CS 432/637
INTERACTIVE COMPUTER GRAPHICS

Display Issues
Week 5

David Breen
Department of Computer Science
Drexel University

Based on material from Ed Angel, University of New Mexico

Objectives

• Consider perceptual issues related to displays
• Introduce chromaticity space
  – Color systems
  – Color transformations
• Standard Color Systems

Perception Review

• Light is the part of the electromagnetic spectrum between ~350-750 nm
• A color \( C(\lambda) \) is a distribution of energies within this range
• The human visual system has three types of cones on the retina, each with its own spectral sensitivities
• Consequently, only three values, the tristimulus values, are “seen” by the brain

Tristimulus Values

• The human visual center has three cones with sensitivity curves \( S_1(\lambda) \), \( S_2(\lambda) \), and \( S_3(\lambda) \)
• For a color \( C(\lambda) \), the cones output the tristimulus values:
  \[
  T_1 = \int S_1(\lambda)C(\lambda)d\lambda \\
  T_2 = \int S_2(\lambda)C(\lambda)d\lambda \\
  T_3 = \int S_3(\lambda)C(\lambda)d\lambda 
  \]

Three Color Theory

• Any two colors with the same tristimulus values are perceived to be identical
• Thus a display (CRT, LCD, film) must only produce the correct tristimulus values to match a color
• Is this possible? Not always
  – Different primaries (different sensitivity curves) in different systems

The Problem

• The sensitivity curves of the human are not the same as those of physical devices
• Human: curves centered in blue, green, and green-yellow
• CRT: RGB
• Print media: CMY or CMYK
• Which colors can we match and, if we cannot match, how close can we come?
Representing Colors

- Consider a color \( C(\lambda) \)
- It generates tristimulus values \( T_1, T_2, T_3 \)
  - Write \( C = (T_1, T_2, T_3) \)
  - Conventionally, we assume \( T_1, T_2, T_3 \geq 0 \) because there is a maximum brightness we can produce and energy is nonnegative
- \( C \) is a point in color solid

Producing Colors

- Consider a device such as a CRT with RGB primaries and sensitivity curves

\[
T_1 = \int R(\lambda) C(\lambda) \, d\lambda \\
T_2 = \int G(\lambda) C(\lambda) \, d\lambda \\
T_3 = \int B(\lambda) C(\lambda) \, d\lambda
\]

Matching

- This \( T_1, T_2, T_3 \) is dependent on the particular device
- If we use another device, we will get different values and these values will not match those of the human cone curves
- Need a way of matching and a way of normalizing

Color Systems

- Various color systems are used
  - Based on real primaries:
    - NTSC RGB
    - UVW
    - CMYK
    - HSV
    - HLS
  - Theoretical
    - XYZ
- Prefer to separate brightness (luminance) from color (chromatic) information
  - Reduce to two dimensions

Producing Other Colors

- However colors producible on one system (its color gamut) is not necessarily producible on any other
- Note that if we produce all the pure spectral colors in the 350-750 nm range, we can produce all others by adding spectral colors
- With real systems (CRT, film), we cannot produce the pure spectral colors
- We can project the color solid of each system into chromaticity space (of some system) to see how close we can get

Color Gamuts

- Spectral colors
- Printer colors
- Producible color on CRT but not on printer
- Unproducible color
- Producible color on both CRT and printer
**Color Systems**

- Most correspond to real primaries
  - National Television Systems Committee (NTSC) RGB matches phosphors in CRTs
  - Film both additive (RGB) and subtractive (CMY) for positive and negative film
  - Print industry CMYK (K = black)
  - K used to produce sharp crisp blacks
  - Example: ink jet printers

**RGB, CMY, CMYK**

- Assuming 1 is max of a primary
  - C = 1 – R
  - M = 1 – G
  - Y = 1 – B
- Convert CMY to CMYK by
  - K = min(C, M, Y)
  - C' = C – K
  - M' = M – K
  - Y' = Y – K

