

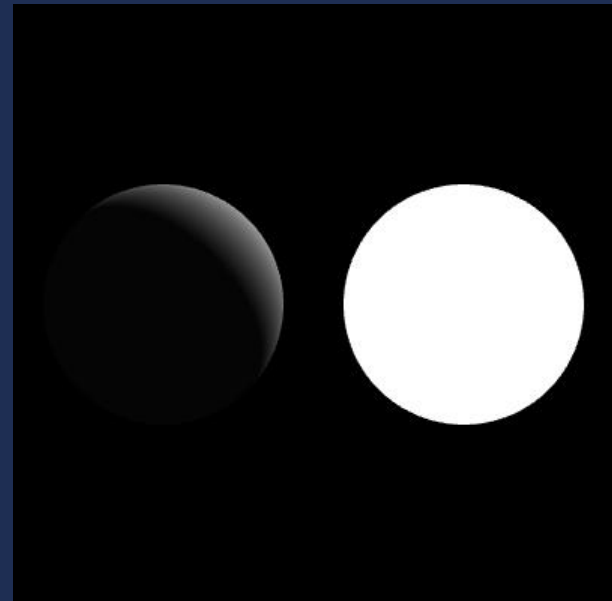


OpenGL 编程

光照效果

光照效果

- 从生理学的角度上讲，眼睛之所以看见各种物体，是因为光线直接或间接的从它们那里到达了眼睛
- 人类对于光线强弱的变化的反应，比对于颜色变化的反应来得灵敏，光线很大程度上表现了物体的立体感
 - 无任何光照效果
 - 使用简单光照效果



光照效果

- OpenGL对于光照效果提供了直接的支持
 - 调用某些函数，便可实现简单的光照效果
- 基本知识：
 - 建立光照模型
 - 法线向量
 - 控制光源
 - 控制材质
 - 选择光照模型



建立光照模型

- 画面的形成
 - 某些物体本身就会发光
 - 而其它物体虽然不会发光，但可以反射来自其它物体的光
 - 光通过各种方式传播，最后进入人的眼睛
- 计算机很难准确模拟各种光线的传播
- 无需精确的模拟各种光线，只需找到一种近似的计算方式，使最终结果让我们认为它是真实的

建立光照模型

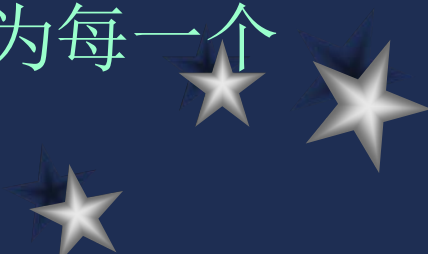
- OpenGL在处理光照时采用如下近似:
 - 光照系统分为三部分
 - 光源：光的来源
 - 材质：接受光照的各种物体的表面，材质特点决定了物体反射光线的特点
 - 光照环境：一些额外的参数，将影响最终的光照画面

建立光照模型

- 光线的传播
 - 镜面反射
 - 漫反射
- 物体在反射光线时，可看成是这两种反射的叠加
- 光照效果形成的因素：
 - 光源发出的光线，可以分别设置其经过镜面反射和漫反射后的光线强度
 - 对于被光线照射的材质，也可以分别设置光线经过镜面反射和漫反射后的光线强度

法线向量

- 法线方向用一个向量来表示：
 - 对于指定某物体，指定光源后，根据反射定律可计算出光的反射方向，进而计算出光照效果的画面
- OpenGL并不会自动计算出多边形所构成的物体的表面的每个点的法线：
 - 为了实现光照效果，在代码中为每一个顶点指定其法线向量



法线向量

- 指定法线向量的方式:
 - 与指定颜色的方式有雷同之处
 - 在指定颜色时，只需指定每一个顶点的颜色，OpenGL就可以自行计算顶点之间的其它点的颜色
 - 颜色一旦被指定，除非再指定新的颜色，否则以后指定的所有顶点都将以这一向量作为自己的颜色
 - 用glNormal*函数可指定法线向量（单位）
 - glTranslate*或glRotate*函数不会改变法线向量, glScale*函数很可能导致法线向量的不正确
 - OpenGL提供了修正措施，但由此也带来了各种开销
 - 在使用法线向量的时，尽量避免使用glScale*函数
 - 即使使用，也最好保证各坐标轴进行等比例缩放

控制光源

- OpenGL仅仅支持有限数量的光源
 - GL_LIGHT0到GL_LIGHT7
 - 使用glEnable函数可以开启它们
 - glEnable(GL_LIGHT0)，可以开启第0号光源
 - 开启过多光源会导致程序运行速度的下降
- 成百上千的电灯：近似手段进行编程
- 每一个光源都可以设置其属性，可通过glLight*函数完成
 - 第一个参数指明是设置哪一个光源的属性
 - 第二个参数指明是设置该光源的哪一个属性
 - 第三个参数则是指明把该属性值设置成多少

