Computação Gráfica e Interfaces

2017-2018

Fernando Birra

Iluminação e Sombreamento - WebGL

2017-2018

Fernando Birra

Objetivos

- Implementar um modelo de iluminação em WebGL
- Iluminação nos vértices vs. fragmentos
  - Gouraud shading
  - Phong shading

Escolha do Referencial

- Um modelo de iluminação é responsável por determinar a cor final dum objeto, num determinado ponto da sua superfície, observada a partir dum determinado ponto de vista e conhecidas as propriedades das luzes e do material que compõe o objeto.

- O modelo de iluminação de Phong assenta na relação espacial entre um conjunto de vetores, aplicados no ponto **P**, onde se procede à avaliação do modelo:

  - vetor **L** - vetor que aponta para a fonte de luz considerada

  - vetor **N** - vetor define a direção perpendicular à superfície

  - vetor **V** - vetor que aponta para o observador

- A relação (ângulos formados) entre quaisquer destes vetores é independente do referencial usado (referencial do mundo ou referencial da câmara)

Escolha do Referencial

- Podemos optar por avaliar o modelo usando o referencial do mundo ou da câmara
- Usar o referencial da câmara tem algumas vantagens…
- Para usar o referencial da câmara é preciso transformar pontos de World Coordinates para Eye Coordinates (Ref. da câmara), multiplicando-os pela matriz $M_{ModelView}$.
- Note-se que a matriz $M_{ModelView}$ é composta por duas partes:
  - $M_{ModelView} = M_{View} * M_{Model}$

Transformações de modelação para instanciar objetos na cena

Transformação que converte pontos do referencial do mundo (WC) para o referencial da câmara. Ver lookAt()

Escolha do Referencial

- O que fazer relativamente às luzes?
- Para se efetuar a determinação da iluminação no referencial da câmara, a posição (luzes pontuais) ou direção (luzes direcionais) de cada fonte de luz terão que ser convertidas para o referencial da câmara.
- Há 3 hipóteses para a especificação da posição/direção das fontes de luz:
  - Especificada diretamente no referencial da câmara
    - Não será necessário transformar a posição/direção pois já se encontra definida no referencial pretendido.
  - Especificadas no referencial do mundo
    - Será necessário converter do referencial do mundo para o da câmara.
  - Especificadas no referencial dum objeto específico
    - Será necessário converter do referencial do objecto em particular para o da câmara

Escolha do Referencial

- Há 3 hipóteses para a especificação da posição/direção das fontes de luz:
  - Especificada diretamente no referencial da câmara
    - Não será necessário transformar a posição/direção pois já se encontra definida no referencial pretendido.
  - Especificadas no referencial do mundo
    - Será necessário **converter do referencial do mundo para o da câmara**.
  - Especificadas
    - Será neces
      para o da c

Usar a matriz Mview (ver função lookAt()) para transformar a posição da luz de WC para o referencial da câmara e usar $((Mview)^T)^{-1}$ para transformar a direção.

Escolha do Referencial

- Há 3 hipóteses para a especificação da posição/direção das fontes de luz:
  - Especificada diretamente no referencial da câmara
    - Não será necessário transformar a posição/direção pois já se encontra definida no referencial pretendido.
  - Especificadas no re
    - Será necessário câmara.
  - Especificadas no referencial dum objeto específico
    - Será necessário **converter do referencial do objecto em particular para o da câmara**

Usar a matriz Mmodelview a aplicar ao objeto em causa para transformar a posição da luz do referencial do objeto para o referencial da câmara e usar $((Mmodelview)^T)^{-1}$ para transformar a direção.

