



Lecture 22: 课程回顾

SSE315: 计算机图形学
Computer Graphics

陈壮彬

软件工程学院

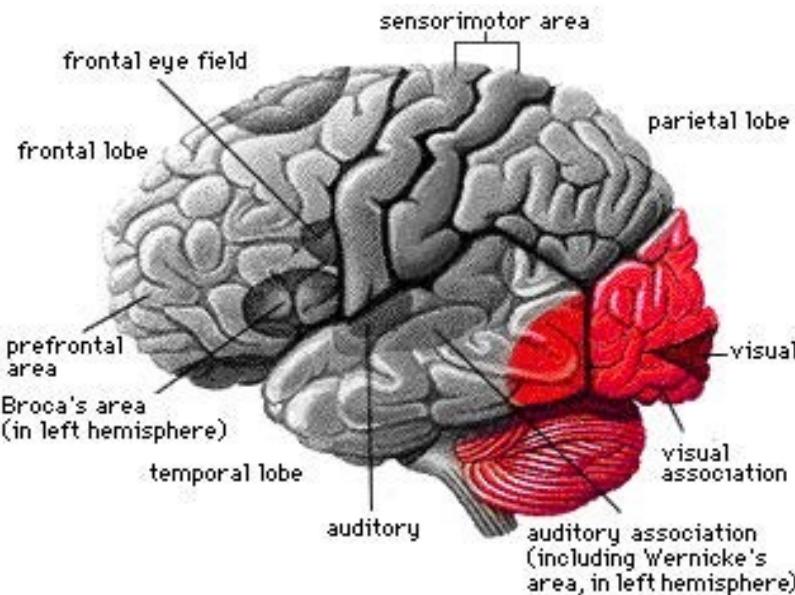
chenzhb36@mail.sysu.edu.cn

计算机图形学的第一印象



**什么是计算机图形学？
为什么我们需要它？**

Why virtual information?



大脑中有近 30% 的部分专门用于处理视觉信息！

眼睛是我们头部接收信息最快、最高效的"接口"

一图胜千言

MACINTOSH (1984)



APPLECOLOR HIGH-RESOLUTION RGB
AND MACINTOSH II (1987)



像素

2020: 8k monitor
7680x4320 (~95MB)



2020 virtual reality headset:
2x2160x2160 @ 90Hz => 2.3GB/s

苹果 Vision Pro
有 2300 万像素！

什么是计算机图形学? again...

com•put•er graph•ics /kəm'pyoodər 'grafiks/ *n.*
The use of computation to turn **digital information** into
sensory stimuli.

感官刺激

计算机图形学无处不在！

游戏 Games



王者荣耀

无主之地 3 (2019)



游戏 Games



黑神话：悟空

电影 Movies



阿凡达 2009 vs. 阿凡达 2022



动画 Animation



疯狂动物城
(2016)



冰雪公主 2
(2019)

设计 Design



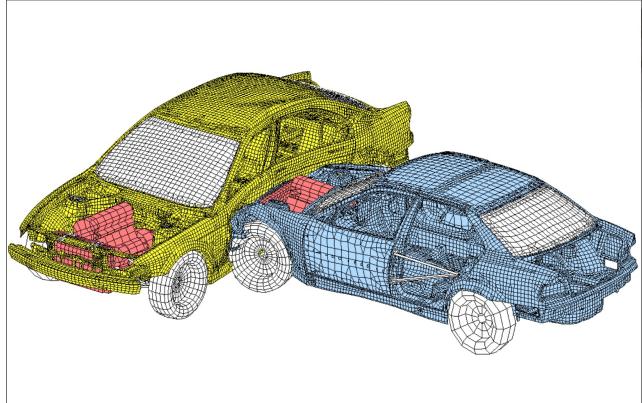
汽车建模



汽车模型



汽车碰撞
实验



设计 Design



宜家 Ikea - 75% 的展示图片都是渲染出来的

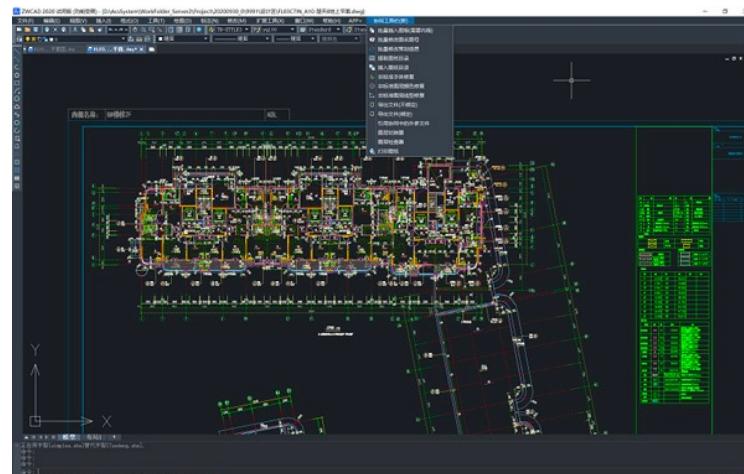
计算机辅助设计 CAD



SolidWorks



AutoCAD



中望 CAD

仿真 Simulation

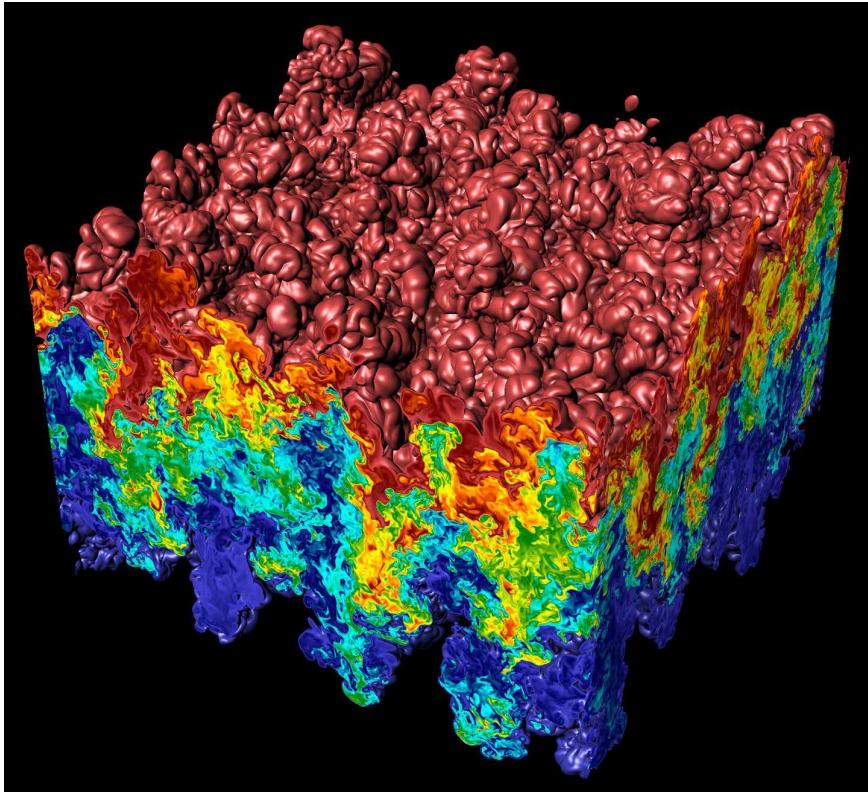


沙尘暴

《星际穿越》中的黑洞



科学/数学研究可视化

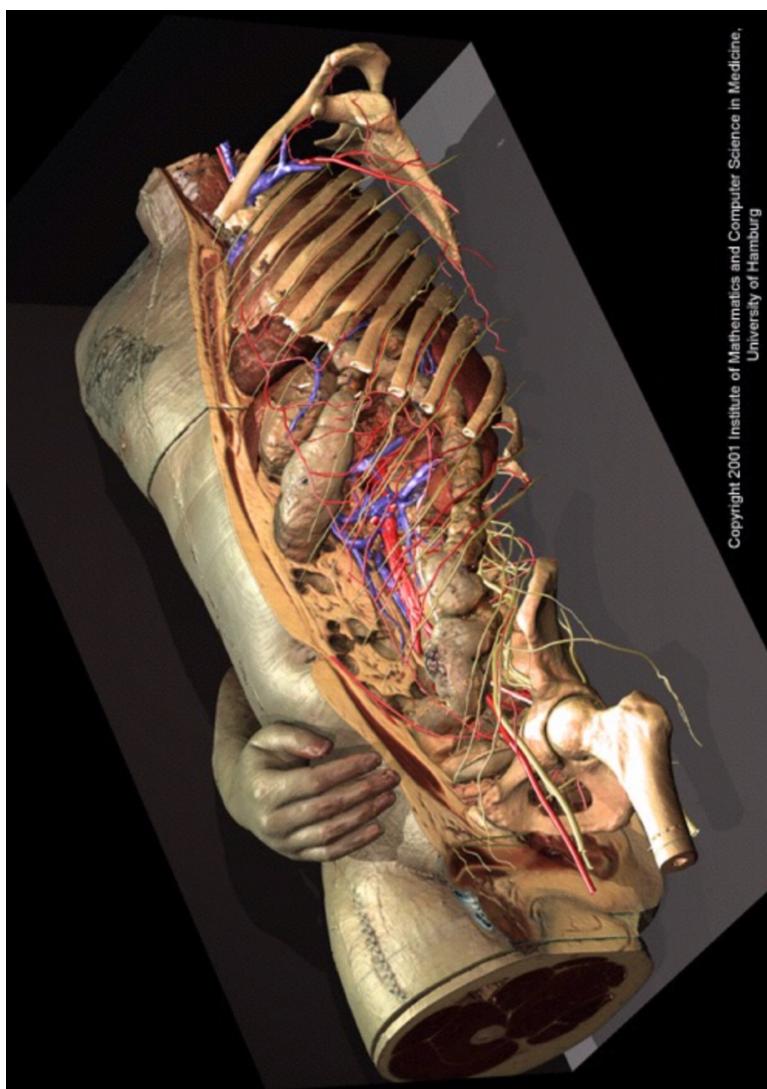


瑞利-泰勒
不稳定性

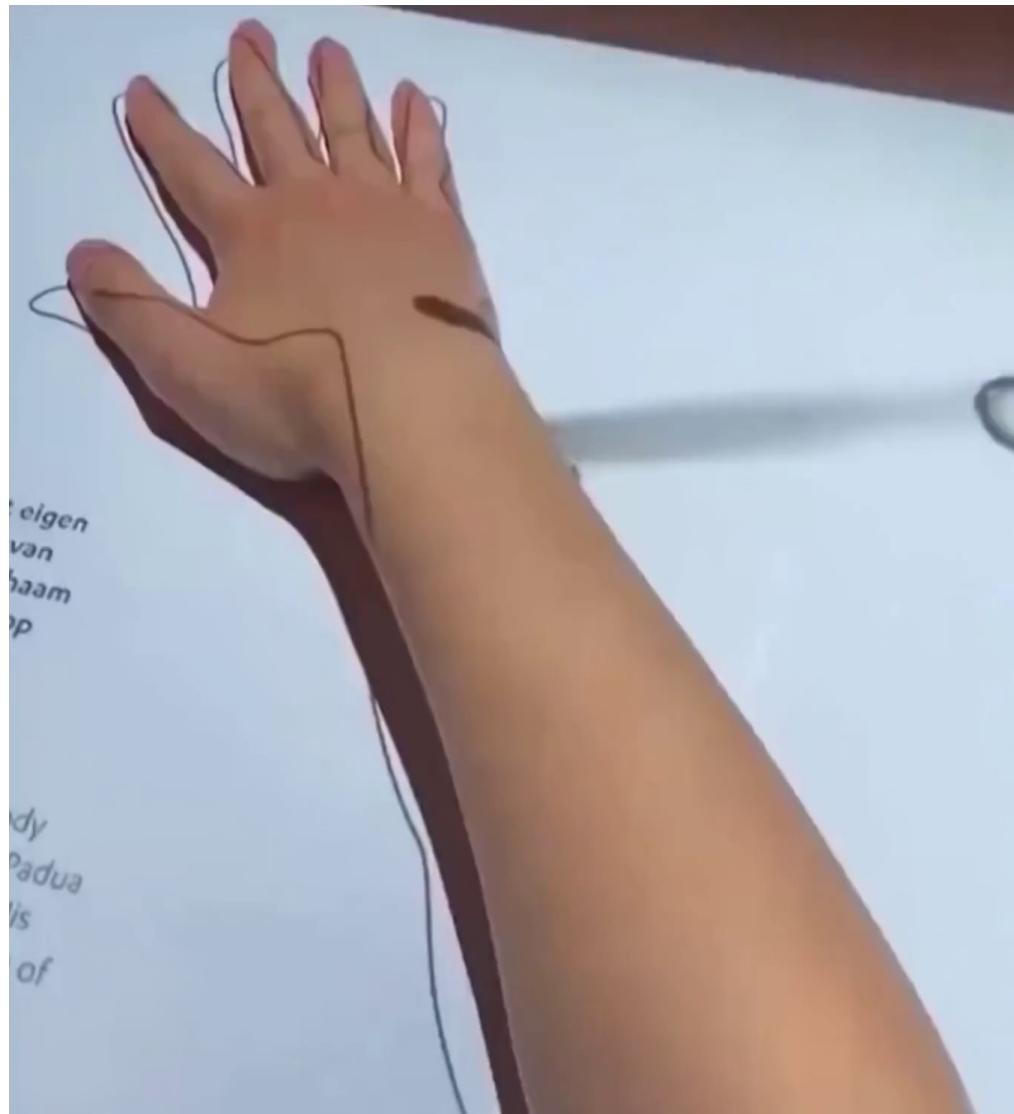


NASA 宇宙观测

医学/解剖学可视化



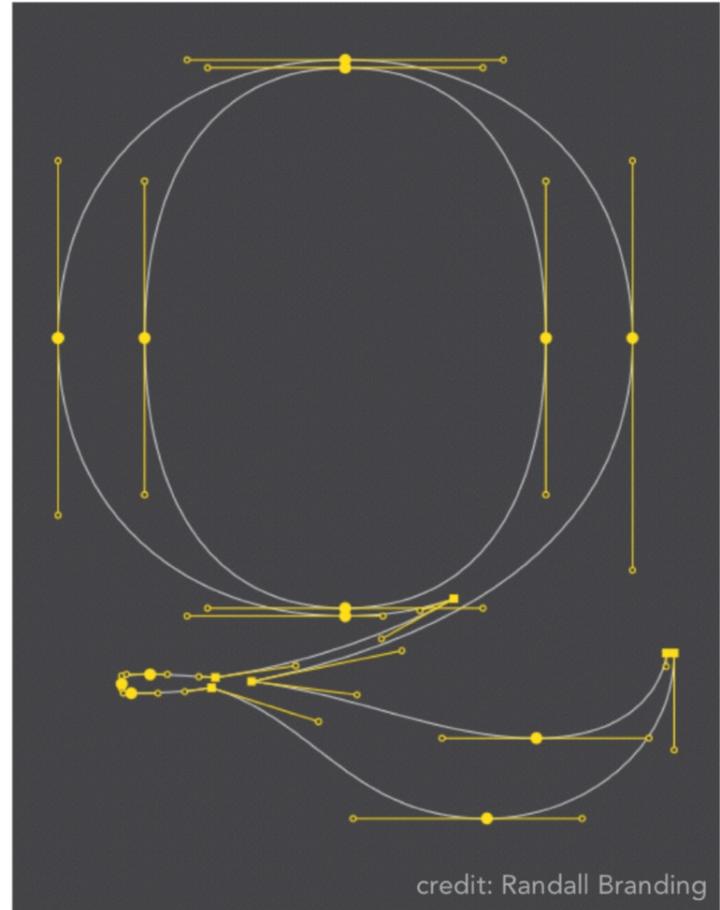
Copyright 2001 Institute of Mathematics and Computer Science in Medicine,
University of Hamburg



字体排版 Typography

The Quick Brown
Fox Jumps Over
The Lazy Dog

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 01234567890



credit: Randall Branding

计算机图形学基石

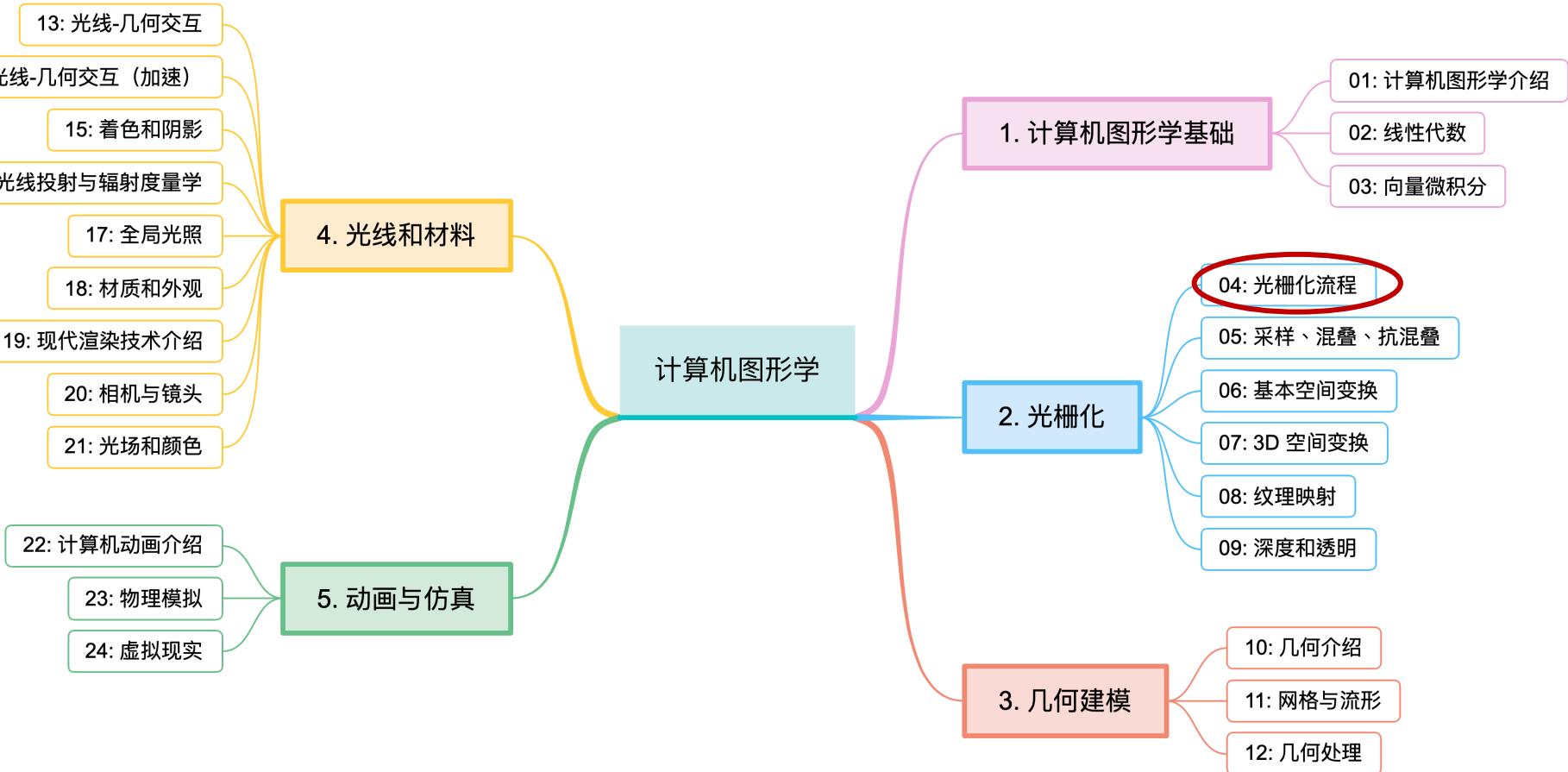
□ 所有这些图形学的应用均需要**复杂的理论和系统**

□ 理论 Theory

- **Basic representations** (如何对形状/运动进行数字编码)
- **Sampling & aliasing** (如何获取并重现信号?)
- **Numerical methods** (如何以数值的方式处理信号?)
- **Radiometry & light transport** (光是如何表现的?)
- **Perception** (这些与人类有什么关系? 人是如何感知的?)
- ...

□ 系统 Systems

- 并行、异构的处理
- 专门的图形编程语言
- ...



光栅化

Rasterization

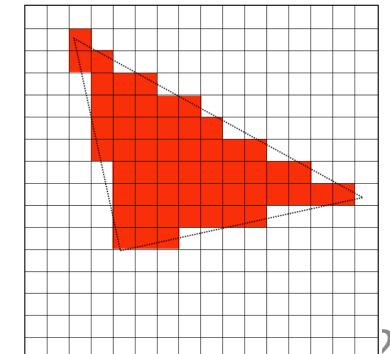
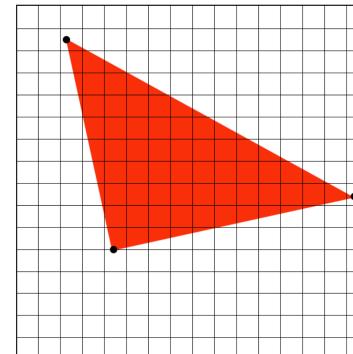
两种把图像“放到”屏幕上的技术

□ 光栅化 Rasterization

- 对于每个图形基元 (primitive)，如三角形，点亮哪些像素
- 非常快（在 GPU 中数十亿个三角形每秒）
- 很难（但并非不可能）实现真实图片效果
- 非常适合 2D vector art, fonts, quick 3D preview

□ 光线追踪 Ray tracing

- 对于每个像素，我们可以看到哪些图形基元
- 很容易实现真实图片效果
- 一般来讲很慢
- 随着课程进行我们将介绍更多



3D 图像生成的流程 (Pipeline)

□ 从一个 pipeline 的角度讨论图像生成：

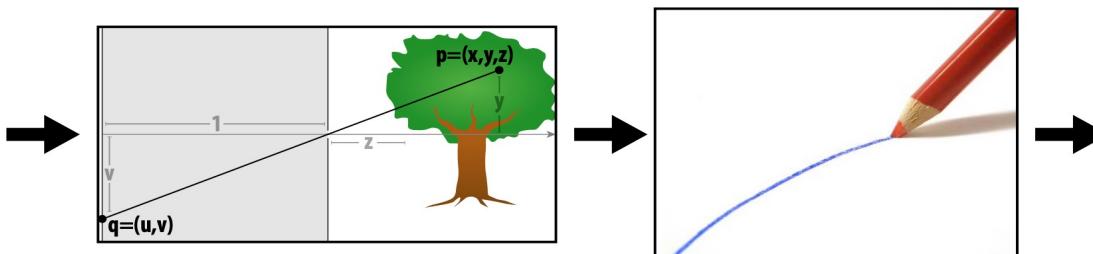
- INPUTS —— 我们想画什么图像？
- STAGES —— 输入的转换序列 → 输出
- OUTPUTS —— 最终图像

□ 比如，我们第一节课画立方体的 pipeline

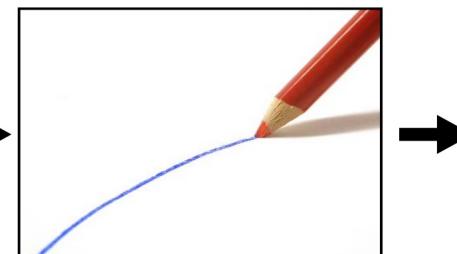
VERTICES
A: (1, 1, 1) E: (1, 1,-1)
B: (-1, 1, 1) F: (-1, 1,-1)
C: (1,-1, 1) G: (1,-1,-1)
D: (-1,-1, 1) H: (-1,-1,-1)

EDGES
AB, CD, EF, GH,
AC, BD, EG, FH,
AE, CG, BF, DH

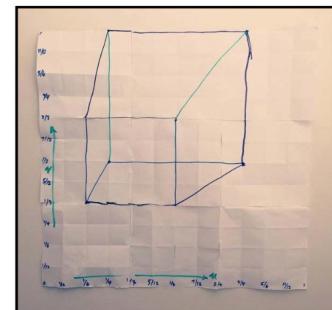
INPUT



**PERSPECTIVE
PROJECTION
STAGE**



**LINE
DRAWING
STAGE**



OUTPUT

光栅化流程 Rasterization pipeline

口 基于光栅化的现代实时图像生成

- Input: 图形基元 - 基本都是点 (points) 和三角形 (triangles)
 - 可能有其他属性 (例如颜色)
- Output: 图像 (可能有深度、透明度等)

口 我们的目标: 理解其中不同阶段的细节*



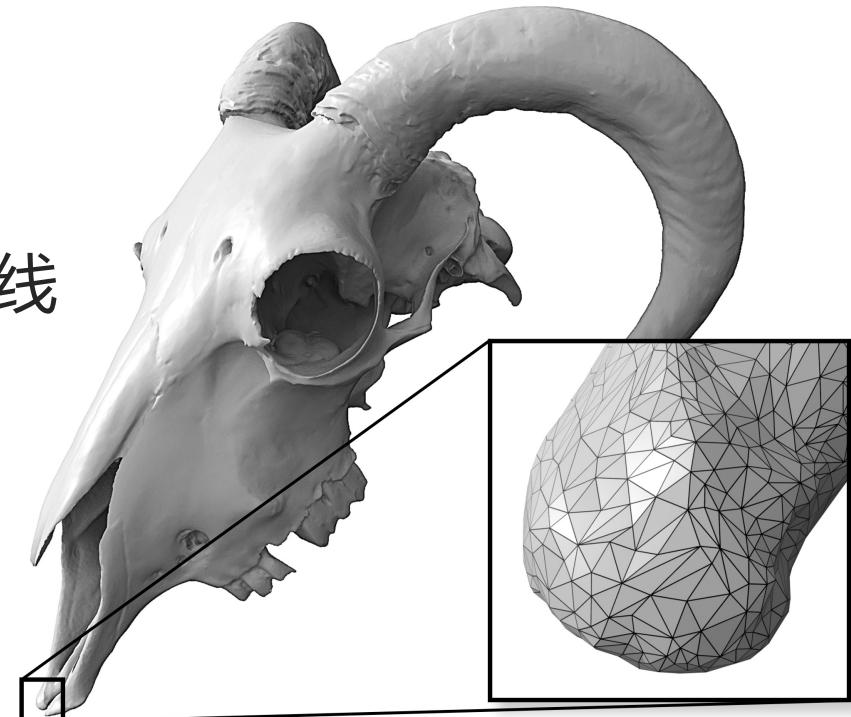
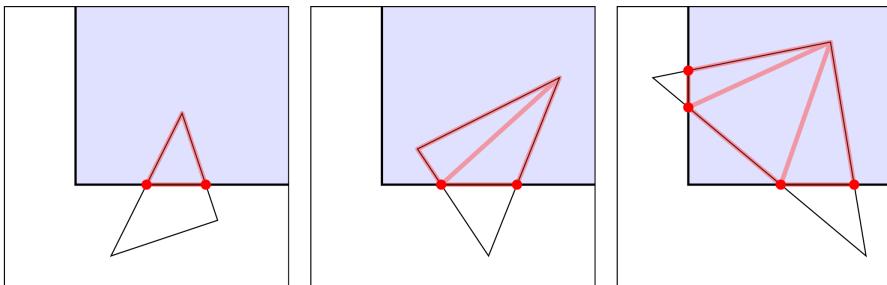
Why triangles?

口光栅化管道将所有的图形基元转换成三角形

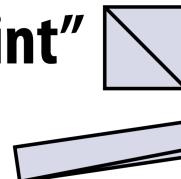
口为什么选择三角形？

- 可以近似任何形状
- 总是平面的，有定义明确的法线
- 定义良好的三角形顶点插值方法（重心插值法）

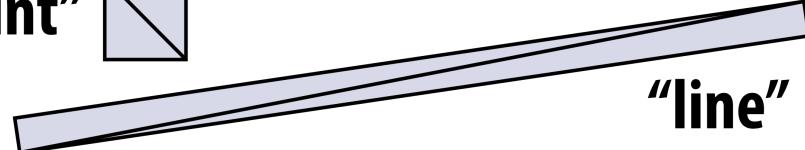
口关键原因：一旦所有东西都被简化为三角形，就可以专注于设计一个深度优化的绘制流程



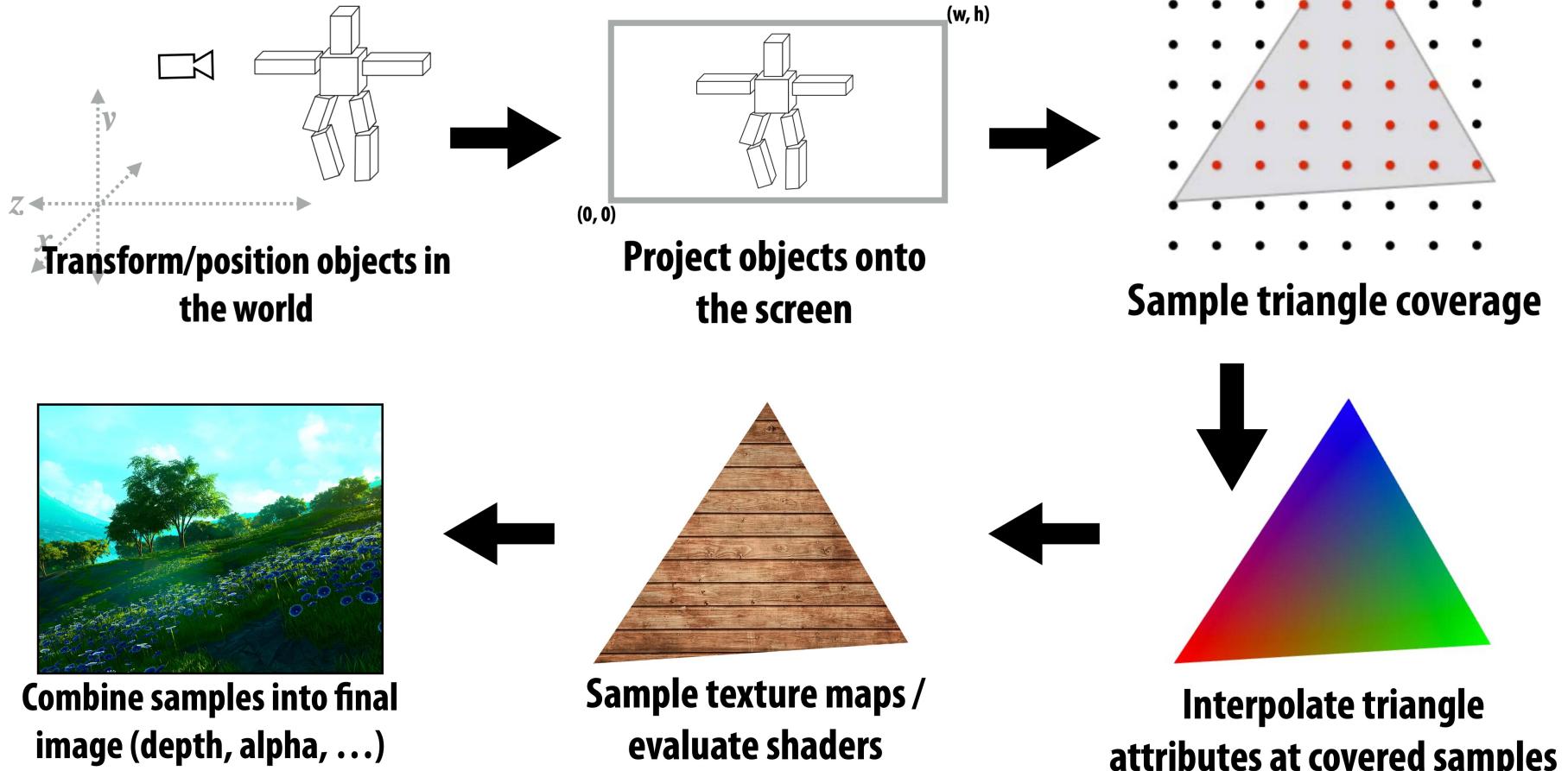
"point"



"line"



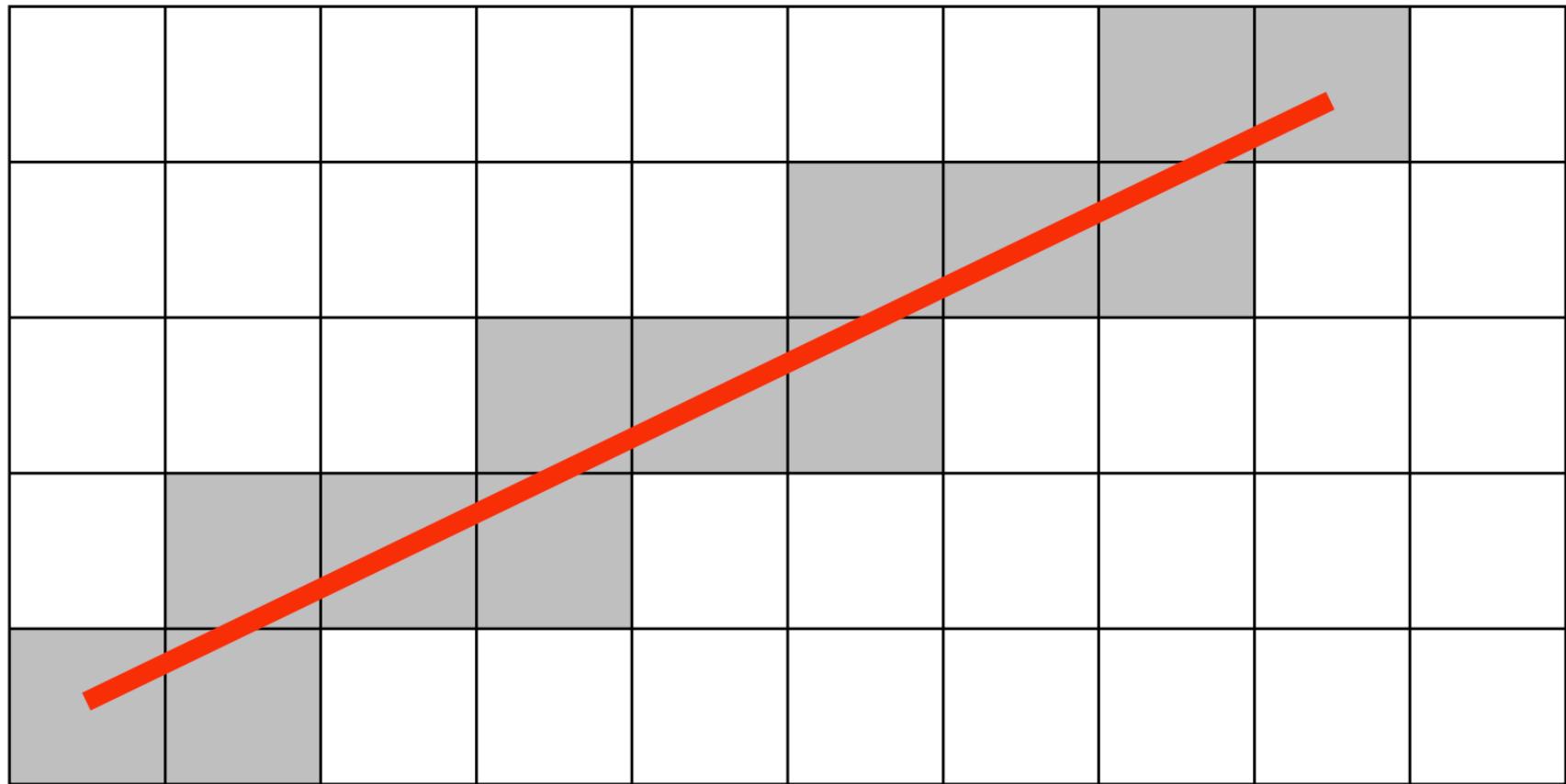
光栅化流程中的步骤



- “真实世界” 的标准光栅化流程 (OpenGL/Direct3D)

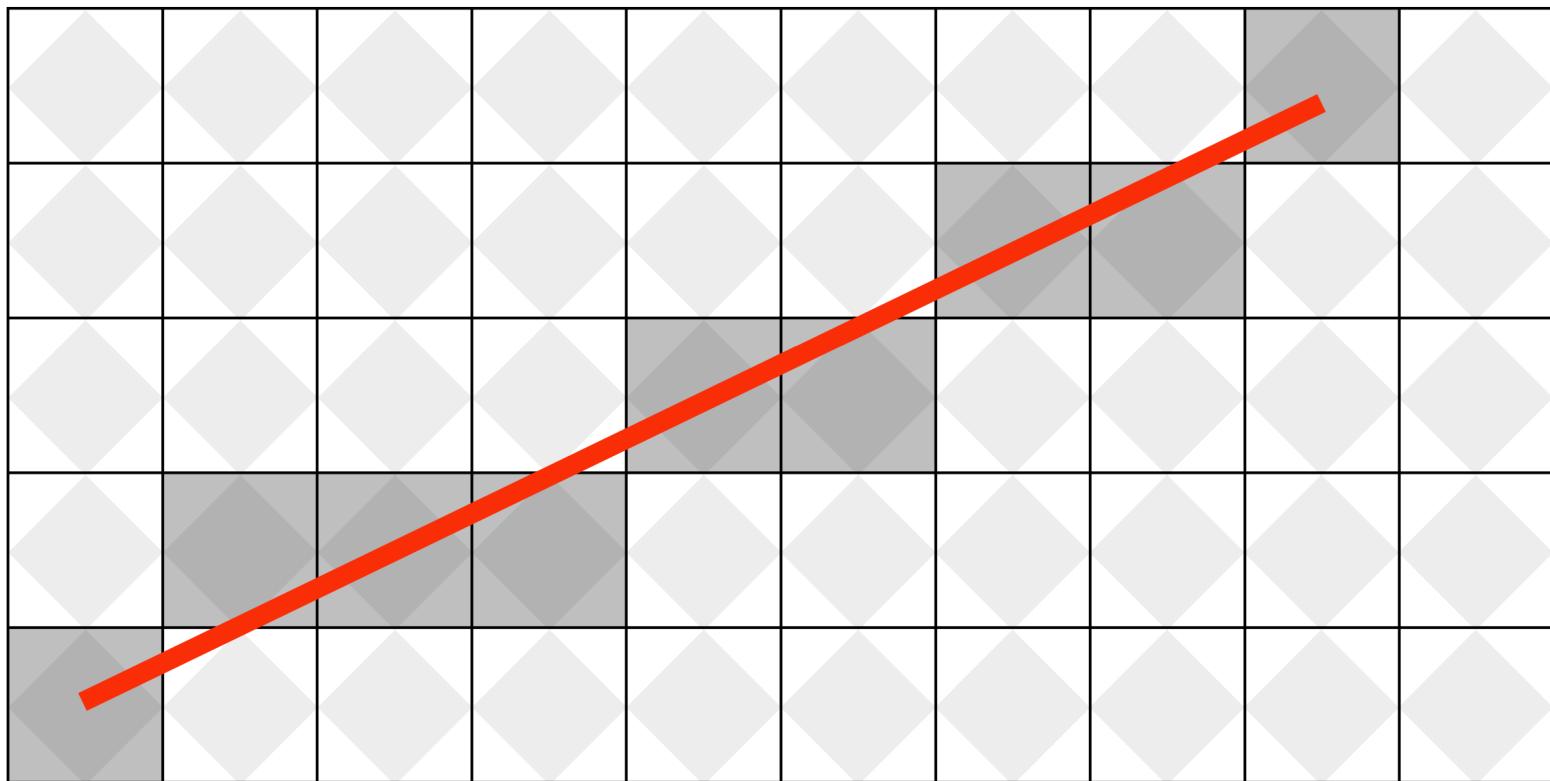
我们应该填充哪些像素来绘制直线？

点亮所有被线条穿过的像素？



我们应该填充哪些像素来绘制直线？

钻石规则 Diamond rule (现代 GPU 使用):
如果线穿过响应的“钻石”，就点亮像素



增量线光栅化

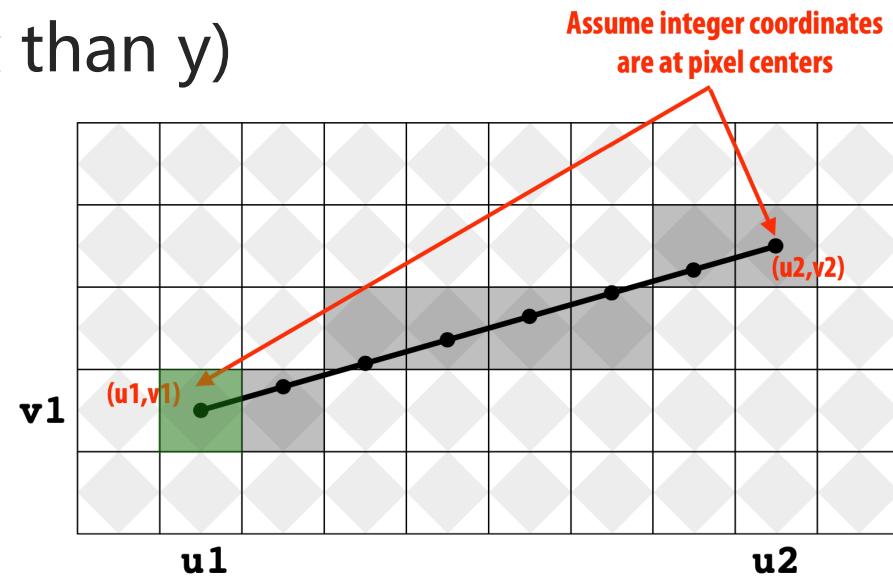
□ 我们用两个点表示一条线: $(u_1, v_1), (u_2, v_2)$

□ 线的斜率 $s = (v_2 - v_1) / (u_2 - u_1)$

□ 考虑一个具体的例子

- $u_1 < u_2, v_1 < v_2$ (line points toward upper-right)
- $0 < s < 1$ (more change in x than y)

