



下载APP



24 | 如何模拟光照让3D场景更逼真？（下）

2020-08-17 月影

跟月影学可视化

[进入课程 >](#)



讲述：月影

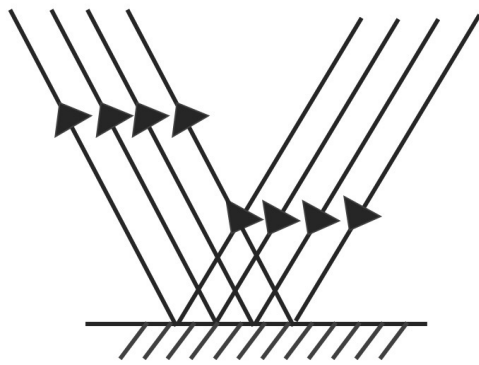
时长 10:34 大小 9.69M



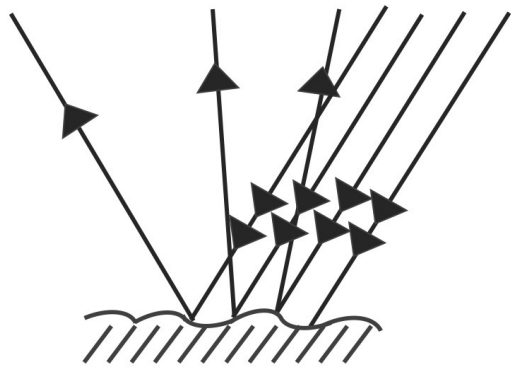
你好，我是月影。今天，我们接着来讲，怎么模拟光照。

上节课，我们讲了四种光照的漫反射模型。实际上，因为物体的表面材质不同，反射光不仅有漫反射，还有镜面反射。





镜面反射

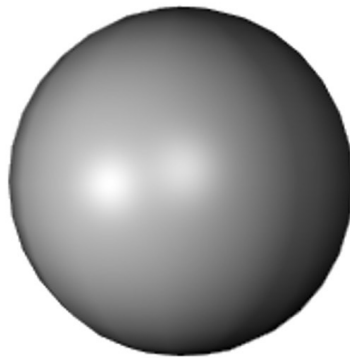


漫反射

镜面反射与漫反射

什么是镜面反射呢？如果若干平行光照射在表面光滑的物体上，反射出来的光依然平行，这种反射就是镜面反射。镜面反射的性质是，入射光与法线的夹角等于反射光与法线的夹角。

越光滑的材质，它的镜面反射效果也就越强。最直接的表现就是物体表面会有闪耀的光斑，也叫镜面高光。但并不是所有光都能产生镜面反射，我们上节课讲的四种光源中，环境光因为没有方向，所以不参与镜面反射。剩下的平行光、点光源、聚光灯这三种光源，都是能够产生镜面反射的有向光。

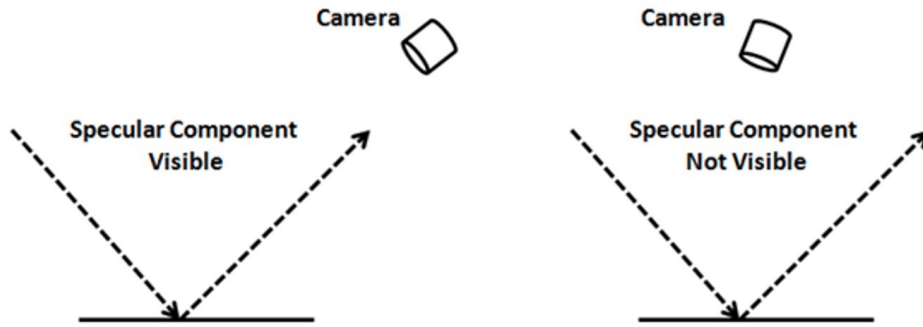


镜面高光

那么今天，我们就来讨论一下如何实现镜面反射，然后将它和上节课的漫反射结合起来，就可以实现标准的光照模型，也就是 Phong 反射模型了，从而能让我们实现的可视化场景更加接近于自然界的效果。

如何实现有向光的镜面反射？

首先，镜面反射需要同时考虑光的入射方向以及相机也就是观察者所在的方向。



观察者与光的入射方向

接着，我们再来说怎么实现镜面反射效果，一般来说需要 4 个步骤。

第一步，求出反射光线的方向向量。这里我们以点光源为例，要求出反射光的方向，我们可以直接使用 GLSL 的内置函数 `reflect`，这个函数能够返回一个向量相对于某个法向量的反射向量，正好就是我们要的镜面反射结果。

