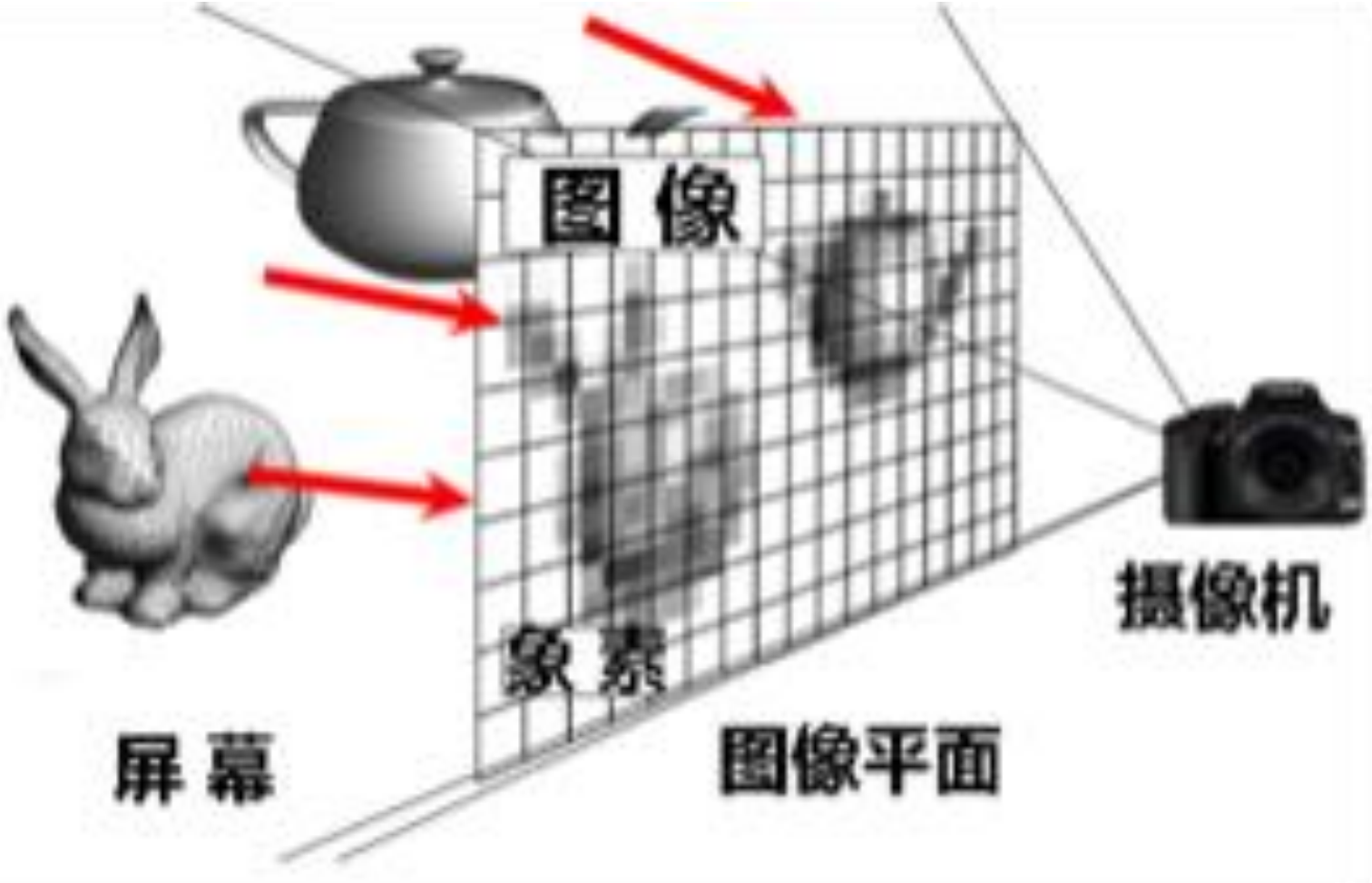
中国科学技术大学
University of Science and Technology of China

计算机图形学
Computer Graphics

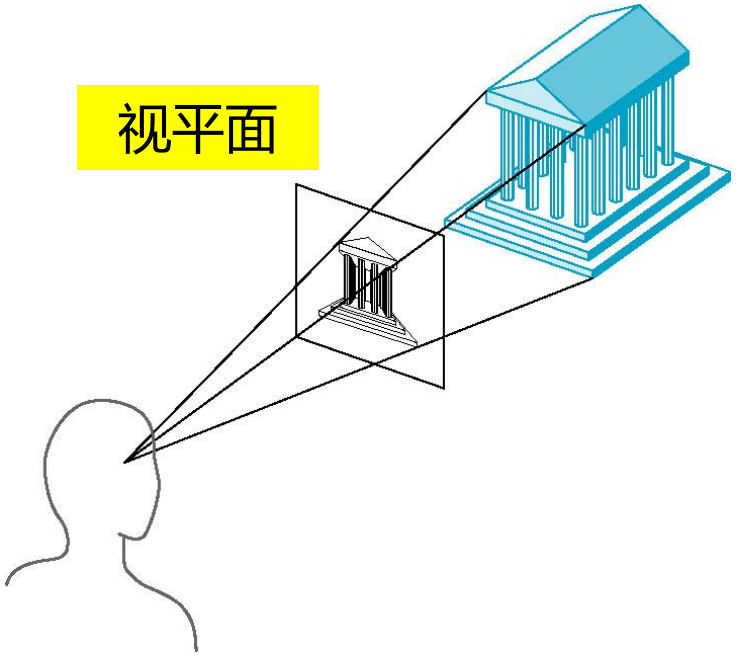
陈仁杰
renjiec@ustc.edu.cn
http://staff.ustc.edu.cn/~renjiec

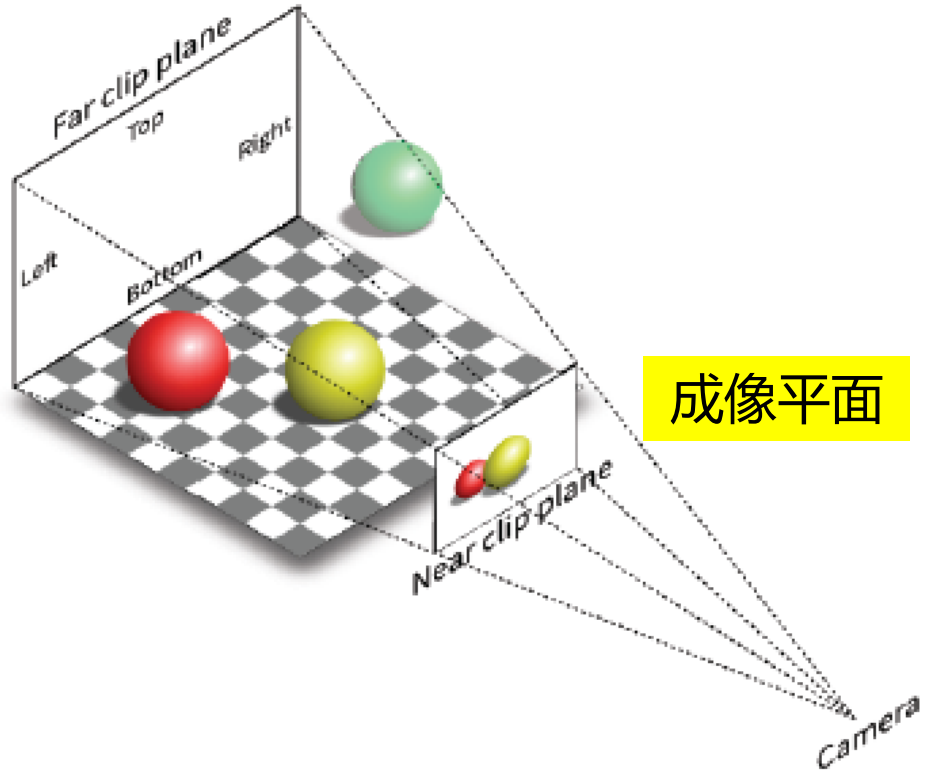
实时渲染流水线/管线
Realtime Rendering Pipeline

渲染/绘制：将3D模型成像到2D图像平面

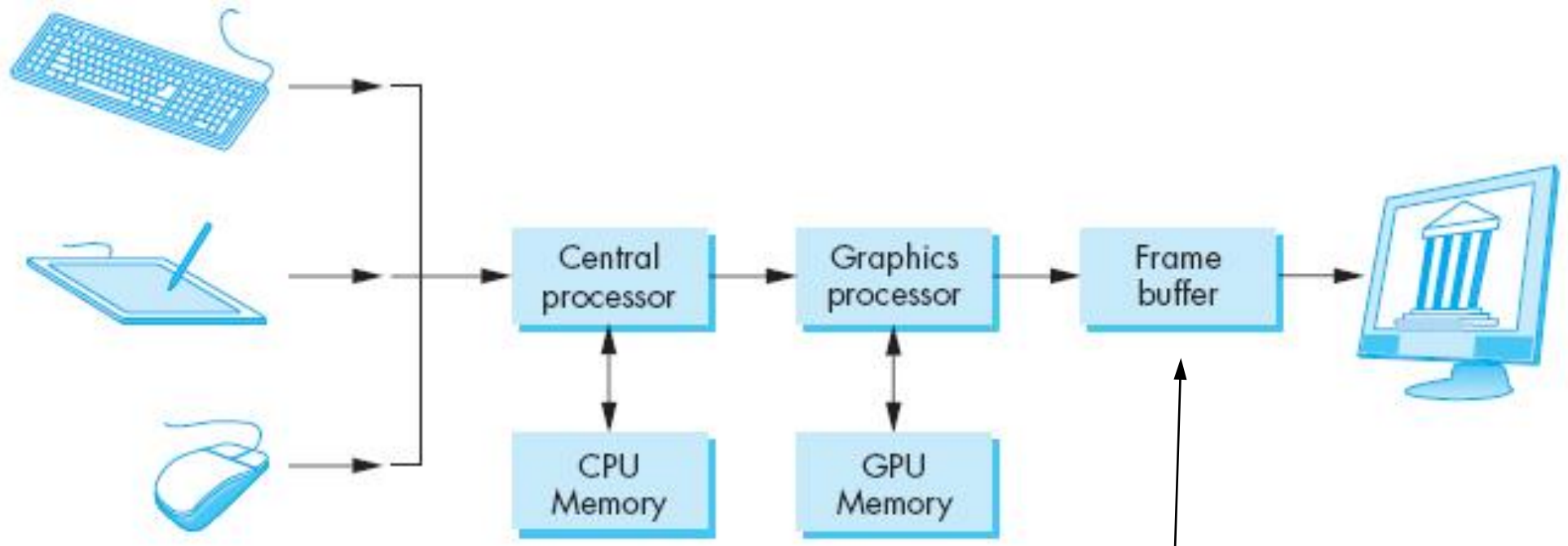


渲染/绘制：3D场景→2D图像





3D应用程序



输入设备

图像在帧缓冲区形成

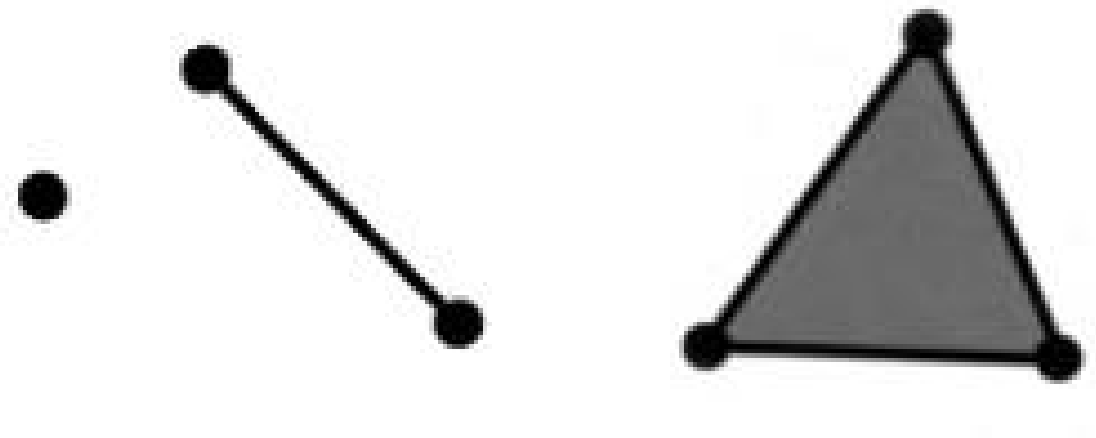
输出设备

渲染的两个主要阶段

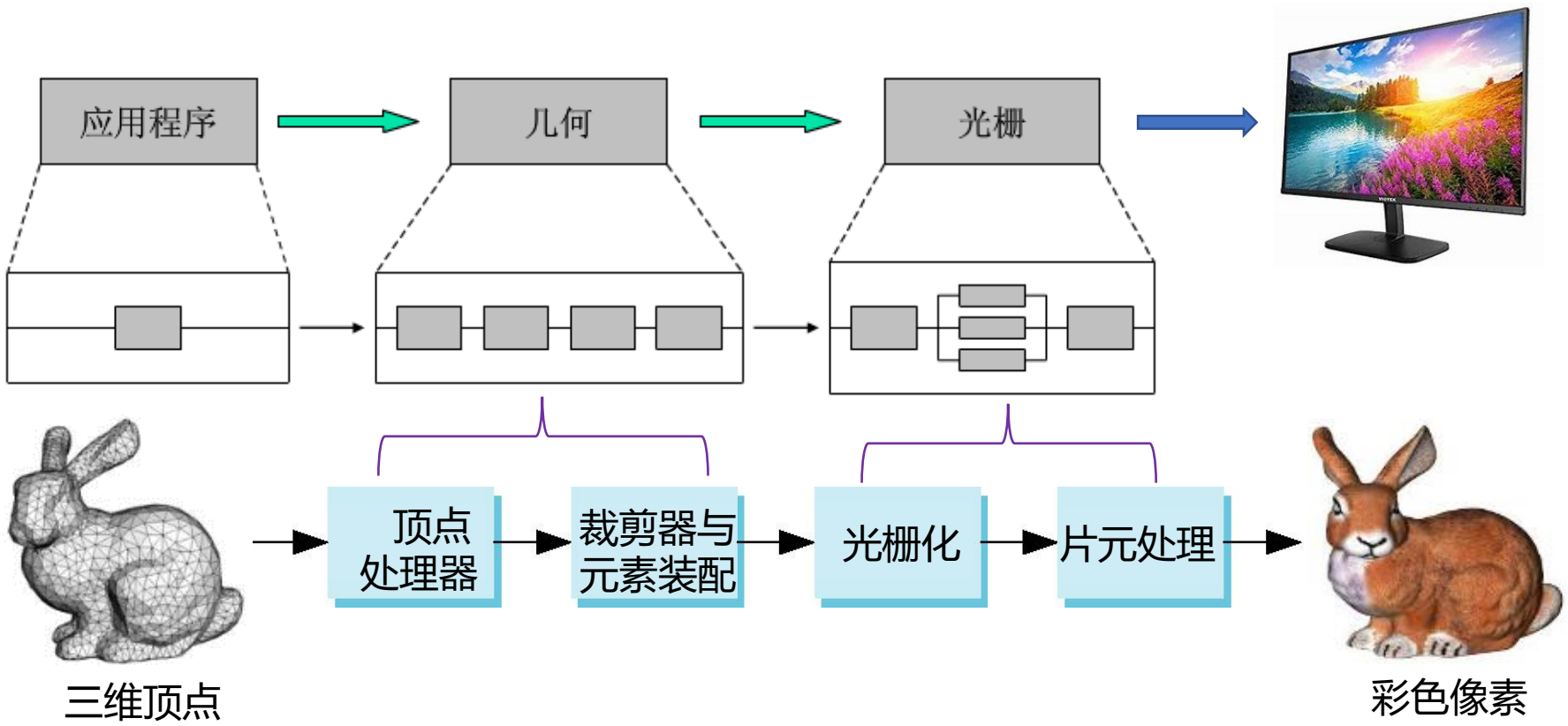
- 几何阶段
  - 将3D模型投影变换到图像平面
  - 决定可见的图元（图形元素）
- 光栅阶段
  - 决定可见的片元 (fragment)
  - 决定片元的颜色成为彩色像素

