

INGMAR PRESENTS

OPENGL FOR OUTLAWS II

FROM SCRATCH TO SKETCH



© 2018

Ingmar Kroon
Minor ASE semester 3 2018

Minor begeleider: Wouter Brinksma

INTRODUCTIE

Dit document is bedoeld voor geïnteresseerden die meer willen weten over low level computer graphics.
In deze reader gebruiken we OpenGL; een open source cross platform graphics API.

In deze reader wordt er uitgebreid aandacht besteed aan de benodigde concepten om te kunnen renderen in 3D, inclusief de wiskunde.

In deze cursus gaan we zelf programma's schrijven in C++.

C++ is een taal die goed aansluit op OpenGL omdat C++ ook cross platform en low level is.

Voordat er geprogrammeerd kan worden moet de omgeving voorbereid worden, dit wordt stap-voor-stap behandeld in deze reader.

In deze cursus zal worden gewerkt met Visual Studio 2017 in Windows 10.

In het begin besteden we vooral aandacht aan de OpenGL API.

Hoe rendert OpenGL een punt, een driehoek of een vierkant?

Nadat we kennis hebben gemaakt met de API worden de fundamentele concepten uitgelegd van graphics zoals lineaire algebra.

Zodra we snappen hoe de OpenGL API werkt én hoe graphics werken, gaan we de kennis combineren door zelf stap-voor-stap een 3D renderer bouwen.

De bedoeling is dat er zo weinig mogelijk 'magie' wordt gebruikt.

Alle complete code voorbeelden uit deze reader zijn terug te vinden op github:

https://github.com/ingmar12345/OPENGL_FOR_OUTLAWS/

De grootste uitdaging tijdens het maken van deze reader was om de inhoud zo duidelijk mogelijk over te brengen, daarom zijn er tientallen afbeeldingen / 3D visualisaties om de stof helder te maken.

Ik hoop dat dit document een duidelijke basis legt om te beginnen met graphics programmeren.

INHOUD

Introductie	2
Inhoud.....	3
1. Over OpenGL	6
Wat is OpenGL?	6
Hardware	7
Pipeline	8
Vertices.....	9
Vertex Shader	9
Primitives genereren	10
Rasteren.....	10
Fragment Shader	11
Framebuffer operaties.....	11
2. Omgeving opzetten	12
Gereedschappen voorbereiden.....	12
Je eerste programma opzetten	14
3. OpenGL.....	19
Shaders	19
Hoe werken shaders?	19
Vertex Array Object.....	20
Vertex shader	21
Fragment shader.....	22
Voorbeeld code shaders	23
Vertex renderen	24
Simpele driehoek.....	27
Geïndexeerd renderen	30
Meerdere vertex attributen	32
Uniforms	34
4. Mathematiek	36
Vector	37
Matrix	39
Transformaties.....	40
Eenheidsmatrix.....	40
Translatiematrix.....	40
Schalingsmatrix.....	40

Rotatiematrix.....	41
Matrices in OpenGL.....	42
Volgorde van transformaties.....	43
Quaternionen	46
Complexe getallen.....	46
Quaternionen vermenigvuldigen	47
Vector roteren met behulp van rotatie quaternion (Sandwich product).....	48
Rotatie quaternionen combineren.....	50
Rotatie quaternion omzetten naar rotatie matrix	50
Euler hoeken omzetten naar quaternion	51
Coördinatenstelsels	53
Representatie coördinatenstelsel	54
Model-view transformatie.....	54
Projectie transformatie	58
Viewport transformatie.....	61
5. Renderen in 3D	62
3D bestanden	63
OBJ bestand	63
OBJ parser.....	65
Shaders	66
Belichtingscomponenten.....	66
Phong shading	70
Gourad shading	71
Flat shading.....	72
Model	75
Overzicht.....	75
Instantie variabelen.....	76
Belangrijkste implementaties.....	76
Camera	79
Basis klasse camera	80
Statische camera	82
Model camera.....	83
RC camera (remote control).....	85
Belichtingscomponenten.....	87
Overzicht.....	87
Koppelfunctie	87
Shadercomponenten.....	88

Overzicht.....	88
Instantie variabelen.....	88
Implementatie	89
Quaternion	91
Main class	92
Overzicht.....	92
Instantie variabelen.....	93
Belangrijkste implementaties.....	94
6. Externe bronnen.....	96

1. OVER OPENGL

WAT IS OPENGL?

OpenGL is een cross platform graphics API voor rendering op de GPU (Graphics Processing Unit).

OpenGL is een abstractie laag tussen de applicatie (bijvoorbeeld een game) en het graphics systeem wat afgestemd is op de specifieke hardware (GPU).

Deze abstractie laag zorgt er voor dat de applicaties niet afhankelijk zijn van een bepaald type GPU.

OpenGL is low level, maar ook weer niet.

De OpenGL laag maakt geen verschillen tussen de verschillende merken hardware, maar toch is OpenGL het wel zo low level gemaakt dat je de hardware optimaal kan benutten.

Een bestaande engine zoals Unity maakt onderwater gebruik van een graphics API zoals OpenGL.

Unity is zo gemaakt dat men vrij gemakkelijk en snel een 3D applicatie kan ontwikkelen, maar Unity geeft geen optimale controle over de graphics en hardware.

Wat OpenGL ziet zijn driehoekjes, lijntjes en puntjes, geen complete meshes (3D vormen).

Veel operaties in OpenGL zijn dan ook per pixel (fragment), per vertex of per polygoon.

OpenGL regelt niet het weergeven van een venster en het handelen van user input omdat OpenGL crossplatform is.

Elk operating system beheert op haar eigen manier hoe vensters worden weergegeven en hoe bijvoorbeeld een toetsenbord wordt geïnterpreteerd.

Tijdens het coderen van OpenGL code is het mogelijk om de output in een venster weer te geven.

Hier gebruiken we een windowing library voor.

Tijdens de installatie in Chapter 2 zal hier op ingegaan worden.

HARDWARE

OpenGL is zo ontworpen dat het optimaal de kracht kan benutten van een GPU; maar wat is het voordeel van een GPU precies?

GPU staat voor Graphics Processing Unit. Een GPU heeft veel meer cores dan een CPU.

Deze cores zijn niet hetzelfde als CPU cores.

GPU cores zijn zeer efficiënt als het gaat om het uitvoeren van berekeningen in gigantische hoeveelheden data, zoals operaties uitvoeren per vertex, per pixel of per polygoon (we spreken over miljoenen).

In een GPU zitten vaak gigabytes aan videogeheugen; geheugen dat de grafische chip heel snel kan raadplegen waar bijvoorbeeld grote bestanden als textures in zitten opgeslagen.

Video geheugen is minder snel, maar het is beschikbaar voor alle cores.

Wat voor de CPU het RAM of werkgeheugen is, is voor de GPU het VRAM of videogeheugen.

Een GPU wordt niet alleen gebruikt voor graphics; maar kan ook gebruikt worden voor physics berekeningen en het trainen van neurale netwerken.

Nu denk je, is een CPU dan niet overbodig?

Nee, want GPU cores zijn veel simplistischer dan CPU cores en ze beschikken niet over geavanceerde features zoals out-of-order execution en branch prediction.,

De cache in een GPU is veel kleiner dan de cache in een CPU.

Cache geheugen is core specifiek (elke core heeft haar eigen cache).

Voorbeeld:

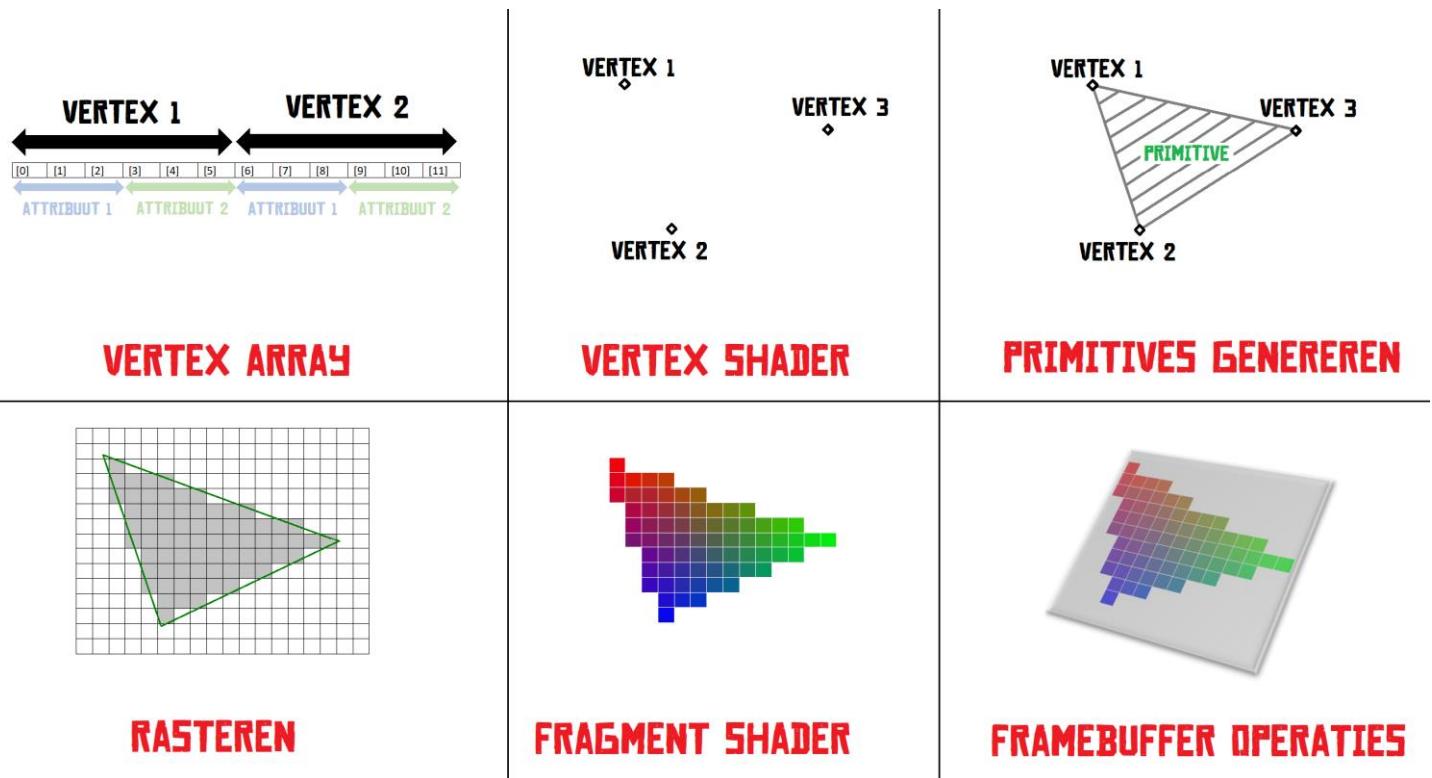
1 grote complexe berekening wordt efficiënter uitgevoerd op een CPU (meer cache, minder cores).

Een groot aantal kleine berekeningen worden sneller uitgevoerd op de GPU (minder cache, meer cores).

Later in deze cursus zal worden behandeld welke OpenGL logica interactie heeft met de GPU.

PIPELINE

OpenGL kan je zien als een soort van fabriek met verschillende fasen.
Sommige fasen worden onderwater uitgevoerd, andere fasen zijn programmeerbaar.
De volgende figuur toont een versimpelde graphics pipeline van OpenGL.



Wat zijn shaders?

Een shader is een klein programmaatje wat parallel uitgevoerd wordt op de GPU.

Wat een shader doet is input omzetten in output.

Een shader is een geïsoleerd programma.

De output gaat weer verder naar de volgende fase in de pipeline.

In OpenGL worden shaders geschreven in GLSL (OpenGL Shading Language).

GLSL is een soort van C achtige taal.

In hoofdstuk 3 gaan we zelf een shader programmeren.

Vertices

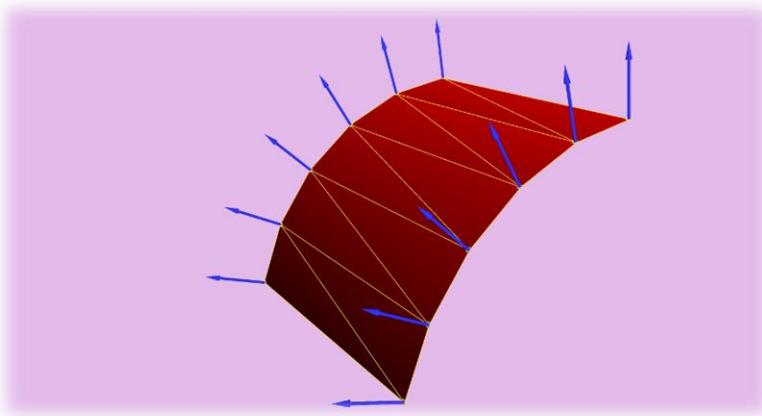
Een vertex (enkelvoud van vertices) is een punt in 3D met een X, Y en Z-coördinaat.

Een vertex kan behalve een 3D positie meerdere attributen bevatten.

Voorbeelden van andere vertex attributen zijn vertex normaal vectoren en texture coördinaten.

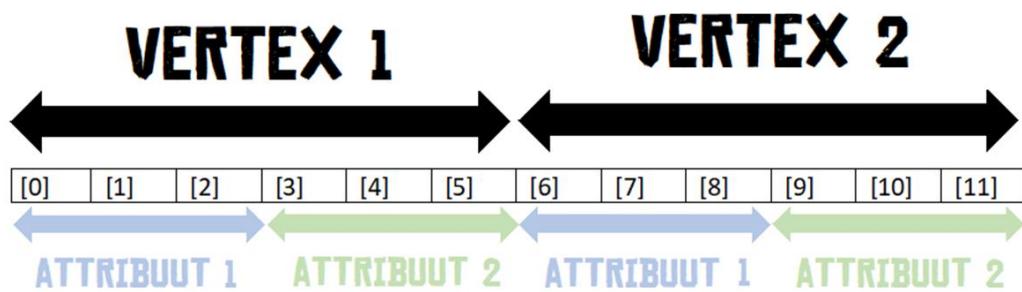
Vertex normaal vectoren spelen een rol bij belichting.

- Vertex posities zijn de hoekpunten van elk polygoon (driehoek met gele contouren)
- Vertex normaal vectoren zijn de blauwe pijlen die haaks op elke vertex (positie) staan



De vertex attributen zijn opgeslagen in een vertex array.

Meerdere attributen kunnen in dezelfde array worden opgeslagen.



Vertex Shader

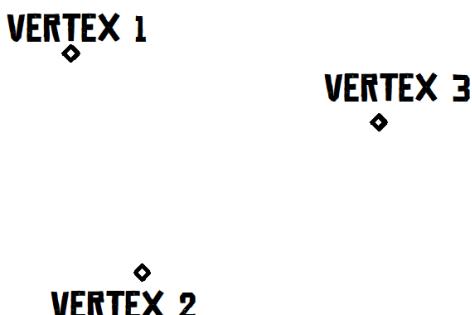
De vertex shader is de eerste programmeerbare fase in de OpenGL pipeline.

De vertex shader manipuleert elke individuele vertex uit de vertex stream (de vertex array).

Vertex attribuut pointers specificeren het format van de vertex attributen in een array.

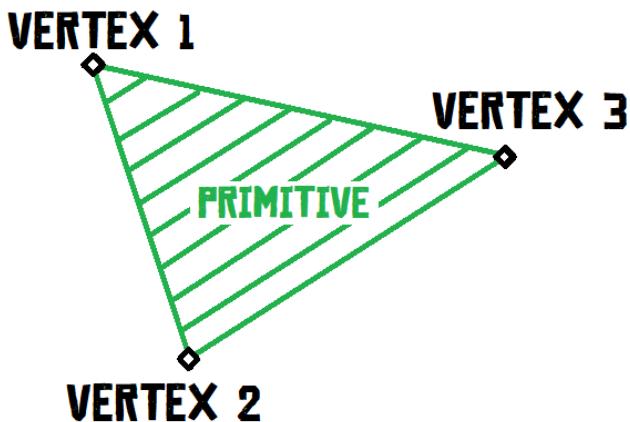
Typische **per-vertex** operaties zijn:

- Vertex posities transformeren van een lokaal coördinatenstelsel (model space) naar een geprojecteerd coördinatenstelsel (clip space).
De output gaat naar de primitive generation en rasterization.
- Vertex attributen die bijdragen aan de pixel kleur (zoals een normal) doorsturen naar de fragment shader. Indien nodig worden deze attributen eerst gemanipuleerd voordat ze ge-output worden.



Primitives genereren

De output van de vertex shader (vertex positions in clip space) gaat naar de primitive generation.
Primitive generation is het proces waarbij driehoeken, lijnen of punten gegenereerd worden uit vertices.
Primitives worden in de volksmond ook wel polygonen genoemd.
De volgorde van de vertices is erg belangrijk, de volgorde bepaalt namelijk welke vertices een primitive vormen.



Het generen van primitives wordt door OpenGL onderwater uitgevoerd.

Er is ook een optionele shader die het mogelijk maakt om operaties uit te voeren **per-primitive**, deze zogenoemde *geometry shader* bevindt zich tussen de primitive generation en de rasterization.
Later zullen we een praktijk voorbeeld uitleggen waarbij deze extra shader goed van pas komt.

Rasteren

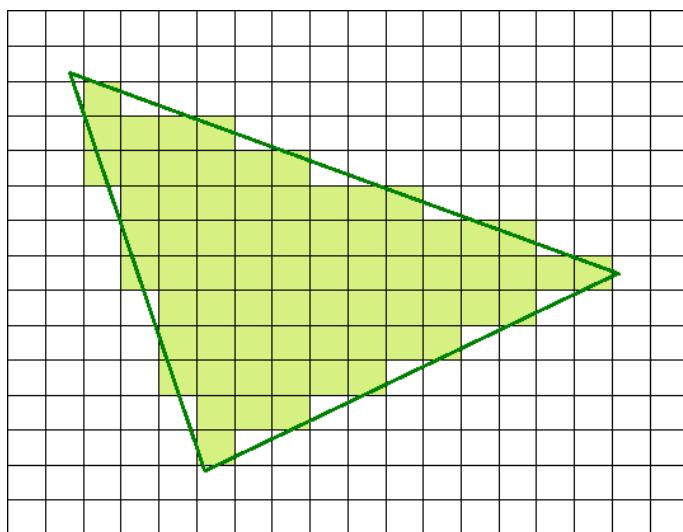
Als eenmaal de primitives bepaald zijn kunnen de primitives gerasterd worden naar pixels.

Rasterization test welke pixels een primitive (punt, lijn of driehoek) bedekken.

Een gerasteriseerd primitive (vlak) bestaat uit zogeheten fragmenten, zie Figuur 2.

Een fragment is een soort tijdelijke ‘pixel’, deze pixels kunnen, afhankelijk van framebuffer operaties, wel of niet worden weergegeven op het scherm.

Het rasteren wordt ook onderwater parallel uitgevoerd door OpenGL.



Fragment Shader

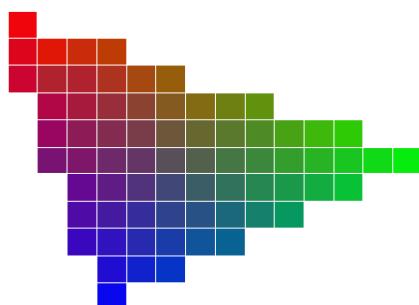
De fragment shader (in de volksmond ook wel “pixel shader” genoemd) is verantwoordelijk voor de kleur van elke pixel (Figuur 3).

Nadat er een primitive (zoals een driehoek) gerastered is, wordt er een lijst van fragmenten naar de fragment shader gestuurd om daar vervolgens gekleurd te worden.

In de fragment shader kan op basis van bijvoorbeeld een belichtingsmodel, de fragmentkleur berekenen. Een driehoek kan meerdere fragmenten produceren, daarom wordt de fragment shader ook **per-fragment** parallel uitgevoerd op de GPU.

Vertex attributen die kunnen bijdragen aan de kleur worden vanuit de vertex shader ge-output en vervolgens ge-input in de fragment shader.

De fragment shader kan zelf geprogrammeerd worden, net als de vertex shader.



Framebuffer operaties

De framebuffer operaties zijn de laatste stappen in de graphics pipeline.

De framebuffer operaties bepalen welke fragmenten worden omgezet in voor de gebruiker zichtbare pixels.

De framebuffer bestaat uit verschillende componenten, waaronder een depth buffer en een color buffer.

De depth buffer beschrijft de diepte van elk fragment (fragment is soort van ‘3 dimensionale pixel’)

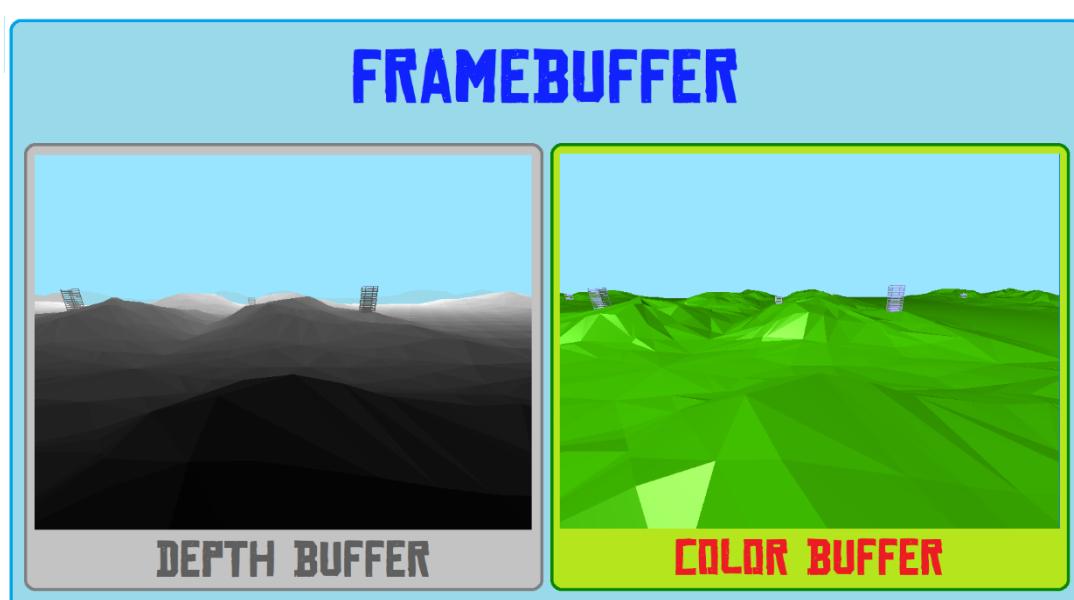
De color buffer beschrijft de kleur van een fragment, deze fragment kleur is bepaald in de fragment shader.

De *scissor test* is een test die bepaalt of fragmenten zich binnen het scherm bevinden, als een fragment zich buiten het scherm bevindt wordt het fragment ‘weggeknippt’.

Na de *scissor test* worden de *depth test* uitgevoerd.

De *depth test* bepaalt op basis van de depth- of Z-buffer welke fragmenten elkaar overlappen zodat de niet zichtbare fragmenten kunnen worden weggegooit.

Bij elke render iteratie moeten de buffers worden opgeschoond om het volgende frame ruimte te geven.



2. OMGEVING OPZETTEN

GEREEDSCHAPPEN VOORBEREIDEN

CMake installeren

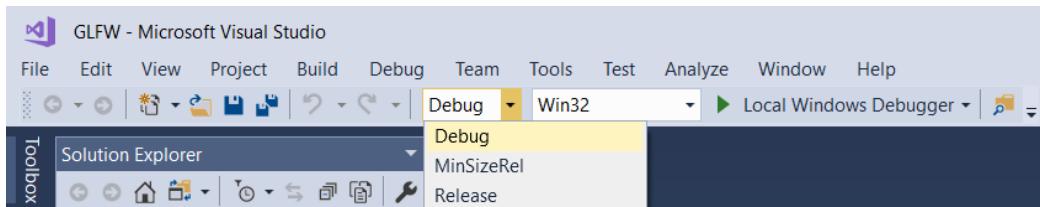
1. Download het zip bestand van <https://github.com/openglsuperbible/sb7code/archive/master.zip>
2. Verplaats het zip bestand naar een passende locatie en unzip het bestand.
3. Ga naar <http://www.cmake.org/> en download de nieuwste versie van de installer voor Windows.
De installer is, tenzij anders aangegeven, te vinden in de download folder.

GLFW builden

OpenGL is cross platform, dus het weergeven van vensters en het regelen van invoer wordt niet door OpenGL zelf geregeld.

Daar is een aparte library voor nodig, in dit geval gebruiken we GLFW.

4. Open de installatie Wizard van CMake.
Vink het hokje aan dat cmake in het pad zit.
Kies vervolgens een passende locatie (zoals `C:/Program Files/CMake/`)
Voer de installatie uit.
5. Open opdrachtprompt en ga naar de volgende map (afhankelijk van de locatie van sb7code-master)
`CD <<locatie>>/sb7code-master/extern/glfw-3.0.4`
6. Wanneer je in de juiste map zit voer het volgende in:
`cmake -G "Visual Studio 15" .` (Visual Studio 15 and 17).
7. Ga naar de map `sb7code-master/extern/glfw-3.0.4` in verkenner en open `GLFW.sln` in Visual Studio.
8. Build de release en debug configuraties (deze zijn te vinden onder het dropdown menu).



9. Ga naar de map `glfw-3.0.4/src/Debug/` en kopieer `glfw3.lib`.
10. Plak het `glfw3.lib` bestand in de map `sb7code-master/lib` en hernoem het naar `glfw3_d.lib`.
11. Ga naar de map `glfw-3.0.4/src/Release/` en kopieer `glfw3.lib`.
12. Plak het `glfw3.lib` bestand in de map `sb7code-master/lib` zonder te hernoemen.

sb7 builden met demo programma's

sb7 kunnen we ook bouwen met cmake, sb7 bevat een headerfile (`sb7.h`) met de functies die we kunnen overerven, zoals een render loop.

Daarnaast worden er ook een aantal voorbereide demo's gebouwd.

Uiteraard gaan we in deze cursus ons eigen programma maken.

1. Open opdrachtprompt en voer het volgende in:
`CD <<<localpath>>>/sb7code-master/build`
2. Wanneer je in de bovenstaande map zit voer het volgende in:
`cmake -G "Visual Studio 15" ..`
3. Bouw de release en debug configuraties in `superbible7.sln` in Visual Studio (dit bestand is te vinden in `sb7code-master/build`)
4. Download het media pakket van openglsuperbible.com/files/superbible7-media.zip
Plaats het media mapje in `sb7code-master/bin/media` (alleen de subfolders van superbible7-media)

De voorbeelden worden gebuild in `sb7code-master/build`

Voer de demo's uit door de `.exe` bestanden te openen in `sb7code-master/bin`

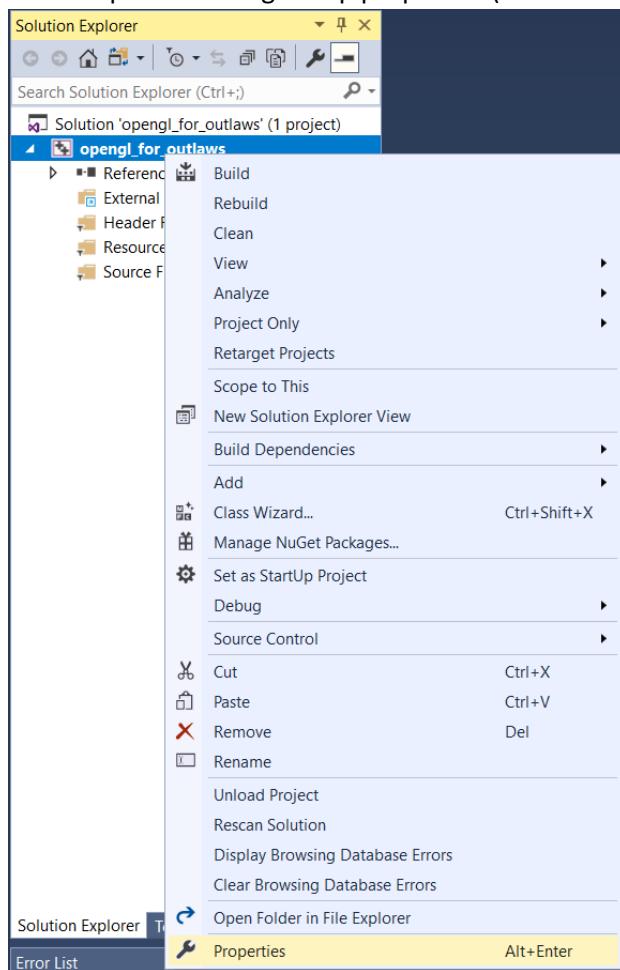
JE EERSTE PROGRAMMA OPZETTEN

Nu GLFW en sb7 zijn geconfigureerd kunnen we zelf een OpenGL project aanmaken in Visual Studio. De volgende stappen beschrijven hoe je zelf een leeg project kunt aanmaken in Visual Studio en daarin OpenGL code kan schrijven.

Hiervoor moeten onder andere bepaalde instellingen worden gewijzigd in Visual Studio zelf.

We zullen eerst een klein simpel programma maken die een gekleurd scherm toont.

1. Open Visual Studio 2017 en kies Visual C++ in het linker menu en maak een Empty Project aan in Visual Studio en geef het een leuke naam, bijvoorbeeld `opengl_for_outlaws`
Kies een handige locatie, bijvoorbeeld in dezelfde map als `sb7code-master`
Maak het project vervolgens aan.
2. Zweef met je muis boven de project tab (2 plusjes) in de Solution Explorer en klik op de rechter muisknop. Klik vervolgens op properties (helemaal onderaan).

