- 为什么要研究真实感图形学？
- 真实感图形学研究什么？

❄21世纪，图形无所不在！

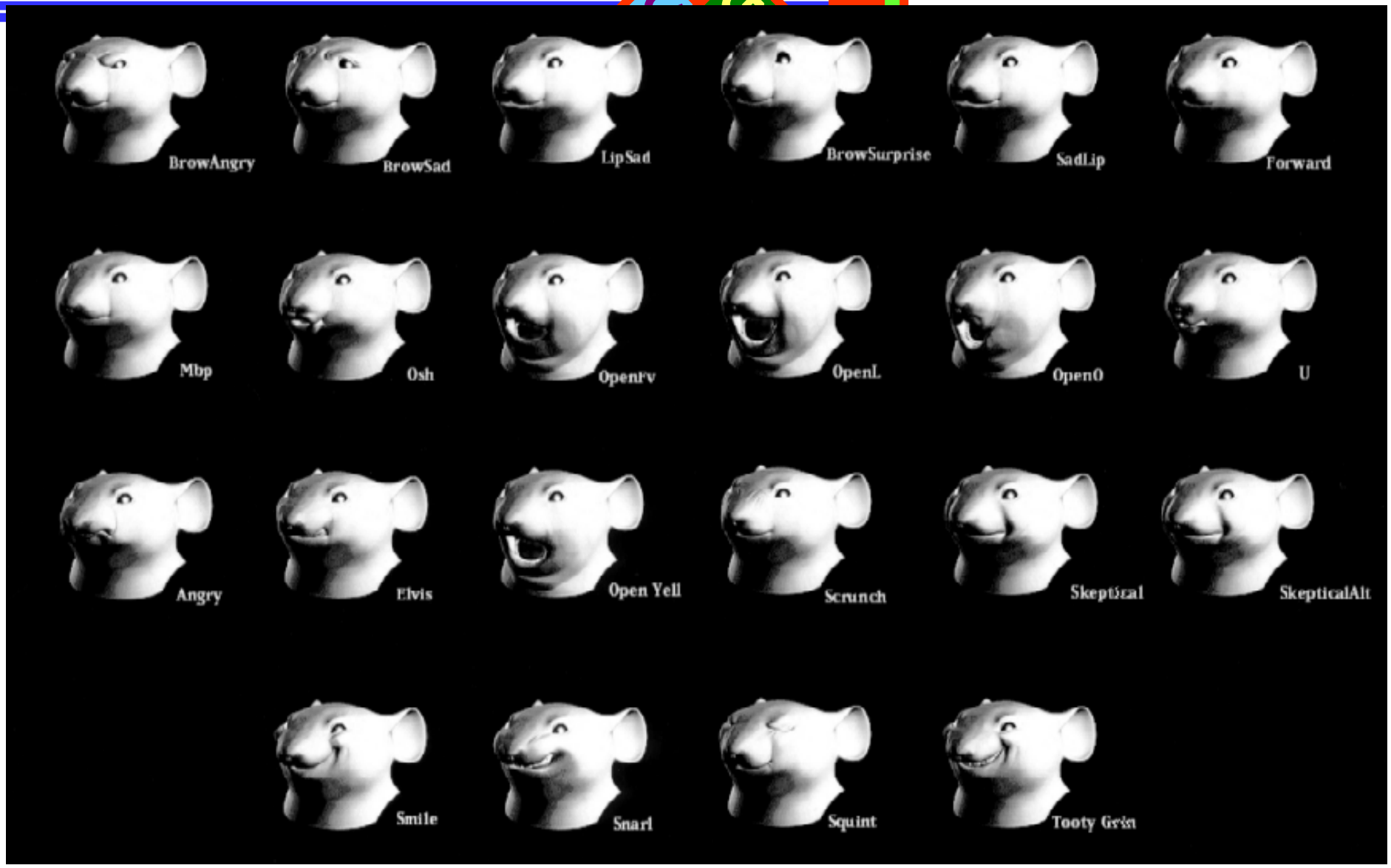
《泰坦尼克号》剧照

Aviator

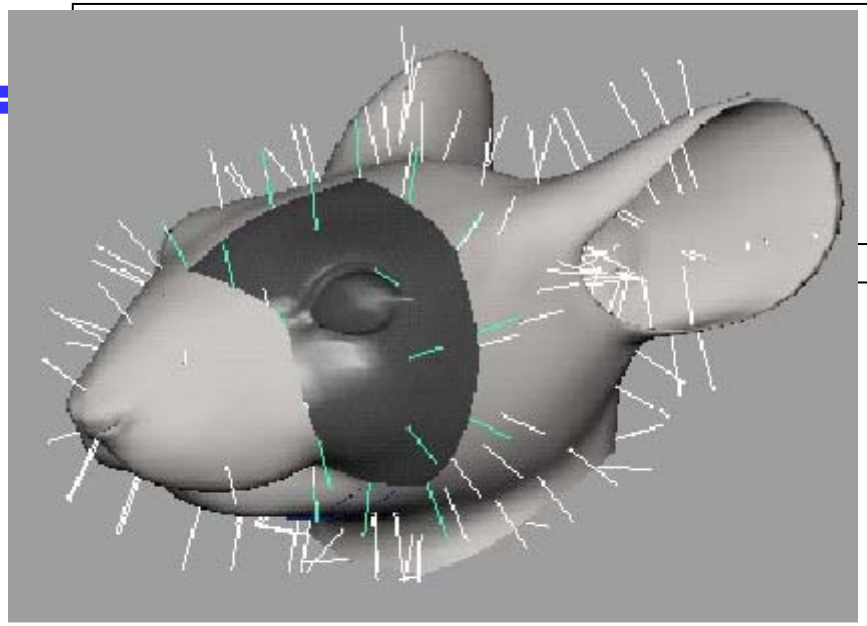
Matte Painting Sky

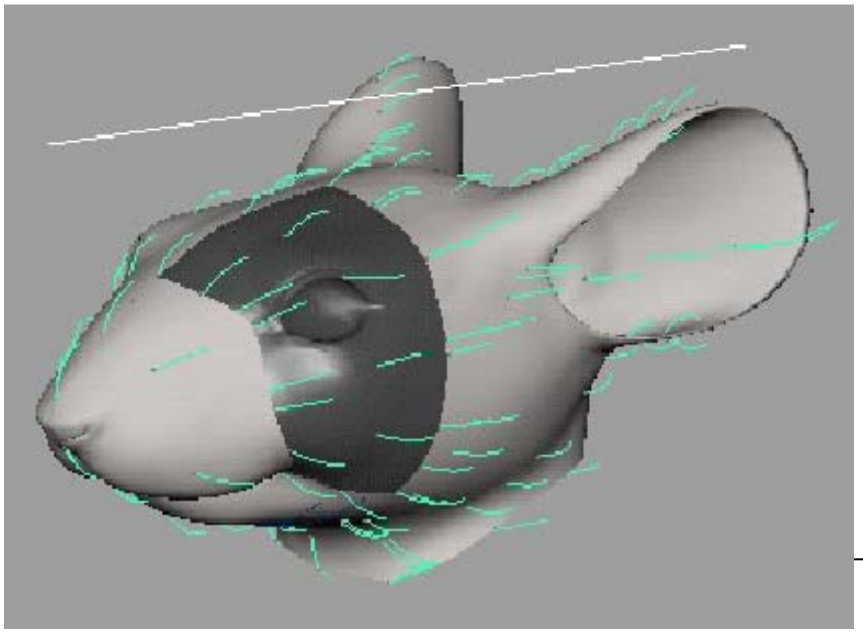
Matte Painting Ground

Staurt Little

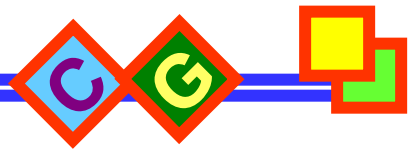


Stuart Little






ORANG

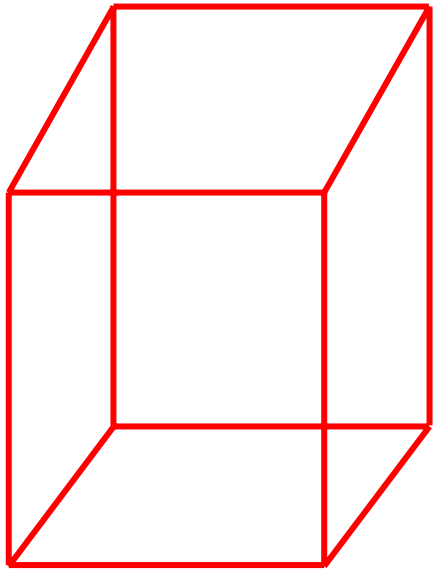


真实感图形的生成技术

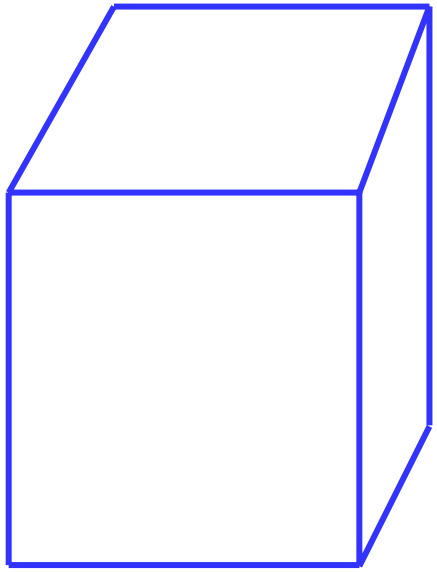
- 消隐技术
- 光照技术
- 物体表面细节的模拟
- 阴影的生成
- 图形反走样技术
- 用OpenGL生成真实感图形

问题的引出

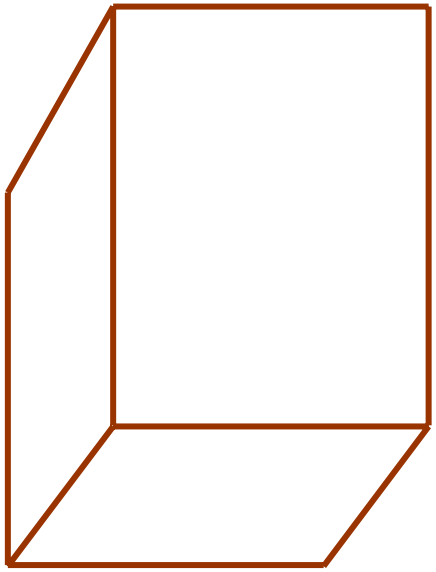
- 投影变换失去了深度信息，往往导致图形的**二义性**



线框图



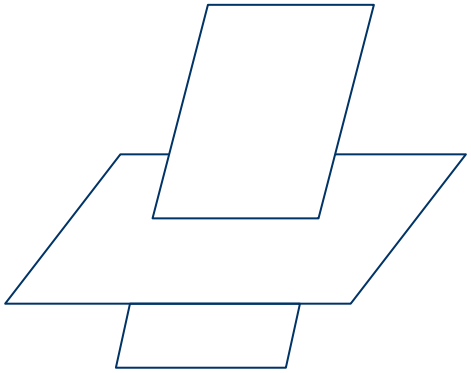
从右上往下看

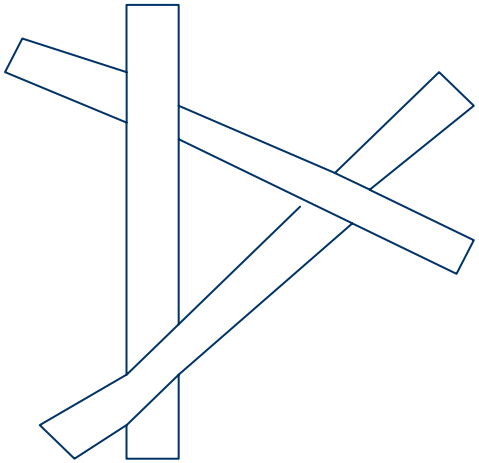


从左下往上看

问题的引出

- 此时，出现真实感图形学的概念。



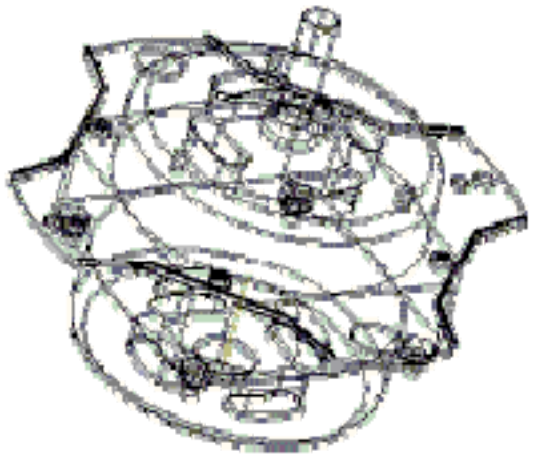


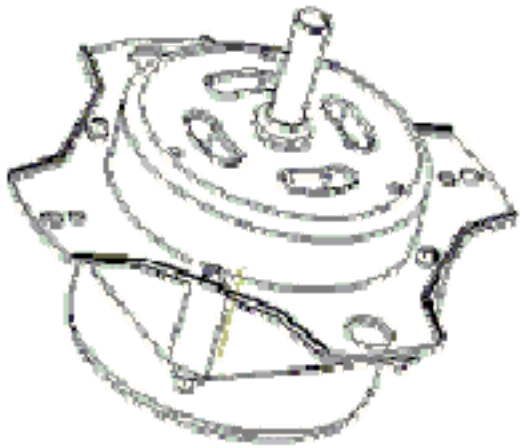
贯串与循环遮挡

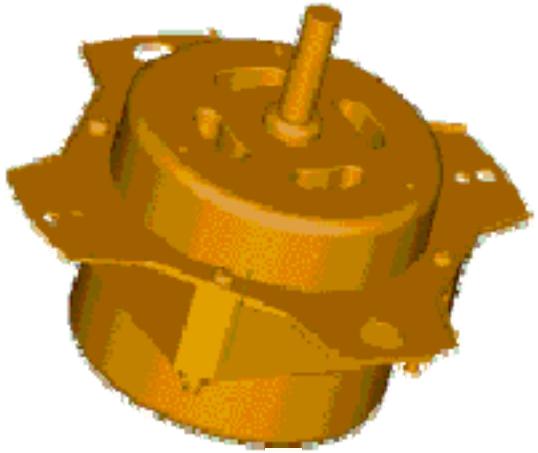
消隐：消除隐藏线、隐藏面

问题的引出

- 为了更加真实：引入灰度、颜色、绘制的概念







产生彩色

- 选择合适的颜色模型----RGB模型、CMY模型
- 为颜色模型中的每一种基色建立光照明方程

为了让物体更像真实的，加光源！

于是出现照明度模型：光照技术

简单光照明模型

当光照射到物体表面时，光线可能被吸收、反射和透射。被物体吸收的部分转化为热，反射、透射的光进入人的视觉系统，使我们能看见物体。为模拟这一现象，我们建立一些数学模型来替代复杂的物理模型，这些模型就称为明暗效应模型或者光照明模型。

三维形体的图形经过消隐后，再进行明暗效应的处理，就基本上可以实现图形的真实感显示。

简单光照明模型

简单光照明模型模拟物体表面对光的反射作用。光源被假定为点光源，反射作用被细分为镜面反射(Specular Reflection)和漫反射(Diffuse Reflection)。

简单光照明模型只考虑物体对直接光照的反射作用，而物体间的光反射作用，只用环境光(Ambient Light)来表示，Phong光照明模型就是这样的一种模型

Phong光照明模型

由物体表面上一点*P*反射到视点的光强I为环境光的反射光强*Ie*、理想漫反射光强*Id*、和镜面反射光*Is*的总和

$$I = I_a K_a + I_p K_d (L \cdot N) + I_p K_s (R \cdot V)^n$$

Phong模型示例_1

理想漫反射

+

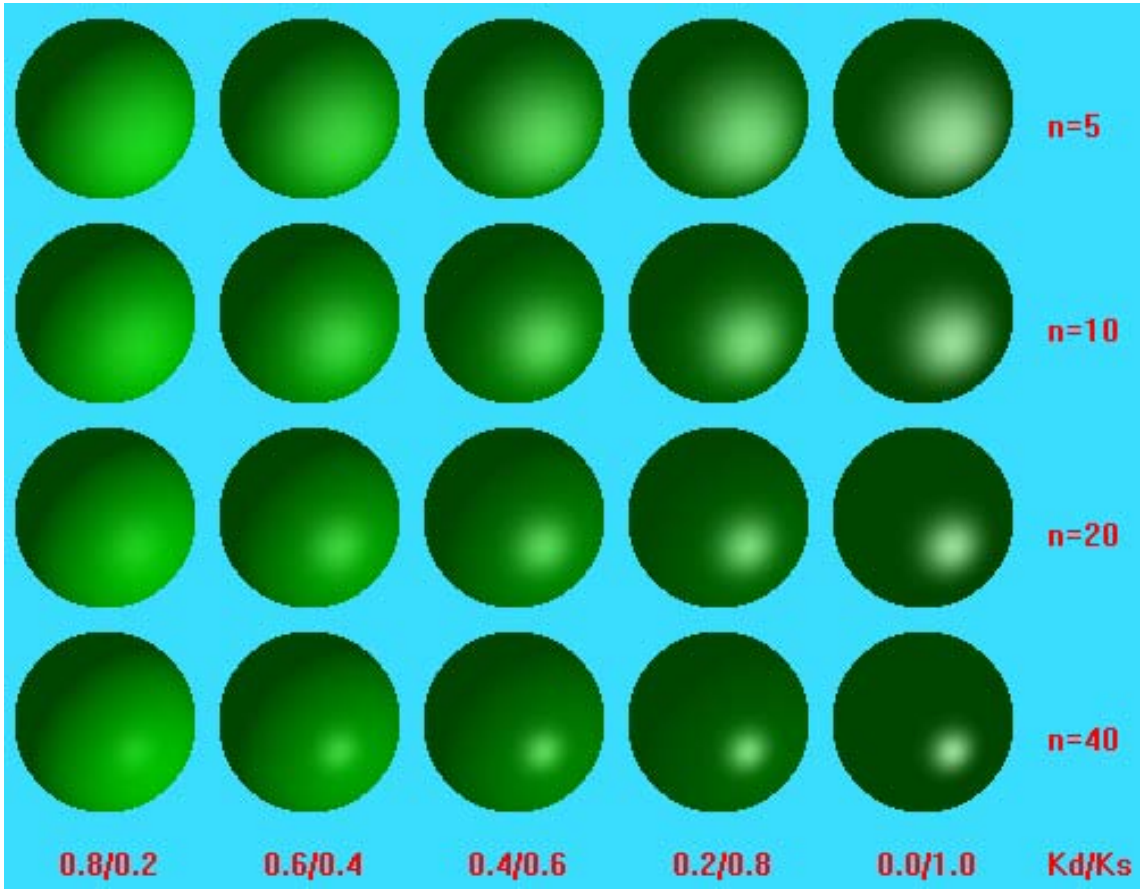
环境光

+

镜面反射

=

Phong模型示例_2



Phong光照明模型

Phong光照明模型是真实感图形学中提出的第一个有影响的光照明模型，生成图象的真实度已经达到可以接受的程度；但是在实际的应用中，由于它是一个经验模型，还具有以下的一些问题：

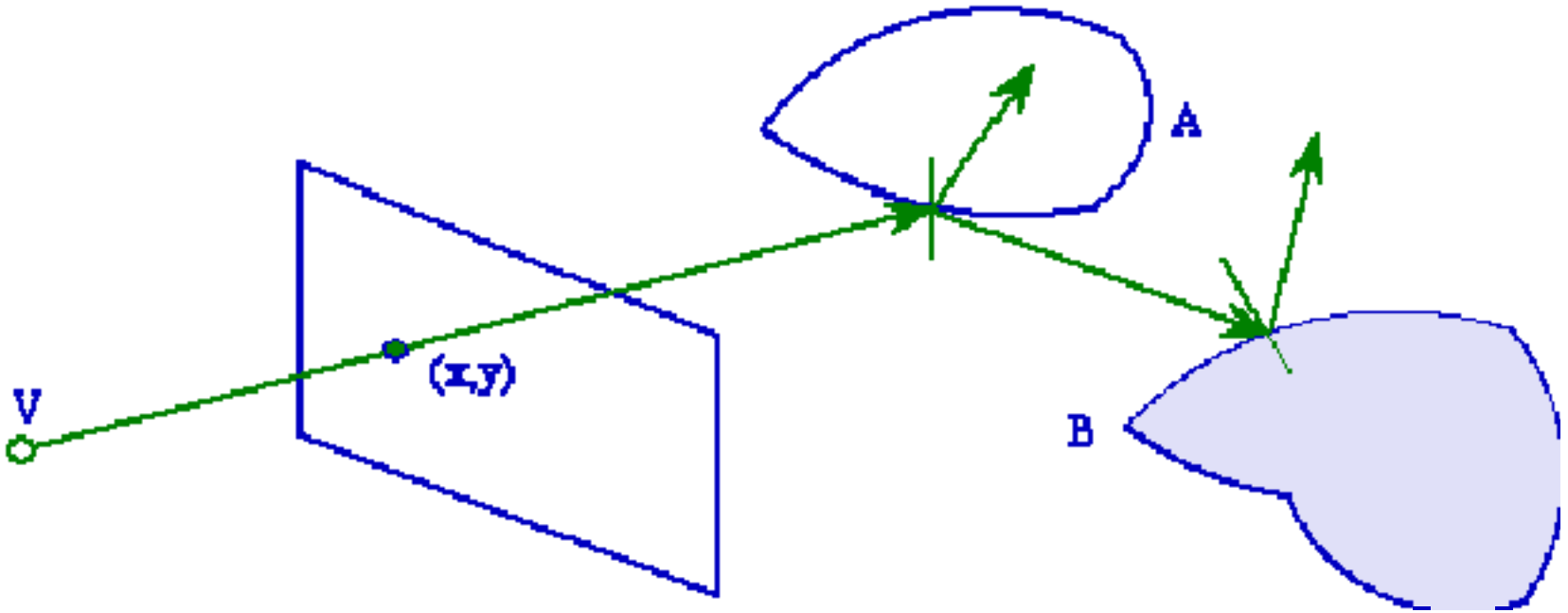
- 用Phong模型显示出的物体象塑料，没有质感；
- 环境光是常量，没有考虑物体之间相互的反射光；
- 镜面反射的颜色是光源的颜色，与物体的材料无关；
- 镜面反射的计算在入射角很大时会产生 失真等

前述几个模型，都只是处理光源直接照射物体表面的光强计算，不能很好的模拟光的折射、反射和阴影，必须要有一个更精确的光照明模型

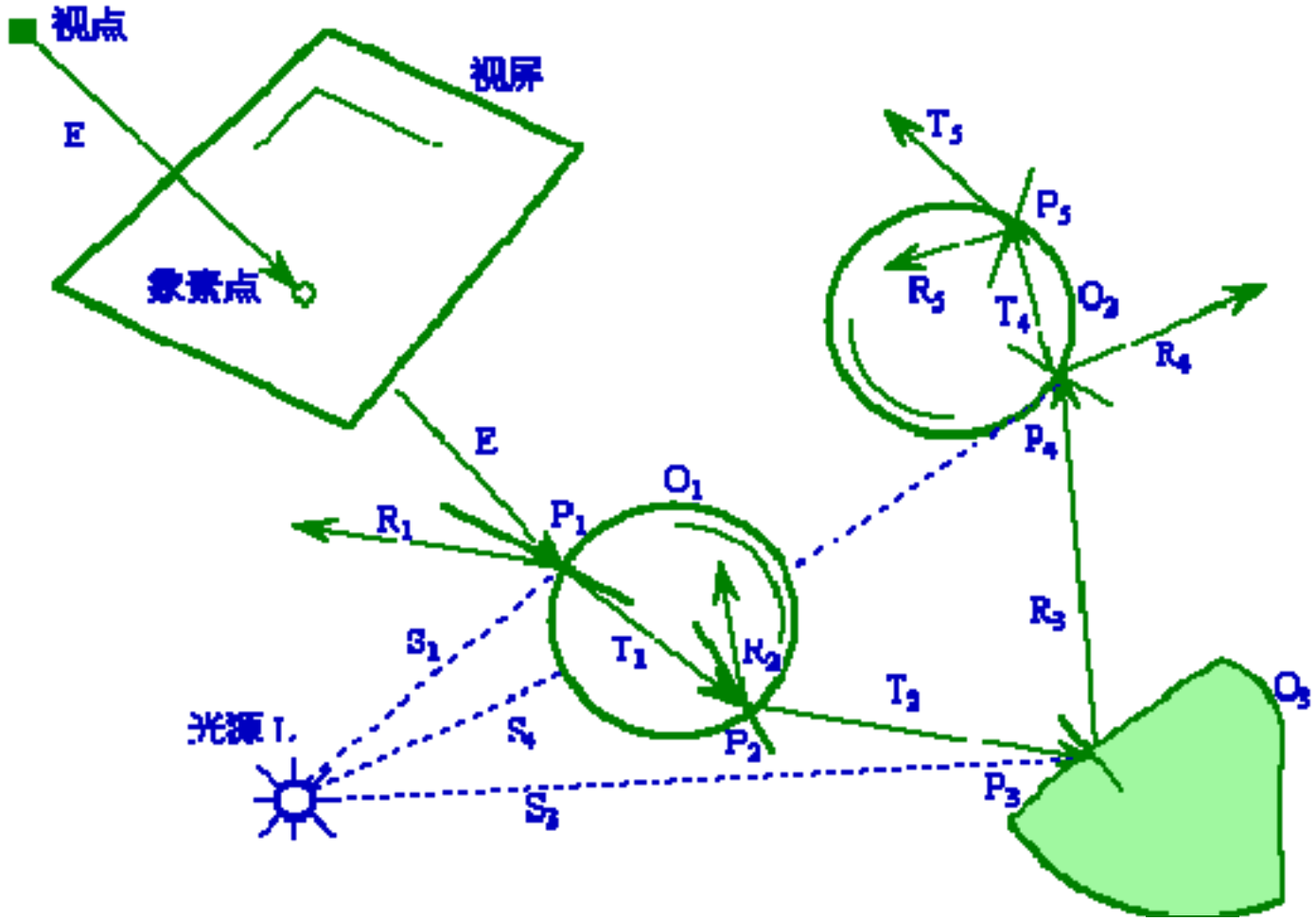
整体光照明模型

整体光照明模型就是这样的一种模型。在现有的整体光照明模型中，主要有光线跟踪和辐射度两种方法，它们是当今真实感图形学中最重要的两个图形绘制技术，在CAD及图形学领域得到了广泛的应用

