble tatt, men heller forklare hvordan effektene ble oppnådd og hvordan programmet er satt sammen. Med prosjektet er det vedlagt programkoden og prosjektfiler, samt en arbeidslogg og gruppekontrakt.

Takk til

Prosjektdeltakerne ønsker å takke faglærer Lars Vidar Magnusson for råd og veiledning i prosjektperioden. Det rettes også en varm takk til medstudenter hjelp og tips.

Matematikk-bibliotek

Da prosjektet var i startfasen, ble det brukt et C++ matematikk-bibliotek kalt OpenGL Mathematics (GLM)[1] for å utføre vektor- og matriseberegninger. GLM ble valgt for å komme raskt i gang med selve grafikkprogrammeringen. Før prosjektet hadde startet, var det allerede bestemt at prosjektet skulle benytte et matematikk-bibliotek laget av gruppen. Biblioteket bestod av C++ klasser for følgende matematiske begreper:

- To-komponentsvektorer
- Tre-komponentsvektorer
- Fire-komponentsvektorer
- 2x2-matriser
- 3x3-matriser
- 4x4-matriser

Matriser

Matematikk-biblioteket bydde på en utfordring når det gjaldt implementasjonen av matrisene, da OpenGL ser på matrisene som en 1-dimensjonal array av float verdier. OpenGL kan ikke jobbe med matriser representert av 2-dimensjonale arrays, som er enklest for mennesker å tolke. I matematikken ser man på matriser som en samling av elementer, der hvert element ligger i en rekke, og en kolonne. Programmeringsmessig vil dette ligne veldig på en 2-dimensjonal array. Utfordringen var å gå vekk fra den 2-dimensjonale array tankegangen, da det var utfordrende å se hvor elementene havnet i en 1-dimensjonal array. OpenGL bruker column-major notasjon på matriser, det vil si at elementene går kolonnevis istedenfor rekkevis.

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

Figur 1

m0	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12	m13	m14	m15

Figur 2

Fire-komponents vektorer ligger i hver kolonne som vist i matrisefiguren over (m0 - m3 utgjør en vektor, og så videre). Implementasjonen av en 4x4-matrise i kode vil se ut som

figuren over, der alle vektorene ligger lineært i arrayen. Matrisene vil ha lineært oppsett i minne, dette gjelder både for OpenGL sin notasjon (column-major) og f.eks DirectX[2] sin notasjon (row-major).

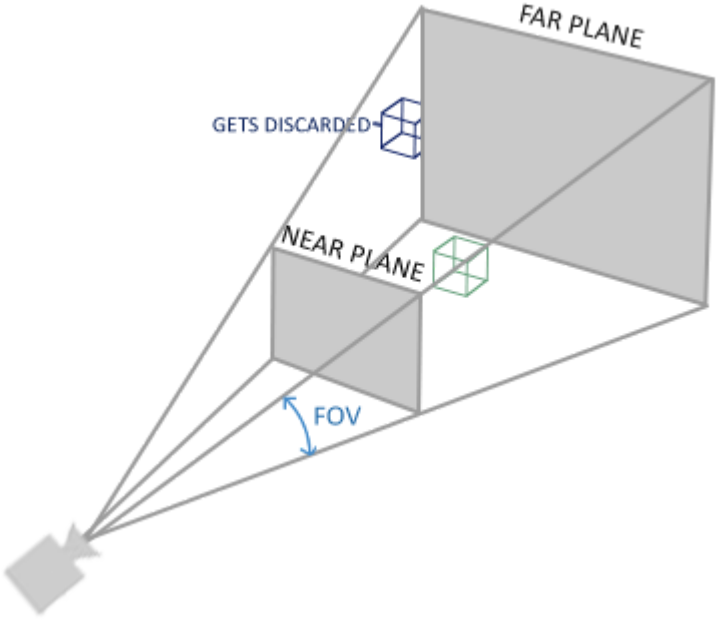
Det ble implementert metoder for å beregne følgende matriser:

- Identitetsmatrise
- Transformasjonsmatriser (scale, rotation og translation matriser)
- Projection matrise
- View matrise
- Multiplikasjon av matriser

Identitetsmatrisen er en matrise som ikke endrer en vektor ved å gange matrisen med vektoren. Om man har en identitetsmatrise m, og en vektor v, vil følgende gjelde:
m * v = v.

Transformasjonsmatrisene blir brukt for å endre vektorer. Scale matrisen endrer størrelsen (lengden) på en vektor. Rotation matrisen roterer en vektor rundt en gitt akse med en gitt vinkel. Translation matrisen flytter en vektor i x-retning, y-retning og z-retning. Alle disse matrisene må ganges med en vektor for å utføre transformasjonen på vektoren.

Projection matrisen prosjekterer 3D-verdenen på skjermen som et 2D-bilde. Projection matrisen inneholder informasjon om volumet av verden som skal vises på skjermen. Det vil si en synsvinkel, forholdet mellom bredden og høyden på skjermen, og to plan i rommet som kalles for near plane og far plane. Om et objekt ligger foran near plane, bak far plane eller utenfor synsvinkelen, vil ikke dette objektet bli tegnet på skjermen.



Figur 3

View matrisen brukes til å endre hele 3D-verdenen, ved å enten flytte den, rotere den, eller skalere den. Ved å flytte verden mot høyre, får man følelsen av å "gå" mot venstre. View matrisen blir derfor vanligvis brukt til å simulere et kamera.

Koordinater til objekter blir transformert gjennom 5 koordinatsystemer før de blir fragmenter på skjermen. Matrisene brukes for å utføre transformeringen mellom koordinatsystemene.

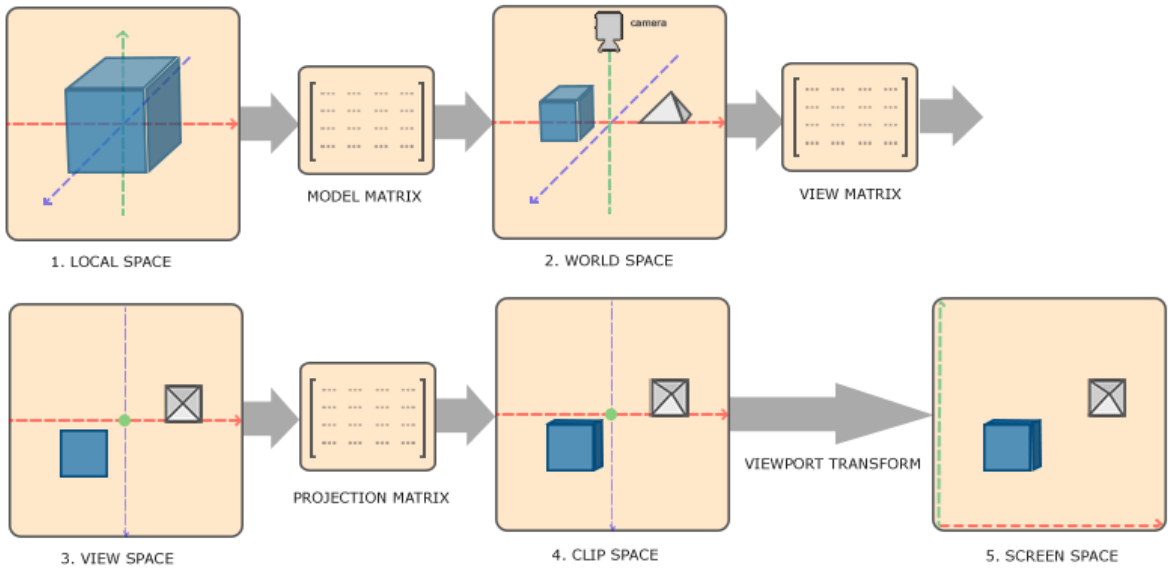
Koordinater til et objekt er i "local-space" etter at de har blitt definert. Det vil si at dette koordinatsystemet er der objektet blir laget. Vanligvis er koordinatsystemets origo i senter av objektene.

Transformasjonsmatrisene ganget sammen utgjør en model matrise som frakter koordinatene fra "local-space" til "world-space". "World-space" er det koordinatsystemet der objektenes koordinater ligger i forhold til en verden.

Ganger man så view matrisen med model matrisen fra venstre, frakter man koordinatene fra "world-space" til "view-space". Koordinatene i "view-space" vil ligge i forhold til synsvinkelen til brukeren (vanligvis kalt kameraet).

Til slutt kan man gange projection matrisen med view*model matrisen for å frakte koordinatene fra "view-space" til "clip-space". I "clip-space" vil koordinater utenfor et spesifikt område bli forkastet, mens koordinater som befinner seg i området blir ivaretatt (dette er spesifisert i projection matrisen).

Koordinatene blir så transformert til "screen-space" ved "viewport-transform" (dette gjør OpenGL automatisk). Det er her koordinatene blir omgjort til fragmenter som blir tegnet på skjermen.



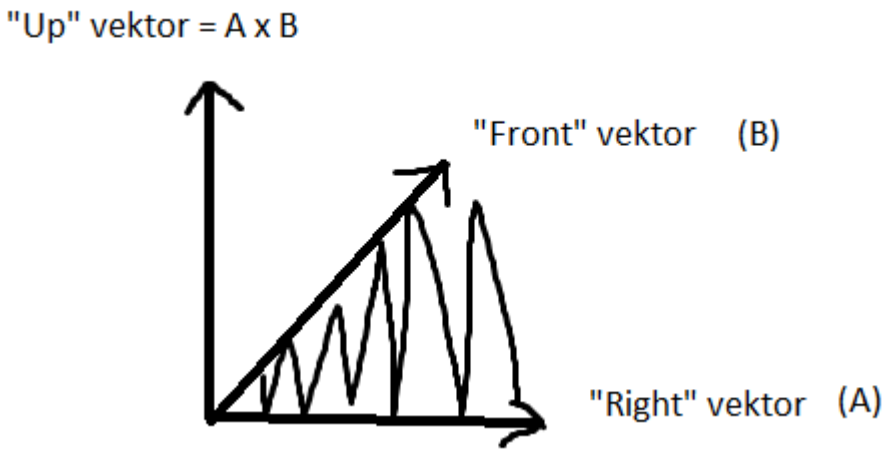
Figur 4

Vektorer

Metoder som trengtes for vektorregning var mindre utfordrende å implementere enn metodene for matriser, på grunn av vektorer er enklere å tenke på som en 1-dimensjonal array. Viktige metoder som ble implementert:

- Kryssprodukt
- Prikkprodukt
- Normalisering
- Aritmetiske operasjoner

Kryssprodukt av to vektorer blir brukt for å få en vektor som står ortogonalt på planet som vektorene danner. Dette blir for eksempel brukt til å finne “Up” vektoren til kameraet.



Figur 5

Prikkproduktet av to vektorer blir brukt for å vite vinkelforholdet mellom vektorene. Resultatet av et prikkprodukt er bare et tall. Prikkprodukt av vektor A og B er definert slik:

A dot B = |A| * |B| * cos(theta), hvor theta er vinkelen mellom vektorene.

