CMSC 498M: Chapter 2c
Overview of OpenGL - continued

Reading:
- See the *OpenGL Programming Guide*, (also known as "The Red Book") by the OpenGL Arch. Rev. Board, Shreiner, Woo, Neider, and Davis.

Overview:
- Lighting and shading in OpenGL
- Texture mapping in OpenGL
- Pixel Buffers and Operations

Lighting in OpenGL
Illumination

Illumination Models:
- Light is a very complex physical phenomenon.
- Most illumination models in graphics are based on simple geometric optics, as opposed to more complex (but realistic) wave optics.

Local Illumination Models: (OpenGL does this)
- Point light sources and direct interactions with light.
- No shadows, no indirect reflection
- Easy to implement and very efficient.

Shading:
- Involves determining the intensity of illumination (and its color) incident at a surface point.
- Can be computationally intensive, so it is common to compute it accurately at a few points (e.g., vertices) and interpolate in between.

Light/Surface Interaction

Light Reflection:
Ambient: A background glow that illuminates all objects, irrespective of light source location.
Diffuse: A uniform scattering of light, characterized by matte (non-shiny) objects, like cloth or foam rubber.
Specular: Shiny (metallic-like) reflection, like a polished wood table.
Reflective (Pure): No light scattering, like a mirror.
Transparent/Translucent: Light passes through material.
Lighting in OpenGL

- Ambient, diffuse, specular illuminations are supported.
- Attenuation is supported: illumination grows dimmer as the distance from the light source increases.
- Users define light sources: position, type, color.
- Front and back sides of polygons may be given different colors.

Lighting in OpenGL

There is a bewildering number of options and parameters that need to be set up in OpenGL in order to use lighting:

- **Lighting/Shading model**: Global parameters that affect how illumination and shading are computed.
- **Light properties**: Options that define the location, colors, and intensities of the lights.
- **Object material properties**: The color of the object and degree of ambient, diffuse, specular reflection.
- **Enabling**: Lighting can be enabled or disabled.

```c
glEnable ( GL_LIGHTING );
glDisable ( GL_LIGHTING );
```
Lights

Lights and Lighting:
- At least 8 light sources: GL_LIGHT0, ..., GL_LIGHT7.
- `glLight*( )`: used to define individual light properties.
- `glLightModel*( )`: used to define global lighting properties.
- To determine the maximum number of lights supported in your implementation use:
  
  ```
  glGetIntegerv ( GL_MAX_LIGHTS, GLint* num_lights )
  ```
- You need to enable (turn on) each light that you plan to use.
  ```
  glEnable ( GL_LIGHT0 );
  glEnable ( GL_LIGHT1 ); ...
  ```

\[\text{glLight}*( )\]

For scalar-valued parameters:

\[\text{glLight}if( )\]

For vector-valued parameters:

\[\text{glLight}ifv( )\]

where:
- (light) can be: GL_LIGHT0, ..., GL_LIGHT7.
- (pname) can be:
  - GL_POSITION: Light position.
  - GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR: Light colors. (RGBA vector)
  - GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION: Parameters for attenuation. (scalar)
Light Position

Lights at infinity:
- The light position is given as a homogenous \([x, y, z, w]\) vector. When \(w = 0\), this defines a light source at infinity.
- Provides additional efficiency, since light vector is the same for all points in the scene.

When to set light position:
- The \([x, y, z, w]\) vector is given in world coordinates. OpenGL needs to convert these into view-frame coordinates. This is done by multiplying this vector times the Modelview transformation.
- But, with each redraw cycle, the viewer typically moves and we alter the Modelview transformation (by calling \texttt{gluLookAt()}).
- The upshot is that light positions need to be set after issuing the \texttt{gluLookAt()} call.
- Other lighting settings can be done only once.