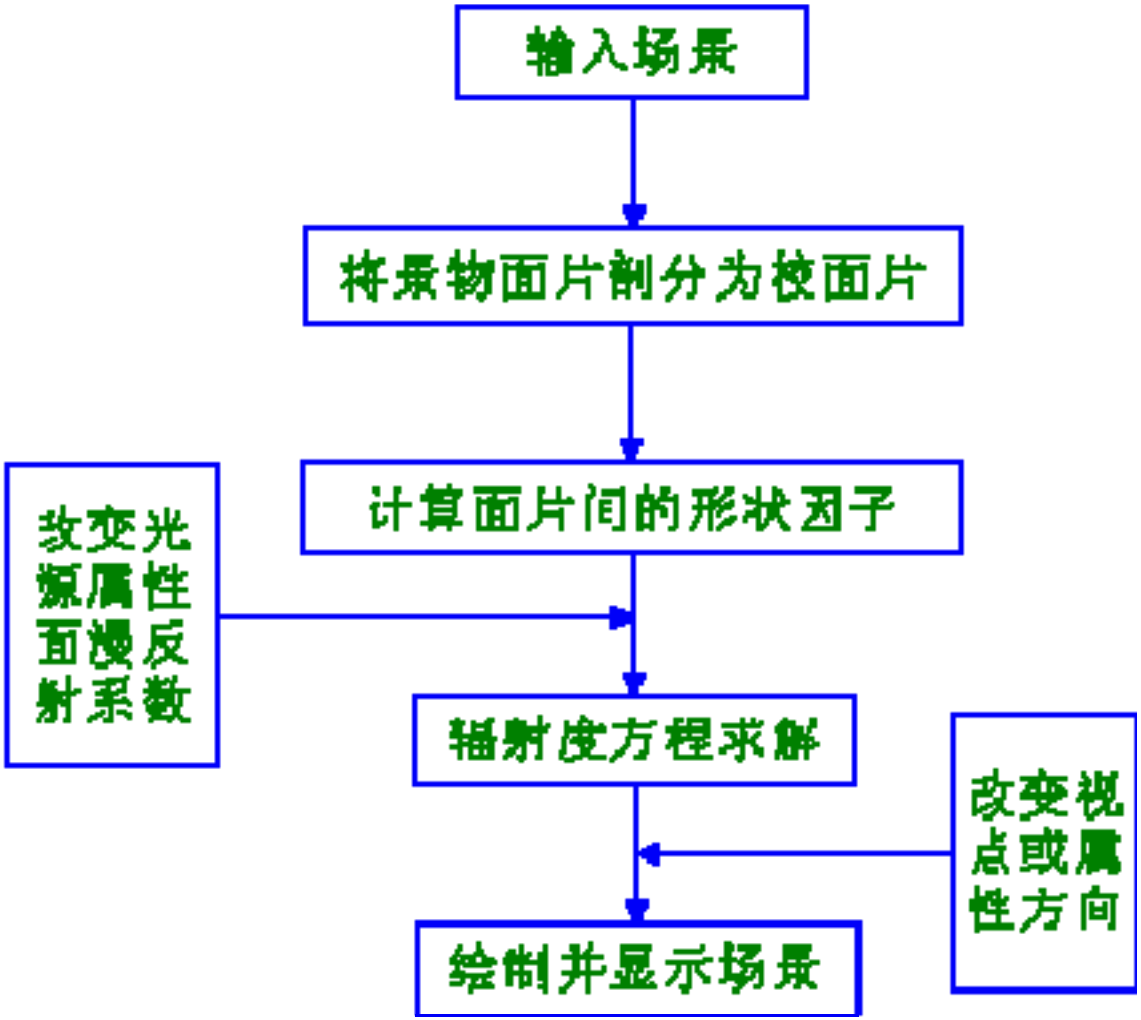
光线跟踪(Ray Tracing)的基本原理



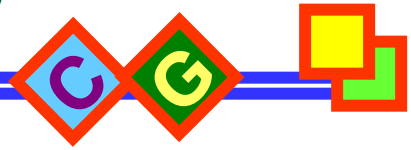
光线跟踪的基本过程


视点
视屏
E
象素点
E
O_1
P_1
R_1
S_1
T_1
K_2
P_2
S_4
光源 I.
S_3
T_2
P_3
O_3
R_3
P_4
O_2
T_4
R_4
T_5
P_5
R_5

辐射度方法(Radioactivity)



模拟物体表面细节



由于前面生成的物体其表面过于光滑和单调，看起来反而不真实，这是因为在现实世界中的物体，其表面通常有它的表面细节，即各种纹理，如刨光的木材表面上有木纹，建筑物墙壁上有装饰图案，机器外壳表面有文字说明它的名称、型号等。它们是通过颜色色彩或明暗度变化体现出来的表面细节，这种纹理称为颜色纹理。另一类纹理则是由于不规则的细小凹凸造成的，例如桔子皮表面的皱纹。称为几何纹理。

- 表面细节多边形
- 纹理映射
- 法向扰动法

Bump texture

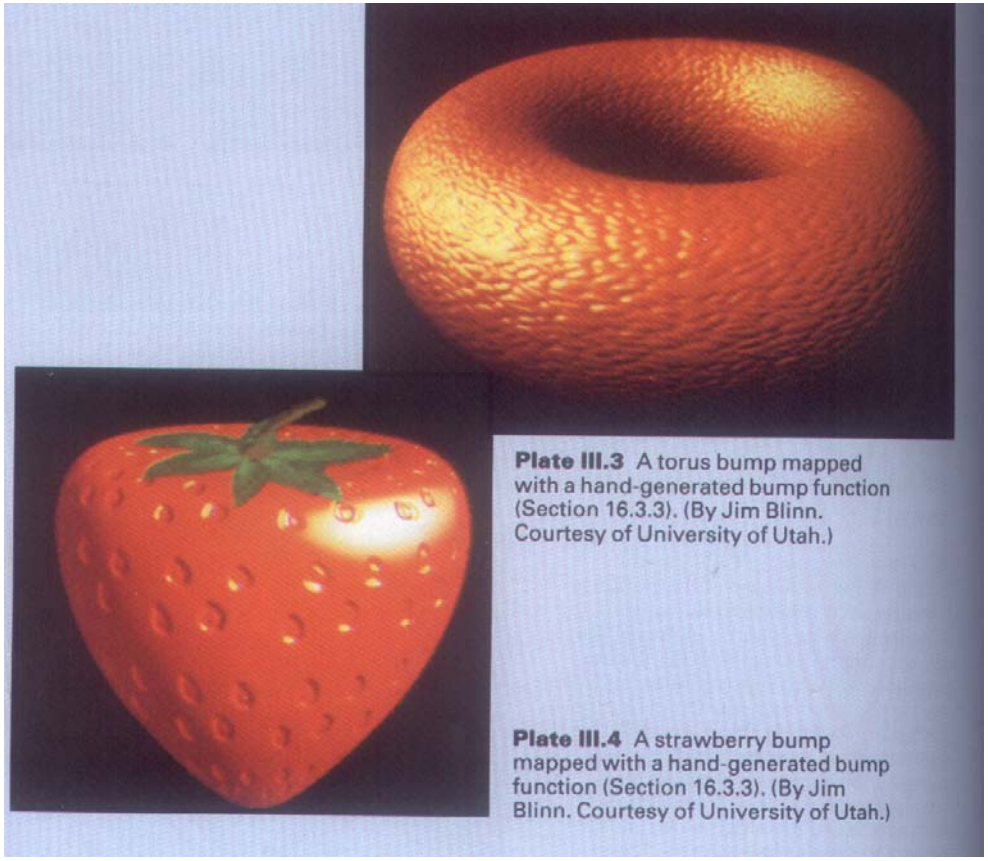


Plate III.3 A torus bump mapped with a hand-generated bump function (Section 16.3.3). (By Jim Blinn. Courtesy of University of Utah.)

Plate III.4 A strawberry bump mapped with a hand-generated bump function (Section 16.3.3). (By Jim Blinn. Courtesy of University of Utah.)

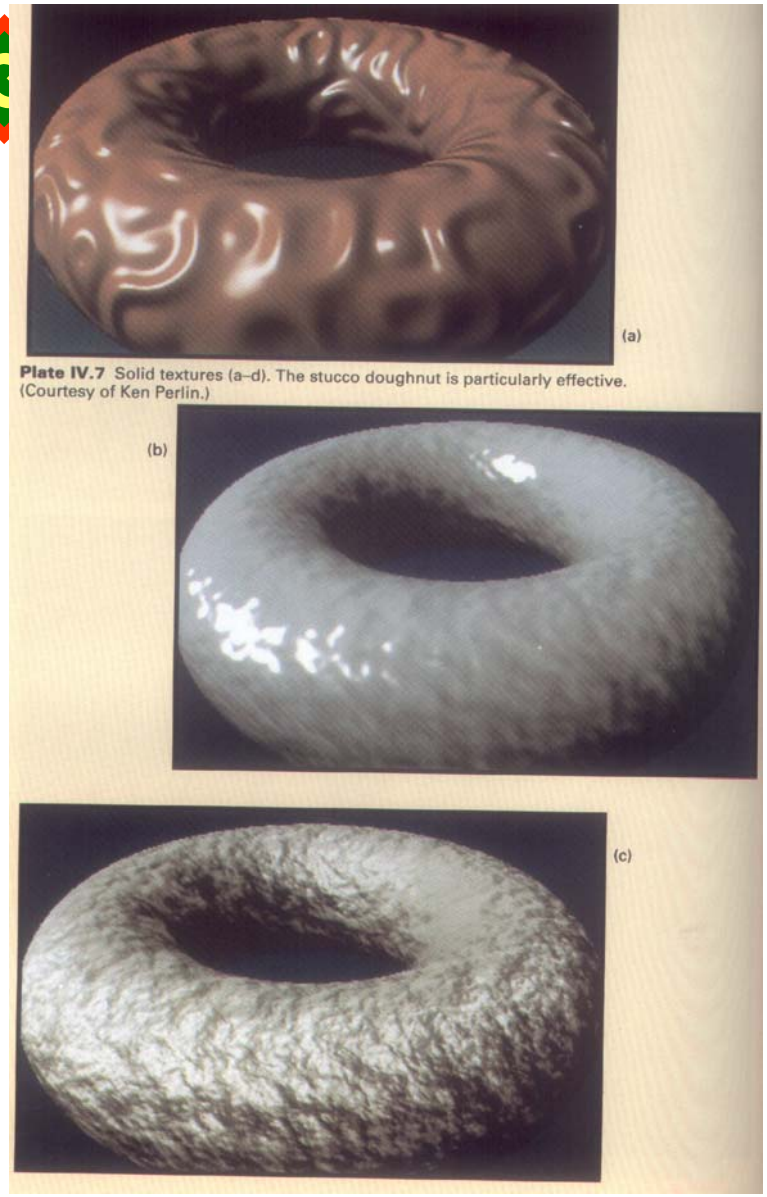
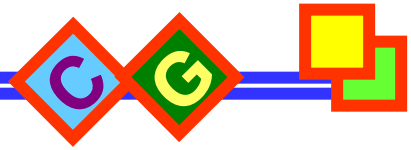


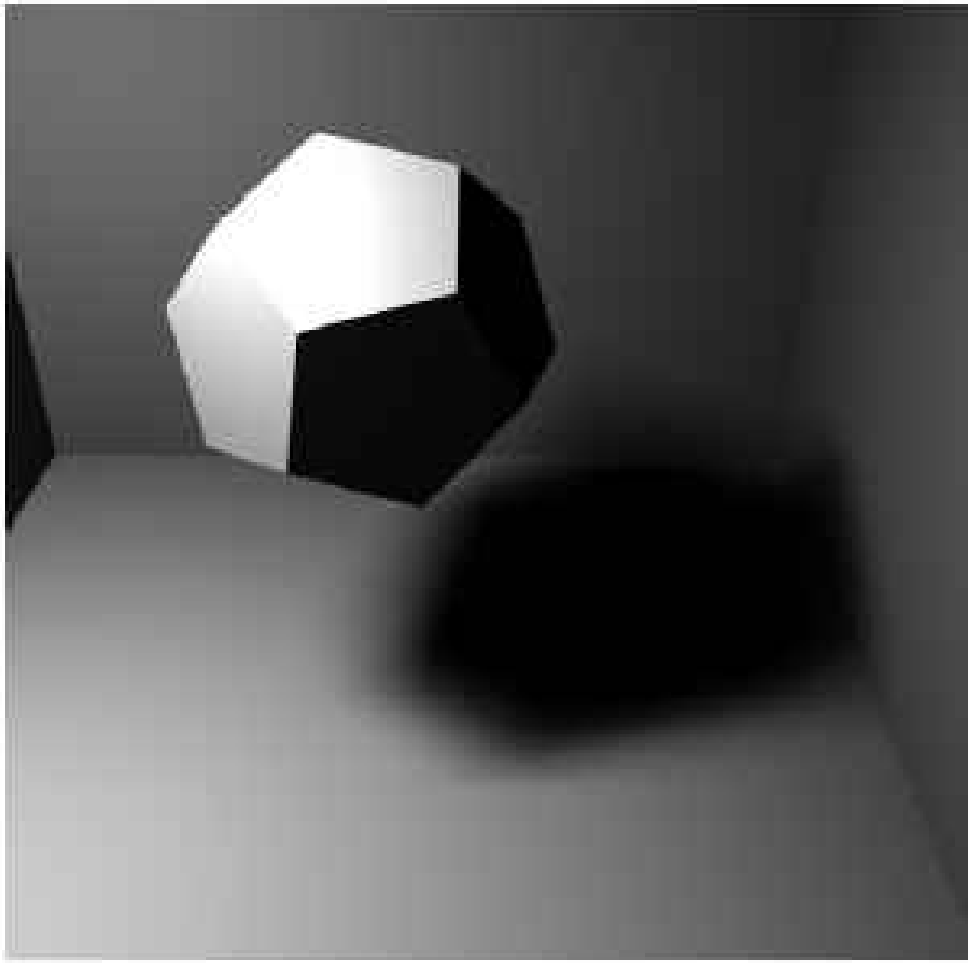
Plate IV.7 Solid textures (a–d). The stucco doughnut is particularly effective. (Courtesy of Ken Perlin.)

阴影的生成



1978年，**Atherton**等人提出了阴影多边形算法 。

阴影的生成



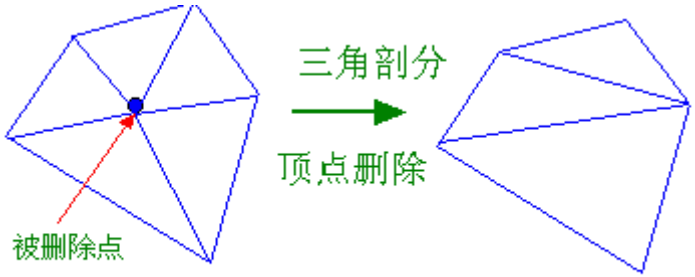
实时真实感图形学技术

就目前的技术而言，主要有两种技术：

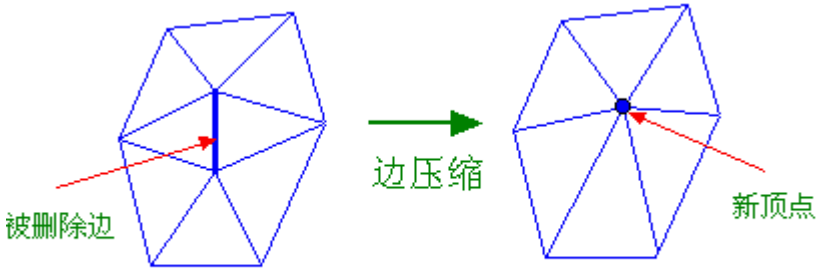
- 通过降低显示三维场景模型的复杂度来实现，这种技术被称为层次细节(LOD：Level of Detail)显示和简化技术，是当前大多数商业实时真实感图形生成系统中所采用的技术。

- 在最近的几年中，又出现了一种全新思想的真实感图形生成技术－基于图象的绘制技术(Image Based Rendering)，它利用已有的图象来生成不同视点下的场景真实感图象，生成图象的速度和质量都是以前的技术所不能比拟的，具有很高的应用前景

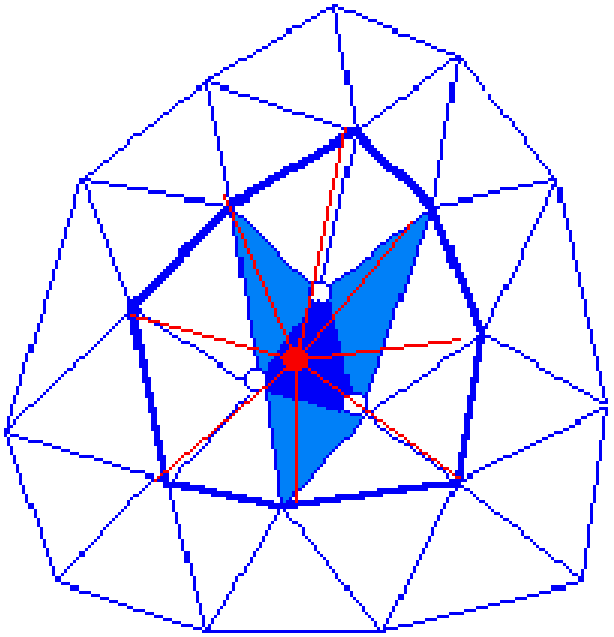
层次细节显示和简化



(A) 顶点删除操作

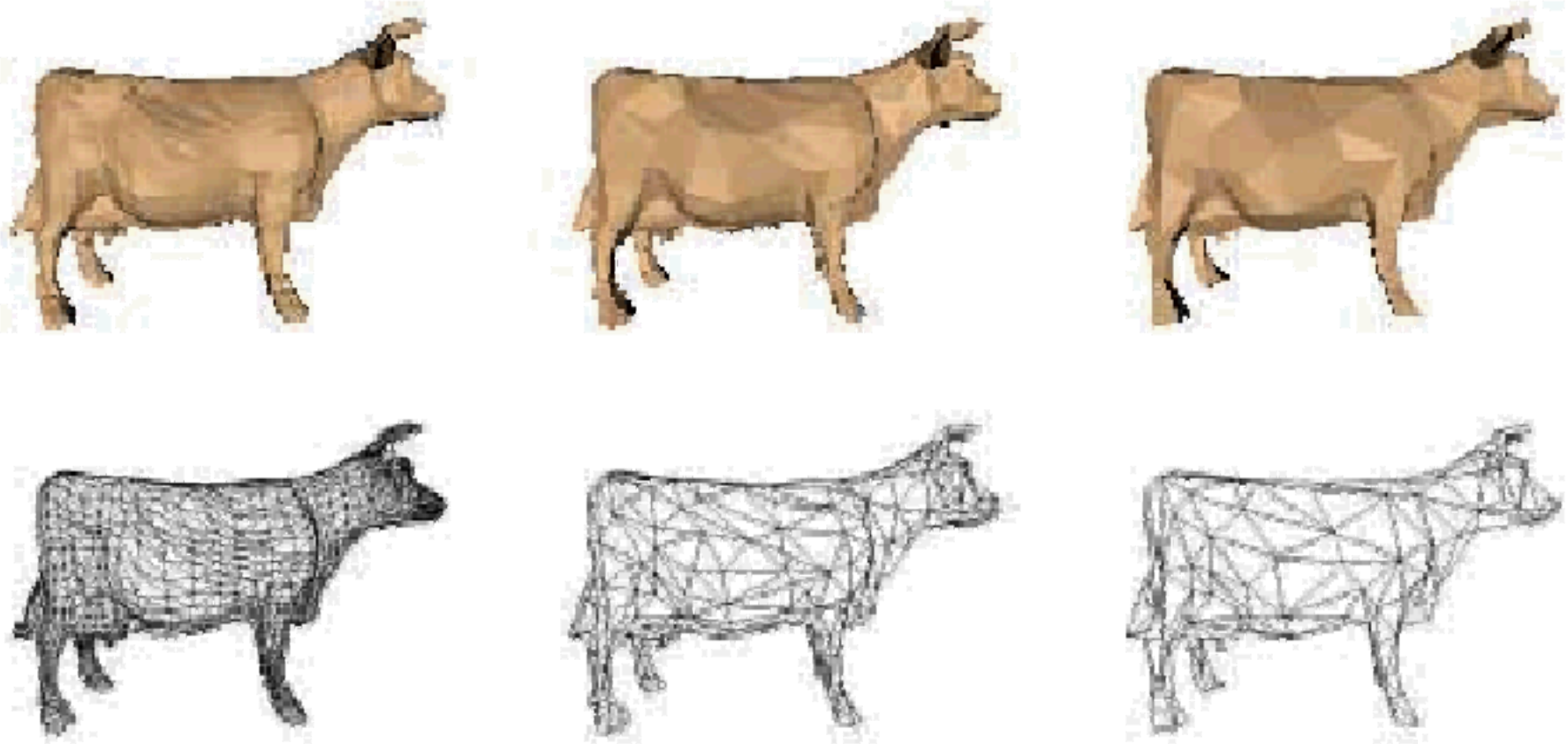


(B) 边压缩操作

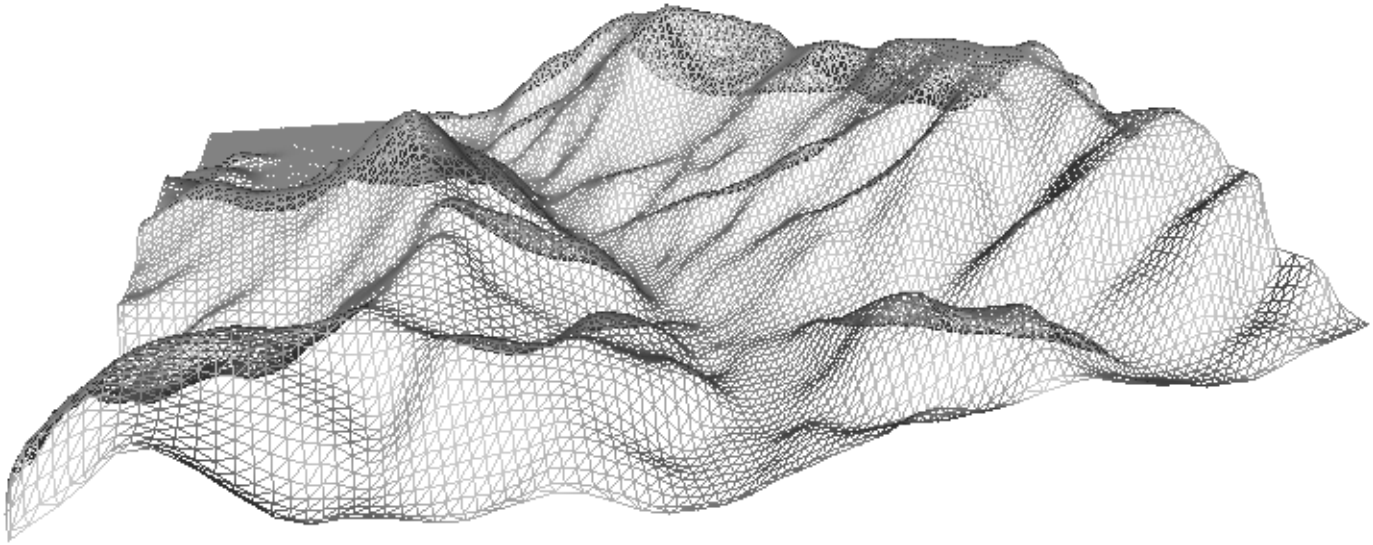


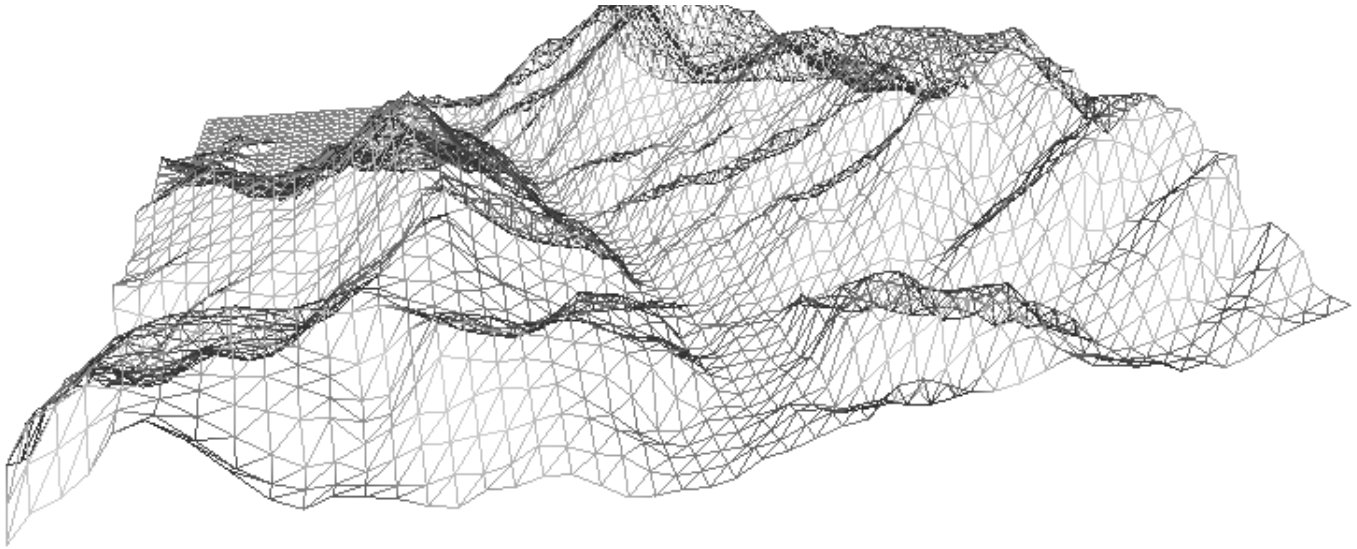
(C) 面片收缩操作

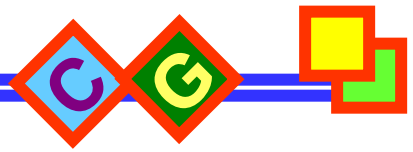
层次细节显示和简化



层次细节显示和简化







真实感图形的生成技术

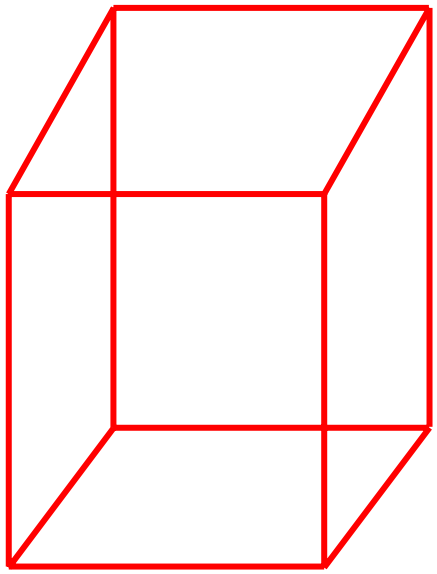
- 消隐技术
- 光照技术
- 物体表面细节的模拟
- 阴影的生成
- 图形反走样技术
- 用OpenGL生成真实感图形