Hvis A og B er enhetsvektorer, det vil si at lengden til disse vektorene er 1, vil A dot B være cosinus til vinkelen. Tallet som blir gitt av et prikkprodukt sier derfor noe om vinkelen mellom vektorene. Om tallet er 0, står vektorene 90 grader på hverandre. Om tallet er mindre enn 0, vil vinkelen mellom vektorene være større enn 90 grader. Om tallet er større enn 0, vil vinkelen mellom vektorene være mindre enn 90 grader. Disse egenskapene til prikkproduktet brukes for eksempel i kalkulasjonen av "diffuse" og "specular" belysning.

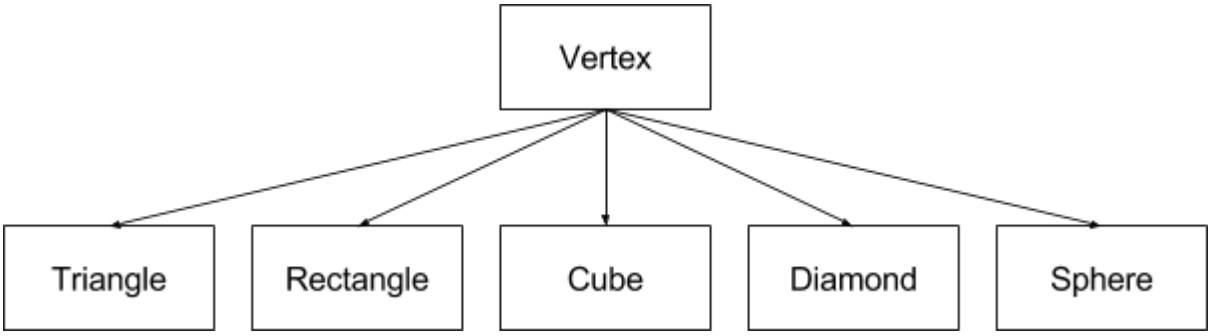
I mange tilfeller er det ønskelig å jobbe med enhetsvektorer (vektorer som har lengde lik 1), som for eksempel ved beregning av prikkprodukt. For å få vektorer til å bli enhetsvektorer, normaliserer man vektorene ved å dele alle komponentene til vektoren med lengden av vektoren.

```
vec3 vec3::normalize(const vec3& v) {
    float vectorLength = sqrt((v.x * v.x) + (v.y * v.y) + (v.z * v.z));
    return vec3(v.x / vectorLength, v.y / vectorLength, v.z / vectorLength);
}
```

Figur 6

3D objekterer og tegnefunksjoner

I programmet er laget flere klasser som tar seg av konstruering av ulike objekter: *Rectangle.h* (rectangel/flate), *Triangle.h* (trekant), *Cube.h* (boks, kube), *Diamond.h* (diamant) og *Sphere.h* (sirkel, ball). Klassene arver fra en overordnet klasse kalt *Vertex.h* som holder på *vertex data* som objekt-klassene konstruerer.



Figur 7

Vertex data og grafikkminnet

For å kunne tegne objekter i programvinduet må objektets vertex data lagres i grafikkminnet og prossesseres i grafikkprogrammene (vertex shader og fragment shader). Hvert objekt har vertex data som inneholder nødvendig informasjon om objektet på hvert punkt. I prosjektet lagres denne dataen slik:

$$data = [point_0,\ normal_0,\ color_0,\ uv_0,\ tangent_0,\ bitangent_0,\ point_1,\ normal_1,\ ...]$$

$point_0$ er en tre-komponent vektor som representerer koordinaten til en av mange punkter som definerer objektet. Denne benyttes, sammen med resten av punktene, til å tegne objektet.

$normal_0$ er en tre-komponent vektor som indikerer hvilken retning som er "utover" for dette punktet. Denne benyttes til lysberegning i fragment shaderen. Mer om dette senere.

$color_0$ er en tre-komponent vektor for fargen som skal tegnes på det punktet, hvis objektet ikke tegnes med tekstur. Vektoren representerer hver farge i RGB, altså rød, grønn og gul.

uv_0 er en to-komponent vektor som er en tekstur koordinat for når objektet tegnes med tekstur. Tekstur-koordinater går fra 0.0 til 1.0, hvor [0.0, 0.0] er nedre venstre hjørne og [1.0, 1.0] er øvre høyre hjørne. Hvis en tekstur har en oppløsning på 200x200 piksler vil for eksempel koordinaten [0.2, 0.8] hente ut pikselen på [40, 160]. Dette gjøres i fragment shaderen.

$tangent_0$ og $bitangent_0$ er tre-komponent vektorer som med $normal_0$ definerer et plan for hvert punkt. Dette planet har en x, y og z akse. Planet benyttes i en teknikk kalt *normal mapping* som gir objekter dybde. Mer om dette senere.

Hvert objekt kan også ha indekser som blir kalt *indices*. Indeksene forteller grafikkortet i hvilken rekkefølge vertex dataen skal tegnes. Grunnen til at man vil bruke indekser er fordi de fleste objekter har vertex data hvor noe av dataen er repeterende. Et hjørne på en firkant må defineres to ganger hvis man skal tegne en firkant uten indekser. Med indekser kan man redusere størrelsen på vertex dataen, og heller legge til indekser som tar mindre plass i grafikkminnet. Indeksene defineres forskjellig for hver objekt, og i prosjektet er det gjort slik at indekser er valgfritt.

Vertex-klassen tar seg av lagring av vertex data og indekser på grafikkminnet, samt en del andre funksjoner som benyttes tett med objekt-klassene. Lagring av vertex data gjøres med *VAO* (Vertex Array Object), *VBO* (Vertex Buffer Oject). Indekser lagres i en *EBO* (Element Buffer Object). Under er litt kode fra storeOnGPU()-funksjonen fra vertex-klassen.

```
glGenVertexArrays(1, &VAO); // Create VAO that stores the buffer objects.
glGenBuffers(1, &VBO); // Create VBO that stores vertex data
glGenBuffers(1, &EBO); // Create EBO that stores indices
glBindVertexArray(VAO); // Bind the VAO before binding and configuring buffers
glBindBuffer(GL_ARRAY_BUFFER, VBO); // Bind the VBO to the GL_ARRAY_BUFFER target
// Copy vertex data into the VBO currently bound to the GL_ARRAY_BUFFER target
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * dataSize(), data().data(), GL_STATIC_DRAW);
```

Figur 8

Først genereres en unik ID for VAO-, VBO- og EBO-bufferen med *glGenVertexArrays* og *glGenBuffers*. Deretter bindes VBO med *glBindBuffer* og vertex dataen legges til med *glBufferData*, hvor type data blir spesifisert, samt størrelsen og en pointer til dataen.

```
if (hasVertices()) {
    // Position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, stride, (void*)(verticeStride() * sizeof(float)));
    glEnableVertexAttribArray(0);
}
if (hasNormals())
{
    // Normals coordinate attribute
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, stride, (void*)(normalStride() * sizeof(float)));
    glEnableVertexAttribArray(1);
}
if (hasColors())
{
    // Colors coordinate attribute
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, stride, (void*)(colorStride() * sizeof(float)));
    glEnableVertexAttribArray(2);
}
if (hasUVs())
{
    // Texture coordinate attribute
    glVertexAttribPointer(3, 2, GL_FLOAT, GL_FALSE, stride, (void*)(uvStride() * sizeof(float)));
    glEnableVertexAttribArray(3);
}
if (hasTangents())
{
    // Tangents coordinate attribute
    glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, stride, (void*)(tangentStride() * sizeof(float)));
    glEnableVertexAttribArray(4);
}
if (hasBitangents())
{
    // Tangents coordinate attribute
    glVertexAttribPointer(5, 3, GL_FLOAT, GL_FALSE, stride, (void*)(bitangentStride() * sizeof(float)));
    glEnableVertexAttribArray(5);
}
if (hasIndices()) {
    // Bind the EBO to the GL_ELEMENT_ARRAY_BUFFER target
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    // Copy indices data into the EBO currently bound to the GL_ELEMENT_ARRAY_BUFFER target
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int) * indices.size(), indices.data(), GL_STATIC_DRAW);
}
```

Figur 9

For å ikke skrive tom data til grafikkminnet, er det gjort slik at bare de nødvendige vertex dataen skrives og pekes til. Så hvis et objekt ikke trenger å ha farger, legges ikke dette med. Hver datatype i vertex dataen for hver sin peker og hentes ut på denne i grafikkprogrammene. Dette gjøres med funksjonen *glVertexAttribPointer* hvor det må legges inn størrelsen på dataen (i bytes) på hver punkt og hvor mange bytes fram den aktuelle vertex dataen ligger.

Hvis objektet har indekser, lagres disse på samme måte som vertex dataen, men i EBO-bufferen. EBO-bufferen bindes til VAO-bufferen slik at man bare trenger å kalle på VAO når man skal tegne et spesifikt objekt.

```
// Bind textures if if it points to a texture
if (material != nullptr) material->bind();
// Bind VAO
glBindVertexArray(VAO);
// Calculate the model matrix for each object and pass it to shader before drawing
mat4 translate = mat4::makeTranslate(position);
mat4 rotate = mat4::makeRotate(rotation_degrees, rotation_vector);
mat4 scale = mat4::makeScale(scale_vector);
mat4 model = translate * this->position * rotate * this->rotate * scale * this->scale;
shader->setMat4("model", model);
if (!scaleTexture)
    shader->setVec2("scale", vec2(scale_vector.x * uv_scale.x, scale_vector.y * uv_scale.y));
else
    shader->setVec2("scale", vec2(1.0f, 1.0f));
// Draw mesh
if (hasIndices())
    glDrawElements(draw_mode, indices.size(), GL_UNSIGNED_INT, 0);
else
    glDrawArrays(draw_mode, 0, size());
// Unbind textures
if (material != nullptr) material->unbind();
return true;
```

Figur 10

Tegnefunskjonen *drawObject()* i vertex-klassen kalles på når et objekt skal tegnes. Som parameter legges ved den aktuelle shaderen som skal benyttes. Hvis objektet tegnes med tekstur skal denne legges med. Man kan endre posisjonen, skalere og rotere objektet hvis ønskelig.

Transformasjonsmatrisen *model* regnes ut med å multiplisere posisjonsmatrisen *translate*, rotasjonsmatrisen *rotate* og skalermatrisen *scale*, i den rekkefølgen. Hver objekt har også sine egne transformasjonsmatriser som kan defineres i konstrueringen av objektet. Hvis objektet er statisk (beveger seg ikke), er det anbefalt å definere transformasjonsmatrisen utenfor løkken slik at det sparer prosessoren for noen unødvendige kalkulasjoner.

