Iluminación en OpenGL

Informática Gráfica I

Material de: **Elena Gómez** y **Rubén Rubio**
Basado en apuntes de: **Antonio Gavilanes** y **Ana Gil**
{mariaelena.gomez,rubenrub}@ucm.es

UNIVERSIDAD COMPLUTENSE MADRID

Contenido

Iluminación en OpenGL

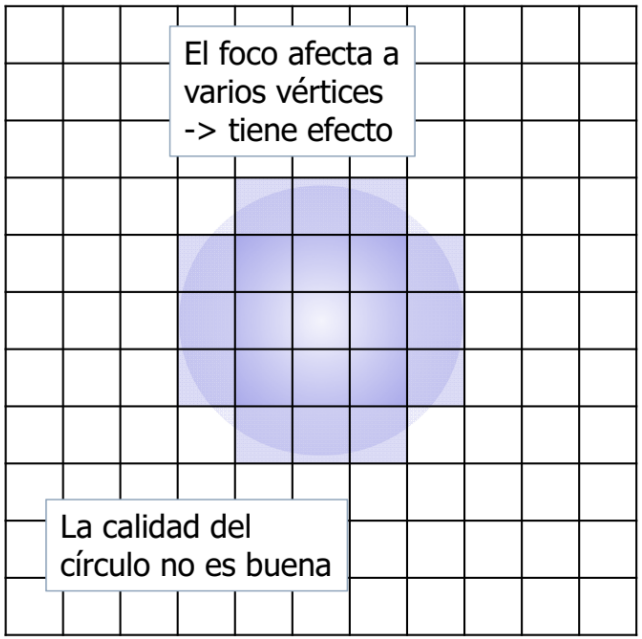
- Las primeras versiones de OpenGL (y las actuales a través del *compatibility profile*) disponían una API para configurar fuentes de luz globales en la escena.
- Se manipulaban con funciones como `glLight`, `glLightModel`, `glMaterial` y `glEnable(GL_LIGHTING)`.
- OpenGL 3 considera obsoleto el soporte para iluminación y no está disponible en el *core profile*. La iluminación ahora la ha de implementar el programador en los shaders.
- En la asignatura hemos implementado un shader `light` relativamente genérico con una intefaz semejante a la de OpenGL 2.

Iluminación en OpenGL

- El shader `light` realiza los cálculos de iluminación para cada fragmento (aproximadamente, para cada píxel de la imagen).
- Se utilizan las normales que se obtienen para cada fragmento por interpolación de las normales asignadas a los vértices.
- Alternativamente, se podría hacer el cálculo en el shader de vértices, interpolando luego el color obtenido. La calidad (sobre todo de la luz especular) dependería entonces en gran medida del **número de vértices de la malla**.

Iluminación en OpenGL

- La calidad del sombreado de Gourard (es decir, si se implementase en el shader de vértices) depende, en gran medida, del número de vértices de la malla.



Rejilla (11 x 11)

Rectángulo

Iluminación en OpenGL

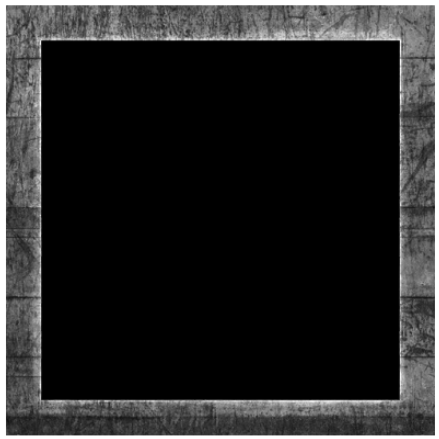
- Los **cálculos de iluminación en coordenadas de la cámara**.
  Una vez transformados los elementos geométricos (puntos y vectores) por la matriz de modelado y vista.
- Los **elementos geométricos de las fuentes de luz** (posición o dirección de la luz) también tienen que ser transformados por la **matriz de modelado y vista**.
  Hay que configurar las luces antes que los objetos a los que pueden iluminar.

Iluminación con lighting maps

- Para la iluminación se tiene en cuenta el material del objeto (tres colores para sus componentes ambiente, difuso y especular).
- Una alternativa de configuración más fina son los **lighting maps**.
- Se pueden utilizar dos texturas simultáneamente para ampliar la definición del material de los objetos a nivel de fragmentos:
  - *Diffuse map*: texturas de coeficientes de reflexión difusa
  - *Specular map*: textura de coeficientes de reflexión especular
- Se pueden utilizar otros datos, por ejemplo, para incorporar rugosidad al material se utiliza una textura de vectores normales (*bump mapping*).

