Ćwiczenie 5. Oświetlenie

Witold Alda

Spis treści

1 Co robimy w tym ćwiczeniu? 1

2 Informacje wprowadzające 1

3 Shadery do wypróbowania 3
3.1 Oświetlenie z kilku punktowych źródeł światła. 3
3.2 Oświetlenie reflektorem 5
3.3 Cartoon shading 7
3.4 Symulowanie mgły 9

1 Co robimy w tym ćwiczeniu?

Ćwiczenie jest w całości poświęcone światłu i modelom oświetlenia. Opiera się w dużej mierze na przykładach z tutoriali Tut 09, Tut 10 i Tut 11. Na początek proponuję obejrzeć i wypróbować gotowe przykłady. W Tut_09 omówione jest podstawowe oświetlenie wyliczane w wierzchołkach i następnie liniowo interpolowane. W Tut_10 zastosowane jest światło punktowe, interpolowane między wierzchołkami lub wyliczane wprost dla piksela (warto zwrócić uwagę na różnice w jakości oświetlenia liczonego tymi dwoma sposobami). W Tut_11 pokazano efekty oświetlenia połyskliwego (z odblaskami) liczonego według trzech modeli: Phonga, Blinna-Phonga i Gaussa.

Następnie należy w wybranych przykładach zmienić standardowe shadery na nieco inne, przedstawione poniżej. Wiąże się to z dokonaniem niewielkich modyfikacji w kodach OpenGL.

2 Informacje wprowadzające

W omawianych przykładach wykorzystujemy najprostszy model oświetlenia, zwany modelem ADS (od składników *Ambient*, *Diffuse* i *Specular*). Charakteryzuje się

on uwzględnieniem tylko jednokrotnego odbicia światła od powierzchni obiektu – tego, które jest skierowane wprost do obserwatora (kamery). Pozostałe odbicia – pomijamy.

Obliczenia oświetlenia są wykonywane w shaderach. W tym celu dostarcza się do nich (w postaci zmiennych Uniform) niezbędne atrybuty. W przykładach są one zgromadzone w strukturze `ProgramData` i zawierają:

- informacje o atrybutach źródła światła: kierunek padania światła, jego intensywność i barwę, w przypadku punktowego źródła światła – współrzędne jego położenia, w przypadku światła *ambient* – jego intensywnośći barwę; liczba tych atrybutów zmienia się od przykładu do przykładu;
- dwie macierze przekształceń, które są charakterystyczne nie tylko dla omawianych przykładów, ale w ogóle dla powszechnie przyjętego stylu programowania oświetlenia (opis poniżej)

Pierwsza z wspomnianych wyżej macierzy po prostu przenosi nas do układu związanego z kamerą (*camera space*). Używana była jawnie już wcześniej (po raz pierwszy w przykładach z tutoriala 06). W shaderze nosi nazwę `modelToCameraMatrix`. Może przyjmować różne wartości dla różnych obiektów (a nawet dla pojedynczych wierzchołków).

W przykładach z oświetleniem mamy dwa obiekty – płaszczyznę, która jest swego rodzaju podstawą sceny, oraz cylinder. Przed rysowaniem każdego z nich, w funkcji `display()`, jest ustawiana wartość macierzy funkcją `glUniformMatrix4fv`. Obliczenie samej wartości, w tych przykładach, jest dosyć rozbudowane, ale generalnie sprowadza się do podstawienia bieżącej macierzy transformacji.

Druga z macierzy `normalModelToCameraMatrix` jest ważna i charakterystyczna dla oswietlenia. Jest to macierz 3x3, której zadaniem jest zachowanie poprawnego charakteru wektorów normalnych po przeskalowaniu.

Zauważmy, że równolegle z transformacją wierzchołków, następuje transformacja stowarzyszonych z nimi normalnych. To jest logiczne, że normalne obracają się i przesuwają razem z wierzchołkami

Zwróćmy uwagę, że w programie są użyte trzy macierze transformacji. Pierwsza z nich, `modelToWorldMatrix` służy do rozmieszczenia poszczególnych obiektów i ich elementów na scenie. Macierz ta zmienia się dla każdego obiektu. Dla ustalenia uwagi: drzewko wczytane z pliku xml ma współrzędne lokujące je na początku układu współrzędnych. `modelToWorldMatrix` przesuwa je na właściwe miejsce.

Drugą macierzą jest `worldToCameraMatrix`. Przedstawia ona współrzędne wszystkich obiektów (czyli współrzędne świata) względem położenia kamery. Macierz ta zmienia się, gdy wymuszamy ruch kamery w trakcie przelotu.

Trzecią macierzą jest `cameraToClipMatrix`, która wykonuje rzutowanie perspektywiczne. Jej wartości zmieniają się tylko gdy zmieniamy rozmiary i/lub proporcje okna na ekranie (u nas robi to funkcja `reshape()`).

