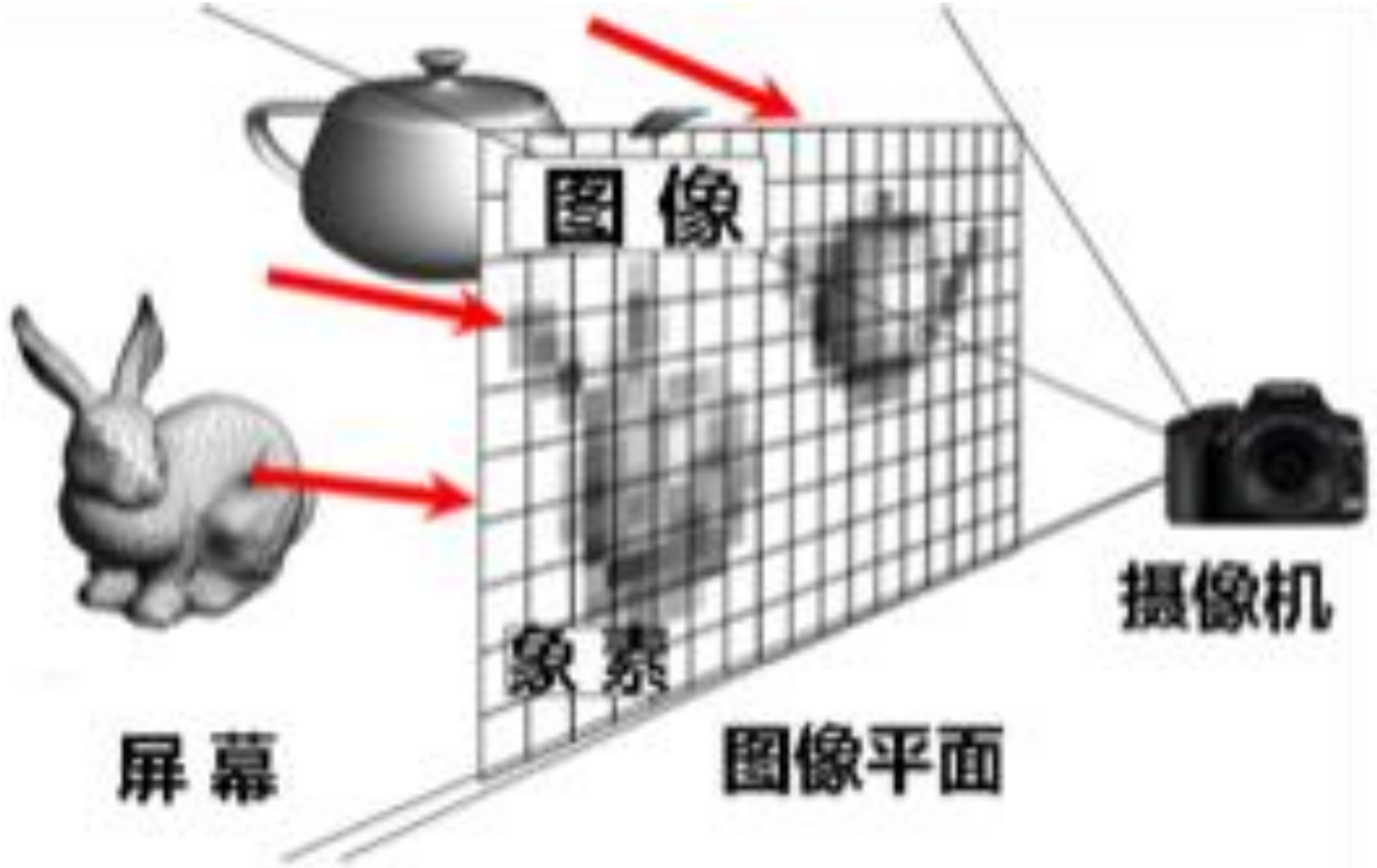
计算机图形学
Computer Graphics

刘利刚
lgliu@ustc.edu.cn
http://staff.ustc.edu.cn/~lgliu

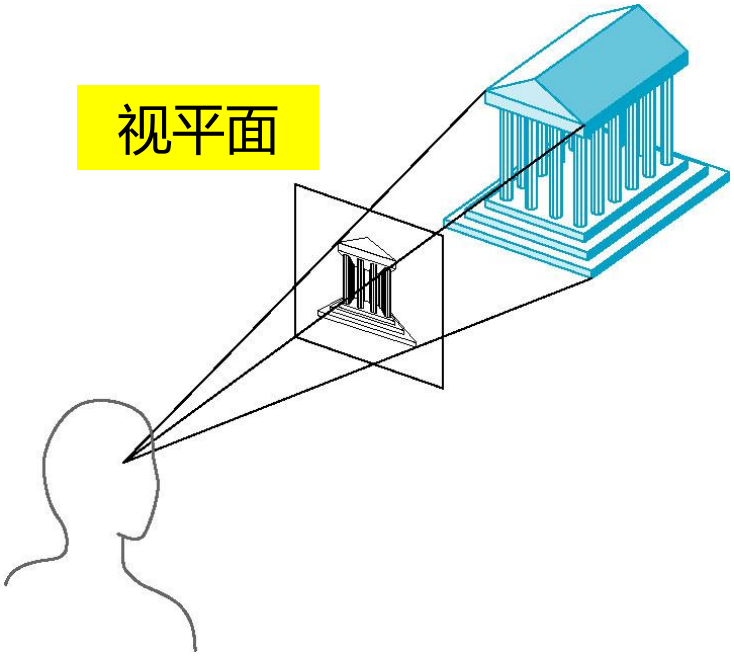
Graphics&Geometric Computing Lab
@USTC

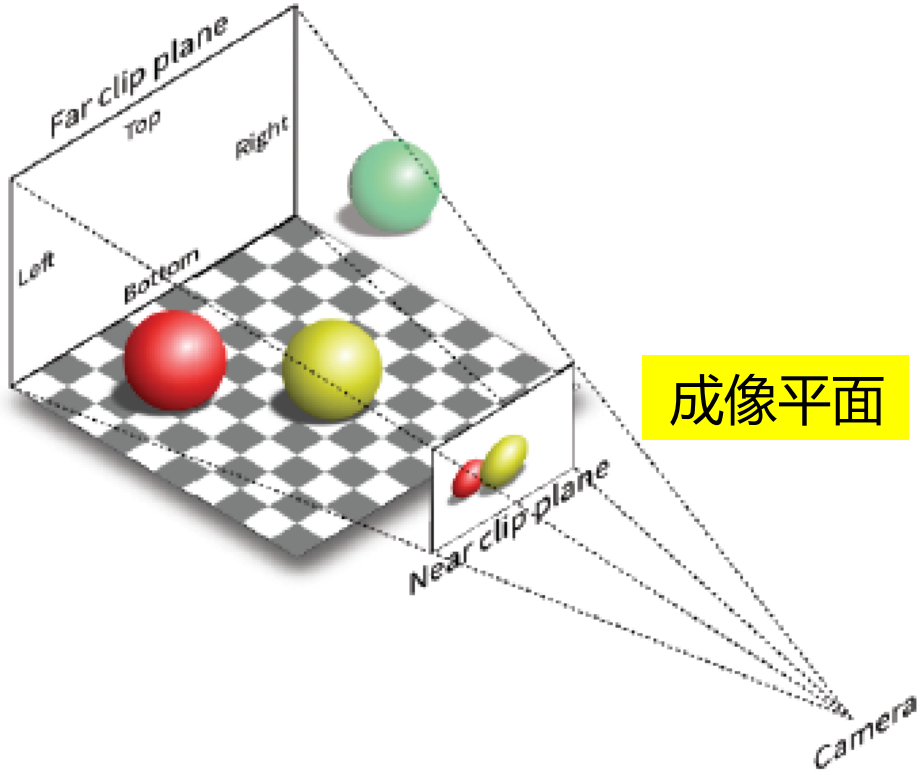
渲染流水线/管线
Rendering Pipeline

渲染：将3D模型成像到2D图像平面

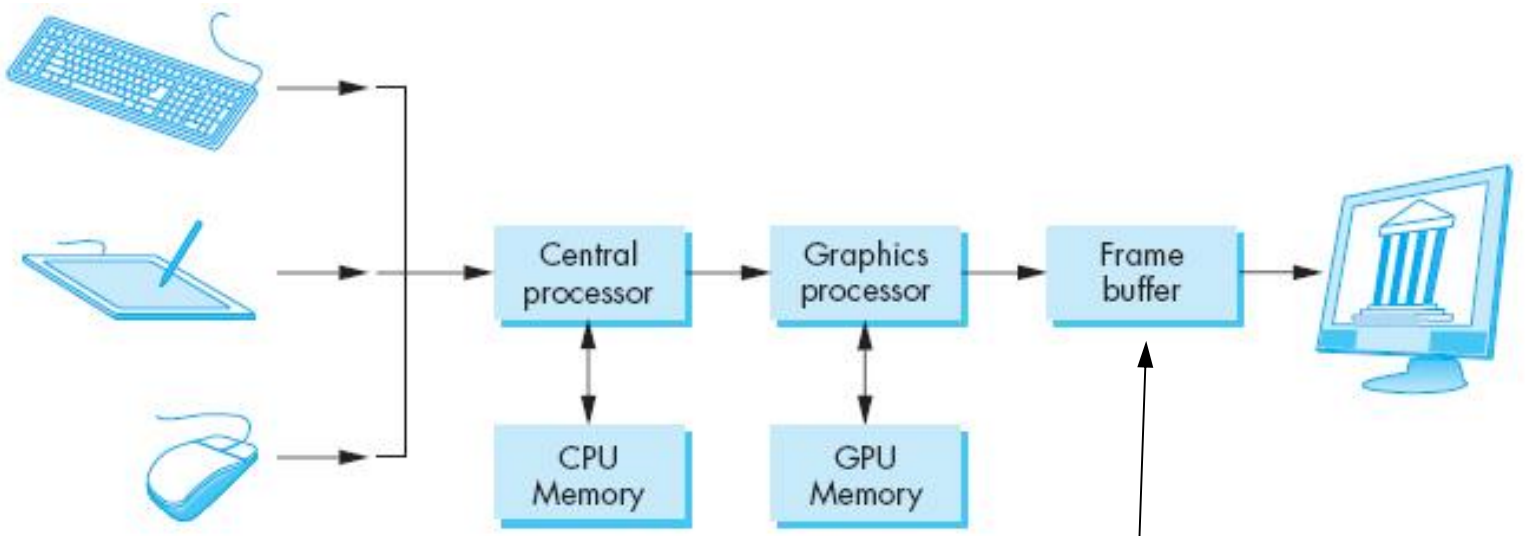


渲染：3D场景→2D图像


视平面


Far clip plane
Top
Right
Left
Bottom
成像平面
Near clip plane
Camera

3D应用程序



输入设备

图像在帧缓冲区形成

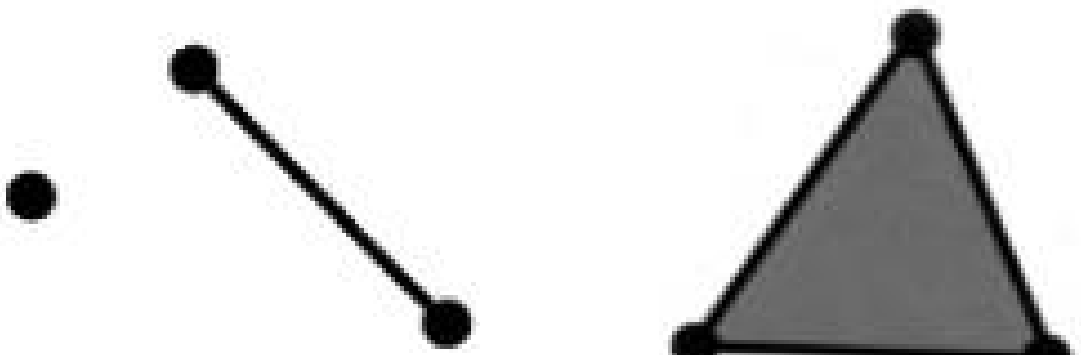
输出设备

渲染的两个主要阶段

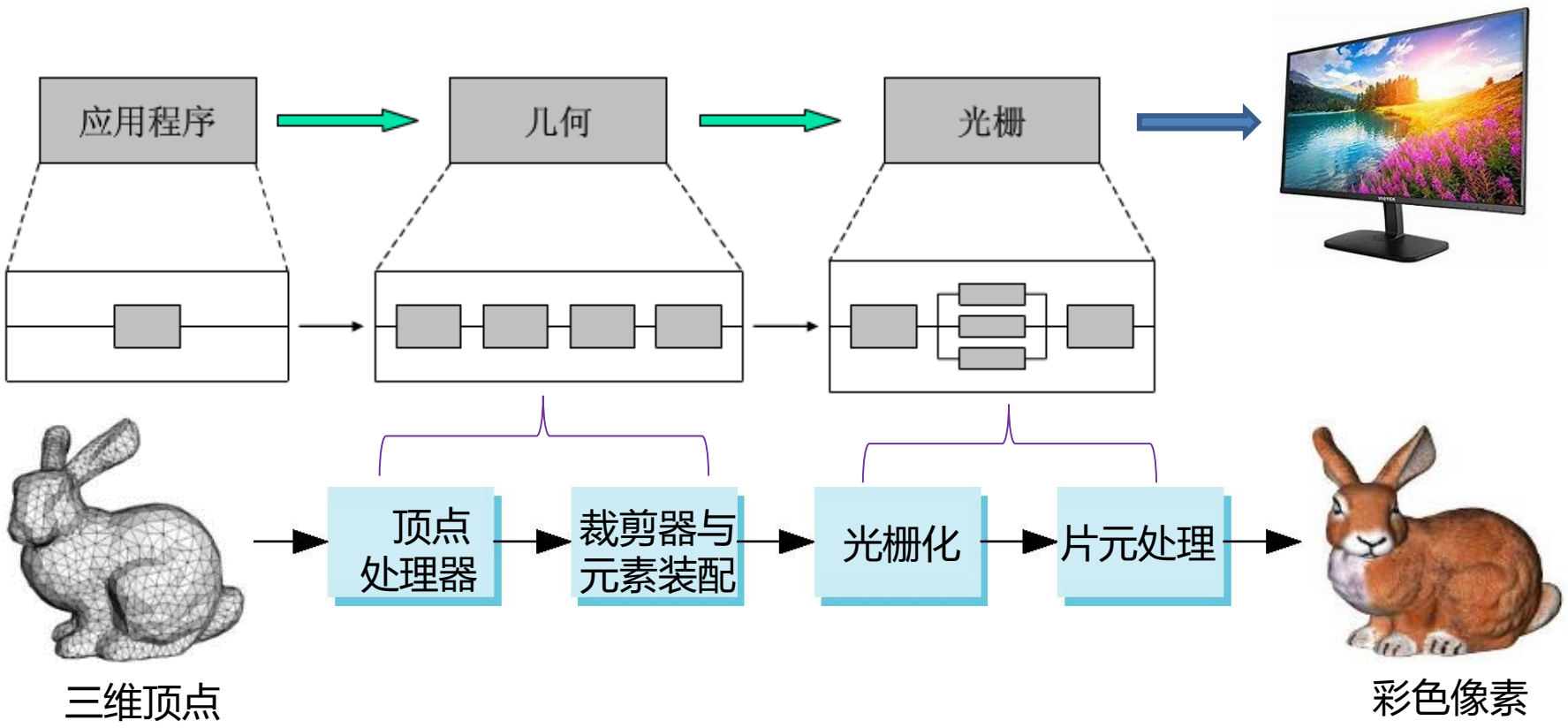
- 几何阶段
  - 将3D模型投影变换到图像平面
  - 决定可见的图元（图形元素）
- 光栅阶段
  - 决定可见的片元 (fragment)
  - 决定片元的颜色成为彩色像素

