CIS 441/541: Introduction to Computer Graphics
Lecture 11: lighting and materials, CW-CCW, 2D texture in action

May 10th, 2013
Hank Childs, University of Oregon
- Tues OH canceled
  - NOT TODAY: Fri OH: 2-4

- Other:
  - Will make signup document this weekend
  - Will grade this weekend
  - Project 2B coming Weds.
    - Get on project 2A!!

- Special colloquium on Thursday
Project #2A (8%), Due Thurs May 16th, 6am

- Goal: OpenGL program that does regular colors and textures
- New VTK-based project2A.cxx
- New CMakeLists.txt (but same as old ones)

Clear what to do?
Outline

- Lighting in OpenGL
- Shading in OpenGL
- Materials in OpenGL
Lighting

- glEnable(GL_LIGHTING);
  - Tells OpenGL you want to have lighting.

- Eight lights
  - Enable and disable individually
    - glEnable(GL_LIGHT0)
    - glDisable(GL_LIGHT7)
  - Set attributes individually
    - glLightfv(GL_LIGHTi, ARGUMENT, VALUES)
glLightfv parameters

- **GL_AMBIENT**

  `params` contains four fixed-point or floating-point values that specify the ambient RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped. The initial ambient light intensity is (0, 0, 0, 1).

- **GL_DIFFUSE**

  `params` contains four fixed-point or floating-point values that specify the diffuse RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped. The initial value for `GL_LIGHT0` is (1, 1, 1, 1). For other lights, the initial value is (0, 0, 0, 0).

- **GL_SPECULAR**

  `params` contains four fixed-point or floating-point values that specify the specular RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped. The initial value for `GL_LIGHT0` is (1, 1, 1, 1). For other lights, the initial value is (0, 0, 0, 0).

- **GL_POSITION**

  `params` contains four fixed-point or floating-point values that specify the position of the light in homogeneous object coordinates. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped.

  The position is transformed by the modelview matrix when `glLight` is called (just as if it were a point), and it is stored in eye coordinates. If the w component of the position is 0, the light is treated as a directional source. Diffuse and specular lighting calculations take the light's direction, but not its actual position, into account, and attenuation is disabled. Otherwise, diffuse and specular lighting calculations are based on the actual location of the light in eye coordinates, and attenuation is enabled. The initial position is (0, 0, 1, 0); thus, the initial light source is directional, parallel to, and in the direction of the -z axis.
For each light source, we can set an RGBA for the diffuse, specular, and ambient components:

```gl
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
GLfloat diffuse0[4] = { 0.7, 0.7, 0.7, 1 };
gllightfv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
... // set ambient, specular, position
glDisable(GL_LIGHT1); // do we need to do this?
...

... // do we need to do this?
```
How do we tell OpenGL about the surface normals?

- **Flat shading:**
  
  ```
  glNormal3f(0, 0.707, -0.707);
  glVertex3f(0, 0, 0);
  glVertex3f(1, 1, 0);
  glVertex3f(1, 0, 0);
  ```

- **Smooth shading:**
  
  ```
  glNormal3f(0, 0.707, -0.707);
  glVertex3f(0, 0, 0);
  glNormal3f(0, 0.707, +0.707);
  glVertex3f(1, 1, 0);
  glVertex3f(1, 0, 0);
  glVertex3f(1, 0, 0);
  ```
Distance and Direction

• The source colors are specified in RGBA
• The position is given in homogeneous coordinates
  - If \( w =1.0 \), we are specifying a finite location
  - If \( w =0.0 \), we are specifying a parallel source with the given direction vector
• The coefficients in the distance terms are by default \( a=1.0 \) (constant terms), \( b=c=0.0 \) (linear and quadratic terms). Change by

\[
\begin{align*}
a &= 0.80; \\
g1Lightf(GL_LIGHT0, GLCONSTANT_ATTENUATION, a);
\end{align*}
\]
Ten parameters (ones you will never use)

**GL_SPOT_DIRECTION**

*params* contains three fixed-point or floating-point values that specify the direction of the light in homogeneous object coordinates. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped.

The spot direction is transformed by the upper 3x3 of the modelview matrix when *glLight* is called, and it is stored in eye coordinates. It is significant only when *GL_SPOT_CUTOFF* is not 180, which it is initially. The initial direction is (0, 0, -1).

**GL_SPOT_EXPONENT**

*params* is a single fixed-point or floating-point value that specifies the intensity distribution of the light. Fixed-point and floating-point values are mapped directly. Only values in the range [0, 128] are accepted.