Det sendes også en skalerings vektor til den aktuelle shaderen på uniformen “scale”. I shaderen skaleres UV-koordinatene for hvert punkt. Slik vil ikke teksturene strekkes. Dette er også valgfritt for hvert objekt, dvs. man kan velge om teksturen skal strekkes eller ikke.

```
UV = vec2(aUVs.x * scale.x, aUVs.y * scale.y);
```

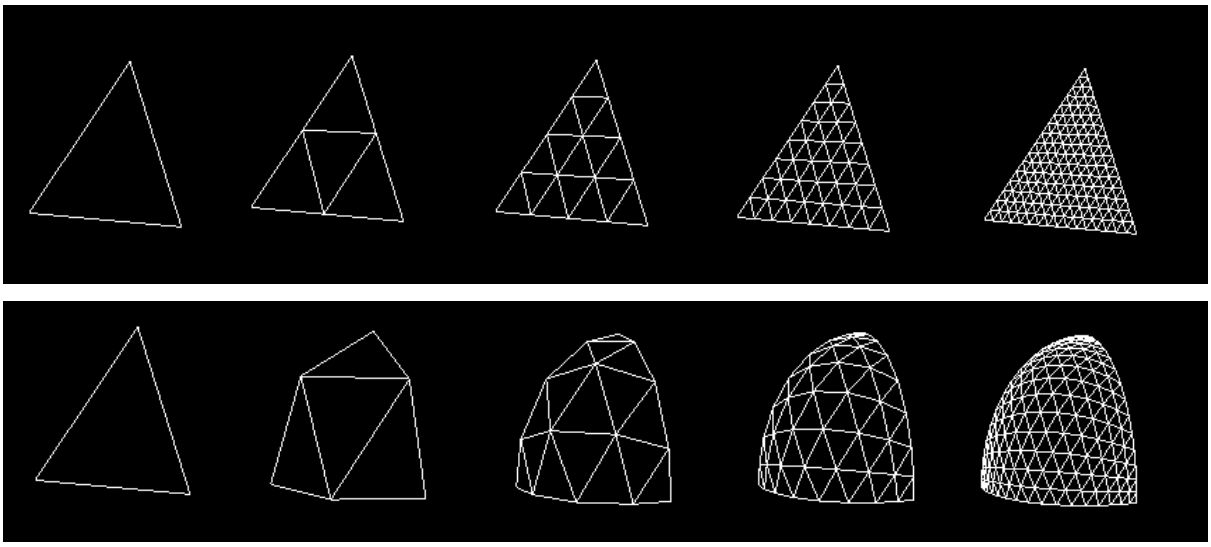
Figur 11

Hvis objekter har indekser benyttes funksjonen *glDrawElements* til å tegne objektet. Der oppgis antall indekser som skal tegnes, hvor det alltids oppgis størrelsen på indeks-listen.

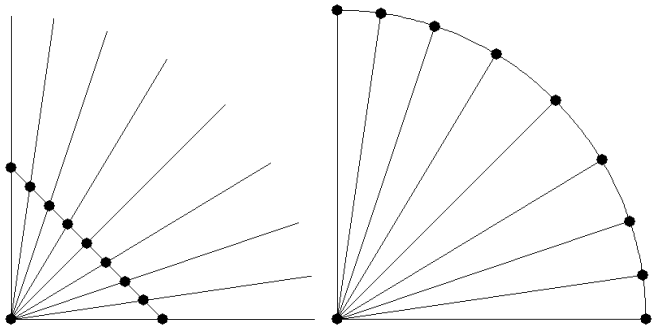
Hvis objektet ikke har indekser, benyttes funksjonen *glDrawArrays* hvor det oppgis hvor mye vertex data som skal tegnes.

Sphere-klassen

Å tegne en kule viste seg å bli vanskeligere enn først forestilt. Det ble tilslutt benyttet en teknikk med hjelp fra et innlegg på StackOverflow[3]. Først må man dele opp en pyramide i mange deler, for så å normalisere alle punktene med midtpunktet av pyramiden og flytte de lik distance ut fra midtpunktet. Figur 21 viser hvordan en trekant deles opp og hvordan ser ut etter normaliseringen. Figur 22 viser normaliseringen og flyttingen av punkter.



Figur 12



Figur 13

Desto flere ganger pyramiden deles opp, desto "bedre kvalitet" får kulen, men vertex dataen vil bli mye større (eksponentiell større). Oppdelingfunksjonen *subdivide* i vertex-klassen benyttes til oppdeling av et vilkårlig objekt. For hvert kall deler den opp hvert polygon i 3 nye polygoner.

```
std::vector<vec3> Vertex::subdivide(const std::vector<vec3>& vertex_data)
{
    std::vector<vec3> subdivided;
    for (unsigned int index = 0; index < vertex_data.size(); index += 3)
    {
        // Vertices

        vec3 v1 = vertex_data.at(index + 0);
        vec3 v2 = vertex_data.at(index + 1);
        vec3 v3 = vertex_data.at(index + 2);

        vec3 va = vec3::midpoint(v1, v2);
        vec3 vb = vec3::midpoint(v2, v3);
        vec3 vc = vec3::midpoint(v1, v3);

        subdivided.push_back(v1);
        subdivided.push_back(va);
        subdivided.push_back(vc);

        subdivided.push_back(va);
        subdivided.push_back(vb);
        subdivided.push_back(vc);

        subdivided.push_back(va);
        subdivided.push_back(v2);
        subdivided.push_back(vb);

        subdivided.push_back(vc);
        subdivided.push_back(vb);
        subdivided.push_back(v3);
    }
    return subdivided;
}
```

Figur 14

```
for (int d = 0; d < times; d++)
{
    if (hasVertices()) vertices = subdivide(vertices);
    if (hasNormals()) normals = subdivide(normals);
    if (hasColors()) colors = subdivide(colors);
    if (hasUVs()) uvs = subdivide(uvs);
    if (hasTangents()) tangents = subdivide(tangents);
    if (hasBitangents()) bitangents = subdivide(bitangents);
}
```

Figur 15

For å lage en kule trenger man bare å oppgi diameter og "kvalitet". Funksjonen som lager kulen er i Sphere-klassen og ser slik ut:

```
void Sphere::createSphere(const float width, const unsigned int quality) {
    float radius = width / 2.0f;

    Diamond diamond = Diamond(width);

    vertices = diamond.vertices;
    uvs = diamond.uvs;
    subdivide(quality);

    for (int i = 0; i < vertices.size(); i++)
        vertices.at(i) = vec3::scale(vec3::normalize(vertices.at(i)), width);
}
```

Figur 16

Først deles vertex dataen opp i flere deler. Deretter blir hvert punkt normalisert og skalert med diameteren *width*.

Teksturer

Alle teksturer som benyttes med objektene blir lastet med Texture-klassen i prosjektet. Vanlige bildefiler blir lastet med konstruktøren, hvor filstien til teksturen oppgis som parameter.

```
Texture::Texture(char const * path)
{
    glGenTextures(1, &id);

    int components;
    unsigned char *data = stbi_load(path, &width, &height, &components, 0);
    if (!data) {
        printf("Error: Texture failed to load at path: %s", path);
        stbi_image_free(data);
    }

    GLenum format;
    if (components == 1)
        format = GL_RED;
    else if (components == 3)
        format = GL_RGB;
    else if (components == 4)
        format = GL_RGBA;

    // Bind texture and generate mipmap
    glBindTexture(GL_TEXTURE_2D, id);
    glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);

    // Set texture parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    // Free data and unbind texture
    stbi_image_free(data);
    glBindTexture(GL_TEXTURE_2D, 0);

}
```

Figur 50

Det genereres en unik ID med *glGenTextures*. Deretter leses filen med *stbi_load*, en funksjon fra det eksterne biblioteket STB[17] laget av Sean Barrett. Det ble bestemt tidlig og ikke bruke tid på å lage egne bildeimporteringsfunksjoner så derfor er dette biblioteket benyttet. Dette biblioteket aksepterer også flere forskjellige bildetyper. Deretter hentes ut fargekomponentet/formatet fra filen. Vanligvis blir dette RGB eller RGBA.

glBindTexture aktiverer ID som ble generert så alt som gjøres videre fester seg til denne. Teksturen lages som en 2D tekstur med *glTexImage2D* hvor formatet, høyden, bredden og fargedata til bilde blir lagt med som parametere.

glGenerateMipmap genererer *mipmap* for teksturen. Kort sagt lager den ekstra teksturer i mindre størrelser som benyttes desto lenger unna teksturen som rendres. Det er ikke nødvendig å benytte teksturer av høy kvalitet når teksturen rendres langt unna kamera, så teksturen byttes ut med en versjon som er av lavere kvalitet for å spare på ressurser.

glTexParameteri setter hvordan teksturen skal "kles" objektflatene. Hvis teksturen ikke strekker over hele objektflaten bestemmes det at teksturen skal repeteres. Ofte er teksturene lagd slik at de er sømløse (seamless) slik at de repeteres med jevn overgang. Derfor benyttes denne parameteren. glTexPaameteri setter også filtreringsmetoden for mipmap. Denne er satt til lineær (linear).

Det finnes flere typer teksturer som går sammen og representerer et objekt på ulike måter. I prosjektet er det tilrettelagt for *diffuse*-, *specular*-, *normal*-, *displacement*- og *ambient occlusion* teksturer. For å binde disse sammen i en pakke ble det laget en enkel Material-klassen som tar inn disse ulike teksturene. Material-objekter blir benyttet når 3D objekter skal tegnes.

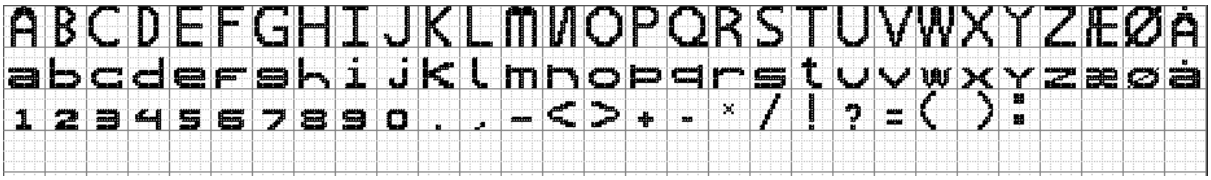
```
class Material
{
private:
    bool diffuseBound = false, specularBound = false, normalBound = false, displacementBound = false, AOBound = false;
public:
    Texture diffuse;
    Texture specular;
    Texture normal;
    Texture displacement;
    Texture ambient_occlusion;
```

Figur 51

Material-klassen holder også oversikt over hvilke tekstur-typer som er lagt til.