基本图元

- 2D
  - 点、线
- 3D
  - 点、线和三角形

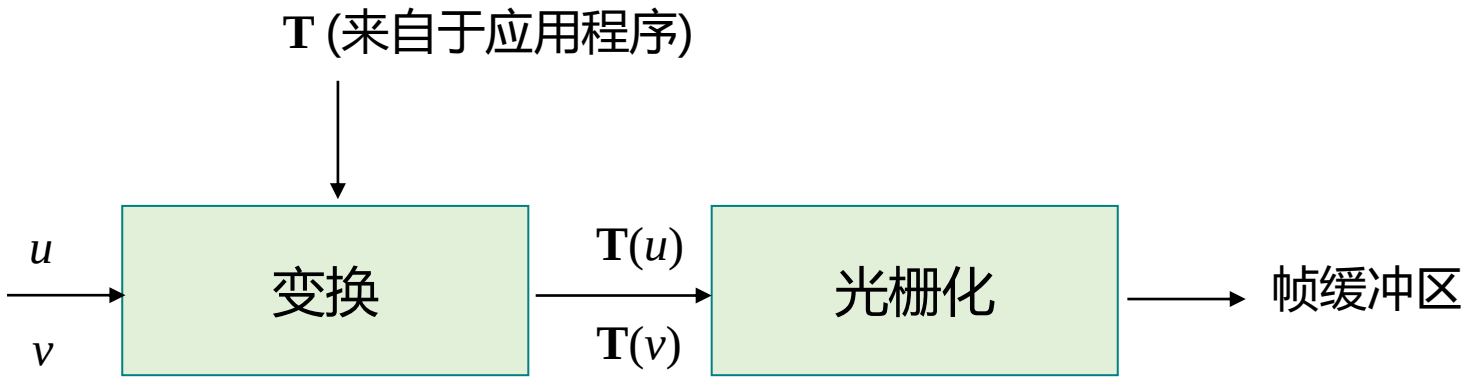


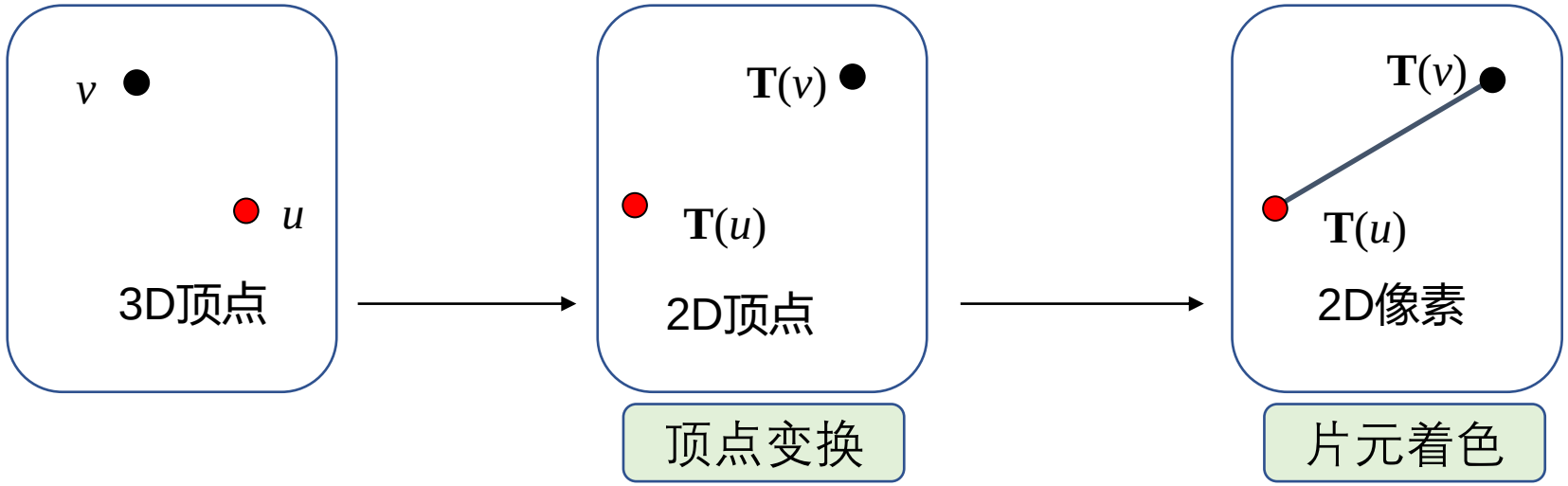
渲染管线：渲染计算的流水线



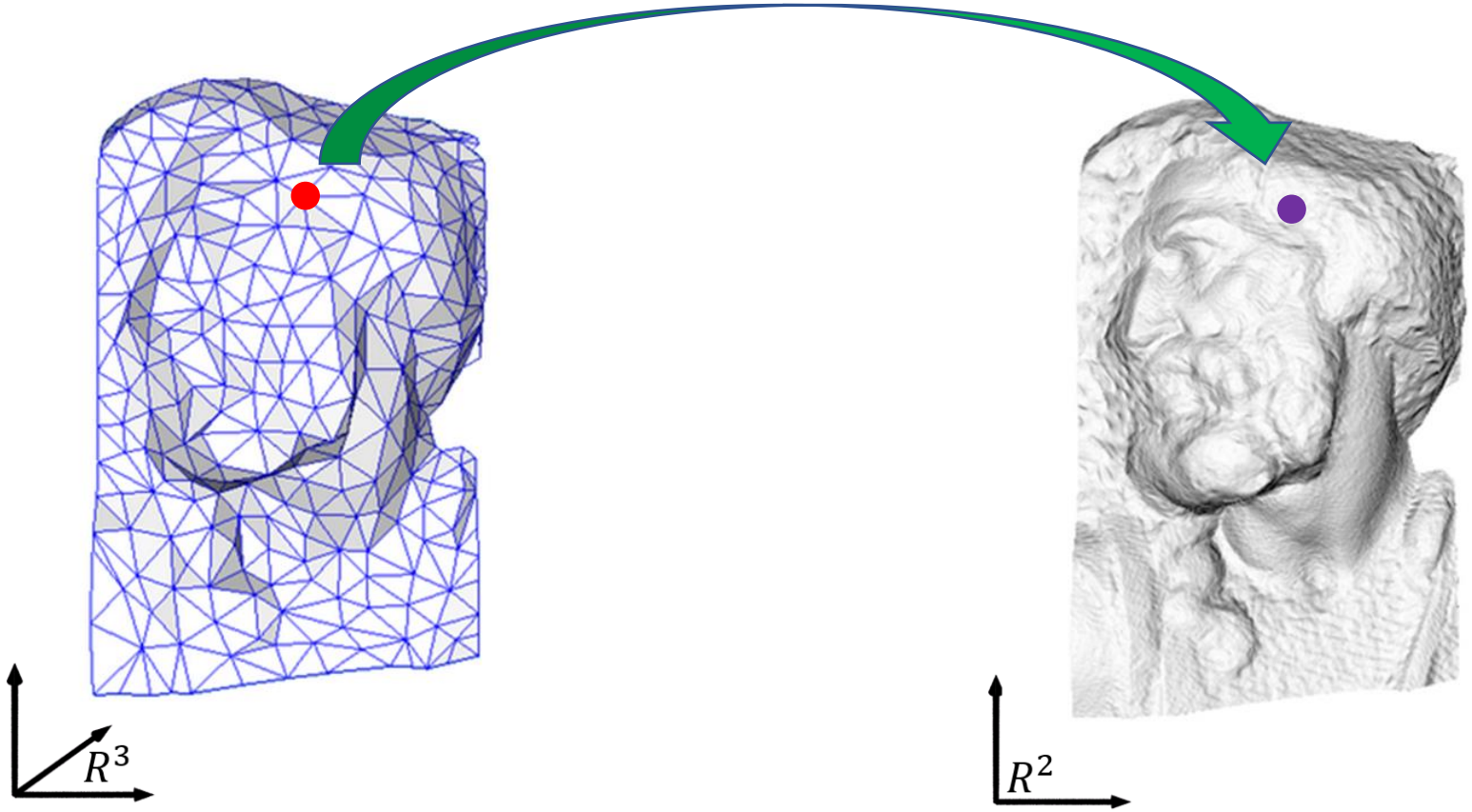
渲染流水线的过程

两个主要过程：几何+像素


T (来自于应用程序)
u
v
变换
T(u)
T(v)
光栅化
帧缓冲区


v
u
3D顶点
T(v)
T(u)
2D顶点
T(v)
T(u)
2D像素
顶点变换
片元着色

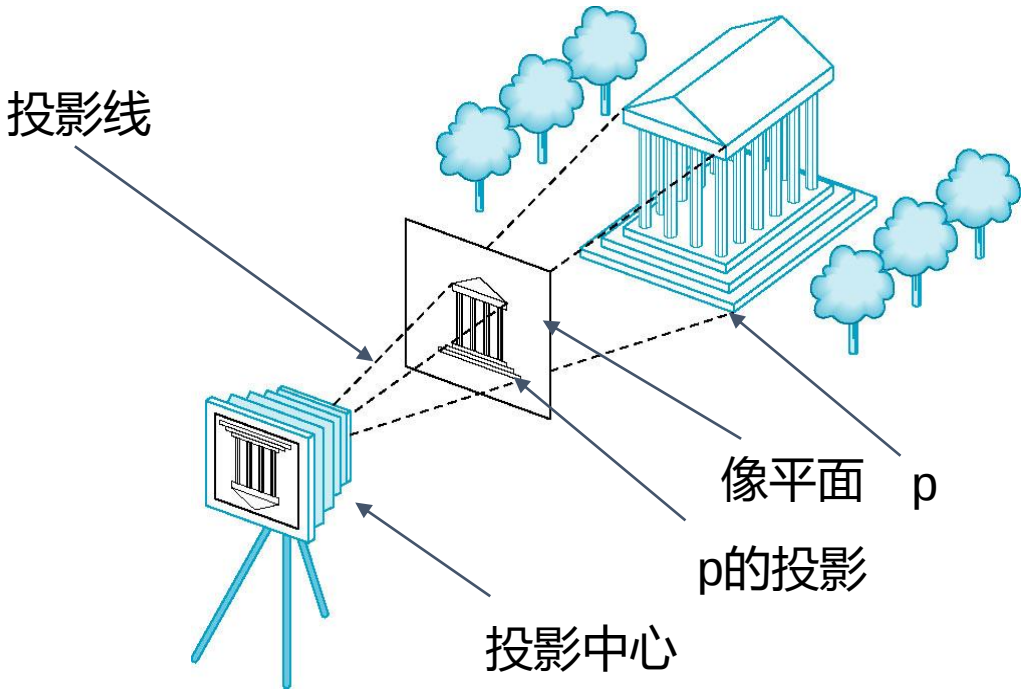
过程1：几何变换
逐顶点计算屏幕坐标

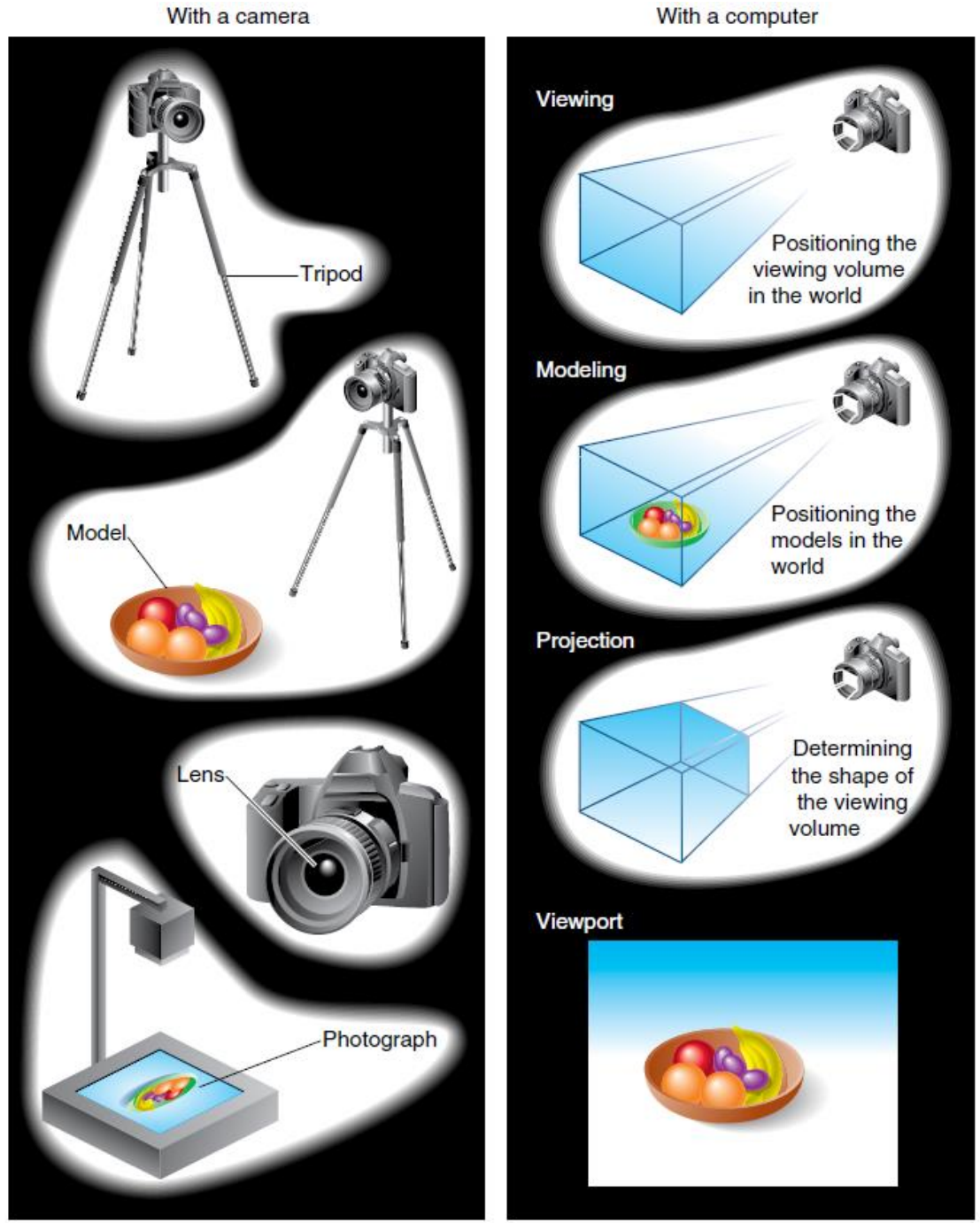


成像模型

- 3D场景在虚拟相机（眼睛）下的投影

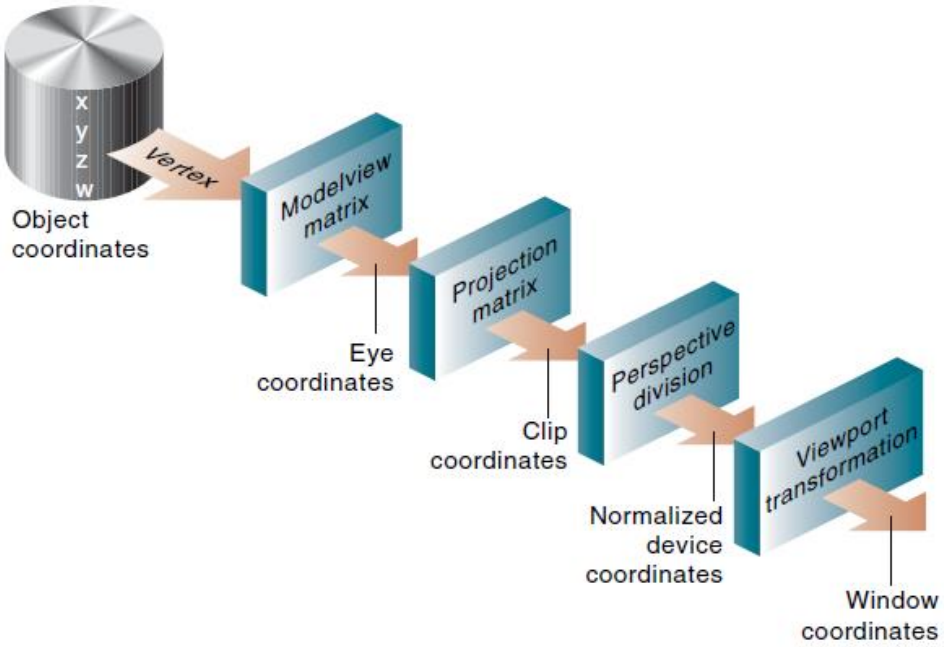
- 应用程序指定
  - 3D模型
  - 相机参数
    - 位置
    - 朝向
    - 焦距
    - 视角




With a camera
With a computer
Tripod
Model
Lens
Photograph
Viewing
Positioning the viewing volume in the world
Modeling
Positioning the models in the world
Projection
Determining the shape of the viewing volume
Viewport

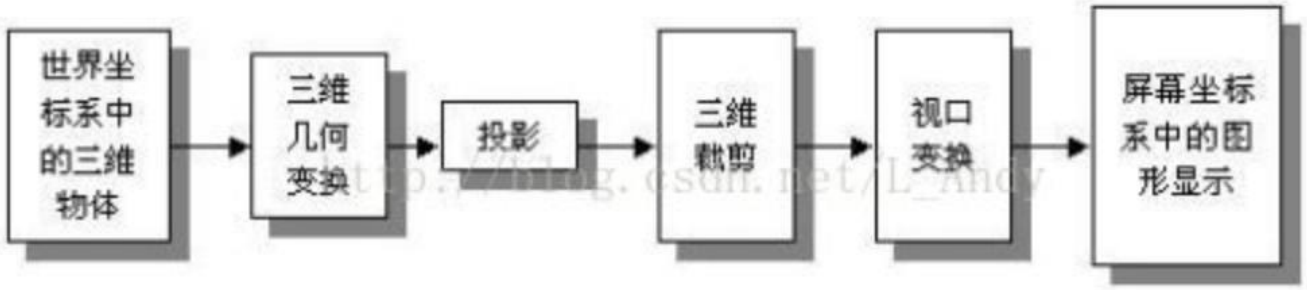
各种坐标系

- 对象坐标系或建模坐标系
- 世界坐标系
- 视点坐标系或照相机坐标系
- 裁剪坐标系
- 规范化的设备坐标系
- 窗口坐标系或屏幕坐标系

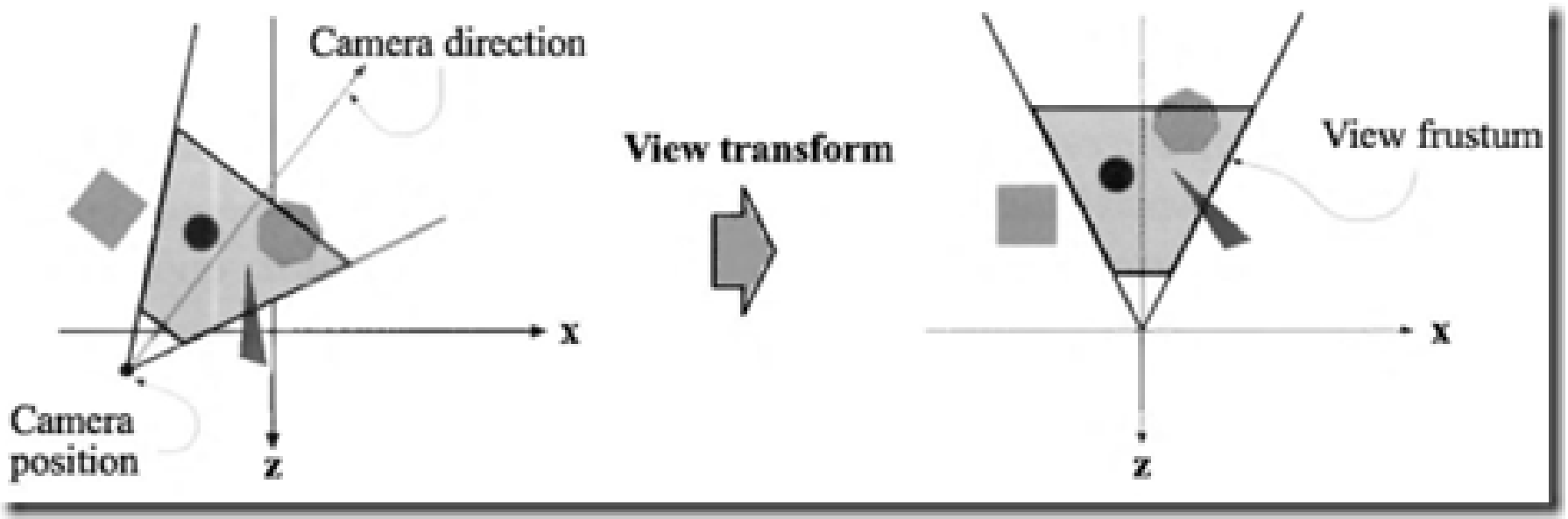


顶点处理

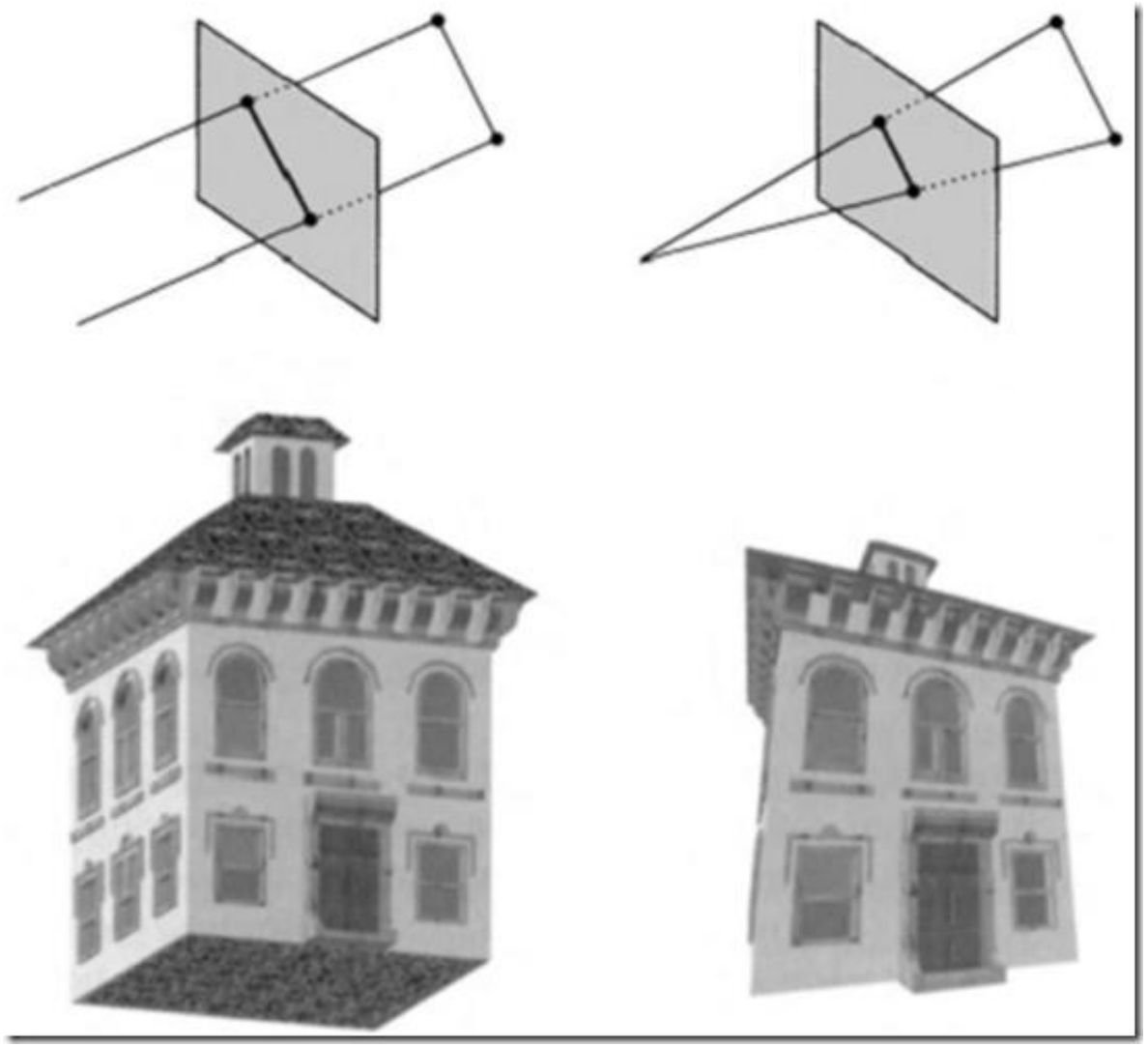
- 流水线中大部分工作是把对象在一个坐标系中表示转化为另一坐标系中的表示:
  - 世界坐标系
  - 照相机(眼睛)坐标系
  - 屏幕坐标系
- 坐标的每个变换相当于一次矩阵乘法
  - 最终的顶点变换为多个矩阵的乘法：MVP
  - 窗口变换