基本图元

- 2D
  - 点、线
- 3D
  - 点、线和三角形

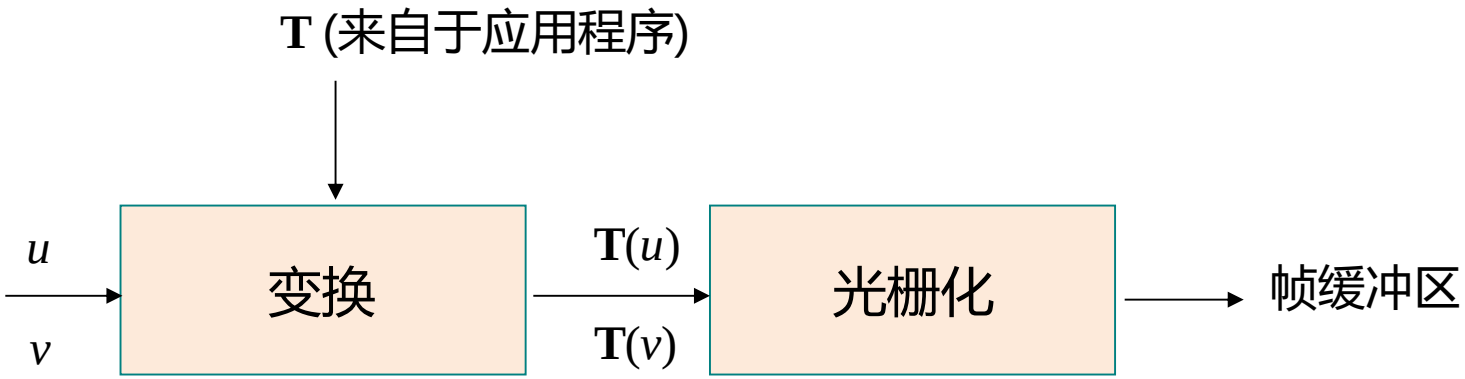


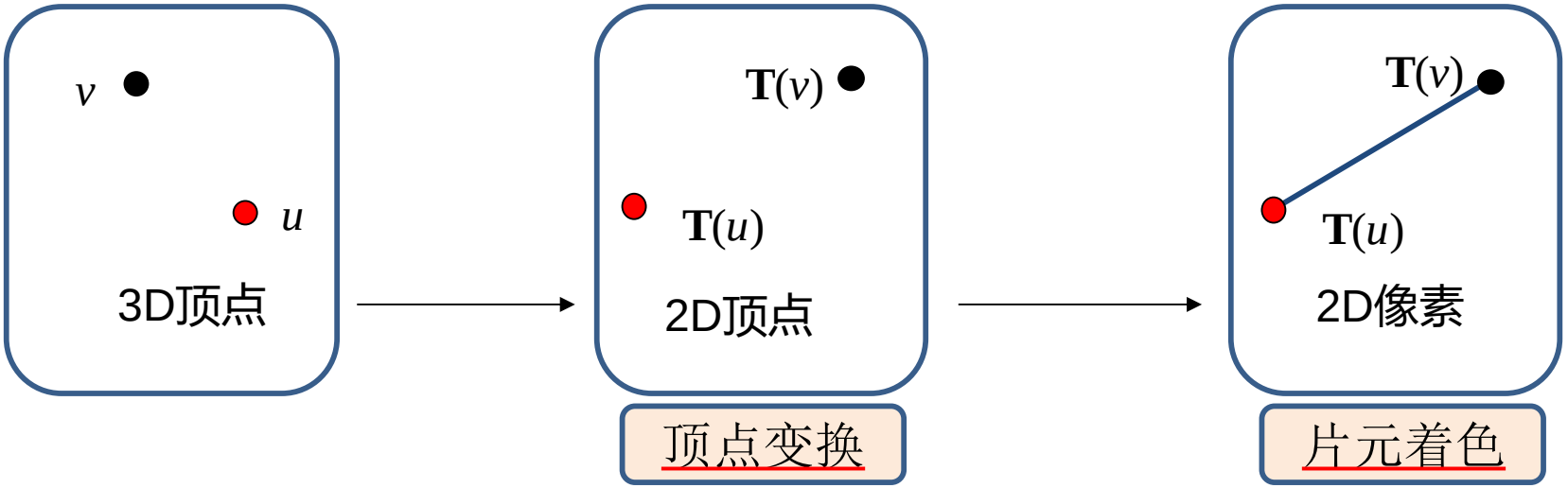
渲染管线：渲染计算的流水线



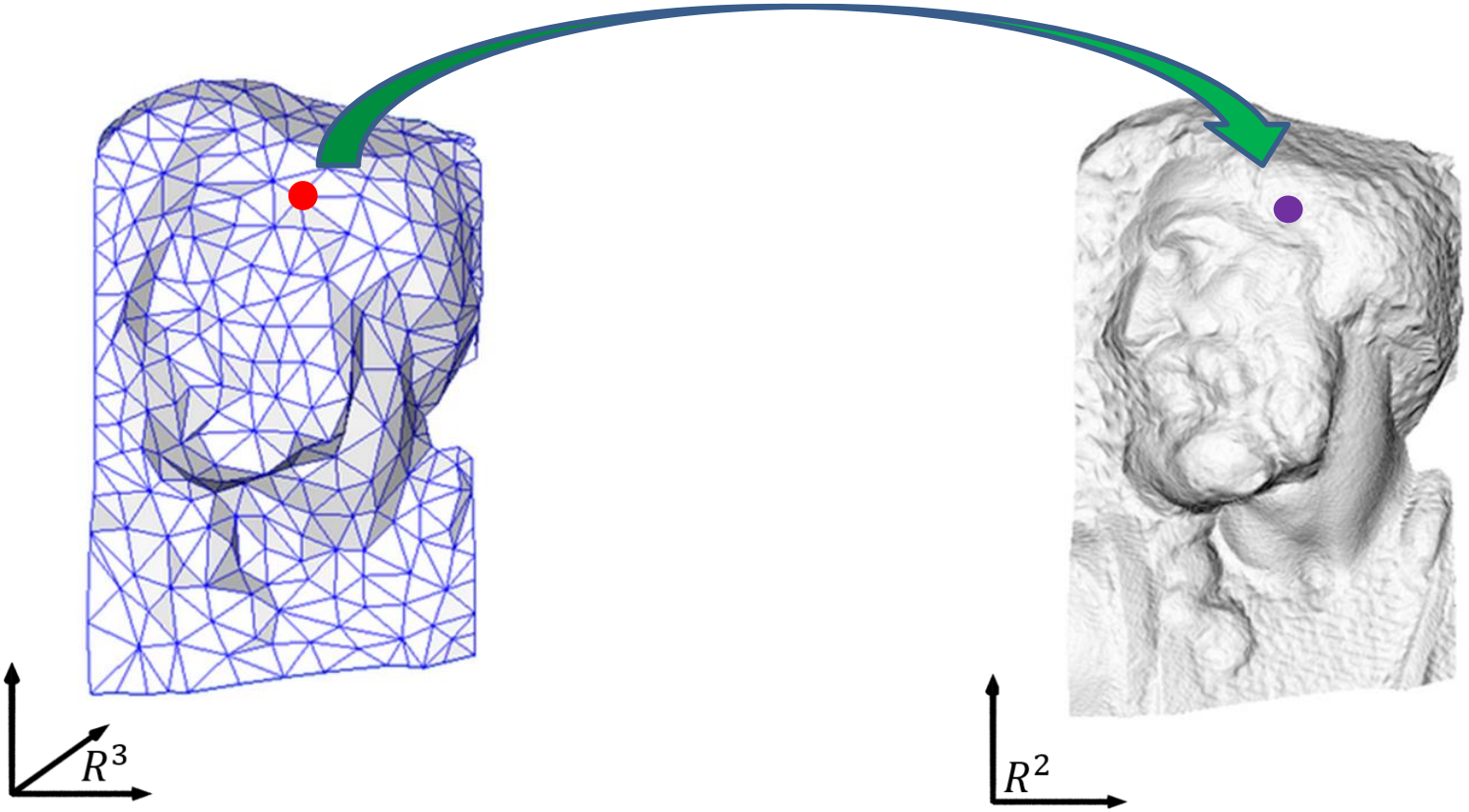
渲染流水线的过程

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

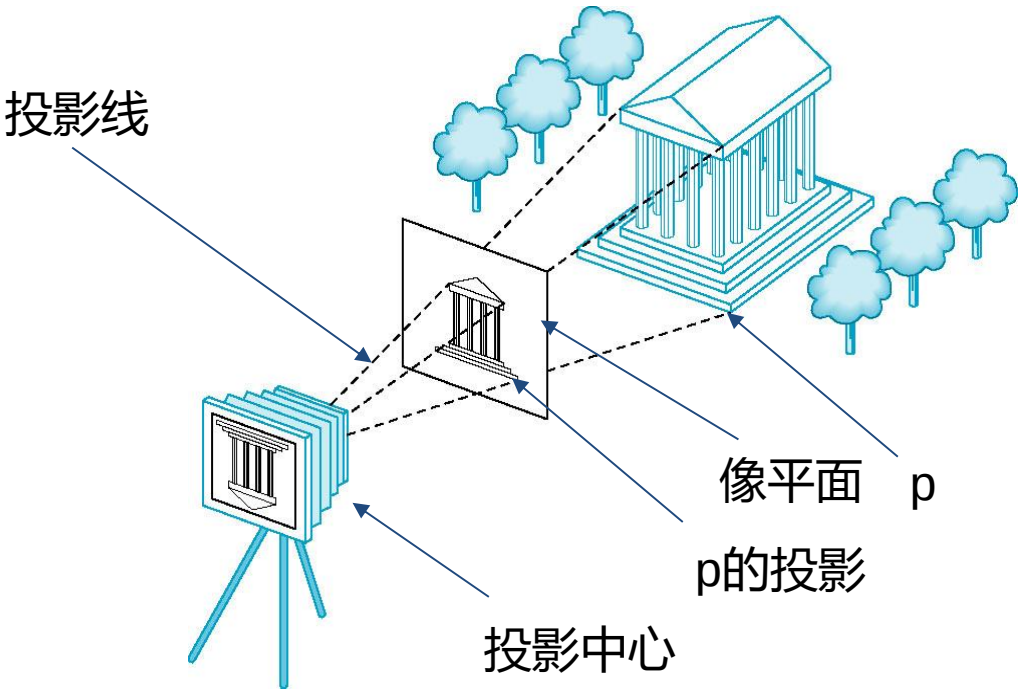
过程1：几何变换
逐顶点计算屏幕坐标

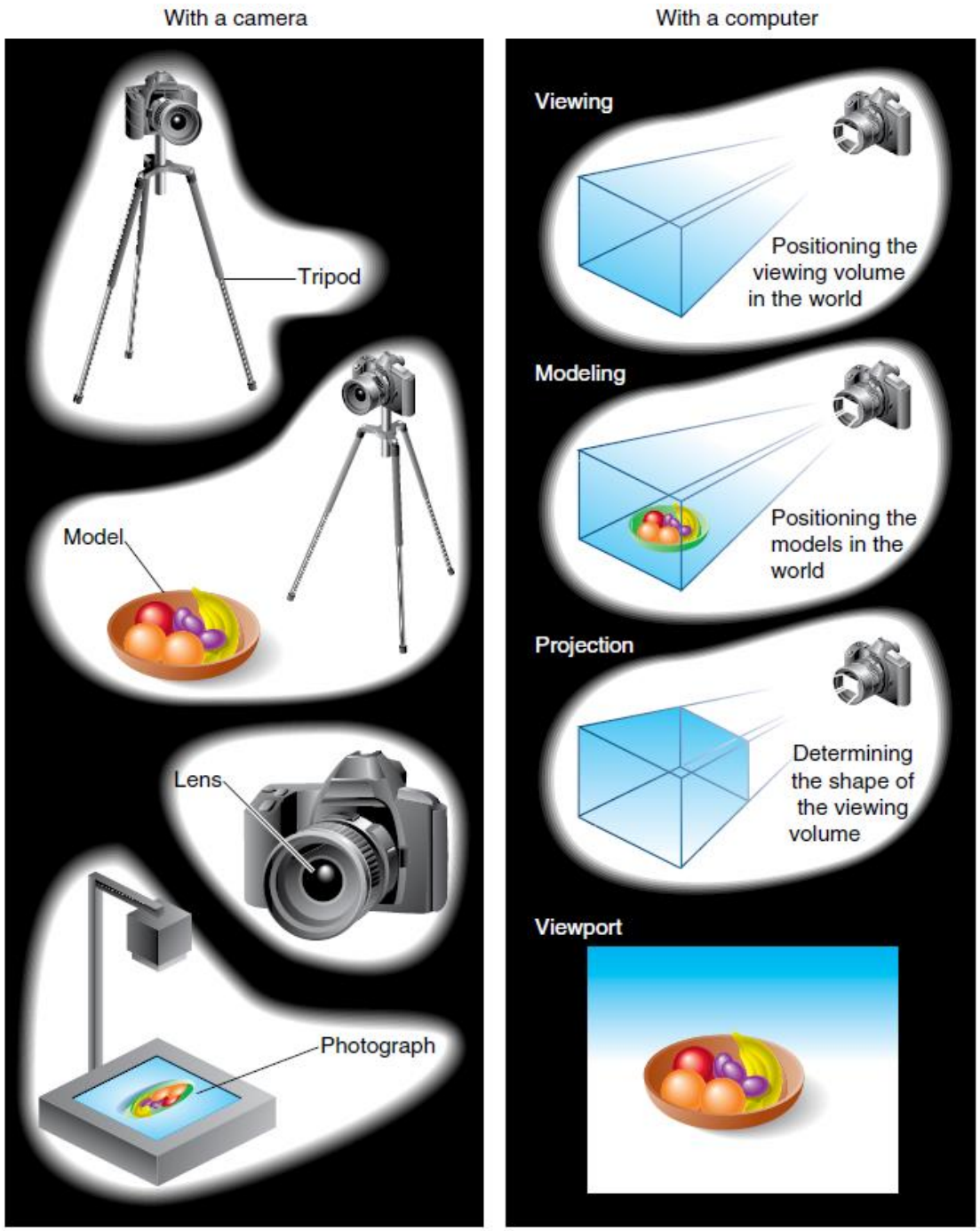


成像模型

- 3D场景在虚拟相机（眼睛）下的投影

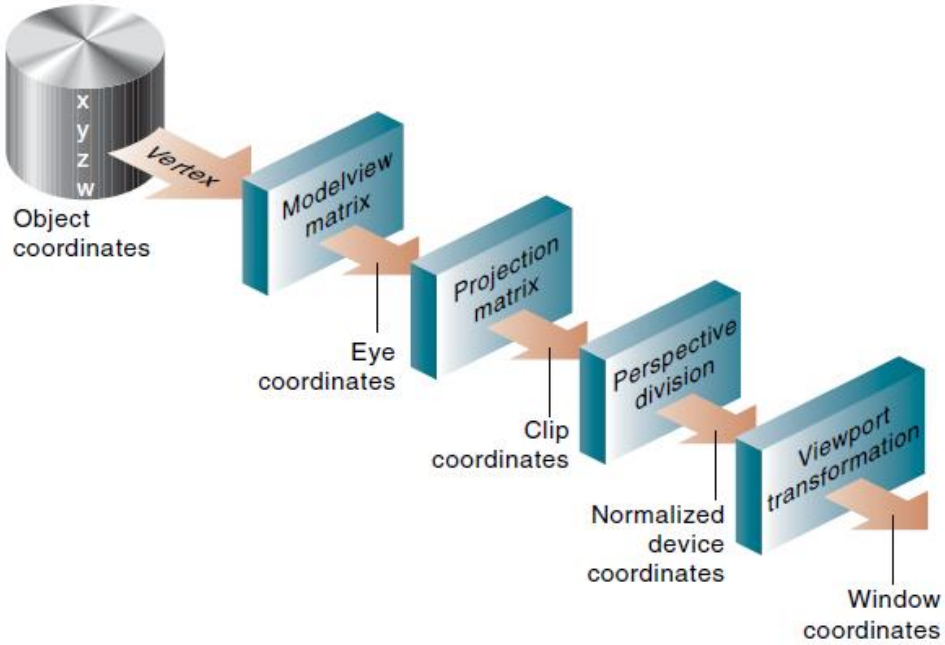
- 应用程序指定
  - 3D模型
  - 相机参数
    - 位置
    - 朝向
    - 焦距
    - 视角




