Shading in OpenGL

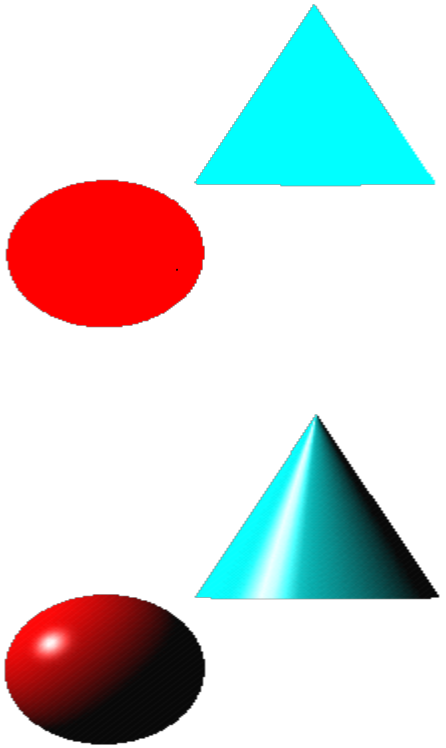
Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Introduce the OpenGL shading methods
  - per vertex vs per fragment shading
  - Where to carry out
- Discuss polygonal shading
  - Flat
  - Smooth
    - Gouraud shading
    - Phong shading
    - Blinn-Phong shading

Shading Principles

- Shading simulates how objects reflect light
  - material composition of object
  - light's color and position
  - global lighting parameters
- Usually implemented in
  - vertex shader for faster speed
    - Gouraud shading
  - fragment shader for nicer shading
    - Phong shading

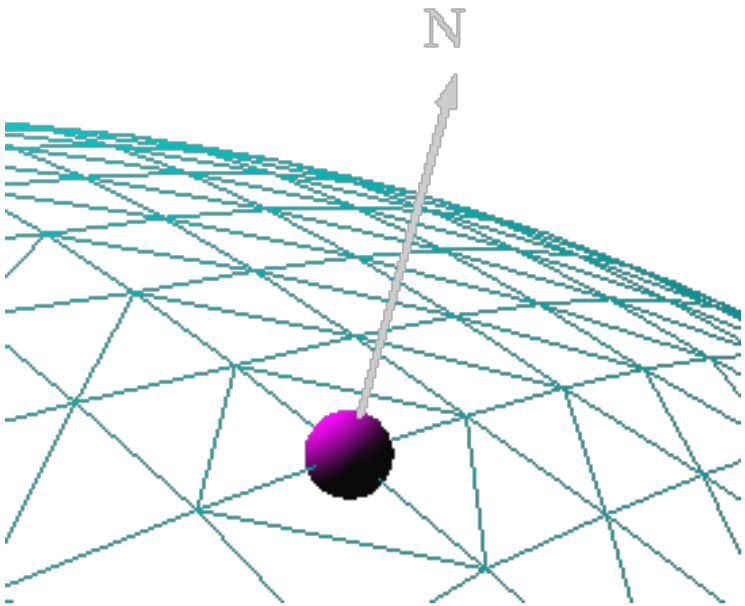


OpenGL shading

- Need to specify:
  - Normals
  - Material properties
  - Lights
- Get computed values in application or send attributes to shaders

Surface Normals

- Normals define how a surface reflects light
  - Application usually provides normals as a vertex attribute
  - Current normal is used to compute vertex's color
  - Use unit normals for proper lighting
    - scaling affects a normal's length