复制代码

```
1 // 求光源与点坐标的方向向量
2 vec3 dir = (viewMatrix * vec4(pointLightPosition, 1.0)).xyz - vPos;
3
4 // 归一化
5 dir = normalize(dir);
6
7 // 求反射向量
8 vec3 reflectionLight = reflect(-dir, vNormal);
```

第二步，我们要根据相机位置计算视线与反射光线夹角的余弦，用到原理是向量的点乘。

复制代码

```
1 vec3 eyeDirection = vCameraPos - vPos;
2 eyeDirection = normalize(eyeDirection);
3 // 与视线夹角余弦
4 float eyeCos = max(dot(eyeDirection, reflectionLight), 0.0);
```

第三步，我们使用系数和指数函数设置镜面反射强度。指数越大，镜面越聚焦，高光的光斑范围就越小。这里，我们指数取 50.0，系数取 2.0。系数能改变反射亮度，系数越大，反射的亮度就越高。

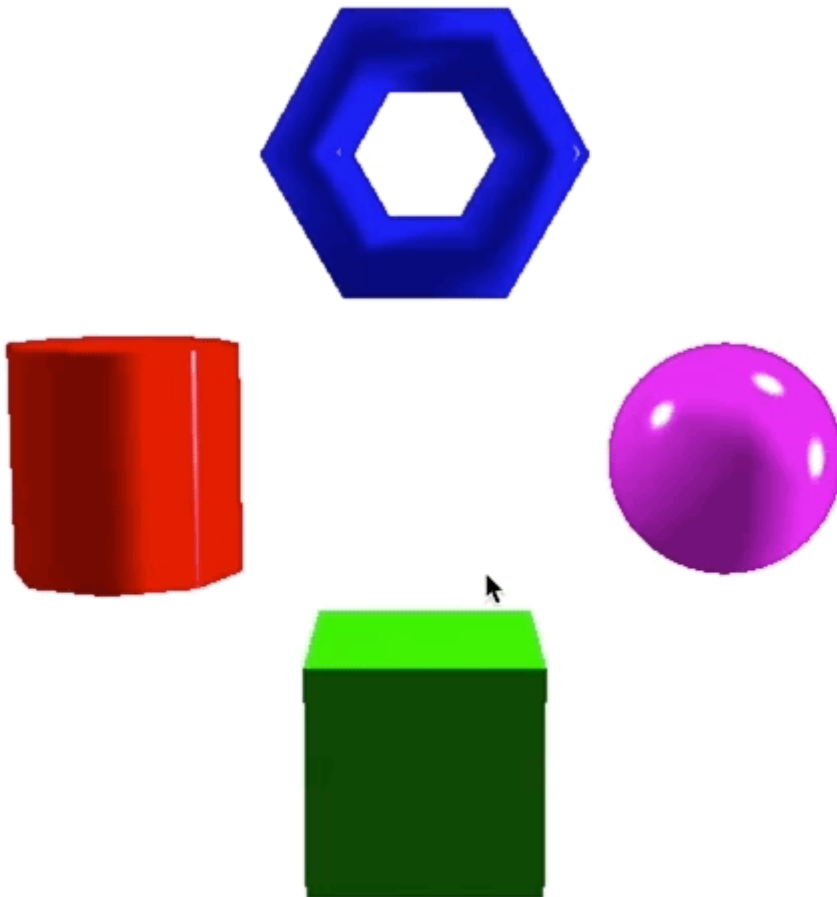
[复制代码](#)

```
1 float specular = 2.0 * pow(eyeCos, 50.0);
```

最后，我们将漫反射和镜面反射结合起来，就会让距离光源近的物体上，形成光斑。

[复制代码](#)

```
1 // 合成颜色
2 gl_FragColor.rgb = specular + (ambientLight + diffuse) * materialReflection;
3 gl_FragColor.a = 1.0;
```



上面的代码是以点光源为例来实现的光斑，其实只要有向光，都可以用同样的方法求出镜面反射，只不过对应的入射光方向计算有所不同，也就是着色器代码中的 `dir` 变量计算方式不一样。你可以利用我上节课讲的内容，自己动手试试。