Text rendering

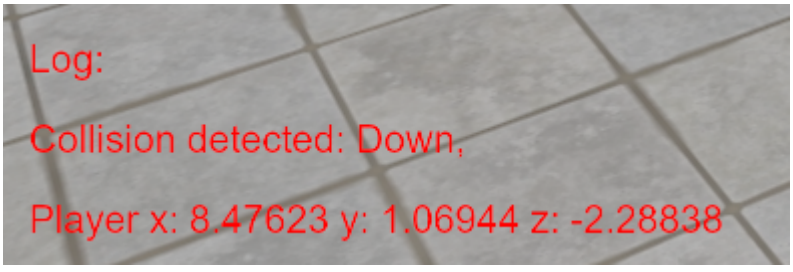
Tidlig i prosjektet ble det diskutert text rendering for å enklere gi informasjon og tilbakemelding av variabler i sanntid. I starten arbeidet gruppen med en populær tekstendring løsning kalt *bitmap font*. Dette innebærer å laste inn en spritesheet av tegn og dele opp i hvert enkelt symbol i mindre kvadrater kalt *glyphs*. Med en streng og en enkel for-løkke kan programmet hente ut tegn fra spritesheet med et koordinatsystem som peker til hvert tegn.



Figur 17

Figur 18

En slik løsning kan være tidkrevende å implementere, spesielt hvis det ønskes tekst med fint utseende. Denne løsningen har blitt implementert av prosjektdeltaker Vegard Strand i et tidligere prosjekt, men med java spesifikke klasser som ikke fantes erstatninger til i C++. Etter en del søking etter oppskrift eller en enkel algoritme i C++ ble det bestemt at det vil være for tidkrevende å implementere. Prosjektdeltakerne vil ikke bruke mye tid eller fokus på tekst rendering. Derfor tas det i bruk et ferdiglaget tekst rendering bibliotek kalt *freetype*[4].



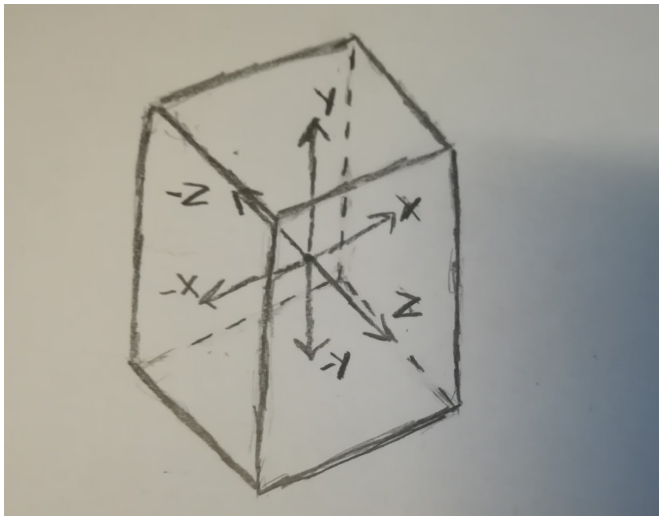
Figur 19

Freetype genererer tegn med vektorgrafikk og lager fine avstander mellom ord (noe som kan være tidkrevende å lage med bitmap font). Biblioteket trenger en filsti til en fontfil og vil med det generere alt som behøves. Oppsett i OpenGL er kopiert fra nettsiden LearnOpenGL[5] og det har blitt laget enkle metoder bygd rundt dette eksemplet for tekstendring. Dette biblioteket kan bruke en del ressurser av initialiseringen av prosjektet, spesielt første gang prosjektet blir kjørt på en ny maskin.

Kollisjonsdeteksjon

For å skape en dynamisk virtuell miljø vil det som oftest alltid kreve kollisjonsdeteksjon. Dette innebærer å sjekke skjæringspunkter mellom to eller flere objekter, og er nødvendig for at spill eller simulasjon skal virke livlig. Det ble bestemt og benytte en *hitbox*-løsning pakket rundt OpenGL-rendrede objekter. En kube er en av de enkleste formene for kollisjonsdeteksjon, noe som er implementert i prosjektet. Det var ment som en start til videreutvikling av andre hitbox løsninger med dessverre grunnet en del bugs og mangel på tid ble dette løsningen som ble brukt.

Kollisjonsdeteksjonløsningen bygger seg på en lang if setning som sjekker om spilleren kolliderer med andre objekter sin hitbox ut fra posisjonen til objektet i 3D space og avgrenser dens hitbox med ut fra posisjonen og størrelsen til objektet. Dette betyr at det blir en konstant sjekk av mengder som returnerer om spillerens hitbox eksisterer i en annens hitbox.



Figur 20

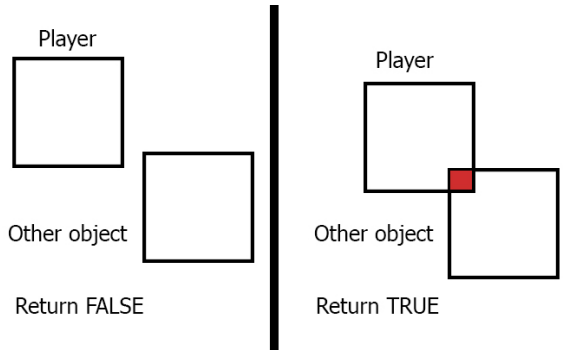
For å holde styr på koordinatene for andre objekter i 3D space tas det i bruk en Entity klasse som holder på verdier som objektets posisjon i x, y, z retning, størrelse (scale), fast(solid) og hvor stor prosent av hitboxen skal dekke størrelsen av objektet. Prosent delen kommer av at det ønsker en hitbox som er større enn objektet, dette kan komme av at det ønskes at spilleren skal "interacte" med et objekt. For eksempel spilleren skal plukke noe i spillverden, da er det gunstig å ha en hitbox som gir tilbakemeldinger om at spilleren er nærme nok objekter slik at spilleren kan plukke den opp.

Kollisjonsdeteksjon Løsning kamera(Player) mot object B 3D space:

$P = Player$
$O = other\ object$

$$
\begin{aligned}
&f(P,O) = \\
&(P_{positionZ} + P_{hitboxZ}) \geq O_{positionZ} + (O_{hitboxZ} \cdot O_{scaleZ} - O_{scaleZ} \cdot 2) \wedge \\
&(P_{positionZ} + P_{hitboxZ}) \leq O_{positionZ} + (O_{hitboxZ} \cdot O_{scaleZ}) \wedge \\
&(P_{positionY} - P_{hitboxY}) \leq O_{positionY} + (O_{hitboxY} \cdot O_{scaleY}) \wedge \\
&(P_{positionY} + P_{hitboxY}) \geq O_{positionY} + (O_{hitboxY} \cdot O_{scaleY} - O_{scaleY} \cdot 2) \wedge \\
&(P_{positionX} - P_{hitboxX}) \leq O_{positionX} + (O_{hitboxX} \cdot O_{hitboxX}) \wedge \\
&(P_{positionX} + P_{positionX}) \geq O_{positionX} + (O_{hitboxX} \cdot O_{ScaleX} - O_{scaleX} \cdot 2)
\end{aligned}
$$

Visualisert:



Figur 21

Denne sjekken vil kun gi true om spilleren treffer en annet objekt. Deretter er det en siste sjekk som tester om objektet spilleren treffer er solid (fast). Hvis objektet er solid kan ikke spilleren gå igjennom objektet.

Det er også behov for kollesinsdetetkjson som sjekker under spilleren. Dette kreves for blant annet for gravitasjon og sjekke kontakt med bakken.

Kollisjon under spilleren, bakken.:

$$
\begin{aligned}
&P = Player \\
&O = other\ object\ /\ floor
\end{aligned}
$$

$$
\begin{aligned}
&f(P,O) = \\
&(P_{positionY} + P_{hitboxY}) \leq O_{positionY} \wedge \\
&(P_{positionX} - P_{hitboxX}) \leq O_{positionX} + (O_{hitboxX} \cdot O_{hitboxX}) \wedge \\
&(P_{positionX} + P_{positionX}) \geq O_{positionX} + (O_{hitboxX} \cdot O_{ScaleX} - O_{scaleX} \cdot 2) \wedge \\
&(P_{positionZ} + P_{hitboxZ}) \geq O_{positionZ} + (O_{hitboxZ} \cdot O_{scaleZ} - O_{scaleZ} \cdot 2) \wedge \\
&(P_{positionZ} + P_{hitboxZ}) \leq O_{positionZ} + (O_{hitboxZ} \cdot O_{scaleZ}) \wedge
\end{aligned}
$$

Gravitasjon er en konstant kraft ned med -y. I prosjektet vil det alltid legges til en liten sum på -0.01 etter hvert bilde hvis det ikke er noe kollisjon under kameraet. Hvis kollisjon er oppdaget under spilleren vil kraften ned -y settes til null fram til det ikke er oppdaget kollisjon under spilleren.

Hopping er relativt enkelt å implementer med en kraft y som påvirker spilleren oppover men vil bli dratt ned igjen av tyngdekraften -y og vil falle tilbake igjen.

I prosjektet vil det søkes etter kollisjon mot alle enheter som kan kollideres med kameraet. Dette er svært ineffektivt og burde lage en løsning som sjekker objekter kun i nærheten av kameraet. En populær løsning er å benytte en søketre som deler opp "verden" i deler og vil gjøre kollisjonsdeteksjon aktivt i de delene kameraet befinner seg i. populære søketrær er: Quadtree[6], Octree[7], Binary Space Partitoning (BSP)[8], Sweep and prune (SAP)[9].

Det er mye mer som kan implementeres i kollisjonsdeteksjon. Det ble funnet en artikkel på hvordan en mer dynamisk kollisjonsdeteksjon med blant annet sliding[10]. Hvis det skulle jobbes videre med dette prosjektet ville denne artikkelen bli brukt som mal for videreutvikling.