**Color models**

- **RGB model**
  - specify color by red, green, & blue components
  - additive model
  - 8 bits/component = 24 bits total per pixel
  - in the end, all others reduce to RGB for display

- **HSV model**
  - hue, saturation, & value
  - easier for people to think about
  - hue = primary wavelength (i.e., basic color)
  - saturation = measure of how pure light is
    - high is pure, low means it is mixed w/ white/gray
  - value = intensity or brightness (dark vs. light)
  - direct conversion to RGB

- **CMY model**
  - represent in terms of mixtures of pigments
  - gets color from light it absorbs & doesn’t reflect
  - mix Cyan, Magenta, Yellow
    - subtractive primaries
      - cyan = no red, magenta = no green, yellow = no blue
    - used by printers and artists
  - Color “matching” is quite difficult for printed output

**Color Systems**

- Most correspond to real primaries
  - National Television Systems Committee (NTSC) RGB matches phosphors in CRTs
  - Film both additive (RGB) and subtractive (CMY) for positive and negative film
  - Print industry CMYK (K = black)
  - K used to produce sharp crisp blacks
  - Example: ink jet printers
Color Matrix

- Exists a 3 x 3 matrix to convert from representation in one system to representation in another:

\[
\begin{bmatrix}
T'_1 \\
T'_2 \\
T'_3
\end{bmatrix} = M \begin{bmatrix}
T_1 \\
T_2 \\
T_3
\end{bmatrix}
\]

- Example: XYZ to NTSC RGB
  - find in colorimetry references
  - Can take a color in XYZ and find out if it is producible by transforming and then checking if resulting tristimulus values lie in (0,1)

YIQ

- NTSC Transmission Colors
- Here Y is the luminance
  - Arose from need to separate brightness from chromatic information in TV broadcasting

\[
\begin{bmatrix}
Y \\
I \\
Q
\end{bmatrix} = \begin{bmatrix}
0.299 & 0.587 & 0.114 \\
0.596 & -0.275 & -0.321 \\
0.212 & -0.523 & 0.311
\end{bmatrix} \begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

- Note luminance shows high green sensitivity

Gamma

- Intensity vs CRT voltage is nonlinear
  \( I = cV^\gamma \)
- Can use a lookup table to correct
- Human brightness response is logarithmic
  - Equal steps in gray levels are not perceived equally
  - Can use lookup table
- CRTs cannot produce a full black
  - Limits contrast ratio

Shading

Objectives

- Learn to shade objects so their images appear three-dimensional
- Introduce the types of light-material interactions
- Build a simple reflection model---the Phong model--- that can be used with real time graphics hardware

Why we need shading

- Suppose we build a model of a sphere using many polygons and color it with \texttt{glColor}. We get something like

- But we want
Shading

• Why does the image of a real sphere look like

• Light-material interactions cause each point to have a different color or shade

• Need to consider
  – Light sources
  – Material properties
  – Location of viewer
  – Surface orientation

Scattering

• Light strikes A
  – Some scattered
  – Some absorbed

• Some of scattered light strikes B
  – Some scattered
  – Some absorbed

• Some of this scattered light strikes A, and so on

Rendering Equation

\[ I(x, y) = g(x, y) e(x, y) + \int_{\text{surf}} s(x, y, z) I(y, z) dz \]

• The infinite scattering and absorption of light can be described by the rendering equation
  – Cannot be solved in general
  – Ray tracing is a special case for perfectly reflecting surfaces

• Rendering equation is global and includes
  – Shadows
  – Multiple scattering from object to object

Global Effects

Translucent surface

Shadow

Multiple reflection

Local vs Global Rendering

• Correct shading requires a global calculation involving all objects and light sources
  – Incompatible with pipeline model which shades each polygon independently (local rendering)

• However, in computer graphics, especially real time graphics, we are happy if things “look right”
  – Exist many techniques for approximating global effects

Light-Material Interaction

• Light that strikes an object is partially absorbed and partially scattered (reflected)

• The amount reflected determines the color and brightness of the object
  – A surface appears red under white light because the red component of the light is reflected and the rest is absorbed

• The reflected light is scattered in a manner that depends on the smoothness and orientation of the surface
Light Sources

General light sources are difficult to work with because we must integrate light coming from all points on the source.