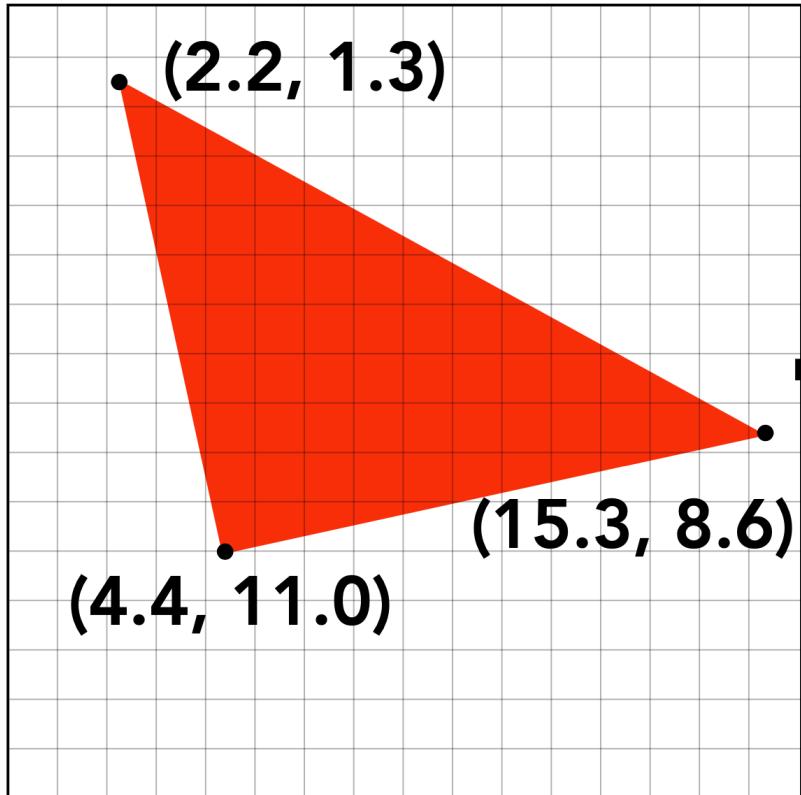
```
v = v1;  
for(u=u1; u<=u2; u++)  
{  
    v += s;  
    draw(u, round(v))  
}
```



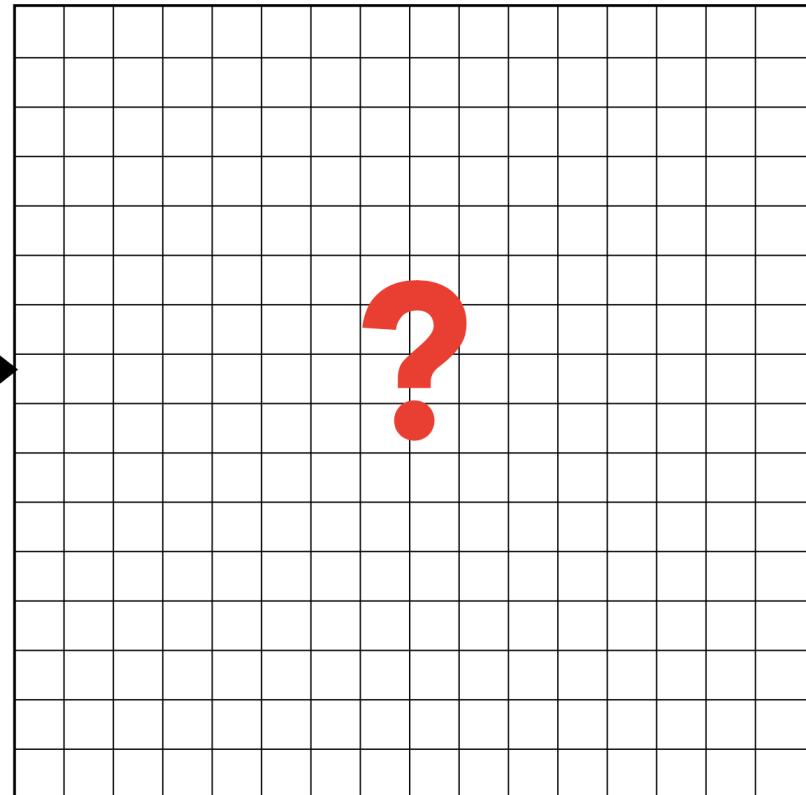
易于实现，但不是当代软件/硬件画线的唯一方式

三角形呢？

□ 填充屏幕中哪些像素才能绘制三角形？

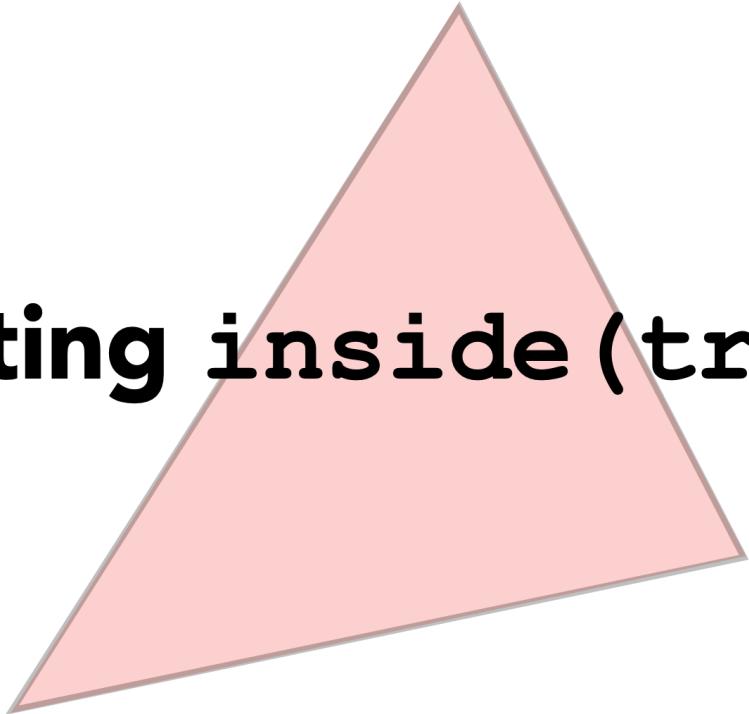


**Input: position of triangle
vertices projected on screen**

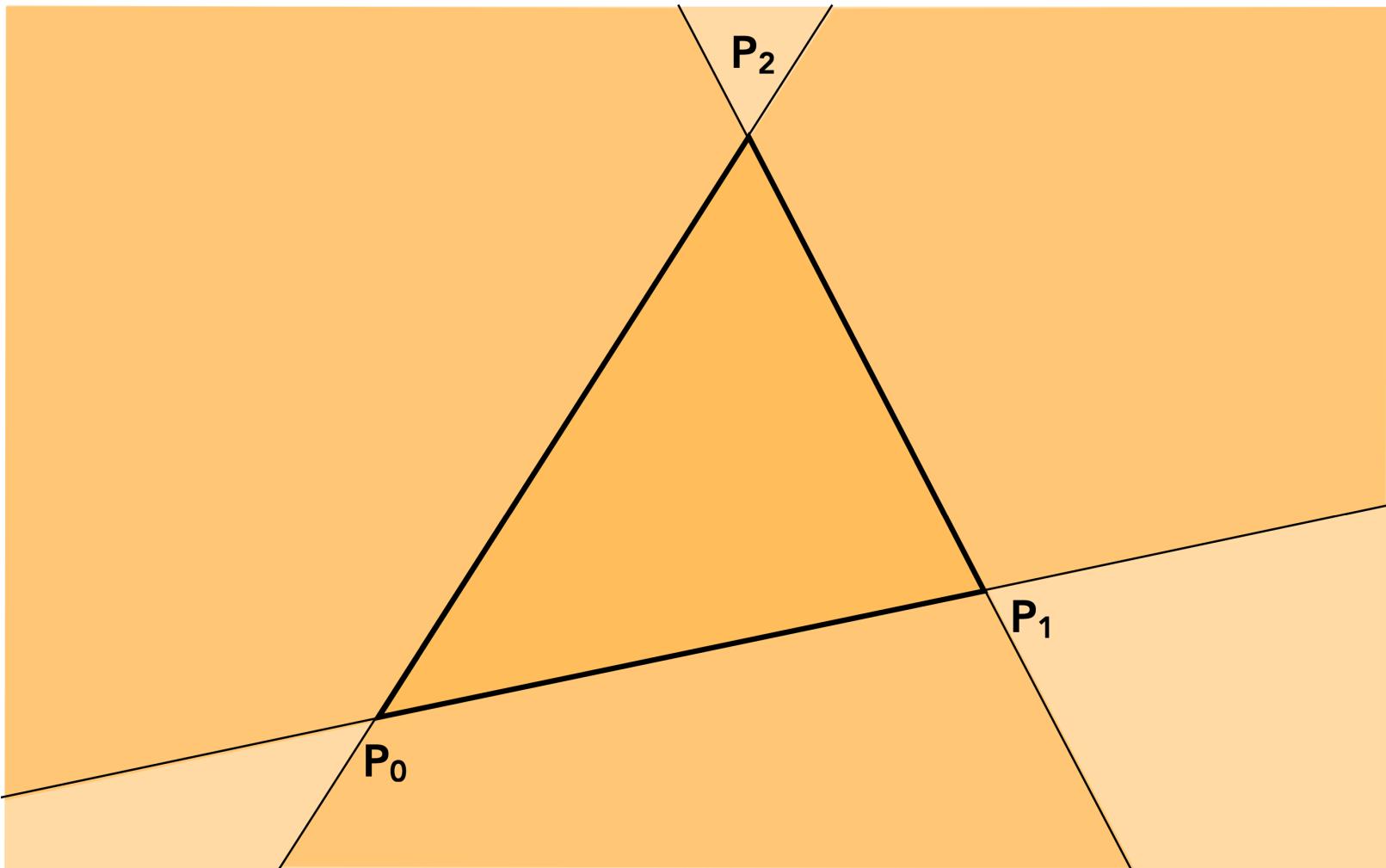


**Output: set of pixel values
approximating triangle**

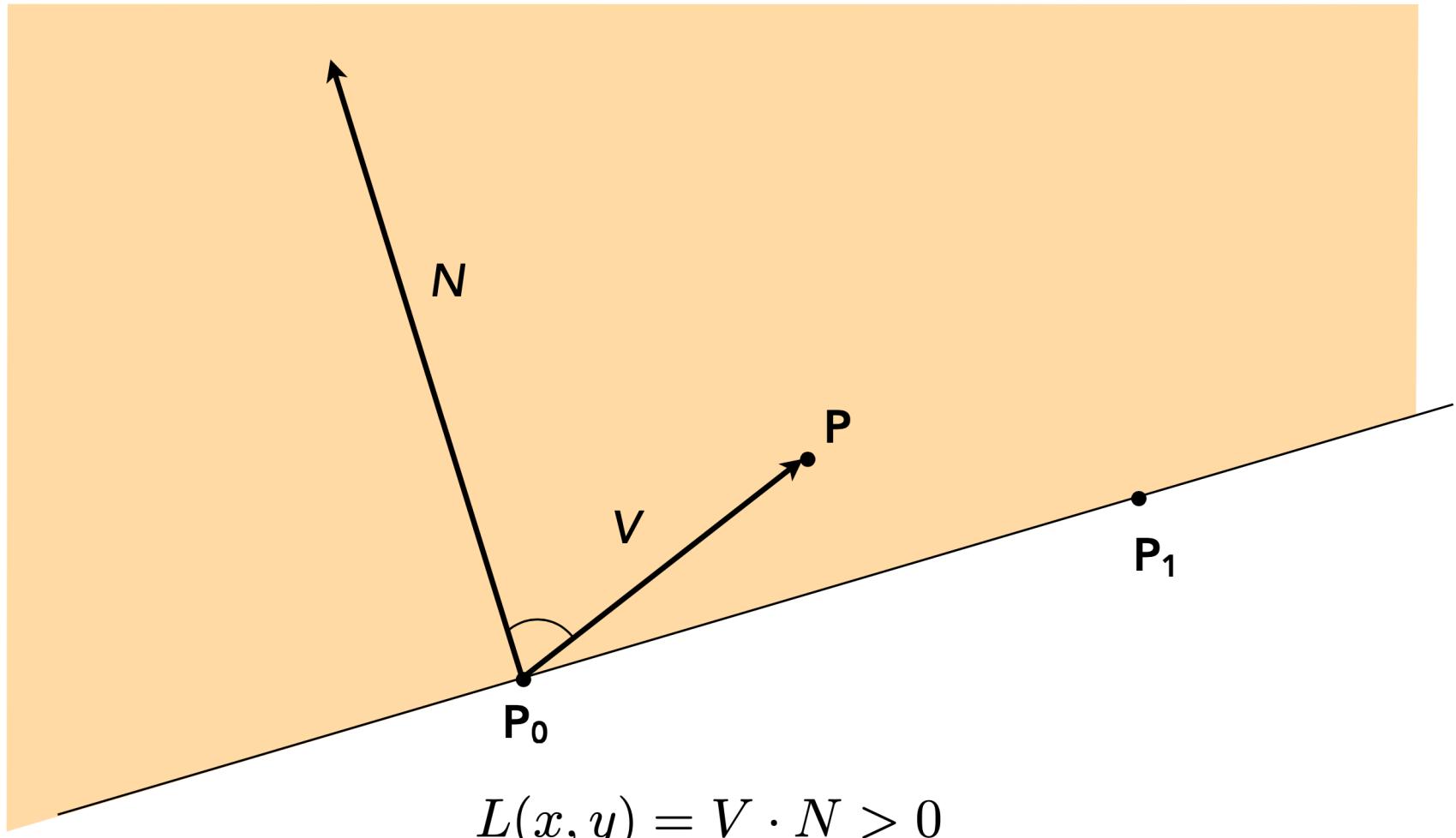
Evaluating inside(tri , x , y)



三角形 = 三个半平面的交集

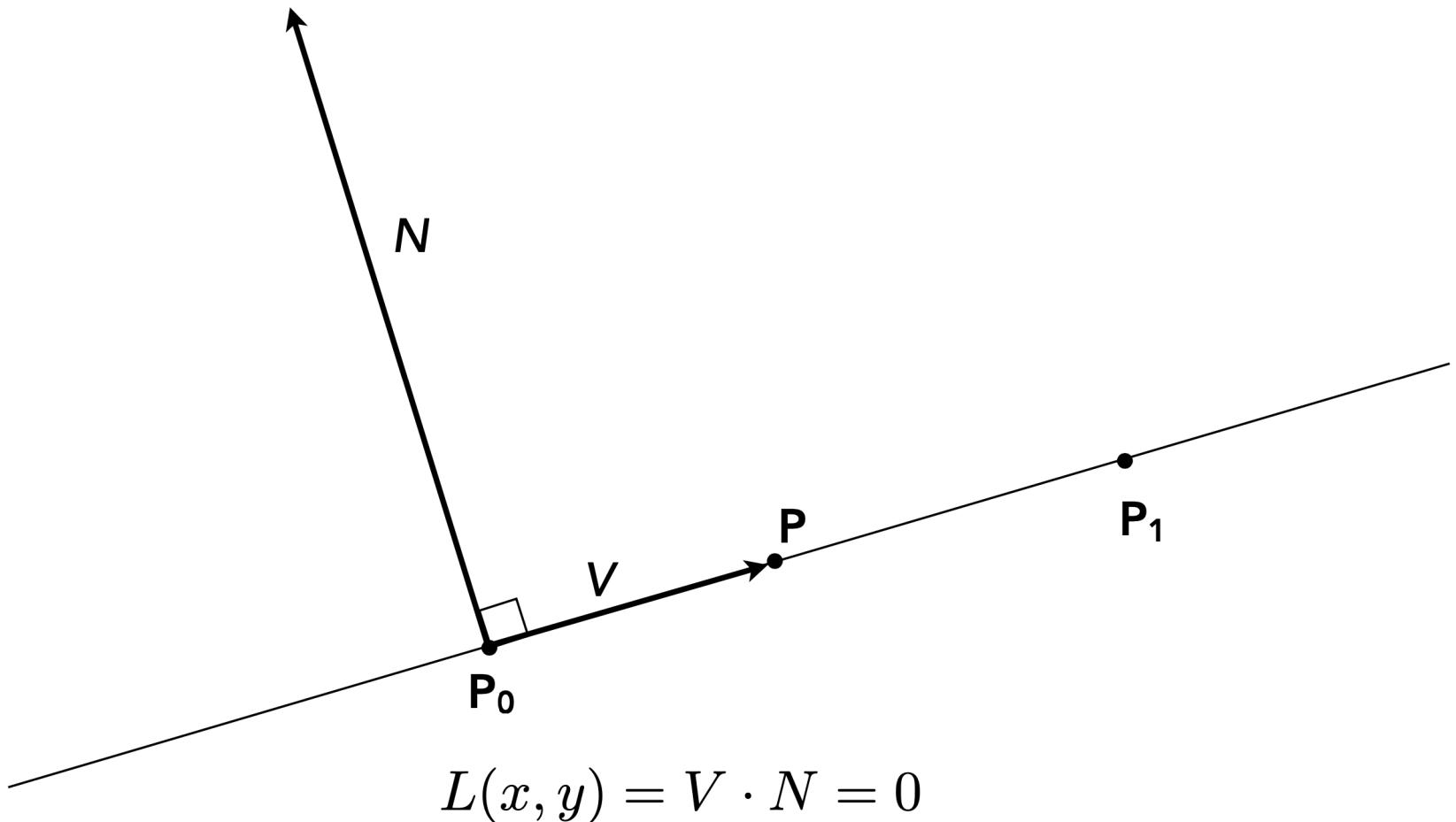


直线方程测试

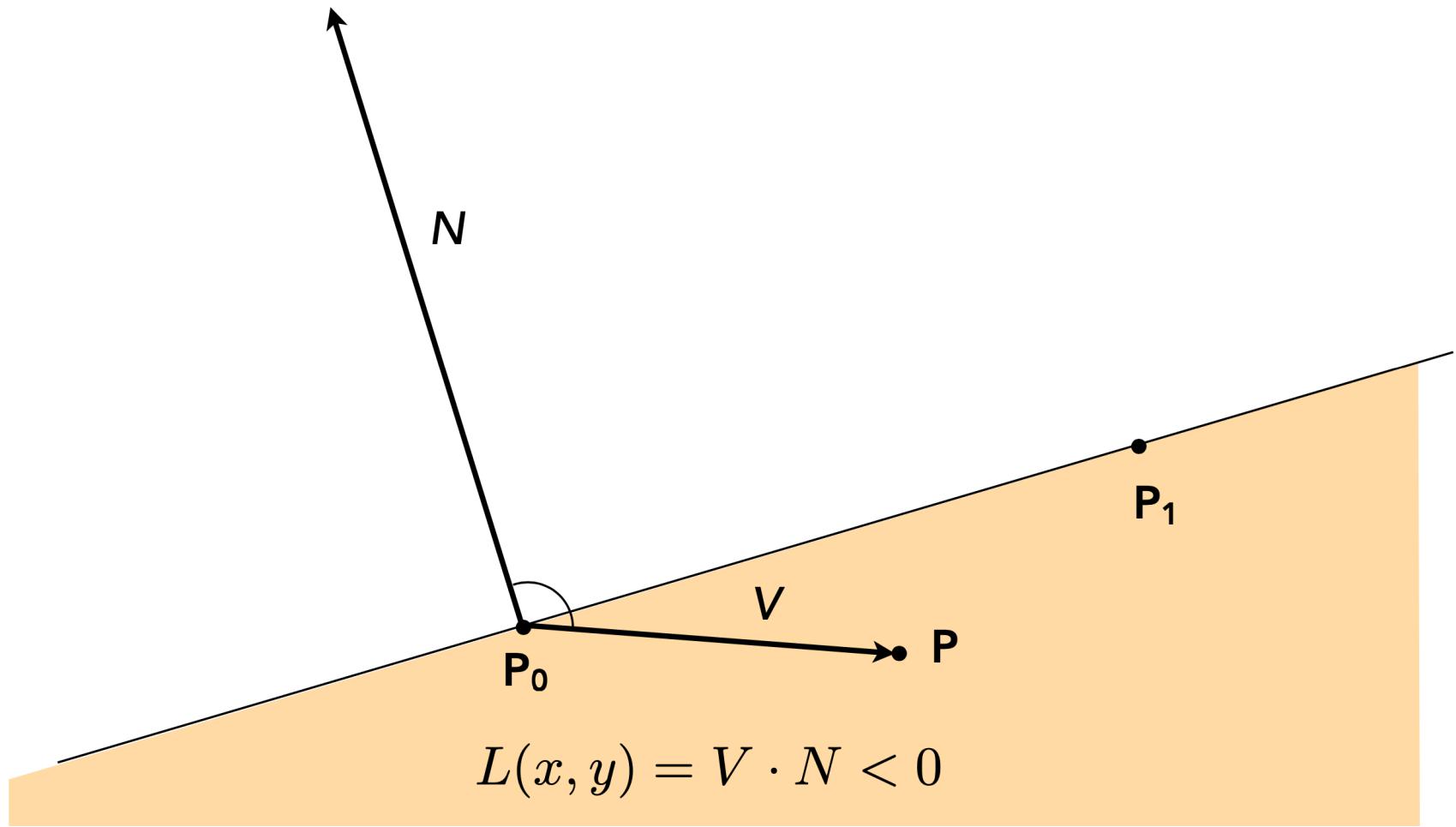


$$L(x, y) = V \cdot N > 0$$

直线方程测试



直线方程测试

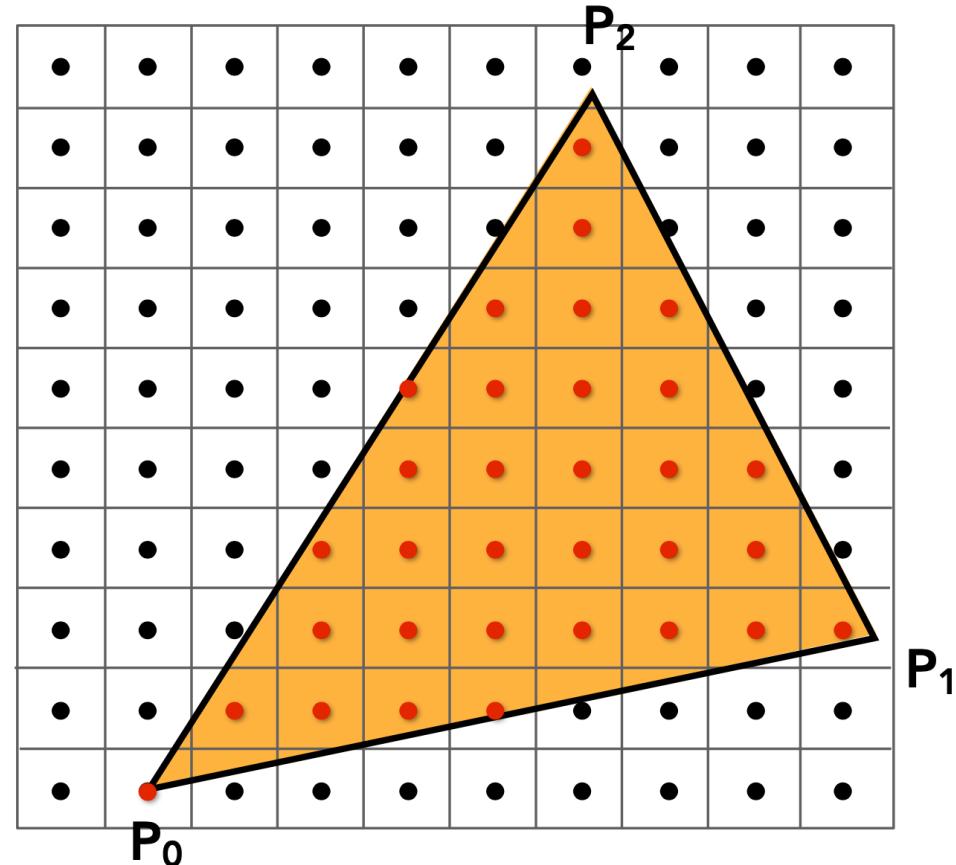


三角形内点测试：三线测试

Sample point $s = (sx, sy)$ is inside the triangle if it is inside all three lines.

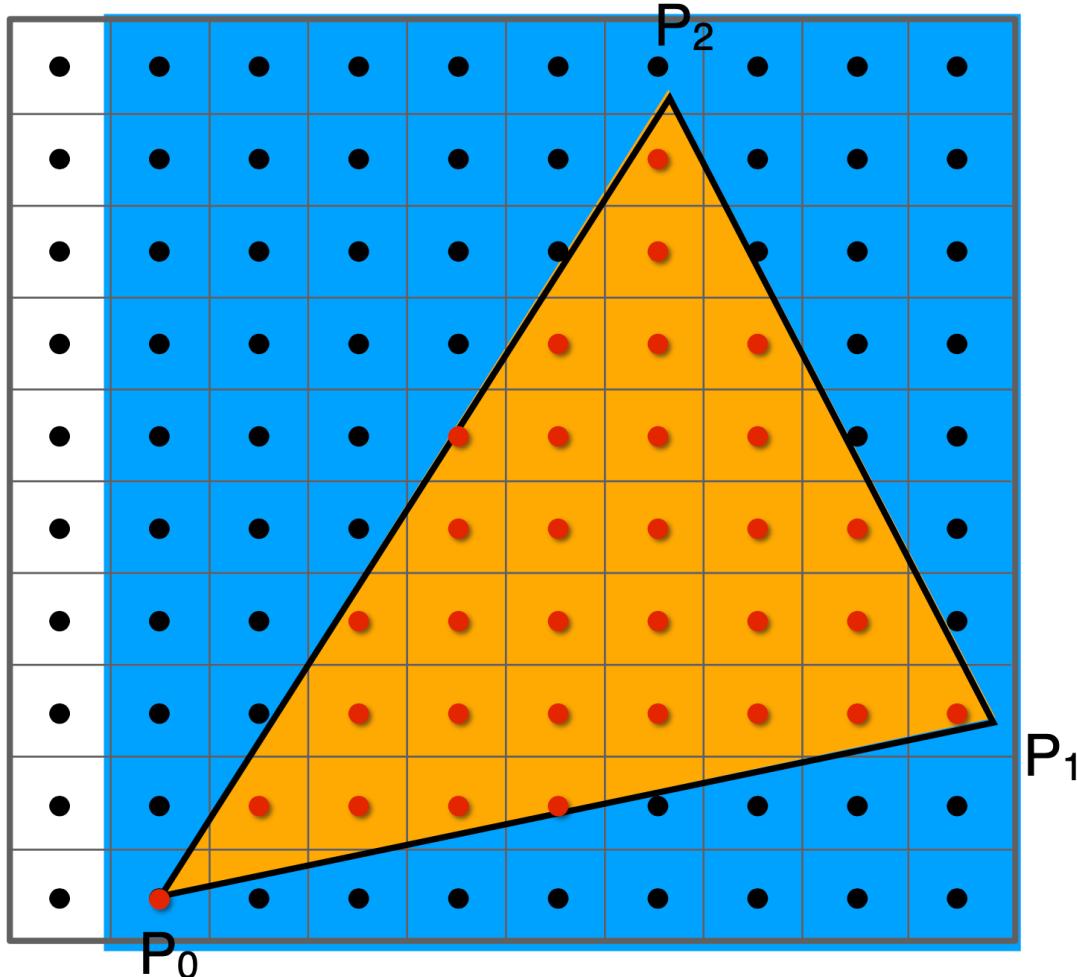
$inside(sx, sy) =$
 $L_0(sx, sy) > 0 \ \&\&$
 $L_1(sx, sy) > 0 \ \&\&$
 $L_2(sx, sy) > 0;$

Note: actual implementation of $inside(sx, sy)$ involves \leq checks based on edge rules



如何加速三角形的绘制过程？

包围盒加速三角形遍历



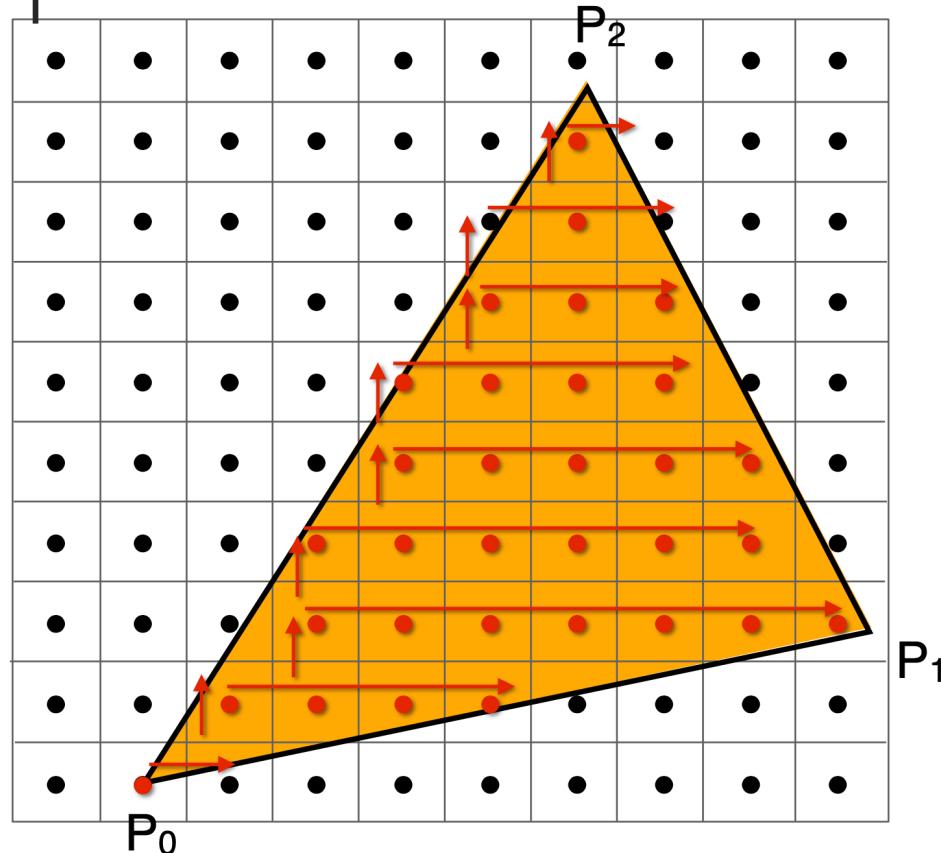
Always efficient?

增量三角形遍历

□ 通过“增量式”技术提高半平面检查的计算效率

□ 增量式方法以一种与数据在内存中的排列方式相一致的模式访问内存，这可以大大提高渲染的速度

- backtrack
- zigzag
- Hilbert/Morton curves



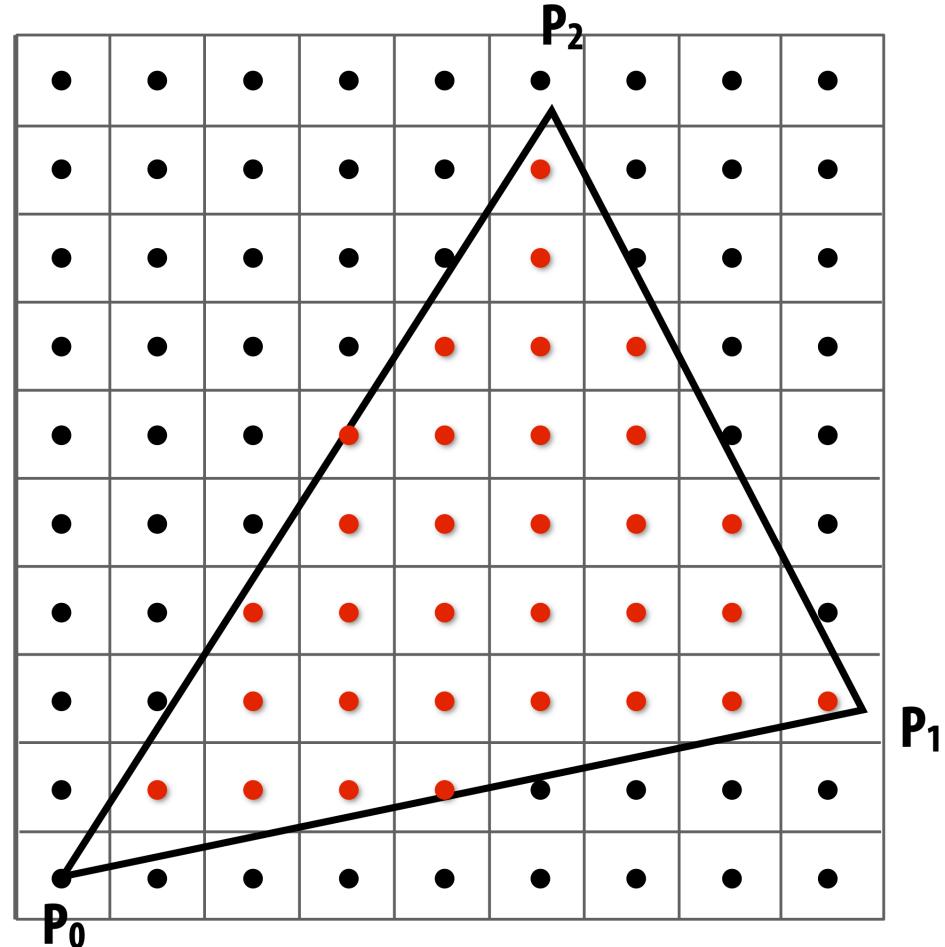
并行覆盖检测

口增量式遍历是串行的；现代硬件支持高度并行计算

口替代方案：并行测试三角形“包围盒”内的所有样本点

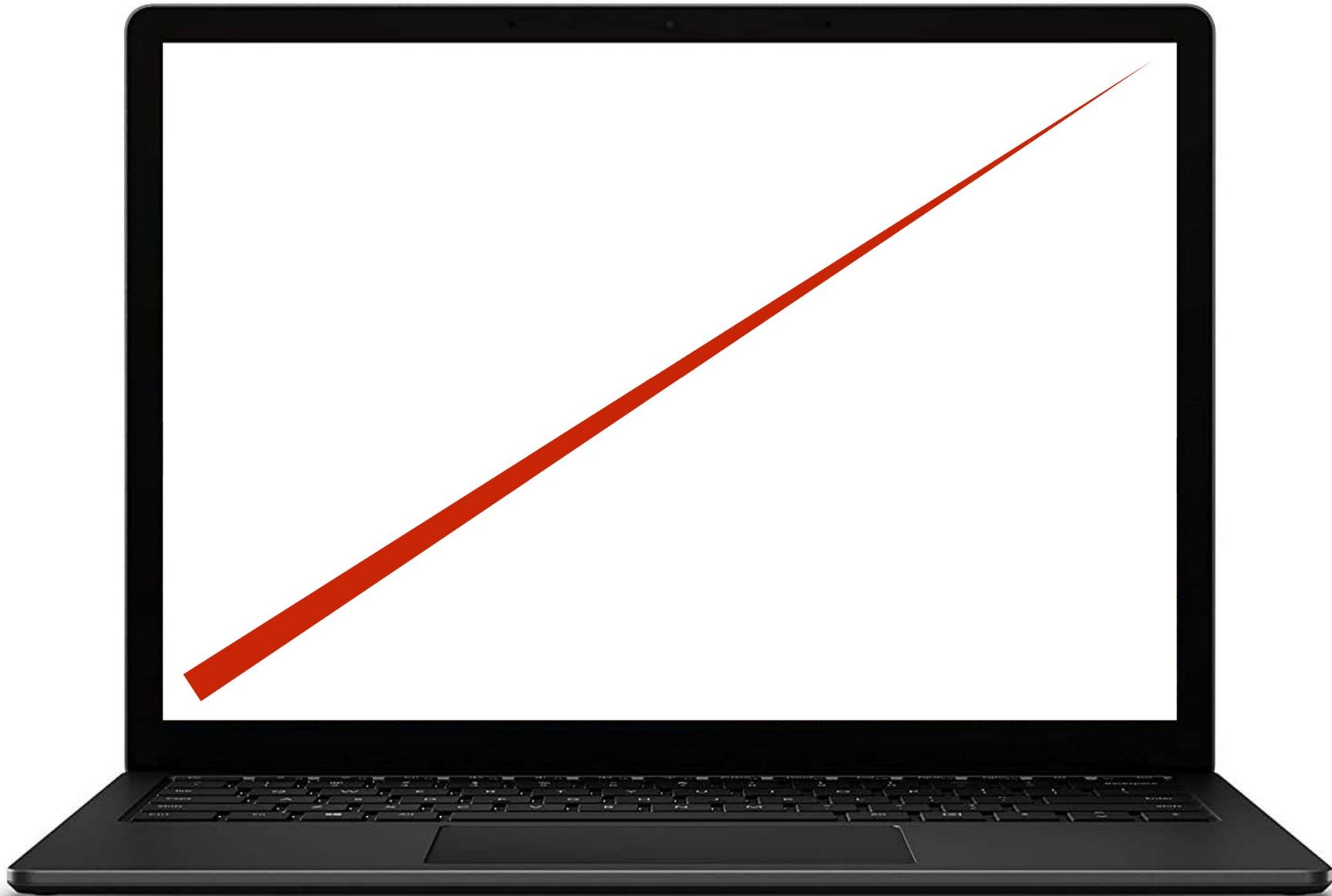
口大量的并行可以抵消额外的测试（大多数三角形覆盖许多样本，特别是超采样时）

口现代图形处理单元 (GPU) 具有专用硬件，可以高效地进行 point-in-triangle 测试



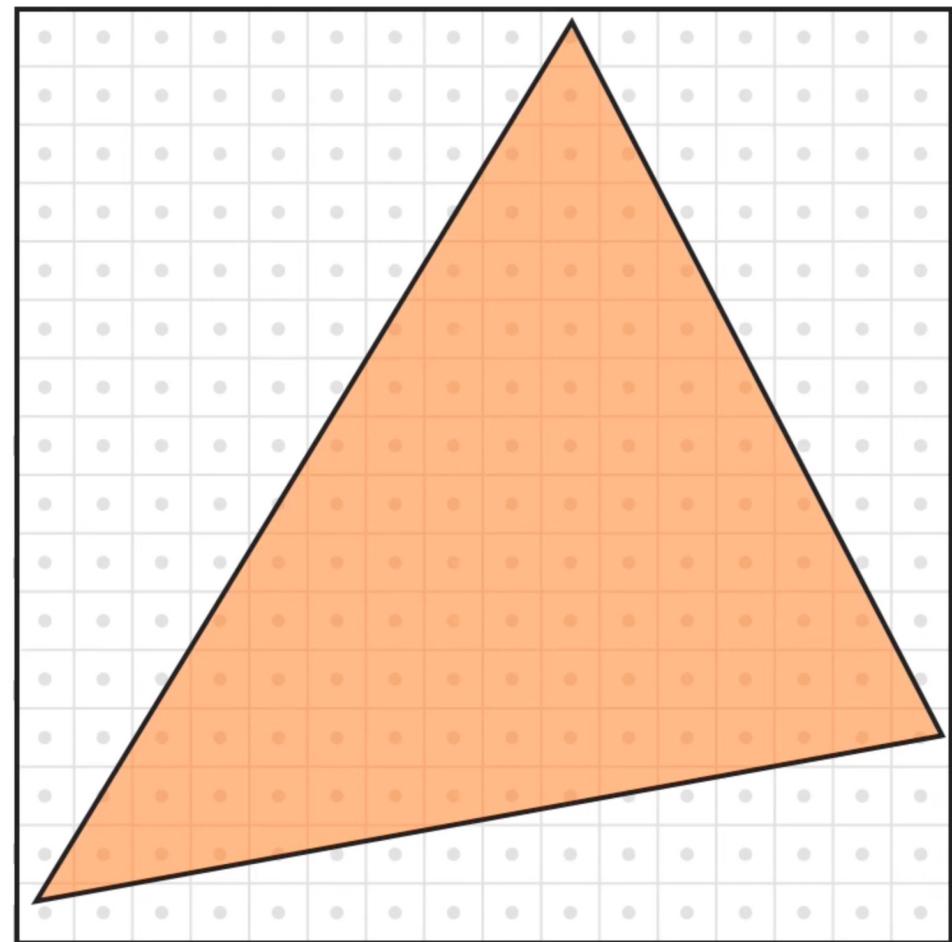
Q：在什么样的场景中，初级的并行策略仍然非常低效？

Naïve approach can be (very) wasteful...