模型及视图变换

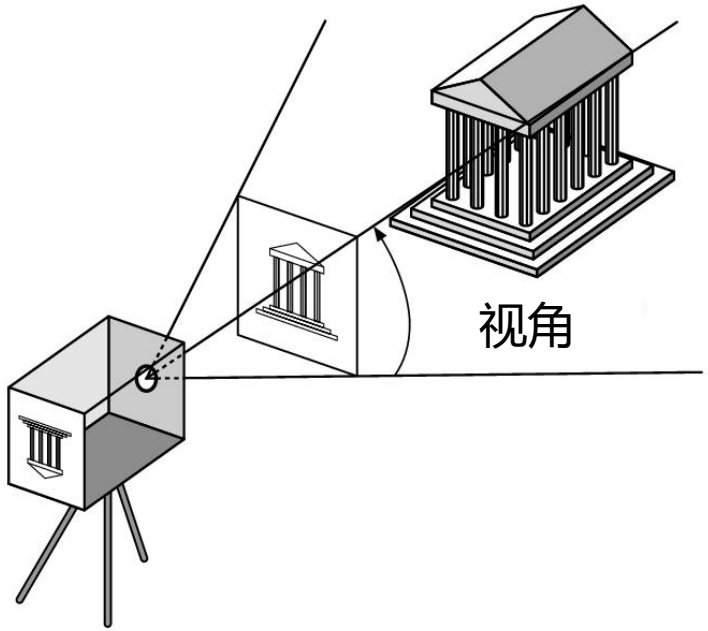

Camera direction
View transform
View frustum
x
z
Camera position
x
z

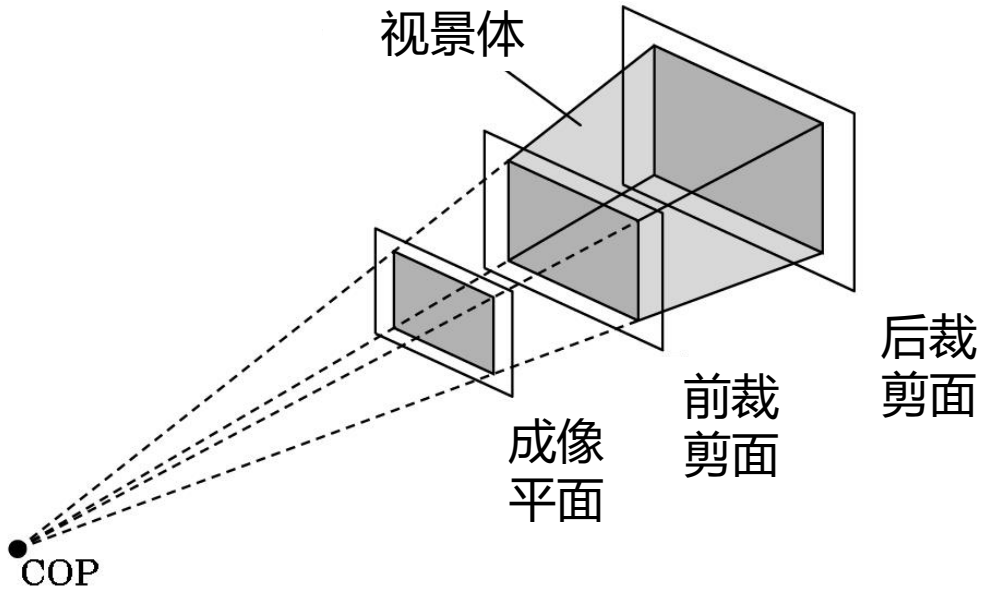
投影变换



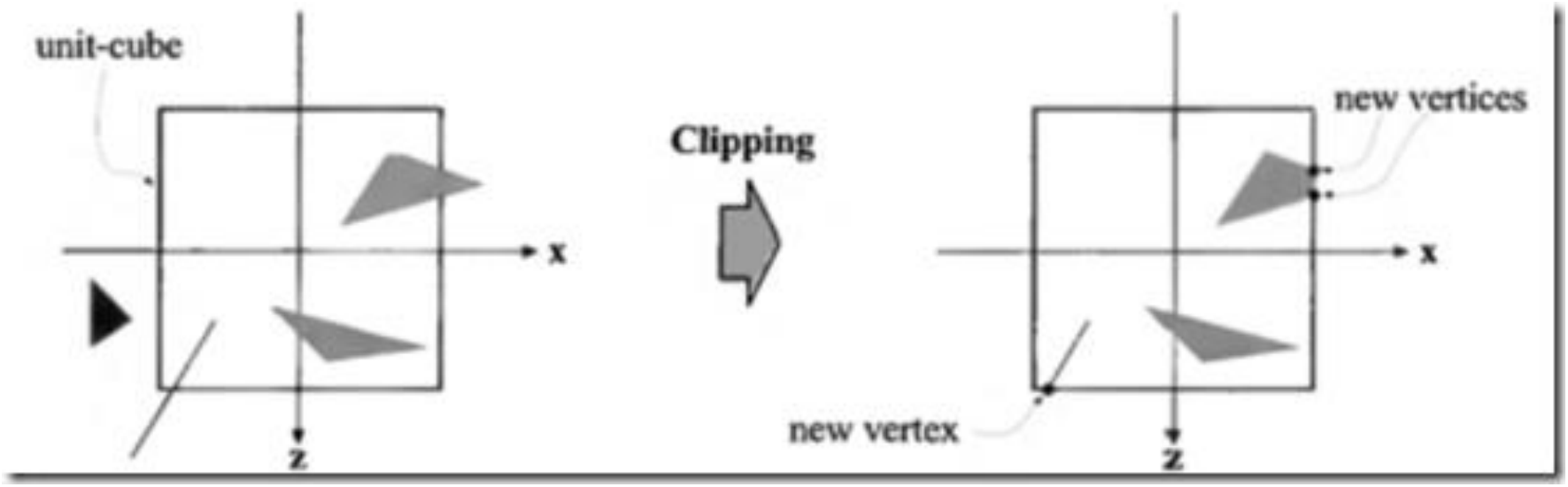
裁剪：视景体裁剪

- 超出可视范围的几何元素需要裁剪掉，不参加后面的计算
  - 视景体范围

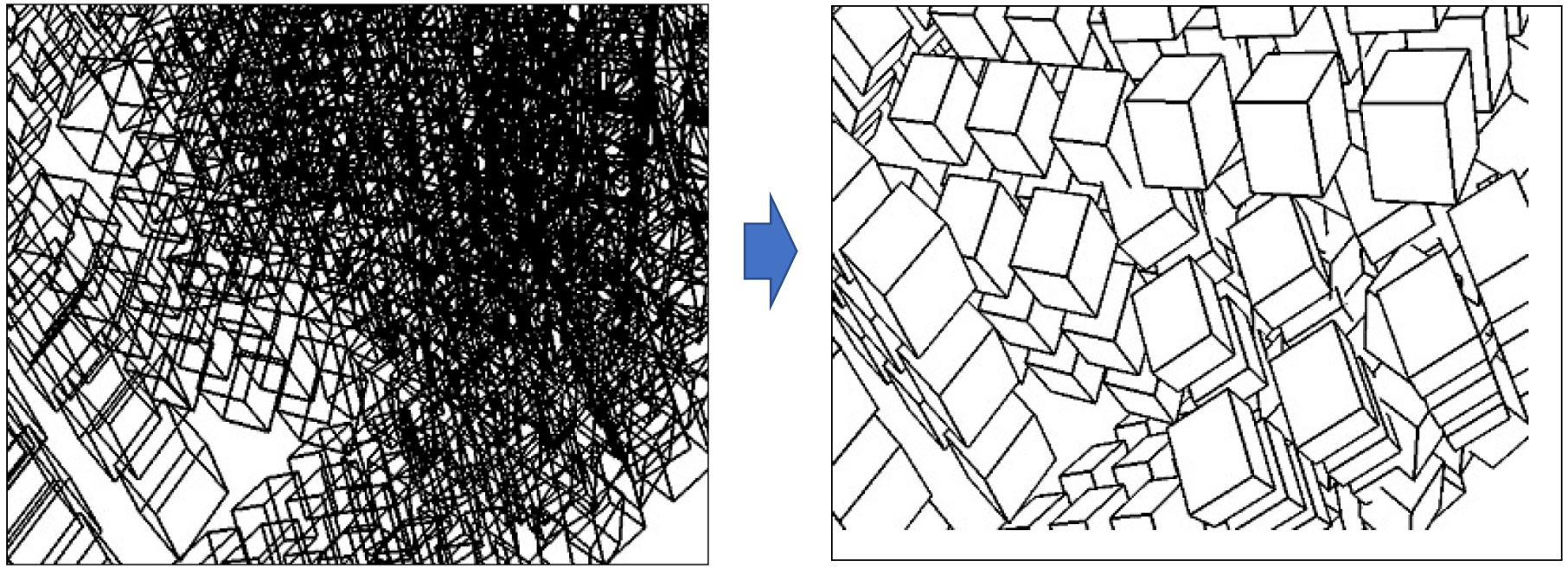




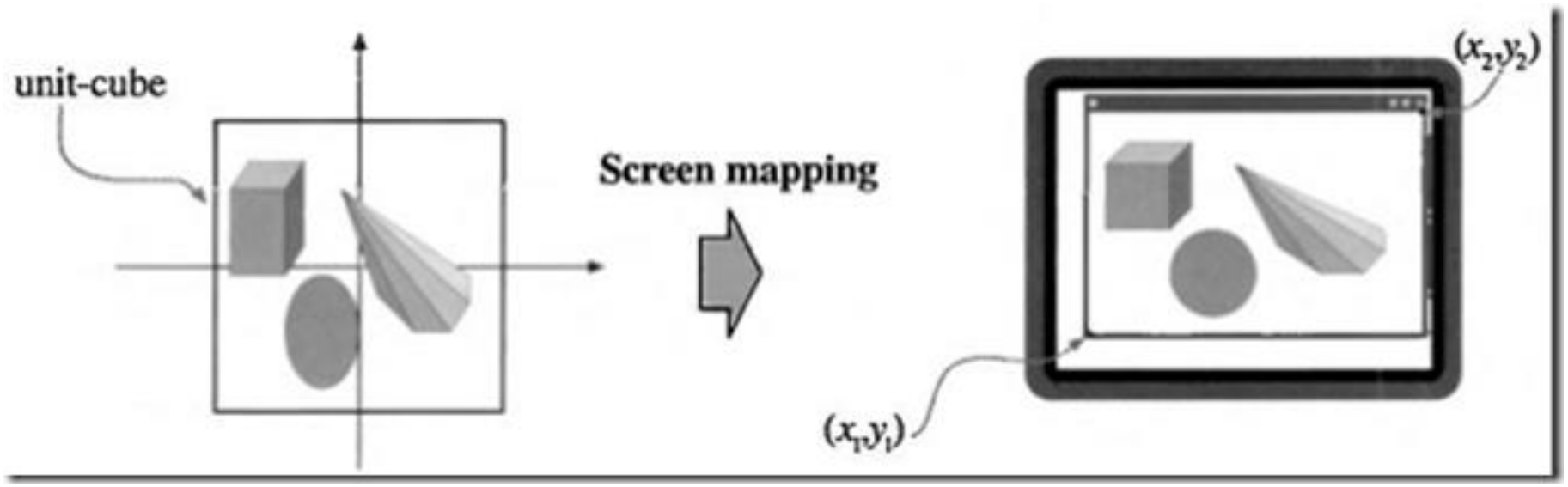
裁剪：窗口裁剪



消隐：消除隐藏面



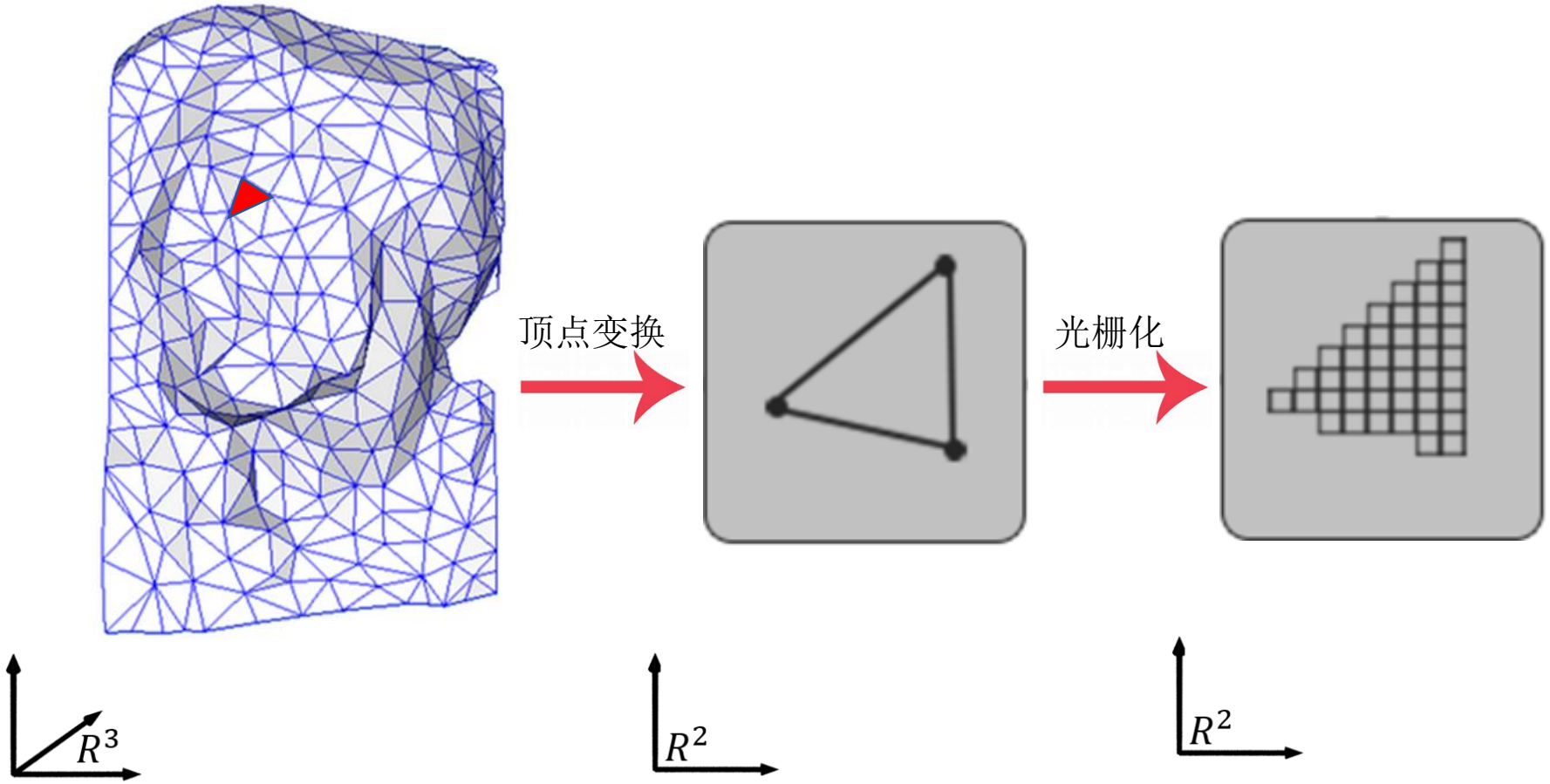
屏幕映射变换



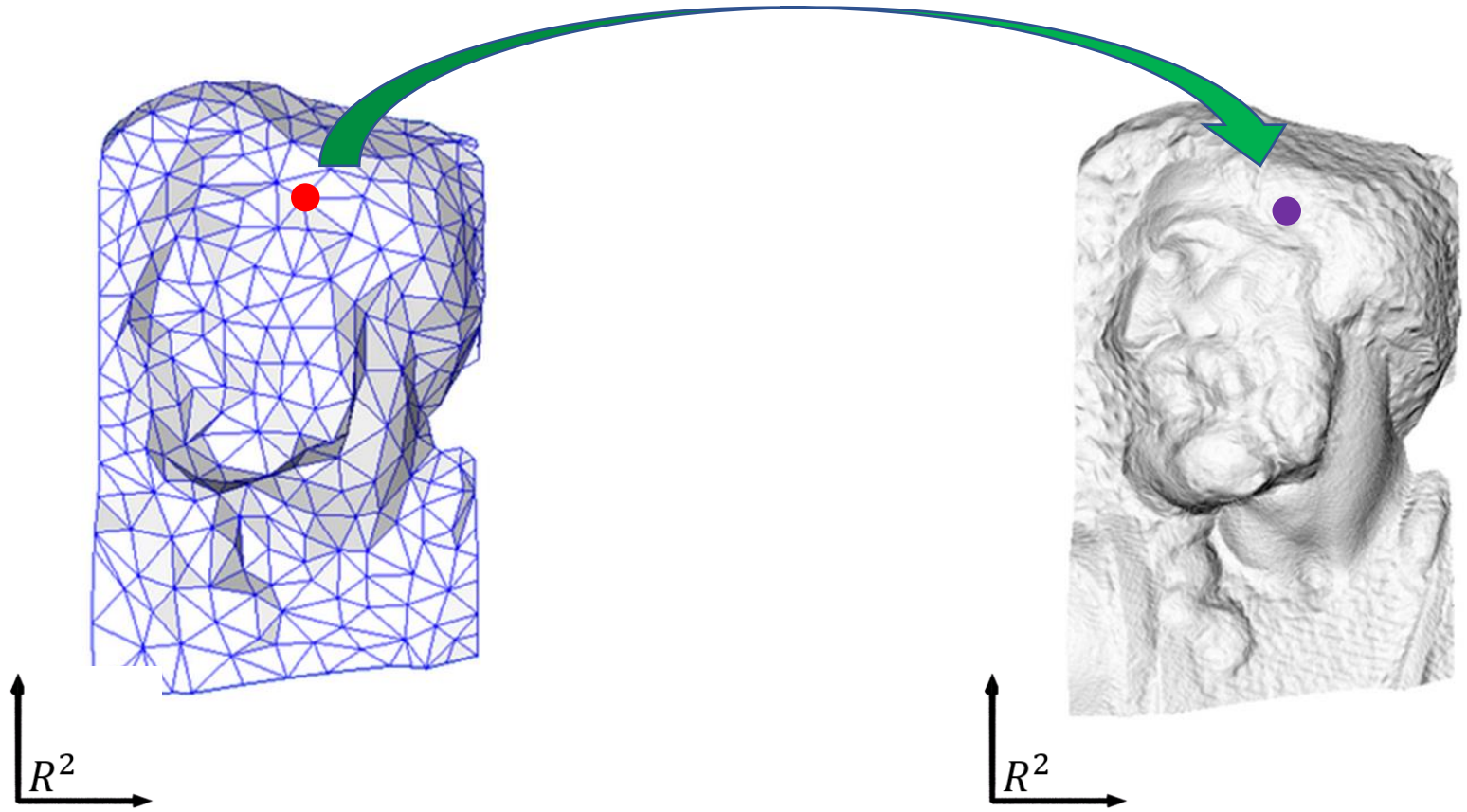
光栅化：找到所有片元(fragment)

- 如果一个对象不被裁掉，那么在帧缓冲区中相应的像素就必须被赋予颜色
- 光栅化程序为每个对象生成一组片段
- 片元是“潜在的像素”（未着色的像素）
  - 在帧缓冲区中有一个位置
  - 具有若干属性（如颜色、深度等）
- 光栅化程序在对象上对顶点属性进行插值（重心坐标）