消隐技术

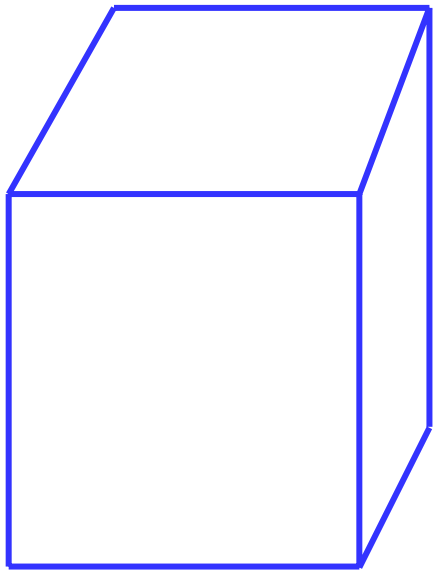
- 消隐技术的综合介绍
- 消除隐藏线
- 多面体隐藏线消除

消隐技术的综合介绍

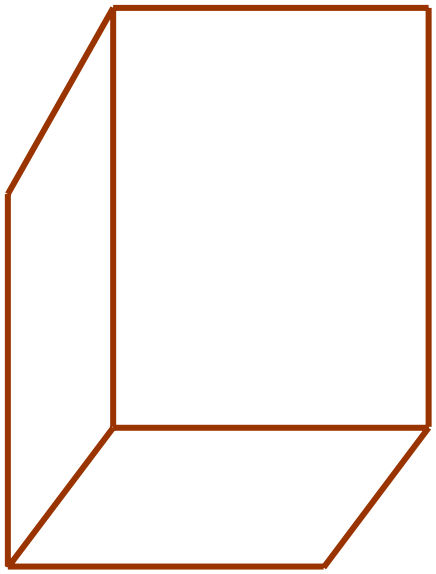
- 投影变换失去了深度信息，往往导致图形的二义性



线框图



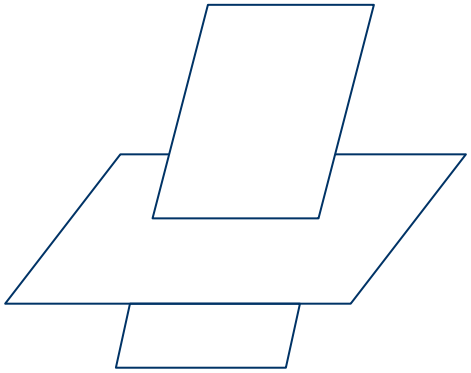
从右上往下看

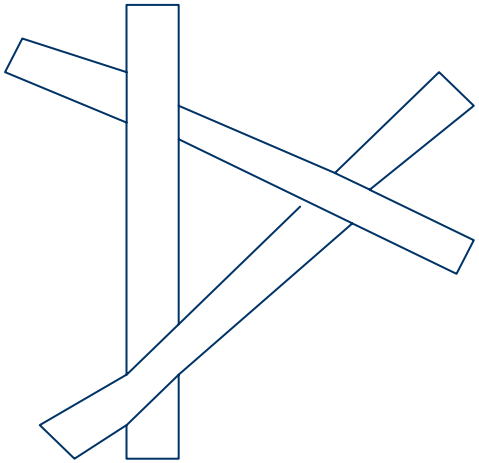


从左下往上看

问题的引出

- 此时，出现真实感图形学的概念。





贯串与循环遮挡

消隐：消除隐藏线、隐藏面

消隐技术的综合介绍

- 要消除二义性，就必须在绘制时消除被遮挡的不可见的线或面，习惯上称作消除隐藏线和隐藏面，简称为消隐。
- 经过消隐得到的投影图称为物体的真实图形。

判别可见面算法的分类

按照实现时所基于的坐标系分为

物空间算法和像空间算法。

判别可见面算法的分类

物空间算法是在定义、描述物体的世界坐标系中实现的。

```
//以场景中的物体为循环核心即处理单元
for(场景中的每一个物体)
{
   将该物体与场景中的其它物体进行比较，确定其表面的可见部分；
   显示该物体表面的可见部分；
}
```

物空间算法优点：

- 精度高，与机器精度相同
- 在工程应用方面特别有用

判别可见面算法的分类

像空间算法是在观看物体的屏幕坐标系下实现的。

```
//以窗口内的每个像素为循环核心
for(窗口内的每一个像素)
{
    确定与此像素对应的距离视点最近的物体;
    用该物体表面的颜色显示像素;
}
```

像空间算法的计算仅局限于屏幕的分辨率

判别可见面算法的分类

物空间算法和像空间算法显著区别在于算法所需要的计算量不同。

例：假设场景中有k个物体，平均每个物体表面由h个多边形构成，显示区域中有m*n个像素

- 物空间算法需要的计算量为（kh）*（kh）
- 像空间算法需要的计算量为 (mn)*(kh)

判别可见面算法的分类

为什么判别可见面算法并非在物空间实现？

- 1 判别可见面的算法离不开排序
- 2 排序一般是基于体、面、边或点到视点的距离
- 3 判别可见面的算法的效率很大程度上取决于排序的效率。

判别可见面算法的分类

为什么判别可见面算法并非在物空间实现？

- 4 扫描线的方式实现像空间算法时容易利用连贯性质，使得像空间算法更具效率。

连贯性：是物体特征变化趋势具有局部不变性。

判别可见面算法的分类

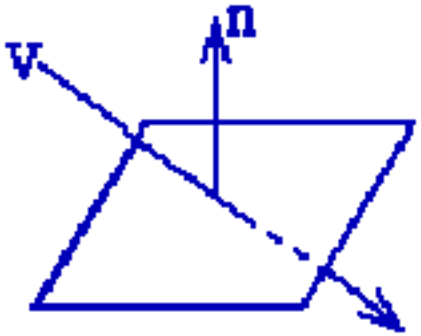
提高消隐算法效率的方法：

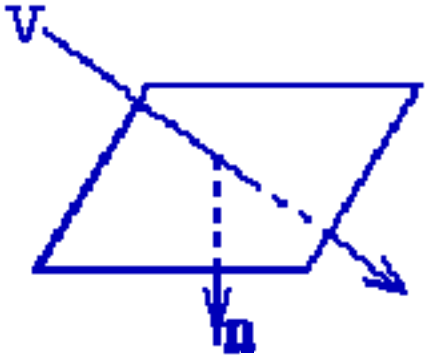
- 利用连贯性
- 包围盒技术
- 背面剔除
- 区域分割技术
- 物体分层表示

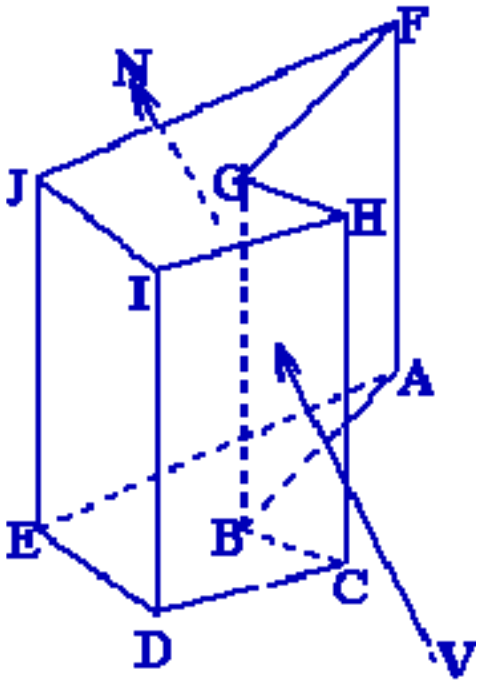
消除隐藏线

最基本的运算

线消隐中，最基本的运算为：判断面对线的遮挡关系。体也要分解为面，再判断面与线的遮挡关系。在遮挡判断中，要反复地进行线线、线面之间的求交运算。



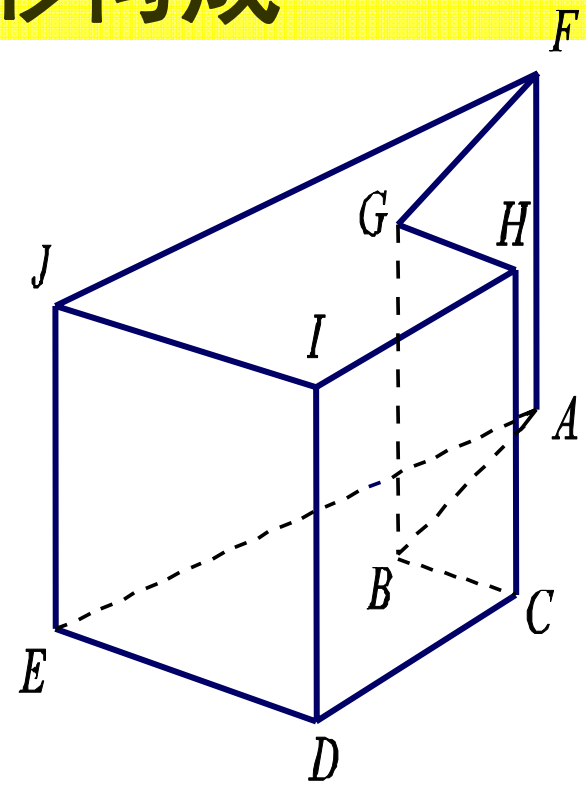




多面体：由表面多边形构成

条件：

多面体是用线框方式表示的，并且如果存在多个多面体，则多面体之间是互不相交的。



BC, BA, BG， EA为隐藏线

各种消隐方法都假定构成对象的不同面不能相互贯穿

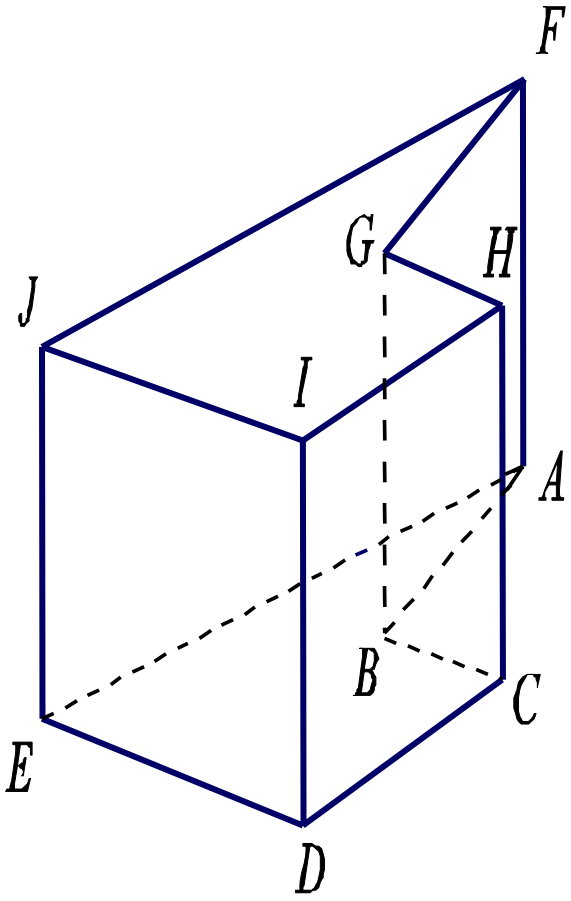
也不能有循环遮挡的情况，如果有这种情况，可把它们剖分成互不贯串和不循环遮挡的情况。

例如用图中的虚线便可把原来循环遮挡的三个平面，分割成不互相循环遮挡的四个面。

贯穿和循环遮挡

隐藏线的产生：

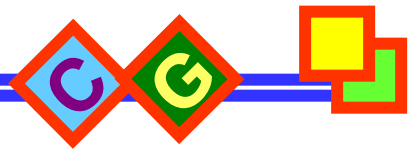
在给定的观察方向下，某些棱(或棱的一部分)被表面多边形的遮挡成为不可见。



BC, BA, BG， EA为隐藏线

多面体隐藏线消除归结为：

在给定的观察方向下，给定一条空间线段P_1P_2和一个多边形π，**判断线段是否是被多边形遮挡**。如果遮挡，求出遮挡部分。



线段和一个多边形进行隐藏性判断涉及的运算包括:

投影变换、

平面上线段和多边形的求交、

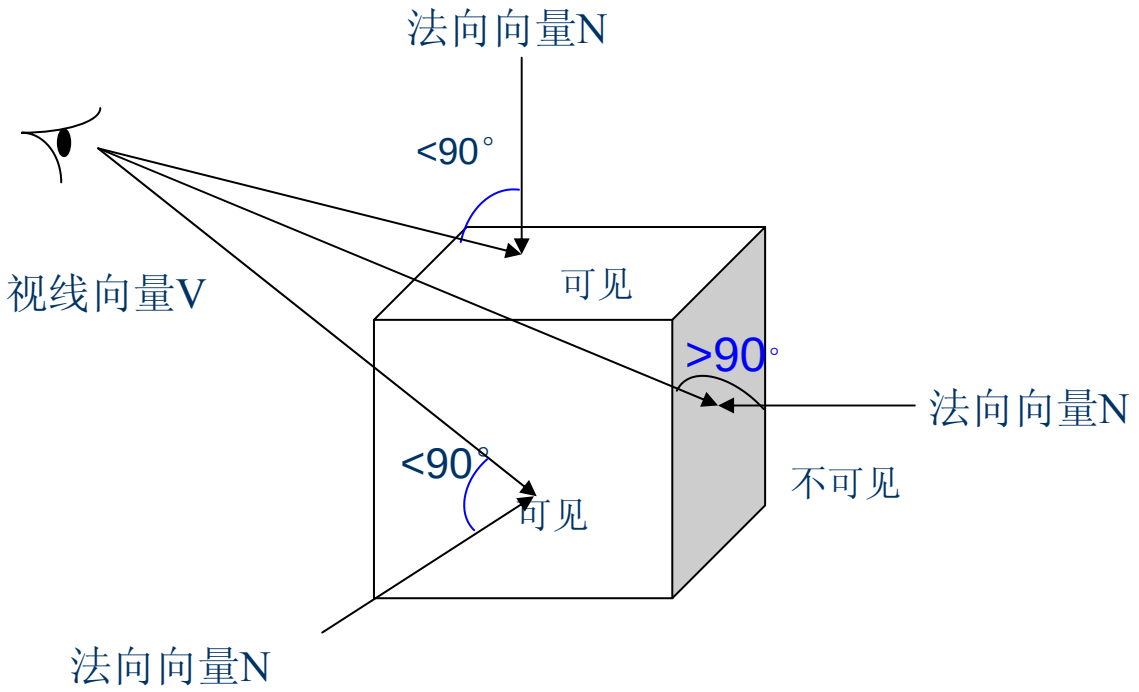
判断点是否在多边形内、

空间射线和平面求交

消除自隐藏线、隐藏面

当视线与某个多边形的内法向量夹角余弦大于0，则这个多边形称为“朝前的面”

是潜在可见面，它可能完全可见，也可能被其他的多边形遮挡成为部分可见或完全隐藏

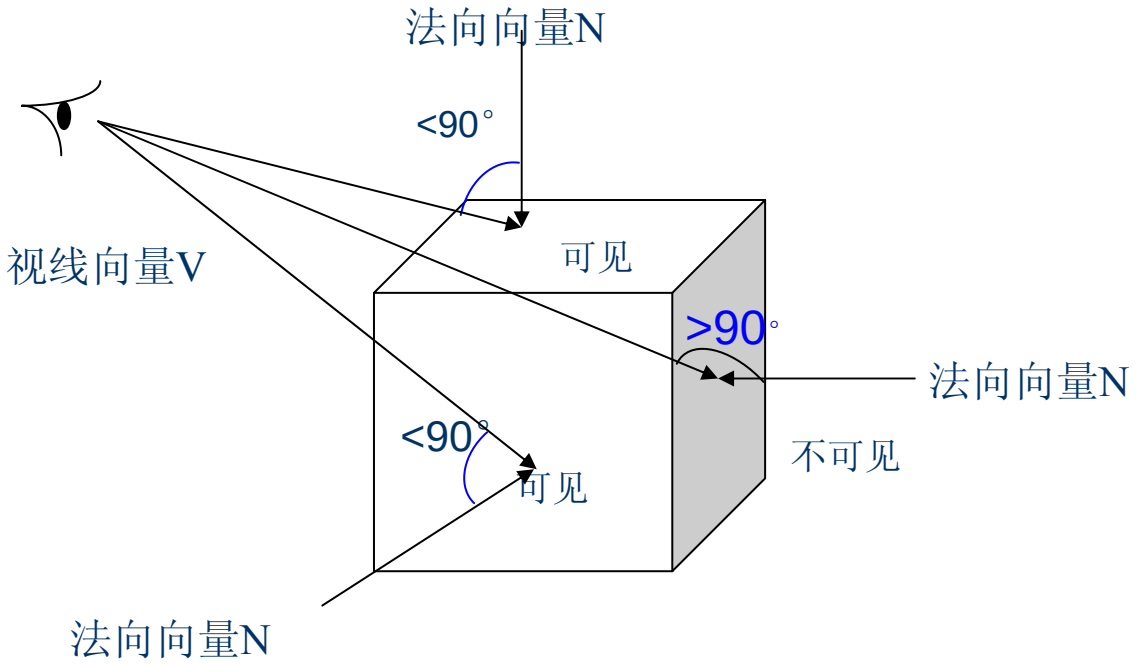


消除自隐藏线、隐藏面

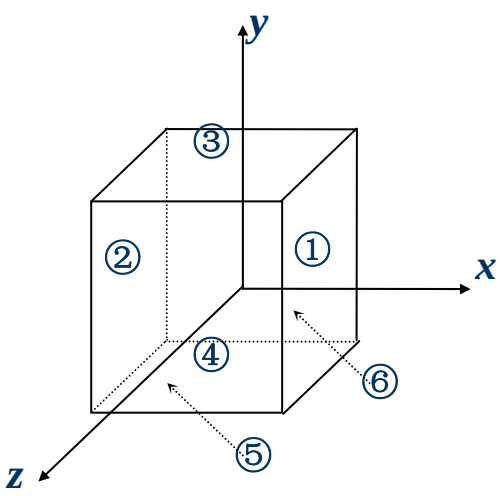
当视线与某个多边形的内法向量夹角余弦小于0，则这个多边形称为“朝后的面”

是自隐藏面，被“朝前的面”遮挡或完全隐藏

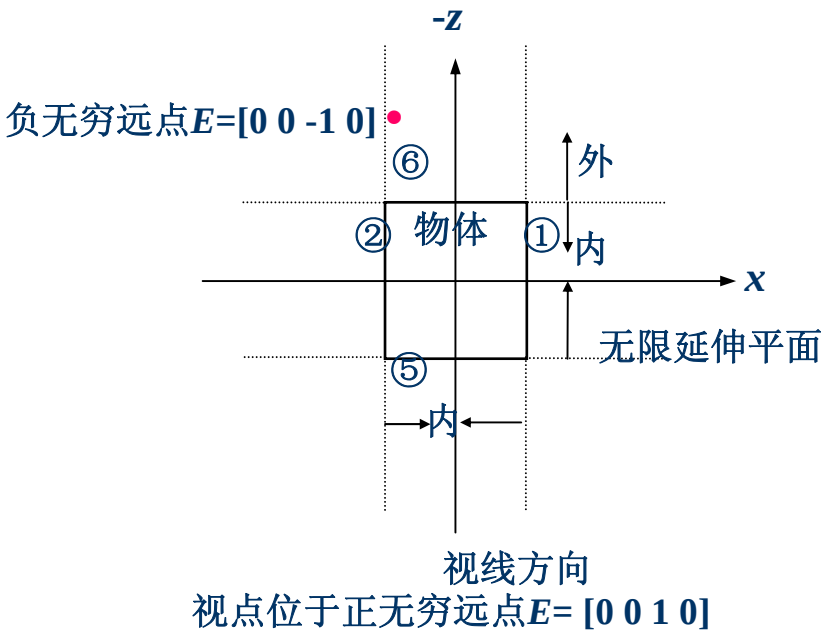
两个自隐藏面的交线为自隐藏线



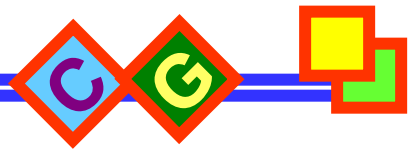
深度测试



深度测试是在观察坐标系下判断线段与多边形的前后关系，分粗略测试和精确测试



深度测试



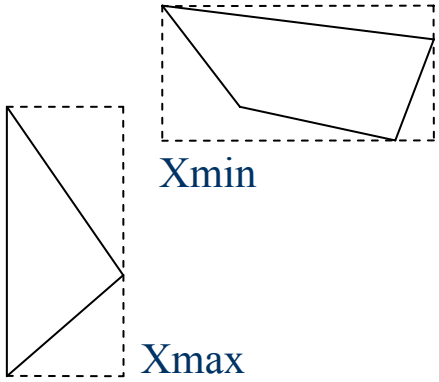
把多边形顶点的最大Z坐标和线段端点的最小Z坐标进行比较。如果前者小于或等于后者，则说明多边形完全在线段之后，线段完全可见；如果前者大于后者，这时线段仍有可能完全位于多边形之前，可以采用精确测试予以判断：

从线段两端点$p_1(x_1, y_1, z_1)$和$p_2(x_2, y_2, z_2)$各作一条与Z轴平行的直线，假设这两条直线与多边形所在平面的交点分别为$M_1(x_1, y_1, z_1')$和$M_2(x_2, y_2, z_2')$，若$z_1' \leqslant z_1$且$z_2' \leqslant z_2$，则线段完全可见，否则，要确定被多边形隐藏的子线段。

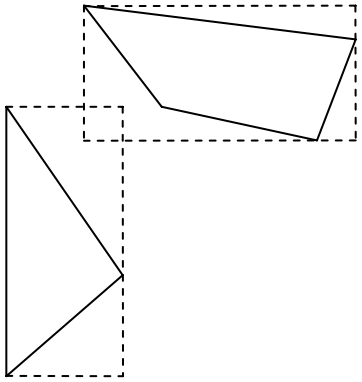
包围盒测试

在投影平面上包含线段或多边形的、边平行于投影平面坐标轴的最小矩形

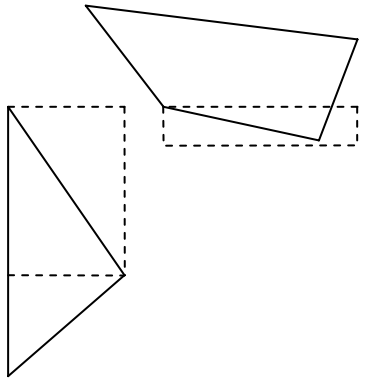
包围盒不相交，线段和多边形也不相交，线段完全可见，无需就线段和多边形的遮挡关系进行进一步判断。



不重叠,不可能互相遮蔽



测试无确定结果



对每条边进行最小最大测试

消除隐藏面

主要算法包括:

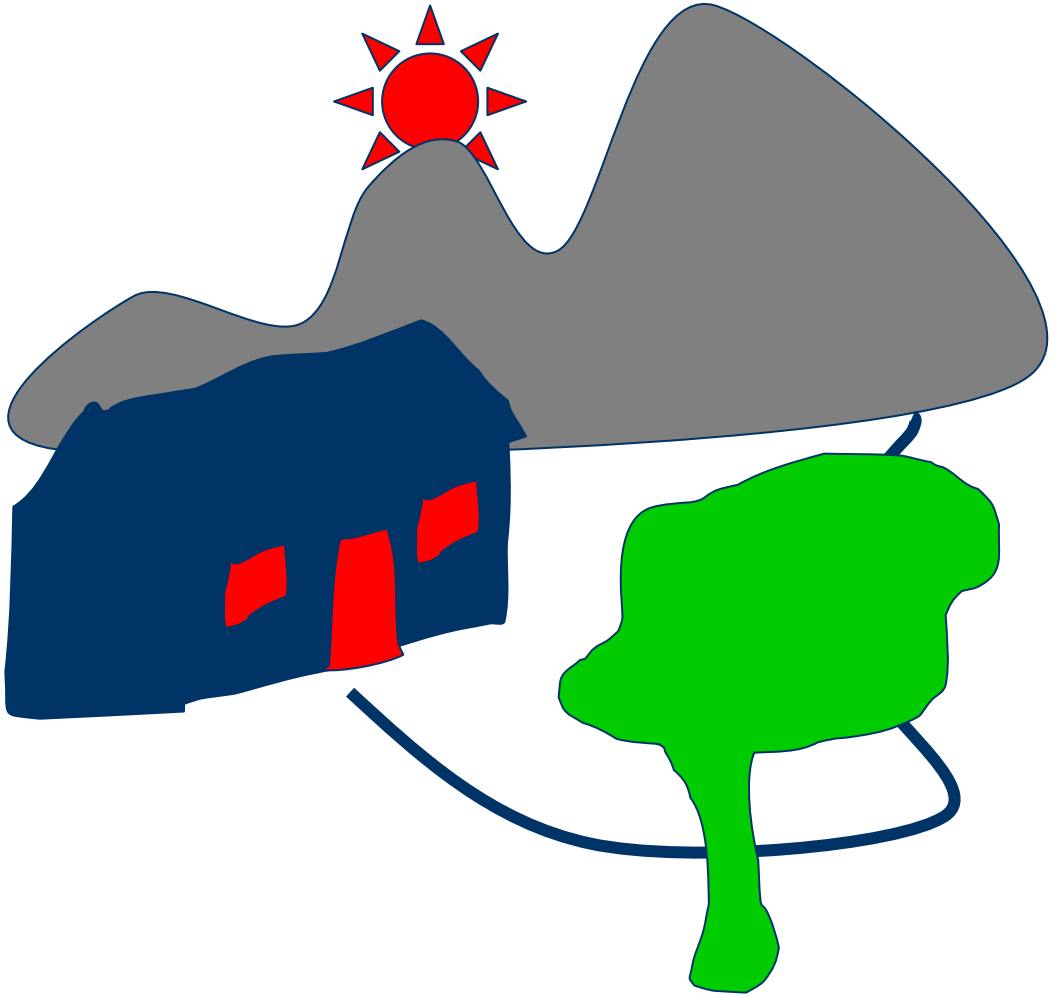
- 画家算法
- Z缓冲区（Z-Buffer）算法
- 扫描线Z-buffer算法
- 区间扫描线算法
- 区域子分割算法 (Warnack算法)
- 光线投射算法

