Lecture 7, Visual realism
Part 1: Lighting and rendering

Yong-Jin Liu

liuyongjin@tsinghua.edu.cn

Material by S.M. Lea (UNC)
Visual Realism Requirements

• Light Sources
• Materials (e.g., plastic, metal)
• Shading Models
• Depth Buffer Hidden Surface Removal
• Textures
• Reflections
• Shadows
Rendering Objects

• We know how to model mesh objects, manipulate a jib camera, view objects, and make pictures.
• Now we want to make these objects look visually interesting, realistic, or both.
• We want to develop methods of rendering a picture of the objects of interest: computing how each pixel of a picture should look.
Much of rendering is based on different shading models, which describe how light from light sources interacts with objects in a scene.

- It is impractical to simulate all of the physical principles of light scattering and reflection.
- A number of approximate models have been invented that do a good job and produce various levels of realism.
Rendering

• Rendering: deciding how a pixel should look
• Example: compare wireframe (left) to wireframe with hidden surface removal (right)
Rendering (2)

- Example: Compare mesh objects drawn using wire-frame, flat shading, smooth (Gouraud) shading
Rendering (3)

- Compare to images with specular highlights, shadows, textures
Shading Models: Introduction

• Assume to start with that light has no color, only brightness: \( R = G = B \)
• Assume we also have a point source of light (sun or lamp) and general ambient light, which doesn't come directly from a source, but through windows or scattered by the air.
  – Ambient light comes equally from all directions.
  – Point-source light comes from a single point.
Shading Models: Introduction (2)

- When light hits an object, some light is absorbed (and turns into heat), some is reflected, and some may penetrate the interior (e.g., of a clear glass object).
- If all the light is absorbed, the object appears black and is called a blackbody.
- If all the light is transmitted, the object is visible only through refraction (Ch. 12.).
Shading Models: Introduction (3)

• When light is reflected from an object, some of the reflected light reaches our eyes, and we see the object.
  – **Diffuse** reflection: some of the light slightly penetrates the surface and is re-radiated uniformly in all directions. The light takes on some fraction of the color of the surface.
  – **Specular** reflection: more mirror-like. Light is reflected directly from the object’s outer surface, giving rise to highlights of approximately the same color as the source. The surface looks shiny.
Shading Models: Introduction (4)

• In the simplest model, specular reflected light has the same color as the incident light. This tends to make the material look like plastic.

• In a more complex model, the color of the specular light varies over the highlight, providing a better approximation to the shininess of metal surfaces.
Most surfaces produce some combination of diffuse and specular reflection, depending on surface characteristics such as roughness and type of material.

The total light reflected from the surface in a certain direction is the sum of the diffuse component and the specular component.

- For each surface point of interest, we compute the size of each component that reaches the eye.
Reflected Light Model

• Finding Reflected Light: a model
  – Model is not completely physically correct, but it provides fast and relatively good results on the screen.
  – Intensity of a light is related to its brightness. We will use $I_s$ for intensity, where $s$ is R or G or B.
Calculating Reflected Light

- To compute reflected light at point P, we need 3 vectors: normal $\mathbf{m}$ to the surface at P and vectors $\mathbf{s}$ from P to the source and $\mathbf{v}$ from P to the eye. We use world coordinates.
Calculating Reflected Light (2)

• Each face of a mesh object has an inside and an outside.
• Normally the eye sees only the outside (front, in Open-GL), and we calculate only light reflected from the outside.
Calculating Reflected Light (3)

• If the eye can see inside, we must also compute reflections from the inside (back, in OpenGL).
  - If $\mathbf{v} \cdot \mathbf{m} > 0$, the eye can see the face and lighting must be calculated.
  - `glLightModeli(GL_LIGHT_MODEL_TWO_SIDES, GL_TRUE)` calculates lighting for both front and back faces - in case you have open boxes, for example.
Calculating Diffuse Light

• Diffuse scattering is assumed to be independent of the direction from the point, $P$, to the location of the viewer’s eye.

• Because the scattering is uniform in all directions, the orientation of the facet $F$ relative to the eye is not significant, and $I_d$ is independent of the angle between $\mathbf{m}$ and $\mathbf{v}$ (unless $\mathbf{v} \cdot \mathbf{m} < 0$, making $I_d = 0$.)

• The amount of light that illuminates the facet does depend on the orientation of the facet relative to the point source: the amount of light is proportional to the area of the facet that it sees: the area subtended by a facet.
Calculating Diffuse Light (2)

• The intensity depends on the projection of the face perpendicular to $\mathbf{s}$ (Lambert's law). Left: $I_d$; right: $I_d \cos \theta$
Calculating Diffuse Light (3)

- For $\theta$ near $0^\circ$, brightness varies only slightly with angle, because the cosine changes slowly there. As $\theta$ approaches $90^\circ$, the brightness falls rapidly to 0.
- We know $\cos \theta = (\mathbf{s} \cdot \mathbf{m})/(|\mathbf{s}||\mathbf{m}|)$.
- $I_d = I_s \rho_d \frac{(\mathbf{s} \cdot \mathbf{m})}{(|\mathbf{s}||\mathbf{m}|)}$.
  - $I_s$ is the intensity of the source.
  - $\rho_d$ is the diffuse reflection coefficient and depends on the material the object is made of.
Calculating Diffuse Light (4)

• \( \mathbf{s} \cdot \mathbf{m} < 0 \) implies \( I_d = 0 \).
• So to take all cases into account, we use
  \[ I_d = I_s \rho_d \max \left[ \frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}| |\mathbf{m}|}, 0 \right]. \]
Example: Spheres Illuminated with Diffuse Light.

- Spheres with reflection coefficients from 0 to 1 by 0.2.
- The source intensity is 1.0. Background intensity is 0.4.
- The sphere is totally black when $\rho_d = 0.0$, and the shadow in its bottom half (where $(\mathbf{s} \cdot \mathbf{m})/(||\mathbf{s}|| ||\mathbf{m}||) < 0$) is also black.
Calculating Diffuse Light (4)

• The light intensity falling on facet $S$ from the point source is known to decrease as the inverse square of the distance between $S$ and the source. We could try to incorporate this in our model.

• Experiments show that using this law yields pictures with exaggerated depth effects.

• Also, we sometimes model light sources as if they lie “at infinity”. Using an inverse square law in such a case would quench the light entirely.

• So we ignore distance effects in calculating light intensity.
Calculating the Specular Component

- Real objects do not scatter light uniformly in all directions; a specular component is added to the shading model.
- Specular reflection causes highlights, which can add significantly to realism of a picture when objects are shiny.
Calculating the Specular Component (2)

• A simple model for specular light was developed by Phong. It is easy to apply.
  – The highlights generated by the Phong model give an object a plastic-like or glass-like appearance.
  – The Phong model is less successful with objects that are supposed to have a shiny metallic surface, although you can roughly approximate them with OpenGL by careful choices of certain color parameters.
Calculating the Specular Component (3)

• Most of the light reflects at equal angles from the (smooth and/or shiny) surface, along direction $\mathbf{r}$, the reflected direction.
Calculating the Specular Component (2)

• In Ch. 4, we found that $\mathbf{r} = -\mathbf{s} + 2 \mathbf{m} \frac{(\mathbf{s} \cdot \mathbf{m})}{(|\mathbf{m}|^2)}$ (mirror reflection direction).

• For surfaces that are not mirrors, the amount of reflected light decreases as the angle $\phi$ between $\mathbf{r}$ and $\mathbf{v}$ (vector from reflection point to eye) increases.

• For a simplified model, we say the intensity decreases as $\cos^f \phi$, where $f$ is chosen experimentally between 1 and 200.
Calculating the Specular Component (3)

- The effect of \( f \): large \( f \) values give concentrated highlights; smaller ones give larger dimmer highlights.
Calculating the Specular Component (4)

- $\cos \varphi = \frac{\mathbf{r} \cdot \mathbf{v}}{|\mathbf{r}| |\mathbf{v}|}$

- $I_{sp} = I_s \rho_s \left(\frac{\mathbf{r} \cdot \mathbf{v}}{|\mathbf{r}| |\mathbf{v}|}\right)^f$.
  - $\rho_s$ is the specular reflection coefficient, which depends on the material.

- If $\mathbf{r} \cdot \mathbf{v} < 0$, there is no reflected specular light.

- $I_{sp} = I_s \rho_s \max\left(\left(\frac{\mathbf{r} \cdot \mathbf{v}}{|\mathbf{r}| |\mathbf{v}|}\right)^f, 0\right)$. 
Speeding up Calculations for Specular Light

• Find the halfway vector \( h = s + v \).