Iluminación con lighting maps

- *Diffuse map*: una textura que define, por fragmento, el coeficiente de reflexión difuso. Sustituye a los coeficientes difuso y ambiente del material.
- *Specular map*: una textura, de colores grises, que define, por fragmento, el coeficiente de reflexión especular.



Specular map: Los bordes son de metal y brillan (grises), el interior es de madera y no produce brillos (negro).

El shader light

- Los parámetros de la iluminación se pasan al shader light a través de atributos uniformes.
- El material del objeto es un atributo de tipo struct Material:

```
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

uniform Material material;  // material de la entidad
```

- En C++ esto se corresponde con la clase Material, que carga sus atributos sobre los del struct del shader.

La clase Material

```
class Material
{
public:
    Material() = default; virtual ~Material() = default;
    Material(glm::vec3 color, float shininess=8.0); // color mat.
    void upload(Shader& shader) const;
    // [...] Getter y setters de los atributos
    void setCopper();

protected:
    // Coeficientes de reflexión
    glm::vec3 ambient, diffuse, specular;
    GLfloat expF = 0; // exponente para la reflexión especular
};
```

La clase Material

```
void Material::upload(Shader& lighting) const
{
    lighting.setUniform("material.ambient", ambient);
    lighting.setUniform("material.diffuse", diffuse);
    lighting.setUniform("material.specular", specular);
    lighting.setUniform("material.shininess", expF);
}

void Material::setCopper()
{
    ambient = {0.19125, 0.0735, 0.0225};
    diffuse = {0.7038, 0.27048, 0.0828};
    specular = {0.256777, 0.137622, 0.086014};
    expF = 12.8;
}
```

La clase EntityWithMaterial

Como en anteriores ocasiones, añadimos un nuevo tipo de entidad, `EntityWithMaterial`, para renderizar las entidades con material iluminadas usando el shader `light`.

```
class EntityWithMaterial : public Abs_Entity {
public:
    EntityWithMaterial();
    void setMaterial(const Material& m) { material = m; };

protected:
    Material material;
};
```

La clase EntityWithMaterial

```
EntityWithMaterial::EntityWithMaterial() {
    mShader = Shader::get("light");
}

void
EntityWithMaterial::render(mat4 const& modelViewMat) const
{
    mShader->use();
    // Carga los atributos del material en la GPU
    mMaterial.upload(*mShader);
    upload(modelViewMat * mModelMat);
    mMesh->render();
}
```

El shader light

- Las características de las luces se pasan al shader a través de atributos uniformes de tipo array de structs específicos:

```
const int NR_DIR_LIGHTS = 2;
const int NR_POS_LIGHTS = 4;
const int NR_SPOT_LIGHTS = 4;

uniform DirLight dirLights[NR_DIR_LIGHTS];
uniform PosLight posLights[NR_POS_LIGHTS];
uniform SpotLight spotLights[NR_SPOT_LIGHTS];
```

- El número máximo de luces de cada tipo está acotado por sendas constantes (también lo estaba en la API de OpenGL 2), pero esas constantes se pueden cambiar.

El shader light

El shader combina la luz producida por cada fuente y calculada con funciones específicas que detallaremos más adelante.

```
void main() {
    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(- FragPos);
    vec3 result = vec3(0);

    // (1) Directional lighting
    for (int i = 0; i < NR_DIR_LIGHTS; i++)
        if (dirLights[i].enabled)
            result += calcDirLight(dirLights[i], norm, viewDir);

    // [...] Lo mismo (2) Point lights y (3) Spot lights
    FragColor = vec4(result, 1.0);
}
```

El shader light

En lo siguiente se describen los tres tipos de fuentes de luz, `DirLight`, `PosLight` y `SpotLight`, siempre con los siguiente elementos:

1. sus parámetros, declarados en el `struct` del shader `light` en GLSL.
2. la clase de C++ que reproduce estos parámetros y permiten configurar las luces.
3. el código de la función del shader que implementa el tipo de luz.

