





Computergraphik II

Übung 02

OpenGL Teil 2: Shader, Uniforms und (einfache) Beleuchtung

In dieser Einheit werden wir uns im ersten Teil mit der Entwicklung eines einfachen Blinn-Phong-Beleuchtungsshadere für eine Lichtquelle beschäftigen. Im zweiten Teil werden wir uns dann mit verschiedenen Möglichkeiten Uniforms zu verwalten beschäftigen, um dann mehrere Lichtquellen mit Hilfe von Uniform-Buffer-Objects(UBOs) animiert, in die Szene zu integrieren.

Tastaturbelegung:

-  : Programm beenden
-  /  +  : Wireframe an/aus

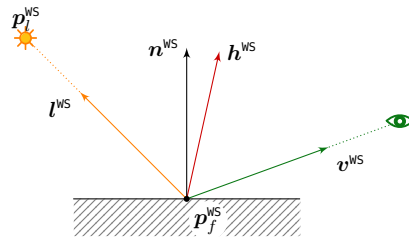
1. Beleuchtung im Shader

In `data / shaders / example.fs.glsl` ist der entsprechende Shader bereits vorbereitet. Die Varyings werden im Vertexshader (`data/shaders/example.vs.glsl`) entsprechend zugewiesen. Die Beleuchtung wird, anders als im Computergraphik1 Framework, im Worldspace berechnet.

调整片段着色器, 以便为给定光源 `LightData one_light` 计算漫射照明部分 `D` :

(a) Passen Sie den Fragmentshader so an, dass der diffuse Beleuchtungsanteil D für die gegebene Lichtquelle `LightData one_light` berechnet wird:

$$D = \max(\langle \mathbf{n}^{\text{WS}}, \mathbf{l}^{\text{WS}} \rangle, 0) \cdot c_l \cdot c_m$$



其中 \mathbf{n}^{WS} 是世界空间中的归一化法线, \mathbf{l}^{WS} 是光源的归一化方向向量, c_l 或 c_m 是光源或材料的颜色。

Wobei \mathbf{n}^{WS} die **normierte** Normale im Worldspace, \mathbf{l}^{WS} der **normierte** Richtungsvektor zur Lichtquelle und c_l bzw. c_m die Farben der Lichtquelle bzw. des Materials sind.

为了使光源的行为更加真实, 其影响受到作为到被照射物体的距离 d 的函数的阻尼因子 a (衰减) 的限制。

- (b) Um das Verhalten der Lichtquelle realistischer zu gestalten wird deren Einfluss durch einen Dämpfungsfaktor a (engl. attenuation) in Abhängigkeit von der Distanz d zum beleuchteten Objekt beschränkt.

$$a := \frac{1}{1 + d^2}$$
$$d := \|\mathbf{p}_l^{\text{WS}} - \mathbf{p}_f^{\text{WS}}\|$$

Berechnen Sie den Dämpfungsfaktor und wenden Sie ihn auf D an.

计算阻尼系数并将其应用于 D 。

Die OpenGL Shading Language (GLSL) ist die Sprache in der wir, in OpenGL, Shader programmieren können. Die Sprache folgt im Wesentlichen der C/C++ Syntax mit einigen eingebauten Typen und Funktionen. Die GLSL bietet uns wie glm^a Typen, um Matrizen (`mat2`, `mat3`, `mat4`) und Vektoren (`vec2`, `vec3`, `vec4`) zu repräsentieren, sowie die üblichen Operationen auf diesen Datentypen. In der folgenden Liste stehen `vecX` bzw `matX` für die jeweiligen Vektor und Matrixtypen.

`vecX a, vecX b; a [+,-,*,/] b;`

Führt den Operator [+,-,*,/] **elementweise** aus.

`vecX a, matX M; M*a;`

Matrix-Vektor Multiplikation.

`matX M, matX N; M*N;`

Matrix-Matrix Multiplikation.

`float dot(vecX a, vecX b)` [↗](#)

Berechnet das Skalarprodukt zwischen a und b : $\langle a, b \rangle$.

`float length(vecX a)` [↗](#)

Berechnet die euklidische Länge des Vektors a : $\|a\| = \sqrt{\sum_{i=0}^{i < X} a_i^2}$

`float distance(vecX a, vecX b)` [↗](#)

Berechnet die euklidische Distanz zwischen den Vektoren a und b : `length(a-b)`

`vecX normalize(vecX a)` [↗](#)

Normiert den Vektor a : $a/\text{length}(a)$

`vec3 cross(vec3 a, vec3 b)` [↗](#)

Berechnet das Kreuzprodukt zwischen a und b : $a \times b$

`vecX max(vecX a, vecX b)` [↗](#)

Berechnet das elementweise Maximum zwischen a und b .