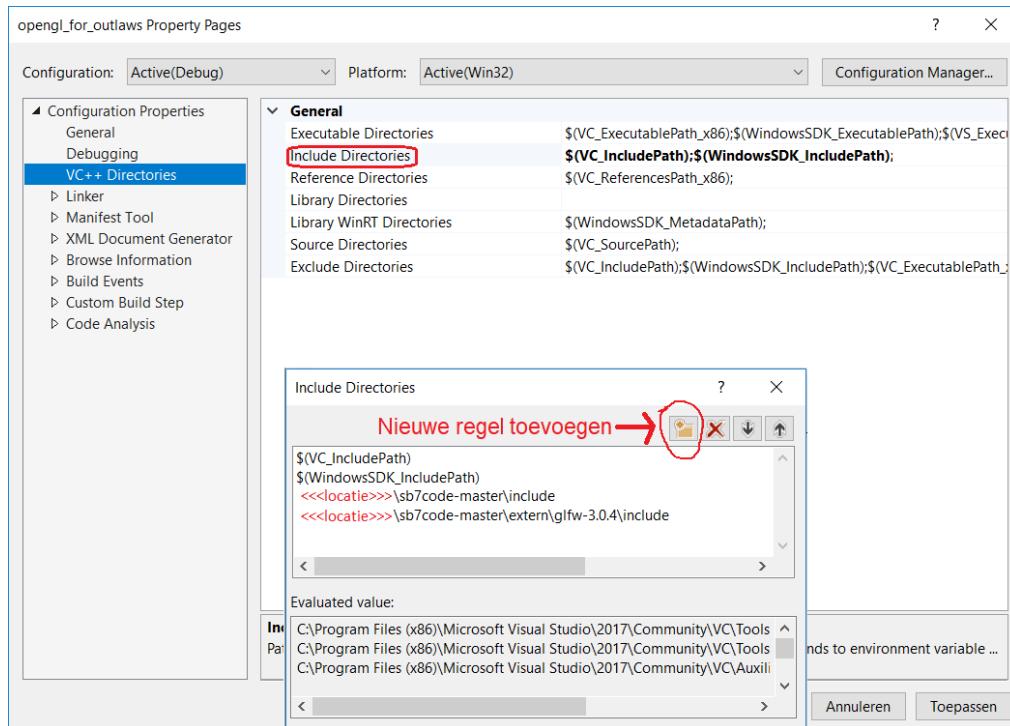


3. Ga naar VC++ directories (menu links) klik op het pijltje rechts bij 'Include Directories', vervolgens klik je op <Edit...>

Voeg de volgende locaties toe in het menu wat verschijnt

<<<locatie>>>/sb7code-master/extern/glfw-3.0.4/include

<<<locatie>>>/sb7code-master/include

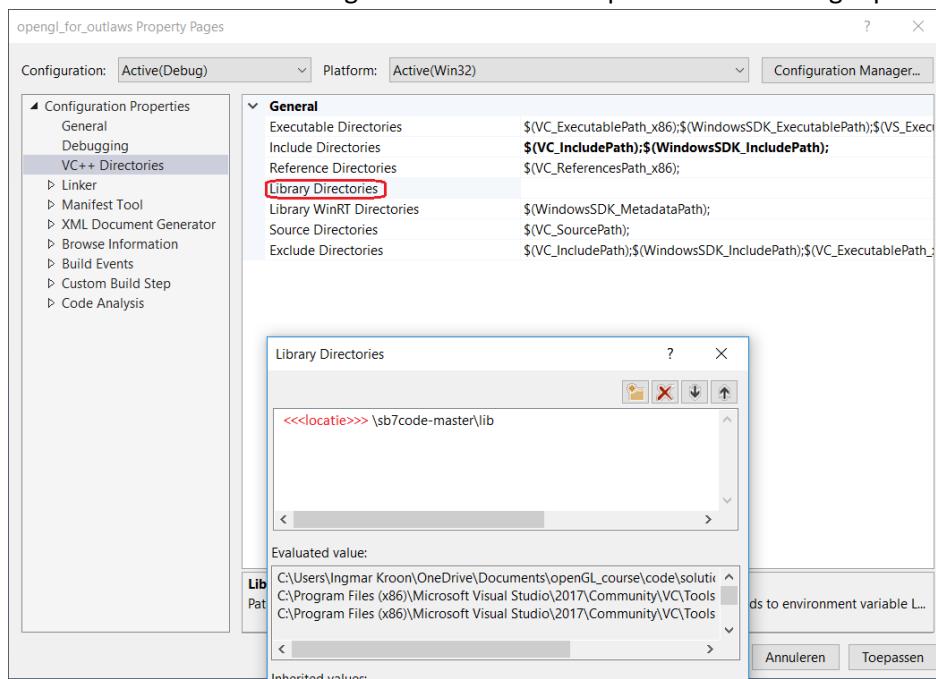


4. Voeg het volgende pad toe aan 'Library Directories'.

<<<localpath>>>/sb7code-master/lib

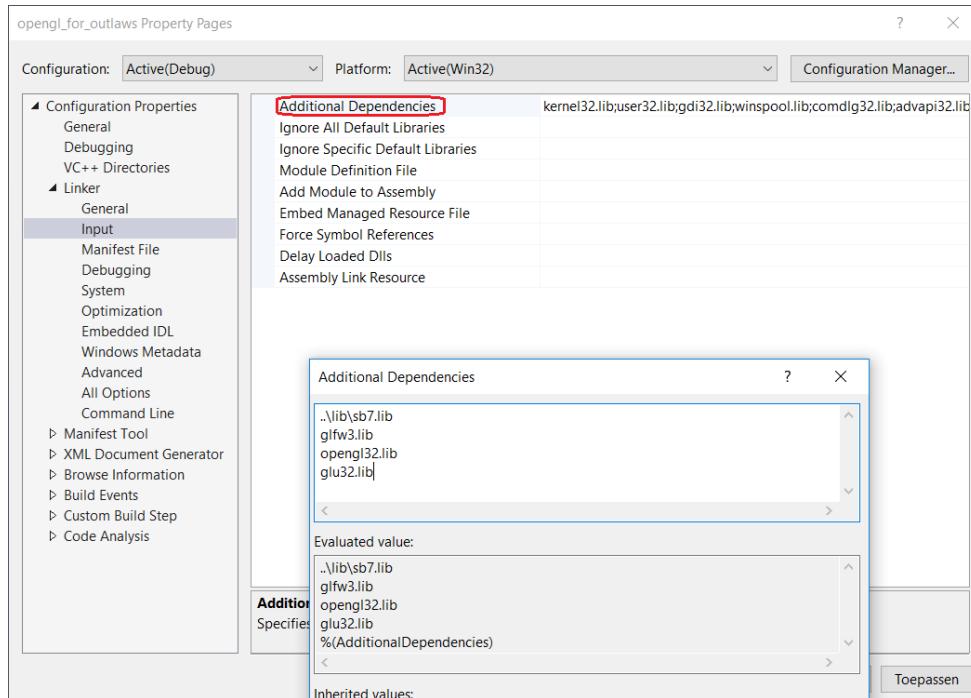
De 'Library Directories' bevinden zich, net als 'Include Directories', ook in VC++ directories.

Het venster voor het toevoegen van locaties kan op dezelfde manier geopend worden.



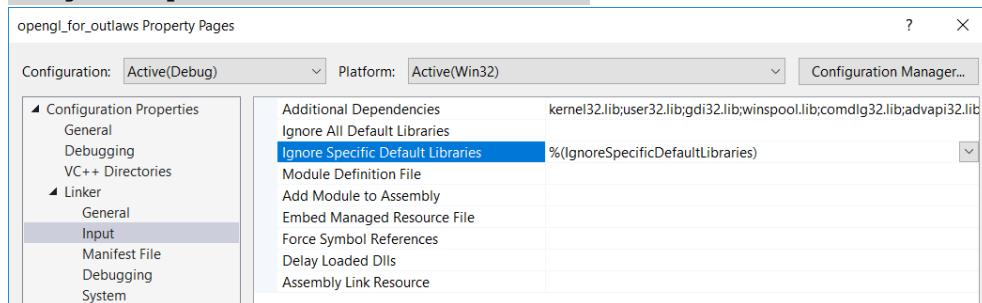
5. Ga naar ‘Input’ onder het menu ‘Linker’ en voeg de volgende items toe aan ‘Additional Dependencies’ gescheiden door een puntkomma.

```
..\lib\sb7.lib
glfw3.lib
opengl32.lib
glu32.lib
```



6. Voeg het volgende item toe bij ‘Ignore Specific Default Libraries’.

```
% (IgnoreSpecificDefaultLibraries)
```



7. Sluit de Property window en voeg een nieuw leeg cpp bestand toe aan het mapje ‘Source Files’ in de Solution Explorer. Het is helemaal aan jou hoe je dit bestand wil noemen, maar in de voorbeelden noemen we dit bestand `firstprogram.cpp`

8. Voeg de volgende code toe aan `firstprogram.cpp`
 Include altijd de `sb7.h` header file en herleid de class van `sb7::application`.

Helemaal onderaan zet je `DECLARE_MAIN(<<<classname>>>)` neer, dit geeft aan dat de huidige file de main entry point is van onze applicatie.

```
#include <sb7.h>

class firstprogram : public sb7::application
{
    void init()
    {
        static const char title[] = "OpenGL sample";

        sb7::application::init();

        memcpy(info.title, title, sizeof(title));
    }

    // deze functie wordt bij elke render iteratie uitgevoerd
    void render(double currentTime)
    {
        glClearColor(0.0f, 1.0f, 0.0f, 1.0f); // achtergrond kleur
        glClear(GL_COLOR_BUFFER_BIT); // framebuffer opschonen
    }
};

// dit bestand is geregistreerd als het opstart punt en voert de opstart logica van sb7 uit
DECLARE_MAIN(firstprogram)
```

!!!! BELANGRIJK !!!!:

In `sb7.h` zitten 2 regels die verandert moeten worden, doe je dit niet, dan kunnen er errors optreden.

Op regel 29 ontbreekt een underscore voor `WIN32`

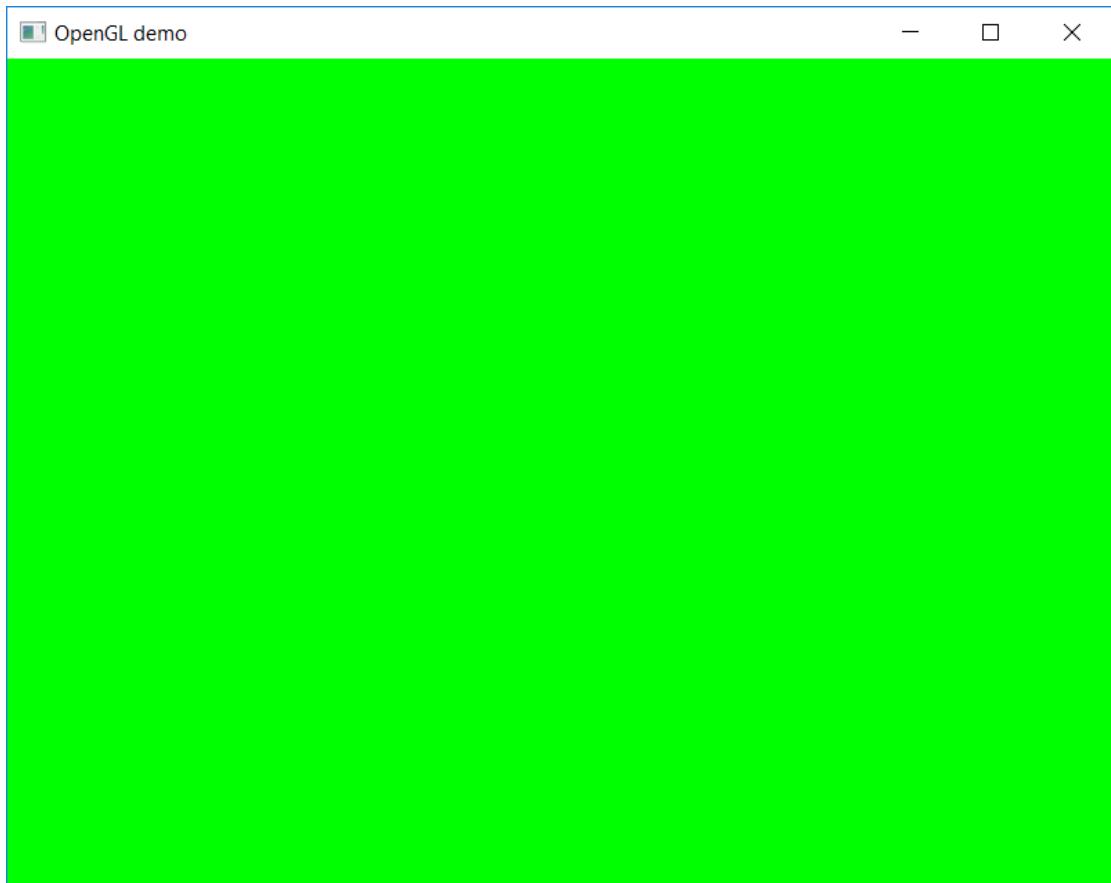
Verander regel 29 naar `#ifdef _WIN32`

```
26     ifndef __SB7_H__
27     define __SB7_H__
28
29     ifndef WIN32 → #ifdef _WIN32
30         pragma once
31         define _CRT_SECURE_NO_WARNINGS 1
32
33         define WIN32_LEAN_AND_MEAN 1
34         include <Windows.h>
35     else
36         include <unistd.h>
37         define Sleep(t) sleep(t)
38     endif
```

Op regel 180 moet `strcpy` worden verandert in `strcpy_s`

```
178     virtual void init()
179     {
180         strcpy → strcpy_s(info.title, "OpenGL SuperBible Example");
181         info.windowWidth = 800;
182         info.windowHeight = 600;
183     ifdef __APPLE__
```

9. Build de solution
10. Bij het uitvoeren van de solution verschijnt er automatisch een venster.
Dit venster genereert een groene achtergrond als de code correct wordt uitgevoerd.



3. OPENGL

SHADERS

Hoe werken shaders?

Aan het einde van het vorige hoofdstuk hebben we een zeer eenvoudig programma geschreven met een gekleurde achtergrond.

In dit hoofdstuk gaan we het programma uitbreiden met shaders.

Shaders zijn nodig om geometrie te renderen, zoals puntjes en driehoekjes.

Shaders zijn een belangrijk deel in de OpenGL pipeline (hoofdstuk 1), omdat shaders het toelaten om operaties uit te voeren per vertex, per fragment of zelfs per primitive.

De shaders worden geschreven in GLSL; een taal gebaseerd op C.

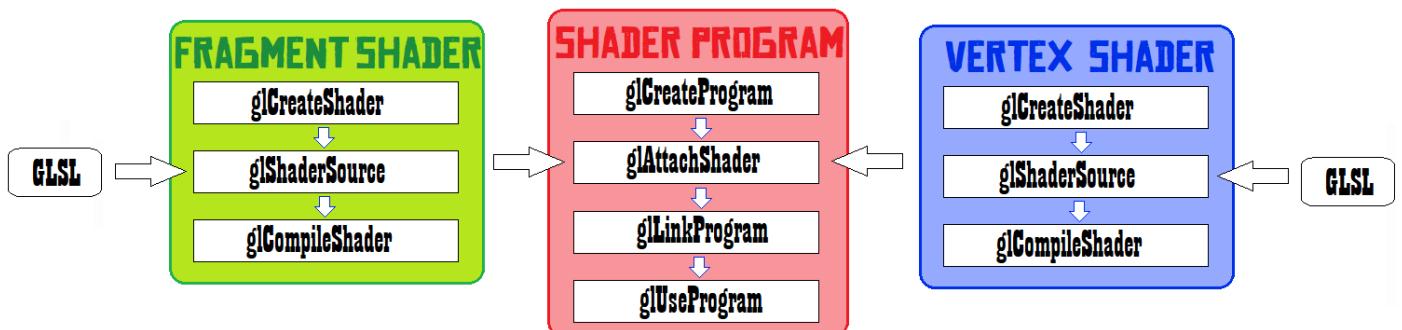
Zoals eerder besproken zijn shaders kleine geïsoleerde programmaatjes die parallel worden uitgevoerd op de GPU.

De shaders die we nu gaan gebruiken zijn de vertex shader en de fragment shader.

De broncode van een shader wordt geplaatst in een enkel shader object en wordt vervolgens gecompileerd.

Met behulp van een **shader programma** kunnen shaders gekoppeld worden aan een object.

Het volgende figuur geeft weer hoe een shader programma kan worden aangemaakt.



Vertex Array Object

Een vertex is een verzameling van attributen om een punt in 3D te beschrijven.

We hebben gezien dat een vertex in eerste instantie een positie is met een XYZ-coördinaat, maar er zijn ook andere vertex attributen mogelijk zoals texture coördinaten en normals (normaal vectoren).

Een **Vertex Array Object** is een soort van container die alle logica bevat voor een renderbaar object.

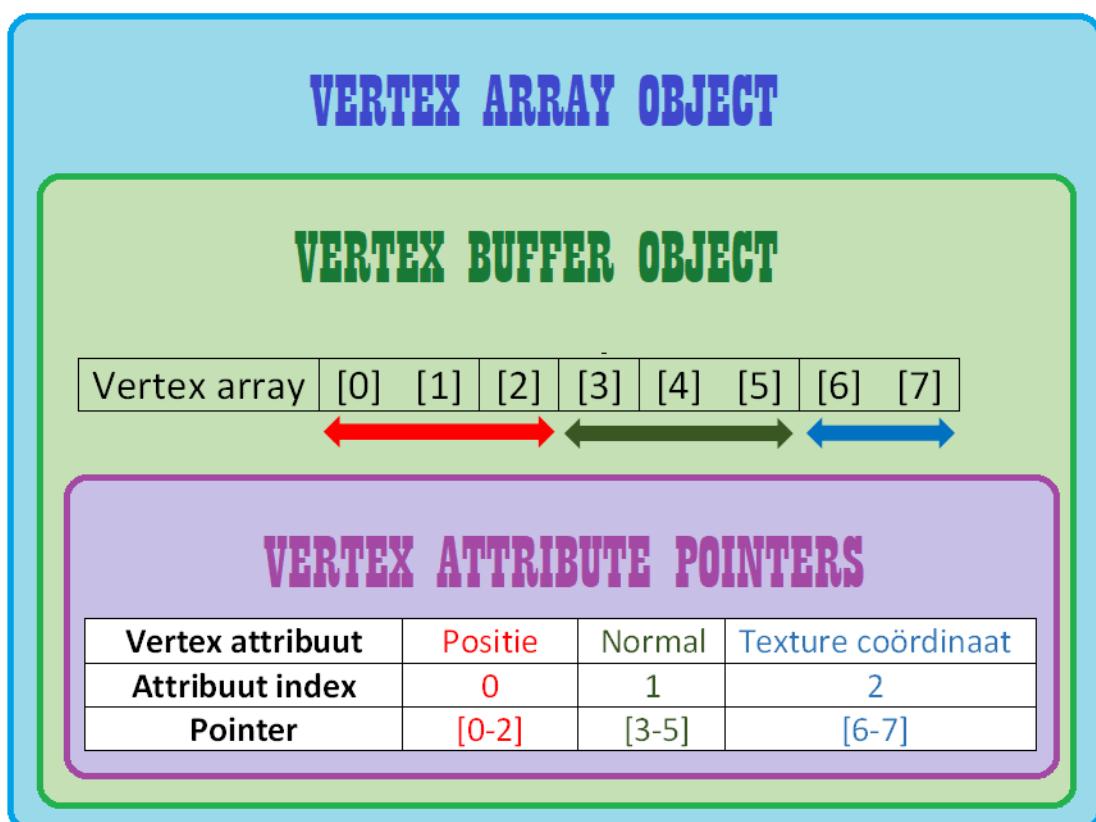
Deze "container" bevat alle benodigde input voor de vertex shader: vertex attributen.

Met andere woorden: de OpenGL pipeline wordt gevoedt vanuit een Vertex Array Object.

Belangrijke onderdelen van een Vertex Array Object zijn *Vertex Buffer Objects* en *Vertex Attribute Pointers*:

- Een Vertex Buffer Object is een object waarin vertex data geplaatst kan worden.
De bron van een Vertex Buffer Object wordt als een 1 dimensionale array verwacht.
Deze array bevat slechts de data (alle vertex attributen op een hoop).
De data in een Vertex Buffer Object wordt opgeslagen in het videogeheugen van de grafische hardware.
Het is mogelijk om meerder buffer objecten te koppelen aan een Vertex Array Object.
- Vertex Attribute Pointers verwijzen naar de juiste locatie van elk vertex attribuut in de array.
Elk attribuut krijgt een index, deze index kan vervolgens in de vertex shader gebruikt worden om de specifieke attributen op te halen.

Voor elke "render call" moet het juiste Vertex Array Object gekoppeld worden om een actief shader programma van de gewenste data te voorzien.



Vertex shader

Input

In de vertex shader worden alle vertex attributen opgehaald, afhankelijk van het op dat moment gekoppelde Vertex Array Object.

Het Vertex Array Object bevat immers de vertex attributen van een object (zoals een donut).

De vertex attributen kunnen in de vertex shader gemanipuleerd worden voordat ze ge-output worden naar de volgende fase in de pipeline.

Naast vertex attributen kennen we ook uniforms, dit zijn globale variabelen in de shader pipeline die niet afhankelijk zijn van een Vertex Array Object maar van een shader programma zelf.

In de vertex shader kan bijvoorbeeld de vertex positie verandert worden d.m.v. transformaties (zie hoofdstuk 5)..

Transformatiematrices zijn een mooie voorbeelden van uniforms.

Output

De eindpositie van een vertex wordt naar `gl_position` geschreven.

`gl_position` verwacht coördinaten in clip space.

Clip space is voor te stellen als een denkbeeldige kubus waarvan elke dimensie (XYZ) een bereik heeft van 1 tot -1, alles wat buiten dit bereik valt (dus buiten de kubus) wordt weggooit.

In de praktijk moeten objecten vaak eerst getransformeerd worden vanuit hun eigen lokale coördinatenstelsel naar clip space

De waarde in `gl_position` wordt ge-output naar de primitive assembly; waar primitives (bijvoorbeeld driehoeken) worden gecreëerd.

Attributen die bijdragen aan de pixel kleur (bijvoorbeeld een texture coördinaat of een normal), worden door de vertex shader ge-output naar de fragment shader met het `out` keyword.

In de fragment shader worden deze attributen gebruikt voor het bepalen van de pixel kleur.

Op de volgende pagina staat een voorbeeld met GLSL code van een vertex shader.

Fragment shader

De tweede programmeerbare fase in de OpenGL pipeline is de fragment shader.

Input die veel gebruikt wordt in de fragment shader zijn normals.

In de fragment shader vinden vaak lichtberekeningen plaats, daar worden normals veel voor gebruikt.

De output van de fragment shader is de fragment kleur.

Stel, we renderen een driehoek.

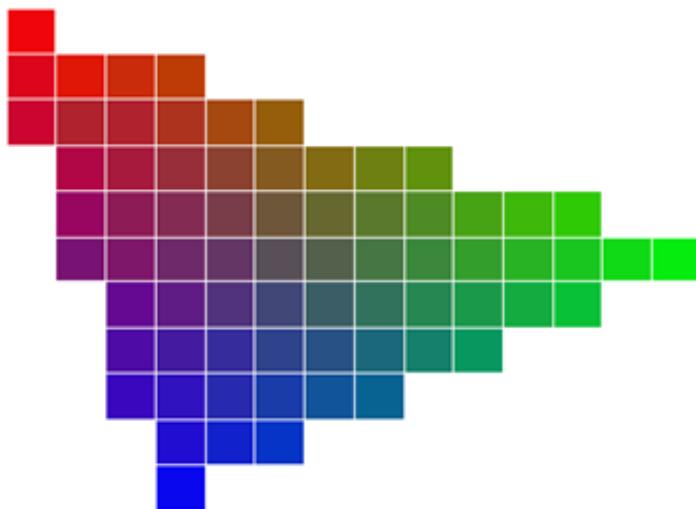
Deze driehoek bestaat uit 3 vertices (hoekpunten).

Elke vertex heeft een andere kleur



De primitive generation genereert vervolgens een driehoek die vervolgens gerasterd wordt naar pixels.

Omdat elk hoekpunt van de gerasterde driehoek een andere kleur heeft, wordt de kleur van de fragmenten tussen deze hoekpunten geïnterpolateerd, dit levert een vloeiende kleurovergang op tussen de hoekpunten.



Voorbeeld code shaders

De volgende code knipsels geven een impressie van shader implementaties.

Je hoeft op dit moment nog niet alles te begrijpen, maar het belangrijkste is om te zien hoe variabelen worden ge-input en ge-output.

Voorbeeld GLSL code vertex shader

```
#version 420 core

// input; vertex attributen; gebonden aan vertex (VAO)
layout (location = 0) in vec3 vertexPosition;
layout (location = 1) in vec3 vertexNormal;

// output; naar fragment shader
out vec3 worldspaceNormal;
out vec3 worldspacePosition;

// input; uniforms of globale variabelen; gebonden aan shader programma
uniform mat4 model;           // lokaal coördinatenstelsel matrix
uniform mat4 view;            // camera coördinatenstelsel matrix
uniform mat4 projection;      // projectiematrix

void main(void)
{
    // vertex positie in clip space berekenen
    vec4 clipSpacePosition = projection * view * model * vec4(vertexPosition, 1.0f);

    // output; naar primitive assembly
    gl_Position = clipSpacePosition;

    // vertex attributen manipuleren voor fragment shader
    worldspaceNormal = vec3(model * vec4(vertexNormal, 1.0f));
    worldspacePosition = vec3(model * vec4(vertexPosition, 1.0f));
}
```

Voorbeeld GLSL code fragment shader

```
#version 420 core

// input; output van vertex shader
in vec3 worldspaceNormal;
in vec3 worldspacePosition;

// output; fragment kleur
out vec4 color;

// input; belichtings parameters
uniform vec3 modelColor;
uniform vec3 lightPos;

void main(void)
{
    // bereken de fragment kleur op basis van gegeven parameters ...
    ...
    vec3 result = doSomeCalculations ...

    color = vec4(result, 1.0f);
}
```

VERTEX RENDEREN

We beginnen met het renderen van een enkele vertex.

Eerst voegen we een functie toe die eenmalig wordt uitgevoerd zodra het programma opstart.

```
virtual void startup()
{
}
```

Vervolgens voegen we een methode toe die het geheugen opruimt bij het afsluiten van het programma.

Deze functie zullen we later verder invullen.

```
virtual void shutdown()
{
}
```

Voor het maken van een shader programma maken we ook een aparte functie.

Hierin gebeurt het volgende:

- Vertex en fragment shader aanmaken
- Shader programma aanmaken
- Shaders compileren
- Shaders linken aan shader programma
- Shader programma retourneren

```
GLuint compile_shaders(void)
{
}
```

We voegen eerst de vertex shader broncode toe in `compile_shaders`.

De bovenste regel geeft aan dat we versie 4.5 van GLSL gebruiken.

In de main functie kunnen we shader code uitvoeren.

In dit geval hardcoden we een punt in de shader in clip space (-1.0 t/m 1.0).

Omdat onze vertex zich in clip space bevindt kunnen we de positie rechtstreeks toewijzen aan `gl_Position`

```
static const char * vs_source[] =
{
    "#version 450 core\n"
    "void main(void)\n"
    "{\n        gl_Position = vec4(0.0, 0.0, 0.5, 1.0);\n    }\n};
```

Vervolgens kunnen we de fragment shader code toevoegen.

Ook hier geven we aan dat we versie 4.5 van GLSL willen gebruiken.

Met het **out** keyword geven we aan dat het om een output variabele gaat.

De output wordt rechtstreeks vertaald in de fragment kleur voor een gerasteriseerde primitive.

```
static const char * fs_source[] =
{
    "#version 450 core\n"
    "out vec4 color;\n"
    "void main(void)\n"
    "{\n        color = vec4(0.0, 0.8, 1.0, 1.0);\n    }\n};
```

Nadat we de shaders hebben geschreven, kunnen we de volgende code toevoegen aan `compile_shaders`.

`glCreateShader` maakt een nieuw shader object aan op basis van de target (vertex of fragment shader).

`glShaderSource` voegt de broncode toe aan het shader object.

`glCompileShader` compileert de shader code.

`glCreateProgram` maakt een shader programma waaraan shader objecten kunnen worden gelinkt.

`glLinkProgram` koppelt alle shader objecten aan een shader programma

`glDeleteShader` verwijdert het shader object. Zodra de shader objecten gelinkt zijn aan een programma object is de shader onderwater al toegevoegd, dus zijn de shader objecten overbodig.

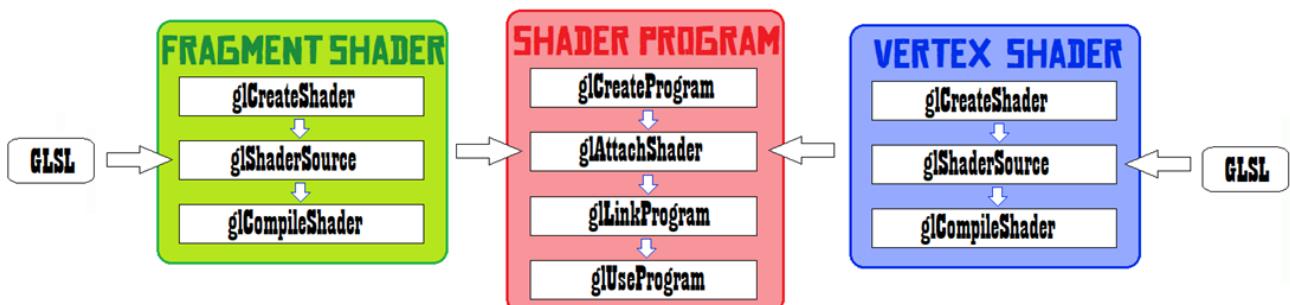
```
GLuint vertex_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex_shader, 1, vs_source, NULL);
glCompileShader(vertex_shader);

GLuint fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragment_shader, 1, fs_source, NULL);
glCompileShader(fragment_shader);

GLuint program = glCreateProgram();
glAttachShader(program, vertex_shader);
glAttachShader(program, fragment_shader);
glLinkProgram(program);

glDeleteShader(vertex_shader);
glDeleteShader(fragment_shader);

return program;
```



We hebben nu een functie die een shader programma bouwt en deze retourneert aan de hand van een ID.

Er zijn 2 globale variabelen die we moeten toevoegen:

- Shader programma ID
- Vertex Array Object

```
private:  
    GLuint rendering_program; // shader programma ID  
    GLuint vao; // Vertex Array Object ID
```

Voeg de volgende code toe aan startup():

```
// laat compile_shaders() een shader programma genereren  
rendering_program = compile_shaders();  
  
// koppel de ID van het vertex array object aan de context  
glBindVertexArray(vao);  
  
// vertex array object aanmaken  
glGenVertexArrays(  
    1,      // grootte  
    &vao   // object ID  
);  
  
glBindVertexArray(0);
```

Voeg de volgende code toe aan render():

```
glUseProgram(rendering_program); // koppel het shader programma aan de context  
  
glPointSize(40.0f);  
  
// geef aan welke primitives gerendert worden uit onze data  
glDrawArrays(  
    GL_POINTS, // primitive type  
    0, // n.v.t.  
    1 // aantal vertices die gerendert worden  
);
```

In

startup() functie is het vertex array object gekoppeld aan de context.

In de render() functie wordt het shader programma gekoppeld aan de context.

Zodra het programma afgesloten wordt kunnen deze objecten worden weggegooid.

Voeg de volgende code toe aan shutdown():

```
glDeleteVertexArrays(1, &vao);  
glDeleteProgram(rendering_program);
```

Het resultaat is een blauw punt op een groene achtergrond.



De volledige broncode kan men vinden in het mapje `1vertex`

SIMPELE DRIEHOEK

Tot nu toe hebben we een enkele vertex gerendert

Om meerdere vertices te renderen is het niet efficiënt om dit in de shader te hardcoden.

De shaders zullen voortaan gevoed worden vanuit het programma zelf .

In dit voorbeeld gaan we een driehoek tekenen.

Eerder hebben we al een shader programma en een Vertex Array Object aangemaakt.

We beginnen door een variabele toe te voegen voor een Vertex Buffer Object.

```
private:  
    GLuint rendering_program; // shader programma ID  
    GLuint vao; // Vertex Array Object ID  
    GLuint vbo; // Vertex Buffer Object ID
```

De vertex coördinaten van onze driehoek kunnen we vervolgens toevoegen aan de `startup()` methode in een array.

