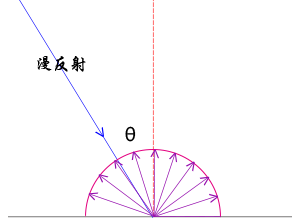
**概念**

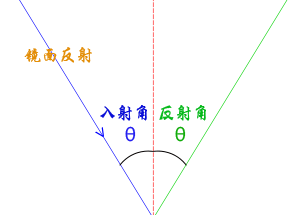
1. 漫反射



光照到物体上，有一部分光会进行漫反射，即在各个方向进行反射

漫反射光的颜色 = 几何体表面基色 x 光线颜色 x 光线入射角余弦值

1. 镜面反射



光照到物体上，有一部分光会进行镜面反射

镜面反射光的颜色 = 几何体表面基色 x 光线颜色 x 视线与反射光线的夹角余弦值

1. 环境光照

光照到大气上会出现漫反射，这就成为了环境光，环境光没有光源

环境反射光颜色 = 几何体表面基色 x 环境光颜色

1. 真实物体的颜色

真实物体的颜色=漫反射光的颜色+镜面反射光的颜色+环境反射光颜色

**法向量**

垂直与面的线称为法向量，光线与法向量的夹角就是入射角

**漫反射光的颜色示例**

1. 着色器

    <!-- 顶点着色器 -->

    <script id="vertexShader" type="x-shader/x-vertex">

        attribute vec4 a\_position;//attribute声明vec4类型变量apos

        attribute vec4 a\_color;// attribute声明顶点颜色变量

        attribute vec4 a\_normal;//顶点法向量变量

        uniform vec3 u\_lightColor;// uniform声明平行光颜色变量

        uniform vec3 u\_lightPosition;// 光源位置

        uniform mat4 u\_matrix;  // 物体移动矩阵

        varying vec4 v\_color;//varying声明顶点颜色插值后变量

        void main(){

            // 位置

            gl\_Position = u\_matrix\*a\_position;

            // 计算物体旋转后的法向量

            vec3 normal = normalize((u\_matrix\*a\_normal).xyz);

            // 计算光源与照射点的单位向量

            vec3 lightDirection = normalize(vec3(gl\_Position) - u\_lightPosition);

            // 计算法向量与光向量的余弦值

            float dot = max(dot(lightDirection, normal), 0.0);

            // 漫反射颜色 = 光源颜色 \* 物体颜色 \* 余弦值

            vec3 reflectedLight = u\_lightColor \* a\_color.rgb \* dot;

            // 得出点漫反射后的颜色

            v\_color = vec4(reflectedLight, a\_color.a);

        }

    </script>

    <!-- 片元着色器 -->

    <script id="fragmentShader" type="x-shader/x-fragment">

        precision mediump float;

        varying vec4 v\_color;

        void main() {

            gl\_FragColor =  v\_color;

        }

    </script>

normalize归一化，该函数使向量变成长度为1的单位向量

dot函数用于计算2个向量的点积，当2个向量的长度为1时，其值为余弦值

1. 建立点数据

        // 指定绘图的顶点坐标和对应的纹理坐标

        const points = [

            -0.5,-0.5,0.0, 1.0,0.0,0.0,1.0, 0.0,0.0,1.0,

            -0.5,0.5,0.0,  1.0,0.0,0.0,1.0, 0.0,0.0,1.0,

            0.5,-0.5,0.0,  1.0,0.0,0.0,1.0, 0.0,0.0,1.0,

        ];

        const source = new Float32Array(points);

前3个为点坐标，中间4个为颜色，后面3个为点的法向量

**整体代码**

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <script src="http://www.yanhuangxueyuan.com/versions/threejsR92/build/three.js"></script>

</head>

<body style="margin: 0px;">

    <canvas id="canvas"></canvas>

    <!-- 顶点着色器 -->

    <script id="vertexShader" type="x-shader/x-vertex">

        attribute vec4 a\_position;//attribute声明vec4类型变量apos

        attribute vec4 a\_color;// attribute声明顶点颜色变量

        attribute vec4 a\_normal;//顶点法向量变量

        uniform vec3 u\_lightColor;// uniform声明平行光颜色变量

        uniform vec3 u\_lightPosition;// 光源位置

        uniform mat4 u\_matrix;  // 物体移动矩阵

        varying vec4 v\_color;//varying声明顶点颜色插值后变量

        void main(){

            // 位置

            gl\_Position = u\_matrix\*a\_position;

            // 计算物体旋转后的法向量

            vec3 normal = normalize((u\_matrix\*a\_normal).xyz);

            // 计算光源与照射点的单位向量

            vec3 lightDirection = normalize(vec3(gl\_Position) - u\_lightPosition);

            // 计算法向量与光向量的余弦值

            float dot = max(dot(lightDirection, normal), 0.0);

            // 漫反射颜色 = 光源颜色 \* 物体颜色 \* 余弦值

            vec3 reflectedLight = u\_lightColor \* a\_color.rgb \* dot;

