CSE 5542 - Real Time Rendering
Week 6,7,8
OpenGL Perspective Matrix

Courtesy: Prof. H-W. Shen
Perspective Transform
\[
\begin{align*}
x_e - 2e &= x_p - \frac{xe}{d} \\
\Rightarrow x_p &= \frac{xe}{d} - 2e
\end{align*}
\]
Some for \( y \):
\[
\begin{align*}
y_e - 2e &= y_p - \frac{ye}{d} \\
\Rightarrow y_p &= \frac{ye}{d} - 2e
\end{align*}
\]

Basic perspective projection matrix:
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \times \begin{bmatrix}
x_e \\
y_e \\
z_e \\
1
\end{bmatrix} = \begin{bmatrix}
x_e' \\
y_e' \\
z_e' \\
1
\end{bmatrix}
\]

Because
\[
\begin{bmatrix}
1 & 0 & 0 & 7 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \times \begin{bmatrix}
x_e \\
y_e \\
z_e \\
2e
\end{bmatrix} = \begin{bmatrix}
x_e \\
y_e \\
z_e \\
2e
\end{bmatrix}
\]
Normalization:
\[
\begin{align*}
x_p &= \frac{xe}{d} - 2e \\
y_p &= \frac{ye}{d} - 2e \\
z_p &= z_e - d
\end{align*}
\]

But we need normalize:
\[
\begin{align*}
[x_{min}, x_{max}] &\rightarrow [-1, 1] \\
[y_{min}, y_{max}] &\rightarrow [-1, 1]
\end{align*}
\]

Visualize this:
\[
\begin{align*}
\overrightarrow{X_{n}} &\rightarrow \overrightarrow{X_{p}} \\
\overrightarrow{y_{n}} &\rightarrow \overrightarrow{y_{p}} \\
\overrightarrow{X_{n}} &\rightarrow \overrightarrow{X_{p}}
\end{align*}
\]

We have:
\[
\frac{X_p - X_{min}}{X_{max} - X_{min}} = \frac{X_n - (-1)}{1 - (-1)} = \frac{X_n + 1}{2}
\]

\[
\begin{align*}
x_n &= \frac{2(X_p - X_{min})}{X_{max} - X_{min}} - 1 \\
&= \frac{2}{X_{max} - X_{min}} \times X_p - \frac{X_{max} + X_{min}}{X_{max} - X_{min}}
\end{align*}
\]

Remember earlier we have:
\[
X_p = \frac{xe}{d} - 2e
\]

So now we have
\[
X_n = \frac{2}{X_{max} - X_{min}} \times \frac{xe}{d} - \frac{X_{max} + X_{min}}{X_{max} - X_{min}}
\]
\[
\begin{bmatrix}
2 & X_{\text{max}} - X_{\text{min}} & \frac{X_{\text{max}} + X_{\text{min}}}{2} \\
0 & \frac{X_{\text{max}} - X_{\text{min}}}{2} & \frac{X_{\text{max}} + X_{\text{min}}}{2} \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix} = M_p
\]

Why? Because

\[
M_p \times \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}
\]

Normalize \( (x', y', z', w') \)

\[
\begin{align*}
x' &= \frac{2x - 1}{x_{\text{max}} - x_{\text{min}}} - x_{\text{max}} + x_{\text{min}} \\
y' &= \frac{2y - 1}{y_{\text{max}} - y_{\text{min}}} - y_{\text{max}} + y_{\text{min}} \\
z' &= \frac{2z - 1}{z_{\text{max}} - z_{\text{min}}} - z_{\text{max}} + z_{\text{min}} \\
w' &= 1
\end{align*}
\]

But we are not done yet.