Let op dat we de data nu initialiseren buiten onze shader.

```
// gehardcode driehoek  
float vertices[] = {  
    -0.5f, -0.6f, 0.0f,  
    0.5f, -0.5f, 0.0f,  
    0.0f, 0.5f, 0.0f  
};
```

Vervolgens kunnen we een Vertex Array Object en een Vertex Buffer Object generen en deze koppelen aan de context.

```
// vertex array object aanmaken
glGenVertexArrays(
    1,          // aantal vertex array objecten
    &vao        // object ID
);

// Vertex Buffer Object aanmaken
glGenBuffers(
    1,          // aantal buffer objecten
    &vbo        // object ID
);

// koppel het vertex array object aan de context
glBindVertexArray(vao);

// koppel het buffer object aan de juiste target (GL_ARRAY_BUFFER)
glBindBuffer(GL_ARRAY_BUFFER, vbo);
```

Plaats onze array met vertex posities in het GPU geheugen.

```
// plaats de daadwerkelijke array met data in het GPU geheugen
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

De volgende tabel geeft het formaat weer van een vertex attribuut voor positie (XYZ-coördinaten).

We hebben 3 attributen van elke 3 (X, Y en Z) van elk 32-bit (4 byte).

De afstand tussen elke positie is 3, dat noemen we de stride.

	Vertex 1			Vertex 2			Vertex 2		
Vertex attribute	X	Y	Z	X	Y	Z	X	Y	Z
Buffer	4 bytes	4 bytes	4 bytes	12 bytes			12 bytes		
Stride	3			3			3		

Op basis van deze gegevens kan een **Vertex Attribute Pointer** gegenereerd worden voor positie.

```
// vertex attribuut instellen voor positie
glVertexAttribPointer(
    0,                  // index; later nodig in shader
    3,                  // aantal waardes; X, Y en Z
    GL_FLOAT,           // data type
    GL_FALSE,           // normaliseren ja/nee
    3 * sizeof(float), // geheugen grootte (3 keer 32 bits float)
    (void*)0            // offset; begin vanaf ...
);

// activeer vertex attribuut met index 0
 glEnableVertexAttribArray(0);
```

Pas in de render functie de `glDrawArrays` functie aan.

In plaats van een punt willen we een driehoek genereren als primitive. Ook rendereren we nu 3 vertices.

```
glDrawArrays(  
    GL_TRIANGLES, // primitive type  
    0, // n.v.t.  
    3 // aantal verticles die getekent worden  
)
```

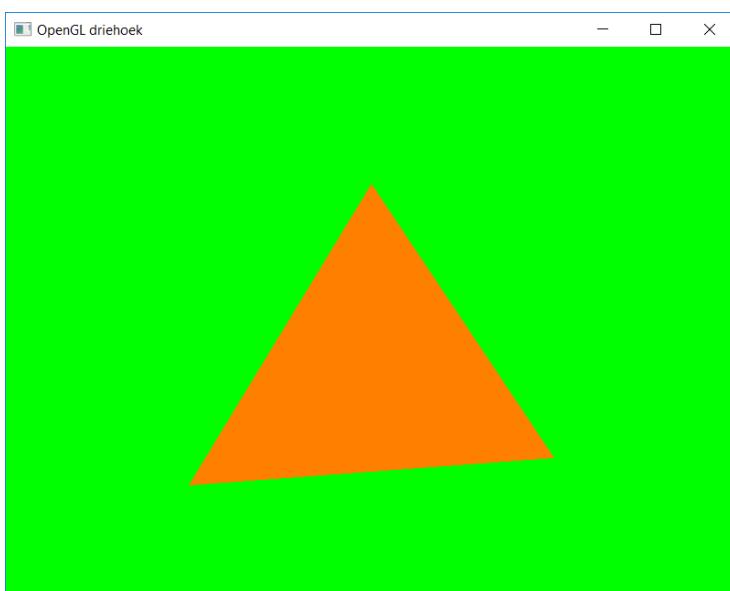
Het enige wat we nu moeten doen is de vertex shader aanpassen.

De vertex shader moet de vertex attributen ophalen die we hebben ingesteld in de Vertex Attribute Pointers.

`layout (location = 0)` specificeert het formaat van de vertex input, we willen vertex attribuut op index 0 gebruiken.

```
static const char * vs_source[] =  
{  
    "#version 420 core\n"  
    "\n"  
    "layout (location = 0) in vec3 vertPos;\n"  
    "\n"  
    "void main(void)\n"  
    "{\n        gl_Position = vec4(vertPos.x, vertPos.y, vertPos.z, 1.0f);\n    }\n"};
```

Het verwachte resultaat levert een oranje driehoek op.



De volledige broncode kan men vinden in het mapje [2driehoek](#)

GEÏNDEXEERD RENDEREN

Zoals eerder besproken ziet OpenGL alleen maar driehoekjes, lijntjes en puntjes.

Om een driehoek te tekenen zijn 3 punten nodig.

Voor een rechthoek zijn 4 punten nodig.

Als we een recht hoek zouden tekenen aan de hand van een enkele vertex array zijn er 6 punten nodig omdat een rechthoek uit 2 driehoeken bestaat.

We weten dat 2 punten van de driehoeken elkaar overlappen, dus willen we het liefst deze 2 punten hergebruiken.

Een oplossing hiervoor is geïndexeerd renderen.

Per driehoek verwijst je alleen naar de index van de juiste vertex, in plaats van dat je voor elke driehoek de complete vertices specificeert.

Voor de index data bestaat er een apart object binnen OpenGL: Element Buffer Object.

Dit object is qua structuur vergelijkbaar met een Vertex Buffer Object, maar in plaats van vertex data bevat een Element Buffer Object index data.

Een Element Buffer Object is net als een Vertex Buffer Object, onderdeel van een Vertex Array Object.

Eerst voegen we een referentie toe voor ons Element Buffer Object.

```
private:  
    GLuint rendering_program; // shader programma ID  
    GLuint vao; // Vertex Array Object ID  
    GLuint vbo; // Vertex Buffer Object ID  
    GLuint ebo; // Element Buffer Object ID
```

Vervolgens kan de vertex en index data worden geinitialiseerd.

```
// alle benodigde vertices voor een rechthoek  
float vertices[] = {  
    0.5f, 0.5f, 0.0f, // rechtsboven  
    0.5f, -0.5f, 0.0f, // rechtsonder  
    -0.5f, -0.5f, 0.0f, // linksonder  
    -0.5f, 0.5f, 0.0f // linksboven  
};  
  
// de vertex indices  
int indices[] = { // begin vanaf 0  
    0, 1, 3, // eerste driehoek  
    1, 2, 3 // tweede driehoek  
};
```

Buffer genereren voor een Element Buffer Object.

```
glGenVertexArrays(1, &vao); // Vertex Array Object aanmaken  
glGenBuffers(1, &vbo); // Vertex Buffer Object aanmaken  
glGenBuffers(1, &ebo); // Element Buffer Object aanmaken
```

Daarna kan de Element Buffer Object gekoppeld worden aan de juiste target en kunnen we de data specificeren voor deze buffer.

```
// element buffer object koppelen aan de context (GL_ELEMENT_ARRAY_BUFFER)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);

// plaats de daadwerkelijke index data in het buffer geheugen
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

Nu we de indices correct hebben gebufferd moet er nog 1 ding gebeuren.

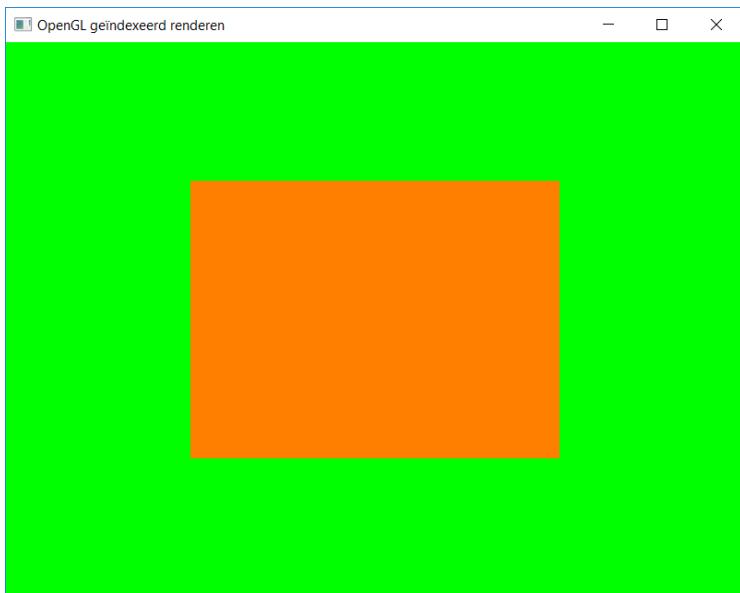
We hadden eerder een functie die specificeert welk soort primitives we willen renderen uit onze data:

`glDrawArrays`.

Omdat we nu onze primitives generen op basis van indices, gebruiken we `glDrawElements`.

```
// geef aan welke primitives gerendert worden uit onze data
// (specifiek voor indexed rendering)
glDrawElements(
    GL_TRIANGLES, // primitive type
    6, // aantal indices
    GL_UNSIGNED_INT, // n.v.t.
    0
);
```

Resultaat:



De volledige broncode kan men vinden in het mapje [3indices_renderen](#)

MEERDERE VERTEX ATTRIBUTEN

Een vertex kan meer attributen bevatten dan alleen de positie.

Hier voor moet extra vertex attributen worden aangemaakt.

We willen in dit geval elke vertex een andere kleur geven.

Een kleur heeft net als een positie 3 kanalen: RGB (0 tot 1)

De vertex array ziet er als volgt uit met extra data voor kleur.

De eerste 3 waarden (vanaf links) zijn steeds de X, Y en Z coördinaten van een vertex.

De laatste 3 waarden zijn de RGB kanalen van een vertex.

```
// alle benodigde vertices voor een rechthoek
float vertices[] = {
    0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f, // rechtsboven
    0.5f, -0.5f, 0.0f, 1.0f, 1.0f, 0.0f, // rechtsonder
    -0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f, // linksonder
    -0.5f, 0.5f, 0.0f, 0.0f, 0.0f, 1.0f // linksboven
};
```

Vervolgens kunnen we een extra Vertex Attribute Pointer specificeren voor de kleur.

Omdat er 3 extra waarden aan toegevoegd zijn moet het geheugen worden uitgebreid met 12 bytes.

Want 3 extra float waarden zijn 3×4 bytes = 12 bytes.

Ook is de stride (aantal waarden voor een enkele vertex) groter geworden naar 6.

	Vertex 1						Vertex 2					
Vertex attribuut	x	y	z	r	g	b	x	y	z	r	g	b
Buffer	6 × 4 bytes						24 bytes					
Stride	6						6					

```
// vertex attribuut instellen voor positie
glVertexAttribPointer(
    0, // locatie
    3, // aantal waarden
    GL_FLOAT, // data type
    GL_FALSE,
    6 * sizeof(float), // stride; 3 extra waarden dus 3 + 3 = 6
    (void*)0 // begin positie van de buffer data
);

	glEnableVertexAttribArray(0);

// vertex attribuut instellen voor kleur
glVertexAttribPointer(
    1, // locatie
    3, // aantal waarden
    GL_FLOAT, // data type
    GL_FALSE,
    6 * sizeof(float), // stride
    (void*)(3 * sizeof(float)) // begin positie van de kleur buffer (3)
);

	glEnableVertexAttribArray(1);
```

Nu alle attributen juist gebufferd zijn kunnen we de shaders aanpassen zodat elke vertex zijn eigen kleur krijgt.

De locatie in de shader is hetzelfde als de locatie gespecificeerd in onze pointer naar het desbetreffende vertex attribuut (0 voor positie, 1 voor kleur).

```
static const char * vs_source[] =
{
    "#version 420 core
"
    "layout (location = 0) in vec3 vertPos;
    "layout (location = 1) in vec3 vertColor;
"
    "out vec3 fragColor;
"
    "void main(void)
    "{
        "    gl_Position = vec4(vertPos, 1.0f);
        "    fragColor = vertColor;
    }
};
```

De kleur van de vertex willen we toewijzen in de fragment shader, deze sturen we door met het **out** keyword.

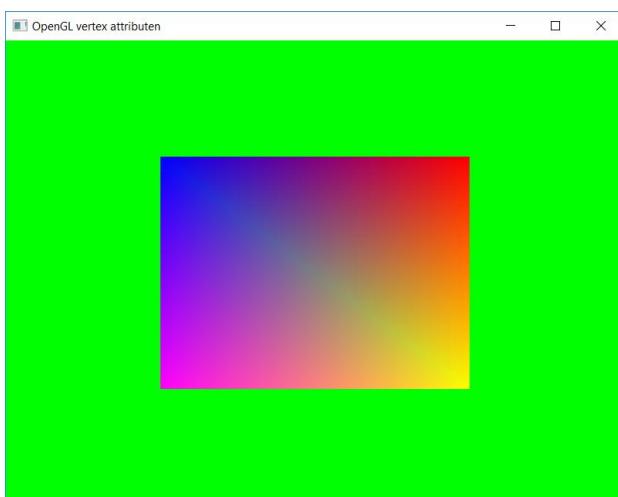
In de fragment shader halen we dezelfde variabele op met het **in** keyword en kan de kleur meteen ge-output worden.

```
static const char * fs_source[] =
{
    "#version 420 core
"
    "in vec3 fragColor;
    "out vec4 color;
"
    "void main(void)
    "{
        "    color = vec4(fragColor, 1.0f);
    }
};
```

Het resultaat levert een gekleurde rechthoek op.

Let op dat de kleur vloeidend is geïnterpolateerd tussen de hoekpunten van elke driehoek.

De volledige broncode kan men vinden in het mapje [4extra_vertex_attributen](#)



UNIFORMS

Uniforms zijn globale variabelen die op elk moment kunnen worden opgehaald in elke shaderfase.

Uniforms zijn gebonden aan een shader programma, en niet als vertex attributen aan een Vertex Array Object (model; bijv. een donut).

Een transformatie matrix willen we bijvoorbeeld uitvoeren op alle vertices binnen een object.

Niet elke vertex heeft een eigen transformatiematrix als vertex attribuut, dat is zeer onpraktisch.

Uniforms kunnen in de applicatie worden gespecificeerd op basis van:

- Data type
- Variabele in applicatie
- Naam waarmee de variabele kan worden opgehaald in een shader
- Shader programma

In elke shader fase kan een uniform worden opgehaald.

We breiden ons programma (met de gekleurde rechthoek) uit met een simpele rotatie om de Z-as (deze staat immers haaks op ons “beeldscherm”).

Voor transformaties maken we gebruik van de vmath bibliotheek.

Deze moet eerst ge-include worden.

De onderliggende wiskunde wordt in het volgende hoofdstuk uitgediept.

```
#include <sb7.h>
#include <vmath.h>
```

Een uniform is gebonden aan een object ID. Deze maken we eerst aan.

Ook maken we een instantie aan voor een 4x4 matrix.

```
private:
    GLuint rendering_program; // shader programma ID
    GLuint vao; // Vertex Array Object ID
    GLuint vbo; // Vertex Buffer Object ID
    GLuint ebo; // Element Buffer Object ID

    GLuint transform_location; // uniform object locatie
    vmath::mat4 rotation_matrix; // transformatie matrix
};
```

Vervolgens kunnen we in de render functie een rotatie matrix maken met behulp van vmath::rotate. We willen een constante rotatie om de Z-as uitvoeren op basis van de tijd.

```
// koppel het shader programma aan de context
glUseProgram(rendering_program);

// koppel het vertex array object aan de context
glBindVertexArray(vao);

// maak een rotatie matrix aan die om de Z-as roteert
rotation_matrix = vmath::rotate(
    (float)currentTime * 81.0f, // rotatie
    0.0f, // X-as
    0.0f, // Y-as
    1.0f // Z-as
);

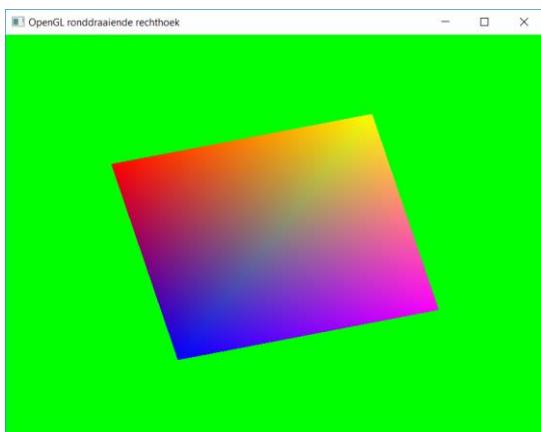
// koppel het uniform object aan een transformatie matrix
glUniformMatrix4fv(
    transform_location, // locatie van uniform object variabele
    1, // aantal elementen
    GL_FALSE, // transponeren ja/nee
    rotation_matrix // matrix die we willen specificeren voor de uniform
);
```

Nu het uniform object gekoppeld is aan onze rotatie matrix kunnen we de uniform in de vertex shader ophalen.

De rotatie matrix kan vervolgens worden vermenigvuldigd met de vector die de vertex positie representeert.

```
static const char * vs_source[] =
{
    "#version 420 core\n"
    "\n"
    "layout (location = 0) in vec3 vertPos;\n"
    "layout (location = 1) in vec3 vertColor;\n"
    "\n"
    "out vec3 fragColor;\n"
    "\n"
    "uniform mat4 rotation;\n"
    "\n"
    "void main(void)\n"
    "{\n        gl_Position = rotation * vec4(vertPos, 1.0f);\n        fragColor = vertColor;\n    }\n};
```

Het resultaat levert een ronddraaiende rechthoek op. Code in mapje [5draainde_rechthoek](#)



4. MATHEMATIEK

Tot nu toe hebben we een punt, een driehoek en een rechthoek gerendert in OpenGL.

We hebben nu een basis gelegd waarop we verder kunnen bouwen.

Om in 3D te kunnen renderen, hebben we bepaalde wiskunde nodig; lineaire algebra.

Het verplaatsen, roteren en projecteren van 3D geometrie zijn typische gevallen waarbij we lineaire algebra nodig hebben.

We hebben ook te maken met verschillende coördinatenstelsels.

Bijvoorbeeld de berekeningen om vertices van het ene naar het andere coördinatenstelsel te transformeren.

Het is handig om een beetje voorkennis te hebben van wiskunde alvorens dit hoofdstuk te beginnen.

Voor matrix en vector operaties in OpenGL gebruiken we de vmath bibliotheek van de auteurs van OpenGL SuperBible.

Voor quaternionen gebruiken we een zelfgemaakte implementatie.

In dit hoofdstuk gaan we in op het volgende:

- Wat een vector is en hoe deze vectoren bijvoorbeeld vertex posities kunnen representeren of een richting
- Wat een matrix is en hoe deze transformaties kan representeren zoals translaties en rotaties
- Hoe we kunnen rekenen met vectors en matrices
- Wat quaternionen zijn en hoe we daar mee kunnen rekenen (en waarom ze nuttig zijn)
- Hoe de verschillende transformatiematrices zijn opgebouwd die een vertex van model space naar clip space transformeren

VECTOR

Eerder hebben we besproken wat een vertex is; die onder andere het attribuut voor een positie bevat.

Deze positie is een XYZ-coördinaat.

Een XYZ coördinaat kan ook beschouwd worden als een 3-delige vector in een 3D ruimte.

Een vector kan naast een positie ook een richting representeren met een lengte.

Vectors zijn een van de belangrijkste fundamenten voor het manipuleren van 3D geometrie.

Naast vectors in 3 dimensies (\mathbb{R}^3) zijn er ook vectors in 2 dimensies (\mathbb{R}^2), of een n aantal dimensies (\mathbb{R}^n).

Een eenheidsvector is een vector met een lengte van 1. Bijvoorbeeld: (1,0,0).

Om van een normale vector (met een andere lengte dan 1) een eenheidsvector te maken moet deze geschaald worden, dit noemen we normaliseren.

De eenheidsvector heeft dezelfde richting als voorheen, alleen is de magnitude 1.

Neem als voorbeeld de vector (3, 0, -5):

- Ga 3 positieve eenheden over de X-as
- Ga 0 eenheden over de Y-as
- Ga 5 negatieve eenheden over de Z-as

Een vector kan je bijvoorbeeld gebruiken om een richting te representeren waar een camera heen wijst.

Onderstaande figuur geeft een vector \vec{v} weer met een willekeurige magnitude.

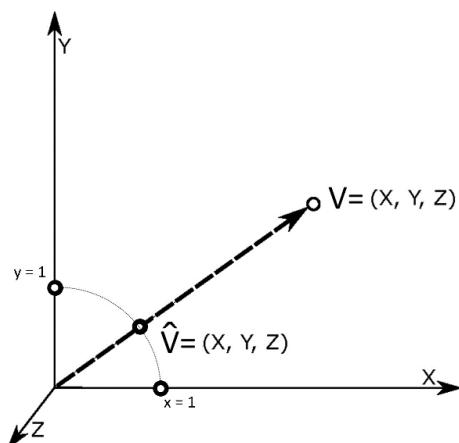
Vector \hat{v} is hierin een eenheidsvector met een magnitude van 1.

Als we vector \vec{v} normaliseren (magnitude 1 maken), krijgen we een eenheidsvector \hat{v} .

Eenheidsvectoren worden vaak gebruikt om een richting te representeren.

Om een vector met een willekeurige magnitude te normaliseren moet de vector gedeeld worden door zijn eigen lengte.

$$\hat{v} = \frac{\vec{v}}{\|\vec{v}\|}$$



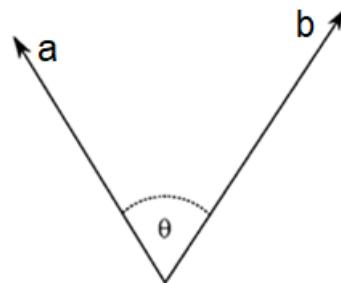
Inwendig product

Met behulp van het inwendig product kan de hoek tussen 2 vectoren berekent worden.

Het inwendig product $a \cdot b$ levert een scalair (enkele waarde) als resultaat op.

Het inwendig product wordt onder andere toegepast om de invalshoek te berekenen tussen de lichtvector en de normaalvector.

$$\cos \theta = \frac{ab}{|a||b|}$$



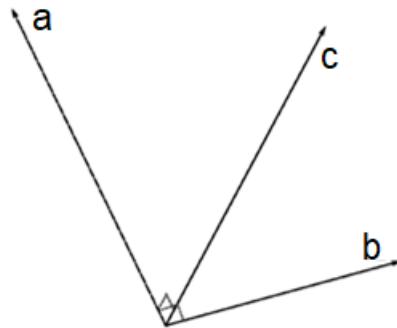
Kruisproduct

Het kruisproduct $a \times b$ levert een derde vector op die loodrecht op het denkbeeldige oppervlak staat van vector a en b .

Indien het resultaat van een kruisproduct wordt genormaliseerd, dan krijgen we de normaalvector van een oppervlak, oftewel een vector met lengte 1 die haaks op het denkbeeldige oppervlak staat van a en b .

Indien we a en b omdraaien in de formule, dan wijst vector c in de tegenovergestelde richting.

$$\begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix}$$



Norm van een vector

De magnitude, lengte of norm van een vector kan als volgt berekend worden:

$$|\vec{v}| = \sqrt{\vec{v}_x^2 + \vec{v}_y^2 + \vec{v}_z^2}$$

Je kan deze formule zien als een driedimensionale Stelling van Pythagoras.

MATRIX

Een matrix is een collectie van getallen gepresenteerd in rijen en kolommen.

Matrices worden onder andere gebruikt voor het manipuleren van een punt in een 3D ruimte (vector).

We willen bijvoorbeeld weten wat de nieuwe positie is van een 3D punt na een rotatie om een bepaald punt.

Zo'n rotatie kan gepresenteerd worden in een matrix, met daarin de rotatie parameters.

Matrices kunnen vermenigvuldigd worden met vectoren, andere matrices en zelfs met een scalair (enkele waarde).

Hoofddiagonaal

Elke matrix heeft een hoofddiagonaal, dit zijn de waarden die lopen van linksboven naar rechtsonder.

$$\begin{bmatrix} A_{00} & A_{10} & A_{20} & A_{30} \\ A_{01} & \textcolor{blue}{A_{11}} & A_{21} & A_{31} \\ A_{02} & A_{12} & \textcolor{blue}{A_{22}} & A_{32} \\ A_{03} & A_{13} & A_{23} & \textcolor{blue}{A_{33}} \end{bmatrix}$$

Matrix-vector vermenigvuldiging

Een matrix die vermenigvuldigt wordt met een vector levert een nieuwe vector op.

Het onderstaande voorbeeld geeft een matrix-vector vermenigvuldiging weer (4x4 matrix en 4-delige vector).

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} a_1 \cdot x + a_2 \cdot y + a_3 \cdot z + a_4 \cdot w \\ b_1 \cdot x + b_2 \cdot y + b_3 \cdot z + b_4 \cdot w \\ c_1 \cdot x + c_2 \cdot y + c_3 \cdot z + c_4 \cdot w \\ d_1 \cdot x + d_2 \cdot y + d_3 \cdot z + d_4 \cdot w \end{bmatrix}$$

Matrix-matrix vermenigvuldiging

Het is ook mogelijk om een matrix met een matrix te vermenigvuldigen, dit levert een nieuwe matrix op.

Onderstaande voorbeeld geeft een matrix-matrix vermenigvuldiging weer (3x3 matrices).

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{bmatrix} = \begin{bmatrix} a_1 \cdot x_1 + a_2 \cdot y_1 + a_3 \cdot z_1 & a_1 \cdot x_2 + a_2 \cdot y_2 + a_3 \cdot z_2 & a_1 \cdot x_3 + a_2 \cdot y_3 + a_3 \cdot z_3 \\ b_1 \cdot x_1 + b_2 \cdot y_1 + b_3 \cdot z_1 & b_1 \cdot x_2 + b_2 \cdot y_2 + b_3 \cdot z_2 & b_1 \cdot x_3 + b_2 \cdot y_3 + b_3 \cdot z_3 \\ c_1 \cdot x_1 + c_2 \cdot y_1 + c_3 \cdot z_1 & c_1 \cdot x_2 + c_2 \cdot y_2 + c_3 \cdot z_2 & c_1 \cdot x_3 + c_2 \cdot y_3 + c_3 \cdot z_3 \end{bmatrix}$$

Getransponeerde matrix

Bij een getransponeerde matrix worden de rijen als kolommen geschreven en de kolommen als rijen.

De getransponeerde matrix kan ook verkregen worden door de matrix te spiegelen om de hoofddiagonaal.

$$A = \begin{bmatrix} A_{00} & \textcolor{green}{A_{10}} & A_{20} & A_{30} \\ A_{01} & \textcolor{blue}{A_{11}} & \textcolor{green}{A_{21}} & A_{31} \\ A_{02} & A_{12} & \textcolor{blue}{A_{22}} & \textcolor{green}{A_{32}} \\ A_{03} & A_{13} & A_{23} & \textcolor{blue}{A_{33}} \end{bmatrix} \quad A^T = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ \textcolor{blue}{A_{10}} & \textcolor{blue}{A_{11}} & \textcolor{blue}{A_{12}} & \textcolor{blue}{A_{13}} \\ \textcolor{green}{A_{20}} & \textcolor{green}{A_{21}} & \textcolor{blue}{A_{22}} & \textcolor{orange}{A_{23}} \\ \textcolor{green}{A_{30}} & \textcolor{green}{A_{31}} & \textcolor{orange}{A_{32}} & \textcolor{blue}{A_{33}} \end{bmatrix}$$

TRANSFORMATIES

Eerder hebben we besproken wat vectors en matrices zijn en hoe deze vermenigvuldigt kunnen worden met elkaar.

Het is mogelijk om een vertex positie te manipuleren door haar vector te vermenigvuldigen met een matrix. Dit noemen we transformeren.

Drie veelvoorkomende transformaties zijn de translatie, schaling en rotatie.

Transformaties zijn bijvoorbeeld nuttig om coördinaten tussen verschillende coördinatenstelsels te verplaatsen.

Om een transformatie uit te voeren op een vector vermenigvuldigen we de vector met de desbetreffende transformatiematrix.

Eenheidsmatrix

Als een eenheidsmatrix vermenigvuldigt wordt met een vector resulteert dit in dezelfde vector. Zie dit als een vermenigvuldiging van elk vector component met een scalair 1:

Alle waarden in de hoofddiagonaal zijn 1 bij een eenheidsmatrix, en de rest van de waarden zijn 0.

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}$$

Translatiematrix

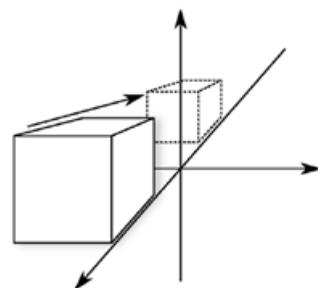
Een translatie is in lineaire algebra de term voor een verplaatsing.

Met behulp van een translatiematrix kan een vector over verschillende assen verplaatst worden.

Het volgende voorbeeld represeneert een translatie van een vector m.b.v. een translatiematrix.

De t waarden in de laatste kolom representeren de daadwerkelijke translaties voor de X, Y en Z-as.

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & t_x \\ 0.0 & 1.0 & 0.0 & t_y \\ 0.0 & 0.0 & 1.0 & t_z \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ v_w \end{bmatrix}$$

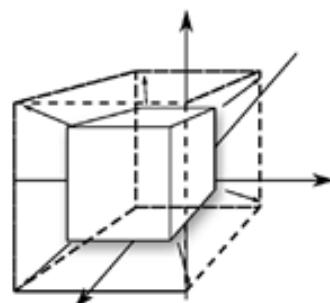


Schalingsmatrix

Met behulp van schalen kan een object vergroot en verkleint worden over de verschillende assen.

De s waarden uit de schalingsmatrix worden vermenigvuldigd met de vector componenten om een nieuwe geschaalde vector op te leveren.

$$\begin{bmatrix} s_x & 0.0 & 0.0 & 0.0 \\ 0.0 & s_y & 0.0 & 0.0 \\ 0.0 & 0.0 & s_z & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} v_x \cdot s_x \\ v_y \cdot s_y \\ v_z \cdot s_z \\ v_w \end{bmatrix}$$

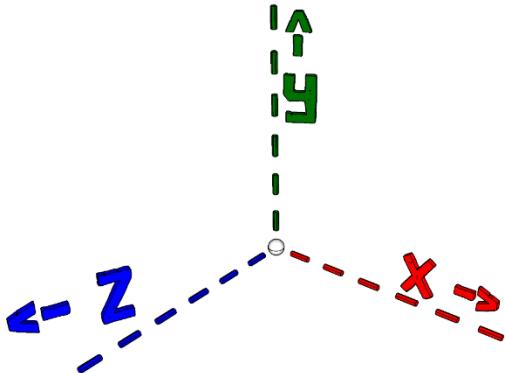


Rotatiematrix

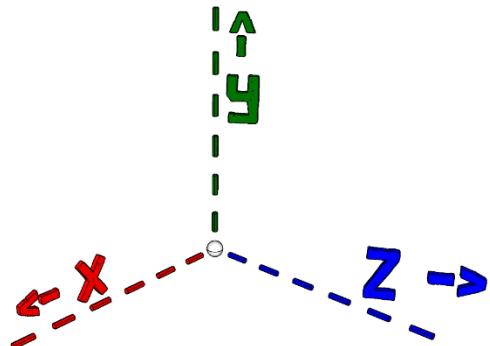
Hoeken van Euler

De hoeken van Euler beschrijven hoe 3 hoeken een oriëntatie beschrijven in een driedimensionale ruimte. Elke Euler hoek beschrijft een rotatie om een as. Deze assen zijn 3 vectoren die orthogonaal (loodrecht) ten opzichte van elkaar. Voor het gemak noemen we deze assen de X, Y en Z-as.