如何实现完整的 Phong 反射模型？

那在自然界中，除了环境光以外，其他每种光源在空间中都可以存在不止一个，而且因为几何体材质不同，物体表面也可能既出现漫反射，又出现镜面反射。

可能出现的情况这么多，分析和计算起来也会非常复杂。为了方便处理，我们可以把多种光源和不同材质结合起来，形成标准的反射模型，这一模型被称为 **Phong 反射模型**。

Phong 反射模型的完整公式如下：

$$I_p = k_a i_a + \sum_{m \in \text{lights}} \left(k_d \left(\hat{L}_m \cdot \hat{N} \right) i_{m,d} + k_s \left(\hat{R}_m \cdot \hat{V} \right)^\alpha i_{m,s} \right)$$

公式里的 k_a 、 k_d 和 k_s 分别对应环境反射系数、漫反射系数和镜面反射系数。 \hat{L}_m 是入射光， N 是法向量， \hat{R}_m 是反射光， V 是视线向量。 i 是强度，漫反射和镜面反射的强度可考虑因为距离的衰减。 α 是和物体材质有关的常量，决定了镜面高光的范围。

根据上面的公式，我们把多个光照的计算结果相加，就能得到光照下几何体的最终颜色了。不过，这里的 Phong 反射模型实际上是真实物理世界光照的简化模型，因为它只考虑光源的光作用于物体，没有考虑各个物体之间的反射光。所以我们最终实现出的效果也只是自然界效果的一种近似，不过这种近似也高度符合真实情况了。

在一般的图形库或者图形框架中，会提供符合 Phong 反射模型的物体材质，比如 ThreeJS 中，就支持各种光源和反射材质。

下面，我们来实现一下完整的 Phong 反射模型。它可以帮助你在这个模型有更深入的理解，让你以后使用 ThreeJS 等其他图形库，也能够更加得心应手。整个过程分为三步：定义光源模型、定义几何体材质和实现着色器。

1. 定义光源模型

我们先来定义光源模型对象。环境光比较特殊，我们将它单独抽象出来，放在一个 ambientLight 的属性中，而其他的光源一共有 5 个属性与材质无关，我列了一张表放在了下面。

属性（作用）/光源	点光源	平行光	聚光灯
direction 方向 (定义光照方向)	无	有	有
position 位置 (定义光源位置)	有	无	有
color 颜色 (定义光的颜色)	有	有	有
decay 衰减 (光强度随着距离而减小)	有	无	有
angle 角度 (光传播的角度范围)	无	无	有



这样，我们就可以定义一个 Phong 类。这个类由一个环境光属性和其他三种光源的集合组合而成，表示一个可以添加和删除光源的对象。它的主要作用是添加和删除光源，并把光源的属性通过 uniforms 访问器属性转换成对应的 uniform 变量，主要的代码如下：

[复制代码](#)


```

1 class Phong {
2   constructor(ambientLight = [0.5, 0.5, 0.5]) {
3     this.ambientLight = ambientLight;
4     this.directionalLights = new Set();
5     this.pointLights = new Set();
6     this.spotLights = new Set();
7   }
8
9   addLight(light) {
10    const {position, direction, color, decay, angle} = light;
11    if(!position && !direction) throw new TypeError('invalid light');
12    light.color = color || [1, 1, 1];
13    if(!position) this.directionalLights.add(light);
14    else {
15      light.decay = decay || [0, 0, 1];
16      if(!angle) {
17        this.pointLights.add(light);
18      } else {
19        this.spotLights.add(light);
20      }
21    }
22  }
23
24  removeLight(light) {
25    if(this.directionalLights.has(light)) this.directionalLights.delete(light);
26    else if(this.pointLights.has(light)) this.pointLights.delete(light);
27    else if(this.spotLights.has(light)) this.spotLights.delete(light);

