Lights, Materials, and Transparency in OpenGL

Francisco Santos

Departamento de Engenharia Informática
Instituto Superior Técnico

Computer Graphics, 2009
Outline

1. Lighting
   - Introduction
   - Light Model
   - Defining Light Sources
   - Defining Materials

2. Transparency
   - Introduction
   - Blending Function
   - Rendering Order
Outline

1. Lighting
   - Introduction
   - Light Model
   - Defining Light Sources
   - Defining Materials

2. Transparency
   - Introduction
   - Blending Function
   - Rendering Order
Why do we need light?

- An object appears 3D because of the small variations of its surface color.
- These colors depend on the interaction between one or more light sources and the object’s material.
- When light strikes an object, part of it is absorbed and the other is reflected, possibly striking other objects.
- We need global calculations to determine an object’s color but OpenGL works on a vertex-by-vertex basis.
Outline

1. Lighting
   - Introduction
   - Light Model
     - Defining Light Sources
     - Defining Materials

2. Transparency
   - Introduction
   - Blending Function
   - Rendering Order
An Overview of the Light Model

- **Light sources**: emit red, green, and blue light.
- **Materials**: reflect red, green, and blue light.
- Model breaks down light into four components: **ambient**, **diffuse**, **specular**, and **emissive**.
- Each component is computed independently and then added together in the end.
- Light source types: **point sources**, **spotlights**, and **ambient sources**.
- Point sources: located at infinity or at a finite distance from the objects.
Model indirect light sources.
Backfaces will not appear entirely darkened.
OpenGL supports an ambient value per light.

\[ i_{amb} = m_{amb} \otimes s_{amb} \]  \hspace{1cm} (1)
Diffuse Light Component

- Light is scattered equally in all directions.
- Model dull surfaces (e.g. cardboard).
- Intensity depends on: angle between light source \( \mathbf{l} \) and surface normal \( \mathbf{n} \).
- Intensity does \textbf{not} depend on the viewer’s position.
- \textbf{Maximal} intensity: \( \mathbf{l} \) coincides with \( \mathbf{n} \).

\[
\mathbf{i}_{\text{diff}} = (\mathbf{n} \cdot \mathbf{l}) \mathbf{m}_{\text{diff}} \otimes \mathbf{s}_{\text{diff}}
\]
Specular Light Component

- Model surface highlights.
- Viewer sees the surface’s curvature, the light source’s direction and its location.
- Intensity depends on: angle between light reflection $r$ and viewpoint $v$.
- **Maximal** intensity: $r$ coincides with $v$.

$$i_{spec} = (r \cdot v)^{m_{shi}} m_{spec} \otimes s_{spec} \quad (3)$$
Emissive Light Component

- Model glowing objects (i.e., that generate photons).
- Unaffected by the viewpoint or the light source’s position.
- Does not contribute to the scene’s overall lighting.

\[ i_{emi} = m_{emi} \]  

(4)
Outline

1. Lighting
   - Introduction
   - Light Model
   - Defining Light Sources
   - Defining Materials

2. Transparency
   - Introduction
   - Blending Function
   - Rendering Order
Before lighting calculations can be performed

**Turn on the lights**

- `glEnable(GL_LIGHTING)`: activate lighting calculations.
- `glEnable(GL_LIGHT0)`: activate the first light source.

**Compute normals**

- `glNormal3*()`: assign normals (per vertex or poly).

**Activate depth buffer: light only frontmost surfaces**

- `glutInitDisplayMode(GLUT_DEPTH | ...)`
- `glClear(GL_DEPTH_BUFFER_BIT | ...)`
- `glEnable(GL_DEPTH_TEST)`
The light source configuration function

```c
void glLight*(GLenum light, GLenum par, type val)
```

- **GLenum light**: which of OpenGL's eight lights is being configured: `GL_LIGHT0`...`GL_LIGHT7`.
- **GLenum par**: the name of the parameter to change.
- **type val**: the parameter's new value.
Changing the Light Source’s Color

Example: setting the color of the light source

```c
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };  
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };  
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };  

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);  
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);  
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);  
```

- **Warning!** All color values are 4D vectors (RGBA).
Creating a Point Light

- Has a position in space.
- Shines uniformly in all directions.
- `GL_POSITION`: a 4D vector: $(x, y, z, w)$.
- For point lights, $w = 1.0$.