混合方法：平铺三角形遍历

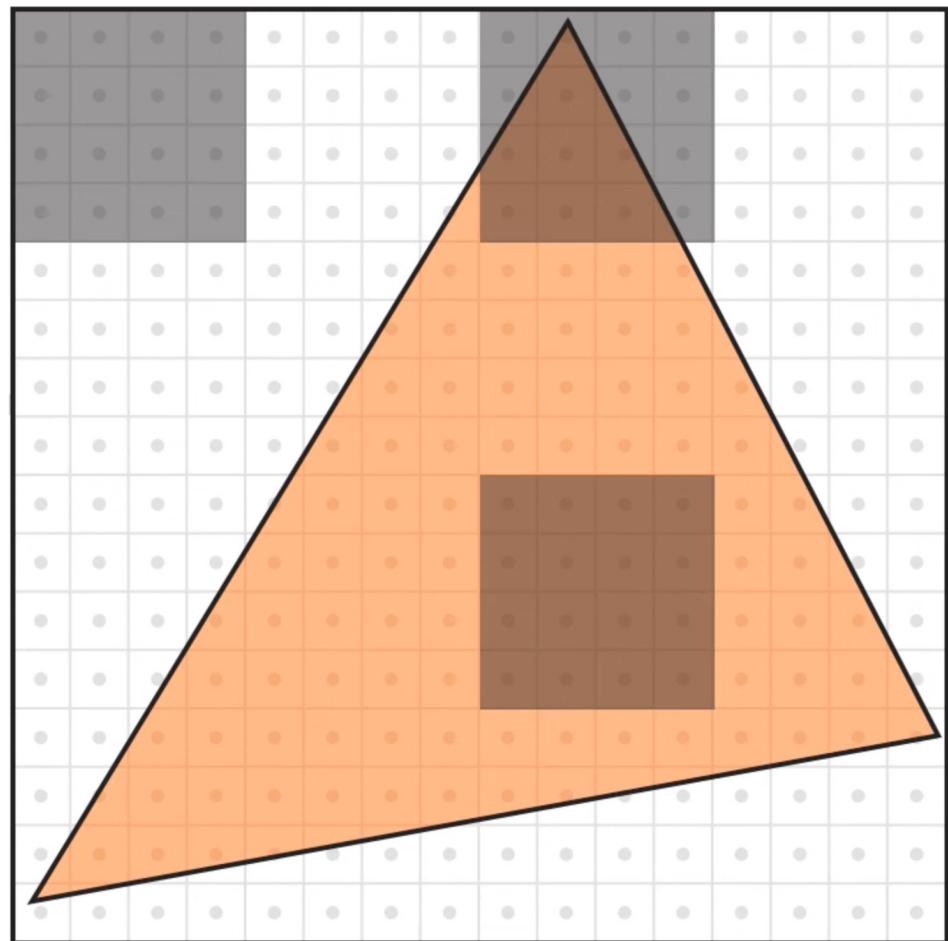
口理念：从粗到细



混合方法：平铺三角形遍历

□理念：从粗到细

□首先，检查块是否与三角形相交（**怎么做？**）

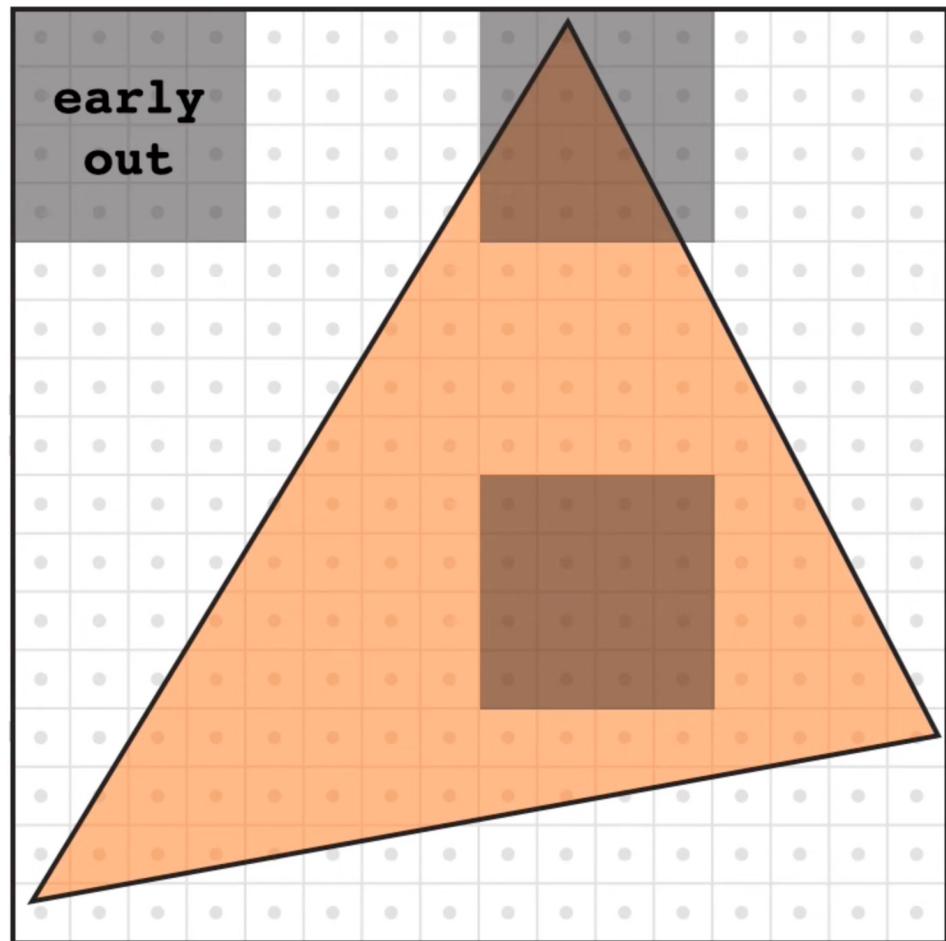


混合方法：平铺三角形遍历

口理念：从粗到细

口首先，检查块是否与三角形相交（**怎么做？**）

口如果没有，完全跳过此块
（“提前退出 early out”）



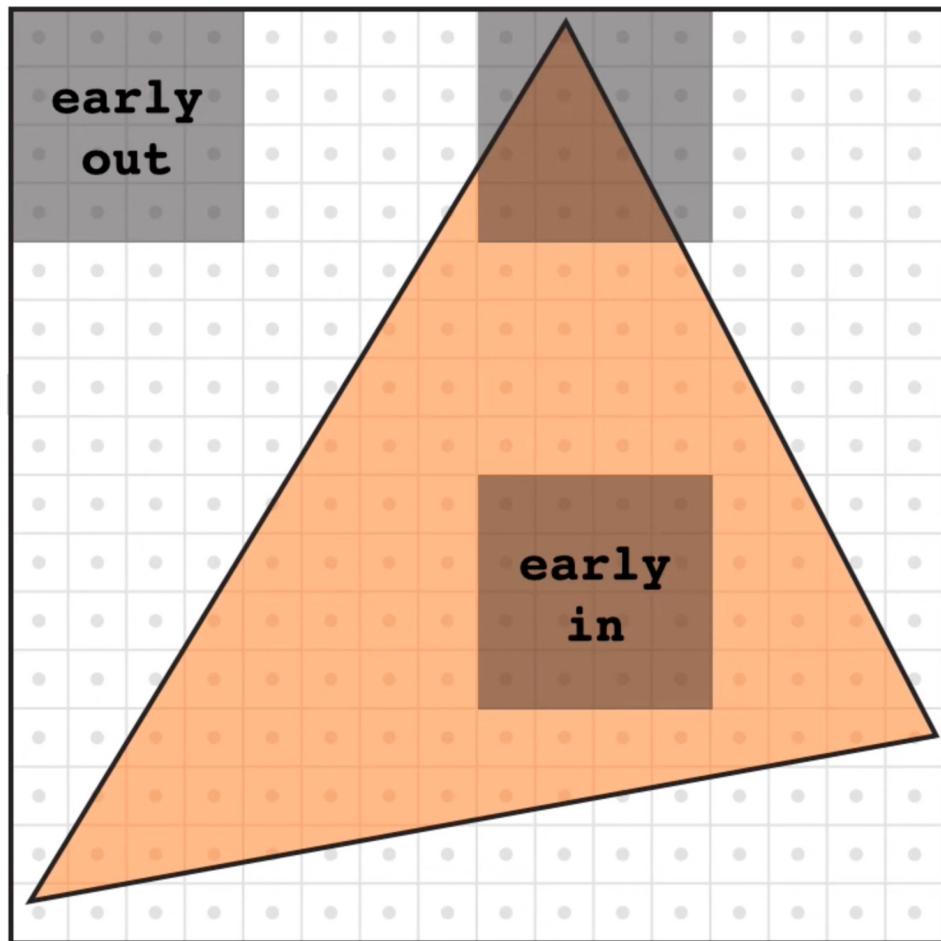
混合方法：平铺三角形遍历

口理念：从粗到细

口首先，检查块是否与三角形相交（**怎么做？**）

口如果没有，完全跳过此块
（“提前退出 early out”）

口如果块包含在三角形内，则
块内所有点都被覆盖了
（“早期进入 early in”）



混合方法：平铺三角形遍历

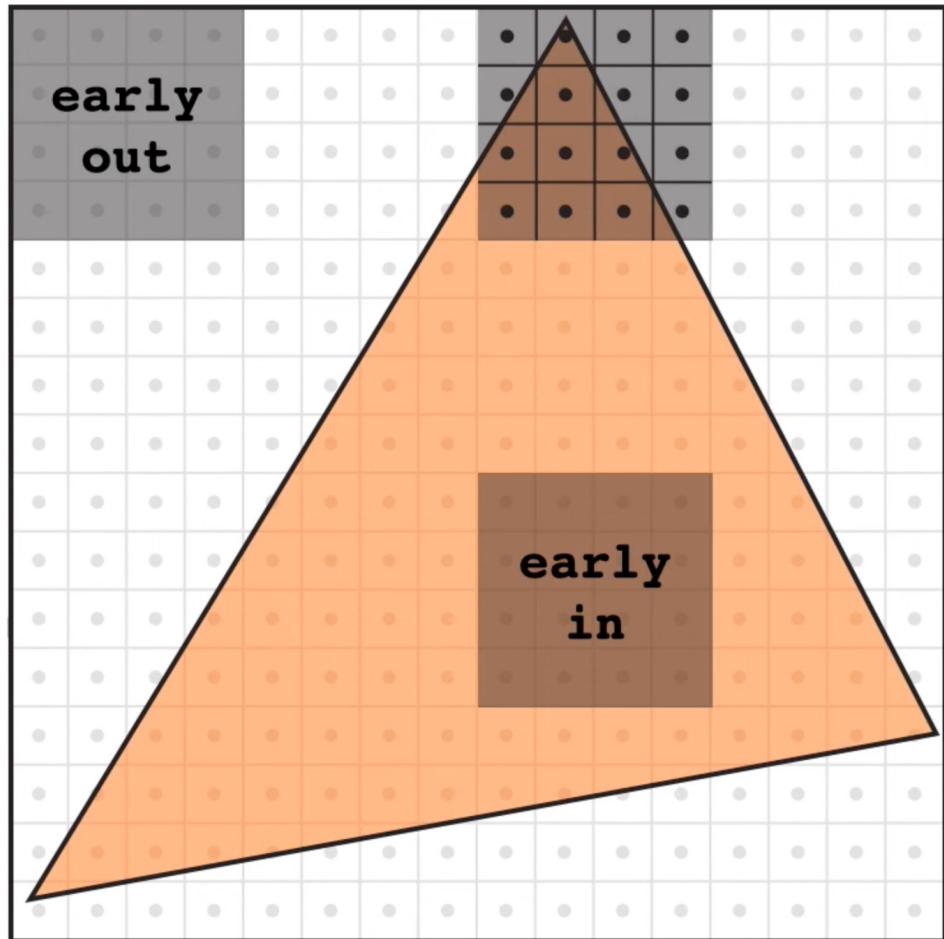
口理念：从粗到细

口首先，检查块是否与三角形相交（**怎么做？**）

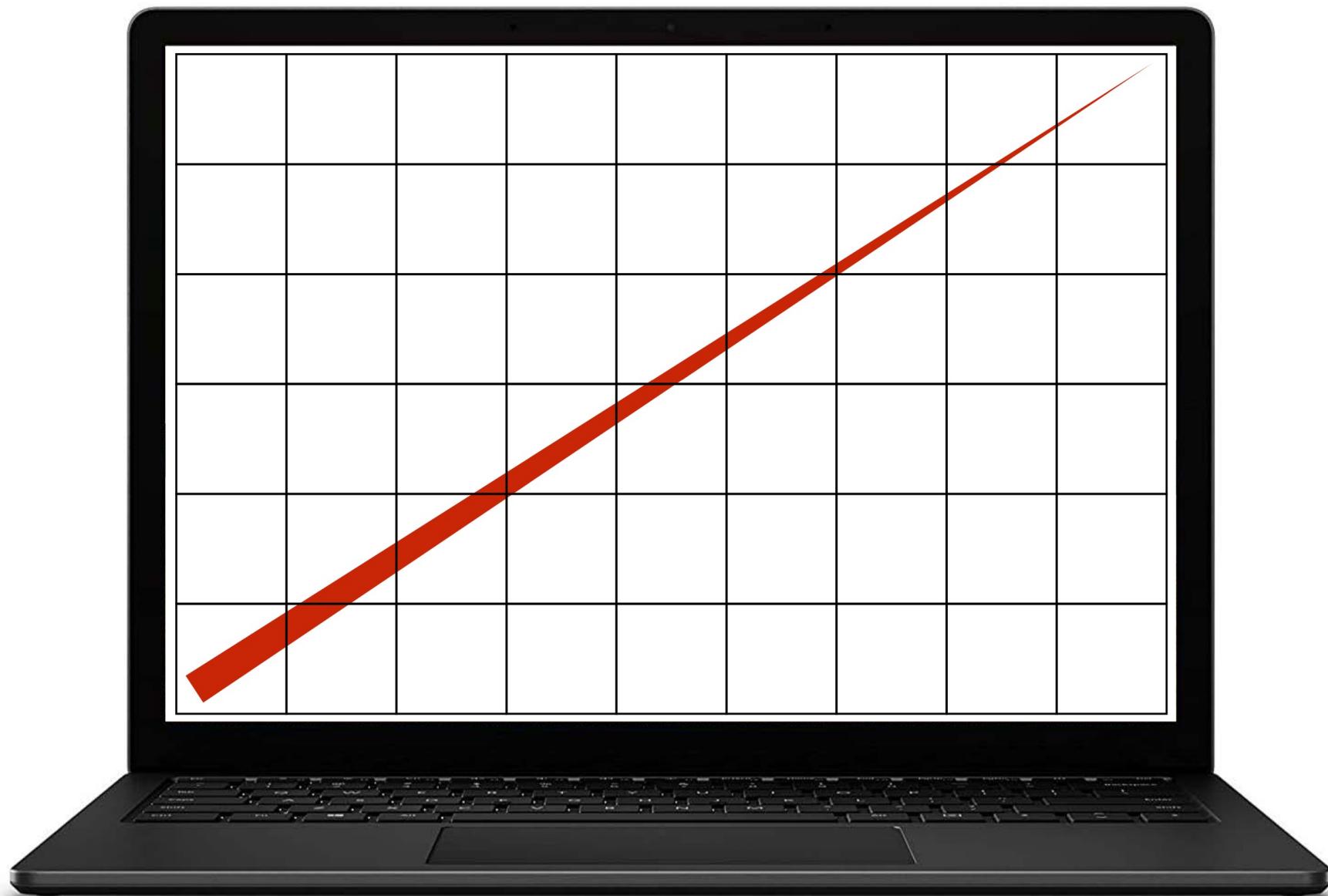
口如果没有，完全跳过此块
（“提前退出 early out”）

口如果块包含在三角形内，则
块内所有点都被覆盖了
（“早期进入 early in”）

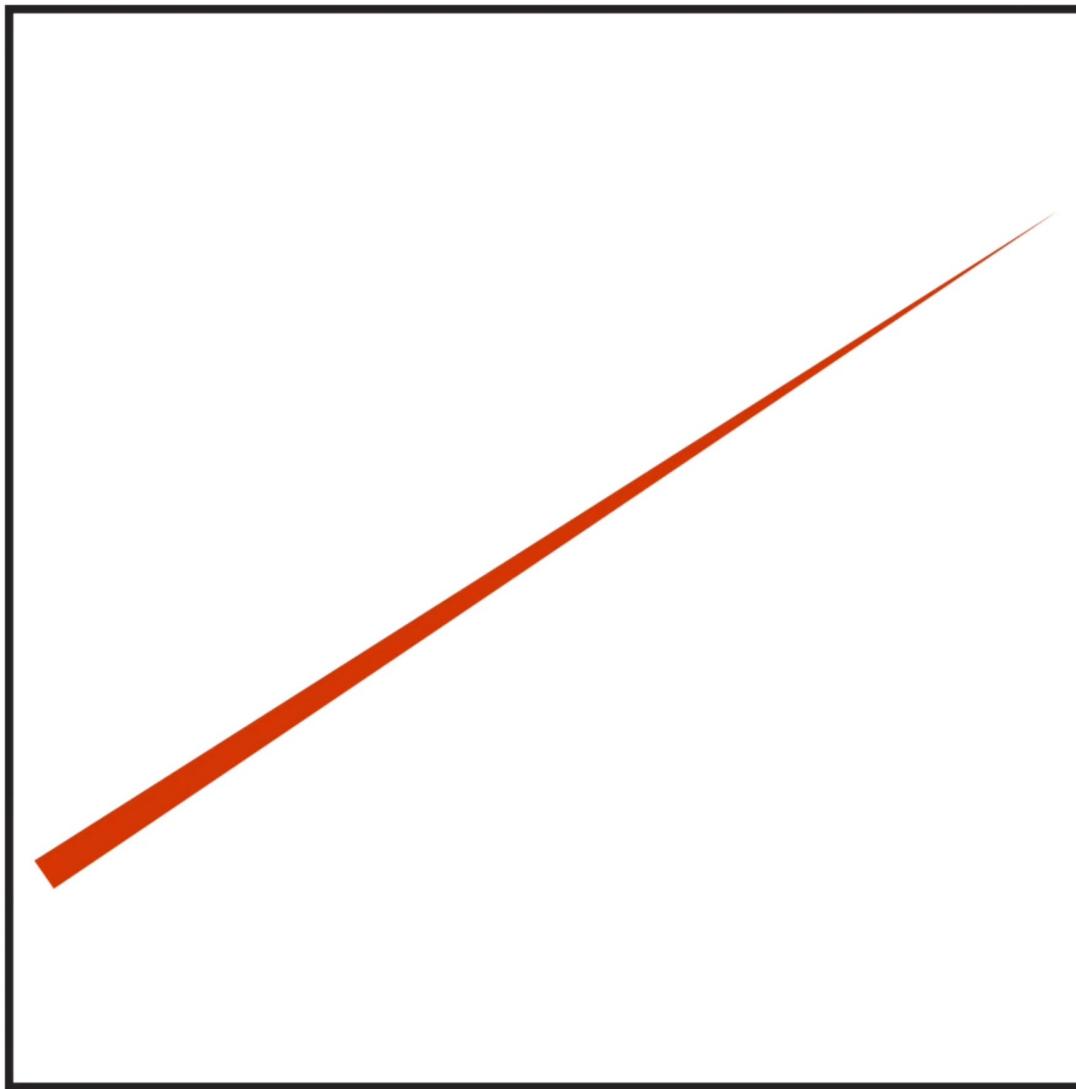
口否则，并行测试块中的每个
采样点



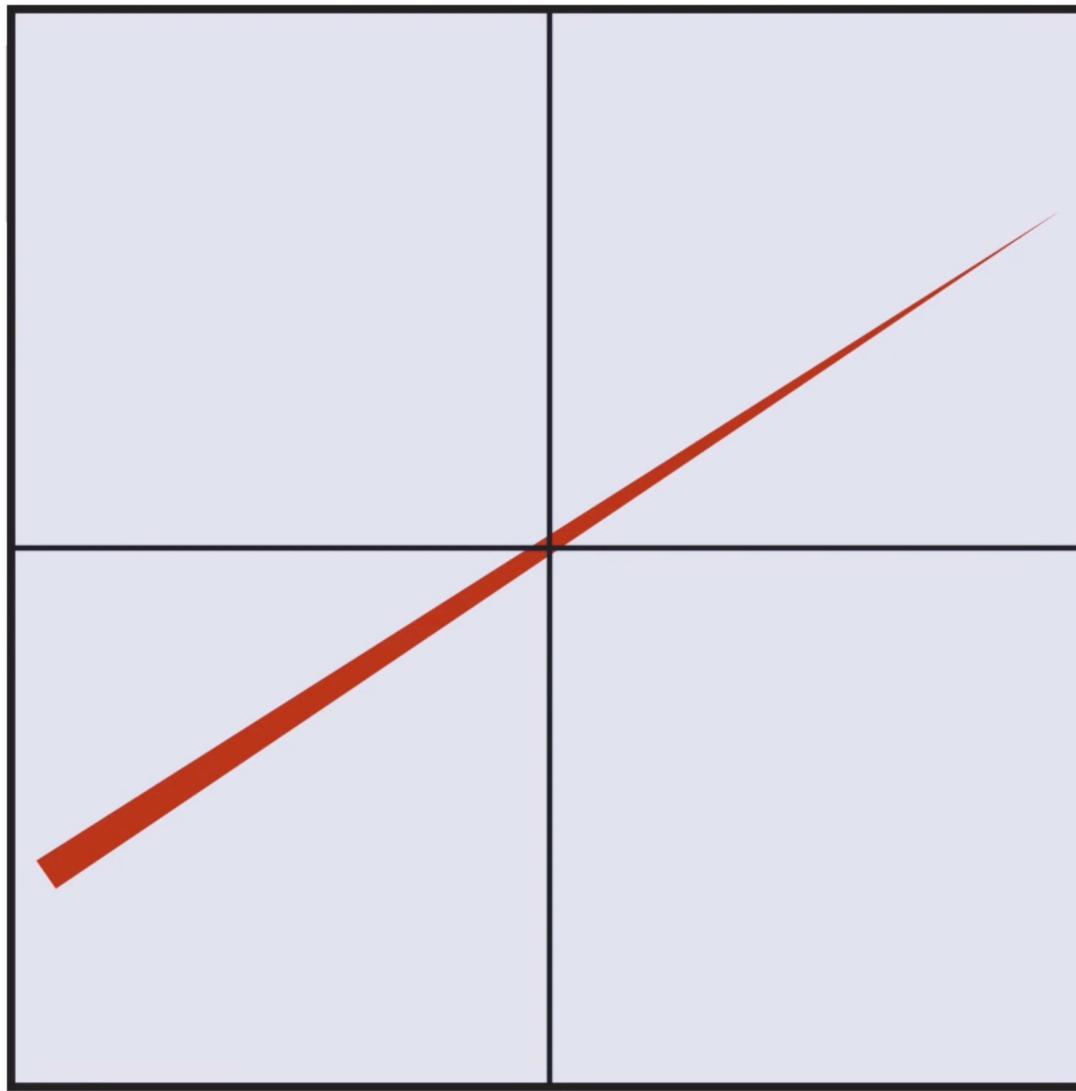
Can we do even better for this example?