We need to map \( z' \) range between \([-\text{near}, \text{far}]\) to \([-1, 1]\). This requires change of \( M_p \).

\[
\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = M_p^x \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \quad \text{(copy x from \( M_p \) in \( \text{Eq.} \)}
\]

\[
\begin{bmatrix} 0 \\ 0 \\ \frac{1}{-d} \\ 0 \end{bmatrix} \quad \text{we need to find} \quad A \quad \text{and} \quad B
\]

\[
\begin{bmatrix} x' \\ x'' \\ y'' \\ z'' \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ A \\ B \end{bmatrix} = \begin{bmatrix} xx \\ xx \\ xx + xx \\ xx + xx \end{bmatrix}
\]

\[
xx \text{ means } \frac{\text{\textit{don't care}}}{}
\]

\[
\Rightarrow \quad \frac{A \cdot \text{near} + B}{\text{near}} = -1
\]

\[
A \cdot \text{near} + B = -\text{near}
\]

\[
\Rightarrow \quad -Ad + B \cdot d = -\text{near} \rightarrow \boxed{1}
\]
Also:
\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & 0 & A & B \\
0 & 0 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
X \\
x \\
x \\
\end{bmatrix}
\begin{bmatrix}
0 \\
0 \\
1 \\
\end{bmatrix}
= \begin{bmatrix}
X \\
1 \\
\end{bmatrix}
\begin{bmatrix}
0 \\
A - x + B \\
\frac{N + F}{F - N} d \\
\end{bmatrix}
\]

\[
\Rightarrow A - x + B = \frac{1}{\frac{N + F}{F - N} d}
\]

Also:
\[
\begin{bmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
X \\
\frac{N + F}{F - N} d \\
x \\
\end{bmatrix}
\begin{bmatrix}
0 \\
1 \\
\end{bmatrix}
= \begin{bmatrix}
X \\
\frac{N + F}{F - N} d \\
\end{bmatrix}
\begin{bmatrix}
0 \\
\frac{N + F}{F - N} d \\
\end{bmatrix}
\]

That is:
\[
\begin{bmatrix}
2 & 0 & \frac{N + F}{F - N} d & 0 \\
0 & \frac{N + F}{F - N} d & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
= M_2
\]

Scale the whole matrix by \(d\):
\[
\begin{bmatrix}
2d & 0 & \frac{N + F}{F - N} d & 0 \\
0 & \frac{N + F}{F - N} d & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

So we have the new projection matrix:
\[
\begin{bmatrix}
X \\
\frac{N + F}{F - N} d \\
0 \\
\end{bmatrix}
\begin{bmatrix}
0 \\
1 \\
\end{bmatrix}
\]

Now, if we set the image plane at the near plane,
\[
\text{that is, } d = N.
\]

Then we have the following final matrix
\[
\begin{bmatrix}
\frac{2N}{x_{\text{max}} - x_{\text{min}}} & 0 & \frac{x_{\text{max}} + x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} & 0 \\
0 & \frac{2N}{y_{\text{max}} - y_{\text{min}}} & \frac{y_{\text{max}} + y_{\text{min}}}{y_{\text{max}} - y_{\text{min}}} & 0 \\
0 & 0 & \frac{-(N+R)}{F-N} & -\frac{2NF}{F-N} \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

OpenGL perspective projection matrix
Code for GL

Courtesy:
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
GLM

OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL).

Provides classes and functions designed and implemented following as strictly as possible the GLSL conventions and functionalities.

When a programmer knows GLSL, he knows GLM as well, making it really easy to use.
```cpp
glm::mat4 myMatrix;
glm::vec4 myVector;

// fill myMatrix and myVector somehow

glm::vec4 transformedVector = myMatrix * myVector;

// Again, in this order! this is important.
```
GLSL

mat4 myMatrix;
vec4 myVector;

// fill myMatrix and myVector somehow
vec4 transformedVector = myMatrix * myVector;

// Yeah, it's pretty much the same than GLM
Identity

glm::mat4 myIdentityMatrix = glm::mat4(1.0f);
Translate

GLM -
#include <glm/transform.hpp> // after <glm/glm.hpp>
glm::mat4 myMatrix = glm::translate(10.0f, 0.0f, 0.0f);
glm::vec4 myVector(10.0f, 10.0f, 10.0f, 0.0f);
glm::vec4 transformedVector = myMatrix * myVector;

