Karlsruher Institut für Technologie
Lehrstuhl für Computergrafik

Prof. Dr.-Ing. Carsten Dachsbacher Dipl.-Inform. Florian Simon

1. Übungsblatt zur Vorlesung Interaktive Computergrafik im SS 2014

Besprechung am Mittwoch, 30.04.2014

Kopieren und übersetzen Sie das Programm von der Vorlesungswebseite (Windows: Visual Studio 2010, Linux: `make && make test`).
In `main.cpp` finden Sie die Klasse `CRender`, deren Methode `void sceneRender();` für das OpenGL-Rendering eines Objektes und die Übergabe der notwendigen Parameter an die GLSL-Shader zuständig ist.
Für das Rendering sind die Vertex- und Fragment-Shader `lighting.vp.glsl` und `lighting.fp.glsl` vorgesehen, die vorerst nur die Vertex-Attribute weitergeben, bzw. einen konstanten Farbwert in das Render-Target schreiben.
Die Anwendung verfügt über einen *Render-Mode*, der bestimmt, welche Technik (diffuse Beleuchtung, spekulare Beleuchtung, Ambient-Occlusion, Texture-Mapping, Normal-Mapping usw.) verwendet wird, und kann über das AntTweakBar-GUI eingestellt werden. Hiermit können auch noch weitere Parameter wie z.B. die Lichtquellen eingestellt werden. Die Kamera lässt sich mit der Maus steuern.

Hinweise:

- Wenn Sie die Shader editieren, können Sie sie zur Laufzeit mit der r-Taste neu laden.
- Linux-User: Zum Kompilieren von GLEW müssen libXi und libXmu installiert sein!

Aufgabe 1 *Beleuchtungsberechnung pro Vertex*

Implementieren Sie zunächst die Funktion

```
void computeLighting( in  vec3  lightPositionWS,
                      in  vec3  normalWS,
                      in  vec3  worldPosition,
                      out float diffuse,
                      out float specular );
```

im Vertex-Shader, die mit dem Blinn-Phong-Modell eine Beleuchtungsberechnung pro Vertex durchführt. Sie können davon ausgehen, dass Welt- und Objektkoordinaten identisch sind, d.h. die Model-View-Matrix enthält nur die Kameraabbildung. Die verwendeten `uniform`-Variablen sind:

```
const int nLights = 2;
uniform vec3 lightPositionWS[ nLights ];
uniform vec4 lightColor [ nLights ];
uniform vec3 vCameraPosition;
```

Sie benötigen die folgenden Vektoren für die Beleuchtungsberechnung:

- die Richtung zur Lichtquelle L
- die Richtung zum Betrachter V
- den Half-Way Vektor $H = \frac{V+L}{||V+L||}$
- Berechnen Sie daraus einen diffusen $\langle L,N \rangle$ und spekularen $\langle H,N \rangle^n$ Reflexionskoeffizienten

Aufgabe 2 *Beleuchtungsberechnung pro Fragment*

Als nächstes soll die Beleuchtungsberechnung im Fragment-Shader vorgenommen werden. Verwenden Sie dazu die interpolierte Vertex-Normale und -Position im World-Space, die im Fragment-Shader über die `in`-Variablen zur Verfügung stehen:

```
in vec3 vNormal;
in vec3 vWorldPosition;
```

Für die Beleuchtungsberechnung können Sie die `computeLighting`-Methode wiederverwenden, indem Sie sie in den Fragment-Shader kopieren und anpassen.

Aufgabe 3 *Normal-Mapping im Tangentenraum*

In der Vorlesung haben Sie Normal-Mapping und den Tangentenraum kennengelernt. Ändern Sie zunächst `computeLighting` so ab, dass die Beleuchtungsberechnung im Tangentenraum stattfindet.
Transformieren Sie dazu L und V (in Weltkoordinaten) mit der `matTangentSpace`-Matrix in den Tangentenraum und berechnen Sie anschließend wie bisher den Half-Way-Vektor und die Koeffizienten. (Überlegen Sie sich, wie Sie die Normale in den Tangentenraum "transformieren" müssen.)
Um schließlich Normal-Mapping zu implementieren, müssen Sie für die Beleuchtungsberechnung nur noch die interpolierte Normale durch die Normale aus der Normal-Map (`in mat4 normalTS`) ersetzen.
Um die Rendering-Performance zu steigern, werden oft möglichst viele Berechnungen vom Fragment- in den Vertex-Shader verschoben. Welche Berechnungen können Sie pro Vertex ausführen? Welche Berechnungen verbleiben dann im Fragment-Shader? Hinweis: Sie benötigen den Tangentenraum nicht im Fragment-Shader!

Aufgabe 4 *Post-Processing: Tone-Mapping und Gammakorrektur*

Die bisher berechneten Bildwerte befinden sich, je nach Intensität der Lichtquellen, nicht unbedingt im darstellbaren Display-Wertebereich $[0, 1]$. Um die Werte auf den darstellbaren Bereich zu bringen, sollte daher *Tone-Mapping* durchgeführt werden. In dieser Aufgabe soll ein einfacher globaler Tone-Mapping-Operator implementiert werden.
Ein weiteres Problem ist, dass das Ansprechverhalten von (kalibrierten) Displays in der Regel nichtlinear ist, d.h. ein Wert von 1.0 ist z.B. nicht doppelt so hell wie ein Wert von 0.5. Um dies zu beheben, muss eine *Gammakorrektur* durchgeführt werden, die das nichtlineare Ansprechverhalten kompensiert.

Implementieren Sie Gammakorrektur und Tone-Mapping in `postprocess.fp.glsl`, gemäß folgender Formel:

$$c' = (c \cdot 2^e)^{1/\gamma},$$

wobei c und c' der ursprüngliche bzw. angepasste Farbwert ist, und e und γ die `exposure`- bzw. `gamma`-Werte aus den entsprechenden `uniform`-Variablen sind.
Beachten Sie, dass Farb-Texturen in der Regel schon gammakorrigiert sind! Um die Gammakorrektur nach der Beleuchtungsberechnung zu kompensieren, passen Sie in der `main`-Funktion von `lighting.fp.glsl` den ausgelesenen Texturwert `texColor.rgb` wie folgt an:

$$c' = c^\gamma,$$

wobei c und c' erneut der ursprüngliche bzw. angepasste Farbwert, und γ der Gammawert ist.