XIAMEN UNIVERSITY

COMPUTER GRAPHICS

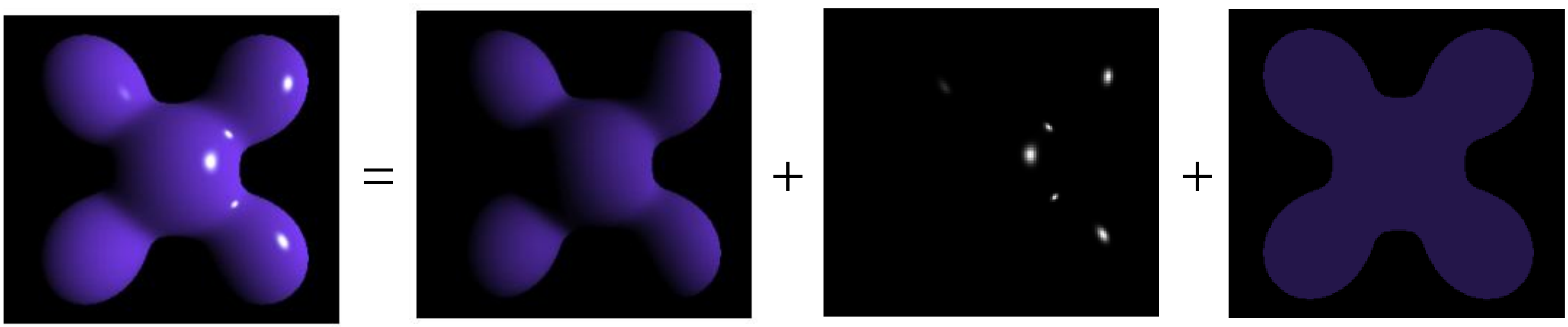
第七章 光照 II

陈中贵
厦门大学信息学院
http://graphics.xmu.edu.cn

Phong光照模型

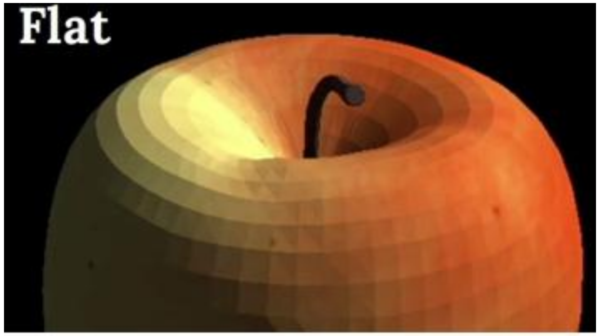
- 反射光强 = 漫反射光+镜面光+环境光

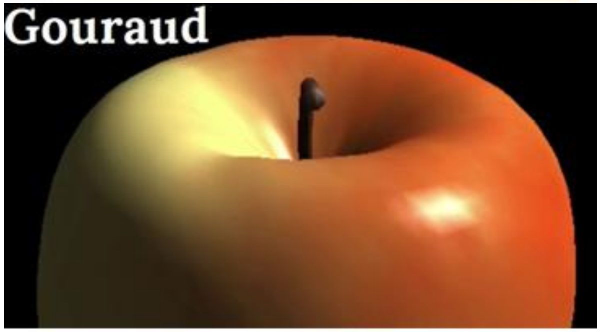
$$
\begin{aligned}
I_{total} &= I_{diffuse} + I_{specular} + I_{ambient} \\
&= k_d \cdot \cos\theta \cdot L_d + k_s \cdot L_s \cdot \cos^{\alpha}\varphi + k_a \cdot L_a
\end{aligned}
$$

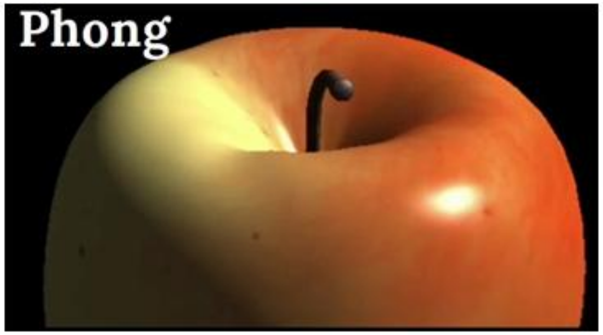


多边形网格的明暗处理

- 在多边形网格中每个多边形为平面，那么存在唯一的法向量
- 三种明暗处理的方法
  - 平面着色(flat shading)
  - Gouraud着色(Gouraud shading)或插值着色
  - Phong着色(Phong shading)

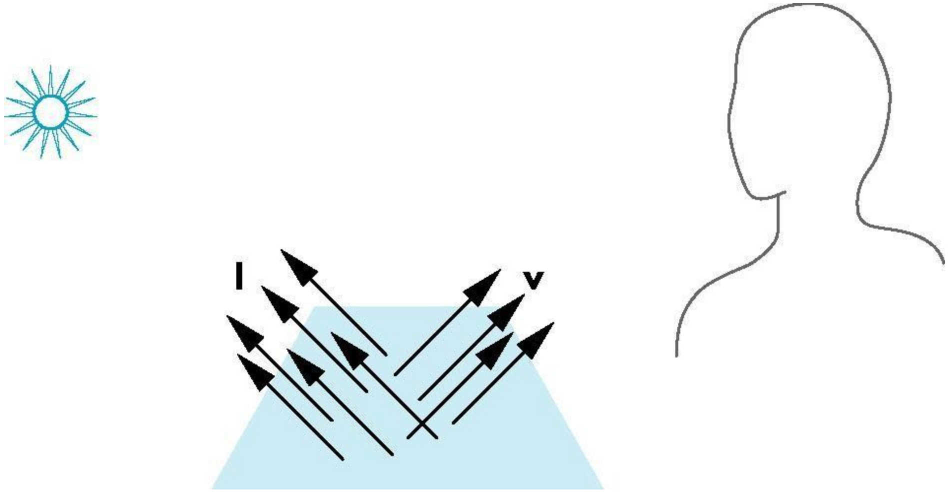






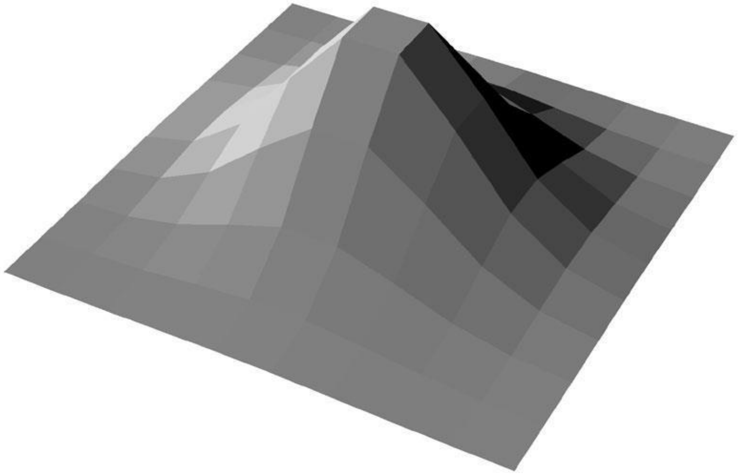
平面明暗处理

- 在同一多边形上法向**n**为常向量
- 视点在无穷远，视点方向**v**是常向量
- 光源在无穷远，入射方向**l**也是常向量
- 从而对于每个多边形，只需要计算其上一点的颜色，其它点的颜色与它相同



