Implementation: Shading

Introduction to Computer Graphics

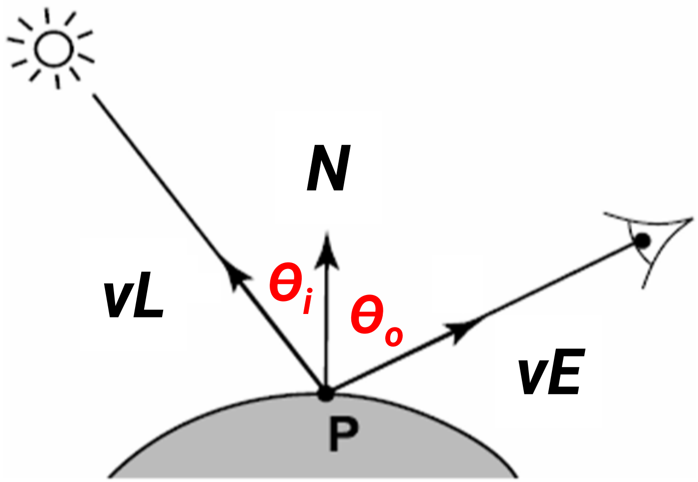
Yu-Ting Wu

Goals

- Introduce how to define **point/directional lights** and **object materials with the *Phong lighting model*** in an *OpenGL* program
- Introduce how to calculate **ambient** and **diffuse** lighting in the Vertex Shader in the fashion of ***Gouraud shading***

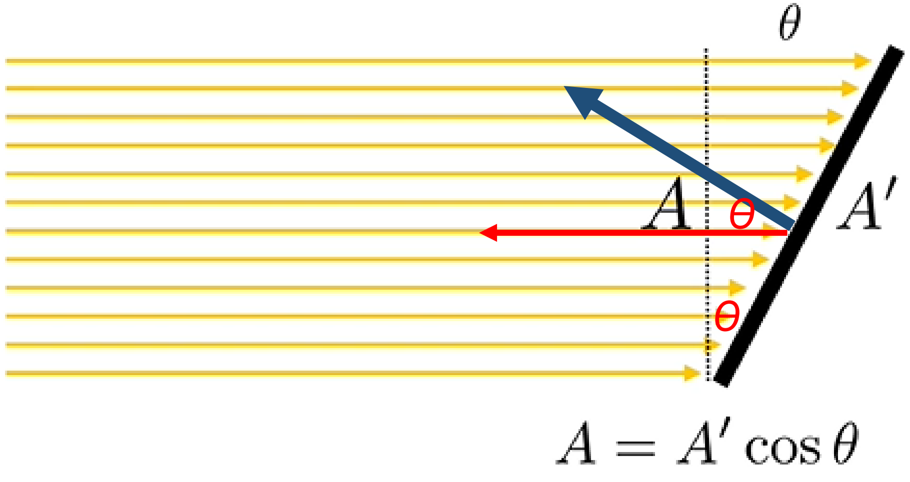
Recap: Shading

- Shading refers to the process of altering the color of an object/surface/polygon in the 3D scene
- In physically-based rendering, shading tries to approximate the **local behavior** of lights on the object's surface, based on things like
  - Surface orientation (normal) ***N***
  - Lighting direction ***vL*** (and θ_i)
  - Viewing direction ***vE*** (and θ_o)
  - Material properties
  - Participating media
  - etc.



Recap: Lambertian Cosine Law

- Illumination on an oblique surface is less than on a normal one
- Generally, illumination falls off as cosΘ

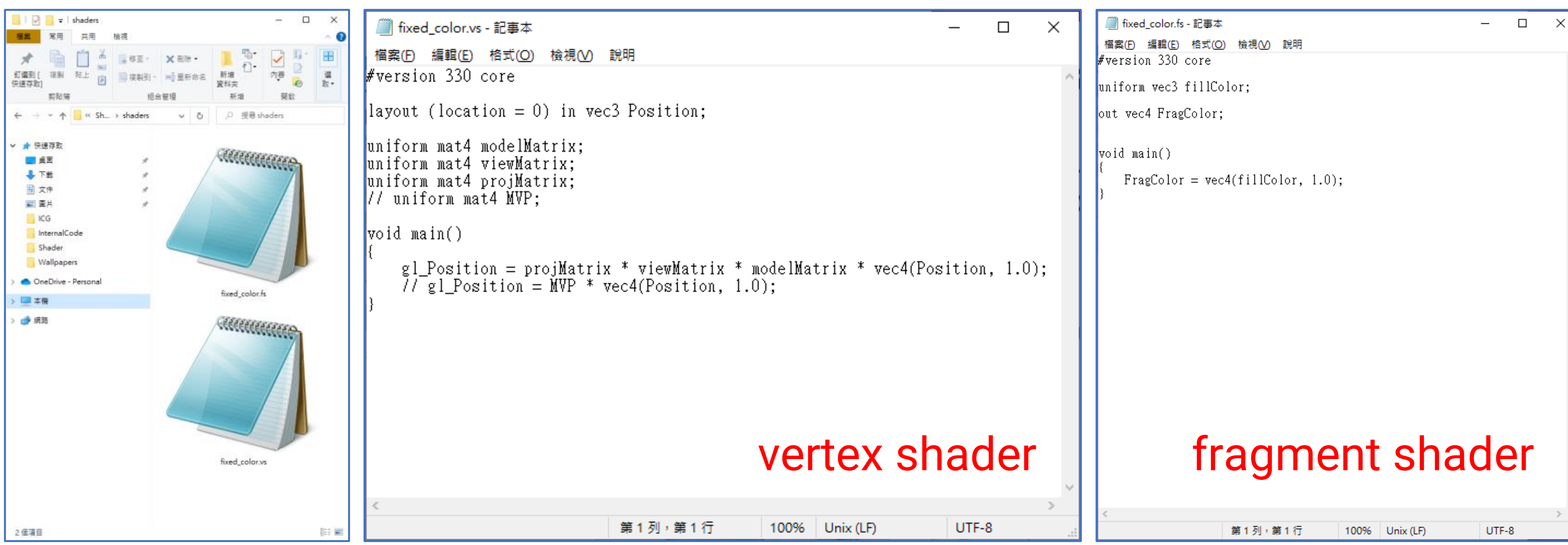


$$E = \frac{\Phi}{A'} = \frac{\Phi \cos\theta}{A}$$

Recap: Shaders

- Shaders: small C-like program that runs in a **per-vertex (Vertex Shader)** or **per-fragment (Fragment Shader)** manner **on the GPU in parallel**

the file extension does not matter!



Recap: Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 Position;
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projMatrix;

void main() {
    gl_Position = projMatrix * viewMatrix *
                  modelMatrix * vec4(Position, 1.0);
}
```

Vertex attribute

- **glEnableVertexAttribArray(0)**

uniform variables communicated with the CPU

- Get location by **glGetUniformLocation**
- Set value by **glUniformXXX**

the main program **executed per vertex**

gl_Position: built-in variable for the Clip Space coordinate

Recap: Fragment Shader

```
#version 330 core
uniform vec3 fillColor;
```

uniform variables communicated with the CPU

- Get location by **glGetUniformLocation**
- Set value by **glUniformXXX**

```
out vec4 FragColor;
```

Output: fragment data

the main program **executed per fragment**

```
void main() {
    FragColor = vec4(fillColor, 1.0);
}
```

Recap: Communicate with Shaders

```
locMVP = glGetUniformLocation(shaderProgId, "MVP");
glUniformMatrix4fv(locMVP, 1, GL_FALSE, glm::value_ptr(MVP));
```

CPU

1

2

Vertex Shader

GPU

