CS6533/CS4533 Lectures 11-12 Slides/Notes

Shading and Illumination; Compositing
(Notes, Ch 14, Notes)

By Prof. Yi-Jen Chiang
CSE Dept., Tandon School of Engineering
New York University

1

* **Continued on Shading and Illumination:**

- First we reviewed the ``Overall Formula’’ of the Phong Reflection Model (shown in the next 2 slides) as presented last time, which is implemented in the sample program ``Handout: rotate-cube-shading.cpp”.
- Discussed the sample program ``Handout: rotate-cube-shading.cpp” (complete sample program has been posted at
  https://cse.engineering.nyu.edu/cs653/Rotate-Cube-Shading.tar.gz)
- Some screenshots of the sample program with annotations are then shown next.
- Showed a demo of the sample program ``Rotate-Cube-Shading’’.
- Then we finished up the last 2 pages (p.14 & p.15) of the last lecture notes/slides “Lecture-9-10.pdf” on Normal Matrix.

* **New topic: Composition Techniques.**

2

Note: component-wise multiplications: $K_a \cdot L_a$, $K_d \cdot L_d$, $K_s \cdot L_s$. They are attenuated differently.

Overall formula:

global ambient light

$$I = K_a \cdot L_{a\text{-global}} + \sum_{\text{light } i} (\text{Attenuation})_i \cdot \Big[K_a \cdot L_a + K_d \cdot L_d \cdot \max\{(\ell \cdot n), 0\}$$
$$+ [\text{if } \ell \cdot n \geq 0] \cdot K_s \cdot L_s \cdot \big(\max\{(r \cdot v), 0\}\big)^{\alpha} \Big]_i$$

replaced with $(n \cdot h)$

(1) If light i is a distant (directional) light, then

① $(\text{Attenuation})_i = 1$

② vector ℓ (from pt p to light source i) $= -$(distant light direction L)

i.e. $\ell = -L$

$L \quad \ell = -L$

(Also, use this ℓ to compute $h = \text{normalize}(\ell + v)$)

(2) If light i is a point source, then

① $(\text{Attenuation})_i = \dfrac{1}{a + bd + cd^2}$

where d = distance from pt p to the light source position.

a, b, c: constant, linear, quadratic attenuations

3

(3) If light i is a spotlight then

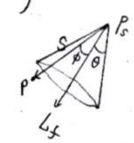
① $(\text{Attenuation})_i = \dfrac{1}{a + bd + cd^2} \cdot (\text{spotlight\_attenuation})_i$ — a, b, c, d are as in (2) point source.

In particular $d = |\overrightarrow{P_s P}|$

$\ell = \text{normalize}(\overrightarrow{P P_s}) = -s$

$\therefore s = -\ell$

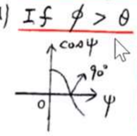
② $(\text{spotlight\_attenuation})_i = ?$



(Let $L_f = \text{normalize}(L_f)$)

θ: spotlight cut-off angle $\theta \in [0, 90°]$

(a) If $\phi > \theta$ then contribution $= 0$



In the range $[0, 90°]$

$\phi > \theta \iff \cos\phi < \cos\theta$

$\iff (L_f \cdot s) < \cos\theta$

$\iff L_f \cdot (-\ell) < \cos\theta$

$\therefore$ If $L_f \cdot (-\ell) < \cos\theta$ then $(\text{spotlight\_attenuation})_i = 0$