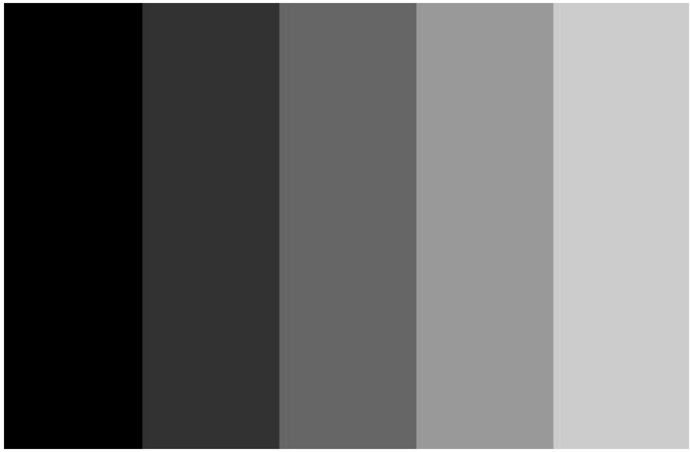
特点

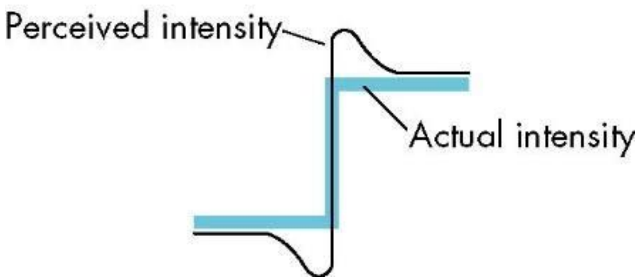
- 网格中每个多边形的颜色不同
  - 如果多边形网格表示的是一个光滑曲面，那么这种效果显然是不令人满意的



人类视觉系统

- 人类视觉系统对光强的变化非常敏感，称为旁侧抑制特性 (lateral inhibition)
- 注意下图边界的条状效果，称为Mach带
- 没有办法避免这种情形，只有给出更平滑的明暗处理方法





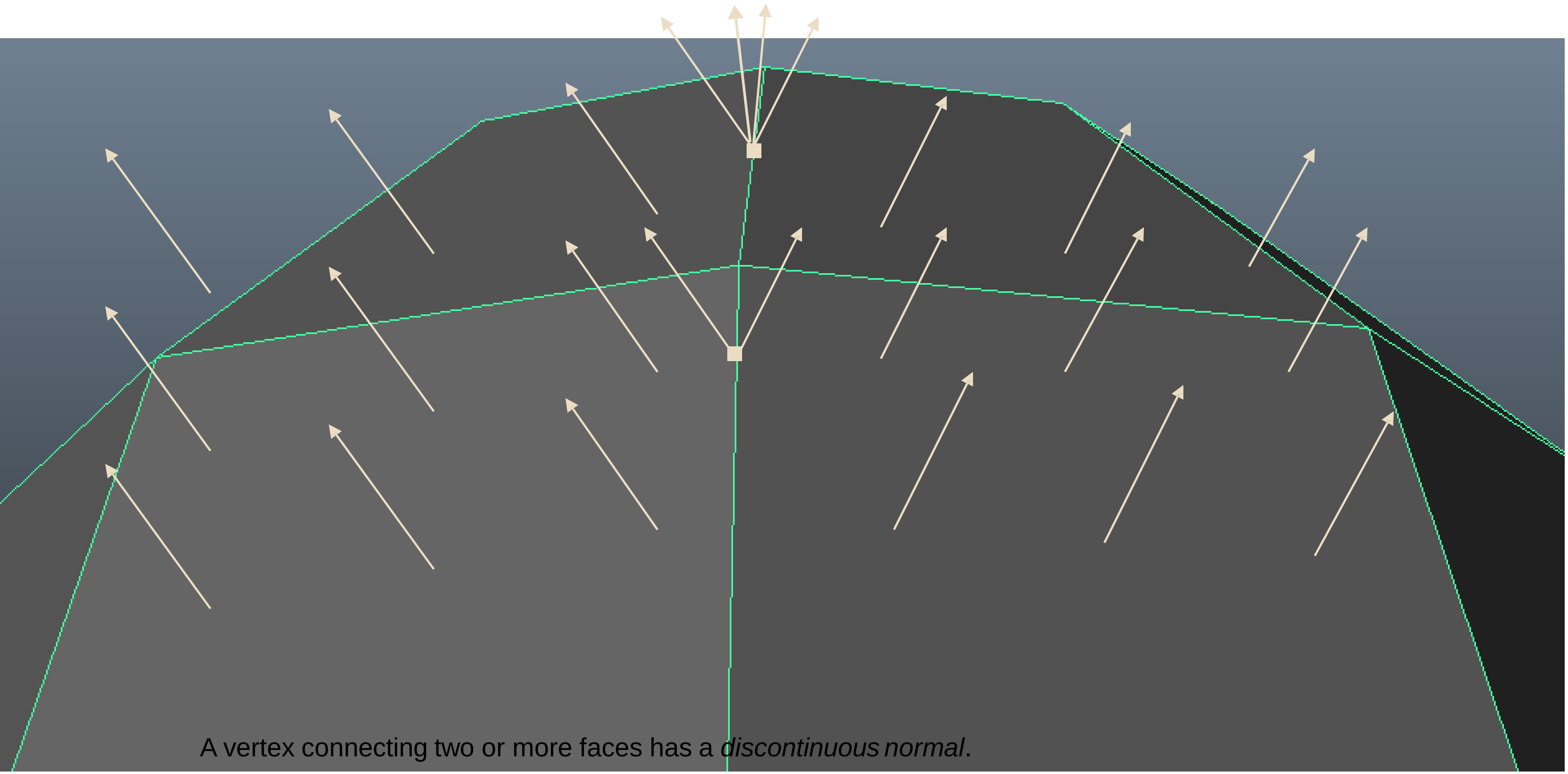
光滑着色

- Gouraud (1971)
  - ADS 计算在每一个顶点进行，通过光栅化插值颜色.
- Phong (1973)
  - 通过光栅化插值法向，然后在每个片段上进行ADS计算.
- Blinn-Phong (1977)
  - 和以上Phong着色一样,优化其中的反射光线计算，提高效率

三种方法都需要计算法向！！！

Gouraud着色

- 如何计算顶点法向?

