Projektion

Kurz & Knapp

- Was ist der Untschied zwischen einer orthographischen und einer perspektivischen Projektion?
- Woran kann man das in einer gegebenen Projektionsmatrix erkennen?
- Sind near und far positiv oder negativ angegeben?
- Wo schneidet die z-Kurve nach der Projektion die Near- und die Farplane?
- Wie verhält sich die Kurve darüber hinaus, wohin konvergiert sie fur $z \to \infty$?
- Was bedeuten in der Projektionsmatrix t, b und l, r, wie steht das im Zusammenhang zu w, h, a und dem Öffnungswinkel der Kamera?

Aufgabe 1

Gegeben eine Transformation mit einer Projektionsmatrix

$$\mathbf{P} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, \quad \mathbf{v}' = \mathbf{P}\mathbf{v},$$

bestimmen Sie den maximalen Wert von x der in NDC auf 1 abgebildet wird, abhängig von z, also die Funktion $x_{\max}(z)$. Berechnen Sie dann diese Funktion für $z = -n$ und $z = -f$.

Hinweis: Die Parameter n und f werden als Absolutbetrag angegeben und nur für reguläre z-Werte negiert verwendet.

Lösung

Betrachtung von $(H(\mathbf{P}\mathbf{v}))_x$, umstellen nach x.

$-\frac{2nx}{(r-l)z} - \frac{r+l}{r-l} = 1$
$\leftrightarrow \quad -\frac{2nx}{r-l} - \frac{z(r+l)}{r-l} = z$
$\leftrightarrow \quad -\frac{2nx}{r-l} = z + \frac{z(r+l)}{r-l} = \frac{z(r-l)+z(r+l)}{r-l}$
$\leftrightarrow \quad -2nx = z((r-l)+(r+l)) = z2r$
$\leftrightarrow \quad x = -\frac{zr}{n}$

Für $z = -n \to x = -\frac{-nr}{n} = r$.

Für $z = -f \to x = -\frac{-fr}{n} = \frac{fr}{n}$.

Test für $z = -f$: $-\frac{2nfr}{-nf(r-l)} - \frac{r+l}{r-l} = \frac{2r-(r+l)}{r-l} = \frac{r-l}{r-l} = 1$.

Aufgabe 2

Im Tarball `renderpipeline.tar.gz` finden Sie eine kleine C++ Implementierung einer Renderpipeline mit Rasterisierung wie bisher in der Vorlesung besprochen. Zum Übersetzen ist es nötig `libpng++` und `glm` zu installieren, auf Debian-Derivaten sind die Pakete einfach verfügbar:

```
apt install libpng++ libglm-dev
```

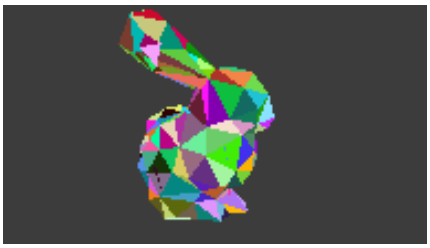
Das Programm verarbeitet Eingaben auf der Kommandozeile, erzeugt ein Dreiecksnetz, wendet Transformationen auf die Dreiecke des Netzes an, rasterisiert die so transformierten Dreiecke und schreibt am Ende ein Bild ins Dateisystem. Zur besseren Darstellung wird jedes Dreieck mit einer zufälligen Farbe gezeichnet. Diesen Ablauf können Sie in der Datei `main.cpp` ablesen.

Die Kommandozeilenschnittstelle ist in `cmdline.cpp` definiert, die entsprechenden Variablen und Initialwerte finden Sie in `cmdline.h`. Diese Dateien müssen nicht verändert werden.

In `raster.cpp` finden Sie eine Implementierung der SD-Rasterisierung wie sie im ersten Blatt mit Asymptote als Aufgabe gestellt war. Lesen Sie den Code durch und prüfen Sie, ob Sie alles nachvollziehen können. Der einzig neue Teil sollte die Verwendung eines sog. *Framebuffers* sein, der als Ausgabe des Renderns verwendet wird. Dabei handelt es sich in unserem Fall nur um ein im Speicher vorgehaltenes PNG Bild, das später gespeichert wird.

Informationen zum Aufruf des Programms können Sie sich mittels `./renderpipeline --help` schön aufbereitet ansehen.