Iluminação nos vértices

Iluminação nos vértices

- Vamos proceder à determinação duma cor em cada vértice, usando o modelo de iluminação

- Essa cor será depois passada com variável **varying** para ser interpolada durante a conversão em fragmentos (pixels)

Iluminação nos vértices

Vertex Shader

Vertex Shader
Início

```
attribute vec4 vPosition;
attribute vec4 vNormal;

uniform mat4 mModelView;
uniform mat4 mNormals;     // inverse transpose of modelView
uniform mat4 mProjection;

varying vec4 fColor;

void main()
{
    fColor = …
    gl_Position = mProjection * mModelView * vPosition;
}
```

Posição do vértice

Normal no vértice

A cor que irá ser calculada após a avaliação do modelo

Vertex Shader
Os parâmetros* do modelo

Posição da luz

```
const vec4 lightPosition = vec4(0.0, 1.8, 1.3, 1.0);
```

```
const vec3 materialAmb = vec3(1.0, 0.0, 0.0);
const vec3 materialDif = vec3(1.0, 0.0, 0.0);
const vec3 materialSpe = vec3(1.0, 1.0, 1.0);
const float shininess = 6.0;
```

coeficientes de reflexão do material Ka, Kd, Ks (RGB)

```
const vec3 lightAmb = vec3(0.2, 0.2, 0.2);
const vec3 lightDif = vec3(0.7, 0.7, 0.7);
const vec3 lightSpe = vec3(1.0, 1.0, 1.0);
```

intensidades da fonte de luz: Ia, Id, Is (RGB)

```
vec3 ambientColor = lightAmb * materialAmb;
vec3 diffuseColor = lightDif * materialDif;
vec3 specularColor = lightSpe * materialSpe;
```

Ia*Ka, Id*Kd, Is*Ks

* Embora se estejam a usar constantes, os valores deveriam corresponder a variáveis do tipo uniform, passadas pela aplicação para o shader.

Vertex Shader
Referencial da Câmara

```
attribute vec4 vPosition;

uniform mat4 mModelView;    // model-view transformation

void main()
{
    ...
    // position in camera frame
    vec3 posC = (mModelView * vPosition).xyz;
    ...
}
```

descartar w

Posição do vértice no referencial da câmara

Vertex Shader
Determinação de L(Variante I)

Exemplo com **lightPosition** especificado em coordenadas da câmara (luz move-se solidária com a câmara)

```
void main()
{
    ...
    vec3 L; // Normalized vector pointing to light at vertex


    if(lightPosition.w == 0.0)
        L = normalize(lightPosition.xyz);
    else
        L = normalize(lightPosition.xyz - posC);
    ...
}
```

Vertex Shader
Determinação de N

```
uniform mat4 mNormals;

void main()
{
    ...

    // normal vectors are transformed to camera frame using a
    // a matrix derived from mModelView (see MV.js code)

    vec3 N = normalize( (mNormals * vNormal).xyz);

    ...
}
```

Matriz calculada a partir da matriz ModelView, mas adequada à transformação de vetores*:
$mNormals = ((mModelView)^T)^{-1}$

* Ver final da secção 6.8 do livro Interactive Computer Graphics A Top-Down approach with WebGL, 7th edition.

Vertex Shader
Determinação de L(Variante II)

```
uniform mat4 mView;          // Matriz resultante de lookAt(), p.ex.
uniform mat4 mViewNormals;   // Matriz inversa da transposta de mView

void main()
{
    ...
    vec3 L; // Normalized vector pointing to light at vertex


    if(lightPosition.w == 0.0)
        L = normalize((mViewNormals*lightPosition).xyz);
    else
        L = normalize((mView*lightPosition).xyz - posC);
    ...
}
```

Exemplo com **lightPosition** especificado em coordenadas do mundo (luz permanece fixa à cena)

Se lightPosition é especificado em WC, então necessita ser transformado para o referencial da câmara

* Neste exemplo necessitaríamos de passar ao shader a matriz Mview, para além da matriz MmodelView. Assim como a sua homóloga para transformar vértices.

