Schritt 6. Kamera Matrizen mit GLM

Die Geometrie war am Schritt 4 bereits sichtbar, aber leider nicht vollständig im Fenster zu sehen. Dies liegt daran, dass wir alle relevanten Transformationsmatrizen auf die Einheitsmatrix gesetzt haben: Jeder Vertex kam genauso wie er definiert worden war im *Normalized Device Space* - also dem Space, indem *geclipped* wird - an. Um die Geometrie vollständig im Fenster sehen zu können, brauchen wir im Grunde eine vernünftige Kamera.

Eine Kamera im Sinne der CG wird wie folgt definiert:

- Sie hat eine Position im Raum und eine Orientierung. Beides bezieht sich natürlich aufs Referenzkoordinatensystem, also den *World Space*. Position und Orientierung werden zur View-Matrix zusammengefasst.
- Sie hat ein Frustum, welches ein Quader oder ein Pyramidenstumpf sein kann. (→ orthografisch vs. perspektivisch). Das perspektivische Frustum definiert sich über die beiden Fields-of-View (horizontal & vertikal), sowie die Near & Far Clipping Planes. Aber diese Werte beziehen sich nicht mehr aufs Welkoordinatensystem, sondern aufs lokale Kamerakoordinatensystem (!): eine Near Clipping Plane von 0.1 ist zum Beispiel 0.1 OpenGL Einheiten von der Kamera entfernt, und zwar in Richtung der View Direction.

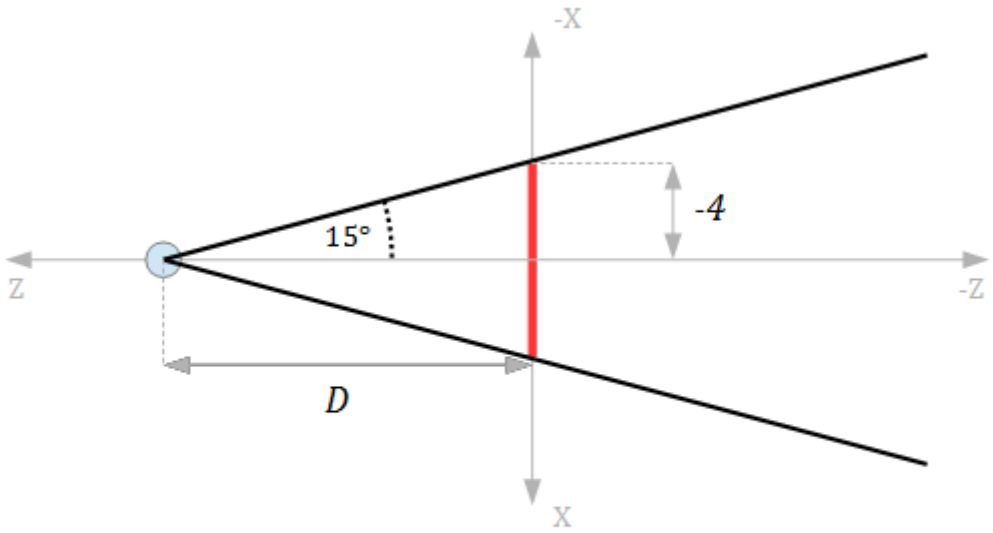
GLM bietet einige intuitiv verwendbare Grundfunktionen, um die Kameramatrizen berechnen zu können: `glm::lookAt` und `glm::perspective`. (Und für den Fall dass wir eine orthografische Kamera wollen, `glm::ortho`)

```
glm::mat4 viewTf = glm::lookAt< float >( glm::vec3( 0.f, 0.f, 15.f ), glm::vec3( 0.f,
0.f, 0.f ), glm::vec3( 0.f, 1.f, 0.f ) );
glm::mat4 projectionTf = glm::perspective< float >( glm::radians( 30.f ),
windowAspectRatio, 1.f, 30.f );
```

Wie setzen unsere Kamera mit einem horizontalen FoV von 30 Grad an. Um Verzerrungen zu vermeiden, muss der vertikale FoV dann entsprechend ans Seitenverhältnis des Viewports angepasst werden. GLM verlangt als Argumente für `glm::perspective` den hFov und das Seitenverhältnis, also die Breite des Viewports dividiert durch die Höhe. Die Werte für zNear und zFar sind relativ beliebig: zFar sollte irgendwo zwischen 0.1f und 1.f liegen, zFar sollte so gewählt werden, dass die gesamte Geometrie im Frustum liegt. Weiters ist zu beachten, dass alle Winkel in GLM in Radian angegeben werden sollen; es gibt die Hilfsfunktion `glm::radians`, um Angaben in Grad zu konvertieren.

Eine gute Ausgangslage für die Kamer ist immer, sie entlang der negativen Z-Achse blicken zu lassen. Wir erreichen dies, indem wir sie auf (0, 0, *D*) setzen - wobei *D* ein positiver Wert sein sollte – und nach (0, 0, 0) blicken lassen. *D* wurde im Sample so gewählt, dass unsere Beispielgeometrie gerade noch auf den Viewport passt. (Dies lässt sich über den Tangenssatz recht leicht berechnen, D=4/tan(15°), siehe Abbildung.) Die

Funktion `glm::lookAt` benötigt dann noch einen dritten Input, die *Up-Direction*. Wir möchten, dass diese Richtung der positiven y-Achse entspricht.



Unsere Beispielgeometrie liegt auf Z=0 und hat eine max. Ausdehnung von 4 OpenGL Einheiten in +X und -X. In Y wären es weniger (3), aber wir wollen die ganze Geometrie in der Viewport bringen. Da der Viewport quadratisch ist, ist das Field-of-View in X und Y gleich: 30°.

Als Letztes wollen wir erreichen, dass die Kamera um die Y-Achse rotiert. Wir definieren uns dafür eine `sf::Clock` – einen Timer also.

```
glm::mat3 rotation = glm::mat3( glm::rotate( glm::mat4(1.f),
glm::radians(timer.getElapsedTime().asSeconds()), glm::vec3(0.f, 1.f, 0.f) ) );
glm::vec3 camera = rotation * glm::vec3(0.f, 0.f, 15.f);
viewTf = glm::lookAt( camera, glm::vec3(0.f, 0.f, 0.f), glm::vec3(0.f, 1.f, 0.f) );
```

Sie sollen hier hauptsächlich mitnehmen, dass:

- Man in GLM `mat4` ganz einfach in `mat3` umwandeln kann - was in diesem Fall sogar bei Multiplikation mit einer Position ok ist, weil die Matrix ja nur eine Rotation ausdrückt und keine Translation – genau wie in GLSL.
- Man in GLM einen `vec3` oder auch `vec4` ganz einfach mit einer Matrix multiplizieren kann, genau wie in GLSL.
- Die Reihenfolge auch gleich ist wie in GLSL: erst Matrix, dann Vektor.
- Man diese per-Frame Berechnung in das UPDATE der Anwendung packen würde.