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

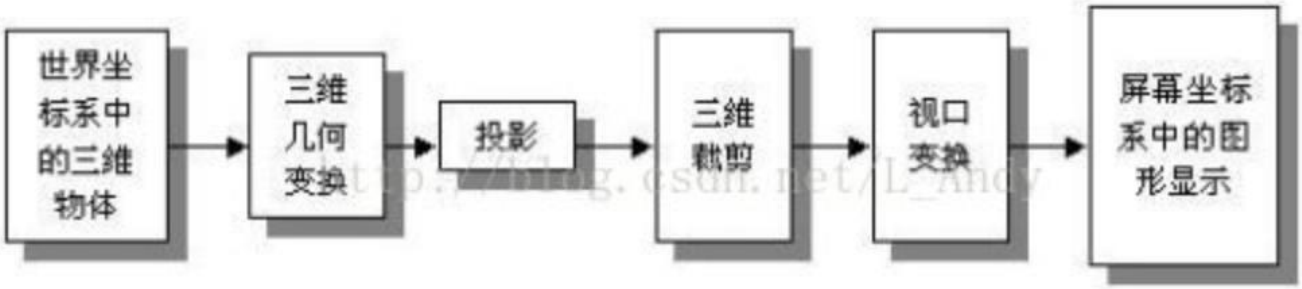
各种坐标系

- 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

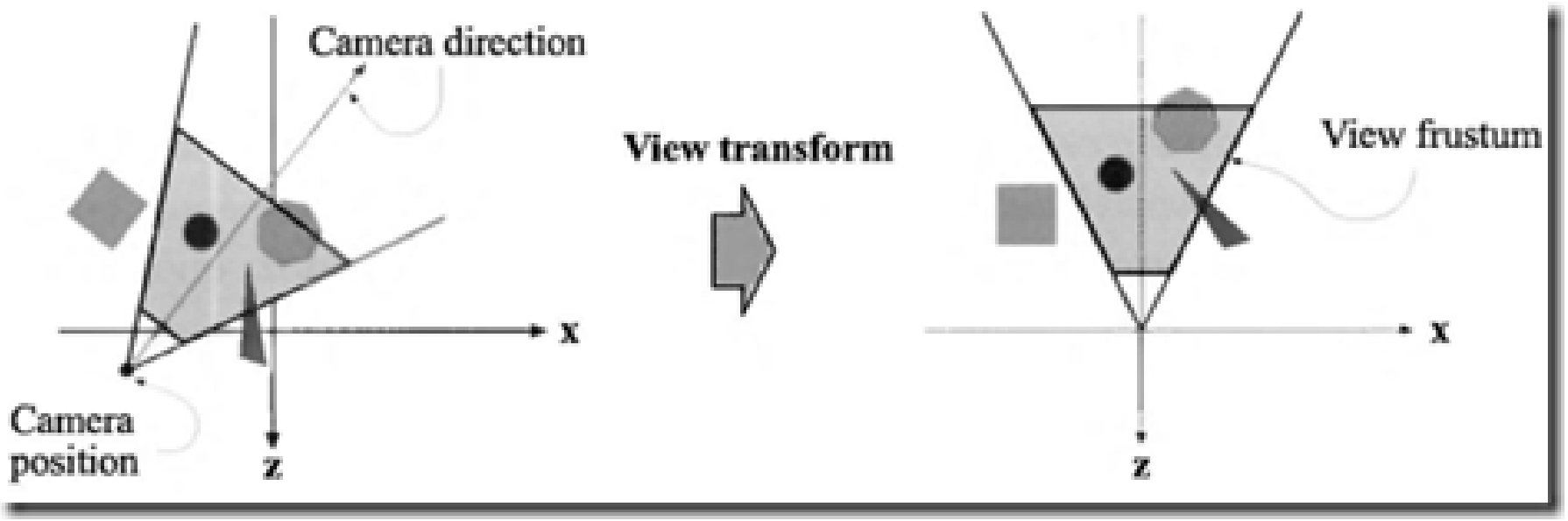


顶点处理

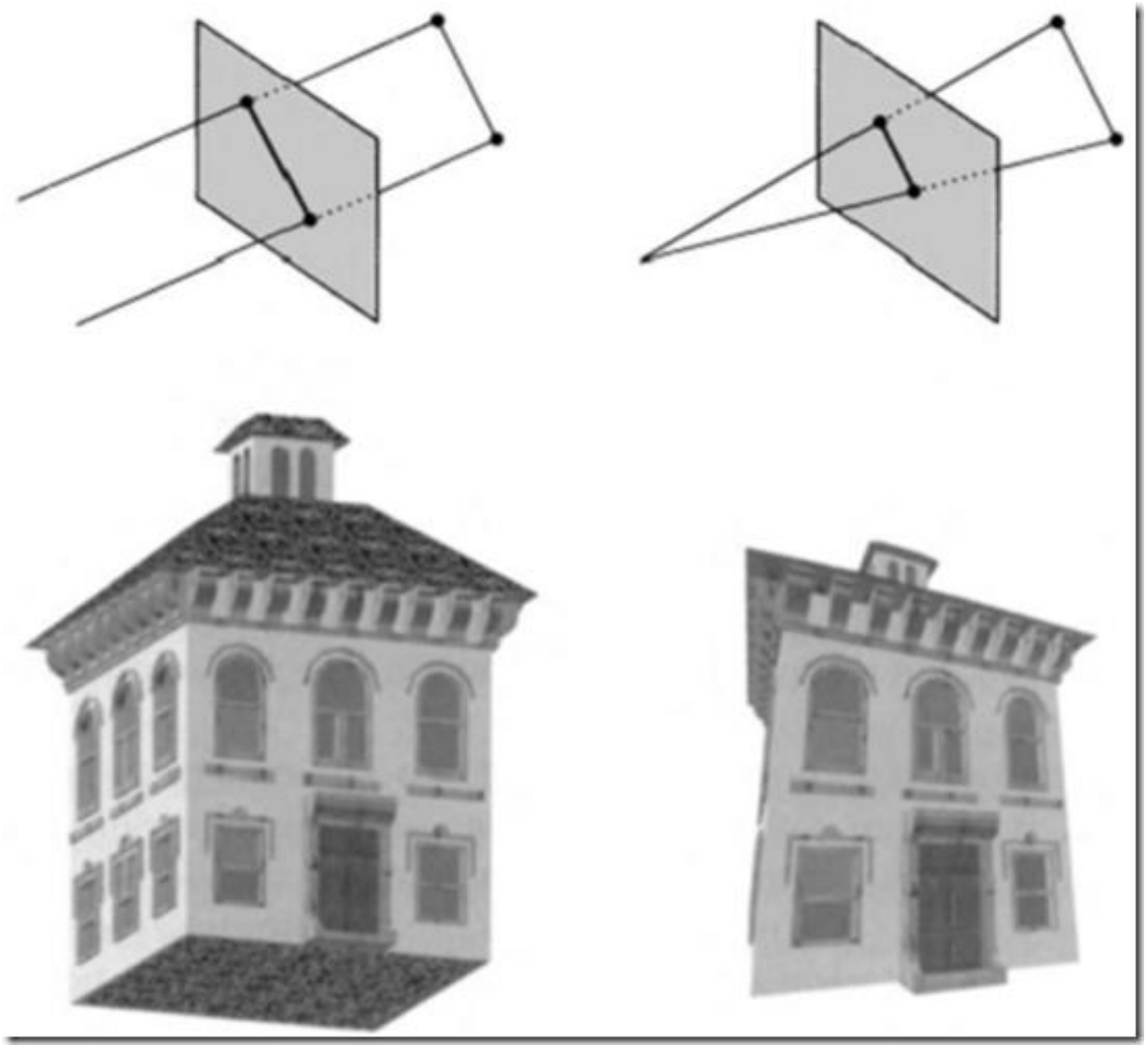
- 流水线中大部分工作是把对象在一个坐标系中表示转化为另一坐标系中的表示：
  - 世界坐标系
  - 照相机(眼睛)坐标系
  - 屏幕坐标系
- 坐标的每个变换相当于一次矩阵乘法
  - 最终的顶点变换为多个矩阵的乘法：MVP
  - 窗口变换