Vertex Shader
Determinação de L(Variante III)

Exemplo com **lightPosition** especificado em coordenadas do objeto (luz permanece fixa em relação ao objeto inicial/primitivo)

```
void main()
{
    ...
    vec3 L; // Normalized vector pointing to light at vertex


    if(lightPosition.w == 0.0)
        L = normalize((mNormals*lightPosition).xyz);
    else
        L = normalize((mModelView*lightPosition).xyz - posC);
    ...
}
```

Se lightPosition é especificado em Object Coordinates, então necessita ser transformado para o referencial da câmara

Vertex Shader
Determinação de V (Variante I)

Projeção perspetiva com o centro de projeção na origem

```
void main()
{
    ...

    // Eye is at origin in camera frame (see lookAt())
    // thus V = -posC (for perspective projection only)

    vec3 V = normalize(-posC);

    ...
}
```

Vertex Shader
Determinação de V (Variante II)

Projeção paralela com as projetantes alinhadas com o eixo Z

```
void main()
{
    ...

    // Eye is at origin in camera frame (see lookAt())
    // thus V = -posC (for perspective projection only)

    vec3 V = vec3(0,0,1); // Point towards the viewer located at (0,0,+inf)

    ...
}
```

Vertex Shader
Determinação de R (Modelo de Phong)

```
void main()
{
    ...

    vec3 R = reflect(L,N);

    ...
}
```

Vertex Shader
Determinação de H (Modelo de Phong-Blinn)

```
void main()
{
    ...

    vec3 H = normalize(L+V);

    ...
}
```

Vertex Shader
Reflexão difusa

```
...
void main()
{
    ...
    float diffuseFactor = max( dot(L,N), 0.0 );

    vec3 diffuse = diffuseFactor * diffuseColor;
    ...
}
```

Impede retro-iluminação!

A reflexão difusa é máxima quando a luz incide perpendicularmente à superfície

diffuseColor tem o valor pré-calculado de Id * Kd

Vertex Shader
Reflexão especular

```
...
void main()
{
    ...
    float specularFactor = pow(max(dot(N,H), 0.0), shininess);
    vec3 specular = specularFactor * specularColor;

    if( dot(L,N) < 0.0 ) {
        specular = vec3(0.0, 0.0, 0.0);
    }
    ...
}
```

intensidade da reflexão especular

specularColor tem o valor previamente calculado de Is * Ks

No caso da luz estar a incidir no lado de trás da face, não há qualquer reflexão especular.

Vertex Shader
Adição das várias componentes

```
…

void main()
{
    ...
    // add all 3 components from the illumination model
    // (ambient, diffuse and specular)

    fColor = vec4(ambientColor + diffuse + specular, 1.0);
    ...
}
```

ambientColor tem o valor previamente calculado de Ia * Ka

Vertex Shader
Completo

```
const vec4 lightPosition = vec4(0.0, 1.8, 1.3, 1.0);

const vec3 materialAmb = vec3(1.0, 0.0, 0.0);
const vec3 materialDif = vec3(1.0, 0.0, 0.0);
const vec3 materialSpe = vec3(1.0, 1.0, 1.0);
const float shininess = 6.0;

const vec3 lightAmb = vec3(0.2, 0.2, 0.2);
const vec3 lightDif = vec3(0.7, 0.7, 0.7);
const vec3 lightSpe = vec3(1.0, 1.0, 1.0);

vec3 ambientColor = lightAmb * materialAmb;
vec3 diffuseColor = lightDif * materialDif;
vec3 specularColor = lightSpe * materialSpe;

attribute vec4 vPosition;
attribute vec4 vNormal;

uniform mat4 mModelView;// model-view transformation
uniform mat4 mNormals;  // model-view transformation
for normals
uniform mat4 mView;         // view transformation
uniform mat4 mViewNormals;  // view transf. for vectors
uniform mat4 mProjection;   // projection matrix

varying vec4 fColor;

void main()
{
    vec3 posC = (mModelView * vPosition).xyz;

    vec3 L;

    if(lightPosition.w == 0.0)
       L = normalize((mViewNormals*lightPosition).xyz);
    else
       L = normalize((mView*lightPosition).xyz - posC;

    vec3 V = vec3(0,0,1);
    vec3 H = normalize(L+V);
    vec3 N = normalize( (mNormals * vNormal).xyz);

    float diffuseFactor = max( dot(L,N), 0.0 );
    vec3 diffuse = diffuseFactor * diffuseColor;

    float specularFactor = pow(max(dot(N,H), 0.0),
shininess);
    vec3 specular = specularFactor * specularColor;

    if( dot(L,N) < 0.0 ) {
        specular = vec3(0.0, 0.0, 0.0);
    }

    gl_Position = mProjection * mModelView * vPosition;

    fColor = vec4(ambientColor + diffuse + specular,
1.0);

}
```