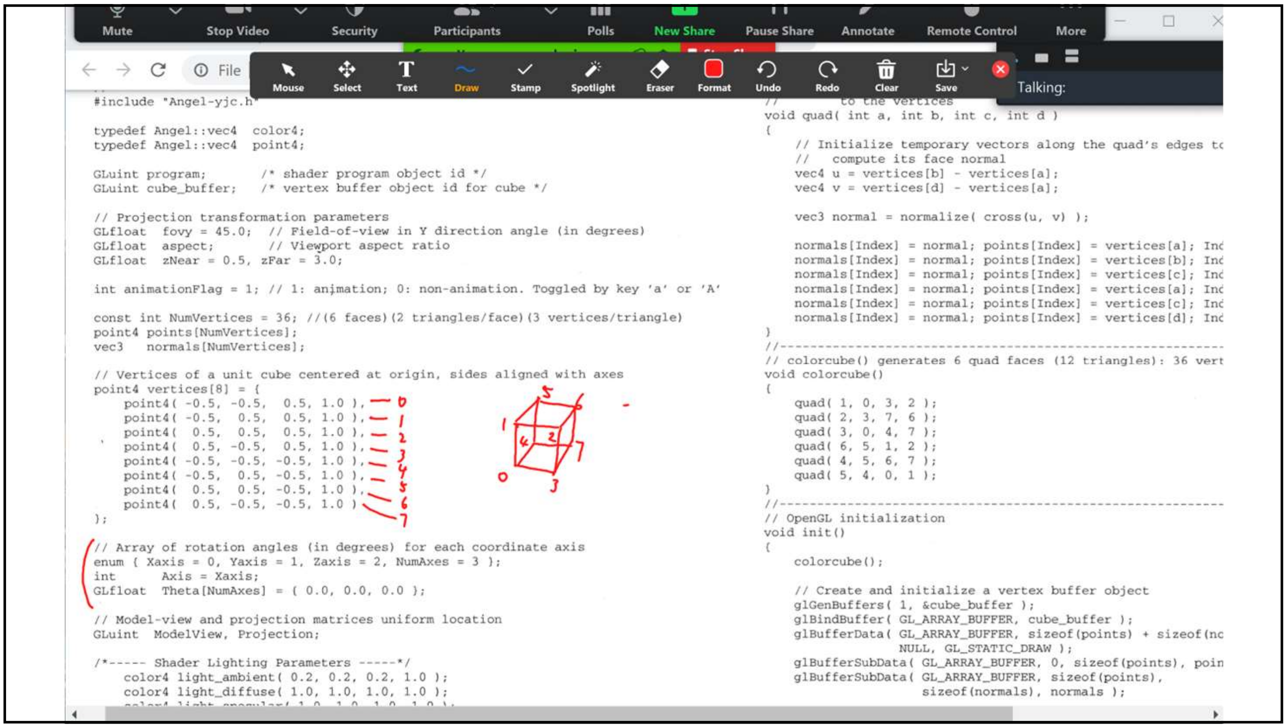
(b) Else,

$(\text{spotlight\_attenuation})_i = (\cos\phi)^e = [L_f \cdot (-\ell)]^e$.

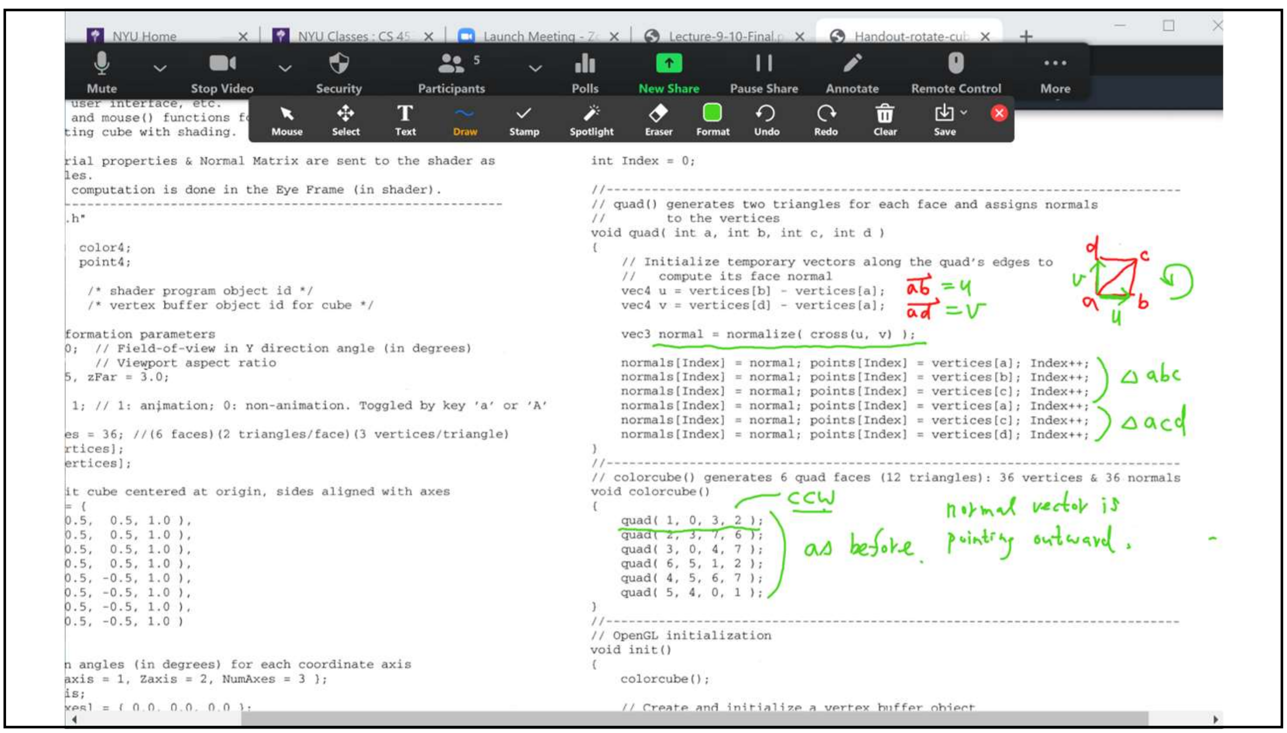
e: spotlight exponent

(Combining (a), (b): $(\text{spotlight\_attenuation})_i = [\text{if } L_f \cdot (-\ell) \geq \cos\theta] \cdot [L_f \cdot (-\ell)]^e$)