隐藏面的消除—画家算法

- 消隐的基本（核心）问题： 排序
  - 整体排序：画家算法
  - 点排序：Z缓冲器算法

隐藏面的消除—画家算法

- 画家作画



隐藏面的消除—画家算法

算法思想：

- 按多边形离观察者的远近来建立一张表
  - 距观察者远的优先级低，近的优先级高。
- 如果这张表能正确地建立好，那么只要从优先级低的多边形开始，依次把多边形的颜色填入帧缓冲存储器中以形成该多边形的图形
- 直到优先级最高的多边形的图形送入帧缓冲器后，整幅图就显示好了。

画家算法

画家消隐算法

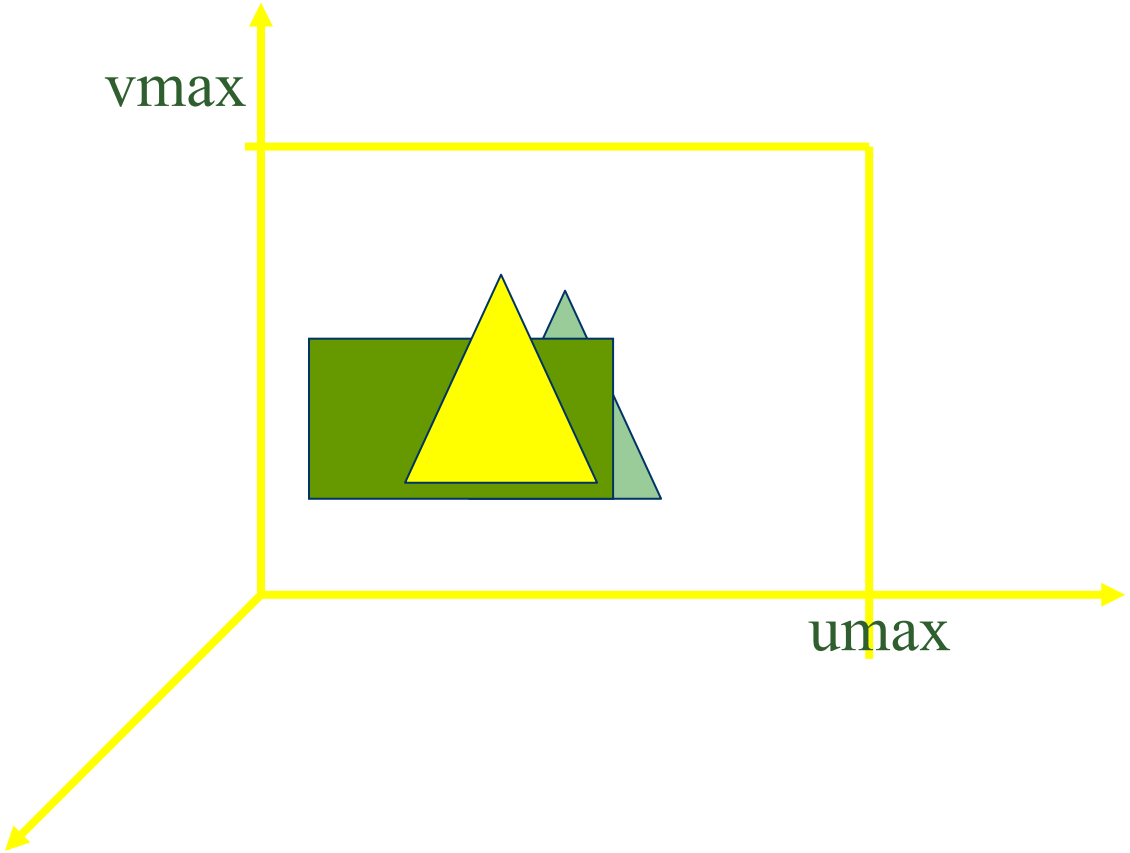
{

对场景中的多边形按深度进行排序，
形成深度优先级表；
按从远到近的顺序显示多边形；

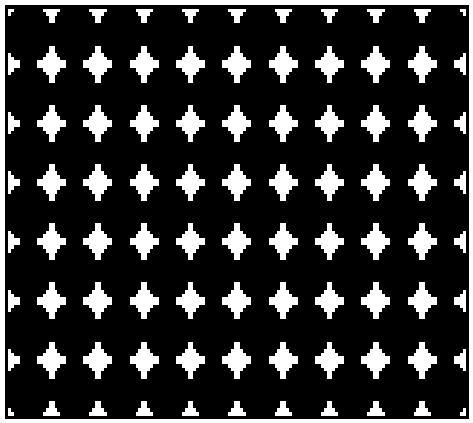
}

隐藏面的消除— Z缓冲器算法

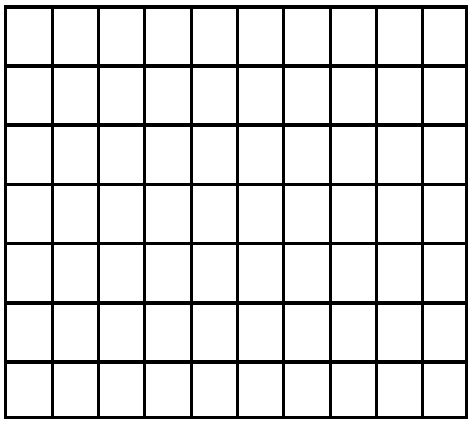
- 深度与可见性



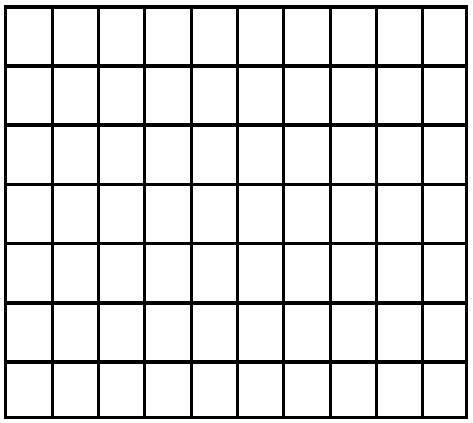
Z（深度）缓冲器



绘图窗口



帧缓冲器用于存放
对应象素的颜色



Z缓冲器用于存放
对应象素的深度值

Z缓冲器算法

```
Z缓冲器消隐算法
{
  for(v=0; v < vmax; v++)     //初始化
    for(u=0; u<umax; u++)
    {
        置Z缓冲器的第(u,v)单元的深度值为-1（最小的深度值）;
        置帧缓冲器的第(u,v)单元的颜色值为背景色;
    }
```

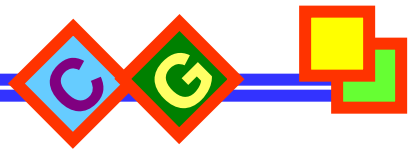
隐藏面的消除— Z缓冲器算法

```
for(每一个多边形）
   for(多边形投影区域内的每一个像素）
   {
     计算多边形在当前像素(u,v)处的深度值，记为
   d;
     if(d>Z缓冲器的第(u,v)单元的值）
       {
         置Z缓冲器的第(u,v)单元的深度值为d;
         置帧缓冲器的第(u,v)单元的颜色值为当前多边
   形颜色值;
       }
   }
}
```

隐藏面的消除— Z缓冲器算法

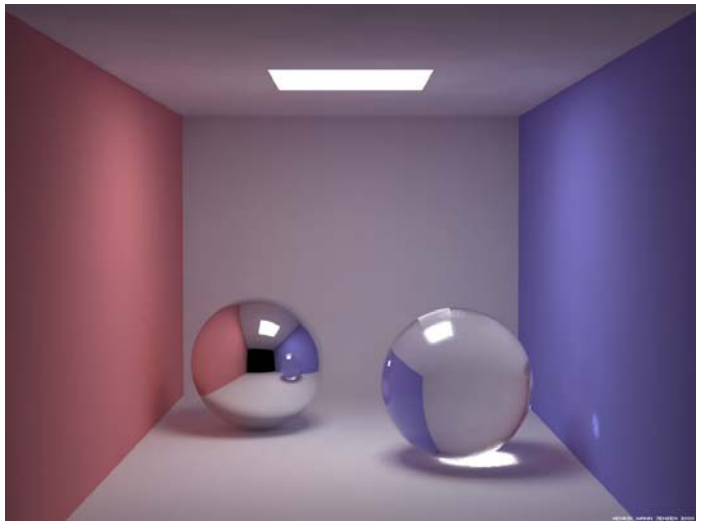
- 优点
  - 算法简单、稳定
  - 便于硬件加速
  - 不需要整个场景的几何数据
- 缺点
  - 需要Z缓冲器
  - 计算复杂度大

需要计算的像素深度值次数
*=多边形个数*多边形平均占据的像素个数*



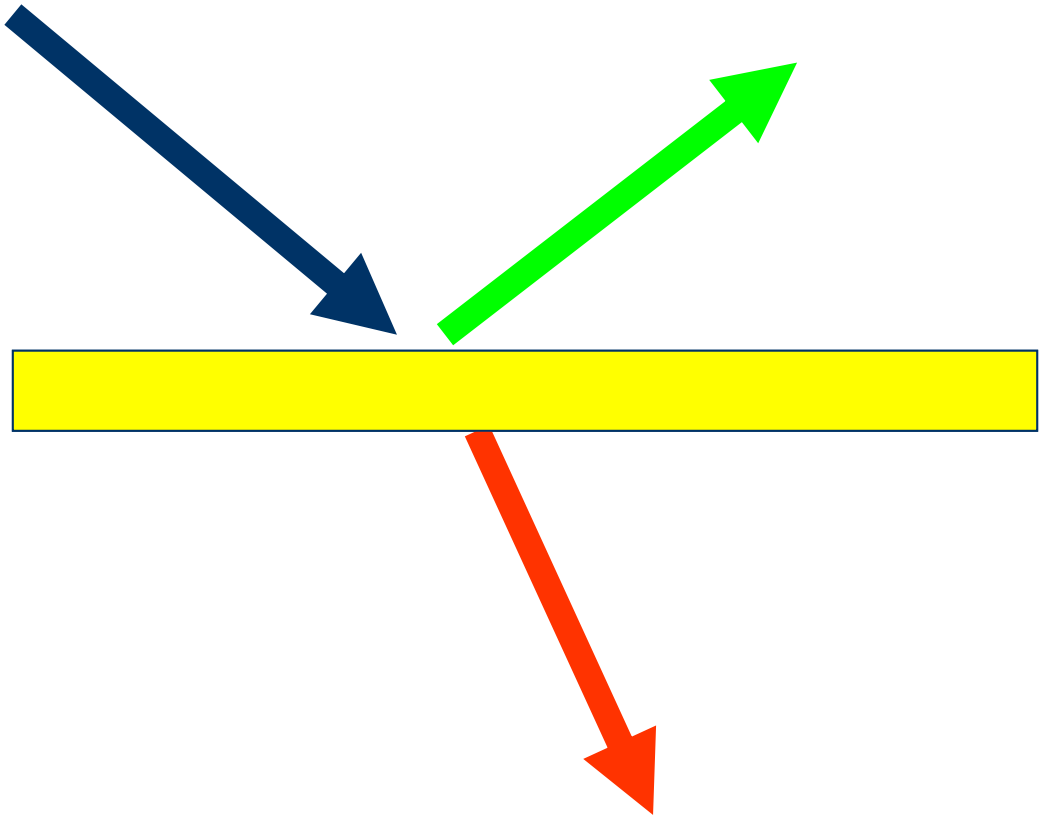
- 简单光照明模型
- 光线跟踪
- 辐射度方法

光照产生的场景



简单光照明模型（1/19）

- 光的反射、透射与转化



简单光照明模型（2/19）

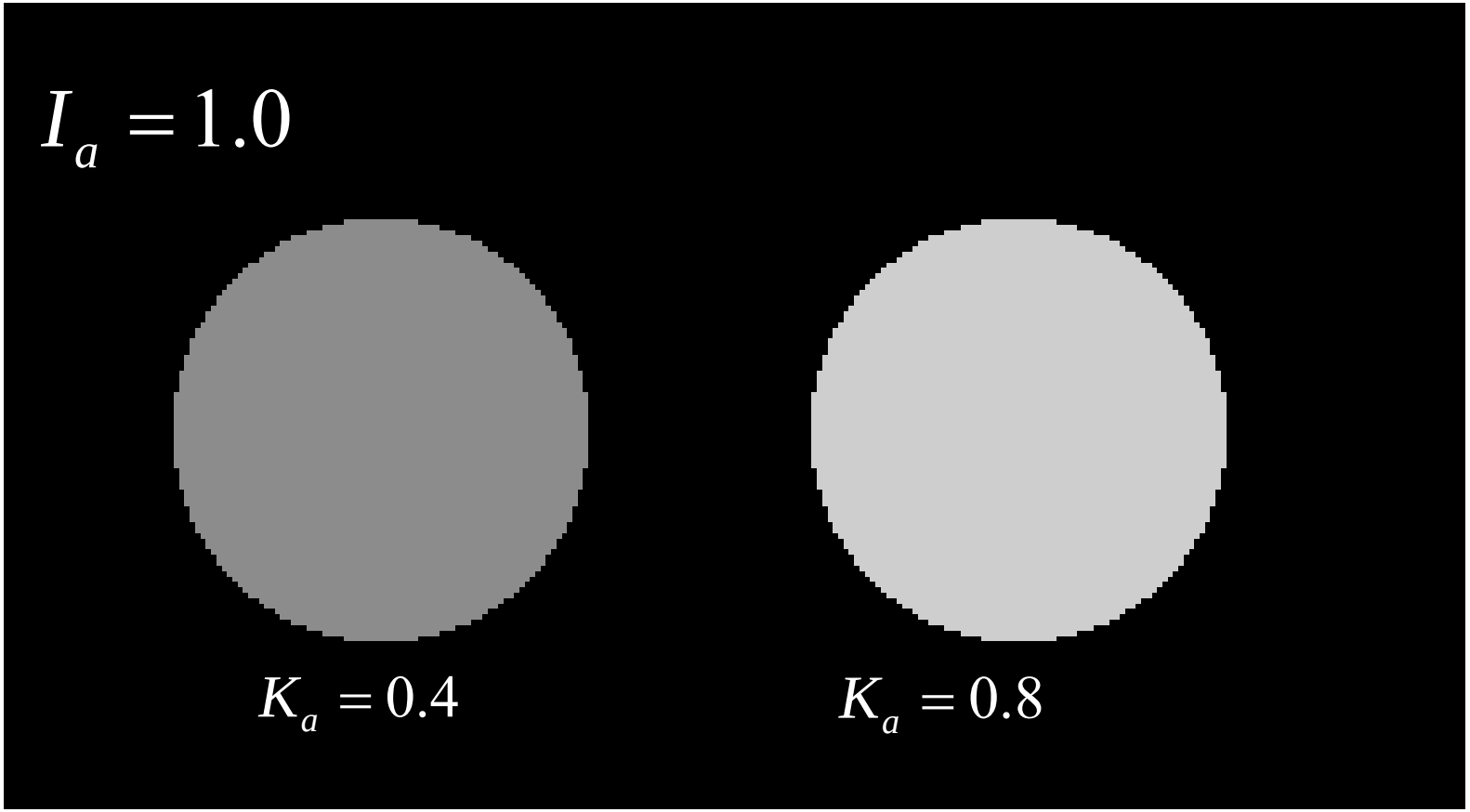
- 环境光
  - **ambient light 或 background light**
  - 是对光线复杂传播现象的抽象描述
  - 在空中的任何位置、任何方向强度相等
  - 光照明方程

$$I_e = K_a I_a$$

  - I_a 环境光亮度
  - K_a 环境光反射系数

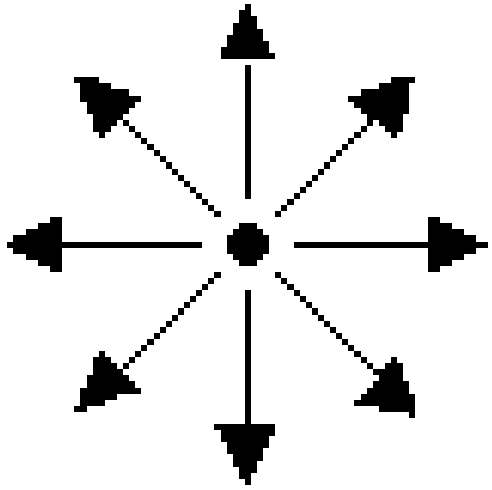
简单光照明模型（3/19）

- 例子



简单光照明模型（4/19）

- 点光源
  - 向周围辐射等强度的光

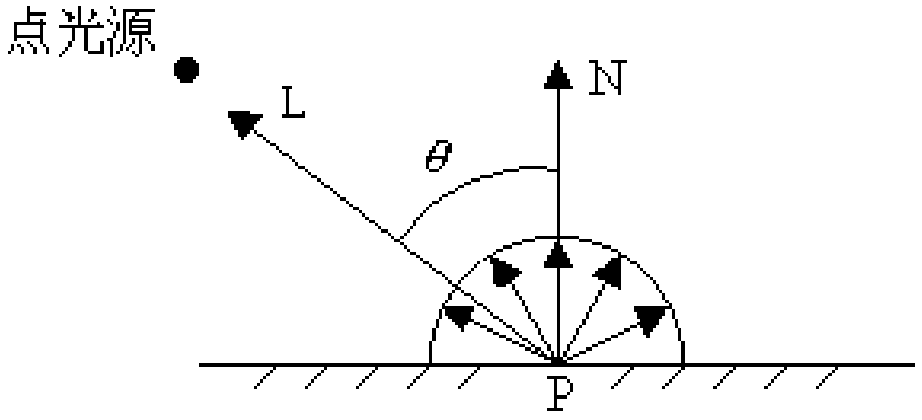


简单光照明模型（5/19）

- 漫反射
  - 粗糙、无光泽物体（如粉笔）表面对光的反射
  - 光照明方程

$$I_d = I_p K_d \cos\theta \qquad \theta \in [0, \frac{\pi}{2}]$$

  - I_p 点光源的亮度
  - K_d 漫反射系数
  - θ 入射角

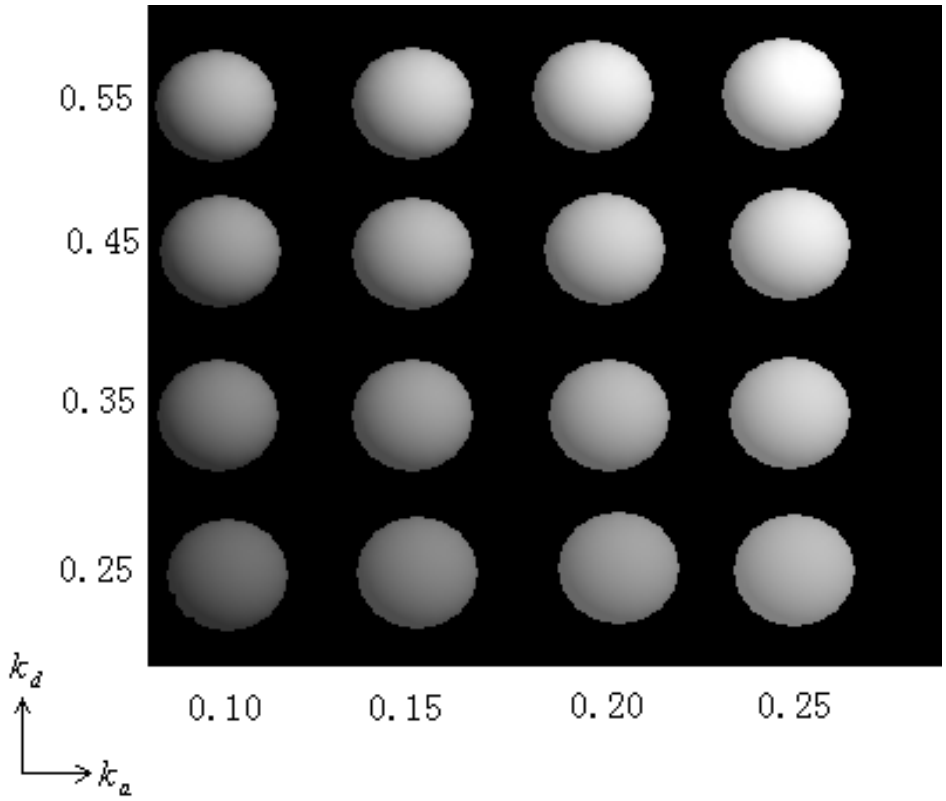


简单光照明模型（6/19）

- 将环境光与漫反射结合起来

$$I = I_e + I_d = I_a K_a + I_p K_d (L \cdot N)$$

- 例子

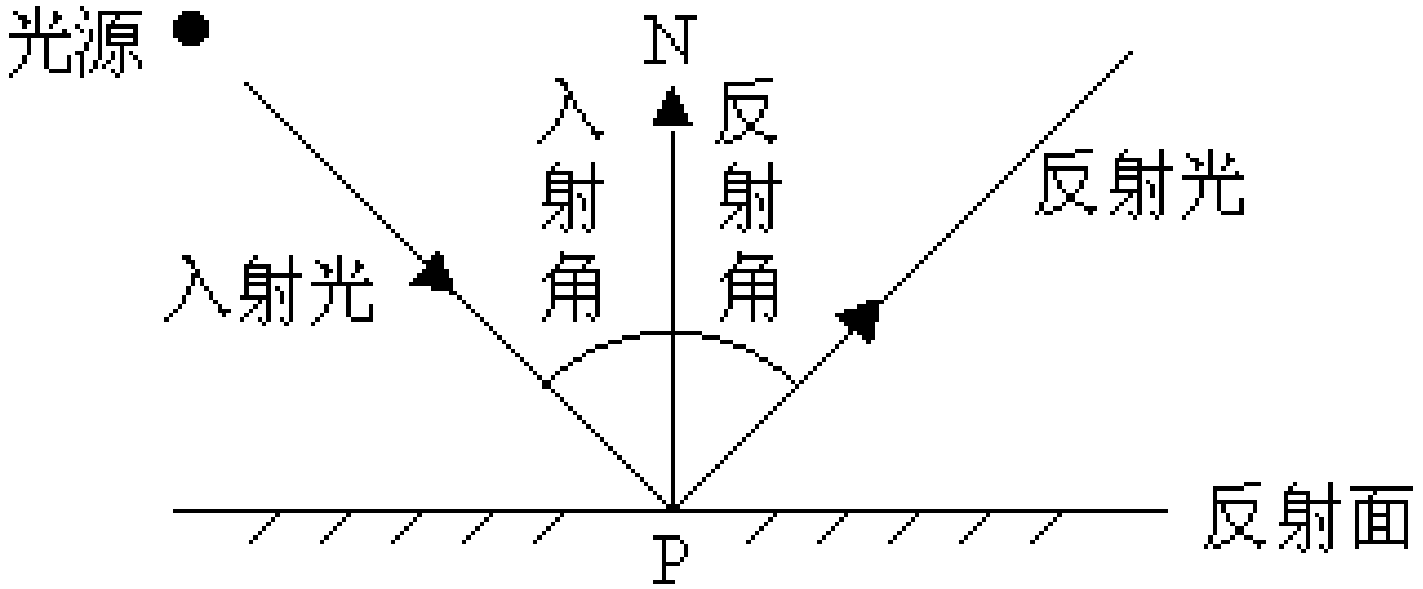


简单光照明模型（7/19）

- 镜面反射
  - 光滑物体（如金属或塑料）表面对光的反射
- 高光
  - 入射光在光滑物体表面形成的特别亮的区域

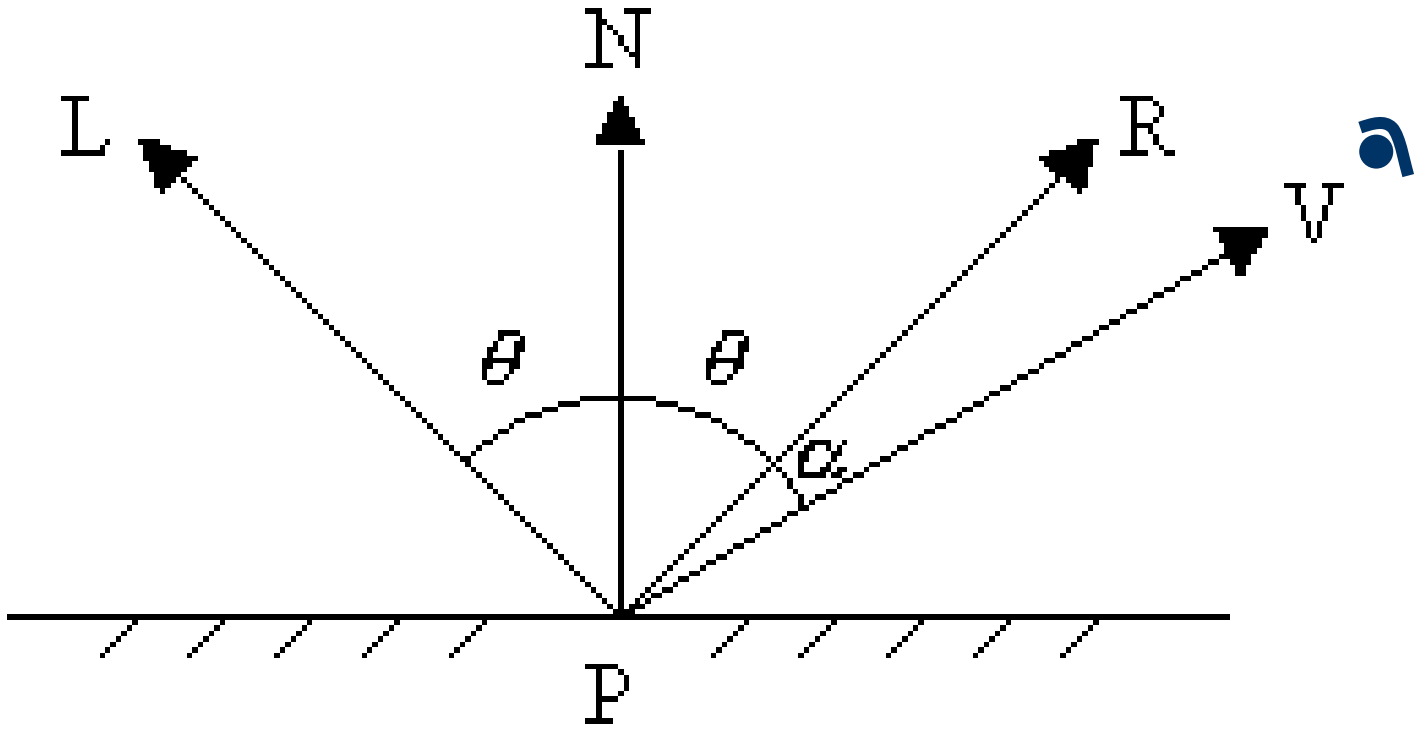
简单光照明模型（8/19）

- 理想镜面反射

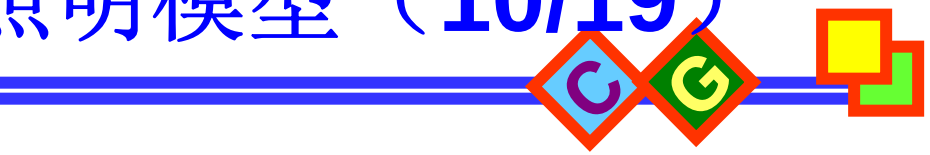


简单光照明模型（9/19）

- 非理想镜面反射



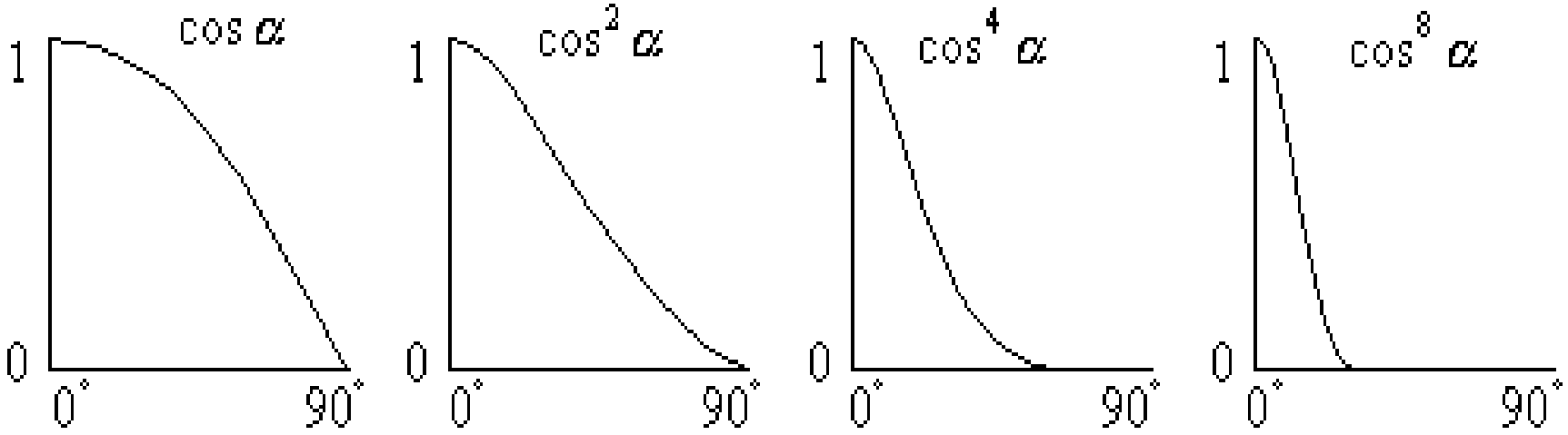
简单光照明模型（10/19）



- 光滑表面的镜面反射光的空间分布

$$I_s = I_p K_s \cos^n \alpha$$

  - **n----镜面反射指数**



简单光照明模型（11/19）

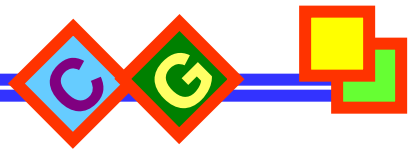
- 环境光、漫反射与镜面反射结合起来

$$I = I_e + I_d + I_s$$
$$= I_a K_a + I_p [K_d (L \cdot N) + K_s (V \cdot R)^n]$$