```
#version 330 core
layout (location = 0) in vec3 Position;
uniform mat4 MVP;
void main() {
    gl_Position = MVP * vec4(Position, 1.0);
}
```

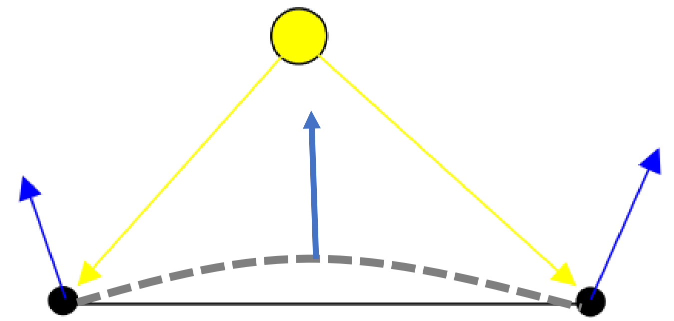
Implementation of Lighting and Shading

- Lighting and shading can be implemented either in the **vertex shader (compute per vertex and interpolate color)** or **fragment shader (interpolate vertex attributes and compute per fragment)**
- It can also be implemented in **all coordinate spaces**, such as world space or camera space

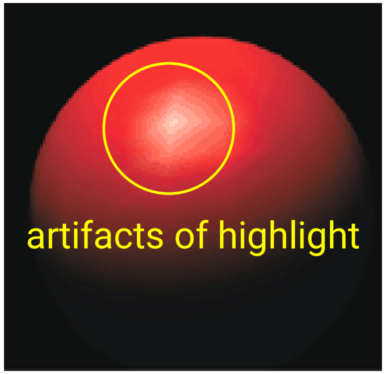
Recap: Gouraud and Phong Shading

- **Gouraud shading**: compute lighting at vertices and interpolate the lighting color
- **Phong shading**: interpolate normal and compute lighting

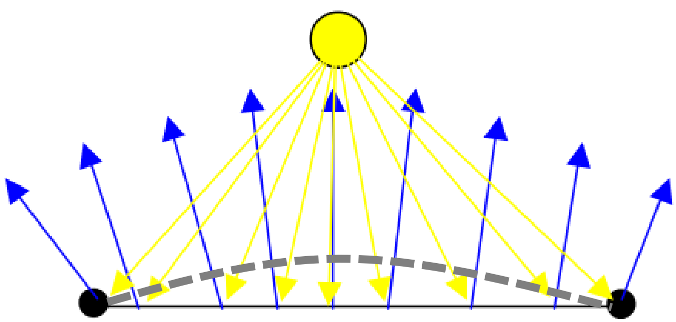
Gouraud shading

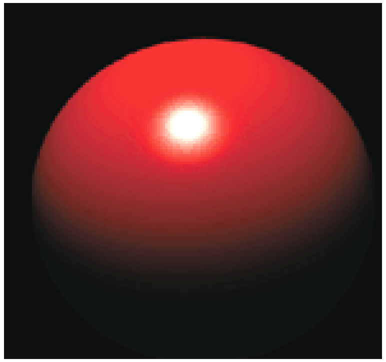


lighting color is interpolated

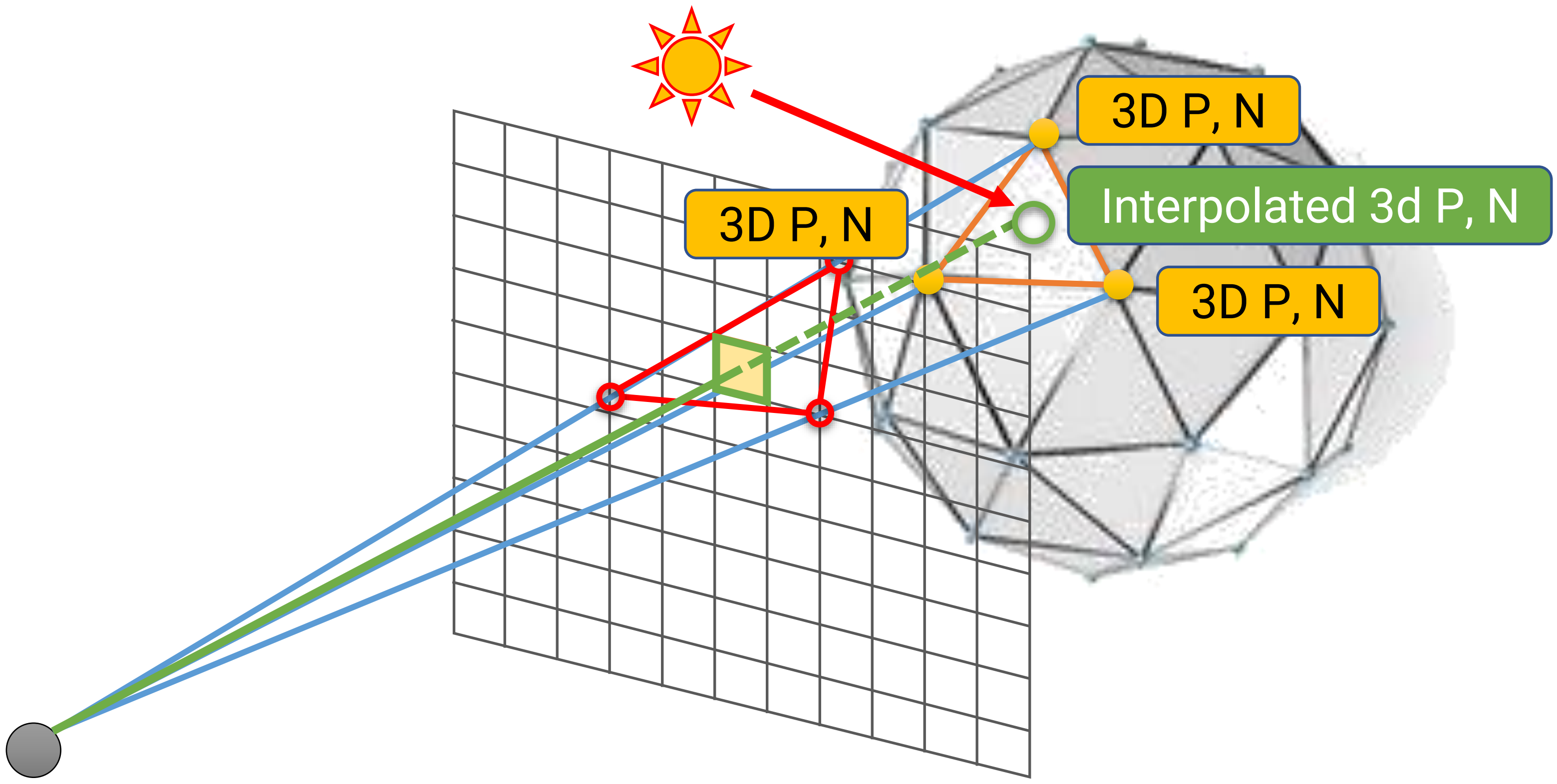


Phong shading





Recap: Gouraud and Phong Shading (cont.)



Recap: Vertex Attribute Interpolation

- Example: interpolate **world-space vertex position** and **world-space vertex normal**

Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 Position;
layout (location = 1) in vec3 Normal;

// Transformation matrix.
uniform mat4 worldMatrix;
uniform mat4 normalMatrix;
uniform mat4 MVP;

// Data pass to fragment shader.
out vec3 iPosWorld;
out vec3 iNormalWorld;

void main()
{
    gl_Position = MVP * vec4(Position, 1.0);

    // Pass vertex attributes.
    vec4 positionTmp = worldMatrix * vec4(Position, 1.0);
    iPosWorld = positionTmp.xyz / positionTmp.w;

    iNormalWorld = (normalMatrix * vec4(Normal, 0.0)).xyz;
}
```

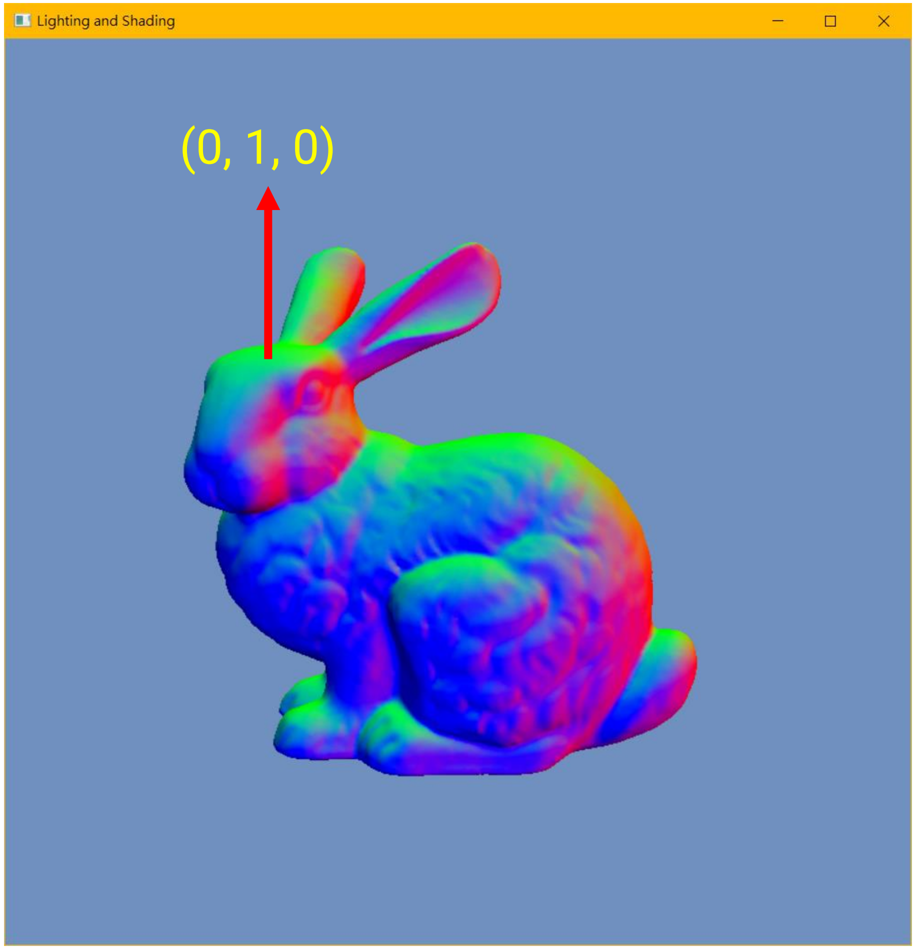
Tell OpenGL you want to interpolate these attributes