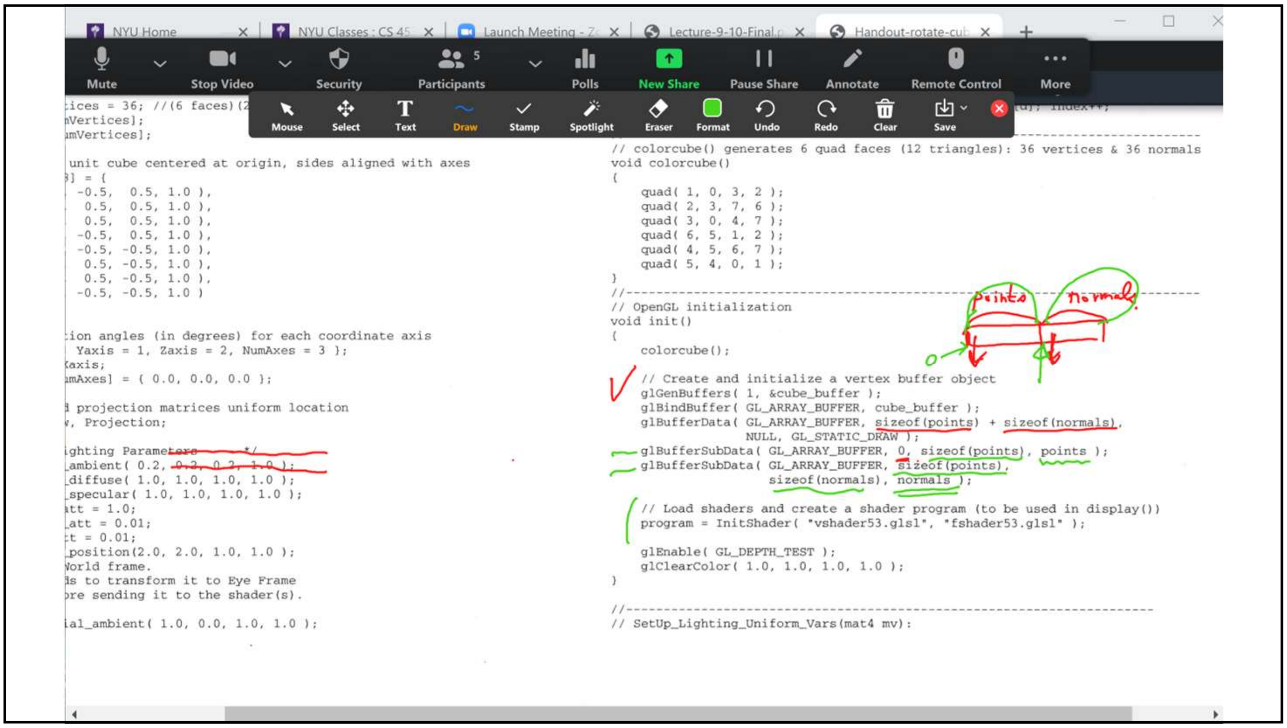
4


#include "Angel-yjc.h"
typedef Angel::vec4 color4;
typedef Angel::vec4 point4;
GLuint program; /* shader program object id */
GLuint cube_buffer; /* vertex buffer object id for cube */
// Projection transformation parameters
GLfloat fovy = 45.0; // Field-of-view in Y direction angle (in degrees)
GLfloat aspect; // Viewport aspect ratio
GLfloat zNear = 0.5, zFar = 3.0;
int animationFlag = 1; // 1: anjmation; 0: non-animation. Toggled by key 'a' or 'A'
const int NumVertices = 36; //(6 faces)(2 triangles/face)(3 vertices/triangle)
point4 points[NumVertices];
vec3 normals[NumVertices];
// Vertices of a unit cube centered at origin, sides aligned with axes
point4 vertices[8] = {
point4(-0.5, -0.5, 0.5, 1.0), — 0
point4(-0.5, 0.5, 0.5, 1.0), — 1
point4(0.5, 0.5, 0.5, 1.0), — 2
point4(0.5, -0.5, 0.5, 1.0), — 3
point4(-0.5, -0.5, -0.5, 1.0), — 4
point4(-0.5, 0.5, -0.5, 1.0), — 5
point4(0.5, 0.5, -0.5, 1.0), — 6
point4(0.5, -0.5, -0.5, 1.0) — 7
};
// Array of rotation angles (in degrees) for each coordinate axis
enum { Xaxis = 0, Yaxis = 1, Zaxis = 2, NumAxes = 3 };
int Axis = Xaxis;
GLfloat Theta[NumAxes] = { 0.0, 0.0, 0.0 };
// Model-view and projection matrices uniform location
GLuint ModelView, Projection;
/*----- Shader Lighting Parameters -----*/
color4 light_ambient(0.2, 0.2, 0.2, 1.0);
color4 light_diffuse(1.0, 1.0, 1.0, 1.0);
void quad(int a, int b, int c, int d)
// Initialize temporary vectors along the quad's edges to
// compute its face normal
vec4 u = vertices[b] - vertices[a];
vec4 v = vertices[d] - vertices[a];
vec3 normal = normalize(cross(u, v));
normals[Index] = normal; points[Index] = vertices[a];
normals[Index] = normal; points[Index] = vertices[b];
normals[Index] = normal; points[Index] = vertices[c];
normals[Index] = normal; points[Index] = vertices[a];
normals[Index] = normal; points[Index] = vertices[c];
normals[Index] = normal; points[Index] = vertices[d];
// colorcube() generates 6 quad faces (12 triangles): 36 vert
void colorcube()
quad(1, 0, 3, 2);
quad(2, 3, 7, 6);
quad(3, 0, 4, 7);
quad(6, 5, 1, 2);
quad(4, 5, 6, 7);
quad(5, 4, 0, 1);
// OpenGL initialization
void init()
colorcube();
// Create and initialize a vertex buffer object
glGenBuffers(1, &cube_buffer);
glBindBuffer(GL_ARRAY_BUFFER, cube_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points) + sizeof(nc
NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(points), poin
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points),
sizeof(normals), normals);