过程1：几何变换
逐顶点计算屏幕坐标



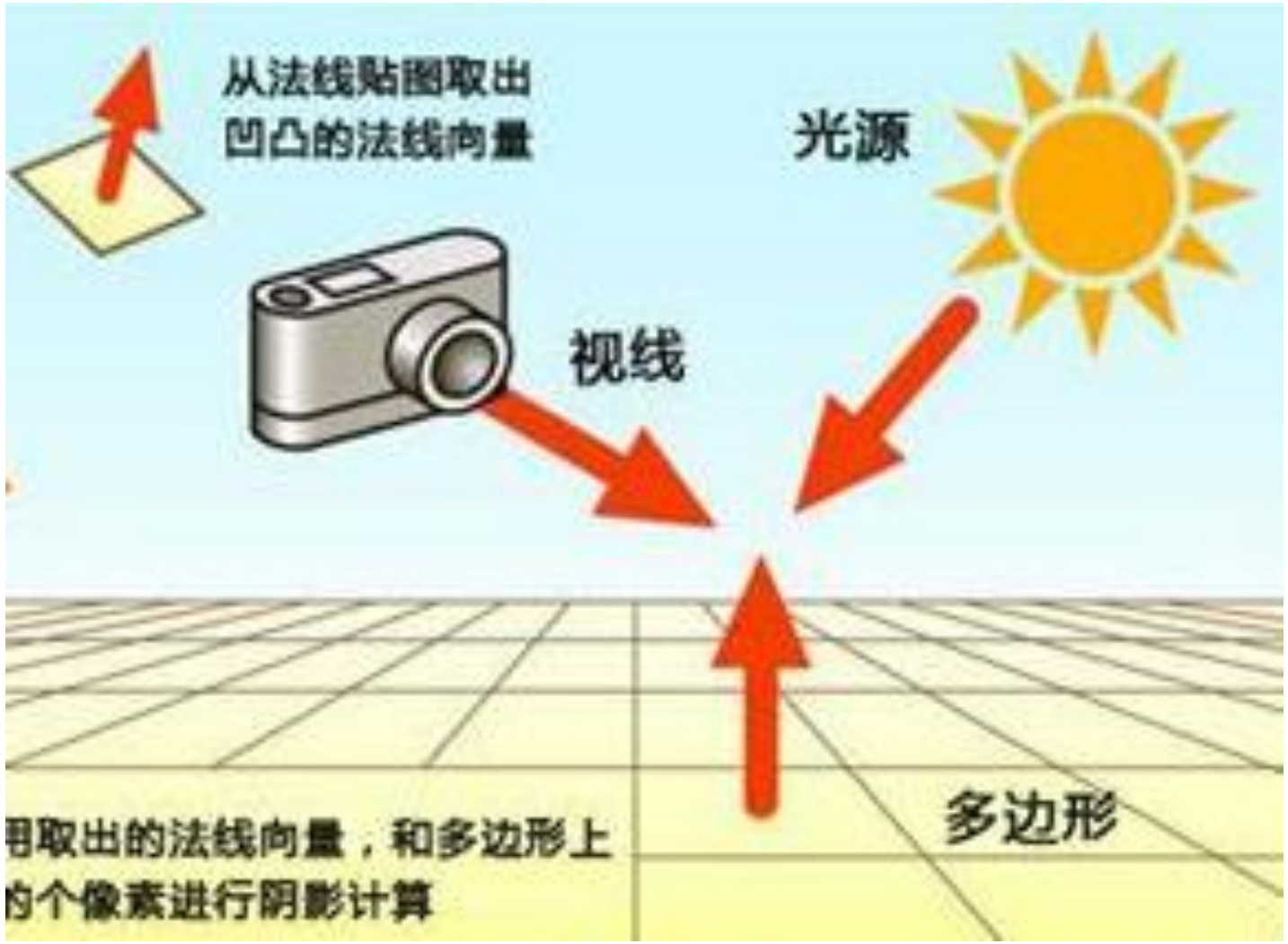
过程2：像素着色
逐片元计算颜色



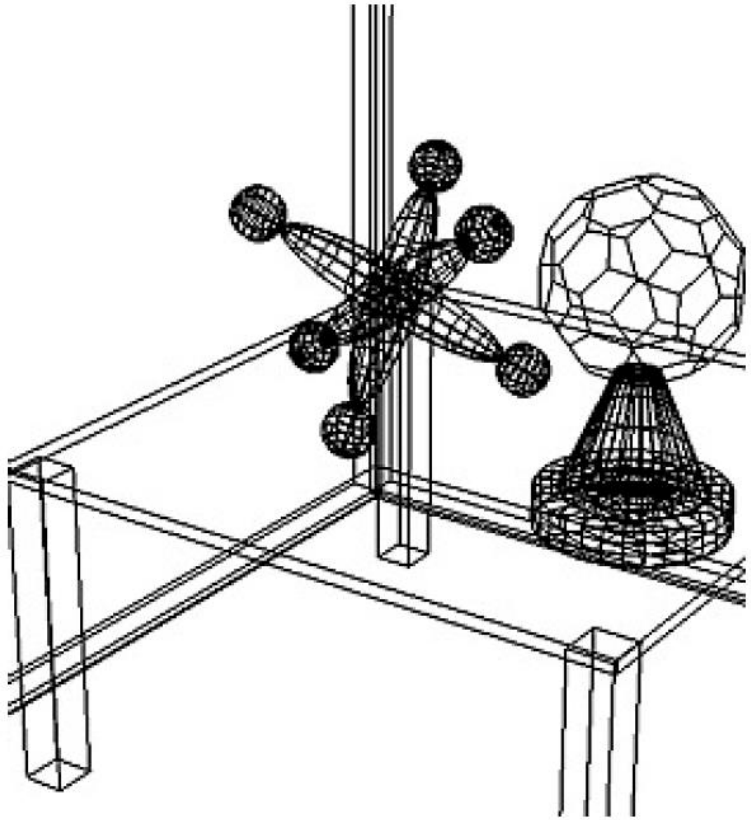
着色：让3D物体更有空间感

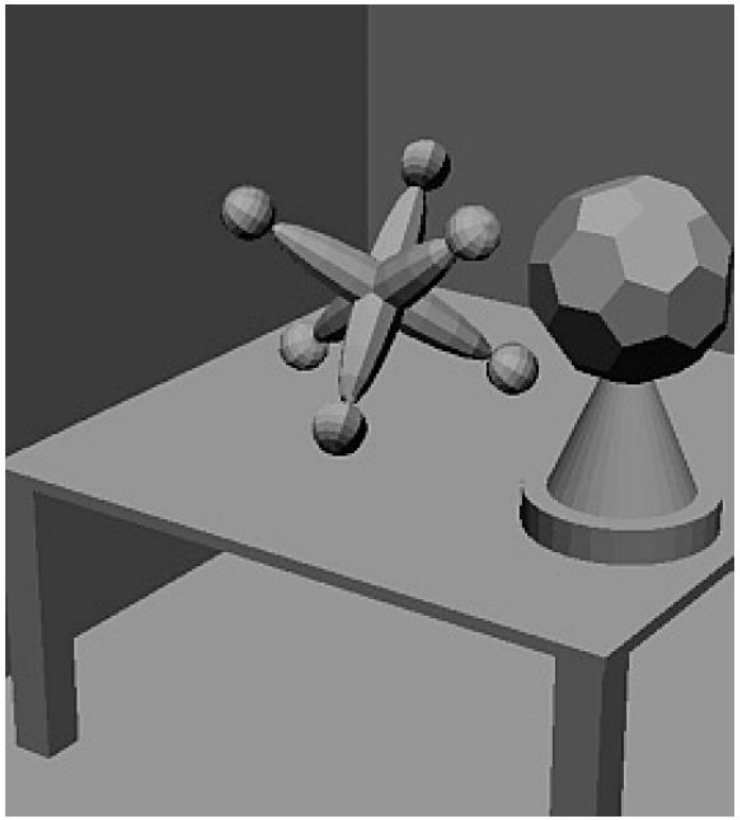
- 光：产生物体的不同部分的明暗变化
- 需模拟光对顶点的明暗（颜色）计算

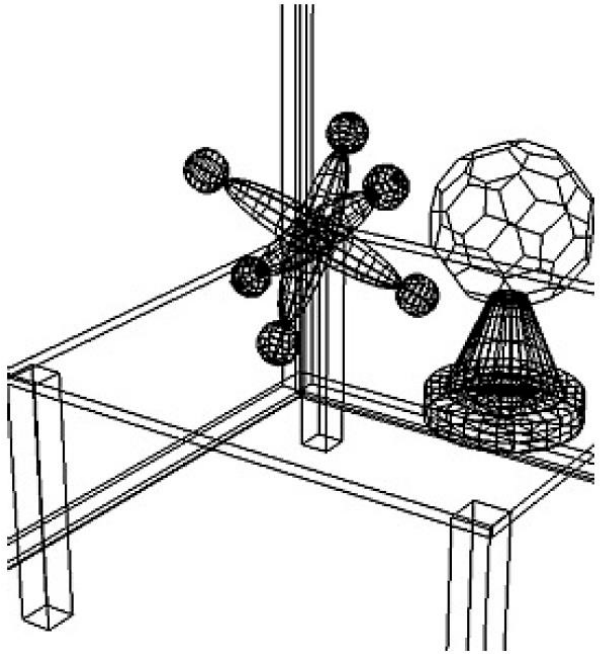
着色：光的物理性质及计算

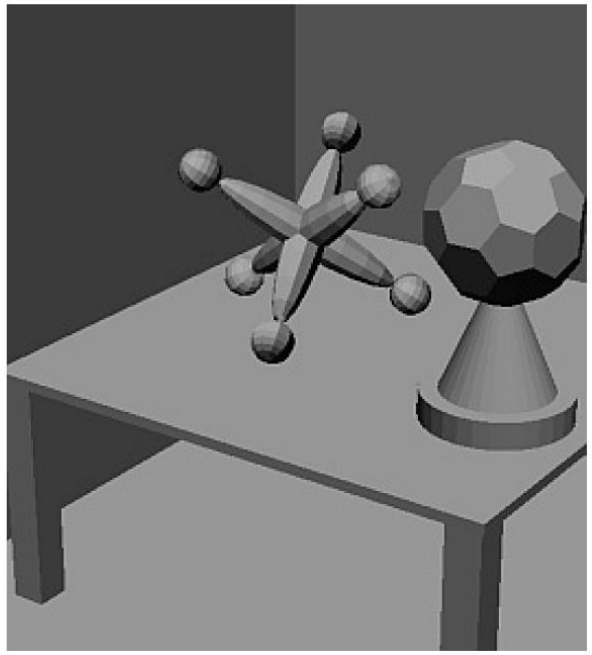


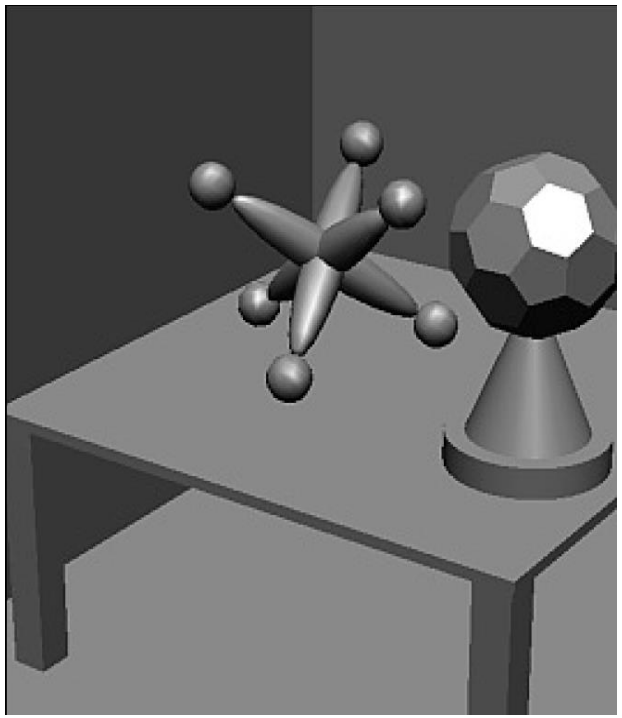
片元着色

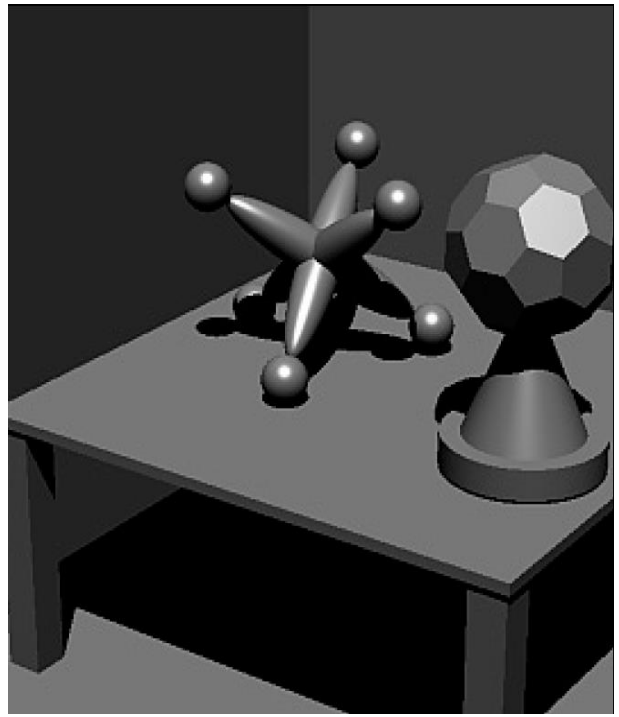


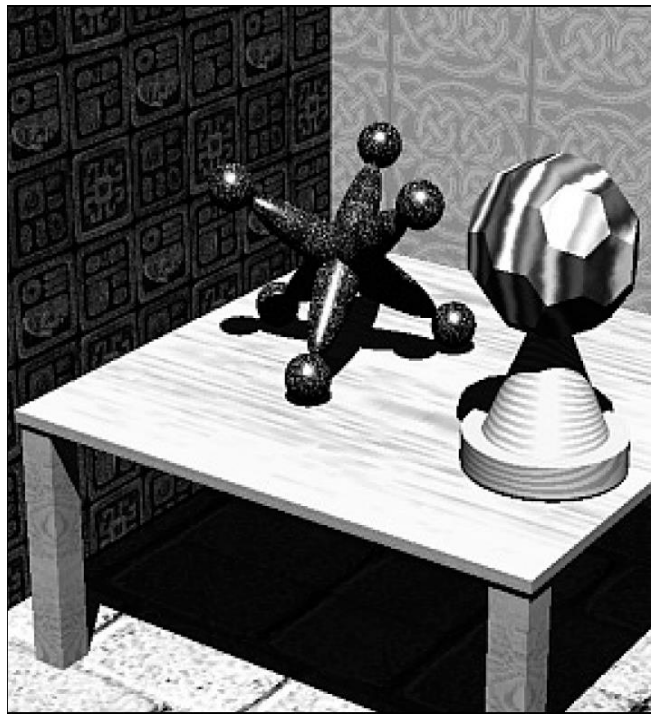






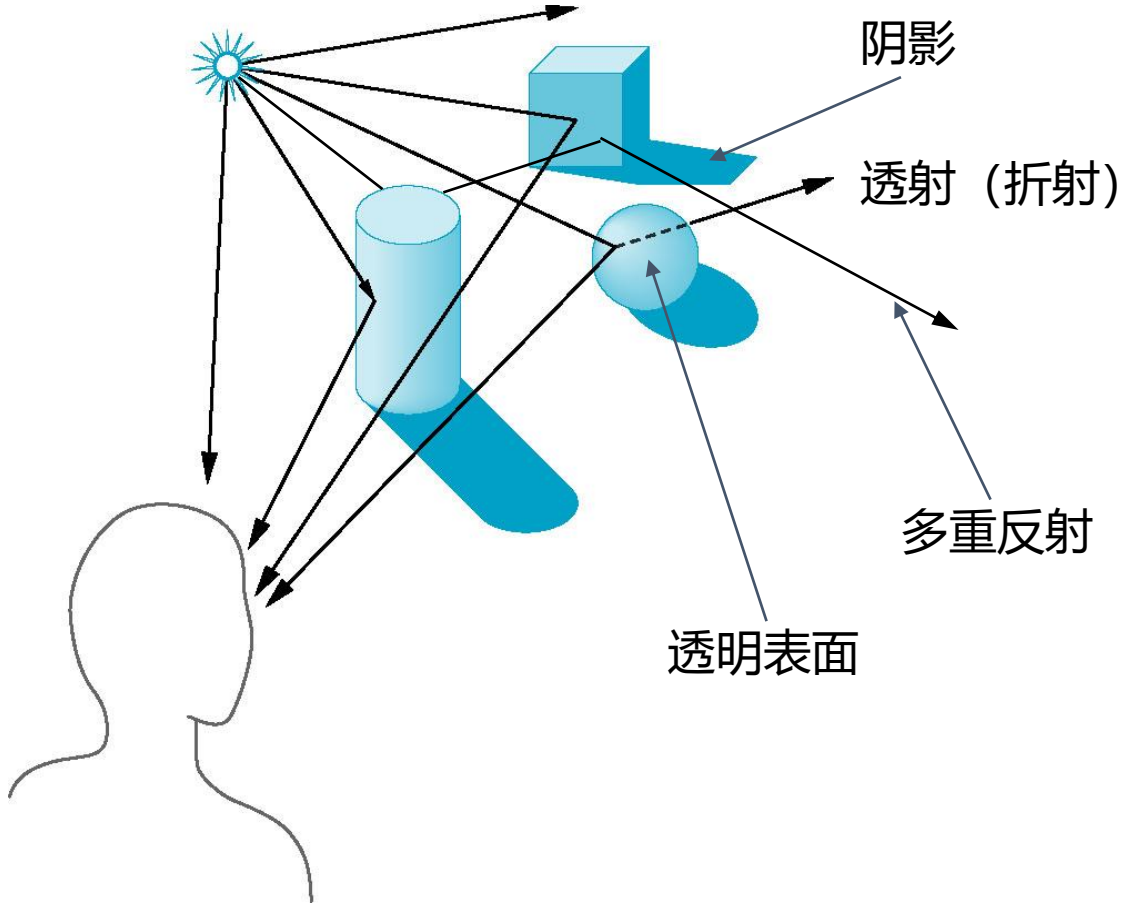






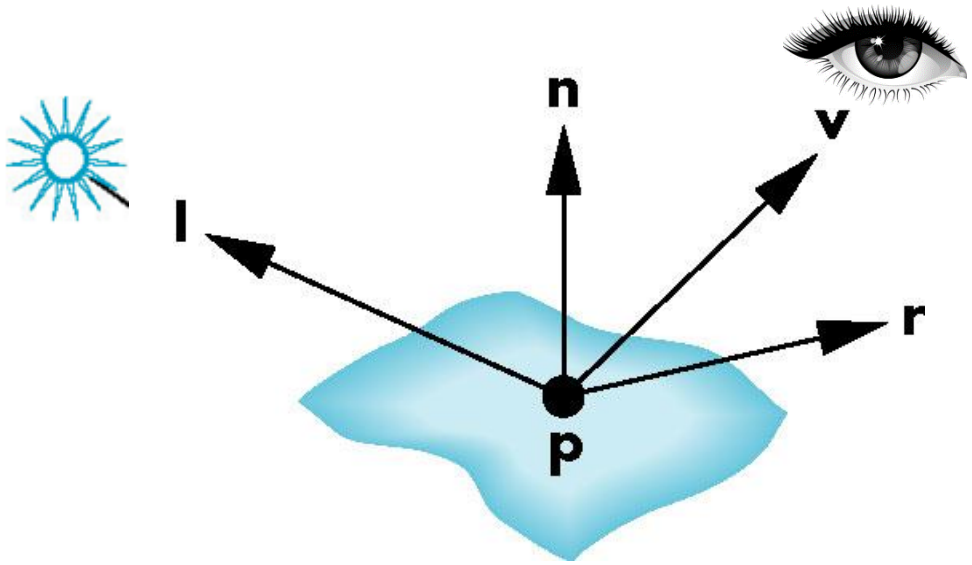
光照模型

- 局部光照模型
  - 简单、经验性
- 全局光照模型
  - 复杂、物理的



局部光照模型

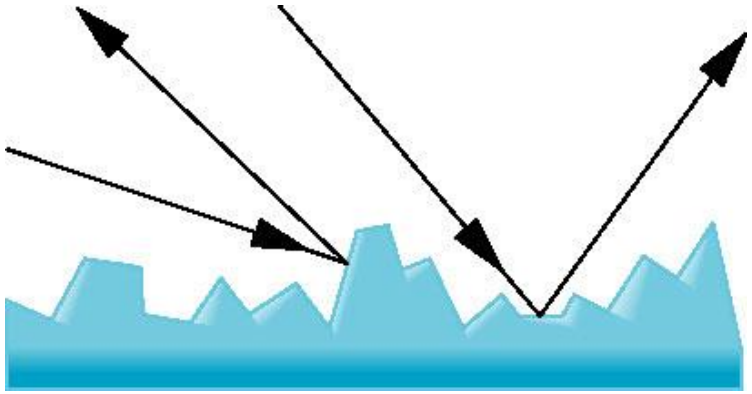
- 3个主要部分
  - 环境光 (Ambient)
  - 漫反射 (Diffuse)
  - 镜面反射 (Specular)



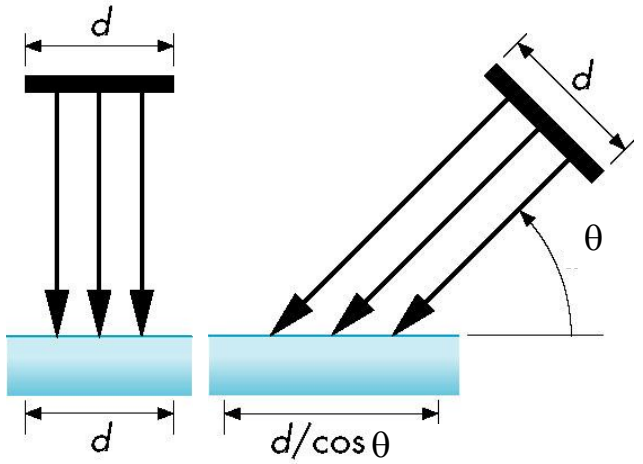
- $I = k_a I_a + k_d I_d (I \cdot n) + k_s I_s \, (v \cdot r)^{\alpha}$

漫反射 (Lambertian项)

- 模拟粗糙表面：光向各个方向均匀地反射
- 反射光的比例正比于入射光的竖直分量
  - 即反射光 ~ $\cos\theta_I$
  - $I_d = k_d I_l \cos\theta_i$



Lambertian曲面

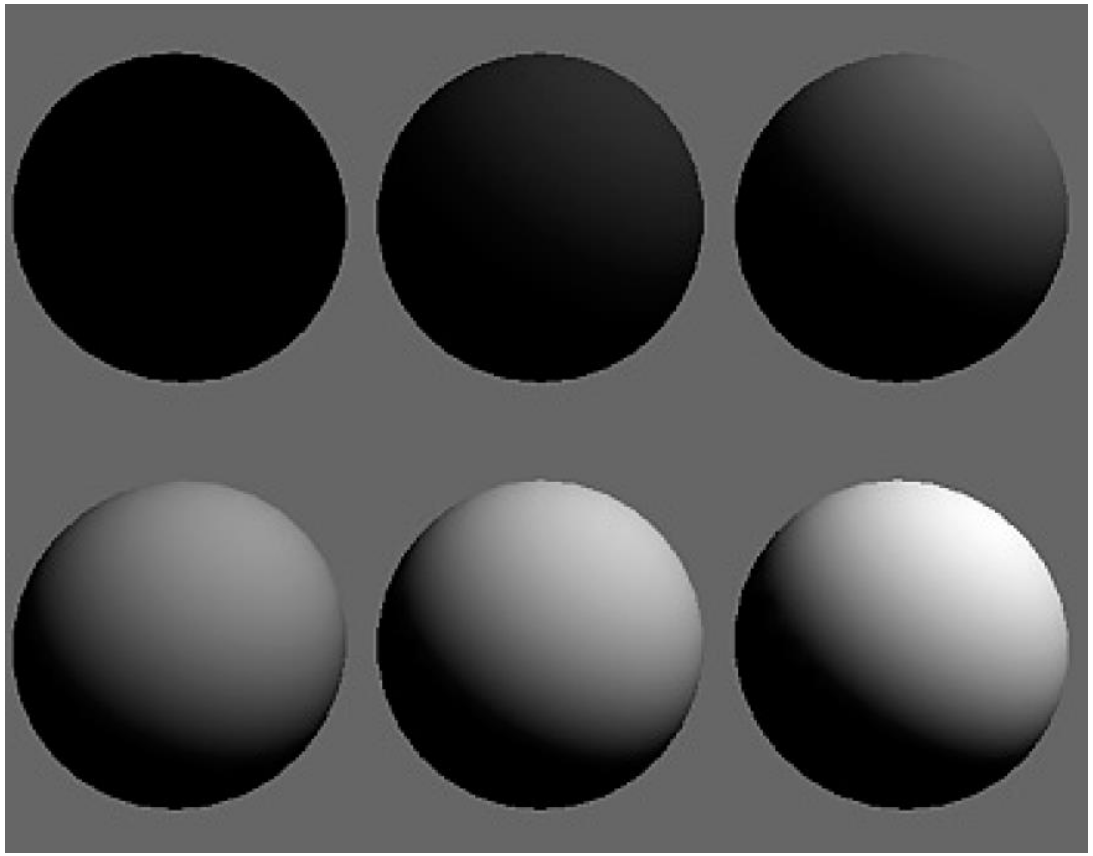


(a) (b)