Fragment Shader

```
#version 330 core

// Data from vertex shader.
in vec3 iPosWorld;
in vec3 iNormalWorld;

out vec4 FragColor;

void main()
{
    vec3 N = normalize(iNormalWorld);
    vec3 visColor = 0.5 * N + 0.5;
    FragColor = vec4(N, 1.0);
}
```

Recap: Vertex Attribute Interpolation (cont.)

visualize world-space position as color



visualize world-space normal as color

Programs

Overview

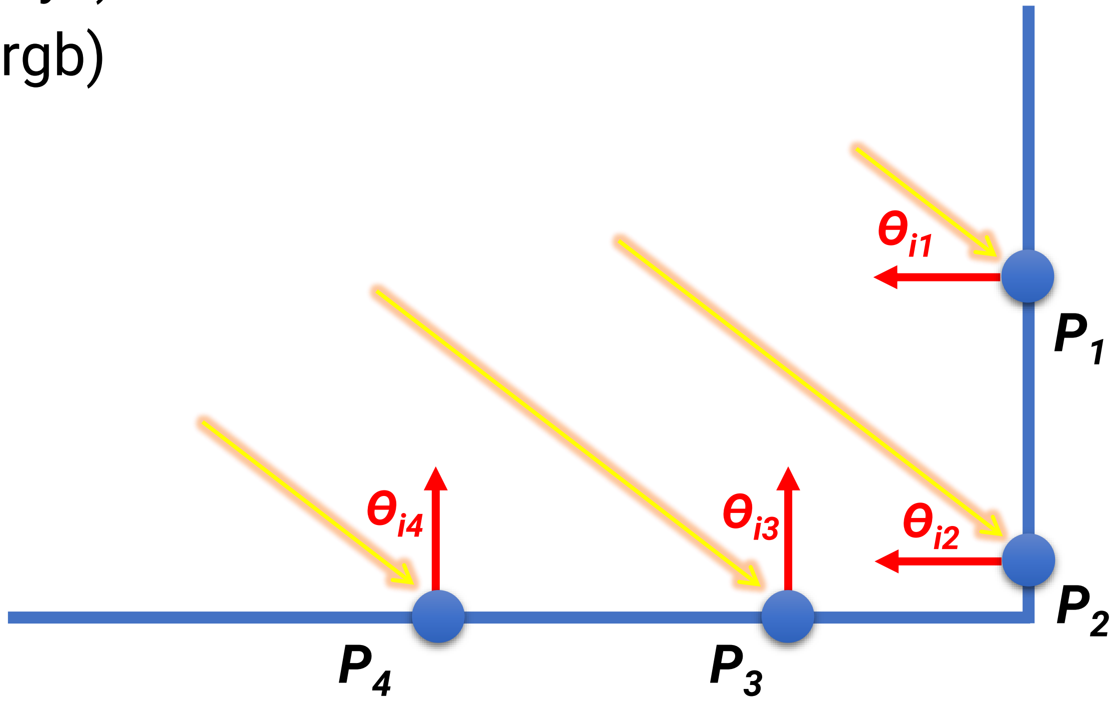
- The sample program implements **Gouraud shading** with a point light and a directional light in the Vertex Shader
- Only the diffuse and the ambient term are computed
  - Specular term is part of your homework assignment #2

Data Structure: Lights

- Defined in *light.h*
- Two types of lights implemented
    - Directional light
    - Point light

Recap: Directional Light

- Describes an emitter that deposits illumination from the **same direction** at every point in space
- Described by
  - Light direction (***D***, xyz)
  - Light radiance (***L***, rgb)

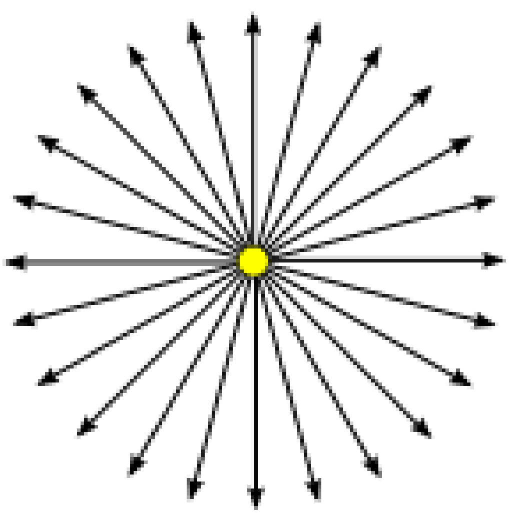


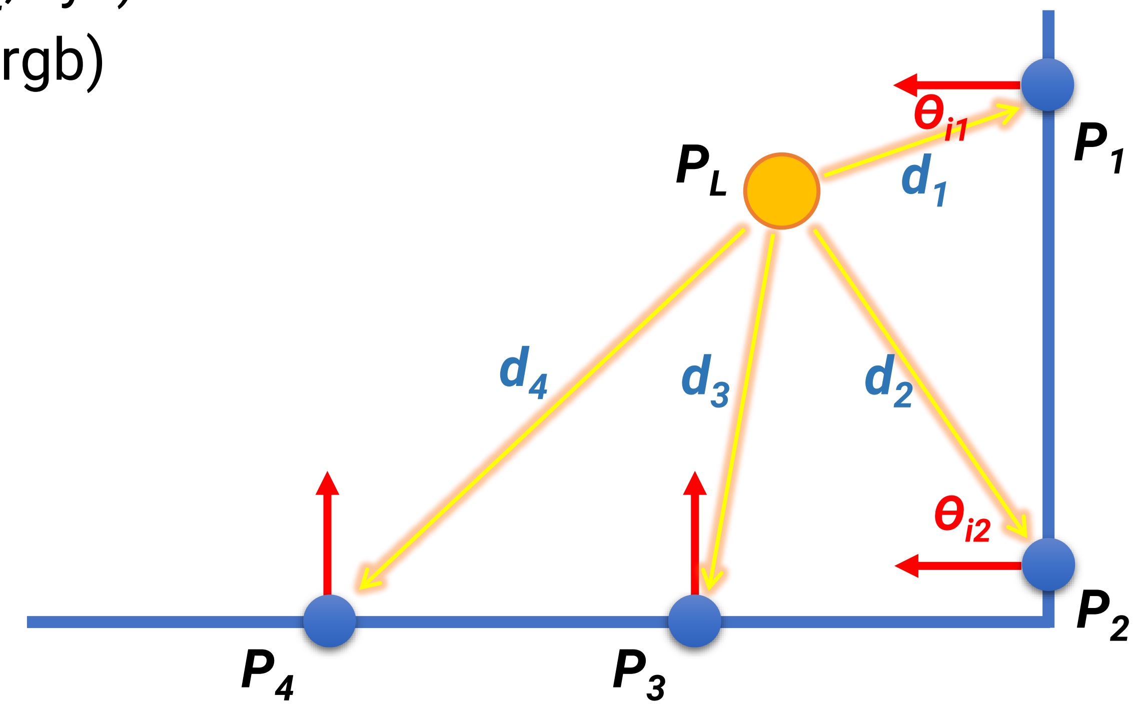
Data Structure: Directional Light

```cpp
// DirectionalLight Declarations.
class DirectionalLight
{
public:
    // DirectionalLight Public Methods.
    DirectionalLight() {
        direction = glm::vec3(1.5f, 1.5f, 1.5f);
        radiance = glm::vec3(1.0f, 1.0f, 1.0f);
    };
    DirectionalLight(const glm::vec3 dir, const glm::vec3 L) {
        direction = dir;
        radiance = L;
    }

    glm::vec3 GetDirection() const { return direction; }
    glm::vec3 GetRadiance()  const { return radiance;  }

private:
    // DirectionalLight Private Data.
    glm::vec3 direction;
    glm::vec3 radiance;
};
```

Recap: Point Light

- An isotropic point light source that emits the same amount of light in all directions
- Described by
  - Light position (***P_L***, xyz)
  - Light intensity (***I***, rgb)