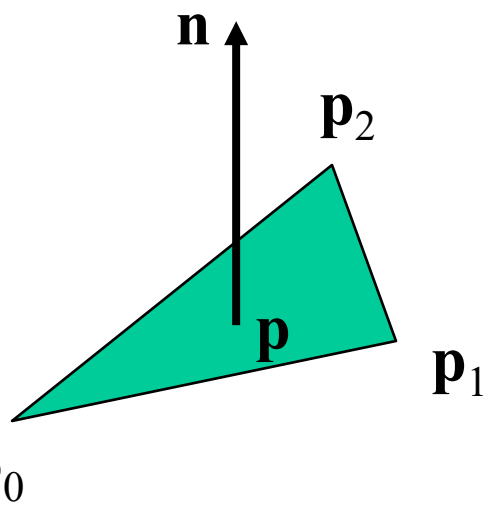


Normal for Triangle

plane $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$

$\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)$

normalize $\mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|$

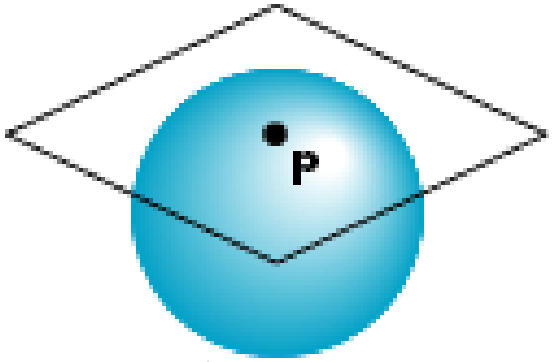


Note that right-hand rule determines outward face

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

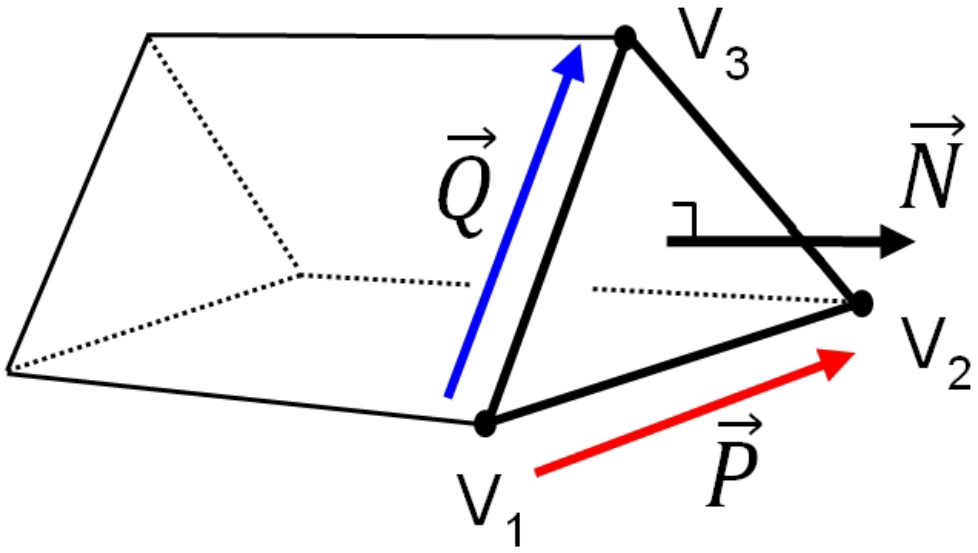
$$\partial \mathbf{p}/\partial u = [\partial x/\partial u, \partial y/\partial u, \partial z/\partial u]T$$
$$\partial \mathbf{p}/\partial v = [\partial x/\partial v, \partial y/\partial v, \partial z/\partial v]T$$

- Normal given by cross product

$$\mathbf{n} = \partial \mathbf{p}/\partial u \times \partial \mathbf{p}/\partial v$$

Cross Product

Useful for finding outward normal vectors



$$\vec{P} = V_2 - V_1$$
$$\vec{Q} = V_3 - V_1$$
$$\vec{N} = \vec{P} \times \vec{Q}$$

assuming P=(a,b,c) and Q=(d,e,f),

N = P x Q = (bf-ce, cd-af, ae-bd)

Specifying a Point Light Source

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

```
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);
```

- 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

Moving Light Sources

- Light sources are geometric objects whose positions or directions are affected by the model-view matrix
- Depending on where we place the position (direction) setting function, we can
  - 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 (1)

- Define the surface properties of a primitive

Property	Description
Diffuse	Base object color
Specular	Highlight color
Ambient	Low-light color
Emission	Glow color
Shininess	Surface smoothness

- you can have separate materials for front and back

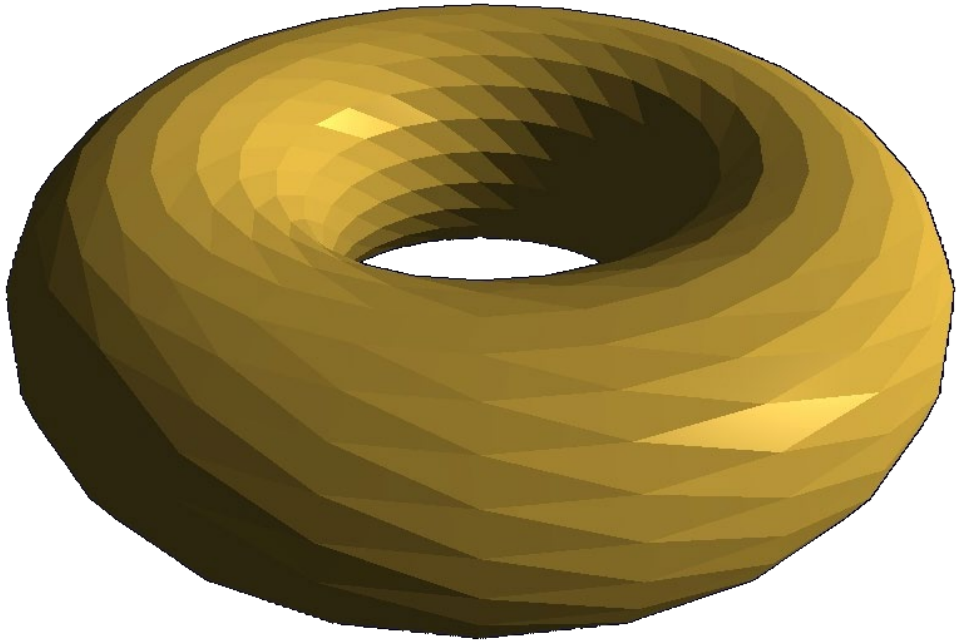
Material Properties (2)

- Material properties should match the terms in the light model
- Specifies amount of reflected light
  - An object appears red because it reflects red component of light
- w component gives opacity

```
vec4 ambient  = vec4(0.2, 0.2, 0.2, 1.0);
vec4 diffuse  = vec4(1.0, 1.0, 1.0, 1.0);
GLfloat shine = 100.0
```

Flat Shading

- Use triangle normal across all fragments of triangle
- One color per triangular facet



Smooth Shading

- Set a new normal at each vertex
- Easy for sphere model
  - If centered at origin $\mathbf{n} = \mathbf{p}$
- Note *silhouette edges*

Smooth Shading Implementation

- Gouraud (1971)

  ADS computations done *per vertex*, then interpolated by the rasterizer.

- Phong (1973)

  Normal vectors interpolated by rasterizer, then ADS computations done *per fragment*.

- Blinn-Phong (1977)

  Same as Phong shading (above), with reflection computation slightly optimized for performance.

