Computer Graphics – WebGL, Teil 6

1 Beleuchtung

In diesem Teil der Übung werden wir die Beleuchtung von Objekten einbauen. Dazu gibt es verschiedene Möglichkeiten, sowohl für das verwendete Beleuchtungsmodell, als auch für das Koordinatensystem und den Shader, in welchem die Berechnung stattfindet.

Die Berechnung der Beleuchtung im Vertex Shader war lange der Standard, da das sogenannte *Pixel Shading*, d. h. die Berechnung im Fragment Shader, zu aufwendig für die Grafikhardware war. Mit den heutigen Grafikkarten ist dies jedoch kein Problem mehr, und Pixel Shading liefert deutlich bessere Ergebnisse, weshalb wir dies hier verwenden.

Die Berechnung der Beleuchtung erfolgt am besten im Kamera-Koordinatensystem. Dabei ist die Kamera selber im Nullpunkt des Koordinatensystem und der Betrachter schaut entlang der z-Achse. Andere Vektoren zur Berechnung der Beleuchtung müssen daher in dieses Koordinatensystem transformiert werden, was mit der Multiplikation mit der ModelView-Matrix geschieht.

2 Normalenvektoren und Transformationen

Um die Beleuchtung berechnen zu können, werden die Normalenvektoren zu den Flächen benötigt. Obwohl WebGL eigentlich in der Lage wäre, diese für Dreiecke zu berechnen, müssen sie doch explizit angegeben werden. Grund dafür ist, dass oft nicht die korrekten Normalenvektoren auf die Flächen benutzt werden, sondern interpolierte Normalenvektoren. Damit lassen sich auch gekrümmte Flächen mit Dreiecken simulieren. Alle Objekte, die wir mit Beleuchtung zeichnen möchten, müssen deshalb einen weitern Buffer für die Normalenvektoren pro Vertex erhalten und diese an den VertexShader übergeben. Da die eigentliche Berechnung der Beleuchtung dann aber im Fragment Shader erfolgt, müssen die Normalenvektoren vom Vertex Shader weiter an den Fragment Shader übergeben werden.

Normalenvektoren können nicht mit der ModelViewMatrix transformiert werden, da sie sich unter Transformationen anders verhalten. Stattdessen müssen sie mit der transponierten Inversen der oberen linken 3x3-Submatrix der ModelView-Matrix multipliziert werden. Die Berechnung der transponierten Inversen muss nicht ausprogrammiert werden, sondern es gibt in der glMatrix-Bibliothek eine eigenene Funktion `mat3.normalFromMat4()`, die diese Matrix aus der ModelView-Matrix berechnet. Auch diese Matrix muss an den Vertex Shader übergeben werden. Generell muss diese Matrix immer dann auch neu übergeben werden, wenn sich die ModelView-Matrix ändert.

```
gl.uniformMatrix4fv (ctx.uModelViewMatrixId, false, modelViewMatrix);
mat3.normalFromMat4 (normalMatrix, modelViewMatrix);
gl.uniformMatrix3fv (ctx.uNormalMatrixId, false, normalMatrix);
```

3 Vertex Shader

Bei der Berechnung der Beleuchtung im Fragment Shader hat der Vertex Shader die Aufgabe, die Attribute, die ja per Vertex angegeben werden, mittels varying-Variablen an den Fragment Shader weiterzuleiten. Im folgenden ist ein Vertex Shader angegeben, der dies bereitstellt. Die entsprechenden Attribute werden wie bisher mittels Buffer im JavaScript File angegeben, und die uniforms werden in JavaScript direkt gesetzt.

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexColor;
attribute vec3 aVertexNormal;
attribute vec2 aVertexTextureCoord;

uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;
uniform mat3 uNormalMatrix;

varying vec3 vColor;
varying vec2 vTextureCoord;
varying vec3 vNormalEye;
varying vec3 vVertexPositionEye3;

void main ()
{
    // calculate the vertex position in eye coordinates
    vec4 vertexPositionEye4 = uModelViewMatrix * vec4 (aVertexPosition, 1.0);
    vVertexPositionEye3 = vertexPositionEye4.xyz / vertexPositionEye4.w;

     // calculate the normal vector in eye coordinates
    vNormalEye = normalize (uNormalMatrix * aVertexNormal);

    // set texture coordinates for fragment shader
    vTextureCoord = aVertexTextureCoord;
```