Example: light at $(-1.0, 0.5, 2.0)$

```c
GLfloat light_position[] = { -1.0, 0.5, 2.0, 1.0 };  
gllightfv(GL_LIGHT0, GL_POSITION, light_position);
```
Creating a Directional Light

- Positioned infinitely away.
- Light rays are parallel.
- \texttt{GL\_POSITION}: a 4D vector: $(x, y, z, w)$.
- For directional lights, $(x, y, z)$ is a direction and $w = 0.0$.

Example: light pointing toward negative z axis

```c
GLfloat light_position[] = { 0.0, 0.0, 1.0, 0.0 };
gllightfv(GL\_LIGHT0, GL\_POSITION, light\_position);
```
Creating a Spotlight

- Light shape confined to a cone.
- `GL_SPOT_DIRECTION`: a 3D vector.
- `GL_SPOT_EXPONENT`: how concentrated the light is.
- `GL_SPOT_CUTOFF`

Example: spot at (0.0, 5.0, 0.0), aimed at negative y axis

```c
GLfloat spot_pos[] = { 0.0, 5.0, 0.0, 1.0 };  
GLfloat spot_dir[] = { 0.0, -1.0, 0.0 };   
gllightfv(GL_LIGHT0, GL_POSITION, spot_pos);  
gllightfv(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0); 
gllightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_dir);  
```
Global Ambient Light

- Each light can contribute to the global illumination (GL_AMBIENT color component).
- We can also set the scene’s ambient light independently:

Example: setting the scene’s ambient light

```c
GLfloat amb[] = { 0.2, 0.2, 0.2, 1.0 }; 
gllightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
```
Transforming Lights

Lights can be transformed like any other geometric entity.

Example: rotating spot, idle object and observer

```c
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_dir);
...
glPushMatrix();
goRotatef(theta, 0.0, 0.0, 1.0);
glLightfv(GL_LIGHT0, GL_POSITION, spot_pos);
goPopMatrix();
glutSolidSphere(0.5, 10, 10);
```
Outline

1. Lighting
   - Introduction
   - Light Model
   - Defining Light Sources
   - Defining Materials

2. Transparency
   - Introduction
   - Blending Function
   - Rendering Order
Specifying a Material

The material configuration function

```c
void glMaterial*(GLenum face, GLenum par, type val)
```

- **GLenum face**: to which faces should the material be applied to: `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.
- **GLenum par**: the name of the parameter to change.
- **type val**: the parameter’s new value.
An Example of a Material Definition

Example: changing the material state

```c
GLfloat ambient[] = { 0.3, 0.0, 0.0, 1.0 };  
GLfloat diffuse[] = { 0.6, 0.0, 0.0, 1.0 };  
GLfloat specular[] = {0.8, 0.6, 0.6, 1.0 };  
GLfloat shininess = 32.0; /* [0..128] */  
glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);  
glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);  
glMaterialfv(GL_FRONT, GL_SPECULAR, specular);  
glMaterialf(GL_FRONT, GL_SHININESS, shininess);
```

- Only **one** set of material properties per state.
- Color components are **4D** vectors (RGBA).
- Color values indicate fraction of light reflected.
1. Lighting
   - Introduction
   - Light Model
   - Defining Light Sources
   - Defining Materials

2. Transparency
   - Introduction
   - Blending Function
   - Rendering Order
Why do we need the Alpha Value?

- Consider a fragment with an RGBA value assigned to it.
- RGB is the fragment’s color and $A$, alpha, is its opacity.
- Alpha is used to blend the fragment’s color with that of the pixel stored in the framebuffer.
- Without blending, new fragments would simply replace the colors in the framebuffer.
- We can model transparent and translucent objects by setting $A$ to values less than 1.
Before we can use Blending

Activate Blending

- `glEnable(GL_BLEND)`

Define a blending function

- `glBlendFunc*()`: how fragment colors are combined with pixel colors in the framebuffer.