Sample Lighting Setup

```c
void setUpMyLighting ( ) {
    // light intensity and location
    GLfloat ambientIntensity[4] = { 0.9, 0.0, 0.0, 1.0 }; // red
    GLfloat diffSpecIntensity[4] = { 1.2, 1.2, 1.2, 1.0 }; // white
    GLfloat position[4] = { 2.0, 4.0, 5.0, 1.0 }; 
    // global lighting options
    glShadeModel ( GL_SMOOTH ); // (or GL_FLAT)
    glEnable ( GL_LIGHTING ); // enable lighting
    glEnable ( GL_LIGHT0 ); // enable light 0
    // set up light 0 properties
    glLightfv ( GL_LIGHT0, GL_AMBIENT, ambientIntensity );
    glLightfv ( GL_LIGHT0, GL_DIFFUSE, diffSpecIntensity );
    glLightfv ( GL_LIGHT0, GL_SPECULAR, diffSpecIntensity );
    glLightfv ( GL_LIGHT0, GL_POSITION, position );
}
```
Drawing Objects with Lighting

`glMaterial*()`: with lighting use `glMaterial*()` to specify colors:
- Object colors under illumination are computed as a component-wise multiplication of the light colors and material colors.
- Material properties can be specified differently for ambient, diffuse, and specular reflection.
- In addition to this emission (glowing) can be defined:
  - Lights do not influence emission.
  - Emissive objects do not illuminate other objects.

`glNormal*()`:
- Used to specify vertex surface normals for shading.

---

`glMaterial*()`

For scalar-valued parameters:

```
float glMateriali(GLenum face, GLenum pname, float param)
```

For vector-valued parameters:

```
void glMaterialfv(GLenum face, GLenum pname, const float param*)
```

where:

- `(face)` can be:
  - `GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`:
    Indicates which side of the polygon is being colored. (Recall that front face is the side from which vertices are listed in CCW order.)

- `(pname)` can be:
  - `GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION`:
    Material colors (RGBA vectors). (You can set both ambient and diffuse at once using `GL_AMBIENT_AND_DIFFUSE`.)
  - `GL_SHININESS`:
    Specular illumination exponent (called $\alpha$ above).
Sample Drawing with Lighting

```c
void doMyDrawing() {
    GLfloat red[4] = {1.0, 0.0, 0.0, 1.0}; // RGBA object color (red)
    // set material color
    glMaterialfv(GL_FRONT_AND_BACK,
                GL_AMBIENT_AND_DIFFUSE, red);
    glBegin(GL_POLYGON); // draw polygon
        glNormal3f(...); glVertex3f(...);
        glNormal3f(...); glVertex3f(...);
        glNormal3f(...); glVertex3f(...);
    glEnd();
}
```

In flat shading only one normal is needed.

---

**glColorMaterial***()

**glColorMaterial**: Allows a simpler but more limited way to specify material properties using glColor***().

To use glColorMaterial, it must be enabled (and can be disabled):
```
    glEnable(GL_COLOR_MATERIAL);
    ...  
    glDisable(GL_COLOR_MATERIAL);
```

**Example**:
```
    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_DIFFUSE);
    glColor3f(0.2, 0.5, 0.8); // changes the diffuse material color
    (Draw objects here)
    glColorMaterial(GL_FRONT, GL_SPECULAR);
    glColor3f(0.9, 0.0, 0.2); // changes the specular material color
    (Draw objects here)
    glDisable(GL_COLOR_MATERIAL);
```
Texture Maping in OpenGL

Texture and Surface Detail

We have seen how to provide color to objects using:

- **Solid colors**: through rasterization.
- **Lighting and shading**: through various lighting and shading models.

Next we consider how to add realism through **surface detail**.

Examples:

- **Natural surfaces**: stone, wood, gravel, grass.
- **Printing and painting**: printed labels, billboards, newspapers.
- **Clothing and fabric**: woven and printed patterns, upholstery.
Image Texturing

Texture Mapping Process

Texture mapping function: Maps from texture to object space. Inverse mapping function: Maps the other way. This is actually what we need in texture mapping—which texel corresponds to a given surface point.
Texturing in OpenGL: Basic Steps

One-Time Initialization: (After window is created.)
- Create and specify a texture object:
  - Create a new texture object. (OpenGL provides identifier.)
  - Provide the associated texture image to OpenGL.