Effective light intensity is attenuated by the cosine of the angle between the direction of the light and the direction from the light to the vertex being lighted, raised to the power of the spot exponent. Thus, higher spot exponents result in a more focused light source, regardless of the spot cutoff angle (see *GL_SPOT_CUTOFF*, next paragraph). The initial spot exponent is 0, resulting in uniform light distribution.

**GL_SPOT_CUTOFF**

*params* is a single fixed-point or floating-point value that specifies the maximum spread angle of a light source. Fixed-point and floating-point values are mapped directly. Only values in the range [0, 90] and the special value 180 are accepted. If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors. The initial spot cutoff is 180, resulting in uniform light distribution.

**GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION**

*params* is a single fixed-point or floating-point value that specifies one of the three light attenuation factors. Fixed-point and floating-point values are mapped directly. Only nonnegative values are accepted. If the light is positional, rather than directional, its intensity is attenuated by the reciprocal of the sum of the constant factor, the linear factor times the distance between the light and the vertex being lighted, and the quadratic factor times the square of the same distance. The initial attenuation factors are (1, 0, 0), resulting in no attenuation.
Spotlights

• Use `glLightv` to set
  - Direction `GL_SPOT_DIRECTION`
  - Cutoff `GL_SPOT_CUTOFF`
  - Attenuation `GL_SPOT_EXPONENT`
  • Proportional to $\cos^\alpha \phi$
What happens with multiple lights?

```c
glEnable(GL_LIGHT0);
glEnable(GL_LIGHT1);
```

- the effects of these lights are additive.
  - Individual shading factors are added and combined
  - Effect is to make objects brighter and brighter
    - Same as handling of high specular factors for 1E
Global Ambient Light

• Ambient light depends on color of light sources
  - A red light in a white room will cause a red ambient term that disappears when the light is turned off

• OpenGL also allows a global ambient term that is often helpful for testing
  - `glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient)`
    • VTK turns this on by default!
      - Affects lighting of materials colored with `glColor`, but not `glTexCoord1f`!!
Outline

- Lighting in OpenGL
- **Shading in OpenGL**
- Materials in OpenGL
Polygonal Shading

• Shading calculations are done for each vertex
  - Vertex colors become vertex shades
• By default, vertex shades are interpolated across the polygon
  - `glShadeModel(GL_SMOOTH);`
• If we use `glShadeModel(GL_FLAT);` the color at the first vertex will determine the shade of the whole polygon
  - We will come back to this in a few slides
Polygon Normals

- Polygons have a single normal
  - Shades at the vertices as computed by the Phong model can be almost same
  - Identical for a distant viewer (default) or if there is no specular component
- Consider model of sphere
- Different normals at each vertex, want single normal
Smooth Shading

- We can set a new normal at each vertex
- Easy for sphere model
  - If centered at origin $\mathbf{n} = \mathbf{p}$
- Now smooth shading works
- Note *silhouette edge*
Mesh Shading

- The previous example is not general because we knew the normal at each vertex analytically.
- For polygonal models, Gouraud proposed we use the average of the normals around a mesh vertex:

\[ \mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|} \]
glShadeModel — select flat or smooth shading

C Specification

```c
void glShadeModel(GLenum mode);
```

Parameters

**mode**

Specifies a symbolic value representing a shading technique. Accepted values are GL_FLAT and GL_SMOOTH. The initial value is GL_SMOOTH.

Description

GL primitives can have either flat or smooth shading. Smooth shading, the default, causes the computed colors of vertices to be interpolated as the primitive is rasterized, typically assigning different colors to each resulting pixel fragment. Flat shading selects the computed color of just one vertex and assigns it to all the pixel fragments generated by rasterizing a single primitive. In either case, the computed color of a vertex is the result of lighting if lighting is enabled, or it is the current color at the time the vertex was specified if lighting is disabled.

Flat and smooth shading are indistinguishable for points. Starting when `glBegin` is issued and counting vertices and primitives from 1, the GL gives each flat-shaded line segment i the computed color of vertex i + 1, its second vertex. Counting similarly from 1, the GL gives each flat-shaded polygon the computed color of the vertex listed in the following table. This is the last vertex to specify the polygon in all cases except single polygons, where the first vertex specifies the flat-shaded color.
GLShadeModel

GLShadeModel affects normals and colors
glMaterial: coarser controls for color

- You specify how much light is reflected for the material type.

- Command:

  \texttt{glMaterialfv(FACE\_TYPE, PARAMETER, VALUE(S))}

- FACE\_TYPE =
  - GL\_FRONT\_AND\_BACK
  - GL\_FRONT
  - GL\_BACK
glMaterialfv Parameters

**GL_AMBIENT**

`params` contains four fixed-point or floating-point values that specify the ambient RGBA reflectance of the material. The values are not clamped. The initial ambient reflectance is (0.2, 0.2, 0.2, 1.0).