Assign alpha values to the scene

- `glColor*()` and `glMaterial*()`: change the objects’ alpha values.
- `gLlght*()`: modify the light sources’ alpha values.
- `glClearColor*()`: change the framebuffer’s clear color.
Outline

1. Lighting
   - Introduction
   - Light Model
   - Defining Light Sources
   - Defining Materials

2. Transparency
   - Introduction
   - Blending Function
   - Rendering Order

Francisco Santos
Lights, Materials, and Transparency in OpenGL
Defining the Blending Function

The blending definition function

```c
void glBlendFunc(GLenum src, GLenum dest)
```

- `GLenum src`: describes the source blending factor.
- `GLenum dest`: describes the destination blending factor.
Source and Destination Blending Factors

### Example: common blending factors

<table>
<thead>
<tr>
<th>Blend Factor</th>
<th>Source Color</th>
<th>Destination Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_ONE_MINUS_SRC_ALPHA</td>
<td>$1, 1, 1, 1$</td>
<td>$(A_D, A_D, A_D, A_D)$</td>
</tr>
<tr>
<td>GL_ONE</td>
<td>$(R_S, G_S, B_S, A_S)$</td>
<td>$(R_D, G_D, B_D, A_D)$</td>
</tr>
<tr>
<td>GL_SRC_COLOR</td>
<td>$(R_S, G_S, B_S, A_S)$</td>
<td>$(R_D, G_D, B_D, A_D)$</td>
</tr>
<tr>
<td>GL_DST_COLOR</td>
<td>$(R_S, G_S, B_S, A_S)$</td>
<td>$(R_D, G_D, B_D, A_D)$</td>
</tr>
</tbody>
</table>

The most common option

```plaintext```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```plaintext```
How the Blending Function Works

The blending function works as follows:

1. **Source Color** $c_S$ is multiplied by the source blending factor $f_S$.
2. **Destination Color** $c_D$ is multiplied by the destination blending factor $f_D$.
3. The two results are added together to produce the blended color.

Mathematically, this can be represented as:

$$
\text{blended color} = c_S \cdot f_S + c_D \cdot f_D
$$
Outline

1 Lighting
   - Introduction
   - Light Model
   - Defining Light Sources
   - Defining Materials

2 Transparency
   - Introduction
   - Blending Function
   - Rendering Order
Why might the Rendering Order be Important?

- The yellow and cyan triangles both have alpha = 0.75.
- Assume $f_S = (A_S, A_S, A_S, A_S)$ and $f_D = 1 - f_S$.
- Since alpha $= 0.75$, the source blending factor becomes: 0.75, and the destination factor: $1 - 0.75 = 0.25$.
- The source fragments have more effect on the final color than the destination fragments.
Combining Opaque and Translucent Polygons

- Using the depth buffer, we can ensure that every **opaque polygon** blocks all objects behind it.
- However, **translucent polygons** must not block objects behind them.
- Solution:
  1. Draw opaque objects with a writeable depth buffer.
  2. Make the depth buffer read-only.
  3. Draw the translucent objects.
- Hence, translucent objects are not drawn if behind an opaque polygon.
- And if they are in front of opaque polygons, their colors blend together.
Using the Depth Buffer in Read-only Mode

The depth mask configuration function

```c
void glDepthMask(GLboolean flag)
```

**GLboolean flag**: makes the depth buffer read-only (GL_FALSE) or writeable (GL_TRUE), the default.
Failing to Use the Depthmask Function

- Depth always Writeable
  - Using readonly depth for translucent objects: 0
  - Draw order: □ (opaque), ■ (opaque), △ (trans), □ (trans).

- Translucents use Read-Only
  - Using readonly depth for translucent objects: 1
For Further Reading

T. Akenine-Möller and Eric Haines
Real-Time Rendering (second edition)
A K Peters, 2002

Dave Shreiner et. al
OpenGL Programming Guide (sixth edition)
Addison-Wesley, 2007

Edward Angel
OpenGL – A Primer (third edition)
Addison-Wesley, 2008
Edward Angel
Addison-Wesley, 2009