GLSL -
vec4 transformedVector = myMatrix * myVector;
Scaling

// Use #include <glm/gtc/matrix_transform.hpp> and #include
<glm/gtx/transform.hpp>

glm::mat4 myScalingMatrix = glm::scale(2.0f, 2.0f, 2.0f);
Rotation

// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>

glm::vec3 myRotationAxis( ?, ?, ?);

glm::rotate( angle_in_degrees, myRotationAxis );
Accumulating Transforms

TransformedVector = TranslationMatrix * RotationMatrix * ScaleMatrix * OriginalVector;
In Code

GLM

glm::mat4 myModelMatrix = myTranslationMatrix * myRotationMatrix * myScaleMatrix;

glm::vec4 myTransformedVector = myModelMatrix * myOriginalVector;

GLSL

mat4 transform = mat2 * mat1;
vec4 out_vec = transform * in_vec;
In Diagrams

Model Coordinates

[[Model Matrix]]

World Coordinates
In Pictures
Camera/Eye Space

```cpp
glm::mat4 ViewMatrix = glm::translate(-3.0f, 0.0f, 0.0f);
```
Camera/Eye Space

```cpp
glm::mat4 CameraMatrix = glm::LookAt(
    cameraPosition, // the position of your camera, in world space
    cameraTarget,   // where you want to look at, in world space
    upVector        // probably glm::vec3(0,1,0),
                    // but (0,-1,0) would make you looking upside-down
);
```

Transform objects from world to eye space
gluLookAt

LookAt(eye, at, up)
Camera Coordinate Frame
Camera Space

Right hand coordinate system

\[ \vec{n} = at - eye \]
\[ \vec{n} = \frac{\vec{n}}{\|\vec{n}\|} \]
\[ \vec{u} = up \times \vec{n} \]
\[ \vec{v} = \vec{n} \times \vec{u} \]

\[
\mathbf{V} = \begin{pmatrix}
    u_x & u_y & u_z & -\mathbf{eye} \cdot \mathbf{u} \\
    v_x & v_y & v_z & -\mathbf{eye} \cdot \mathbf{v} \\
    n_x & n_y & n_z & -\mathbf{eye} \cdot \mathbf{n} \\
    0   & 0   & 0   & 1
\end{pmatrix}
\]
Old Style

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    ...
}
New World

- Create a view matrix

\[
\text{view} = \text{glm::lookAt}(\text{glm::vec3}(0.0, 2.0, 2.0), \text{glm::vec3}(0.0, 0.0, 0.0), \text{glm::vec3}(0.0, 1.0, 0.0));
\]

- Combine with modeling matrices

\[
\text{glm::mat4 model} = \text{glm::mat4}(1.0f);
\text{model} = \text{glm::rotate} (\text{model}, \text{angle}, \text{glm::vec3}(0.0f, 0.0f, 1.0f));
\text{model} = \text{glm::scale} (\text{model}, \text{scale\_size, scale\_size, scale\_size});
\]

\[
\text{glm::mat4 modelview} = \text{view} \ast \text{model};
\]
Working with Old World

```c
// begin to draw your geometry
...
```
Projection Matrices
In Code

// Generates a really hard-to-read matrix, but a normal, standard 4x4 matrix nonetheless

glm::mat4 projectionMatrix = glm::perspective(
    FoV,       // The horizontal Field of View, in degrees : the amount of "zoom".
    // Think "camera lens". Usually between 90° (extra wide) and 30° (quite zoomed in)
    4.0f / 3.0f, // Aspect Ratio. Depends on the size of your window.
    // Notice that 4/3 == 800/600 == 1280/960, sounds familiar?
    0.1f,      // Near clipping plane. Keep as big as possible, or you'll get precision issues.
    100.0f     // Far clipping plane. Keep as little as possible.
);
Effect
In Diagrams

More Code

C++ : compute the matrix

```cpp
glm::mat4 MVPmatrix = projection * view * model;
// Remember : inverted !
```

// GLSL : apply it
```cpp
transformed_vertex = MVP * in_vertex;
```
Combined
Generate Matrix

// Projection matrix : 45°
// Field of View, 4:3 ratio, display range : 0.1 unit <-> 100 units
 glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.0f);

// Camera matrix
 glm::mat4 View = glm::lookAt(
   glm::vec3(4,3,3), // Camera is at (4,3,3), in World Space
   glm::vec3(0,0,0), // and looks at the origin
   glm::vec3(0,1,0)  // Head is up (set to 0,-1,0 to look upside-down)
);

// Model matrix : an identity matrix (model will be at the origin)
 glm::mat4 Model = glm::mat4(1.0f); // Changes for each model!