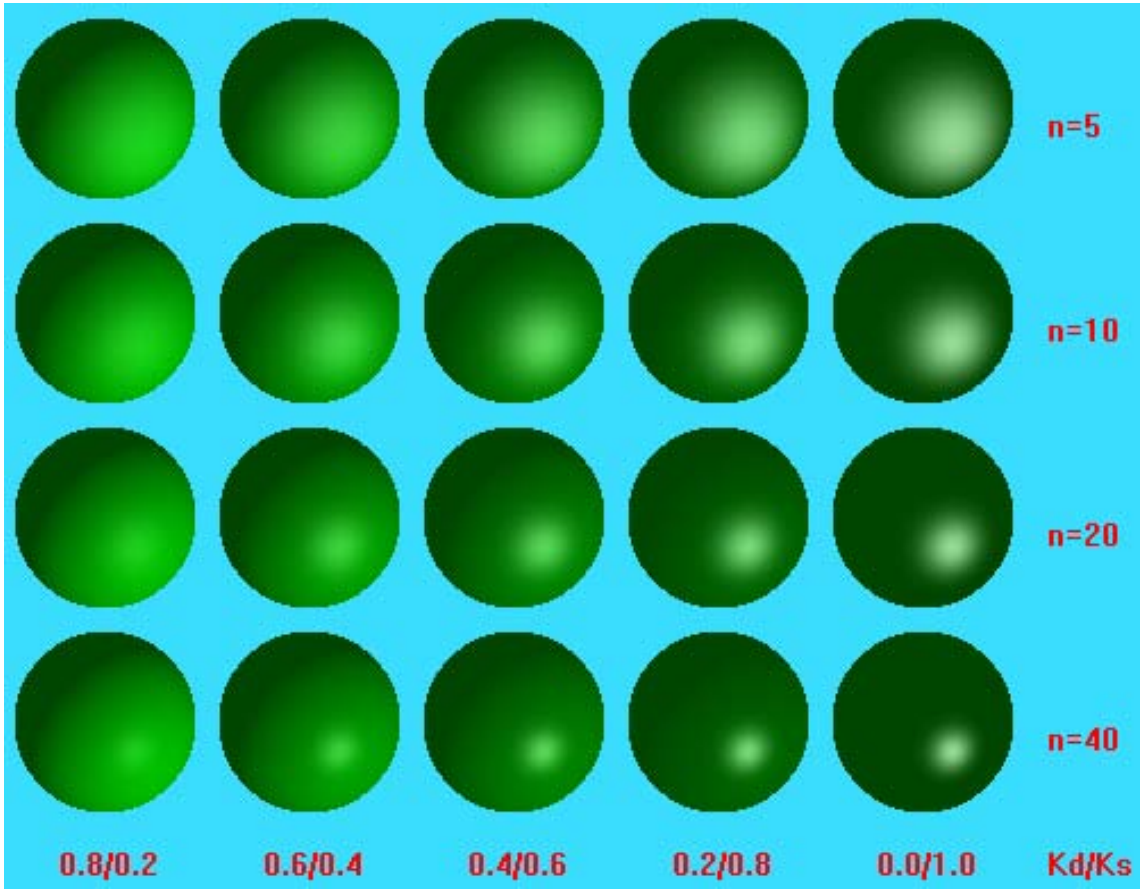
简单光照明方程
也称为**Phong**光照明模型

Phong光照明模型的不足

- Phong光照明模型是真实感图形学中提出的第一个有影响的光照明模型
- 经验模型，**Phong**模型存在不足：
  - 显示出的物体象塑料，无质感变化
  - 没有考虑物体间相互反射光
  - 镜面反射颜色与材质无关
  - 镜面反射大入射角失真现象



Phong模型示例_2

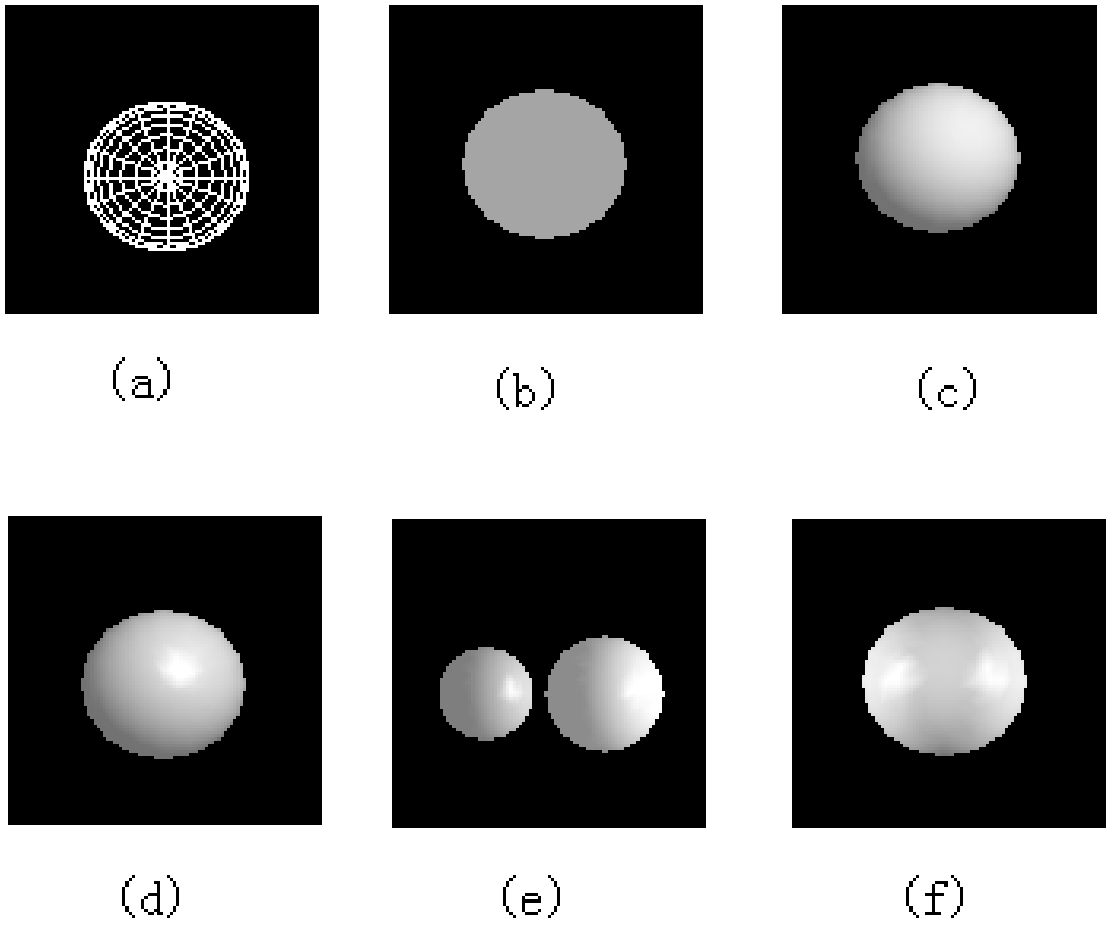


简单光照明模型（16/19）

- 产生彩色
  - 选择合适的颜色模型----RGB模型
  - 为颜色模型中的每一种基色建立光照明方程
- 采用多个光源
  - 采用m个光源的光照明方程

简单光照明模型（19/19）

– 例子

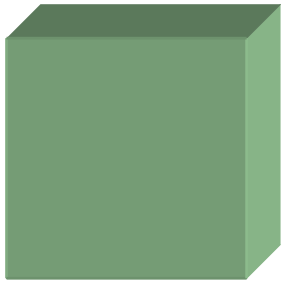


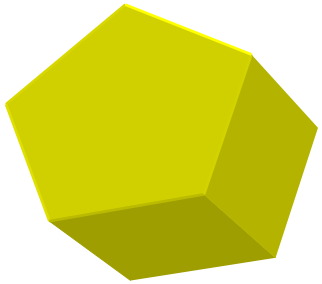
多边形绘制方法

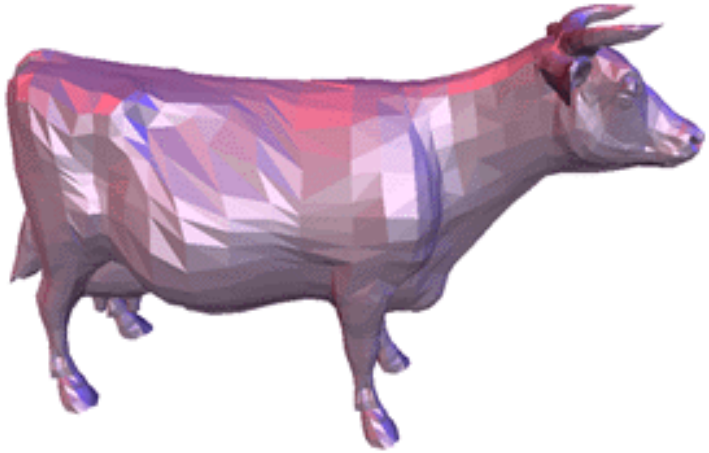
- 均匀着色
  - 方法
    - 任取多边形上一点，利用光照明方程计算出它的颜色
    - 用这个颜色填充整个多边形
  - 适合于如下情况
    - 光源在无穷远处
    - 视点在无穷远处
    - 多边形是物体表面的精确表示

多边形绘制方法

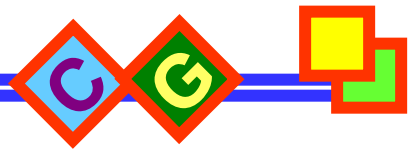
- 优点
  - 每个多边形只需计算一次光照明方程，速度快
- 缺点
  - 相邻多边形颜色过渡不光滑（块效应）
- 例子



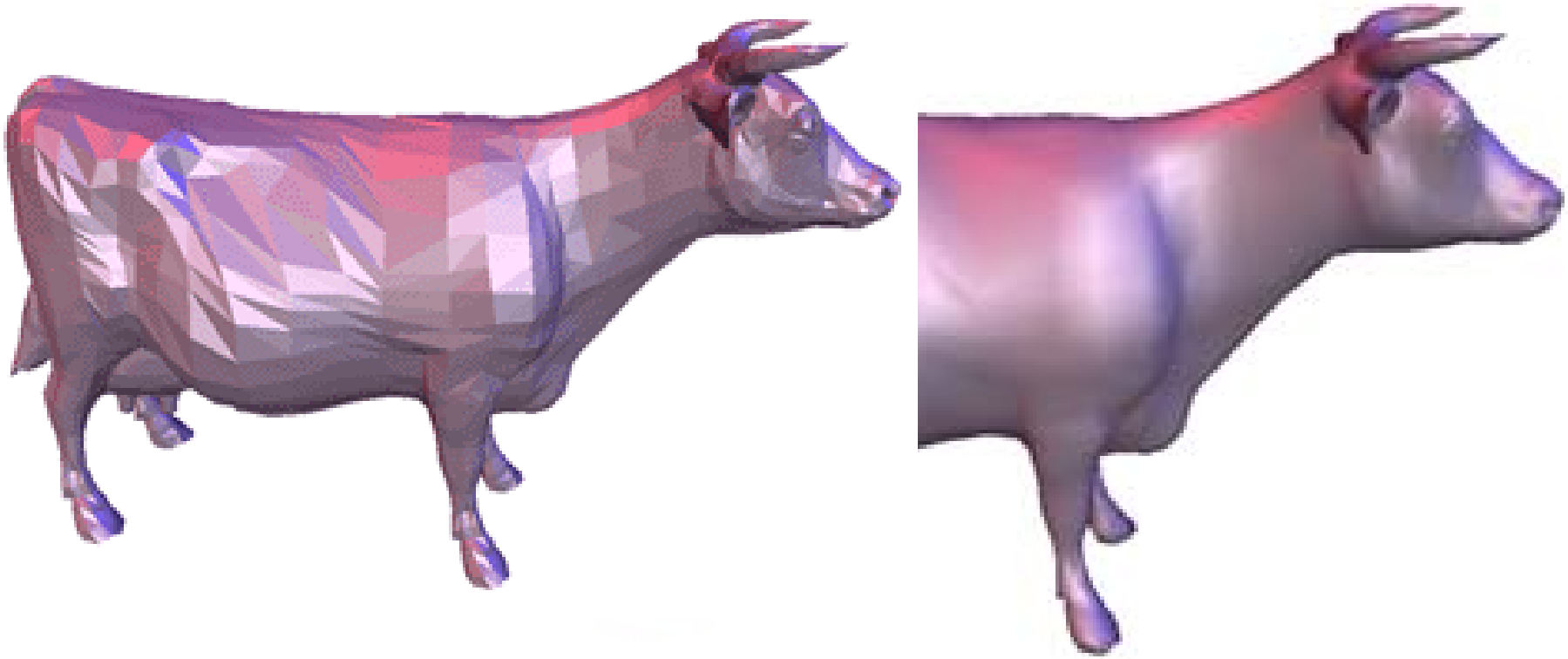




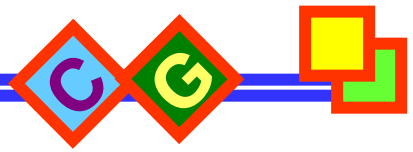
多边形绘制方法



- 光滑着色----插值
  - 颜色插值(Gouraud明暗处理)
  - 法矢量插值(Phong明暗处理)



Gourand 明暗处理效果



两种多边形处理方法比较

Gouraud明暗处理：

- 计算量小
- 不容易产生正确的高光区域(颜色插值)

Phong明暗处理：

- 计算量大
- 能进行好的局部逼近，有真实的高光效果

光线跟踪算法的基本原理

- 自然界中光线的传播过程

光源 → 物体表面 → 物体表面 → 人眼

- 光线跟踪过程----光线传播的逆过程

光线跟踪算法的基本原理

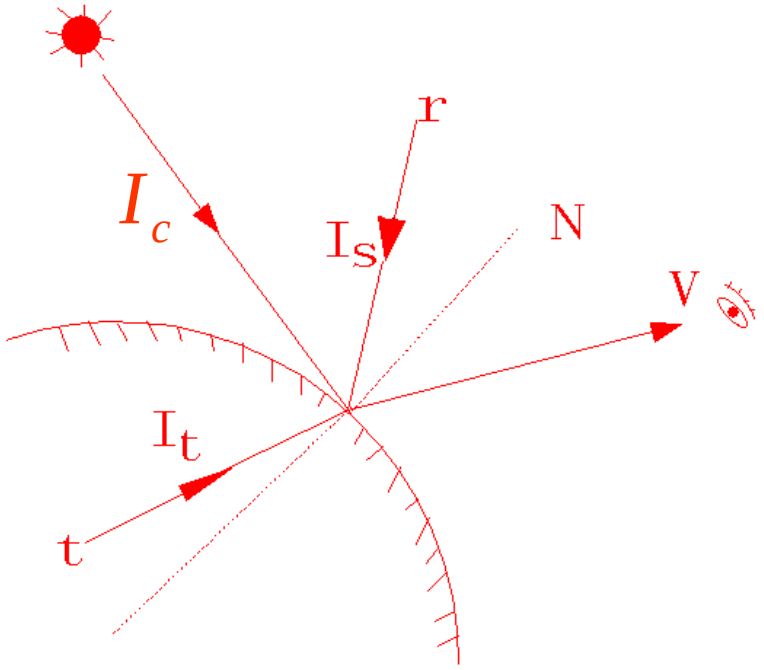
1、光源直接入射的光引起的反射光亮度 I_c

2、沿V的镜面反射方向r来的源自其它物体反射的光投射 I_s 在光滑表面引起的镜面反射光。

3、沿V的规则透射方向t来的源自其它物体反射的光投射 I_t 在透明体表面引起的规则透射光

$$I = I_c + K_s \cdot I_s + K_t \cdot I_t$$

称为整体光照模型



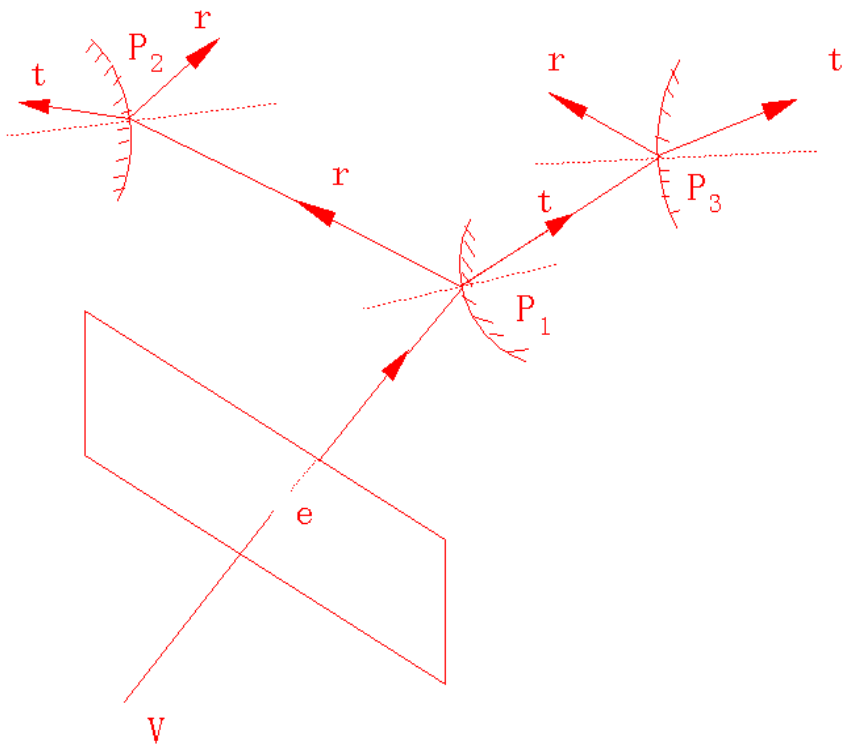
Whitted光照模型示意图

光线跟踪的终止条件

虽然在理想情况下，光线可以在物体之间进行无限的反射和折射，但是在实际的算法进行过程中，我们不可能进行无穷的光线跟踪，因而需要给出一些跟踪的终止条件。在算法应用的意义上，可以有以下的几种终止条件：

- 该光线未碰到任何物体。
- 与光线相交的最近景物面为漫射面。
- 该光线碰到了背景。
- 光线在经过许多次反射和折射以后，就会产生衰减，光线对于视点的光强贡献很小（小于某个设定值）。
- 光线反射或折射次数即跟踪深度大于一定值。

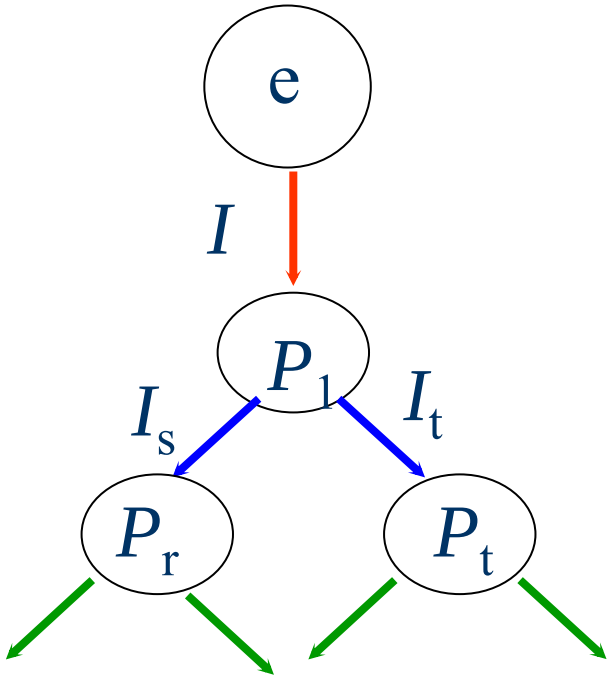
光线跟踪算法的基本原理



Whitted光照模型求解示意图

光线跟踪算法的基本原理

- 光线跟踪过程可以表示为一棵二叉树，称为光线树



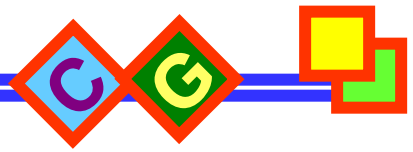
光线树

光线跟踪算法的基本原理

光线跟踪的缺点：

- 耗时多，计算量大
- 容易引起图形走样

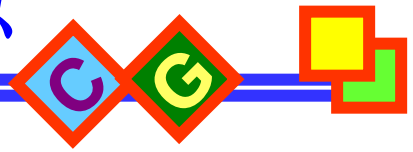
辐射度方法



光线跟踪方法难于模拟景物表面之间的多重漫反射，不能反映色彩渗透现象。

辐射度方法基于物理学的能量平衡原理，采用数值求解技术近似每一个景物表面的辐射度分布，是一个视点独立的算法，应用于虚拟环境的漫游系统

物体表面细节的模拟

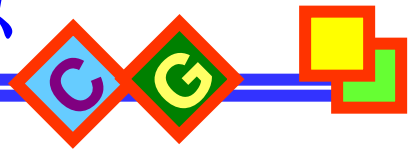


细节：

由物体表面颜色色彩、明暗变化体现出来的细节，取决于物体表面的材质属性

由物体表面不规则的细小凹凸造成的细节，取决于物体本身的的几何形状

物体表面细节的模拟

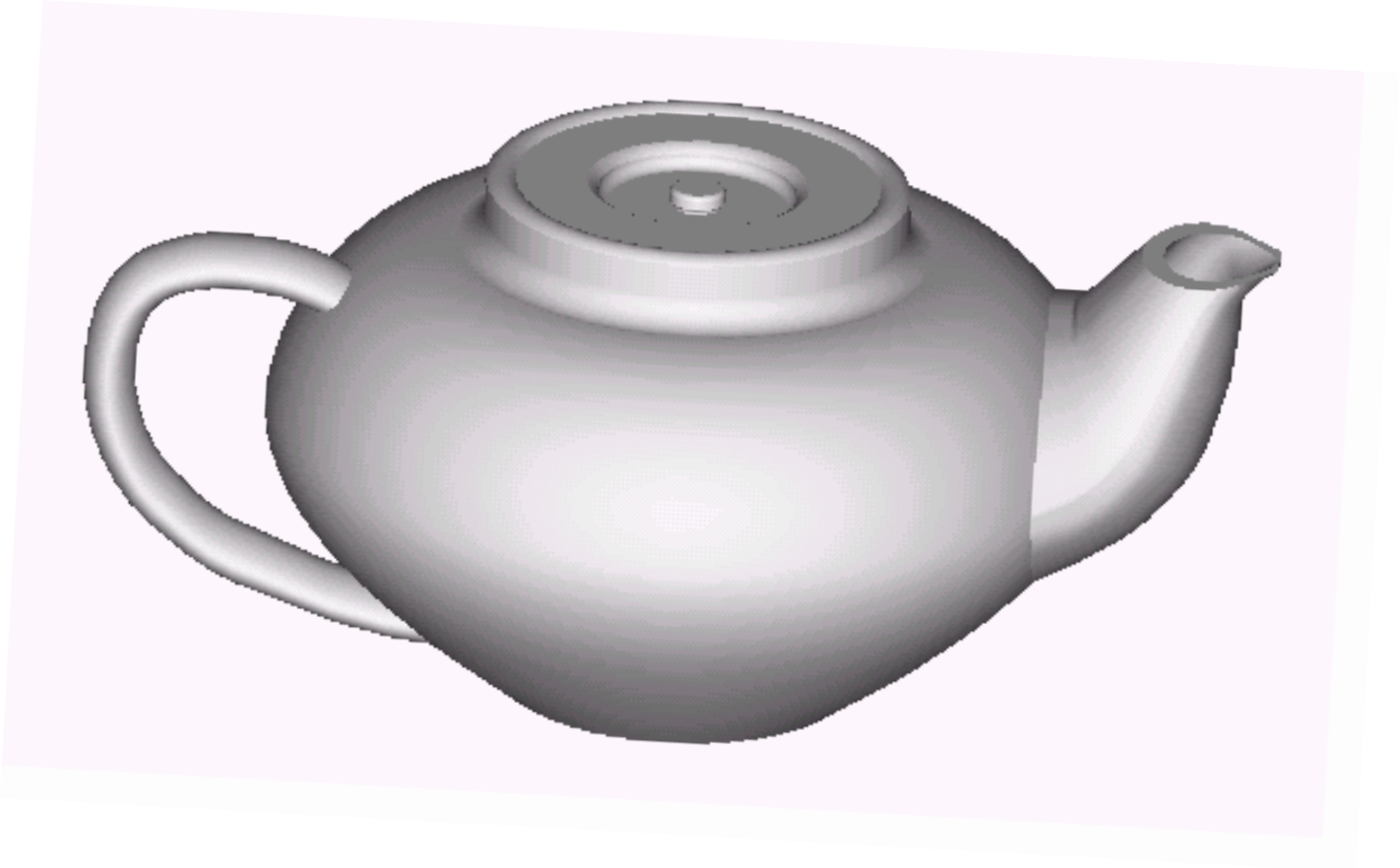


物体表面细节 —看起来象→ 纹理映射

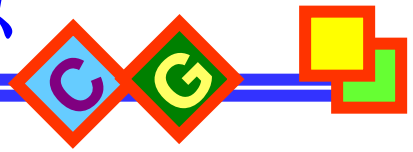
纹理映射
- 颜色纹理映射
- 几何纹理映射

物体表面细节的模拟

模型

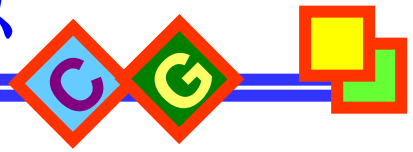


映射（Mapping）



纹理（Texture）

- 纹理是物体表面的细小结构
- 纹理类型
  - 颜色纹理
    - 二维纹理，物体表面花纹、图案
    - 三维纹理，木材纹理
  - 几何纹理
    - 法向扰动