5


int Index = 0;
// quad() generates two triangles for each face and assigns normals
// to the vertices
void quad(int a, int b, int c, int d)
// Initialize temporary vectors along the quad's edges to
// compute its face normal
vec4 u = vertices[b] - vertices[a]; ab = u
vec4 v = vertices[d] - vertices[a]; ad = v
vec3 normal = normalize(cross(u, v));
normals[Index] = normal; points[Index] = vertices[a]; Index++;
normals[Index] = normal; points[Index] = vertices[b]; Index++;
normals[Index] = normal; points[Index] = vertices[c]; Index++;
△abc
normals[Index] = normal; points[Index] = vertices[a]; Index++;
normals[Index] = normal; points[Index] = vertices[c]; Index++;
normals[Index] = normal; points[Index] = vertices[d]; Index++;
△acd
// colorcube() generates 6 quad faces (12 triangles): 36 vertices & 36 normals
void colorcube()
CCW
quad(1, 0, 3, 2);
quad(2, 3, 7, 6);
quad(3, 0, 4, 7);
quad(6, 5, 1, 2);
quad(4, 5, 6, 7);
quad(5, 4, 0, 1);
as before.
normal vector is pointing outward.
// OpenGL initialization
void init()
colorcube();

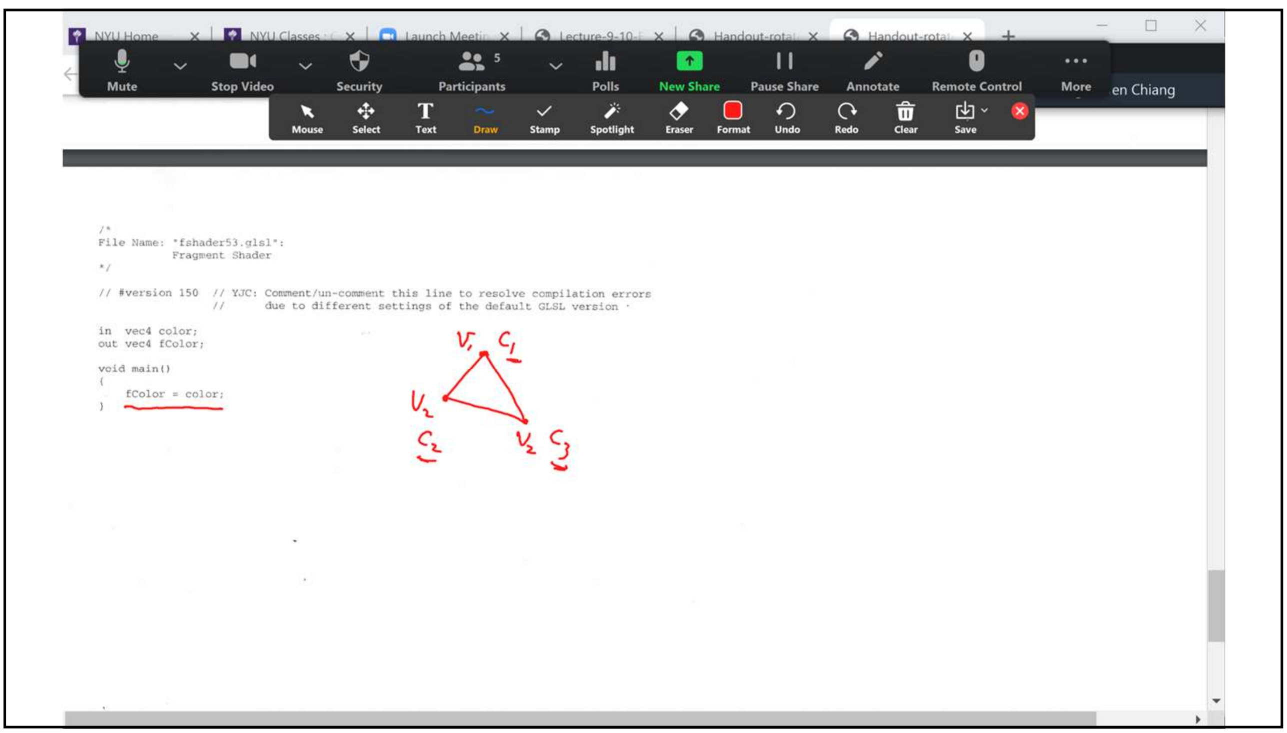
6



```
ices = 36; //(6 faces)(2
nVertices];
umVertices];

 unit cube centered at origin, sides aligned with axes
3] = {
 -0.5,  0.5, 1.0 ),
  0.5,  0.5, 1.0 ),
  0.5,  0.5, 1.0 ),
 -0.5,  0.5, 1.0 ),
 -0.5, -0.5, 1.0 ),
  0.5, -0.5, 1.0 ),
  0.5, -0.5, 1.0 ),
 -0.5, -0.5, 1.0 )

ion angles (in degrees) for each coordinate axis
 Yaxis = 1, Zaxis = 2, NumAxes = 3 };
Xaxis;
umAxes] = { 0.0, 0.0, 0.0 };

d projection matrices uniform location
v, Projection;

ighting Parameters      */
_ambient( 0.2, 0.2, 0.2, 1.0 );
_diffuse( 1.0, 1.0, 1.0, 1.0 );
_specular( 1.0, 1.0, 1.0, 1.0 );
att = 1.0;
_att = 0.01;
t = 0.01;
_position(2.0, 2.0, 1.0, 1.0 );
World frame.
ds to transform it to Eye Frame
ore sending it to the shader(s).

ial_ambient( 1.0, 0.0, 1.0, 1.0 );
```