Simple Light Sources

- Point source
  - Model with position and color
  - Distant source = infinite distance away (parallel)
- Spotlight
  - Restrict light from ideal point source
- Ambient light
  - Same amount of light everywhere in scene
  - Can model contribution of many sources and reflecting surfaces

Surface Types

- The smoother a surface, the more reflected light is concentrated in the direction a perfect mirror would reflect the light.
- A very rough surface scatters light in all directions.

Phong Model

- A simple model that can be computed rapidly
- Has three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To source
  - To viewer
  - Normal
  - Perfect reflector

Ideal Reflector

- Normal is determined by local orientation
- Angle of incidence = angle of reflection
- The three vectors must be coplanar

Lambertian Surface

- Perfectly diffuse reflector
- Light scattered equally in all directions
- Amount of light reflected is proportional to the vertical component of incoming light
  - reflected light \( \approx \cos \theta_i \)
  - \( \cos \theta_i = 1 \cdot n \) if vectors normalized
  - There are also three coefficients, \( k_r, k_g, k_b \) that show how much of each color component is reflected.
Specular Surfaces

- Most surfaces are neither ideal diffusers nor perfectly specular (ideal reflectors).
- Smooth surfaces show specular highlights due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection.

Modeling Specular Reflections

- Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased.

The Shininess Coefficient

- Values of $\alpha$ between 100 and 200 correspond to metals.
- Values between 5 and 10 give surface that look like plastic.
Ambient Light
- Ambient light is the result of multiple interactions between (large) light sources and the objects in the environment.
- Amount and color depend on both the color of the light(s) and the material properties of the object.
- Add \( I_a \) to diffuse and specular terms.

Distance Terms
- The light from a point source that reaches a surface is inversely proportional to the square of the distance between them.
- We can add a factor of the form \( 1/(a + bd + cd^2) \) to the diffuse and specular terms.
- The constant and linear terms soften the effect of the point source.

Light Sources
- In the Phong Model, we add the results from each light source.
- Each light source has separate diffuse, specular, and ambient terms to allow for maximum flexibility even though this form does not have a physical justification.
- Separate red, green, and blue components.
- Hence, 9 coefficients for each point source:
  - \( I_{d_r}, I_{d_g}, I_{d_b}, I_{s_r}, I_{s_g}, I_{s_b}, I_{a_r}, I_{a_g}, I_{a_b} \).
Material Properties

- Material properties match light source properties
  - Nine absorption coefficients
    - $k_{db}$, $k_{dr}$, $k_{sg}$, $k_{sr}$, $k_{sb}$, $k_{ag}$, $k_{ar}$, $k_{ab}$
  - Shininess coefficient $\alpha$

Adding up the Components

For each light source and each color component, the Phong model can be written (without the distance terms) as

$$I = k_{id} I_{id} \cdot n + k_{is} I_{is} (v \cdot r)^{\alpha} + k_{ia} I_{ia}$$

For each color component we add contributions from all sources.

Example

Only differences in these teapots are the parameters in the Phong model.