Lys

Lys kommer som regel fra flere lyskilder spredt rundt oss. Selv fra lyskilder som ikke syntes. Det finnes også flere typer lys, og i prosjektet er det definert tre typer lys: Retningslys (directional light), spotter (spot light) og punktlys (point light). Det er laget en klasse for hver av disse lystypene som arver fra en felles lysklasse *Light.h*. Lysene sendes med hver sine tilhørende verdier til fragment shaderen *object_frag.shader* å lagres som egendefinerte *struct* objekter i hver sin liste. Figur 22 viser PointLight-structen.

```
struct PointLight {
    vec3 position;

    float constant;
    float linear;
    float quadratic;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
```

Figur 22

I fragment shaderen er det laget en funksjon for hver av disse lystypene som regner ut fargen på hvert fragment (piksel). Funksjonene tar inn parametrene *normal* og *view_direction*. Se figur 23.

```
vec3 CalcDirectionLight(DirectionLight light, vec3 normal, vec3 view_direction);
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 Point, vec3 view_direction);
vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 Point, vec3 view_direction);
```

Figur 23

Vektorene normal og view må gjøres om til *unit vectors* og må da normaliseres til maks lengde 1 med funksjonen *normalize()*. Dette må gjøres slik at dot produktet av vektorene gir ut cosinen, og ikke mer enn det. Det trekkes også fra x, y og z koordinatene på view.

```
vec3 normal = normalize(Normal);
vec3 view_direction = normalize(viewPos - Point);
```

Figur 24

Funksjonene for lystypene returnerer hver seg en tre-komponent vektor. I hovedprogrammet *main* i fragment shaderen legges disse sammen i en vektor kalt *result*. Hver liste med lys blir gått igjennom slik at alle lysene er med i beregningen. Se figur 25.

```
vec3 result = vec3(0.0f);
// Calculate directional light(s)
for (int i = 0; i < directionLightCount; i++)
    result += CalcDirectionLight(directionalLights[i], normal, view_direction) * DIRLIGHT_STRENGTH;
// Calculate point light(s)
for (int i = 0; i < pointLightCount; i++)
    result += CalcPointLight(pointLights[i], normal, Point, view_direction) * POINTLIGHT_STRENGTH;
// Calculate spot light(s)
for (int i = 0; i < spotLightCount; i++)
    result += CalcSpotLight(spotLights[i], normal, Point, view_direction) * SPOTLIGHT_STRENGTH;
```

Figur 25

Ambient

I prosjektet ble det planlagt å forsøke å implementere *Global Illumination*, men dette måtte dessverre prioriteres bort for å ha tid til andre effekter. Istedenfor ble det benyttet en enkel metode for å regne *Ambient Lighting*. Selvom det er mørk er det fortsatt en konstant mengde lys i rommet. Denne ambient-konstanten benyttes for alle lyskilder, og kan justeres forskjellig på hvert lys. Formelen benyttet for å regne ambient lys er noe forskjellig for hver lystype.

$$ambient_{dirlight} = ambient_{lys} \cdot texture_{diffuse}(x,y)$$
$$ambient_{pointlight} = ambient_{lys} \cdot texture_{diffuse}(x,y) \cdot attenuation_{lys}$$
$$ambient_{spotlight} = ambient_{lys} \cdot texture_{diffuse}(x,y) \cdot attenuation_{lys} \cdot intensity_{lys}$$

$ambient_{lys}$ er en float-verdi som bestemmes for hver lyskilde. Hver lystype har bestemte standarder på denne verdien, men det kan endres i programmet via lys-klassene.

$texture_{diffuse}(x,y)$ er en tre-komponent vektor for fargen på pikselen på det gjeldende fragmenten. Den finner man ved å avlese pikselen på diffuse teksturen med funksjonen texture(material.diffuse, UV), hvor *material.diffuse* er en 2D tekstur og *UV* er tekstur koordinaten for pikselen. Diffuse teksturer er teksturer som skal være flate og kun gi en fargeantydning til objektet. I prosjektet benyttes flere teksturer for forskjellige objekter. Figur 26 viser *tile* teksturen.

Figur 26

$attenuation_{lys}$ er en reduksjonsfaktor som indikerer hvor mye fargen skal reduseres (bli mørkere). Desto lenger unna lyskilden er, desto mørke skal fargen bli. Formelen for attenuation gis som:

$$attenuation = \frac{1.0}{constant + linear \cdot distance + quadratic \cdot distance^2}$$

Constant, linear og quadratic er variabler som settes for hvert lys. Distance regnes med lengden av lysposisjonen minus x, y og z koordinaten.

$intensity_{lys}$ er lysintensiteten for spotter. Den regnes ut med to variabler, *cutOff* og *outerCutOff*, lagret ved spotter, samt dot-produkten mellom lysretning og motsatt lysretning.

```
float theta = dot(lightDir, normalize(-light.direction));
float epsilon = light.cutOff - light.outerCutOff;
float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);
```

Figur 27

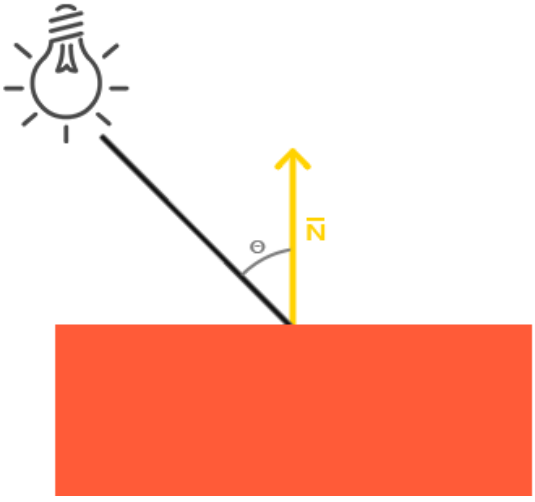
I koden ser ambient beregningen for directional light slik ut:

```
vec3 ambient = light.ambient * vec3(texture(material.diffuse, UV));
```

Figur 28

Diffuse

Diffuse er en annen type lysegenskap som implementeres i prosjektet. Formelen benyttet for å regne diffusen er nesten lik ambient-formelen. Et ekstra steg blir å multipliseres inn dot-produkten av normalen (retningen til objektet) og lysretningen, i programmet kalt *angle*. Når vinkelen mellom lys og objektflate er 0 grader blir dot-produkten 1.0. Da er lyset rettet rett på objektflaten og lysintensiteten er høyest. Hvis vinkelen er 90 blir dot-produktet 0.0. Se på dette som en prosentverdi. Se figur 6.



Figur 29

$$diffuse_{dirlight} = ambient_{lys} \cdot angle \cdot texture_{diffuse}(x,y)$$
$$diffuse_{pointlight} = ambient_{lys} \cdot angle \cdot texture_{diffuse}(x,y) \cdot attenuation_{lys}$$
$$diffuse_{spotlight} = ambient_{lys} \cdot angle \cdot texture_{diffuse}(x,y) \cdot attenuation_{lys} \cdot intensity_{lys}$$

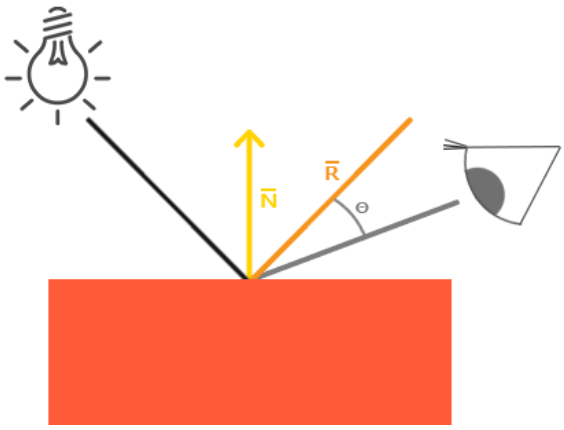
I koden ser diffuse beregningen for directional light slik ut:

```
vec3 diffuse = light.diffuse * angle * vec3(texture(material.diffuse, UV));
```

Figur 30

Specular

Specular sier noe om hvor mye et objekt skal reflektere lys, og hvor på objektene lys reflekteres mer. Den reflektive fargen regnes med en formel lik ambient-formelen, men det multipliseres også inn en verdi kalt *spec*. spec er verdien av dot-produktet mellom view_direction og reflect_direction og denne opphøyes i en *shininess*-verdi som medfølger hver tekstur. Shininess verdien er med på å justere refleksjonen.

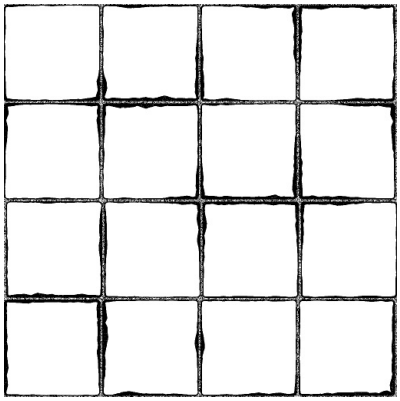


Figur 31

```
float spec = pow(max(dot(view_direction, reflect_direction), 0.0), material.shininess);
```

Figur 32

I prosjektet kombineres dette med spekulær mapping (specular mapping). Det blitt lagt til rette for spekulær mapping i prosjektet, hvor hvert objekt kan ha ekstra tekstur kalt specular. Specular tekstur er en tekstur som indikerer hvor det skal skinne mye og lite, basert på hvor fargen/lysstyrken på den gitte pikselen. Hvis fargen er hvit (100% hvit, [1.0, 1.0, 1.0]), skal det reflektere 100% av fargen. Hvis fargen er svart (0% hvit, [0.0, 0.0, 0.0]), skal det reflektere 0% av fargen. Figur 33 viser en spekulær versjon av tile-teksturen.



Figur 33

Formelen for spekulær ser slik ut:

$$specular_{dirlight} = specular_{lys} \cdot spec \cdot texture_{specular}(x,y)$$
$$specular_{pointlight} = specular_{lys} \cdot spec \cdot texture_{specular}(x,y) \cdot attenuation_{lys}$$
$$specular_{spotlight} = specular_{lys} \cdot spec \cdot texture_{specular}(x,y) \cdot attenuation_{lys} \cdot intensity_{lys}$$

I koden ser spekulær beregningen for directional light slik ut:

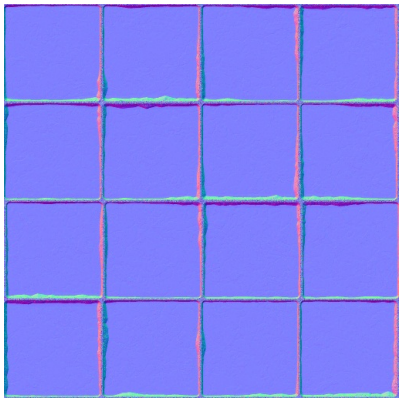
```
vec3 specular = light.specular * spec * vec3(texture(material.specular, UV));
```

Figur 34

Normals

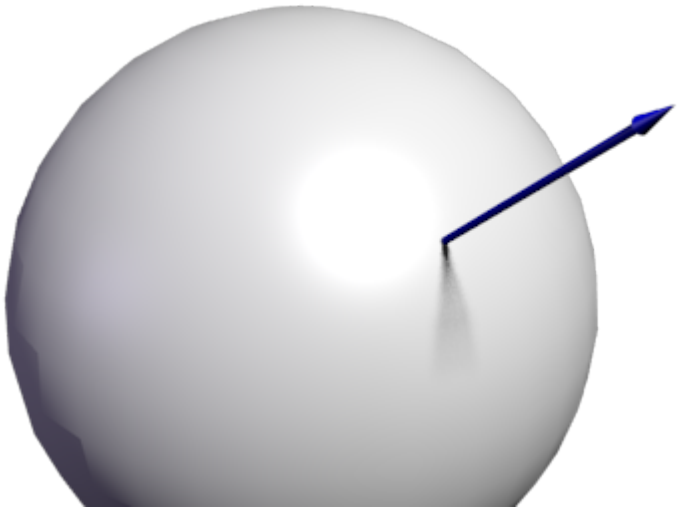
Ojekter og teksturer har ofte små detaljer og formgivinger som ikke stikker godt nok fram. For eksempel så har tile-teksturen fliser som bør stikke litt ut. Det kunne ha blitt lagd realistiske 3D modeller hvor hver flis blir modellert, og dette hadde nok sett best ut. Desverre så vil dette øke størrelsen på vertex dataene i grafikkminnet. Ettersom programmet baserer seg på *realtime rendering*[11], er det ofte nødvendig å spare på ressurser der man kan.