// Our ModelViewProjection : multiplication of our 3 matrices
 glm::mat4 MVP = Projection * View * Model;

// Remember, matrix multiplication is the other way around
GLSL Takes Over

// Get a handle for our "MVP" uniform.
// Only at initialisation time.
GLuint MatrixID = glGetUniformLocation(programID, "MVP");

// Send our transformation to the currently bound shader,
// in the "MVP" uniform
// For each model you render, since the MVP will be different
// (at least the M part)

glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
Use It

in vec3 vertexPosition_modelspace;
uniform mat4 MVP;

void main(){
  // Output position of the vertex, in clip space : MVP * position
  vec4 v = vec4(vertexPosition_modelspace, 1);
  // Transform an homogeneous 4D vector, remember ?
  gl_Position = MVP * v;
}

Old Style
OpenGL Orthogonal Viewing

\[ \text{Ortho}(\text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far}) \]

near and far measured from camera
OpenGL Perspective

Frustum(left, right, bottom, top, near, far)
Using Field of View

• With Frustum it is often difficult to get the desired view

• Perspective(fovy, aspect, near, far) often provides a better interface

![Diagram of perspective with labels: fovy, aspect = w/h, front plane]
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJETION);
    glLoadIdentity();
    gluPerspective(fove, aspect, near, far);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    my_display();    // your display routine
}
Can Still GLM

- Set up the projection matrix

```cpp
glm::mat4 projection = glm::mat4(1.0f);
projection = glm::perspective(60.0f, 1.0f, .1f, 100.0f);
```

- Load the matrix to GL_PROJECTION

```cpp
glMatrixMode(GL_PROJECTION);
glLoadIdentityMatrxf(&projection[0][0]);
```
Why we need shading

• Just attach color \texttt{glColor}

• But
Shading

• Why does the shape?

• Light-material interaction at points -> different color or shade

• Factor
  – Light sources
  – Material properties
  – Location of viewer
  – Surface orientation
Global Effects

- shadow
- multiple reflection
- translucent surface
Light Sources

General Difficult!
Simple Light Sources
Point Sources

Point source
Model with position and color
Distant source = infinite distance away (parallel)
Spot Light

Spotlight
Restrict light from ideal point source
Ambient

Ambient light

Same amount of light in scene

Model contribution of all sources and reflecting surfaces
Light-Matter Interaction
Volume 1
Fundamentals and Applications

JOHN WEINER
P.-T. HO
Indirect/Direct Light
Light strikes object - is partially absorbed & partially scattered (reflected)
Amount reflected determines the color and brightness of the object

Red surface appears red in white light - red component is reflected and rest is absorbed
The Surface

Reflected light is scattered depending on smoothness and orientation of the surface
Surface Type - Smooth

- Very Smooth - more reflected light concentrated in one direction – like a perfect mirror
TBT - specular
Surface Type - Rough

Scatters light in all directions

rough surface
Smooth vs. Rough

Figure 1: Specular and Diffuse Reflection

- Specular Reflection
- Diffuse Reflection

Diagram showing pure diffuse, pure specular, and glossy reflections.
Smooth vs. Rough
The Phong Illumination Model
Phong Model

A simple local 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
Computing \( r \)

Want all three to be unit length

\[ r = 2(l \cdot n)n - l \]
Diffuse
Lambertian Surface

Amount reflected is proportional to vertical component of incoming light

- reflected light $\sim \cos \theta_i$
- Or $I_d \ k_d \ \cos \theta_i$
- $\cos \theta_i = \mathbf{l} \cdot \mathbf{n}$ if vectors normalized
- Three coefficients, $k_{dr}$, $k_{db}$, $k_{dg}$ that measure each color component is reflected
Specular or Glossy Surface
Specular Surfaces

Specular highlights due to incoming light being reflected in directions close to the direction of a perfect reflection