• Then the angle \( \beta \) between \( h \) and \( m \) approximately measures the falloff of intensity. To take care of errors, we use a different \( f \) value, and write

\[
I_{sp} = I_s \rho_s \max[(h \cdot m/(||h||||m||))^f, 0]
\]
Calculating the Specular Component (5)

- From bottom to top, $\rho_s = 0.75, 0.5, 0.25$. From left to right, $f = 3, 6, 9, 25, 200$.
- $\rho_a = 0.1$
- $\rho_d = 0.4$
Ambient Light

- Our desire for a simple reflection model leaves us with far from perfect renderings of a scene.
  - E.g., shadows appear to be unrealistically deep and harsh.
- To soften these shadows, we can add a third light component called *ambient light*. 
The scenes we observe around us always seem to be bathed in some soft non-directional light. This light arrives by multiple reflections from various objects in the surroundings and from light sources that populate the environment, such as light coming through a window, fluorescent lamps, and the like.

We assume a uniform background glow called ambient light exists in the environment. This ambient light source is not situated at any particular place, and it spreads in all directions uniformly.
Calculating Ambient Light

- The source is assigned an intensity, $I_a$.
- Each face in the model is assigned a value for its ambient reflection coefficient, $\rho_a$ (often this is the same as the diffuse reflection coefficient, $\rho_d$), and the term $I_a \rho_a$ is simply added to whatever diffuse and specular light is reaching the eye from each point $P$ on that face.
- $I_a$ and $\rho_a$ are usually arrived at experimentally, by trying various values and seeing what looks best.
Adding Ambient Light to Diffuse Reflection

• The diffuse and ambient sources have intensity 1.0, and $\rho_d = 0.4$. $\rho_a = 0, 0.1, 0.3, 0.5, 0.7$ (left to right).

• Modest ambient light softens shadows; too much ambient light washes out shadows.
Combining Light Contributions and Adding Color

• \[ I = I_a \rho_a + I_s \rho_d \text{ lambert} + I_{sp} \rho_s x \text{ phong}\]

• \[ \text{Lambert} = \max[(s \cdot m)/(||s|| ||m||), 0] \text{ and Phong} = \max[(h \cdot m)/(||h|| ||m||), 0] \]

• To add color, we use 3 separate total intensities like that above, one each for Red, Green, and Blue, which combine to give any desired color of light.

• We say the light sources have three types of color: ambient = \( (I_{ar}, I_{ag}, I_{ab}) \), diffuse = \( (I_{dr}, I_{dg}, I_{db}) \), and specular = \( (I_{spr}, I_{spg}, I_{spb}) \).
Combining Light Contributions and Adding Color (2)

• Generally the diffuse light and the specular light have the same intensity.
• $\rho_s$ is the same for R, G, and B, so specular light is the color of the light source.
• An object’s color (in white light) is specified by 9 coefficients (ambient and diffuse are color of object):
  • ambient reflection coefficients: $\rho_{ar}$, $\rho_{ag}$, and $\rho_{ab}$;
  • diffuse reflection coefficients: $\rho_{dr}$, $\rho_{dg}$, and $\rho_{db}$
  • specular reflection coefficients: $\rho_{sr}$, $\rho_{sg}$, and $\rho_{sb}$
Example

- If the color of a sphere is 30% red, 45% green, and 25% blue, it makes sense to set its ambient and diffuse reflection coefficients to $(0.3K, 0.45K, 0.25K)$ respectively, where $K$ is some scaling value that determines the overall fraction of incident light that is reflected from the sphere.

- Now if it is bathed in white light having equal amounts of red, green, and blue ($I_{sr} = I_{sg} = I_{sb} = I$) the individual diffuse components have intensities $I_r = 0.3 \ K \ I$, $I_g = 0.45 \ K \ I$, $I_b = 0.25 \ K \ I$, so as expected we see a color that is 30% red, 45% green, and 25% blue.
Example (2)

- Suppose a sphere has ambient and diffuse reflection coefficients (0.8, 0.2, 0.1), so it appears mostly red when bathed in white light.
- We illuminate it with a greenish light $I_s = (0.15, 0.7, 0.15)$.
- The reflected light is then given by $(0.12, 0.14, 0.015)$, which is a fairly even mix of red and green, and would appear yellowish.

  - $0.12 = 0.8 \times 0.15$, $0.14 = 0.2 \times 0.7$, $0.015 = 0.1 \times 0.15$
Example

• Because specular light is mirror-like, the color of the specular component is often the same as that of the light source.
  – E.g., the specular highlight seen on a glossy red apple when illuminated by a yellow light is yellow rather than red.
• To create specular highlights for a plastic-like surface, set the specular reflection coefficients $\rho_{sr} = \rho_{sg} = \rho_{sb} = \rho_s$, so that the reflection coefficients are ‘gray’ in nature and do not alter the color of the incident light.
• The designer might choose $\rho_s = 0.5$ for a slightly shiny plastic surface, or $\rho_s = 0.9$ for a highly shiny surface.
Combining Light Contributions and Adding Color (3)

- A list of $\rho$ and $f$ values for various materials for ambient, diffuse, and specular light is given in Fig. 8.17.

- Spheres of different materials (mostly metallic, but one jade at bottom center) are shown at right (Fig. 8.18).
Shading and the Graphics Pipeline

- Shading is applied to a vertex at the point in the pipeline where the projection matrix is applied. We specify a normal and a position for each vertex.
Shading and the Graphics Pipeline (2)

- `glNormal3f (norm[i].x, norm[i].y, norm[i].z)` specifies a normal for each vertex that follows it.
- The modelview matrix transforms both vertices and normals ($\mathbf{m}$), the latter by $\mathbf{M}^{-T}\mathbf{m}$. $\mathbf{M}^{-T}$ is the transpose of the inverse matrix $\mathbf{M}^{-1}$.
- The positions of lights are also transformed.
Shading and the Graphics Pipeline (3)

• Then a color is applied to each vertex, the perspective transformation is applied, and clipping is done.
• Clipping may create new vertices which need to have colors attached, usually by linear interpolation of initial vertex colors.
• If the new point \( a \) is 40% of the way from \( v_0 \) to \( v_1 \), the color associated with \( a \) is a blend of 60% of \((r_0, g_0, b_0)\) and 40% of \((r_1, g_1, b_1)\): 
  \[
  \text{color at point } a = \langle \text{lerp}(r_0, r_1, 0.4), \text{lerp}(g_0, g_1, 0.4), \text{lerp}(b_0, b_1, 0.4) \rangle
  \]
Shading and the Graphics Pipeline (4)

- The vertices are finally passed through the viewport transformation where they are mapped into screen coordinates (along with pseudodepth, which now varies between 0 and 1).
- The quadrilateral is then rendered (with hidden surface removal).
Creating and Using Light Sources in Open-GL

• Light sources are given a number in [0, 7]: GL_LIGHT_0, GL_LIGHT_1, etc.

• Each light has a position specified in homogeneous coordinates using a GLfloat array named, for example, litePos.

• The light is created using glLightfv (GL_LIGHT_0, GL_POSITION, litePos);

• If the position is a vector (4th component = 0), the source is infinitely remote (like the sun).
Point and Vector Light Locations

• The figure shows a local source at \((0, 3, 3, 1)\) and a remote source “located” along vector \((3, 3, 0, 0)\).

• Infinitely remote light sources are often called “directional”. There are computational advantages to using directional light sources, since direction \(s\) in the calculations of diffuse and specular reflections is constant for all vertices in the scene.

• But directional light sources are not always the correct choice: some visual effects are properly achieved only when a light source is close to an object.
Creating and Using Light Sources in OpenGL (2)

• The light color is specified by a 4-component array \([R, G, B, A]\) of \texttt{GLfloat}, named (e.g.) \texttt{amb0}. The A value can be set to 1.0 for now.

• The light color is specified by \texttt{glLightfv (GL\_LIGHT\_0, GL\_AMBIENT, amb0)}; Similar statements specify \texttt{GL\_DIFFUSE} and \texttt{GL\_SPECULAR}. 
Creating and Using Light Sources in OpenGL (3)

• Lights do not work unless you turn them on.
  – In your main program, add the statements
    `glEnable (GL_LIGHTING); glEnable (GL_LIGHT_0);`
  – If you are using other lights, you will need to enable them also.

• To turn off a light, `glDisable (GL_LIGHT_0);`
• To turn them all off, `glDisable (GL_LIGHTING);`
Creating an Entire Light

GLfloat amb0[ ] = {0.2, 0.4, 0.6, 1.0};
    // define some colors
GLfloat diff0[ ] = {0.8, 0.9, 0.5, 1.0};
GLfloat spec0[ ] = {1.0, 0.8, 1.0, 1.0};
gllightfv(GL_LIGHT0, GL_AMBIENT, amb0);
    // attach them to LIGHT0
gllightfv(GL_LIGHT0, GL_DIFFUSE, diff0);
gllightfv(GL_LIGHT0, GL_SPECULAR, spec0);
Creating and Using Light Sources in OpenGL (4)

• Global ambient light is present even if no lights are created. Its default color is \( \{0.2, 0.2, 0.2, 1.0\} \).