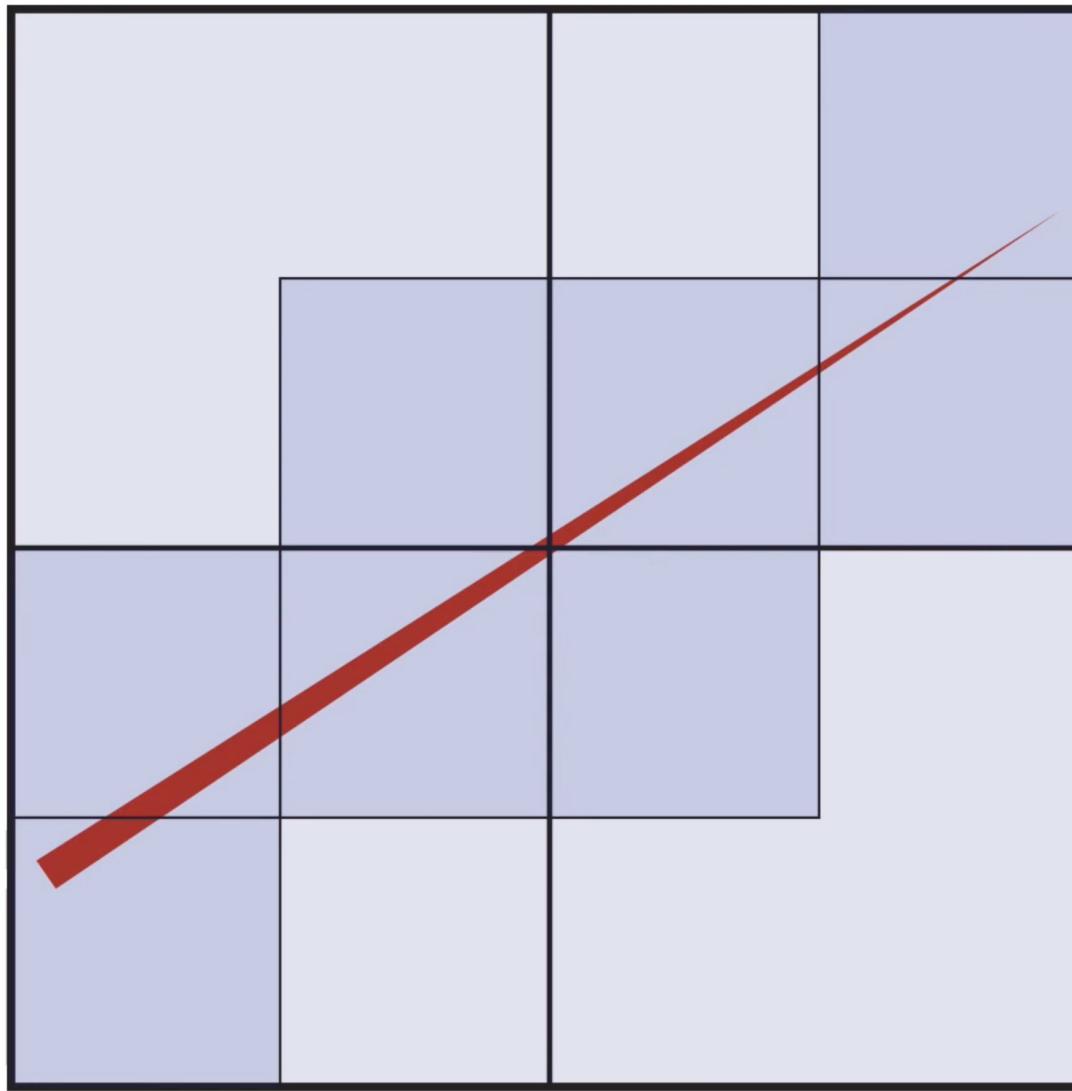
分层策略 Hierarchical strategies



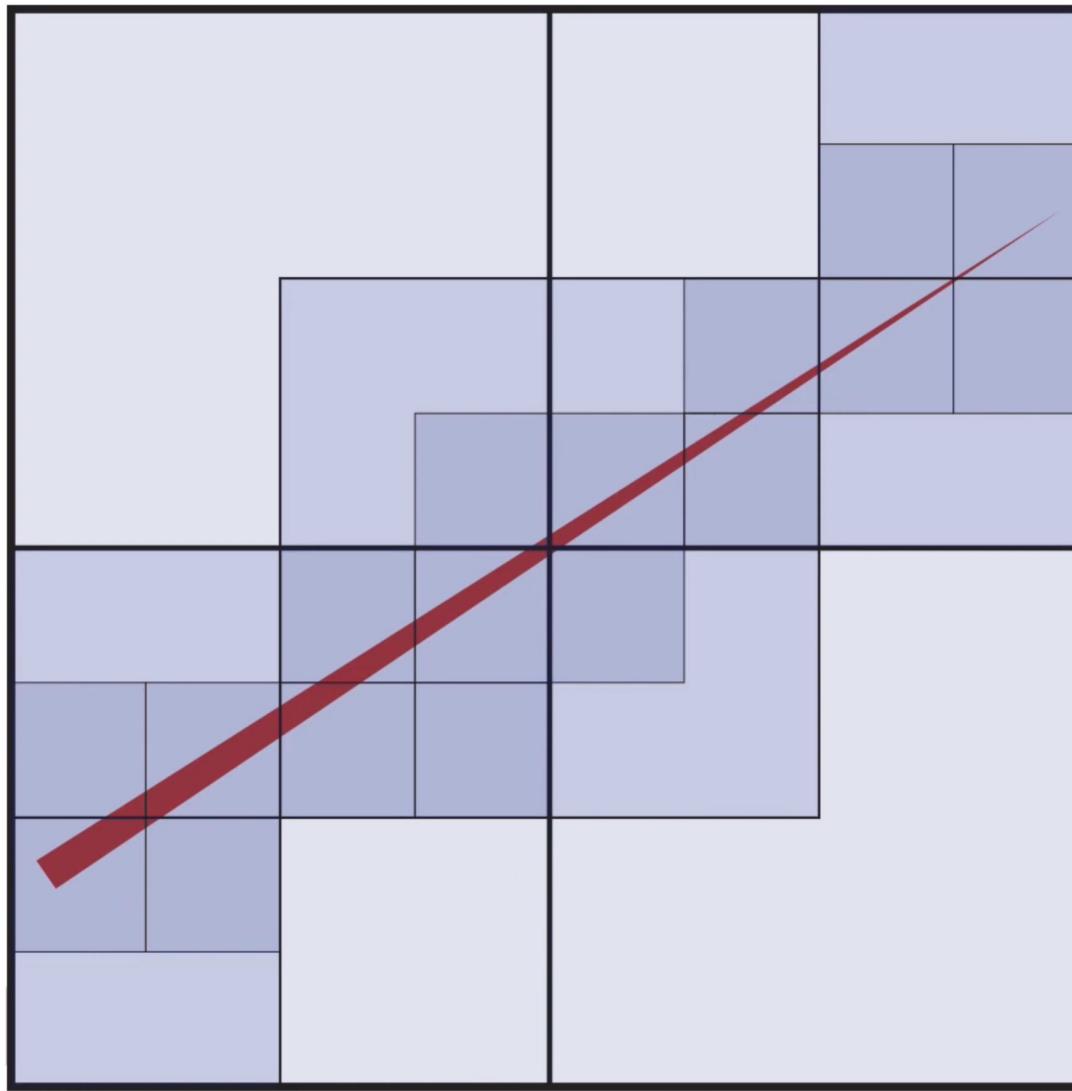
分层策略 Hierarchical strategies



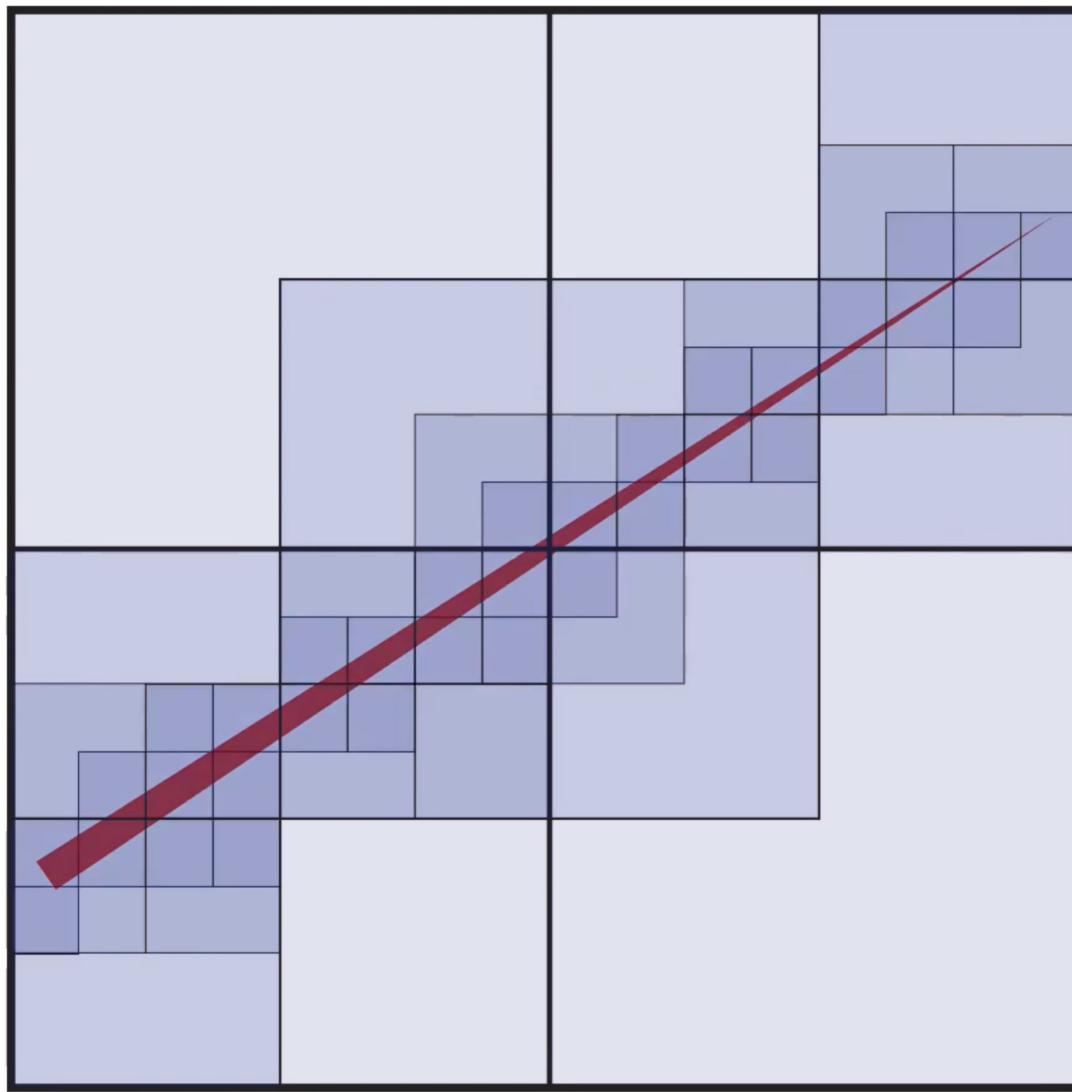
分层策略 Hierarchical strategies



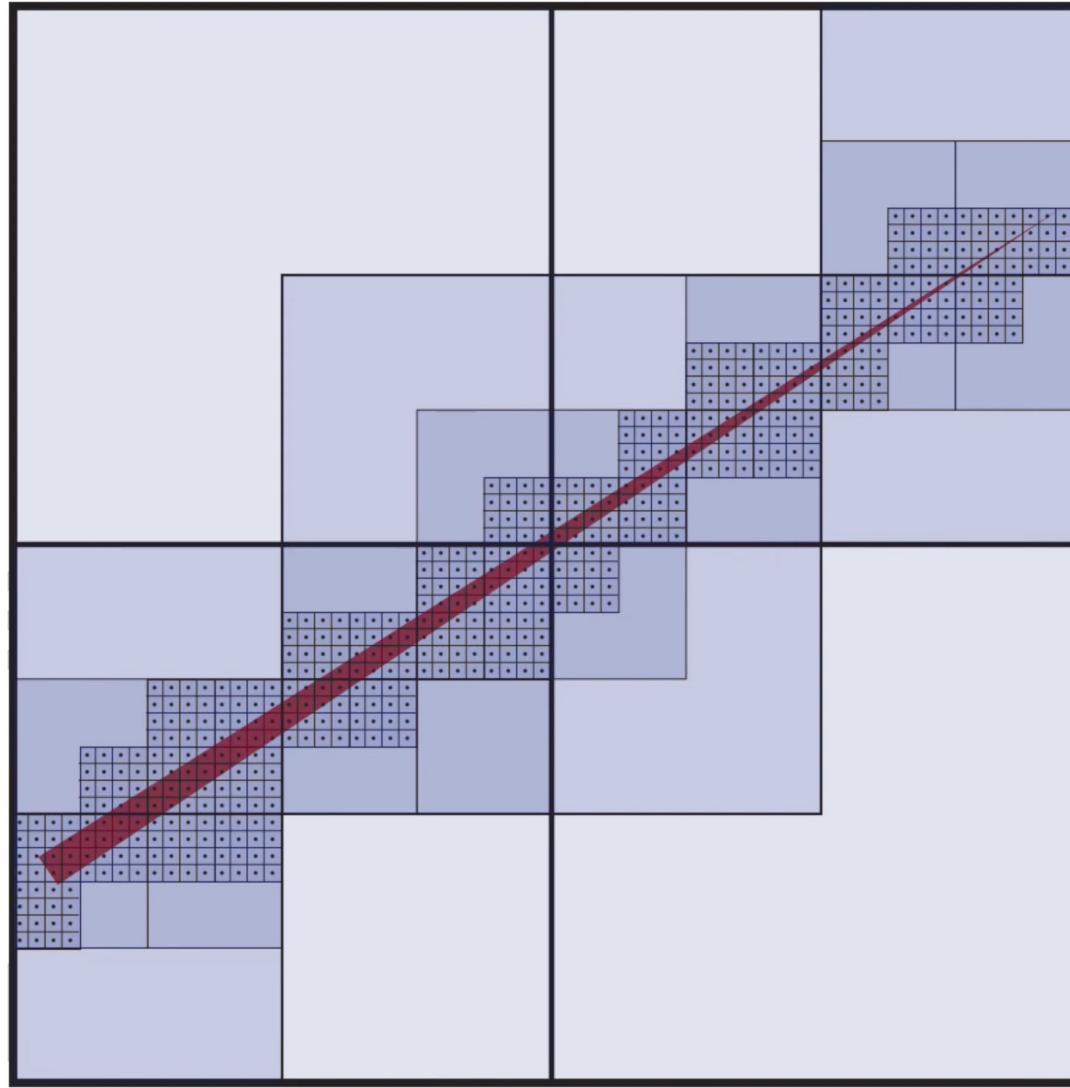
分层策略 Hierarchical strategies



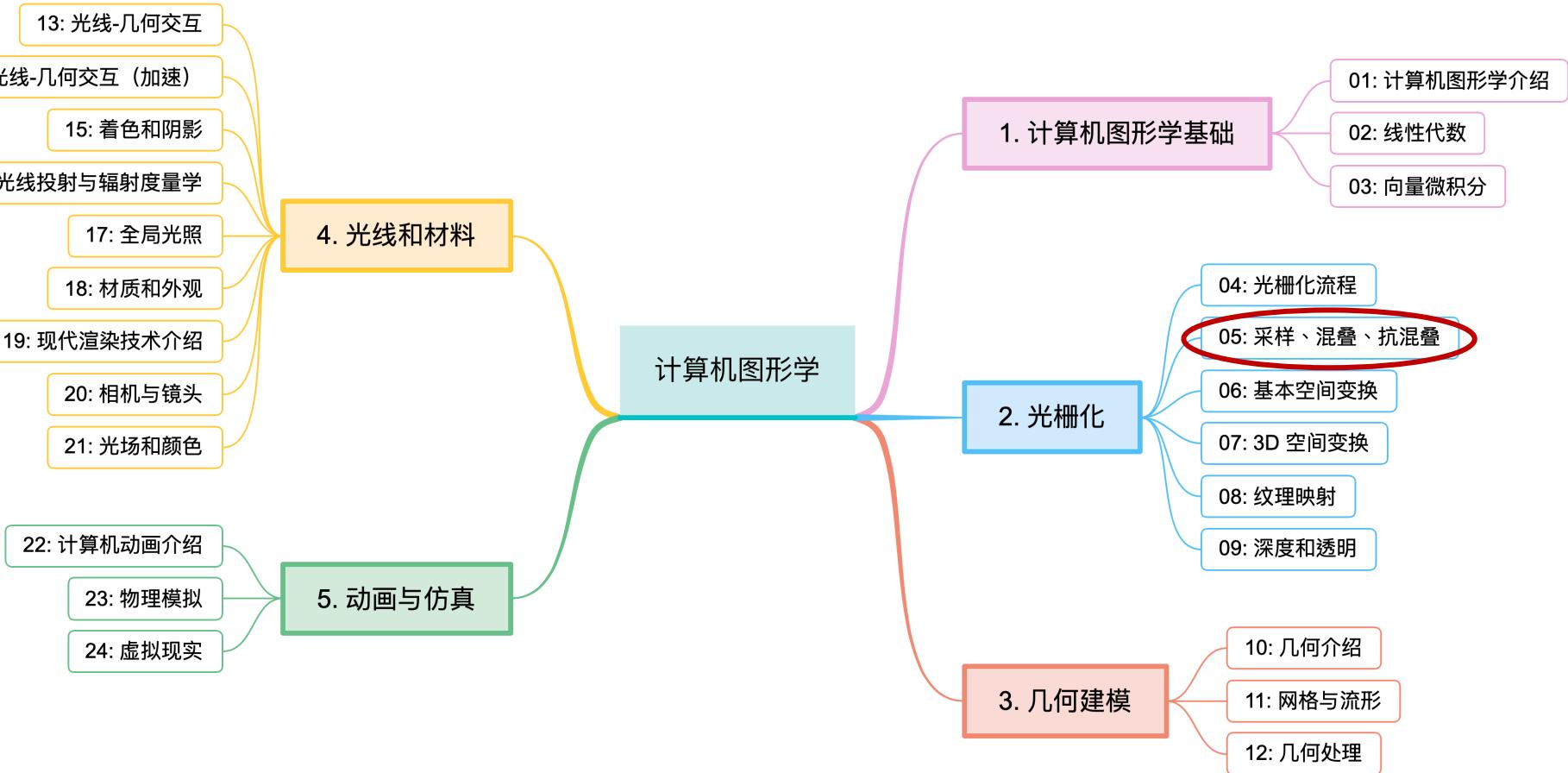
分层策略 Hierarchical strategies



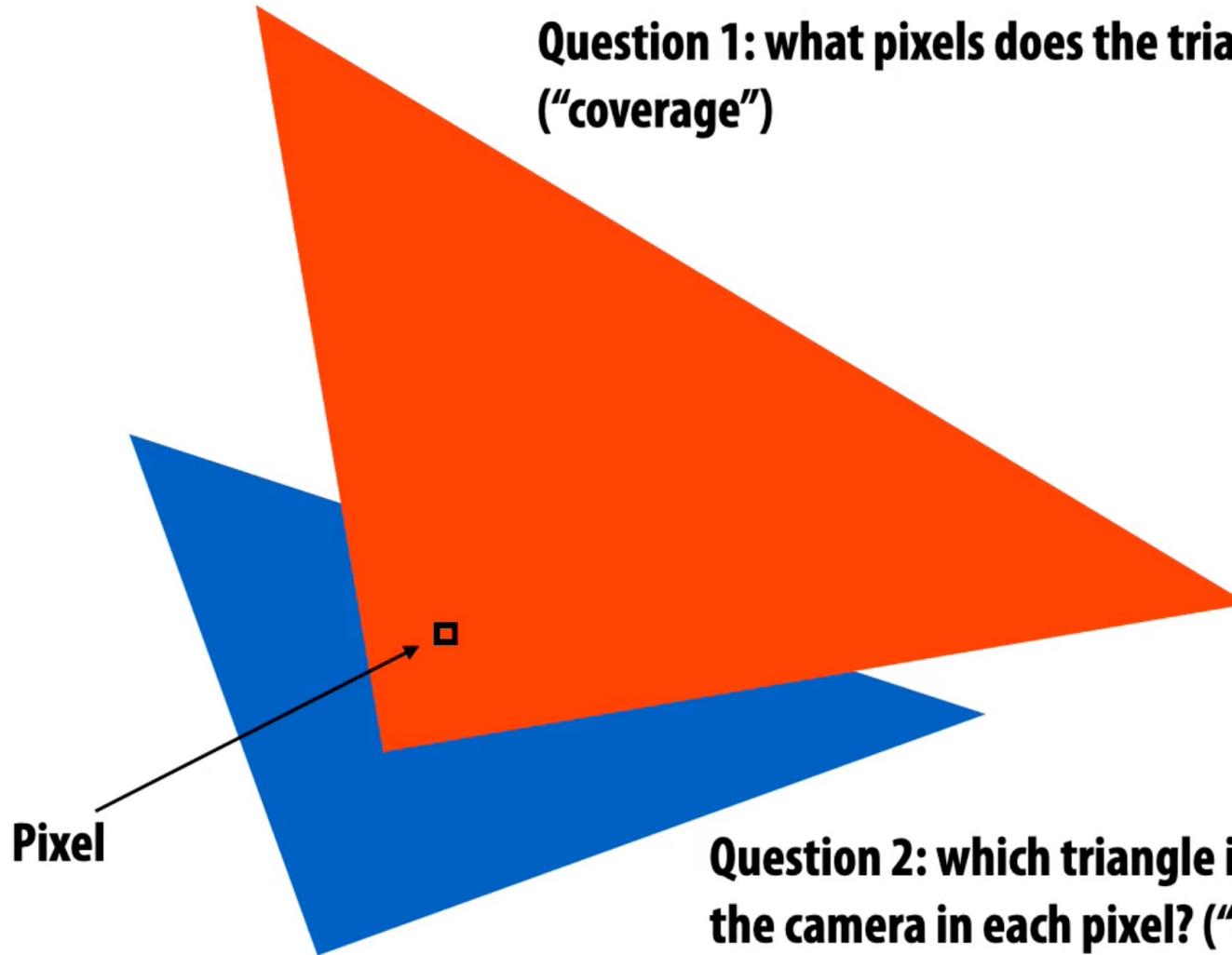
分层策略 Hierarchical strategies



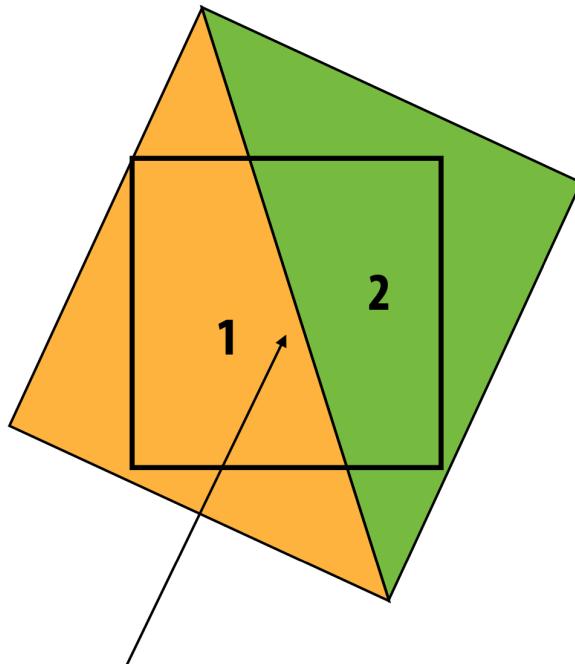
Q: Better way to find finest blocks? A: Maybe: incremental traversal!



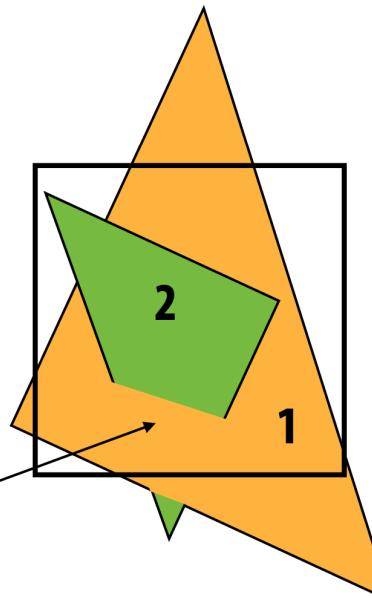
两个 (或多个) 三角形呢？



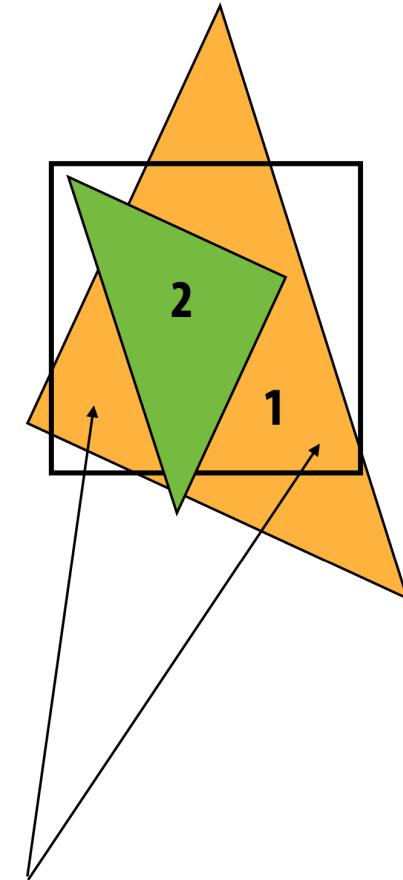
遮挡使覆盖变得更复杂



Pixel covered by triangle 1, other
half covered by triangle 2



Interpenetration of triangles: even trickier



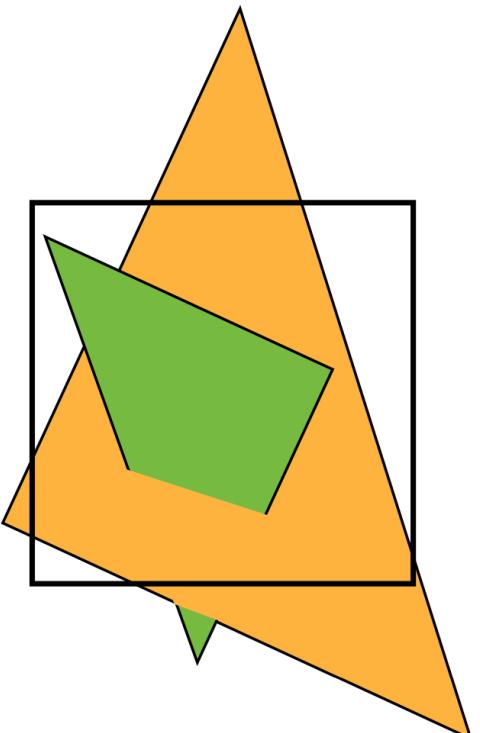
Two regions of triangle 1 contribute to pixel.
One of these regions is not even convex.

如何估计不同三角形
覆盖的百分比?

基于采样的覆盖

口真实的场景很复杂!

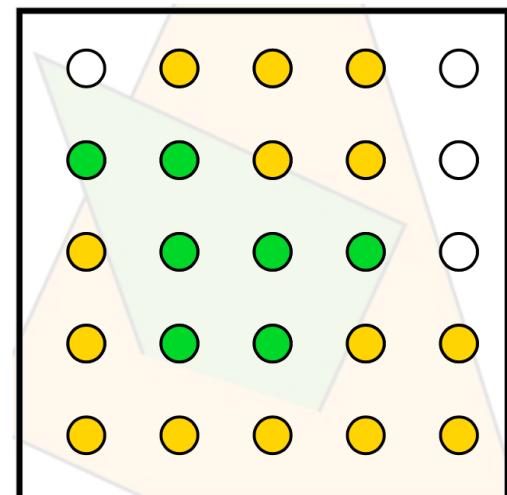
- 遮挡 occlusion、透明度 transparency...
- 以后将介绍更多



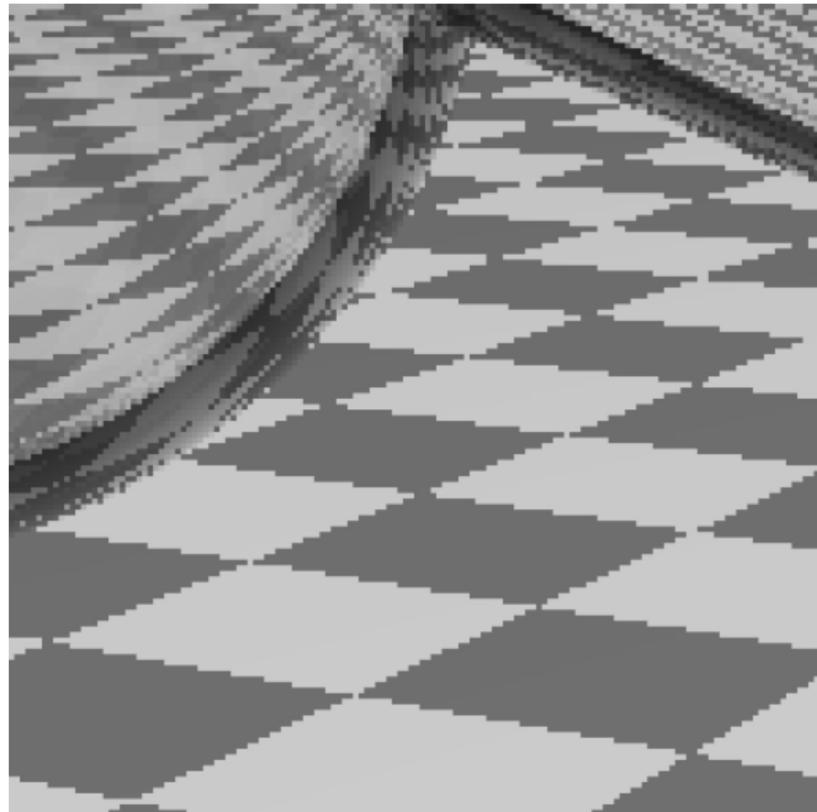
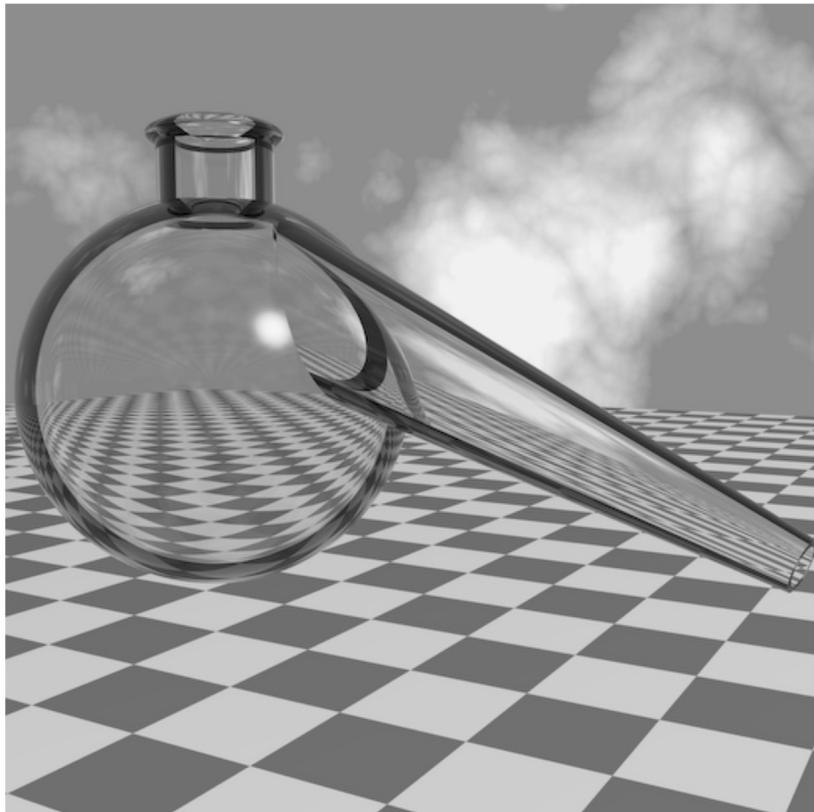
口计算准确的覆盖率是不现实的

口相反：将覆盖率视为采样问题

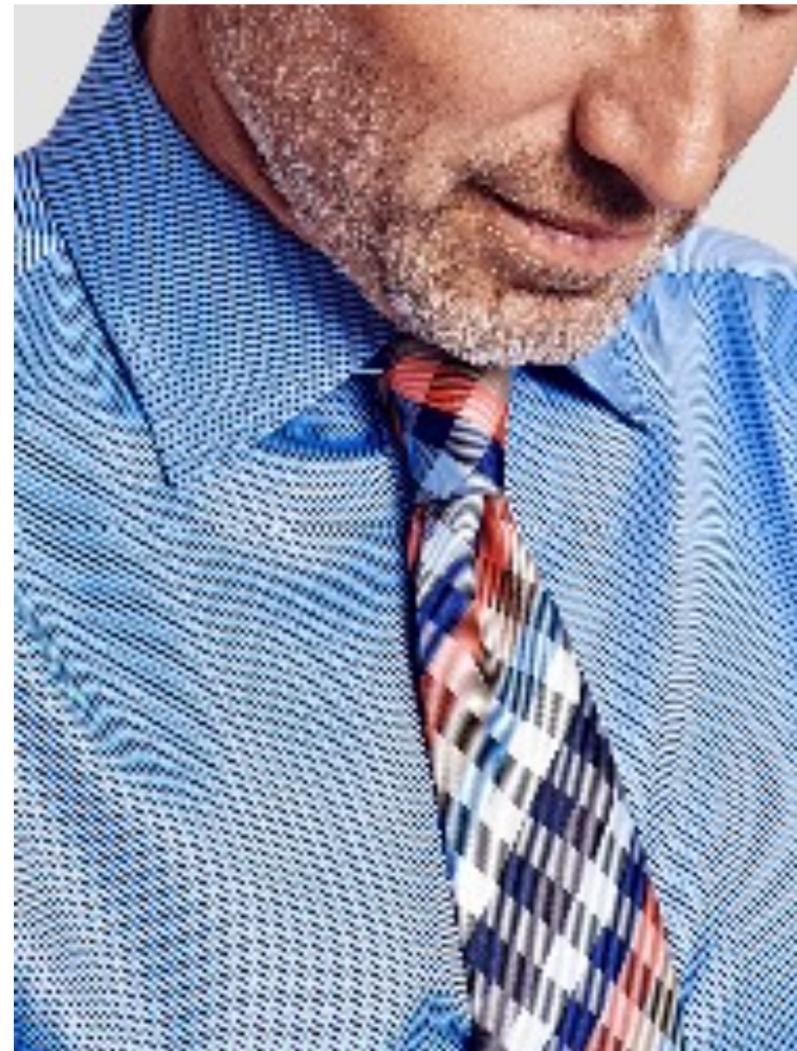
- 不要计算 exact/analytical answer
- 相反地，测试一组采样点
- 有了足够的样本点及合适的位置选择，
就可以开始得到一个很好的估计



Jaggies (楼梯样式)

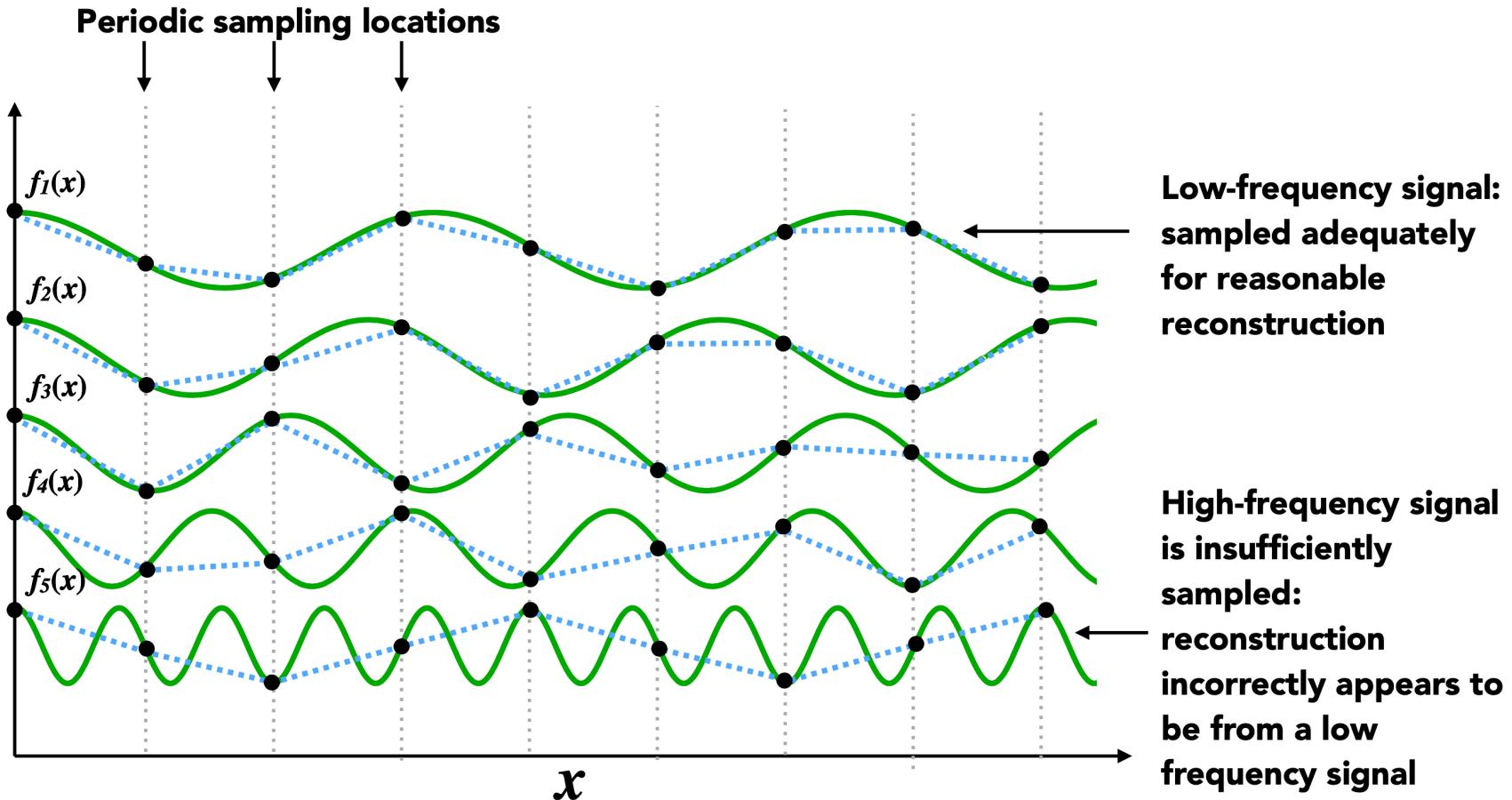


摩尔条纹 Moiré Patterns

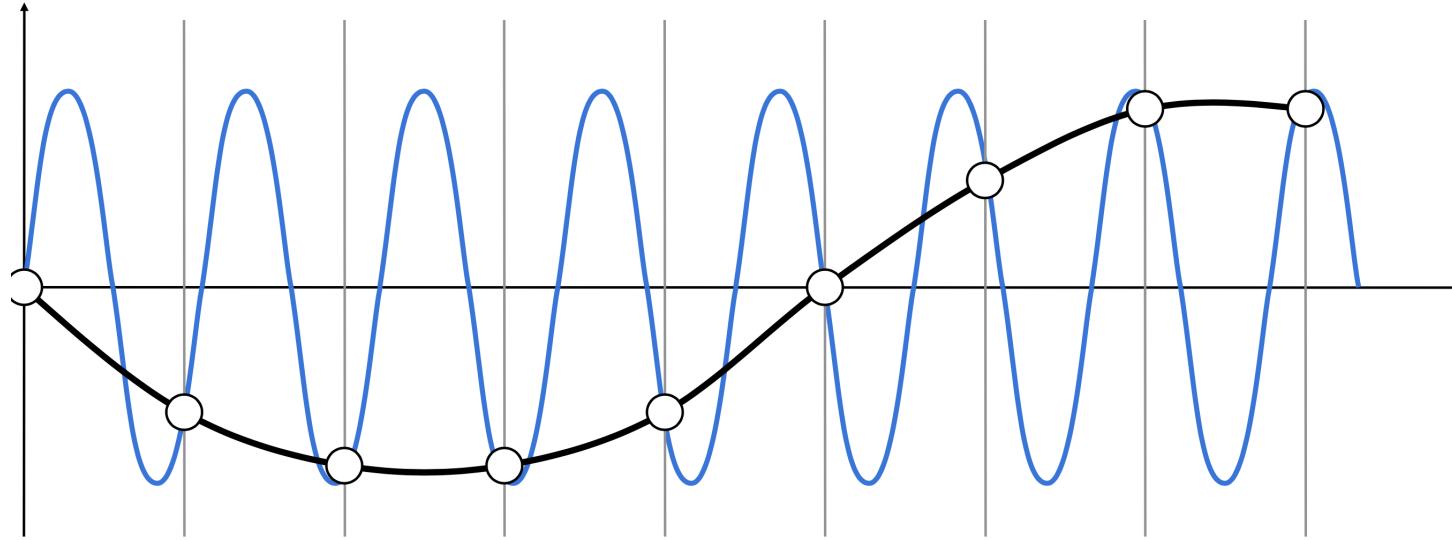


走样
Aliasing

高频信号采样不足会导致走样

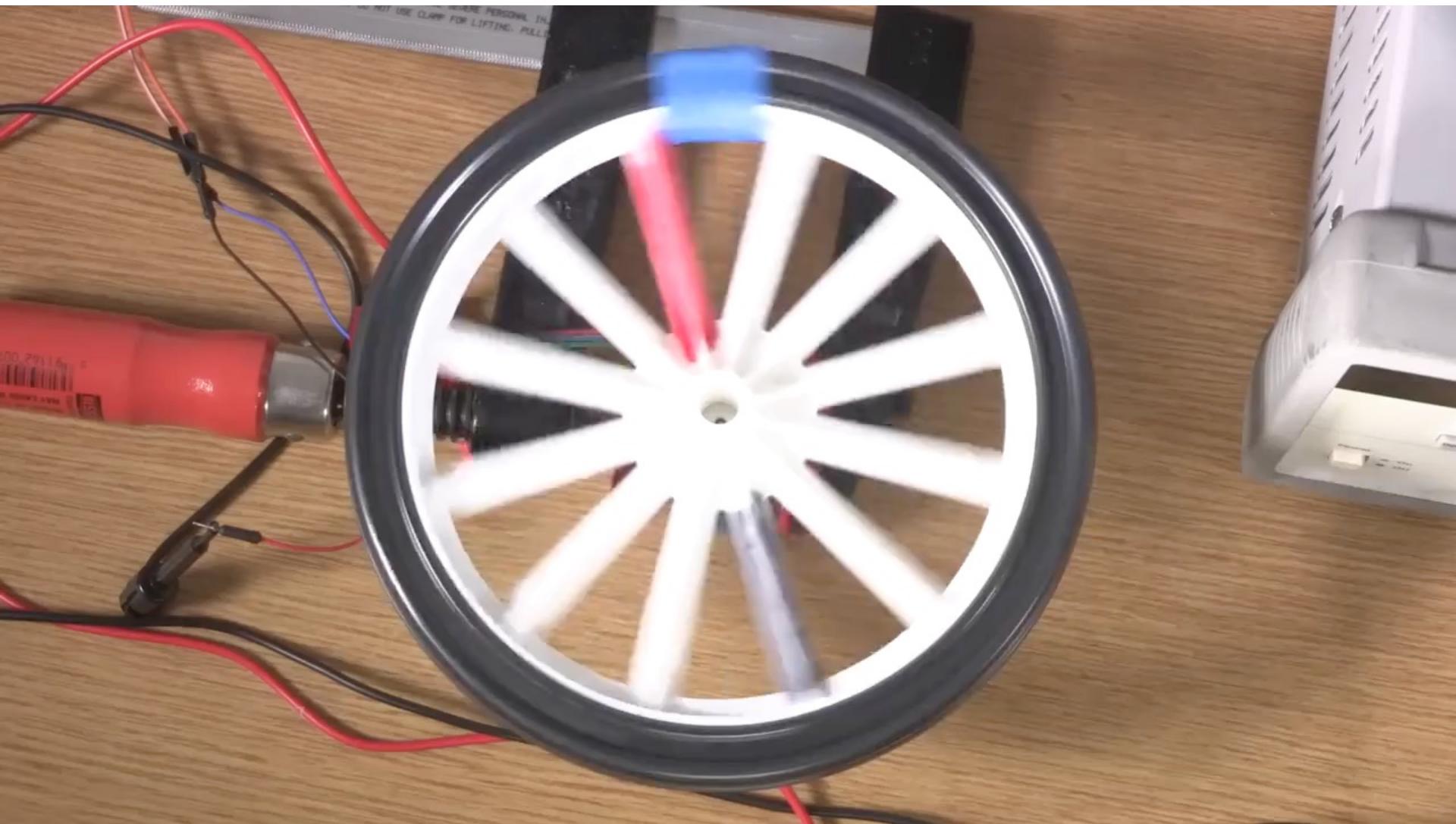


欠采样导致走样



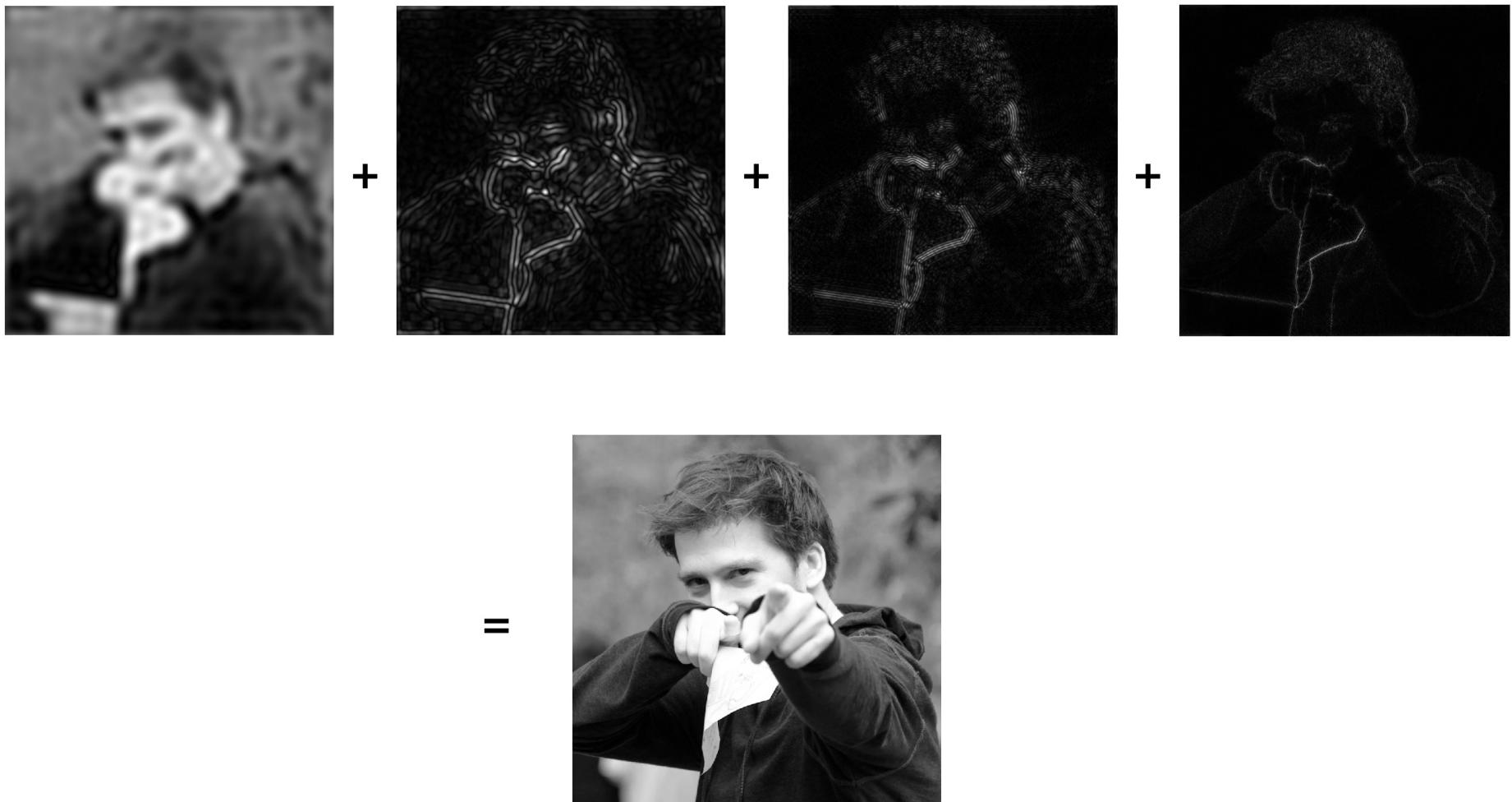
□ 高频信号采样不足：样本错误地看起来来自低频信号
□ 在给定的采样率下无法区分的两个频率被称为
走样 aliasing

时间走样：车轮效果



对于快速旋转的车轮，相机的帧率（时间采样率）太低

图像分解为不同频率的部分



尼奎斯特定理 Nyquist Theorem

若信号的频率低于尼奎斯特频率（定义为采样频率的一半），则不会出现走样现象。

Theorem: We get no aliasing from frequencies in the signal that are less than the Nyquist frequency (which is defined as half the sampling frequency) *