Data Structure: Point Light

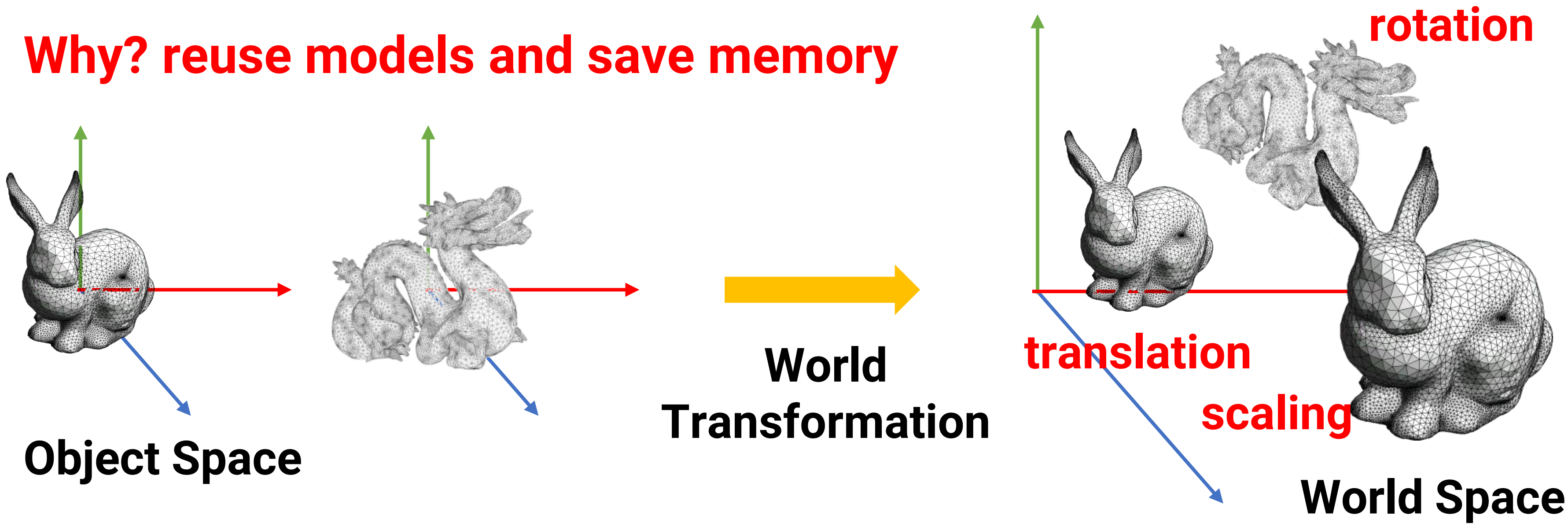
```
// PointLight Declarations.
class PointLight
{
public:
    // PointLight Public Methods.
    PointLight() {
        position = glm::vec3(1.5f, 1.5f, 1.5f);
        intensity = glm::vec3(1.0f, 1.0f, 1.0f);
        CreateVisGeometry();
    }
    PointLight(const glm::vec3 p, const glm::vec3 I) {
        position = p;
        intensity = I;
        CreateVisGeometry();
    }

    glm::vec3 GetPosition() const  { return position;  }
    glm::vec3 GetIntensity() const { return intensity; }

    void Draw() {
        glPointSize(16.0f);
        glEnableVertexAttribArray(0);
        glBindBuffer(GL_ARRAY_BUFFER, vboId);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexP), 0);
        glDrawArrays(GL_POINTS, 0, 1);
        glDisableVertexAttribArray(0);
        glPointSize(1.0f);
    }
```

Data Structure: Point Light (cont.)

```
    void MoveLeft (const float moveSpeed) { position += moveSpeed * glm::vec3(-0.1f,  0.0f, 0.0f); }
    void MoveRight(const float moveSpeed) { position += moveSpeed * glm::vec3( 0.1f,  0.0f, 0.0f); }
    void MoveUp   (const float moveSpeed) { position += moveSpeed * glm::vec3( 0.0f,  0.1f, 0.0f); }
    void MoveDown (const float moveSpeed) { position += moveSpeed * glm::vec3( 0.0f, -0.1f, 0.0f); }

private:
    // PointLight Private Methods.
    void CreateVisGeometry() {
        VertexP lightVtx = glm::vec3(0, 0, 0);
        const int numVertex = 1;
        glGenBuffers(1, &vboId);
        glBindBuffer(GL_ARRAY_BUFFER, vboId);
        glBufferData(GL_ARRAY_BUFFER, sizeof(VertexP) * numVertex, &lightVtx, GL_STATIC_DRAW);
    }

    // PointLight Private Data.
    GLuint vboId;
    glm::vec3 position;
    glm::vec3 intensity;
};

// VertexP Declarations.
struct VertexP
{
    VertexP() { position = glm::vec3(0.0f, 0.0f, 0.0f); }
    VertexP(glm::vec3 p) { position = p; }
    glm::vec3 position;
};
```

Recap: Object Space to World Space

Why? reuse models and save memory



$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translation

`glm::translate`

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

scaling

`glm::scale`

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation (Y)

`glm::rotate`

Data Structure: Scene Object

```
// SceneObject.
struct SceneObject
{
    SceneObject() {
        mesh = nullptr;
        worldMatrix = glm::mat4x4(1.0f);
        Ka = glm::vec3(0.3f, 0.3f, 0.3f);
        Kd = glm::vec3(0.7f, 0.7f, 0.7f);
        Ks = glm::vec3(0.6f, 0.6f, 0.6f);
        Ns = 50.0f;
    }
    TriangleMesh* mesh;
    glm::mat4x4 worldMatrix;
    // Material properties.
    glm::vec3 Ka;  ambient coefficient
    glm::vec3 Kd;  diffuse coefficient
    glm::vec3 Ks;  specular coefficient
    float Ns;  specular exponent (roughness)
};
```

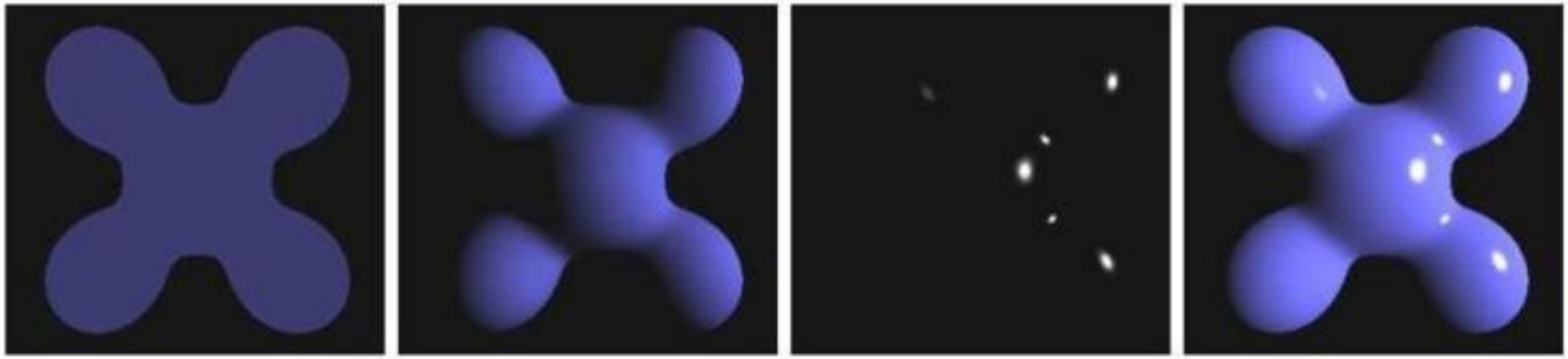
```
// ScenePointLight (for visualization of a point light).
struct ScenePointLight
{
    ScenePointLight() {
        light = nullptr;
        worldMatrix = glm::mat4x4(1.0f);
        visColor = glm::vec3(1.0f, 1.0f, 1.0f);
    }
    PointLight* light;
    glm::mat4x4 worldMatrix;
    glm::vec3 visColor;
};
```

Data Structure: Shaders

- Defined in *shaderprog.h / shaderprog.cpp*
- Add class ***"GouraudShadingDemoShaderProg"***
- Add shaders
  - Vertex shader: ***"gouraud_shading_demo.vs"***
  - Fragment shader: ***"gouraud_shading_demo.fs"***

Recap: Phong Lighting Model

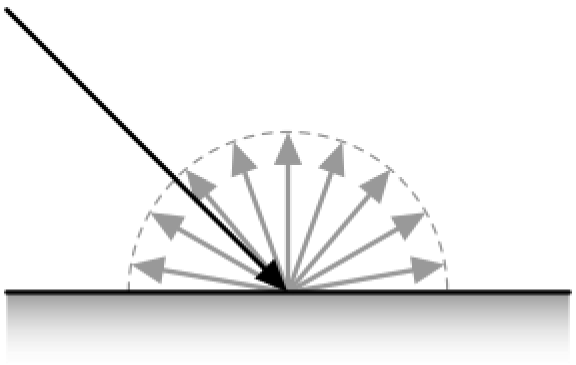
- **Diffuse reflection**
  - Light goes everywhere; colored by object color
- **Specular reflection**
  - Happens only near mirror configuration; usually white
- **Ambient reflection**
  - Constant accounted for global illumination (cheap hack)