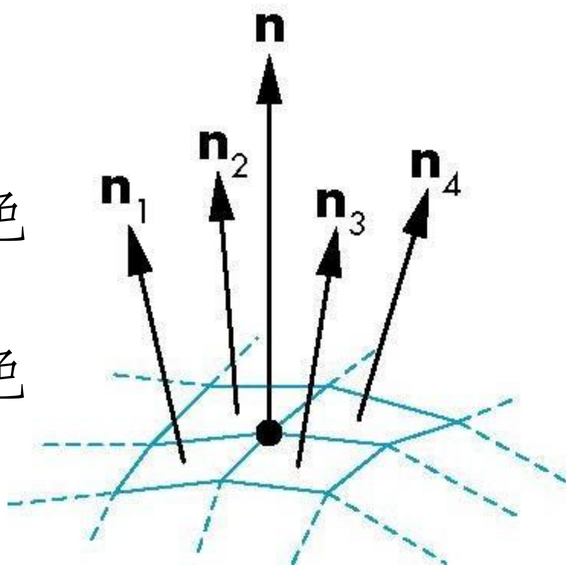


Gouraud着色

- 在网格中每个顶点处有几个多边形交于该点，每个多边形有一个法向，取这几个法向的平均得到该点的法向

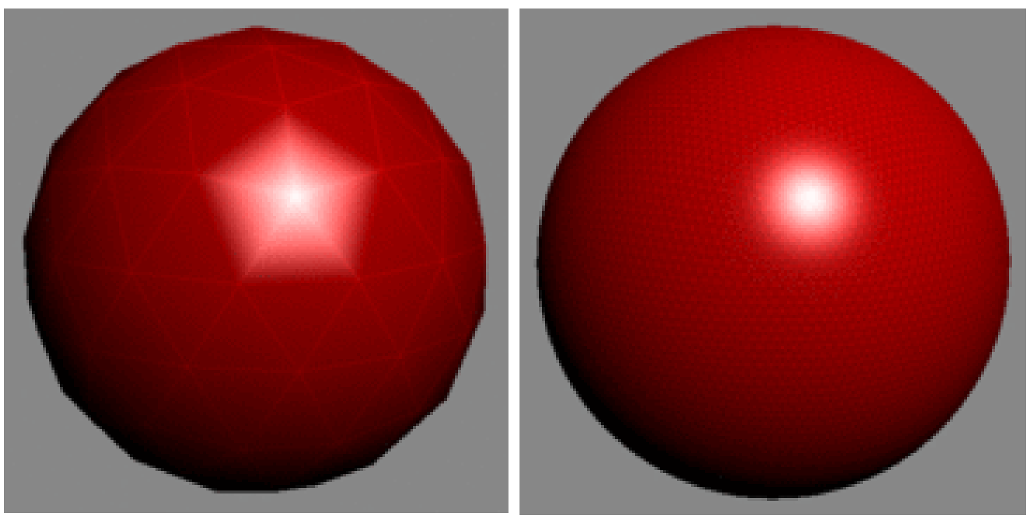
$$n = \frac{n_1 + n_2 + n_3 + n_4}{|n_1 + n_2 + n_3 + n_4|}$$

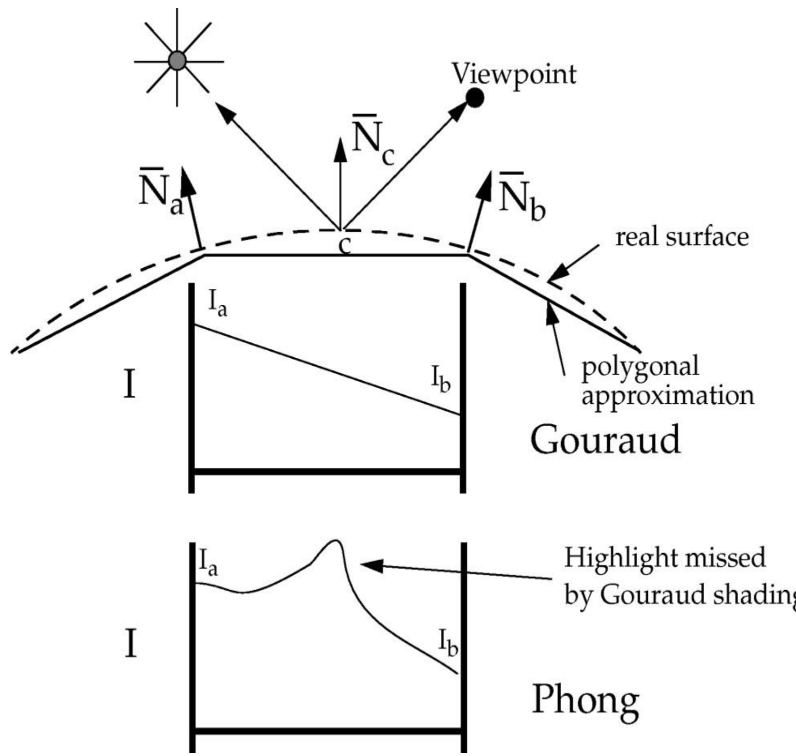
- 然后利用简单光照模型计算出顶点的颜色
- 对多边形内的点，采用线性插值确定颜色



Gouraud着色法缺点

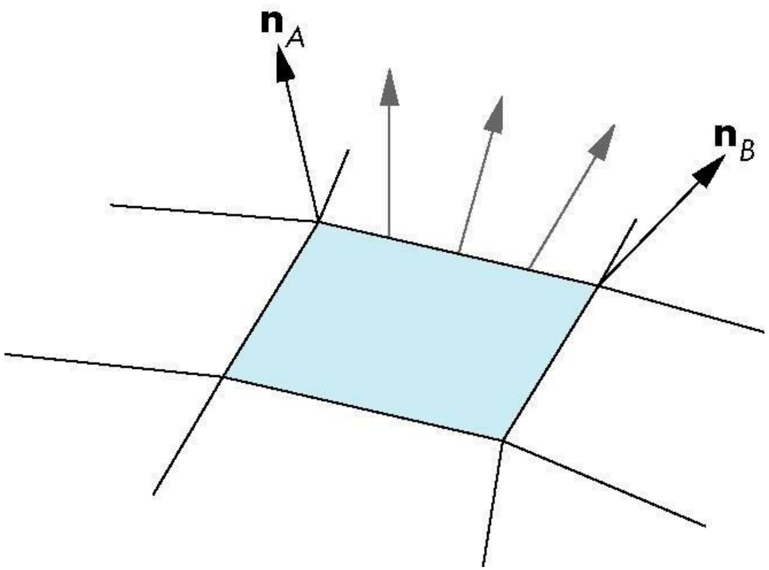
- 着色后仍然可以看出一个个小平面的效果
- 渲染一些与位置相关的光照效果（比如高光）时有问题