All require the presence of *surface normal* vectors

Gouraud Shading

- Computes a color for each vertex using
  - Surface normals
  - Diffuse and specular reflections
  - Viewer's position and viewing direction
  - Ambient light
  - Emission
- Vertex colors are interpolated across polygons by the rasterizer
  - *Phong shading* does the same computation per pixel, interpolating the normal across the polygon
    - more accurate results

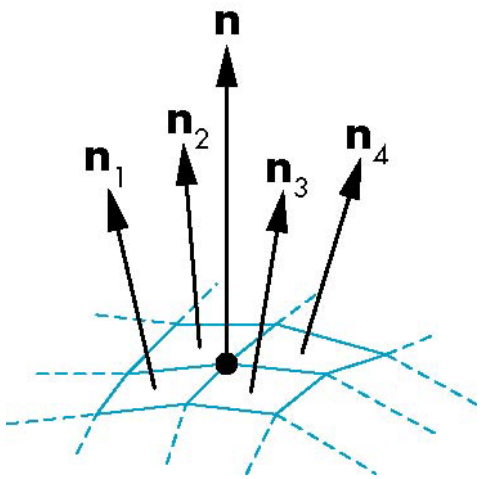
Polygonal Shading

- In per vertex shading, shading calculations are done for each vertex
  - Vertex colors become vertex shades and can be sent to the vertex shader as a vertex attribute
  - Alternately, we can send the parameters to the vertex shader and have it compute the shade
- By default, vertex shades are interpolated across an object if passed to the fragment shader as a varying variable (smooth shading)
- We can also use uniform variables to shade with a single shade (flat shading)

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 the normals around a mesh vertex

$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4)/\,|\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$



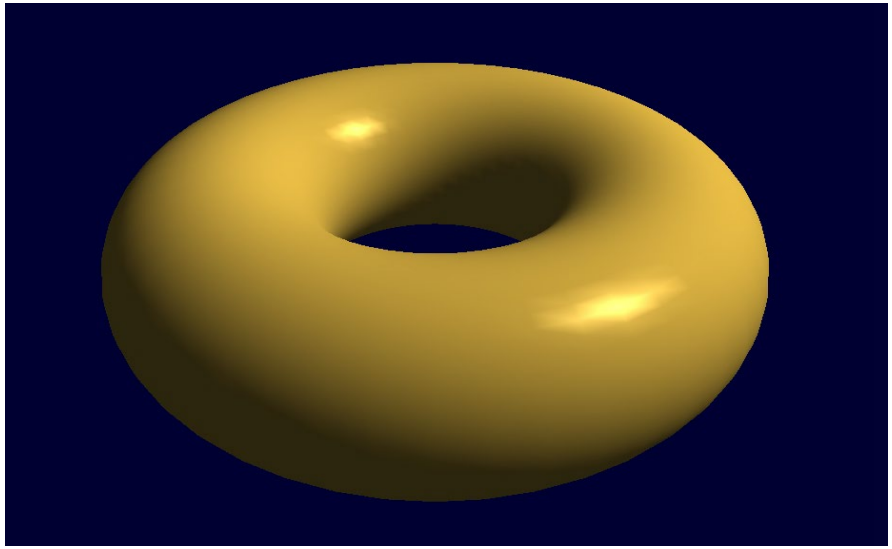
Gouraud and Phong Shading

- Gouraud Shading
  - Find average normal at each vertex (vertex normals)
  - Apply Phong illumination equation 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