漫反射参数的影响

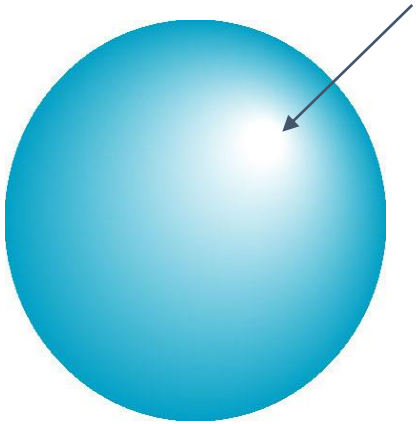
漫反射系数依次为0.0, 0.2, 0.4, 0.6, 0.8, 1.0

光强为1.0, 背景光强为0.4



镜面反射计算模型

- 模拟在镜面反射方向附近的聚集光现象



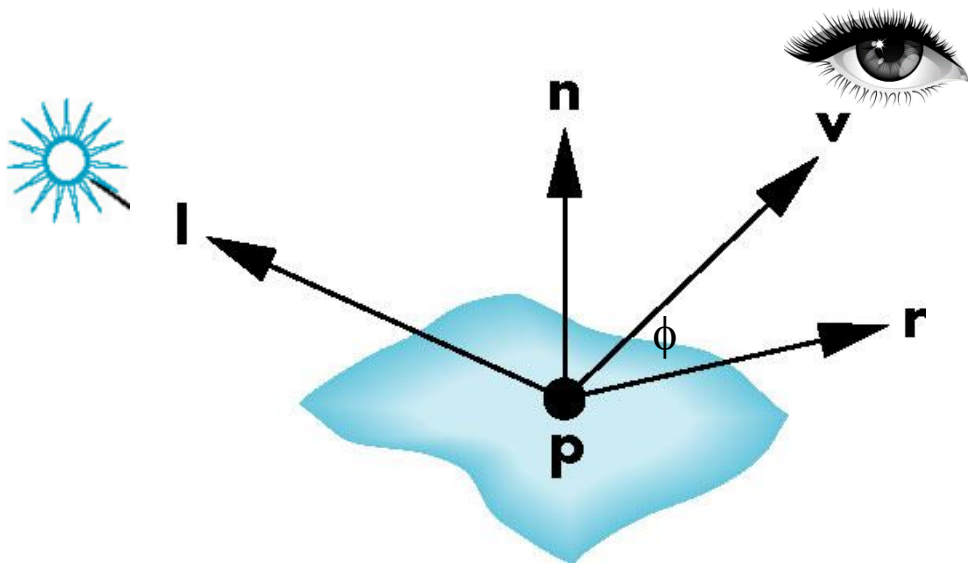
n: 法向量

l: 入射光方向

r: 反射方向

v: 视点方向

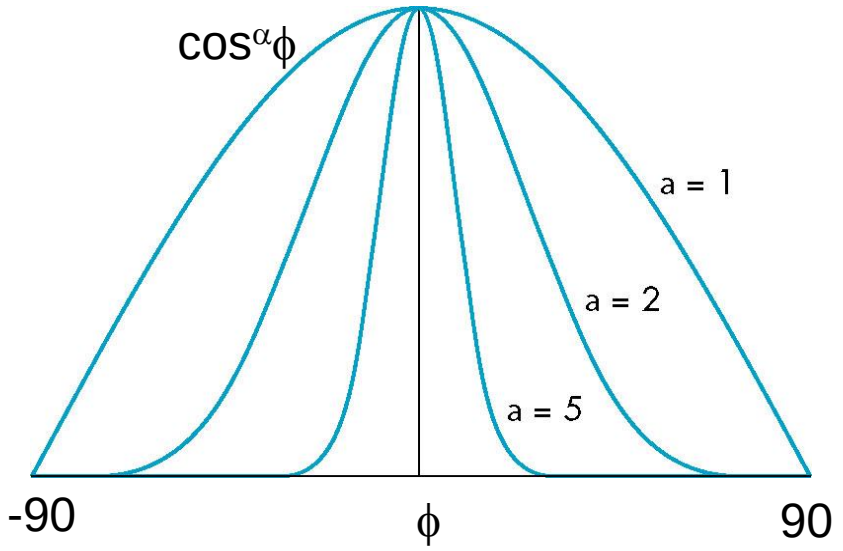
ϕ: r与v的夹角



$$I_r = k_s I \cos^{\alpha}\phi$$

高光系数

- $\alpha \in [100,200]$，对应于金属材料
- $\alpha \in [5,10]$，材料类似于塑料

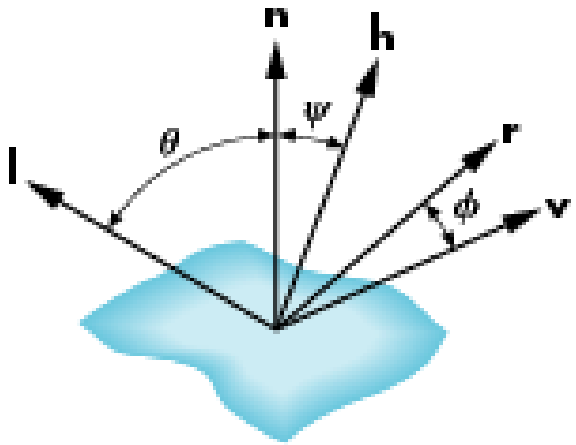


镜面反射的Blinn-Phong修正计算

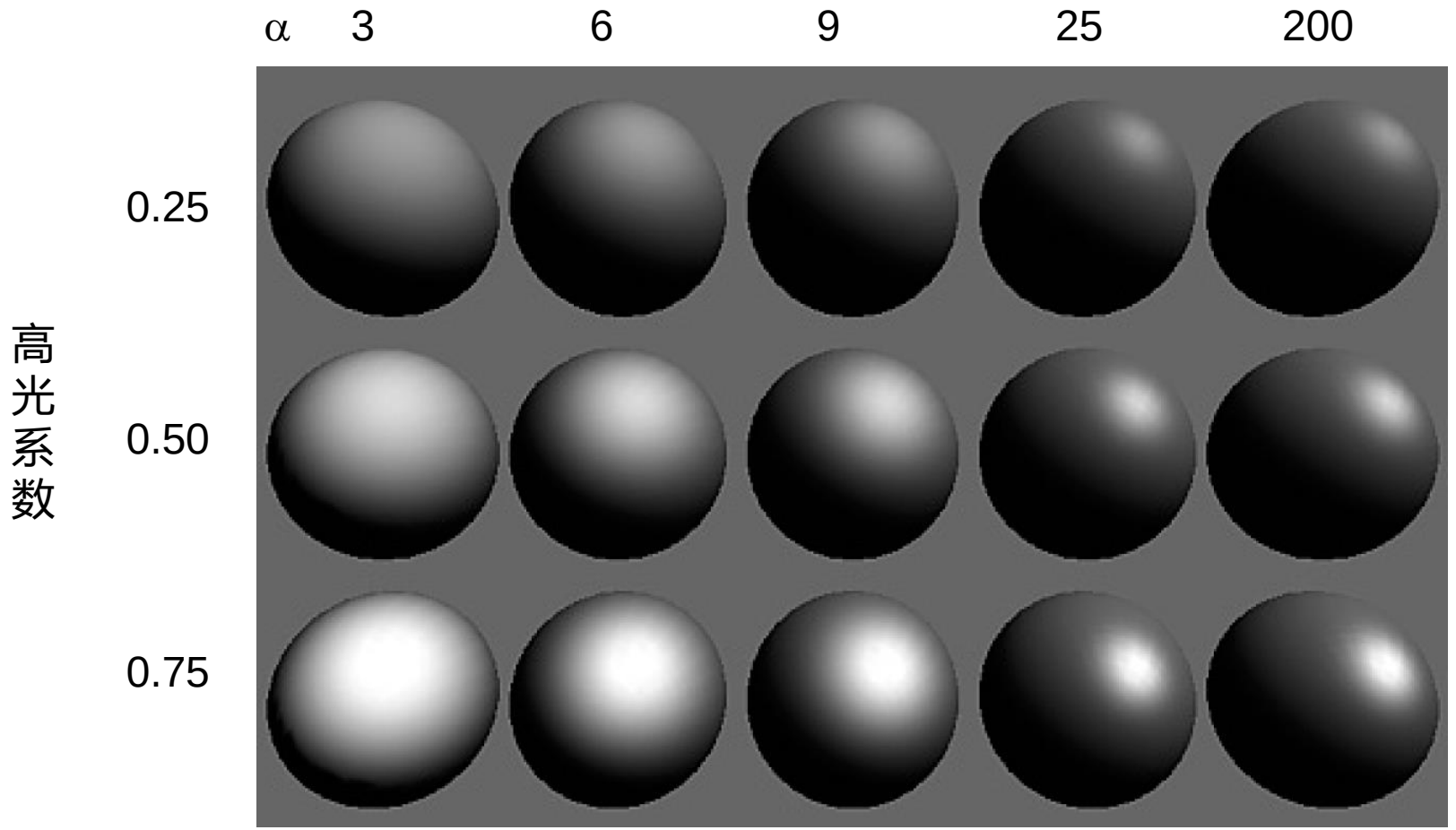
- Blinn利用中分向量给出了一个近似算法，减少计算量、提高计算效率
- h是I和v的平分单位向量，即

$$h = \frac{I+v}{|I+v|}$$

- 用$(n \cdot h)^{\beta}$代替$(v \cdot r)^{\alpha}$
  - 参数β恰当选取，以匹配高光度



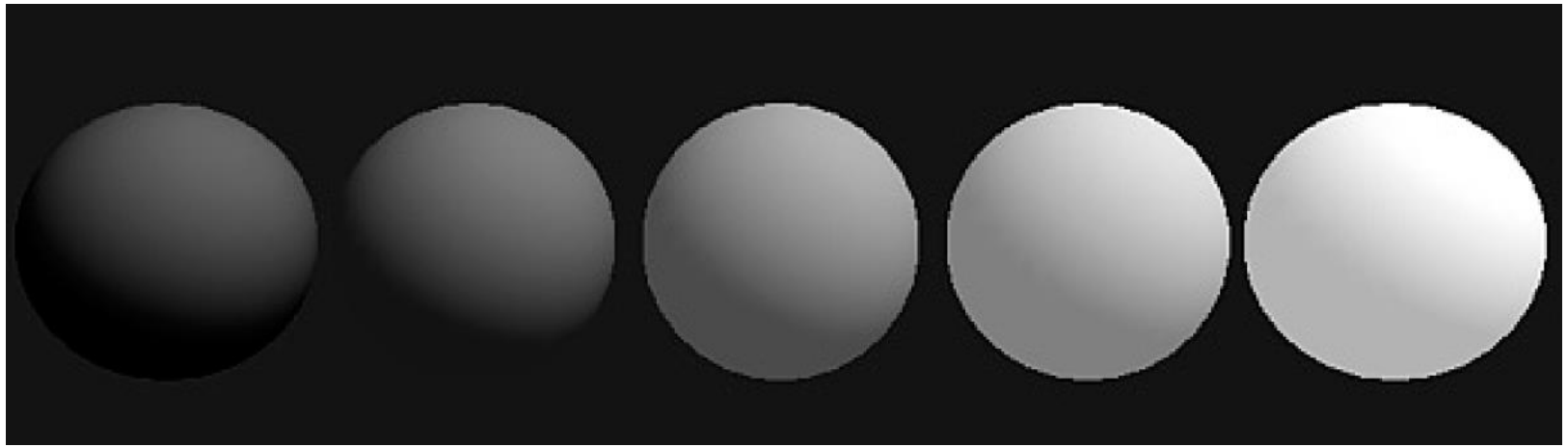
镜面反射参数的影响



环境光

- 背景光：模拟多次反射后的效果的近似
  - 让场景光照不到的地方看起来不是全黑
  - 每个地方都具有相同的强度
- 环境光强 $I_a = [I_{ar}, I_{ag}, I_{ab}]$为常值

环境光参数的影响



向漫反射光中加入不同的环境光的效果

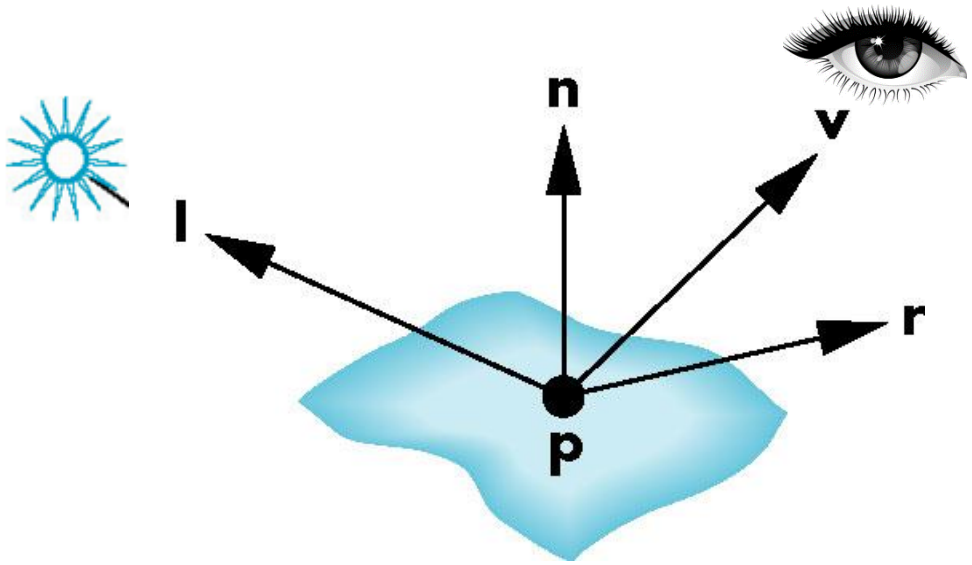
两种光强都是1, 漫反射系数为0.04

环境光反射系数依次为0, 0.1, 0.3, 0.5, 0.7

Recap: 局部光照模型

- 3个主要部分
  - 环境光 (Ambient)
  - 漫反射 (Diffuse)
  - 镜面反射 (Specular)

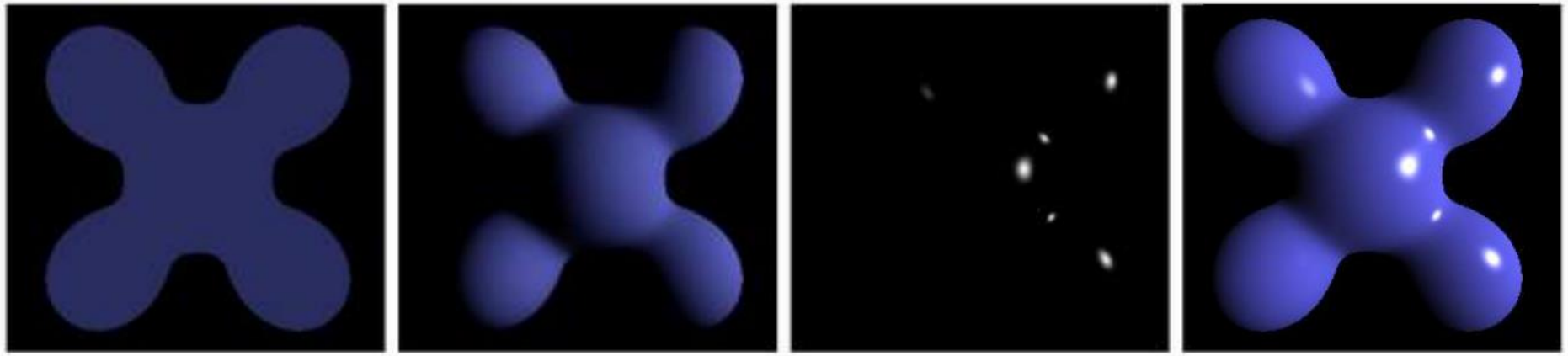
- $I = k_a I_a + k_d I_d (I \cdot n) + k_s I_s (v \cdot r)^{\alpha}$



光源与材质属性

- 三原色中每种分量单独处理
  - 九个系数 k_{dr}, k_{dg}, k_{db}, k_{sr}, k_{sg}, k_{sb}, k_{ar}, k_{ag}, k_{ab}
- 材质
  - 高光系数α
- 多个光源
  - 每个光源的结果叠加在一起

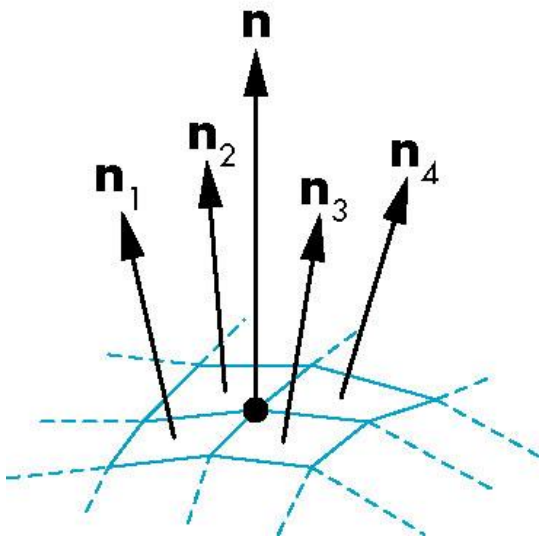
着色结果



Ambient + Diffuse + Specular = Blinn-Phong Shading

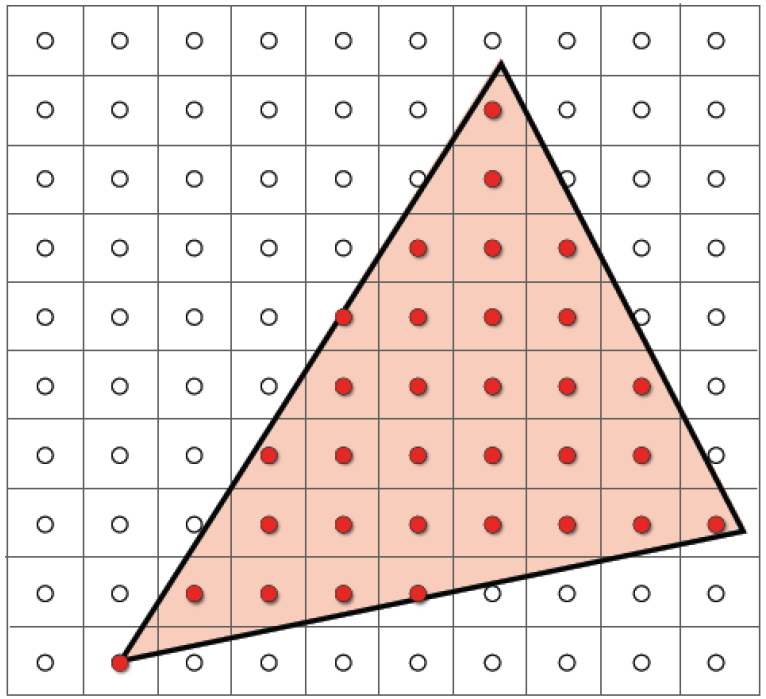
顶点的着色

- 顶点的法向
  - 由原始数据给出
  - 由相邻面的法向（加权）平均得到
  - 其他估计方法



像素（片元）的着色

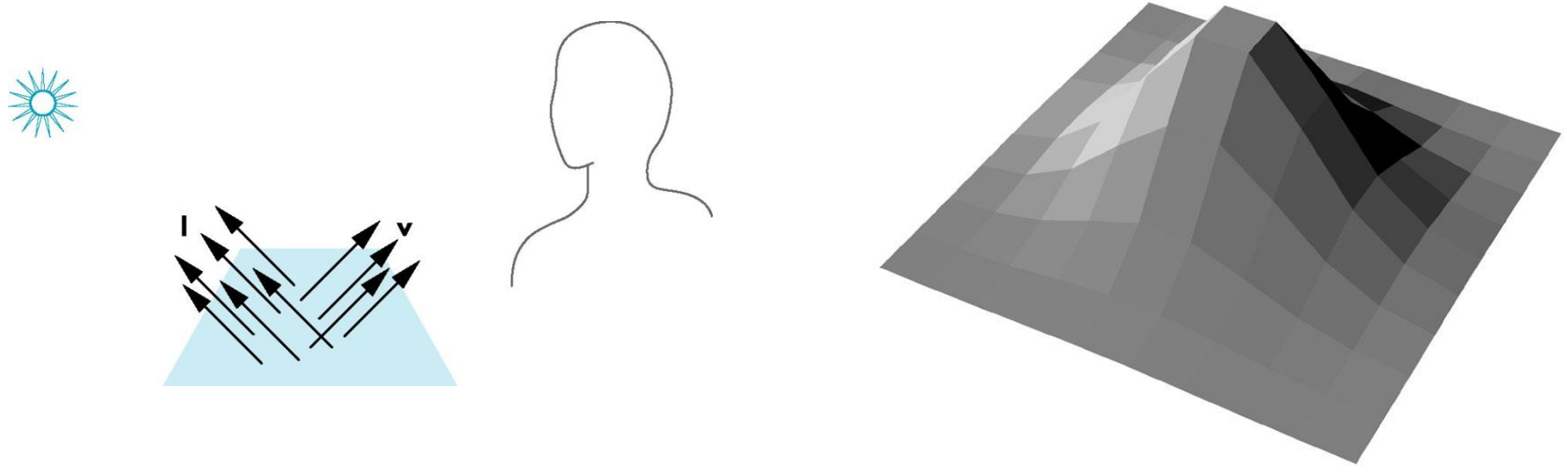
非顶点的像素的颜色？



由顶点处的信息插值得到！

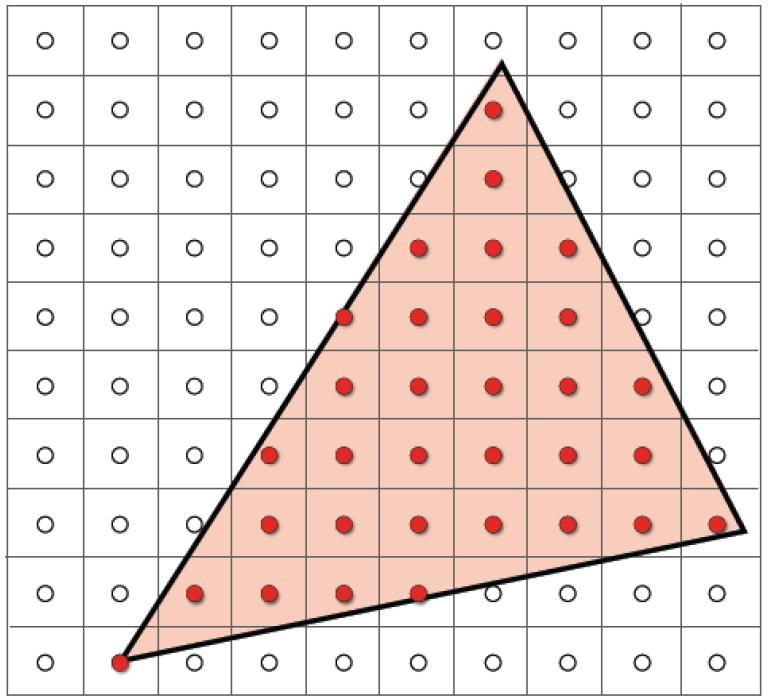
1. Flat Shading (Shade each triangle)

- 每个三角形中的像素的法向都一样（三角形的法向）
- 相当于：视点在无穷远，光源在无穷远
  - 视点方向v和入射方向l都是常量



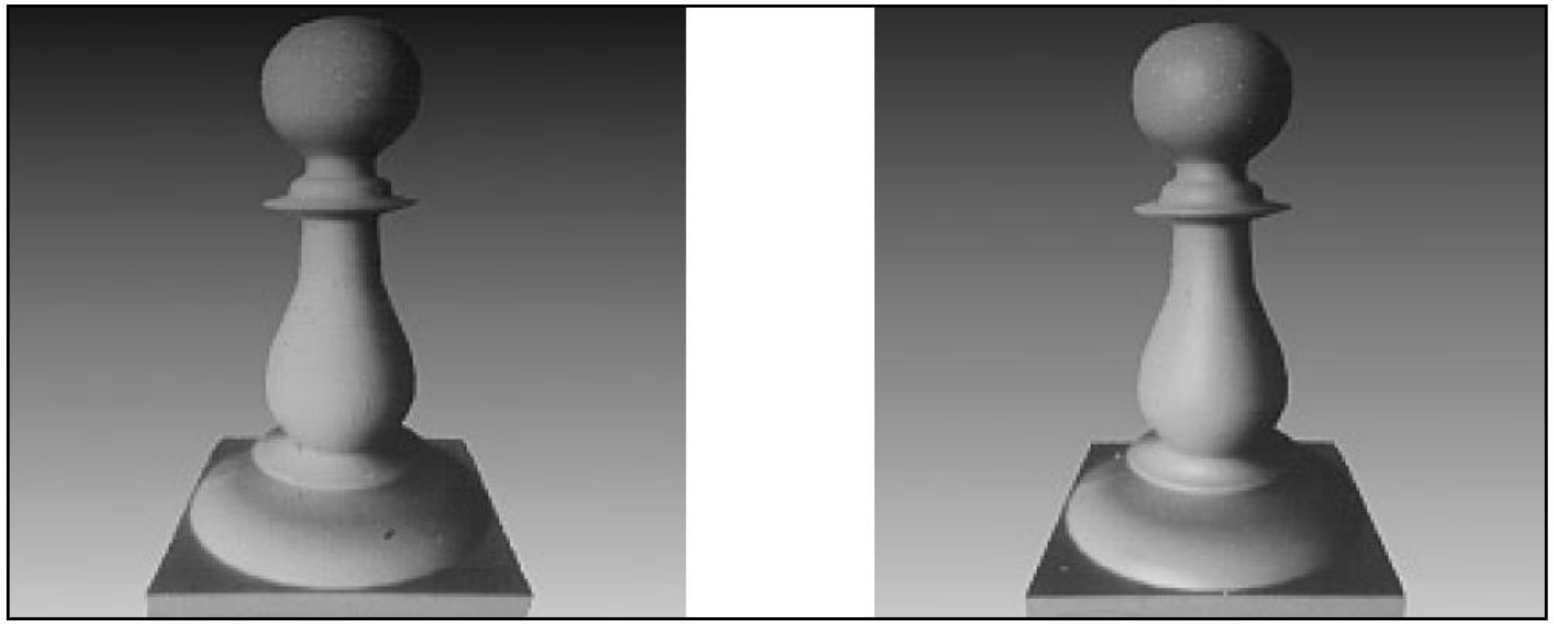
2. Gouraud Shading (Shade each vertex)

- 有3个顶点的颜色插值得到像素的颜色
- OpenGL 提供的方法



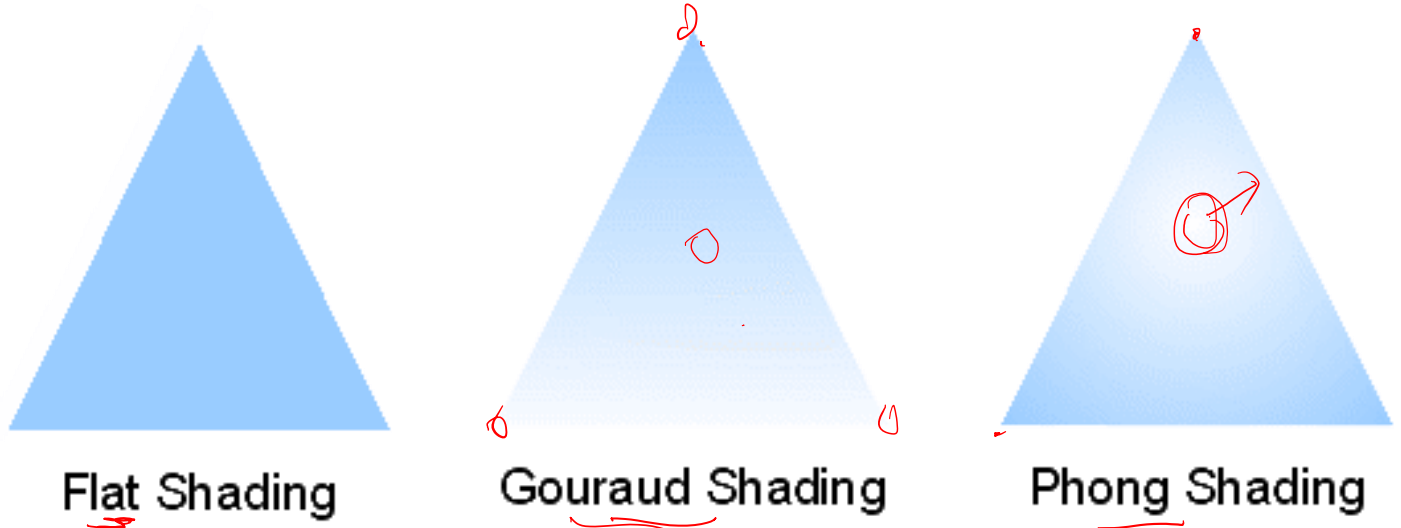
3. Phong Shading (Shade each pixel)