Shading in OpenGL

Objectives

- Introduce the OpenGL shading functions
- Discuss polygonal shading
  - Flat
  - Smooth
  - Gouraud

Steps in OpenGL shading

1. Enable shading and select model
2. Specify normals
3. Specify material properties
4. Specify lights
Normals

- In OpenGL the normal vector is part of the state
- Set by glNormal*()
  - glNormal3f(x, y, z);
  - glNormal3fv(p);
- Usually we want to set the normal to have unit length so cosine calculations are correct
- Length can be affected by transformations
- Note the scale does not preserve length
  - glEnable(GL_NORMALIZE) allows for autonormalization at a performance penalty

Normal for Triangle

\[
\text{plane: } n \cdot (p_i \cdot p_0) = 0 \\
n = (p_1 \cdot p_0) \times (p_2 \cdot p_0) \\
\text{normalize } n \leftarrow n / |n| \\
P_0
\]

Note that right-hand rule determines outward face

Enabling Shading

- Shading calculations are enabled by
  - glEnable(GL_LIGHTING)
- Once lighting is enabled, glColor() ignored
- Must enable each light source individually
  - glEnable(GL_LIGHTi) i=0,1,.....
- Can choose light model parameters
  - glLightModeli(parameter, GL_TRUE)
    - GL_LIGHT_MODEL_LOCAL_VIEWER use simplifying distant viewer assumption in calculation if turned off
    - GL_LIGHT_MODEL_TWO_SIDED shades both sides of polygons independently
    - GL_LIGHT_MODEL_AMBIENT ambient only shading when no light sources are enabled. Need to supply an ambient light color.

Defining a Point Light Source

- For each light source, we can set an RGB for the diffuse, specular, and ambient parts, and the position

\[
\text{GL float diffuse0}[] = \{1.0, 0.0, 0.0, 1.0\}; \\
\text{GL float ambient0}[] = \{1.0, 0.0, 0.0, 1.0\}; \\
\text{GL float specular0}[] = \{1.0, 0.0, 0.0, 1.0\}; \\
\text{GL float light0_pos}[] = \{1.0, 2.0, 3.0, 1.0\}; \\
\text{glEnable(GL_LIGHTING)}; \\
\text{glEnable(GL_LIGHT0)}; \\
\text{glLightv(GL_LIGHT0, GL_POSITION, light0_pos)}; \\
\text{glLightv(GL_LIGHT0, GL_AMBIENT, ambient0)}; \\
\text{glLightv(GL_LIGHT0, GL_DIFFUSE, diffuse0)}; \\
\text{glLightv(GL_LIGHT0, GL_SPECULAR, specular0)};
\]

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
- Light's position is transformed by the model-view matrix and stored in eye coordinates

Light Distance Drop-off

- Specular & diffuse attenuation function
  - \( 1/(a + bd + cd^2) \)
- Just linear drop-off "looks right"
- The coefficients in the distance terms are by default \( a=1.0 \) (constant terms), \( b=c=0.0 \) (linear and quadratic terms). Change by
  \[
  a = 0.80; \\
  \text{glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, a)}; \\
  b = 0.60; \\
  \text{glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, b)}; \\
  c = 1.20; \\
  \text{glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, c)};
  \]
Spotlights

• Use \texttt{glLightfv} to set
  – Direction \texttt{GL\_SPOT\_DIRECTION}
  – Cutoff \texttt{GL\_SPOT\_CUTOFF}
  – Attenuation \texttt{GL\_SPOT\_EXponent}

  • Proportional to $\cos^\theta \phi$

  \[
  \theta \quad \phi \quad 0
  \]

• By default, \texttt{GL\_SPOT\_CUTOFF} = 180.0
  – light is emitted in all directions

• Restricted to $[0.0, 90.0]$ (unless 180.0)

  \[
  \text{GLfloat} \quad \text{spot\textunderscore direction} = \{-1.0, -1.0, 0.0\}; \quad \text{gl\textunderscore lightfv}(\text{GL\_LIGHT0}, \text{GL\_SPOT\_DIRECTION}, \text{spot\textunderscore direction}); \quad \text{gl\textunderscore lightf}(\text{GL\_LIGHT0}, \text{GL\_SPOT\_CUTOFF}, 45.0); \quad \text{gl\textunderscore lightf}(\text{GL\_LIGHT0}, \text{GL\_SPOT\_EXponent}, 3.0);
  \]