```
    // set color for fragment shader
    vColor = aVertexColor;

    // calculate the projected position
    gl_Position = uProjectionMatrix * vertexPositionEye4;
}
```

4 Fragment Shader

Die Berechnung der Beleuchtung erfolgt dann im FragmentShader. Am besten werden die 3 Terme für ambiente, diffuse und spiegelnde Beleuchtung separat berechnet und dann addiert. Damit man die Beleuchtung ein- und ausschalten kann, empfiehlt es sich, eine Variable (hier uEnableLighting) einzuführen. Das erleichtert einerseits das Debugging, andererseits kann man zum Beispiel auch die Lichtquelle darstellen, die selber nicht beleuchtet wird.

Anschliessend ein Gerüst für den Fragment Shader, der noch mit den eigentlichen Berechnungen (siehe unten) ergänzt werden muss:

```
precision mediump float;

uniform bool uEnableTexture;
uniform bool uEnableLighting;

uniform vec3 uLightPosition;
uniform vec3 uLightColor;

varying vec3 vColor;
varying vec2 vTextureCoord;
varying vec3 vNormalEye;
varying vec3 vVertexPositionEye3;

uniform sampler2D uSampler;

const float ambientFactor = 0.2;
const float shininess = 10.0;
const vec3 specularMaterialColor = vec3 (0.4, 0.4, 0.4);

void main ()
{
    vec3 baseColor = vColor;
    if (uEnableTexture)
    {
        baseColor = texture2D (uSampler,
            vec2 (vTextureCoord.s, vTextureCoord.t)) .rgb;
    }

    if (uEnableLighting)
```

```
    {
        // ambient lighting
        vec3 ambientColor = ambientFactor * baseColor.rgb;

        // calculate light direction as seen from the vertex position
        vec3 lightDirectionEye = ...;
        vec3 normal = normalize (vNormalEye);

        // diffuse lighting
        float diffuseFactor = ...;
        vec3 diffuseColor = ...;

        // specular lighting
        vec3 specularColor = vec3 (0, 0, 0);

        if (diffuseFactor > 0.0)
        {
           vec3 reflectionDir = ...;
           vec3 eyeDir = ...;
           float cosPhi = ...;
           float specularFactor = ...;
           specularColor = ...;
        }