La clase Light

La clase abstracta `Light` proporciona la interfaz y atributos comunes a todas las fuentes de luz.

```
class Light {
public:
    virtual ~Light() = default;
    // Enciende o apaga la luz
    bool enabled() const; void setEnabled(bool enabled);
    // Carga sus atributos en la GPU
    virtual void upload(Shader& shader,
                        const glm::mat4& modelViewMat) const;
    void unload(Shader& shader);

    void setAmb(const glm::vec3& ind);
    void setDiff(const glm::vec3& ind);
    void setSpec(const glm::vec3& ind);                    // [sigue]
```

La clase Light

```
protected:
    Light(std::string name);
    Light(const std::string& name, int id);

    // Identificador de la luz (en el shader)
    std::string lightID;
    // Si la luz está apagada o encendida
    bool bEnabled;

    // Atributos lumínicos y geométrico de la fuente de luz
    glm::vec3 ambient = {0.1, 0.1, 0.1};
    glm::vec3 diffuse = {0.5, 0.5, 0.5};
    glm::vec3 specular = {0.5, 0.5, 0.5};
};
```

La clase Light

```
Light::Light(const std::string& name, int id)
 : lightID(name + "[" + std::to_string(id) + "]")
{
   // lightID: dirLights[0], posLights[1], etc.
}
```

`lightID` es el identificador del `struct` de la luz en el shader, que contiene al menos:

- Los 3 colores de sus componentes (`ambient`, `diffuse` y `specular`) de tipo `vec3`.
- Un booleano `enabled` que indica si la luz está encendida.

La clase Light

```
void Light::upload(Shader& shader,
                   glm::mat4 const& modelViewMat) const
{
    // Transfer light properties to the GPU
    shader.setUniform(lightID + ".ambient", ambient);
    shader.setUniform(lightID + ".diffuse", diffuse);
    shader.setUniform(lightID + ".specular", specular);
    shader.setUniform(lightID + ".enabled", bEnabled);
}

void Light::unload(Shader& shader) {
    shader.setUniform(lightID + ".enabled", false);
}
```

Luz direccional

Su atributo específico es

- `direction`: la direccion de la luz en coordenadas de vista.

```
struct DirLight {
    vec3 direction;

    vec3 ambient;    // atributos comunes
    vec3 diffuse;
    vec3 specular;

    bool enabled;
};
```

La clase DirLight

```
class DirLight : public Light {
public:
    explicit DirLight(int id = 0);

    virtual void upload(...) const override;
    void setDirection(const glm::vec3& dir);

protected:
    glm::vec4 direction;
};

void DirLight::setDirection(const glm::vec3& dir) {
    direction = glm::vec4(dir, 0.0);  // 0 = dirección
}
```

La clase DirLight

```
void DirLight::upload(Shader& shader,
                      mat4 const& modelViewMat) const
{
   Light::upload(shader, modelViewMat);
   shader.setUniform(lightID + ".direction",
                     vec3(modelViewMat * direction));
}
```

Implementación de DirLight en el shader

El cálculo de las componentes es el mismo que en `simple_light` y común para todas las fuentes de luz.

```
void calcComponents(vec3 lightDir, vec3 normal, vec3 viewDir,
   vec3 lightAmbient, vec3 lightDiffuse, vec3 lightSpecular,
   out vec3 ambient, out vec3 diffuse, out vec3 specular)
{
   float diff = max(dot(normal, lightDir), 0.0);  // diffuse
   // Specular shading
   vec3 reflectDir = reflect(- lightDir, normal);
   float spec = pow(max(dot(viewDir, reflectDir), 0.0),
                    material.shininess);
   ambient  = lightAmbient * material.ambient;
   diffuse  = diff * lightDiffuse * material.diffuse;
   specular = spec * lightSpecular * material.specular;
}
```

Implementación de DirLight en el shader

La función `calcDirLight` calcula el color producido por una luz direccional usando la función anterior `calcComponents`.

```
vec3 calcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
    vec3 lightDir = - light.direction;
    vec3 ambient, diffuse, specular;

    calcComponents(lightDir, normal, viewDir,
                   light.ambient, light.diffuse, light.specular,
                   ambient, diffuse, specular);

    // Simplemente suma las tres componentes
    return ambient + diffuse + specular;
}
```

Luz posicional

Su atributos específicos son

- `position`: posición de la luz en coordenadas de vista.
- `constant`, `linear` y `quadratic` son los coeficientes del factor de atenuación de la luz

$$f_a = \frac{1}{k_c + k_l \cdot d + k_q \cdot d^2}$$

La intensidad de todos los componentes de luz se multiplica por este factor, donde d es la distancia del fragmento a `position`.

```
struct PosLight {
   vec3 position;
   float constant, linear, quadratic;
   // [...] Atributos comunes
};
```

