Übungsblatt 7

Willkommen zur siebten Übung der Veranstaltung *Generative Computergrafik*. Ziel dieses Übungsblatts ist, dass Sie sich etwas intensiver mit OpenGL vertraut machen. Die nachfolgende Aufgabe ist bis zum 19. Juni 2019 um 10:00 Uhr über `https://read.mi.hs-rm.de` **abzugeben und anschließend die im Praktikum vorzuführen**.

Aufgabe 1. Schreiben Sie ein OpenGL-Programm, das ein Dreiecks-Netz, welches im Objekt-Fileformat **obj** als Indexed Faceset gespeichert ist, einlesen und darstellen kann. Ihr Programm soll glfw (siehe `https://www.glfw.org`) als GUI Toolkit verwenden. Starten Sie dazu mit dem Skript `RenderWindow.py`, welches Sie auf der Veranstaltungswebseite finden.

Das Objekt-Fileformat **obj** ist ein einfaches ASCII Fileformat, um die Geometrie und andere Eigenschaften (Farben, Textur, ...) von Objekten zu beschreiben. Das **obj**-Fileformat unterstützt sowohl polygonale Objekte (definiert durch Punkte, Linien und Polygone) als auch Freiform-Objekte (definiert durch Kurven und Flächen). Unter `http://paulbourke.net/dataformats/` finden Sie eine komplette Beschreibung des Formats. In einem **obj**-File werden

- *Objekt-Punkte (vertices)* durch ein vorangestelltes `v`
- *Texturkoordinaten* durch ein vorangestelltes `vt`
- *Vertex-Normalen* durch ein vorangestelltes `vn`

gekennzeichnet.

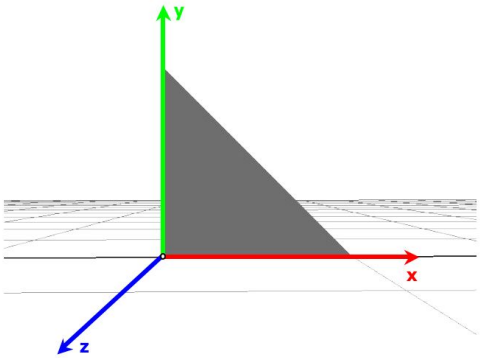
Die einzelnen Polygone (*faces*) werden durch ein vorangestelltes `f`, gefolgt von den Punktindizes (für `v`, `vt` und `vn` jeweils durch / getrennt) gekennzeichnet. Ein Dreieck wird also in der Form

```
f v/vt/vn v/vt/vn v/vt/vn
```

dargestellt. Sind keine Texturkoordinaten vorhanden, werden zwei aufeinanderfolgende Slashes geschrieben (//). Sind weder Texturkoordinaten noch Normalenvektoren vorhanden, werden die Punktindizes alleine, ganz ohne Slashes geschrieben.

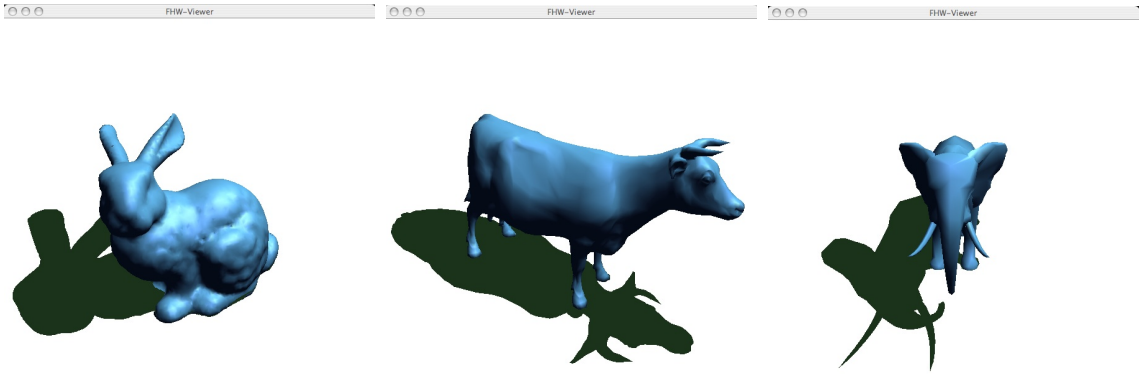
Die folgende *Beispiel-Datei* definiert ein Dreieck. Hierzu werden zunächst die drei Eckpunkte $(0.0, 0.0, 0.0)$, $(1.0, 0.0, 0.0)$, $(0.0, 1.0, 0.0)$ definiert. Anschließend folgt die Definition der Normalenvektoren $(0.0, 0.0, 1.0)$, $(0.0, 0.0, 1.0)$, $(0.0, 0.0, 1.0)$. Die letzte mit `f` beginnende Zeile definiert schließlich ein Dreieck mit den Eckpunkten $(1, 2, 3)$, wobei Punkt 1 die Normale 1, Punkt 2 die Normale 2 und Punkt 3 die Normale 3 zugewiesen bekommt. Texturkoordinaten sind keine vorhanden.

```
v  0.0 0.0 0.0
v  1.0 0.0 0.0
v  0.0 1.0 0.0

vn 0.0 0.0 1.0
vn 0.0 0.0 1.0
vn 0.0 0.0 1.0

f 1//1 2//2 3//3
```