De volgende figuur toont een rechtshandig en linkshandig coördinaten stelsel.



RECHTSHANDIG



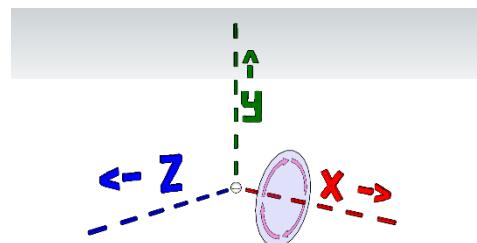
LINKSHANDIG

Met behulp van de rotatiematrix kan een object om de X, Y en Z-as gedraaid worden.

De rotatiematrix die gebruikt wordt hangt af van om welke as we een rotatie willen uitvoeren.

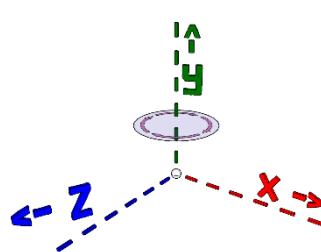
X-as rotatiematrix (pitch):

$$R_x(\theta) = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & \cos \theta & -\sin \theta & 0.0 \\ 0.0 & \sin \theta & \cos \theta & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \cdot \cos \theta + v_z \cdot -\sin \theta \\ v_y \cdot \sin \theta + v_z \cdot \cos \theta \\ v_w \end{bmatrix}$$



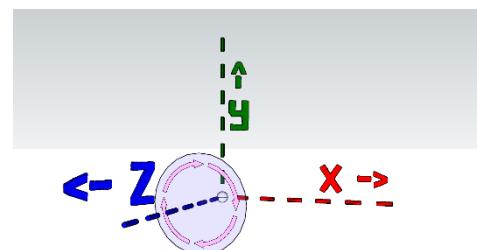
Y-as rotatiematrix (yaw):

$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0.0 & \sin \theta & 0.0 \\ 0.0 & 1.0 & 0 & 0.0 \\ -\sin \theta & 0.0 & \cos \theta & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} v_x \cdot \cos \theta + v_z \cdot \sin \theta \\ v_y \\ v_x \cdot -\sin \theta + v_z \cdot \cos \theta \\ v_w \end{bmatrix}$$



Z-as rotatiematrix (roll):

$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0.0 & 0.0 \\ \sin \theta & \cos \theta & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} v_x \cdot \cos \theta + v_y \cdot -\sin \theta \\ v_x \cdot \sin \theta + v_y \cdot \cos \theta \\ v_z \\ v_w \end{bmatrix}$$



De rotatiematrices hierboven zijn gebaseerd op een rechtshandig coördinatenstelsel.

De linkshandige variant van een rotatiematrix kan verkregen worden door de rechtshandige rotatiematrix te transponeren (minus-teken spiegelt). Hetzelfde geldt andersom.

Het is ook mogelijk om alle 3 rotaties te combineren door de matrices van elke as met elkaar te vermenigvuldigen.

Als we de drie X, Y en Z rotatie matrices met elkaar vermenigvuldigen krijgen we een enkele rotatie matrix voor alle 3 assen. Dit proces heet ook wel 'concatteneren'.

Standaard wordt de volgorde ZYX toegepast.

Een veelgemaakte fout is dat er gedacht wordt dat de volgorde van concatteneren geen invloed heeft op het resultaat, dit is echter heel belangrijk.

Euler rotaties zijn ook gevoelig voor gimball lock, een probleem waarbij 2 rotatie assen dezelfde rotatie uitvoeren. Nadere uitleg volgt.

We spreken het volgende af:

X-as rotatie	Y-as rotatie	Z-as rotatie
$\cos \phi = c_\phi$	$\cos \theta = c_\theta$	$\cos \psi = c_\psi$
$\sin \phi = s_\phi$	$\sin \theta = s_\theta$	$\sin \psi = s_\psi$

$$R_z(\psi)R_y(\theta)R_x(\phi) = \begin{bmatrix} c_\theta \cdot c_\psi & c_\phi \cdot s_\psi + s_\phi \cdot s_\theta \cdot c_\psi & s_\phi \cdot s_\psi - c_\phi \cdot s_\theta \cdot c_\psi & 0.0 \\ -c_\theta \cdot s_\psi & c_\phi \cdot c_\psi - s_\phi \cdot s_\theta \cdot s_\psi & s_\phi \cdot c_\psi + c_\phi \cdot s_\theta \cdot s_\psi & 0.0 \\ s_\theta & -s_\phi \cdot c_\theta & c_\phi \cdot c_\theta & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Matrices in OpenGL

OpenGL ziet een 4x4 matrix niet als een tweedimensionale array, maar als enkele array met 16 float waarden.

De volgorde van deze array is gebaseerd op de kolommen (verticaal).

De 4x4 matrix links kan gepresenteerd worden als een 1 dimensionale array (rechts).

$$\begin{bmatrix} A_{00} & A_{10} & A_{20} & A_{30} \\ A_{01} & A_{11} & A_{21} & A_{31} \\ A_{02} & A_{12} & A_{22} & A_{32} \\ A_{03} & A_{13} & A_{23} & A_{33} \end{bmatrix}$$

```
float A[] {
    A00, A01, A02, A03, // eerste kolom
    A10, A11, A12, A13, // tweede kolom
    A20, A21, A22, A23, // derde kolom
    A30, A31, A32, A33 // vierde kolom
};
```

Volgorde van transformaties

Het vermenigvuldigen van matrices is niet commutatief, dat wil zeggen dat de verschillende volgorden waarin je de matrices elk een ander resultaat opleveren.

We leggen hier 2 voorbeelden uit, gecombineerde translatie-rotatie en meerdere rotaties onderling.

Translatie en rotatie

Stel, je hebt:

- Een vector in het oorspronkelijke (eenheids-)coördinatenstelsel A.
- Een 4x4 matrix die de positie en oriëntatie beschrijft van coördinatenstelsel B.

Als de vector in coördinatenstelsel A vermenigvuldigt wordt met de matrix van coördinatenstelsel B, dan resulteert dit in een nieuwe vector die getransformeerd is naar coördinatenstelsel B.

Op deze manier kunnen alle vertices van een object vermenigvuldigt worden met een 4x4 matrix, zodat het hele object naar een coördinatenstelsel met een andere positie en oriëntatie transformeert.

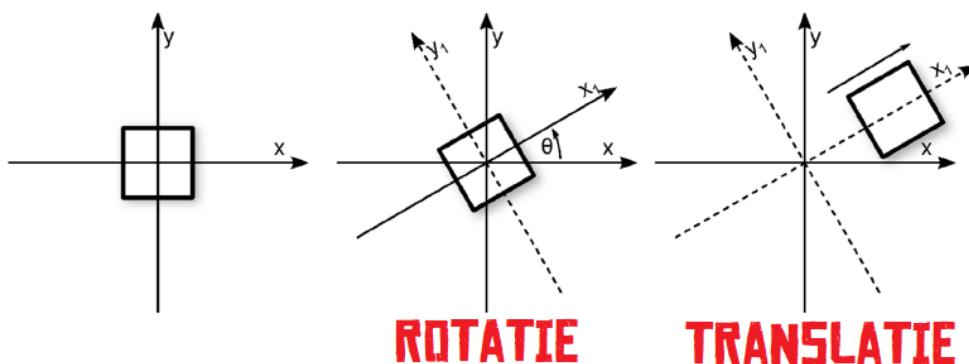
De mogelijkheden zijn eindeloos, je kan de vertices niet alleen van coördinatenstelsel A naar coördinatenstelsel B transformeren, maar vervolgens ook van coördinatenstelsel B naar coördinatenstelsel C, etc.

Het is mogelijk om matrices in verschillende volgordes te vermenigvuldigen.

Alleen de volgorde van de verschillende matrices heeft invloed op het resultaat.

Figuur 5 illustreert een rotatie gevolgd door een translatie (over de nieuw ontstane "X-as" x_1).

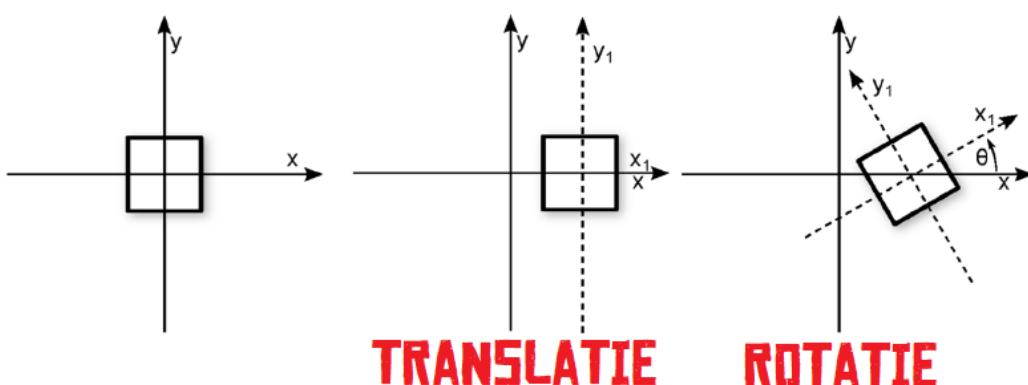
Figuur 1



ROTATIE TRANSLATIE

Figuur 6 illustreert een translatie (over de oorspronkelijk X-as) gevolgd door een rotatie.

Figuur 2



TRANSLATIE ROTATIE

Gimball lock

In het vorige voorbeeld hebben we gezien dat de volgorde van translaten en roteren het resultaat kan beïnvloeden.

Ditzelfde geldt voor Euler rotaties onderling.

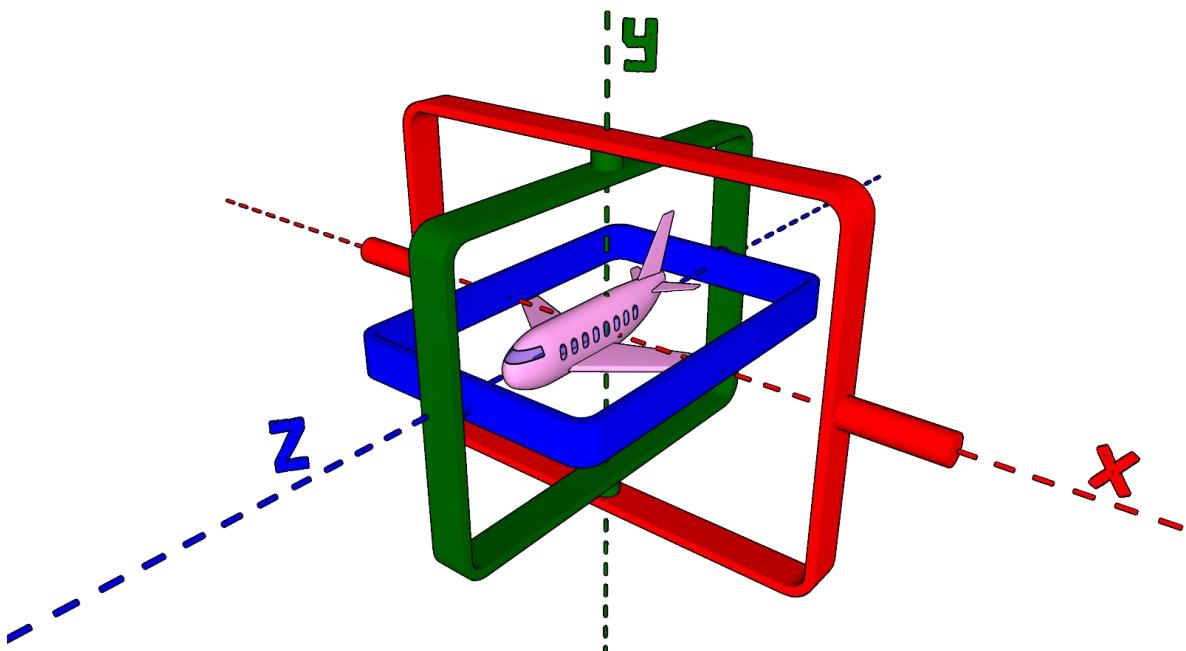
Stel we hebben een vliegtuig wat we willen roteren over de X, Y en Z-as met behulp van een rotatiematrix, dan kunnen we dit representeren in een "Gimball".

- De rode as en ring representeren de rotatie om de X-as; oftewel de pitch van het "vliegtuig".
- De groene as en ring representeren de rotatie om de Y-as; oftewel de yaw van het "vliegtuig".
- De blauwe as en ring representeren de rotatie om de Z-as; oftewel de roll van het "vliegtuig".

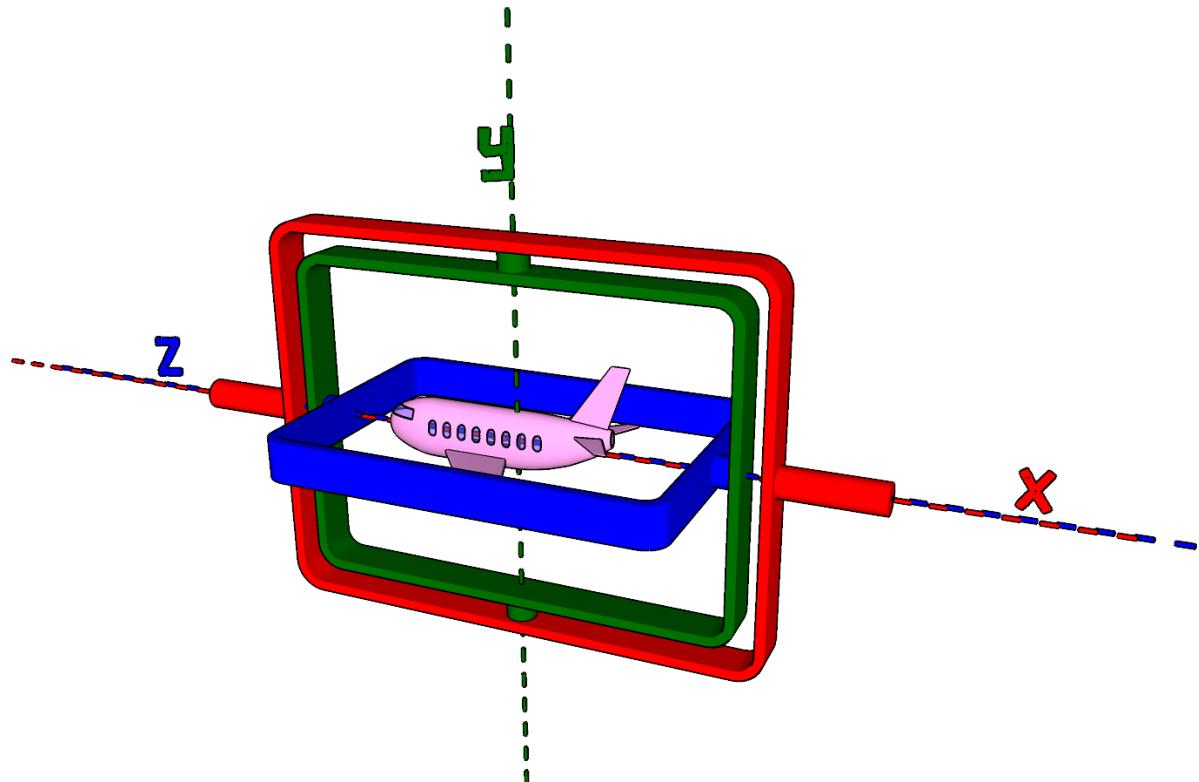
De binnenste ring (blauw) is de eerste rotatie die wordt uitgevoerd op het vliegtuig, de middelste ring (groen) de tweede rotatie, en de buitenste ring de derde rotatie.

Als we dit vertalen naar lineaire algebra krijgen we een rotatiematrix gepresenteerd als $R_z(\theta)R_y(\theta)R_x(\theta)$

Let er specifiek op dat alle 3 assen orthogonaal zijn, vanuit deze (begin) oriëntatie is het mogelijk om elke rotatie vrij uit te voeren.



Als het vliegtuig 90° om de Y-as rechtsom draait ziet de gimball er als volgt uit:



Het resultaat brengt een negatief gevolg met zich mee: zowel de X-as als de Z-as zijn nu parallel.

Als we nu het vliegtuig roteren om de X-as krijgen we hetzelfde effect als roteren om de Z-as.

Het vliegtuig kan nu geen pitch beweging meer maken vanuit deze positie.

Een regel is dat de eerste rotatie (binnenste ring) altijd correct wordt uitgevoerd.

Stel dat de eerste rotatie een pitch beweging represeneert dan is deze beweging niet afhankelijk van de andere rotaties.

In de ideale situatie staan alle rotatie assen haaks op elkaar ten opzichte van het vliegtuig, ook nadat het vliegtuig in verschillende richtingen is geroteerd, helaas is dit niet mogelijk met "Euler angles".

QUATERNIONEN

Rotaties kunnen beschreven worden als matrices, maar ook als quaternionen.

Quaternionen hebben als voordeel dat ze compact zijn (slechts 4 waarden) en ze maken het mogelijk om rotaties uit te voeren zonder Gimball lock.

Een volledige implementatie is te vinden in `quaternion.h`

Complexe getallen

Voordat we met quaternionen gaan rekenen moeten we eerst weten wat een complex getal is.

Een complex getal bestaat uit een reëel gedeelte en een imaginair gedeelte.

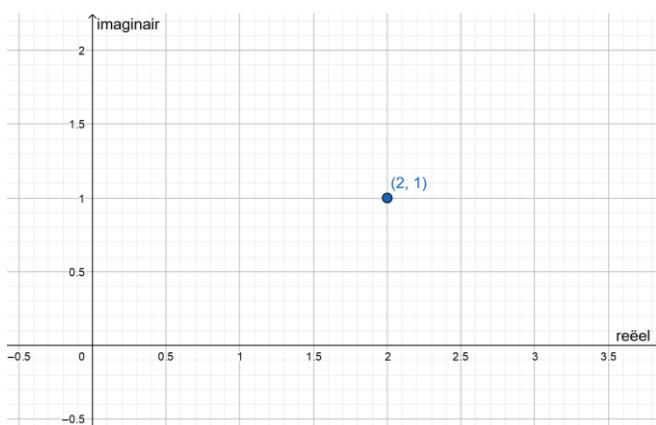
In tegenstelling tot een complex getal beperkt een quaternion zich niet tot 1 imaginair deel maar 3 imaginaire delen.

Zoals een reële getallen op een lijn kunnen worden weergegeven kunnen complexe getallen op een vlak worden weergegeven (en quaternionen in een 4 dimensionale ruimte).

Voor het imaginaire gedeelte geldt $i = \sqrt{-1}$

Een complex getal kan gerepresenteerd in een 2 dimensionaal assenstelsel.

Onderstaande assenstelsel toont een complex getal waarvan het reële gedeelte 2 is en het imaginaire gedeelte i of $1i$.



Waarom zijn complexe getallen nuttig?

Het complexe getal is uitgevonden om bepaalde vergelijkingen te kunnen oplossen waar normaal geen oplossing voor is.

Imaginaire getallen zijn te vergelijken met negatieve getallen, onder negatieve getallen zijn bepaalde vraagstukken onmogelijk, bijvoorbeeld het kopen van een negatief aantal aardappelen bij de groenteboer, dit is niet mogelijk, maar conceptueel gezien wel met negatieve getallen.

Een voorbeeld waarbij imaginaire getallen nuttig zijn met negatieve wortels:

$$0 = x^2 + 4$$

$$x^2 = -4$$

$$x = \sqrt{-4} \text{ of } x = -\sqrt{-4}$$

We weten dat $i = \sqrt{-1}$ dus we kunnen bovenstaande als volgt vereenvoudigen:

$$x = \sqrt{-1} \cdot \sqrt{4}$$

$$x = 2i \text{ of } x = -2i$$

Indien we geen complexe getallen hadden bestond er geen oplossing (negatieve wortel trekken is onmogelijk).

Quaternionen vermenigvuldigen

i, j en k zijn de imaginaire uitbreiding van het assenstelsel voor een quaternion.

Een quaternion a kan als volgt gedefinieerd worden:

$$a = a_{\text{re\"eel}} + a_i i + a_j j + a_k k$$

Vermenigvuldigingsregels

Voor quaternion rotaties is vermenigvuldigen de belangrijkste operatie die we gaan gebruiken.

Belangrijke regels voor het vermenigvuldigen van quaternionen zijn:

$$i^2 = j^2 = k^2 = i \cdot j \cdot k = -1$$

$$j \cdot k = i$$

$$i \cdot k = j$$

$$j \cdot i = k$$

Het vermenigvuldigen van de imaginaire delen is niet-commutatief, dat wil zeggen dat de verschillende volgorden van vermenigvuldigen niet hetzelfde resultaat opleveren.

$$k \cdot j = -i$$

$$k \cdot i = -j$$

$$i \cdot j = -k$$

Als we 2 quaternionen willen vermenigvuldigen zijn we de bovenstaande regels toepassen.

De onderstaande tabel geeft weer hoe 2 quaternionen met elkaar vermenigvuldigt worden:

x	1	i	j	k
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

VOORBEELD

Stel we willen quaternion a vermenigvuldigen met quaternion b

$$a = 5 + 2i + 6j + 4k$$

$$b = 4 + 3i + 5j + 2k$$

Deze quaternionen kunnen we representeren in onderstaande vermenigvuldigingstabel.

x	5	$2i$	$6j$	$4k$
4	20	$8i$	$24j$	$16k$
$3i$	$15i$	-6	$18k$	$-12j$
$5j$	$25j$	$-10k$	-30	$20i$
$2k$	$10k$	$4j$	$-12i$	-8

Vervolgens kunnen we het re\"ele deel en de imaginaire delen afzonderlijk optellen

re\"eel	$20 - 6 - 30 - 8$	-24
i	$8i + 15i + 20i - 12i$	$31i$
j	$24j - 12j + 25j + 4j$	$41j$
k	$16k + 18k - 10k + 10k$	$34k$

Het nieuwe quaternion c wat uit deze vermenigvuldiging volgt is:

$$-24 + 31i + 41j + 34k$$

Vermenigvuldigings formule

Voor het implementeren van quaternion multiplicatie kan een formule gebruikt worden.

Voor de quaternion componenten spreken we het volgende af:

reëel component	w
i component	x
j component	y
k component	z

$$ab = c$$

$$c_x = a_w b_x + a_x b_w + a_y b_z - a_z b_y$$

$$c_y = a_w b_y + a_y b_w + a_z b_x - a_x b_z$$

$$c_z = a_w b_z + a_z b_w + a_x b_y - a_y b_x$$

$$c_w = a_w b_w - a_x b_x - a_y b_y - a_z b_z$$

Vector roteren met behulp van rotatie quaternion (Sandwich product)

Stel, we willen een positie gerepresenteerd als vector \vec{v} roteren om een as met θ graden.

Deze as wordt gepresenteerd als een eenheidsvector \hat{u} .

We kunnen deze rotatie uitvoeren met behulp van het Sandwich product.

1. Rotatie en positie omzetten in quaternioenen

De eerste stap is om van positie vector \vec{v} een quaternion te maken.

De XYZ-componenten van \vec{v} kunnen gerepresenteerd worden op de imaginaire assen.

Voor een positie geldt dat het reële gedeelte 0 is, zo'n quaternion wordt ook wel een puur quaternion genoemd.

Het quaternion voor de positie noemen we p

Vector $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$ kan gepresenteerd worden als quaternion $p = 0 + v_x i + v_y j + v_z k$

Vervolgens kan de rotatie worden omgezet naar een quaternion.

Hiervoor hebben we 2 gegevens nodig: de rotatie θ in graden en de rotatie-as gepresenteerd als eenheidsvector \hat{u}

Uit rotatie-as $\hat{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$ en hoek θ volgt het quaternion $q = \cos\left(\frac{\theta}{2}\right) + (u_x i + u_y j + u_z k) \sin\left(\frac{\theta}{2}\right)$

Een rotatie quaternion dient altijd genormaliseerd te zijn.

Voor het normaliseren van quaternioenen geldt:

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2} = 1$$

2. Vermenigvuldigen

Voor het vector-quaternion rotatie geldt het Sandwich product:

$$p' = qpq^*$$

- p is het (te roteren) positie quaternion
- q is het rotatie quaternion
- q^* is de geconjugeerde van q waarbij geldt dat $q^* = q_w - q_xi - q_yj - q_zk$
Voor een geconjugeerd quaternion geldt dat de waarden voor i, j en k negatief zijn.
- p' is het nieuwe geroteerde quaternion.

De naam Sandwich product valt als volgt te verklaren:

q en q^* zijn de boven- en onderkant van de ‘sandwich’ (het broodje).

p zijn de groenten en het vlees (‘ge-sandwiched’ tussen het broodje)

Uiteraard kan de vermenigvuldigingstabel toegepast worden voor deze quaternion rotatie, eerst bereken je qp , vervolgens kan het tussenantwoord vermenigvuldigt worden met q^* .

$$p' = (qp)q^*$$

3. Geroteerde quaternion omzetten naar vector

De nieuwe positie v' van quaternion p' zit gebakken in het imaginaire deel.

Indien de rotatie correct berekent wordt is het reële gedeelte van p' gelijk aan 0.

Uit quaternion $p' = 0 + v'_x i + v'_y j + v'_z k$ volgt de nieuwe vector $\vec{v}' = \begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix}$.

EXTRA voor de verstrooide professoren

Een quaternion kan opgesplitst worden in een scalair gedeelte (w) en een vector gedeelte (xyz).

$p = (0, \vec{v})$ waarbij geldt dat \vec{v} de positie vector is.

$q = (q_w, \vec{q})$ waarbij geldt dat \vec{q} het vector gedeelte is van q oftewel (q_x, q_y, q_z) en

$q^* = (q_w, -\vec{q})$ waarbij geldt dat $-\vec{q}$ het vector gedeelte is van q^* oftewel $(-q_x, -q_y, -q_z)$

$p' = qpq^*$ kan met behulp van bovenstaande notatie vereenvoudigd worden tot een zeer efficiënte formule met 2 inwendige producten en 1 kruisproduct. Dit vereist minder zwevende kommagetal operaties dan de traditionele aanpak.

$$\vec{v}' = 2(\vec{q} \cdot \vec{v})\vec{q} + (q_w^2 - \vec{q} \cdot \vec{q})\vec{v} + 2q_w(\vec{q} \times \vec{v})$$

Rotatie quaternionen combineren

2 rotatie quaternionen q_1 en q_2 kunnen gecombineerd worden door ze te vermenigvuldigen.

Let op, het vermenigvuldigen van quaternion is niet-commutatief.

$$q'_1 = q_2 q_1$$

VOORBEELD

In de praktijk komt het voor dat we een oriëntatie willen bijhouden van een object, zoals een vliegtuig.

Deze oriëntatie kan gepresenteerd worden als rotatie quaternion q_1

Vervolgens willen we de oriëntatie manipuleren met een rotatie van x aantal graden om een willekeurige as.

Deze rotatie kunnen we omzetten naar een ‘tijdelijk’ rotatie quaternion q_2

Om een nieuw oriëntatie quaternion q'_1 te berekenen vermenigvuldigt men de 2 rotatie quaternionen q_1 en q_2 (rechts naar links)

$$q'_1 = q_2 q_1$$

Rotatie quaternion omzetten naar rotatie matrix

We hebben gezien dat een vector \vec{v} geroteerd kan worden met een rotatie quaternion q met behulp van het Sandwich product $p' = q \cdot p \cdot q^*$, waarbij $p = (0, \vec{v})$

Het is ook mogelijk om een rotatie matrix af te leiden van een rotatie quaternion.

Dit is nuttig omdat GLSL geen ondersteuning biedt voor quaternionen, maar wel voor matrices.

De van quaternion q afgeleide rotatiematrix R kan vervolgens worden vermenigvuldigt met een vector \vec{v} .

$$R = \begin{bmatrix} 1 - 2(q_y^2 - q_z^2) & 2(q_x q_y - q_w q_z) & 2(q_w q_y + q_x q_z) & 0 \\ 2(q_x q_y + q_w q_z) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_w q_x) & 0 \\ 2(q_x q_z - q_w q_y) & 2(q_w q_x + q_y q_z) & 1 - 2(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

VOORBEELD

We willen vector \vec{v} roteren met een rotatie beschreven als een quaternion q

Eerst zetten we rotatie quaternion q om naar een rotatie matrix R

Vervolgens passen we de regels voor de vector-matrix vermenigvuldiging toe zoals beschreven in MATRIX

$$\vec{v}' = R \vec{v}$$
$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ 1.0 \end{bmatrix} = \begin{bmatrix} 1 - 2(q_y^2 - q_z^2) & 2(q_x q_y - q_w q_z) & 2(q_w q_y + q_x q_z) & 0 \\ 2(q_x q_y + q_w q_z) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_w q_x) & 0 \\ 2(q_x q_z - q_w q_y) & 2(q_w q_x + q_y q_z) & 1 - 2(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1.0 \end{bmatrix}$$

Euler hoeken omzetten naar quaternion

Euler hoeken worden veel gebruikt voor gebruikersinput (frond end), want ze zijn in 3D, dus intuïtief. Omdat Euler hoeken gevoelig zijn voor Gimball lock, is het niet verstandig om rotaties hiermee uit te voeren.

In de back end wordt daarom vaak gerekend met quaternioen, in 4D.

Het omzetten van Euler oriëntatie naar quaternion gaat als volgt:

De eerste stap is om van elke hoek (ϕ, θ, ψ) een rotatie quaternion te maken.

$$q_x = \begin{bmatrix} \cos(\phi/2) \\ \sin(\phi/2) \\ 0 \\ 0 \end{bmatrix} \quad q_y = \begin{bmatrix} \cos(\theta/2) \\ 0 \\ \sin(\theta/2) \\ 0 \end{bmatrix} \quad q_z = \begin{bmatrix} \cos(\psi/2) \\ 0 \\ 0 \\ \sin(\psi/2) \end{bmatrix}$$

De resulterende rotatie quaternioen kunnen als volgt gecombineerd worden.