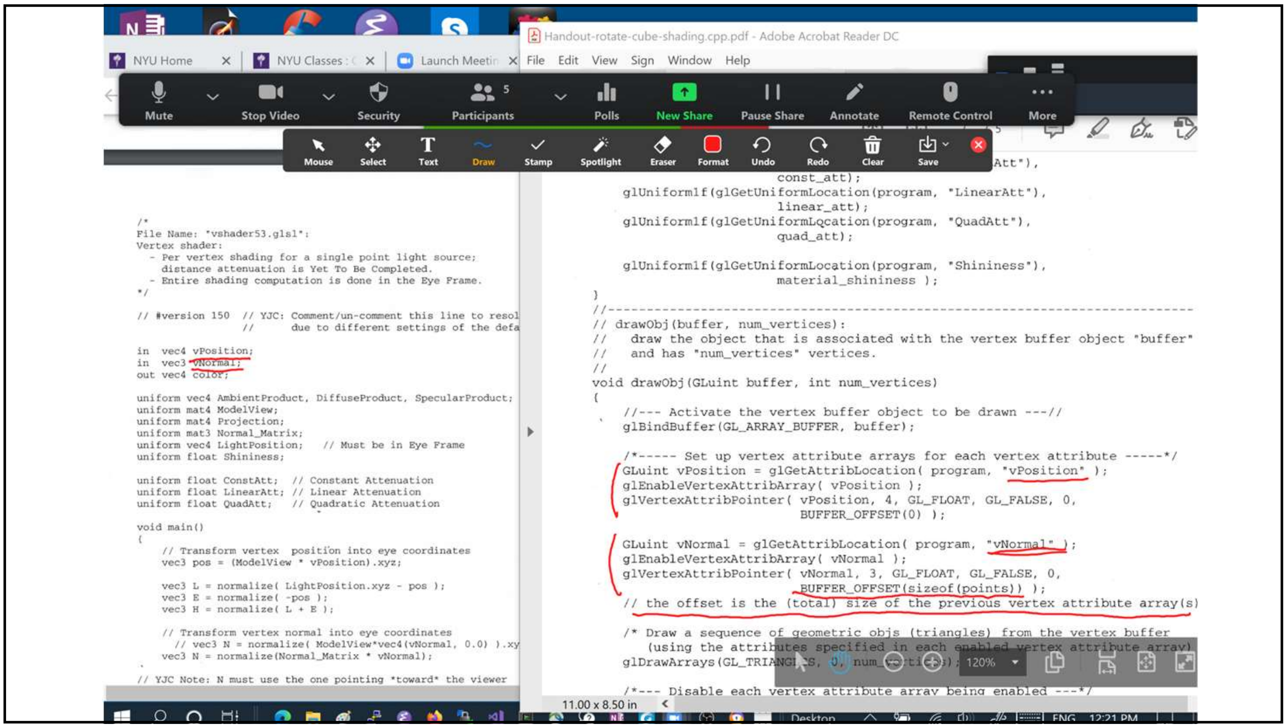
```
// colorcube() generates 6 quad faces (12 triangles): 36 vertices & 36 normals
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
//----------------------------------------------------------------------
// OpenGL initialization
void init()
{
    colorcube();

    // Create and initialize a vertex buffer object
    glGenBuffers( 1, &cube_buffer );
    glBindBuffer( GL_ARRAY_BUFFER, cube_buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof(points) + sizeof(normals),
                  NULL, GL_STATIC_DRAW );
    glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(points), points );
    glBufferSubData( GL_ARRAY_BUFFER, sizeof(points),
                     sizeof(normals), normals );

    // Load shaders and create a shader program (to be used in display())
    program = InitShader( "vshader53.glsl", "fshader53.glsl" );

    glEnable( GL_DEPTH_TEST );
    glClearColor( 1.0, 1.0, 1.0, 1.0 );
}

//----------------------------------------------------------------------
// SetUp_Lighting_Uniform_Vars(mat4 mv):
```