如何减少走样？

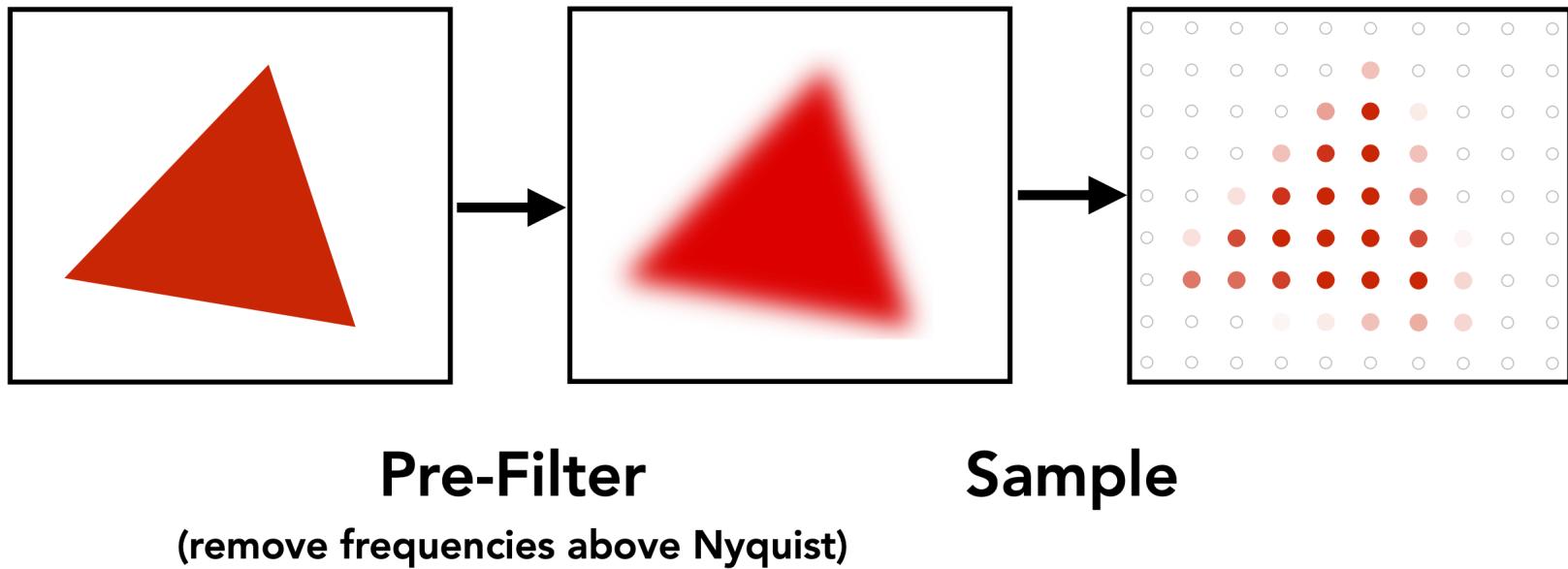
口增加采样率（增加奈奎斯特频率）

- 更高分辨率的显示器、传感器、帧缓冲区...
- 但是：成本高昂，可能需要非常高的分辨率

口抗走样

- 在采样前去除（或降低）超过尼奎斯特频率的信号
- 怎样在采样前滤除高频？

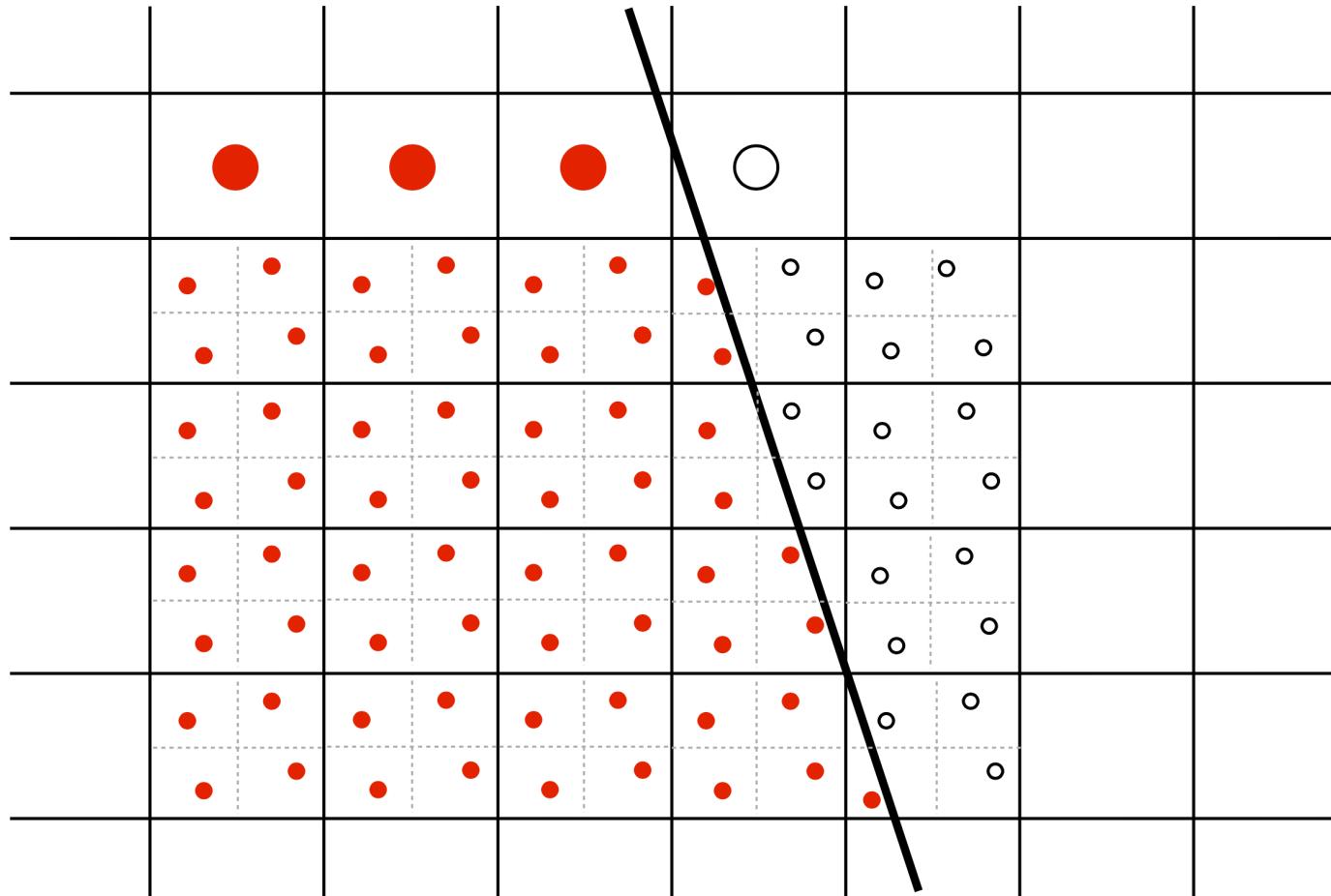
抗走样的采样



注意光栅化三角形中锯齿的值为红色与白色的中间值

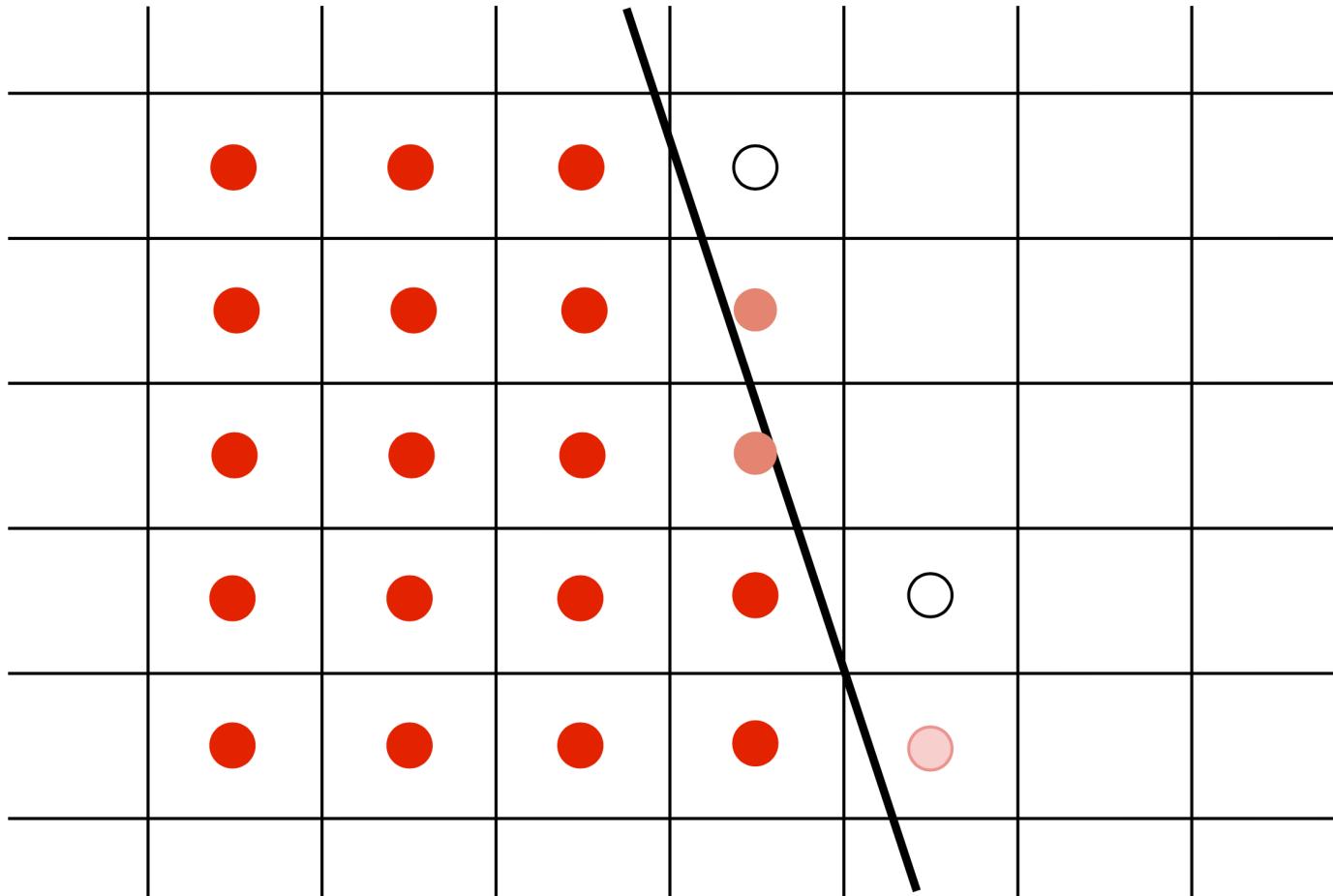
超采样

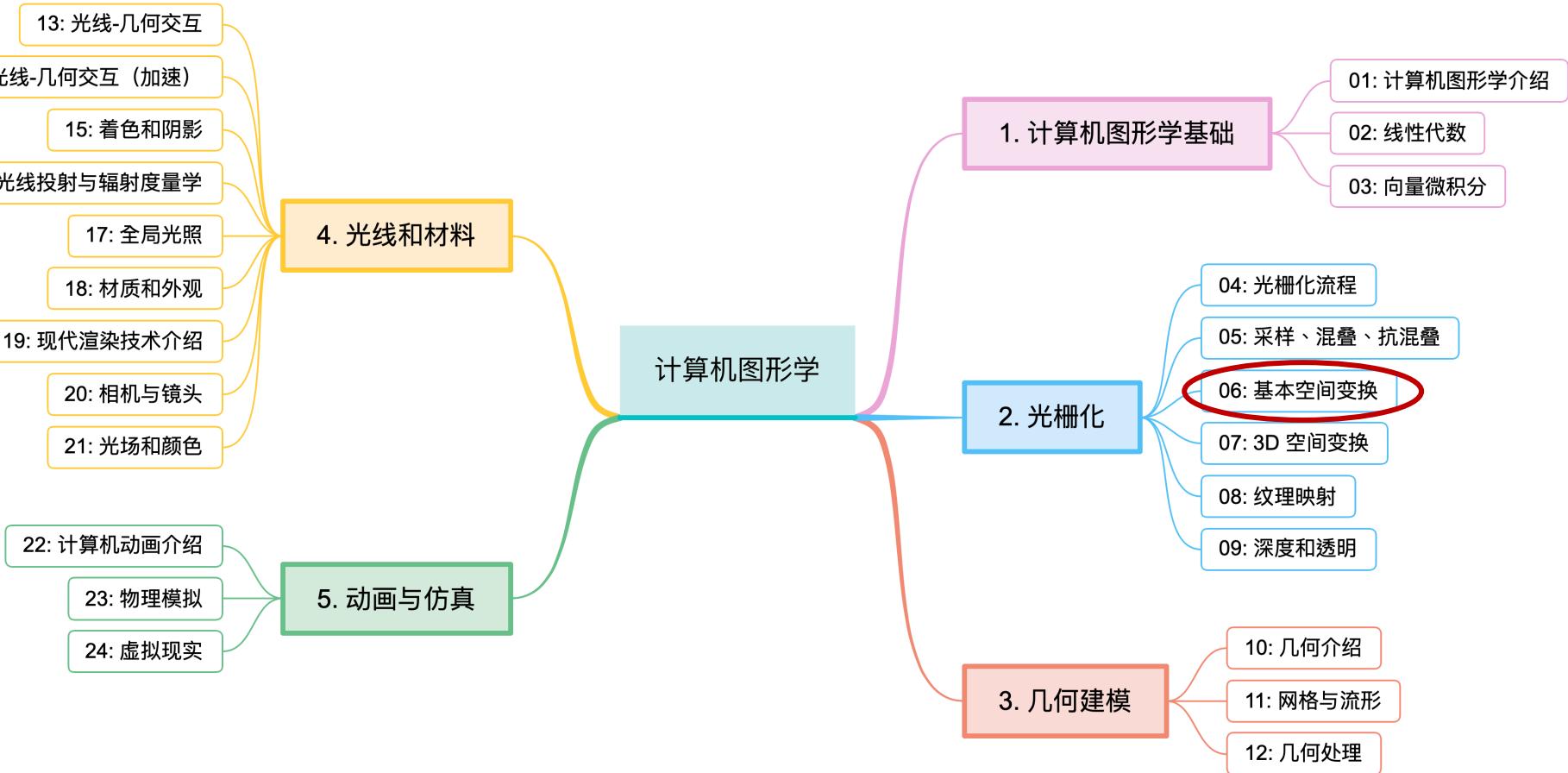
对每个像素里的 $n \times n$ 个样本取平均



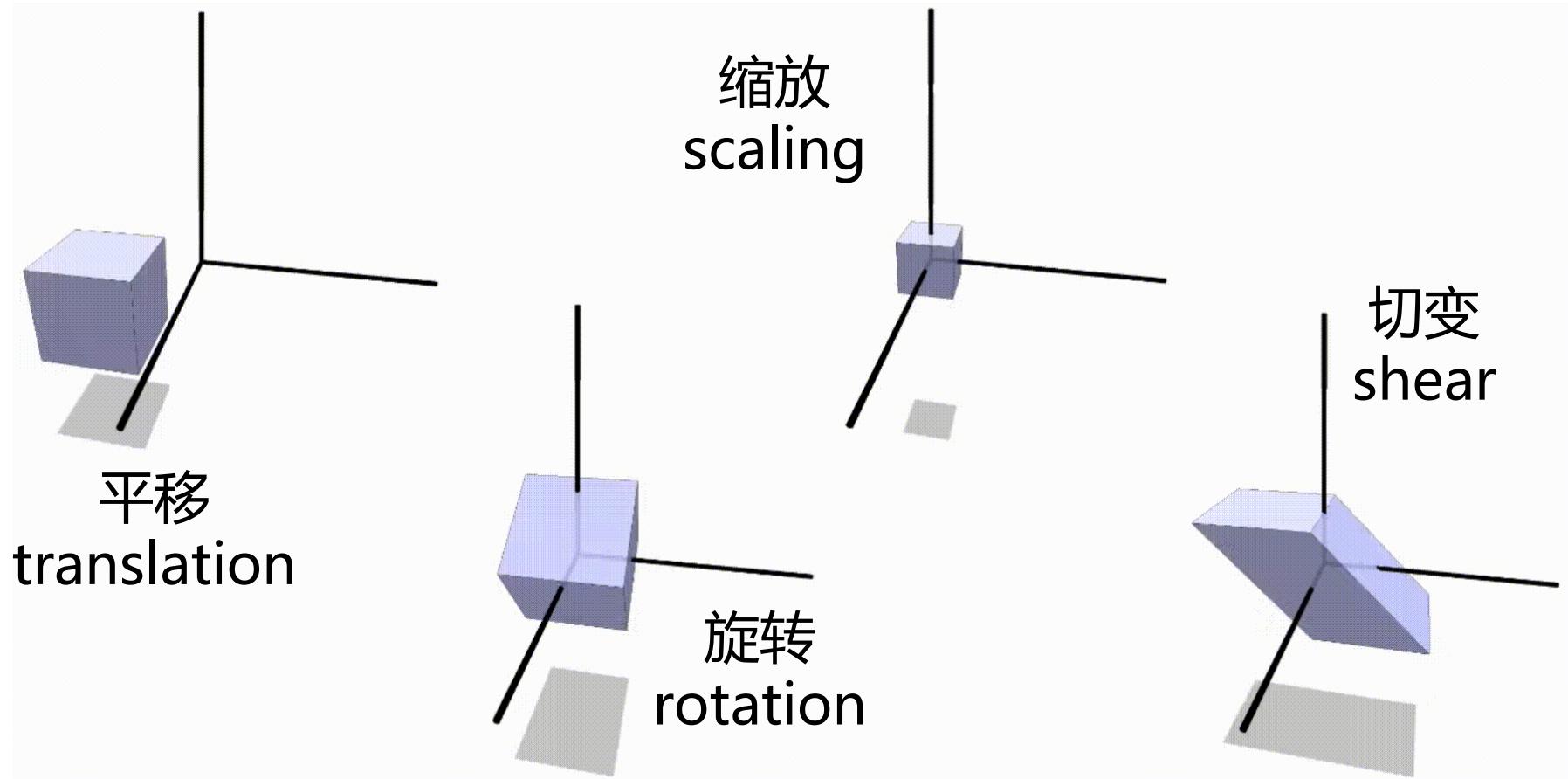
超采样

对每个像素里的 $n \times n$ 个样本取平均



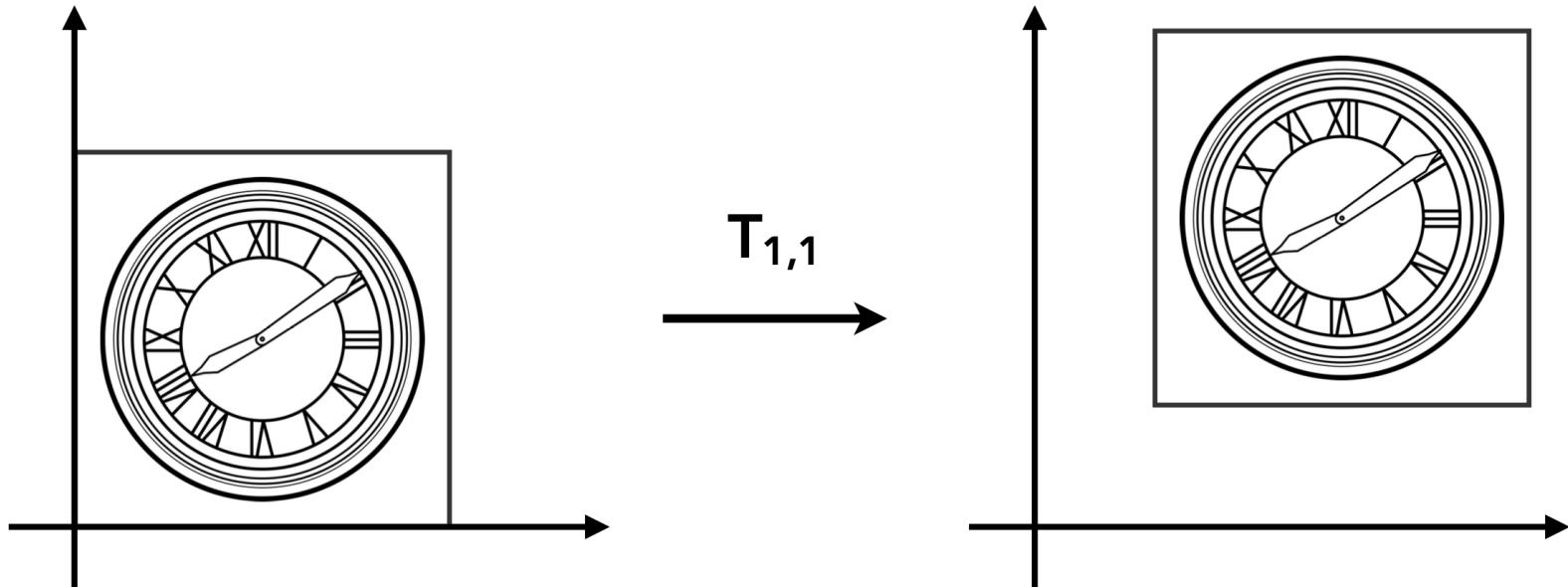


变换的种类



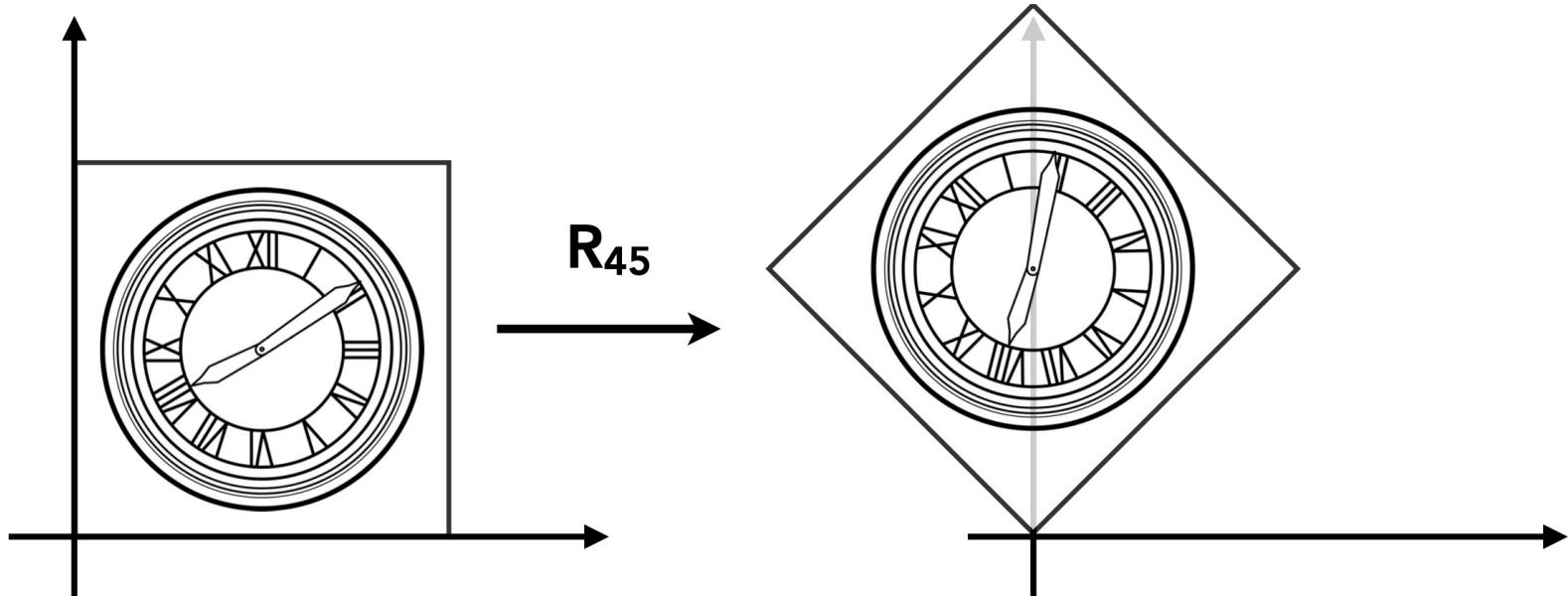
2D 视角的变换

□ 平移 Translation



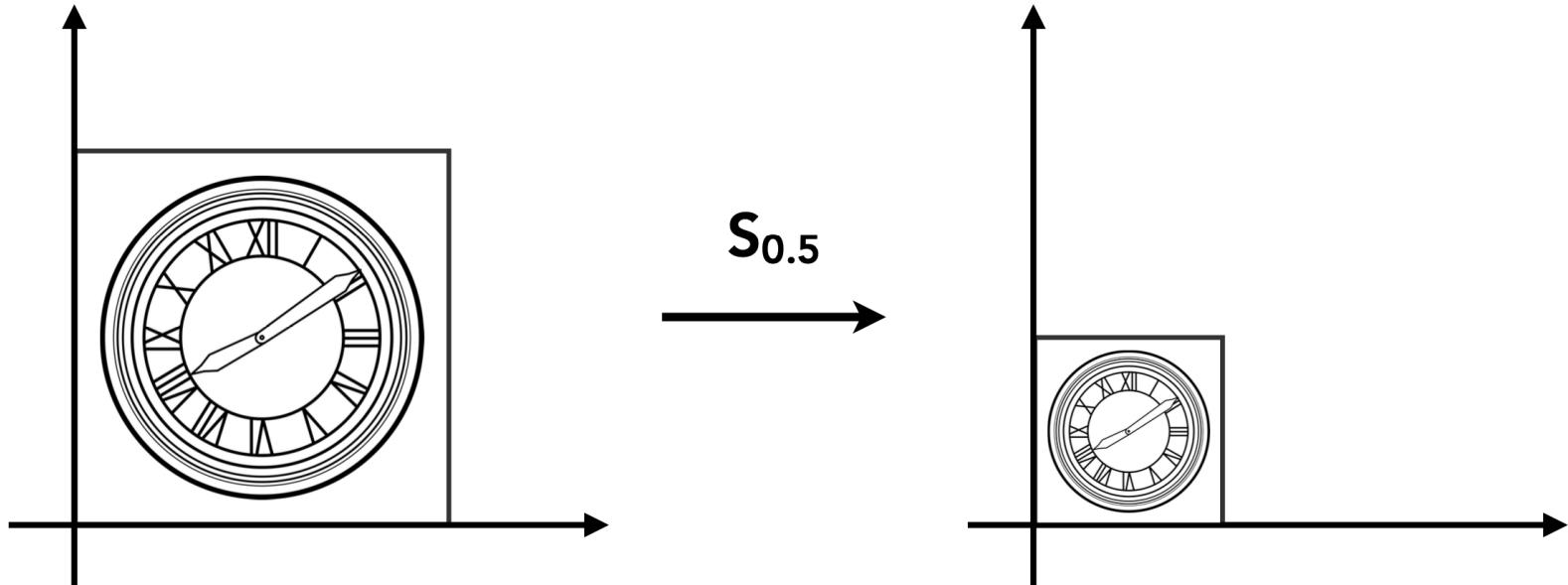
2D 视角的变换

□ 旋转 Rotation



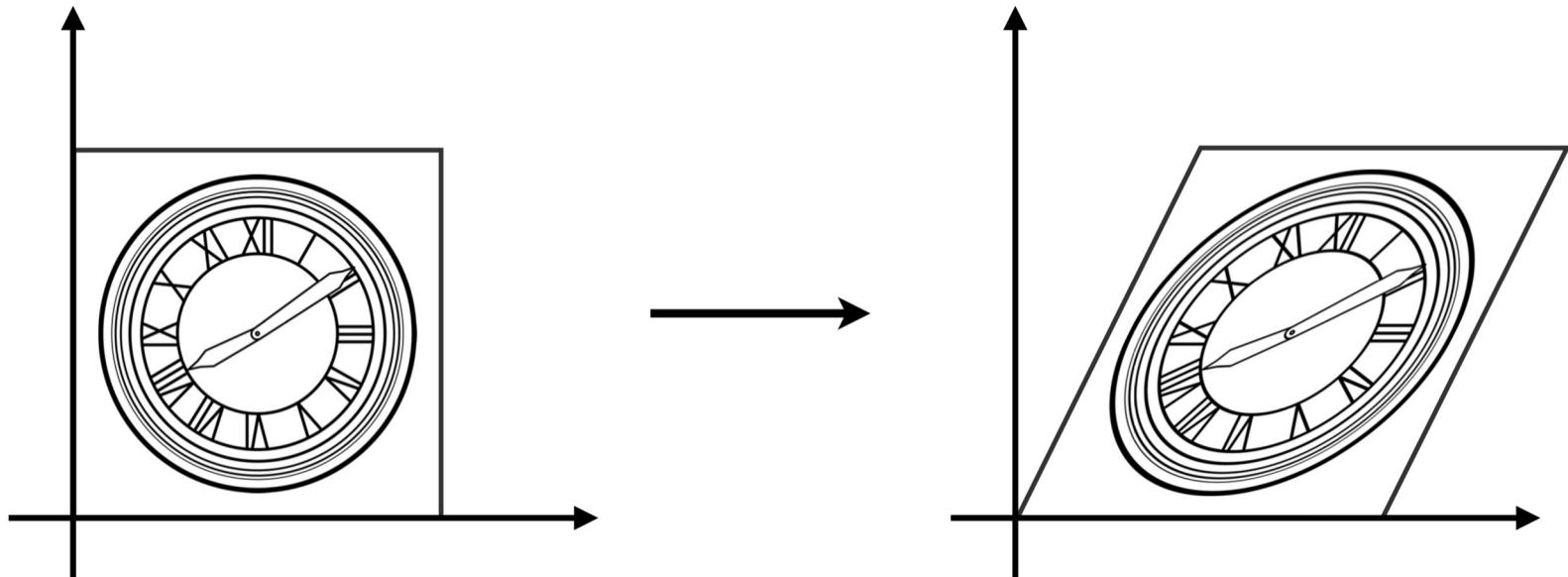
2D 视角的变换

□缩放 Scaling



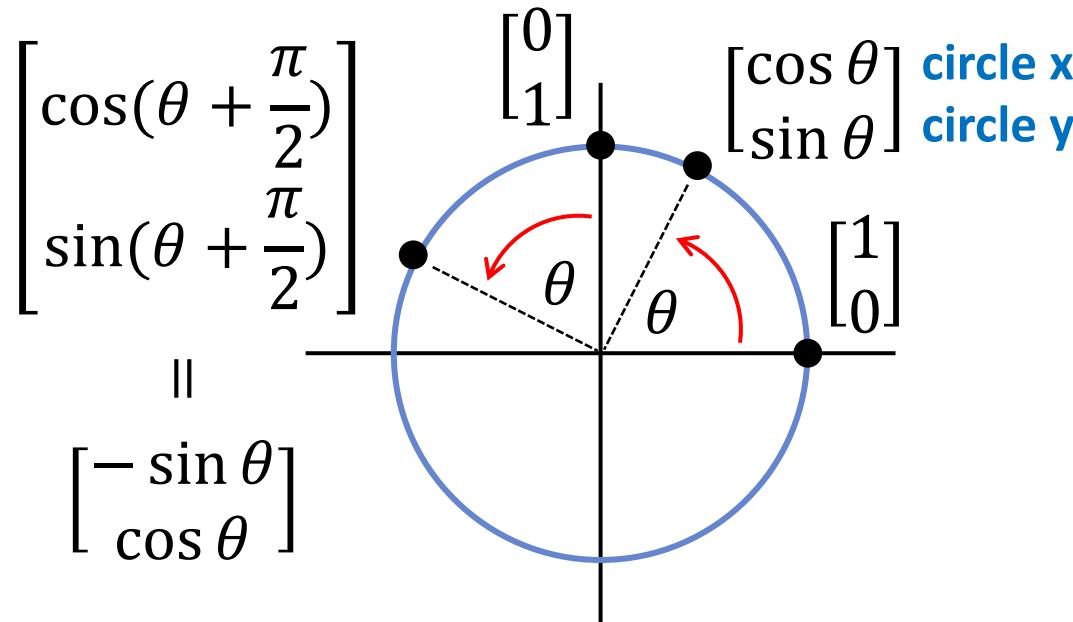
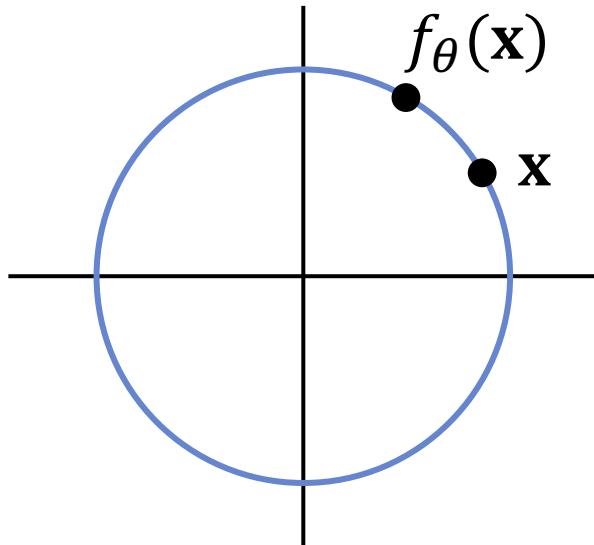
2D 视角的变换

口切变 Shear



2D 旋转 – 矩阵表示

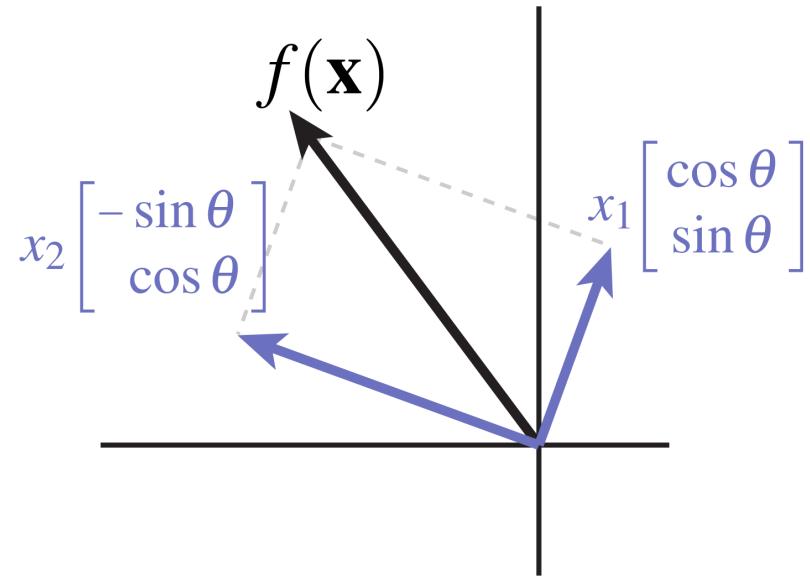
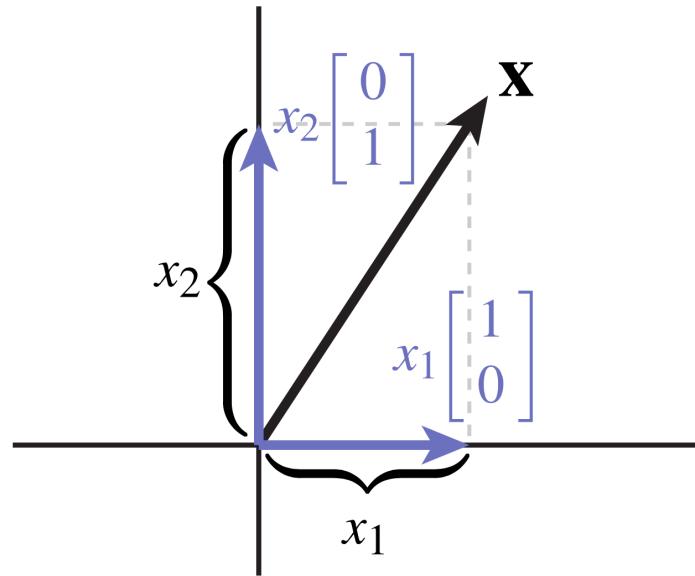
□ 旋转保留了长度和原点，因此，角度为 θ 的 2D 旋转将点 \mathbf{x} 映射到点 $f_\theta(\mathbf{x})$ ，且 $f_\theta(\mathbf{x})$ 处在半径为 $|\mathbf{x}|$ 的圆上



- 如果我们逆时针旋转 θ , $\mathbf{x} = (1, 0)$ 将映射到哪个点?
□ $\mathbf{x} = (0, 1)$ 呢?

更一般的向量 $\mathbf{x} = (x_1, x_2)$ 呢?

2D 旋转 – 矩阵表示



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$f(\mathbf{x}) = x_1 \begin{bmatrix} \cos \theta \\ \sin \theta \end{bmatrix} + x_2 \begin{bmatrix} -\sin \theta \\ \cos \theta \end{bmatrix}$$

□ 我们要如何用矩阵表示一个 2D 旋转函数 $f_\theta(x)$?

即用矩阵表示
一个线性映射

$$f_\theta(\mathbf{x}) = \begin{bmatrix} \cos \theta & -\sin(\theta) \\ \sin \theta & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

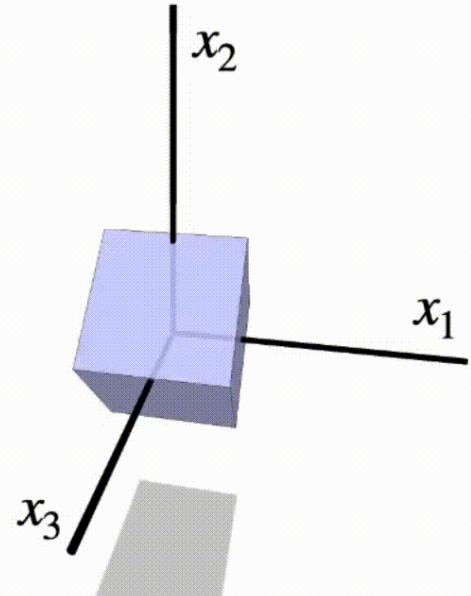
3D 旋转

□Q: 在 3D 中, 我们如何沿着 x_3 坐标轴旋转?

□A: 对 x_1 和 x_2 应用相同的转换, 保持 x_3 不变

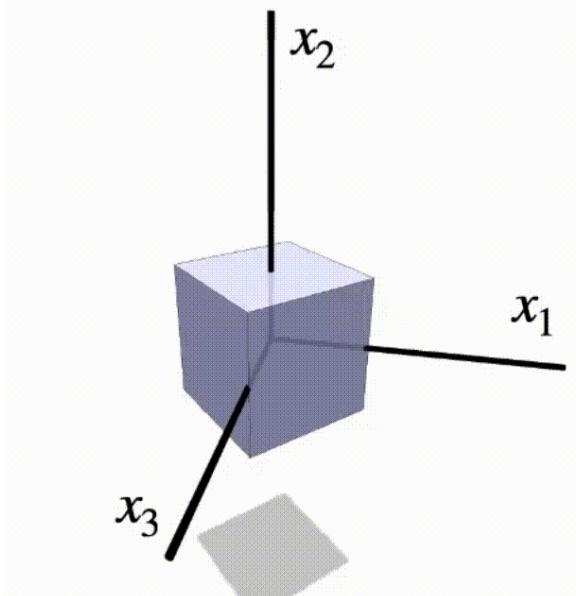
rotate around x_1

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin(\theta) \\ 0 & \sin \theta & \cos(\theta) \end{bmatrix}$$



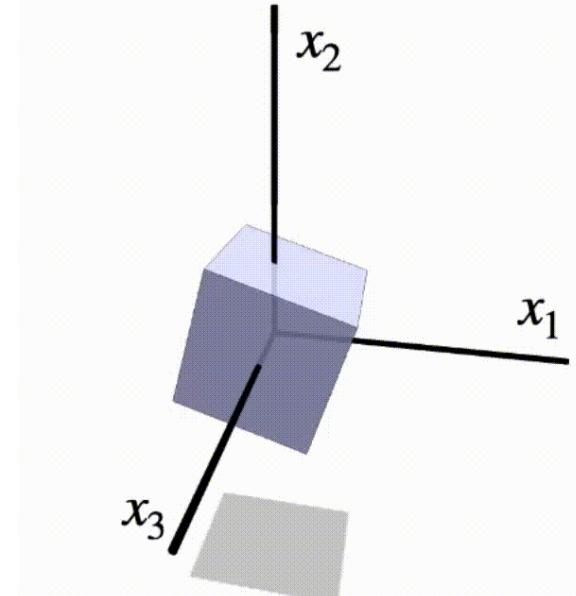
rotate around x_2

$$\begin{bmatrix} \cos \theta & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos(\theta) \end{bmatrix}$$



rotate around x_3

$$\begin{bmatrix} \cos \theta & -\sin(\theta) & 0 \\ \sin \theta & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



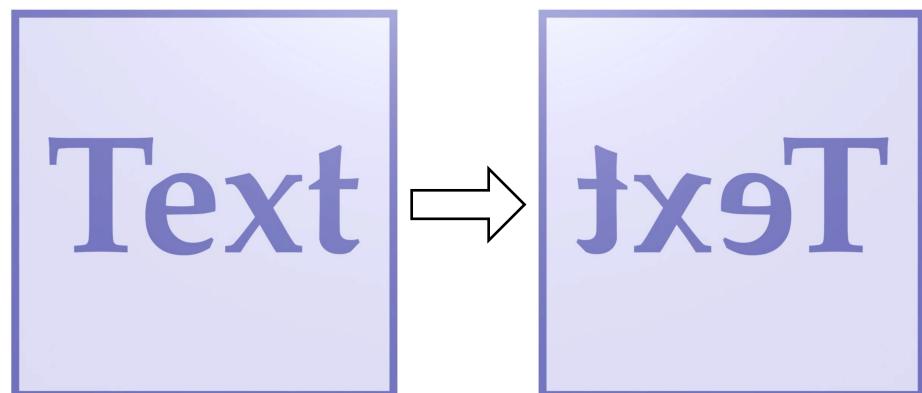
反射 Reflection

- Q: 是否所有满足 $Q^T Q = I$ 的正交矩阵均描述了一个旋转?
- 旋转必须保持**原点 origin**、**距离 distance** 和**方位 orientation**
- 考虑如下的矩阵

$$Q = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad Q^T Q = \begin{bmatrix} (-1)^2 & 0 \\ 0 & 1 \end{bmatrix} = I$$

- Q: 这个矩阵是否描述了一个旋转? 如果不是, 哪个量未能保持?