Iluminação nos vértices

Fragment Shader

Fragment Shader Completo

```
precision mediump float;

varying vec4 fColor;

void main() {
    gl_FragColor = fColor;
}
```

O vertex shader irá calcular a cor do vértice e atribuir o valor a esta variável.

A cor final do pixel será a cor interpolada a partir dos valores dos vértices.

Iluminação nos fragmentos

Iluminação nos fragmentos

- A aplicação do modelo de iluminação ao nível de cada fragmento produz resultados mais realistas (melhor seguimento de superfícies curvas)

- O fragment shader terá que ter acesso aos vetores **N**, **L**, **V,** no referencial da câmara, correspondente ao pixel (fragmento) que está a ser gerado

- O problema é que os atributos (vPosition e vNormal) são passados juntamente com os vertices (vertex data), pelo que os valores necessários terão que ser propagados para o fragment shader (via variáveis varying por forma a serem interpoladas)

- Finalmente, com tudo disponível, o fragment shader poderá aplicar o modelo e determinar a cor final do pixel

Iluminação nos fragmentos Vertex Shader

Vertex Shader
Posição no ref. câmara

```
const vec4 lightPosition = vec4(0.0, 1.8, 1.3, 1.0);

attribute vec4 vPosition;   // vertex position in modelling coordinates
attribute vec3 vNormal;     // vertex normal in modelling coordinates

uniform mat4 mModelView;    // model-view transformation
uniform mat4 mNormals;      // model-view transformation for normals
uniform mat4 mView;         // view transformation (for points)
uniform mat4 mViewNormals;  // view transformation (for vectors)
uniform mat4 mProjection;   // projection matrix

varying vec3 fNormal;
…

void main(){
    vec3 posC = (mModelView * vPosition).xyz;
    ...
}
```

Vertex Shader
Interpolação do vetor N

```
...
attribute vec4 vNormal;     // vertex normal in modelling coordinates
...
varying vec3 fNormal;       // normal vector in camera space

void main(){
    ...
    // compute normal in camera frame
    fNormal = (mNormals * vNormal).xyz;

    ...
}
```

o vetor normal irá ser interpolado no interior dos triângulos, tendo um valor diferente para cada fragmento

Vertex Shader
Interpolação do vetor L

```
...
const vec4 lightPosition = vec4(0.0, 1.8, 1.3, 1.0);
...
```

In World Coordinates

```
varying vec3 fLight;        // light vector in camera space

void main(){
    ...
    // compute light vector in camera frame
    if(lightPosition.w == 0.0)
        fLight = normalize((mViewNormals * lightPosition).xyz);
    else
        fLight = normalize((mView*lightPosition).xyz - posC);

    ...
}
```

o vetor fLight irá ser interpolado no interior dos triângulos, tendo um valor diferente para cada fragmento

Vertex Shader
Interpolação do vetor V

```
...
varying vec3 Viewer;        // view vector in camera space

void main(){
    ...
    // Compute the view vector

    // fViewer = -posC; // Perspective projection

    fViewer = vec3(0,0,1); // Parallel projection only
    ...
}
```

o vetor fViewer irá ser interpolado no interior dos triângulos, tendo um valor diferente para cada fragmento