模型及视图变换

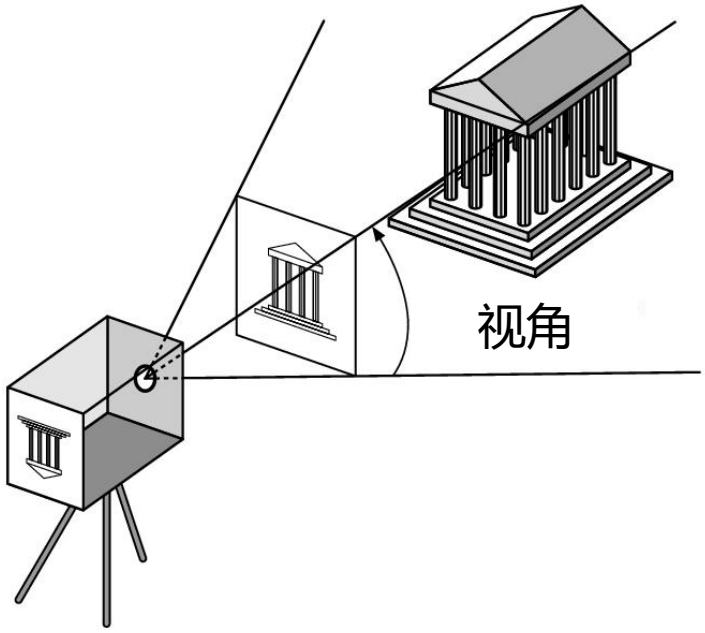

Camera direction
x
Camera
position
z
View transform
View frustum
x
z

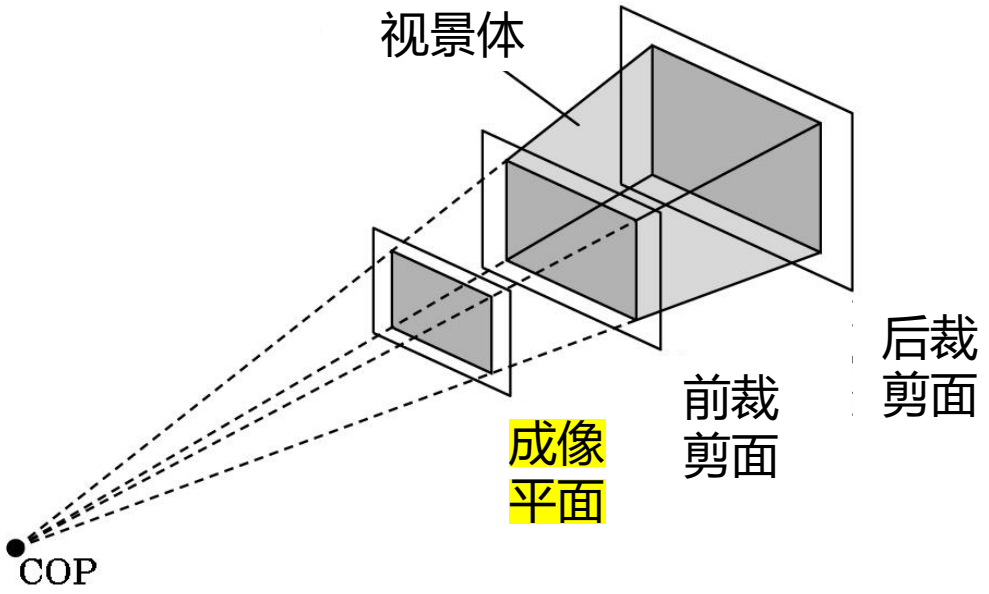
投影变换



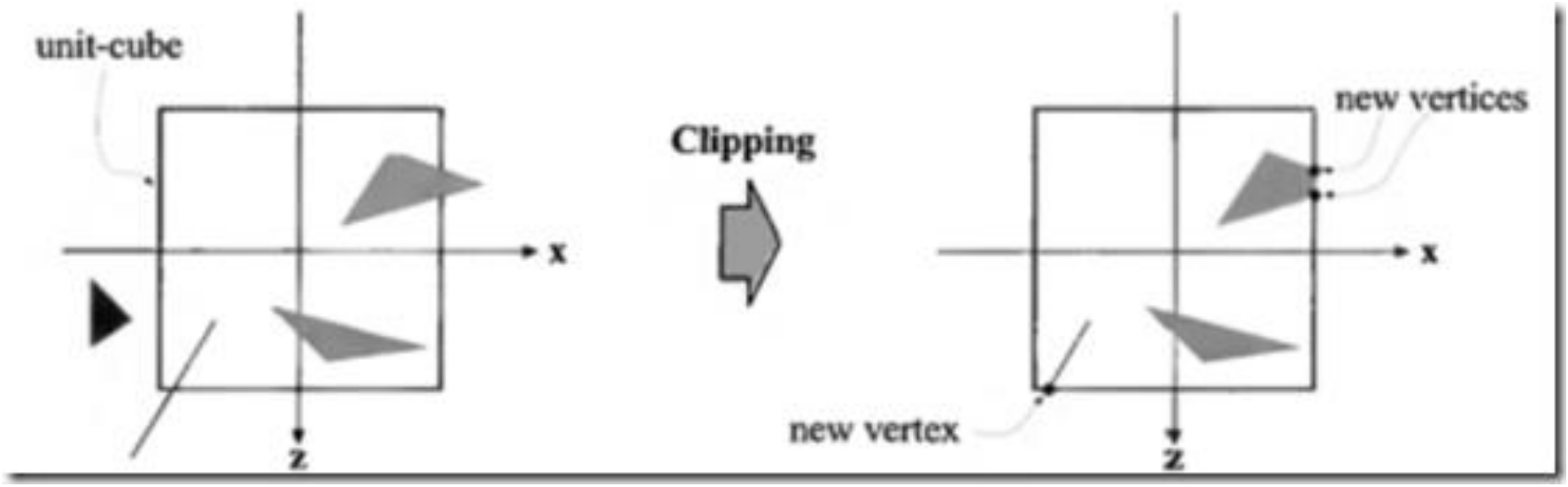
裁剪：视景体裁剪

- 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

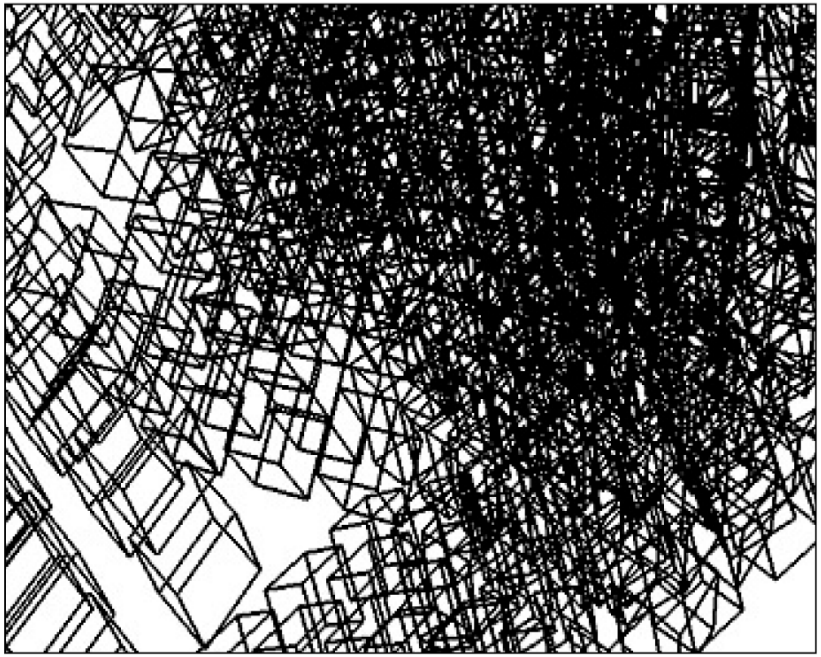


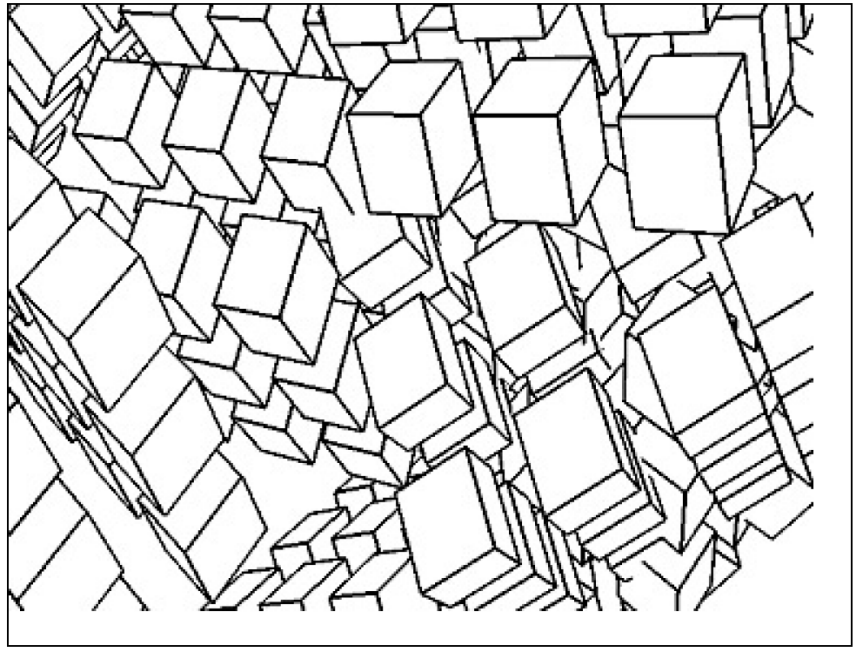


裁剪：窗口裁剪

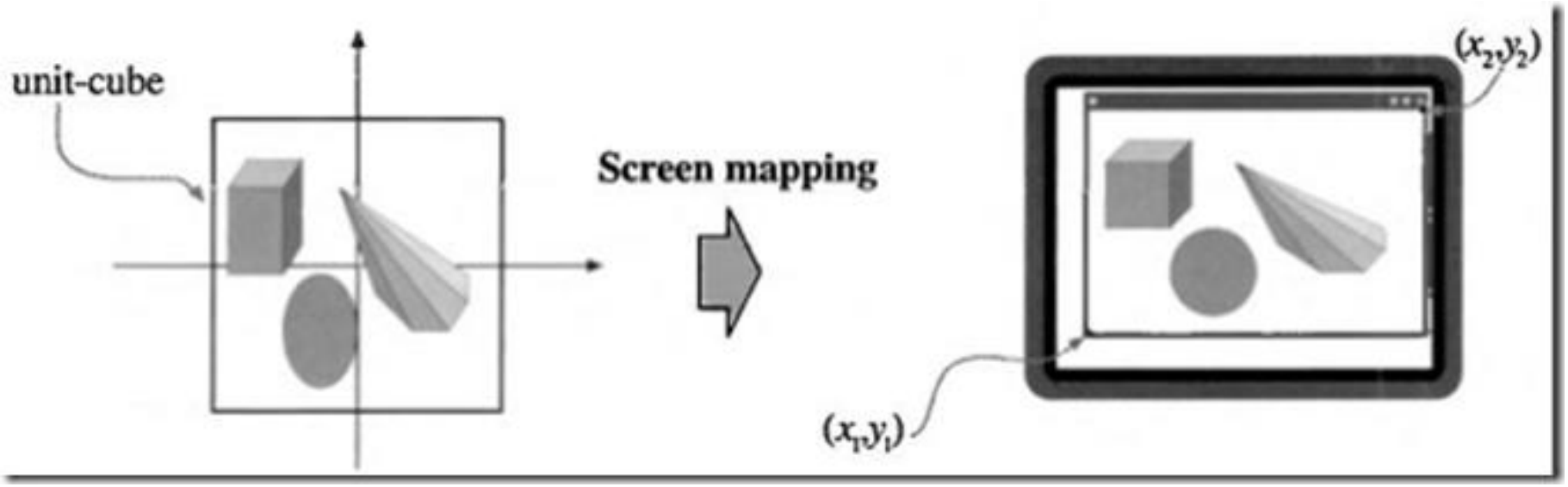


消隐：消除隐藏面





屏幕映射变换

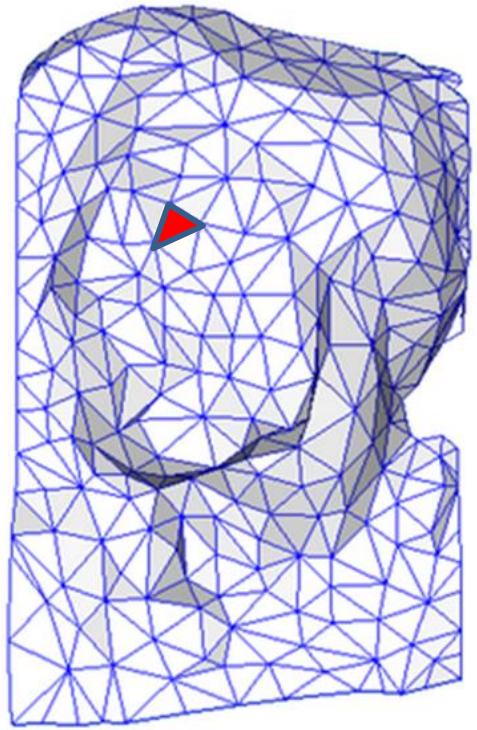


光栅化：找到所有片元

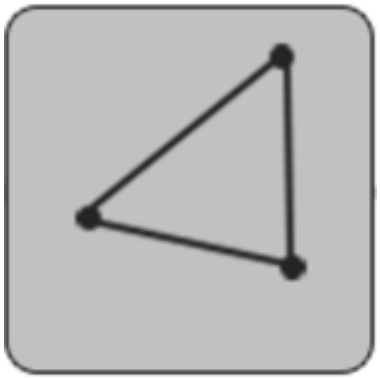
- 如果一个对象不被裁掉，那么在帧缓冲区中相应的像素就必须被赋予颜色
- 光栅化程序为每个对象生成一组片段
- 片段是“潜在的像素”
  - 在帧缓冲区中有一个位置
  - 具有颜色和深度属性
- 光栅化程序在对象上对顶点属性进行插值（重心坐标）