Not Ideal Mirror

specular highlight
Specular Reflections

\[ I_r \sim k_s I \cos^\alpha \phi \]

- reflected intensity
- shininess coeff
- incoming intensity
- absorption coeff

Diagram showing the relationship between incoming and reflected intensities.
The Shininess Coefficient

$\alpha$ between

- 100 and 200 correspond to metals
- 5 and 10 give surface that look like plastic
Ambient Light

• Result of multiple interactions between (large) light sources and objects in environment
• Amount and color depend on both color of light(s) and material properties of the object
• Add $k_a l_a$ to diffuse and specular terms

reflection coef intensity of ambient light
Light Source

• We add results from each light source

• Each light source
  – diffuse, specular, and ambient terms
  – Separate red, green and blue components

• Hence, 9 coefficients for each point source
  – \( I_{dr}, I_{dg}, I_{db}, I_{sr}, I_{sg}, I_{sb}, I_{ar}, I_{ag}, I_{ab} \)
Material Properties

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

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

\[ I = k_d l_d (l \cdot n) + k_s l_s (v \cdot r)^a + k_a l_a \]

For each color component we add contributions from all sources.
Distance Terms

- Light inversely proportional to square of distance
- A factor added to diffuse and specular components:
  \[
  \frac{I}{(a + bd + cd^2)}
  \]
  \[
  d = \left\| \vec{p} - \vec{L} \right\|
  \]
  \[
  a, b, c \text{ – constants}
  \]
- Constant, linear terms soften effect of point source
Modified Phong Model

- Specular term requires new reflection and view vector for each vertex

- Blinn approximation – using halfway vector

- And more efficient
The Halfway Vector

\[ \mathbf{h} \text{ is normalized vector halfway between } \mathbf{l} \text{ and } \mathbf{v} \]

\[ \mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{||\mathbf{l} + \mathbf{v}||} \]
Using the halfway vector

• Replace \((v \cdot r)^a\) by \((n \cdot h)^b\)

• \(b\) is chosen to match shininess

• Halfway angle half of angle \(r\) and \(v\) if coplanar

• Modified Phong, Blinn lighting model
  – OpenGL standard
Teapot Gallery

Only differences in these teapots are the parameters in the modified Phong model.
Computation of Vectors

• $l$ and $v$ are specified by the application
• Can compute $r$ from $l$ and $n$

• Problem is determining $n$
• Depends on underlying representation of surface

• OpenGL leaves determination of normal to application
  – Exception for GLU quadrics and Bezier surfaces was deprecated
Computing $r$

Vectors unit length

\[
\begin{align*}
\text{left} \Delta & : L + S = N(N \cdot L) \\
\text{right} \Delta & : N(N \cdot L) + S = R \\
R & = 2(L \cdot N)N - L
\end{align*}
\]
Plane Normals

- Equation of plane: \( ax + by + cz + d = 0 \)
- Plane is determined by three points \( p_0, p_1, p_2 \) or normal \( \mathbf{n} \) and \( p_0 \)
- Normal can be obtained by
  \[
  \mathbf{n} = (p_2 - p_0) \times (p_1 - p_0)
  \]
Normal to Sphere

• Implicit function $f(x,y,z)=0$
• Normal given by gradient
• Sphere $f(p)=p \cdot p^T - 1$

$n = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}]^T = p$
Parametric Form

• For sphere
  \[ x = x(u,v) = \cos u \sin v \]
  \[ y = y(u,v) = \cos u \cos v \]
  \[ z = z(u,v) = \sin u \]

• Tangent plane determined by vectors

  \[ \frac{\partial p}{\partial u} = [\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}]^T \]
  \[ \frac{\partial p}{\partial v} = [\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}]^T \]

• Normal given by cross product

  \[ n = \frac{\partial p}{\partial u} \times \frac{\partial p}{\partial v} \]
Transforming Normals
Normals

Algebra tricks to derive correct transform

Incorrect to transform like points
Transforming Normal

\[ p \rightarrow Mp \quad n \rightarrow Qn \quad Q = ? \]

\[ n^T p = 0 \]

\[ n^T Q^T Mp = 0 \quad \Rightarrow \quad Q^T M = I \]

\[ Q = (M^{-1})^T \]