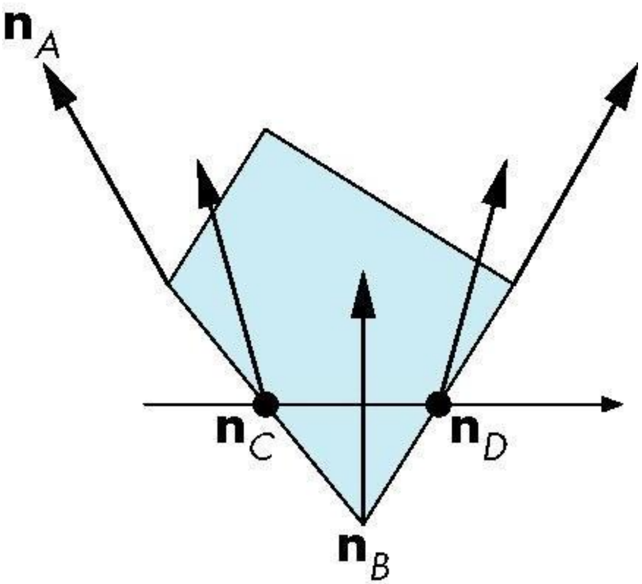




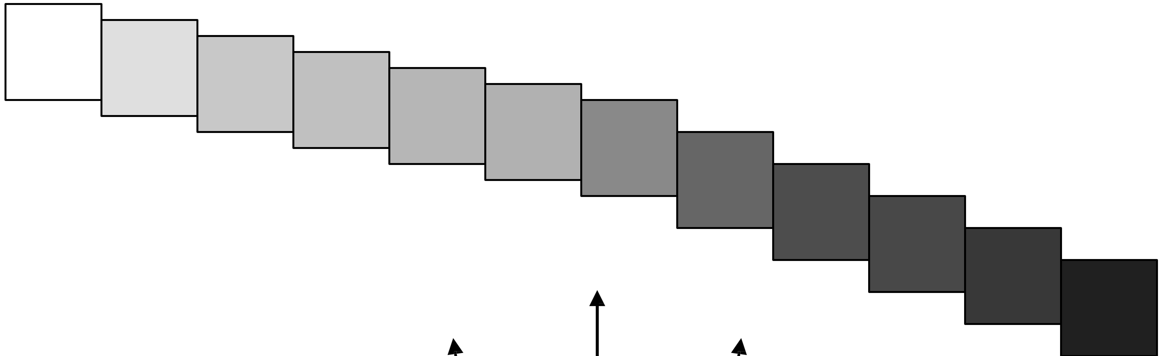
Phong明暗处理

- 与Gouraud方法不同，Phong方法是根据每个顶点的法向，插值出多边形内部各点的法向，然后基于光照模型计算各点的颜色

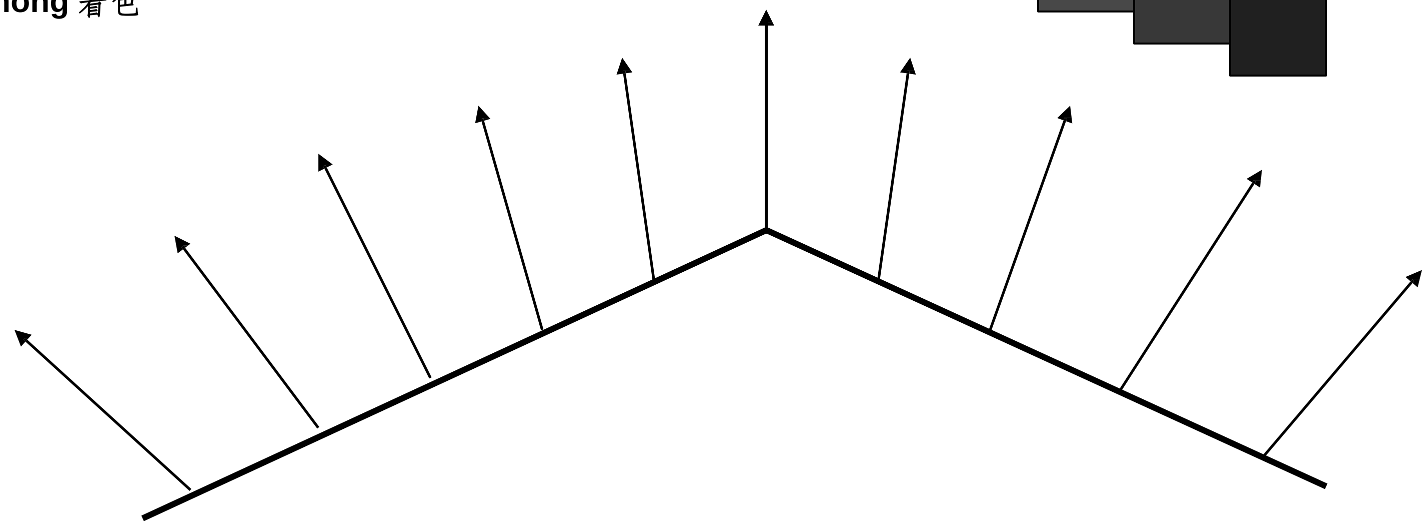


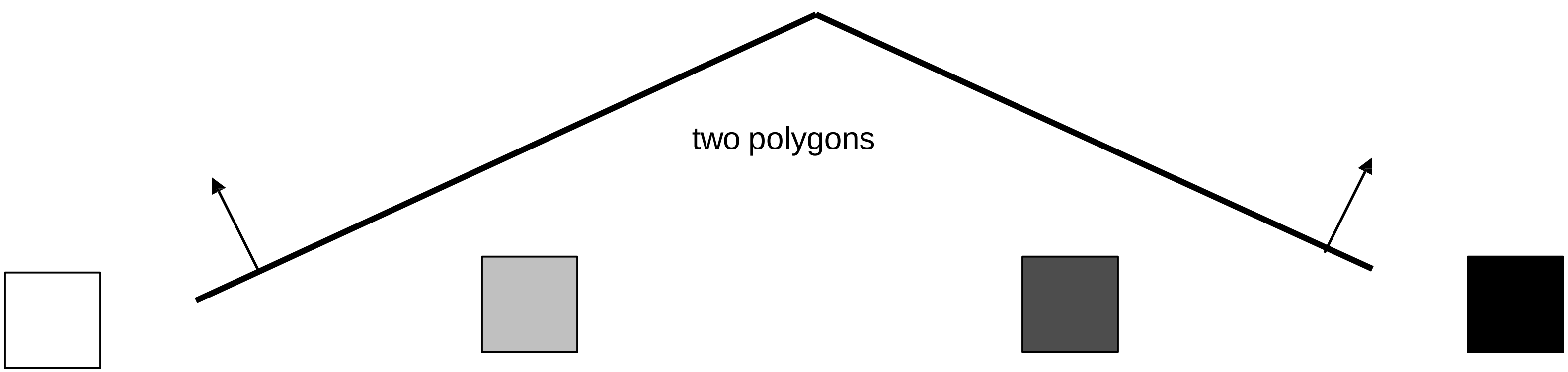


Gouraud 着色



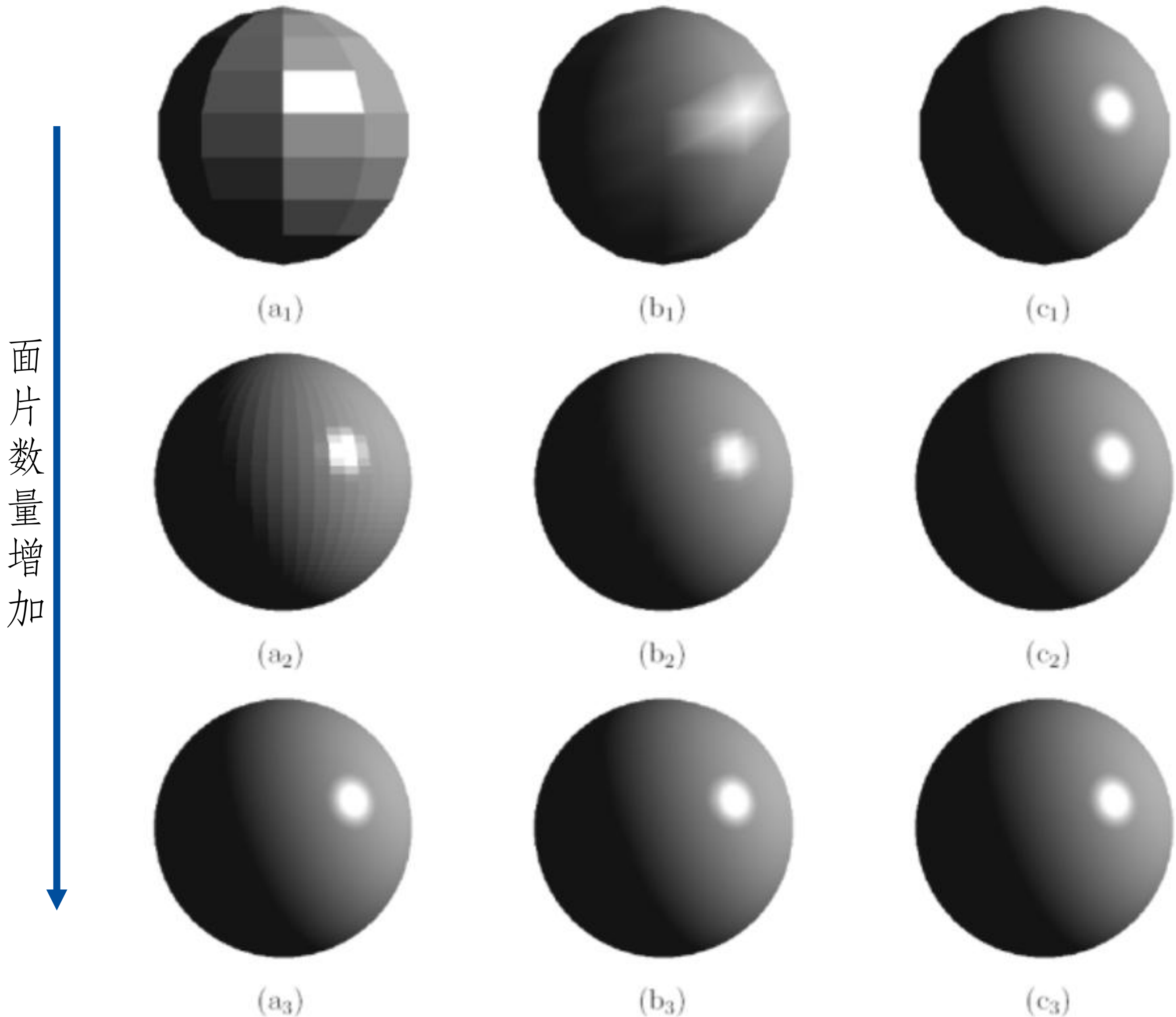
Phong 着色





着色模式

- 平面着色(flat shading)
  - 每个多边形只会呈现一个颜色，这个颜色由面法向量和光照计算得来。在该模型中，每个多边形中只有多边形的面存在法向量，而其各个顶点没有。
- Gouraud着色(Gouraud shading)或插值着色
  - 插值颜色，通常先计算多边形每个顶点的光照，再通过双线性插值计算三角形区域中其它像素的颜色。
- Phong着色(Phong shading)
  - 插值法向量，多边形中每个顶点都有一个法向量，通过这些法向量与光照计算，来得到每个点的颜色。在使用有限数量的多边形时，对顶点法向量进行插值可以给出近似平滑的曲面效果



Flat→Gouraud→Phong Shading

OpenGL中的光照

- 全局环境光：模拟到达附近所有物体的低级别辉光，处处一致

```
float globalAmbient[4] = { 0.6, 0.6f, 0.6f, 1.0f };
```

- 定向光：有方向，无需源位置，如阳光
  - 例子：红色定向光

```
float dirLightAmbient[4] = { 0.1f, 0.0f, 0.0f, 1.0f };
float dirLightDiffuse[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
float dirLightSpecular[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
float dirLightDirection[3] = { 0.0f, 0.0f, -1.0f };
```

- 点光源：

Example of a red positional light at location (5, 2, -3):

```
float posLightAmbient[4] = { 0.1f, 0.0f, 0.0f, 1.0f };
float posLightDiffuse[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
float posLightSpecular[4] = { 1.0f,0.0f, 0.0f, 1.0f };
float posLightLocation[3] = { 5.0f, 2.0f, -3.0f };
```

聚光灯

- 聚光灯（spotlight）同时具有位置和方向。
- “锥形”效果使用 0° ～90° 的截光角 ϑ 来模拟
- 使用衰减指数可以模拟随光束角度的强度变化
- 衰减指数会影响当角度 φ 增加时，强度因子趋于 0 的速率