控制光源

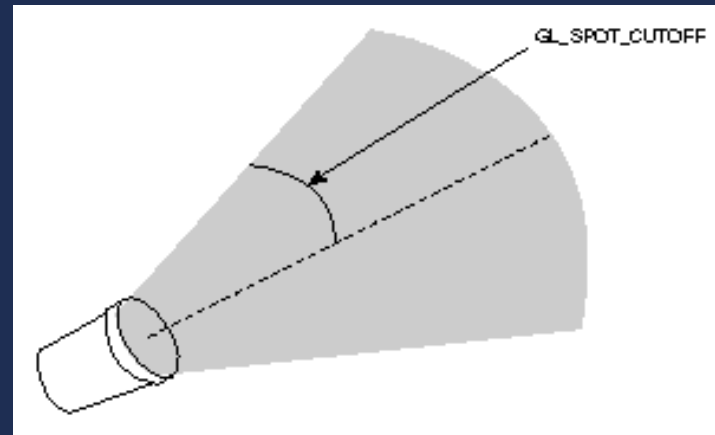
- GL_AMBIENT/DIFFUSE/SPECULAR属性
 - 表示了光源所发出的光的反射特性（颜色），每个属性由四个值表示：RGBA
 - GL_AMBIENT：表示该光源所发出的光，经过非常多次的反射后，最终遗留在整个光照环境中的强度（颜色）
 - GL_DIFFUSE：表示该光源所发出的光，照射到粗糙表面时经过漫反射，所得到光的强度
 - GL_SPECULAR：表示该光源所发出的光，照射到光滑表面时经过镜面反射，所得到的光的强度

控制光源

- GL_POSITION属性：光源所在位置
 - 由四个值 (X, Y, Z, W) 表示
 - 方向性光源：W为零，则表示该光源位于无限远处，前三个值表示了它所在的方向
 - 位置性光源：W不为零， X/W 表示光源位置，各种矩阵变换函数如`glRotate*`等同样有效
 - 方向性光源计算快：在视觉效果允许的情况下，应该尽可能的使用方向性光源
- 方向性光源：不会用到后两类属性

控制光源

- GL_SPOT_DIRECTION、GL_SPOT_EXPONENT、GL_SPOT_CUTOFF属性
 - 将光源作为聚光灯使用（位置性光源）
 - 比如手电筒
 - GL_SPOT_DIRECTION属性
 - 由三个值组成一个向量，即光源发射的方向



控制光源

- GL_SPOT_EXPONENT属性
 - 只有一个表示聚光程度的值：零，表示光照范围内向各方向发射的光线强度相同；正数，表示光照向中央集中
 - 数值越大，聚光效果就越明显
- GL_SPOT_CUTOFF属性
 - 只有一个表示角度的值，它是光源发射光线所覆盖角度的一半，其取值范围在0到90之间，也可以取180这个特殊值



控制光源

- GL_CONSTANT (LINEAR、QUADRATIC)_ATTENUATION属性
 - 只对位置性光源有效：表示了光源所发出的光线的直线传播特性
 - 光线的强度随着距离的增加而减弱，OpenGL将趋势抽象成函数：
衰减因子= $1/(k_1+k_2*d+k_3*d^2)$ d表示距离
 - 初始强度乘衰减因子，得到对应距离的光线强度
 - 设置三常数，可控制光线在传播过程中减弱趋势

控制材质

- 材质也需设置众多属性，用glMaterial*函数来设置，该函数有三个参数：
 - 第一个参数表示指定哪一面的属性，可以是GL_FRONT、GL_BACK或者GL_FRONT_AND_BACK
 - 第二、第三个参数与glLight*函数的第二、三个参数作用类似

控制材质

- glMaterial*函数可指定的材质属性
 - GL_AMBIENT、DIFFUSE、SPECULAR属性
 - GL_AMBIENT: 各种光线照射到该材质上, 经过多次反射后最终遗留在环境中的光线强度
 - GL_DIFFUSE: 光线照射到该材质上, 经过漫反射后形成的光线强度
 - GL_SPECULAR: 光线照射到该材质上, 经过镜面反射后形成的光线强度
 - GL_AMBIENT和GL_DIFFUSE都取相同的值, 可以达到比较真实的效果
 - 使用GL_AMBIENT_AND_DIFFUSE可以同时设置GL_AMBIENT和GL_DIFFUSE属性