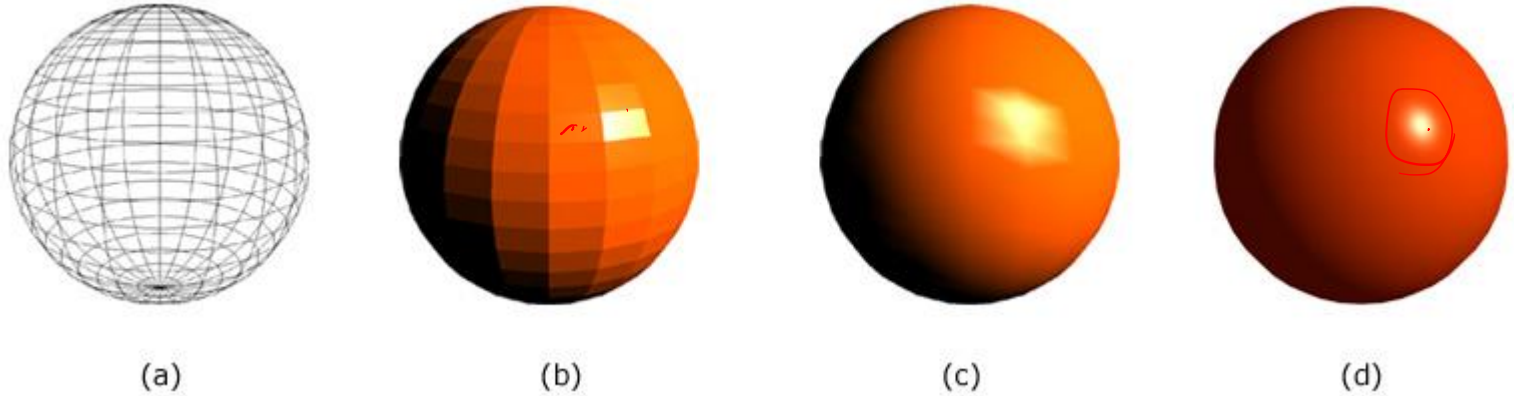
- 根据每个顶点的法向，插值出三角形内部各点的法向，然后基于光照模型计算出各点的颜色



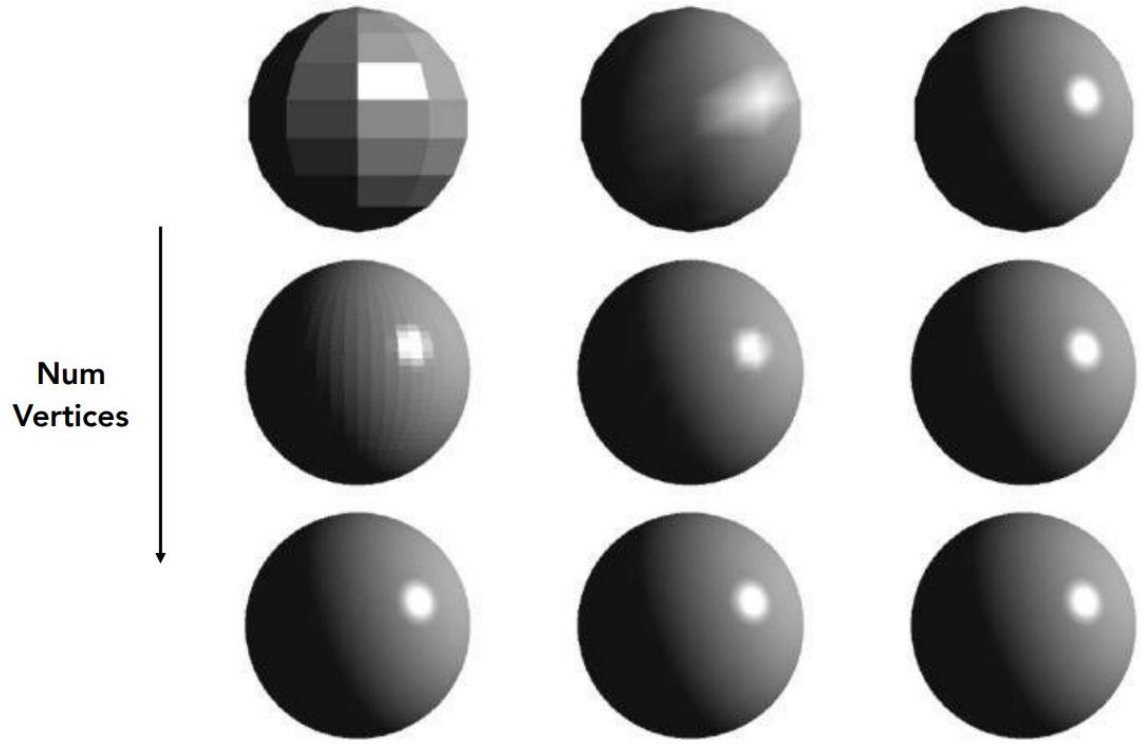
Gouraud　　Phong

Shading Comparisons (Face, Vertex, Pixel)





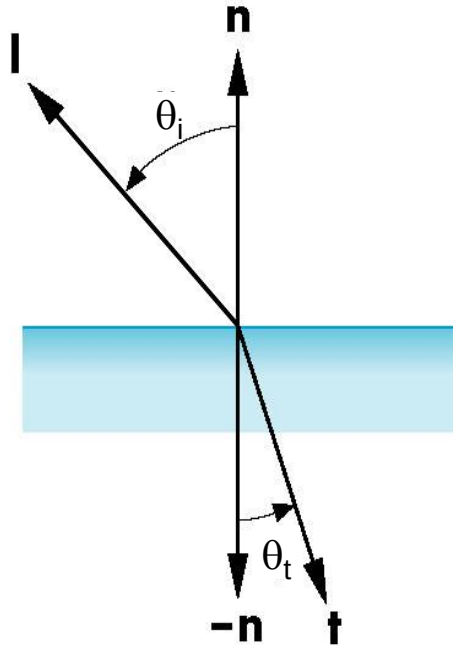
Shading Comparisons (Face, Vertex, Pixel)



Shading freq. :	Face	Vertex	Pixel
Shading type :	Flat	Gouraud	Phong

更复杂的光照模型

- 折射（透明体、水等）
- 多次反射
- 焦散
- 环境映射
- …



说明

- 对于顶点的着色在几何处理流程中就做好了，顶点的颜色作为顶点的属性传入到片元处理流程中使用
- 片元处理流程还可使用其他顶点属性
  - 法向
  - 深度
  - 纹理坐标
  - ...

图形渲染API

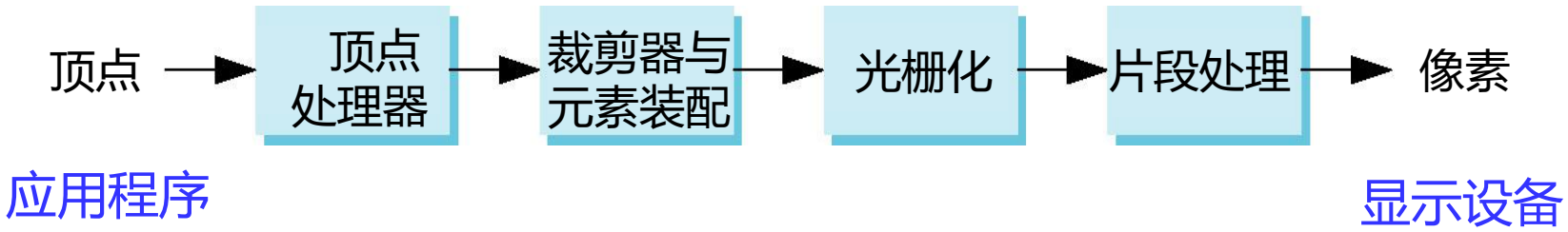
Application Program Interfaces

图形渲染API

- OpenGL：开放的接口，跨平台
- DirectX3D： Microsoft, Windows操作系统支持较好
- Vulkan: next generation of OpenGL

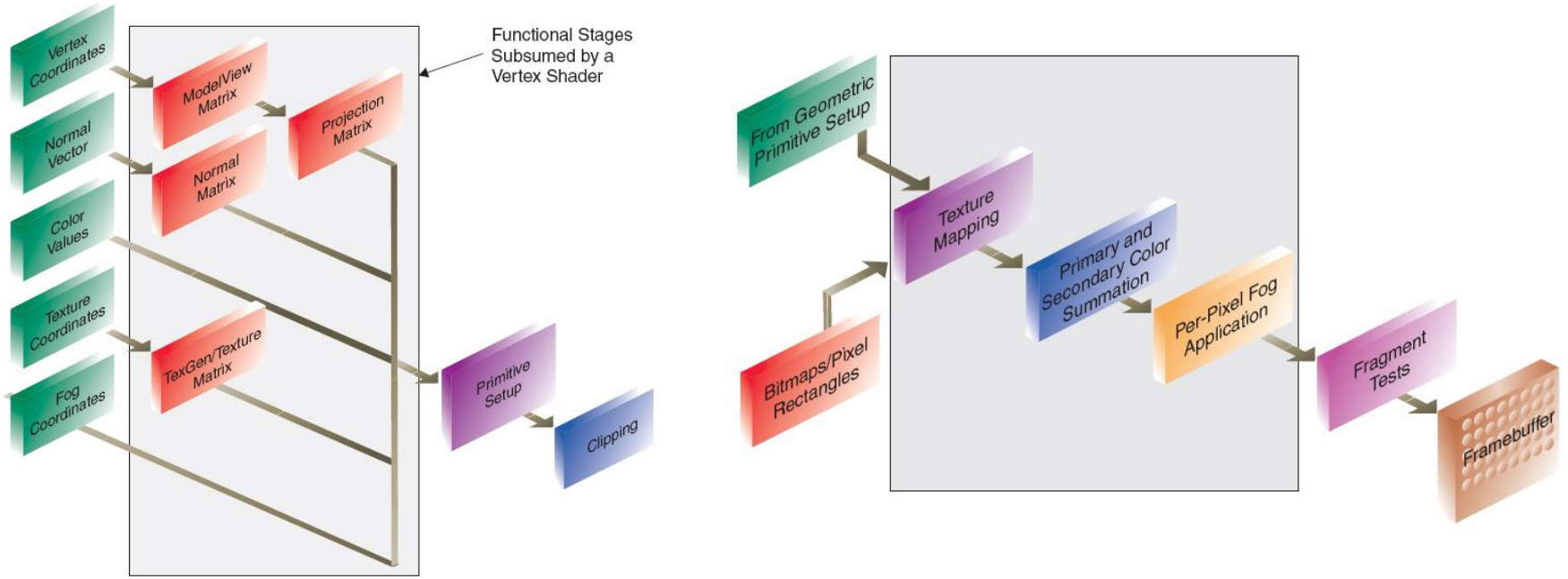
固定（不可编程）渲染管线

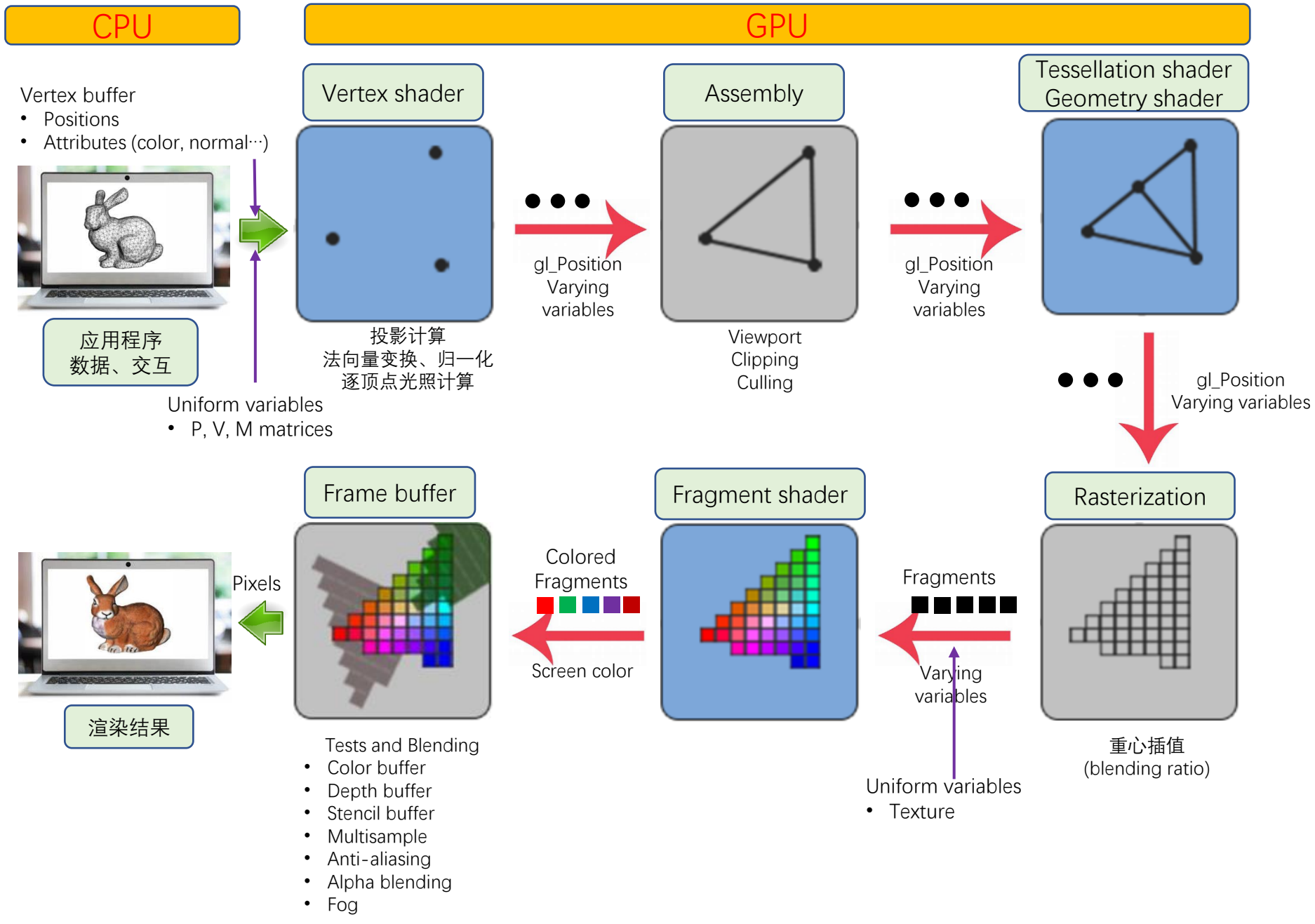
- 按照API提供的接口函数逐步操作、处理每个对象
- 优点：对初级用户使用方便
- 缺点：渲染效果一般，无法进行高级渲染
- 流水线体系



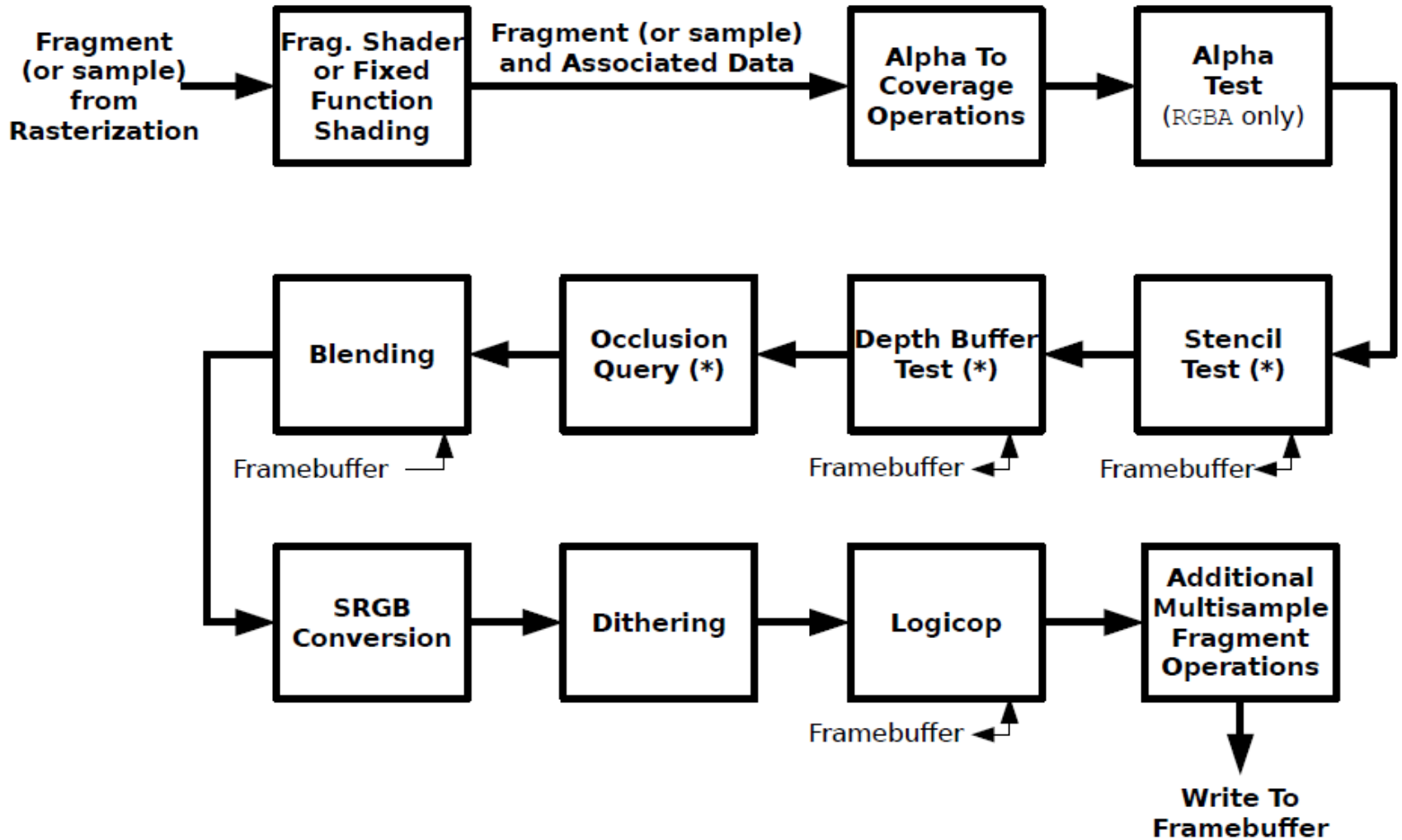
可编程渲染管线

- 固定管线：固定的处理模式（如Phong光照模型），至多有些参数可调，不够灵活
- 可编程管线
  - Vertex processor、fragment processor：programmable




CPU
GPU
Vertex buffer
• Positions
• Attributes (color, normal···)
应用程序
数据、交互
Uniform variables
• P, V, M matrices
Vertex shader
投影计算
法向量变换、归一化
逐顶点光照计算
gl_Position
Varying
variables
Assembly
Viewport
Clipping
Culling
gl_Position
Varying
variables
Tessellation shader
Geometry shader
gl_Position
Varying variables
Rasterization
重心插值
(blending ratio)
Fragments
Varying
variables
Uniform variables
• Texture
Fragment shader
Colored
Fragments
Screen color
Frame buffer
Tests and Blending
• Color buffer
• Depth buffer
• Stencil buffer
• Multisample
• Anti-aliasing
• Alpha blending
• Fog
Pixels
渲染结果

Fragment Shader



逐片元操作

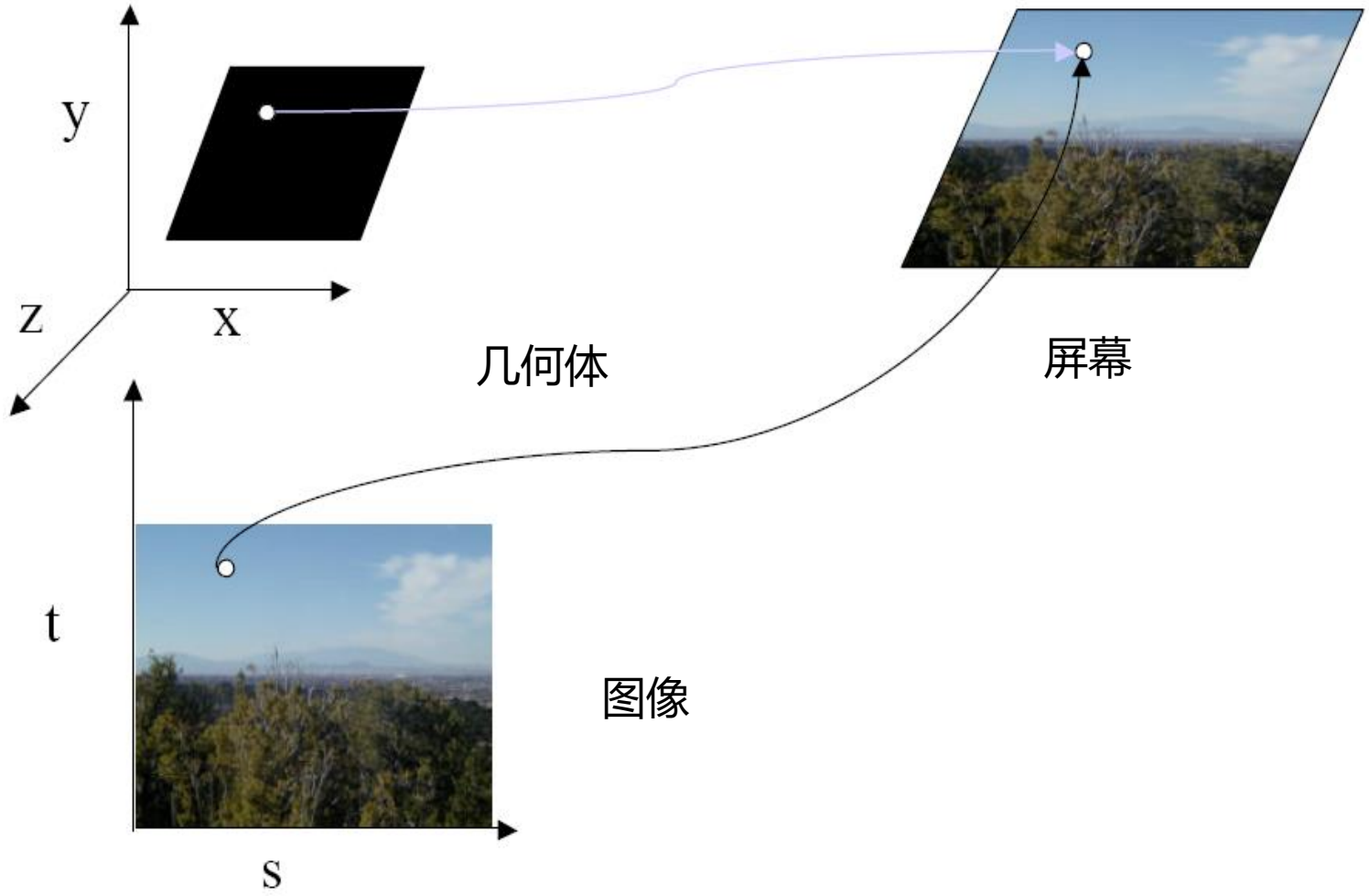
- 每个片元经过片元着色器处理后，还可以执行以下一些操作
  - 剪切测试（scissor test）
  - 多重采样片元操作（multisample fragment operations）
  - 模板测试（stencil test）
  - 深度测试（depth test）
  - 融混（blending）
  - 抖动（dithering）
  - 逻辑操作