```

```
28   }
29
30   get uniforms() {
31     const MAX_LIGHT_COUNT = 16; // 最多每种光源设置16个
32     this._lightData = this._lightData || {};
33     const lightData = this._lightData;
34
35     lightData.directionalLightDirection = lightData.directionalLightDirection
36     lightData.directionalLightColor = lightData.directionalLightColor || {value: new Float32Array(3)}
37
38     lightData.pointLightPosition = lightData.pointLightPosition || {value: new Float32Array(3)}
39     lightData.pointLightColor = lightData.pointLightColor || {value: new Float32Array(3)}
40     lightData.pointLightDecay = lightData.pointLightDecay || {value: new Float32Array(1)}
41
42     lightData.spotLightDirection = lightData.spotLightDirection || {value: new Float32Array(3)}
43     lightData.spotLightPosition = lightData.spotLightPosition || {value: new Float32Array(3)}
44     lightData.spotLightColor = lightData.spotLightColor || {value: new Float32Array(3)}
45     lightData.spotLightDecay = lightData.spotLightDecay || {value: new Float32Array(1)}
46     lightData.spotLightAngle = lightData.spotLightAngle || {value: new Float32Array(1)}
47
48     [...this.directionalLights].forEach((light, idx) => {
49       lightData.directionalLightDirection.value.set(light.direction, idx * 3);
50       lightData.directionalLightColor.value.set(light.color, idx * 3);
51     });
52
53     [...this.pointLights].forEach((light, idx) => {
54       lightData.pointLightPosition.value.set(light.position, idx * 3);
55       lightData.pointLightColor.value.set(light.color, idx * 3);
56       lightData.pointLightDecay.value.set(light.decay, idx * 3);
57     });
58
59     [...this.spotLights].forEach((light, idx) => {
60       lightData.spotLightPosition.value.set(light.position, idx * 3);
61       lightData.spotLightColor.value.set(light.color, idx * 3);
62       lightData.spotLightDecay.value.set(light.decay, idx * 3);
63       lightData.spotLightDirection.value.set(light.direction, idx * 3);
64       lightData.spotLightAngle.value[idx] = light.angle;
65     });
66
67     return {
68       ambientLight: {value: this.ambientLight},
69       ...lightData,
70     };
71   }
72 }
```

有了这个类之后，我们就可以创建并添加各种光源了。我在下面的代码中，添加了一个平行光和两个点光源，你可以看看。

 复制代码

```
1  const phong = new Phong();
2  // 添加一个平行光
3  phong.addLight({
4    direction: [-1, 0, 0],
5  });
6  // 添加两个点光源
7  phong.addLight({
8    position: [-3, 3, 0],
9    color: [1, 0, 0],
10 });
11
12 phong.addLight({
13   position: [3, 3, 0],
14   color: [0, 0, 1],
15 });
```


2. 定义几何体材质

定义完光源之后，我们还需要定义几何体的**材质**（material），因为几何体材质决定了光反射的性质。

在前面的课程里，我们已经了解了一种与几何体材质有关的变量，即物体的反射率（MaterialReflection）。那在前面计算镜面反射的公式，`float specular = 2.0 * pow(eyeCos, 50.0);`中也有两个常量 2.0 和 50.0，把它们也提取出来，我们就能得到两个新的变量。其中，2.0 对应 `specularFactor`，表示镜面反射强度，50.0 指的是 `shininess`，表示镜面反射的光洁度。

这样，我们就有了 3 个与材质有关的变量，分别是 `matrrialReflection`（材质反射率）、`specularFactor`（镜面反射强度）、以及 `shininess`（镜面反射光洁度）。

然后，我们可以创建一个 `Matrrial` 类，来定义物体的材质。与光源类相比，这个类非常简单，只是设置这三个参数，并通过 `uniforms` 访问器属性，获得它的 `uniform` 数据结构形式。

 复制代码

```
1  class Material {
2    constructor(reflection, specularFactor = 0, shininess = 50) {
3      this.reflection = reflection;
4      this.specularFactor = specularFactor;
5      this.shininess = shininess;
```



```
6    }
7
8    get uniforms() {
9        return {
10            materialReflection: {value: this.reflection},
11            specularFactor: {value: this.specularFactor},
12            shininess: {value: this.shininess},
13        };
14    }
15 }
16
```

那么, 我们就可以创建 `material` 对象了。这里, 我一共创建 4 个 `material` 对象, 分别对应要显示四个几何体的材质。

[复制代码](#)

```
1 const material1 = new Material(new Color('#0000ff'), 2.0);
2 const material2 = new Material(new Color('#ff00ff'), 2.0);
3 const material3 = new Material(new Color('#008000'), 2.0);
4 const material4 = new Material(new Color('#ff0000'), 2.0);
```

有了 `phong` 对象和 `material` 对象, 我们就可以给几何体创建 `WebGL` 程序了。那我们就使用上面四个 `WebGL` 程序, 来创建真正的几何体网格, 并将它们渲染出来吧。具体代码如下:

[复制代码](#)

```
1 const program1 = new Program(gl, {
2     vertex,
3     fragment,
4     uniforms: {
5         ...material1.uniforms,
6         ...phong.uniforms,
7     },
8 });
9 const program2 = new Program(gl, {
10    vertex,
11    fragment,
12    uniforms: {
13        ...material2.uniforms,
14        ...phong.uniforms,
15    },
16 });
17 const program3 = new Program(gl, {
18    vertex,
```

```
19     fragment,
20     uniforms: {
21         ...material3.uniforms,
22         ...phong.uniforms,
23     },
24 });
25 const program4 = new Program(gl, {
26     vertex,
27     fragment,
28     uniforms: {
29         ...material4.uniforms,
30         ...phong.uniforms,
31     },
32
```

3. 实现着色器

接下来，我们重点看一下，支持 phong 反射模型的片元着色器代码是怎么实现的。这个着色器代码比较复杂，我们一段一段来看。

首先，我们来看光照相关的 uniform 变量的声明。这里，我们声明了 vec3 和 float 数组，数组的大小为 16。这样，对于每一种光源，我们都可以支持 16 个。

[复制代码](#)

```
1  #define MAX_LIGHT_COUNT 16
2  uniform mat4 viewMatrix;
3
4  uniform vec3 ambientLight;
5  uniform vec3 directionalLightDirection[MAX_LIGHT_COUNT];
6  uniform vec3 directionalLightColor[MAX_LIGHT_COUNT];
7  uniform vec3 pointLightColor[MAX_LIGHT_COUNT];
8  uniform vec3 pointLightPosition[MAX_LIGHT_COUNT];
9  uniform vec3 pointLightDecay[MAX_LIGHT_COUNT];
10 uniform vec3 spotLightColor[MAX_LIGHT_COUNT];
11 uniform vec3 spotLightDirection[MAX_LIGHT_COUNT];
12 uniform vec3 spotLightPosition[MAX_LIGHT_COUNT];
13 uniform vec3 spotLightDecay[MAX_LIGHT_COUNT];
14 uniform float spotLightAngle[MAX_LIGHT_COUNT];
15
16 uniform vec3 materialReflection;
17 uniform float shininess;
18 uniform float specularFactor;
```

接下来, 我们实现计算 phong 反射模型的主题逻辑。事实上, 处理平行光、点光源、聚光灯的主体逻辑类似, 都是循环处理每个光源, 再计算入射光方向, 然后计算漫反射以及镜面反射, 最终将结果返回。

[复制代码](#)

```
1 float getSpecular(vec3 dir, vec3 normal, vec3 eye) {
2     vec3 reflectionLight = reflect(-dir, normal);
3     float eyeCos = max(dot(eye, reflectionLight), 0.0);
4     return specularFactor * pow(eyeCos, shininess);
5 }
6
7 vec4 phongReflection(vec3 pos, vec3 normal, vec3 eye) {
8     float specular = 0.0;
9     vec3 diffuse = vec3(0);
10
11     // 处理平行光
12     for(int i = 0; i < MAX_LIGHT_COUNT; i++) {
13         vec3 dir = directionalLightDirection[i];
14         if(dir.x == 0.0 && dir.y == 0.0 && dir.z == 0.0) continue;
15         vec4 d = viewMatrix * vec4(dir, 0.0);
16         dir = normalize(-d.xyz);
17         float cos = max(dot(dir, normal), 0.0);
18         diffuse += cos * directionalLightColor[i];
19         specular += getSpecular(dir, normal, eye);
20     }
21
22     // 处理点光源
23     for(int i = 0; i < MAX_LIGHT_COUNT; i++) {
24         vec3 decay = pointLightDecay[i];
25         if(decay.x == 0.0 && decay.y == 0.0 && decay.z == 0.0) continue;
26         vec3 dir = (viewMatrix * vec4(pointLightPosition[i], 1.0)).xyz - pos;
27         float dis = length(dir);
28         dir = normalize(dir);
29         float cos = max(dot(dir, normal), 0.0);
30         float d = min(1.0, 1.0 / (decay.x * pow(dis, 2.0) + decay.y * dis + decay.z));
31         diffuse += d * cos * pointLightColor[i];
32         specular += getSpecular(dir, normal, eye);
33     }
34
35     // 处理聚光灯
36     for(int i = 0; i < MAX_LIGHT_COUNT; i++) {
37         vec3 decay = spotLightDecay[i];
38         if(decay.x == 0.0 && decay.y == 0.0 && decay.z == 0.0) continue;
39
40         vec3 dir = (viewMatrix * vec4(spotLightPosition[i], 1.0)).xyz - pos;
41         float dis = length(dir);
42         dir = normalize(dir);
43
44         // 聚光灯的朝向
```