控制材质

- glMaterial*函数
 - GL_SHININESS属性：镜面指数(单值)，取值范围是0到128，该值越小，表示材质越粗糙
 - GL_EMISSION属性：由四个值组成，表示一种颜色，OpenGL认为该材质本身就微微的向外发射光线
 - GL_COLOR_INDEXES属性：仅在颜色索引模式下使用，使用复杂且使用范围较小



选择光照模型

- 光照模型包括四个部分的内容
 - 全局环境光线的强度
 - 视点位置是在较近位置还是在无限远处
 - 物体正面与背面是否分别计算光照
 - 镜面颜色（即GL_SPECULAR属性所指定的颜色）的计算是否从其它光照计算中分离出来，并在纹理操作以后在进行应用



选择光照模型

- 通过函数glLightModel*设置（两个参数：属性+属性值）
 - GL_LIGHT_MODEL_AMBIENT表示全局环境光线强度，由四个值组成
 - GL_LIGHT_MODEL_LOCAL_VIEWER
 - GL_TRUE：表示近处观看
 - GL_FALSE：表示在远处观看
 - GL_LIGHT_MODEL_TWO_SIDE
 - GL_TRUE：执行双面光照计算
 - GL_FALSE：不执行双面光照计算

选择光照模型

- 通过函数glLightModel*设置
 - GL_LIGHT_MODEL_COLOR_CONTROL表示颜色计算方式：
 - GL_SINGLE_COLOR: 表示按通常顺序操作，先计算光照，再计算纹理
 - GL_SEPARATE_SPECULAR_COLOR, 表示将GL_SPECULAR属性分离出来，先计算光照的其它部分，待纹理操作完成后再计算GL_SPECULAR
 - 后者通常可以使画面效果更为逼真

最后准备

- OpenGL默认是关闭光照处理

- 打开光照处理功能

- `glEnable(GL_LIGHTING)`

- 关闭光照处理功能

- `glDisable(GL_LIGHTING)`

- 举例



小结

- 介绍了OpenGL光照的基本知识
 - OpenGL把光照分解为光源、材质、光照模式三个部分，根据这三个部分的各种信息，以及物体表面的法线向量，可以计算得到最终的光照效果
 - 光源、材质和光照模式具有各自属性，用函数 `glLight*`, `glMaterial*`, `glLightModel*` 设置
 - 可使用多个光源来实现各种逼真的效果，但光源数量的增加将导致程序运行效率下降
 - 使用光照过程中，属性的种类和数量都非常繁多，需要很多的经验才能熟练的设置各种属性，从而形成逼真的光照效果



OpenGL 编程

显示列表

显示列表

- 面临的问题
 - 使用OpenGL，只要调用一系列的函数就可以了，但可能出现问题
 - 某个模型由数千个多边形来近似，为了产生这数千个多边形，需要不停地调用glVertex*函数
 - 如果我们需要每秒钟绘制60幅画面，则每秒调用的glVertex*函数次数就会超过数十万次，乃至接近百万次

显示列表

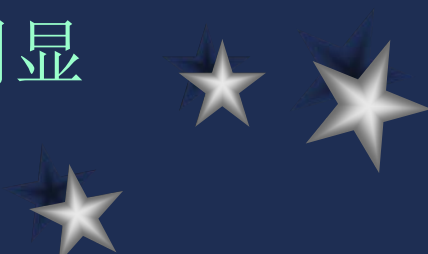
- 考虑这样一段代码

```
const int segments = 100;
const GLfloat pi = 3.14f;
int i;
glLineWidth(10.0);
glBegin(GL_LINE_LOOP);
for(i=0; i<segments; ++i)
{
    GLfloat tmp = 2*pi*i/segments;
    glVertex2f(cos(tmp), sin(tmp));
}
glEnd();
```



显示列表

- 绘制圆环的代码：
 - 如果我们在每次绘制图象时调用这段代码，则虽然可以达到绘制圆环的目的
 - 但是 \cos 、 \sin 等开销较大的函数被多次调用，浪费了CPU资源
 - 如果每一个顶点不是通过 \cos 、 \sin 等函数得到，而是使用更复杂的运算方式得到，则浪费的现象就更加明显



显示列表

- 上述两类问题的共同点
 - 程序多次执行了重复的工作，导致CPU资源浪费和运行速度的下降
- 显示列表：
 - 编写程序时，当遇到需要反复调用的一组函数，可创建一个显示列表，把这些函数装入其中，并在需要的地方调用这个显示列表



显示列表

- 显示列表作用：
 - 预先编译、解析需反复调用的函数，将结果存储在显示系统内部，以提高执行时的效率
 - 适合于显示列表中的函数较多或网络远程执行OpenGL时情景
- 使用显示列表的四个步骤：
 - 分配显示列表编号
 - 创建显示列表
 - 调用显示列表
 - 销毁显示列表