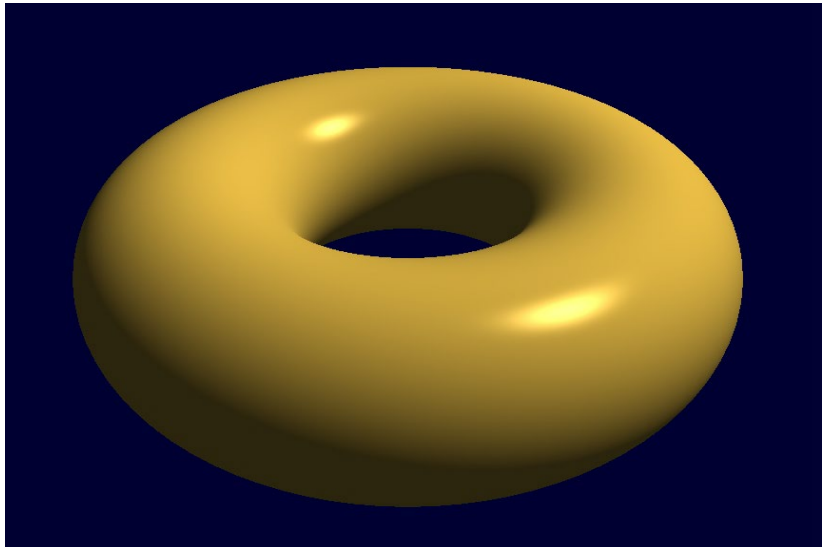
Comparison

- 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

Smooth Shading: Gouraud vs Phong



Gouraud

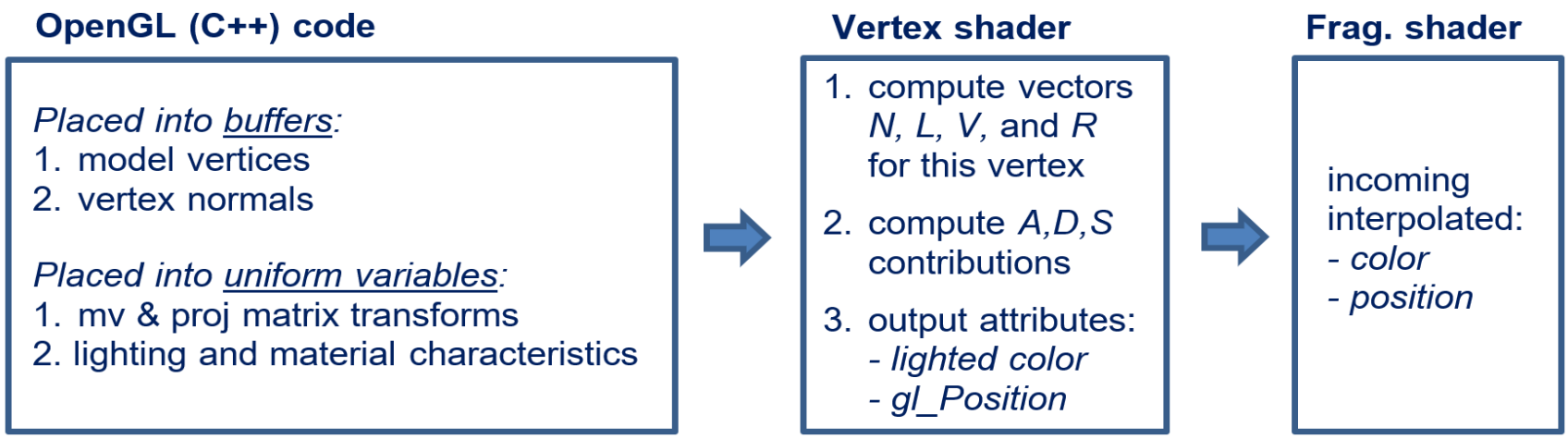


Phong

Normalization

- Cosine terms in lighting calculations can be computed using dot product
- Unit length vectors simplify calculation
- Usually we want to set the magnitudes to have unit length but
  - Length can be affected by transformations
  - Note that scaling does not preserve length
- GLSL has a normalization function

Gouraud Shading Implementation



Gouraud Shading: Vertex Shader(1)

```
// 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;
```

Gouraud Shading: Vertex Shader (2)

```
void
main()
{
    // transform vertex position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;

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

    // transform vertex normal into eye coordinates
    vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
```

Gouraud Shading: Vertex Shader (3)

```
    // compute terms in the illumination equation
    // ambient lighting term
    vec4 ambient = AmbientProduct;

    // diffuse lighting term
    float Kd = max( dot(L, N), 0.0 );
    vec4  diffuse = Kd*DiffuseProduct;

    // specular lighting term
    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);

    // add lighting terms to form color
    color = ambient + diffuse + specular;
    color.a = 1.0;

    gl_Position = Projection * ModelView * vPosition;
}
```

Gouraud Shading: Fragment Shader

```
// Fragment Shader
in vec4 color;

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

Gouraud Shading Implementation

C++/OpenGL application:

```
. . .
// initial light location
glm::vec3 initialLightLoc = glm::vec3(5.0f, 2.0f, 2.0f);
// properties of white light (global and positional) used in this scene
float globalAmbient[4] = { 0.7f, 0.7f, 0.7f, 1.0f };
float lightAmbient[4] = { 0.0f, 0.0f, 0.0f, 1.0f };
float lightDiffuse[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
float lightSpecular[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
// gold material properties
float* matAmb = Utils::goldAmbient();     // Utils.java file includes definition for Gold, Silver, Bronze
float* matDif = Utils::goldDiffuse();
float* matSpe = Utils::goldSpecular();
float matShi = Utils::goldShininess();
. . .
void setupVertices(void)  {
    . . .
    // load the torus normal vectors into the second buffer
    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
    glBufferData(GL_ARRAY_BUFFER, nvalues.size()*4, &nvalues[0], GL_STATIC_DRAW);
}
```

(continued…)

```
void display(GLFWwindow* window, double currentTime ) {
    // setup of model and view matrices and rendering program as in earlier examples.
    . . .
    // get uniforms for MV and projection (as before), plus matrix transform for normal vectors:
    nLoc = glGetUniformLocation(renderingProgram, "norm_matrix");
    // set up lights based on the current light's position
    currentLightPos = glm::vec3(initialLightLoc.x, initialLightLoc.y, initialLightLoc.z);
    installLights(vMat);
    . . .
    // mv matrix for normal vector is the inverse transpose of MV.
    invTrMat = glm::transpose(glm::inverse(mvMat));
    . . .
    // put the matrices into corresponding uniforms, now including the inverse transpose for normals:
    glUniformMatrix4fv(nLoc, 1, GL_FALSE, glm::value_ptr(invTrMat));
    . . .
    // bind the vertices buffer to vertex attribute #0 in the vertex shader (as before)
    . . .
    // bind vertices buffers to vertex attributes, now including the normals buffer (to vertex attribute #1):
    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
    glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, 0);
    glEnableVertexAttribArray(1);
    . . .
    glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);
}
```

(continued…)

```
void installLights(glm::mat4 vMatrix)  {
    // convert light's position to view space,
    // and save it in a float array in preparation for sending to the vertex shader
    lightPosV = glm::vec3(vMatrix * glm::vec4(currentLightPos, 1.0));
    lightPos[0] = lightPosV.x;
    lightPos[1] = lightPosV.y;
    lightPos[2] = lightPosV.z;

    // get the locations of the light and material fields in the shader
    globalAmbLoc = glGetUniformLocation(renderingProgram, "globalAmbient");
    ambLoc = glGetUniformLocation(renderingProgram, "light.ambient");
    posLoc = glGetUniformLocation(renderingProgram, "light.position");
    // . . . etc. for diffuse, specular, and position – and for material components