På bakgrunn av dette er det bestemt å tilrettelegge for *Normals*. Dette er den siste lyseffekten som benyttes i prosjektet. Et triks som på mange måter skaper illusjon om dybden i en tekstur. I tillegg til diffuse- og spekulær tekstur, legges det også med en *normal tekstur*, kalt normal map eller bump map.



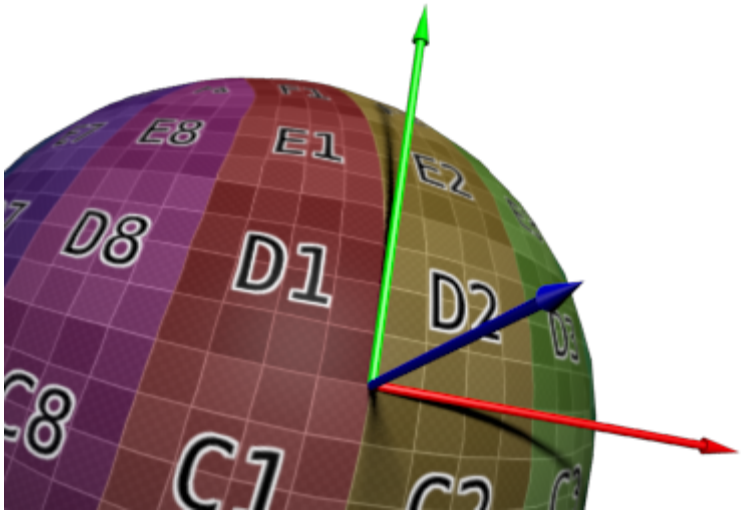
Figur 35

Hvert objekt i prosjektet har normal vektorer, en vektor som forteller hvilken retning flaten til objektet peker ut (90 grader).



Figur 36

I tillegg til dette må *tangents* og *bitangents* legges med. Med disse vektorene lages det et plan for hver vertice. Tangents og bitangents regnes ut i funksjonen calculateTangents() i Vertex-klassen. Figuren under viser hvordan planet ser ut for en vertice på en sirkel.



Figur 37

Med denne informasjonen kan den inverse *TBN matrisen* beregnes[12] slik at vi kan transformere alt fra *model space* til *tangent space*. Denne utregningen blir gjort i vertex shaderen *object_vert.shader*.

```
mat3 normalMatrix = transpose(inverse(mat3(model)));
vec3 T = normalize(normalMatrix * aTangent);
vec3 B = normalize(vec3(model * vec4(aBitangent, 0.0)));
vec3 N = normalize(normalMatrix * aNormals);

mat3 TBN = transpose(mat3(T, B, N));
for (int i = 0; i < lightCount; i++)
    TangentLightPos[i] = TBN * lightPositions[i];
TangentViewPos = TBN * viewPos;
TangentPoint = TBN * Point;
```

Figur 38

Listen *TangentLightPos[]*, *TangentView* og *TangentPoint* sendes videre til fragment shaderen hvor normalen blir beregnet for hvert fragment. Dette gjøres i funksjonen *CalcNormals()* som returnerer fargen på fragmentet.

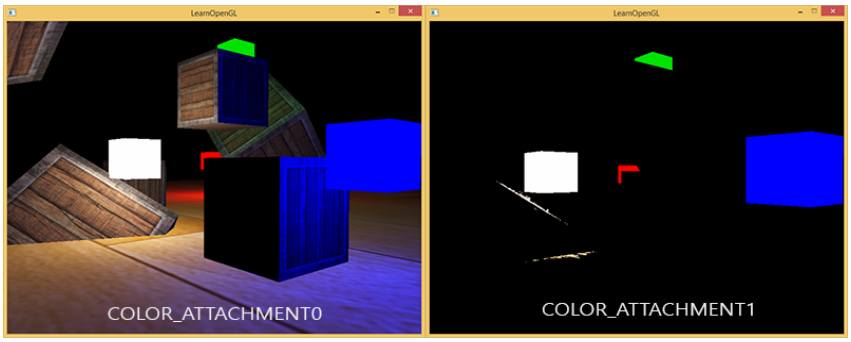
```
vec3 CalcNormals(vec3 tangentLightPos)
{
    // obtain normal from normal map in range [0,1]
    vec3 normal = texture(material.normal, UV).rgb;
    // transform normal vector to range [-1,1], normal is in tangent space
    normal = normalize(normal * 2.0 - 1.0);
    // angle
    vec3 light_direction = normalize(tangentLightPos - TangentPoint);
    float angle = max(dot(light_direction, normal), 0.0);
    // get diffuse color
    vec3 color = texture(material.diffuse, UV).rgb;
    // ambient & diffuse
    vec3 ambient = 0.1 * color;
    vec3 diffuse = 0.5 * color;
    // specular
    vec3 view_direction = normalize(TangentViewPos - TangentPoint);
    vec3 reflect_direction = reflect(-light_direction, normal);
    vec3 halfway_direction = normalize(light_direction + view_direction);
    float specular = pow(max(dot(normal, halfway_direction), 0.0), 32.0) * 0.2;

    return (ambient + diffuse * angle + specular * angle);
}
```

Figur 39

Normalene benytter de samme formelene som for ambient, diffuse og specular, men her gjøres alt i tangent space.

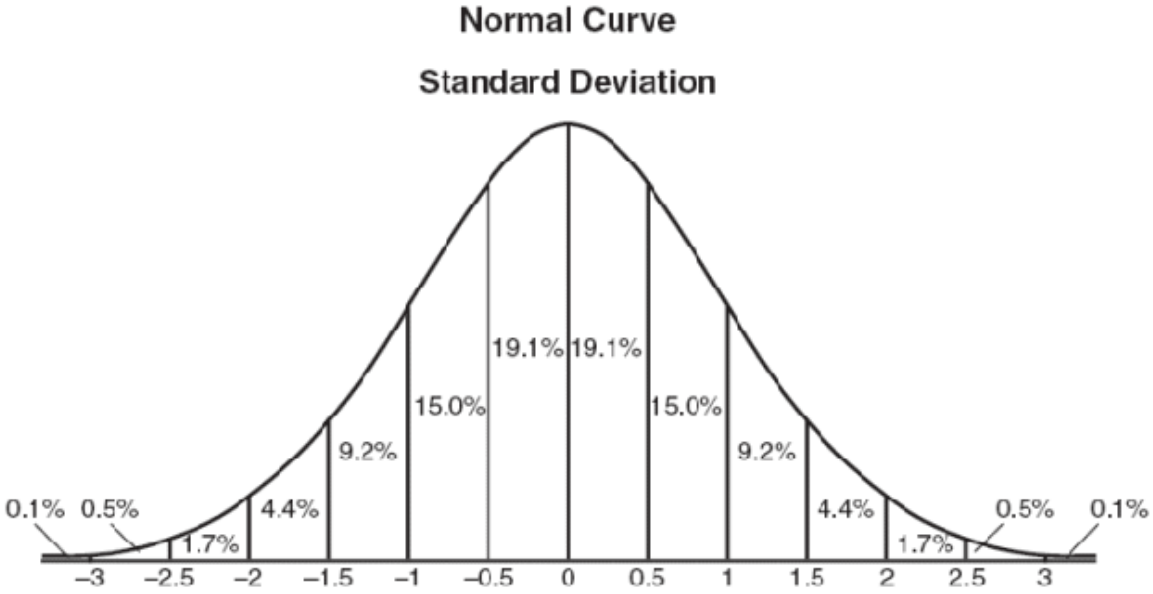
Bloom

Det ble implementert en lyseffekt kalt “bloom”, som gir en glødende effekt på farger som er sterke i verdenen. Måten det blir gjort på, er at fragment shaderen har to utganger for farge. Den ene utgangen ble brukt til å tegne verdenen som normalt med lyskalkulasjoner. Den andre utgangen sendte kun ut fargen som er kalkulert, om rgb-verdien til denne fargen var over en viss grense. Hvis den ikke var det, ville det bli sendt svart ut på denne utgangen. Det ble da laget to “versjoner” av verdenen, en vanlig, og en svart verden med noen skarpe farger. Disse bildene ble lagret i to forskjellige “color attachments” i en “framebuffer”.

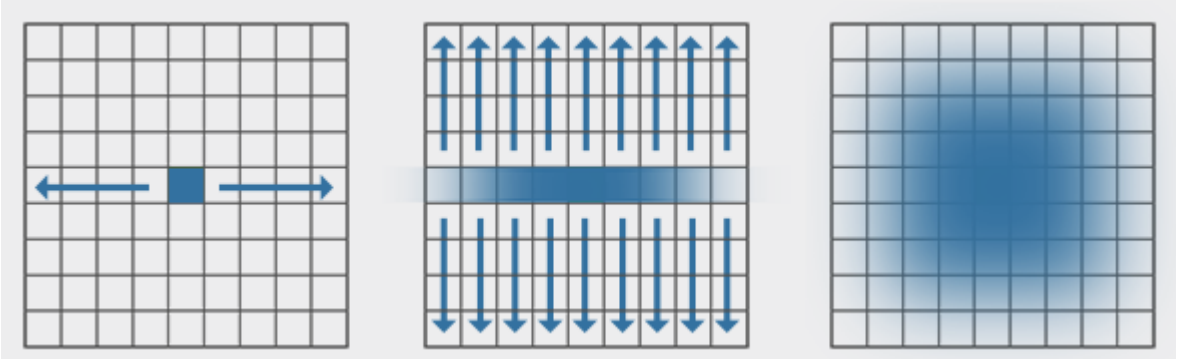


Figur 40

For å kunne utføre bloom effekten, måtte det utføres en type blur effekt på bildet med de skarpe fargene. Gaussian blur[13] ble brukt, som er basert på Gaussian kurven[14]. Gaussian blur metoden angir vekter på “samples” rundt et fragment (vektene utgjør en “Gaussian kernel”), og utfører horisontal og vertikal “blurring” annenhver gang basert på verdien av en boolean. “Samples” nærmest midten (nærmest fragmentet) blir vektet mest, mens “samples” lenger unna fragmentet blir vektet mindre.