7



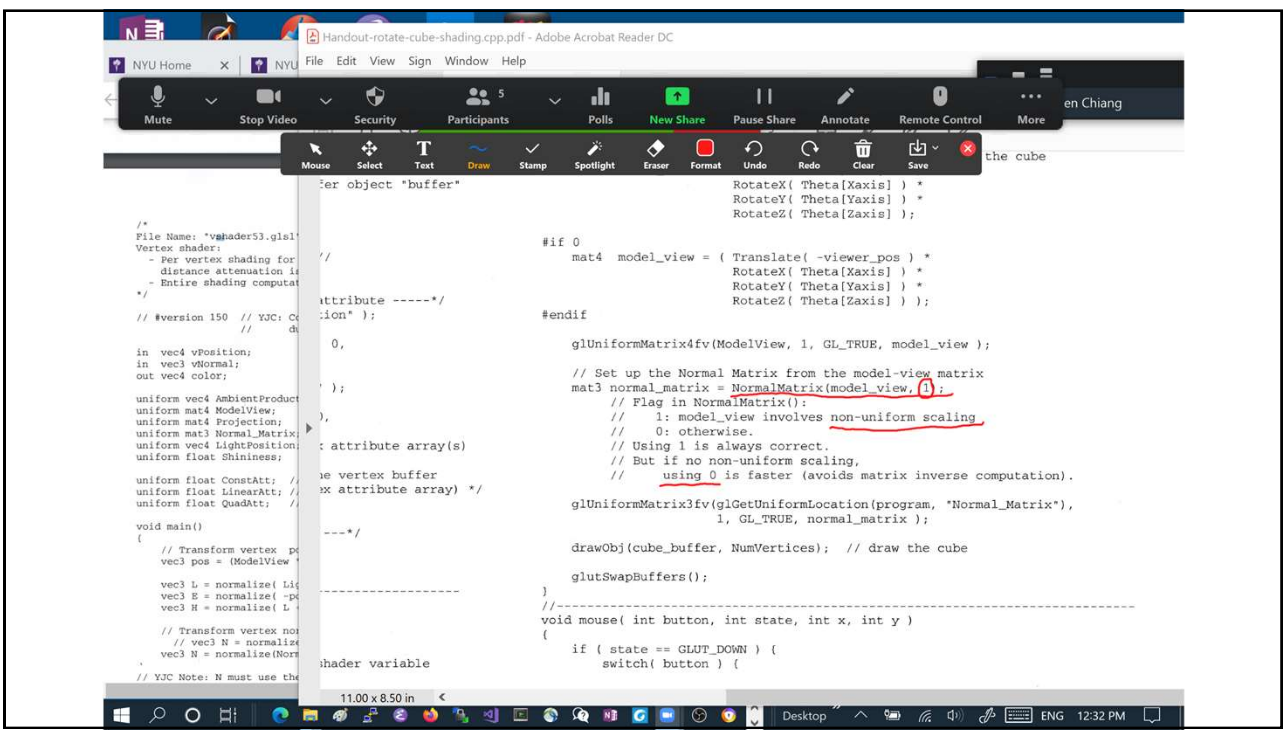
```
/*
File Name: "fshader53.glsl":
          Fragment Shader
*/

// #version 150  // YJC: Comment/un-comment this line to resolve compilation errors
                 //      due to different settings of the default GLSL version

in  vec4 color;
out vec4 fColor;

void main()
{
    fColor = color;
}
```


8



Handout-rotate-cube-shading.cpp.pdf - Adobe Acrobat Reader DC
/*
File Name: "vshader53.glsl":
Vertex shader:
- Per vertex shading for a single point light source;
distance attenuation is Yet To Be Completed.
- Entire shading computation is done in the Eye Frame.
*/
// #version 150 // YJC: Comment/un-comment this line to resol
// due to different settings of the defa
in vec4 vPosition;
in vec3 vNormal;
out vec4 color;
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform mat3 Normal_Matrix;
uniform vec4 LightPosition; // Must be in Eye Frame
uniform float Shininess;
uniform float ConstAtt; // Constant Attenuation
uniform float LinearAtt; // Linear Attenuation
uniform float QuadAtt; // Quadratic Attenuation
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)).xy
vec3 N = normalize(Normal_Matrix * vNormal);
// YJC Note: N must use the one pointing *toward* the viewer
const_att);
glUniform1f(glGetUniformLocation(program, "LinearAtt"),
linear_att);
glUniform1f(glGetUniformLqcation(program, "QuadAtt"),
quad_att);
glUniform1f(glGetUniformLocation(program, "Shininess"),
material_shininess);
}
//--
// drawObj(buffer, num_vertices):
// draw the object that is associated with the vertex buffer object "buffer"
// and has "num_vertices" vertices.
//
void drawObj(GLuint buffer, int num_vertices)
{
//--- Activate the vertex buffer object to be drawn ---//
glBindBuffer(GL_ARRAY_BUFFER, buffer);
/*----- Set up vertex attribute arrays for each vertex attribute -----*/
GLuint vPosition = glGetAttribLocation(program, "vPosition");
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
GLuint vNormal = glGetAttribLocation(program, "vNormal");
glEnableVertexAttribArray(vNormal);
glVertexAttribPointer(vNormal, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(sizeof(points)));
// the offset is the (total) size of the previous vertex attribute array(s)
/* Draw a sequence of geometric objs (triangles) from the vertex buffer
(using the attributes specified in each enabled vertex attribute array)
/*--- Disable each vertex attribute array being enabled ---*/

9



Handout-rotate-cube-shading.cpp.pdf - Adobe Acrobat Reader DC
RotateX(Theta[Xaxis]) *
RotateY(Theta[Yaxis]) *
RotateZ(Theta[Zaxis]);
#if 0
mat4 model_view = (Translate(-viewer_pos) *
RotateX(Theta[Xaxis]) *
RotateY(Theta[Yaxis]) *
RotateZ(Theta[Zaxis]));
#endif
glUniformMatrix4fv(ModelView, 1, GL_TRUE, model_view);
// Set up the Normal Matrix from the model-view matrix
mat3 normal_matrix = NormalMatrix(model_view, 1);
// Flag in NormalMatrix():
// 1: model_view involves non-uniform scaling
// 0: otherwise.
// Using 1 is always correct.
// But if no non-uniform scaling,
// using 0 is faster (avoids matrix inverse computation).
glUniformMatrix3fv(glGetUniformLocation(program, "Normal_Matrix"),
1, GL_TRUE, normal_matrix);
drawObj(cube_buffer, NumVertices); // draw the cube
glutSwapBuffers();
}
//--
void mouse(int button, int state, int x, int y)
{
if (state == GLUT_DOWN) {
switch(button) {