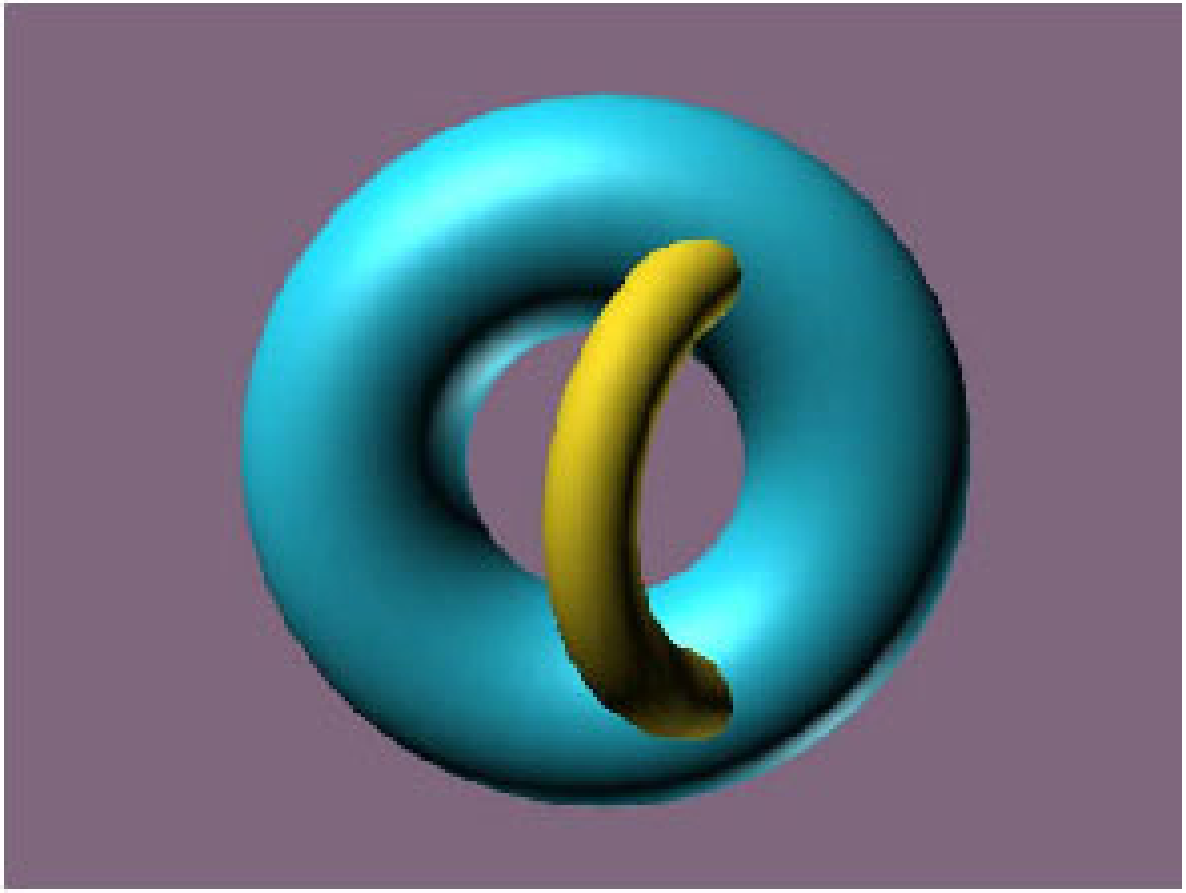


纹理映射

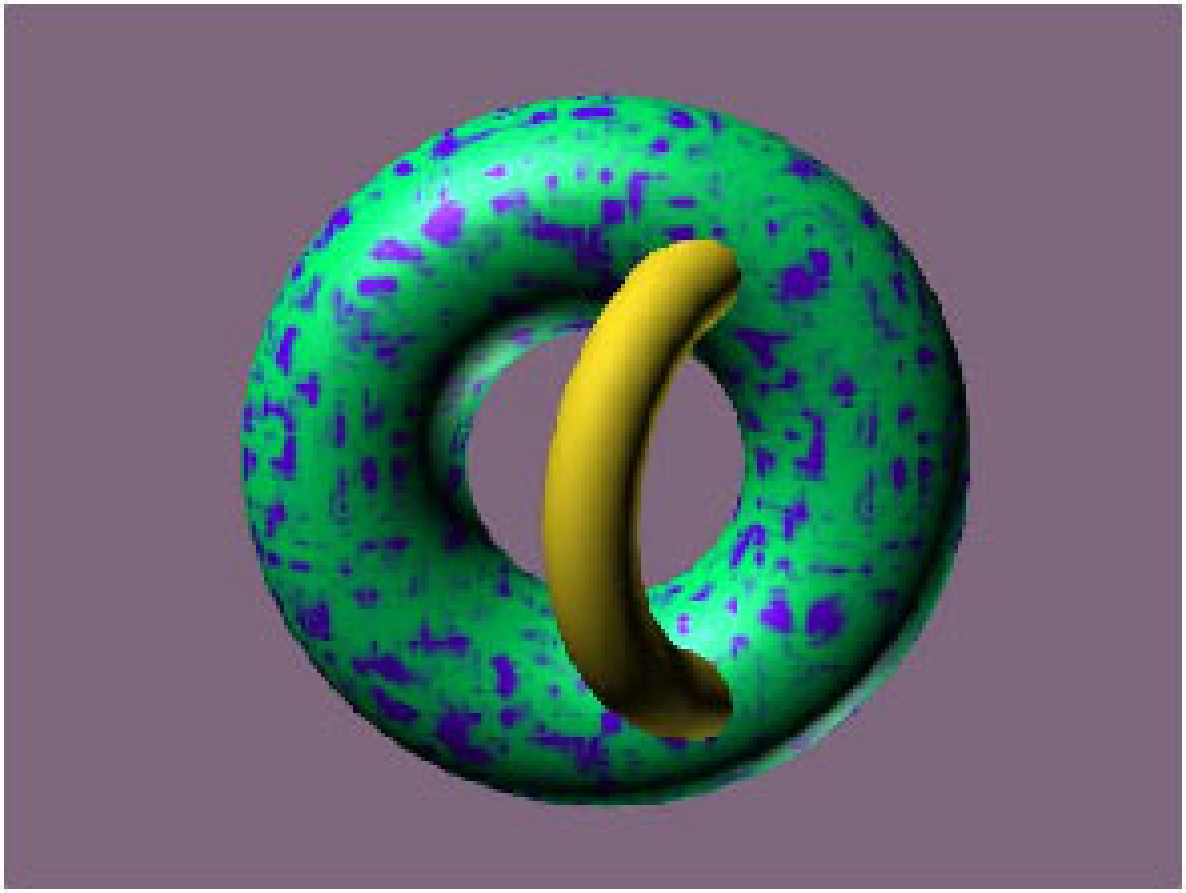
- 纹理映射是把纹理图象值映射到三维物体的表面的技术
- 纹理映射的问题
  - 改变物体的属性，可以产生纹理的效果，对简单光照明模型而言
    - 改变漫反射系数来改变物体的颜色
    - 改变物体表面的法向量

物体表面细节的模拟

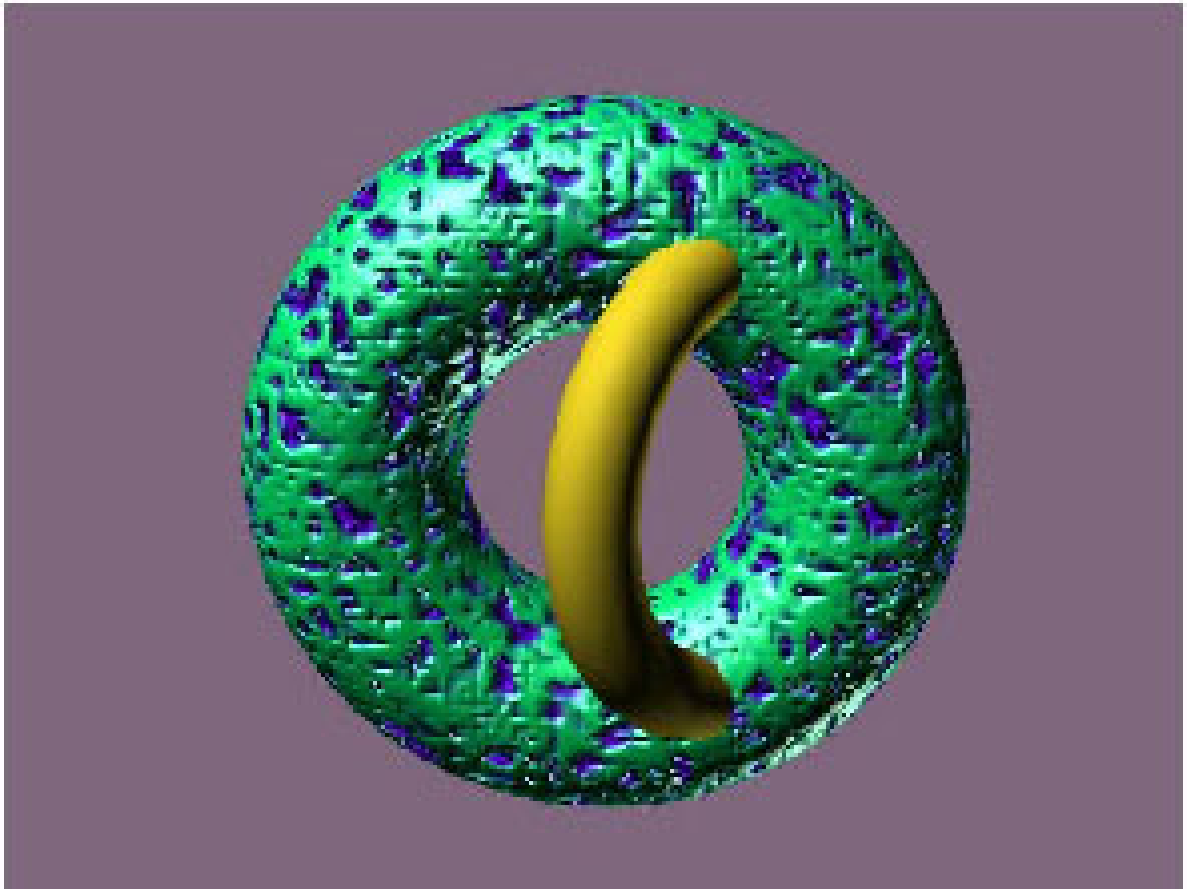
原始模型



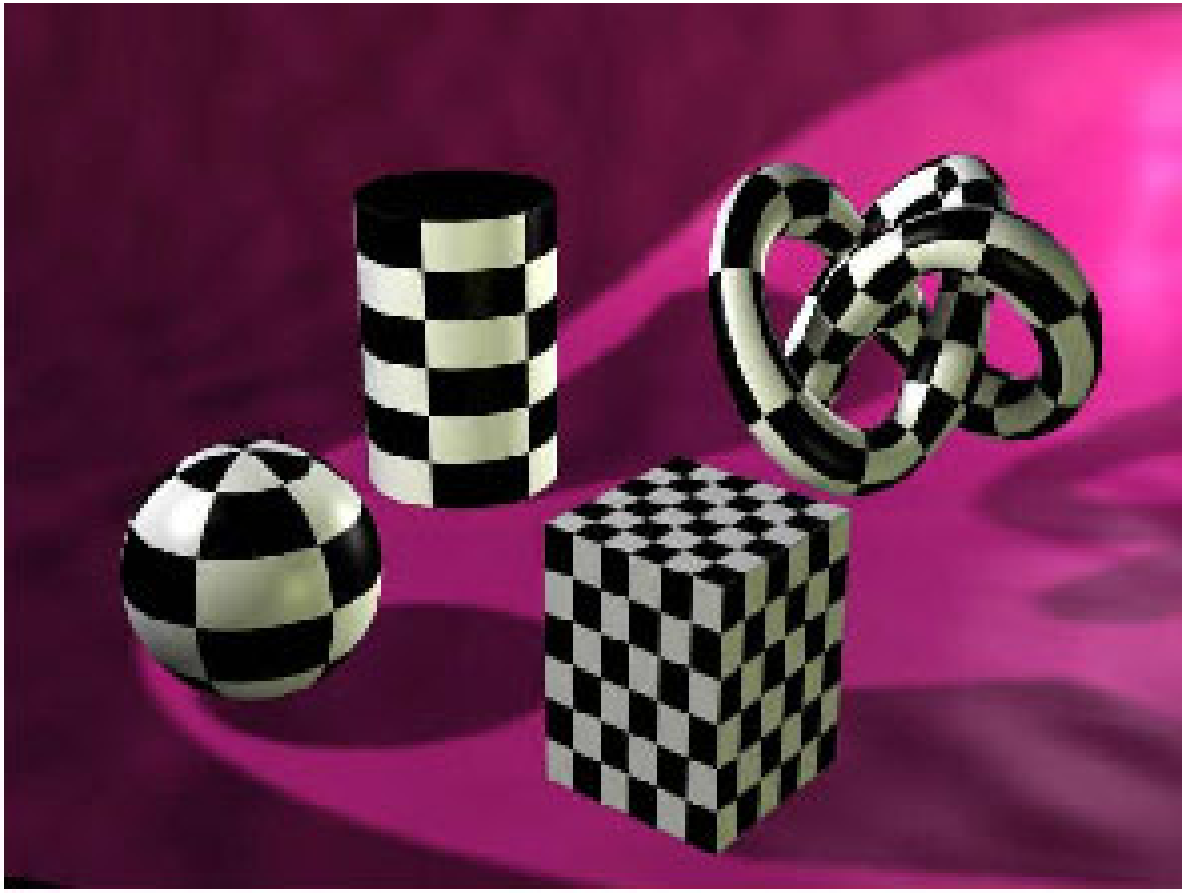
二维纹理



几何纹理

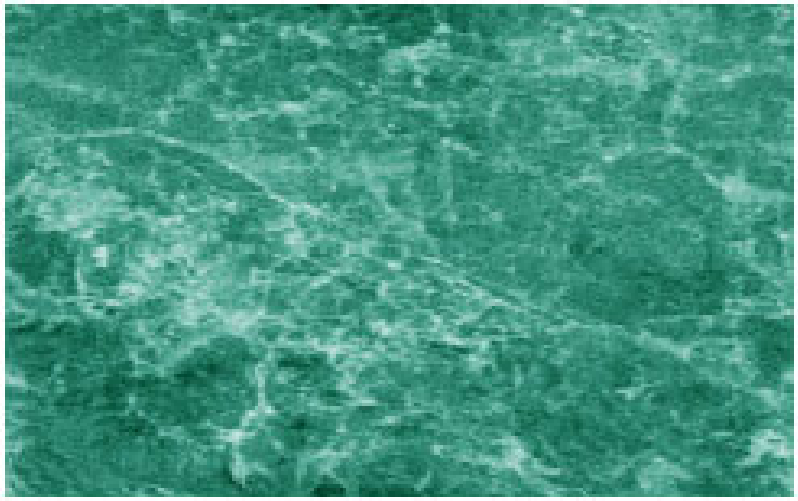


物体表面细节的模拟



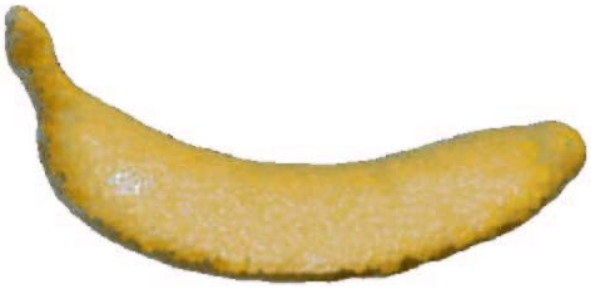
物体表面细节的模拟

物体表面细节的模拟

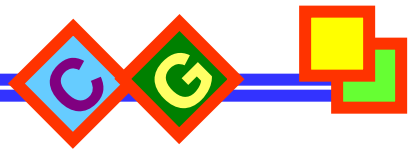


物体表面细节的模拟

纹理

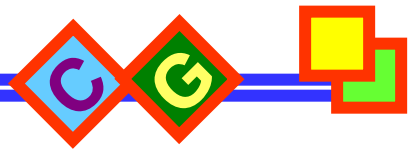


几何纹理映射技术



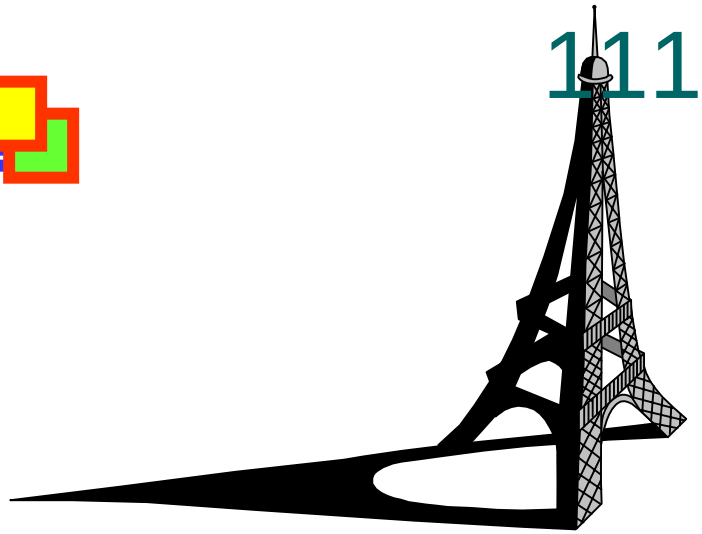
- 法向扰动法
  - 目标
    - 产生几何纹理，模拟凸凹不平的物体表面
  - 应用
    - 自然界中植物的表皮等
  - 方法
    - 对物体表面微观形状进行扰动

环境映射技术



- 目的是模拟景物镜面反射效果
- 本质是颜色映射技术
- 使用的纹理图像是当前被绘制物体周边的景物
  - 首先将纹理图像映射到中介面
  - 再将结果映射到最终的三维物体表面

阴影的生成



- 概念
  - 什么是阴影
    - 光源不能直接照射的区域
    - 对光源来说，不可见的面（隐藏面）

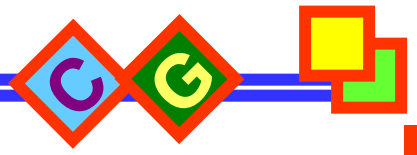
- **判断视点、光源以及物体之间的位置关系**
  - 从视点可见，从光源也可见
  - 从视点可见，从光源不可见
  - 相对于部分光源可见，相对于另一部分光源不可见

阴影的生成

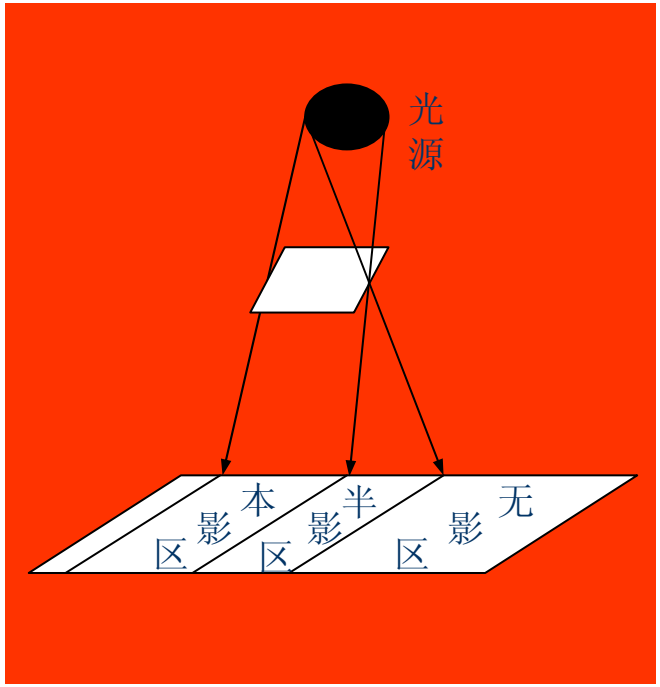
- 当观察方向与光源方向重合时
  - 观察者看不到任何阴影
  - 可以不进行阴影测试

- 当观察方向与光源方向不一致
- 或光源多且光源体制比较复杂时
  - 必须进行阴影处理

阴影的生成



- 阴影由两部分组成
  - *本影*
    - 任何光线都照不到的区域
    - 呈现为全黑的轮廓分明的区域
  - *半影*
    - 可接收到分布光源照射的部分光线的区域
    - 通常位于本影周围，呈现为半明半暗的区域



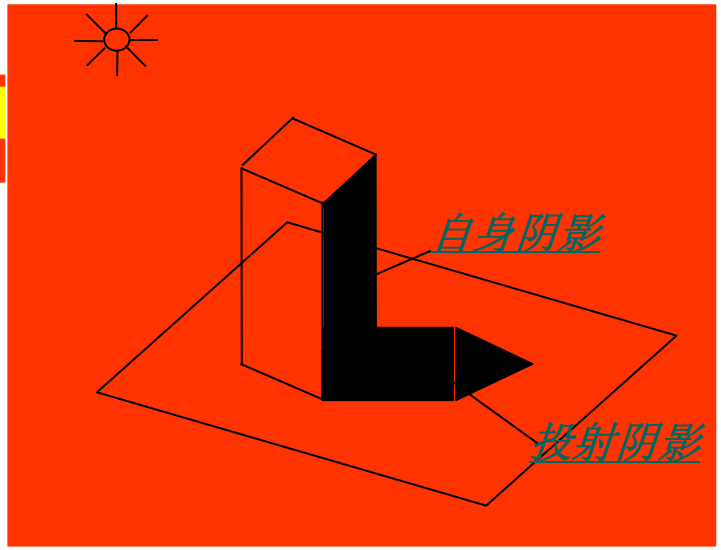
- *点光源*
  - 只能产生本影
- *位于有限距离内的分布光源*
  - 可同时产生本影和半影
    - 需要的阴影计算量大

阴影的生成

- 计算阴影的过程
- 相当于两次消隐过程
  - *对每个光源进行消隐*
  - *对视点的位置进行消隐*

- 好处
  - *改变视点位置，*第一次消隐过程不必重新计算

阴影的生成



- **产生的本影包括**
  - *自身阴影面*
    - 假设视点在点光源位置，用*背面剔除*的方法求出
  - *投射阴影*
    - 从光源向物体的所有可见面投射光线
    - 将这些面投影到场景中得到投影面
    - 将这些投影面与场景中其它平面求交线，可得阴影多边形

阴影的生成

- Z缓冲器算法
  - 步骤

1. 将所有景物变换到光源坐标系中，利用Z缓冲器算法按光线方向对景物进行消隐，把那些距光源最近的物体表面上点的深度值保存在阴影缓冲器中

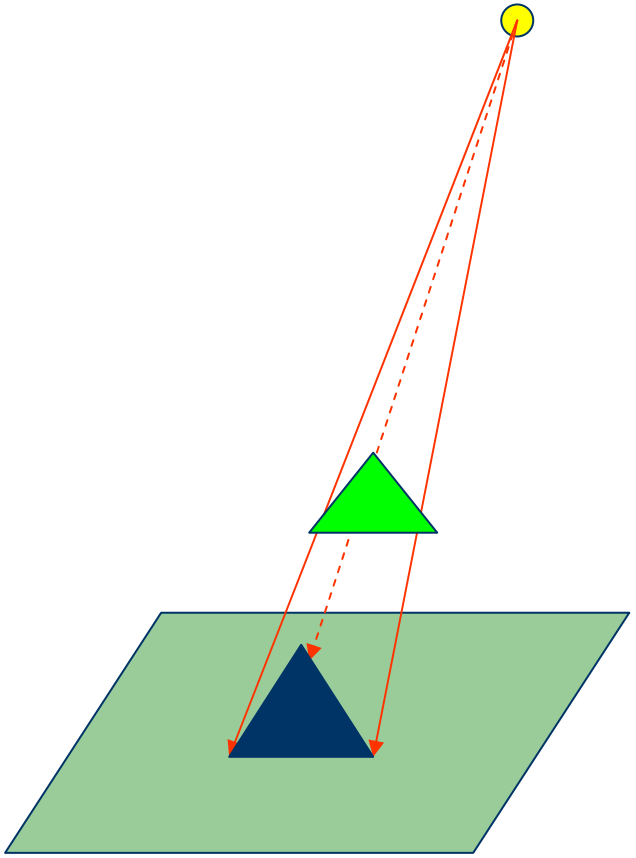
2. 利用Z缓冲器算法按视线方向对景物进行消隐，将得到的每一个可见点变换到光源坐标系中，若它在光源坐标系中的深度值比阴影缓冲器中相应单元的值小，则说明该可见点位于阴影中，否则不是