Spotlights

• By default, direction is (0.0, 0.0, -1.0),
  – so light points down the negative z-axis.
  Also, keep in mind that a

• Spotlight's direction is transformed by
  the model\textunderscore view matrix

• Result is stored in eye coordinates

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 allows a global ambient term
  that is often helpful

  \[
  \text{gl\textunderscore lightModelfv}(\text{GL\_LIGHT\_MODEL\_AMBIENT}, \text{global\textunderscore ambient});
  \]

Moving Light Sources

• Light sources are geometric objects whose
  positions or directions are affected by the
  model\textunderscore view matrix

• Position is transformed and set in eye-coords

• Depending on where we place the position
  (direction) setting function, we can
  – Move the light source(s) with the object(s) or view
  – Fix the object(s) and move the light source(s)
  – Fix the light source(s) and move the object(s)
  – Move the light source(s) and object(s)
    independently

Moving Light Sources

• Keeping light stationary in world
  – Define light position after \texttt{LookAt}

• Move light independently
  – Define after \texttt{LookAt}
  – Surround with \texttt{Push} & \texttt{PopMatrix}
  – Define transformations and light position

• Move with camera
  – Define position before \texttt{LookAt}
Light Source Fixed to View

/* in init() */
glViewport(0, 0, (GLsizei) w, (GLsizei) h);
gLoadIdentity();
if (w <= h)
  glOrtho(-1.5, 1.5, -1.5*h/w, 1.5*h/w, -10.0, 10.0);
else
  glOrtho(-1.5*w/h, 1.5*w/h, -1.5, 1.5, -10.0, 10.0);
gLoadIdentity(); /* later in init() */
glMatrixMode(GL_MODELVIEW);
GLfloat light_position[] = { 0.0, 1.0, 0.0, 1.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);

Light Source Fixed in World

void display(void)
{
  GLfloat light_position[] = { 0.0, 0.0, 1.5, 1.0 };
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  gluLookAt(0.0,0.0,5.0, 0.0,0.0,0.0, 0.0,1.0,0.0);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glutSolidTorus(0.275, 0.85, 8, 15);
  glPopMatrix();
  glFlush();
}

Moving Light Source

void display(void) /* light rotates around a torus */
{
  GLfloat light_position[] = { 0.0, 0.0, 1.5, 1.0 };
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  gluLookAt(0.0,0.0,5.0, 0.0,0.0,0.0, 0.0,1.0,0.0);
  glPushMatrix();
  glRotatef(spin, 1.0, 0.0, 0.0);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glPopMatrix();
  glPopMatrix();
  glutSolidTorus(0.275, 0.85, 8, 15);
  glFlush();
}

Moving Light Source & Object

void display(void) /* light rotates around a torus */
{
  GLfloat light_position[] = { 0.0, 0.0, 1.5, 1.0 };
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  gluLookAt(0.0,0.0,5.0, 0.0,0.0,0.0, 0.0,1.0,0.0);
  glPushMatrix();
  glRotatef(spin0, 1.0, 0.0, 0.0);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glPopMatrix();
  glRotatef(spin1, 0.0, 1.0, 0.0);
  glutSolidTorus(0.275, 0.85, 8, 15);
  glPopMatrix();
  glFlush();
}

Material Properties

- Material properties are also part of the OpenGL state and match the terms in the Phong model
- Set by glMaterialv()

  GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};
  GLfloat diffuse[] = {1.0, 0.8, 0.0, 1.0};
  GLfloat specular[] = {1.0, 1.0, 1.0, 1.0};
  GLfloat shine = 100.0
  glMaterial(GL_FRONT, GL_AMBIENT, ambient);
  glMaterial(GL_FRONT, GL_DIFFUSE, diffuse);
  glMaterial(GL_FRONT, GL_SPECULAR, specular);
  glMaterial(GL_FRONT, GL_SHININESS, shine);