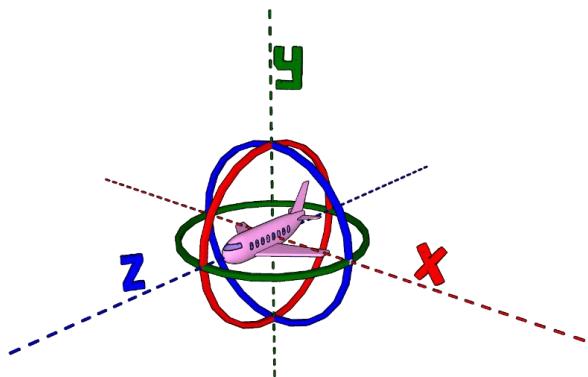
We spreken af:

X-as:	Y-as:	Z-as:
$\cos(\phi/2) = c_\phi$	$\cos(\theta/2) = c_\theta$	$\cos(\psi/2) = c_\psi$
$\sin(\phi/2) = s_\phi$	$\sin(\theta/2) = s_\theta$	$\sin(\psi/2) = s_\psi$

$$q_{zyx} = \begin{bmatrix} c_\phi \cdot c_\theta \cdot c_\psi + s_\phi \cdot s_\theta \cdot s_\psi \\ s_\phi \cdot c_\theta \cdot c_\psi - c_\phi \cdot s_\theta \cdot s_\psi \\ c_\phi \cdot s_\theta \cdot c_\psi + s_\phi \cdot c_\theta \cdot s_\psi \\ c_\phi \cdot c_\theta \cdot s_\psi - s_\phi \cdot s_\theta \cdot c_\psi \end{bmatrix}$$

Vliegtuig voorbeeld

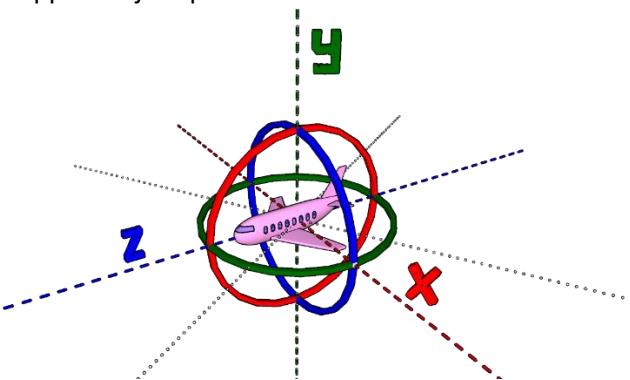
Stel, we willen de oriëntatie van een vliegtuig beschrijven met quaternioen.



1. Oriëntatie quaternion aanmaken op basis van Euler hoeken (orthogonaal).
2. We willen het vliegtuig 32 graden roteren om de Y-as, dus zetten we deze (Euler) rotatie om naar een tijdelijk rotatie quaternion
3. Oriëntatie quaternion vermenigvuldigen met rotatie quaternion, dit resulteert in een nieuwe oriëntatie quaternion.
4. Rotatiematrix afleiden van nieuwe oriëntatie quaternion.

Deze rotatiematrix kan vermenigvuldigd worden met de vertices van het vliegtuig, maar ook met een richtingsvector om de richting te bepalen (handig voor translatie).

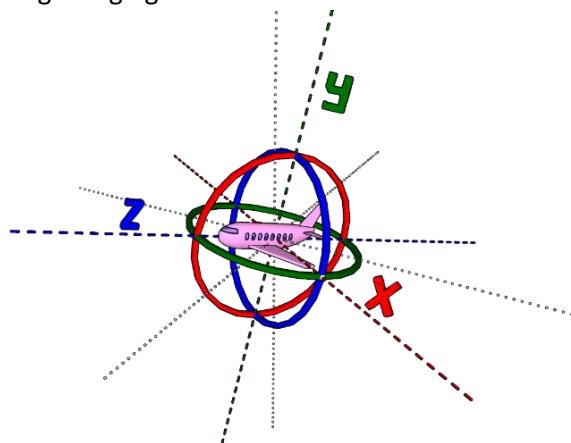
De vage gestippelde lijn represeneert de oude oriëntatie.



Het oriëntatie quaternion kan vervolgens gemanipuleerd door deze te vermenigvuldigen met andere rotatie quaternioen.

Bijvoorbeeld een rotatie om de X-as van 15 graden omhoog (beschreven als een Euler rotatie, intuïtief). Ook hier geldt weer:

1. Euler rotatie omzetten naar rotatie quaternion
2. rotatiematrix afleiden van nieuw oriëntatiequaternion en deze matrix gebruiken voor vector vermenigvuldigingen.



COÖORDINATENSTELSELS

We weten nu wat vectors en matrices zijn en hoe we van het ene coördinatenstelsel naar het andere transformeren.

In OpenGL willen we uiteindelijk de 3D objecten projecteren naar clip space, een soort van 3 dimensionale projectie.

Voordat we daadwerkelijk de 3D data kunnen projecteren, moeten alle 3D objecten zich eerst in het juiste coördinatenstelsel bevinden met als oorsprong de camera, dit coördinatenstelsel heet view space.

Om de data in view space te krijgen gaan er meerdere transformaties aan vooraf.

In OpenGL onderscheiden we de volgende coördinatenstelsels:

Coördinaatstelsel	Representatie
Model space	Coördinatenstelsel ten opzichte van de oorsprong van het 3D object
World space	Coördinatenstelsel ten opzichte van de globale oorsprong (wereld waar het 3D object zich in bevindt)
View space	Coördinatenstelsel ten opzichte van de camera (of het oog)
Clip space	Coördinatenstelsel waar de vertex posities zich in bevinden nadat ze geprojecteerd zijn. Clip space is als het ware een eenheidskubus met 3 dimensies, de derde dimensie kan geïnterpreteerd worden als een soort 'Z-buffer' (depth test; overlapping). Alle posities buiten deze eenheidskubus worden niet geprojecteerd. Zodra de vertex shader een positie schrijft naar <code>gl_position</code> , verwacht OpenGL deze coördinaat in clip space
Normalized Device Coordinate (NDC) space	In principe bevinden coördinaten zich in NDC space als de clip space coördinaten gedeeld worden door hun eigen w-component. Dit heeft als gevolg dat w altijd gelijk wordt aan 1.0, als w eerst niet gelijk was aan 1.0, dan worden de xyz-componenten geschaald met $1.0/w$.



Representatie coördinatenstelsel

Het is mogelijk om de positie (translatie) en oriëntatie (rotatie) van een coördinatenstelsel te representeren in 1 enkele matrix.

De eerste 3 elementen van de eerste 3 kolommen zijn vectoren die een oriëntatie representeren.

Deze 3 vectoren staan altijd haaks op elkaar en zijn meestal eenheidsvectoren (om bijvoorbeeld een coördinatenstelsel representeren).

Als de vectoren van een eenheidslengte zijn noemen we dit orthonormaal, indien dit niet het geval is noemen we dat orthogonaal.

Onderstaande matrix represeneert een gecombineerde rotatie-translatie matrix.

De 3×3 submatrix linksboven (α) represeneert een rotatie (of oriëntatie) en de laatste kolom (β) represeneert een translatie (of positie).

$$\begin{bmatrix} \alpha_{0,0} & \alpha_{1,0} & \alpha_{2,0} & \beta_0 \\ \alpha_{0,1} & \alpha_{1,1} & \alpha_{2,1} & \beta_1 \\ \alpha_{0,2} & \alpha_{1,2} & \alpha_{2,2} & \beta_2 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Om een coördinaat van een 3D object van model space naar clip space te transformeren gebruikt men verschillende transformatie matrices, hieronder wordt hier dieper op ingegaan.

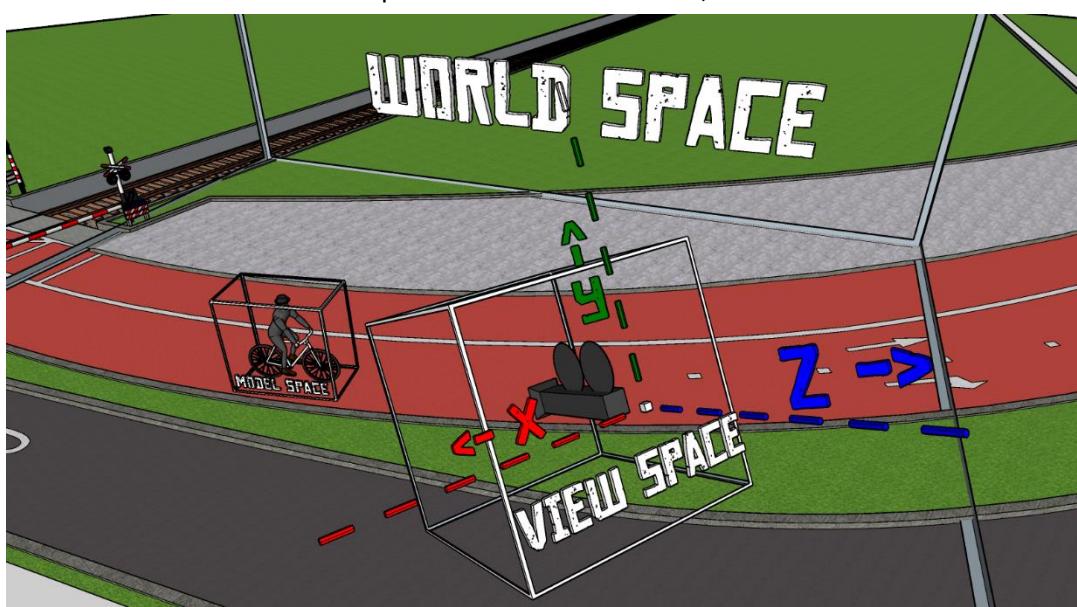
Model-view transformatie

De model view transformatie wordt gebruikt om coördinaten van model space (een fiets bijvoorbeeld) naar world space te transformeren.

Standaard is de OpenGL "camera" (view space) gericht op de negatieve Z-as (richting de kijker, het "scherm").

Met andere woorden, de standaard richting van de OpenGL is de richting van de webcam in een laptop.

In theorie draait de wereld in OpenGL om de camera heen, en niet de camera om de wereld.



De model-viewmatrix is eigenlijk een concattenatie van 2 transformatie matrices:

- Model-world transformatie: model space -> world space
- World-view transformatie: world space -> view space

Model-world transformatie

De model-world transformatie transformeert een object van model space naar world space.

Elk object heeft zijn eigen coördinatenstelsel ten opzichte van de wereld en dus een eigen rotatie-translatie matrix.

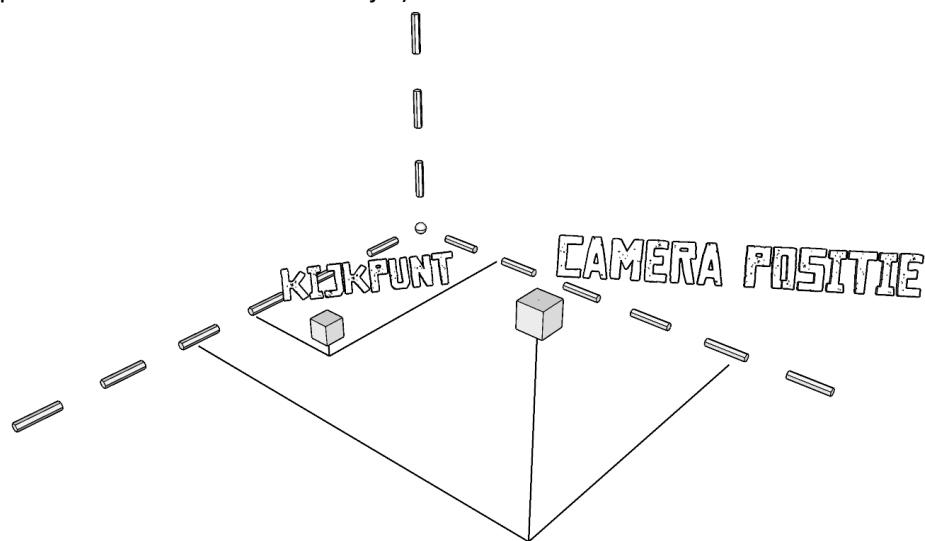
World-view transformatie

De world-view transformatie transformeert elk object van world space naar view space.

View space is het coördinatenstelsel met de positie en oriëntatie van de camera.

Met behulp van een view matrix kunnen we het coördinatenstelsel creëren van de camera ten opzichte van de wereld.

Om een view matrix te maken hebben we in principe 2 punten nodig: een camerapositie en een kijkpunt (positie waar de camera heen kijkt).



Om de camera op de juiste manier te oriënteren hebben we 2 vectors nodig, een vector die omhoog wijst (zodat boven en onder onderscheiden kunnen worden) en een vector die van de camerapositie naar het kijkpunt wijst.

De eenheidsvector die omhoog wijst noemen we u .

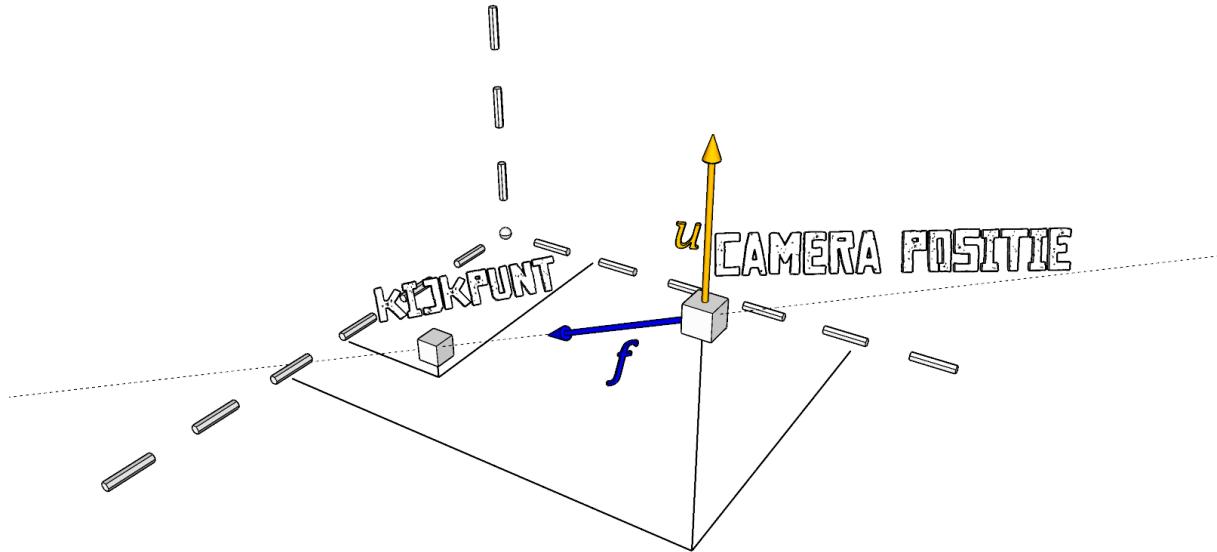
U weten we aannemen dat deze vector altijd omhoog wijst.

De eenheidsvector van de camera naar het kijkpunt noemen we f .

f representeert onze Z-as.

f kan als volgt berekent worden:

$$f = \frac{\text{kijkpunt} - \text{camerapositie}}{\|\text{kijkpunt} - \text{camerapositie}\|}$$

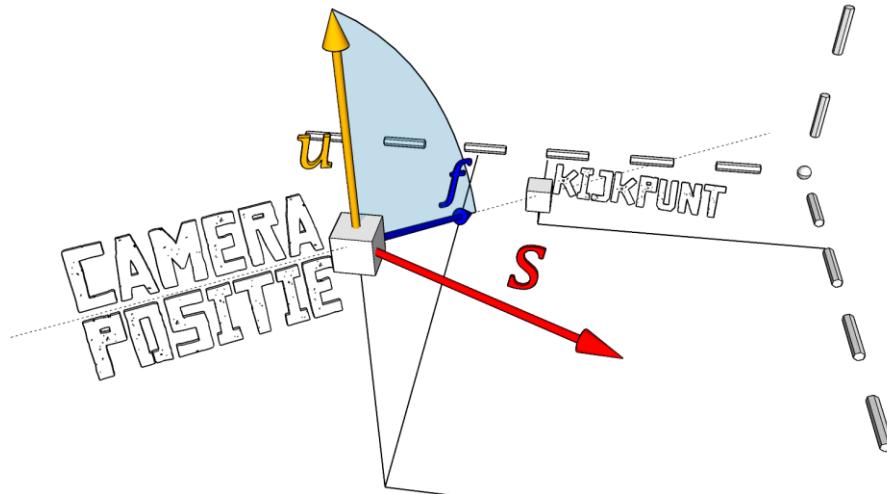


Nu we vector f hebben berekent kunnen we de zijwaartse eenheidsvector berekenen die de X-as gaat representeren.

Als we het kruisproduct van f en u nemen krijgen we een nieuwe vector die haaks op het denkbeeldige oppervlak tussen f en u staat.

Deze zijwaartse vector noemen we s .

$$s = f \times u$$

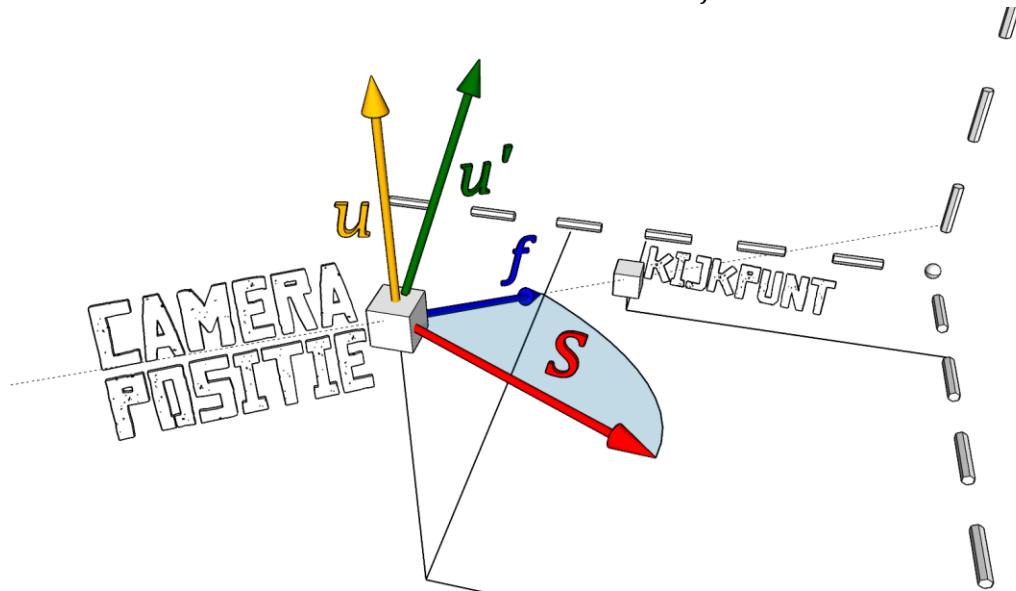


Er is nog 1 vector die ontbreekt: de vector die omhoog wijst en loodrecht staat op het oppervlak van vector s en f .

We creëren een nieuwe vector u' die het assenstelsel compleet maakt met een Y-as.

Ook hier kunnen we het kruisproduct toepassen.

$$u' = s \times f$$



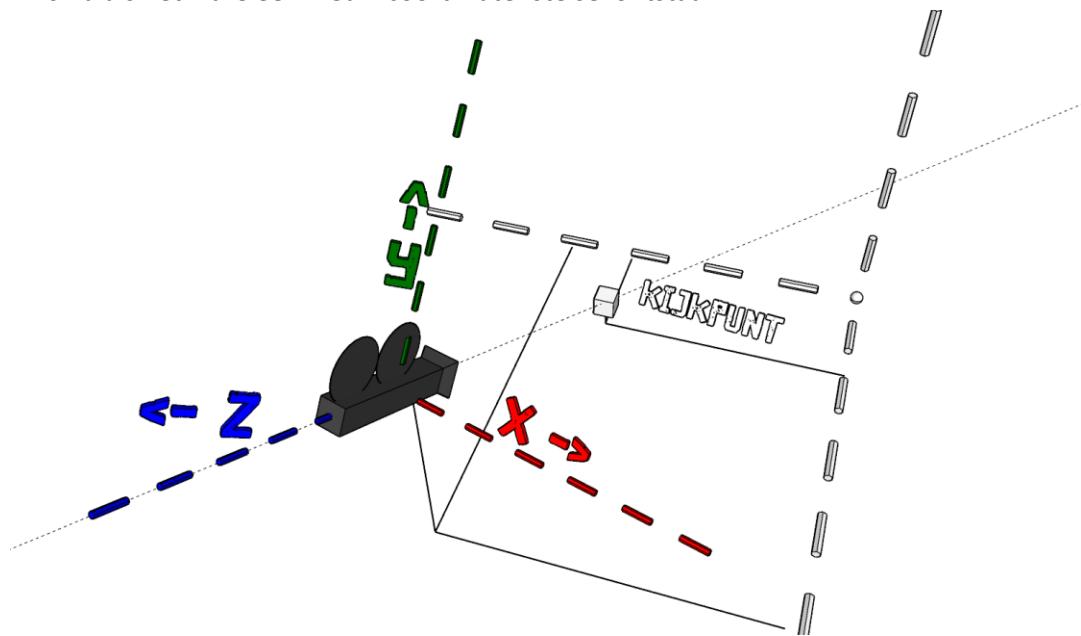
We hebben nu 3 eenheidsvectoren die loodrecht op elkaar staan: $s u' f$

Vervolgens kunnen we de view matrix bouwen op basis van de oriëntatie en de positie van de camera in world space.

$$\text{view matrix} = \begin{bmatrix} s_x & u_x' & f_x & -camera_x \\ s_y & u_y' & f_y & -camera_y \\ s_z & u_z' & f_z & -camera_z \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Als we de model-world matrix vermenigvuldigen met de world-view matrix krijgen we 1 gecombineerde model-viewmatrix.

Er is nu als het ware een nieuw coördinatenstelsel ontstaan:



Projectie transformatie

Als alle 3D coördinaten zich in view space bevinden kunnen we de coördinaten omzetten naar clip space.
Alle objecten buiten clip space vallen zijn niet geprojecteerd.

Met behulp van de projectiematrix kan berekent worden welke vertices binnen het ‘frustum’ van de camera vallen en welke geprojecteerd worden binnen de ‘clip space kubus’.

We onderscheiden 2 soorten projecties: de parallel projectie en de perspectief projectie.

Perspectief projectie

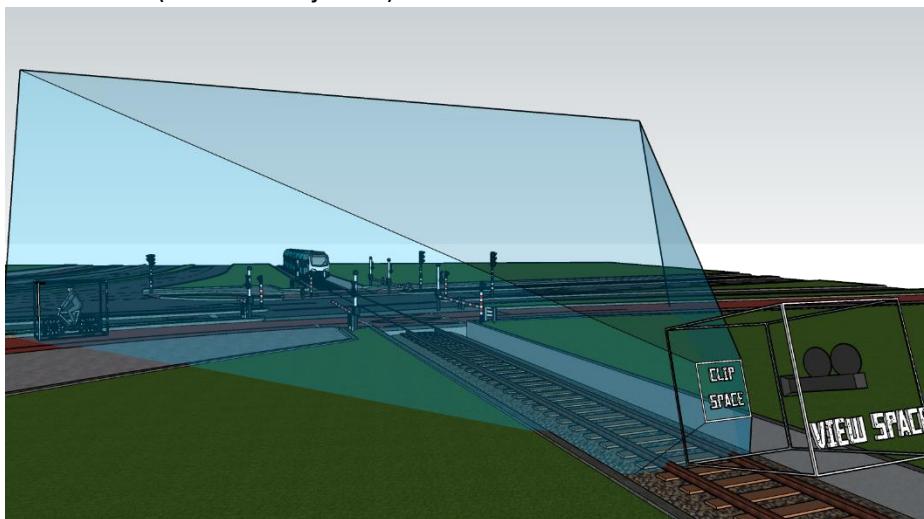
Een perspectief projectie is vrij vanzelfsprekend, het is eigenlijk op de manier waarop wij mensen de wereld om ons heen waarnemen met onze ogen.

Hoe groter de afstand tussen het oog en het object, hoe kleiner het object wordt waargenomen.

Een treinspoor ziet er als volgt uit in een perspectief projectie:



Het frustum (zichtbare objecten) ziet er dan uit als een soort trechter



Parallel projectie

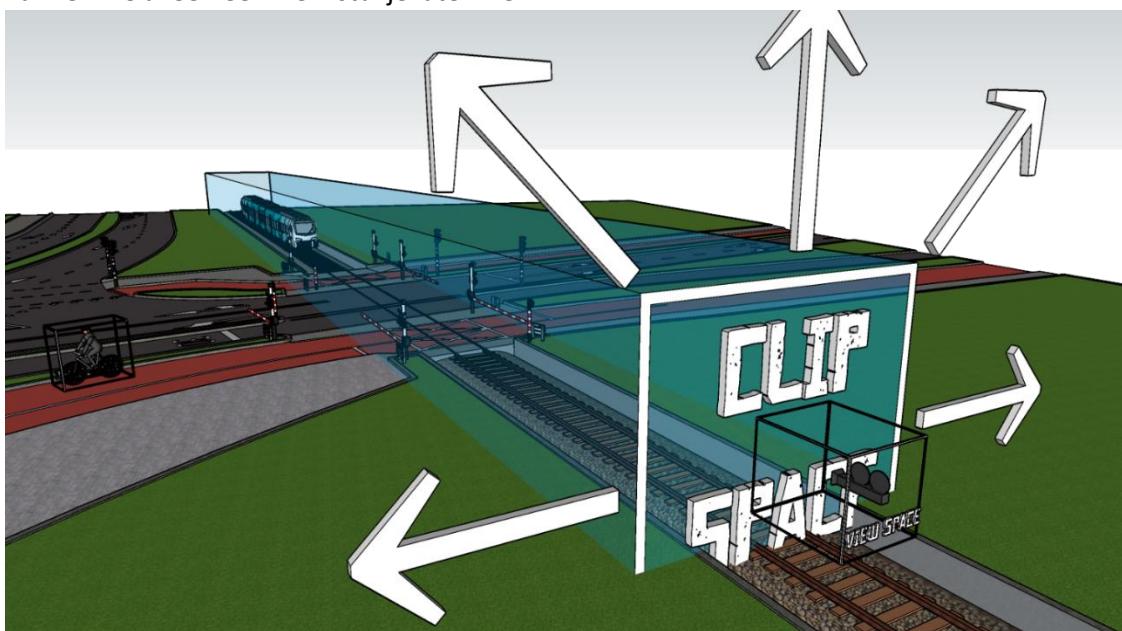
Bij een perspectief of orthografische projectie worden alle vertices parallel geprojecteerd op het scherm. De objecten zijn, onafhankelijk van de kijkafstand, altijd even groot.

Een parallel projectie kan handig zijn om bijvoorbeeld een plattegrond te projecteren met de exacte dimensies van de geometrie.

Als een treinspoor parallel projecteren ziet dit er als volgt uit:



In theorie is ons frustum en projectievlak oneindig hoog en breed. Omdat een computer scherm dit niet kan tonen kunnen we alleen een klein stukje laten zien.

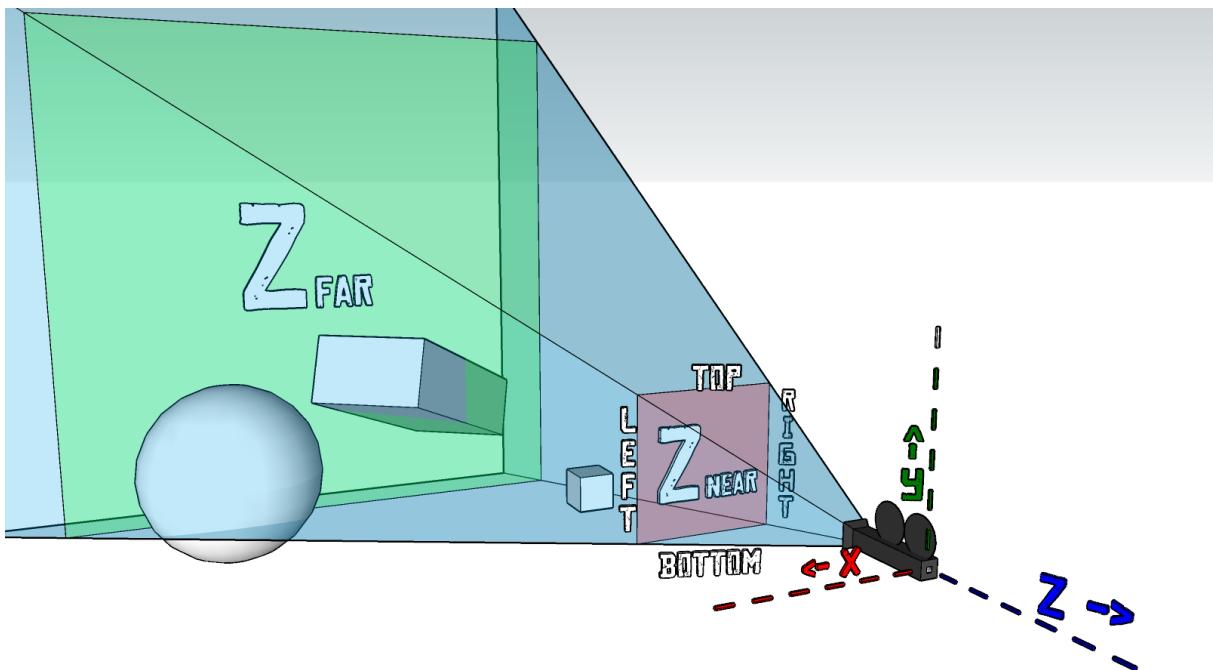


Projectiematrices

Om view space coördinaten om te zetten in clip space coördinaten hebben we weer een transformatiematrix nodig.

Een projectiematrix kan gebouwd worden op basis van de volgende parameters:

- Z_{far} is de maximale afstand van objecten die geprojecteerd worden, dit kunnen we representeren als een groen vlak in het frustum.
- Z_{near} is de minimale afstand van objecten die geprojecteerd worden, dit kunnen we representeren als een rood vlak in het frustum.
- Top, Right, Left en Bottom zijn de uiterste coördinaten van het rode vlak Z_{near}



Een perspectief projectiematrix ziet er als volgt uit:

$$\begin{bmatrix} \frac{2 \cdot Z_{near}}{right - left} & 0.0 & \frac{right + left}{right - left} & 0.0 \\ 0.0 & \frac{2 \cdot Z_{near}}{top - bottom} & \frac{top + bottom}{top - bottom} & 0.0 \\ 0.0 & 0.0 & \frac{Z_{near} + Z_{far}}{Z_{near} - Z_{far}} & \frac{2 \cdot Z_{near} \cdot Z_{far}}{Z_{near} - Z_{far}} \\ 0.0 & 0.0 & -1.0 & 0.0 \end{bmatrix}$$

Een parallel projectiematrix ziet er zo uit:

$$\begin{bmatrix} \frac{2}{right - left} & 0.0 & 0.0 & \frac{left + right}{left - right} \\ 0.0 & \frac{2}{top - bottom} & 0.0 & \frac{bottom + top}{bottom - top} \\ 0.0 & 0.0 & \frac{2}{Z_{near} - Z_{far}} & \frac{Z_{near} + Z_{far}}{Z_{far} - Z_{near}} \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Clip space en Z-buffer

Alle geometrie die zich tussen het rode oppervlak én het groene oppervlak én binnen het frustum bevinden worden geprojecteerd in clip space.