阴影的生成

- 优点
  - 算法简单
- 缺点
  - 每个光源需要一个阴影缓冲器

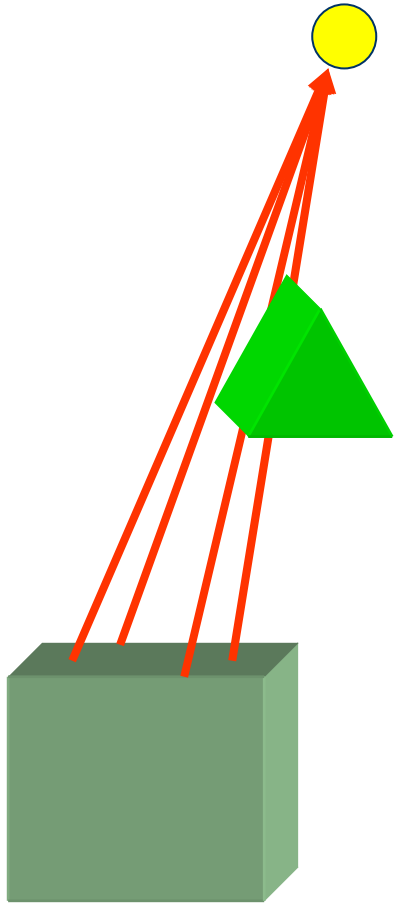
阴影的生成

- 阴影细节多边形
  - 算法步骤
    1. 在景物空间中，利用裁剪算法求出被光源直接照射的多边形或其一部分
    2. 将这些多边形作为表面细节贴在物体表面上

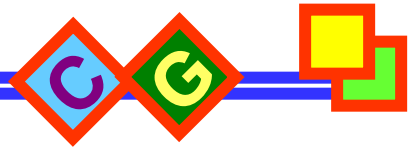


阴影的生成

- 光线跟踪
  - 从可见点P向光源发出测试光线，若该光线在到达光源之前与其它物体相交，则P点位于阴影区域中



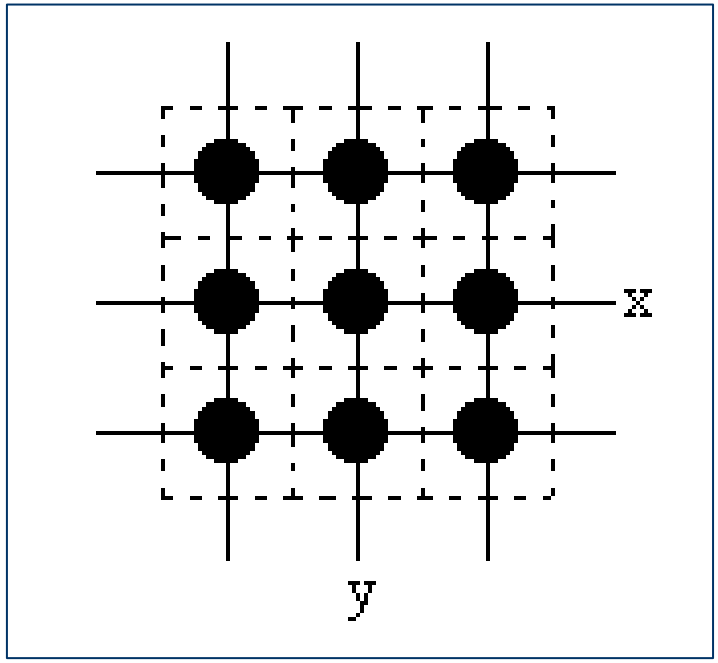
图形反走样技术



- 混淆现象（**Aliasing**）
- 反混淆方法（**Antialiasing**）
- 采样定理

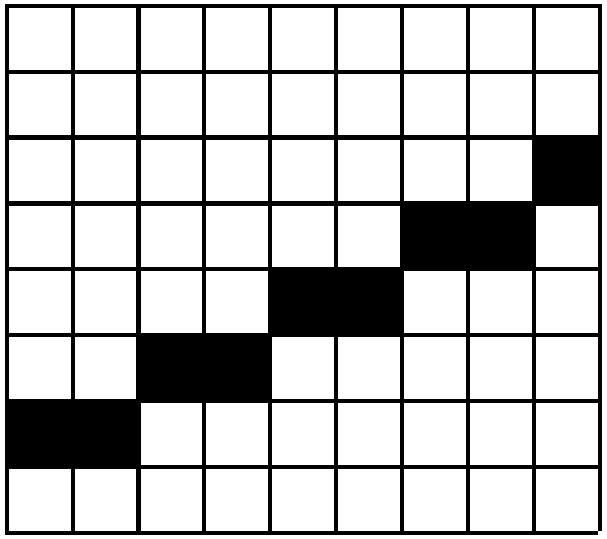
像素的表示

- 中心在**(x,y)**，边长为**1**的正方形

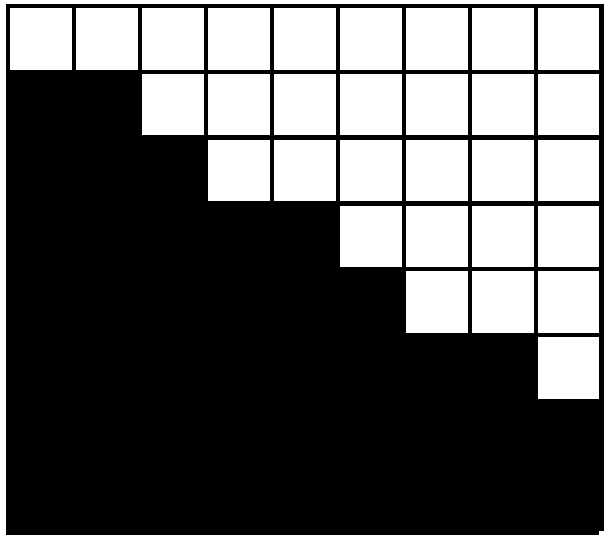


混淆现象（1/3）

- 不光滑(阶梯状）的图形边界



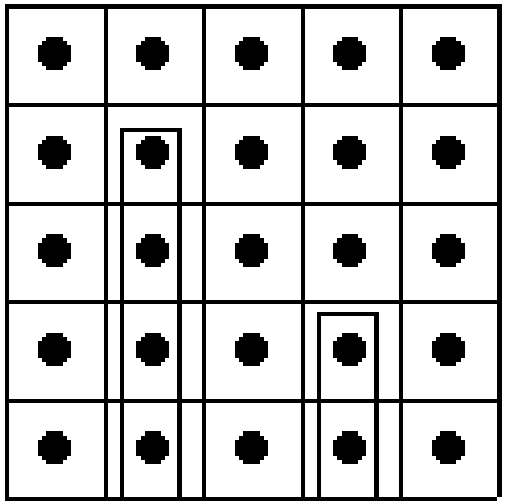
(a)



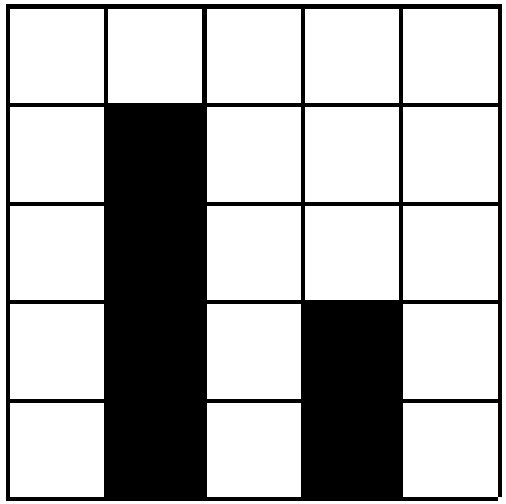
(b)

混淆现象（2/3）

- 图形细节失真



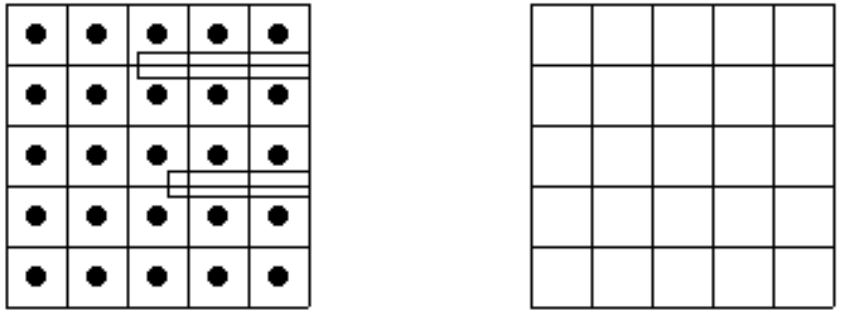
(a)

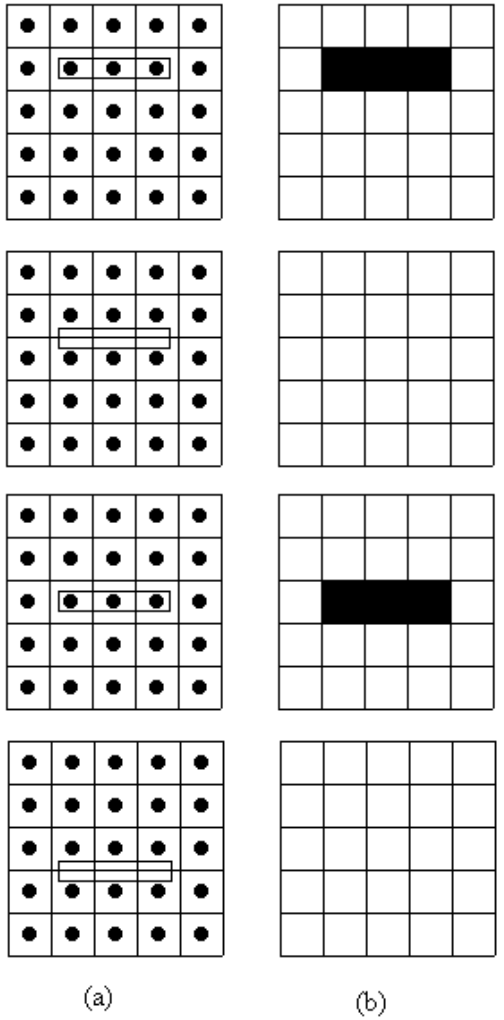


(b)

混淆现象（3/3）

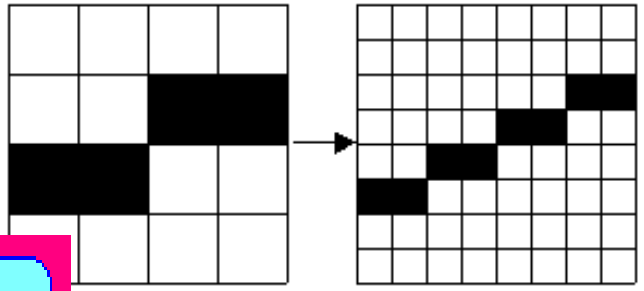
- 狭小图形的遗失与动态图形的闪烁

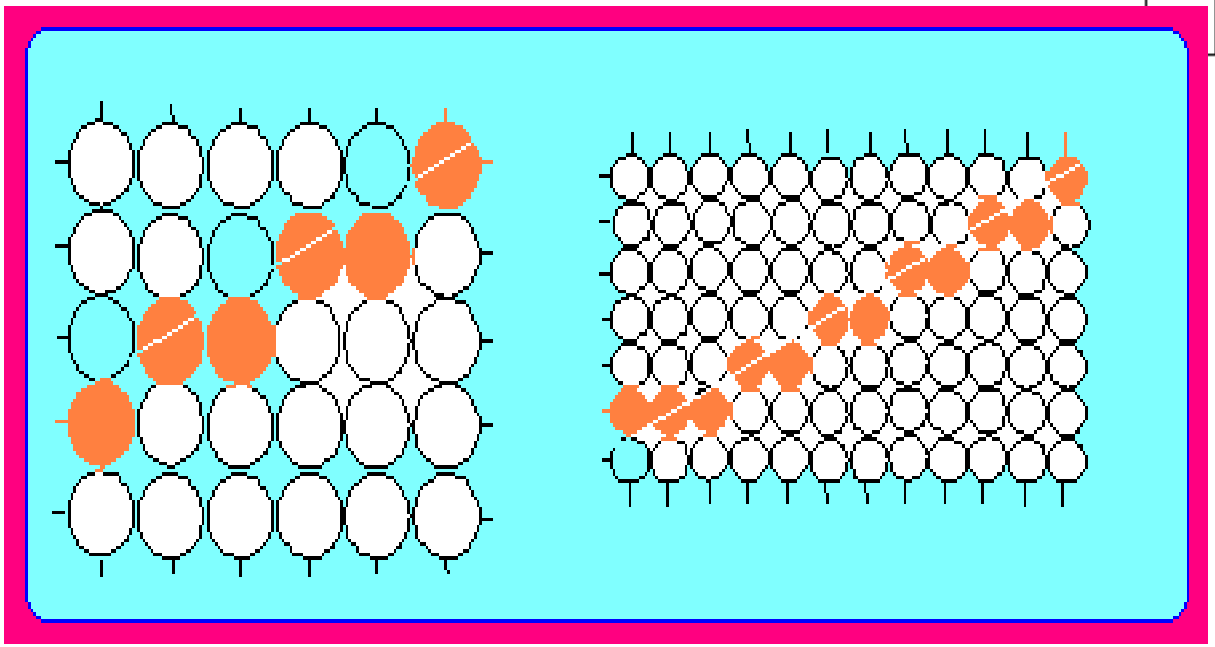




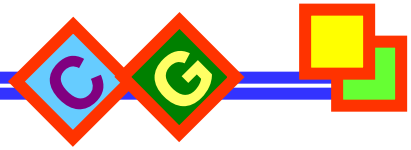
反混淆方法（1/9）

- 什么是反混淆
  - 在图形显示过程中，用于减少或消除混淆现象的方法
- 提高分辨率的反混淆方法





反混淆方法（2/9）

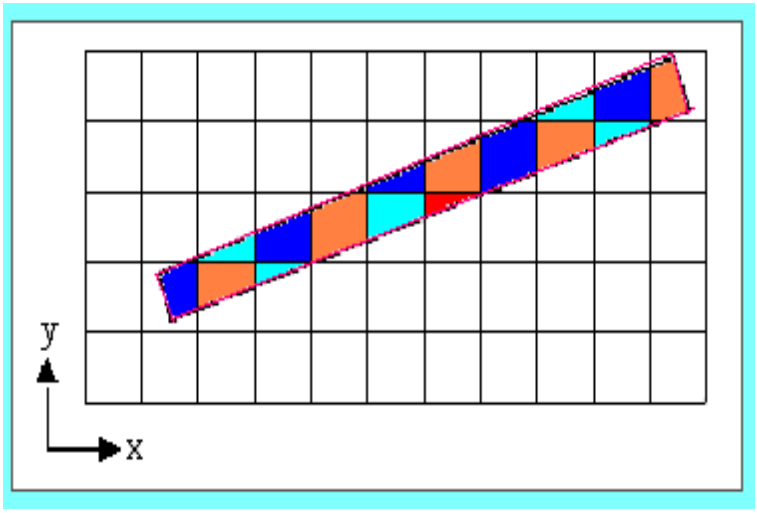


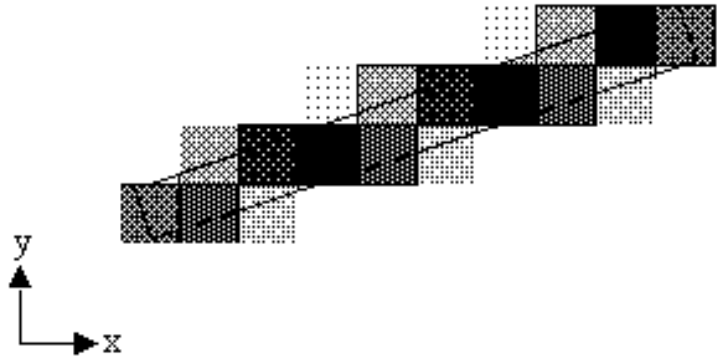
- 非加权区域采样方法
  - 两点假设
    1、像素是数学上抽象的点，它的面积为0，它的亮度由覆盖该点的图形的亮度所决定；
    2、直线段是数学上抽象直线段，它的宽度为0。
  - 现实
    - 像素的面积不为0；
    - 直线段的宽度至少为1个像素；
  - 假设与现实的矛盾是导致混淆出现的原因之一

反混淆方法（3/9）

- 解决方法：改变直线段模型
- 实现步骤：

1、将直线段看作具有一定宽度的狭长矩形；
2、当直线段与某象素有交时，求出两者相交区域的面积；
3、根据相交区域的面积，确定该象素的亮度值





OpenGL介绍

一、OpenGL的背景知识

- 是近几年发展起来的一个性能卓越的三维图形标准；
- 是在SGI等多家计算机公司倡导下，以SGI的GL三维图形库为基础制定的一个通用共享的开放式三维图形标准；
- **Microsoft、SGI、IBM、DEC、SUN、HP**等都采用了**OpenGL**作为三维图形标准；
- 许多硬件厂商提供对**OpenGL**的支持，是一个工业标准
- **OpenGL**独立于硬件设备、窗口系统和操作系统，以其为基础开发的应用程序可以在各种平台间移植。
- **OpenGL**可以用各种编程语言进行调用

OpenGL介绍

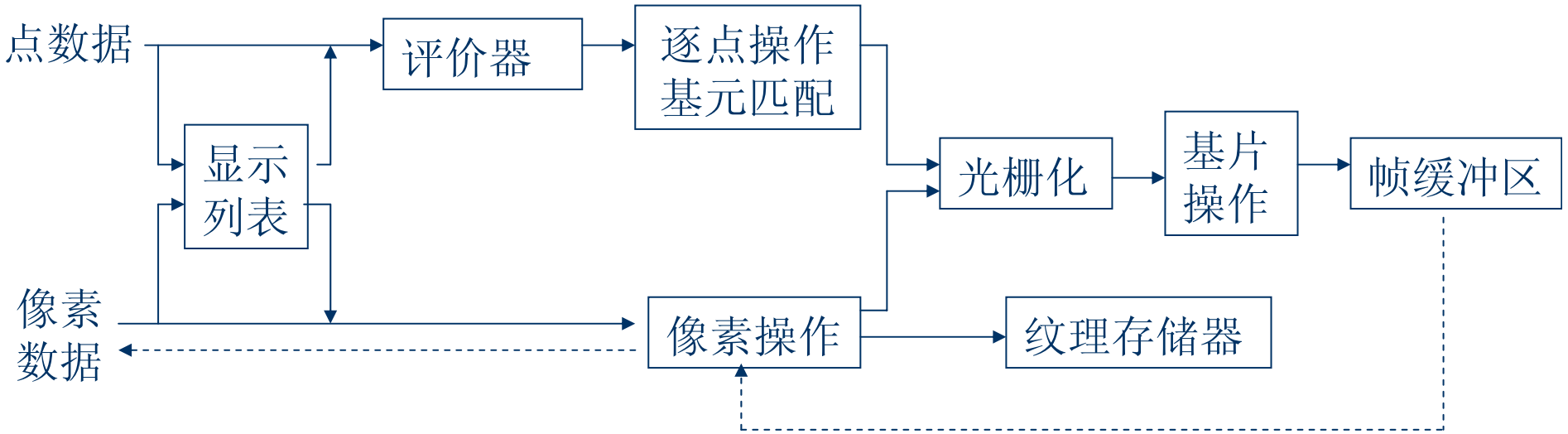
OpenGL：

图形硬件的软件接口

OpenGL是一种应用程序编程接口，而不是一种编程语言

Microsoft提供的OpenGL软件实现位于Opengl32.dll动态链接库中，位于Windows的System32目录

OpenGL的绘制流程和原理



OpenGL介绍

OpenGl函数及结构

- **OpenGL核心库，函数以gl开头**
- **OpenGL实用库，函数以glu开头**
- **辅助库，函数以aux开头(帮助初学者练习之用)**
- **Windows专用函数，用于连接OpenGL和Windows窗口系统，以wgl开头**
- **OpenGL实用函数工具包(GLUT)提供了与任意屏幕窗口系统进行交互的函数库，函数以glut开头。**

基本的OpenGL语法

函数命名的约定：

glColor3f(……)

函数库名　根命令　参数数量　参数类型

- 函数库名指明函数来自于哪个库
- 根命令表示这个函数相对应的OpenGL命令
- 参数数量和参数类型表示这个函数将接受的参数的个数和类型

OpenGL定义的常量都以GL开头，所有的字母大写，单词间以下划线来分隔，如GL_COLOR_BUFFER_BIT

OpenGL介绍

头文件：

```
#include  <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

注意，若使用

```
#include <GL/glut.h>
```

则不需要引入gl.h和glu.h，因为GLUT保证了它们的正确引入

可以引入C++程序需要的头文件

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

OpenGL介绍—命令后缀参数类型

后缀定义	数据类型	相应的C语言类型	OpenGL类型定义
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

二、OpenGL的主要功能

- 绘制模型：提供了绘制点、线、多边形、球、锥、多面体、茶壶等复杂的三维物体以及贝塞尔、NURBS等复杂曲线或曲面的绘制函数。

- 各种变换：提供了平移、旋转、变比和镜像四种基本变换以及平行投影和透视投影两种投影变换。通过变换实现三维的物体在二维的显示设备上显示。

- 着色模式：提供了RGBA模式和颜色索引两种颜色的显示方式。

- 光照处理：在自然界我们所见到的物体都是由其材质和光照相互作用的结果，OpenGL提供了辐射光(Emitted Light)、环境光(Ambient Light)、漫反射光(Diffuse Light)和镜面光(Specular Light)。材质是指物体表面对光的反射特性，在OpenGL中用光的反射率来表示材质。

二、OpenGL的主要功能

➢纹理映射(Texture Mapping)：将真实感的纹理粘贴在物体表面，使物体逼真生动。纹理是数据的简单矩阵排列，数据有颜色数据、亮度数据和alpha数据。

➢位图和图像：提供了一系列函数来实现位图和图像的操作。位图和图像数据均采用像素的矩阵形式表示。

➢制作动画：提供了双缓存(Double Buffering)技术来实现动画绘制。双缓存即前台缓存和后台缓存，后台缓存用来计算场景、生成画面，前台缓存用来显示后台缓存已经画好的画面。当画完一帧时，交互两个缓存，这样循环交替以产生平滑动画。

二、OpenGL的主要功能

- 选择和反馈：**OpenGL**为支持交互式应用程序设计了选择操作模式和反馈模式。在选择模式下，则可以确定用户鼠标指定或拾取的是哪一个物体，可以决定将把哪些图元绘入窗口的某个区域。而反馈模式，**OpenGL**把即将光栅化的图元信息反馈给应用程序，而不是用于绘图。
- 反走样技术
- 深度暗示**(Depth Cue)**
- 运动模糊**(Motion Blur)**
- 雾化**(Fog)**

OpenGL生成基本图形

OpenGL提供了描述点、线、多边形的绘制机制。它们通过glBegin()函数和glEnd()函数配对来完成。glBegin()函数有一个类型为Glenum的参数，它的取值见下表。gLEnd()函数标志着形状的结束，该函数没有参数。

Mode 的值	解释
GL_POINTS	一系列独立的点
GL_LINES	每两点相连成为线段
GL_POLYGON	简单凸多边形的边界
GL_TRIANGLES	三点相连成为一个三角形
GL_QUADS	四点相连成为一个四边形
GL_LINE_STRIP	顶点相连成为一系列线段
GL_LINE_LOOP	顶点相连成为一系列线段，连接最后一点与第一点
GL_TRIANGLE_STRIP	相连的三角形带
GL_TRIANGLE_FAN	相连的三角形扇形
GL_QUAD_STRIP	相连的四边形带

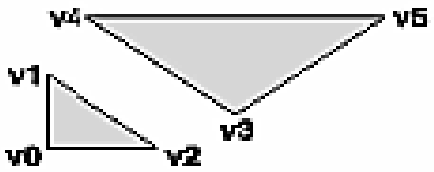
OpenGL生成基本图形

GL_POINTS

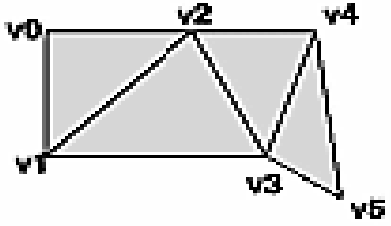
GL_LINES

GL_LINE_STRIP

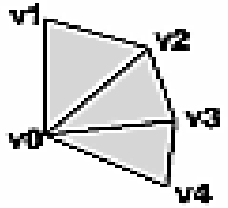
GL_LINE_LOOP



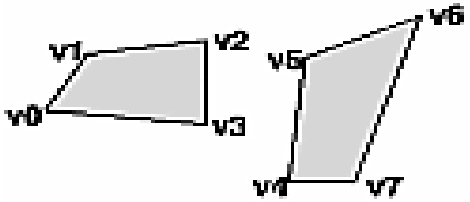
GL_TRIANGLES



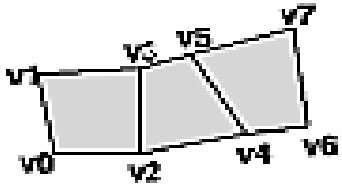
GL_TRIANGLE_STRIP



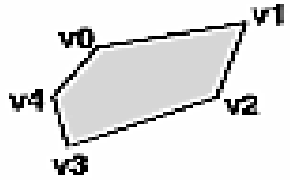
GL_TRIANGLE_FAN



GL_QUADS



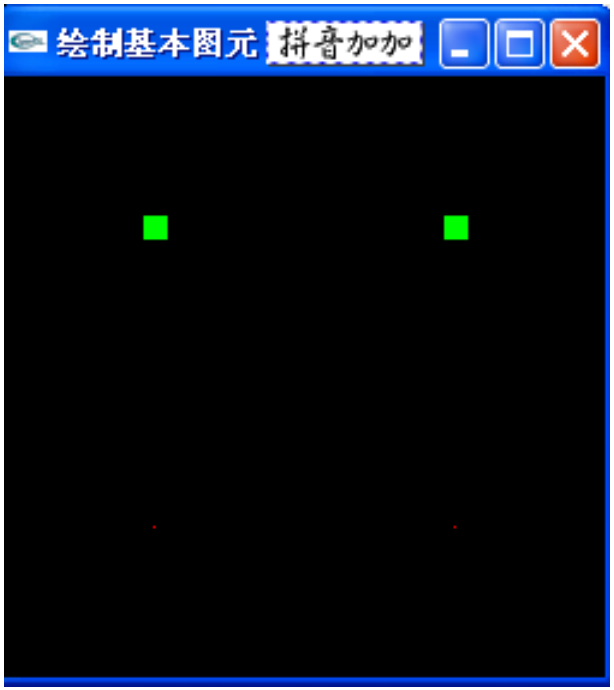
GL_QUAD_STRIP



GL_POLYGON

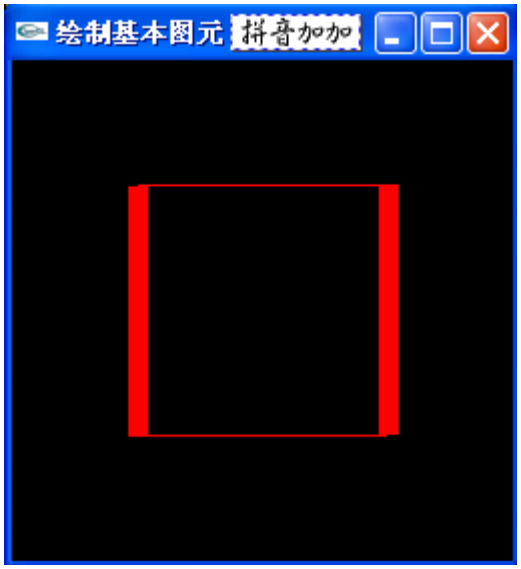
OpenGL生成基本图形－点