    //  then set the uniform light and material values in the shader
    glProgramUniform4fv(renderingProgram, globalAmbLoc, 1, globalAmbient);
    glProgramUniform4fv(renderingProgram, ambLoc, 1, lightAmbient);
    glProgramUniform4fv(renderingProgram, posLoc, 1, lightPos);
    // . . . etc. for the remaining uniforms
}
```

(continued…)

Vertex Shader

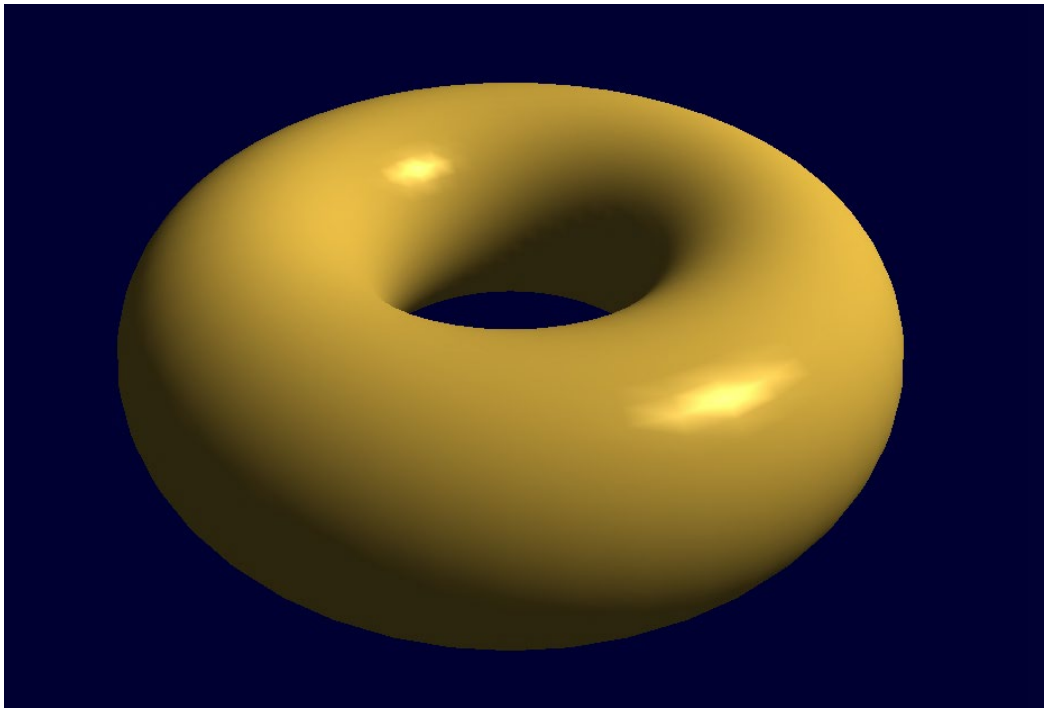
```
#version 430
layout (location=0) in vec3 vertPos;
layout (location=1) in vec3 vertNormal;
out vec4 varyingColor;
struct PositionalLight
{   vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec3 position;
};
struct Material
{   vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    float shininess;
};
uniform vec4 globalAmbient;
uniform PositionalLight light;
uniform Material material;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;
```

(continued…)

```
. . .
void main(void)
{   vec4 color;
    // convert vertex position to view space
    // convert normal to view space
    // calculate view space light vector (from vertex to light)
    vec4 P = mv_matrix * vec4(vertPos,1.0);
    vec3 N = normalize((norm_matrix * vec4(vertNormal,1.0)).xyz);
    vec3 L = normalize(light.position - P.xyz);
    // view vector is equivalent to the negative of view space vertex position
    vec3 V = normalize(-P.xyz);
    //  R is reflection of -L with respect to surface normal N
    vec3 R = reflect(-L, N);
    // ambient, diffuse, and specular contributions
    vec3 ambient = ((globalAmbient * material.ambient)
                        + (light.ambient * material.ambient)).xyz;
    vec3 diffuse = light.diffuse.xyz * material.diffuse.xyz * max(dot(N,L), 0.0);
    vec3 specular = material.specular.xyz * light.specular.xyz
                        * pow(max(dot(R,V), 0.0f), material.shininess);
    // send the color output to the fragment shader
    varyingColor = vec4((ambient + diffuse + specular), 1.0);
    // send the position to the fragment shader, as before
    gl_Position = proj_matrix * mv_matrix * vec4(vertPos,1.0);
}
```

Fragment Shader