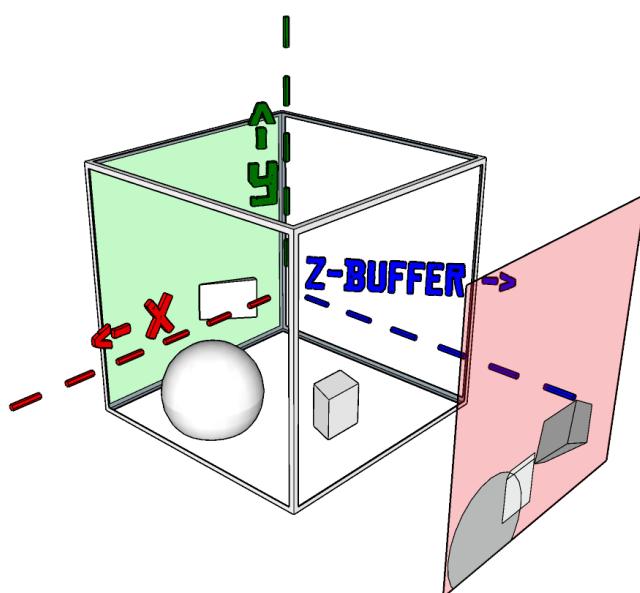
Het volgende model toont een daadwerkelijke perspectief projectie in een 3D kubus, dit is als het ware de definitie van clip space.

Elke dimensie in clip space bevindt zich tussen 1 en -1.

Zoals te zien zijn de objecten vervormd door de perspectief projectie.

De mini kubus vlakbij de camera lijkt in verhouding bijna even groot als de veel grotere balk en bol.

Het rode 2D vlak geeft weer hoe de verschillende objecten elkaar overlappen, hoe grijzer het object hoe groter de diepte of Z-buffer.



Viewport transformatie

De laatste transformatie is de viewport transformatie.

Eerst worden de clip space coördinaten omgezet naar Normalized Device Space, dit gebeurt door alle clip space coördinaten te delen door hun eigen W-component.

Vervolgens kan worden de NDC coördinaten omgezet naar scherm coördinaten met behulp van de viewport transformatie.

NDC coördinaten hebben een aspect ratio van 1:1 (vierkant).

Een scherm is in de praktijk vaak rechthoekig.

Deze viewport transformatie 'rekt' de NDC kubus uit zodat deze overeenkomt met de aspect ratio van het scherm.

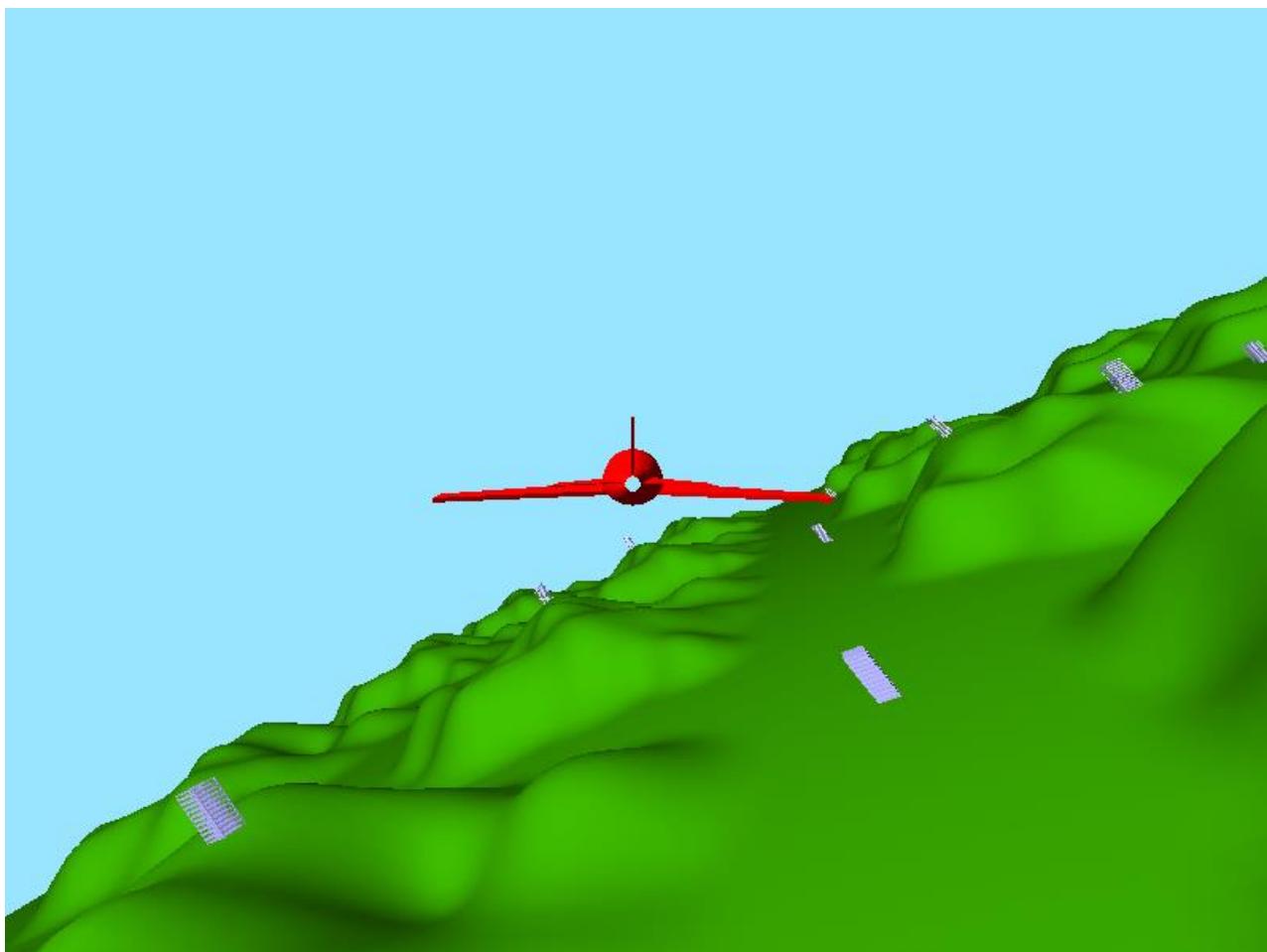
5. RENDEREN IN 3D

In dit hoofdstuk zullen we een uitgebreid programma schrijven voor een vliegsimulatie in 3D.

We zullen hiervoor alle eerder opgedane kennis bij toepassen.

Daarvoor zijn een aantal dingen nodig:

- 3D bestanden
- Shaders voor belichting
- Transformatie matrices (model-view-projection)
- Logica abstract maken zodat we de deel componenten gescheiden houden
 - Model class; beschrijft een te renderen object; beheert componenten zoals de model matrix, positie en richting
 - Camera class; beschrijft de verschillende camera's; beheert de view en projectiematrix
 - Light class; beheert belichtingscomponenten zoals een lichtbron positie
 - Shader class; bevat alle logica voor het aanmaken van een shader object
- Main class



3D BESTANDEN

Voor de voorbeelden die we tot nu toe hebben gemaakt gebruikten we steeds een simpele rechthoek, in 3D is in de meeste gevallen de geometrie vele malen complexer.

Het is niet efficiënt om elk 3D model te hardcoden zoals onze rechthoek, daarom gaan we vanaf nu vertex data van een model inladen uit een bestand.

Dit heeft als bijkomend voordeel dat we zelf gemakkelijk modellen kunnen ontwerpen in 3D tools zoals SketchUp en Blender.

Vervolgens kunnen we dit bestand uitlezen in ons programma en de data structureren voor OpenGL.

OBJ bestand

In deze cursus zullen we OBJ bestanden gebruiken, OBJ is een open standaard voor het opslaan van 3D data.

Een OBJ bestand heeft een bepaalde structuur.

Het bevat de vertex posities, texture coördinaten en vertex normaal vectoren.

Ook worden polygonen gedefinieerd m.b.v. van indices naar bovenstaande vertex attributen.

Componenten

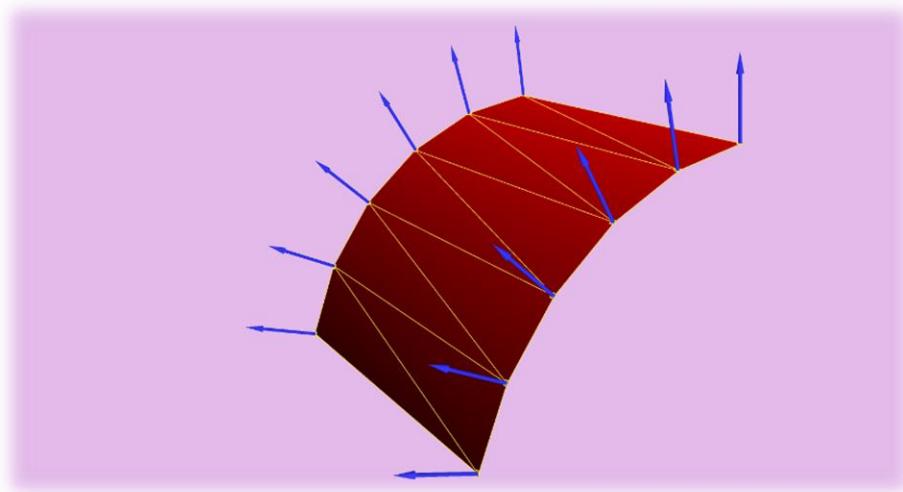
Vertex coördinaat

Een vertex coördinaat beschrijft de positie van een vertex (een polygoon heeft 3 hoeken, dus 3 vertices).

Vertex normaal

Een **vertex normaal** is een eenheidsvector die haaks op een vertex staat (blauwe pijl).

Normals worden onder andere gebruikt om belichting te berekenen.



Texture coördinaat

In deze cursus gebruiken we voorlopig nog geen textures, dus texture coördinaten zijn momenteel niet relevant.

Indices

Een polygoon is opgebouwd uit drie vertices.

Om een OBJ bestand zo klein mogelijk te houden wordt elk vertex attribuut slechts 1 keer gedefinieerd.

Toch wordt een vertex attribuut meerdere keren gebruikt, zoals 2 aangrenzende driehoeken die beide 2 vertex posities delen.

In een OBJ bestand worden polygonen gedefinieerd op basis van indices.

Voor elke vertex in een polygoon, 3 in totaal, wordt verwezen naar 3 vertex attribuut indices.

- Een vertex positie wordt aangekondigd met `v`
- Een vertex texture coördinaat wordt aangekondigd met `vt`
- Een vertex normaal wordt aangekondigd met `vn`
- Een Polygoon wordt aangekondigd met `f`
 - De vertex attribuut indices zijn gescheiden door backslashes
 - De vertices zijn gescheiden door spaties.
 - Bijvoorbeeld: `f pos1/texcoord1/norm1 pos2/texcoord2/norm1 pos3/texcoord3/norm1`

Formaat

Een voorbeeld van een OBJ bestand:

```
mtllib rcplane.mtl
g Mesh1 Group1 Model
usemtl FrontColor
v 73 15 -47.4434
vt -5.90551 -0.831578
vn 1.7465e-15 0.382683 -0.92388
v 88 12.941 -48.2963
vt -6.49606 -0.919322
vn 8.62818e-16 0.258819 -0.965926
v 73 12.941 -48.2963
vt -5.90551 -0.919322
f 1/1/1 2/2/2 3/3/2

v 88 15 -47.4434
vt -6.49606 -0.831578
f 2/2/2 1/1/1 4/4/1

v 73.2556 16.9411 -46.6394
vt -5.91557 -0.748859
f 4/4/1 1/1/1 5/5/1
```

	Vertex 1				Vertex 2				Vertex 3		
	v	vt	vn		v	vt	vn		v	vt	vn
driehoek 1 indices	1	1	1		2	2	2		3	3	2
driehoek 2 indices	2	2	2		1	1	1		4	4	1

	Vertex coördinaat (v)	Texture coördinaat (vt)	Vertex normaal (vn)
index 1	73, 15, -47.44	-5.90, -0.83	1.74, 0.38, -0.92
index 2	88, 12.94, -48.29	-6.49, -0.91	8.62, 0.25, -0.96
index 3	73, 12.94, -48.29	5.90, -0.91	n.v.t.
index 4	88, 15, -47.44	-6.49 -0.83	n.v.t.
index 5	73.25, 16.94, -46.63	-5.91, -0.74	n.v.t.

OBJ parser

Voor deze cursus is een unieke parser geschreven om OBJ bestanden in te lezen.

De parser beschikt niet over geavanceerde mogelijkheden zoals onderscheid maken in hiërarchie, maar de pure basis voor uitlezen van de nodige vertex attributen.

In het mapje exercizes zitten 2 bestanden (`objreader.h` en `objreader.cpp`).

Bij de includes voegen we de headerfile toe van de OBJ lezer.

```
#include <sb7.h>
#include <vmath.h>
#include "objloader.h"
```

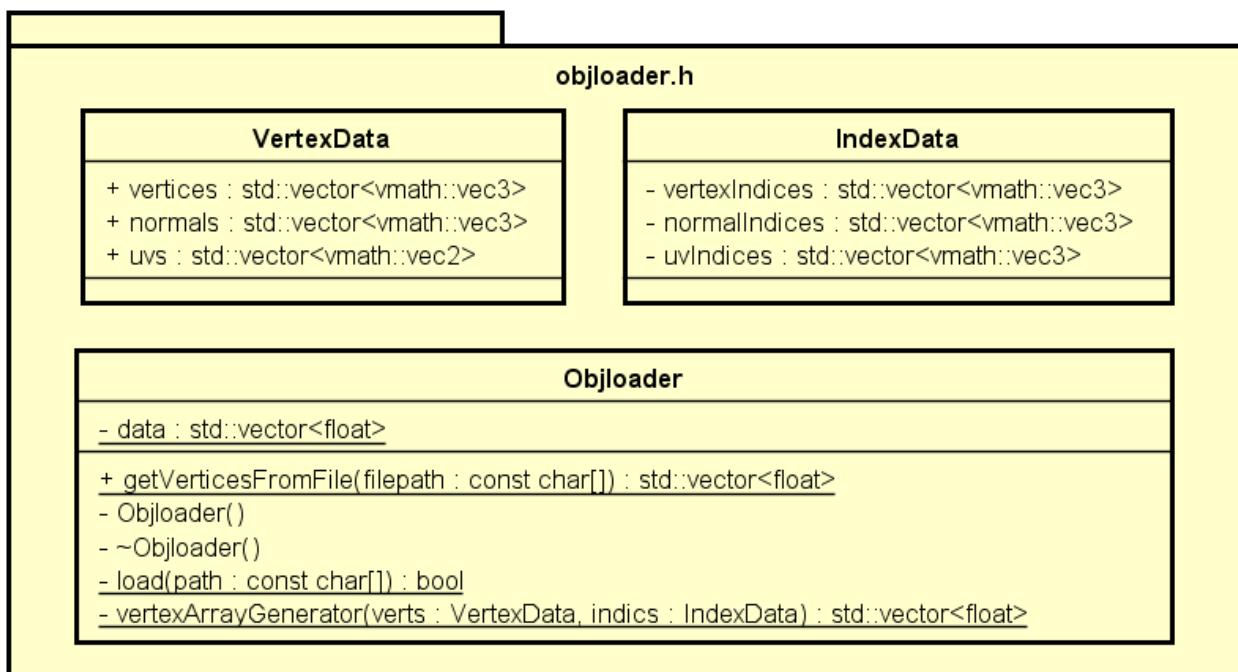
De vertex data uit het OBJ bestand slaan we op in een vector, omdat een vector een variabele lengte heeft. In deze vector worden de attributen opgeslagen van de vertex coördinaten en de vertex normaal vectoren. Merk op dat we niet langer een index buffer gebruiken.

OpenGL profiteert slechts van geïndexeerd renderen als gedeelde vertices precies dezelfde attributen bevatten.

Niet elke vertex positie heeft dezelfde vertex normaal, dit is een unieke combinatie geworden.

Een vertex positie kan meerdere keren voorkomen, maar steeds met een andere vertex normaal.

Geïndexeerd renderen voegt om deze reden weinig toe.



SHADERS

Om de modellen 'realistisch' te belichten, kunnen we shaders gebruiken.

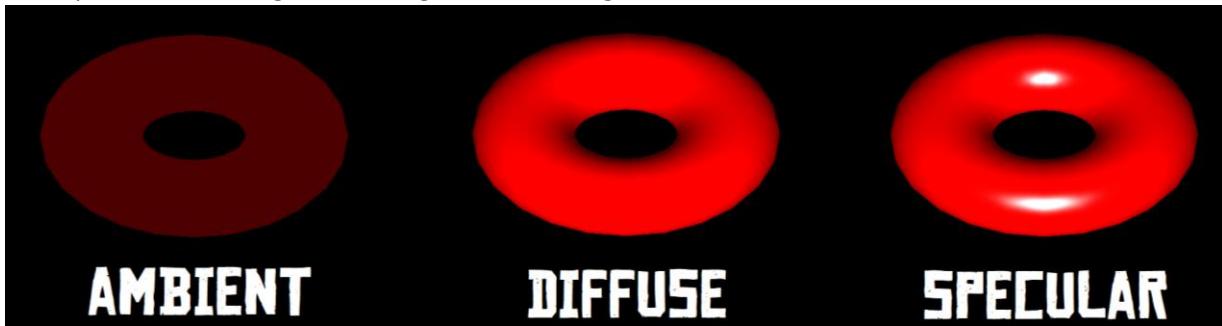
We gaan 3 verschillende shading technieken bespreken:

- Phong shading (belichtingskleur berekenen per fragment)
- Gourad shading (belichtingskleur berekenen per vertex)
- Flat shading (belichtingskleur berekenen per polygoon)

Belichtingscomponenten

Elk van de bovenstaande shading technieken is opgebouwd uit 3 componenten; de ambient, diffuse en specular component. Het resultaat krijgt men door alle 3 componenten bij elkaar op te tellen.

De implementatie hangt af van de gekozen shading techniek.



Ambient

Zonder een lichtbron zijn objecten vaak niet helemaal donker, maar wel zichtbaar in de omgeving.

Om dat effect te krijgen is er een factor nodig die de object kleur donkerder laat lijken.

Als deze verdonkeringsfactor vermenigvuldigt wordt met de oorspronkelijke object kleur, dan krijgt men de *ambient component*.

Diffuse

Diffuse belichting heeft een bepaald effect waarbij de invalshoek van een lichtbron de kleurintensiteit van een oppervlak bepaald.

De diffuse factor is afhankelijk van de hoek tussen 2 vectoren; de normaalvector en de lichtvector.

Hoe groter de diffuse factor, hoe lichter de kleur, hoe kleiner de diffuse factor, hoe donkerder de kleur.

1. Lichtvector berekenen

De lichtvector is de vector die wijst van de lichtbron naar de vertex positie.

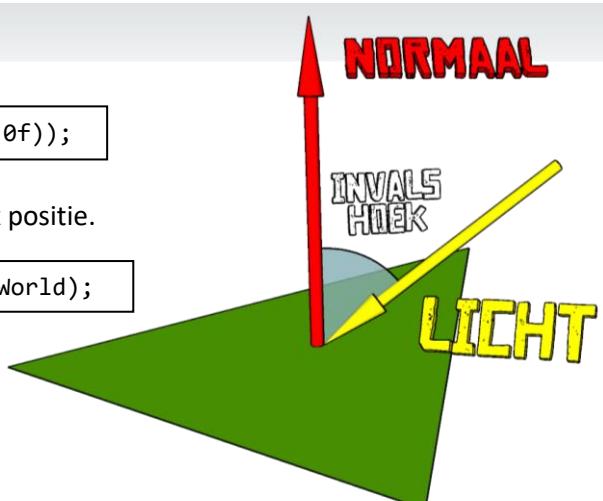
De lichtvector kan berekent worden m.b.v. de vertex positie en de lichtbron positie (beide in world space).

De vertex positie moet eerst worden vermenigvuldigt met de model matrix om de world space coördinaten te bemachtigen.

```
vertPosWorld = vec3(model * vec4(vertPos, 1.0f));
```

De lichtvector is de lichtbron positie min de vertex positie.

```
vec3 lightVec = normalize(lightPos - vertPosWorld);
```



2. Normaalvector bepalen

Naast een vertex positie hebben we ook de vertex normaal (per vertex).

De vertex normaal is een normaalvector die haaks op een vertex staat.

Omdat de belichting afhankelijk is van de wereld moet de normaal moet eerst omgezet worden naar world space.

Een normaal represeneert alleen een richting, dus als we de reguliere model matrix vermenigvuldigen met de normal kan dit resulteren in ongewenste translatie en schaling.

Het translatiegedeelte van de model matrix (rechterkolom) kan geëlimineerd worden door de bovenste 3x3 submatrix te nemen.

De schaling kan worden tegengegaan door de getransponeerde en inverse van de model matrix te berekenen.

Transponeren houdt in dat de rijen en kolommen van de matrix worden omgedraaid.

Dit resulteert in een zogenoemde *normal matrix*; een model matrix zonder ongewenste schaling en translatie.

De juiste normaalvector kan berekend worden door de oorspronkelijke normal te vermenigvuldigen met de *normal matrix*.

```
vertNormWorld = mat3(transpose(inverse(model))) * vertNorm;
```

Vervolgens kan de normaalvector ge-output worden naar de fragment shader.

Hier wordt de normaalvector genormaliseerd.

```
vec3 normVec = normalize(vertNormWorld);
```

VERTEX NORMAAL VS POLYGOON NORMAAL

De oorspronkelijke vertex normaal staat haaks op een vertex, en niet, zoals velen denken, op een oppervlak van een polygoon.

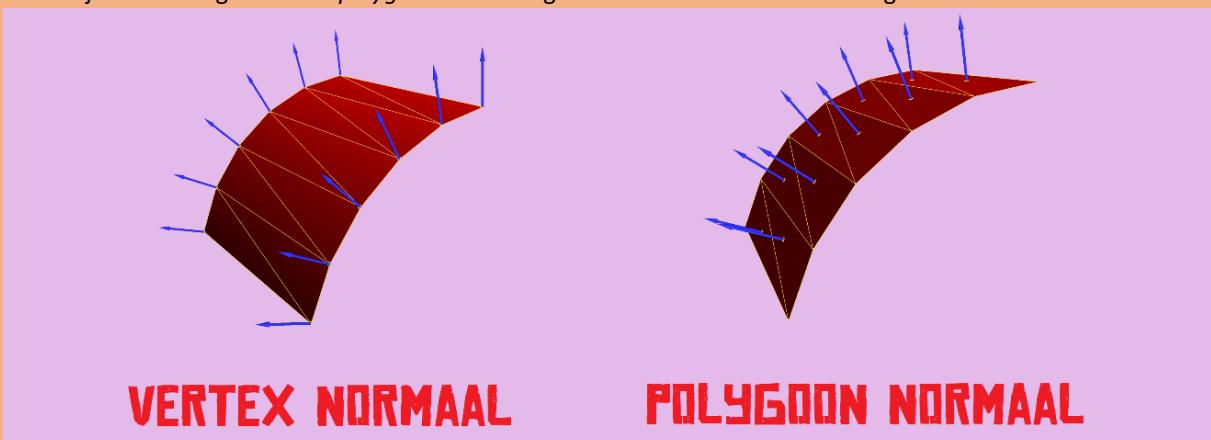
De vertex normaal is afhankelijk van alle aangrenzende polygoon oppervlakken van de vertex.

De polygoon normaal moet eerst uitgerekend worden voordat deze gebruikt kan worden.

Hier wordt later op ingegaan.

Bij Phong en Gouraud shading wordt de *vertex normaal* gebruikt als input voor de lichtberekeningen.

Bij Flat shading wordt de *polygoon normaal* gebruikt voor de lichtberekeningen.



3. Diffuse factor berekenen

De diffuse factor krijgt men door het inwendig product te nemen van de normaalvector en de lichtvector.

De diffuse factor is op deze manier afhankelijk van de invalshoek van het licht.

```
float diffuseFactor = max(dot(normVec, lightVec), 0.0f);
```

4. Diffuse component berekenen

De diffuse component kan berekend worden door de diffusefactor te vermenigvuldigen met de model kleur.

```
vec3 diffuse = diffuseFactor * modelColor;
```

Specular

Het specular component geeft een soort van reflecterend effect richting de camera.

De specular factor is afhankelijk van 2 vectors: de kijkvector en de reflectievector.

1. Kijkvector berekenen

```
vec3 viewVec = normalize(viewPos - vertPosWorld);
```

De kijkvector is de vector tussen de camerapositie en de vertex positie (in world space).

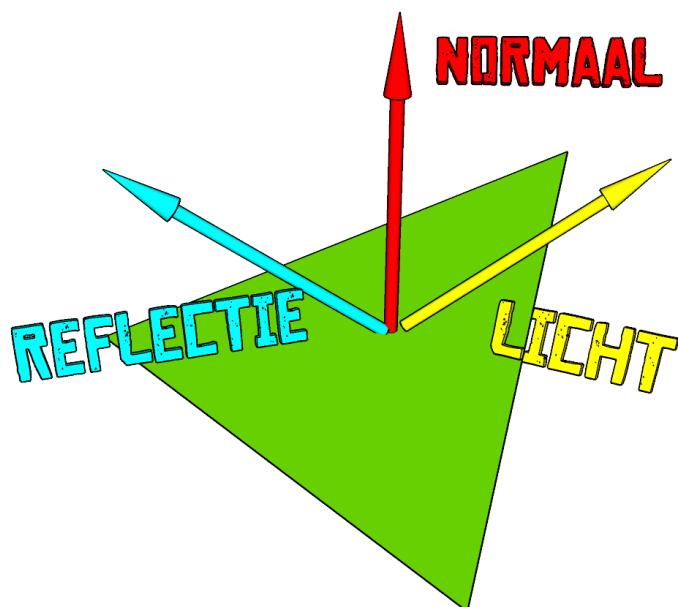
De kijkvector kan berekent worden door de vertex positie af te trekken van de camerapositie.

Net als de lichtvector en normaalvector dient de kijkvector genormaliseerd te worden.

2. Reflectievector berekenen

In GLSL zit een ingebouwde functie om de reflectie vector te berekenen, deze functie neemt als input de normaalvector en de lichtvector (in omgekeerde richting).

```
vec3 reflectVec = reflect(-lightVec, normVec);
```



3. Reflectiefactor berekenen

De specular factor is het inwendig product van de kijkvector en reflectievector tot een gegeven macht.

Hoe groter deze macht, hoe kleiner het reflectieoppervlak en vice versa.

```
float specFactor = pow(max(dot(viewVec, reflectVec), 0.0f), 16);
```

4. Glansfactor en reflectiekleur bepalen

De glansfactor bepaalt de glans van het reflectieoppervlak.

De reflectiekleur bepaalt de kleur van het reflectieoppervlak.

```
float specShine = 1.5;
vec3 specColor = vec3(1.0f, 1.0f, 1.0f);
```

5. Specular component berekenen

De specularcomponent kan berekent worden door de specularfactor te vermenigvuldigen met de glansfactor en de reflectiekleur.

```
vec3 specular = specShine * specFactor * specColor;
```

Als de ambient, diffuse en specular componenten bij elkaar opgeteld worden, dan krijgt men de resulterende belichtingscomponent.

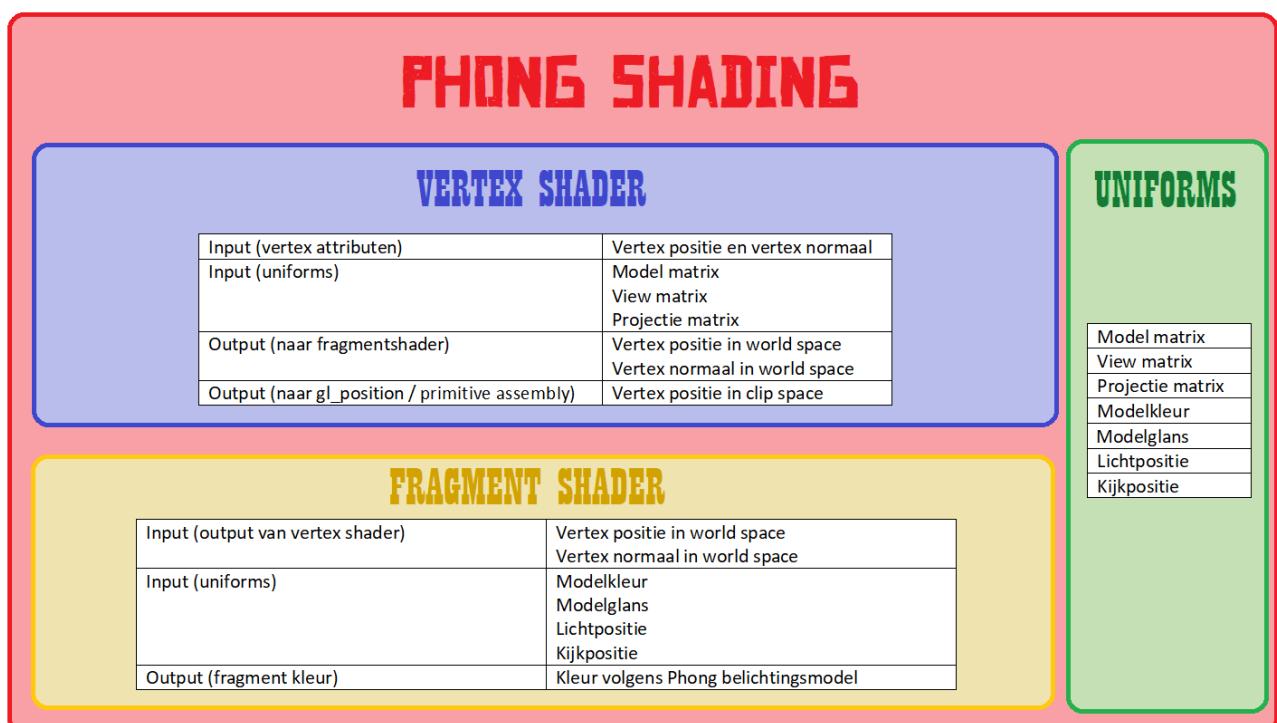
Phong shading

Bij Phong shading worden de vertex attributen (positie en normaal) gemanipuleerd in de vertex shader. De vertex positie en vertex normaal in world space worden ge-output naar de fragment shader. In de fragment shader worden de belichtingscomponenten berekent die de kleur bepalen (per fragment). Dit resulteert in een mooie effen belichting.



Overzicht shader programma

De volledige implementatie kan men vinden in `phong_vertex.glsl` en `phong_fragment.glsl`



Gourad shading

De Gourad shading lijkt heel erg op Phong shading.

Het enige verschil is dat de lichtcomponenten niet in de fragment shader berekent worden, maar in de vertex shader.

De resulterende kleur wordt vervolgens ge-output naar de fragment shader.

In de fragment shader worden geen berekeningen uitgevoerd.