• To change this value, create a `GLfloat` array of values `newambient` and use the statement

\[
glLightModelfv (GL_LIGHT_MODEL_AMBIENT, newambient);
\]

• Default light values are \( \{0, 0, 0, 1\} \) for ambient for all lights, \( \{1, 1, 1, 1\} \) for diffuse and specular for `LIGHT_0`, and \( \{0, 0, 0, 1\} \) for all other lights.
Creating and Using Spotlights in Open-GL

• A spotlight emits light only in a cone of directions; there is no light outside the cone. Inside the cone, \( I = I_s (\cos \beta)^\varepsilon \), where \( \cos \beta \) uses the angle between \( d \) and a line from the source to \( P \).
Creating and Using Spotlights in OpenGL (2)

• To create the spotlight, create a `GLfloat` array for \( \mathbf{d} \). Default values are \( \mathbf{d} = \{0, 0, 0, 1\} \), \( \alpha = 180^\circ \), \( \varepsilon = 0 \): a point source.

• Then add the statements
  
  – `glLightf (GL_LIGHT_0, GL_SPOT_CUTOFF, 45.0);` (45.0 is \( \alpha \) in degrees)
  
  – `glLightf (GL_LIGHT_0, GL_SPOT_EXPONENT, 4.0);` (4.0 is \( \varepsilon \))
  
  – `glLightfv (GL_LIGHT_0, GL_SPOT_DIRECTION, \mathbf{d});`
Attenuation of Light with Distance

- OpenGL also allows you to specify how rapidly light diminishes with distance from a source.
- OpenGL attenuates the strength of a positional light source by the following attenuation factor:

\[
\text{atten} = \frac{1}{k_c + k_l D + k_q D^2}
\]

- where \( k_c, k_l, \) and \( k_q \) are coefficients and \( D \) is the distance between the light’s position and the vertex in question.
Attenuation of Light with Distance (2)

• These parameters are controlled by function calls:
• `glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);`
• and similarly for `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`.
• The default values are $k_c = 1$, $k_l = 0$, and $k_q = 0$ (no attenuation).
Changing the OpenGL Light Model

• **The color of global ambient light**: specify its color using:

```c
GLfloat amb[] = {0.2, 0.3, 0.1, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
```

• **Is the viewpoint local or remote?** OpenGL computes specular reflections using the “halfway vector” \( \mathbf{h} = \mathbf{s} + \mathbf{v} \). The true directions \( \mathbf{s} \) and \( \mathbf{v} \) are normally different at each vertex in a mesh. OpenGL uses \( \mathbf{v} = (0, 0, 1) \), along the positive z-axis, to increase rendering speed. To use the true value of \( \mathbf{v} \) for each vertex, execute

```c
glLightModeli(GL_LIGHT_MODEL_LOCALVIEWER, GL_TRUE);
```
Changing the OpenGL Light Model (2)

- **Are both sides of a polygon shaded properly?** Each polygonal face in a model has two sides. When modeling, we tend to think of them as the “inside” and “outside” surfaces. The convention is to list the vertices of a face in counter-clockwise (CCW) order as seen from outside the object.

- **OpenGL has no notion of inside and outside.** It can only distinguish between “front faces” and “back faces”. A face is a **front face** if its vertices are listed in counter-clockwise (CCW) order as seen by the eye.
Changing the OpenGL Light Model

(3)

• For a space-enclosing object (a) all visible faces are front faces; OpenGL draws them properly with the correct shading.

• If a box has a face removed (b), OpenGL does not shade back faces properly. To force OpenGL to shade back faces, use:

  • `glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);`

• OpenGL reverses the normal vectors of any back-face and performs shading properly.

• Replace `GL_TRUE` with `GL_FALSE` (the default) to turn off this facility.
Moving Light Sources in OpenGL

• To move a light source independently of the camera:
  – set its position array,
  – clear the color and depth buffers,
  – set up the modelview matrix to use for everything except the light source and push it
  – move the light source and set its position
  – pop the matrix
  – set up the camera, and draw the objects.
void display()
{
  GLfloat position[ ] = {2, 1, 3, 1}; //initial light position
    // clear color and depth buffers
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glPushMatrix();
  glRotated(...);
  glTranslated(...);     // move the light
  glLightfv(GL_LIGHT0, GL_POSITION, position);
  glPopMatrix();
  gluLookAt(...);        // set the camera position
  <.. draw the object ..>
  glutSwapBuffers(); }

Code: Independent Motion of Light
Code: Light Moves with Camera

GLfloat pos[ ] = {0,0,0,1};
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0, GL_POSITION, position); // light at (0,0,0)
gluLookAt(...); // move light and camera
// draw the object
Materials in OpenGL

• A material is defined by $\rho_a$, $\rho_s$, $\rho_d$, and $f$ for each of the R, G, B components.
  – Fig. 8.17 contains a table of coefficients for various materials.

• To set your own material properties, create a `GLfloat` array of reflectivities $\rho$, one each for R, G, and B.
  – Set the A reflectivity to 1.0.
Materials in OpenGL (2)

• Then call `glMaterialfv (arg1, arg2, arrayname);`
  – `arg1` can be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.
  – `arg2` can be `GL_DIFFUSE`, `GL_AMBIENT`, `GL_SPECULAR`, `GL_AMBIENT_AND_DIFFUSE`, or `GL_EMISSION`.
  – `GL_EMISSION`: A surface may be emissive (glow) like the sun. We give it an intensity $I_e$, as a `GLfloat` array of 4 values for the emissive color. The $I_e$ is added to the other light intensities.
Materials in OpenGL (3)

- OpenGL adds together the contributions of emissive sources, ambient light, and light from sources to find the total light in a scene for each of red, green, and blue.
- The contributions from each light (spot or regular) are calculated separately before adding.
- OpenGL does not allow the total light intensity to be more than 1.0; it is set to 1.0 if the sum exceeds 1.0.
light x y z R G B ! Gets next unused light number
background R G B
globalAmbient R G B
ambient R G B ! ρ values
diffuse R G B ! ρ values
specular R G B ! ρ values: R = G = B usually
specularExponent f
emissive R G B ! Colors

• Remember to enable lighting and your particular light(s) in your main program.
Example

light 3 4 5 .8 .8 .8 ! bright white light at (3, 4, 5)
background 1 1 1 ! white background
globalAmbient .2 .2 .2 ! a dark gray global ambient light
ambient .2 .6 0 ! material coefficients
diffuse .8 .2. 1 ! red material
specular 1 1 1 ! bright specular spots – the color of the source
exponent 20 ! set the Phong exponent
scale 4 4 4 sphere
Shading Using SDL (2)

- The current material properties are loaded into each object’s mtrl field at the time it is created (see the end of `Scene :: getObject()` in the source example `Shape.cpp` of Appendix 3 and online).

- When an object draws itself using its `drawOpenGL()` method, it first passes its material properties to OpenGL (see `Shape::tellMaterialsGL()`), so that at the moment it is actually drawn OpenGL has these properties in its current state.
Shading Models: Calculating the Color for Each Screen Pixel

- For some objects, we want to see the faces (e.g., the barn); for others, we want to see the underlying surface (e.g., the torus).
- If a face should appear as a separate surface (barn), we attach the same normal to all its vertices.
- If a face approximates an underlying surface (torus), we attach the normal to the underlying surface at each vertex.
Shading Models (2)

- The normal vectors determine the shading type.
  - Flat shading emphasizes individual polygons (barn).
  - Smooth (Gouraud or Phong) shading emphasizes the underlying surface (torus).
- For both kinds of shading, the vertices are passed down the graphics pipeline, shading calculations attach a color to each vertex, and ultimately the vertices are converted to screen coordinates and the face is painted pixel by pixel with the appropriate color.
Shading Models: Raster Fill

- For a convex polygon, screen pixels are colored from bottom to top and left to right across the polygon.
- The display actually colors from top to bottom, but we can think of it as working from bottom to top.
Shading Models: Raster Fill (2)

• We assume the polygons are *convex*. Filling only convex polygons can be made highly efficient, since at each scan-line there is a single unbroken run of pixels inside the polygon.

• Most implementations of OpenGL exploit this and always fill convex polygons correctly, but do not guarantee to fill non-convex polygons properly.

• The figure shows an example where the face is a convex quadrilateral.
Example

• The screen coordinates of each vertex are noted. The lowest and highest points on the face are $y_{\text{bott}}$ and $y_{\text{top}}$, respectively.
• The tiler first fills in the row at $y = y_{\text{bott}}$ (in this case a single pixel), then the one at $y_{\text{bott}} + 1$, etc.
• At each scan-line, say $y_s$ in the figure, there is a leftmost pixel, $x_{\text{left}}$, and a rightmost pixel, $x_{\text{right}}$.
• The tiler moves from $x_{\text{left}}$ to $x_{\text{right}}$, placing the desired color in each pixel. So the tiler is implemented as a simple double loop, looking in pseudo code like
Example (2): Tiler Operation

for (int y = ybott; y <= ytop; y++) // for each scan-line
{
    // find xleft and xright
    for (int x = xleft; x <= xright; x++) // for each relevant pixel across this scan-line
    {
        // find the color c for this pixel
        // put c into the pixel at (x, y)
    }
}

• Hidden surface removal is also easily accomplished within this double loop.
Shading Models: Flat Shading

- We must calculate the correct color \( c \) for each pixel from the vertex colors.
- Flat Shading: Use the same color for every pixel in a face - usually the color of the first vertex.
Shading Models: Flat Shading (2)

• Edges appear more pronounced than they would on a real object because of a phenomenon in the eye known as lateral inhibition:
  – A discontinuity in intensity across a boundary causes the eye to manufacture a Mach band (named after the discoverer, Ernst Mach); edges are sharpened.