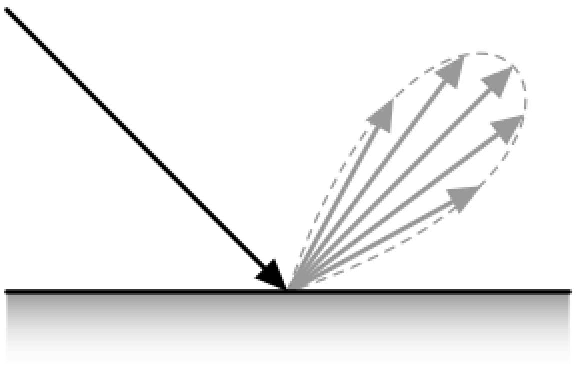
ambient diffuse specular

Recap: Material Property

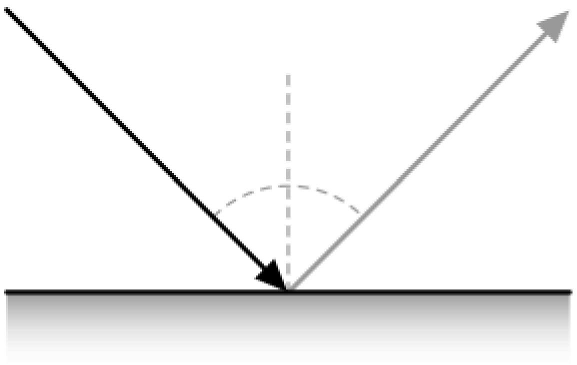
- Highly related to surface types
- The **smoother** a surface, the more reflected light is concentrated in the direction a **perfect mirror** would reflect the light



diffuse



glossy



specular

Data Structure: Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 Position;
layout (location = 1) in vec3 Normal;

// Transformation matrices.
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 normalMatrix;
uniform mat4 MVP;
```

Data Structure: Vertex Shader (cont.)

```
// Material properties.
uniform vec3 Ka;
uniform vec3 Kd;
uniform vec3 Ks;
uniform float Ns;
// Light data
uniform vec3 ambientLight;
uniform vec3 dirLightDir;
uniform vec3 dirLightRadiance;
uniform vec3 pointLightPos;
uniform vec3 pointLightIntensity;
```

Data Structure: Vertex Shader (cont.)

```
// Data pass to fragment shader
out vec3 iLightingColor;

void main() {
    gl_Position = MVP * vec4(Position, 1.0);

    // Compute vertex lighting in view space.
    vec4 tmpPos = viewMatrix * worldMatrix * vec4(Position, 1.0);
    vec3 vsPosition = tmpPos.xyz / tmpPos.w;
    vec3 vsNormal = (normalMatrix * vec4(Normal, 0.0)).xyz;
    vsNormal = normalize(vsNormal);
```

Data Structure: Vertex Shader (cont.)

```
// ---------------------------------------------------------
// Ambient light.
vec3 ambient = Ka * ambientLight;
// ---------------------------------------------------------
// Directional light.
vec3 vsLightDir = (viewMatrix * vec4(-dirLightDir, 0.0)).xyz;
// Diffuse and Specular.
vec3 diffuse =
        Diffuse(Kd, dirLightRadiance, vsNormal, vsLightDir);
vec3 specular = Specular();
vec3 dirLight = diffuse + specular;
```

Data Structure: Vertex Shader (cont.)

```
// Point light.
tmpPos = viewMatrix * vec4(pointLightPos, 1.0);
vec3 vsLightPos = tmpPos.xyz / tmpPos.w;
vsLightDir = normalize(vsLightPos - vsPosition);
float distSurfaceToLight = distance(vsLightPos, vsPosition);
float attenuation = 1.0f / (distSurfaceToLight * distSurfaceToLight);
vec3 radiance = pointLightIntensity * attenuation;
// Diffuse and Specular.
diffuse = Diffuse(Kd, radiance, vsNormal, vsLightDir);
specular = Specular();
vec3 pointLight = diffuse + specular;
```

Data Structure: Vertex Shader (cont.)

```
    // -----------------------------------------------------
    iLightingColor = ambient + dirLight + pointLight;
}

vec3 Diffuse(vec3 Kd, vec3 I, vec3 N, vec3 lightDir) {
    return Kd * I * max(0, dot(N, lightDir));
}
vec3 Specular( /* Put the parameters here. */ ) {
    // Try to implement yourself!
    return vec3(0.0, 0.0, 0.0);
}
```

Data Structure: Shaders (cont.)

- *"GouraudShadingDemoShaderProg.h"*

```
// GouraudShadingDemoShaderProg Declarations.
class GouraudShadingDemoShaderProg : public ShaderProg
{
public:
    // GouraudShadingDemoShaderProg Public Methods.
    GouraudShadingDemoShaderProg();
    ~GouraudShadingDemoShaderProg();

    GLint GetLocM() const { return locM; }
    GLint GetLocV() const { return locV; }
    GLint GetLocNM() const { return locNM; }
    GLint GetLocKa() const { return locKa; }
    GLint GetLocKd() const { return locKd; }
    GLint GetLocKs() const { return locKs; }
    GLint GetLocNs() const { return locNs; }
    GLint GetLocAmbientLight() const { return locAmbientLight; }
    GLint GetLocDirLightDir() const { return locDirLightDir; }
    GLint GetLocDirLightRadiance() const { return locDirLightRadiance; }
    GLint GetLocPointLightPos() const { return locPointLightPos; }
    GLint GetLocPointLightIntensity() const { return locPointLightIntensity; }
```

locations of uniform matrix variables

locations of uniform material variables

locations of uniform light data variables

Data Structure: Shaders (cont.)

```
protected:
    // GouraudShadingDemoShaderProg Protected Methods.
    void GetUniformVariableLocation();
```
override from the base class

```
private:
    // GouraudShadingDemoShaderProg Public Data.
    // Transformation matrix.
    GLint locM;
    GLint locV;
    GLint locNM;
    // Material properties.
    GLint locKa;
    GLint locKd;
    GLint locKs;
    GLint locNs;
    // Light data.
    GLint locAmbientLight;
    GLint locDirLightDir;
    GLint locDirLightRadiance;
    GLint locPointLightPos;
    GLint locPointLightIntensity;
};
```

Data Structure: Shaders (cont.)

- *"GouraudShadingDemoShaderProg.cpp"*

```
GouraudShadingDemoShaderProg::GouraudShadingDemoShaderProg()
{
    locM = -1;
    locV = -1;
    locNM = -1;
    locKa = -1;
    locKd = -1;
    locKs = -1;
    locNs = -1;
    locAmbientLight = -1;
    locDirLightDir = -1;
    locDirLightRadiance = -1;
    locPointLightPos = -1;
    locPointLightIntensity = -1;
}

GouraudShadingDemoShaderProg::~GouraudShadingDemoShaderProg()
{}
```

Data Structure: Shaders (cont.)

- *“GouraudShadingDemoShaderProg.cpp”*

```
void GouraudShadingDemoShaderProg::GetUniformVariableLocation()
{
    ShaderProg::GetUniformVariableLocation();
    locM = glGetUniformLocation(shaderProgId, "worldMatrix");
    locV = glGetUniformLocation(shaderProgId, "viewMatrix");
    locNM = glGetUniformLocation(shaderProgId, "normalMatrix");
    locKa = glGetUniformLocation(shaderProgId, "Ka");
    locKd = glGetUniformLocation(shaderProgId, "Kd");
    locKs = glGetUniformLocation(shaderProgId, "Ks");
    locNs = glGetUniformLocation(shaderProgId, "Ns");
    locAmbientLight = glGetUniformLocation(shaderProgId, "ambientLight");
    locDirLightDir = glGetUniformLocation(shaderProgId, "dirLightDir");
    locDirLightRadiance = glGetUniformLocation(shaderProgId, "dirLightRadiance");
    locPointLightPos = glGetUniformLocation(shaderProgId, "pointLightPos");
    locPointLightIntensity = glGetUniformLocation(shaderProgId, "pointLightIntensity");
}
```

Data Structure: Main Program

- The flow of the main program remains the same

```
int main(int argc, char** argv)
{
    // Setting window properties.
```

Initialize window properties and GLEW

```
    // Initialization.
    SetupRenderState();
    SetupScene();
    CreateShaderLib();

    // Register callback functions.
```

Register callback functions

```
    // Start rendering loop.
    glutMainLoop();

    return 0;
}
```

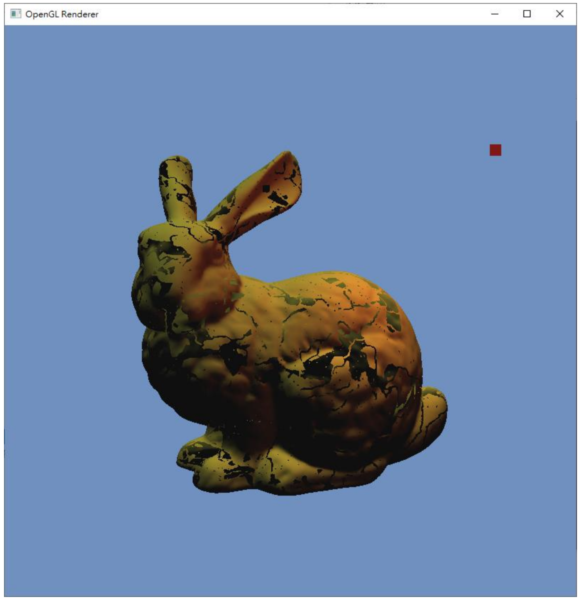
Data Structure: Main Program (cont.)

- Remember to enable ***"depth test"*** by calling

  *glEnable(**GL_DEPTH_TEST**);*

  Otherwise, the Z-buffer will not work

```
void SetupRenderState()
{
    // glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glEnable(GL_DEPTH_TEST);

    glm::vec4 clearColor = glm::vec4(0.44f, 0.57f, 0.75f, 1.00f);
    glClearColor(
        (GLclampf)(clearColor.r),
        (GLclampf)(clearColor.g),
        (GLclampf)(clearColor.b),
        (GLclampf)(clearColor.a)
    );
}
```



Data Structure: Main Program (cont.)