La clase PosLight

```
class PosLight : public Light {
public:
    explicit PosLight(int id = 0);

    virtual void upload(...) const override;

    void setPosition(const glm::vec3& dir);
    void setAttenuation(GLfloat kc, GLfloat kl, GLfloat kq);

protected:
    glm::vec4 position;
    // Coeficientes de atenuación
    GLfloat constant = 1, linear = 0, quadratic = 0;
};
```

La clase PosLight

```
void PosLight::setPosition(const glm::vec3& pos) {
    position = glm::vec4(pos, 1.0); // 1 = posición
}

void PosLight::setAttenuation(GLfloat kc, GLfloat kl,
                              GLfloat kq) {
    constant = kc;
    linear = kl;
    quadratic = kq;
}
```

La clase PosLight

```
void PosLight::upload(Shader& shader,
                      mat4 const& modelViewMat) const
{
   Light::upload(shader, modelViewMat);

   shader.setUniform(lightID + ".position",
                     vec3(modelViewMat * position));

   shader.setUniform(lightID + ".constant", constant);
   shader.setUniform(lightID + ".linear", linear);
   shader.setUniform(lightID + ".quadratic", quadratic);
}
```

Implementación de PosLight en el shader

Las novedades en el cálculo de la luz posicional son que

- La dirección de la luz se calcula uniendo la posición del fragmento con la posición de la luz.
- Se añade un factor de atenuación en función de la distancia.

```
float calcAttenuation(float dist, float kc, float kl, float kq)
{
   return 1.0 / (kc + kl * dist + kq * (dist * dist));
}
```

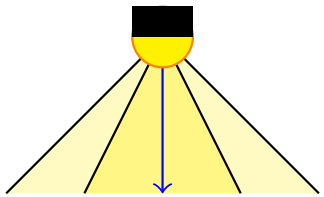
Implementación de PosLight en el shader

```
vec3 calcPosLight(PosLight light, vec3 normal, vec3 fragPos,
                  vec3 viewDir) {
    vec3 lightDir = normalize(light.position - fragPos);
    vec3 ambient, diffuse, specular;
    calcComponents(lightDir, normal, viewDir,
                   light.ambient, light.diffuse, light.specular,
                   ambient, diffuse, specular);

    // Atenuación
    float attenuation = calcAttenuation(
        distance(light.position, fragPos),
        light.constant, light.linear, light.quadratic);

    return attenuation * (ambient + diffuse + specular);
}
```

Foco

Es un subtipo de luz posicional. Sus atributos específicos son

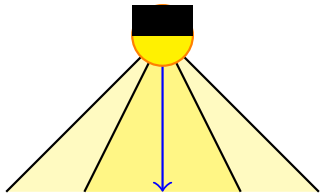
- `direction`: dirección a la que apunta el foco.
- `cutOff` y `outerCutOff`: cosenos de los ángulos interior y exterior del cono de luz.



$$f_i = \text{clamp}\left(\frac{\cos\theta - \texttt{outerCutOff}}{\texttt{cutOff} - \texttt{outerCutOff}}, 0, 1\right)$$

La intensidad de las componente difusa y especular se multiplica por este factor (además de por f_a), donde θ es el ángulo entre la dirección del foco y el rayo que une el framento con el foco.

Foco



$$f_i = \text{clamp}\left(\frac{\cos\theta - \texttt{outerCutOff}}{\texttt{cutOff} - \texttt{outerCutOff}}, 0, 1\right)$$

- El foco ilumina con plena intensidad en el radio interior del cono de luz.
- La luz se atenúa del radio interior al radio exterior.
- El foco no tiene ningún efecto fuera del radio externo.

```
struct SpotLight {
    vec3 direction;
    float cutOff, outerCutOff;
    // [...] Atributos comunes con PosLight
};
```

La clase SpotLight

```
class SpotLight : public PosLight {
public:
    SpotLight(const glm::vec3& pos = {0, 0, 0}, int id = 0);

    virtual void upload(...) const override;
    void setDirection(const glm::vec3& dir);
    void setCutoff(float inner, float outer);

protected:
    // Atributos del foco
    glm::vec3 direction = {0, 0, -1};
    GLfloat cutoff = 0.91, outerCutoff = 0.82;
    GLfloat exp = 0;
};
```