• GL implements this type of coloring using `glShadeModel(GL_FLAT);`
Shading Models: Flat Shading (3)

• Specular highlights are rendered poorly with flat shading:
  – If there happens to be a large specular component at the representative vertex, that brightness is drawn uniformly over the entire face.
  – If a specular highlight doesn’t fall on the representative point, it is missed entirely.

• Consequently, we usually do not include the specular reflection component in the shading computation.
Shading Models: Smooth Shading

- Smooth (Gouraud) Shading
- Tries to de-emphasize edges between faces.
Shading Models: Smooth Shading (2)

- Smooth shading uses linear interpolation of colors between vertices:
- Color at left edge (or right edge) is linear interpolation between colors at top and bottom of left (right) edge (interpolation in y).
- Top has color c4 and bottom has color c1:
- Left has color \( c_2 = \text{lerp} \left( c_1, c_4, f \right) \), where \( f = \frac{y_{\text{left}} - y_{\text{bottom}}}{y_{\text{top}} - y_{\text{bottom}}} \).
Shading Models: Smooth Shading (3)

- Likewise, the color $c_3$ at the right edge is a linear interpolation of the top and bottom colors of the right edge.
- Color along a scanline is a linear interpolation (in $x$) of colors at left ($c_2$) and right ($c_3$) edges.
- Calculations must be made separately for R, G, and B.
Shading Models: Smooth Shading (4)

- In practice, for faster execution, the incremental difference in the color between 2 adjacent pixels in the line is calculated and used across the line: \( c_{\text{diff}} = \frac{(c_3 - c_2)}{(x_3 - x_2)} \), and \( c(x+1) = c(x) + c_{\text{diff}} \).
- OpenGL implements this coloring with \texttt{glShadeModel(GL\_SMOOTH)};
Shading Models: Smooth Shading (4)

- Why do the edges disappear with this technique?
- When rendering $F$, the colors $c_L$ and $c_R$ are used, and when rendering $F'$ the colors $c_L'$ and $c_R'$ are used.
- But since $c_R = c_L'$ there is no abrupt change in color at the edge along the scanline. $c_L$ and $c_R$ are calculated using normals that are found by averaging the face normals of all the faces that abut the edge.
Shading Models: Smooth Shading (5)

• Because colors are formed by interpolating rather than computing colors at every pixel, Gouraud shading does not render highlights well.

• Consequently, we usually do not include the specular reflection component in the shading computation.
Phong Shading

• Very good, but very time-consuming! Takes 6 to 8 times longer than Gouraud shading.
• Not implemented by OpenGL.
• Interpolates to find the normal vector at each pixel and uses its normalized value to determine color.
Phong Shading (2)

- The Phong interpolation for normal vectors simply substitutes the 3 components of a normal vector for the 3 components of a color in the Gouraud interpolation.
Comparison of Phong and Gouraud Shading

- Gouraud (left) and Phong (right): notice presence of specular reflection in Phong version.
Removing Hidden Surfaces

• Painter's Algorithm: uses depth buffer to remove hidden surfaces.

• Principal limitations:
  – It requires a large amount of memory.
  – It often renders an object that is later obscured by a nearer object (so time spent rendering the first object is wasted).
Removing Hidden Surfaces (2)

- The figure shows a depth buffer associated with the frame buffer. For every pixel \( p[i][j] \) on the display the depth buffer stores a \( b \) bit quantity \( d[i][j] \). The value of \( b \) is usually in the range of 12 to 30 bits.
- During rendering, depth buffer contains pseudodepth of closest object encountered so far at pixel \((i, j)\).
Removing Hidden Surfaces (3)

• As the tiler proceeds pixel by pixel across a scan-line filling the current face, it tests whether the pseudodepth of the current face is less than the depth $d[i][j]$ stored in the depth buffer at that point.

  – If so, the color of the closer surface replaces the color $p[i][j]$ and this smaller pseudodepth replaces the old value in $d[i][j]$. 
Removing Hidden Surfaces (4)

- Faces can be drawn in any order. If a remote face is drawn first, the color of some of the pixels in the face will later be replaced by the colors of a nearer face.
- Note that this algorithm works for objects of any shape including curved surfaces, because it finds the closest surface based on a point-by-point test.
Removing Hidden Surfaces (5)

– Initialize the depth buffer to 1.0 and frame buffer to background color.

– While rendering the scene (drawing the pixels of the face), if the face has $d_1(i, j) > d(i, j)$, do not draw it (it is hidden). Or else draw it and set $d(i, j) = d_1(i, j)$.

– Pseudodepth $d$ is the $3^{rd}$ component of a projected point: $(aP_z + b)/(-P_z)$; $a$ and $b$ are chosen so $0 \leq d \leq 1$. 
Removing Hidden Surfaces (6)

- For speed, pseudodepth $d$ may be calculated by interpolation.
OpenGL and Hidden Surface Removal

- OpenGL supports a depth buffer and uses the algorithm described to do hidden surface removal.
- You must instruct OpenGL to create a depth buffer when it initializes the display mode:
  ```c
  glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB);
  ```
- and enable depth testing with
  ```c
  glEnable(GL_DEPTH_TEST);
  ```
- Then each time a new picture is to be created the depth buffer must be initialized using:
  ```c
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // clear screen
  ```
Adding Texture to Faces

- Makes surfaces look more realistic: e.g., brick texture or wood texture.
- Texture is painted on or wrapped around surfaces.
Adding Texture to Faces (2)

- Texture is a function $\text{tex} (s, t)$ which sets a color or intensity value for each value of $s$ and $t$ between 0 and 1.0.
- The value of the function $\text{tex}$ is between 0 and 1 (dark and light).
Adding Texture to Faces (3)

- Example: image texture (left) and procedural texture (right).
Procedural Textures

• To produce the procedural texture shown, we implement the following function:

```c
float fakeSphere (float s, float t) {
    float r = sqrt((s - 0.5)*(s - 0.5) + (t - 0.5)*(t - 0.5));
    if (r > 0.3) return 1 - r/0.3; else return 0.2; }
```

// 0.2 is the background
• Function varies from white at center to black at edges of sphere.
• Any function that can be computed for each value of s and t can provide a texture.
Image Textures

• Textures are often formed from bitmap representations of images (such as a digitized photo, digital camera image, clip art, or an image previously generated in some program).

• Such a texture consists of a 2D array, say $\text{txtr}[r][c]$, of color values (often called “texels”).
Image Textures (2)

• Image textures usually have integer coordinates. They are transformed into values between 0 and 1 by dividing by the size of the image: \( s = \frac{x}{x_{\text{max}}} \) and \( t = \frac{y}{y_{\text{max}}} \).

• Likewise, the value \( \text{tex} \) of a texture image is transformed to values between 0 and 1 by dividing by the maximum value in the image: \( \text{tex} = \frac{c}{c_{\text{max}}} \).
Image Textures (3)

- Sample code: Given a texture image $\text{txtr } [c][r]$ (where $c$ is column number and $r$ is row number) we calculate

- $\text{Color3 tex (float s, float t) \{return txtr [(int) s*C] [(int) t*R];\}}$, where $C$ and $R$ are the maximum number of columns and rows.
Uses of Textures

• The value $\text{tex} (s, t)$ can be used in a variety of ways:
  – It can be used as the color of the face itself as if the face were glowing;
  – It can be used as the ambient, diffuse, or specular reflection coefficients to modulate the amount of light reflected from the face;
  – It can be used to alter the normal vector to the surface to give the object a bumpy appearance.
Mapping Texel to Screen Pixel

- With a texture function in hand, the next step is to map it properly onto the desired surface, and then to view it with a camera.
- A single texture is mapped onto two different objects: a planar polygon and a cylinder.
Mapping Texel to Screen Pixel (2)

• For each object, there is some transformation, say $T_{tw}$ (for texture to world) that maps texture $(s, t)$ values to points $(x, y, z)$ on the object’s surface.

• The camera takes a snapshot of the scene from some angle, producing the view shown. We call the transformation from points in 3D to points on the screen $T_{ws}$ (from world to screen), so a point $(x, y, z)$ on a surface is seen at pixel location $(sx, sy) = T_{ws}(x, y, z)$.