```
#version 430
in vec4 varyingColor;
out vec4 fragColor;
. . .
// uniform declarations identical to those in the vertex shader (not shown here)
. . .
// interpolate lighted color (interpolation of gl_Position is automatic)
void main(void) {
    fragColor = varyingColor;
}
```



Phong Shading Implementation

OpenGL (C++) code

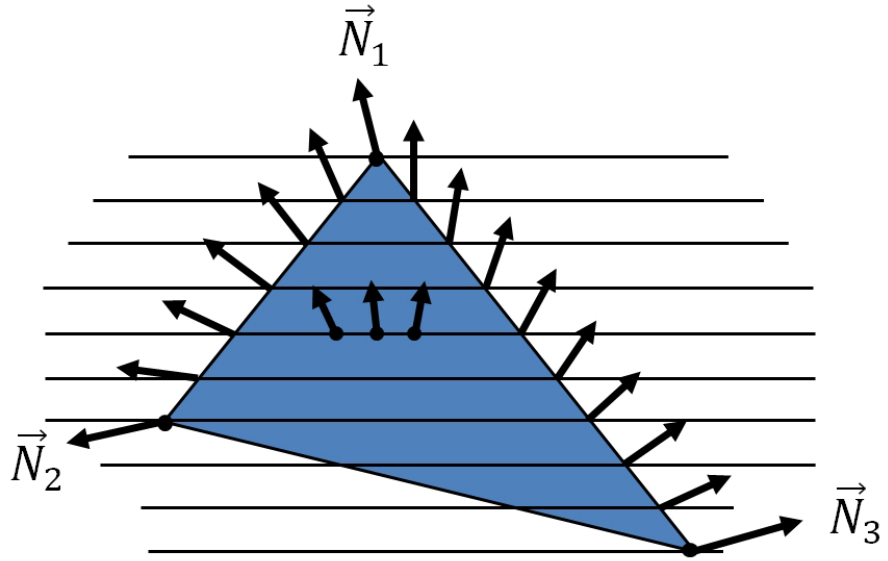
(same as for Gouraud shading)

Vertex shader

1. compute: vectors *N,L,V*
2. output attributes: *N, L, V, gl_Position*

Fragment shader

1. incoming interpolated: *N, L, V*
2. compute: *R, θ, φ*
3. compute ADS contributions
4. output color



Interpolation of normal vectors

Phong Shading: Vertex Shader (1)

```
// Vertex Shader
in vec4 vPosition;
in vec3 vNormal;

// output values that will be interpolatated per-fragment
out vec3 fN;
out vec3 fV;
out vec3 fL;

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;
```

Phong Shading: Vertex Shader (2)

```
void main()
{
    fN = vNormal;
    fV = vPosition.xyz;
    fL = LightPosition.xyz;

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

    gl_Position = Projection*ModelView*vPosition;
}
```

Phong Shading: Fragment Shader (1)

```
// Fragment Shader

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

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 V = normalize(fV);
    vec3 L = normalize(fL);
```

Phong Shading: Fragment Shader (2)

```
    vec3 H = normalize( L + V );
    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
    // is 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;
}
```

Phong Shading Implementation

Vertex Shader

```
#version 430
layout (location=0) in vec3 vertPos;
layout (location=1) in vec3 vertNormal;
out vec3 varyingNormal;        // eye-space vertex normal
out vec3 varyingLightDir;      // vector pointing to the light
out vec3 varyingVertPos;       // vertex position in eye space
. . .
// structs and uniforms same as for Gouraud shading
. . .
void main(void)
{   // output vertex position, light direction, and normal to the rasterizer for interpolation
    varyingVertPos=(mv_matrix * vec4(vertPos,1.0)).xyz;
    varyingLightDir = light.position - varyingVertPos;
    varyingNormal=(norm_matrix * vec4(vertNormal,1.0)).xyz;
    gl_Position=proj_matrix * mv_matrix * vec4(vertPos,1.0);
}
```

Fragment Shader

```
. . .
// inputs correspond to outputs of fragment shader.
// structs and uniforms same as for Gouraud shading.
```

```
. . .
void main(void)
{   vec3 L = normalize(varyingLightDir);
    vec3 N = normalize(varyingNormal);
    vec3 V = normalize(-varyingVertPos);
    // compute light reflection vector with respect to N:
    vec3 R = normalize(reflect(-L, N));
    // get the angle between the light and surface normal:
    float cosTheta = dot(L,N);
    // angle between the view vector and reflected light:
    float cosPhi = dot(V,R);
    // compute ADS contributions (per pixel), and combine to build output color:
    vec3 ambient = ((globalAmbient * material.ambient)
                        + (light.ambient * material.ambient)).xyz;
    vec3 diffuse = light.diffuse.xyz * material.diffuse.xyz * max(cosTheta,0.0);
    vec3 specular = light.specular.xyz * material.specular.xyz
                        * pow(max(cosPhi,0.0), material.shininess);
    fragColor = vec4((ambient + diffuse + specular), 1.0);
}
```

Blinn-Phong Implementation

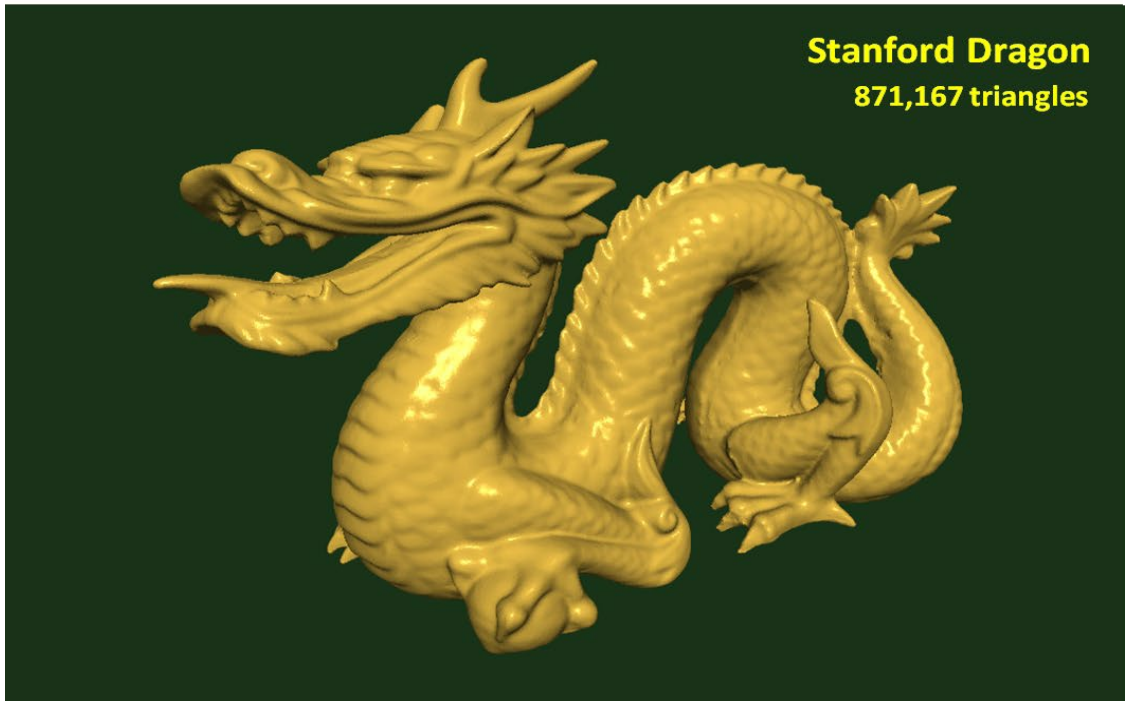
Vertex Shader