Aufgabe 2.1

Beginnen Sie damit, ein Dreieck in NDC zu rendern. Die Standardwerte der Kommandozeilenparameter sind für diesen Fall eingestellt, Sie können diesen Modus explizit mittels `-s ndc-tri` (bzw. `--scene ndc-tri`) angeben. In der Szenenauswahl wird dann der erste Zweig ausgeführt (`scene.cpp:9`), in dem bisher ein Platzhalter-Dreieck erzeugt wird. Überlegen Sie hier was gültige NDC-Koordinaten für ein Dreieck sein könnten, wie es aussehen sollte und tragen Sie die Koordinaten ein. Wird auf der Kommandozeile explizit ein z-Wert angegeben (z.B. `-Z 0.5`), so soll dieser verwendet werden.

Wenn Sie das Programm nun ausführen werden Sie wahrscheinlich kein Ergebnis sehen. Was noch fehlt ist die Transformation des NDC-Dreiecks in Window-Koordinaten, wie es der Rasterisierer erwartet. Erweitern Sie dazu die Funktion `window_transform` in `main.cpp` und füllen Sie den Inhalt von `W_transform`. In der Abbildung oben links sehen Sie ein *beispielhaftes* Ergebnis mit `-w 192 -h 108` (das natürlich von Ihrem konkret gewählten Dreieck abweichen kann).

Hinweise.

- GLM Matrizen werden in Spalten-Ordnung geschrieben, d.h. insbesondere beim Erzeugen einer neuen Matrix werden Vektoren angegeben, die als separate Spalten der Matrix zu verstehen sind. Der Hinweis steht hier prominent, da die Matrizen im Code deshalb transponiert stehen.
- Sie können Vektoren und Matrizen aus GLM mittels `cout << to_string(...) << endl;` ausgeben.
- Zum Debuggen können Sie im Rasterisierer `overlaps_pixel` anpassen, z.B. um nur die Bounding Box zu prüfen.
- Beachten Sie, dass in der Rasterisierung erwartet wird, dass die Dreickeskanten gegen den Uhrzeigersinn verlaufen.
- Aus einem `vec3 v` können Sie via `vec4(v, 1.0f)` einen homogenen Vektor erzeugen, aus einem `vec4 w` via `v = w` einen kartesischen. Aber: Hier wird nicht implizit dehomogenisiert, es wird einfach die letzte Komponente verworfen!

Als Ausgabe wird standardmäßig die Datei `out.png` angelegt, der Hintergrund ist dunkelgrau, das Dreieck in einer zufälligen Farbe. In der Abbildung oben sehen Sie in der Mitte ein beispielhaftes Ergebnis. Prüfen Sie was passiert, wenn Sie z im gültigen Bereich verschieben. Was passiert und warum?

Aufgabe 2.2

Verwenden Sie den Kommandozeilenschalter `-s es-tri`. Nun soll als 'Szene' ein Dreieck im Eye Space erzeugt werden. Gehen Sie wie in der vorigen Aufgabe beschrieben vor.

Passen Sie weiterhin die Funktion `perspective_projection_transform` an und verwenden Sie diese korrekt in `PW_transform`. Achten Sie darauf, dass der Rasterisierer nach wie vor Eingangsdaten in Window-Koordinaten erwartet.

Prüfen Sie auch hier, was passiert wenn Sie z im gültigen Bereich verschieben. Was passiert und warum?

Aufgabe 2.3

Verwendne Sie nun den Kommandozeilenschalter `-s bunny`. Damit wird ein größeres Mesh das in Weltkoordinaten angegeben ist geladen. An `scene.cpp` ist in diesem Fall keine Änderung nötig.

Die Eye Space Paramter Kameraposition (`-p`), Blick-Richtung (`-d`) und Up-Vektor (`-u`) können alle auf der Kommandozeile angegeben werden und haben als Standardeinstellung den $(0,0,0)^\mathsf{T}$, $(0,0,-1)^\mathsf{T}$ und $(0,1,0)^\mathsf{T}$.

Implementieren Sie `viewing_transform` und `VPW_transform` in `main.cpp` und prüfen Sie, ob Sie an Position $(-75,75,60)^\mathsf{T}$ mit Blickrichtung $(1,-1,0)^\mathsf{T}$ und Up-Vektor $(0,0,1)^\mathsf{T}$ das selbe Ergebnis erhalten, wie in der Abbildung oben rechts.

Mit dem beiliegenden Pyhton-Skript können Sie (sofern Sie `ffmpeg` installiert haben) eine Animation um das Modell als Video rendern. Ihre Lösung sollte zum Beispiel-Video passen.

Lösung

Siehe beiliegenden Code.

Happy Hacking :)