^aGenau genommen ist es andersherum, glm wurde nach den GLSL-Typen und Funktionen modelliert.

2. Uniforms und Uniform Buffer Objects

单个静态光源不是很令人印象深刻，并且由于所有光源参数（位置和颜色）都在着色器中固定，因此每次光源位置发生变化时都需要调整着色器代码。将光参数（类似于相机参数）指定为制服是有意义的。

Eine einzelne, statische Lichtquelle ist nicht sonderlich eindrucksvoll und da alle Lichtparameter (Position und Farbe) fest in dem Shader gesetzt werden, müsste man jedes Mal, wenn sich die Lichtposition ändert den Shader-Code anpassen. Sinnvollerweise werden Lichtparameter (ähnlich wie Kameraparameter) als Uniforms spezifiziert.

调整片段着色器，使光源参数由两个相应的制服组成（`Position (location= 5)` 和 `Color (location= 6)`）。

- (a) Passen Sie den Fragmentshader so an, dass die Lichtparameter aus zwei entsprechenden Uniforms gesetzt werden (`Position (location = 5)` und `Farbe (location = 6)`).

在 `cg2application.cpp` 中的 `void CG2App::render_one_frame()` 中，模拟了八个光源灯[8]。选择一个并相应地放两Uniforms。

- (b) In der `void CG2App::render_one_frame()` in `cg2application.cpp` werden acht Lichtquellen `lights[8]` simuliert. Wählen Sie eine aus und setzen Sie die beiden Uniforms entsprechend.

```
void glUniform4f(GLint location, float x, float y, float z, float w)
void glUniform4fv(GLint location, GLsizei count, const GLfloat* value) ↗
```

Setzt den Wert des Uniforms an der gegebenen location. Entweder aus vier Werten, oder aus einem Array value. Mit `glUniform4fv` können auch Uniform Arrays gesetzt werden, mit `count` wird angegeben wie viele Werte gesetzt werden sollen.

设置给定location的Uniform值。来自四个值，或来自数组值。也可以使用`glUniform4fv`设置均匀性数组，`count`指定应设置多少个值。

Größere Mengen Uniforms auf diese Art und Weise zu setzen ist nicht sehr effizient. Die Menge an klassischen Uniform locations ist begrenzt, außerdem sind die Werte der Uniforms Teil des Shader States. Werden also mehrere Shader verwendet, die auf identische Uniforms Zugriff haben sollen, müssen diese für jeden Shader gesetzt werden. In der OpenGL gibt es natürlich für dieses Problem eine Lösung: Uniform Buffer Objects. Generell kann man alle Daten, die zum Rendering benötigt werden in Buffern ablegen, so auch die Daten der Uniforms. Da die Buffer über ihren Inhalt keinerlei Informationen halten (außer dessen Größe in Bytes) muss die Struktur des Buffers im Shader beschrieben werden. In unserem Beispiel würde das so aussehen:

```
struct LightData{
    vec4 position_ws;
    vec4 color;
    vec4 speed; // This can be ignored !!!
};

// std140 definiert das padding-Verhalten
layout(std140, binding = 0) // binding = 0 : Das 0te UBO
uniform globalLightDataBlock // globalLightDataBlock ist der Name des Uniform-Blocks
{
    LightData lights[8]; // Struktur der eigentlichen Daten
}globalLights; // Name der Uniform-Block-Instanz

// Zugriff auf die Werte in der Instanz (den Daten aus dem UBO)访问实例中的值（来自UBO的数据）
globalLights.lights[0].position;
```

GLSL

缓冲区中数据的结构来自此信息。布局`std140`指定了预期的内存对齐。由于我们在这里只使用`vec4`，因此可确保将值实际紧密地打包在内存中（无填充）。绑定指示应该在哪个统一缓冲区读取数据。

Die Struktur der Daten im Buffer ergibt sich aus diesen Informationen. Das `layout std140` spezifiziert das erwartete Memory alignment. Da wir hier nur `vec4` verwenden, ist sichergestellt, dass die Werte tatsächlich dicht gepackt im Speicher liegen (kein padding). Das `binding` gibt an, an welchem Uniform-Buffer die Daten gelesen werden sollen.

```
void glBindBufferBase(GLenum target, GLuint index, GLuint buffer) ↗
```