```
. . .
out vec3 varyingHalfVector;
. . .
void main(void)
{  . . .
   varyingHalfVector = (varyingLightDir + (-varyingVertPos)).xyz;
   . . .
}
```

Fragment Shader

```
. . .
in vec3 varyingHalfVector;
. . .
void main(void)       // note: it is no longer necessary to compute R in the fragment shader
{  . . .
   float cosPhi = dot(normalize(varyingHalfVector), N);
   . . .
   vec3 specular = light.specular.xyz * material.specular.xyz
                        * pow(max(cosPhi,0.0), material.shininess*3.0);
   . . .
}
```


Studio 522 Dolphin
34,014 triangles


Stanford Dragon
871,167 triangles

Combining Light and Texture

fragColor = textureColor * (ambientLight + diffuseLight) + specularLight

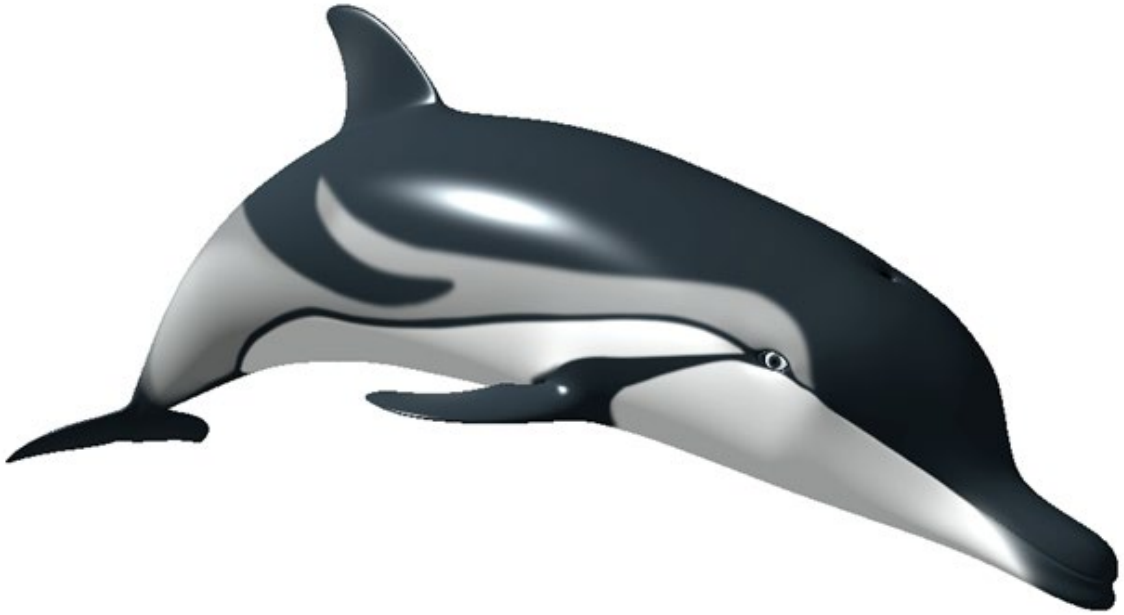
or

fragColor = textureColor * (ambientLight + diffuseLight + specularLight)

or

lightColor = (ambLight * ambMaterial) + (diffLight * diffMaterial) + specLight

fragColor = 0.5 * textureColor + 0.5 * lightColor



Fragment Shaders

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Fragment Shader

- A shader that's executed for each "potential" pixel
  - fragments still need to pass several tests before making it to the framebuffer
- There are lots of effects we can do in fragment shaders
  - Per-fragment lighting
  - Texture and bump Mapping
  - Environment (Reflection) Maps

Shader Example: Height Fields

- A height field is a function $y = f(x, z)$
  - y represents height of point for a location in the x-z plane
- Heights fields are usually rendered as a rectangular mesh of triangles or rectangles sampled from a grid
  - samples $y_{ij} = f(x_i, z_j)$

Displaying a Height Field

- First, generate a mesh data and use it to initialize data for a VBO