• The value $(s^*, t^*)$ on the texture finally arrives at pixel $(sx, sy) = T_{ws}(T_{tw}(s^*, t^*))$. 
The rendering process actually goes the other way: for each pixel at \((sx, sy)\) there is a sequence of questions:

- What is the closest surface seen at \((sx, sy)\)? To answer this requires solving the hidden surface removal problem, at least for this pixel.
- To what point \((x, y, z)\) on this surface does \((sx, sy)\) correspond?
- To which texture coordinate pair \((s, t)\) does this point \((x, y, z)\) correspond?
- So we need the inverse transformation, something like \((s, t) = T_{tw}^{-1}(T_{ws}^{-1}(sx, sy))\), that reports \((s, t)\) coordinates given pixel coordinates. This inverse transformation can be hard to obtain or easy to obtain, depending on the surface shapes.
Pasting Texture onto a Flat Surface

• We first examine the most important case: mapping texture onto a flat surface. This is a modeling task.
We need to associate points on the flat surface with points in the texture. We do so by assigning a texture coordinate to each vertex. In OpenGL:

```
gBegin(GL_QUADS); // define a quadrilateral and position texture on it
gTexCoord2f(0.0, 0.0); glVertex3f(1.0, 2.5, 1.5);
gTexCoord2f(0.0, 0.6); glVertex3f(1.0, 3.7, 1.5);
gTexCoord2f(0.8, 0.6); glVertex3f(2.0, 3.7, 1.5);
gTexCoord2f(0.8, 0.0); glVertex3f(2.0, 2.5, 1.5);
gEnd();
```
Pasting Texture onto a Flat Surface

(3)

- The function `glTexCoord2f(s, t)` sets the current texture coordinates to \((s, t)\), and they are attached to subsequently defined vertices.
- Normally each call to `glVertex3f()` is preceded by a call to `glTexCoord2f()`, causing each vertex to get a new pair of texture coordinates.
- Attaching a \(P_i\) to each \(V_i\) is equivalent to prescribing a polygon \(P\) in texture space that has the same number of vertices as \(F\).
Pasting Texture onto a Flat Surface (3)

- Usually $P$ has the same shape as $F$ as well; then the portion of the texture that lies inside $P$ is pasted without distortion onto the whole of $F$.

- When $P$ and $F$ have the same shape the mapping is clearly affine; it is a scaling, possibly accompanied by a rotation and a translation.
Pasting Texture onto a Flat Surface (4)

- The figure shows a common case where the four corners of a texture rectangle are associated with the four corners of a face in the 3D scene. (The texture coordinates \((s, t)\) associated with each corner are noted on the 3D face.)
Pasting Texture onto a Flat Surface (5)

• In this example, the texture is a 640 by 480 pixel bitmap, and it is pasted onto a rectangle with aspect ratio 640/480, so it appears without distortion.

• Note: the texture coordinate values $s$ and $t$ still vary from 0 to 1.
Tiling a Texture

- The figure shows the use of texture coordinates that **tile** the texture, making it repeat.
Tiling a Texture (2)

• We use s and t values that are larger than 1.0.
• An s value of 2.67, for example, causes the renderer to use \( s = 0.67 \).
• The integer part is the current number of repeats of the pattern.
• s and t values are sent down the graphics pipeline along with each vertex of the face.
• If the rectangle is larger than the texture, OpenGL automatically tiles, although you may choose to clamp the texture (stretch it).
Adding Texture Coordinates to Mesh Objects

- Mesh objects have 3 lists: vertex, normal, face. Add a list of texture coordinates.
- The texture coordinate list stores the coordinates \((s_i, t_i)\) to be associated with various vertices. We add an array of elements, each of which has the type

```cpp
class TxtrCoord { public: float s, t; }
```

to hold all of the coordinate pairs of interest for the mesh.
Adding Texture Coordinates to Mesh Objects (2)

- There are several different ways to treat texture for an object. The two most important are:
  1) The mesh object consists of a small number of flat faces, and a different texture is to be applied to each face. Here each face has only a single normal vector but its own list of texture coordinates. So the data associated with each face would be:
    - the number of vertices in the face;
    - the index of the normal vector to the face;
    - a list of indices of the vertices;
    - a list of indices of the texture coordinates;
Adding Texture Coordinates to Mesh Objects (3)

2) The mesh represents a smooth underlying object, and a single texture is to be wrapped around it (or a portion of it). Here each vertex has associated with it a specific normal vector and a particular texture coordinate pair. A single index into the vertex, normal, and texture lists is used for each vertex. The data associated with each face would then be:

- the number of vertices in the face;
- a list of indices of the vertices;
Rendering the Texture

- Like Gouraud shading: done scan line by scan line.
Rendering the Texture (2)

- Complication: calculating s and t by simple linear interpolation will not work; points on a projected surface do not correspond to points on an actual surface and will produce peculiar results.
Rendering the Texture (3)

- Linear vs. correct interpolation example:
Rendering the Texture (4)

- Affine and projective transformations preserve straightness, so line $L_e$ in eye space projects to line $L_s$ in screen space, and similarly the texels we wish to draw on line $L_s$ lie along the line $L_t$ in texture space that maps to $L_e$. 

![Diagram](image)
The question is this: if we move in equal steps across $L_s$ on the screen, how should we step across texels along $L_t$ in texture space?

The figure shows the line $AB$ in 3D being transformed into the line $ab$ in 3D by matrix $M$. Point $A$ maps to $a$, $B$ maps to $b$. 
Rendering the Texture (6)

- Consider the point $R(g)$ that lies fraction $g$ of the way between $A$ and $B$. It maps to some point $r(f)$ that lies fraction $f$ of the way from $a$ to $b$. The fractions $f$ and $g$ are **not** the same.
- As $f$ varies from 0 to 1, how exactly does $g$ vary? That is, how does motion along $ab$ correspond to motion along $AB$?
Rendering the Texture (7)

• We denote the homogeneous coordinate version of \( a \) by \( \alpha \), and name its components \((\alpha_1, \alpha_2, \alpha_3, \alpha_4)\).

• \( a \) is found from \( \alpha \) by perspective division: \( a = (\alpha_1/\alpha_4, \alpha_2/\alpha_4, \alpha_3/\alpha_4) \).

• Since \( M \) maps \( A \) to \( a \) we know \( \alpha = M(A,1)^T \) where \((A, 1)^T \) is the column vector with components \( A_1, A_2, A_3, \) and 1.

• Similarly, \( \beta = M(B,1)^T \) (homogeneous version of \( b \)).
Rendering the Texture (8)

- \( R(g) = \text{lerp}(A, B, g) \)
- \( M \) maps \( \alpha \) to \( A \) and \( \beta \) to \( B \), so \( R(g) \) maps to \( M(\text{lerp}(A, B, g),1)^T = \text{lerp}(\alpha, \beta, g) \).
- \( \text{Lerp}(\alpha, \beta, g) = \{ \text{lerp}(\alpha_1, \beta_1, g), \text{lerp}(\alpha_2, \beta_2, g), \text{lerp}(\alpha_3, \beta_3, g), \text{lerp}(\alpha_4, \beta_4, g) \} \).
Rendering the Texture (9)

- \( R(f) = \text{lerp}(a, b, f). \)

- \( \text{Lerp}(a, b, f) = (\frac{\text{lerp}(\alpha_1, \beta_1, g)}{\text{lerp}(\alpha_4, \beta_4, g)}, \frac{\text{lerp}(\alpha_2, \beta_2, g)}{\text{lerp}(\alpha_4, \beta_4, g)}, \frac{\text{lerp}(\alpha_3, \beta_3, g)}{\text{lerp}(\alpha_4, \beta_4, g)}) \).

- But also \( \text{lerp}(a, b, f) = \{\text{lerp}(a_1/a_4, b_1/b_4, f), \text{lerp}(a_2/a_4, b_2/b_4, f), \text{lerp}(a_3/a_4, b_3/b_4, f)\}. \)

- So (after some algebra)
  \[ g = f / \text{lerp}(b_4/a_4, 1, f). \]
Rendering the Texture (11)

- \(g\) and \(f\) are both fractions, but different fractions.
- The figure shows how \(g\) and \(f\) are related for different values of \(\frac{b_4}{a_4}\).
Rendering the Texture (10)

- We can also find the point $R(g)$ which maps into $r(f)$: Component 1 is shown, and the other two are similar.

$$ R_1 = \frac{\text{lerp}(\frac{A_1}{a_4}, \frac{B_1}{b_4}, f)}{\text{lerp}(\frac{1}{a_4}, \frac{1}{b_4}, f)} $$

- This interpolation is called hyperbolic interpolation.