- A: No! 它代表了一个沿着 y 轴的反射, 因此未能保持方位

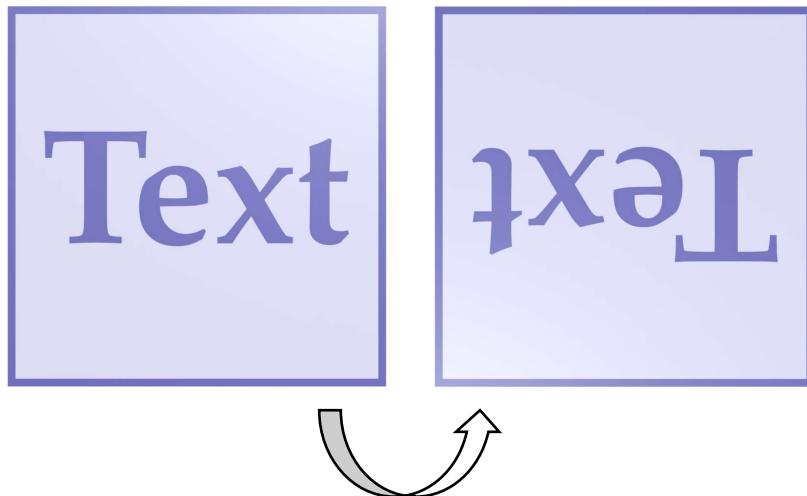


正交变换 Orthogonal transformations

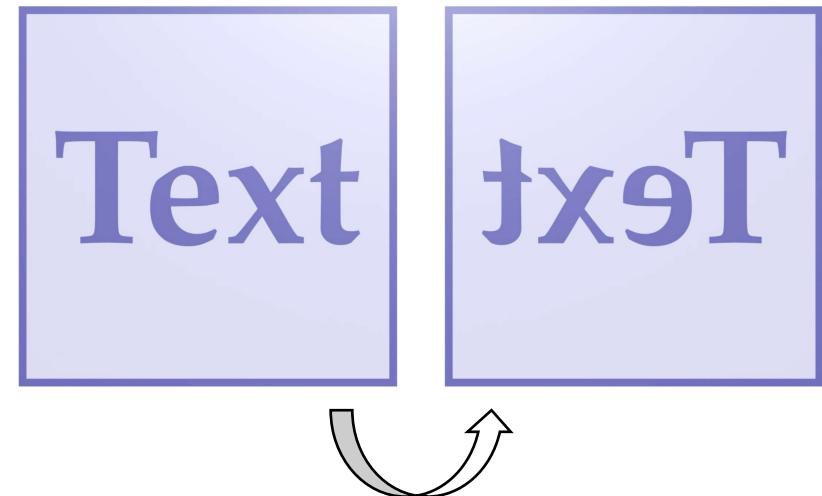
保持 **距离 distance** 和 **原点 origin** 的变换称为正交变换

用矩阵 $Q^T Q = I$ 表示

- **Rotations** additionally preserve orientation: $\det(Q) > 0$
- **Reflections** reverse orientation: $\det(Q) < 0$



rotation



reflection

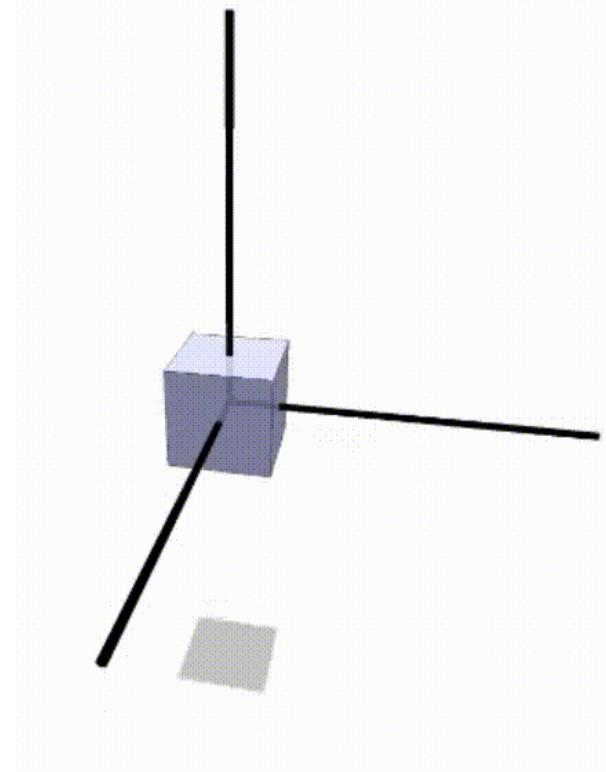
缩放 Scaling

□ 每个向量 \mathbf{u} 映射到标量倍数

$$f(\mathbf{u}) = a\mathbf{u}, \quad a \in \mathbb{R}$$

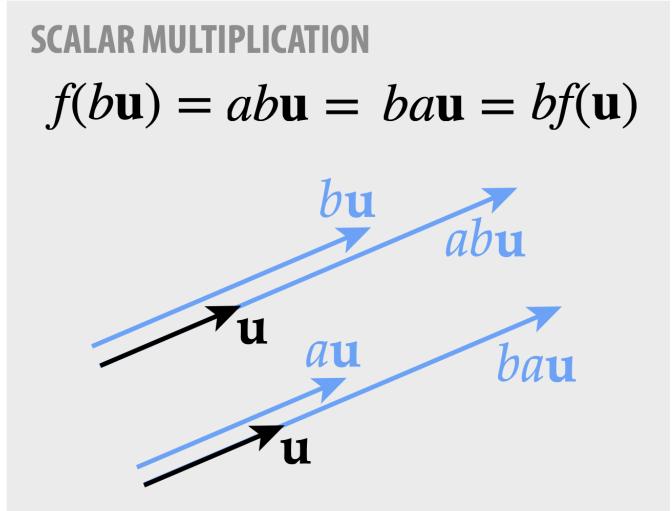
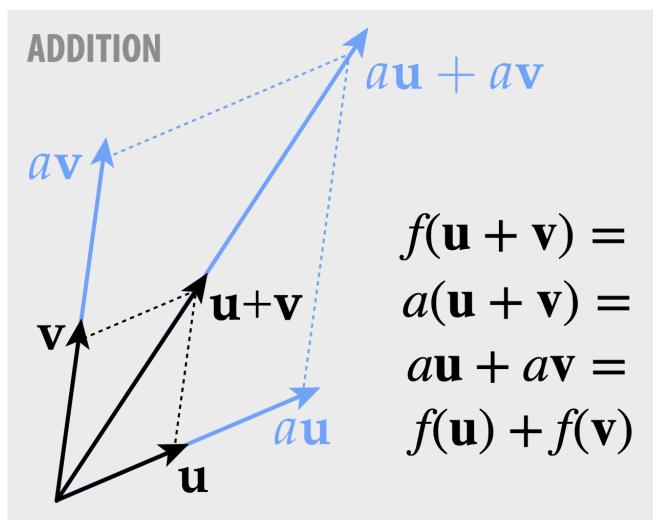
□ 保持了所有向量的方向 direction*

$$\frac{\mathbf{u}}{|\mathbf{u}|} = \frac{a\mathbf{u}}{|a\mathbf{u}|} \quad * \text{假定 } a \neq 0, \mathbf{u} \neq \mathbf{0}$$



□ Q：缩放是线性变换吗？

□ A：是的



缩放 – 矩阵表示

□假设我们想以倍数 a 缩放向量 $\mathbf{u} = (u_1, u_2, u_3)$, 应该如何用矩阵表示此操作?

□只需构建一个对角矩阵 D , 其对角线上的元素为 a

$$\underbrace{\begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix}}_D \underbrace{\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}}_{\mathbf{u}} = \underbrace{\begin{bmatrix} au_1 \\ au_2 \\ au_3 \end{bmatrix}}_{a\mathbf{u}}$$

Q: What happens if a is negative?

非均匀缩放 (轴对齐的)

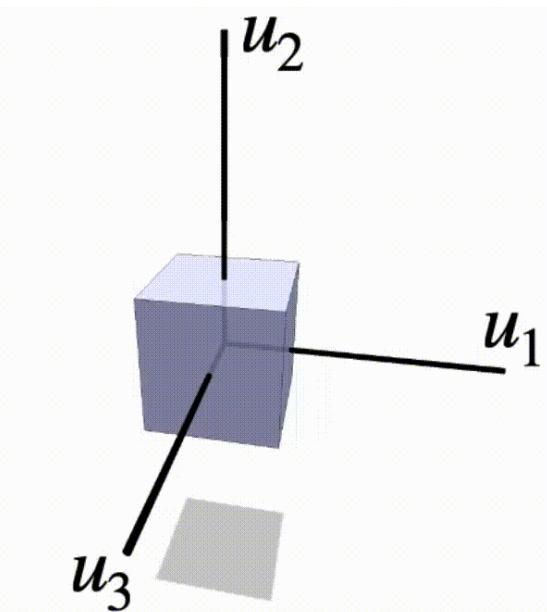
□ 我们还可以按不同的倍数缩放每个轴

$$f(u_1, u_2, u_3) = (au_1, bu_2, cu_3), \quad a, b, c \in \mathbb{R}$$

□ Q: 对应的矩阵表示是什么?

□ A: 只需把 a, b, c 放到矩阵的对角线上

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} au_1 \\ bu_2 \\ cu_3 \end{bmatrix}$$



如果我们想沿着其他非标准轴缩放呢?

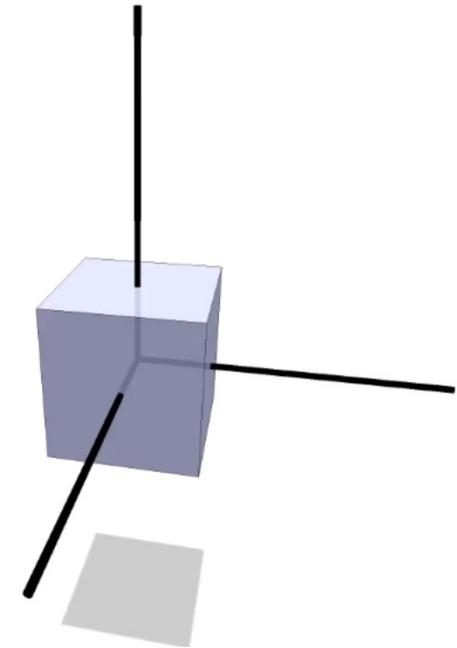
非均匀缩放

□ 我们可以

- 旋转到新的坐标轴 (R)
- 应用对角缩放 diagonal scaling (D)
- 旋转回到原来的坐标轴* (R^T)

□ 注意到整体的变换可以用一个对称矩阵 (symmetric matrix) 表示

$$A := R^T D R$$



$$f(\mathbf{x}) = R^T D R \mathbf{x}$$

Q: 是否所有对称矩阵都表示基于某些坐标轴的非均匀缩放?

*回忆一下对于旋转矩阵 R , 我们有 $R^{-1} = R^T$

切变 Shear

□ 切变将沿着方向 \mathbf{u} 移动 \mathbf{x} , 移动的大小由 \mathbf{x} 沿着固定向量 \mathbf{v} 的距离决定:

$$f_{\mathbf{u}, \mathbf{v}}(\mathbf{x}) = \mathbf{x} + \langle \mathbf{v}, \mathbf{x} \rangle \mathbf{u}$$

点积计算 \mathbf{x} 沿着 \mathbf{v} 的距离

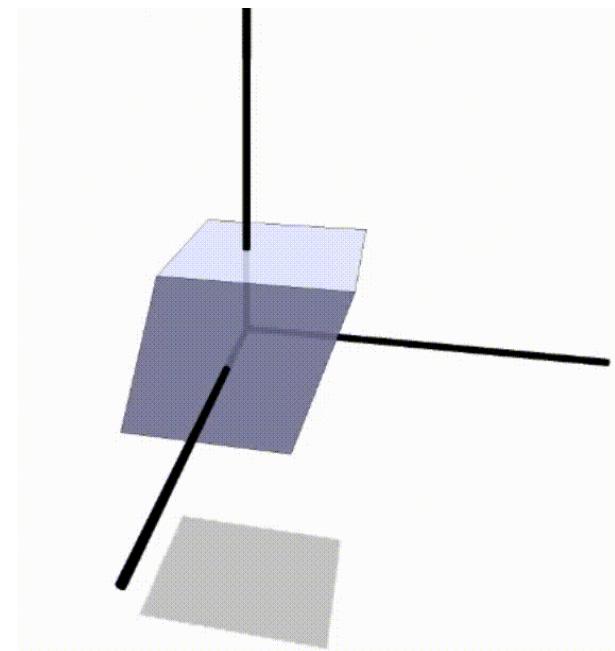
□ Q: 此变换是否线性?

□ A: Yes, 可用如下矩阵表示

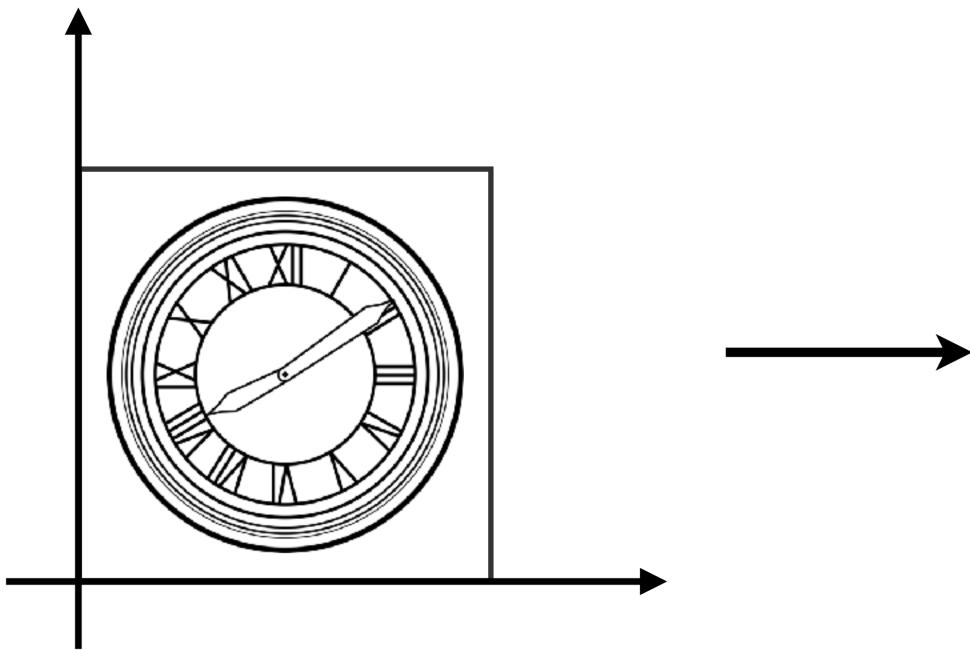
$$A_{\mathbf{u}, \mathbf{v}} = I + \mathbf{u}\mathbf{v}^T$$

□ 例子

$$\begin{aligned} \mathbf{u} &= (\cos(t), 0, 0) & A_{\mathbf{u}, \mathbf{v}} &= \begin{bmatrix} 1 & \cos(t) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ \mathbf{v} &= (0, 1, 0) \end{aligned}$$



切变 Shear – 2D



Hints:

Horizontal shift is 0 at $y=0$

Horizontal shift is a at $y=1$

Vertical shift is always 0

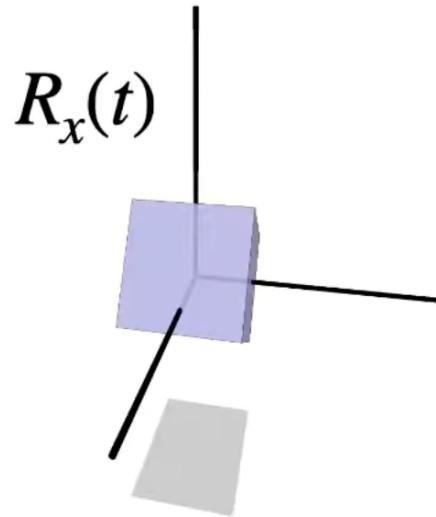
$$f_{\mathbf{u}, \mathbf{v}}(\mathbf{x}) = \mathbf{x} + \langle \mathbf{v}, \mathbf{x} \rangle \mathbf{u}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{cases} \mathbf{u} = [1, 0] \\ \mathbf{v} = [0, a] \end{cases}$$

复合变换 Composite Transformations

根据这些基本变换 (旋转、反射、缩放、切变...), 我们现在可以通过**矩阵乘法**进行复合变换



The last
matrix gets
applied first!

平移 Translations

□ 到目前为止，我们忽略了一个基本的变换 – 平移

□ 平移只是在给定点 x 上加一个偏移 u

$$f_u(x) = x + u$$

□ Q: 此变换是否线性?

□ 让我们检查其是否符合定义

additivity

$$f_u(x + y) = x + y + u$$

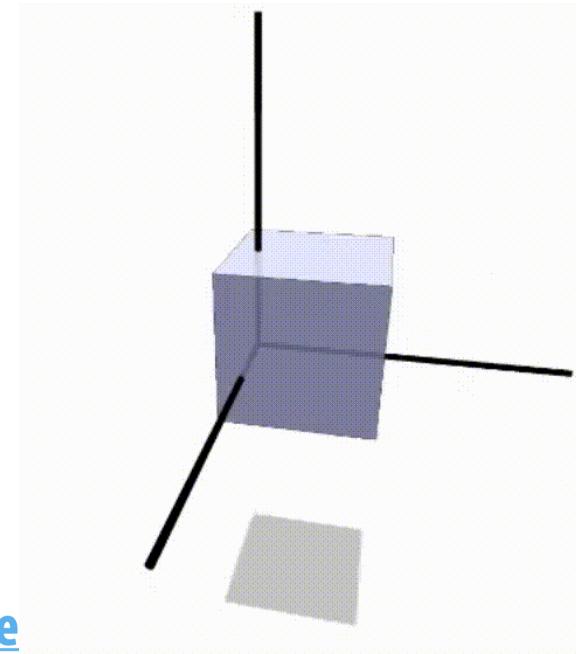
$$f_u(x) + f_u(y) = x + y + 2u$$

homogeneous

$$f_u(ax) = ax + u$$

$$af_u(x) = ax + au$$

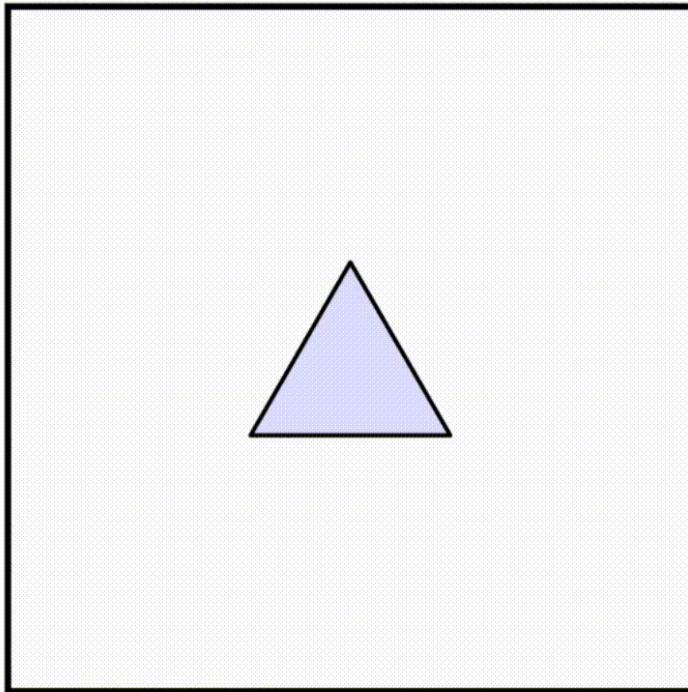
A: No, 平移是仿射, 而不是线性的



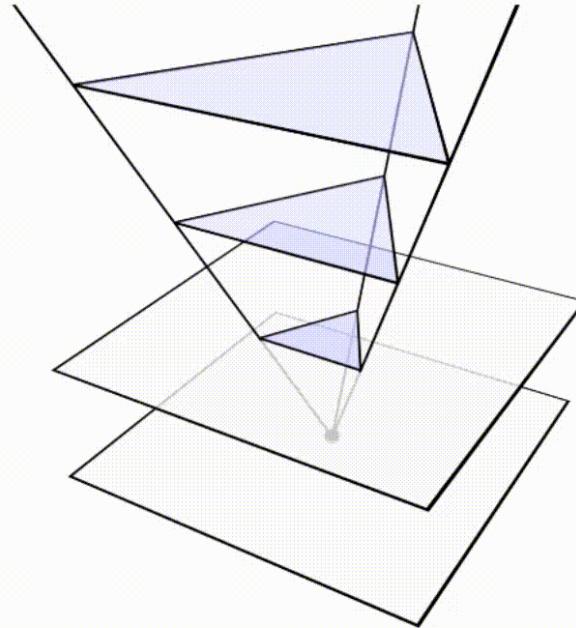
齐次坐标中的平移

口让我们思考一下，如果我们对二维坐标 p 进行平移，齐次坐标 \hat{p} 会发生什么

2D coordinates



homogeneous coordinates

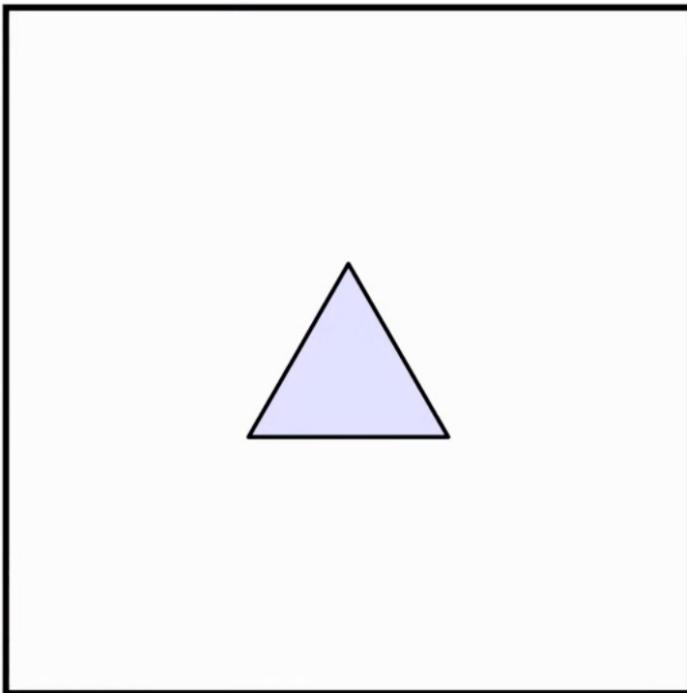


这个看起来像哪种变换？

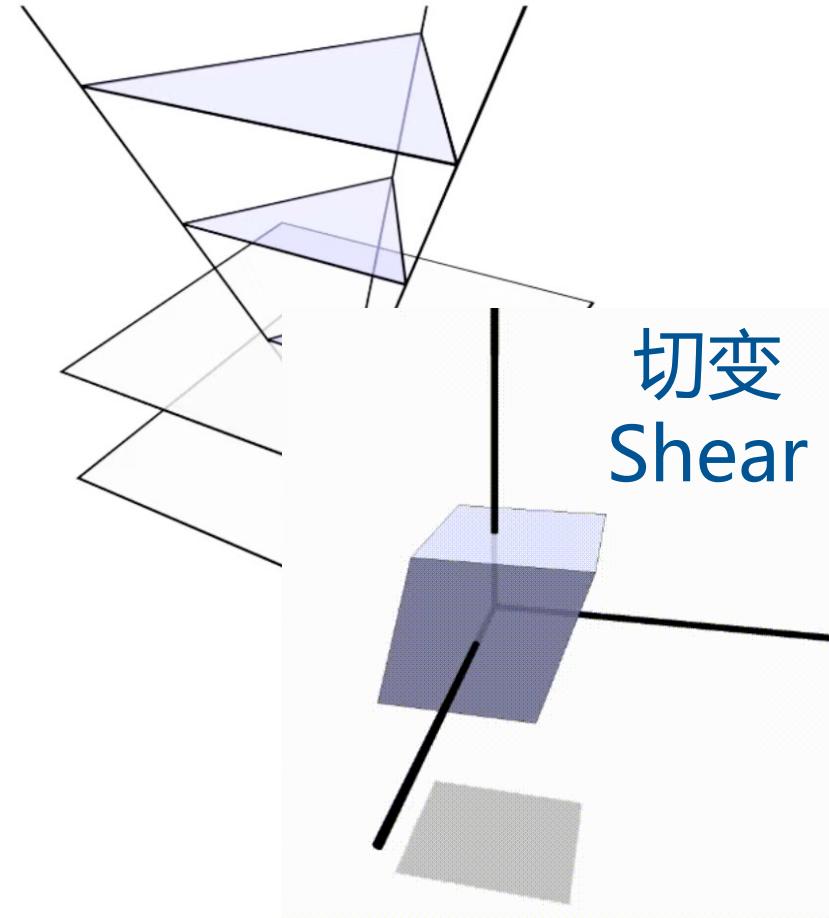
齐次坐标中的平移

口让我们思考一下，如果我们对二维坐标 p 进行平移，齐次坐标 \hat{p} 会发生什么

2D coordinates



homogeneous coordinates



这个看起来像哪种变换？

齐次坐标中的 3D 变换

- 3D (或更高维度) 的情况类似，只需附加一个齐次坐标轴
- 3D 线性变换的矩阵表示只增加一个额外的单位行/列 (identity row/column)；平移同样是切变

rotate (x, y, z) around y by θ

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

point in 3D

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

scale x, y, z
by a, b, c

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

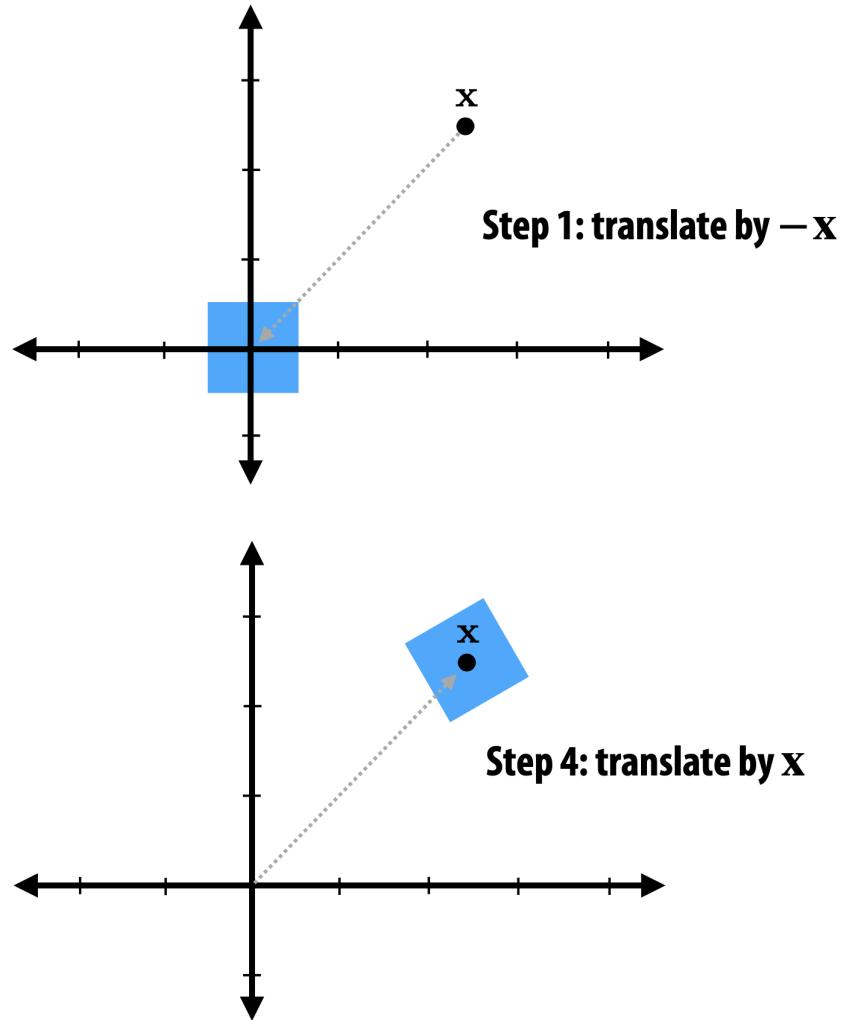
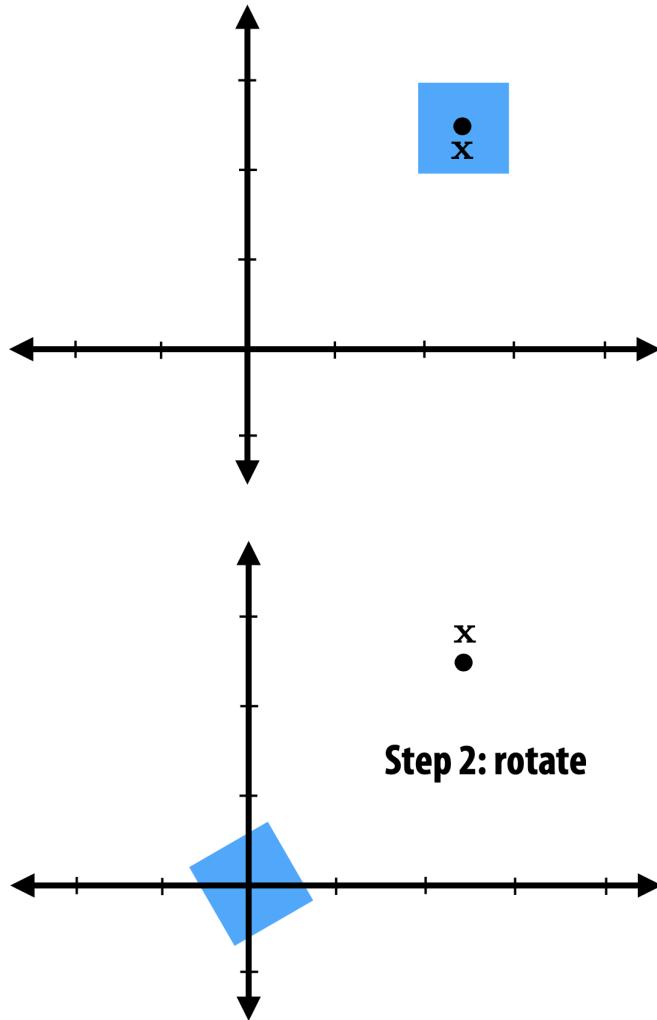
shear (x, y) by z
in (s, t) direction

$$\begin{bmatrix} 1 & 0 & s & 0 \\ 0 & 1 & t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

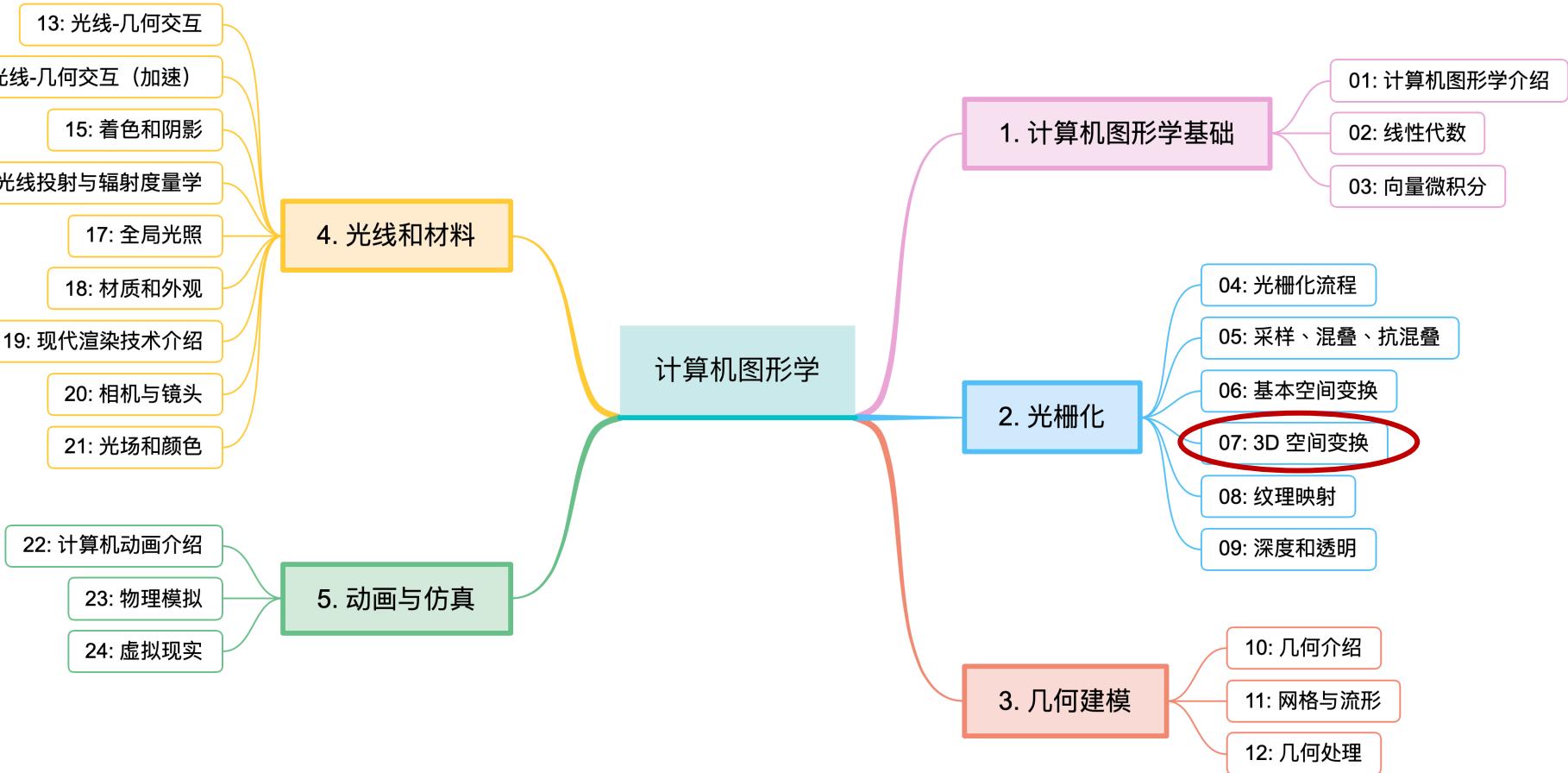
translate (x, y, z)
by (u, v, w)

$$\begin{bmatrix} 1 & 0 & 0 & u \\ 0 & 1 & 0 & v \\ 0 & 0 & 1 & w \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

基于某一点做旋转



Q: What happens if we just rotate without translating first?



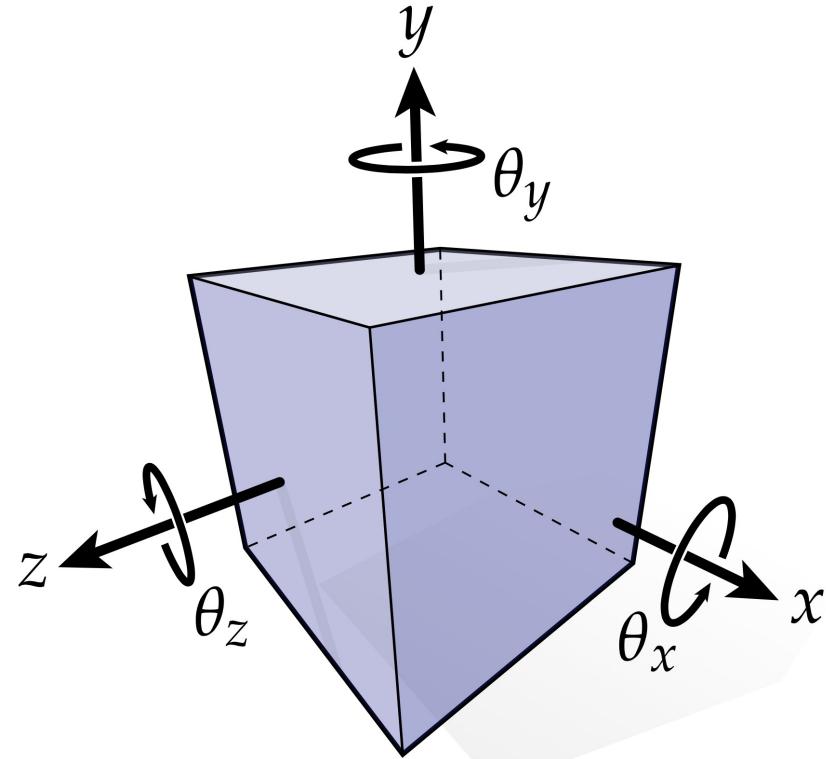
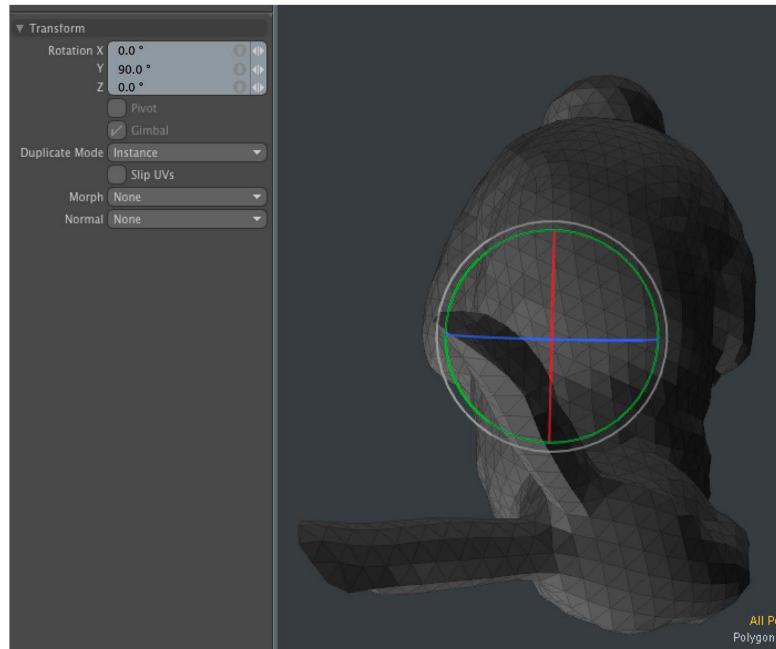
3D 中的旋转表示 – 欧拉角 Euler Angles

□ 如何在 3D 中表示旋转？

□ 一个简单的想法，能否直接将 2D 中的方法用到 3D 中的 x, y, z 轴？

□ 我们称这种方案 (scheme) 为欧拉角，有什么优点？

□ “Gimbal lock” 万向节锁



Gimbal Lock

□ 当使用欧拉角 $\theta_x, \theta_y, \theta_z$ 时，可能会出现无法绕其中一个轴旋转的情况

□ 回顾一下绕三个轴的旋转矩阵

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x \\ 0 & \sin \theta_x & \cos \theta_x \end{bmatrix} \quad R_y = \begin{bmatrix} \cos \theta_y & 0 & \sin \theta_y \\ 0 & 1 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y \end{bmatrix} \quad R_z = \begin{bmatrix} \cos \theta_z & -\sin \theta_z & 0 \\ \sin \theta_z & \cos \theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

□ 这些矩阵的乘积表示基于欧拉角的 3D 旋转

$$R_x R_y R_z = \begin{bmatrix} \cos \theta_y \cos \theta_z & -\cos \theta_y \sin \theta_z & \sin \theta_y \\ \cos \theta_z \sin \theta_x \sin \theta_y + \cos \theta_x \sin \theta_z & \cos \theta_x \cos \theta_z - \sin \theta_x \sin \theta_y \sin \theta_z & -\cos \theta_y \sin \theta_x \\ -\cos \theta_x \cos \theta_z \sin \theta_y + \sin \theta_x \sin \theta_z & \cos \theta_z \sin \theta_x + \cos \theta_x \sin \theta_y \sin \theta_z & \cos \theta_x \cos \theta_y \end{bmatrix}$$

□ 考虑一个特殊例子， $\theta_y = \pi/2$ (so $\cos \theta_y = 0, \sin \theta_y = 1$)

$$\Rightarrow \begin{bmatrix} 0 & 0 & 1 \\ \cos \theta_z \sin \theta_x + \cos \theta_x \sin \theta_z & \cos \theta_x \cos \theta_z - \sin \theta_x \sin \theta_z & 0 \\ -\cos \theta_x \cos \theta_z + \sin \theta_x \sin \theta_z & \cos \theta_z \sin \theta_x + \cos \theta_x \sin \theta_z & 0 \end{bmatrix}$$

Gimbal Lock 万向节锁

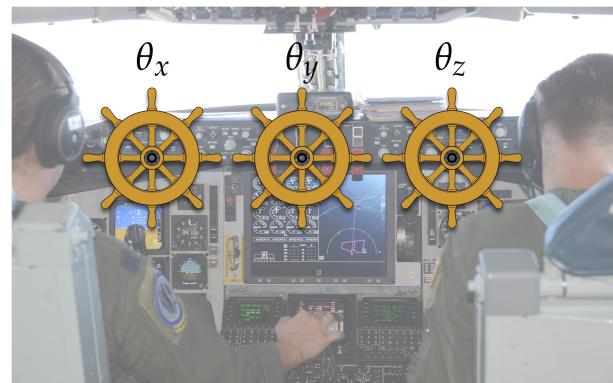
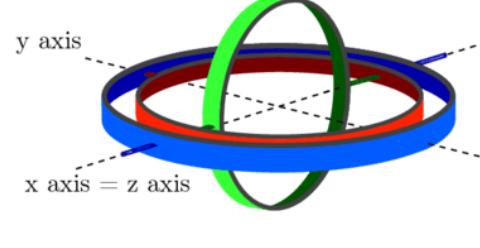
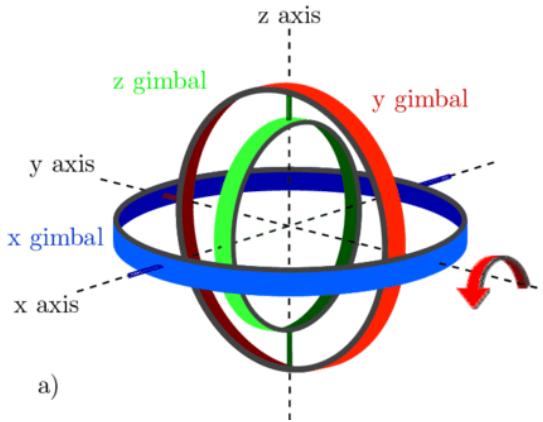
口简化上一张幻灯片中的矩阵，我们得到

$$\begin{bmatrix} 0 & 0 & 1 \\ \sin(\theta_x + \theta_z) & \cos(\theta_x + \theta_z) & 0 \\ -\cos(\theta_x + \theta_z) & \sin(\theta_x + \theta_z) & 0 \end{bmatrix}$$

**no matter how we adjust θ_x , θ_z ,
can only rotate in one plane!**

Q: What does this matrix do?