For each Redrawing:
- Enable texture mapping.
- Draw the textured polygons:
  - Identify which texture is to be used.
  - Specify texture coordinates with vertices.
- Disable texture mapping:
  - when returning to normal drawing mode.

Creating Texture Object(s)

glGenTextures ( GLsizei n, GLuint* textureIDs );
- Returns n currently unused texture IDs in (textureIDs).
- Each texture ID is an integer greater than 0.

glBindTexture ( GLenum target, GLuint textureID );
where (target) is GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D.
- if (textureID) is being used for the first time a new texture object is created and assigned the ID = (textureID). This is now the active texture.
- if (textureID) has been used before, the texture object with ID = (textureID) becomes active.
Specifying a 2-d Texture Object

```
glTexImage2D ( GLenum target, GLint level,  
        GLint internalFormat, GLsizei width, GLsizei height,  
        GLint border, GLenum format, GLenum type,  
        const GLvoid* texels );
```

Example:
```
glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGBA, 128, 128, 0,  
        GL_RGBA, GL_UNSIGNED_BYTE, myImage);
```

• \langle format\rangle and \langle type\rangle are used to specify the way in which the texels are stored in your image array.
• \langle internalFormat\rangle specifies how OpenGL should store the data internally.
• \langle width\rangle and \langle height\rangle give the image size.
• \langle level\rangle and \langle border\rangle have other uses (see documentation).

Specifying How Texture is Applied

How is the color of the texture pixel combined with the existing pixel?

The main issue to do with whether the texture color is combined with existing object color after lighting (modulation) or is just painted on (replacement).

```
glTexEnv{if} ( GLenum target, GLenum pname, \langle TYPE\rangle \langle value\rangle );
```

where \langle target\rangle is: GL_TEXTURE_ENV.

\langle pname\rangle

GL_TEXTURE_ENV_MODE

\langle value\rangle (some common choices)

GL_MODULATE (mix with lighting) or,
GL_REPLACE (just paint this color).
Specifying how Texture is Applied

There are also parameters that specify how the texture is to be mapped. These involve issues such whether the texture should wrap around (repeat) and how to magnify/shrink it.

```c
glTexParameter{if}( GLenum target, GLenum pname, (TYPE)value );
```

where (target) can be: GL_TEXTURE_1D, GL_TEXTURE_2D, ...

<table>
<thead>
<tr>
<th>(pname)</th>
<th>(value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_TEXTURE_WRAP_S</td>
<td>GL_CLAMP, GL_REPEAT</td>
</tr>
<tr>
<td>GL_TEXTURE_WRAP_T</td>
<td>GL_CLAMP, GL_REPEAT</td>
</tr>
<tr>
<td>GL_TEXTURE_MAG_FILTER</td>
<td>GL_NEAREST, GL_LINEAR</td>
</tr>
<tr>
<td>GL_TEXTURE_MIN_FILTER</td>
<td>GL_NEAREST, GL_LINEAR</td>
</tr>
</tbody>
</table>

Beware: OpenGL’s default value for GL_TEXTURE_MIN_FILTER is very strange. Always specify a value for this parameter.

Enable the Texture and Draw

```c
glEnable ( GL_TEXTURE_2D );
```
- Enable 2-d texturing.

```c
glTexCoord2f ( GL_FLOAT s, GL_FLOAT t );
```
- Specify texture coordinates for the next vertex. (As with glNormal() and glColor(), this applies to all subsequent vertices until changed.)
- Irrespective of the image size, texture coordinates s,t always vary from 0 to 1.

```c
glDisable ( GL_TEXTURE_2D );
```
- Disable 2-d texturing, to return to simple coloring.
Texture Initialization:

```glGenTextures (...) ; // create new texture objects
glBindTexture ( ... ); // make this texture active
glTexParameteri ( ... ); // define texture properties
// ... input texture array from file or generate ...
glTexImage2D ( ... ); // provide the texture to OpenGL```

Displaying a Textured Object:

```glEnable ( GL_TEXTURE_2D ); // enable texturing
glBindTexture ( ... ); // activate the desired texture
gBegin ( GL_TRIANGLES ); // draw the object
gTexCoord2f ( ... ); gNormal3f ( ... ); glVertex3f ( ... );
// ... (draw other vertices in the same way)
gEnd ( );
gDisable ( GL_TEXTURE_2D ); // done```

Minimization Filtering and MIP-mapping

What if one screen-space pixel overlaps many texture pixels? Ideally we should average these pixels, but this takes time. So OpenGL just takes one.