逐片元操作的应用

- 深度缓存
  - 隐藏面消除（z buffer algorithm）
  - 阴影绘制（shadow mapping）
- 模板缓存
  - 汽车驾驶模拟显示中的挡风玻璃
  - 多道绘制技术（multipass rendering）
- 融混
  - 透明物体的绘制
- 多重采样与抖动
  - 反走样
- 利用逐片元操作，可以实现很多特殊任务的绘制

OpenGL纹理映射

纹理映射

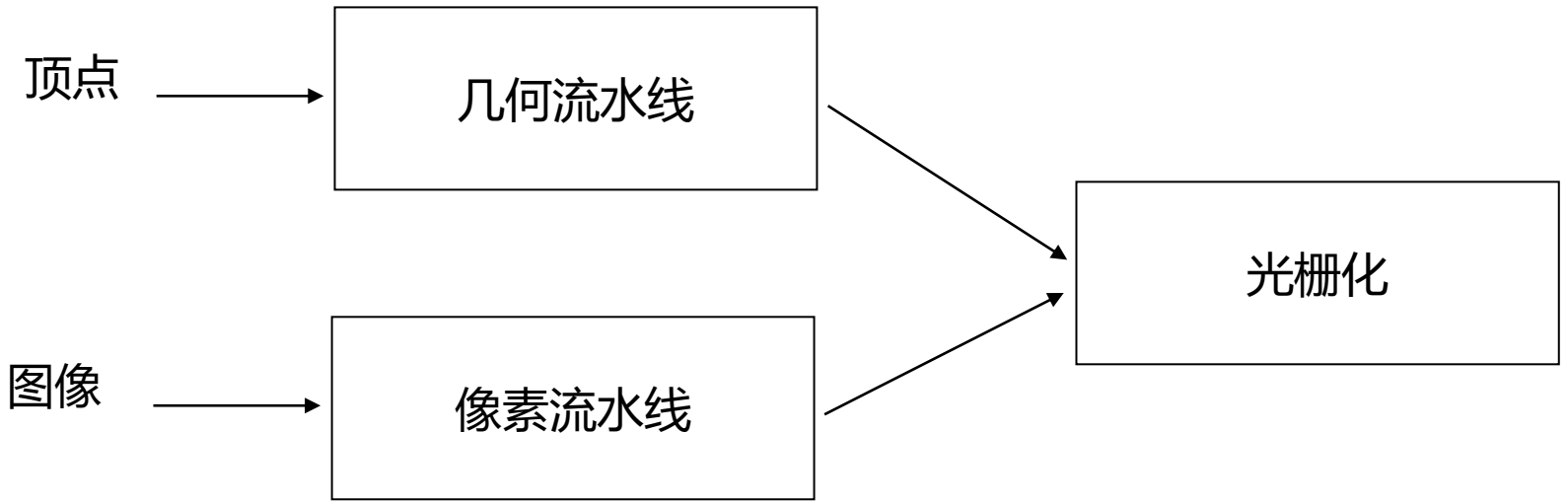


纹理示例

- 纹理（下方）是 256 × 256 的图像，它被映射到一个矩形上，经透视投影后的结果显示在上方

纹理映射与OpenGL流水线

- 图像与几何分别经过不同的流水线，在光栅化时合二为一
  - 复杂纹理并不影响几何的复杂性



指定纹理图像

- 利用CPU内存中的纹理元素数组定义纹理图像
  - `GLubyte my_texels[512][512];`
- 定义纹理图像所用的像素图
  - 扫描图像
  - 由应用程序代码创建
- 激活纹理映射
  - `glEnable(GL_TEXTURE_2D);`
  - OpenGL支持一至四维纹理映射

定义纹理所用的图像

- glTexImage2D(target,level,components,w,h,border,format,type,texels);

  target:纹理的类型，例如：GL_TEXTURE_2D

  level:用于mipmapping(稍后讨论)

  components:每个纹理元素的分量数

  w,h:texels中以像素为单位的宽度与高度

  border:用于光滑处理(稍后讨论)

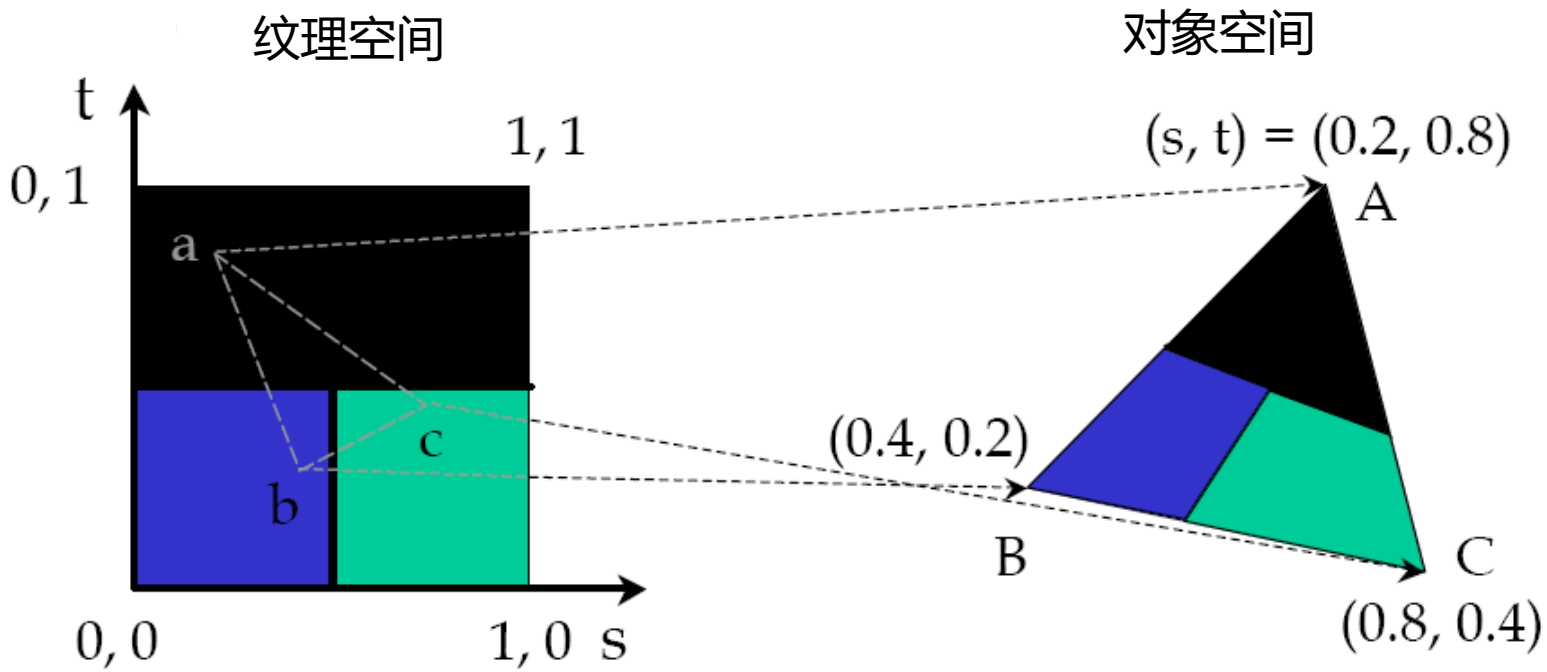
  format与type:描述纹理元素

  texels:指向纹理元素数组的指针

- glTexImage2D(GL_TEXTURE_2D,0,3,512,512,0,GL_RGB,GL_UNSIGNED_BYTE,my_texels);

映射纹理

- 基于参数纹理坐标
- `glTexCoord*()`指定每个顶点对应的纹理坐标



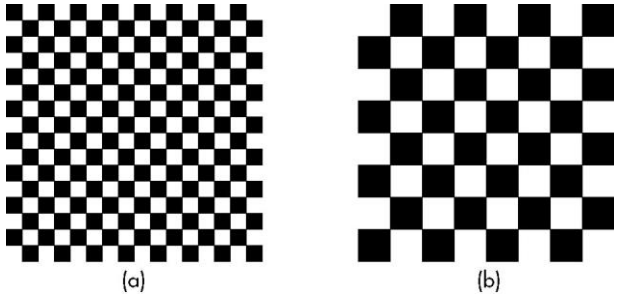
典型代码

```
glBegin(GL_POLYGON);
  glColor3f(r0,g0,b0);
  glNormal3f(u0,v0,w0);
  glTexCoord2f(s0,t0);
  glVertex3f(x0,y0,z0);
  glColor3f(r1,g1,b1);
  glNormal3f(u1,v1,w1);
  glTexCoord2f(s1,t1);
  glVertex3f(x1,y1,z1);
       .
       .
  glEnd();
```

注意为了提高效率，可以采用顶点数组

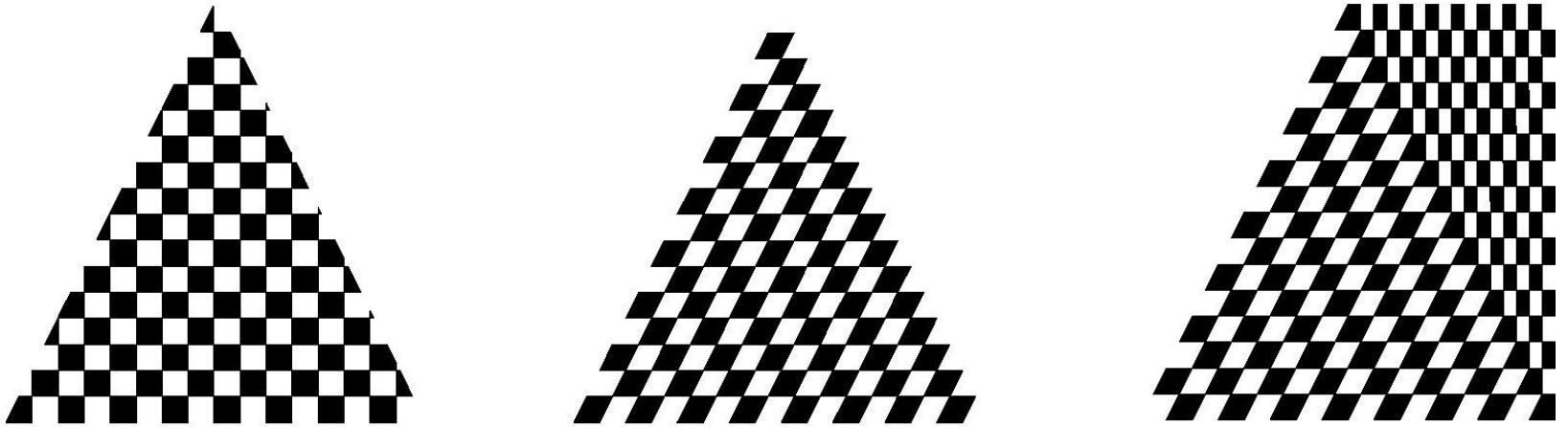
插值

- OpenGL应用双线性插值从给定的纹理坐标中求出适当的纹理元素
- 可以只应用纹理的一部分
  - 方法是只应用纹理坐标的一部分，如最大纹理坐标为(0.5,0.5)



变形

- 对于四边形，从纹理坐标到纹理元素的对应是比较直接的
- 对于一般的多边形，OpenGL需要确定一种方法，确定纹理坐标与纹理元素的对应
  - 可能会出现变形



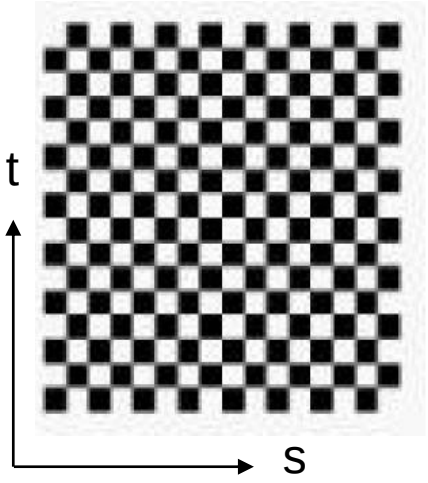
纹理参数

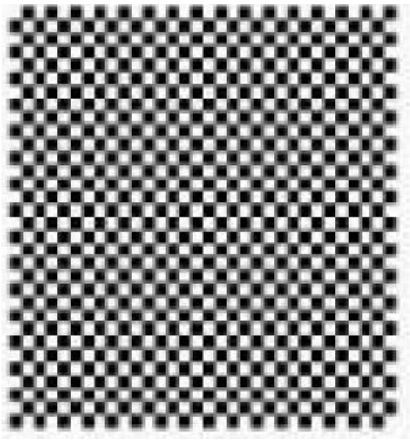
- OpenGL中有许多办法确定纹理的使用方式
  - Wrapping参数确定当s, t的值超出[0,1]区间后的处理方法
  - 应用filter模式就会不采用点取样方法，而是采用区域平均方法
  - Mimmapping技术使得能以不同的分辨率应用纹理
  - 环境参数确定纹理映射与明暗处理的交互作用