10

* Normal Matrix

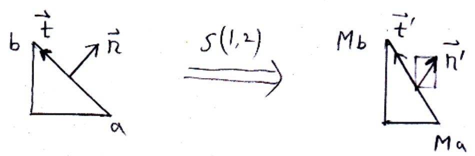
* Typically we perform shading computation in the (eye frame) (i.e. the right-handed eye frame where the eye/camera is at the origin looking at the $-z$ direction. This is the frame obtained by applying LookAt() to the world frame)

Let $\vec{t}$ be the tangent at pt p being shaded.
$\vec{n}$ " normal "

$\vec{n}, \vec{t}$ are in the model frame
M the model-view matrix
i.e. Mp puts p in the eye frame

(1) Suppose M involves (non-uniform scaling) (i.e. scaling factors in x-, y-, z- dimensions are different)

e.g. $S(1,2)$ in 2D



$$\vec{t}' = Mb - Ma = M(b-a) = M\vec{t}$$ is the tangent after transformation

i.e. We can still apply M to $\vec{t}$ to obtain the new tangent $\vec{t}'$ correctly.
But applying M to $\vec{n}$ does NOT give the correct normal vector. (since $\vec{n}'$ is NOT perpendicular to $\vec{t}'$)

(2) Deriving the correct matrix for normal vector: the (normal matrix)

Let $\vec{n} = \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix}$ $\quad \vec{n}\cdot\vec{t} = 0$ The dot product can be expressed as matrix multiplication:

$$\vec{n}\cdot\vec{t} = [\quad]\begin{bmatrix} \\ \end{bmatrix} = (\vec{n})^t\,\vec{t} \quad \text{——} (*)$$
$= 0$

$(\vec{n})^t$: transpose of $\vec{n} = [\ ]$

From (*) we have $0 = (\vec{n})^t\vec{t} = \big((\vec{n})^t \underbrace{M^{-1})(M}_{=I}\vec{t}\big)$ — (**)

But $M = \begin{bmatrix} \ell & T \\ 0 & 1 \end{bmatrix}$ and the 4-th component of $\vec{t}$ is 0 ⇒ We can ignore the 4th column of M. i.e. $\begin{bmatrix} T \\ 1 \end{bmatrix} = \begin{bmatrix} T_x \\ T_y \\ T_z \\ 1 \end{bmatrix}$

⇒ Then the 4th row of the remaining columns are 0 and can be ignored

∴ In (**) we can use ℓ to replace M:

$$\underbrace{\big((\vec{n})^t \ell^{-1}\big)}_{=(\vec{x})^t}\underbrace{(\ell\,\vec{t})}_{=\text{transformed tangent } \vec{t}'} = 0$$

$(\vec{x})^t$ where $\vec{x}$ is the transformed normal, in the form of (*):

cf: In (*): $(\vec{n})^t\vec{t} = 0$
Here: $(\vec{x})^t\vec{t}' = 0$.

∴ $(\vec{x})^t = (\vec{n})^t\ell^{-1} \Rightarrow \vec{x} = \left[(\vec{n})^t\ell^{-1}\right]^t = (\ell^{-1})^t(\vec{n})$

⇒ Desired normal $\vec{x}$ is obtained by $N \overset{\vec{n}}{\begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}}$ where the 3×3 matrix N (normal matrix) is $\big((\ell^{-1})^t\big)$

Simplification:

(scaling factors in x-, y-, z-dim are all the same)

(3) If M only involves translations, rotations, uniform scaling, and LookAt( )

then: translations have no effect on ℓ
LookAt( ) has translation and rotation
$\Rightarrow$ ℓ only involves rotations and uniform scaling

But uniform scaling has no effect after we normalize the transformed normal vector

$\Rightarrow \ell \equiv R$. But $R^{-1} = R^t$

$\therefore (\ell^{-1})^t \equiv (R^{-1})^t = (R^t)^t = R \equiv \ell$

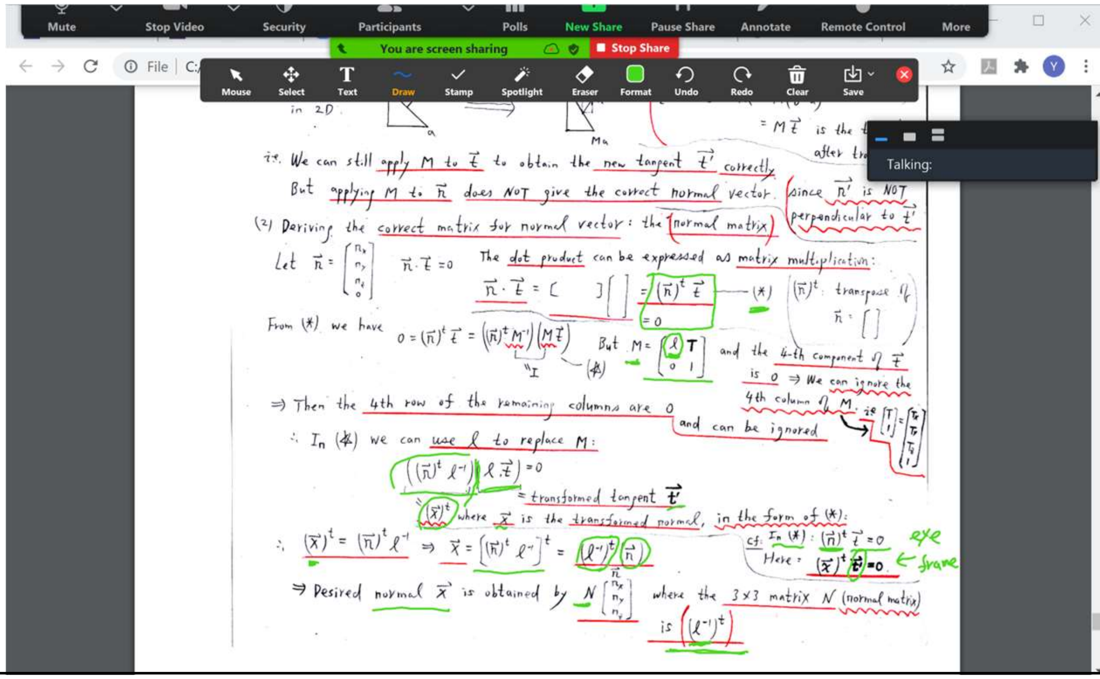
ie We can use (ℓ) to replace $(\ell^{-1})^t$ ①