Front and Back Faces

- The default is shade only front faces which works correct for convex objects
- If we set two sided lighting, OpenGL will shaded both sides of a surface
- Each side can have its own properties which are set by using GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK in glMaterialf

  back faces not visible
  back faces visible
Emissive Term

- We can simulate a light source in OpenGL by giving a material an emissive component.
- This color is unaffected by any sources or transformations.

```c
GLfloat emission[] = {0.0, 0.3, 0.3, 1.0};
glMaterialf(GL_FRONT, GL_EMISSION, emission);
```

Transparency

- Material properties are specified as RGBA values.
- The A value can be used to make the surface translucent.
- The default is that all surfaces are opaque regardless of A.
- Later we will enable blending and use this feature.

Efficiency

- Because material properties are part of the state, if we change materials for many surfaces, we can affect performance.
- We can make the code cleaner by defining a material structure and setting all materials during initialization.

```c
typedef struct materialStruct {
    GLfloat ambient[4];
    GLfloat diffuse[4];
    GLfloat specular[4];
    GLfloat shininess;
} MaterialStruct;
```
- We can then select a material by a pointer.

Polygonal Shading

- Shading calculations are done for each vertex.
  - Vertex colors become vertex shades.
- By default, vertex colors are interpolated across the polygon:
  ```c
  glShadeModel(GL_SMOOTH);
  ```
- If we use `glShadeModel(GL_FLAT);` the color at the first vertex will determine the color of the whole polygon.

Polygon Normals

- Polygons have a single normal.
  - Shades at the vertices as computed by the Phong model can be almost the same.
  - Identical for a distant viewer (default) or if there is no specular component.
- Consider model of sphere.
- Want different normals at each vertex even though this concept is not quite correct mathematically.

Smooth Shading

- We can set a new normal at each vertex.
- Easy for sphere model.
  - If centered at origin $n = p/|p|$
- 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 normals around a mesh:
  \[
  n = \frac{n_1 + n_2 + n_3 + n_4}{|n_1 + n_2 + n_3 + n_4|}
  \]

Gouraud and Phong Shading

- Gouraud Shading
  - Find average normal at each vertex (vertex normals)
  - Apply Phong model at each vertex
  - Interpolate vertex shades across each polygon
- Phong shading
  - Find vertex normals
  - Interpolate vertex normals across polygon
  - Calculate shade with interpolated normal

Comparison

- If the polygon mesh approximates surfaces with high curvatures, Phong shading may look smooth while Gouraud shading may show edges.
- Gouraud shading may miss specular highlights when polygons are large.
- Phong shading requires much more work than Gouraud shading.
  - Usually not available in real time systems.
- Both need data structures to represent meshes so we can obtain vertex normal.

Calculating Normals

- Create vector structure (for normals) the same size as vertex structure.
- For each face:
  - Calculate unit normal
  - Add to normal structure using vertex index
- Normalize all the normals

\[\text{smf file format}\]

- \text{vertices}
  \[
  \begin{align*}
  v & -1 -1 -1 \\
  v & 1 -1 -1 \\
  v & 1 1 -1 \\
  v & 1 -1 1 \\
  v & -1 1 1 \\
  v & -1 1 1 \\
  v & -1 -1 1 \\
  v & 1 -1 -1 \\
  v & 1 1 -1 \\
  v & 1 -1 1 \\
  v & -1 1 1 \\
  \end{align*}
  \]

- \text{triangles}
  \[
  \begin{align*}
  f & 1 4 2 \\
  f & 5 8 7 \\
  f & 1 2 6 \\
  f & 5 8 7 \\
  f & 3 4 4 \\
  f & 1 5 7 \\
  f & 2 1 3 \\
  f & 2 3 6 \\
  \end{align*}
  \]