过程1：几何变换

逐顶点计算屏幕坐标



顶点变换



光栅化

R^3

R^2

R^2

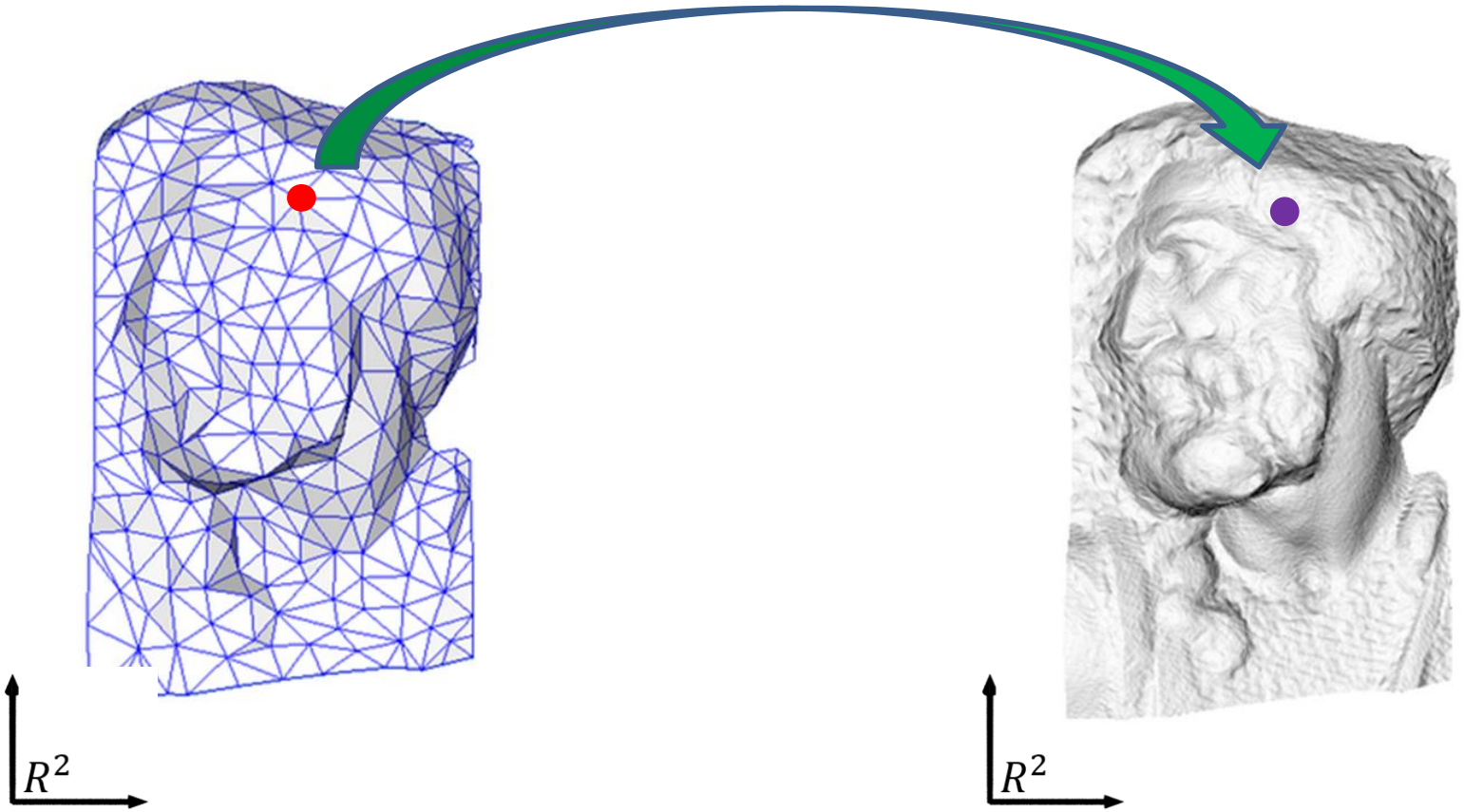
三维模型

可见图元

可见片元

过程2：像素着色

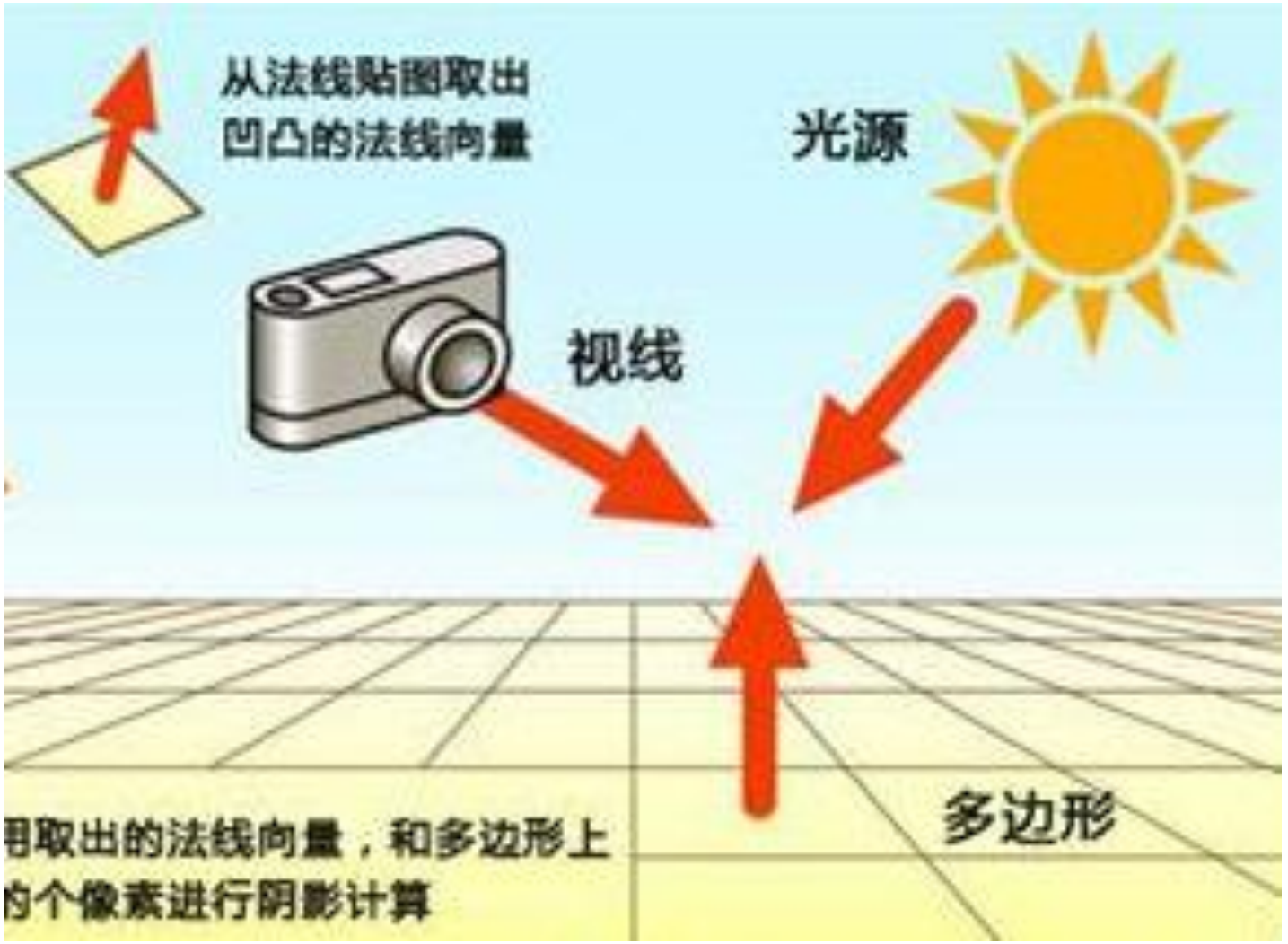
逐片元计算颜色



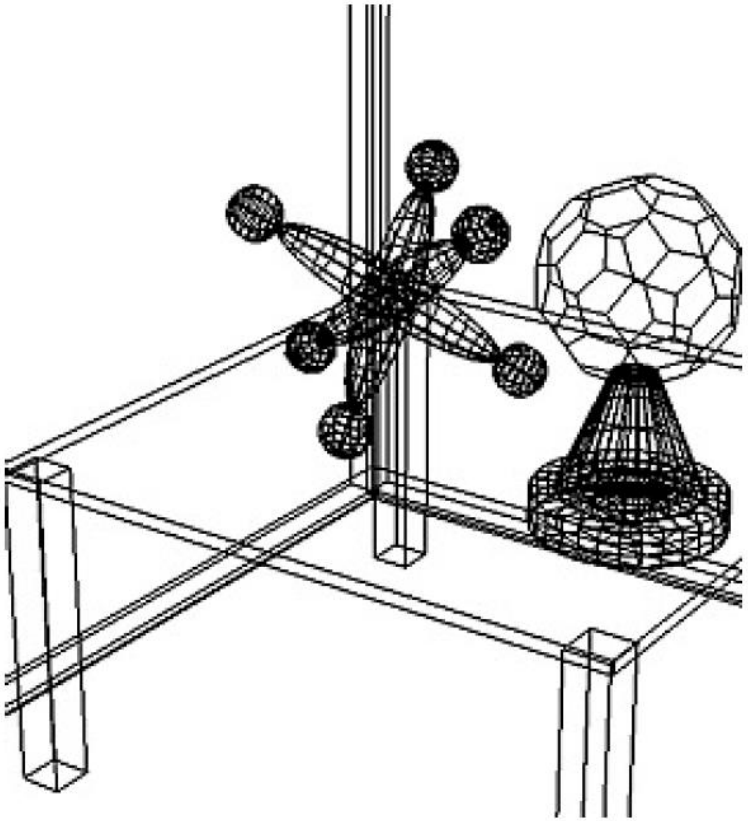
着色：让3D物体更有空间感

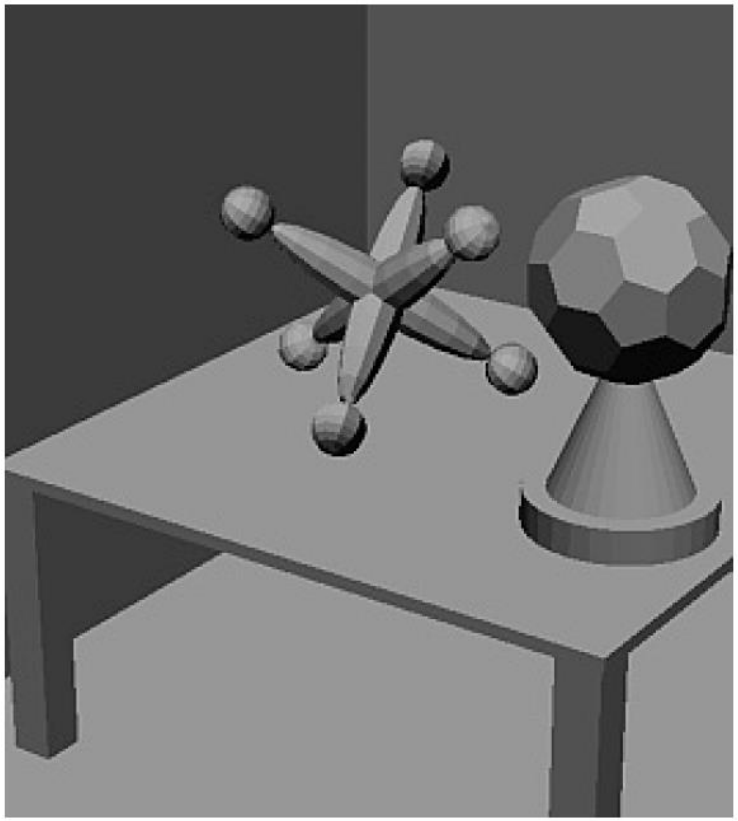
- 光：产生物体的不同部分的明暗变化
- 需模拟光对顶点的明暗（颜色）计算

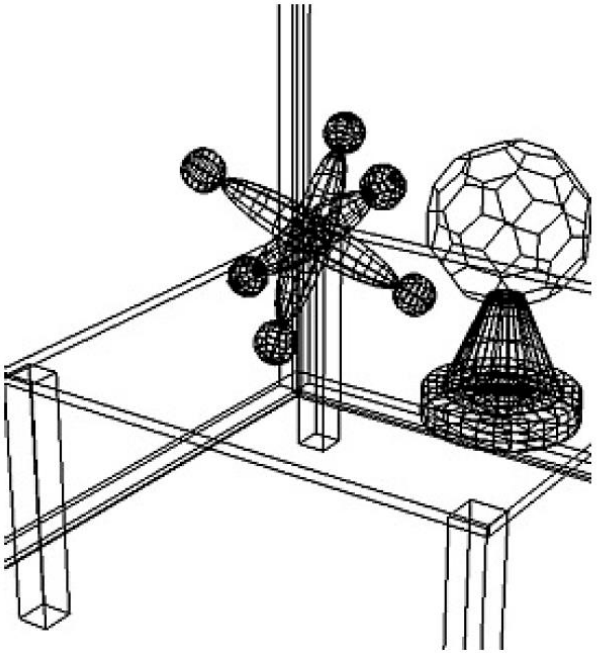
着色：光的物理性质及计算

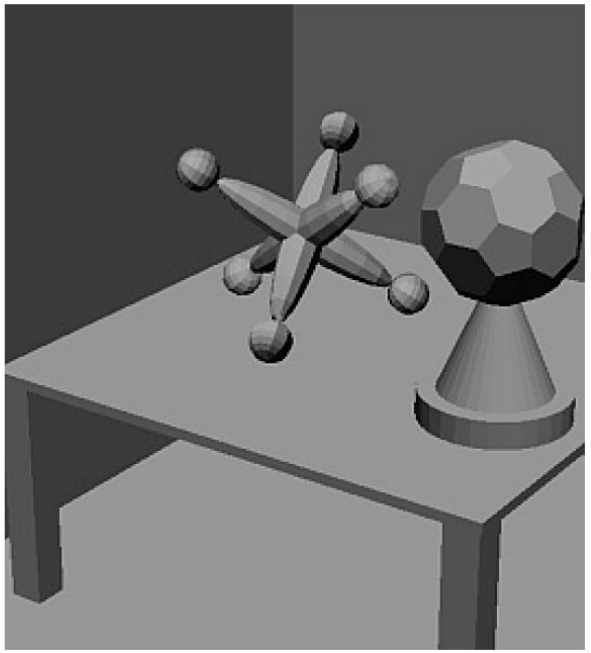


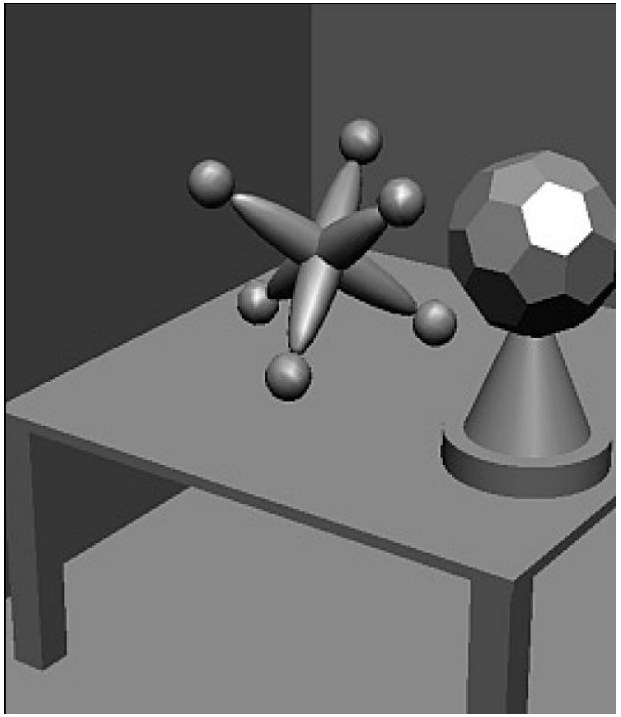
片元着色

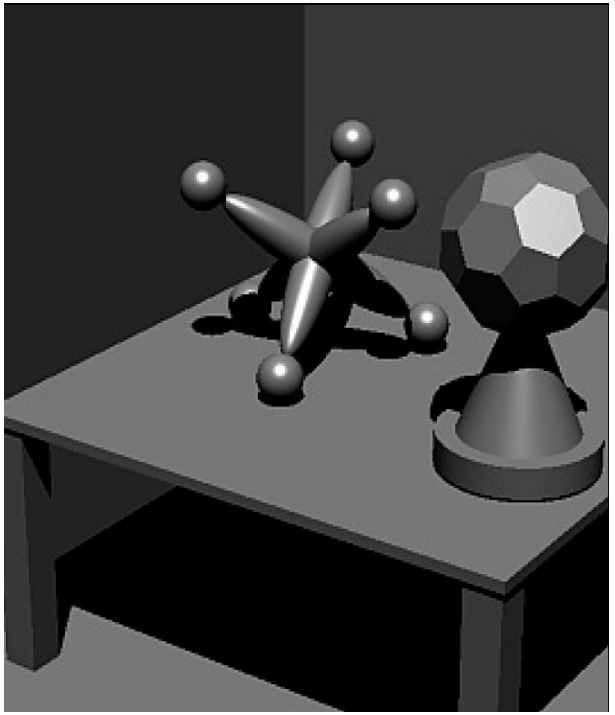


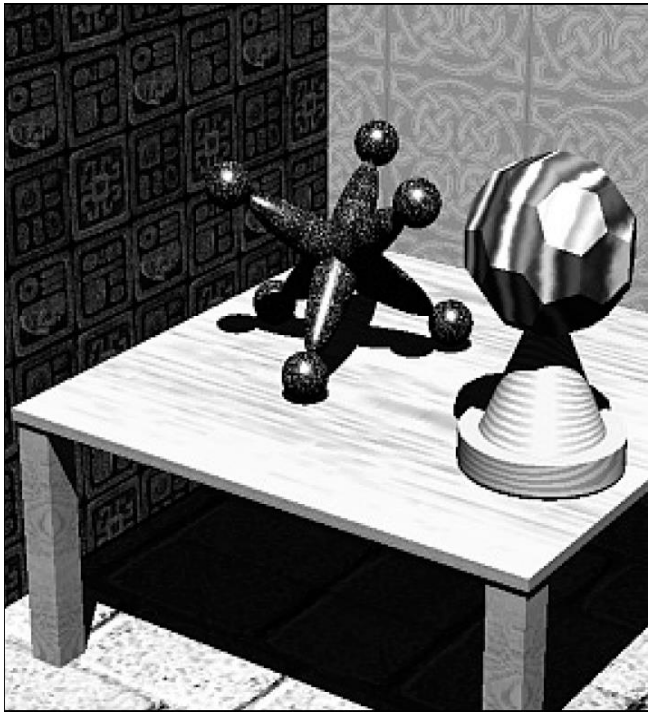






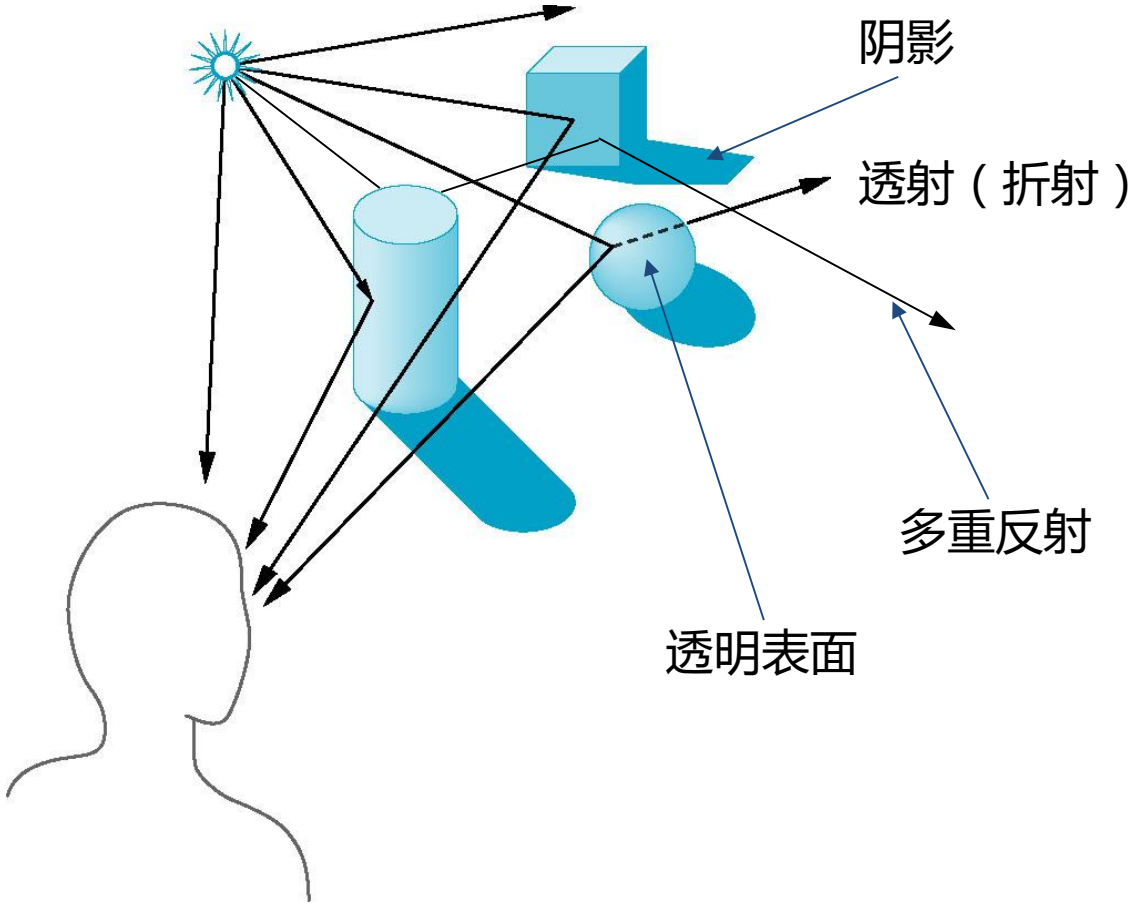






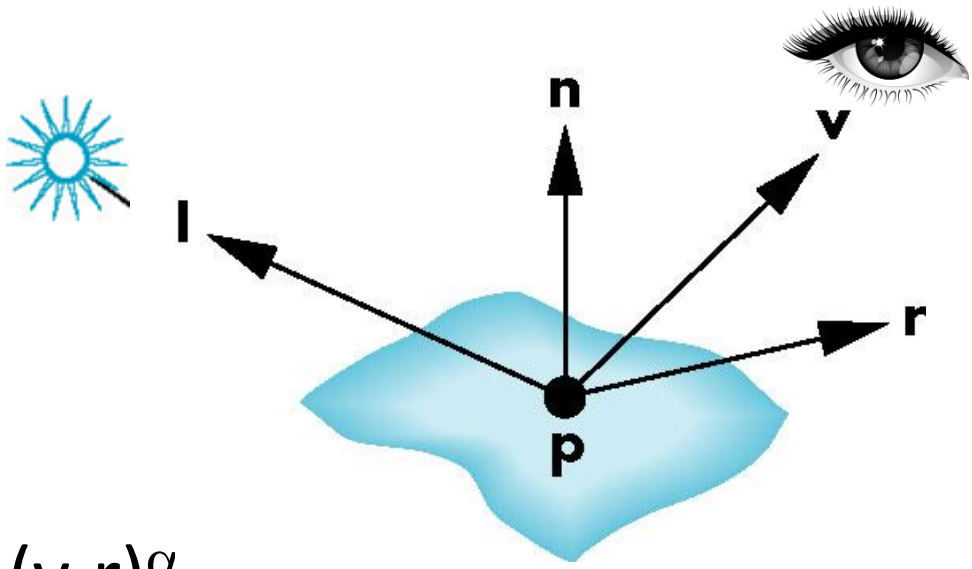
光照模型

- 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

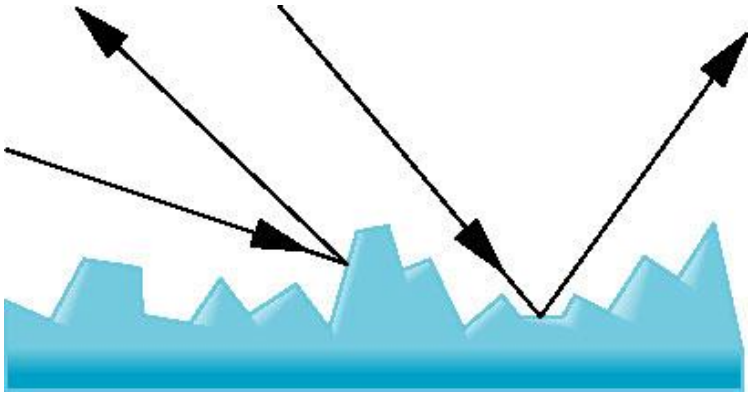
- 3个主要部分
  - 环境光 (Ambient)
  - 漫反射 (Diffuse)
  - 镜面反射 (Specular)



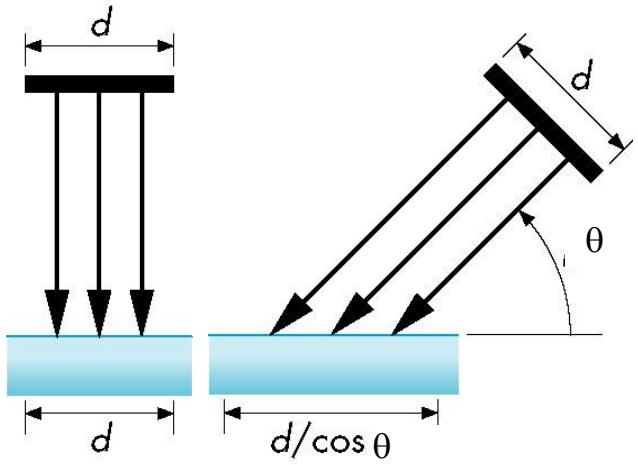
- $I = k_a I_a + k_d I_d \, l \cdot n + k_s I_s (v \cdot r)^{\alpha}$

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 ~ $\cos\theta_I$
  - $I_d = k_d I_l \cos\theta_i$