4 components

(ie We can use the model-view matrix M (4x4) to apply to normal $\vec{n} = \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix}$ ②

① ≡ ②)

Screenshot for Elaboration:



11

New Topic: Compositing Techniques

Recall:
* Fragments : generated by the rasterization of geometric primitives (polygons, etc.)
  Each fragment corresponds to a single pixel

* Compositing Techniques : compositing, α-blending
  * How do we model transparent objects? —— the alpha channel

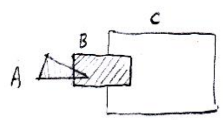
    RGBA (RGBα) color : (r, g, b, α)

    $\alpha \rightarrow$ opacity : 1 opaque ↕ 0 transparent

    (transparency $= 1 - \alpha$)

  * α value controls how the RGB values are written to the frame buffer

    e.g.



    B is opaque, blocking C in the overlapped portion.
    A is transparent, the portion overlapped with B is blended with the color of B (blending the colors of A & B)

    * Many fragments, each coming from a different object, may correspond to the same pixel $\Rightarrow$ each such fragment contributes to the color of the pixel;
      the final color of the pixel is obtained by (blending) the fragment colors;
      The corresponding objects are (blended) or (composited) together

* When a polygon is processed, pixel-size fragments are computed.
  The fragments are assigned colors based on the shading model used

  Regard the fragment as the (source pixel)
  the frame-buffer pixel as the (destination pixel)

  Previously : z-buffer, opaque : source pixel is closer to viewer $\Rightarrow$ source pixel (replaces) the destination pixel

  destination pixel : : $\Rightarrow$ source pixel is (blocked) no action.

  Now : blend the source and destination pixels in various ways color of source pixel: $s = [s_r \ s_g \ s_b \ s_a]$ source blending factor $b = [b_r \ b_g \ b_b \ b_a]$

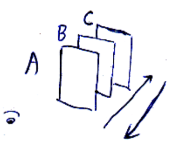
destination: $d = [d_r \ d_g \ d_b \ d_a]$, destination: $c = [c_r \ c_g \ c_b \ c_a]$

* $\boxed{d' \leftarrow bs + cd.}$

compositing: replace d with $d' = [b_r s_r + c_r d_r \quad b_g s_g + c_g d_g \quad b_b s_b + c_b d_b \quad b_a s_a + c_a d_a]$

the resulting r, g, b, a values are clamped to $[0.0, 1.0]$ $\begin{pmatrix} \geq 1 \Rightarrow 1.0 \\ \leq 0 \Rightarrow 0.0 \end{pmatrix}$

"Over" operation: back-to-front.



$$\begin{cases} C_{d'} = \alpha_s C_s + (1-\alpha_s) C_d \\ \alpha_{d'} = \alpha_s + (1-\alpha_s)\alpha_d. \end{cases}$$

$(1-\alpha_s)$ → transparency. the fraction that the "behind color" survives

Depth Cueing and Fog

* Depth Cueing: create illusion of depth by drawing objects farther from the viewer dimmer

$\boxed{\text{See Handout for full details}}$ **

$(A \text{ over } (B \text{ over } C))$: back to front
$((A \text{ over } B) \text{ over } C)$: front to back.

* Fog Effect: extend depth cueing.

create the illusion of partially translucent space (fog) between the object and the viewer, by blending in a (distance-dependent color) as each fragment is processed

f: fog factor, given by the fog equation $f(z)$, $(f = f(z))$

C_s: fragment color

C_f: fog color

z: distance between a (fragment) being rendered and the (viewer) given in the (eye coordinates)

(** Note: The Handout for the ``Over'' operation has been posted at NYU Brightspace:
``Handouts -> Over-Op-Associativity.pdf'')

Resulting color : $C_{s'} = f\,C_s + (1-f)\,C_f$ — (*)

fog-mode	fog equation $f(z) = f$		
linear fog	$f = \dfrac{end - z}{end - start}$	linear, depth-cueing effect	
exponential fog	$f = e^{-(density \cdot z)}$	exponential	fog effect
exponential square fog	$f = e^{-(density \cdot z)^2}$	Gaussian	fog effect

* f specified is clamped to [0,1] and then used in (*) to compute $C_{s'}$

o From fog equation:
$z \uparrow$ $f \downarrow$ (f is clamped to [0,1])
Plugging f into (*)
$\Rightarrow$ C_f has more weight
i.e. when object is farther ($z \uparrow$)
we see more of the fog color (C_f).