Use Q and not M to transform Normals
Q: What about Rotation?
What about Translation?
General Case

Parametric normals for other simple cases

- Quadrics
- Parameteric polynomial surfaces
  - Bezier surface patches
Polygonal Shading
Polygonal Shading

• Per vertex shading
  – shading calculations are done for each vertex
  – Vertex colors become vertex shades
  – sent to the vertex shader as a vertex attribute
  – Or send parameters to vertex shader & compute shade

• By default, vertex shades interpolated across object if passed to fragment shader as a varying variable (smooth shading)

• We can also use uniform variables to shade with a single shade (flat shading)
Polygon Normals

- Triangles have a single normal
  - Shades at vertices as computed by the Phong model
  - Identical for a distant viewer (default) or if there is no specular component
- Want different normals at each vertex even though
Smooth Shading

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

- Previous example is not general since normal at each vertex is known analytically.
- For polygonal models, Gouraud proposed we use the average of normals around a mesh vertex.

\[ n = \frac{n_1 + n_2 + n_3 + n_4}{|n_1 + n_2 + n_3 + n_4|} \]
Interpolation

\[ \Delta I_s = \frac{\Delta x}{x_b - x_a} (I_b - I_a) \]

\[ I_{s,n} = I_{s,n-1} + \Delta I_s \]

\[ \Delta N_{sx} = \frac{\Delta x}{x_b - x_a} (N_{bx} - N_{ax}) \]

\[ \Delta N_{sy} = \frac{\Delta x}{x_b - x_a} (N_{by} - N_{ay}) \]

\[ \Delta N_{sz} = \frac{\Delta x}{x_b - x_a} (N_{bz} - N_{az}) \]
Gouraud and Phong Shading

• Gouraud Shading
  – Find average normal at each vertex (vertex normals)
  – Apply modified Phong model at each vertex
  – Interpolate vertex shades across each polygon

• Phong shading
  – Find vertex normals
  – Interpolate vertex normals across edges
  – Interpolate edge normals across polygon
  – Apply modified Phong model at each fragment
Comparison

![Comparison of rendering techniques](image)

- **Flat shading**
- **Gouraud shading**
- **Phong shading**

- A teapot illustrates the difference in shading techniques.
- A sphere is used to demonstrate the effects of these techniques.
- The images show the flat, Gouraud, and Phong shading of a sphere and a teapot.
Flat Shading
Gouraud Shading
Phong Shading
Gouraud versus Phong
Comparison

• If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges

• Phong shading requires much more work than Gouraud shading
  – Until recently not available in real time systems
  – Now can be done using fragment shaders

• Both need data structures to represent meshes so we can obtain vertex normals
Comparison

• Specular highlights more accurate in Phong
  – vertex highlight is much sharper
  – a highlight can occur within a polygon
• Mach banding greatly reduced
• Cost is higher because reflection model applied per pixel
Mach Band
Perspective Effects

- Anomalies occur because interpolation is carried out in screen space, after perspective transformation.
- Suppose P2 much more distant than P1. P is midway in screen space so gets 50 : 50 intensity (Gouraud) or normal (Phong).
- ... but in world coordinates it is much nearer to P1 than P2.
Averaging Normals

- Averaging normals of adjacent faces works
- Beware corrugated surfaces where averaging unduly smooths
GL SL functions
Lay of the Land

• OpenGL shading methods
  – per vertex vs per fragment shading
  – Where to carry out

• Discuss polygonal shading
  – Flat
  – Smooth
  – Gouraud
OpenGL shading

- Need
  - Normals
  - Material properties
  - Lights

- State-based shading functions have been deprecated
  - glNormal, glMaterial, glLight