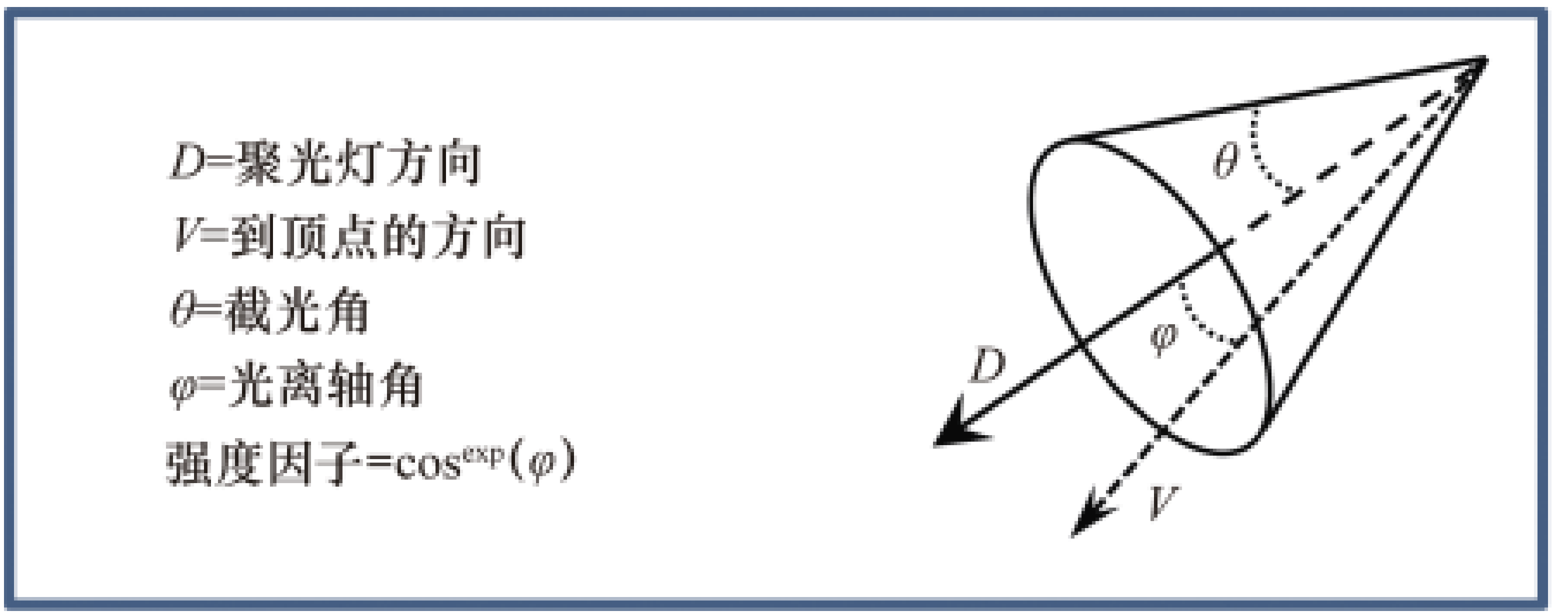


图 7.2　聚光灯参数

聚光灯

- 位于(5,2,-3)向下照射 *z* 轴负方向的红色聚光灯可以表示为

```
float spotLightAmbient[4] = { 0.1f, 0.0f, 0.0f, 1.0f };
float spotLightDiffuse[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
float spotLightSpecular[4] = { 1.0f,0.0f, 0.0f, 1.0f };
float spotLightLocation[3] = { 5.0f, 2.0f, -3.0f };
float spotLightDirection[3] = { 0.0f, 0.0f, -1.0f };
float spotLightCutoff = 20.0f;
float spotLightExponent = 10.0f;
```

材质

- 通过指定 4 个值（环境光反射、漫反射、镜面反射在ADS 光照模型中模拟材质。
- 第 4 个值叫作光泽度，用来为所选材质建立一个合适的镜面高光
  - 例如，要模拟锡铅合金的效果，可以指定如下值：

```
float pewterMatAmbient[4] = { .11f, .06f, .11f, 1.0f };
float pewterMatDiffuse[4] = { .43f, .47f, .54f, 1.0f };
float pewterMatSpecular[4] = { .33f, .33f, .52f, 1.0f };
float pewterMatShininess = 9.85f;
```

材质

- 更多材质属性见：

  Barradeu, N., http://www.barradeau.com/nicoptere/dump/materials.html

材质	环境光RGBA 漫反射RGBA 反射RGBA	光泽度
黄金	0.2473, 0.1995, 0.0745, 1.0 0.7516, 0.6065, 0.2265, 1.0 0.6283, 0.5558, 0.3661, 1.0	51.200
玉	0.1350, 0.2225, 0.1575, 0.95 0.5400, 0.8900, 0.6300, 0.95 0.3162, 0.3162, 0.3162, 0.95	12.800
珍珠	0.2500, 0.2073, 0.2073, 0.922 1.0000, 0.8290, 0.8290, 0.922 0.2966, 0.2966, 0.2966, 0.922	11.264
银	0.1923, 0.1923, 0.1923, 1.0 0.5075, 0.5075, 0.5075, 1.0 0.5083, 0.5083, 0.5083, 1.0	51.200

图 7.3 其他材质的 ADS 系数

实现 ADS 光照

- Gouraud 着色（双线性光强插值法）

（1）确定每个顶点的颜色，并进行光照相关计算。

（2）允许正常的栅格化过程在插入像素时对颜色也进行插值（同时也对光照进行插值）。

在 OpenGL 中，大多数光照计算在顶点着色器中完成的，片段着色器仅传递并展示自动插值的光照后的颜色。

OpenGL (C++) 代码

放入缓冲区：
1.模型顶点
2.顶点法向量
放入统一变量：
1.MV和PROJ矩阵变换
2.光照和材质特性

→

顶点着色器

1.根据顶点计算*N*、*L*、*V*和*R*向量
2.计算*A*、*D*、*S*分量
3.输出属性、
-光照后的颜色
-gl_positon

→

片段着色器

传入插值
-颜色
-位置

实现Gouraud 着色

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, initialLichtLoc.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 = gl.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;
```

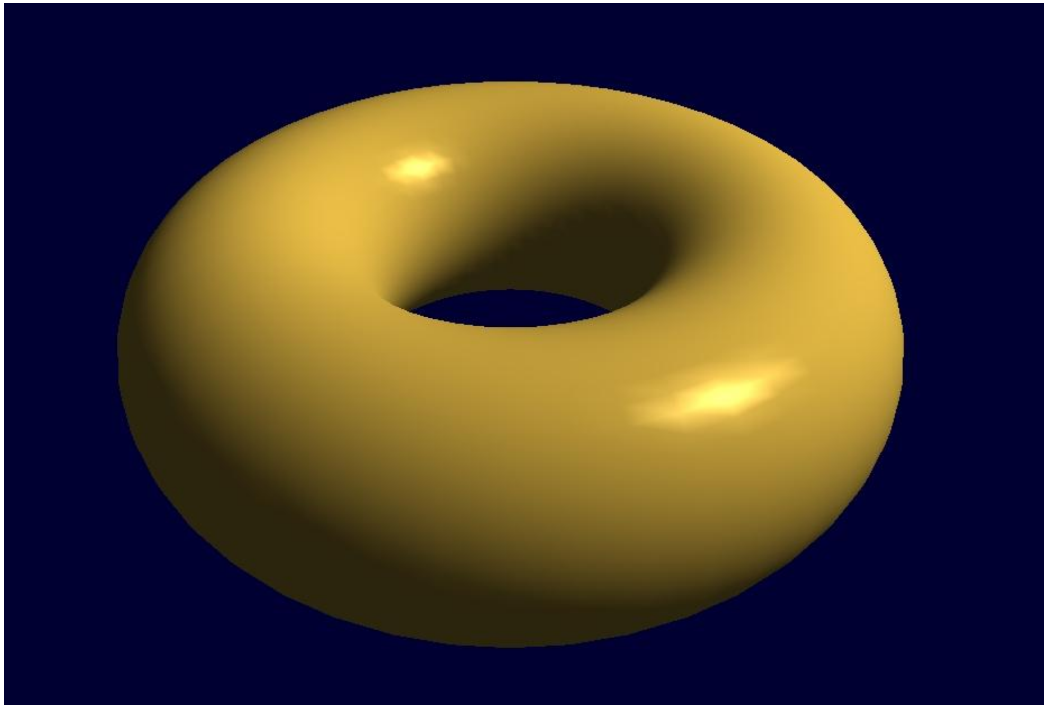
(continued…)

```
. . .
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);
}
```

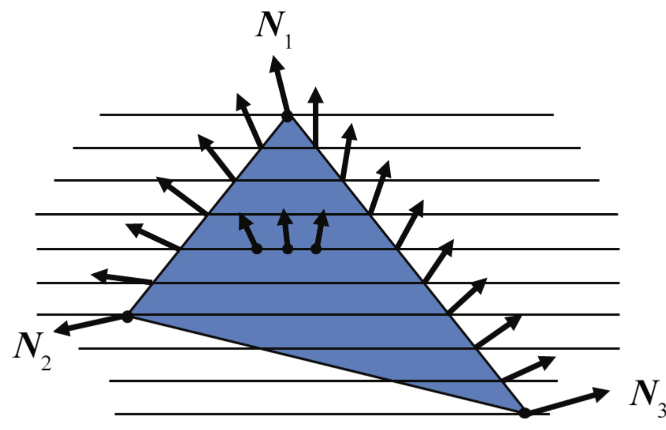
Fragment Shader