Vertex Shader
Completo

```
const vec4 lightPosition = vec4(0.0, 1.8, 1.3, 1.0);

attribute vec4 vPosition;   // vertex position in modelling coordinates
attribute vec4 vNormal;     // vertex normal in modelling coordinates

uniform mat4 mModelView;    // model-view transformation
uniform mat4 mNormals;      // model-view transformation for normals
uniform mat4 mView;         // view transformation (for points)
uniform mat4 mViewNormals;  // view transformation (for vectors)
uniform mat4 mProjection;   // projection matrix

varying vec3 fNormal;       // normal vector in camera space
varying vec3 fLight;        // Light vector in camera space
varying vec3 fViewer;       // View vector in camera space

void main(){
    // compute position in camera frame
    vec3 posC = (mModelView * vPosition).xyz;

    // compute normal in camera frame
    fNormal = (mNormals * vNormal).xyz;
```

Vertex Shader
Completo

```
...
// compute light vector in camera frame
if(lightPosition.w == 0.0)
    fLight = normalize((mViewNormals * lightPosition).xyz);
else
    fLight = normalize((mView*lightPosition).xyz - posC);

    // Compute the view vector
    // fViewer = -fPosition; // Perspective projection
    fViewer = vec3(0,0,1); // Parallel projection only

    // Compute vertex position in clip coordinates (as usual)
    gl_Position = mProjection * mModelView * vPosition;
}
```

Iluminação nos fragmentos
Fragment Shader

Fragment Shader
Os parâmetros* do modelo

```
const vec3 materialAmb = vec3(1.0, 0.0, 0.0);
const vec3 materialDif = vec3(1.0, 0.0, 0.0);
const vec3 materialSpe = vec3(1.0, 1.0, 1.0);
const float shininess = 6.0;
```

coeficientes de reflexão do material Ka, Kd, Ks (RGB)

```
const vec3 lightAmb = vec3(0.2, 0.2, 0.2);
const vec3 lightDif = vec3(0.7, 0.7, 0.7);
const vec3 lightSpe = vec3(1.0, 1.0, 1.0);
```

intensidades da fonte de luz: Ia, Id, Is (RGB)

```
vec3 ambientColor = lightAmb * materialAmb;
vec3 diffuseColor = lightDif * materialDif;
vec3 specularColor = lightSpe * materialSpe;
```

Ia*Ka, Id*Kd, Is*Ks

* Embora se estejam a usar constantes, os valores deveriam corresponder a variáveis do tipo uniform, passadas pela aplicação para o shader.

Fragment Shader
Determinação de L,N,V e H

```
void main()
{
    ...

    vec3 L = normalize(fLight);
    vec3 V = normalize(fViewer);
    vec3 N = normalize(fNormal);

    vec3 H = normalize(L+V);

    ...
}
```

Fragment Shader
Reflexão difusa

```
...
void main()
{
    ...
    float diffuseFactor = max( dot(L,N), 0.0 );

    vec3 diffuse = diffuseFactor * diffuseColor;
    ...
}
```

Impede retro-iluminação!

A reflexão difusa é máxima quando a luz incide perpendicularmente à superfície

diffuseColor tem o valor pré-calculado de Id * Kd

Fragment Shader
Reflexão especular

```
...
void main()
{
    ...
    float specularFactor = pow(max(dot(N,H), 0.0), shininess);
    vec3 specular = specularFactor * specularColor;

    if( dot(L,N) < 0.0 ) {
        specular = vec3(0.0, 0.0, 0.0);
    }
    ...
}
```

intensidade da reflexão especular

specularColor tem o valor previamente calculado de Is * Ks

No caso da luz estar a incidir no lado de trás da face, não há qualquer reflexão especular.