- Send attributes to shaders
Normalization of Vectors

- Cosine terms in lighting computed using dot product
- Unit length vectors simplify calculation
- Set the magnitudes to have unit length but
  - Length can be affected by transformations
  - Note that scaling does not preserved length
- GLSL has a normalization function
Normal for Triangle

plane \quad n \cdot (p - p_0) = 0

n = (p_2 - p_0) \times (p_1 - p_0)

normalize n \leftarrow n / |n|

Note that right-hand rule determines outward face
Point Light Source

For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and for the position.

```cpp
vec4 diffuse0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 ambient0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 specular0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 light0_pos = vec4(1.0, 2.0, 3.0, 1.0);
```
Distance and Direction

• Source colors are specified in RGBA
• Position is given in homogeneous coordinates
  – If w =1.0, point source
  – If w =0.0 parallel source given direction vector
• Coefficients in distance terms
  – Quadratic $1/(a+b*d+c*d^2)$
  – d is distance from point rendered to light source
Spotlights

• Derive from point source
  – Direction
  – Cutoff
  – Attenuation Proportional to $\cos^a \phi$
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

• A global ambient term helpful for testing
Moving Light Sources

• Geometric objects whose positions, directions are affected by model-view matrix

• Depending on where we place the position (direction)
  – Move the light source(s) with the object(s)
  – 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
Material Properties

- Material properties should match light model

- Reflectivities

- \( w \) component gives opacity

```
vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);
vec4 diffuse = vec4(1.0, 0.8, 0.0, 1.0);
vec4 specular = vec4(1.0, 1.0, 1.0, 1.0);
GLfloat shine = 100.0
```
Front and Back Faces

- Every face has a front and back
- For many objects, we never see the back face so ignore
- we can handle in shader

back faces not visible  back faces visible
Emissive Term

- Simulate a light source in OpenGL by giving a material an emissive component

- This component is unaffected by any sources or transformations
Transparency

- Material properties as RGBA values
- A value can be used to make surface translucent
- Default is all surfaces are opaque regardless of A
- Enable blending
// vertex shader
in vec4 vPosition;
in vec3 vNormal;
out vec4 color;  // vertex shade

// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
void main()
{
    // Transform vertex position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;

    vec3 L = normalize(LightPosition.xyz - pos);
    vec3 E = normalize(-pos);
    vec3 H = normalize(L + E);

    // Transform vertex normal into eye coordinates
    vec3 N = normalize(ModelView*vec4(vNormal, 0.0)).xyz;
// Compute terms in the illumination equation
vec4 ambient = AmbientProduct;

float Kd = max(dot(L, N), 0.0);
vec4 diffuse = Kd*DiffuseProduct;
float Ks = pow(max(dot(N, H), 0.0), Shininess);
vec4 specular = Ks * SpecularProduct;
if( dot(L, N) < 0.0 ) specular = vec4(0.0, 0.0, 0.0, 1.0);
gl_Position = Projection * ModelView * vPosition;

color = ambient + diffuse + specular;
color.a = 1.0;
}
// fragment shader

in vec4 color;

void main()
{
    gl_FragColor = color;
}
// vertex shader
in vec4 vPosition;
in vec3 vNormal;

// output values that will be interpolated per-fragment
out vec3 fN;
out vec3 fE;
out vec3 fL;

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;
void main()
{
    fN = vNormal;
    fE = vPosition.xyz;
    fL = LightPosition.xyz;

    if( LightPosition.w != 0.0 ) {
        fL = LightPosition.xyz - vPosition.xyz;
    }

    gl_Position = Projection*ModelView*vPosition;
}
// fragment shader

// per-fragment interpolated values from the vertex shader
in vec3 fN;
in vec3 fL;
in vec3 fE;

uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;
void main()
{
    // Normalize the input lighting vectors

    vec3 N = normalize(fN);
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);

    vec3 H = normalize( L + E );
    vec4 ambient = AmbientProduct;
float Kd = max(dot(L, N), 0.0);
    vec4 diffuse = Kd*DiffuseProduct;

float Ks = pow(max(dot(N, H), 0.0), Shininess);
vec4 specular = Ks*SpecularProduct;

// discard the specular highlight if the light's behind the vertex
if( dot(L, N) < 0.0 )
    specular = vec4(0.0, 0.0, 0.0, 1.0);

    gl_FragColor = ambient + diffuse + specular;
    gl_FragColor.a = 1.0;
Next

FIGURE 10.41  Rendered teapots.