Result: A jagged appearance, aliasing.

MIP-mapping: Precompute averages and build hierarchy based on powers of 2.

To render: OpenGL gets appropriate level in the MIP-map, and use this pixel. This smooths out the jagged lines.
MIP-mapping in OpenGL

How to Set Up a MIP Mapped Texture:

```c
// request MIP-map for min filter
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER,
    GL_NEAREST_MIPMAP_LINEAR);

// present image to OpenGL
glTexImage2D(/* ...as before... */, myImage);

// compute MIP-maps

// (always the same)

// (internal format)

// (image size)

// (image format)

// (image type)

// (the image)
```

Pixel Buffers and Operations
Pixel Buffers and Operations

Pixel Buffers: OpenGL maintains from one to many pixel buffers. These buffers store different types of information and have different functions. We will discuss:

Buffer concepts: bitmaps, pixmaps, depth.
OpenGL Buffers: color-, depth-, accumulation-, and stencil buffers.
Transfer: reading and writing pixel buffers.
Imaging Operations: user operations, bitblt, blend.

Bitmaps and Pixmaps

Pixel: A picture element.
Bitmap: A 2D array of single-bit pixels (0/1 or black/white).
Pixmap: Stack of bitmaps. The number of bits per pixel is called its depth.

To represent full RGB color, it is sufficient to have 24-bit depth, 8 bits each for red, green, and blue.
Bitmaps and Pixmaps

**Bitmap (depth = 1)**

**Pixmap (depth = 8)**

**Pixmap (depth = 24)**

OpenGL Buffers

**Color Buffer**: Stores image color information.
- **RGB**: Red, green, blue
- **RGBA** or **RGBA**: Alpha-channel used for blending operations, such as transparency.

**Depth Buffer**: Stores distance to object pixel.
- Used for hidden surface removal - the closest pixel survives.
- Also called the **Z-buffer** (z-coordinate stores distance).

**Accumulation Buffer**: Used for composing and blending images.
- Useful for achieving affects such as **motion blur**.

**Stencil Buffer**: 
- Useful for masking operations.
Buffer Creation/Specification in GLUT

void glutInitDisplayMode ( unsigned int mode ); where mode is the bitwise OR of GLUT display mode bit masks:

- **GLUT_RGBA**: Select an RGBA (direct) color. (Default)
- **GLUT_RGB**: (Same as GLUT_RGBA).
- **GLUT_INDEX**: Select indexed color.
- **GLUT_SINGLE**: Use single buffering. (Default)
- **GLUT_DOUBLE**: Use double buffering.
- **GLUT_ACCUM**: Allocate space for accumulation buffer.
- **GLUT_ALPHA**: Allocate space for color blending.
- **GLUT_DEPTH**: Allocate space for depth buffer.
- **GLUT_STENCIL**: Allocate space for stencil buffer.

Example:

```c
    glutInitDisplayMode ( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
```

Reading and Writing Buffers

- **glReadPixels** ( x, y, width, height, format, type, *pixels )
- **glRasterPos2i** ( x, y )
- **glDrawPixels** ( width, height, format, type, *pixels )
- **glCopyPixels** ( x, y, width, height, format, type, buffer )

where:

- **format**:
  - GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA,
  - GL_COLOR_INDEX, GL_DEPTH_COMPONENT, ...
- **type**:
  - GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, GL_FLOAT, ...
  - GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_SHORT_5_6_5, ...
- **buffer**:
  - GL_COLOR, GL_DEPTH, GL_STENCIL
Summary

Topics Covered:
- Lighting and shading in OpenGL
- Texture mapping in OpenGL
- Pixel Buffers and Operations

Please see OpenGL documentation and tutorials on web pages (like NeHe) for further information.