La clase SpotLight

```
SpotLight::SpotLight(const glm::vec3& pos, int id)
{
    lightID = "spotLights[" + std::to_string(id) + "]";
    position = glm::vec4(pos, 1.0);
};

void SpotLight::setCutoff(float inner, float outer)
{
    cutoff = cos(glm::radians(inner));
    outerCutoff = cos(glm::radians(outer));
}
```

La clase SpotLight

```
void SpotLight::upload(Shader& shader,
                       glm::mat4 const& modelViewMat) const
{
   PosLight::upload(shader, modelViewMat);

   shader.setUniform(lightID + ".direction",
                     vec3(modelViewMat * vec4(direction, 0.0)));

   shader.setUniform(lightID + ".cutOff", cutoff);
   shader.setUniform(lightID + ".outerCutOff", outerCutoff);
}
```

Implementación de SpotLight en el shader

```
vec3 calcSpotLight(SpotLight light, vec3 normal, vec3 fragPos,
                   vec3 viewDir) {

    // [...] Ambiente, difusa, especular y atenuación
    // se calculan igual que en calcPosLight

    // Intensidad del foco
    float theta = dot(lightDir, normalize(-light.direction));
    float epsilon = light.cutOff - light.outerCutOff;
    float intensity = clamp((theta - light.outerCutOff)
                            / epsilon, 0.0, 1.0);

    return attenuation *
            (ambient + intensity * (diffuse + specular));
}
```

Luces en las escenas

- La clase `Scene` (o sus subclases) contiene gran parte de las luces de la escena en forma de atributos.
- Se construyen y definen sus atributos en `Scene::init`.
- En `Scene::render` de `Scene` se hace `upload` de todas ellas.
- En `Scene::unload` se desactivan (de la GPU) para que no afecten a otras escenas.
- Otros métodos de la clase pueden activarlas o desactivarlas, por ejemplo, en respuesta a eventos del teclado.

Luces en las entidades

- El resto de las luces aparecen en la escena porque suelen formar parte de entidades; por ejemplo, la luz de una lámpara de una habitación, los focos de un coche, o la luz de exploración de una nave sobre un planeta.
- Estas luces se declaran como atributos de la entidad de la que forman parte y se construyen y establecen sus características en la constructora de esa entidad
- Si una luz está asociada a una entidad y se mueve de forma solidaria con la entidad (por ejemplo, los focos de los faros de un coche, el foco de exploración de una nave, etc.), debe fijarse su posición en el `render()` de la entidad (ver luces dinámicas, más adelante)

Luces estáticas (o fijas)

- Son las luces (remotas, locales, focos o no) que no cambian su posición durante la renderización de la escena.
- La forma de comprobar que una luz es estática consiste en mover la cámara y ver que los efectos de la luz no cambian.
- La posición debe restablecerse cada vez que cambie la matriz de modelado-vista.
- Cuando la cámara se mueve, hay que volver a fijar la posición de la luz (pues se espera en coordenadas de vista). Por ello, al principio de `Scene::render`, se llama a su su método `upload(shader, cam.viewMat())`.

Luces dinámicas

- Luces que cambian de posición, independientemente de la cámara
  - **Focos en objetos de la escena que se mueven** (por ejemplo, la luz de exploración debajo de una nave)
  - La posición inicial se fija en la constructora de la entidad que las contiene; generalmente esa posición suele ser el punto $(0, 0, 0)$
  - La posición se actualiza después de multiplicar por la matriz `modelMat`, en el `render()` de la entidad que las contiene

Luces dinámicas

- Luces que se mueven con la cámara
  - **Luz de minero posicional** (que puede ser foco o no) en la cámara
  - La posición se establece una vez, después de que la matriz de modelado-vista se ha fijado
  - Si se trata de un foco colocado en la cámara, su dirección de emisión es $(0, 0, -1)$
  - La posición se fija al inicializar y se actualiza en `Scene::render` llamando con ella a `upload`

Observaciones

- Los cálculos de la iluminación en OpenGL se realizan en coordenadas de cámara.
- Hay que establecer las fuentes de luz y encenderlas antes de renderizar los objetos que van a iluminar.
- Los focos deben iluminar vértices de una malla para que se vea el cono de emisión. Si no lo hacen su luz no aparece.
- Texturas: el shader `light` no soporta texturas pero se podría extender para que lo hiciera y mezclara los colores como en el shader `texture`.