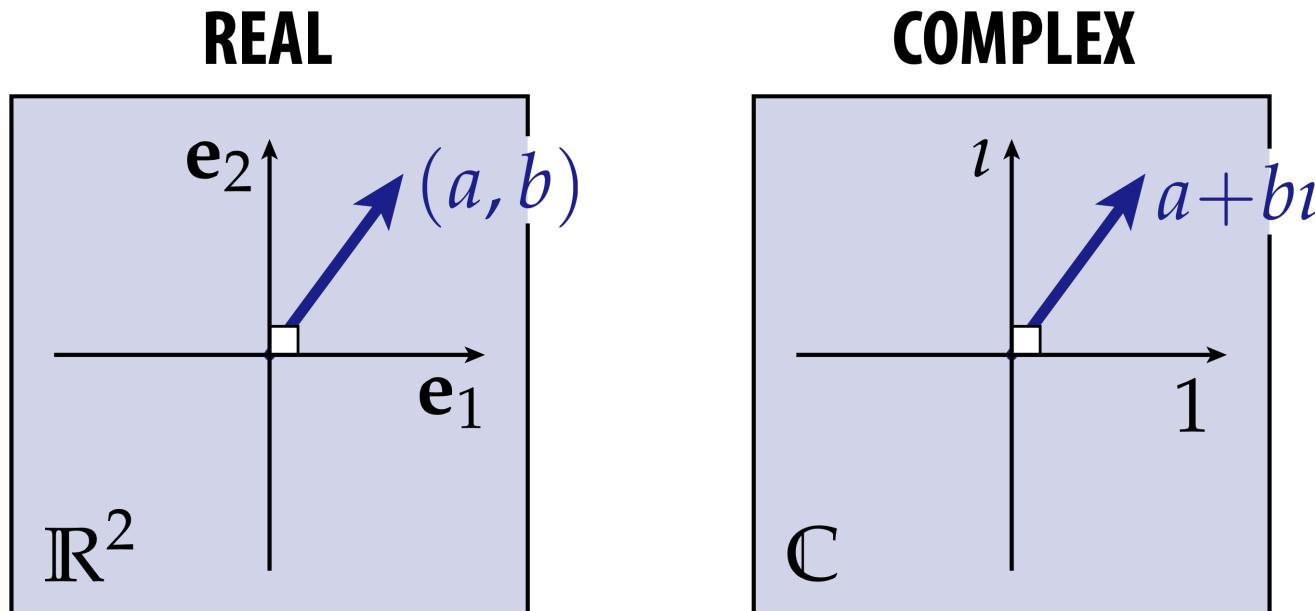
口我们被“锁定”在一个单一的旋转轴上



口对飞机控制是个很糟糕的设计！

复数 Complex numbers

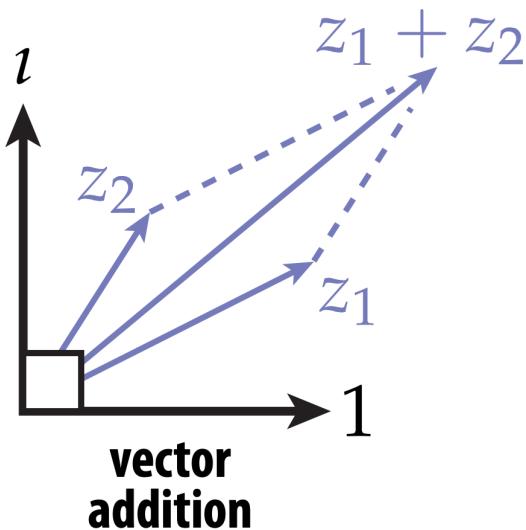
- 复数可以理解成一个 2 维向量
- 使用 “1” 和 “ i ” 而不是 e_1, e_2 来表示两个坐标基
- 除此之外，其他表现与 2 维空间无异



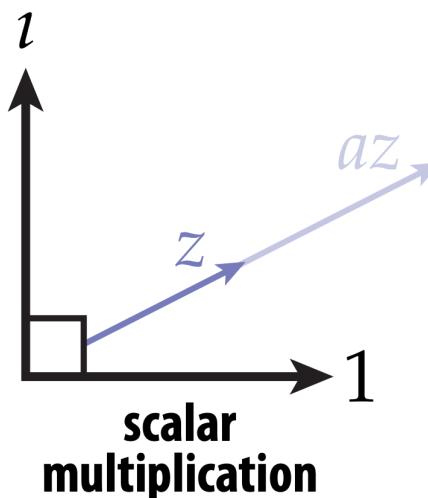
- 除了一个非常有用的关注两个向量乘积的新操作

复数算术 Complex arithmetic

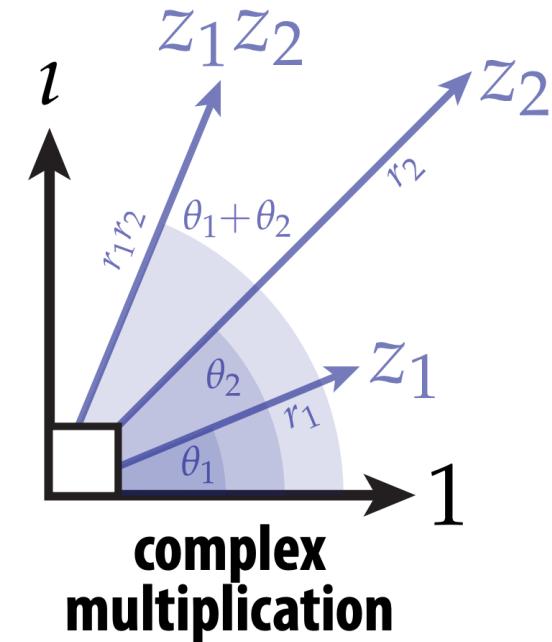
□ 与以前相同的操作，再加上一个



vector
addition



scalar
multiplication



complex
multiplication

□ 复数乘法

- 角度 (angles) 相加
- 长度 (magnitudes) 相乘

“POLAR FORM”*:

$$z_1 := (r_1, \theta_1)$$

$$z_2 := (r_2, \theta_2)$$

$$z_1 z_2 = (r_1 r_2, \theta_1 + \theta_2)$$

have to be more
careful here!



*Not quite how it really works, but basic idea is right.

2D 旋转: Matrices vs. Complex

□ 假设我们想将向量 \mathbf{u} 旋转角度 θ , 然后再旋转角度 φ

REAL / RECTANGULAR	COMPLEX / POLAR
$\mathbf{u} = (x, y)$	$u = re^{i\alpha}$
$\mathbf{A} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$	$a = e^{i\theta}$
$\mathbf{B} = \begin{bmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{bmatrix}$	$b = e^{i\phi}$
$\mathbf{A}\mathbf{u} = \begin{bmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \end{bmatrix}$	$abu = re^{i(\alpha+\theta+\phi)}$
$\mathbf{B}\mathbf{A}\mathbf{u} = \begin{bmatrix} (x \cos \theta - y \sin \theta) \cos \phi - (x \sin \theta + y \cos \theta) \sin \phi \\ (x \cos \theta - y \sin \theta) \sin \phi + (x \sin \theta + y \cos \theta) \cos \phi \end{bmatrix}$	
$= \cdots \text{some trigonometry} \cdots =$	
$\mathbf{B}\mathbf{A}\mathbf{u} = \begin{bmatrix} x \cos(\theta + \phi) - y \sin(\theta + \phi) \\ x \sin(\theta + \phi) + y \cos(\theta + \phi) \end{bmatrix}$	
	求导操作也很容易

图形中的普遍主题：

某个操作通常有许多“等价”
表示 representations

要选择让问题更简单（或更
高效、准确）的那一个

四元数 Quaternions

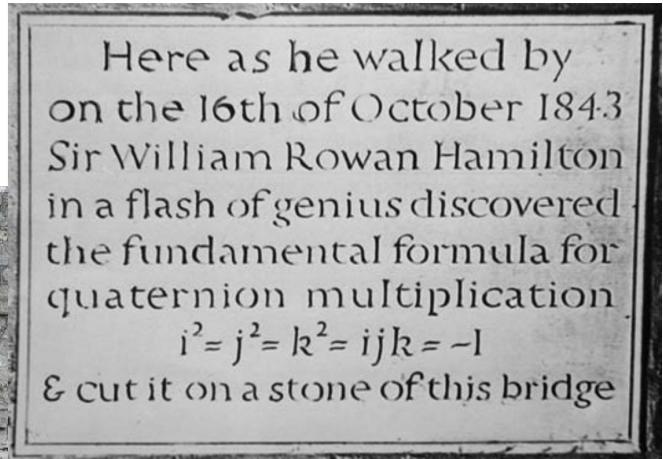
- 有点类似复数，不过是针对 3D 旋转的
- 不同于 2D 旋转仅需 2 个部分 (长度+角度)，只有 3 个部分无法进行 3D 旋转



William Rowan Hamilton
(1805-1865)



(Not Hamilton)



坐标系中的四元数

□ 汉密尔顿：为了模仿 2D 复数的方式进行 3D 旋转，实际上需要 **4 个坐标**

□ 一个实数，三个虚数

"H" is for Hamilton!

$$\mathbb{H} := \text{span}(\{1, i, j, k\})$$

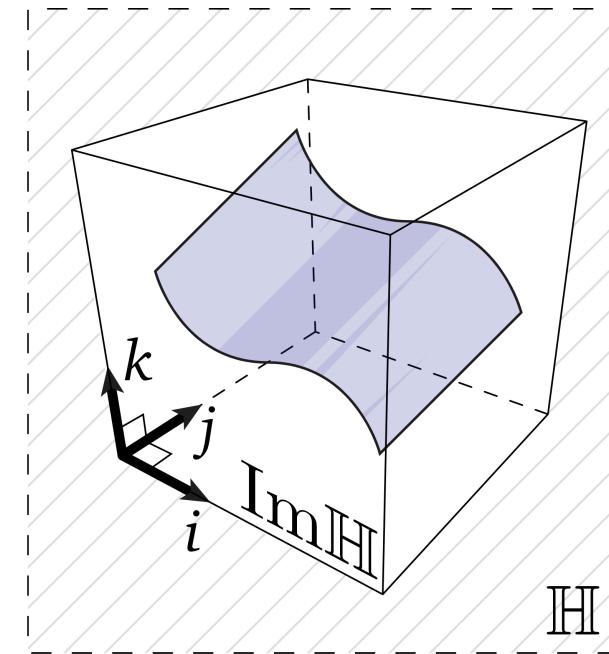
□ 四元数乘积由以下计算决定

$$\left\{ \begin{array}{l} i^2 = j^2 = k^2 = ijk = -1 \\ ij = k, ji = -k \\ jk = i, kj = -i \\ ki = j, ik = -j \end{array} \right.$$

以及一些规则 (分配律 *distributivity*、结合律 *associativity* 等)

□ Warning：乘积不满足交换律 (为什么？)

$$\text{For } q, p \in \mathbb{H}, \quad qp \neq pq$$



基于四元数的 3D 变换

口四元数在图形学中的主要应用：3D 旋转 Rotations

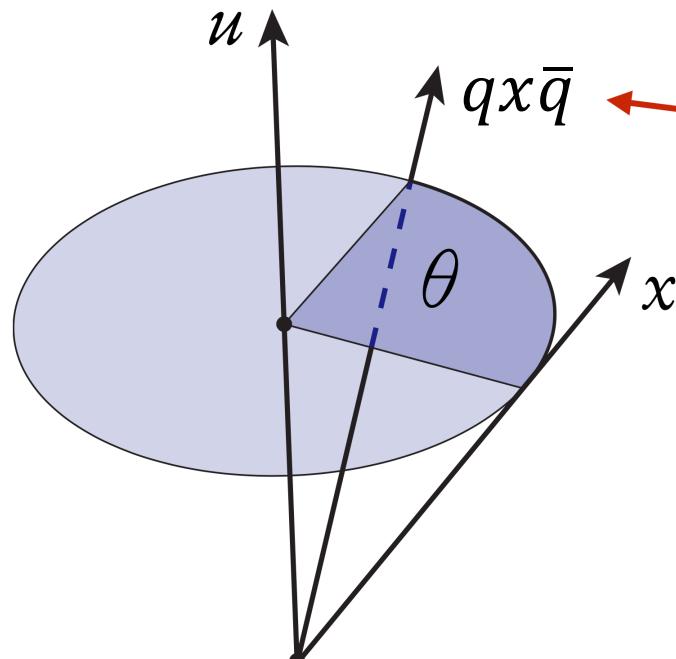
口考虑向量 x (pure imaginary) 和单位四元数 q

$$x \in \text{Im}(\mathbb{H})$$

$$q = w + xi + yj + zk$$

$$q \in \mathbb{H}, \quad |q|^2 = 1$$

$$w^2 + x^2 + y^2 + z^2 = 1$$



always expresses
some rotation

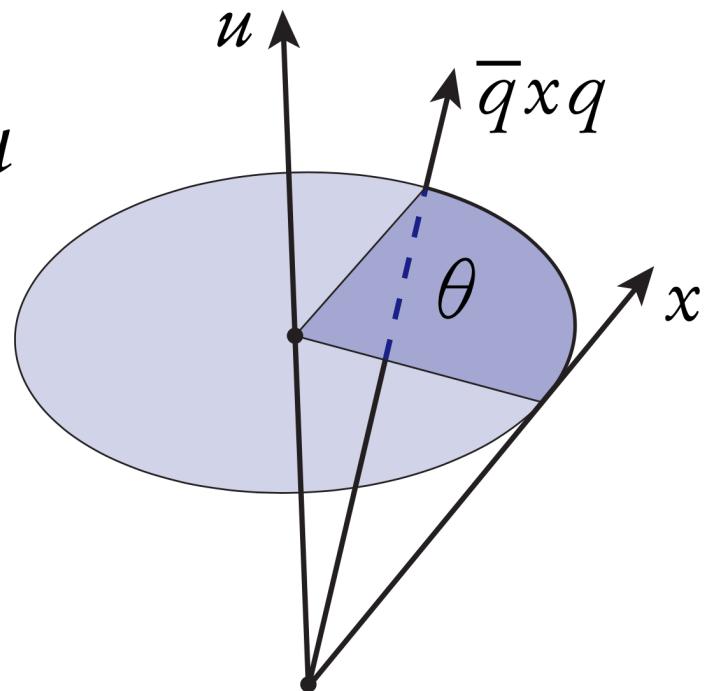
\bar{q} 是 q 的共轭复数 (conjugate),
通过改变虚部的符号得到, i.e.,
 $\bar{q} = w - xi - yj - zk$

回顾从轴/角度旋转

口给定轴 \mathbf{u} , 角度 θ , 表示旋转的四元数 q 为

$$q = \cos(\theta/2) + \sin(\theta/2)\mathbf{u}$$

Angle **Unit vector**



基于四元数的 3D 变换

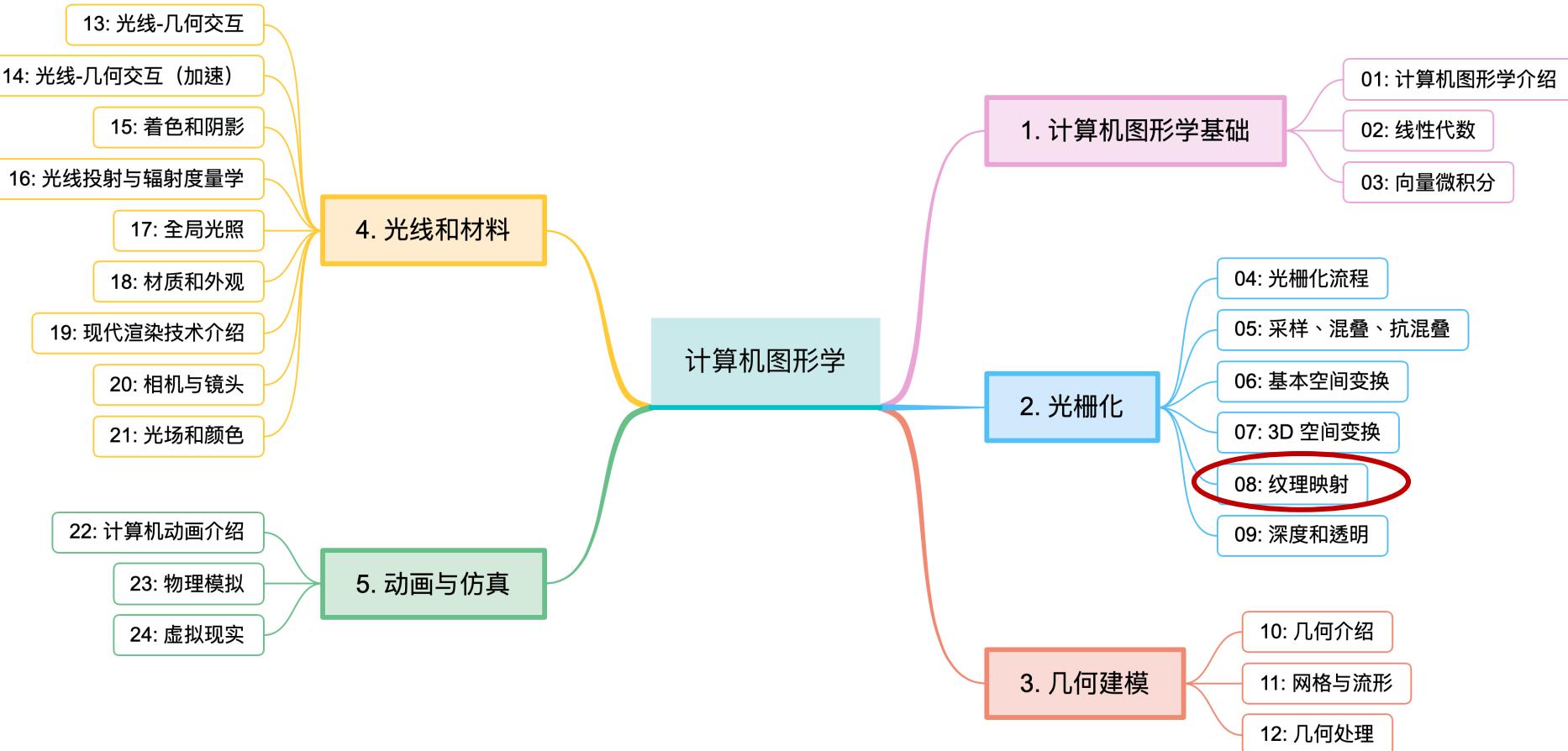
□ 给定 3D 向量 $x = (0, \mathbf{v})$ 和单位四元数 $q = (a, \mathbf{u})$

□ 旋转结果 $qx\bar{q}$

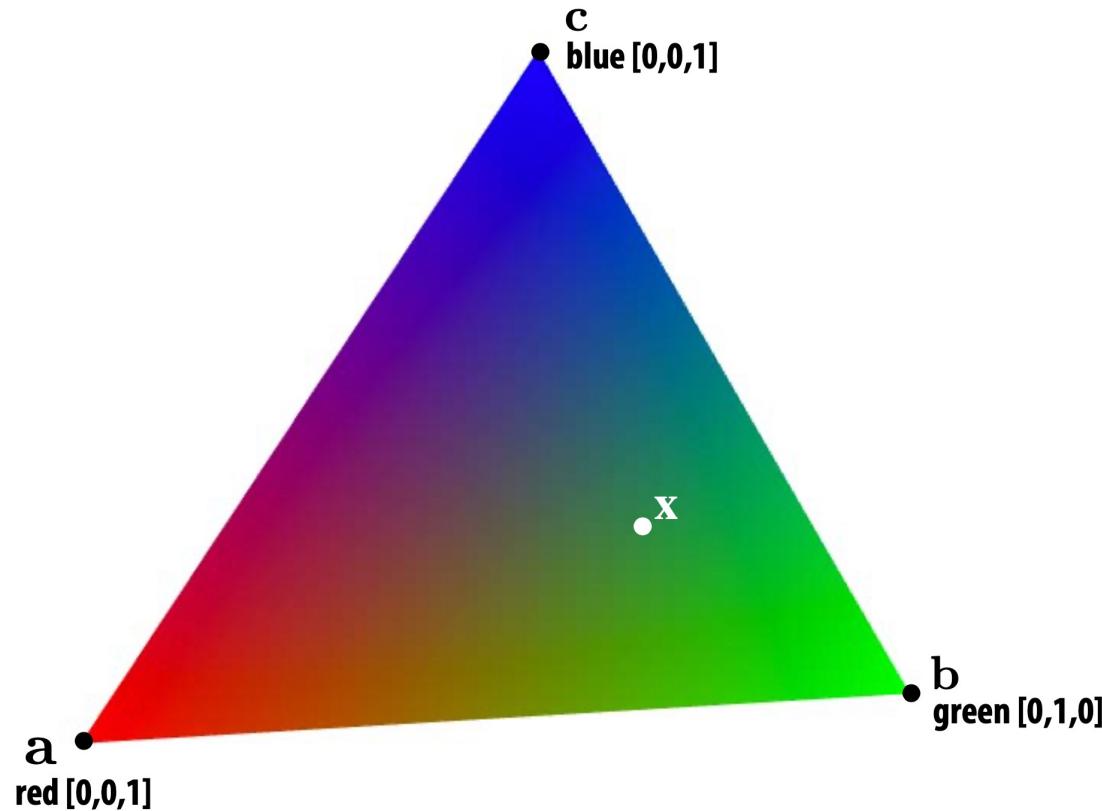
$$\begin{aligned} qx\bar{q} &= (-\mathbf{u} \cdot \mathbf{v}, a\mathbf{v} + \mathbf{u} \times \mathbf{v})(a, -\mathbf{u}) \\ &= (0, (a^2 - \mathbf{u} \cdot \mathbf{u})\mathbf{v} + 2(\mathbf{u} \cdot \mathbf{v})\mathbf{u} + 2a(\mathbf{u} \times \mathbf{v})) \end{aligned}$$

□ 比矩阵更容易记忆 (和操作)

$$\begin{bmatrix} \cos \theta + u_x^2 (1 - \cos \theta) & u_x u_y (1 - \cos \theta) - u_z \sin \theta & u_x u_z (1 - \cos \theta) + u_y \sin \theta \\ u_y u_x (1 - \cos \theta) + u_z \sin \theta & \cos \theta + u_y^2 (1 - \cos \theta) & u_y u_z (1 - \cos \theta) - u_x \sin \theta \\ u_z u_x (1 - \cos \theta) - u_y \sin \theta & u_z u_y (1 - \cos \theta) + u_x \sin \theta & \cos \theta + u_z^2 (1 - \cos \theta) \end{bmatrix}$$



考虑采样 $\text{color}(x, y)$

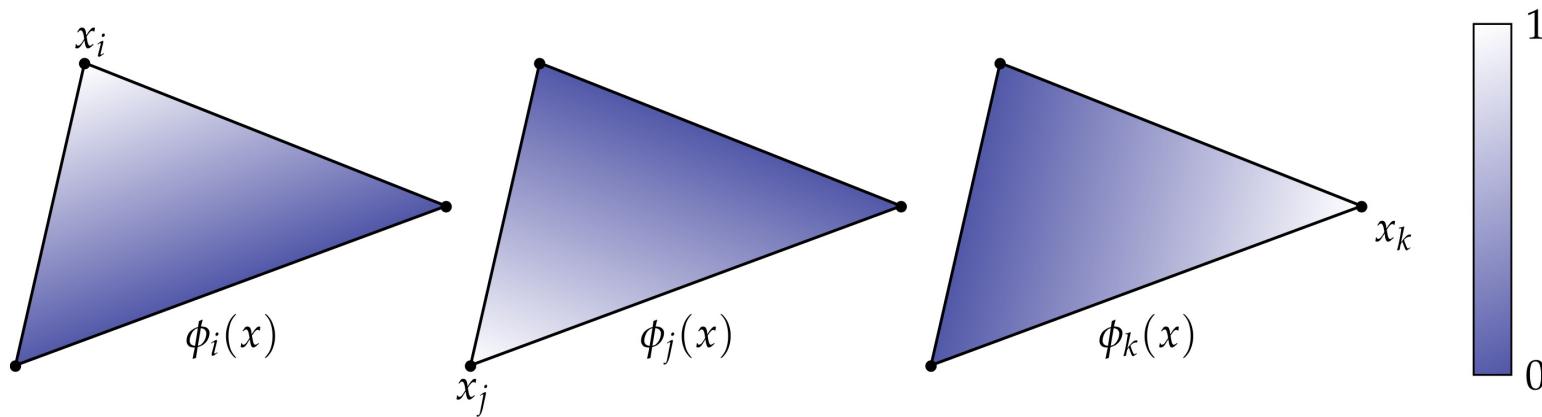
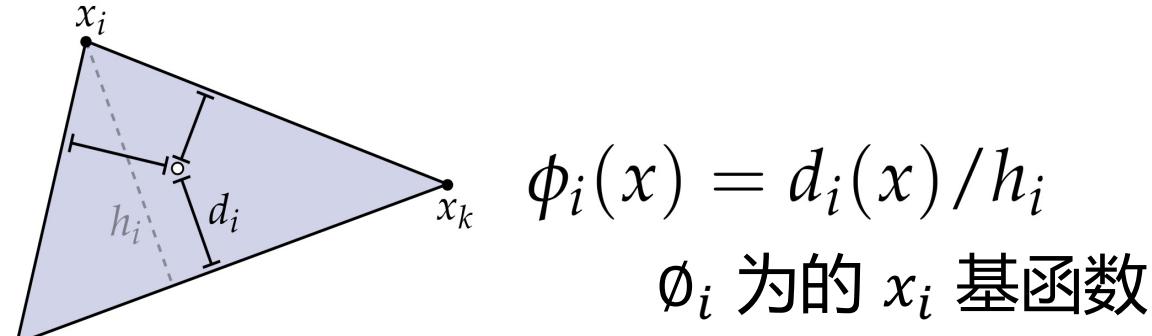


- 三角形在点 x 是什么颜色?
- 一个标准策略：基于顶点做颜色插值

回顾 2D 中的线性插值

□ 我们可以为三角形构造类似的函数

□ 对于给定的点 x , 测量其到每条边的距离; 然后除以三角形的高度:

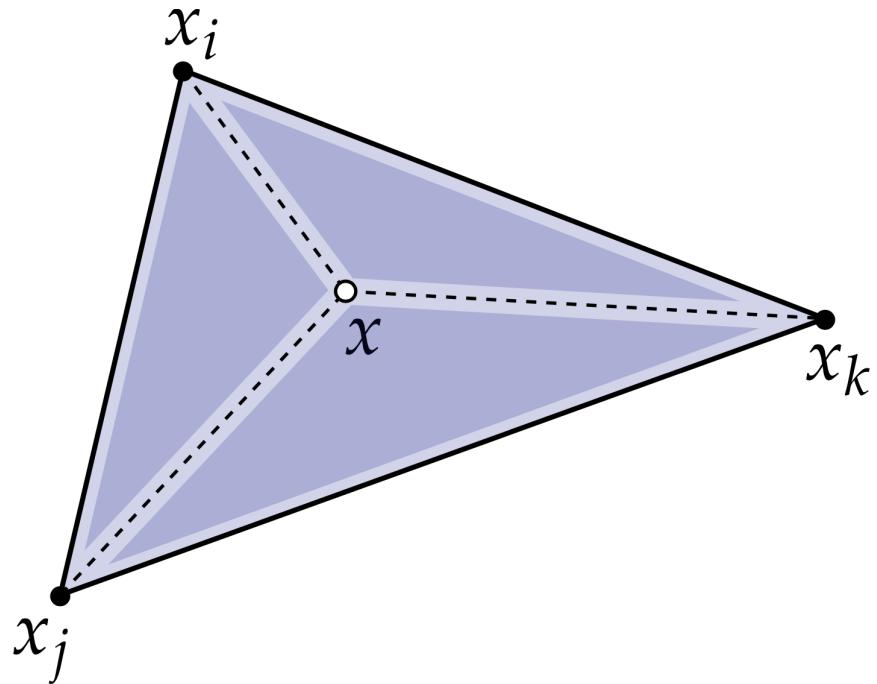


□ 通过线性组合得到插值函数: $\hat{f}(x) = f_i\phi_i + f_j\phi_j + f_k\phi_k$

Q: Is this the same as the (ugly) function we found before?

2D 插值的另一种方式

口还可以利用三角形之间的面积比值找到同样的基函数



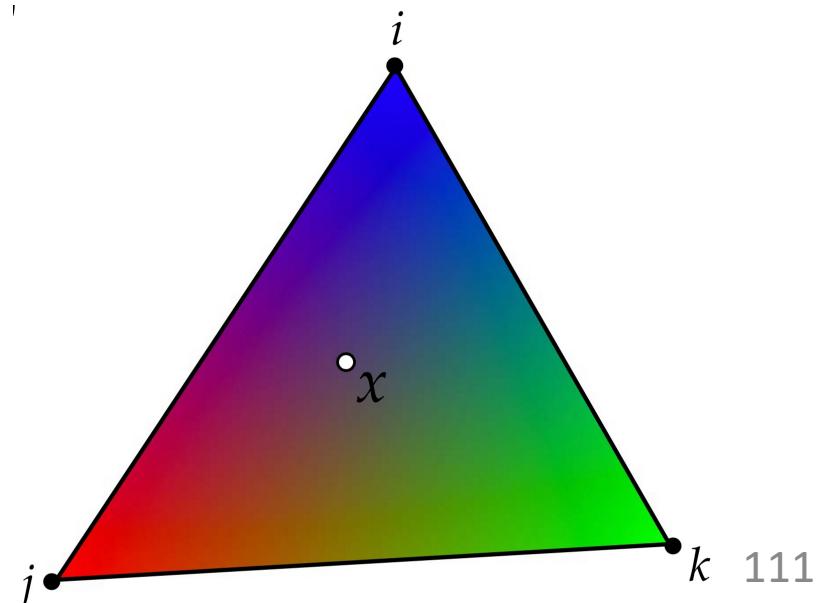
$$\phi_i(x) = \frac{\text{area}(x, x_j, x_k)}{\text{area}(x_i, x_j, x_k)}$$

Q: Do you buy it? (Why or why not?)

重心坐标 Barycentric Coordinates

- 无论以何种方式计算，对于一个给的定点，其三个函数 $\phi_i(x), \phi_j(x), \phi_k(x)$ 的值被称为重心坐标
- 可用于对与顶点关联的任何属性进行插值 (颜色 color*、纹理坐标 texture coordinates 等)
- 重心坐标的**符号**直接告诉我们该点是否通过了三角形光栅化的半平面测试 (half-plane tests)

$$\begin{aligned}color(x) &= color(x_i)\phi_i + \\& color(x_j)\phi_j + color(x_k)\phi_k\end{aligned}$$



*Note: we haven't explained yet how to encode colors as numbers! We'll talk about that in a later lecture...

纹理映射 Texture Mapping

把一张 2D 的图片
包裹 3D 物体



纹理映射的很多使用场景

口定义表面反射率的变化 (variation in surface reflectance)



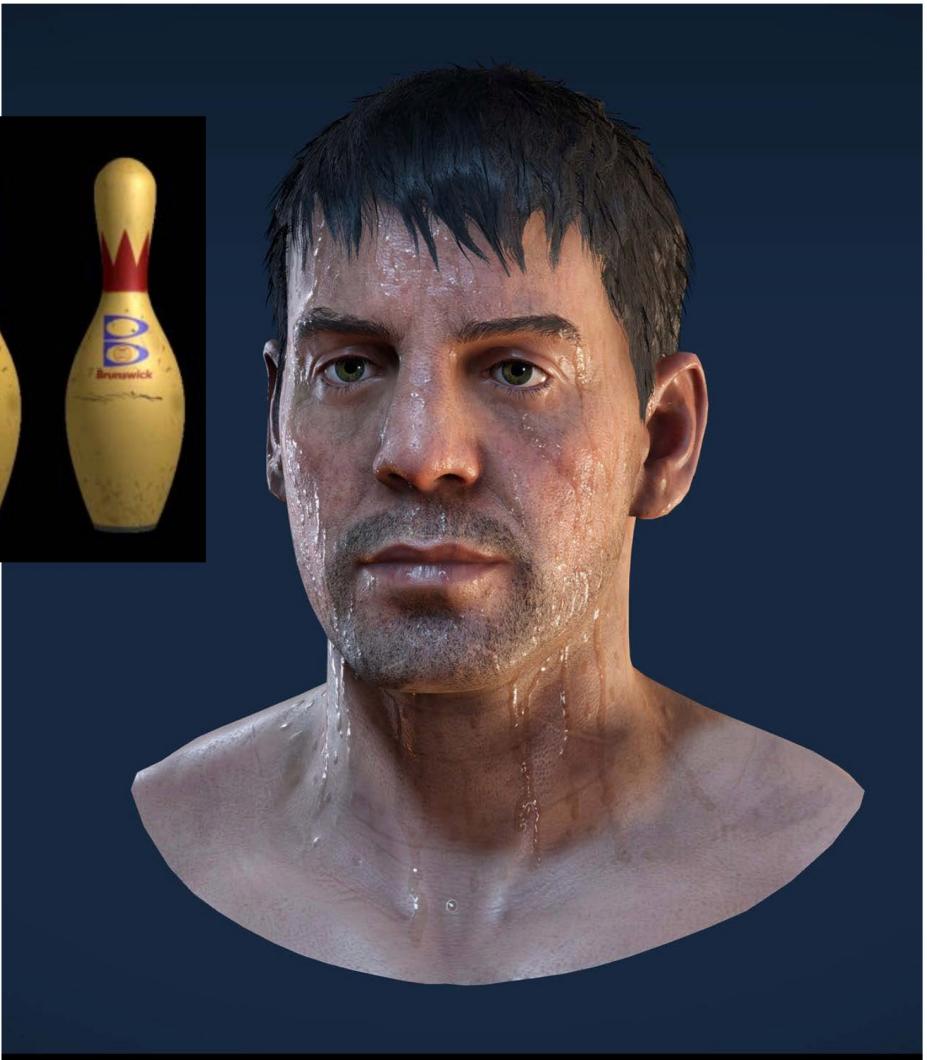
Pattern on ball

Wood grain on floor

描述表面材料特性



Multiple layers of texture maps for color, logos, scratches, etc.