- For efficiency, every constant term (everything except $f$) is stored rather than recalculated.
Rendering the Texture (12)

- If the matrix transforming the points on the surface is affine, equal steps along line AB do correspond to equal steps along line ab; however, if the matrix is a perspective projection, they do not.
Rendering the Texture Incrementally: the Barn

- The left edge of the face has endpoints \( a \) and \( b \).
- The face extends from \( x_{\text{left}} \) to \( x_{\text{right}} \) across scan-line \( y \). We need to find appropriate texture coordinates \((s_{\text{left}}, t_{\text{left}})\) and \((s_{\text{right}}, t_{\text{right}})\) to attach to \( x_{\text{left}} \) and \( x_{\text{right}} \), respectively, which we can then interpolate across the scan-line.
Example: the Barn (2)

- To find $s_{\text{left}}(y)$, the value of $s_{\text{left}}$ at scan-line $y$:
- Texture coordinate $s_A$ is attached to point $a$, and $s_B$ is attached to point $b$: $s_A$ and $s_B$ have been passed down the pipeline along with $A$ and $B$.
- If scan-line $y$ is fraction $f$ of the way between $y_{\text{bott}}$ and $y_{\text{top}}$ ($f = (y - y_{\text{bott}})/(y_{\text{top}} - y_{\text{bott}})$), then we know the proper texture coordinate to use is

$$
  s_{\text{left}}(y) = \frac{\text{lerp}(\frac{s_A}{a_4}, \frac{s_B}{b_4}, f)}{\text{lerp}(\frac{1}{a_4}, \frac{1}{b_4}, f)}
$$
Example: the Barn (3)

- $T_{\text{left}}$ is found similarly.
- $s_{\text{left}}$ and $t_{\text{left}}$ have the same denominator: a linear interpolation between values $1/a_4$ and $1/b_4$.
- The numerator and denominator terms can be found incrementally for each $y$.
- But to find $s_{\text{left}}$ and $t_{\text{left}}$ we must still perform an explicit division at each value of $y$. 
Example: the Barn (4)

- The pair \((s_{\text{right}}, t_{\text{right}})\) is calculated in a similar fashion. They have denominators that are based on values of \(a_4'\) and \(b_4'\) that arise from the projected points \(a'\) and \(b'\).
- Once \((s_{\text{left}}, t_{\text{left}})\) and \((s_{\text{right}}, t_{\text{right}})\) have been found, the scan-line can be filled.
- For each \(x\) from \(x_{\text{left}}\) to \(x_{\text{right}}\) the values \(s\) and \(t\) are found, again by hyperbolic interpolation.
The pipeline: Various points are labeled with the information available at that point. Each vertex $V$ is associated with a texture pair $(s, t)$ and a vertex normal.
Graphics Pipeline (2)

• The vertex $V$ is transformed by the modelview matrix $M$ (and the normal is multiplied by the inverse transpose of $M$), giving vertex $A = (A_1, A_2, A_3)$ and a normal $n'$ in eye coordinates.

• Shading calculations are done using this normal, producing the color $c = (c_r, c_g, c_b)$.

• The texture coordinates $(s_A, t_A)$ (the same as $(s, t)$) are still attached to $A$. 
Graphics Pipeline (3)

- Clipping is now done. This can cause some vertices to disappear and others to be formed.
  - When a vertex $D$ is created, we must determine its position $(d_1, d_2, d_3, d_4)$ and attach to it the appropriate color and texture point.
  - The clipping algorithm forms the position components $d_i$ by linear interpolation: $d_i = lerp(a_i, b_i, t)$, for $i = 1, \ldots, 4$, for some $t$. $d_4$ is also formed this way.

- Vertex $A$ undergoes the perspective transformation, producing $\alpha = (\alpha_1, \alpha_2, \alpha_3, \alpha_4)$. The texture coordinates and color $c$ are not changed.
Graphics Pipeline (4)

• Linear interpolation is also used to form both the color components and the texture coordinates.
• After clipping, the face still consists of a number of vertices, and to each is attached a color and a texture point.
• For point A the information is stored in the array \((a_1, a_2, a_3, a_4, s_A, t_A, c, 1)\).
The Graphics Pipeline (5)

• Now perspective division is done. Since for hyperbolic interpolation we need terms such as $s_A/a_4$, $1/a_4$, and $t_A/a_4$, we divide every item in the array that we wish to interpolate hyperbolically by $a_4$ to obtain the array $(x, y, z, 1, s_A/a_4, t_A/a_4, c, 1/a_4)$.

  – We could also divide the color components in order to obtain slightly more realistic Gouraud shading.
The Graphics Pipeline (6)

• The first 3 components \((x, y, z) = \left(\frac{a_1}{a_4}, \frac{a_2}{a_4}, \frac{a_3}{a_4}\right)\) are the position of the point in normalized device coordinates. The third component is pseudodepth.

• The first two components are scaled and shifted by the viewport transformation; we shall continue to call the screen coordinate point \((x, y, z)\).

• So the renderer receives the array \((x, y, z, 1, \frac{s_A}{a_4}, \frac{t_A}{a_4}, c, \frac{1}{a_4})\) for each vertex of the face to be rendered.

• It is simple to render texture using hyperbolic interpolation, since the required values \(\frac{s_A}{a_4}\) and \(\frac{1}{a_4}\) are available for each vertex.
Applying Textures

• Glowing Objects
  – The intensity $I$ is replaced by the value of the texture. (In color, the replacement is for each of the R, G, B components.)

\[
I = \text{texture}(s, t)
\]

• In OpenGL, use \texttt{glTexEnvf} (GL\_TEXTURE\_ENV, GL\_TEX\_ENV\_MODE, GL\_REPLACE);
  – \texttt{GL\_REPLACE} and \texttt{GL\_DECAL} are equivalent.
Applying Textures (2)

• Modulating Reflection Coefficient
  – Color of an object is color of its diffuse light component; vary diffuse reflection coefficient.
  
  \[ I = \text{texture} (s, t) \times \left[ I_a \rho_a + I_d \rho_d \times \text{lambert} \right] + I_{sp} \rho_s \times \text{phong}. \]
  
  • The specular component is the color of the light, not the object.

• In OpenGL, use `glTexEnvf`:

  ```
  (GL_TEXTURE_ENV, GL_TEX_ENV_ENV, GL_TEX_ENV_ENV_MODE, GL_MODULATE);
  ```
Example: Rotating Cube with 6 Textures

• The code uses a number of OpenGL functions to establish the six textures and to attach them to the walls of the cube.

• OpenGL uses textures that are stored in pixel maps, or pixmaps for short. We view a pixmap as a simple array of pixel values, each pixel value being a triple of bytes to hold the red, green, and blue color values:

```cpp
class RGB{
    // holds a color triple – each with 256 possible intensities
    public: unsigned char r,g,b; }
```
Example (2)

- The RGBpixmap class stores the number of rows and columns in the pixmap, as well as the address of the first pixel in memory.

```cpp
class RGBpixmap{
  public:
    int nRows, nCols; // dimensions of the pixmap
    RGB* pixel; // array of pixels
    int readBMPFile(char * fname); // read BMP file into this pixmap
    void makeCheckerboard();
    void setTexture(GLuint textureName); }
```
Example (3)

• Here it has only three methods for mapping textures. Other methods and details are discussed in Chapter 9.
• The method `readBMPFile()` reads a (24-bit) BMP file and stores the pixel values in its `pixmap` object; its implementation is available on the book’s companion website.
• Our example OpenGL application will use six textures. To create them we first make an `RGBpixmap` object for each:

```
RGBpixmap pix[6];  // create six (empty) pixmaps
```
• We then load the desired texture image into each one.
• Finally each one is passed to OpenGL to define a texture.
Example (4)

- **Making a procedural texture.**
- We create a checkerboard texture using the method `makeCheckerboard()` and store it in `pix[0]`:
  
  ```c
  pix[0].makeCheckerboard();
  ```
- The method creates a 64 by 64 pixel array, where each pixel is an RGB triple \((c, c, 0)\), where \(c\) jumps back and forth between 0 and 255 every 8 pixels.
- The two colors of the checkerboard are black \((0,0,0)\), and yellow: \((255,255,0)\).
- The pixel map is laid out in memory as one long array of bytes: row by row from **bottom** to **top**, **left** to **right** across a row.
- The function sets the address of the first pixel of the pixmap, which is later passed to `glTexImage2D()` to create the actual texture for OpenGL.
Example (5): Code

```c++
void RGBpixmap:: makeCheckerboard()
// make checkerboard pattern
{
    nRows = nCols = 64;
    numPixel = new RGB[3 * nRows * nCols];
    if (!numPixel) {cout << "out of memory!"; return;}
    long count = 0;
    for(int i = 0; i < nRows; i++)
        for(int j = 0; j < nCols; j++)
            {
                int c = (((i/8) + (j/8)) %2) * 255;
                numPixel[count].r = c; // red
                numPixel[count].g = c; // green
                numPixel[count++].b = 0; // blue
            }
}
```
Example (6)

- Once we have a pixel map, we must bind it to a unique integer (its name) to refer to it in OpenGL.
- We arbitrarily assign the names 2001, 2002, ..., 2006 to our six textures.
  - OpenGL can supply unique names: If we need six unique names we can build an array to hold them: 
    \[
    \text{GLuint name[6];}
    \]
    and then call \text{glGenTextures(6,name)}. OpenGL places six unique integers in 
    \text{name[0],...,name[5]}, and we subsequently refer to the \text{i}th texture using \text{name[i]}.
- The texture is created by making certain calls to OpenGL, which we encapsulate in the method: 
  \text{void RGBpixmap :: setTexture(GLuint textureName)}
Example (7): Code

```c
void RGBpixmap :: setTexture(GLuint textureName)
{
    glBindTexture(GL_TEXTURE_2D, textureName);
    glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MAG_FILTER,GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER,GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
    nCols, nRows, 0, GL_RGB, GL_UNSIGNED_BYTE,
pixel);
}
```
Example (8)

- The call to `glBindTexture()` binds the given name to the texture being formed. When this call is made at a later time, it will make this texture the active texture.