Eine **obj** Datei kann mehrere Vertex-, Vertexnormalen- und Face-Blöcke enthalten. Die **Nummerierung der Vertices und Vertexnormalen ist fortlaufend** und beginnt nicht etwa für jeden Block wieder bei 1.

Unter `cvmr.info/teaching/computergrafik/` finden Sie fünf unterschiedliche Modelle (Stanford-Bunny, Cow, Elephant, Squirrel, Squirrel_ar) im **obj**-Fileformat. Ihr Programm soll beim Aufruf von `python oglViewer.py object.obj` die Daten aus der Datei `object.obj` einlesen und das zugehörige Modell in einem GLUT-Fenster darstellen.



Ihr ***obj**-Viewer* soll dabei (mindestens) die folgenden Features besitzen:

- *Einlesen eines obj-Files und Darstellung dessen Inhalts als Polygonnetz mit Hilfe von VBOs.*
- *Schattierte Darstellung* eines *beleuchteten Modells* (siehe obige Abbildungen). Die Eigenschaften von Lichtquelle(n) und Objektmaterial(ein) können Sie dabei beliebig wählen.
- *Navigation* mit Hilfe der *Maus*
  - Bei gedrückter *linker Maustaste* soll das Modell mittels Arcball-Metapher (sieh auch Vorlesungsfolie 194) rotiert werden können.

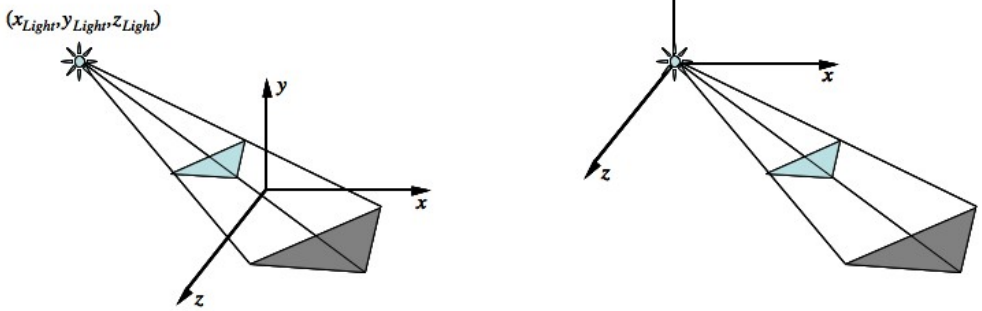
  - Bei gedrückter *mittlerer Maustaste* soll an das Modell heran- bzw. weggezoomt werden können.
  - Bei gedrückter *rechter Maustaste* soll das Modell verschoben werden können.

- *Umschalten* zwischen Orthogonal- und Zentral- *Projektion* über die Tasten [O] und [P]. In beiden Fällen soll die Kamera geeignet angepasst werden, wenn die Fenstergröße geändert wird, sodass keine Verzerrungen entstehen.

- *Wechseln* der Hintergrund- und Objekt-Farbe. Als Hintergrund- und Objekt-Farbe sollen, unabhängig voneinander über die Tasten [S], [W], [R], [B] und [G], mindestens die Farben *Schwarz*, *Weiß*, *Rot*, *Blau* und *Gelb* eingestellt werden können.

