

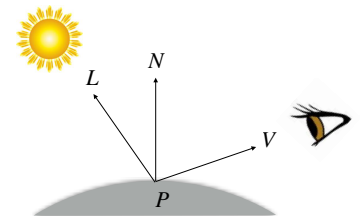
Iluminação e sombreamento (WebGL)

Objetivos

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

Escolha do Referencial

- Um modelo de iluminação é responsável por determinar a cor final dum objeto, num determinado ponto P 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 - aponta para a fonte de luz considerada
 - vetor N - define a direção perpendicular à superfície
 - vetor V - 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 de iluminação 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 Object Coordinates para Eye Coordinates (Ref. da câmara), multiplicando-os pela matriz $\mathbf{M}_{ModelView}$
- Note-se que a matriz $\mathbf{M}_{ModelView}$ é composta por duas partes:

○ $\mathbf{M}_{ModelView} = \mathbf{M}_{View} * \mathbf{M}_{Model}$

Transformações de modelação para instanciar objetos na

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 objeto 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 no referencial de um objeto específico
 - Será necessário Usar a matriz \mathbf{M}_{View} (ver função `lookAt()`) para transformar a posição da luz de WC para o referencial da câmara e usar $(\mathbf{M}_{View}^T)^{-1}$ 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.
 - Especificada no referencial do objeto
 - Usar a matriz $\mathbf{M}_{ModelView}$ a aplicar ao objeto em causa para transformar a posição da luz do referencial do objeto para o referencial da câmara e usar $(\mathbf{M}_{ModelView}^T)^{-1}$ para transformar a direção.
câmara.
 - Especificadas no referencial dum objeto específico
 - Será necessário converter do referencial do objecto em particular para o da câmara

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 como 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;
```

Posição do vértice

Normal no vértice

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

```
varying vec4 fColor;
```

A cor que irá ser calculada após a avaliação do modelo

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

Vertex Shader: os parâmetros* do modelo

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

Posição da luz

```
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;
```

$I_a * K_a$, $I_d * K_d$, $I_s * K_s$

* 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 $\mathbf{M}_{ModelView}$, mas adequada à transformação de vetores*: $\mathbf{M}_{Normals} = (\mathbf{M}_{ModelView}^T)^{-1}$

* 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)

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

```
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);
    ...
}
```

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(mNormalsLight*lightPosition).xyz);
    else
        L = normalize(mModelViewLight*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 perspectiva 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()  
{
```

```
...
```

Impede retro-iluminação!

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

```
float diffuseFactor = max( dot(L,N), 0.0 );
```

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

```
...  
}
```

diffuseColor tem o valor pré-calculado de $I_d * K_d$

Vertex Shader: reflexão especular

```
...  
void main()  
{
```

intensidade da reflexão especular

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

specularColor tem o valor
previamente calculado de $I_s * K_s$

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

```
...  
}
```

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 $I_a * K_a$

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); // Projeção paralela...
    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 vértices (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 fNormal 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 fViewer; // 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 = -posC; // 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 K_a , K_d , K_s
(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: I_a , I_d , I_s (RGB)

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

$I_a * K_a$, $I_d * K_d$, $I_s * K_s$

* 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()  
{  
    ...
```

Impede retro-iluminação!

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

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

diffuseColor tem o valor pré-calculado de $I_d * K_d$

Fragment shader: reflexão especular

```
...  
void main()  
{
```

intensidade da reflexão especular

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

specularColor tem o valor
previamente calculado de $I_s * K_s$

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

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 $I_a * K_a$

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);  
}
```