Wrapping模式

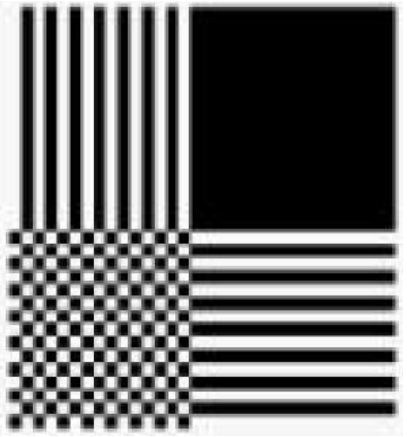
- 截断：若s,t>1就取1，若s,t<0就取0
- 重复：应用s,t模1的值

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP)
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT)
```





GL_REPEAT



GL_CLAMP

纹理对象

- 纹理也是状态的一部分
  - 如果不同的对象具有不同的纹理，那么OpenGL需要从处理器内存向纹理内存传送大量数据
- 新版本OpenGL提供了纹理对象功能
  - 每个纹理对象是一个图像
  - 纹理内存可以保存多个纹理对象

多重纹理

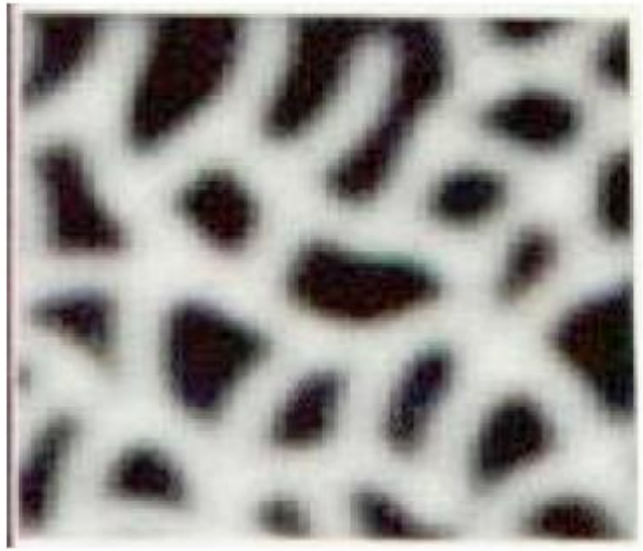
- 通常一个几何对象上只有一个纹理
- 有许多渲染效果需要多次应用纹理
  - 例：已有纹理的表面上有其它物体的阴影
  - 此时需要给阴影加上纹理
- 多重纹理的工作流程

几何对象　　纹理单元0　　纹理单元1　　纹理单元2　　帧缓冲区

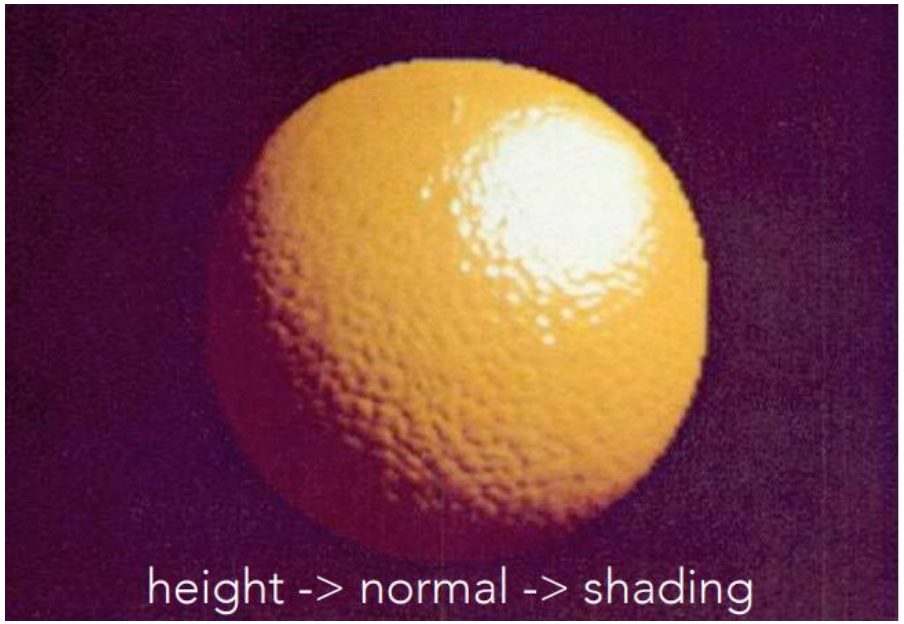
凹凸映射

凹凸映射

- 纹理图片可以表示其他信息，比如几何！

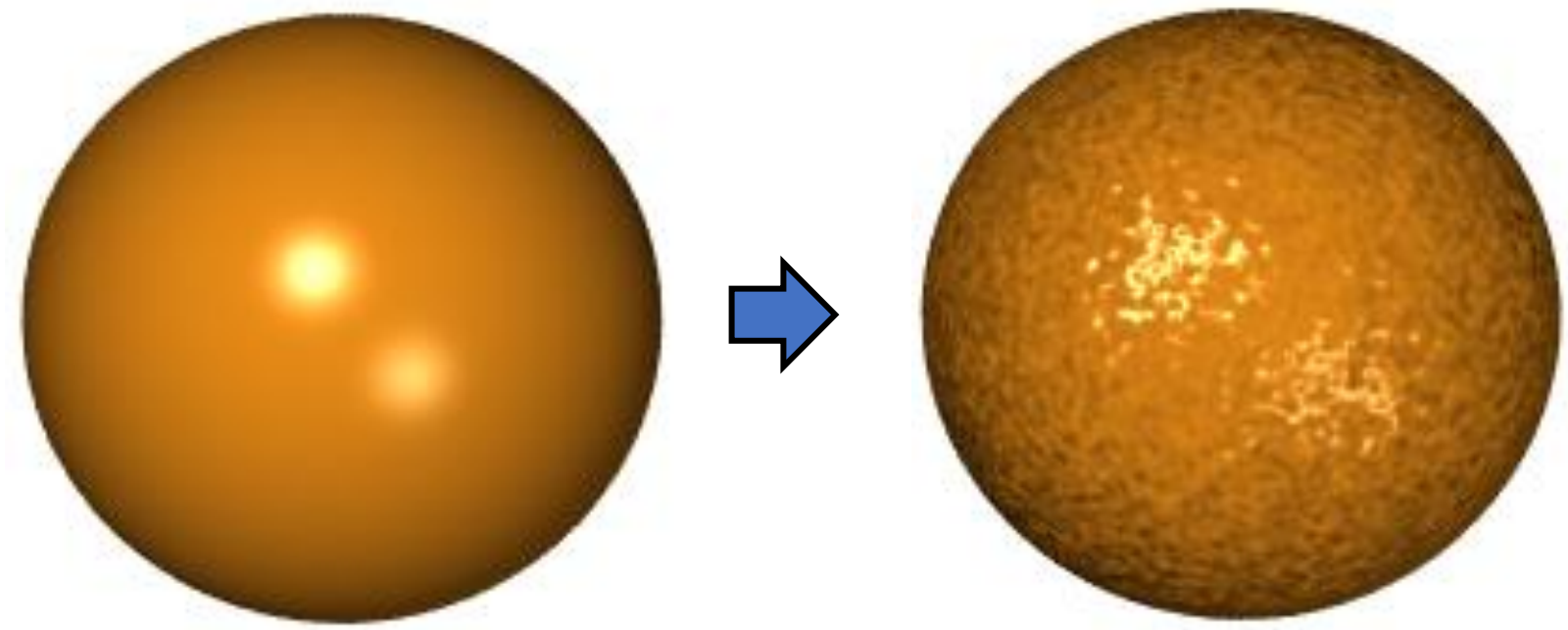


Height Map



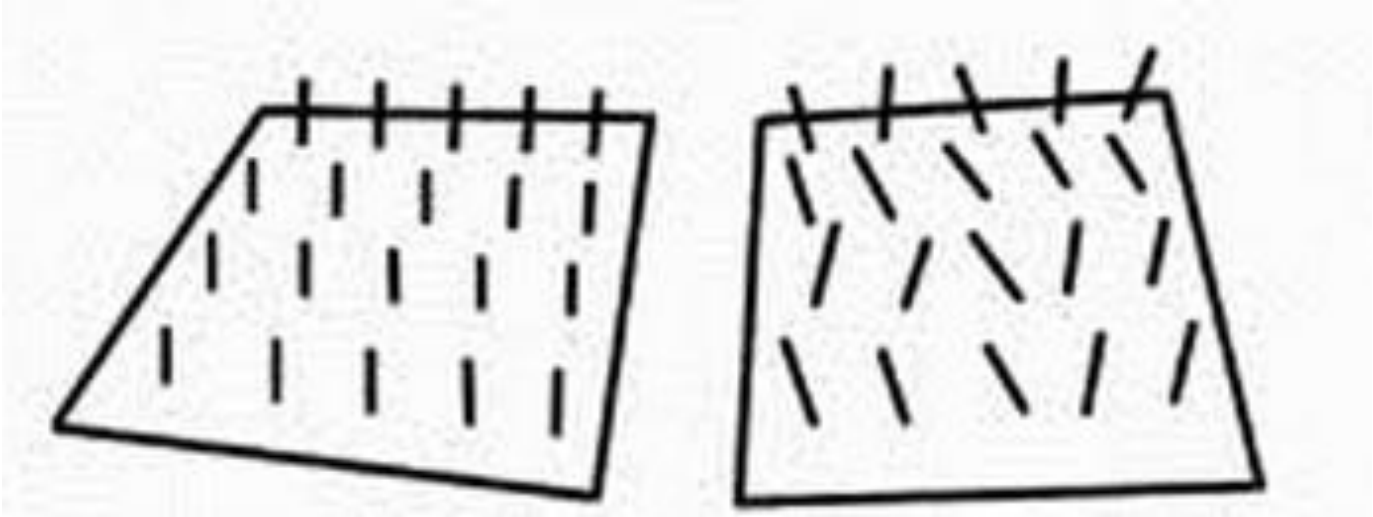
基本想法

- 对曲面的法向进行随机扰动
- 如此得到的图像就会显现出形状变化的错觉



法向

- 曲面上一点的法向量刻划曲面在该点的形状
- 如果用小量扰动曲面法向，那么就会得到具有小变化的曲面
- 如果在生成图像时进行这种扰动，那么就会从光滑的模型得到具有复杂表面模型的图像

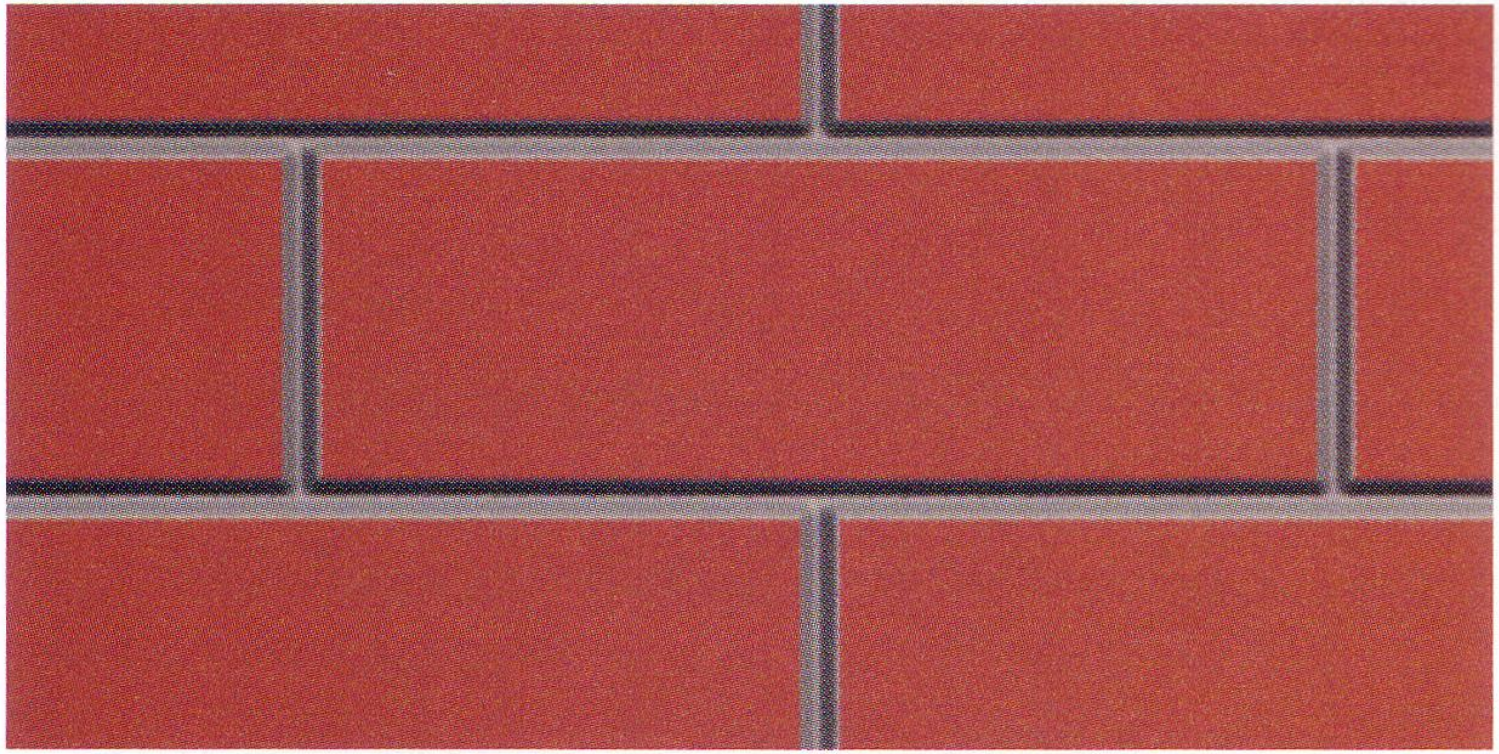


法向扰动方法

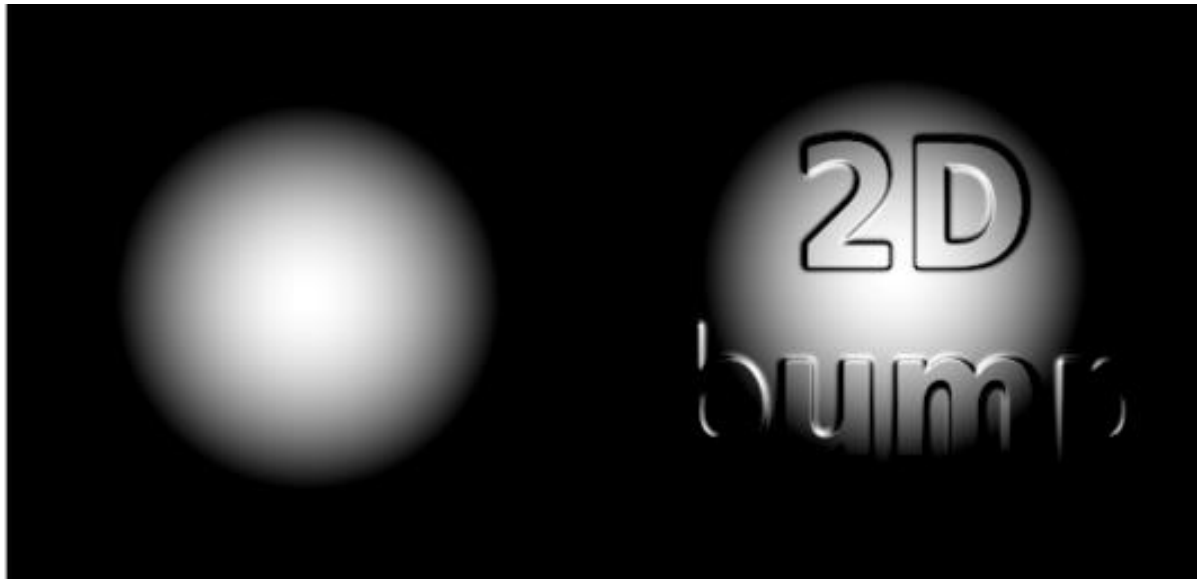
- 可以有多种方法对法向进行扰动
- 下述法向适用于参数曲面
  - 设曲面方程为$S(u,v)$
  - 在一点的单位法向为$n = \frac{S_u \times S_v}{\|S_u \times S_v\|}$
  - 假设曲面在法向移位$d(u,v)$ $(|d(u,v)| \ll 1)$
  - 那么变化后的法向近似为

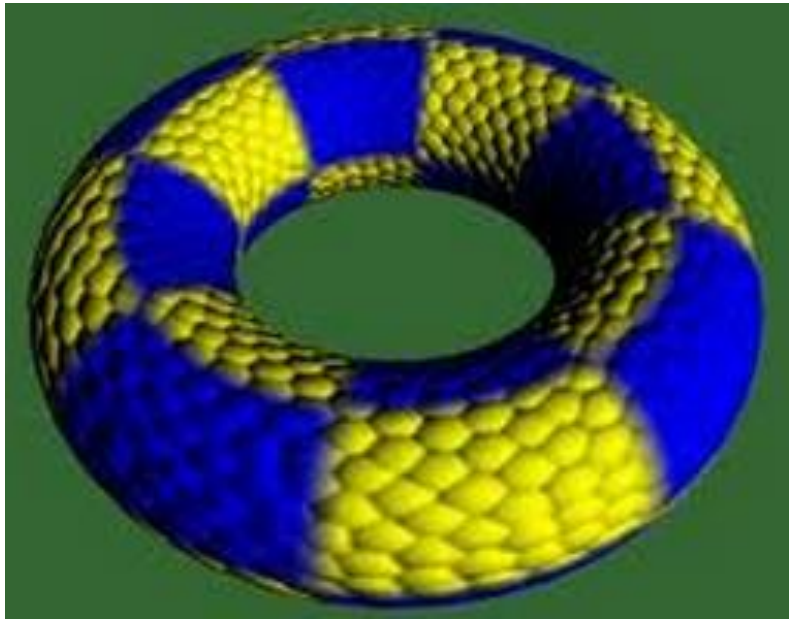
$$n' = n + dun \times S_u + dvn \times S_v$$

结果示例



2D bump





计算方法

- 曲面表示的偏导数计算
  - 基于曲面的表示解析计算，或者
  - 在当前点让u与v改变很小，找到四个最近点，用差商代替微商
- 位移函数的偏导数计算
  - 根据定义方式解析计算

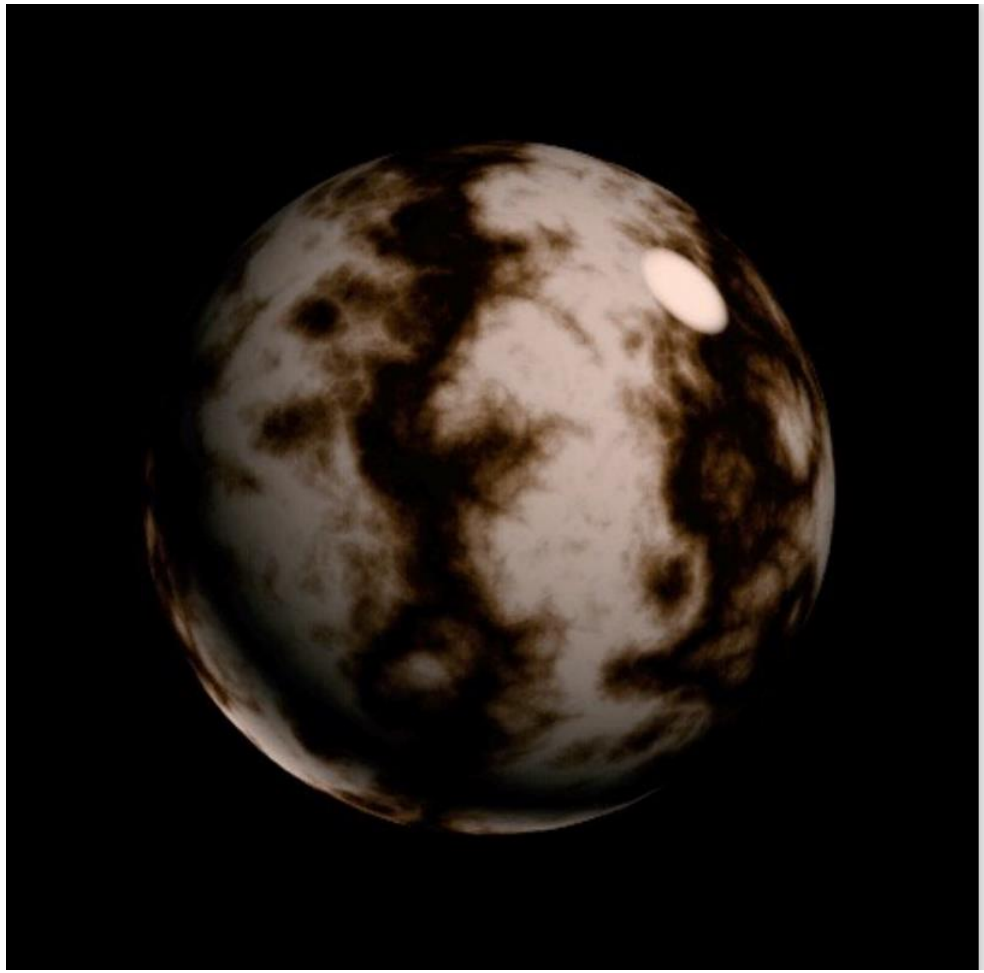
其他纹理

程序纹理（过程纹理）

- Procedural texture

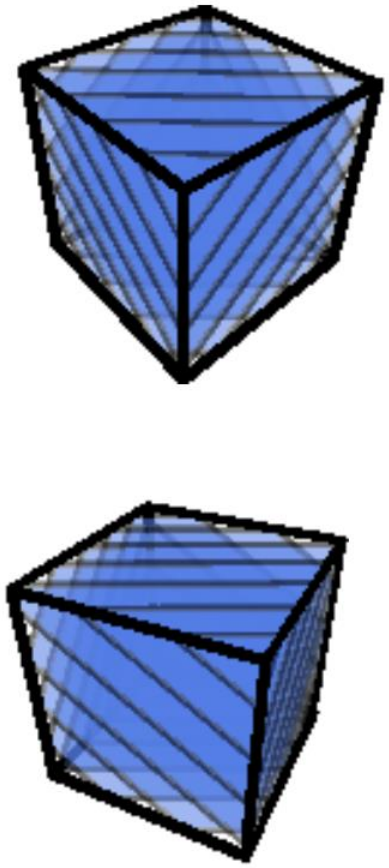
立体纹理

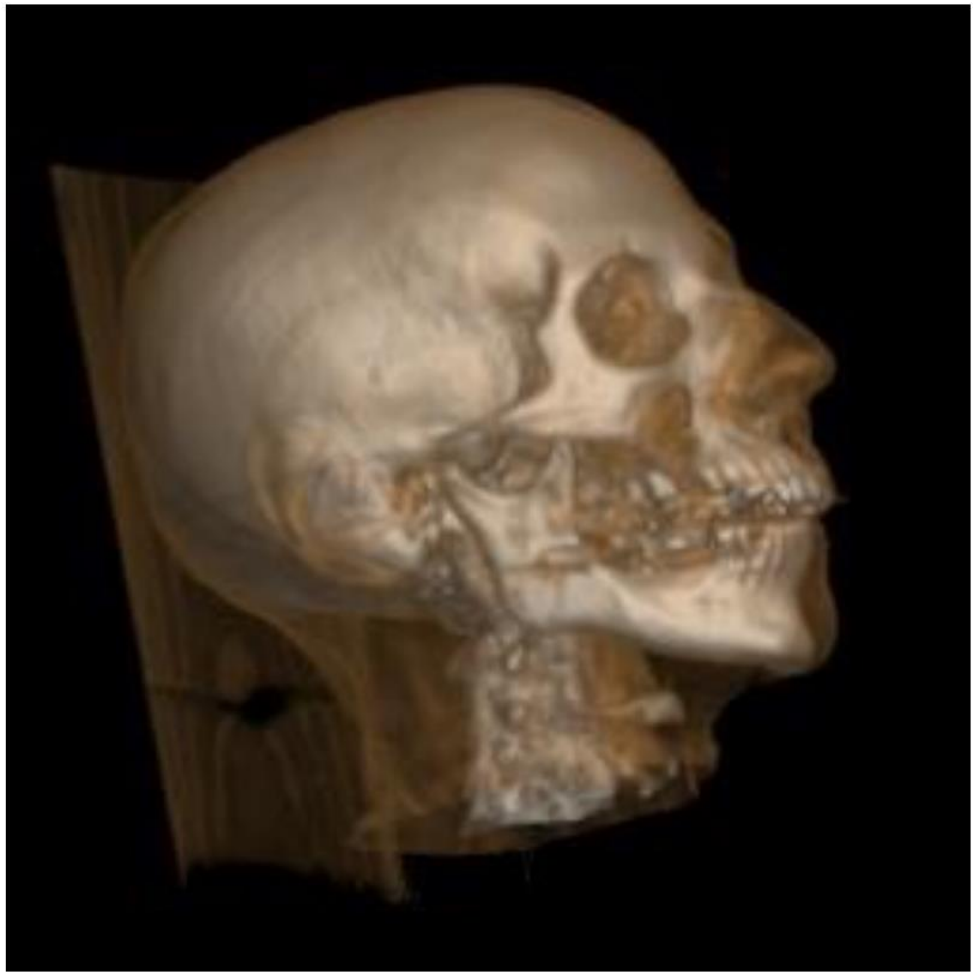
- solid texture
  - Perlin noise



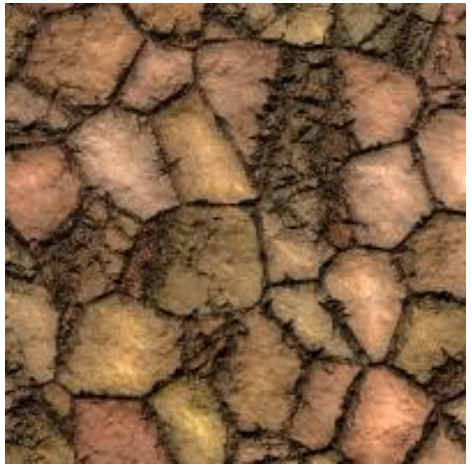
Volumetric Textures

- Volume Rendering

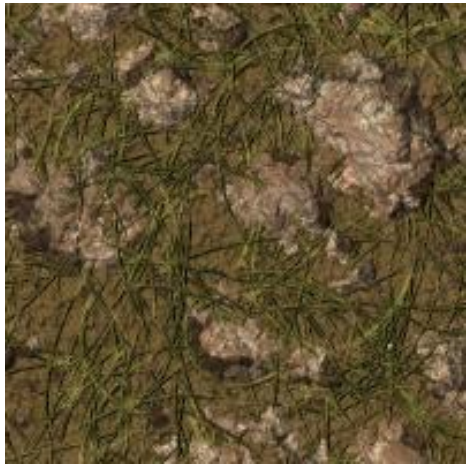
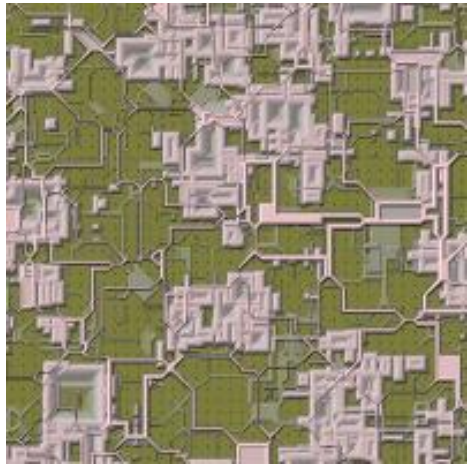




更多的纹理例子





Learn OpenGL

- https://learnopengl-cn.github.io/01%20Getting%20started/06%20Textures/

OpenGL
核心模式与立即渲染模式
扩展
状态机
对象
让我们开始吧
附加资源

核心模式与立即渲染模式

早期的OpenGL使用立即渲染模式（Immediate mode，也就是固定渲染管线），这个模式下绘制图形很方便。OpenGL的大多数功能都被库隐藏起来，开发者很少有控制OpenGL如何进行计算的自由。而开发者迫切希望能有更多的灵活性。随着时间推移，规范越来越灵活，开发者对绘图细节有了更多的掌控。立即渲染模式确实容易使用和理解，但是效率太低。因此从OpenGL3.2开始，规范文档开始废弃立即渲染模式，并鼓励开发者在OpenGL的核心模式(Core-profile)下进行开发，这个分支的规范完全移除了旧的特性。

当使用OpenGL的核心模式时，OpenGL迫使我们使用现代的函数。当我们试图使用一个已废弃的函数时，OpenGL会抛出一个错误并终止绘图。现代函数的优势是更高的灵活性和效率，然而也更难于学习。立即渲染模式从OpenGL**实际**运作中抽象掉了很多细节，因此它在易于学习的同时，也很难让人去把握OpenGL具体是如何运作的。现代函数要求使用者真正理解OpenGL和图形编程，它有一些难度，然而提供了更多的灵活性，更高的效率，更重要的是可以更深入的理解图形编程。

这也是为什么我们的教程面向OpenGL3.3的核心模式。虽然上手更困难，但这份努力是值得的。

现今，更高版本的OpenGL已经发布（写作时最新版本为4.5），你可能会问：既然OpenGL 4.5 都出来了，为什么我们还要学习OpenGL 3.3？答案很简单，所有OpenGL的更高的版本都是在3.3的基础上，引入了额外的功能，并没有改动核心架构。新版本只是引入了一些更有效率或更有用的方式去完成同样的功能。因此，所有的概念和技术在现代OpenGL版本里都保持一致。当你的经验足够，你可以轻松使用来自更高版本OpenGL的新特性。

> 当使用新版本的OpenGL特性时，只有新一代的显卡能够支持你的应用程序。这也是为什么大多数开发者基于较低版本的OpenGL编写程序，并只提供选项启用新版本的特性。

在有些教程里你会看见更现代的特性，它们同样会以这种红色注释方式标明。

扩展

OpenGL的一大特性就是对扩展(Extension)的支持，当一个显卡公司提出一个新特性或者渲染上的大优化，通常会以扩展的方式在驱动中实现。如果一个程序在支持这个扩展的显卡上运行，开发者可以使用这个扩展提供的一些更先进更有效的图形功能。通过这种方式，开发者不必等待一个新的OpenGL规范面世，就可以使用这些新的渲染特性了，只需要简单地检查一下显卡是否支持此扩展。通常，当一个扩展非常流行或者非常有用的时候，它将最终成为未来的OpenGL规范的一部分。

使用扩展的代码大多看上去如下：

```
if(GL_ARB_extension_name)
{
    // 使用硬件支持的全新的现代特性
}
else
{
    // 不支持此扩展：用旧的方式去做
}
```

使用OpenGL3.3时，我们很少需要使用扩展来完成大多数功能，当需要的时候，本教程将提供适当的指示。

Thank you!

Questions?