```
#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;
}
```



实现Phong 着色

- Bui Tuong Phong 在犹他大学读研究生期间开发了一种平滑的着色算法， 1973 年发表论文
- 法向量 N 和光向量 L仅有顶点包含这些信息， Phong 着色将*N* 和 *L* 在顶点着色器中进行计算，并在栅格化期间插值



OpenGL (C++) 代码

（与Gouraud着色相同）

顶点着色器

1. 计算向量*N*、*L*、*V*。
2. 输出属性*N*、*L*、*V*、gl_position。

片段着色器

1. 传入插值*N*、*L*、*V*。
2. 计算*R*、*θ*、*φ*。
3. 计算ADS分量。
4. 输出颜色。

实现Phong着色

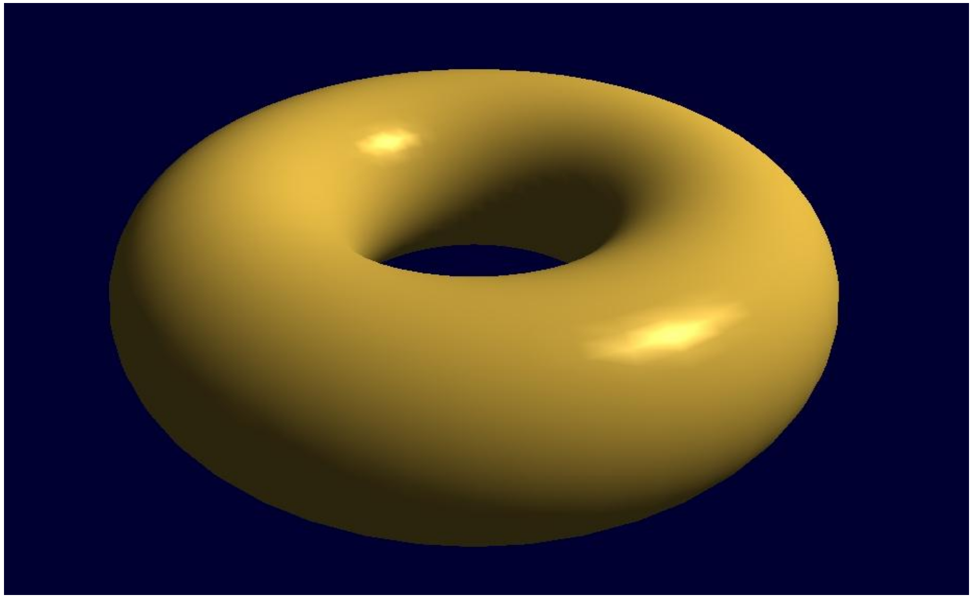
Vertex Shader

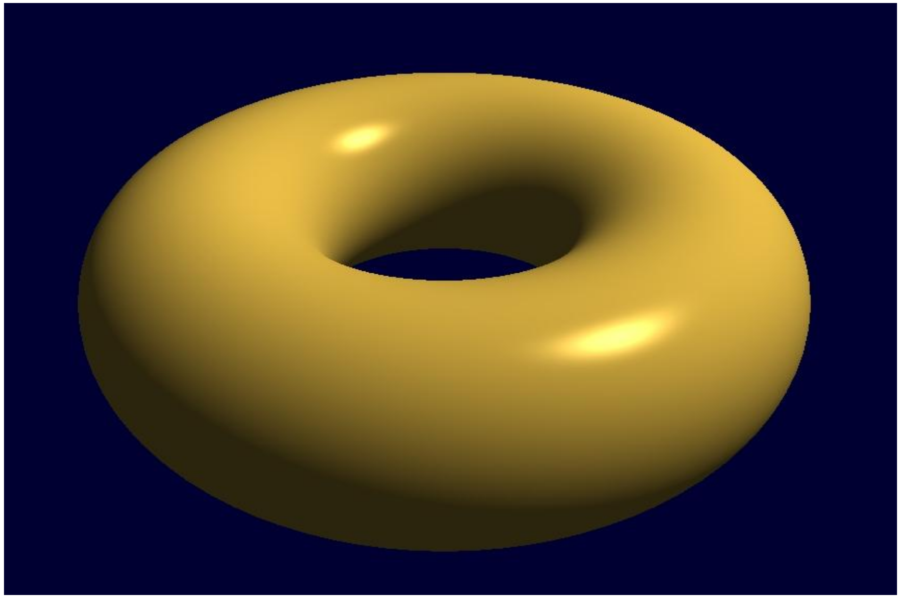
```
#version 430
layout (location=0) in vec3 vertPos;
layout (location=1) in vec3 vertNormal;
out vec3 varyingNormal;        // eye-space vertex normal
out vec3 varyingLightDir;      // vector pointing to the light
out vec3 varyingVertPos;       // vertex position in eye space
. . .
// structs and uniforms 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

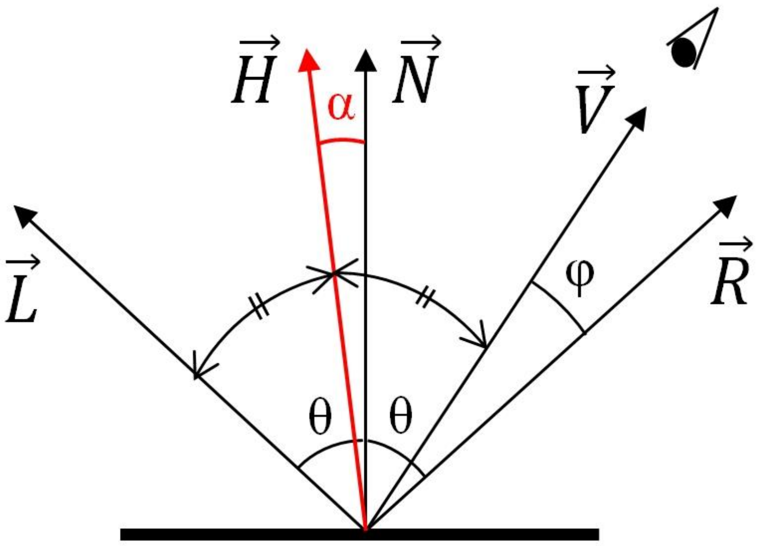
```
. . .
//  inputs correspond to outputs of fragment shader.
// structs and uniforms same as for Gouraud shading.
. . .
void main(void)
{   vec3 L = normalize(varyingLightDir);
    vec3 N = normalize(varyingNormal);
    vec3 V = normalize(-varyingVertPos);
    // compute light reflection vector with 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);
}
```