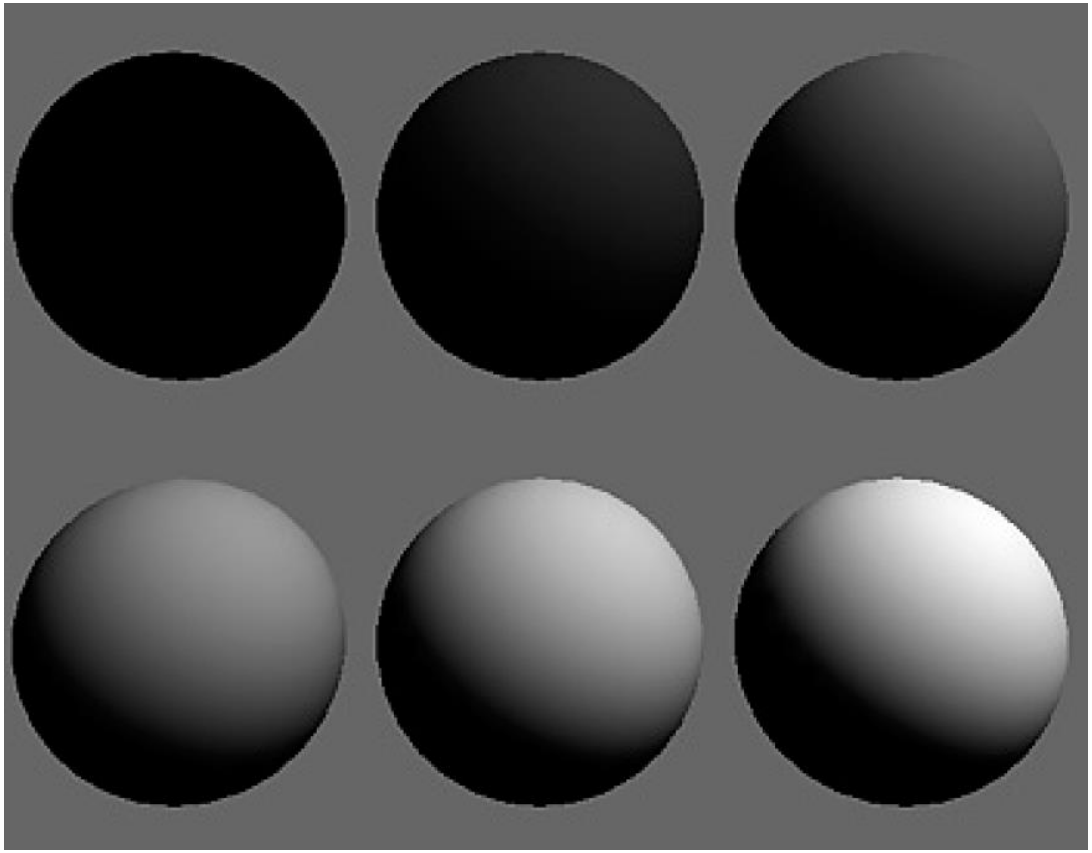
Lambertian曲面



漫反射参数的影响

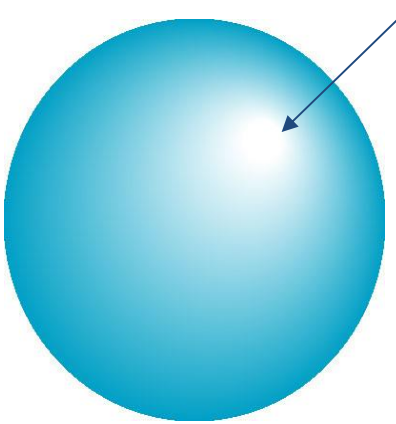
漫反射系数依次为
0.0, 0.2, 0.4, 0.6,
0.8, 1.0

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



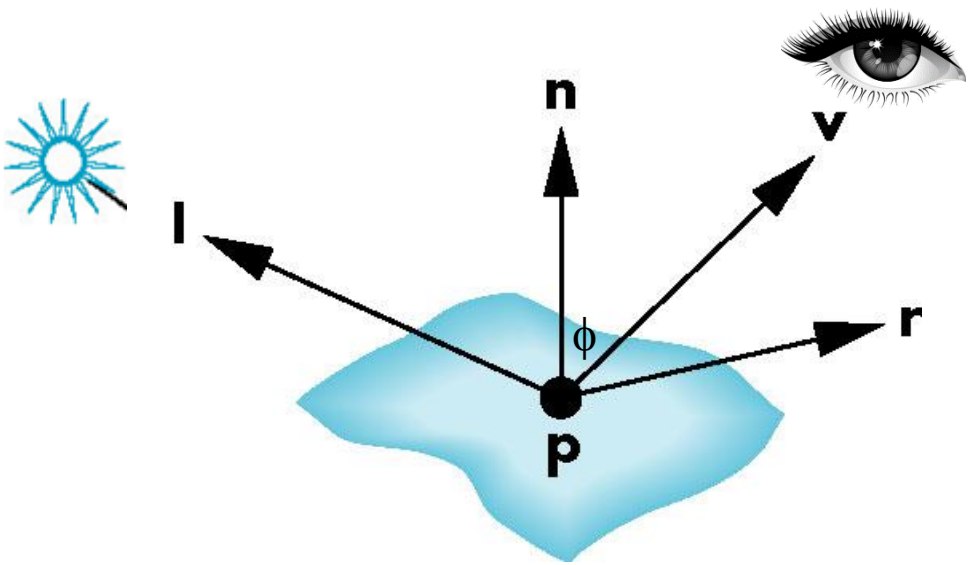
n: 法向量

l: 入射光方向

r: 反射方向

v: 视点方向

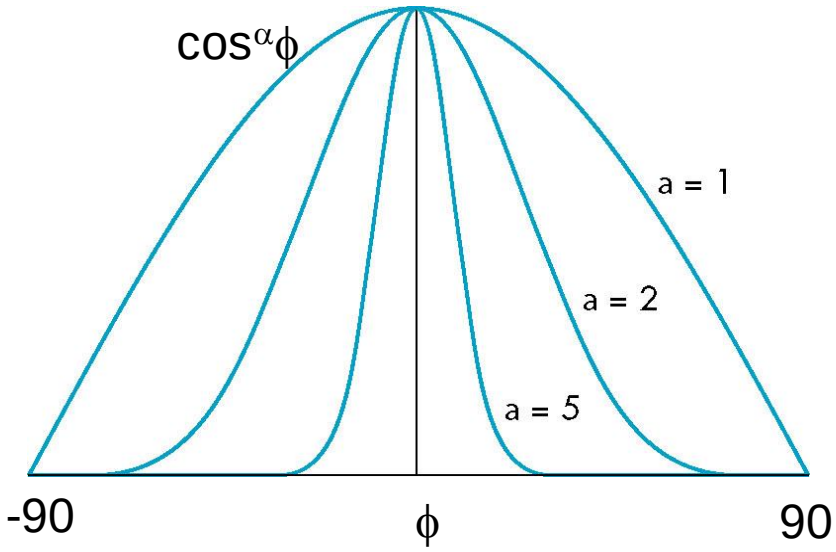
ϕ: r与v的夹角



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

高光系数

- 如果α的值介于100到200之间，那么对应于金属材料
- 如果α的值介于5到10之间，材料类似于塑料

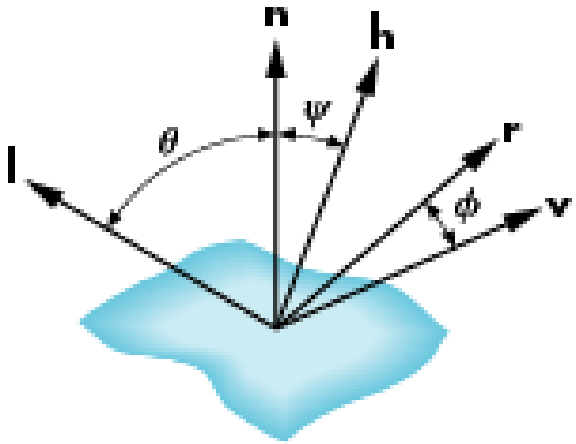


镜面反射的Blinn-Phong修正计算

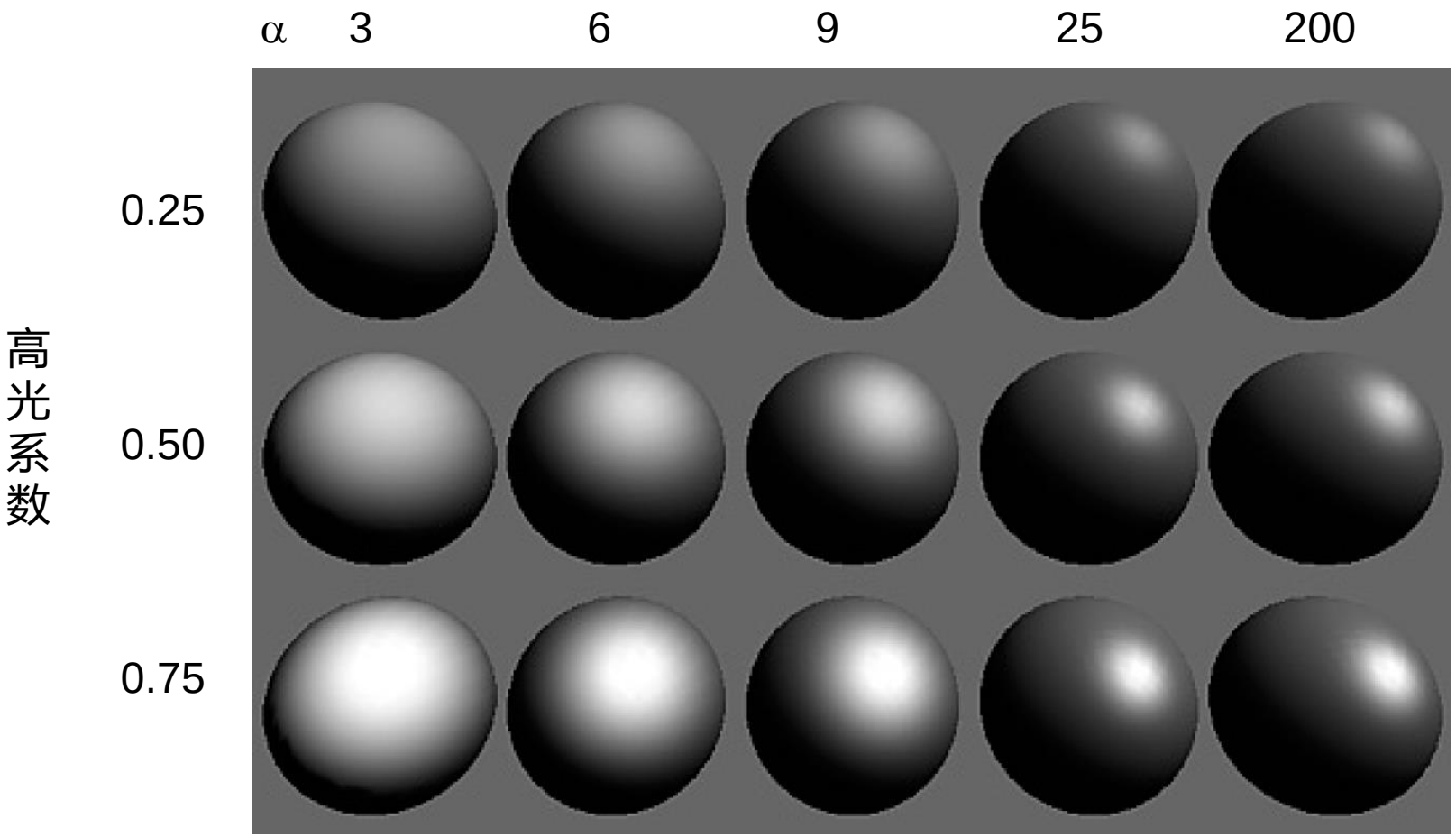
- Blinn利用中分向量给出了一个近似算法，减少计算良、提高计算效率
- h是l和v的平分单位向量，即

$$h = (l+v)/|l+v|$$

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



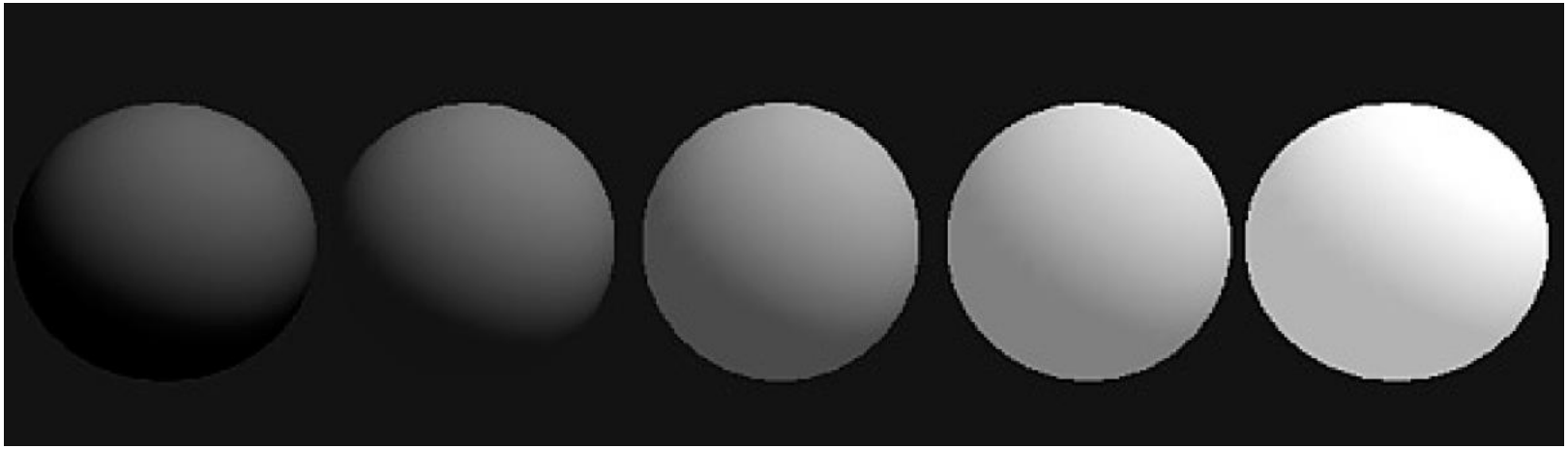
镜面反射参数的影响



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



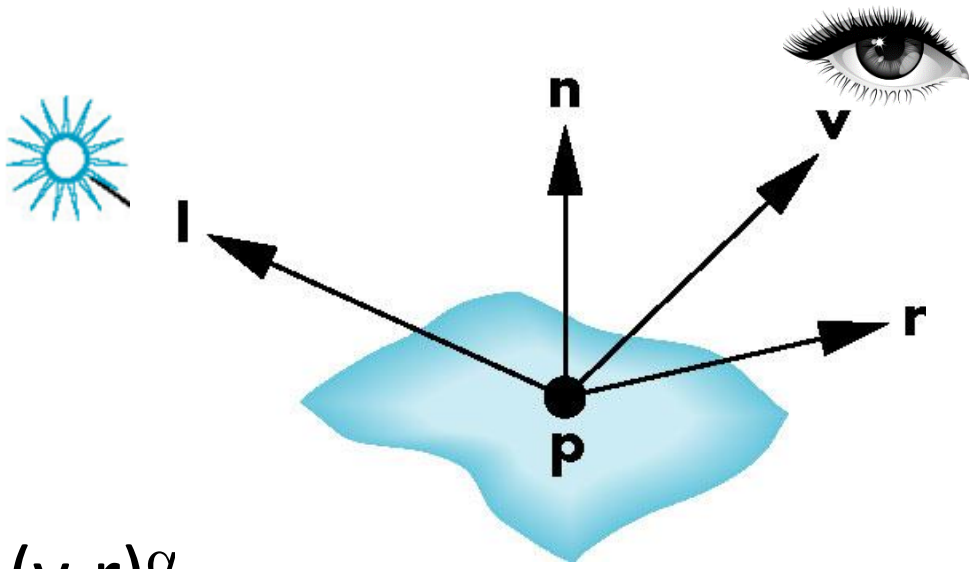
向漫反射光中加入不同的环境光的效果

两种光强都是1.0, 漫反射系数为0.04

环境光反射系数依次为0.0, 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)