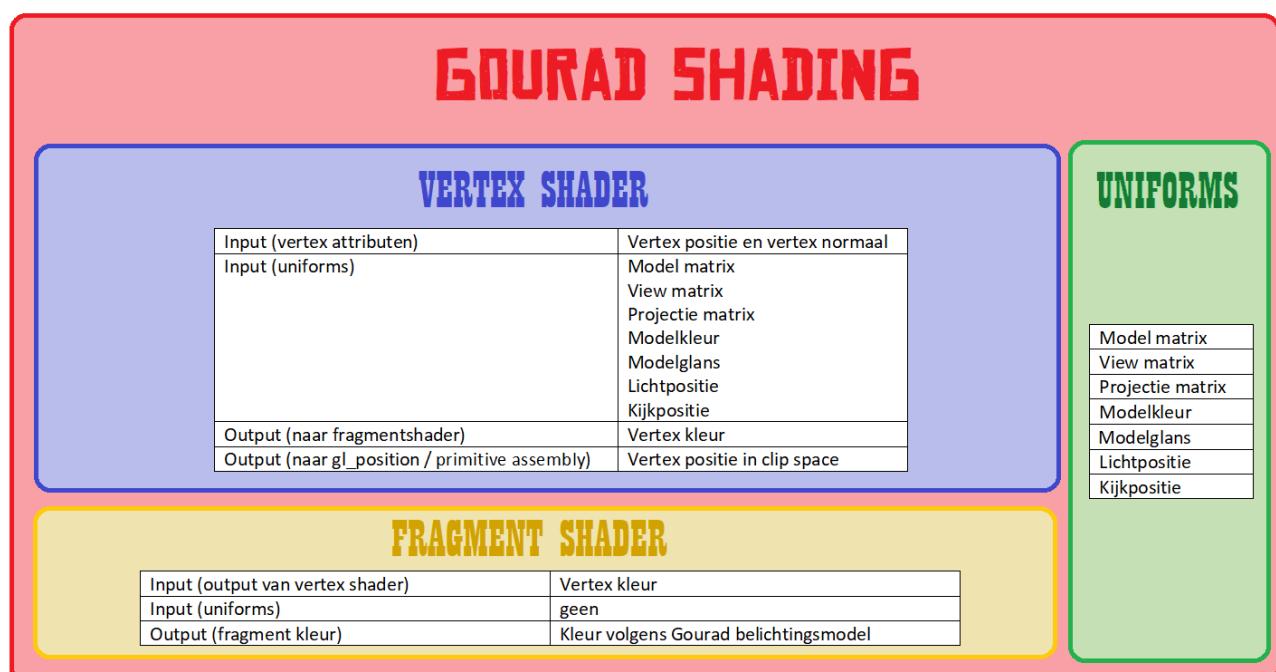
Gourad shading heeft als resultaat dat de belichting niet PER FRAGMENT wordt berekend maar PER VERTEX.

Bij Gourad shading zijn de contouren van de vertices zichtbaarder dan bij Phong shading.



Overzicht shader programma

De volledige implementatie kan men vinden in `gourad_vertex.glsl` en `gourad_fragment.glsl`



Flat shading

Flat shading is een techniek waarbij de normaal per polygoon gebruikt wordt bij de belichtingsberekeningen, en niet de normaal per vertex.

De polygoon normaal moet om die reden eerst berekend worden.

Geometry shader

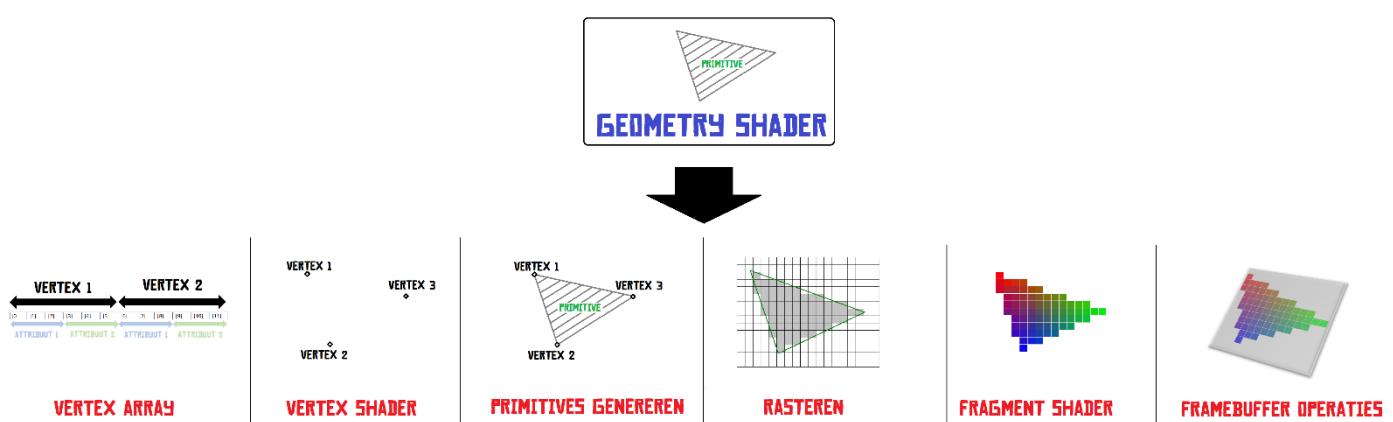
Omdat een vertex shader alleen PER VERTEX operaties uitvoert, is het niet mogelijk om daar de polygoonnormaal te berekenen.

Gelukkig bestaat er een extra (optionele) shader die het mogelijk maakt om operaties PER PRIMITIVE (polygoon) uit te voeren

Deze shader heet de geometry shader.

De geometry shader wordt uitgevoerd NA het primitives genereren, en VOOR het rasteren.

Dus tussen de vertex shader en de fragment shader in.



De input van de geometry shader is een primitive, een driehoek in dit geval.

Het formaat van de input primitive (triangles) wordt gespecificeerd in de shader.

Ook moet het formaat van de output (triangle_strip) gespecificeerd worden, inclusief het aantal vertices per primitive.

Het is, net als in elke andere shader, mogelijk om input op te halen uit de vorige shader (vertex shader) en input door te sturen naar de volgende shader (fragment shader).

Omdat een primitive in dit geval opgebouwd is uit 3 vertices, moet elke input een 3 delige array zijn.

De output wordt toegewezen per vertex, voor elk polygoon moet er dus 3 keer een vertex ge-output worden. Vervolgens wordt de primitive als geheel ge-output.

Een enkele vertex wordt ge-output vóór het commando EmitVertex().

Een primitive wordt ge-output vóór het commando EndPrimitive().

```
#version 420

layout(triangles) in; // input formaat
layout(triangle_strip, max_vertices=3) out; // output formaat

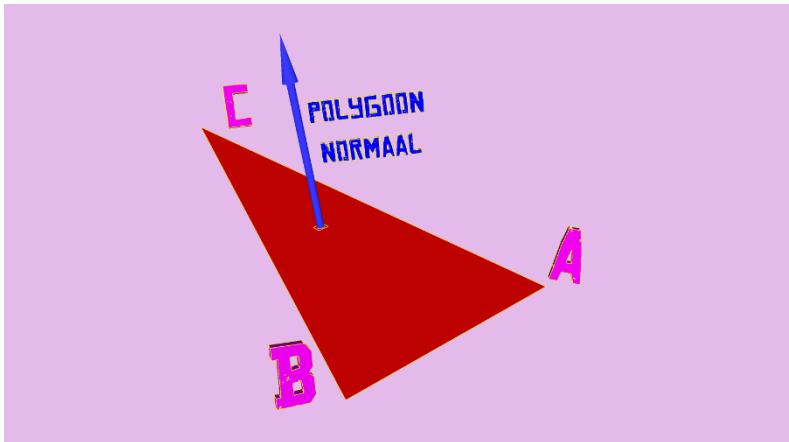
in vec3 vertPosWorld[3];
out vec3 geoPosWorld;    // vertex world positie
out vec3 geoNormWorld;   // polygoon normaal
```

Uit de vertex shader inputten we steeds de vertex world posities.

Dit zijn alle hoekpunten van een polygoon.

Uit deze hoekpunten kunnen we de polygoon normaal berekenen door het kruisproduct te nemen van 2 zijden gepresenteerd als vectoren ab en ac.

```
// oppervlakte normaal berekenen m.b.v. 2 driehoekszijden
vec3 ab = vertPosWorld[1] - vertPosWorld[0];
vec3 ac = vertPosWorld[2] - vertPosWorld[0];
geoNormWorld = normalize(cross(ab, ac));
```



De geometry shader verwacht dat we een output primitive genereren voor elke input primitive.

De clip space positie zit gebakken in gl_Position, deze hebben we eerder toegewezen in de vertex shader.

gl_Position moet per vertex, opnieuw, worden toegewezen voordat deze ge-output kan worden.

We hebben daarnaast ook een input meegegeven met daarin world space vertex posities.

Deze world space vertex posities moeten ook worden toegewezen per vertex.

De normaal die we eerder hebben berekend hoeven we niet voor elke vertex apart toe te wijzen.

Voor elke vertex in een polygoon geldt immers dat ze dezelfde normaal delen bij flat shading.

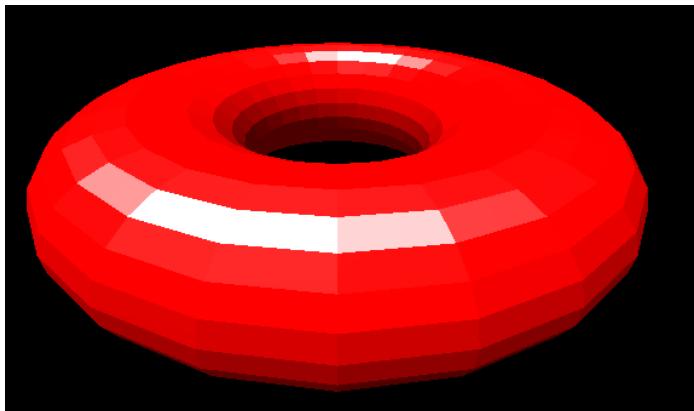
```
// primitive opnieuw opbouwen en output positie toewijzen
for(int i=0; i<3; i++)
{
    // vertex positie blijft hetzelfde
    gl_Position = gl_in[i].gl_Position;
    geoPosWorld = vertPosWorld[i];
    EmitVertex();
}
EndPrimitive();
```

In de fragment shader worden de vertex positie in world space en de nieuwe normaal ge-input om de belichtingscomponenten te berekenen.

Flat shading berekent de belichting per vlak en niet per vertex. Dit resulteert in een blokkerig resultaat waarbij de polygonen duidelijk zichtbaar zijn.

Bij Phong en Gourad shading is het belichtingseffect veel vloeinder omdat daar

We hebben eerder besproken dat de fragment shader de kleur van elk fragment interpoleert op basis van de hoekpunten (vertices) van een polygoon.



Overzicht shaderprogramma

De volledige implementatie kan men vinden in `flat_vertex.glsl`, `flat_geometry.glsl` en `flat_fragment.glsl`

FLAT SHADING

VERTEX SHADER

Input (vertex attributen)	Vertex positie
Input (uniforms)	Model matrix View matrix Projectie matrix
Output (naar geometry shader)	Vertex positie in world space
Output (naar gl_Position / primitive assembly)	Vertex positie in clip space

UNIFORMS

Model matrix
View matrix
Projectie matrix
Modelkleur
Modelglans
Lichtpositie
Kijkpositie

GEOMETRY SHADER

Input (output van vertex shader)	Array met vertex posities in world space
Input (output van primitive assembly)	Array met primitive vertex posities (<code>gl_in[]</code>)
Input (uniforms)	geen
Output (naar gl_Position / rasterizer)	Polygoon, vertex posities in clip space
Output (naar fragment shader)	Vertex positie in world space Polygoon normaal in world space

Model matrix
View matrix
Projectie matrix
Modelkleur
Modelglans
Lichtpositie
Kijkpositie

FRAGMENT SHADER

Input (output van geometry shader)	Vertex positie in world space Polygoon normaal in world space
Input (uniforms)	Modelkleur Modelglans Lichtpositie Kijkpositie
Output (fragment kleur)	Kleur volgens Flat belichtingsmodel

MODEL

De Model class bevat alle logica voor een te renderen object.

Overzicht

Headerfile (`model.h`).

```
model.h

Model

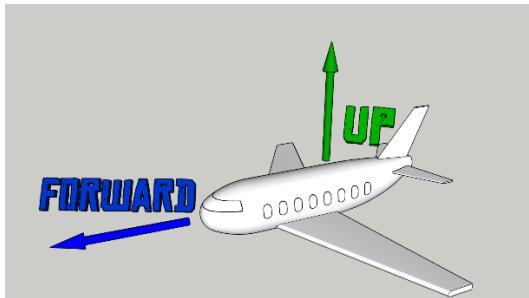
- instance_counter : int
- vao : int
- vbo : int
- shader_program : int
- vertex_array : std::vector<float>
- model_size : int
- model_color : vmath::vec3
- model_shine : float
- translation_matrix : vmath::mat4
- rotation_matrix : vmath::mat4
- scale_matrix : vmath::mat4
- model_matrix : vmath::mat4
- forward_default : const vmath::vec4
- up_default : const vmath::vec4
- forward_local : vmath::vec4
- up_local : vmath::vec4
- orientation : Quaternion
- angles : Euler
- position : vmath::vec3
- speed : float

+ Model(vertex_array : std::vector<float>, model_color : vmath::vec3)
+ Model(vertex_array : std::vector<float>, model_color : vmath::vec3, scale : float)
+ Model(vertex_array : std::vector<float>, model_color : vmath::vec3, scale : float, position : vmath::vec3)
+ ~Model()
+ getNrOfInstances() : int
- initVertices() : void
+ render(shader_program : int) : void
+ cleanUp() : void
+ setScale(factor : float) : void
+ setPosition(position : vmath::vec3) : void
+ setShine(factor : float) : void
+ getForward() : vmath::vec3
+ getUp() : vmath::vec3
+ getPosition() : vmath::vec3
+ getModelMatrix() : vmath::mat4
+ getSpeed() : float
+ printPosition() : void
+ accelerate(delta : float) : void
+ turn(delta : Euler) : void
```

Instantie variabelen

De belangrijkste instantie variabelen van de Model class zijn:

- Vertex Array Object en VertexBuffer Object
De Vertex Array Objects kunnen gekoppeld aan een shader
- Cosmetische eigenschappen zoals kleur en glansfactor
- Oriëntatie quaternion
- Richtingsvectoren (nodig voor verplaatsing en third person camera)



- Positie vector
- Snelheid
- Model matrix en bijbehorende transformatie matrices (translatie, rotatie, schalen)

Belangrijkste implementaties

De belangrijkste implementaties zullen hier worden uitgelegd, de implementatie van alle functies zijn uiteraard te vinden in `model.h`

Constructor

De basis constructor van Model vereist een vertex array en een kleur.

2 optionele parameters zijn een schaalfactor en een positie.

Verder worden in de basis constructor de instantie variabelen geinitialiseerd.

```
Model::Model(std::vector<float>& vertex_arr, vmath::vec3 model_color)
    : vertex_array(vertex_arr)
{
    this->model_color = model_color;

    initVertices();      // initialiseer vertex objecten en pointers
    instance_counter++; // instantie bij op tellen

    // standaard waarden initialiseren
    model_shine = 0.5;           // glans
    speed = 0.0f;                // snelheid
    position = vmath::vec3(0.0f, 0.0f, 0.0f); // positie (oorsprong)
    forward_local = forward_default; // lokale richting voorwaarts
    up_local = up_default;        // lokale richting omhoog

    // Euler hoeken initialiseren
    angles.roll = 0.0f;
    angles.pitch = 0.0f;
    angles.yaw = 0.0f;

    // oriëntatie quaternion aanmaken op basis van Euler hoeken
    orientation.EulerToQuat(angles);

    // transformatie matrices initialiseren
    translation_matrix = vmath::mat4::identity();
    rotation_matrix = vmath::mat4::identity();
    scaling_matrix = vmath::mat4::identity();
}
```

Vertex objecten en pointers

De initVertices() functie initialiseert een Vertex Array Object en Vertex Buffer Object op basis van de in de constructor meegegeven vertex array.

Ook worden vertex attribuut pointers geinitialiseerd voor vertex posities en vertex normaal vectors.

```
void Model::initVertices()
{
    model_size = vertex_array.size();

    glGenVertexArrays(1, &vao); // vertex array object aanmaken
    glGenBuffers(1, &vbo); // Vertex Buffer Object aanmaken

    // koppel het vertex array object aan de context
    glBindVertexArray(vao);

    // vertex buffer object koppelen aan de juiste context (GL_ARRAY_BUFFER)
    glBindBuffer(GL_ARRAY_BUFFER, vbo);

    // plaats de daadwerkelijke vertex data in het buffer geheugen
    glBufferData(GL_ARRAY_BUFFER, model_size * sizeof(float), &vertex_array[0],
    GL_STATIC_DRAW);

    // vertex attribuut pointer instellen voor positie
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);

    // vertex attribuut pointer instellen voor normal
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 *
    sizeof(float)));

    // vertex attributen inschakelen
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);

    // unbind
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
}
```

De cleanup() functie bevat alle logica voor het opruimen van vertex objecten.

Deze functie wordt altijd aangeroepen in de destructor tijdens het afsluiten van het programma.

```
void Model::cleanup()
{
    glDeleteVertexArrays(1, &vao);
    glDeleteBuffers(1, &vbo);
}
```

Render logica

De render() functie bevat alle logica per frame voor het renderen van een model.

- Positie updaten; translatie over de voorwaartse vector met een snelheidsfactor
- De model matrix wordt bij elk frame geüpdate op basis van de schaling-, rotatie- en translatiematrix.
- Vertex array object koppelen en tekencommando uitvoeren. Na het tekencommando wordt het vertex array object weer ontkoppelt

```
void Model::render(GLuint& shader_program)
{
    position += vmath::vec3(forward_local[0], forward_local[1], forward_local[2]) * speed;
    translation_matrix = vmath::translate(position);

    model_matrix = translation_matrix * rotation_matrix * scaling_matrix;

    // matrices en vectors koppelen aan uniforms
    glUniformMatrix4fv(glGetUniformLocation(shader_program, "model"), 1, FALSE, model_matrix);
    glUniform3fv(glGetUniformLocation(shader_program, "modelColor"), 1, model_color);
    glUniform1f(glGetUniformLocation(shader_program, "modelShine"), model_shine);

    glBindVertexArray(vao); // koppel het vertex array object
    glDrawArrays(GL_TRIANGLES, 0, model_size / 2); // teken commando
    glBindVertexArray(0); // ontkoppel het vertex array object
}
```

Rotatie logica

De turn() functie bevat alle logica voor het manipuleren van de huidige oriëntatie

De huidige oriëntatie is gepresenteerd als een quaternion.

Het oriëntatie quaternion wordt gemanipuleerd door deze te vermenigvuldigen met een tijdelijk rotatie quaternion.

Het tijdelijke rotatie quaternion wordt gemaakt op basis van intuïtieve Euler hoeken.

Vervolgens kan de rotatie matrix worden afgeleid van het oriëntatie quaternion.

De modelmatrix, de voorwaartse- en omhoogvector, kunnen vervolgens geüpdate worden door deze te vermenigvuldigen met behulp van deze rotatie matrix.

```
void Model::turn(Euler angles)
{
    // "tijdelijk" rotatie quaternion
    Quaternion rotation;

    // input als Euler hoeken (intuïtief voor gebruiker)
    rotation.EulerToQuat(angles);

    // oriëntatie quaternion vermenigvuldigen met rotatie quaternion
    orientation = rotation * orientation;

    // rotatie matrix afleiden van oriëntatie quaternion
    rotation_matrix = orientation.getRotationMatrix();

    // voorwaartse vector en omhoog vector berekenen
    forward_local = forward_default * rotation_matrix.transpose();
    up_local = up_default * rotation_matrix.transpose();
    normalize(forward_local);
    normalize(up_local);
}
```

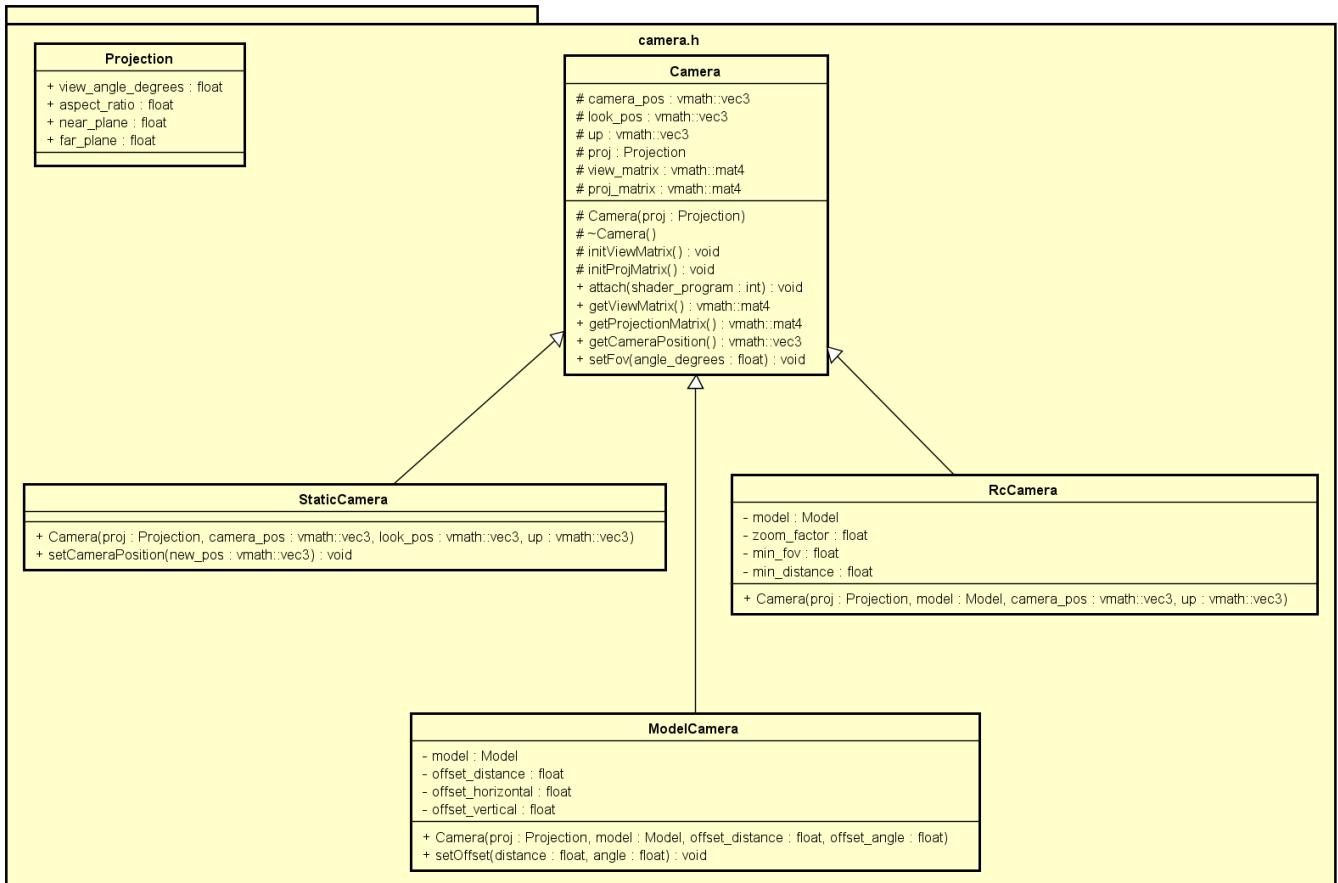
CAMERA

De Camera class beheert alle parameters van een view en projectiematrix op basis van gegeven parameters.

Om de projectie parameters gescheiden te houden van de camera parameters is er voor gekozen om deze een struct onder te brengen. Daarnaast geeft dit meer overzicht.

We kunnen 3 soorten camera's aanmaken:

- Statische camera die naar vast punt kijkt
- Third person camera die een object volgt
- Statische camera die een object volgt met dynamische zoom functie



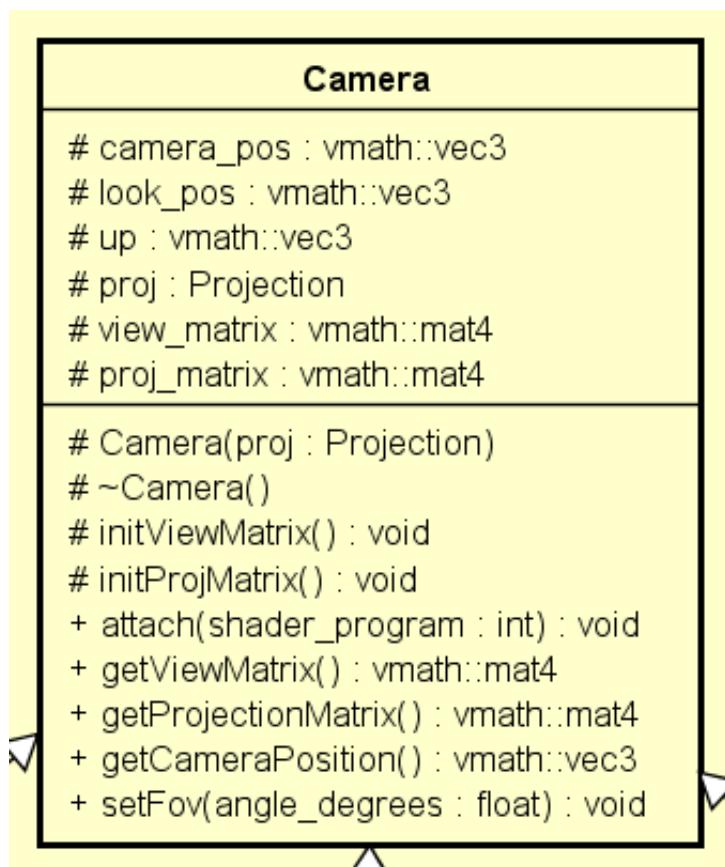
Basis klasse camera

De basis klasse van camera bevat alle gemeenschappelijke logica van een camera, ongeacht wat voor soort camera het is.

Voor elke camera geldt dat er een projectie instantie moet worden meegegeven in de constructor, de overige camera parameters zijn afhankelijk van de specifieke child class constructor.

Overzicht

Headerfile `camera.h`



Constructor

In de basis (parent) constructor wordt de projectiematrix (eenmalig) geïnitialiseert.

```
Camera::Camera(Projection projection_params)
{
    this->proj = projection_params;
    initProjMatrix();
}
```

Koppelfunctie

De attach() functie koppelt de viewmatrix en de projectiematrix als uniforms aan een gegeven shader programma.

Bij elke render iteratie dient de attach() functie te worden aangeroepen.

```
void Camera::attach(int shader_program)
{
    initViewMatrix(); // viewmatrix updaten

    glUniformMatrix4fv(glGetUniformLocation(shader_program, "view"), 1, FALSE, view_matrix);
    glUniformMatrix4fv(glGetUniformLocation(shader_program, "projection"), 1, FALSE, proj_matrix);
}
```

View- en projectiematrix initialiseren

initViewMatrix() initialiseert een viewmatrix op basis van de camerapositie, kijkpositie en omhoog-vector (zie MODEL-VIEW TRANSFORMATIE).

initProjMatrix() initialiseert een (perspectief) projectiematrix aan op basis van de projectie parameters (zie PROJECTIE).

```
// creëer een viewmatrix
void Camera::initViewMatrix()
{
    view_matrix = vmath::lookat(
        camera_pos, // camerapositie
        look_pos,   // kijkpunt
        up          // omhoog-vector
    );
}

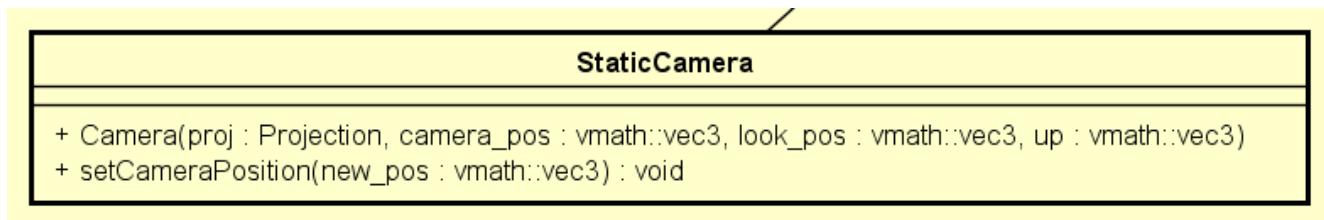
// creëer een projectiematrix
void Camera::initProjMatrix()
{
    proj_matrix = vmath::perspective(
        proj.view_angle_degrees, // frustum-hoek graden
        proj.aspect_ratio,       // beeld verhouding
        proj.near_plane,         // minimale render afstand
        proj.far_plane           // maximale render afstand
    );
}
```

Statische camera

De statische camera is de meest eenvoudige camera die je kan bedenken met een vaste positie en een vast kijkpunt.

Overzicht

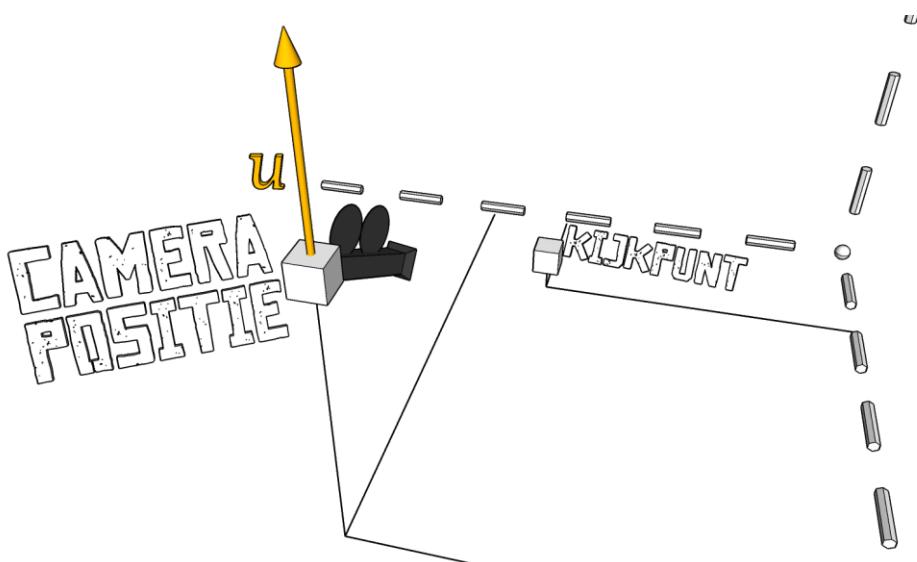
Headerfile `staticcamera.h`



Constructor

De constructor vereist een camerapositie, een kijkpositie en een omhoog-vector om een viewmatrix te kunnen initialiseren.

```
StaticCamera::StaticCamera(Projection proj, vmath::vec3 camera_pos,  
                           vmath::vec3 look_pos, vmath::vec3 up) : Camera(proj)  
{  
    this->camera_pos = camera_pos;  
    this->look_pos = look_pos;  
    this->up = up;  
  
    initViewMatrix();  
}
```

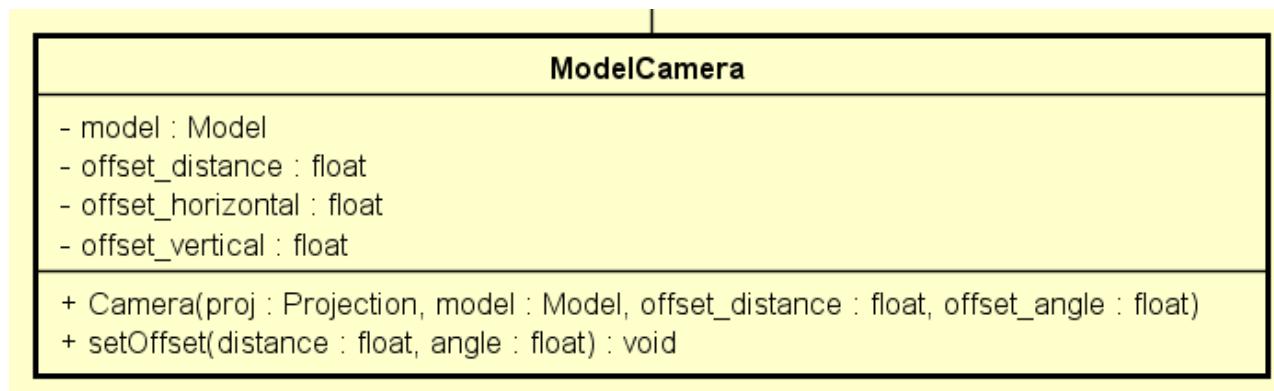


Model camera

De model camera is een third person camera die een gegeven object volgt.