```
//绘制点
   glColor3f (1.0, 0.0, 0.0);
   glBegin(GL_POINTS);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
   glEnd();
   glColor3f(0.0, 1.0, 0.0);
   glPointSize(10.0f);//绘制有宽度的点
   glBegin(GL_POINTS);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();
```



OpenGL生成基本图形—线

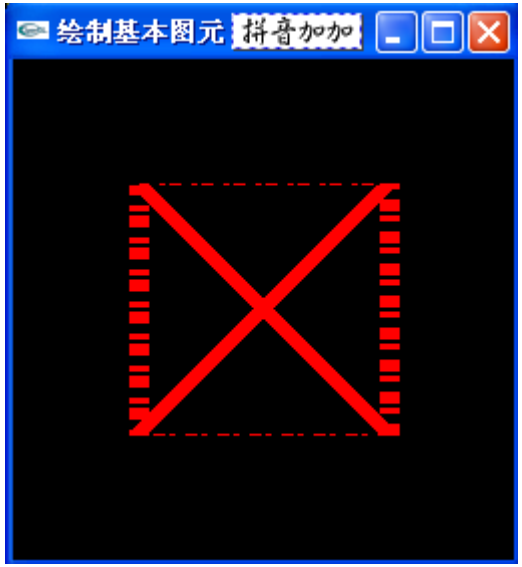

```
//绘制线
  glColor3f (1.0, 0.0, 0.0);
  glBegin(GL_LINES);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.75, 0.25, 0.0);
    glVertex3f (0.75, 0.75, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
  glEnd();
  glLineWidth(10.0f);//设置线的宽度
  glBegin(GL_LINES);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
   glVertex3f (0.75, 0.75, 0.0);
   glVertex3f (0.75, 0.25, 0.0);
  glEnd();
```



OpenGL生成基本图形－点划线

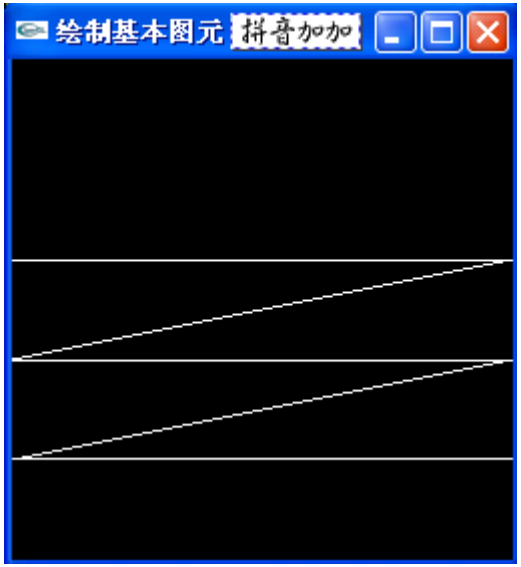

```
//绘制点划线
  glLineStipple(1,0x3f07);//设置点画模式
  glEnable(GL_LINE_STIPPLE);//激活点画线模
  glColor3f (1.0, 0.0, 0.0);
  glBegin(GL_LINES);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.75, 0.25, 0.0);
    glVertex3f (0.75, 0.75, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
  glEnd();
  glLineWidth(10.0f);//设置线的宽度
  glBegin(GL_LINES);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
    glVertex3f (0.75, 0.75, 0.0);
    glVertex3f (0.75, 0.25, 0.0);
  glEnd();
```

```
glDisable(GL_LINE_STIPPLE);//取消点画模式
  glBegin(GL_LINES);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.75, 0.75, 0.0);
    glVertex3f (0.75, 0.25, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
  glEnd();
```



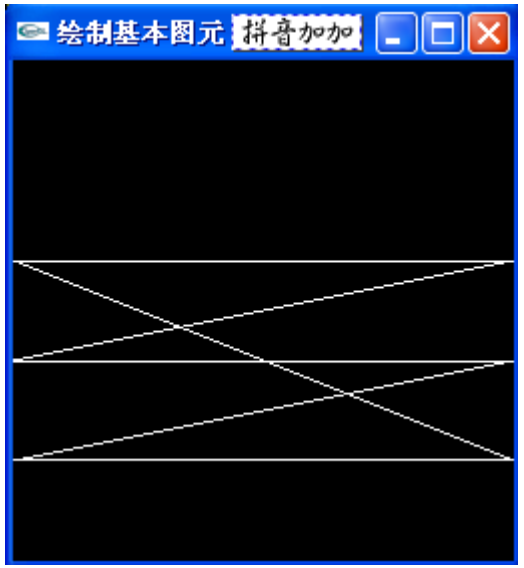
OpenGL生成基本图形—折线


```
//绘制折线
  glBegin(GL_LINE_STRIP);
      glVertex2f(0.0f,0.6f);
      glVertex2f(1.0f,0.6f);
      glVertex2f(0.0f,0.4f);
      glVertex2f(1.0f,0.4f);
     glVertex2f(0.0f,0.2f);
      glVertex2f(1.0f,0.2f);
glEnd();
```



OpenGL生成基本图形－闭合折线


```
glBegin(GL_LINE_LOOP);
    glVertex2f(0.0f,0.6f);
    glVertex2f(1.0f,0.6f);
    glVertex2f(0.0f,0.4f);
    glVertex2f(1.0f,0.4f);
    glVertex2f(0.0f,0.2f);
    glVertex2f(1.0f,0.2f);
glEnd();
```



OpenGL生成基本图形－绘制多边形

- **OpenGL的多边形必须至少有三个顶点**
- 直线不能相交，多边形构成单连通的凸区域
- 一个多边形有前面和后面之分
- 前面和后面可以有不同的属性

Void glPolygonMode(Glenum face,Glenum mode);

该函数控制多边形绘制模式是正面还是反面，是以点、轮廓还是填充的形式画出。默认为正面和反面都以填充的形式画出

face可取：GL_FRONT_AND_BACK、GL_FRONT、GL_BACK

mode可取：GL_POINT、GL_LINE、GL_FILL

OpenGL生成基本图形－绘制多边形

void glFrontFace(Glenum mode);

该函数控制如何确定正面多边形，默认情况下，参数**mode**为**GL_CCW(**逆时针**)**，也可以指定为**GL_CW(**顺时针**)**

void glCullFace(Glenum mode);

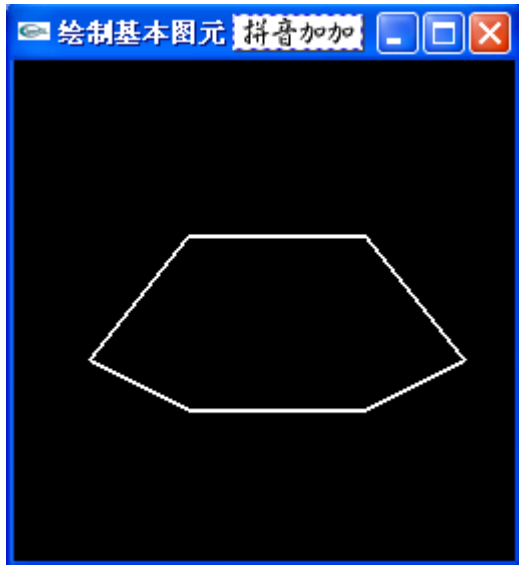
该函数指出在变换到屏幕坐标之前，舍弃哪个面的多边形。**mode**可以为**GL_FRONT**、**GL_BACK** 、**GL_FRONT_AND_BACK**，必须使用

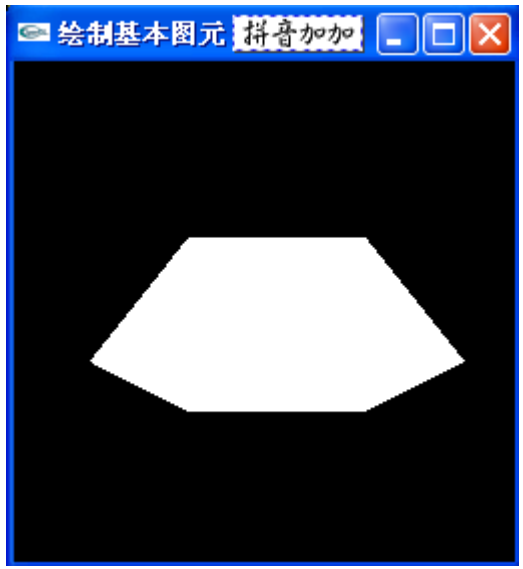
```
glEnable(GL_CULL_FACE);//激活拣选模式

glDisable(GL_CULL_FACE);//使拣选失效
```

OpenGL生成基本图形－绘制多边形


```
//绘制多边形
 glLineWidth(2.0f);
 glPolygonMode(GL_FRONT,GL_LINE);
 glPolygonMode(GL_BACK,GL_FILL);
 glFrontFace(GL_CCW);
// glFrontFace(GL_CW);
 glBegin(GL_POLYGON);
   glVertex2f(-0.3f,0.3f);
   glVertex2f(-0.7f,-0.2f);
   glVertex2f(-0.3f,-0.4f);
   glVertex2f(0.4f,-0.4f);
   glVertex2f(0.8f,-0.2f);
   glVertex2f(0.4f,0.3f);
 glEnd();
```

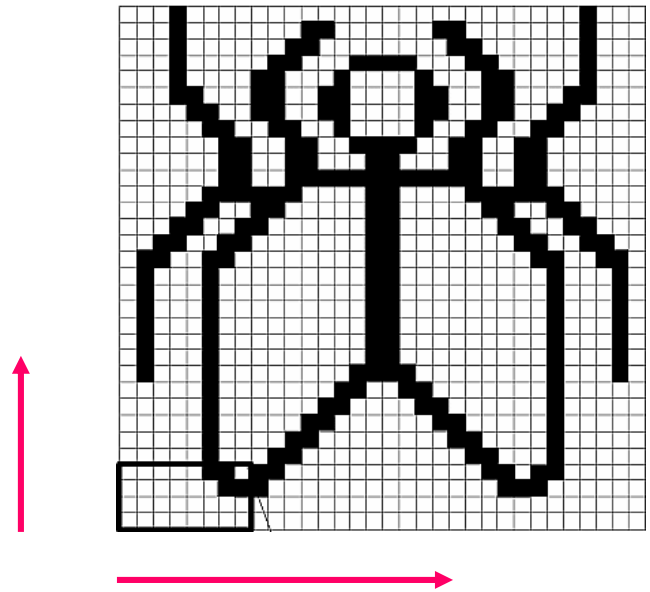
若替换为此函数，结果为





OpenGL生成基本图形－多边形填充模式

- **OpenGL可以让用户自定义多边形填充模式，该填充模式必须是一个32×32的位图**
- **其中的每一个位代表屏幕上的一个点。被填充的位的值为1，未被填充的值为0**
- **将填充模式表示成16进制，表示该位图以一个字节为单位**
- **从左下角开始自左而右自下而上地表示。**



OpenGL生成基本图形－多边形填充模式

void glPolygonStipple(const Glubyte *mask);

该函数为填充多边形定义当前的点画模式

void glEnable(GL_POLYGON_STIPPLE);

该函数使多边形点画模式激活

void glDisable(GL_POLYGON_STIPPLE);

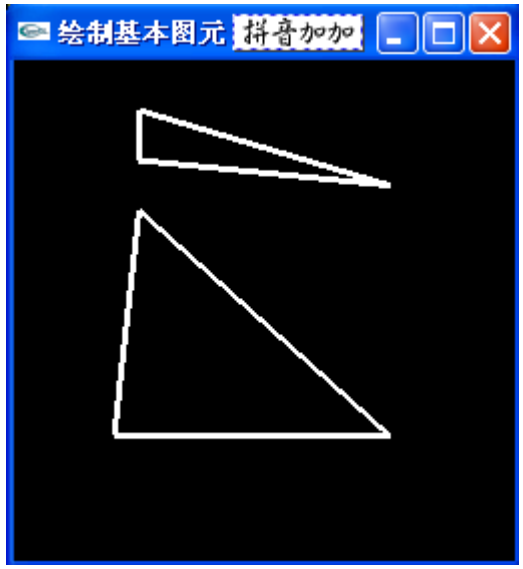
该函数使多边形点画模式失效

OpenGL生成基本图形－多边形填充模式


```
//定义多边形点画模式 为fly
   glLineWidth(2.0f);
   glPolygonMode(GL_FRONT,GL_FILL);
   glFrontFace(GL_CCW);
  glEnable(GL_POLYGON_STIPPLE);
  glPolygonStipple(fly);
  glBegin(GL_POLYGON);
       glVertex2f(-0.3f,0.3f);
              glVertex2f(-0.7f,-0.2f);
       glVertex2f(-0.3f,-0.4f);
       glVertex2f(0.4f,-0.4f);
       glVertex2f(0.8f,-0.2f);
       glVertex2f(0.4f,0.3f);
  glEnd();
  glDisable(GL_POLYGON_STIPPLE);
```

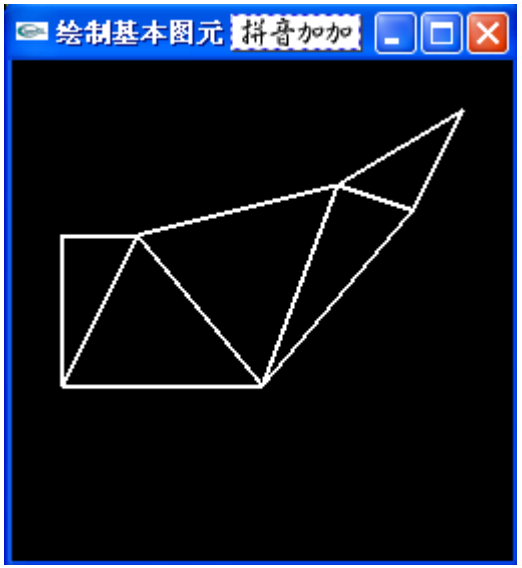
OpenGL生成基本图形—绘制三角形

```
//绘制三角形
  glLineWidth(3.0f);
  glPolygonMode(GL_FRONT,GL_LINE);
  glBegin(GL_TRIANGLES);
     glVertex2f(-0.5f,0.4f);
     glVertex2f(-0.6f,-0.5f);
     glVertex2f(0.5f,-0.5f);

     glVertex2f(-0.5f,0.8f);
     glVertex2f(-0.5f,0.6f);
     glVertex2f(0.5f,0.5f);
  glEnd();
```



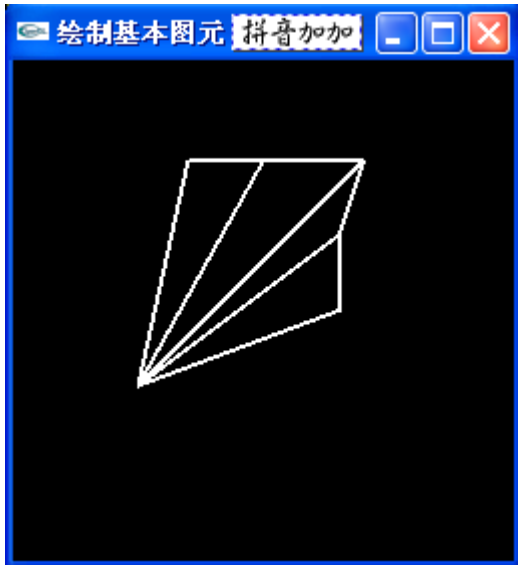
OpenGL生成基本图形—绘制三角形片


```
//绘制三角形片
 glLineWidth(2.0f);
 glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
 glBegin(GL_TRIANGLE_STRIP);
    glVertex2f(-0.8f,0.3f);
    glVertex2f(-0.8f,-0.3f);
    glVertex2f(-0.5f,0.3f);
    glVertex2f(0.0f,-0.3f);
    glVertex2f(0.3f,0.5f);
    glVertex2f(0.6f,0.4f);
    glVertex2f(0.8f,0.8f);
 glEnd();
```



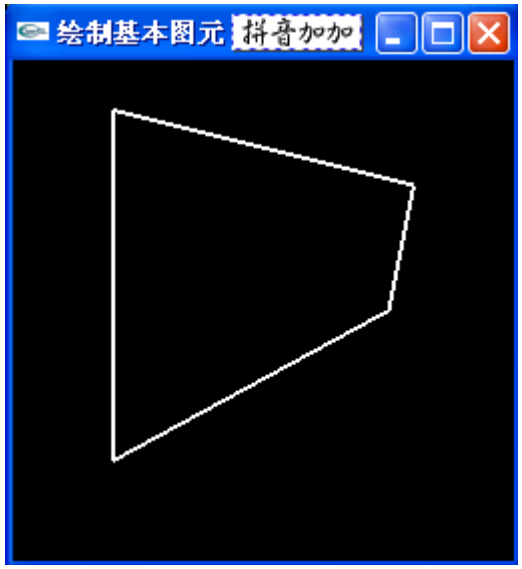
OpenGL生成基本图形—绘制三角形扇


```
//绘制三角形扇
  glLineWidth(2.0f);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
  glBegin(GL_TRIANGLE_FAN);
     glVertex2f(-0.5f,-0.3f);
     glVertex2f(0.3f,-0.0f);
     glVertex2f(0.3f,0.3f);
     glVertex2f(0.4f,0.6f);
     glVertex2f(0.0f,0.6f);
     glVertex2f(-0.3f,0.6f);
  glEnd();
```



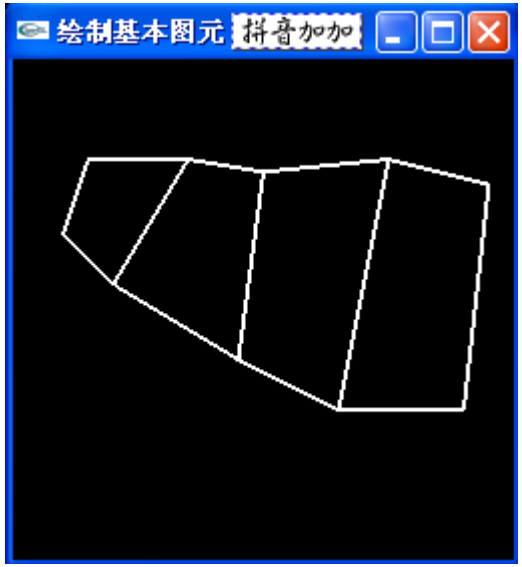
OpenGL生成基本图形—绘制四边形


```
//绘制四边形
  glLineWidth(2.0f);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
  glBegin(GL_QUIDS);
    glVertex2f(-0.6f,0.8f);
    glVertex2f(-0.6f,-0.6f);
    glVertex2f(0.5f,0.0f);
    glVertex2f(0.6f,0.5f);
  glEnd();
```

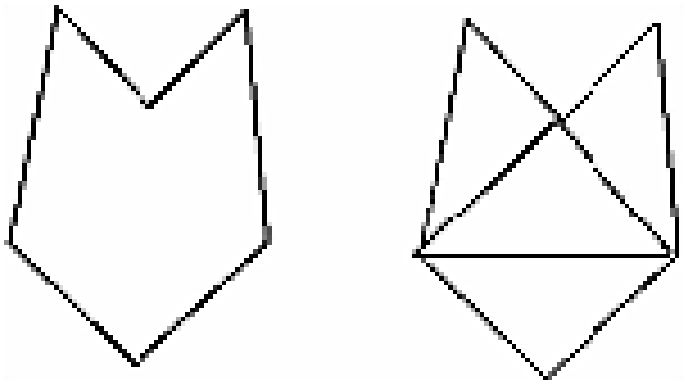


OpenGL生成基本图形—绘制四边形片


```
//绘制四边形片
glLineWidth(2.0f);
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
glBegin(GL_QUAD_STRIP);
  glVertex2f(-0.8f,0.3f);
  glVertex2f(-0.7f,0.6f);
  glVertex2f(-0.6f,0.1f);
  glVertex2f(-0.3f,0.6f);
  glVertex2f(-0.1f,-0.2f);
  glVertex2f(0.0f,0.55f);
  glVertex2f(0.3f,-0.4f);
  glVertex2f(0.5f,0.6f);
  glVertex2f(0.8f,-0.4f);
  glVertex2f(0.9f,0.5f);
glEnd();
```



OpenGL生成基本图形—边的可见性

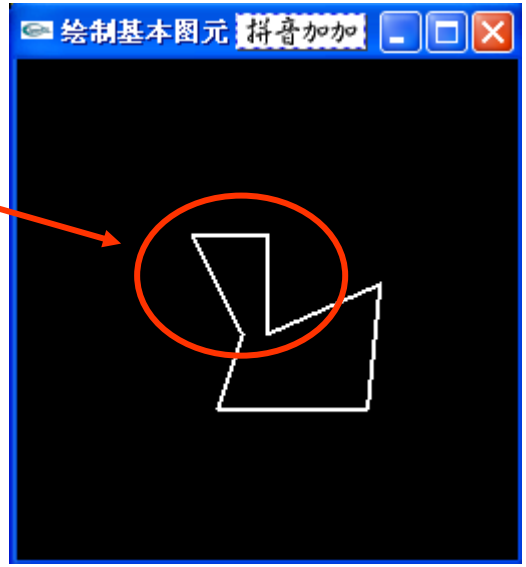


void glEdgeFlag(GLboolean flag);

该函数说明边的可见性，其参数为**GL_TRUE**，则该边可见，为**GL_FALSE**，则该边不可见的，该函数位于两个**glVertex()**函数之前，说明这两个顶点构成的边的可见性。该函数不适合为三角形切片或四边形切边指定顶点。

OpenGL生成基本图形—边的可见性

```
//边的可见性
  glLineWidth(2.0f);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
  glBegin(GL_POLYGON);
    glEdgeFlag(GL_TRUE);
    glVertex2f(-0.3f,0.3f);
    glEdgeFlag(GL_FALSE);
    glVertex2f(-0.1f,-0.1f);
    glEdgeFlag(GL_TRUE);
    glVertex2f(0.0f,-0.1f);
    glVertex2f(0.0f,0.3f);
  glEnd();
 ……
```

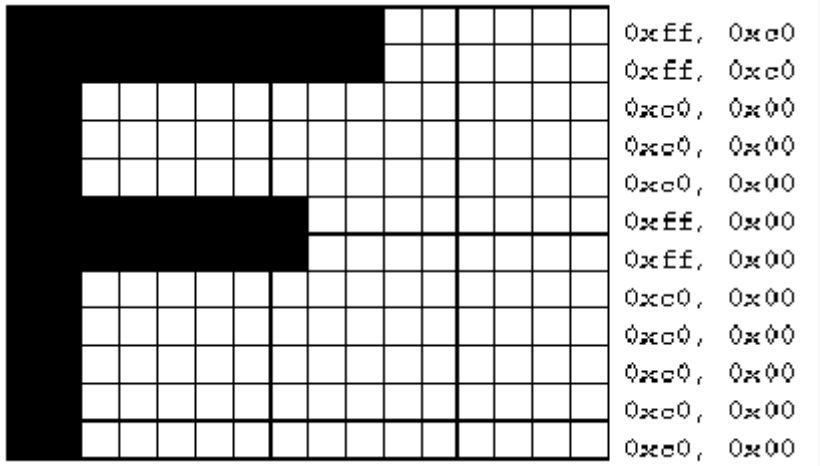


OpenGL生成基本图形—生成英文字符

- 位图字体
- 轮廓字体
- 纹理映射字体

} 三种渲染字体的方法

位图：位图是0和1组成的矩阵，只有每一像素的一位信息。



位图化的F及其数据

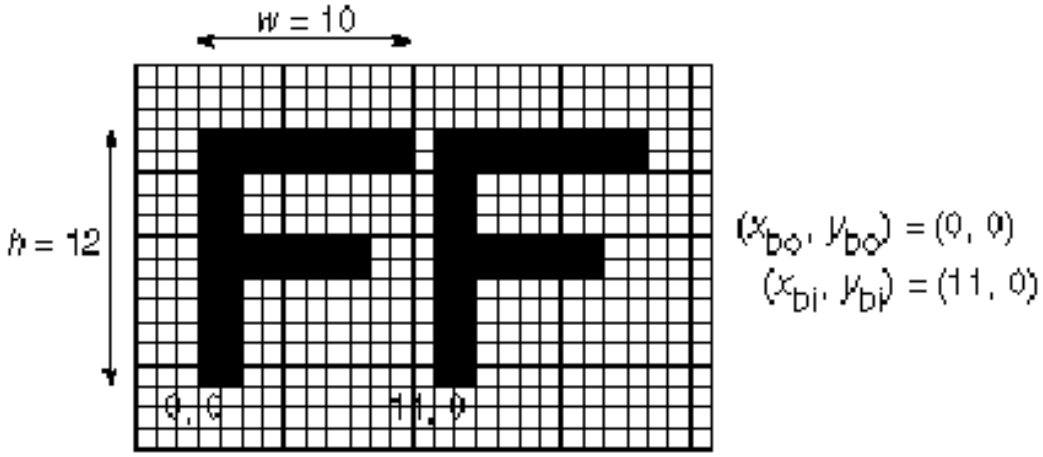
OpenGL生成基本图形—生成英文字符

glRasterPos{234}{sifd}(TYPE x,TYPE y,TYPE z,TYPE w);

该函数设置当前光栅位置，即将要绘制的位图的原点。

void glBitmap(GLsizei width,GLsizei height,GLfloat x_{bo},GLfloat y_{bo},GLfloat x_{bi},GLfloat y_{bi},const GLubyte *bitmap);

该函数用来绘制位图



OpenGL生成基本图形—生成英文字符


```
glPixelStorei(GL_UNPACK_ALIGNMENT,1
);
//设置像素存储模式
 GLubyte rasters[24] = {
      0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0,
0x00,       0xc0, 0x00,   0xff, 0x00, 0xff, 0x00,
0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,   0xff, 0xc0,
0xff, 0xc0};
   glRasterPos2i(0, 0);
   glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
   glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
   glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
```