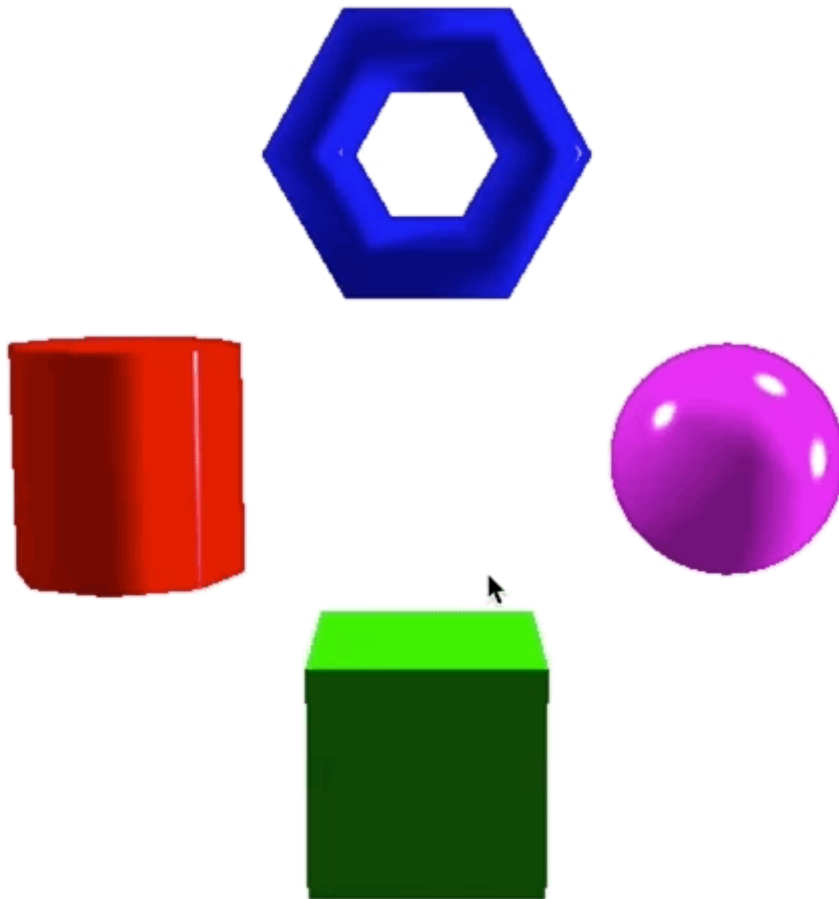
```
45     vec3 spotDir = (viewMatrix * vec4(spotLightDirection[i], 0.0)).xyz;
46     // 通过余弦值判断夹角范围
47     float ang = cos(spotLightAngle[i]);
48     float r = step(ang, dot(dir, normalize(-spotDir)));
49
50     float cos = max(dot(dir, normal), 0.0);
51     float d = min(1.0, 1.0 / (decay.x * pow(dis, 2.0) + decay.y * dis + decay.z));
52     diffuse += r * d * cos * spotLightColor[i];
53     specular += r * getSpecular(dir, normal, eye);
54 }
55
56 return vec4(diffuse, specular);
57
```

最后，我们在 main 函数中，调用 phongReflection 函数来合成颜色。代码如下：

[复制代码](#)

```
1 void main() {
2     vec3 eyeDirection = normalize(vCameraPos - vPos);
3     vec4 phong = phongReflection(vPos, vNormal, eyeDirection);
4
5     // 合成颜色
6     gl_FragColor.rgb = phong.w + (phong.xyz + ambientLight) * materialReflection;
7     gl_FragColor.a = 1.0;
8 }
```

最终呈现的视觉效果如下图所示：



你注意一下上图右侧的球体。因为我们一共设置了 3 个光源，一个平行光、两个点光源，它们都能够产生镜面反射。所以，这些光源叠加在一起后，这个球体就呈现出 3 个镜面高光。