- The calls to `glTexParameteri()` specify that a pixel should be filled with the texel whose coordinates are nearest the center of the pixel, for both enlarged or reduced versions. This is fast but can lead to aliasing effects.

- Finally, the call to `glTexImage2D()` associates the pixmap with this current texture. This call describes the texture as 2D consisting of RGB byte-triples, gives its width, height, and the address in memory (pixel) of the first byte of the bitmap.
Example (9)

- **Making a texture from a stored image.**
- OpenGL offers no support for reading an image file and creating the pixel map in memory.
- The method `readBMPFile()`, available on the book’s companion website, provides a simple way to read a BMP image into a pixmap. For instance,
  ```c
  pix[1].readBMPFile("mandrill.bmp");
  ```
  reads the file mandrill.bmp and creates the pixmap in `pix[1]`.
- Once the pixel map has been created, `pix[1].setTexture()` is used to pass the pixmap to OpenGL to make a texture.
Example (10)

- Texture mapping must also be enabled with `glEnable(GL_TEXTURE_2D)`.
- The call `glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST)` is used to request that OpenGL render the texture properly (using hyperbolic interpolation), so that it appears correctly attached to faces even when a face rotates relative to the viewer in an animation.
Example (11)

- Complete code is in Fig. 8.49. The texture creation, enabling, and hinting is done once, in an initialization routine.
- In `display()` the cube is rotated through angles `xAngle`, and `yAngle`, and the faces are drawn. The appropriate texture must be bound to the face, and the texture coordinates and 3D positions of the face vertices be specified inside a `glBegin()/glEnd()` pair.
- Once the rendering (off screen) of the cube is complete, `glutSwapBuffers()` is called to make the new frame visible.
- Animation is controlled by the callback “idle” function `spinner()`. Whenever there is no user input, `spinner` is called; it alters the rotation angles of the cube slightly, and calls `display()`.
Wrapping Texture on Curved Surfaces

• We want to wrap texture onto a curved surface, such as a beer can or a chess piece.
• We assume as before that the object is modeled by a mesh, so it consists of a large number of small flat faces.
• Each vertex of the mesh has an associated texture coordinate pair \((s_i, t_i)\).
• The main question is finding the proper texture coordinate \((s, t)\) for each vertex of the mesh.
Wrapping Textures on Curved Surfaces

- Easy if the surface is something like a cylinder; hard otherwise.
- Cylinder: $s$ is an angle coordinate, $t$ is a height coordinate.

\[
s = \frac{\theta - \theta_a}{\theta_b - \theta_a}, \quad t = \frac{z - z_a}{z_b - z_a}
\]
Cylinders

• If there are $N$ faces around the cylinder, the $i^{th}$ face has left edge at azimuth $\theta_i = 2\pi i/N$, and its upper left vertex has texture coordinates $(s_i, t_i) = ((2 \pi i/N - \theta_a)/(\theta_b - \theta_a), 1)$.

• Texture coordinates for the other three vertices follow in a similar fashion.
“Shrink-Wrapping” Surfaces of Revolution

- We pretend the surface is a cylinder, but let the texture coordinates move radially and horizontally to the actual surface. The texture may be distorted.
Alternative Methods (3)

- Alternative 1: move texture coordinates along line from centroid of object to point.
- Alternative 2: move texture coordinates along line normal to surface.
Surfaces of Revolution (2)

- The three ways presented to associate texture points with object points can lead to very different results depending on the shape of the object.
Sphere

- We can paste a texture onto a quadrilateral portion of the sphere: both \( s \) and \( t \) are angle coordinates.
- Alternative: use a triangular texture and paste it onto octants.
Sphere (2)

• To map the texture square to the portion lying between azimuth $\theta_a$ to $\theta_b$ and latitude $\varphi_a$ to $\varphi_b$ just map linearly: if vertex $V_i$ lies at $(\theta_i, \varphi_i)$, associate it with texture coordinates $(s_i, t_i) = ((\theta_i - \theta_a)/(\theta_b - \theta_a), (\varphi_i - \varphi_a)/(\varphi_b - \varphi_a))$.

• It is impossible to cover an entire sphere with a flat texture unless you are willing to live with enormous distortions in the texture.
Shrink-wrapping Sphere-like Objects

• Associate texture points as shown below right (one of three methods).
Sphere-like Objects (2)

• The three ways of associating texture points $P_i$ with object vertices $V_i$ are sketched:
  – object-centroid: $P_i$ is on a line from the centroid $C$ through vertex $V_i$; (usually gives best results)
  – object-normal: $P_i$ is the intersection of a ray from $V_i$ in the direction of the face normal;
  – sphere-normal: $V_i$ is the intersection of a ray from $P_i$ in the direction of the normal to the sphere at $P_i$. 
Texturing Interior of Cube

- Vertices on the object can be associated with texture points in the three ways discussed above; the object-centroid and cube-normal are probably the best choices.
Reflection Mapping

- Reflect surroundings in shiny objects
- Chrome mapping: a rough and usually blurry image suggesting the surrounding environment. Texture (left); object (right).
Reflection Mapping (2)

- Environment mapping: recognizable image of environment (cafeteria) reflected in sphere and torus.
Reflection Mapping (3)

• The cafeteria texture is wrapped about a large sphere that surrounds the object, so that the texture coordinates \((s, t)\) correspond to azimuth and latitude about the enclosing sphere.

• We get valuable visual cues from such reflections, particularly when the object moves.

• More realistic reflections can be done using a surrounding cube.
Reflection Mapping (3)

- The six maps can be generated by rendering six images from the point of view of the object (with the object itself removed, of course).
- Alternatively, the textures can be digitized from photos taken by a real camera that looks in the six directions inside an actual room or scene.
Reflection Mapping (4)

• When the object is moving, in chrome or environment mapping, the environment image "flows" over the object.
• Normally, a texture is attached to the object and moves with it.
The Mapping

• What you see at point $P$ on the shiny object is what has arrived at $P$ from the environment in just the right direction to reflect into your eye.

• To find that direction, trace a ray from the eye to the surface, and then find the reflected ray, $\mathbf{r} = \mathbf{u} - 2(\mathbf{u} \cdot \mathbf{m}) \mathbf{m}$. 
The Mapping (2)

- Trace this ray to find where it hits the texture (on the enclosing cube or sphere).
- It is easiest computationally to suppose that the shiny object is centered in, and much smaller than, the enclosing cube or sphere.
- Then the reflected ray emanates approximately from the object’s center, and its direction $r$ can be used directly to index into the texture.
OpenGL Reflection Mapping