```
void SetupScene()
{
    // Scene object --------------------------------------------------------
    mesh = new TriangleMesh();
    mesh->LoadFromFile("models/Bunny/Bunny.obj", true);
    mesh->CreateBuffers();
    mesh->ShowInfo();
    sceneObj.mesh = mesh;

    // Scene lights --------------------------------------------------------
    // Create a directional light.
    dirLight = new DirectionalLight(dirLightDirection, dirLightRadiance);
    // Create a point light.
    pointLight = new PointLight(pointLightPosition, pointLightIntensity);
    pointLightObj.light = pointLight;
    pointLightObj.visColor = ((PointLight*)pointLightObj.light)->GetIntensity();

    // Create a camera and update view and proj matrices.
    camera = new Camera((float)screenWidth / (float)screenHeight);
    camera->UpdateView(cameraPos, cameraTarget, cameraUp);
    float aspectRatio = (float)screenWidth / (float)screenHeight;
    camera->UpdateProjection(fovy, aspectRatio, zNear, zFar);
}
```

```
// SceneObject.
struct SceneObject
{
    SceneObject() {
        mesh = nullptr;
        worldMatrix = glm::mat4x4(1.0f);
        Ka = glm::vec3(0.3f, 0.3f, 0.3f);
        Kd = glm::vec3(0.7f, 0.7f, 0.7f);
        Ks = glm::vec3(0.6f, 0.6f, 0.6f);
        Ns = 50.0f;
    }
    TriangleMesh* mesh;
    glm::mat4x4 worldMatrix;
    // Material properties.
    glm::vec3 Ka;
    glm::vec3 Kd;
    glm::vec3 Ks;
    float Ns;
};
```

```
// ScenePointLight (for visualization of a point light).
struct ScenePointLight
{
    ScenePointLight() {
        light = nullptr;
        worldMatrix = glm::mat4x4(1.0f);
        visColor = glm::vec3(1.0f, 1.0f, 1.0f);
    }
    PointLight* light;
    glm::mat4x4 worldMatrix;
    glm::vec3 visColor;
};
```

Data Structure: Main Program (cont.)

```
void CreateShaderLib()
{
    fillColorShader = new FillColorShaderProg();
    if (!fillColorShader->LoadFromFiles("shaders/fixed_color.vs", "shaders/fixed_color.fs"))
        exit(1);

    gouraudShadingShader = new GouraudShadingDemoShaderProg();
    if (!gouraudShadingShader->LoadFromFiles("shaders/gouraud_shading_demo.vs", "shaders/gouraud_shading_demo.fs"))
        exit(1);
}

static float curRotationY = 0.0f;
const float rotStep = 0.05f;
void RenderSceneCB()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Render a triangle mesh with Gouraud shading. ----------------------------------------
    TriangleMesh* mesh = sceneObj.mesh;
    if (sceneObj.mesh ≠ nullptr) {
        // Update transform.
        curRotationY += rotStep;
        glm::mat4x4 S = glm::scale(glm::mat4x4(1.0f), glm::vec3(1.5f, 1.5f, 1.5f));
        glm::mat4x4 R = glm::rotate(glm::mat4x4(1.0f), glm::radians(curRotationY), glm::vec3(0, 1, 0));
        sceneObj.worldMatrix = S * R;
        glm::mat4x4 normalMatrix = glm::transpose(glm::inverse(camera->GetViewMatrix() * sceneObj.worldMatrix));
        glm::mat4x4 MVP = camera->GetProjMatrix() * camera->GetViewMatrix() * sceneObj.worldMatrix;
```

render the object using "GouraudShadingShader" with object transform, object material, and lighting parameters

increase Y rotation every frame

Data Structure: Main Program (cont.)

```
    gouraudShadingShader->Bind();

    // Transformation matrix.
    glUniformMatrix4fv(gouraudShadingShader->GetLocM(), 1, GL_FALSE, glm::value_ptr(sceneObj.worldMatrix));
    glUniformMatrix4fv(gouraudShadingShader->GetLocV(), 1, GL_FALSE, glm::value_ptr(camera->GetViewMatrix()));
    glUniformMatrix4fv(gouraudShadingShader->GetLocNM(), 1, GL_FALSE, glm::value_ptr(normalMatrix));
    glUniformMatrix4fv(gouraudShadingShader->GetLocMVP(), 1, GL_FALSE, glm::value_ptr(MVP));
    // Material properties.
    glUniform3fv(gouraudShadingShader->GetKa(), 1, glm::value_ptr(sceneObj.Ka));
    glUniform3fv(gouraudShadingShader->GetKd(), 1, glm::value_ptr(sceneObj.Kd));
    glUniform3fv(gouraudShadingShader->GetKs(), 1, glm::value_ptr(sceneObj.Ks));
    glUniform1f(gouraudShadingShader->GetNs(), sceneObj.Ns);
    // Light data.
    if (dirLight ≠ nullptr) {
        glUniform3fv(gouraudShadingShader->GetDirLightDir(), 1, glm::value_ptr(dirLight->GetDirection()));
        glUniform3fv(gouraudShadingShader->GetDirLightRadiance(), 1, glm::value_ptr(dirLight->GetRadiance()));
    }
    if (pointLight ≠ nullptr) {
        glUniform3fv(gouraudShadingShader->GetPointLightPos(), 1, glm::value_ptr(pointLight->GetPosition()));
        glUniform3fv(gouraudShadingShader->GetPointLightIntensity(), 1, glm::value_ptr(pointLight->GetIntensity()));
    }
    glUniform3fv(gouraudShadingShader->GetAmbientLight(), 1, glm::value_ptr(ambientLight));

    // Render the mesh.
    mesh->Draw();

    gouraudShadingShader->UnBind();
}
```

Data Structure: Main Program (cont.)

```
    // Visualize the light with fill color. ---------------------------------------------------------
    // Bind shader and set parameters.
    PointLight* pointLight = pointLightObj.light;
    if (pointLight ≠ nullptr) {
        glm::mat4x4 T = glm::translate(glm::mat4x4(1.0f), (pointLight->GetPosition()));
        pointLightObj.worldMatrix = T;
        glm::mat4x4 MVP = camera->GetProjMatrix() * camera->GetViewMatrix() * pointLightObj.worldMatrix;

        fillColorShader->Bind();

        glUniformMatrix4fv(fillColorShader->GetLocMVP(), 1, GL_FALSE, glm::value_ptr(MVP));
        glUniform3fv(fillColorShader->GetLocFillColor(), 1, glm::value_ptr(pointLightObj.visColor));