Figur 41



Figur 42

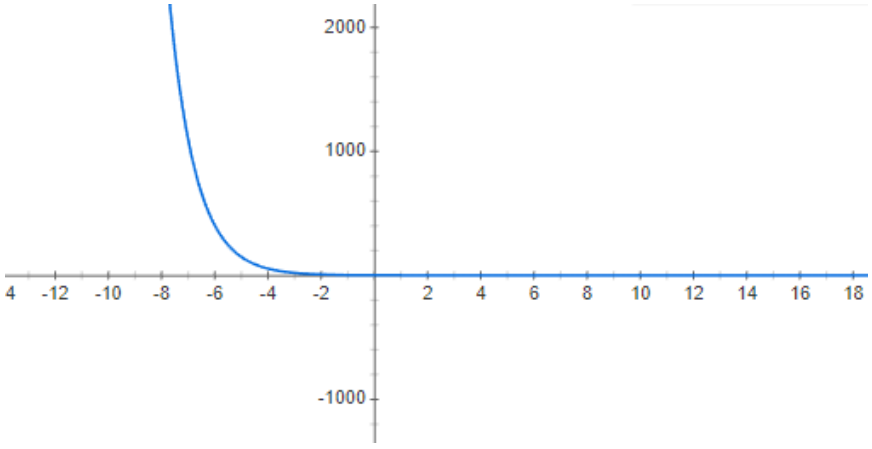
Etter dette, vil bildet med blur effekten bli lagt til i bildet av den vanlige verdien, som sammen vil gi bloom effekten. Dette skjer i en egen shader.

Her blir det vanlige bildet og bildet med påført blur lagt sammen:

```
if (bloom) {
    // Add the original scene and the bloom scene together
    hdrColor += bloomColor;
}

// This function will make the result vector vary from 0 to 1 in all rgb colors, depending on the value of exposure.
// The exposure value is controlled by the "Q" and "E" keys. By using a really high exposure value, the rgb colors will approach 1
// (totally white color, maximum bloom). The lowest exposure value allowed is 0, which will make the rgb colors equal to 0, which is total black.
vec3 result = vec3(1.0) - exp(-hdrColor * exposure);
```

Figur 43

"Exposure" variabelen kan endres fra tastaturet. Blir "exposure" variabelen veldig høy, vil e^(-hdrColor * exposure) leddet bli veldig lite, og "result" vektoren vil bli tilnærmet [1,1,1] (helt hvitt). Derimot, hvis exposure er veldig liten (for eksempel 0), vil "result" bli [0,0,0], som er helt svart.



Figur 44

Volumetriske skyer

Å generere og tegne skyer er ingen enkel oppgave. Det kan fort bli veldig kompleks og kostbart for prosesseringen. Det var et stort ønske om å implementere dette, og ettersom dette er en såpass krevende oppgave ble løsningen basert på masteroppgaven *Real-Time Rendering of Volumetric Clouds*[15] av Rikard Olajos ved Lund University, utgitt 4. oktober 2016.

Løsningen baserer seg på å genere skyer fra en 3D støy tekstur (noise) og benytte *ray traversering* (ray marching) og *ray scattering* til å tegne skyene med realistisk lys. Omgivelsene blir tegnet før skyene og lagret til en Framebuffer. Deretter blir skyene tegnet over dette.

Det er ønskelig å understreke at løsningen tilhører Rikard Olajos og er kun gjenskapt og implementert i dette prosjektet. Følgende filer fra prosjektet vedlagt er basert på Rikard Olajos sin løsning:

cloud_frag.shader.
CloudTexture.h og *CloudTexture.cpp*.
createTexture3DFromEX5()-funksjonen fra *Texture.cpp*.
noise5.txt filen fra tekstur-mappen.

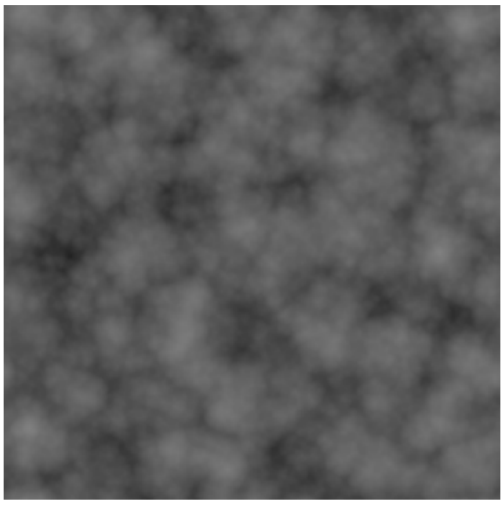
3D Tekstur

I løsningen benyttes to 3D teksturer. 3D sky-teksturen benyttet i løsningen er generert med bruk av Fractional Brownian Motion til å legge lag av flere Cellular Noise støy-teksturer over seg selv med forskjellige frekvenser. Slik vil det se ut som røyk eller skyer. Rikard Olajos har skrevet sin egen støy genererings algoritme *noisegen*[16] som gjør nettopp dette. Det tok vesentlig lang tid (flere minutter) å generere denne teksturen, så dette ble gjort en gang og lagret til filen *noise5.ex5*. Filen blir konvertert i programmet i Tekstur-klassen.

```
/* Read header */
fscanf(fp, "%d", &t->width);
fscanf(fp, "%d", &t->height);
fscanf(fp, "%d", &t->depth);

/* Create image */
std::vector<uint8_t> image;
image.reserve(t->width * t->height * t->depth * 4);

/* Read image*/
uint32_t pixel;
while (fscanf(fp, "%d", &pixel) == 1) {
    image.push_back(pixel >> 24);
    image.push_back(pixel >> 16);
    image.push_back(pixel >> 8);
    image.push_back(pixel >> 0);
}

glGenTextures(1, &t->id);
glBindTexture(GL_TEXTURE_3D, t->id);

glTexImage3D(GL_TEXTURE_3D, 0, GL_RGBA, t->width, t->height, t->depth, 0, GL_RGBA, GL_UNSIGNED_BYTE, image.data());
glGenerateMipmap(GL_TEXTURE_3D);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_REPEAT);
```

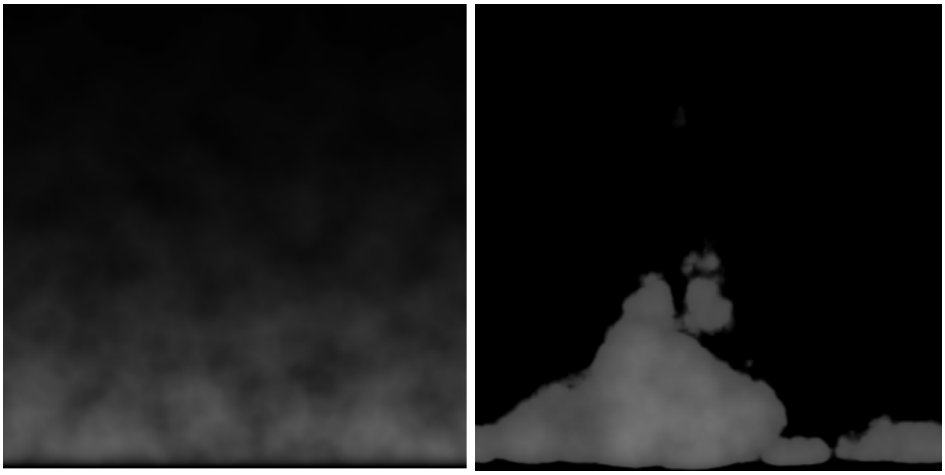
Figur 45

Slik ser teksturen ut når konvertert til 2D:



Figur 46

Denne teksturen alene er ikke nok til å lage skyer. Det finnes utrolig mange forskjellige typer skyer, men denne løsningen tar kun for seg den mest vanlige formen. Denne formen distribuerer meste av skyen mot bunnen og mindre mot toppen. Formelen benyttet for å regne ut probiliteten for hvor skyene skal være basert på høyde: $HD(h) = (1 - e^{-50h}) \cdot e^{-4h}$, hvor h er høyden.

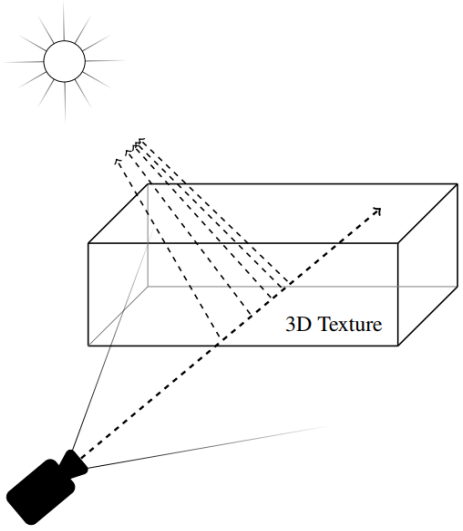


Figur 47

Teksturen over ganges med sky-teksturen og noen av de svake verdiene blir fjernet. Resultatet til høyre er et eksempel på hvordan den endelige teksturen ser ut.

Volumetrisk ray travesering

For å tegne 3D teksturen mer realistisk benyttes ray traversering til å kalkulere hvor mye lys som rekker punktene i skyene. En stråle blir sendt fra kameraet for hver piksel på skjermen. Hvis strålen treffer et punkt A i skyen, blir en ny stråle sendt mot solen og slik kalkuleres hvor mye lys som rekker punkt A.



Figur 48 fra master thesis oppg. s16

Første gangen et punkt i skyen treffes av strålen fra kameraet, lagres denne til en float verdi som sier noe om gjennomsiktigheten til pikselen. Hvis verdien er 0.0, vises alt som er bak skyen. Vis verdien er 1.0 eller høyere vises ikke det bak. Når verdien blir 1.0 eller høyere stoppes traverseringen ettersom alt videre ikke vil vises gjennom skyen uansett. Ray

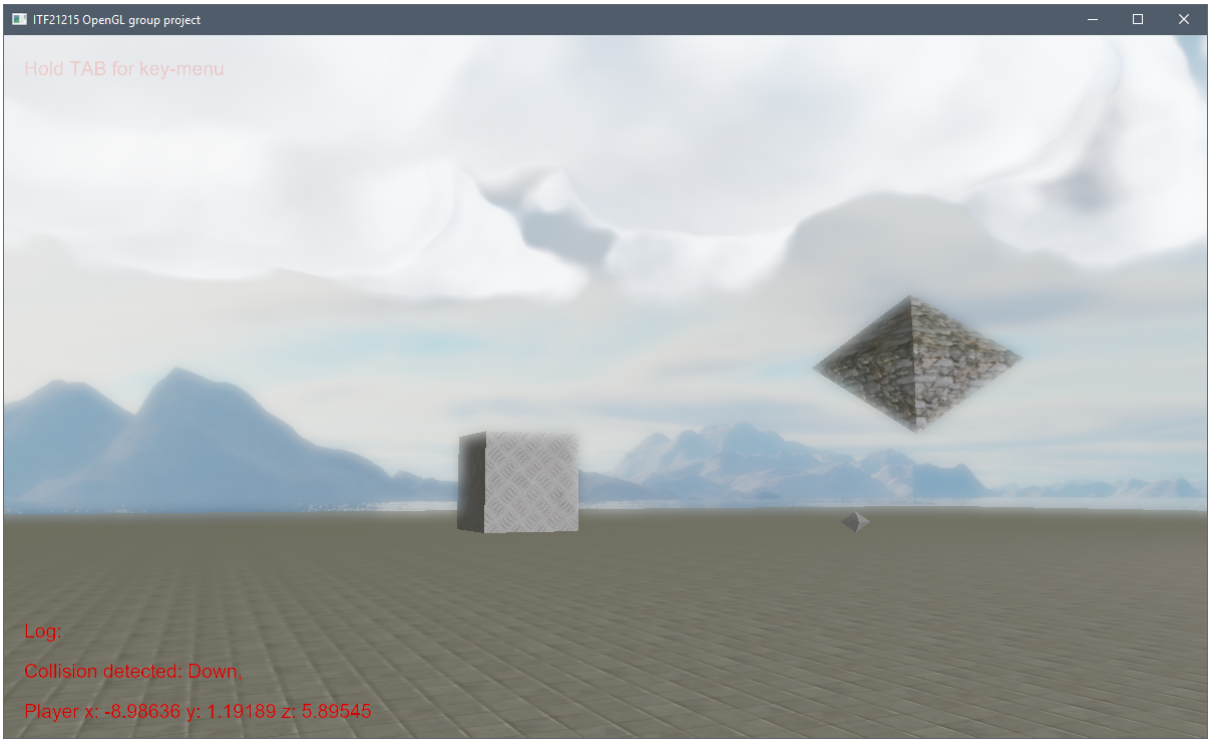
traverseringen begrenses slik at den ikke fortsetter i all evighet. Denne grensen kan endres ved å benytte 1 og 2 på tastaturet, som øker distansen skyene tegnes.