Overzicht

Headerfile `modelcamera.h`



Constructor

De constructor vereist een Model, volg afstand en een hoek.

```
ModelCamera::ModelCamera(Projection proj, Model & model, float offset_distance,  
float angle_degrees) : Camera(proj), model(model)  
{  
    this->offset_distance = offset_distance;  
    offset_horizontal = -1.0f;  
    offset_vertical = tan(-angle_degrees * PI / 180) * offset_horizontal;  
  
    initViewMatrix();  
}
```

Viewmatrix parameters bepalen

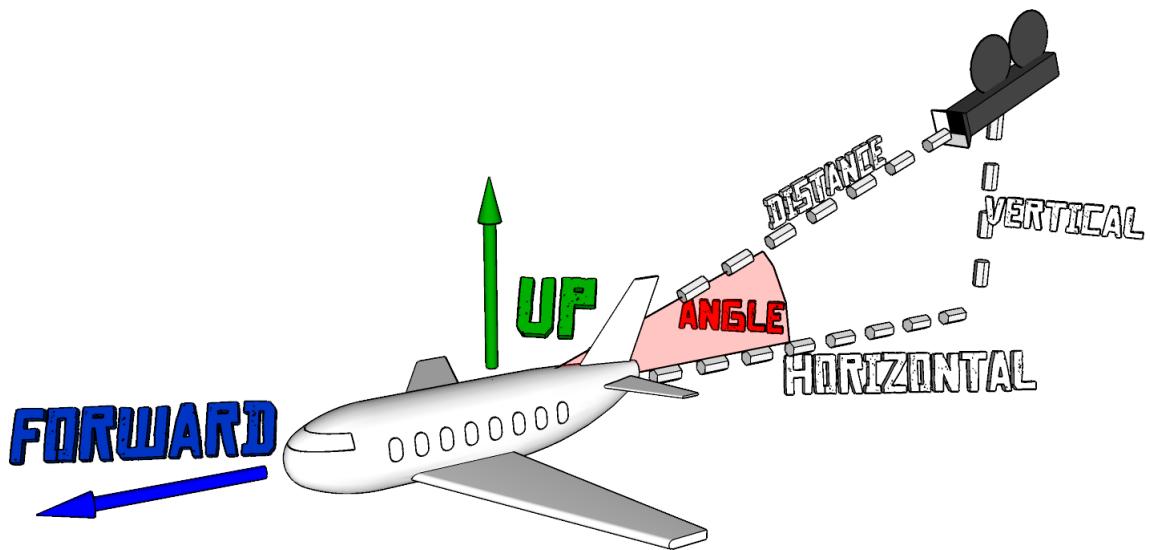
De initViewMatrix() functie van de model camera bepaalt de basis parameters (camerapositie, omhoogvector en kijk positie) van de viewmatrix. De onderstaande figuur toont een overzicht van alle variabelen.

```
void ModelCamera::initViewMatrix()
{
    vmath::vec3 offsetHorizontal = vmath::normalize(model.getForward()) * offset_horizontal;
    vmath::vec3 offsetVertical = vmath::normalize(model.getUp()) * offset_vertical;
    vmath::vec3 offset = vmath::normalize(offsetZ + offsetY);
    camera_pos = model.getPosition() + (offset * offset_distance);

    up = vmath::normalize(model->getUp());

    look_pos = model.getPosition();

    Camera::initViewMatrix();
}
```



Volgafstand en hoek

De functie setOffset() maakt het mogelijk om de volgafstand en kijkhoek te manipuleren.

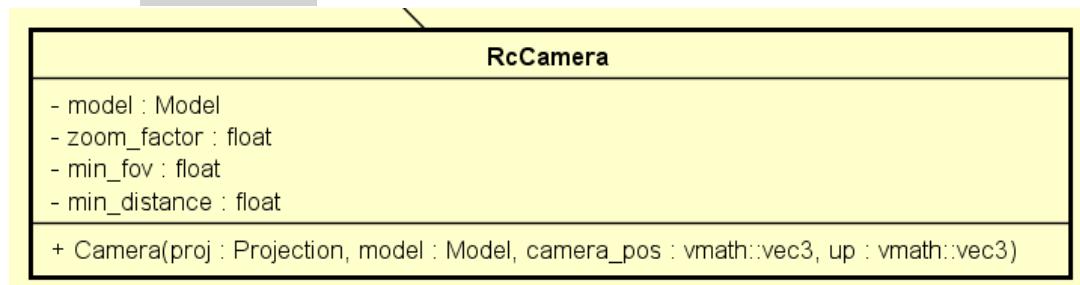
```
void ModelCamera::setOffset(float distance, float angle_degrees)
{
    offset_vertical = tan(angle_degrees * PI / 180) * offset_horizontal;
    offset_distance = distance;
    initViewMatrix();
}
```

RC camera (remote control)

De RC camera is een camera die een gegeven object volgt vanuit een statische positie.

Overzicht

Headerfile `rccamera.h`



Constructor

De constructor vereist een te volgen object, een camerapositie en een omhoog-vector.

In de constructor worden ook de minimale volg afstand en minimale frustum-hoek geïnitialiseerd.

```
RcCamera::RcCamera(Projection proj, Model & model, vmath::vec3 camera_pos,  
vmath::vec3 up) : Camera(proj), model(model)  
{  
    this->camera_pos = camera_pos;  
    this->up = up;  
    min_fov = proj.view_angle_degrees;  
    min_distance = vmath::distance(camera_pos, model.getPosition());  
  
    initViewMatrix();  
}
```

Viewmatrix parameters bepalen

initViewMatrix() bevat de zoom logica.

Er wordt een zoomfactor berekent op basis van de minimale kijkafstand en en de huidige kijkafstand (camera positie – model positie).

Op basis van de zoomfactor wordt de frustum-hoek (field of view) gemanipuleerd.

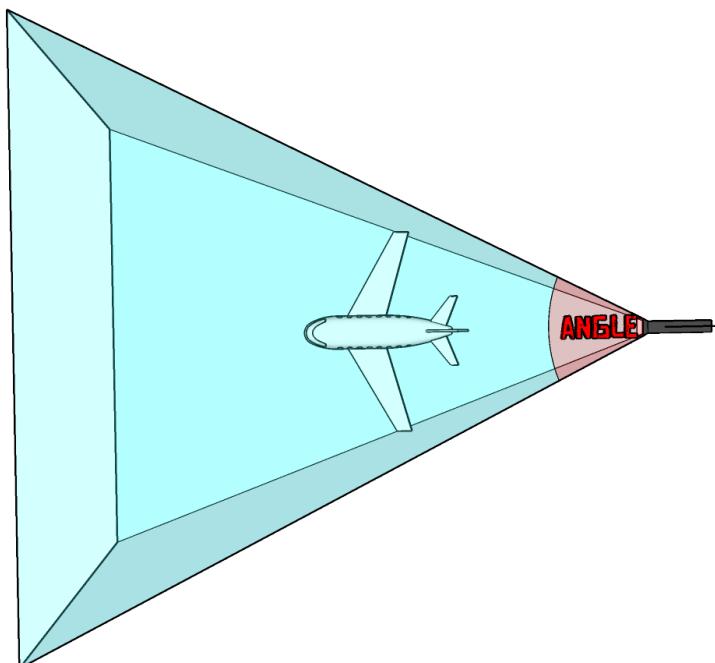
De setFov() functie manipuleert de projectiematrix op basis van een gegeven frustum-hoek.

Hierdoor krijgt men een inzoomend effect.

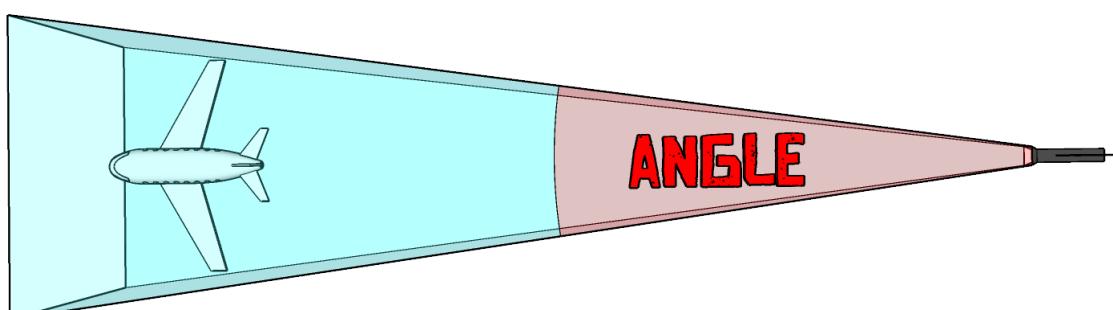
```
void RcCamera::initViewMatrix()
{
    float current_distance = vmath::distance(camera_pos, model.getPosition());
    if (current_distance < min_distance)
    {
        current_distance = min_distance;
    }
    zoomfactor = min_distance / current_distance;
    setFov(min_fov * zoomfactor);
    look_pos = model.getPosition();

    Camera::initViewMatrix();
}
```

Model is dichtbij de camera, frustum-hoek is groot.



Indien het object verder is verwijderd van de camera, wordt de frustum-hoek kleiner (inzoomen).

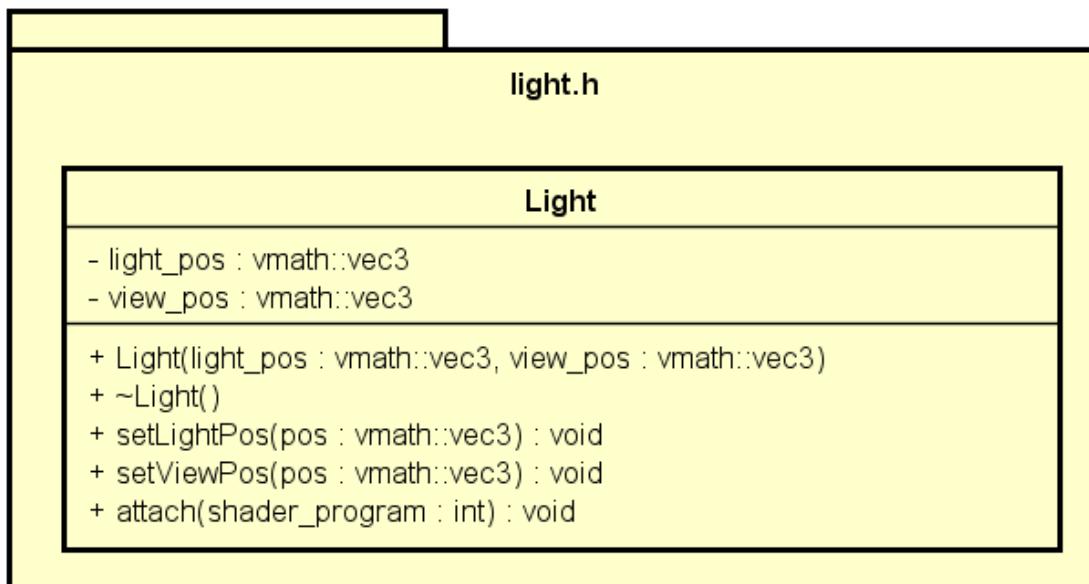


BELICHTINGSCOMPONENTEN

De Light class beheert de belichtingscomponenten (lichtbron positie en kijk positie). m.b.v. de attach of koppel functie kunnen de belichtingscomponenten gekoppeld worden aan een shader programma.

Overzicht

Headerfile `light.h`



Koppelfunctie

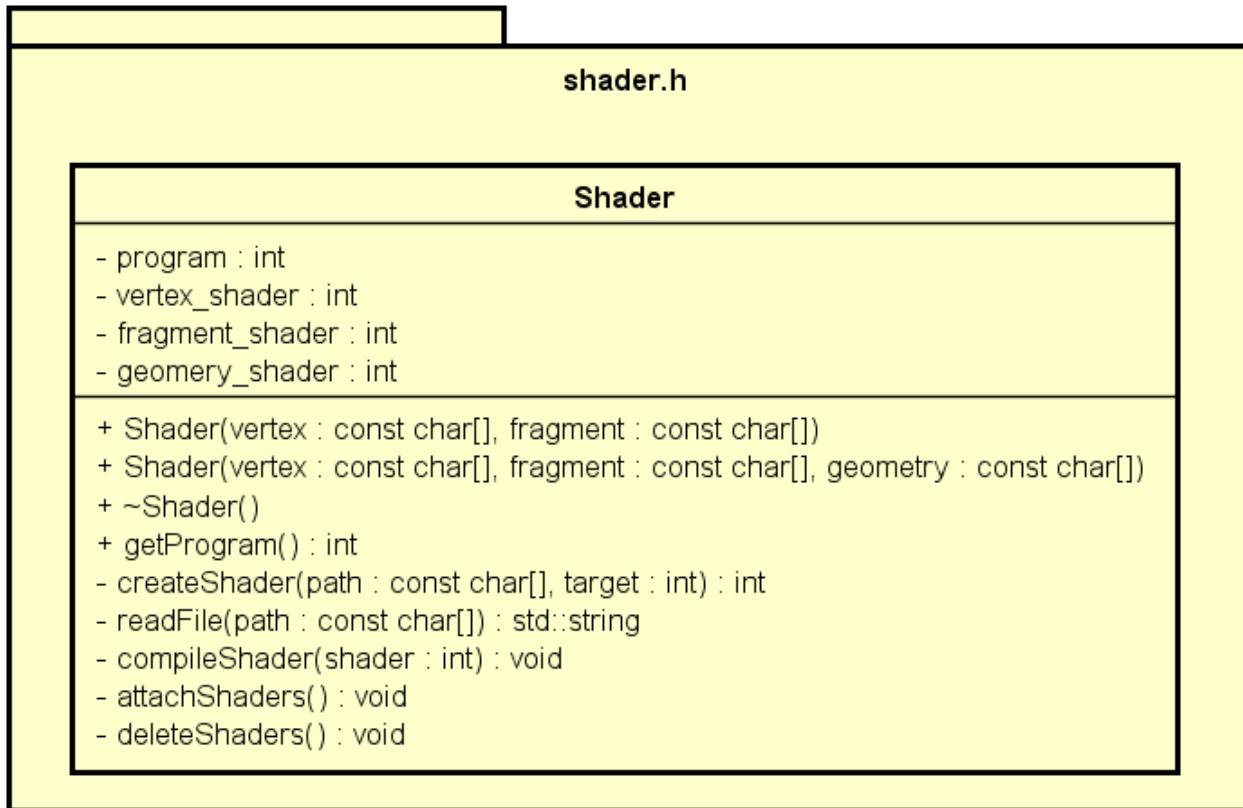
De Attach() functie koppelt de belichtingscomponenten aan een shader programma.

```
// koppel licht parameters als uniforms aan een shader programma
void Light::attach(int shader_program)
{
    glUniform3fv(glGetUniformLocation(shader_program, "lightPos"), 1, light_pos);
    glUniform3fv(glGetUniformLocation(shader_program, "viewPos"), 1, view_pos);
}
```

SHADERCOMPONENTEN

De shader class bevat alle logica voor het aanmaken van een shader programma.

Overzicht



Instantie variabelen

De belangrijkste instantie variabelen van de `Shader` class zijn:

- Programma object
- Vertex shader object
- Fragment shader object
- Geometry shader object (optioneel; alleen nodig bij flat shading)

Implementatie

Constructor

De constructor vereist minimaal een locatie naar een vertex shader en een locatie naar een fragment shader.

```
Shader::Shader(const char * vertex_path, const char * fragment_path)
{
    program = glCreateProgram();

    vertex_shader = createShader(vertex_path, GL_VERTEX_SHADER);
    fragment_shader = createShader(fragment_path, GL_FRAGMENT_SHADER);
    geometry_shader = 0;

    compileShader(vertex_shader);
    compileShader(fragment_shader);

    attachShaders();
    deleteShaders();
}
```

Als optioneel argument kan een locatie naar geometry shader worden meegegeven.

```
Shader::Shader(const char * vertex_path, const char * fragment_path, const char
* geometry_path)
{
    program = glCreateProgram();

    vertex_shader = createShader(vertex_path, GL_VERTEX_SHADER);
    fragment_shader = createShader(fragment_path, GL_FRAGMENT_SHADER);
    geometry_shader = createShader(geometry_path, GL_GEOMETRY_SHADER);

    compileShader(vertex_shader);
    compileShader(fragment_shader);
    compileShader(geometry_shader);

    attachShaders();
    deleteShaders();
}
```

Shader object initialiseren

createShader() maakt een shader object aan op basis van:

- Type shader (`target`), bijv. `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`, of `GL_GEOMETRY_SHADER`.
- Bestandslocatie, de `readFile()` functie leest een bestand uit, vervolgens wordt de resulterende string omgezet naar een char array die als bron dient voor het shader object.

```
GLuint Shader::createShader(const char * path, GLuint target)
{
    std::string source_s = readFile(path);
    const GLchar * source = source_s.c_str();
    GLuint shader = glCreateShader(target);
    glShaderSource(shader, 1, &source, NULL);
    return shader;
}
```

Shader object compileren

Nadat de shader objecten zijn geinitialiseerd kunnen ze gecompileerd worden.

Als het compileren is mislukt wordt er een error output gegenereerd.

Het is niet mogelijk om shaders te debuggen omdat ze uitgevoerd worden op de GPU.

```
void Shader::compileShader(GLuint shader)
{
    glCompileShader(shader);
    int success;
    char infoLog[512];
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(shader, 512, NULL, infoLog);
        OutputDebugString(infoLog);
    }
}
```

Shader objecten koppelen aan programma object

Als de shaders succesvol gecompileerd zijn kunnen ze worden gekoppeld aan het programma object.

Als het koppelen is mislukt wordt ook hier een error output gegenereerd.

Er moet minimaal een vertex shader en fragment shader gekoppeld worden.

```
void Shader::attachShaders()
{
    glAttachShader(program, vertex_shader);
    glAttachShader(program, fragment_shader);
    if (geometry_shader != 0)
        glAttachShader(program, geometry_shader);

    glLinkProgram(program);

    int success_link;
    char infoLog_link[512];
    glGetProgramiv(program, GL_LINK_STATUS, &success_link);
    if (!success_link) {
        glGetProgramInfoLog(program, 512, NULL, infoLog_link);
        OutputDebugString(infoLog_link);
    }
}
```

Shader objecten verwijderen

Nadat de shader objecten gekoppeld zijn aan het shader programma kunnen de shader objecten opgeruimd worden met deleteShaders().

```
void Shader::deleteShaders()
{
    glDeleteShader(vertex_shader);
    glDeleteShader(fragment_shader);
    if (geometry_shader != 0)
        glDeleteShader(geometry_shader);
}
```

Programma object ophalen

De getProgram() functie retourneert een programma object variabele van een Shader instantie.

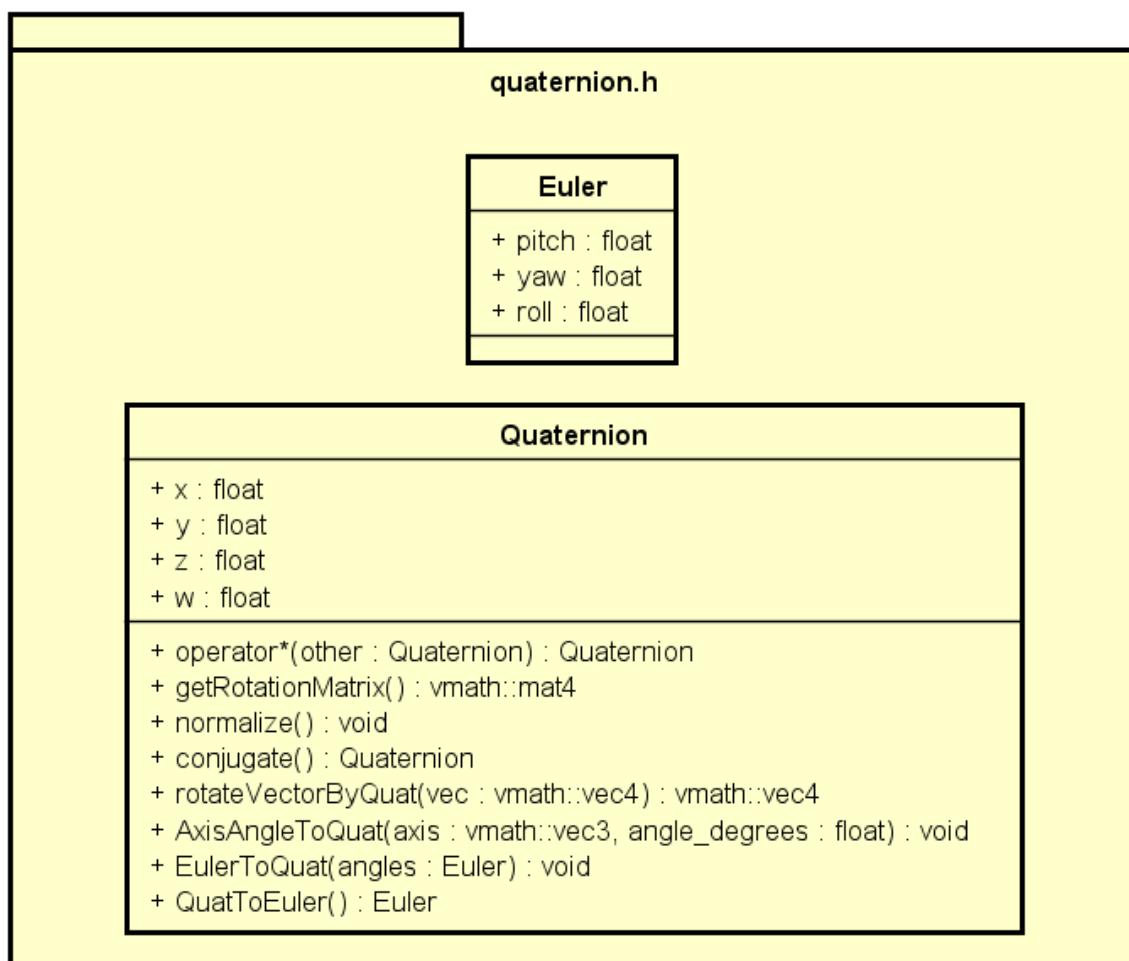
QUATERNION

De Quaternion struct bevat de nodige (rotatie) logica voor het rekenen met quaternions.

Er is ook gekozen om Euler hoeken te representeren in een struct, dit geeft meer overzicht bij berekeningen.

Alle functies van de Quaternion struct zijn gebaseerd op de berekeningen uit hoofdstuk 4.

Headerfile `quaternion.h`



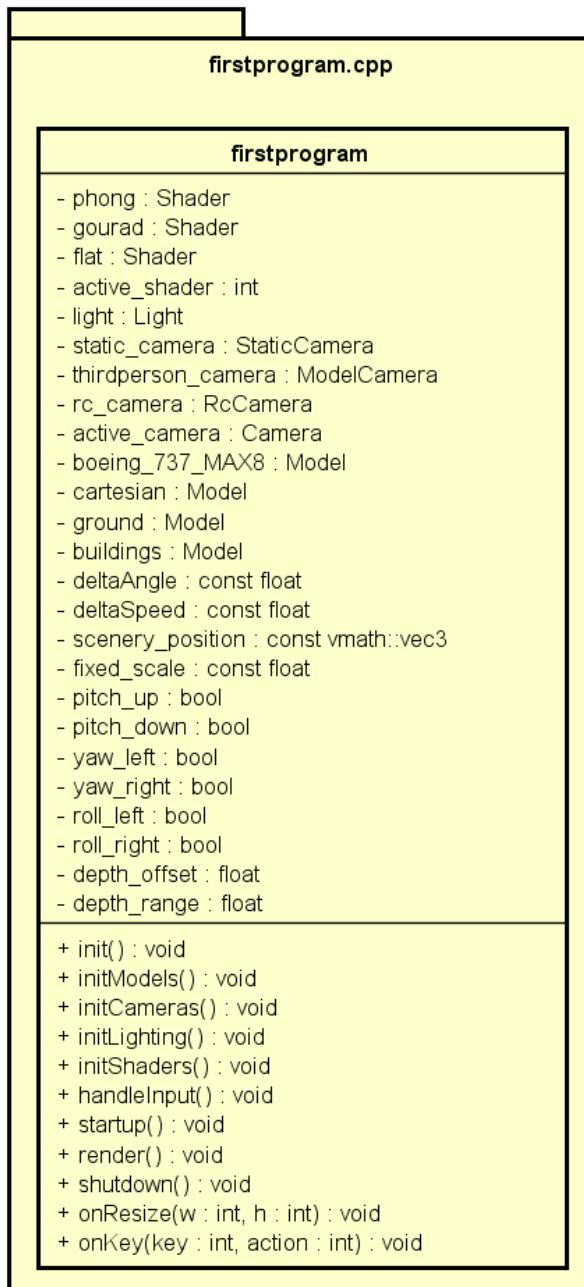
MAIN CLASS

We hebben de nodige componenten behandeld om een volledige 3D renderer te bouwen van scratch.

- We kunnen 3D bestanden inlezen door de `objreader.h` headerfile te includen.
- We kunnen shader objecten aanmaken met behulp van de Shader class.
- We kunnen meerdere objecten tegelijk renderen / beheren met de Model class
- We kunnen verschillende camera's aanmaken: RcCamera, ModelCamera en StaticCamera class
- We kunnen een lichtbron aanmaken met de Light class
- We kunnen berekeningen doen met de Quaterion class

`firstprogram.cpp` is de main class van de applicatie, hier worden alle componenten aan elkaar 'geknoopt'.

Overzicht



Instantie variabelen

Hier volgt de uitleg van de instantie variabelen.

- 4 verschillende typen shaders, tijdens run-time kan er gewisseld worden tussen de verschillende shaders
 - Phong shader
 - Gourad shader
 - Flat shader
 - Depth shader (om de depth- of Z-buffer te visualiseren)
- Light instantie voor de belichtingscomponenten
- 3 verschillende camera's, ook hier geldt dat er tijdens run-time gewisseld kan worden tussen de verschillende camera's
 - Statische camera
 - RC camera
 - Model camera
- Objecten / modellen
 - Vliegtuig
 - Assenstelsel (handig voor calibratie)
 - Heuvellandschap
 - Gebouwen
- Rotatie en acceleratie interval
- Positie van de omgeving (heuvels en gebouwen)
- Standaard schaal die van toepassing is op alle objecten (1 op 1000)
- Booleans om bij te houden welke rotaties van toepassing zijn op het vliegtuig (afhankelijk van user input)
- Depth buffer visualisatie parameters
 - Minimale afstand van de camera
 - Bereik depth buffer spectrum (uiterst zwart tot uiterst wit)

Belangrijkste implementaties

Initialisatie logica

Om alle logica van elkaar te scheiden zijn er aparte initialisatiefuncties

- initModels() initialiseert alle objecten
- initCameras() initialiseert alle camera's
- initLighting() initialiseert de belichtingsobjecten
- initShaders() initialiseert alle shaders

In de StartUp() functie wordt alle initialisatie logica, dus ook de bovenstaande functies, uitgevoerd.

Ook kunnen we hier de instelling inschakelen voor de depth test (op basis van depth buffer overlappende fragmenten negeren) en het niet renderen van backfaces (achterkant polygonen negeren).

```
// eenmalig uitvoeren bij opstarten
virtual void startup()
{
    glEnable(GL_DEPTH_TEST); // depth test inschakelen, overlapping detecteren
    glEnable(GL_CULL_FACE); // backfaces negeren

    initShaders();
    initModels();
    initCameras();
    initLighting();

    depth_range = 800.0f; // bereik; depth buffer visualisatie shader
    depth_offset = 0.2f; // minimale afstand; depth buffer visualisatie shader
    depth_near = 0.2f;
}
```

Render logica

De render() functie handelt alle logica af per render iteratie

```
// logica per render iteratie
virtual void render(double currentTime)
{
    glClearColor(0.6f, 0.9f, 1.0f, 1.0f); // achtergrond kleur

    // framebuffer opschenen (depth buffer en color buffer)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUseProgram(active_shader); // shader programma toewijzen

    // maak de depth visualisatie parameters beschikbaar in de shader
    glUniform1f(glGetUniformLocation(active_shader, "depthRange"), depth_range);
    glUniform1f(glGetUniformLocation(active_shader, "depthOffset"), depth_offset);

    handleInput(); // rotaties afhandelen op basis van actieve gebruikersinput

    light_source->attach(active_shader); // licht koppelen aan het shader programma

    active_camera->attach(active_shader); // camera koppelen aan het shader programma

    // render logica uitvoeren voor model instanties
    boeing_737_MAX8->render(active_shader);
    cartesian_axis->render(active_shader);
    hills->render(active_shader);
    buildings->render(active_shader);
}
```

Gebruikersinput

De onKey() functie captured gebruikersinput, hieronder volgt een samenvatting van alle key mappings:

- pijl toetsen en bracket [] toetsen voor roteren van het vliegtuig (roll, pitch en yaw).
- W en S toetsen voor vertragen / versnellen
- C om de vliegtuig positie te printen
- 1, 2 en 3 toetsen om te wisselen tussen de camera's (respectiefelijk statisch, third person en remote control)
- P toets om over te schakelen naar Phong shading
- G toets om over te schakelen naar Gourad shading
- F toets om over te schakelen naar Flat shading
- D toets om over te schakelen naar depth buffer visualisatie shader
- - en = toetsen om het depth buffer spectrum (afstand tussen zwart en wit) te bepalen
- PAGE DOWN en PAGE UP toetsen om het depth buffer afstand tot de camera te veranderen

De handleInput() functie maakt het mogelijk om op basis van booleans meerdere rotaties tegelijkertijd mogelijk te maken voor het vliegtuig.

Alle code en 3D modellen van de sandbox game kan men vinden in [6sandbox](#)

6. EXTERNE BRONNEN