- OpenGL provides a tool to perform approximate environment mapping where the texture is wrapped about a large enclosing cube:

```c
glTexGenf(GL_S,GL_TEXTURE_GEN_MODE,GL_CUBE_MAP);
glTexGenf(GL_T,GL_TEXTURE_GEN_MODE,GL_CUBE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```
OpenGL Reflection Mapping (2)

- Now when a vertex \( P \) with its unit normal \( \textbf{m} \) is sent down the pipeline, OpenGL calculates a texture coordinate pair \((s, t)\) suitable for indexing into the texture attached to the surrounding cube.

- This is done for each vertex of the face on the object, and the face is drawn as always using interpolated texture coordinates \((s, t)\) for points in between the vertices.
OpenGL Reflection Mapping (3)

- How does OpenGL compute a suitable coordinate pair \((s, t)\)?
- It first finds (in eye coordinates) the reflected direction \(\mathbf{r}\), where \(\mathbf{u}\) is the unit vector (in eye coordinates) from the eye to the vertex \(V\) on the object, and \(\mathbf{m}\) is the normal at \(V\).
OpenGL Reflection Mapping (3)

- It computes
  \[(s, t) = \left( \frac{1}{2} \left( \frac{r_x}{p} + 1 \right), \frac{1}{2} \left( \frac{r_y}{p} + 1 \right) \right), \quad p = \sqrt{r_x^2 + r_y^2 + (r_z + 1)^2} \]

- We must pre-compute a texture that shows what you would see of the environment in a perfectly reflecting cube from an eye position far removed from the cube.

- This expression maps the part of the environment that lies in the hemisphere behind the eye into a circle in the middle of the texture, and the part of the environment in the hemisphere in front of the eye into an annulus (ring) around this circle.

- This texture must be recomputed if the eye changes position.
Highlight Mapping

• A texture image with a concentrated bright spot can "paint" highlights onto images.
• Reflection mapping paints this highlight onto the surface, making it appear to be an actual light source situated in the environment. The highlight created can be more concentrated and detailed than those created using the Phong specular term with Gouraud shading.
• With reflection mapping the coordinates \((s, t)\) into the texture are formed at each vertex, and then interpolated in between. So if the coordinates indexed by the vertices happen to surround the bright spot, the spot will be properly rendered inside the face.
OpenGL Extensions and Environment Mapping

- A number of extensions can be implemented in OpenGL programs that make environment mapping a simpler task.
- The OpenGL Extension Registry is online at: http://oss.sgi.com/projects/ogl-sample/registry/
Adding Shadows

• Shadows make an image much more realistic.
• Where are the objects in relation to the plane?

- A shadow conveys a lot of information; it’s as if you are getting a second look at the object from the viewpoint of the light source.
Adding Shadows (2)

• We will look at 3 methods for generating shadows.
• The tools we develop will be useful only for generating shadows cast by a point light source onto a flat surface.
• In Chapter 12, ray tracing tools will allow us to show accurate shadow shapes for any light source shining on any surface shape.
Shadows as Texture

- The problem is to compute the shape of the shadow that is cast.
- The shape of the shadow is determined by the projections of each of the faces of the box onto the plane of the floor using the source as the center of projection. The shadow is the union of the projections of the six faces.
Shadows as Texture (2)

• After drawing the plane using ambient, diffuse, and specular light contributions, draw the six projections of the box’s faces on the plane using only ambient light.
  – This will draw the shadow in the right shape and color.

• Finally draw the box. (If the box is near the plane parts of it might obscure portions of the shadow.)
Shadows as Texture

• The projection of a vertex $V$ on the original object is given by $V_1 = S + (V - S) \left( \frac{[\mathbf{n} \cdot (A - S)]}{[\mathbf{n} \cdot (V - S)]} \right)$, where $S$ is the position of the light source, $A$ is a point on the plane, and $\mathbf{n}$ is the normal to the plane.
Building Projected Faces

• To make the new face $F'$ produced by $F$, project each of its vertices onto the plane in question.

• Suppose that the plane passes through point $A$ and has normal vector $n$. Consider projecting vertex $V$, producing point $V'$. Point $V'$ is the point where the ray from the source at $S$ through $V$ hits the plane. As developed in the exercises, this point is:

$$V' = S + (V - S) \frac{n \cdot (A - S)}{n \cdot (V - S)}$$

• The exercises show how this can be written in homogeneous coordinates as $V$ times a matrix, which is useful for rendering engines, like OpenGL, that support convenient matrix multiplication.
Shadows Using a Shadow Buffer

- This method works for non-planar surfaces.
- The general idea is that any point hidden from the light source is in shadow.
- A second depth buffer (the shadow buffer) is used. It contains depth information about the scene from the point of view of the light source.
Shadow Buffer Steps

1). Shadow buffer loading. The shadow buffer is first initialized with 1.0 in each element, the largest pseudodepth possible.

Then, using a camera positioned at the light source, each of the faces in the scene is scan converted, but only the pseudodepth of the point on the face is tested.

Each element of the shadow buffer keeps track of the smallest pseudodepth seen so far.
Shadow Buffer Steps (2)

- Example: Suppose point $P$ is on the ray from the source through shadow buffer “pixel” $d[i][j]$, and that point $B$ on the pyramid is also on this ray. If the pyramid is present, $d[i][j]$ contains the pseudodepth to $B$; if it happens to be absent, $d[i][j]$ contains the pseudodepth to $P$. 
Shadow Buffer Steps (3)

• The shadow buffer calculation is independent of the eye position.
  – In an animation where only the eye moves, the shadow buffer is loaded only once.
  – The shadow buffer must be recalculated; however, whenever the objects move relative to the light source.
Shadow Buffer Steps (4)

• **Render the scene.** Each face in the scene is rendered using the eye camera as usual.

• Suppose the eye camera sees point $P$ through pixel $p[c][r]$. When rendering $p[c][r]$ we must find
  – the pseudodepth $D$ from the source to $P$;
  – the index location $[i][j]$ in the shadow buffer that is to be tested;
  – the value $d[i][j]$ stored in the shadow buffer.

• If $d[i][j]$ is less than $D$, the point $P$ is in shadow, and $p[c][r]$ is set using only ambient light.

• Otherwise $P$ is not in shadow and $p[c][r]$ is set using ambient, diffuse, and specular light.
Shadows Using Radiosity Method (Overview)

• Recall that the inclusion of ambient light in the calculation of the light components that reflect from each face of an object is a “catch-all”. Ambient is a concocted light attribute and does not exist in real life.
  – It attempts to summarize the many beams of light that make multiple reflections off the various surfaces in a scene.

• Radiosity is a model that attempts to improve image realism.
  – It attempts to form an accurate model of the amount of light energy arriving at each surface versus the amount of light leaving that surface.
Radiosity Method (Overview)

- The figure shows an example of radiosity: part a shows the scene using radiosity; part b shows it without radiosity.
- Using radiosity increases dramatically the computational time required to render an image.
Radiosity Method (3)

• Neither scene produces a high level of realism.
• We will see in Ch. 12 that the use of ray tracing raises the level of realism dramatically and permits a number of specific effects (including realistic shadows) to be included.
OpenGL 2.0 and the Shading Language GLSL

- **Extensions** allow changes and improvements to be made to OpenGL.
- Extensions are created and specified by companies who wish to test and implement innovations in OpenGL, such as new sets of algorithms, functions, and access to the latest hardware capabilities.
GLSL (2)

- Programmable Shading was an extension which provided the application programmer with function calls directly to the graphics card.
- In OpenGL 2.0 the programmable shading extension was promoted to the core of OpenGL.
- The addition of these new function calls allowed the application programmer to control shading operations directly, avoiding the fixed pipeline shading functions.
- Some of the most difficult shading approaches are bump-mapping and 3D textures; the OpenGL Shading Language (GLSL) removes much of the burden of accomplishing these from the application programmer.
OpenGL Pipeline with GLSL

- The figure is a simplified version of the OpenGL pipeline with the addition of the GLSL.
- The pipeline begins with a collection of polygons (vertices) as input, followed by a **vertex processor** which runs **vertex shaders**.
OpenGL Pipeline with GLSL (2)

• Vertex shaders are pieces of code which take vertex data as input; the data includes position, color, normals, etc. With a vertex shader an application program can perform tasks such as:
  – vertex position transformations; controlled by the modelview and projection matrices;
  – transforming the normal vectors and normalizing them if appropriate;
  – generating and transforming texture coordinates;
  – applying a light model-per-vertex (ambient, diffuse, and specular), and computing color.
Once the vertices have been transformed into the viewplane, primitives are rasterized and **fragments** are formed.

For instance, a primitive such as a point might cover a single pixel on the screen, or a line might cover five pixels on a screen.

The fragment resulting from the point consists of a window’s coordinate and depth information, as well as other associated attributes including color, texture coordinates, depth, and so forth.
• Rasterization also determines the pixel position of the fragment. The values for each such attribute are found by interpolating between the values encountered at the vertices.
• A line primitive that stretches across five pixels results in five fragments and the attributes of these fragments are found by interpolating between the corresponding values at the end-points of the line.
• Some primitives don’t yield any fragments at all, whereas others generate a large number of fragments.
• The output of the “rasterization” stage is a flow of fragments (the pipeline here is actually a suggestion to how OpenGL operates).
Bump Mapping

- Bump mapping is a highly effective method for adding complex texture to objects, without having to alter their underlying geometry.
- For example, this figure shows a close-up view of an orange with a rough pitted surface. However, the actually geometry of the orange is a smooth sphere and the texture has been painted on it.
- At each point, the texture perturbs the normal vector to the surface and consequently perturbs the normal direction that is so used to calculate each specular highlight.
Bump Mapping (2)

- It is important to note that the geometry of the object is still a simple sphere, whereas the perturbing texture can be an image or other source of randomized texture as we have discussed previously.
- Another example, where the surface appears to have been engraved with the dates, “1914 – 1918”. However, the indentations of the engraving are created by perturbing the normal vector.
Bump Mapping (3)

• How is bump mapping done? Given an initial perturbation map with which we wish to perturb the surface normal of an object, we must calculate its effect pixel by pixel on the actual surface normal.

• The derivation of this normal perturbation requires the repeated use of partial derivatives and cross-products between them.

• The calculations are beyond the scope of this book. Fortunately, certain extensions to OpenGL offer some assistance in bump mapping which save the application programmer from having to use the underlying mathematics (partial derivatives, etc).
Bump Mapping (4)

• There is also an excellent tutorial on this subject by Paul Baker at www.paulsprojects.net.

• The figure shows an example of a torus with and without bump mapping that he develops carefully in the tutorial.
Non-Photorealistic Rendering

• There are situations when realism is not the most desirable attribute of an image.
• Instead one might want to emphasize a particular message in an image or omit details that are expensive to render yet of little interest to the intended audience.
Non-Photorealistic Rendering (2)

• For example, one might want to produce a cartoon-like rendering of an engine as shown in the figure. This image gives the impression that it could have been done by hand with a paintbrush.
Non-Photorealistic Rendering (3)

• In another situation, one might want to generate a highly-detailed and precise technical drawing of some machine or a blueprint without regard to the fineries of shading, shadows, and anti-aliasing.

• This technique has been called pen and ink rendering (or engraving).
Non-Photorealistic Rendering (4)

- Another example of non-photorealistic rendering; note that this image is very suggestive of what it represents even though it clearly does not look like an actual photograph of the object.