Fragment Shader
Adição das várias componentes

```
...

void main()
{
    ...
    // add all 3 components from the illumination model
    // (ambient, diffuse and specular)

    gl_FragColor = vec4(ambientColor + diffuse + specular, 1.0);
    ...
}
```

ambientColor tem o valor previamente calculado de Ia * Ka

Fragment Shader
Completo

```
precision mediump float;

varying vec3 fPosition;
varying vec3 fNormal;

const vec3 materialAmb = vec3(1.0, 0.0, 0.0);
const vec3 materialDif = vec3(1.0, 0.0, 0.0);
const vec3 materialSpe = vec3(1.0, 1.0, 1.0);
const float shininess = 6.0;

const vec3 lightAmb = vec3(0.2, 0.2, 0.2);
const vec3 lightDif = vec3(0.7, 0.7, 0.7);
const vec3 lightSpe = vec3(1.0, 1.0, 1.0);

vec3 ambientColor = lightAmb * materialAmb;
vec3 diffuseColor = lightDif * materialDif;
vec3 specularColor = lightSpe * materialSpe;

varying vec3 fLight;
varying vec3 fViewer;
...
```

Fragment Shader
Completo

```
void main() {
    vec3 L = normalize(fLight);
    vec3 V = normalize(fViewer);
    vec3 N = normalize(fNormal);
    vec3 H = normalize(L+V);

    float diffuseFactor = max( dot(L,N), 0.0 );
    vec3 diffuse = diffuseFactor * diffuseColor;

    float specularFactor = pow(max(dot(N,H), 0.0), shininess);
    vec3 specular = specularFactor * specularColor;

    if( dot(L,N) < 0.0 ) {
        specular = vec3(0.0, 0.0, 0.0);
    }

    gl_FragColor = vec4(ambientColor + diffuse + specular, 1.0);
}
```

Transformação de vetores normais

Transformação de pontos vs. vetores normais

- Seja M a matriz que transforma pontos entre dois referenciais

- Precisamos determinar uma matriz M' capaz de transformar vetores normais entre esses mesmos dois referenciais

- Claramente, a matriz M não serve, pois poderá ter incluídas translações e um vetor permanece inalterado perante uma translação.

Transformação de pontos vs. vetores normais

- Considere-se o ponto **p** numa superfície com normal **n**, no referencial de partida, assim como um ponto **x**, de tal modo que

$$(\mathbf{x}-\mathbf{p}).\mathbf{n}=0$$

(**x**-**p**) é um vetor tangente à superfície em p

- Ao aplicar uma transformação M ao ponto **p** obtém-se M**p** e, de forma análoga, **x** será transformado em M**x**.
- Seja **n'** a nova normal, no ponto M**p**, após a transformação.
- Então:

$$(M\mathbf{x}-M\mathbf{p}).\mathbf{n'}=0, \text{ sempre que } (\mathbf{x}-\mathbf{p}).\mathbf{n}=0$$

(M**x**-M**p**) é um vetor no novo referencial que é perpendicular à nova normal (i.e. tangente à superfície no ponto transformado M**p**)

- Ou, de forma equivalente:

$$(M\mathbf{x}-M\mathbf{p})^T\mathbf{n'} = 0 \Leftrightarrow (M(\mathbf{x}-\mathbf{p}))^T\mathbf{n'}=0 \Leftrightarrow (\mathbf{x}-\mathbf{p})^TM^T\mathbf{n'}=0$$

- Claramente, tal verifica-se se:

$$M^T\mathbf{n'} = \mathbf{n} \Leftrightarrow \mathbf{n'} = (M^T)^{-1}\mathbf{n}$$

Transformação de pontos vs. vetores normais

- Se M é uma matriz que transforma pontos dum referencial R1 para um referencial R2
- Então $(M^T)^{-1}$ é a matriz que transforma vetores do referencial R1 para o referencial R2
- Aplicação prática:
    - A matriz ModelView transforma pontos do referencial do objeto para o referencial da câmara (passando pelo referencial do mundo)
    - A matriz inversa da transposta da matriz ModelView transforma os vetores normais do referencial do objeto para o referencial da câmara.
    - O exemplo pode facilmente ser adaptado para transformar outros vetores (direções) do referencial do mundo para o da câmara:
        - usando Mview e $(Mview^T)^{-1}$
- A biblioteca MV.js possui uma função que inverte a transposta da matriz passada como argumento:
    - mNormals = normalMatrix(mModelView)
    - mViewNormals = normalMatrix(mView)