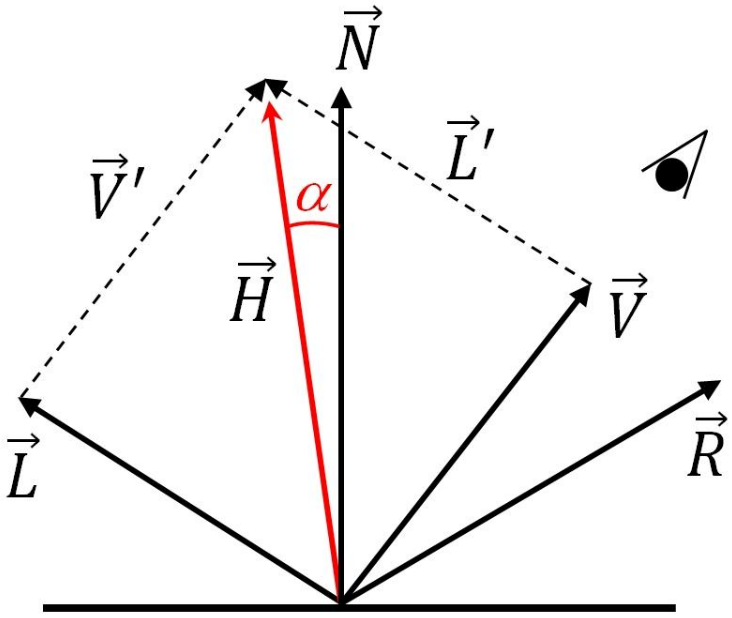
Gouraud着色 vs Phong 着色





实现Blinn-Phong着色





要求: φ

通过计算 α --

$$\alpha = \frac{1}{2}\varphi$$

容易计算 $\vec{H}$: $\vec{H} = \vec{L} + \vec{V}$

实现Blinn-Phong着色

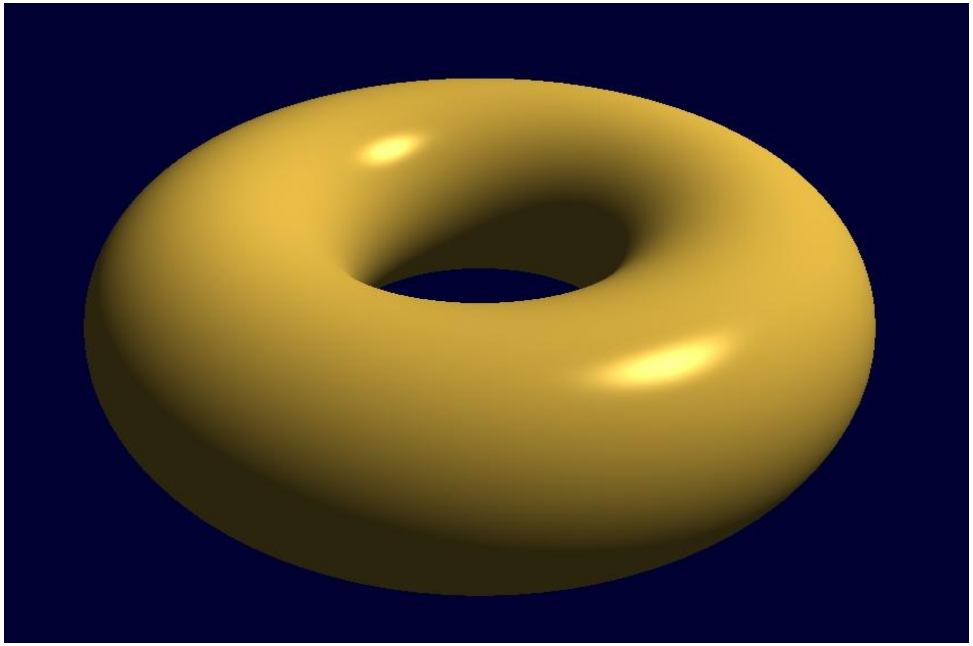
Vertex Shader

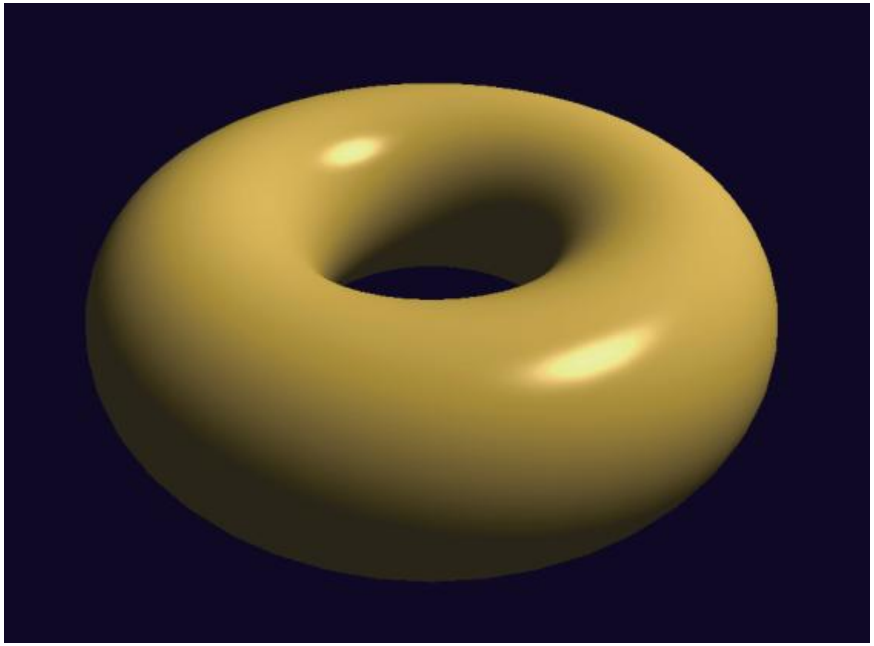
```
. . .
out vec3 varyingHalfVector;
void main(void)
{   . . .
    varyingHalfVector = (varyingLightDir + (-varyingVertPos)).xyz;
    . . .
}
```

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);
    . . .
}
```

Phong 着色 vs Blinn-Phong着色





更多例子

Studio 522 Dolphin
34,014 triangles

Stanford Dragon
871,167 triangles

结合光照与纹理

- 我们结合光照和纹理的方式取决于物体的特性及其纹理的目的
- 情况一：纹理图像很写实地反映了物体真实的表面外观

fragColor = textureColor * (ambientLight + diffuseLight) + specularLight

结合光照与纹理

- 我们结合光照和纹理的方式取决于物体的特性及其纹理的目的
- 情况一：纹理图像很写实地反映了物体真实的表面外观

  fragColor = textureColor * (ambientLight + diffuseLight) + specularLight

- 情况二：镜面高光部分都包含物体表面颜色，如织物或未上漆的木材

  fragColor = textureColor * (ambientLight + diffuseLight + specularLight)

结合光照与纹理

- 我们结合光照和纹理的方式取决于物体的特性及其纹理的目的
- 情况一：纹理图像很写实地反映了物体真实的表面外观

fragColor = textureColor * (ambientLight + diffuseLight) + specularLight

- 情况二：镜面高光部分都包含物体表面颜色，如织物或未上漆的木材

fragColor = textureColor * (ambientLight + diffuseLight + specularLight)

- 情况三：物体本身具有 ADS 材质，并伴有纹理图像，如使用纹理为银质物体表面添加一些氧化痕迹

lightColor = (ambLight * ambMaterial) + (diffLight * diffMaterial) + specLight
fragColor = 0.5 * textureColor + 0.5 * lightColor

结合光照与纹理

- 例子：没有使用材质，并在镜面高光中仅使用光照进行了计算

```
vec4 texColor = texture(sampler, texCoord);
fragColor = texColor * (globalAmbient + lightAmb + lightDiff * max(dot(L,N),0.0))
          + lightSpec * pow(max(dot(H,N),0.0), matShininess*3.0);
```