分配显示列表编号

- OpenGL允许多个显示列表同时存在
 - 不同的显示列表用不同的正整数来区分
- 使用glGenLists函数来自动分配一个没有使用的显示列表编号
 - 该函数有一个参数i，表示要分配i个连续的未使用的显示列表编号
 - 返回的是分配的若干连续编号中最小的一个



分配显示列表编号

- glGenLists函数:
 - glGenLists(3)
 - 如果返回20, 则表示分配了20、21、22这三个连续的编号
 - 如果函数返回零, 表示分配失败
- 使用glIsList函数判断一个编号是否已经被用作显示列表:
 - glIsList(i)



创建显示列表

- 创建显示列表
 - 把各种函数的调用装入到显示列表中
 - glBegin开始， glEnd结束装入
 - glBegin有两个参数
 - 第一个是一个正整数：装入到哪个显示列表
 - 第二个如果为GL_COMPILE，则表示以下的内
容只是装入到显示列表，但现在不执行它们；
如果为GL_COMPILE_AND_EXECUTE，表示在装
入的同时，把装入的内容执行一遍



创建显示列表

- 例如：需要把“设置颜色为红色，且指定一个坐标为(0, 0)的顶点”两条命令装入到编号为list的显示列表中，并且在装入的时候不执行：

```
glNewList(list, GL_COMPILE);  
glColor3f(1.0f, 0.0f, 0.0f);  
glVertex2f(0.0f, 0.0f);  
glEnd();
```



创建显示列表

- 显示列表只装入函数，而不能装入其它内容

```
int i=3;
glNewList(list, GL_COMPILE);
if(i>20)
    glColor3f(1.0f, 0.0f, 0.0f);
glVertex2f(0.0f, 0.0f);
glEnd();
```

- 其中if这个判断就没有被装入到显示列表
- 即使i>20的条件成立，glColor3f也不执行

创建显示列表

- 并非所有函数都可装入到显示列表中：
 - 用于查询的函数无法被装入到显示列表，`glCallList(s)` 不知道如何处理返回值。
 - 在网络方式下，设置客户端状态的函数也无法被装入到显示列表
 - 显示列表被保存到服务器端，设置客户端状态的函数在发送到客户端前已就被执行，客户端无法执行这些函数
 - 分配、创建、删除显示列表的动作也无法被装入到另一个显示列表

调用显示列表

- glCallList函数：调用一个显示列表
 - 单个参数：要调用的显示列表的编号
 - 例如，要调用编号为10的显示列表，直接使用glCallList(10)就可以了
- glCallLists函数：调用一系列
 - 该函数有三个参数
 - 第一个参数：要调用多少个显示列表
 - 第二个参数：显示列表编号的储存格式
 - 第三个参数：显示列表的编号所在的位置

调用显示列表

- 使用该函数前，需要用glListBase函数来设置一个偏移量
 - 假设偏移量为k，且glCallLists中要求调用的显示列表编号依次为11, 12, 13, ..., 则实际调用的显示列表为11+k, 12+k, 13+k, ...
 - ```
GLuint lists[]={1, 3, 4, 8};
glListBase(10);
glCallLists(4, GL_UNSIGNED_INT, lists);
```


则实际上调用的是编号为11, 13, 14, 18的四个显示列表

# 销毁显示列表

- 使用显示列表将会带来一些开销
  - 例如：把各种动作保存到显示列表中会占用一定数量的内存资源
- 销毁显示列表可以回收资源
- 使用glDeleteLists来销毁一串编号连续的显示列表
  - 例：使用glDeleteLists(20, 4);将销毁20, 21, 22, 23四个显示列表



# 销毁显示列表

- 如使用得当，可提升程序的性能
  - 明显的减少函数的调用次数（C/S模式）
  - 保存中间结果，避免一些不必要的计算(Sin)
  - 便于优化(旋转、平移和缩放操作)
- 可为程序的设计带来方便
  - 设置属性时，经常把相关的函数放在一起调用
  - 如把这些设置属性的操作装入到显示列表中，则可实现属性成组的切换
- 即使使用显示列表在某些情况下可以提高性能，但这种提高很可能并不明显 

# 销毁显示列表

---

- 举例：见文档！



# 小结

- 介绍了显示列表的知识和简单的应用

- 将各种函数装到显示列表中，以后调用相当于调用了一组函数；不能存放其它内容
- 使用显示列表的过程是：分配一个未使用的显示列表编号，把函数调用装入显示列表，调用显示列表，销毁显示列表
- 使用显示列表可能提高程序效率，但不一定会明显，显示列表本身也存在一定的开销