        // Render the point light.
        pointLight->Draw();

        fillColorShader->UnBind();
    }
    // ---------------------------------------------------------------------------------------------

    glutSwapBuffers();
}
```

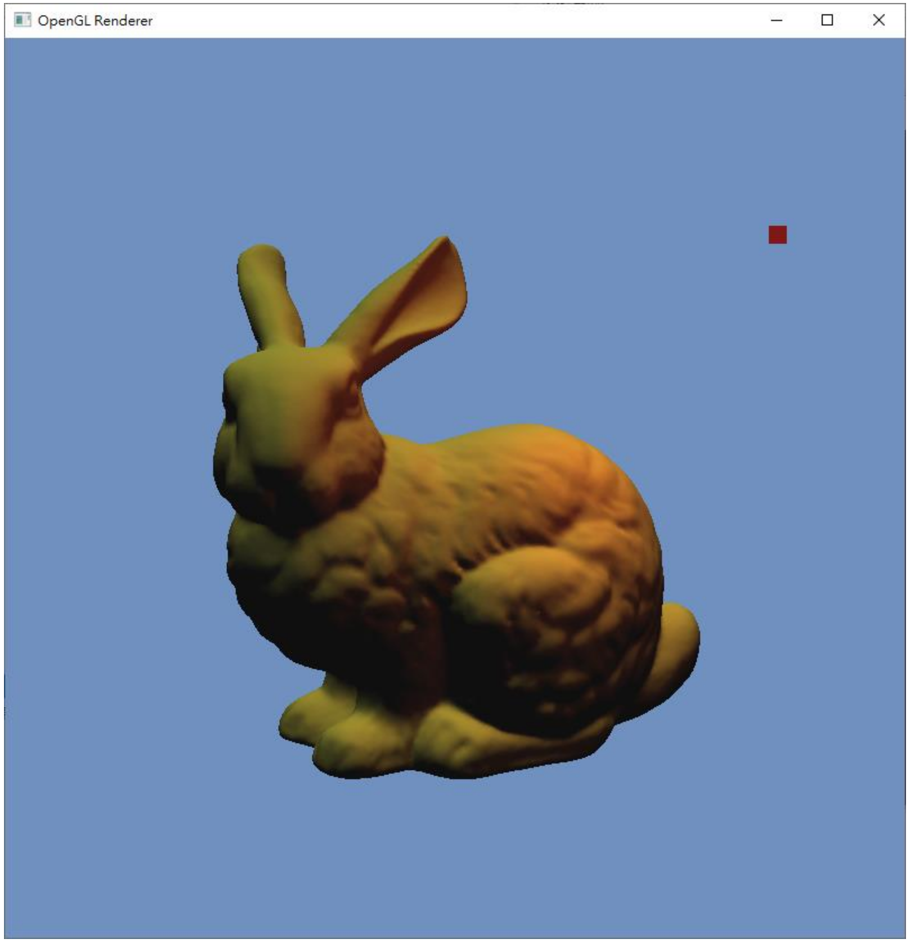
render the point light using "FillColorShader"
With point light transform and color

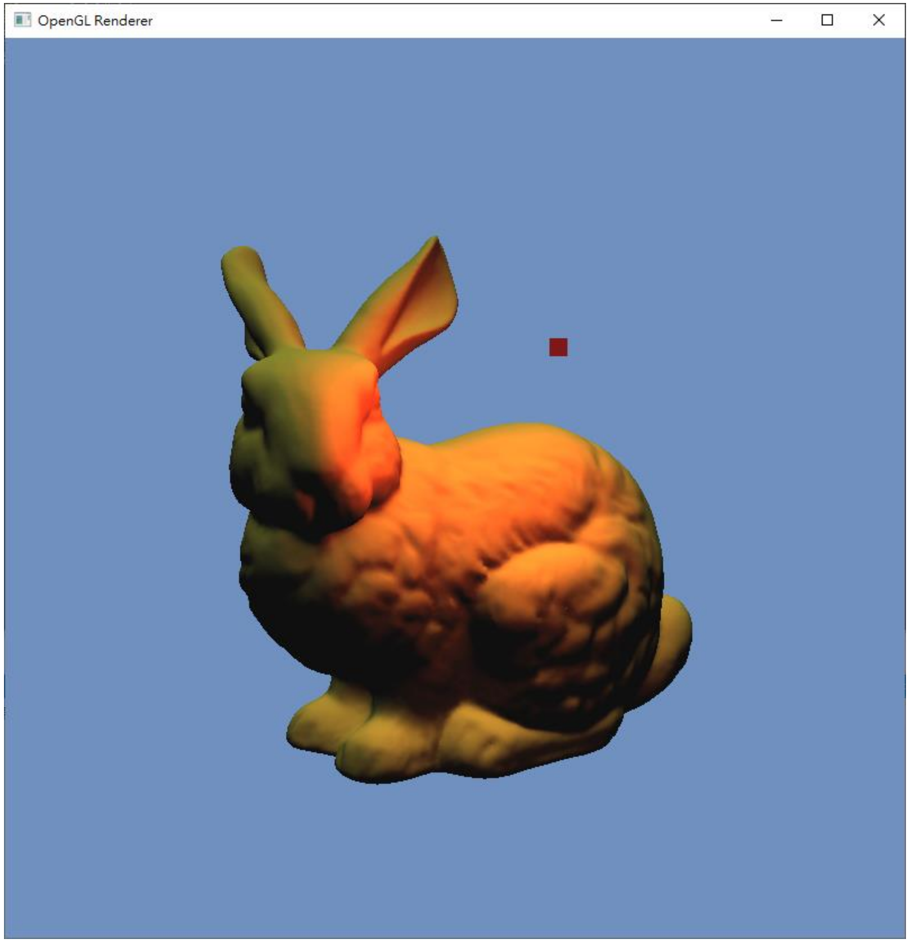
Data Structure: Main Program (cont.)

interactively control the point light with the keyboard

```
void ProcessSpecialKeysCB(int key, int x, int y)
{
    // Handle special (functional) keyboard inputs such as F1, spacebar, page up, etc.
    switch (key) {
    // Rendering mode.