3 Shadery do wypróbowania

W tym punkcie opiszemy kilka shaderów, które mogą wymienić standardowe shadery z tutoriala. Użycie nowych shaderów wymaga wprowadzenia niewielkich zmian do przykładów związanych Są one następujące:

- Oświetlenie z kilku punktowych źródeł światła.
- Oświetlenie reflektorem.
- Oświetlenie przypominające efekt filmu animowanego (tak się nazywa: 'cartoon shading', ale zaznaczmy: starego filmu animowanego).
- Symulowanie mgły.

3.1 Oświetlenie z kilku punktowych źródeł światła.

Rzecz jest w zasadzie prosta: należy zastosować równania oświetlenia dla każdego źródła, a następnie zsumować wyniki. W shaderze oświetlenie jest liczone na podstawie oświetlenia w wierzchołkach i następnie interpolowane. Shader jest napisany w stylu przykładu vertex_point_lighting.cpp z tutorialu Tut_10, ale ma dodane światło *ambient* i *specular* oraz parametry materiałowe dla każdego rodzaju światła (współczynniki Ka, Kd, Ks oraz Shininess). Nie mniej należy go dołączyć do tego właśnie przykładu i zmodyfikować listę zmiennych uniform.

Zasadnicze przetwarzanie odbywa się na poziomie wierzchołków. Poniższy vertex shader jest następujący:

```
#version 330

layout (location = 0) in vec3 position;
layout (location = 2) in vec3 normal;

out vec3 interpColor;

struct LightInfo {
  vec3 lightPos;        // Light position in eye coords.
  vec3 lightIntensity;  // Light intesity (amb., diff., and spec.)
};
```

```
uniform LightInfo lights[5];

// Material parameters
uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity
uniform vec3 Ks;            // Specular reflectivity
uniform float Shininess;    // Specular shininess factor

uniform mat4 modelToCameraMatrix;
uniform mat3 normalModelToCameraMatrix;
uniform Projection
{
mat4 cameraToClipMatrix;
};

vec3 ads( int lightIndex, vec4 position, vec3 norm )
{
    vec3 s = normalize( vec3(lights[lightIndex].lightPos - position) );
    vec3 v = normalize(vec3(-position));
    vec3 r = reflect( -s, norm );
    vec3 I = lights[lightIndex].lightIntensity;
    return
        I * ( Ka +
            Kd * max( dot(s, norm), 0.0 ) +
            Ks * pow( max( dot(r,v), 0.0 ), Shininess ) );
}

void main()
{
vec4 cameraPosition = (modelToCameraMatrix * vec4(position, 1.0));
gl_Position = cameraToClipMatrix * cameraPosition;
  vec3 eyeNorm = normalize( normalModelToCameraMatrix * normal);

    // Evaluate the lighting equation, for each light
    Color = vec3(0.0);
    for( int i = 0; i < 5; i++ )
        interpColor += ads( i, cameraPosition, normCamSpace );

}
```

Odpowiadający mu fragment shader jedynie przekazuje barwę pikseli. Jest

praktycznie powtórzeniem shadera `CpolorPassthrough.frag` z tutoriala Tut_10.

```
#version 330

in vec3 interpColor;

layout( location = 0 ) out vec4 outputColor;

void main() {
    outputColor = vec4(interpColor, 1.0);
}
```

3.2 Oświetlenie reflektorem

W tym przykładzie oświetlenie jest liczone *per pixel* i vertex shader wylicza przetransformowane położenia wierzchołków. Poniższe shadery należy połączyć z przykładem Fragment_Point_Lighting.cpp z tutoriala Tut_10. Poniższe shadery odpowiadają tym o nazwie FragmentLighting z tego tutoriala. Pewne modyfikacje, jak wspomniano są niezbędne, ponieważ przekazywane są dodatkowe zmienne uniform - określające kierunek reflektora i kąt stożka światła, a także dodane parametry materiałowe.

```
#version 330

layout (location = 0) in vec3 position;
layout(location = 1)  in vec4 inDiffuseColor;
layout (location = 2) in vec3 normal;

out vec3 modelSpacePosition;
out vec4 diffuseColor;
out vec3 vertexNormal;

uniform mat4 modelToCameraMatrix;

uniform Projection
{
mat4 cameraToClipMatrix;
};

void main()
{
```

```
gl_Position = cameraToClipMatrix * (modelToCameraMatrix * vec4(position, 1.0));

modelSpacePosition = position;
vertexNormal = normal;
diffuseColor = inDiffuseColor;
}
```