            // 得出点漫反射后的颜色

            v\_color = vec4(reflectedLight, a\_color.a);

        }

    </script>

    <!-- 片元着色器 -->

    <script id="fragmentShader" type="x-shader/x-fragment">

        precision mediump float;

        varying vec4 v\_color;

        void main() {

            gl\_FragColor =  v\_color;

        }

    </script>

    <script>

        // 初始化

        function init(context) {

            //创建程序对象

            const program = context.createProgram();

            //创建顶点着色器

            const vertexShader = context.createShader(context.VERTEX\_SHADER);

            // 获取顶点着色器代码

            const vsSource = document.getElementById('vertexShader').innerText;

            //将着色器源文件传入着色器对象中

            context.shaderSource(vertexShader, vsSource);

            //编译着色器对象

            context.compileShader(vertexShader);

            //把顶点着色对象装进程序对象中

            context.attachShader(program, vertexShader);

            //创建片元着色器

            const fragmentShader = context.createShader(context.FRAGMENT\_SHADER);

            // 获取片元着色器代码

            const fsSource = document.getElementById('fragmentShader').innerText;

            //将着色器源文件传入着色器对象中

            context.shaderSource(fragmentShader, fsSource);

            //编译着色器对象

            context.compileShader(fragmentShader)

            //把片元着色对象装进程序对象中

            context.attachShader(program, fragmentShader);

            //连接webgl上下文对象和程序对象

            context.linkProgram(program);

            //启动程序对象

            context.useProgram(program);

            //将程序对象挂到上下文对象上

            context.program = program;

        }

        // canvas画布

        const canvas = document.getElementById('canvas');

        canvas.width = document.body.clientWidth;

        canvas.height = document.body.clientHeight;

        // webgl上下文

        const context = canvas.getContext('webgl');

        // 初始化上下文

        init(context);

        // 指定将要用来清理绘图区的颜色

        context.clearColor(0.0, 0.0, 0.0, 1.0);

        // 清理绘图区

        context.clear(context.COLOR\_BUFFER\_BIT);

        const a\_position = context.getAttribLocation(context.program, 'a\_position');

        const a\_color = context.getAttribLocation(context.program, 'a\_color');

        const a\_normal = context.getAttribLocation(context.program, 'a\_normal');

        const u\_lightColor = context.getUniformLocation(context.program, 'u\_lightColor');

        const u\_lightPosition = context.getUniformLocation(context.program, 'u\_lightPosition');

        const u\_matrix = context.getUniformLocation(context.program, 'u\_matrix');

        context.uniform3f(u\_lightColor, 1.0, 1.0, 1.0);

        context.uniform3f(u\_lightPosition, 0.5, -0.5, -1.0);

        //模型矩阵

        const modelMatrix = new THREE.Matrix4();

        const mx = new THREE.Matrix4().makeRotationX(0.0);

        const my = new THREE.Matrix4().makeRotationY(0.3);

        const mat = modelMatrix.multiply(my.multiply(mx));

        context.uniformMatrix4fv(u\_matrix, false, mat.elements);

        //缓冲对象

        const sourceBuffer = context.createBuffer();

        //绑定缓冲对象

        context.bindBuffer(context.ARRAY\_BUFFER, sourceBuffer);

        // 指定绘图的顶点坐标和对应的纹理坐标

        const points = [

            -0.5,-0.5,0.0, 1.0,0.0,0.0,1.0, 0.0,0.0,1.0,

            -0.5,0.5,0.0,  1.0,0.0,0.0,1.0, 0.0,0.0,1.0,

            0.5,-0.5,0.0,  1.0,0.0,0.0,1.0, 0.0,0.0,1.0,

        ];

        const source = new Float32Array(points);

        // 写入数据

        context.bufferData(context.ARRAY\_BUFFER, source, context.STATIC\_DRAW);

        // 设置a\_Position的指针属性，以便其能够从缓存中正确提前数据

        context.vertexAttribPointer(

            a\_position,

            3,

            context.FLOAT,

            false,

            10 \* 4,

            0 \* 4

        );

        // 赋能-批处理

        context.enableVertexAttribArray(a\_position);

        // 设置a\_Pin的指针属性，以便其能够从缓存中正确提前数据

        context.vertexAttribPointer(

            a\_color,

            4,

            context.FLOAT,

            false,

            10 \* 4,

            3 \* 4

        )

        //赋能-批处理

        context.enableVertexAttribArray(a\_color);

        // 设置a\_Pin的指针属性，以便其能够从缓存中正确提前数据

        context.vertexAttribPointer(

            a\_normal,

            3,

            context.FLOAT,

            false,

            10 \* 4,

            7 \* 4

        )

        //赋能-批处理

        context.enableVertexAttribArray(a\_normal);

        //渲染

        context.clear(context.COLOR\_BUFFER\_BIT);

        context.drawArrays(context.TRIANGLES, 0, 3);

    </script>

</body>

</html>