- *Darstellung eines einfachen Schattens* des eingelesenen Modells. Das eingelesene Modell soll dabei mit seiner Basis auf der $y = 0$ Ebene stehen. Der Schatten sollte sich dabei über die Taste [H] an- und ausschalten lassen. Bei einem lokalen Beleuchtungsmodell, wie OpenGL es realisiert, kann ein einfacher Schatten (vgl. obige Abb.) folgendendermaßen durch eine zusätzliche projektive Abbildung des darzustellenden Objekts erzeugt werden:

  Wir betrachten einen Schatten, der durch *eine einzige* Punktlichtquelle erzeugt wird und nehmen der Einfachheit halber also an, dass er auf die Ebene $y = 0$ fällt. Dieser Schatten entsteht durch (Zentral-)Projektion des Objekts auf die Ebene $y = 0$. Das Projektionszentrum ist die Position der Lichtquelle. Projiziert man nun die Polygone des Objekts in einem Koordinatensystem, in dem die Lichtquelle der Ursprung ist, erhält man die Eckpunkte der Polygone des Schattens in diesem Licht-Koordinatensytem (siehe Abbildung). Diese Eckpunkte müssen anschließend in das Welt-Koordinatensystem zurücktransformiert werden.



Als konkretes Beispiel sei die *Position der Lichtquelle* $(x_{Light}, y_{Light}, z_{Light})$. Man verschiebt diese nun mittels $\mathbf{t} = (-x_{Light}, -y_{Light}, -z_{Light})$ in den Ursprung und führt anschließend die Projektion $\mathbf{P}$ mit dem Ursprung als Projektionszentrum durch. Nach der Rücktranslation mittels $\mathbf{t} = (x_{Light}, y_{Light}, z_{Light})$ erhält man die Eckpunkte der Polygone des Schattens im Weltkoordinatensystem. Die Transformationen können zum Beispiel wie folgt mit Hilfe der Fixed-Function-Pipeline von OpenGL durchgeführt werden:

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_{Light}} & 0 & 0 \end{pmatrix}$$

```
# projection matrix
p = [1.0, 0, 0, 0, 0, 1.0, 0, -1.0/yLight, 0, 0, 1.0, 0, 0, 0, 0, 0]
glColor3fv(modelColor)
glCallList(modelList)                    # render object normally
glMatrixMode(GL_MODELVIEW)               # use modelview matrix
glPushMatrix()                           # save state
glTranslatef(xLight, yLight, zLight)     # translate back
glMultMatrixf(p)                         # project object
glTranslatef(-xLight, -yLight, -zLight) # move light to origin
glColor3fv(shadowColor)
glCallList(modelList)                    # render object again
glPopMatrix()                            # restore state
```

Beachten Sie dabei, dass die Basis des Modells dabei auf der $y = 0$ Ebene steht, d.h. dass $y_{min} = 0$ für die Boundingbox des Modells gilt. Achten Sie weiterhin darauf, dass ein Schatten in der Regel nicht beleuchtet ist. Bei der beschriebenen Vorgehensweise ist weiterhin zu beachten, dass alle Polygone des Modells für den Schatten übereinander in die $y = 0$ Ebene projiziert werden, was bei der Tiefenberechnung zu Renderingartefakten aufgrund numerischer Ungenauigkeiten führen kann. Die Verwendung des Tiefenpuffers sollte für das Rendering des Schattens daher ausgeschaltet werden.

Der hier vorgestellte „Trick“ zur Erzeugung eines Schattens ist sehr einfach und eigneten sich nicht für die Darstellung von Schatten auf Objekten, die nicht vollständig eben sind. Für weitere Informationen zum Thema *Schattenerzeugung* mit OpenGL lohnt sich ein Blick in

- W. T. Reeves, D. H. Salesin, and R. L. Cook. *Rendering antialiased shadows with depth maps.* SIG. Comp. Graph., 21(4):283–291, Aug. 1987.
- C. Everitt, A. Rege, and C. Cebenoyan. *Hardware Shadow Mapping.* White paper, NVIDIA Corporation, 2001.
- `www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/`