        vec3 color = ambientColor + diffuseColor + specularColor;
        gl_FragColor = vec4 (color, 1.0);
    }
    else
    {
        gl_FragColor = vec4 (baseColor, 1.0);
    }
}
```

Aufgabe 1

Definieren sie die Normalenvektoren für den Würfel und setzen Sie die entsprechenden Attribute für den Shader. Die Normalenvektoren müssen im Vertex Shader normalisiert werden. Da die Normalen durch die Interpolation der varying-Variablen skaliert werden können, sollten sie dann auch im Fragment Shader nochmals normalisiert werden.

5 Lichtquelle und Farbe

Die Position der Lichtquelle und die Lichtfarbe sollen auch an die Shader übergeben werden können. Für die Position und Art der Lichtquelle gibt es verschiedene Möglichkeiten, so kann zum Beispiel eine direktionale Lichtquelle verwendet werden, oder die Lichtquelle kann relativ

zur Kameraposition oder in Weltkoordinaten angegeben werden. Die Angabe in Weltkoordinaten wäre für die Spezifikation der Szene am einfachsten, jedoch muss dann die Umwandlung in Kamerakoordinaten für jeden Vertex (oder Pixel) im Shader erfolgen. Dies ist etwas unnötig, da das Ergebnis für alle Vertices (oder Pixel) das Gleiche wäre. Deshalb ist es zweckmässiger, die Lichtposition in *Kamerakoordinaten* an den Shader zu übergeben.

6 Ambiente und diffuse Beleuchtung

Die ambiente Beleuchtung (siehe Gleichung 1) wird von gleichmässig verteiltem Umgebungslicht erzeugt, das auch wieder gleichmässig abgestrahlt wird.

$$I_a = I_{La} \cdot k_a \tag{1}$$

Die diffuse Beleuchtung (siehe Gleichung 2) simuliert ein mattes Material, welches das Licht gleichmässig in alle Richtungen reflektiert. Die Intensität des reflektierten Lichts ist dabei noch abhängig vom Winkel zwischen der Fläche und der Richtung zur Lichtquelle. Um diese zu berechnen, brauchen wir also die Richtung zur Lichtquelle vom aktuellen Punkt aus gesehen.

$$I_d = I_{Ld} \cdot k_d \cdot \cos\Theta \tag{2}$$

An gekrümmten Flächen lassen sich die Beleuchtungseffekte besser beobachten, deshalb sollten sie ausser dem Würfel auch eine Kugel darstellen. Der Programm-Code, der die Kugel definiert und zeichnet, steht ihnen auf Ilias zur Verfügung.

Aufgabe 2

Implementieren Sie die ambiente und diffuse Beleuchtung. Den Koeffizienten für die ambiente Beleuchtung können Sie im Shader als Konstante angeben.

7 Spiegelnde Beleuchtung

Die Specular Reflection oder spiegelnde Beleuchtung (siehe Gleichung 3) simuliert ein glänzendes Material, dass das Licht hauptsächlich in Reflektionsrichtung abstrahlt.

$$I_s = I_{Ls} \cdot k_s \cdot \cos^{n_s} \Phi \tag{3}$$

Dabei ist Φ der Winkel zwischen der idealen Reflektionsrichtung und der betrachteten Richtung.

Aufgabe 3

Implementieren Sie die spiegelnde Reflektion. Dazu brauchen Sie die Richtung vom aktuellen Punkt zur Kamera. Beachten Sie, dass Sie dabei in Kamerakoordinaten rechnen, weshalb sich die Kamera im Ursprung befindet. Zur Berechnung der Reflektionsrichtung können Sie im Shader die Funktion `reflect()` verwenden.

In Abbildung 1 sind Würfel und Kugel mit diffuser und spiegelnder Beleuchtung dargestellt.

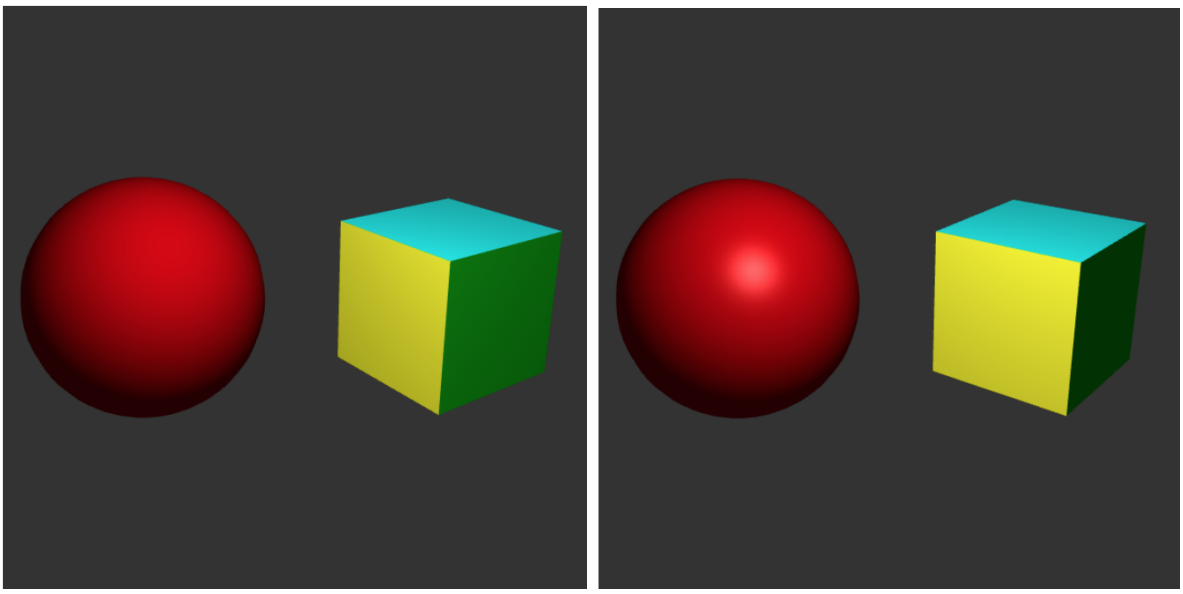

Abbildung 1: Würfel und Kugel, diffus (links) und spiegelnd (rechts) beleuchtet

Aufgabe 4* (optional)

Wie würde ein Shader für ein Computerspiel wie Zelda (Abbildung 2) aussehen?

Abbildung 2: Zelda