Phong 反射模型的局限性

虽然，phong 反射模型已经比较接近于真实的物理模型，不过它仍然是真实模型的一种近似。因为它没有考虑物体反射光对其他物体的影响，也没有考虑物体对光线遮挡产生的阴影。

当然，我们可以完善这个模型。比如，将物体本身反射光（主要是镜面反射光）对其他物体的影响纳入到模型中。另外，我们也要考虑物体的阴影。当我们把这些因素更多地考虑进去的时候，我们的模型就会更加接近真实世界的物理模型。

当我们渲染 3D 图形的时候，要呈现越接近真实的效果，往往要考虑更多的参数，因此所需的计算量也越大，那我们就需要有更强的渲染能力，比如，更好的显卡，更快的 CPU 和 GPU，并且也需要我们尽可能地优化计算的性能。

但是，有很多时候，我们需要在细节和性能上做出平衡和取舍。那性能优化的部分，也是我们课程的重点，我会在性能篇详细来讲。这节课，我们就重点关注反射模型，总结出完整的 Phong 反射模型就可以了。

要点总结

今天，我们把环境光、平行光、点光源、聚光灯这四种光源整合，并且在上节课讲的漫反射的基础上，添加了镜面反射，形成了完整的 Phong 反射模型。在这里，我们实现的着色器代码能够结合四种光源的效果，除了环境光外，每种光源还可以设置多个。

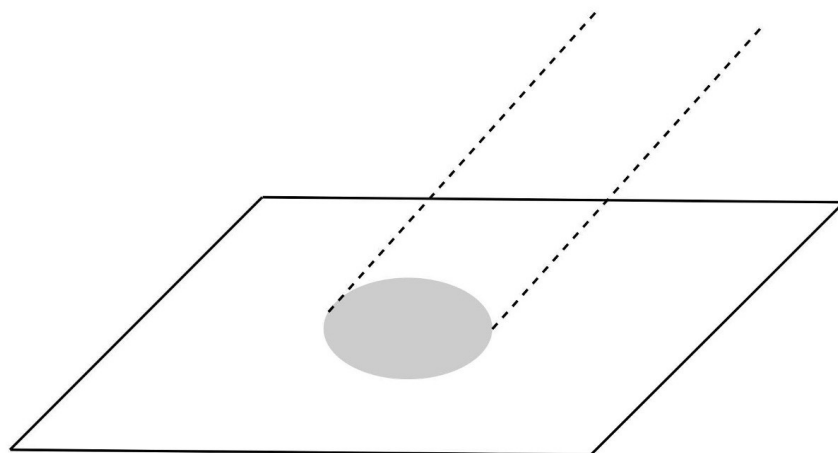
在 Phong 反射模型中，光照在物体上的最终效果，由各个光源的性质（参数）和物体的表面材质共同决定。

Phong 反射模型也只是真实世界的一种近似，因为我们并没有考虑物体之间反射光的相互影响，也没有考虑光线的遮挡。如果把这些因素考虑进去，那我们的模型可以更接近真实世界了。

小试牛刀

我们知道，平行光、点光源和聚光灯是三种常见的方向光，但真实世界还有其他的方向光，比如探照灯，它是一种有范围的平行光，类似于聚光灯，但又不完全一样。你能给物体实现探照灯效果吗？

这里，我先把需要用到的参数告诉你，包括光源方向 `searchLightDirection`、光源半径 `searchLightRadius`、光源位置 `searchLightPosition`、光照颜色 `searchLightColor`。你可以用 OGL 实现探照灯效果，然后把对应的着色器代码写在留言区。



探照灯示意图

而且，探照灯的光照截面不一定是圆形，也可以是其他图形，比如三角形、菱形、正方形，你也可以试着让它支持不同的光照截面。

欢迎在留言区和我讨论，分享你的答案和思考，也欢迎你把这节课分享给你的朋友，我们下节课再见！

源码

课程中完整代码详见 [🔗 GitHub 仓库](#)

推荐阅读

[🔗 Phong 反射模型简介](#)

提建议

跟月影学可视化

系统掌握图形学与可视化核心原理

月影

奇虎 360 奇舞团团长

可视化 UI 框架 SpriteJS 核心开发者



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | 如何模拟光照让3D场景更逼真？（上）

下一篇 25 | 如何用法线贴图模拟真实物体表面

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。