**GL_DIFFUSE**

`params` contains four fixed-point or floating-point values that specify the diffuse RGBA reflectance of the material. The values are not clamped. The initial diffuse reflectance is (0.8, 0.8, 0.8, 1.0).

**GL_SPECULAR**

`params` contains four fixed-point or floating-point values that specify the specular RGBA reflectance of the material. The values are not clamped. The initial specular reflectance is (0, 0, 0, 1).
GL_EMISSION

_params contains four fixed-point or floating-point values that specify the RGBA emitted light intensity of the material. The values are not clamped. The initial emission intensity is (0, 0, 0, 1).

GL_SHININESS

_params is a single fixed-point or floating-point value that specifies the RGBA specular exponent of the material. Only values in the range [0, 128] are accepted. The initial specular exponent is 0.

GL_AMBIENT_AND_DIFFUSE

Equivalent to calling glMaterial twice with the same parameter values, once with GL_AMBIENT and once with GL_DIFFUSE.
OpenGL: very complex model for lighting and colors

- `glMaterial` not used
- `glColor` is no-op

http://www.sjbaker.org/steve/omniv/opengl_lighting.html
Outline

- Lighting in OpenGL
- Shading in OpenGL
- Materials in OpenGL
- Clockwise VS Counter-clockwise
- Quad + textures example
What defines the “front” face?
glFrontFace

**Name**
glFrontFace — define front- and back-facing polygons

**C Specification**

```c
void glFrontFace(GLenum mode);
```

**Python Specification**

```python
glFrontFace(mode) → None
```

**Parameters**

`mode`

Specifies the orientation of front-facing polygons. `GL_CW` and `GL_CCW` are accepted. The initial value is `GL_CCW`.

**Description**

In a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible. Eliminating these invisible polygons has the obvious benefit of speeding up the rendering of the image. To enable and disable elimination of back-facing polygons, call `glEnable` and `glDisable` with argument `GL_CULL_FACE`.

The projection of a polygon to window coordinates is said to have clockwise winding if an imaginary object following the path from its first vertex, its second vertex, and so on, to its last vertex, and finally back to its first vertex, moves in a clockwise direction about the interior of the polygon. The polygon’s winding is said to be counterclockwise if the imaginary object following the same path moves in a counterclockwise direction about the interior of the polygon. `glFrontFace` specifies whether polygons with clockwise winding in window coordinates, or counterclockwise winding in window coordinates, are taken to be front-facing. Passing `GL_CCW` to `mode` selects counterclockwise polygons as front-facing; `GL_CW` selects clockwise polygons as front-facing. By default, counterclockwise polygons are taken to be front-facing.
glCullFace — specify whether front- or back-facing facets can be culled

C Specification

```c
void glCullFace(GLenum mode);
```

Parameters

`mode`

Specifies whether front- or back-facing facets are candidates for culling. Symbolic constants `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK` are accepted. The initial value is `GL_BACK`.

Description

`glCullFace` specifies whether front- or back-facing facets are culled (as specified by `mode`) when facet culling is enabled. Facet culling is initially disabled. To enable and disable facet culling, call the `glEnable` and `glDisable` commands with the argument `GL_CULL_FACE`. Facets include triangles, quadrilaterals, polygons, and rectangles.

`glFrontFace` specifies which of the clockwise and counterclockwise facets are front-facing and back-facing. See `glFrontFace`. 
Something you should have heard of:
Left- and right-handed coordinates

OpenGL: right-handed
DirectX: left-handed
Outline

- Lighting in OpenGL
- Shading in OpenGL
- Materials in OpenGL
- Clockwise VS Counter-clockwise
- Quad + textures example
Live coding example

- Read in 6 textures
- Place each texture on face of a cube
virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
{
    RemoveVTKOpenGLStateSideEffects();
    SetupLight();
    glBegin(GL_QUADS);
    glVertex3f(-10, -10, -10);
    glVertex3f(-10, 10, -10);
    glVertex3f(10, -10, -10);
    glVertex3f(10, 10, -10);
    glEnd();
}
Specifying quads

```cpp
virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
{
    RemoveVTKOpenGLStateSideEffects();
    SetupLight();
    glBegin(GL_QUADS);
    glVertex3f(-10, -10, -10);
    glVertex3f(-10, 10, -10);
    glVertex3f(10, 10, -10);
    glVertex3f(10, -10, -10);
    glEnd();
}
```
What do we expect the output to be?