```
float dx = 1.0/N, dz = 1.0/N;
for( int i = 0; i < N; i++ ) {
    float x = i*dx;

    for( int j = 0; j < N; j++ ) {
        float z = j*dz;
        vertex[Index++] = vec3(      x, f(x,      z), z );
        vertex[Index++] = vec3(      x, f(x,   z+dz), z + dz );
        vertex[Index++] = vec3( x + dx, f(x+dx,z+dz), z + dz );
        vertex[Index++] = vec3( x + dx, f(x+dx,   z), z );
    }
}
```

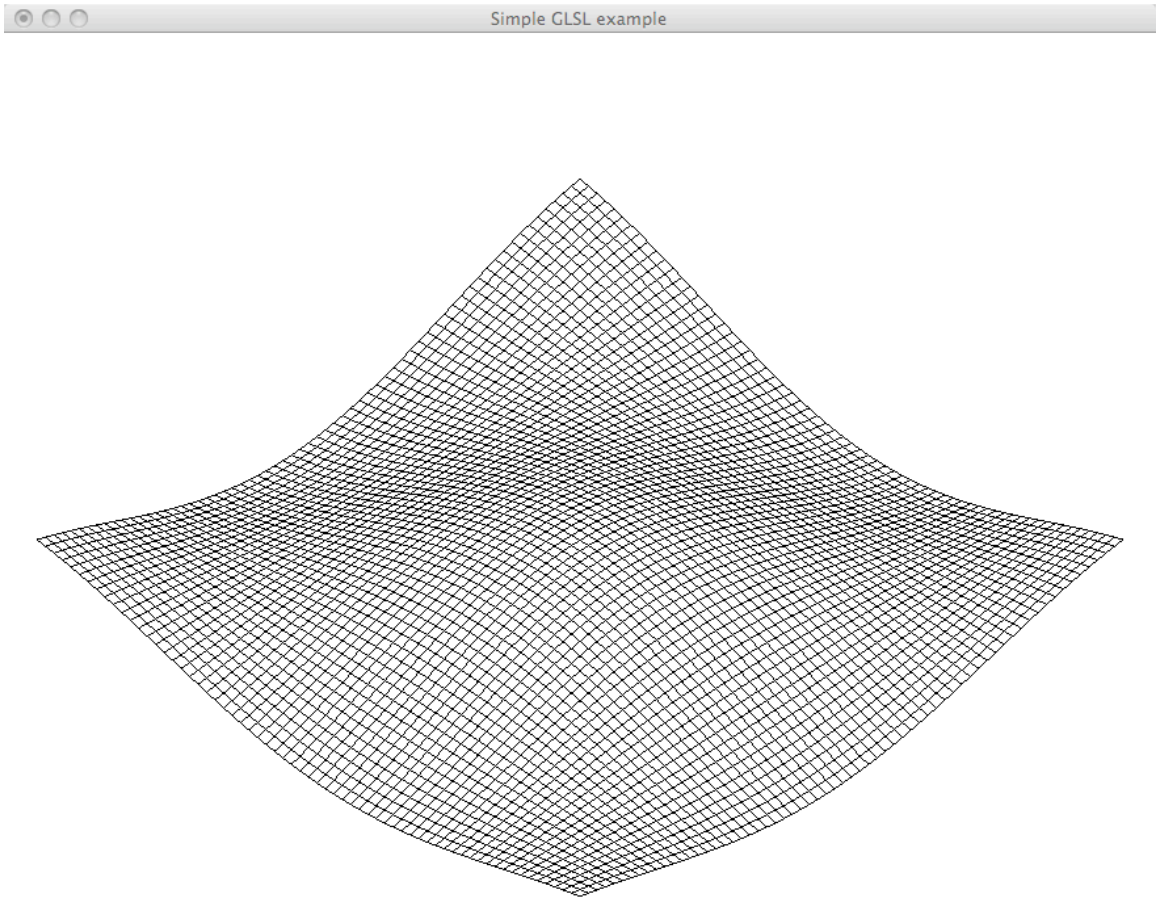
- Finally, display each quad using

```
for( int i = 0; i < NumVertices ; i += 4 )
    glDrawArrays( GL_LINE_LOOP, 4*i, 4 );
```

Time Varying Vertex Shader

```
in vec4 vPosition;
in vec4 vColor;

uniform float time; // in milliseconds
uniform mat4 ModelViewProjectionMatrix;

void main()
{
    vec4  v = vPosition;
    vec4  u = sin( time + 5*v );

    v.y = 0.1 * u.x * u.z;

    gl_Position = ModelViewProjectionMatrix * v;
}
```

Mesh Display



Adding Lighting

- Solid Mesh: create two triangles for each quad
- Display with

```
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

- For better looking results, add lighting
- We'll do per-vertex lighting
  - leverage the vertex shader since we'll also use it to vary the mesh in a time-varying way

Mesh Shader (1)

```
uniform float time, shininess;
uniform vec4 vPosition, lightPosition, diffuseLight, specularLight;
uniform mat4 ModelViewMatrix, ModelViewProjectionMatrix, NormalMatrix;

void main()
{
   vec4  v = vPosition;
   vec4  u = sin( time + 5*v );
   v.y = 0.1 * u.x * u.z;

   gl_Position = ModelViewProjectionMatrix * v;

   vec4 diffuse, specular;
   vec4 eyePosition = ModelViewMatrix * vPosition;
   vec4 eyeLightPos = lightPosition;
```

Mesh Shader (2)

```
    vec3 N = normalize(NormalMatrix * Normal);
    vec3 L = normalize(vec3(eyeLightPos – eyePosition));
    vec3 V = -normalize(eyePosition.xyz);
    vec3 H = normalize(L + E);

   float Kd = max(dot(L, N), 0.0);
   float Ks = pow(max(dot(N, H), 0.0), shininess);
   diffuse  = Kd*diffuseLight;
   specular = Ks*specularLight;
   color    = diffuse + specular;
}
```

Shaded Mesh