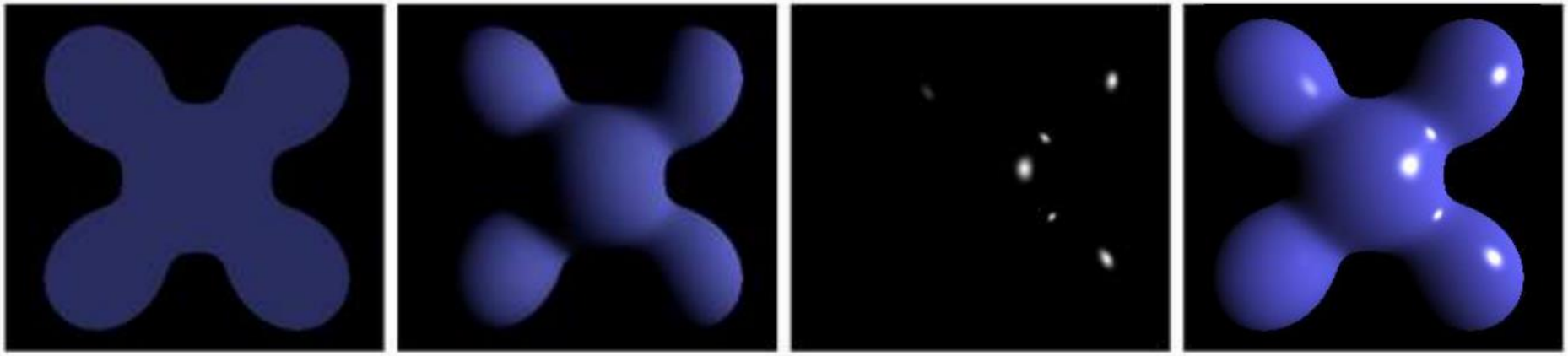


- $I = k_a I_a + k_d I_d \, l \cdot n + k_s I_s (v \cdot r)^{\alpha}$

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

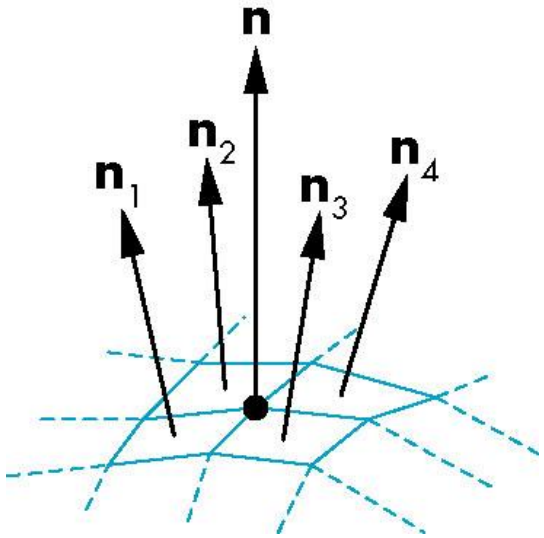
着色结果



Ambient + Diffuse + Specular = Blinn-Phong Shading

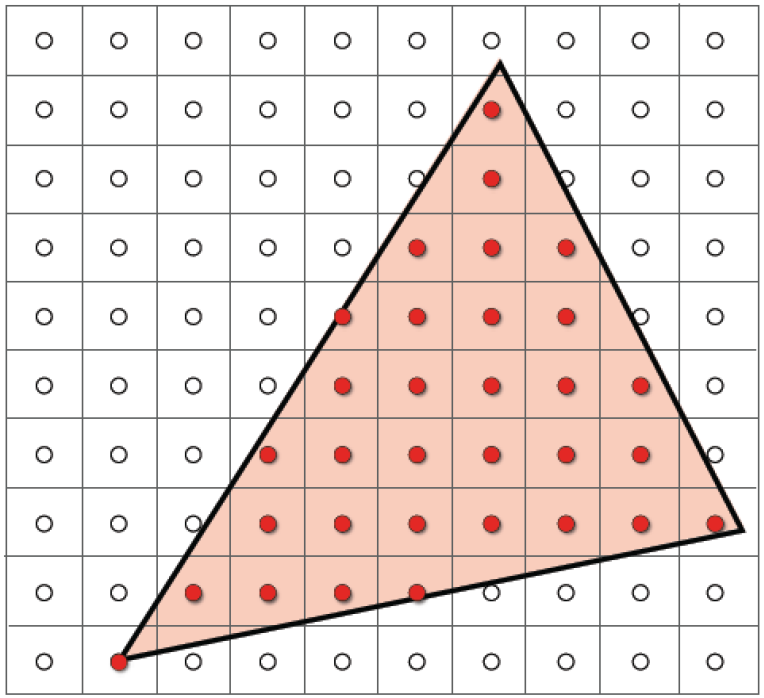
顶点的着色

- 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

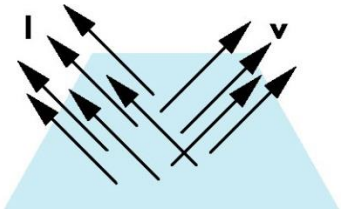
非顶点的像素的颜色？

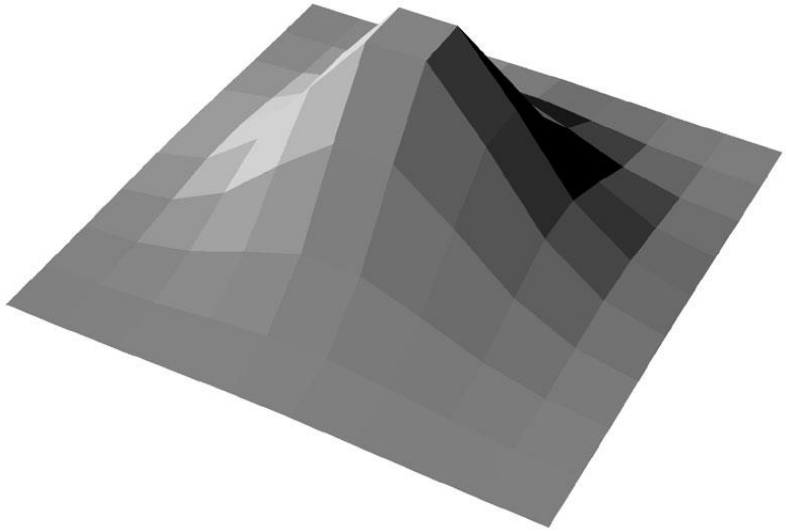


由顶点处的信息插值得到！

1. Flat Shading (Shade each triangle)

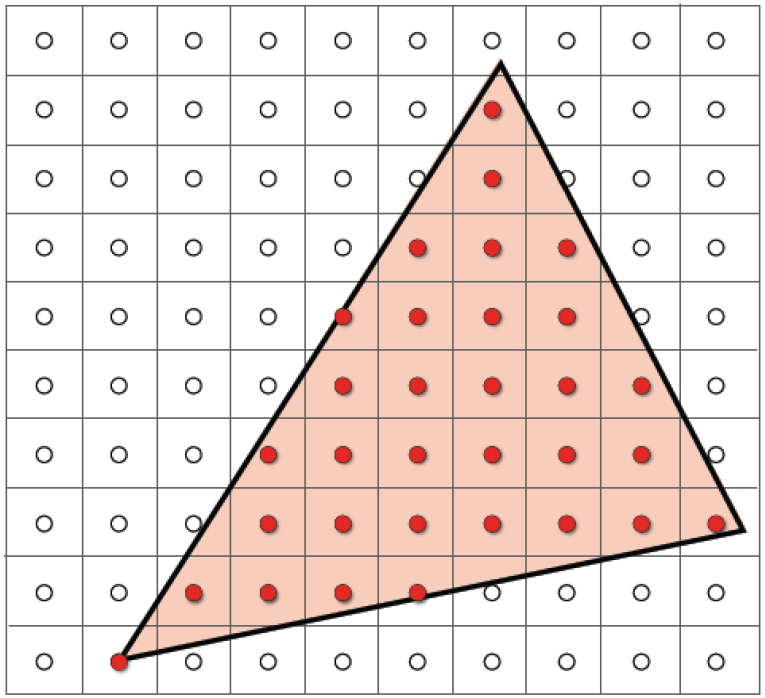
- 每个三角形中的像素的法向都一样（三角形的法向）
- 相当于：视点在无穷远，光源在无穷远
  - 视点方向v和入射方向l都是常量





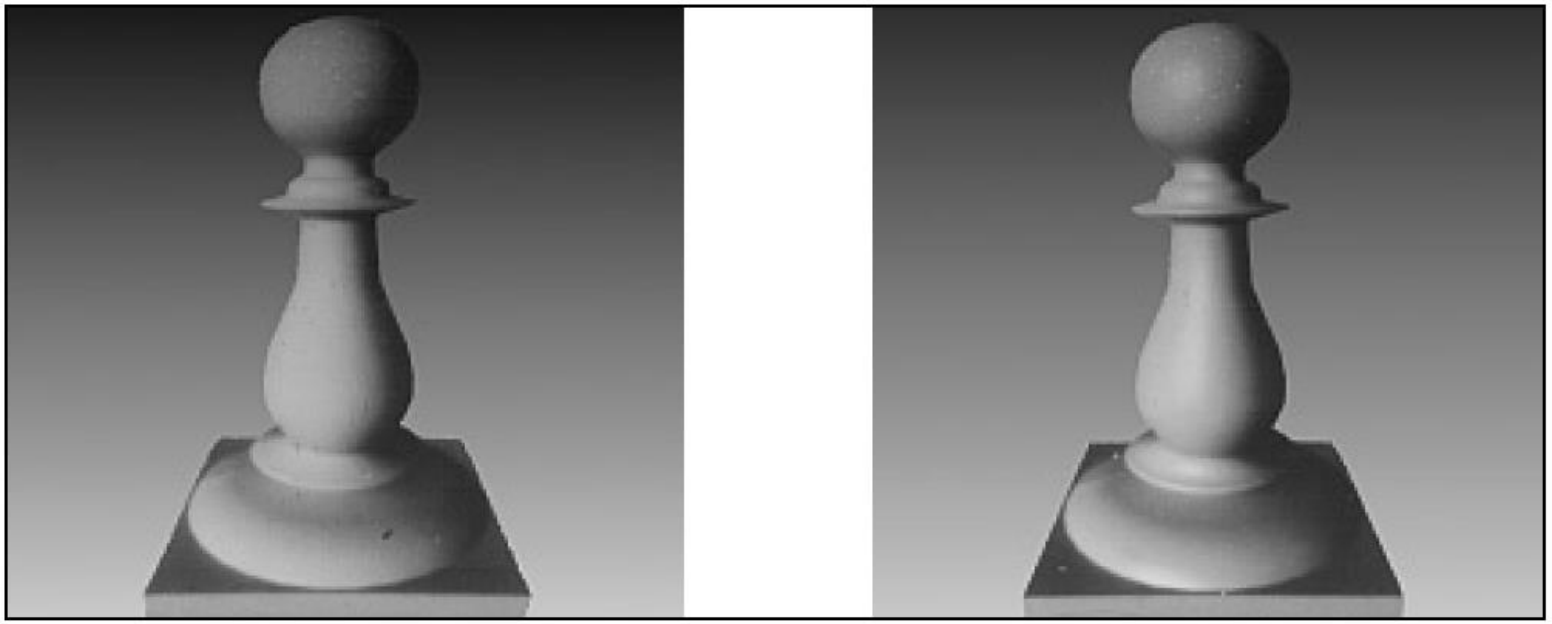
2. Gouraud Shading (Shade each vertex)

- 有3个顶点的颜色插值得到像素的颜色
- OpenGL 提供的方法



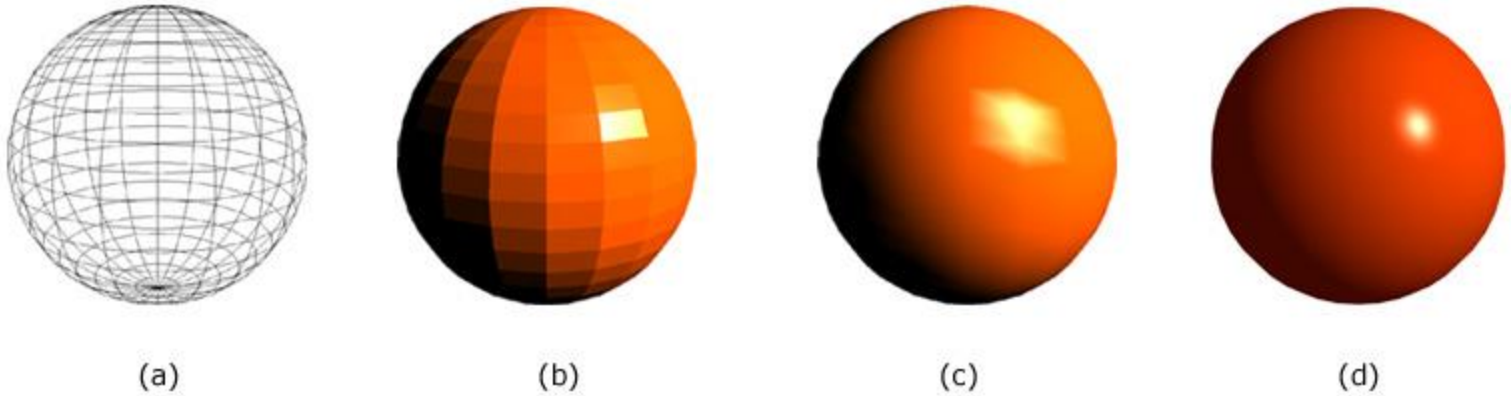
3. Phong Shading (Shade each pixel)

- 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)

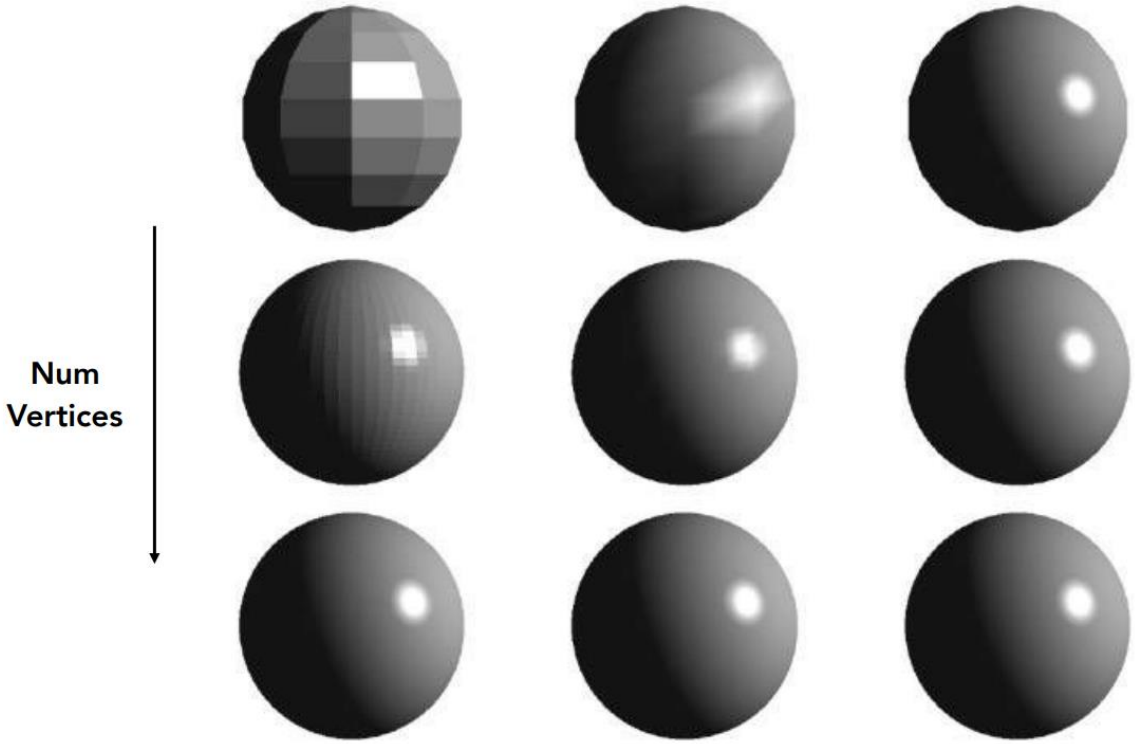
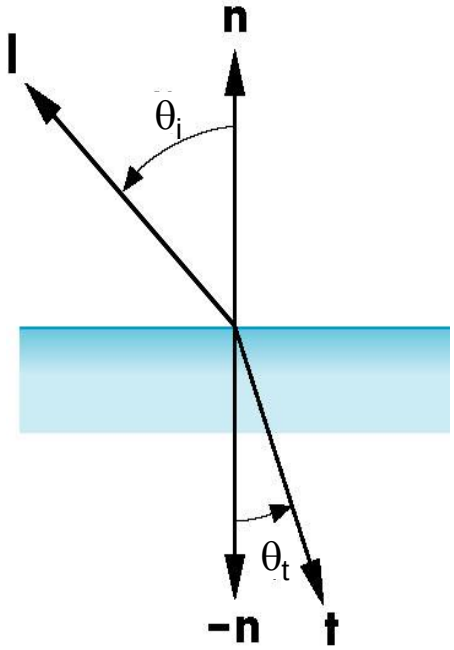


Image credit: Happyman, http://cg2010studio.com/

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
  - ...

图像渲染API

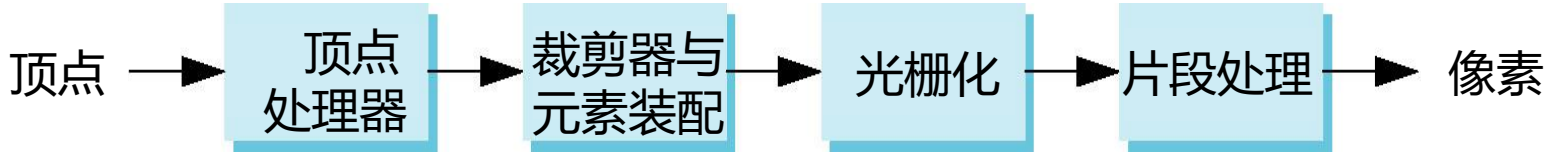
Application Program Interfaces

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，更新快
- Vulkan: next generation of OpenGL