Ist eine mit `glBindBuffer` verwandte Funktion, die es erlaubt an Target Arrays zu binden. Manche Buffer Binding Points (aka. targets) (`GL_ATOMIC_COUNTER_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, `GL_UNIFORM_BUFFER`, `GL_SHADER_STORAGE_BUFFER`) repräsentieren mehr als ein Target. An das Target - beispielsweise `GL_UNIFORM_BUFFER` - kann ganz normal mit `glBindBuffer` gebunden werden. Zusätzlich zu diesem sogenannten *general binding point* gibt es aber noch ein Array von indexed binding points. `glBindBufferBase` bindet buffer an beide, den *general binding point* und den ausgewählten indexed binding point.

是一个与`glBindBuffer`相关的函数，它允许绑定到目标数组。一些缓冲区绑定点（aka. 目标）（`GL_ATOMIC_COUNTER_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, `GL_UNIFORM_BUFFER`, `GL_SHADER_STORAGE_BUFFER`）代表多个目标。对于目标 - 例如，`GL_UNIFORM_BUFFER` - 可以像`glBindBuffer`一样正常绑定。除了这个所谓的通用绑定点之外，还有一系列索引绑定点。`glBindBufferBase`将缓冲区绑定到通用绑定点和选定的索引绑定点。

(c) Legen Sie in `CG2App::init_gl_state()` in `cg2application.cpp` einen entsprechend dimensionierten Buffer in `ubo_lights` an und kopieren Sie alle acht Lichter aus `lights` in den Buffer.

在`cg2application.cpp`中的`CG2App::init_gl_state()`中，在`ubo_lights`中创建一个适当大小的缓冲区，并将所有八个灯复制到缓冲区。

(d) Binden Sie den Buffer an das `GL_UNIFORM_BUFFER` Subtarget 0. (`glBindBufferBase`)

- 调整CG2App::render_one_frame(),以便在每个simulate_lights()之后将所有灯光重新复制到缓冲区(glBufferSubData)
- (e) Passen Sie CG2App::render_one_frame() so an, dass nach jedem simulate_lights() alle Lichter neu in den Buffer kopiert werden.(glBufferSubData)
- 调整着色器,以便从UBO计算所有8个灯光的亮度。
- (f) Passen Sie den Shader so an, dass die Beleuchtung für alle 8 Lichter aus ihrem UBO berechnet wird.



```
void glBufferSubData(GLenum target, GLintptr offset,
                    GLsizeiptr size, const void* data) ↗
```

Diese Funktion kopiert Daten in einen an target gebundenen Buffer, ohne ihn neu anzulegen. Der Parameter offset gibt dabei die Zieladresse relativ zum Bufferanfang an. Hinweis: glBufferData legt einen neuen Buffer an und kopiert die Daten (etwa wie ein malloc(...); memcpy(...);), wohingegen glBufferSubData die Daten in einen bestehenden, ausreichend großen Buffer kopiert (also nur memcpy(...);)!

此函数将数据复制到目标绑定缓冲区而不重新创建它。offset参数指定相对于缓冲区开头的目标地址。注意:glBufferData创建一个新缓冲区并复制数据(例如malloc(...); memcpy(...);),而glBufferSubData将数据复制到现有的足够大的缓冲区(即只有memcpy(...);)!

3. Zusatzaufgabe: Spekularer Beleuchtungsanteil

- (a) Erweitern Sie den Shader so, dass zusätzlich zum diffusen- auch der spekularen Beleuchtungsanteil berechnet wird:

$$S = c_l \cdot u \cdot [\max(\langle \mathbf{n}^{WS}, \mathbf{h}^{WS} \rangle, 0)]^s$$

$$u := \begin{cases} 1, & \text{falls } \langle \mathbf{n}^{WS}, \mathbf{l}^{WS} \rangle \geq 0 \\ 0, & \text{sonst} \end{cases}$$

$$\mathbf{h}^{WS} = \frac{\mathbf{l}^{WS} + \mathbf{v}^{WS}}{\|\mathbf{l}^{WS} + \mathbf{v}^{WS}\|}$$

Den Exponenten s können Sie dabei fest wählen (z.B. $s = 64$).