    // Light control.
    case GLUT_KEY_LEFT:
        if (pointLight != nullptr)
            pointLight->MoveLeft(lightMoveSpeed);
        break;
    case GLUT_KEY_RIGHT:
        if (pointLight != nullptr)
            pointLight->MoveRight(lightMoveSpeed);
        break;
    case GLUT_KEY_UP:
        if (pointLight != nullptr)
            pointLight->MoveUp(lightMoveSpeed);
        break;
    case GLUT_KEY_DOWN:
        if (pointLight != nullptr)
            pointLight->MoveDown(lightMoveSpeed);
        break;
    default:
        break;
    }
}
```

Results





Practices

- Implement specular shading (HW2)
- Implement spotlight (HW2)
- Implement Phong shading (HW2)

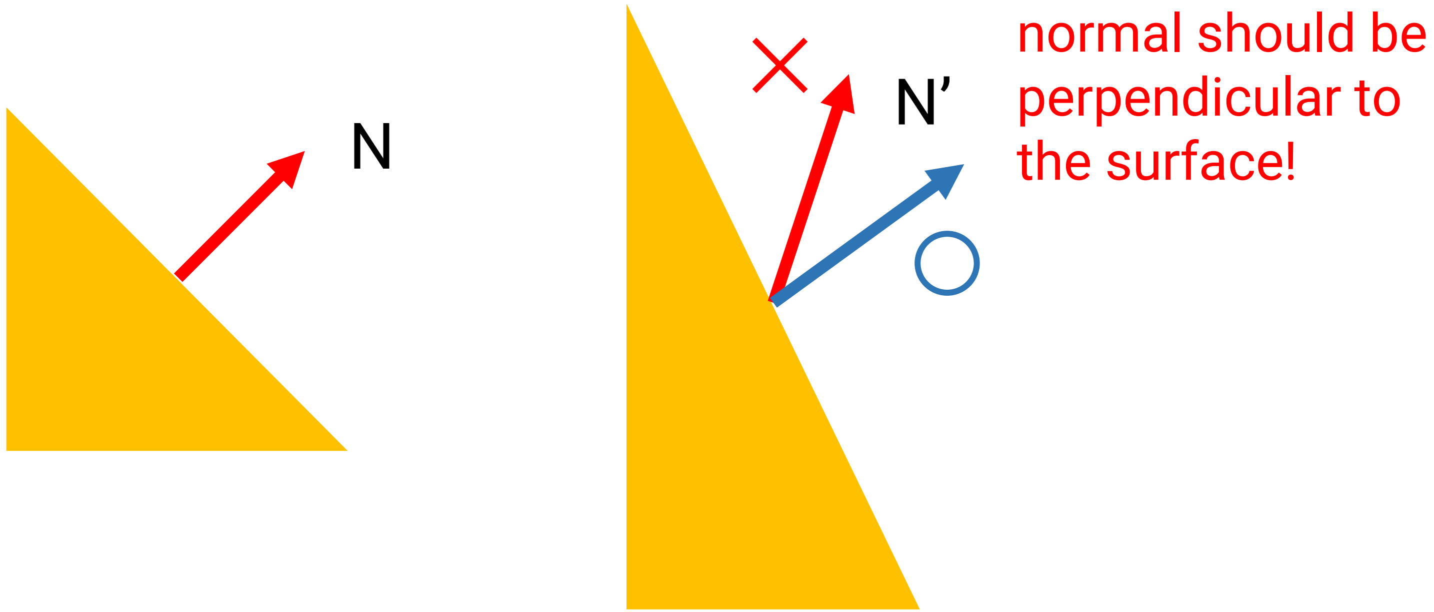
Any Questions?

Normal Matrix

- To transform a point from ***Object Space*** to ***World Space***, we multiply its object-space position by the **world** (**model**) matrix
- How about the vertex normal?
  - We also need to transform the object-space normal to World Space for lighting computation
  - Could we also multiply the object-space normal by the world matrix?

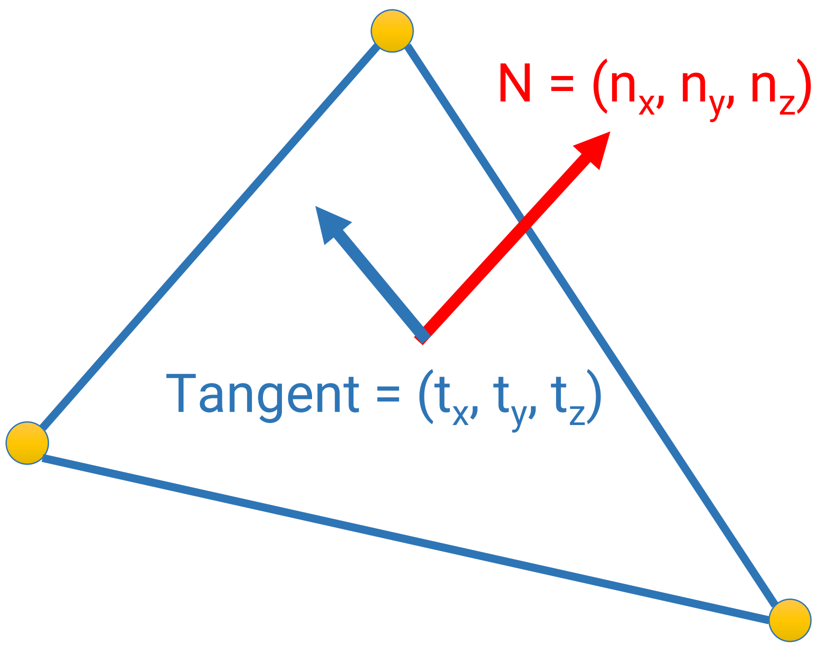
Normal Matrix (cont.)

- If the scaling in a world matrix is **uniform**, you can use the world matrix for transforming the normal directly
- However, if there is a **non-uniform** scaling, the matrix for transforming normal should be different



Normal Matrix (cont.)

- Derivation of the normal matrix



$$(n_x, n_y, n_z, 0) \cdot (t_x, t_y, t_z, 0) = 0$$

$$(n_x, n_y, n_z, 0) \begin{pmatrix} t_x \\ t_y \\ t_z \\ 0 \end{pmatrix} = 0$$

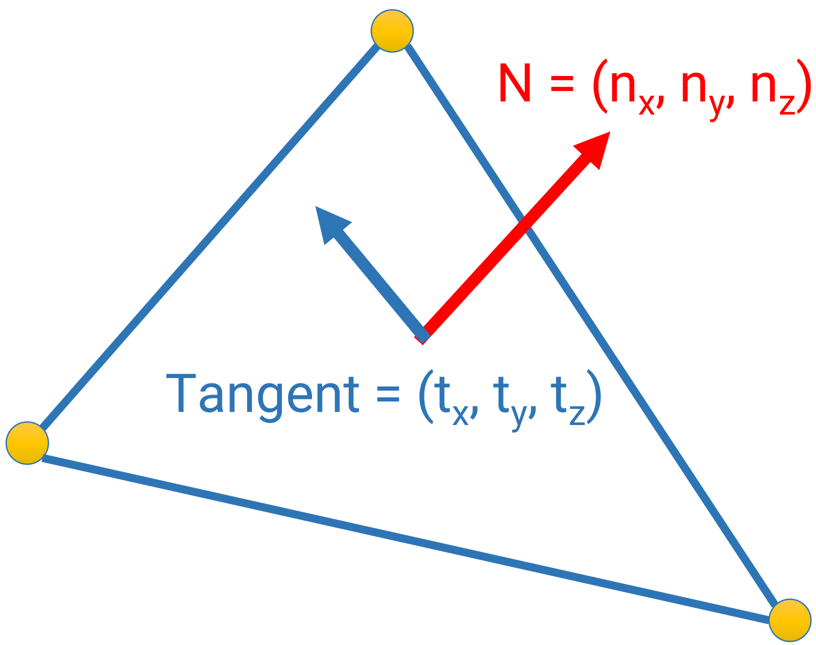
$$\boxed{(n_x, n_y, n_z, 0)\, M^{-1}} \boxed{M \begin{pmatrix} t_x \\ t_y \\ t_z \\ 0 \end{pmatrix}} = 0$$

transform normal

transform vertex

Normal Matrix (cont.)

- Derivation of the normal matrix



Note: if you want to compute lighting in **Camera Space**, the ***M*** should be the **modelview** matrix

$$\begin{pmatrix} n_x^{world} \\ n_y^{world} \\ n_z^{world} \\ 0 \end{pmatrix}^T = (n_x, n_y, n_z, 0)M^{-1}$$

$$(AB)^T = B^T A^T$$

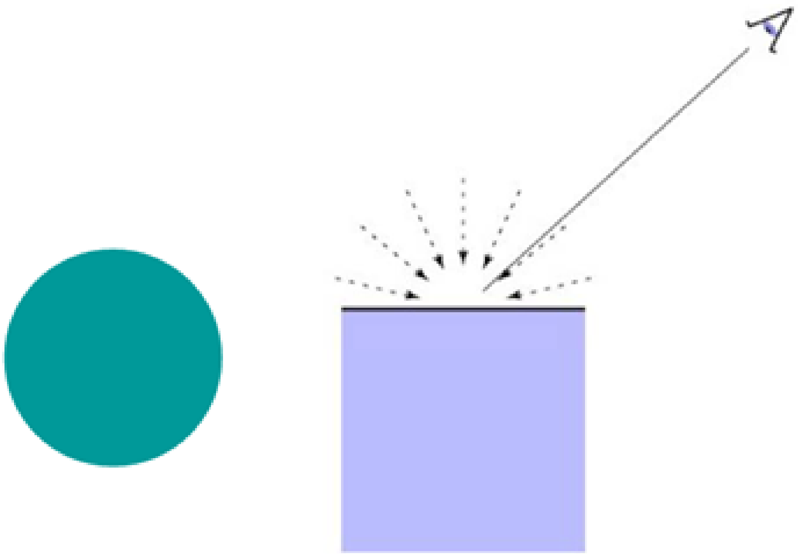
$$\begin{pmatrix} n_x^{world} \\ n_y^{world} \\ n_z^{world} \\ 0 \end{pmatrix} = \boxed{(M^{-1})^T} \begin{pmatrix} n_x \\ n_y \\ n_z \\ 0 \end{pmatrix}$$

normal matrix

(the inverse transpose of world matrix)

Recap: Ambient Shading

- Add constant color to account for disregarded illumination and fill black shadows



the **intensity** of ambient light

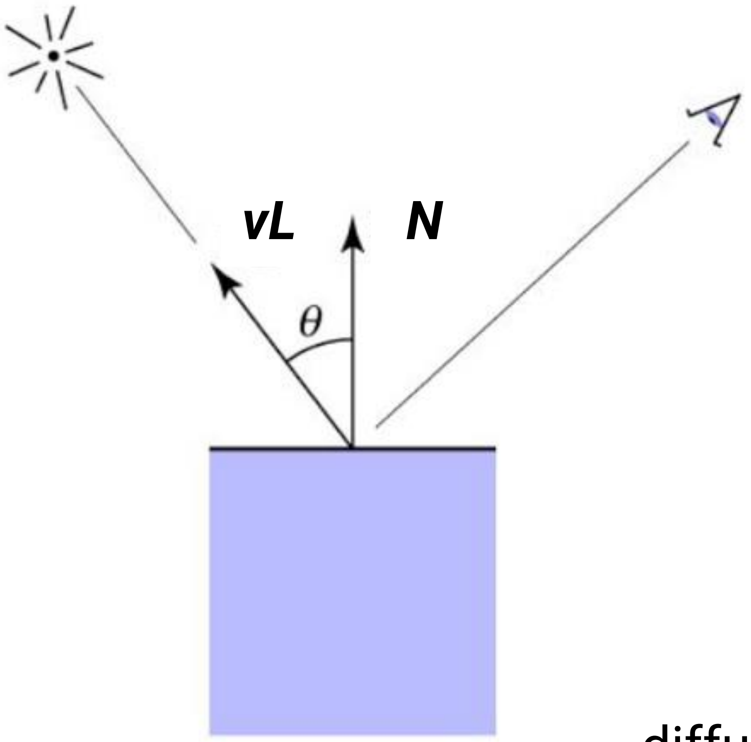
$$L_a = k_a \cdot I_a$$

ambient coefficient

reflected ambient light

Recap: Diffuse Shading

- Applies to diffuse or matte surface



illumination from source

$$L_d = k_d \cdot I \cdot \max(0, N \cdot vL)$$

Lambertian law

diffuse coefficient

diffusely reflected light

Recap: Local Light Attenuation

- The length of the side of a receiver patch is proportional to its distance from the light
- As a result, the average energy per unit area is proportional to the **square of the distance** from the light