Lys i skyen blir tegnet med bruk av *light scattering*. Dette kan fort bli kostbart så en enklere, mindre ressurskrevende metode er å benyttet: Henyey-Greenstein fase funksjon. Denne finner mengden av spredt lys for en bestemt vinkel og konsentrasjon av lysspredning, g. Formelen ser slikt ut:

$$HG(\theta) = \frac{1-g^2}{4\pi(1+g^2 cos\theta)^{3/2}}$$

Konklusjon

Produktet av prosjektet er en 3D visualisert omgivelse som inkluderer flere forskjellige effekter.



Figur 49

Følgende effekter er produsert for prosjektet:

- Collision Detection
- Tyngdekraft
- 3D objekt generatorer for trekant, rektangel, kube, kule og diamant.
- Textur-klasse og Material-klasse som laster inn og aktiverer flere teksturer.
- Tekst-klasse som tegner tekst på skjermen.
- Vektor- og Matrise matematikk-klasser
- Cubemap (skybox)
- Normal mapping
- Specular lighting
- Bloom
- Volumetric Clouds

Skyene legger seg ikke bak objektene. Det var ikke nok tid til å fikse dette.

Terminologi

Glyphs	Et tekst symbol fra en text spritesheet.
Hitbox	Et avgrenset volum til et object. Brukes for kollisjonstesting
Interacte	Å samhandle med noe
Spritesheet	Flere textures delt inn i et bilde. Brukes med et koordinatsystem som peker til hvilke texture som skal lastes. Brukes vanligvis for animasjon med 2d spill.
Realtime Rendering	Simulasjon av virtuelle omgivelser gjort i sanntid.
Tekstur	Et bilde av flaten til et objekt eller detalj.
Column-major matrise	Matriser som har elementer kolonnevis.
Row-major matrise	Matriser som har elementer rekkevis.
Scale matrise	Matriser som endrer lengden til vektorer.
Rotation matrise	Matriser som roterer vektorer rundt en gitt akse med en gitt vinkel.
Translation matrise	Matriser som flytter vektorer i x, y og z retning.
Model matrise	En matrise man får etter å gange sammen scale, rotation og translation matrisene.
Projection matrise	Matriser som definerer synsvinkel, startdybde og sluttdybde (den synlige dybden).
View matrise	Matriser som plasser verden i forhold brukerens perspektiv (verdens kamera).
Near plane	Det planet som vi ser fra. Dette planet er helt nært “øyet”. Objekter som ligger bak øyet blir derfor ikke tegnet på skjermen.
Far plane	Det planet som definerer hvor langt vi klarer å se. Objekter lenger unna enn dette planet blir ikke tegnet på skjermen.

Referanser

Oppdatert:

[1] OpenGl Mathematics library (GLM) Hentet 08.12.17 fra https://glm.g-truc.net/0.9.8/index.html

[2] Wikipedia: DirectX Hentet 08.12.17 fra https://no.wikipedia.org/wiki/DirectX

[3] StackOverflow. *Drawing Sphere in OpenGL without using gluSphere()?* Hentet 08.12.17 fra https://stackoverflow.com/questions/7687148/drawing-sphere-in-opengl-without-using-glusphere

[4] Freetype: program for å rendre text Hentet 08.12.17 fra https://www.freetype.org/

[5] LearnOpenGL: Textrendreing in practice Hentet 08.12.17 fra https://learnopengl.com/#!In-Practice/Text-Rendering

[6] Wikipedia: Quadtree Hentet 08.12.17 fra https://en.wikipedia.org/wiki/Quadtree

[7] Wikipedia: Octree Hentet 08.12.17 fra https://en.wikipedia.org/wiki/Octree

[8] Wikipedia: Binary Space Prtitoning (BSP) Hentet 08.12.17 fra https://en.wikipedia.org/wiki/Binary_space_partitioning

[9] Sweep-and-prune by Pierre Terdiman (11.09.2007) Hentet 08.12.17 fra http://www.codercorner.com/SAP.pdf

[10] Improved collision detection and response by Kasper Fauerby (25.07.2003) Hentet 08.12.17 fra http://www.peroxide.dk/papers/collision/collision.pdf

[11] Wikipedia: Real time rendering Hentet 08.12.17 fra https://en.wikipedia.org/wiki/Real-time_computer_graphics

[12] Txutxi: Tangent Space Matrix (TBN) Hentet 08.12.17 fra http://www.txutxi.com/?p=316

[13] Wikipedia: Gaussian blur Hentet 08.12.17 fra https://en.wikipedia.org/wiki/Gaussian_blur

[14] Wikipedia Gaussian Function Hentet 08.12.17 fra https://en.wikipedia.org/wiki/Gaussian_function

[15] Rikard Olajos (October 4, 2016). *Real-Time Rendering of Volumetric Clouds*. Hentet 01.12.17 fra http://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8893256&fileOId=8893258

[16] Rikard Olajos (14.04.16). *noisegen*. Github. Hentet 01.12.17 fra https://github.com/rikardolajos/noisegen

[17] Sean Barrett. *STB*. Hentet 12.10.17 fra https://github.com/nothings/stb

[18] Visual Studio Community 2017. Hentet 08.12.17 fra https://www.visualstudio.com/downloads/

Figurer

[17] Spritesheet med text glyphs. Laget av Vegard Strand.

[18] Tekst rendering gjort i java med lwjgl. Laget av Vegard Strand

[19] Text Rendering i C++ med freetype fra prosjektet.

[20] Tegning av en hitbox. Laget av Vegard Strand.

[21] Illustrasjon av logikk av to hitboxer. Laget av Vegard Strand.

[22] Programkode fra object_frag.shaderen fra prosjektet

[23] Programkode fra object_frag.shaderen fra prosjektet.

[24] Programkode fra object_frag.shaderen fra prosjektet.

[25] Programkode fra object_frag.shaderen fra prosjektet.

[26] dactilardesign fra FilterForge. 10744-diffuse.jpg fra prosjektet. Hentet fra https://www.filterforge.com/filters/10744.html.

[27] Programkode fra object_frag.shaderen fra prosjektet.

[28] Programkode fra object_frag.shaderen fra prosjektet.

[29] Figur laget av LearnOpenGL. Diffuse. Hentet 07.12.17 fra https://learnopengl.com/#!Lighting/Basic-Lighting

[30] Programkode fra object_frag.shaderen fra prosjektet.

[31] Figur laget av LearnOpenGL. Specular. Hentet 07.12.17 fra https://learnopengl.com/#!Lighting/Basic-Lighting

[32] Programkode fra object_frag.shaderen fra prosjektet.

[33] dactilardesign fra FilterForge. 10744-reflectiveocclusion.jpg fra prosjektet. Hentet fra https://www.filterforge.com/filters/10744.html.

[34] Programkode fra object_frag.shaderen fra prosjektet.

[35] dactilardesign fra FilterForge. 10744-normal.jpg fra prosjektet. Hentet fra https://www.filterforge.com/filters/10744.html.

[36] Bilde som illustrerer en normal på en sirkel. Hentet 08.12.17 fra http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/

[37] Bilde som illustrerer normal, tangent og bitangent på en sirkel. Hentet 08.12.17 fra http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/

[38] Programkode fra object_vert.shaderen fra prosjektet.

[39] Programkode fra object_frag.shaderen fra prosjektet.

[40] Illustrasjon av det vanlige bildet, og det bildet med kun de skarpe fargene. Hentet 03.12.17 fra https://learnopengl.com/#!Advanced-Lighting/Bloom

[41] Illustrasjon av Gaussian kurven, som beskriver størrelsen på vektene jo lenger man er unna senter av kurven. Hentet 03.12.17 fra https://www.pixelstech.net/article/1353768112-Gaussian-Blur-Algorithm

[42] Her blir et fragment gjort blur på horisontalt og vertikalt. Hentet 03.12.17 fra https://learnopengl.com/#!Advanced-Lighting/Bloom

[43] Kodesnippet fra bloom_frag.shaderen fra prosjektet.

[44] Illustrasjon av funksjonen som bestemmer hvor sterk bloom effekten skal være. Hentet 07.12.17 fra www.google.com (funksjonsplot av å google funksjonen e^-x).

[45] Programkode fra Texture.cpp fra prosjektet.

[46] Bilde av Rikard Olajos. Viser hvordan sky-teksturen ser ut i 2D. Fra *Real-Time Rendering of Volumetric Clouds*. Se refereanse [15].

[47] Bilde av Rikard Olajos. Viser hvordan sky-struktur-teksturen ser ut i 2D. Fra *Real-Time Rendering of Volumetric Clouds*. Se refereanse [15].

[48] Figur av Rikard Olajos. Viser hvordan ray travesering foregår med en 3D tekstur. Fra *Real-Time Rendering of Volumetric Clouds*. Se refereanse [15].

[49] Bilde av 3D omgivelsene produsert av prosjektdeltakerne, med bloom effekt og skyer.

[50] Programkode fra Texture.cpp fra prosjektet.

[51] Programkode fra Material.h fra prosjektet.

Vedlegg

Vedlegg 1 Zippet mappe av prosjektet. *ITF21215_Gruppeprosjekt_Prosjektfiler.zip*

Vedlegg 2 Installasjonsveiledning. *README.txt*.

Vedlegg 3 Prosjektkontrakt. *ITF21215_Innleveringskontrakt.pdf*

Vedlegg 4 Aktivitetskart. *ITF21215_Gruppeprosjekt_Prosjektbeskrivelser_Aktivitetskartet.pdf*

Vedlegg 5 Arbeidslogg. *ITF21215_Gruppeprosjekt_Prosjektbeskrivelser_Logg.pdf*