Właściwe obliczenia dla stożka światła odbywają się we fragment shaderze:

```
#version 330

in vec3 modelSpacePosition;
in vec4 diffuseColor;
in vec3 vertexNormal;

struct SpotLightInfo {
    vec4 position;   // Position in eye coords
    vec3 intensity;
    vec3 direction;  // Direction of the spotlight in eye coords.
    float exponent;  // Angular attenuation exponent
    float cutoff;    // Cutoff angle (between 0 and 90)
};
uniform SpotLightInfo Spot;

uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity
uniform vec3 Ks;            // Specular reflectivity
uniform float Shininess;    // Specular shininess factor

layout( location = 0 ) out vec4 outputColor;

vec3 adsWithSpotlight( )
{
    vec3 s = normalize( vec3( Spot.position) - modelSpacePosition );
    vec3 spotDir = normalize( Spot.direction);
    float angle = acos( dot(-s, spotDir) );
    float cutoff = radians( clamp( Spot.cutoff, 0.0, 90.0 ) );
    vec3 ambient = Spot.intensity * Ka;

    if( angle < cutoff ) {
        float spotFactor = pow( dot(-s, spotDir), Spot.exponent );
```

```
        vec3 v = normalize(vec3(-modelSpacePosition));
        vec3 h = normalize( v + s );

        return
            ambient +
            spotFactor * Spot.intensity * (
              Kd * max( dot(s, vertexNormal), 0.0 ) +
              Ks * pow( max( dot(h,vertexNormal), 0.0 ), Shininess )
            );
    } else {
        return ambient;
    }
}

void main() {
    outputColor = vec4(adsWithSpotlight(), 1.0);
}
```

3.3 Cartoon shading

Cartoon shading polega na redukcji poziomów jasności w renderowanym obiekcie, do kilku. Modyfikację na leży wykonać w przykładzie Fragment_Point_Lighting.cpp z tutoriala Tut_10. Vertex shader może wyglądać następująco:

```
#version 330

layout(location = 0) in vec3 position;
layout(location = 2) in vec3 normal;

out vec4 diffuseColor;
out vec3 vertexNormal;
out vec3 modelSpacePosition;

uniform mat4 modelToCameraMatrix;

uniform Projection
{
mat4 cameraToClipMatrix;
};

void main()
```

```
{
gl_Position = cameraToClipMatrix * (modelToCameraMatrix * vec4(position, 1.0));

vertexNormal = normal;
modelSpacePosition = position;
diffuseColor = vec4(1.0);
}
```

Własciwe cieniowanie wykonuje się we *fragment shaderze*. Oświetlenie typu diffuse jest kwantowane na zadaną (w założeniu niewielką) liczbę poziomów jasności - w przykładzie ustawioną na 3.

```
#version 330
in vec3 vertexNormal;
in vec3 modelSpacePosition;

struct LightInfo {
    vec4 position;
    vec3 intensity;
};
uniform LightInfo Light;

uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity

const int levels = 3;
const float scaleFactor = 1.0 / levels;

layout( location = 0 ) out vec4 outputColor;

vec3 toonShade( )
{
    vec3 s = normalize( Light.position.xyz - modelSpacePosition.xyz );
    vec3 ambient = Ka;
    float cosine = dot( s, vertexNormal );
    vec3 diffuse = Kd * floor( cosine * levels ) * scaleFactor;

    return Light.intensity * (ambient + diffuse);
}
```

```
void main() {
    outputColor = vec4(toonShade(), 1.0);
}
```

3.4 Symulowanie mgły

Symulowanie mgły polega na zastąpieniu oryginalnej barwy wyliczonej w punkcie sceny, średnią ważoną tej barwy i zadanej barwy mgły. Waga jest wyliczana na podstawie odległości od obserwatora – im dalej, tym wpływ barwy mgły jest większy. Proponuję wykonać ręczną zmianę fragment shadera z przykładu Fragment_Point_Lighting.cpp z tutoriala Tut_10 (tego samego co poprzednio). Zmianę najlepiej dokonać w shaderze FragmentLighting_PCN, czyli tym, który uwzględnia kolor obiektu ustawiony w programie OpenGL. Zmiana polega na wczytaniu dwóch granicznych wartości zasięgu mgły, jako zmiennych uniform, np. maxDist i minDist, wczytania koloru mgły np. fogColor, a następnie wyliczenia dla składowej z każdego wierzchołka w położeniu position czynnika zamglenia:

$$fogFactor = \frac{maxDist - abs(position.z))}{maxDist - mindist}$$

fogFactor powinien zostać obcięty do przedziału [0,1]. Następnie wyjściowy kolor powinien być wyliczony jako średnia ważona:

$$diffuseColor = mix(indiffuseColor, fogColor, fogFactor);$$

Czy kolejność kolorów w argumentach funkcji mix ma znaczenie?

fogFactor nie musi być wyliczany jako czynnik liniowo zależny od odległości. Czesem lepsze wyniki można uzyskac stosując zależności eksponencjalne:

$$fogFactor = exp(-d/z)$$

$$fogFactor = exp(-(dz)^2)$$

Proszę spróbować ich użyć.