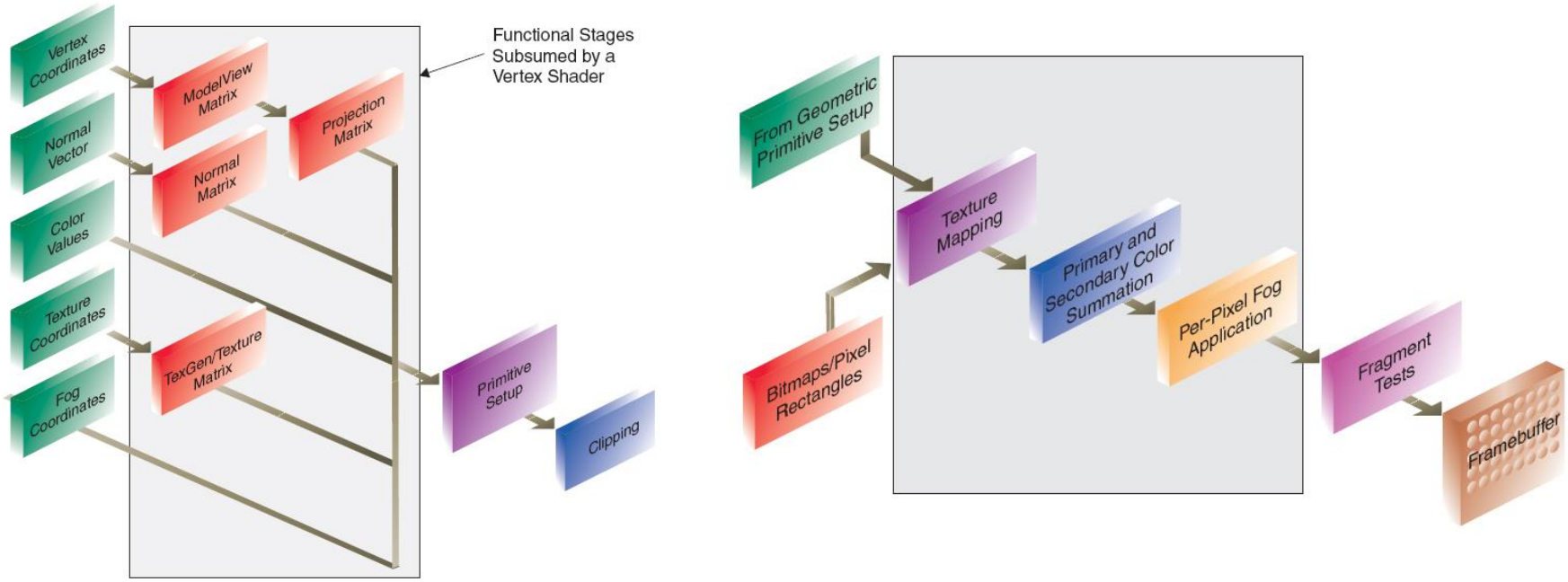
固定（不可编程）渲染管线

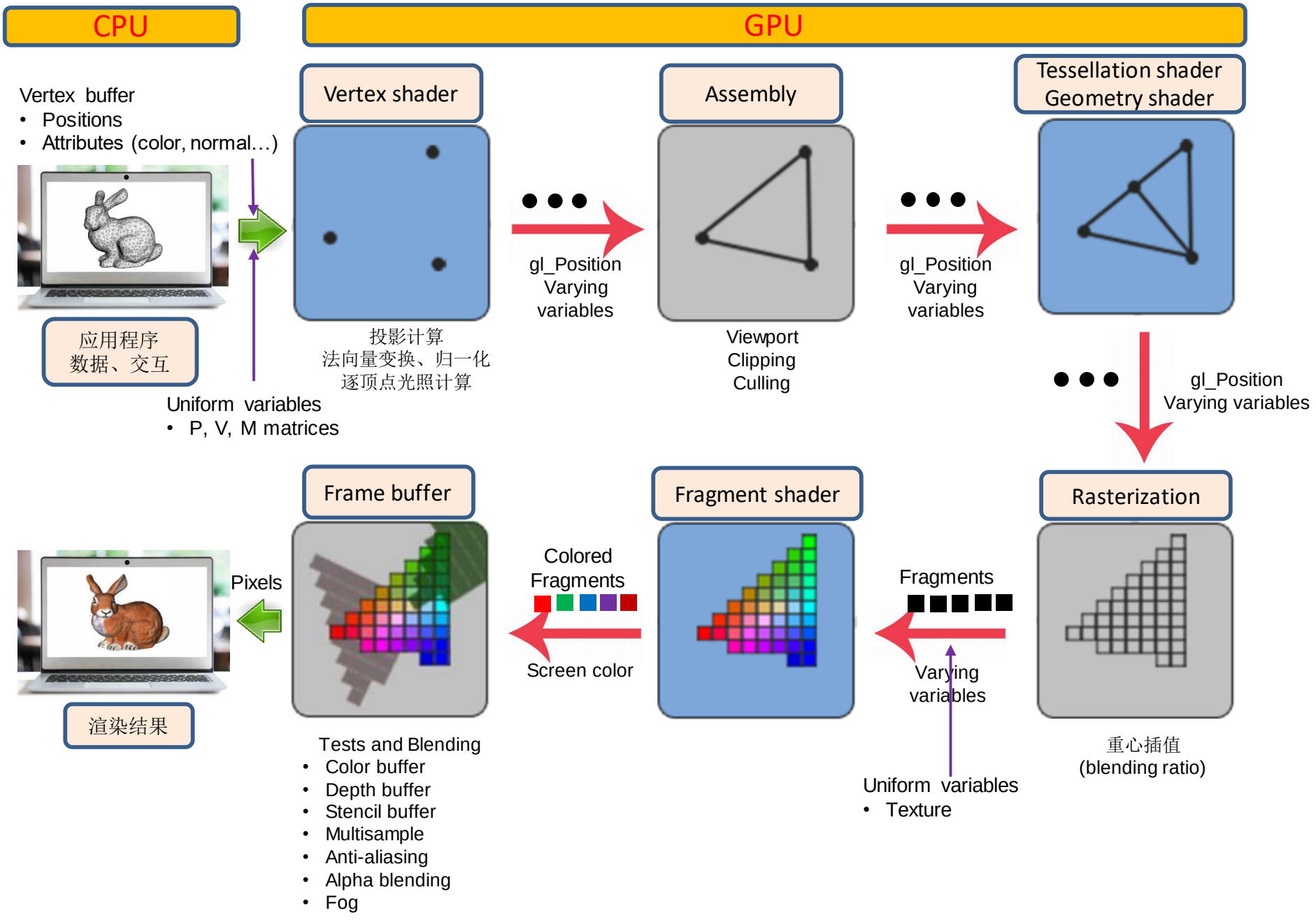
- 按照API提供的接口函数逐步操作、处理每个对象
- 优点：对初级用户使用方便
- 缺点：渲染效果一般，无法进行高级渲染
- 流水线体系



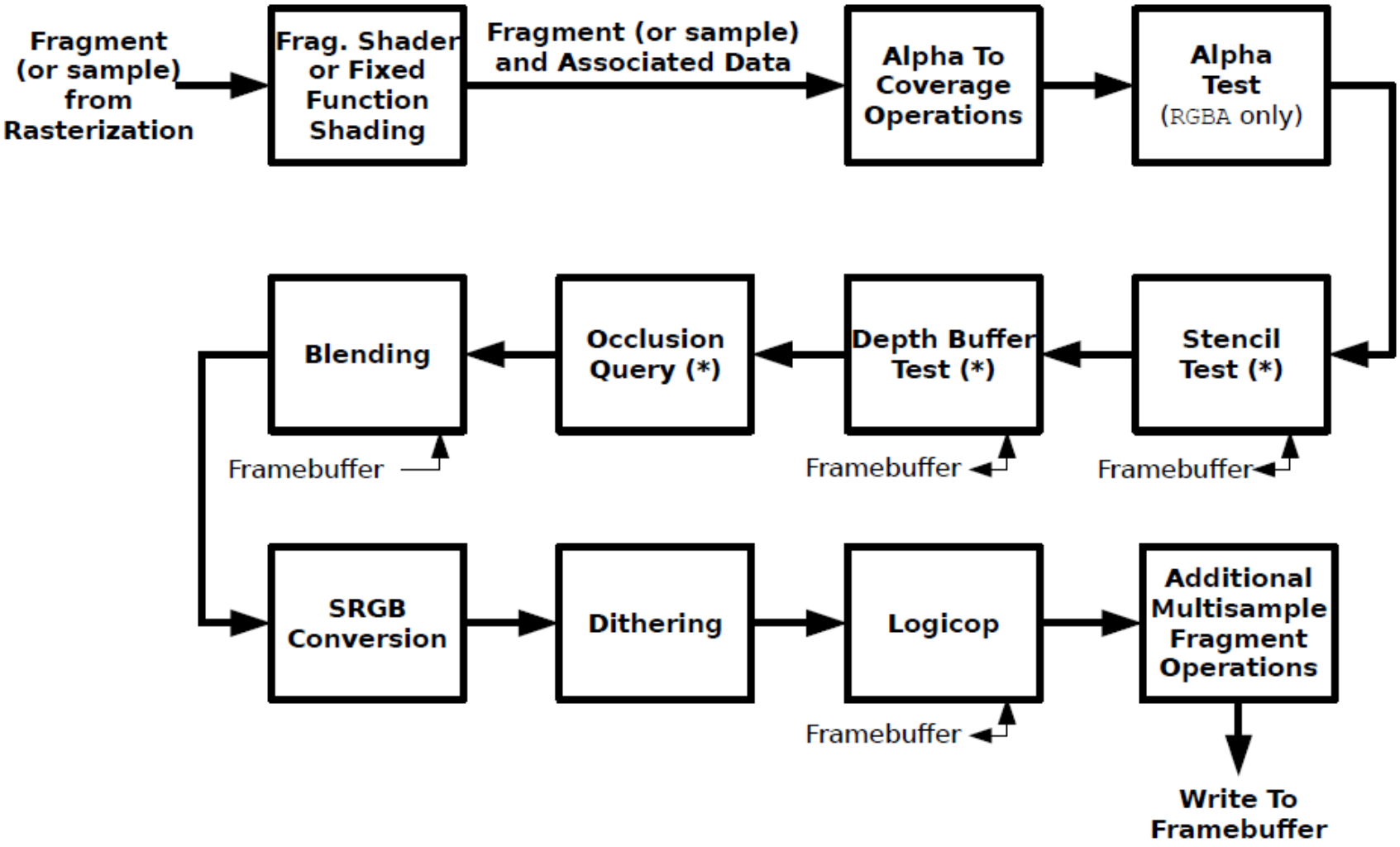
可编程渲染管线

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

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

GPU并行计算

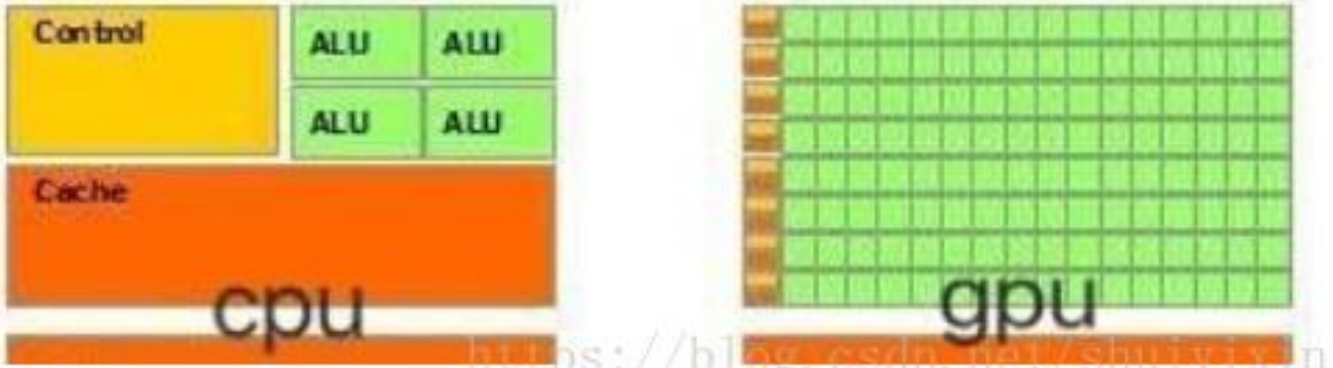
并行计算

- 互相独立（互不依赖）的计算可以并行
- 图形渲染：大量并行运算
  - 顶点变换的计算：逐顶点
  - 片元颜色的计算：逐像素
- 从图形加速卡到GPU

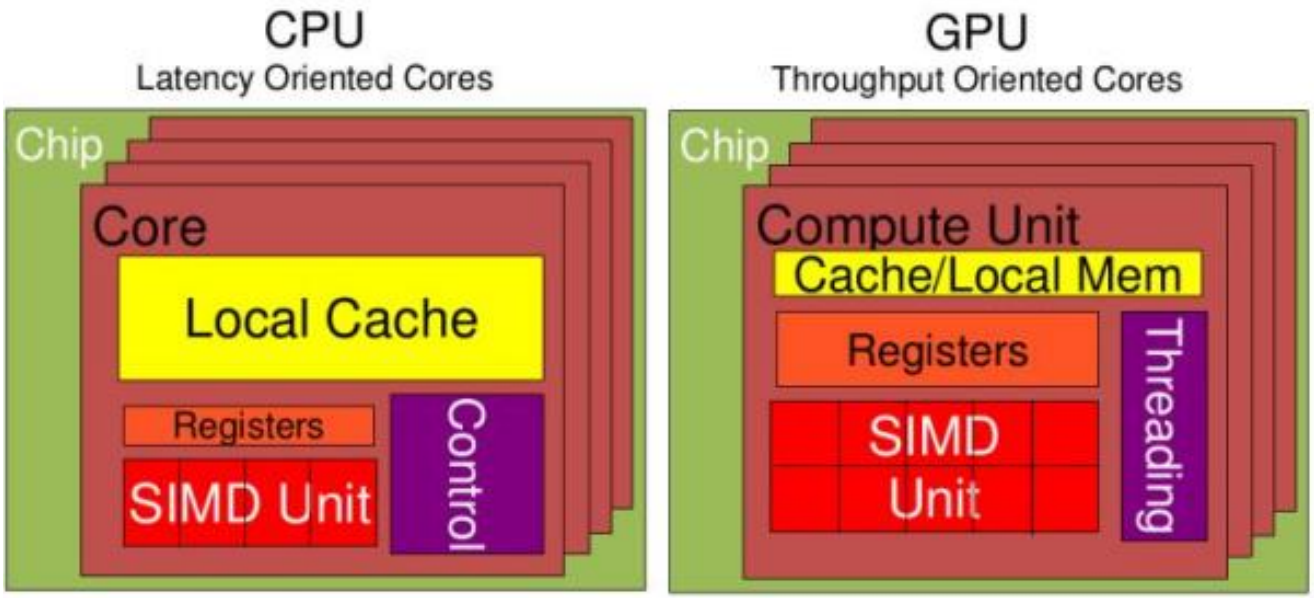
GPU: 基于大吞吐量的设计

- Graphic Processing Unit (GPU)
- 非常多的小的计算单元

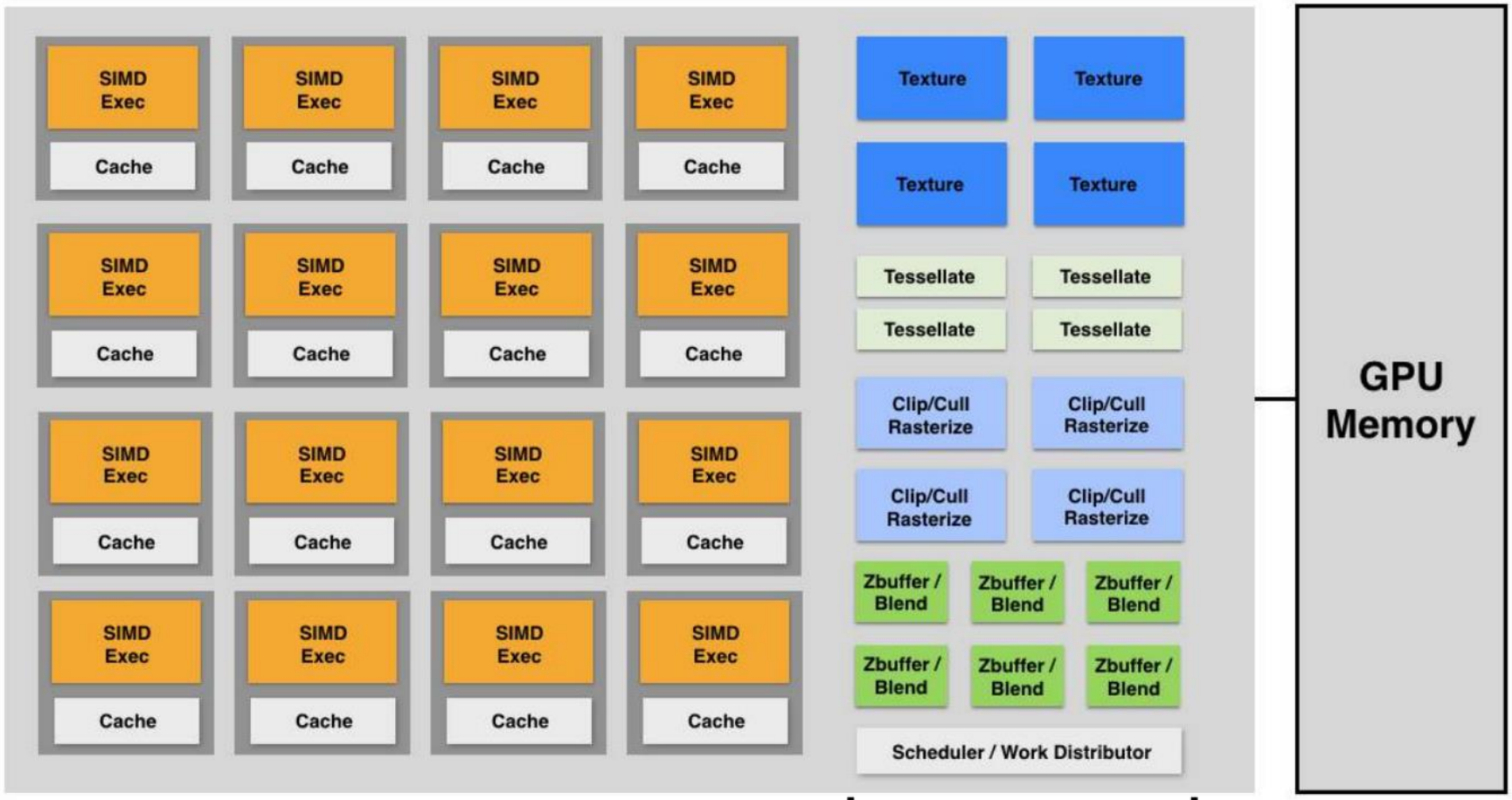
GPU vs. CPU



绿色：计算单元
橙红色：存储单元
橙黄色：控制单元



GPU: Heterogeneous, Multi-Core Processor



Credit: Lingqi Yan

CPU vs. GPU

- 设计架构不一样：目标不同
  - CPU: 复杂逻辑运算
  - GPU: 成百上千个计算核，无复杂逻辑
- CPU擅长
  - 逻辑控制和通用类型数据运算不同
- GPU擅长：大规模并发计算
  - 类型高度统一的、相互无依赖的大规模数据
  - 不需要被打断的纯净的计算环境

GPU vs CPU

GPU

CPU

GPGPU（通用GPU计算）

- 将GPU的高并行性能用于其他需要高密度计算的领域
- OpenGL利用纹理存储器在GPU中计算以及把结果取回内存，流程主要就是
  - 创建OpenGL的环境，接着创建FBO（帧缓存对象）、纹理、设置纹理参数、然后将纹理绑定到帧缓存对象，最后传输数据到纹理，接着用片段着色器对数据进行处理，最后就是取回数据
- OpenCL: Open Computing Language，开放运算语言

Q&A