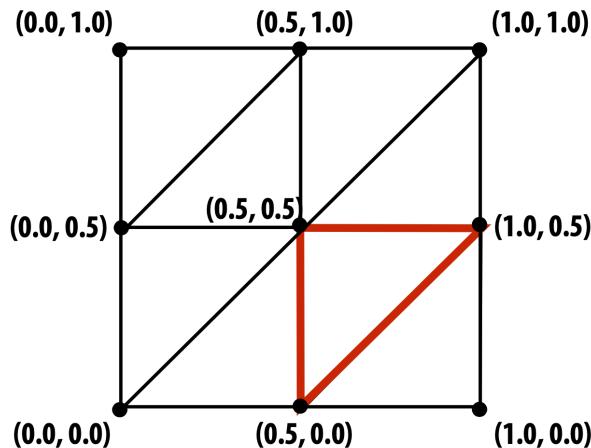
(C)2013 CRYTEK GMBH. ALL RIGHTS RESERVED. RYSE IS A REGISTERED TRADEMARK OF CRYTEK GMBH

RYSE
SON OF ROME

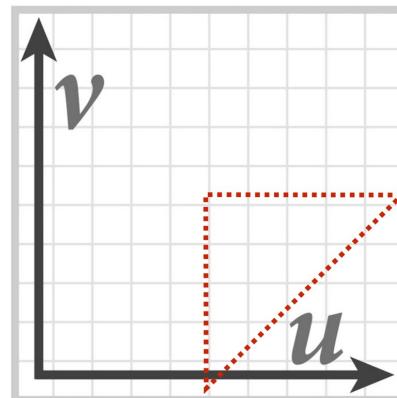
纹理坐标

- “纹理坐标” 定义从表面坐标 (surface coordinates) 到纹理域 (texture domain) 中的点的映射
- 通常通过在三角形顶点线性插值纹理坐标来定义

假设每个立方体面被分割成八个三角形，每个顶点的纹理坐标为 (u, v)

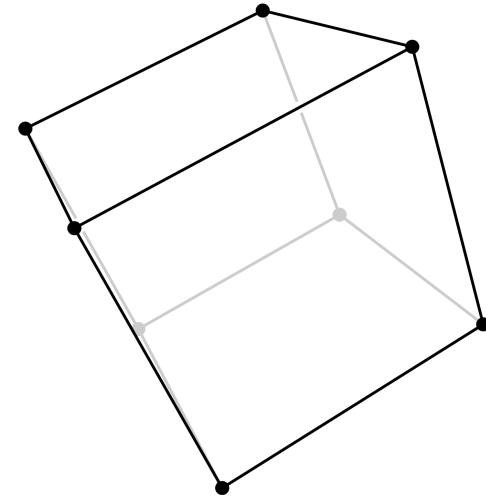


在域 $[0,1]^2$ 中的纹理可以被一个 2048×2048 的图像指定

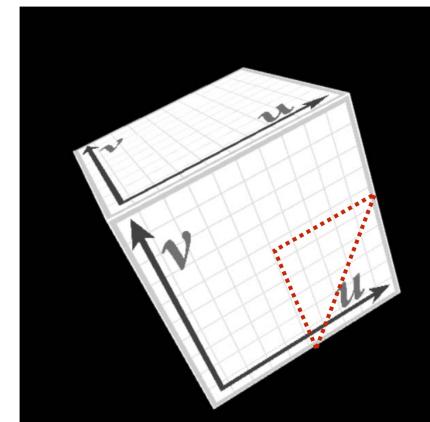


红色表示高亮三角形的位置

example: texture this cube

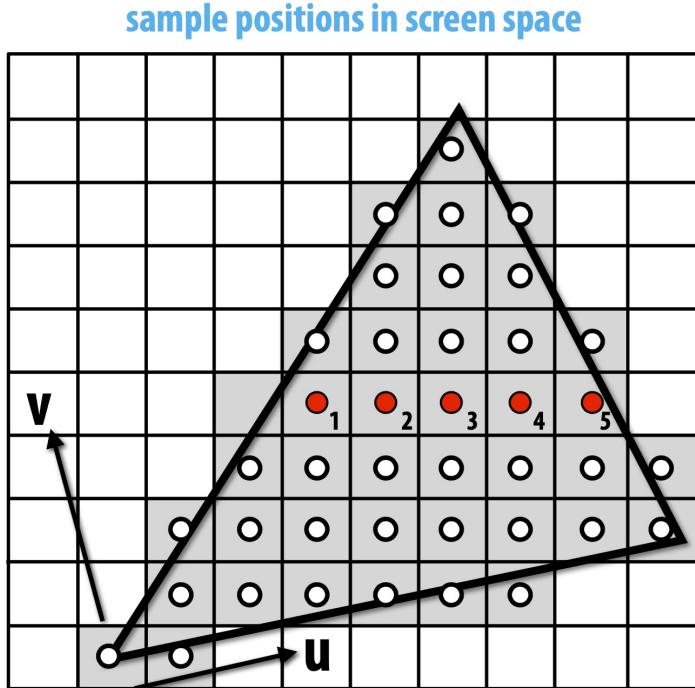


线性插值纹理坐标 & 在纹理中“查找”颜色可得到此图像

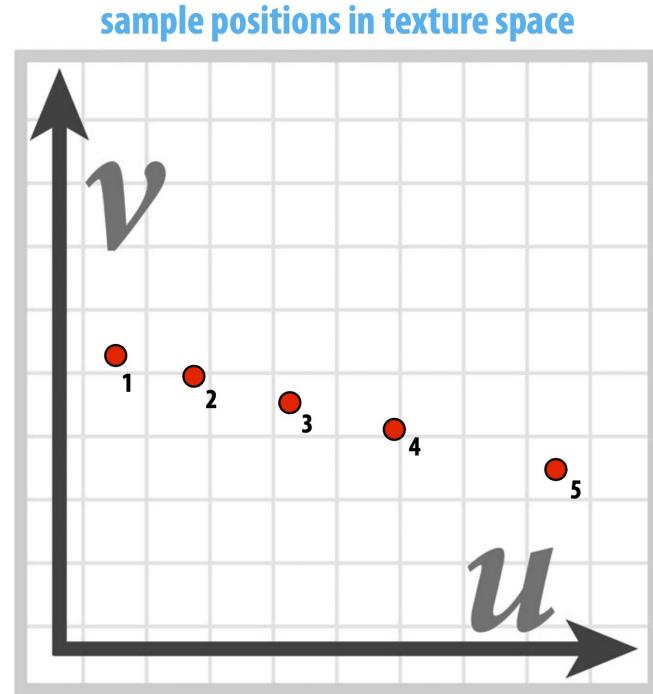


可视化纹理样本

由于三角形是从 3D 投影到 2D 的，屏幕空间中的像素将对应于纹理中不同大小和位置的区域



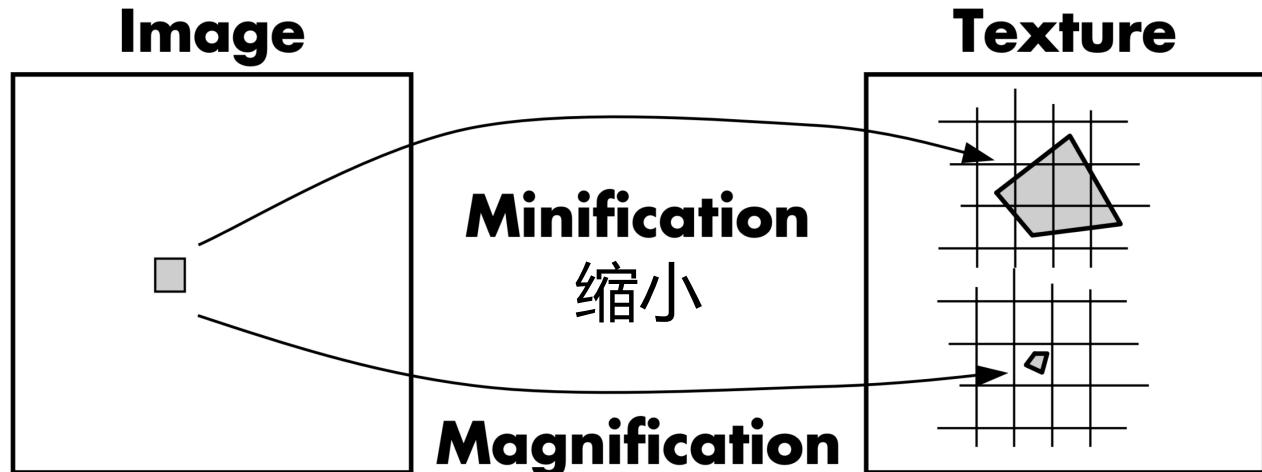
样本位置在屏幕空间中均匀分布
(光栅化器 rasterizer 在这些位置对三角形的外观进行采样)



纹理空间中的采样位置非均匀
(纹理函数在这些位置采样)

不规则的采样模式很难避免走样 (aliasing) 问题

放大与缩小



□ 放大 (更容易)

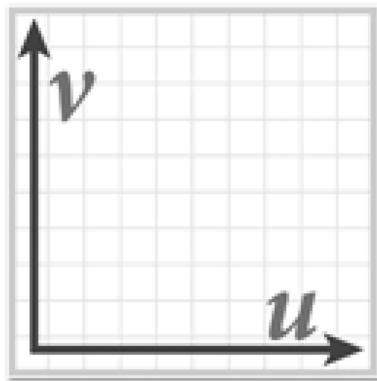
- 例子：摄影机离场景对象很近
- 单屏幕像素映射到纹理的微小区域
- 可以在屏幕像素中心插入值

放大

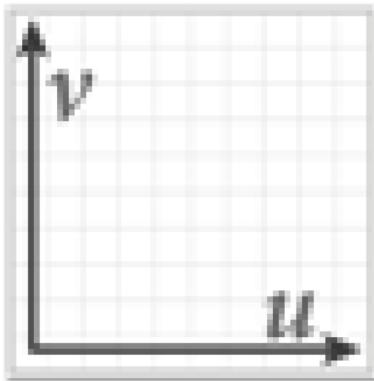
□ 缩小 (更难)

- 示例：摄影机离场景对象很远
- 单屏幕像素映射到纹理的大区域
- 需要计算像素上的平均纹理值以避免走样现象

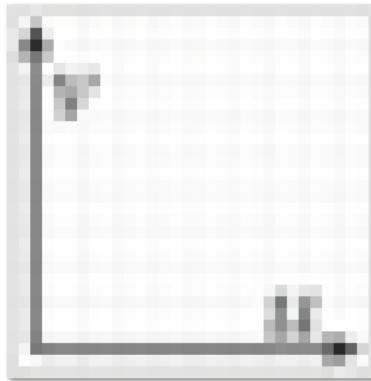
MIP map (L. Williams 83)



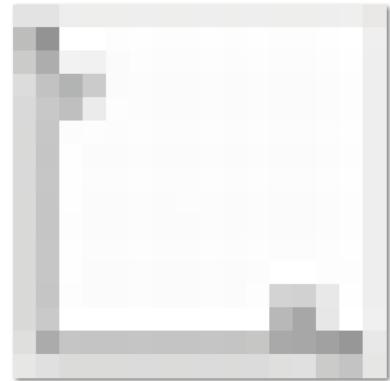
Level 0 = 128x128



Level 1 = 64x64



Level 2 = 32x32



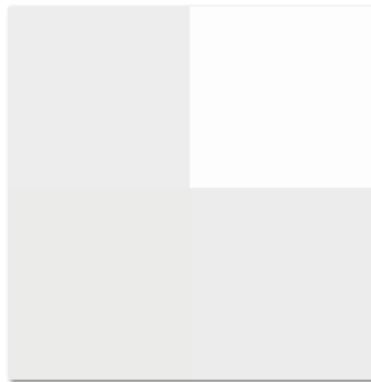
Level 3 = 16x16



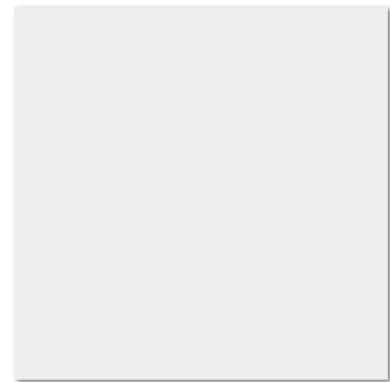
Level 4 = 8x8



Level 5 = 4x4



Level 6 = 2x2



Level 7 = 1x1

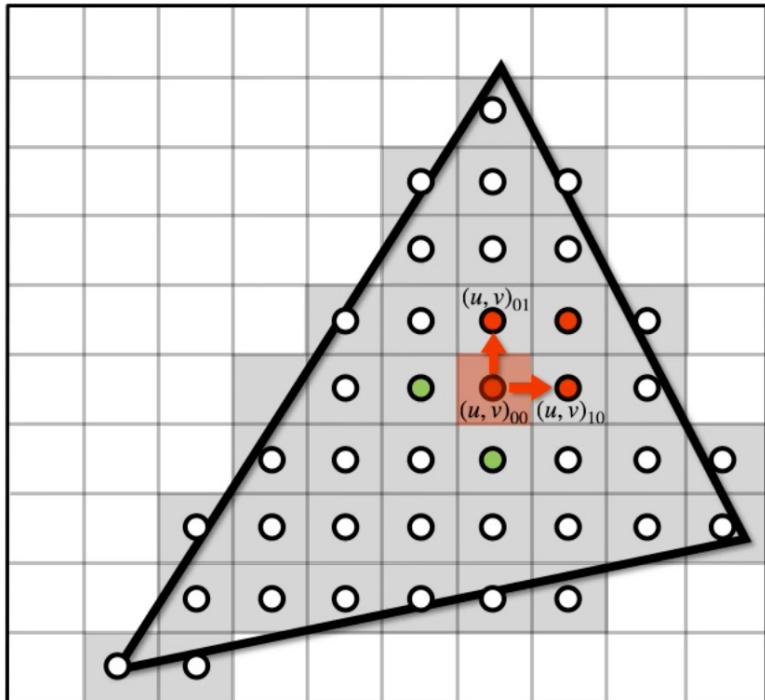
□ 粗略想法：以“各种可能的大小”存储预过滤的图像

□ 较高级别存储纹理空间中较大区域的纹理平均值（下采样）

□ 从适当大小的 MIP 图中查找单个像素

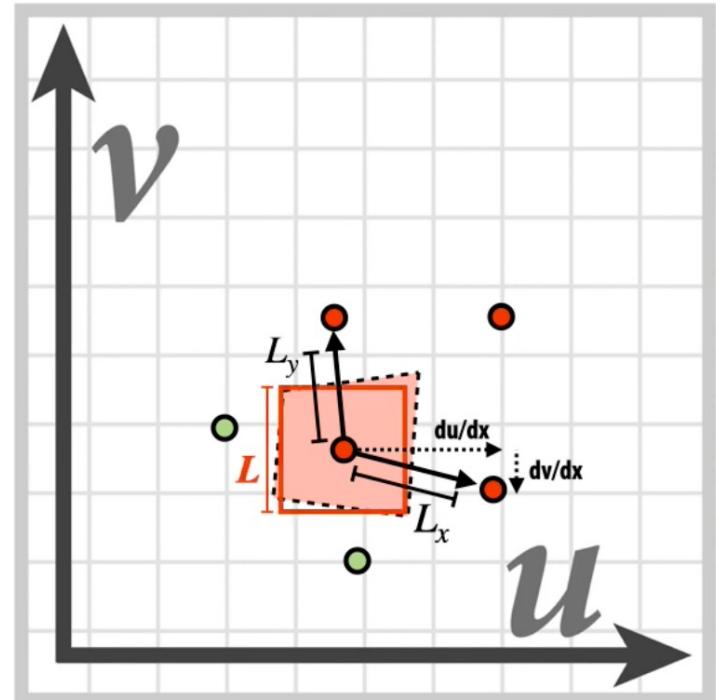
计算 MIP Map 级别

口计算相邻采样的纹理坐标值之间的差异



$$\frac{du}{dx} = u_{10} - u_{00} \quad \frac{dv}{dx} = v_{10} - v_{00}$$

$$\frac{du}{dy} = u_{01} - u_{00} \quad \frac{dv}{dy} = v_{01} - v_{00}$$



$$L_x^2 = \left(\frac{du}{dx} \right)^2 + \left(\frac{dv}{dx} \right)^2 \quad L_y^2 = \left(\frac{du}{dy} \right)^2 + \left(\frac{dv}{dy} \right)^2$$

$$L = \sqrt{\max(L_x^2, L_y^2)}$$

mip-map level: $d = \log_2 L$

三线性过滤 Trilinear filtering

- 对 2D 数据使用双线性滤波；
可以对 3D 数据使用三线性过滤
- 给定一个点 $(u, v, w) \in [0,1]^3$ ，
以及 8 个最近的值 f_{ijk}
- 只需迭代线性过滤：

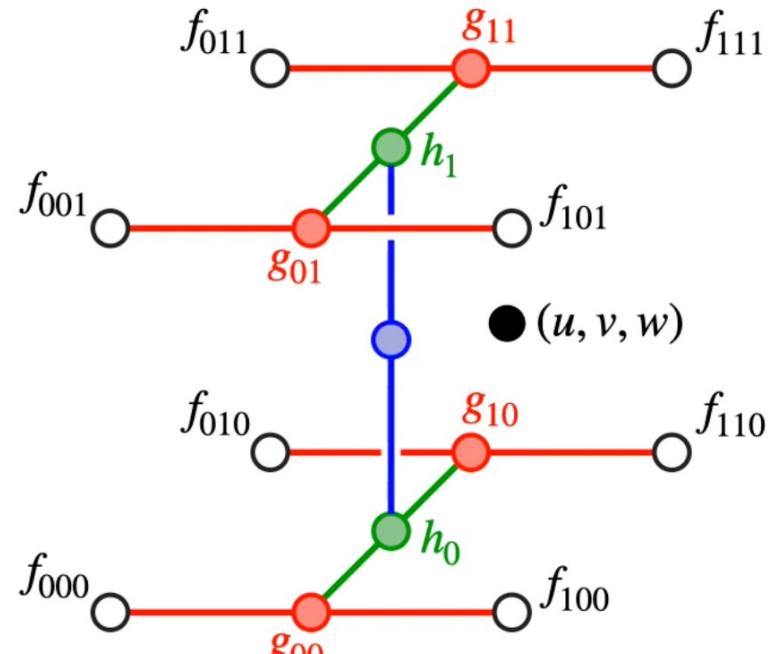
- 沿着 u 做加权平均
- 沿着 v 做加权平均
- 沿着 w 做加权平均

$$g_{00} = (1 - u)f_{000} + uf_{100}$$

$$g_{10} = (1 - u)f_{010} + uf_{110}$$

$$g_{01} = (1 - u)f_{001} + uf_{101}$$

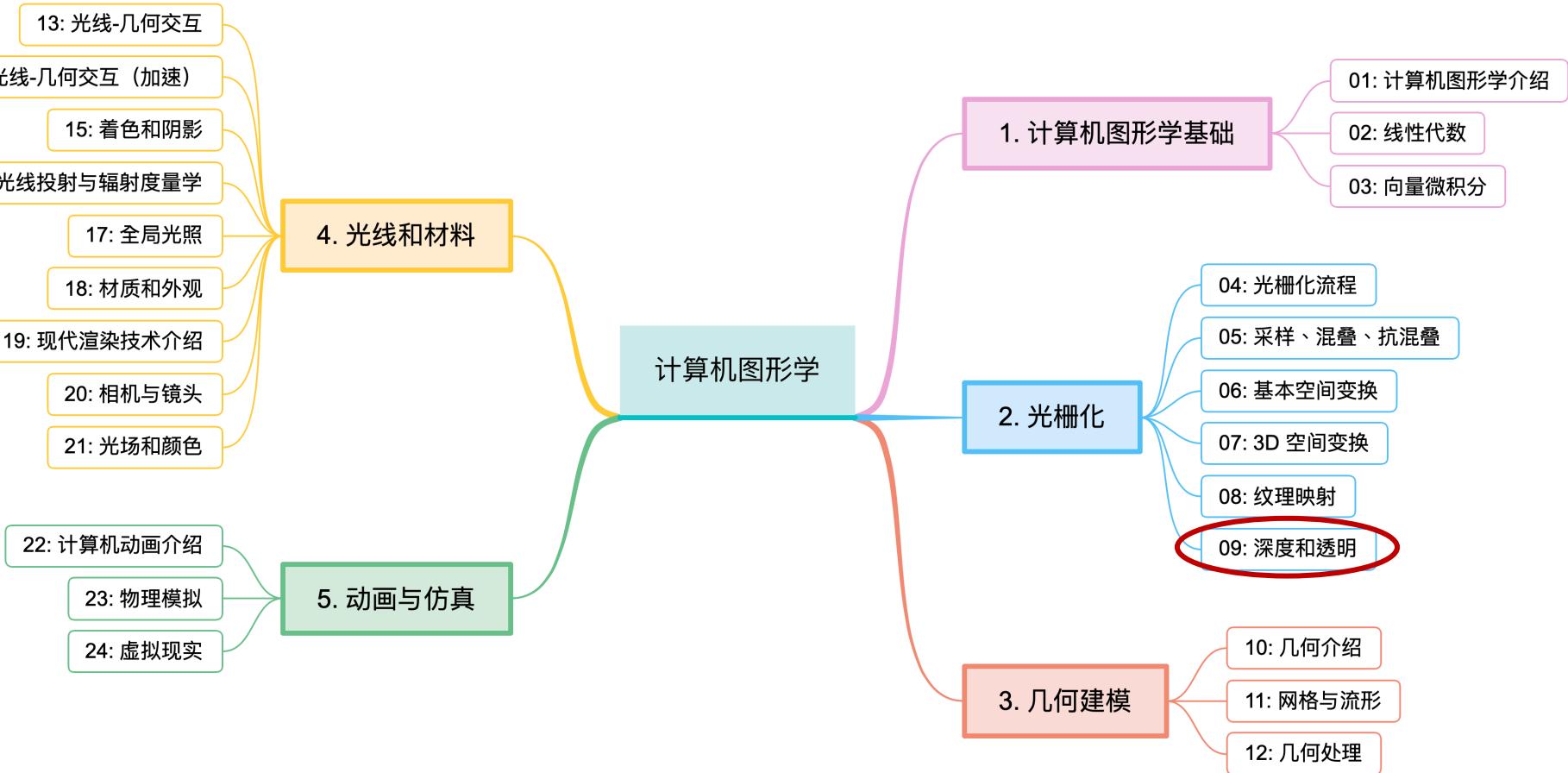
$$g_{11} = (1 - u)f_{011} + uf_{111}$$



$$h_0 = (1 - v)g_{00} + vg_{10}$$

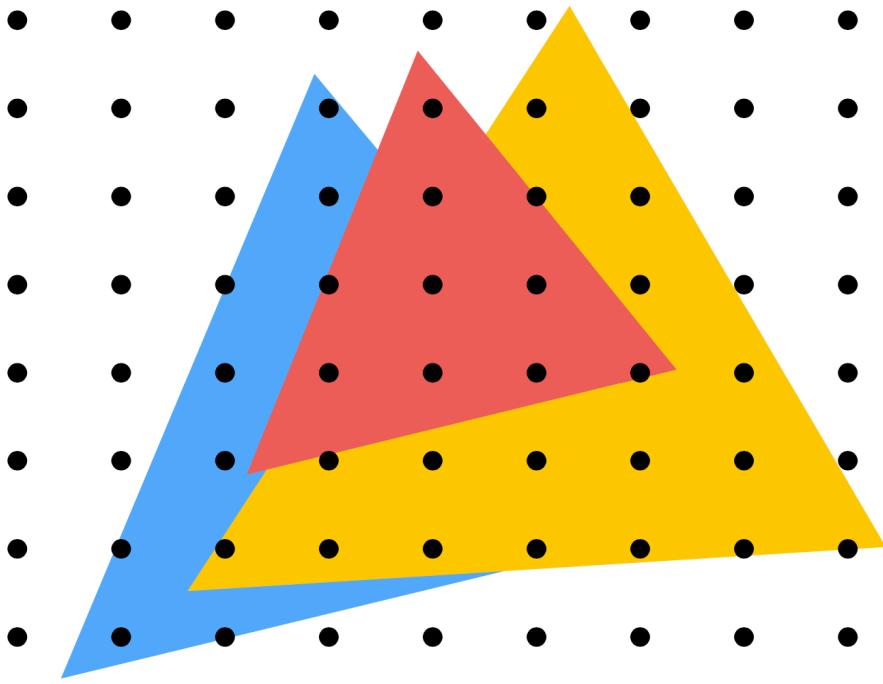
$$h_1 = (1 - v)g_{01} + vg_{11}$$

$$(1 - w)h_0 + wh_1$$

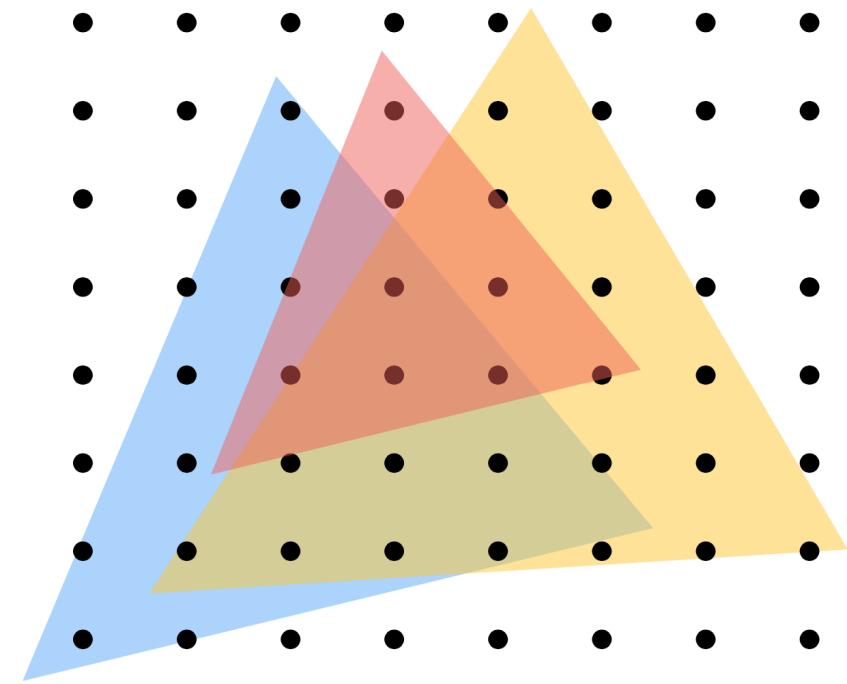


遮挡

口在每个采样点上，哪个三角形可见？



Opaque Triangles



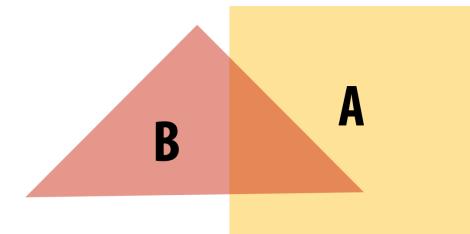
50% transparent triangles

深度缓冲示例

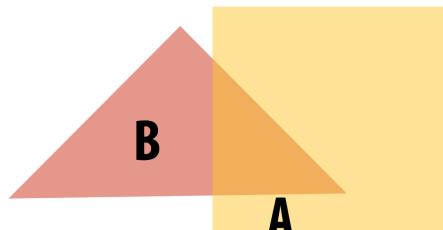


Over 算子 Operator

口将透明度为 α_A 的照片 A 与透明度为 α_B 的照片 B 混合



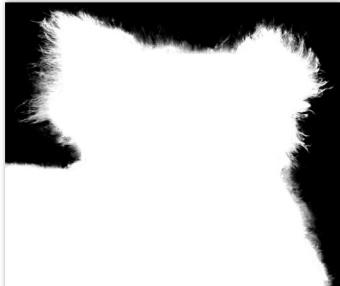
B over A



A over B

Notice: “over” is not commutative

$$A \text{ over } B \neq B \text{ over } A$$



Koala



NYC



Koala over NYC

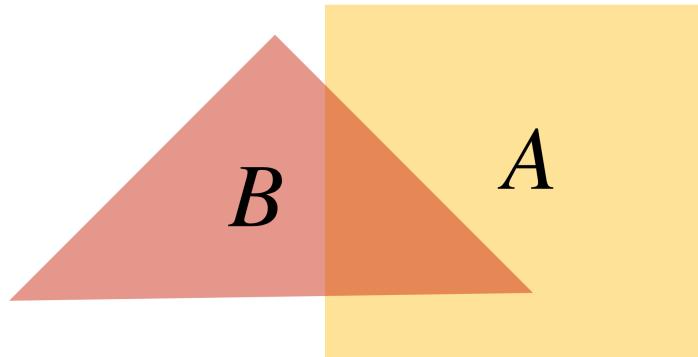
Over 算子：非预乘 α (non-premultiplied α)

□ 将透明度为 α_A 的照片 A 与透明度为 α_B 的照片 B 混合

□ 第一次尝试

$$A = (A_r, A_g, A_b)$$

$$B = (B_r, B_g, B_b)$$



□ 混合颜色

B 允许 A 通过的部分

$B \text{ over } A$

$$C = \alpha_B B + (1 - \alpha_B) \alpha_A A$$

半透明 B 的颜色

半透明 A 的颜色

□ 混合 alpha 值 $\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$

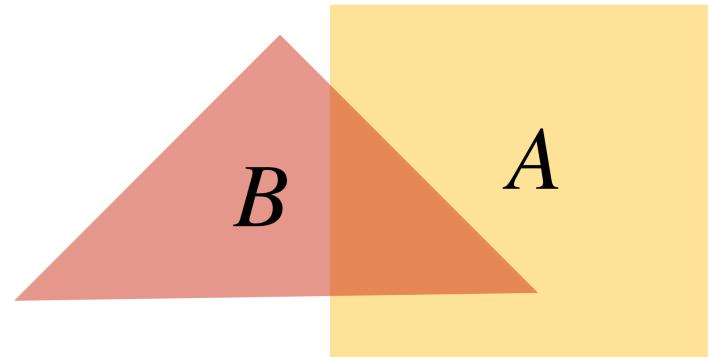
Over 算子：预乘 α (premultiplied α)

口预乘 α : 先将颜色乘以 α , 再进行混合

$$A' = (\alpha_A A_r, \alpha_A A_g, \alpha_A A_b, \alpha_A)$$

$$B' = (\alpha_B B_r, \alpha_B B_g, \alpha_B B_b, \alpha_B)$$

$$C' = B' + (1 - \alpha_B)A'$$



注意到预乘 α 混合 α 的方式与其混合 RGB 的方式一样 (非预乘则以不同的方式混合 α 和 RGB) B over A

口以 “Un-premultiply” 的方式得到最后的颜色 (除以 α_C)

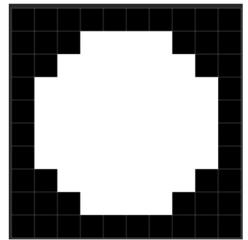
$$(C_r, C_g, C_b, \alpha_C) \Rightarrow (C_r/\alpha_C, C_g/\alpha_C, C_b/\alpha_C)$$

Q: Does this division remind you of anything?

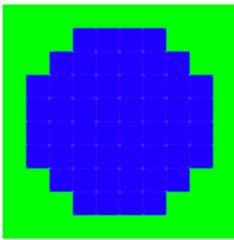
使用或不使用预乘 α 进行混合

口采样一张具有 α 通道的照片，再将其与一个背景混合

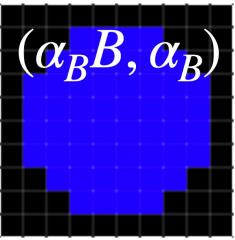
image B



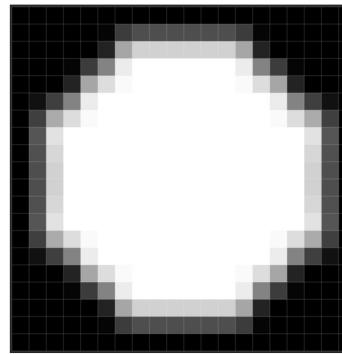
original
alpha



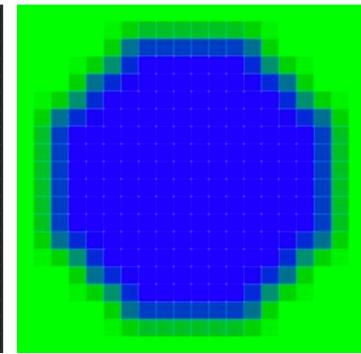
original
color



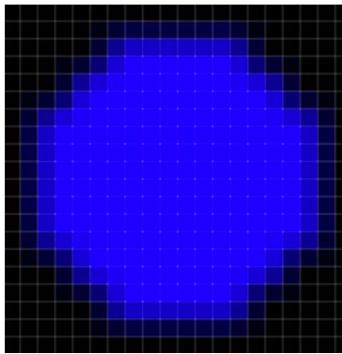
premultiplied
color



upsampled alpha



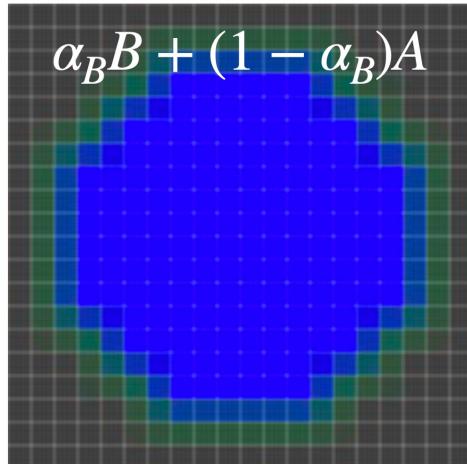
upsampled color



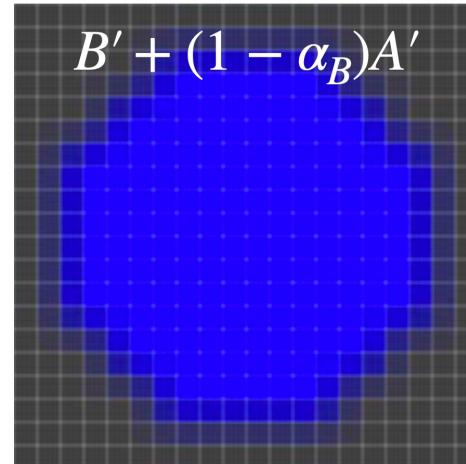
upsampled
premultiplied
color



new background A
 $(\alpha_A = 1)$

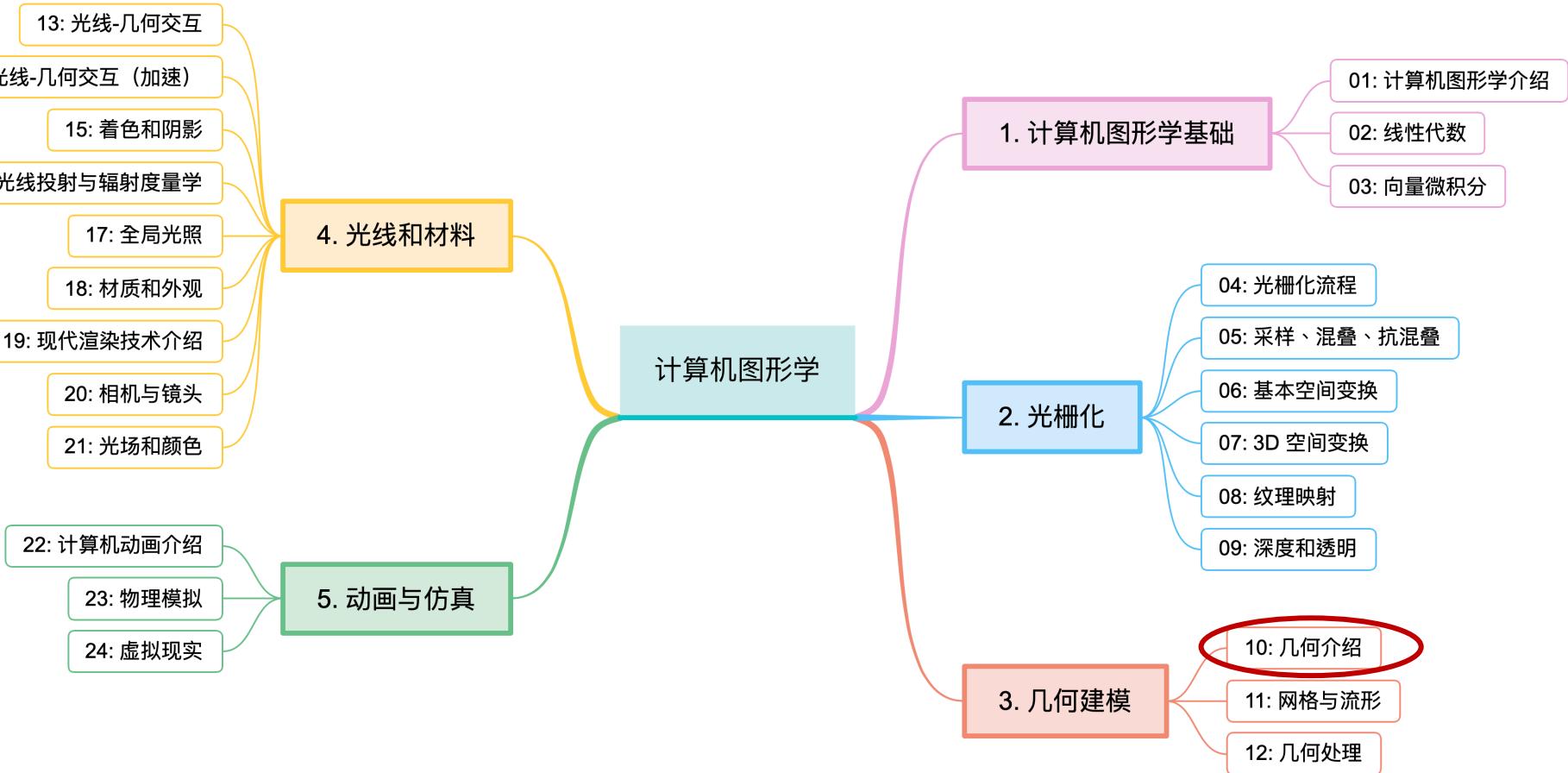


B over A
non-premultiplied



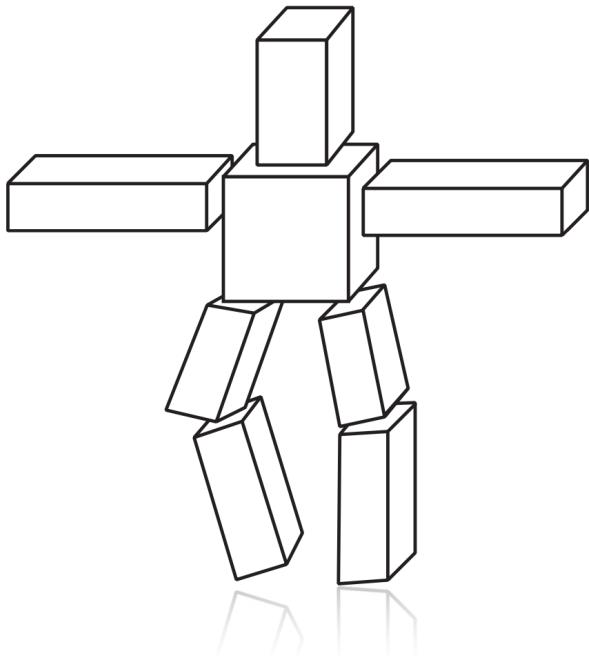
B over A
premultiplied

Q: Why do we get the “green fringe” when we don’t premultiply?



逐步提高模型的复杂度

Transformations



Geometry



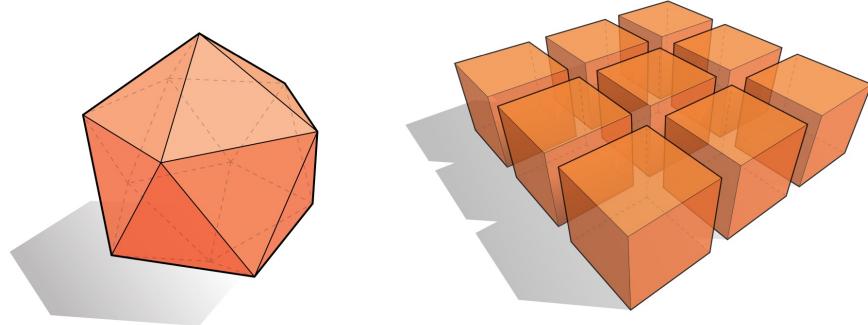
Materials, lighting



很多数字化编码几何图形的方式

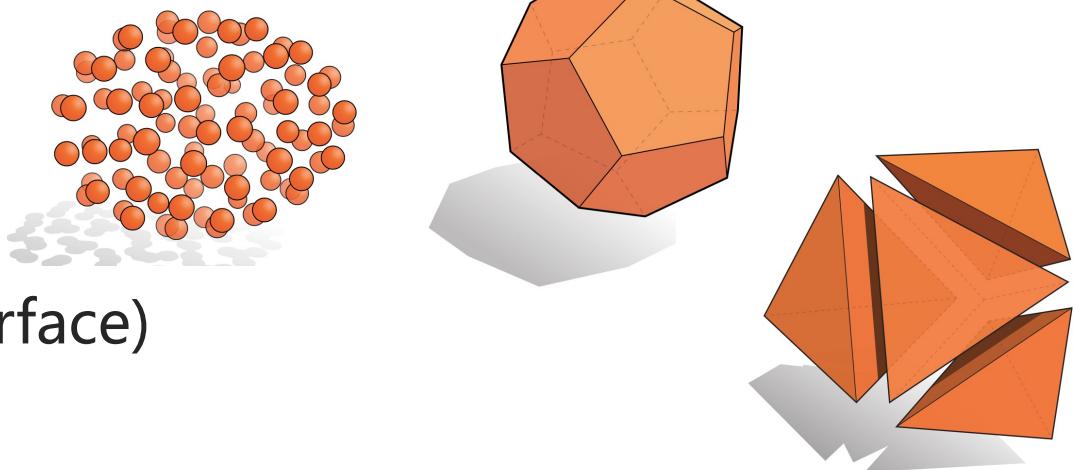
口显式的 (explicit)

- 点云 (point cloud)
- 多边形网格 (polygon mesh)
- 细分 (subdivision), NURBS
- ...



口隐式的 (implicit)

- 水平集 (level set)
- 代数曲面 (algebraic surface)
- L-系统 (L-systems)
- ...

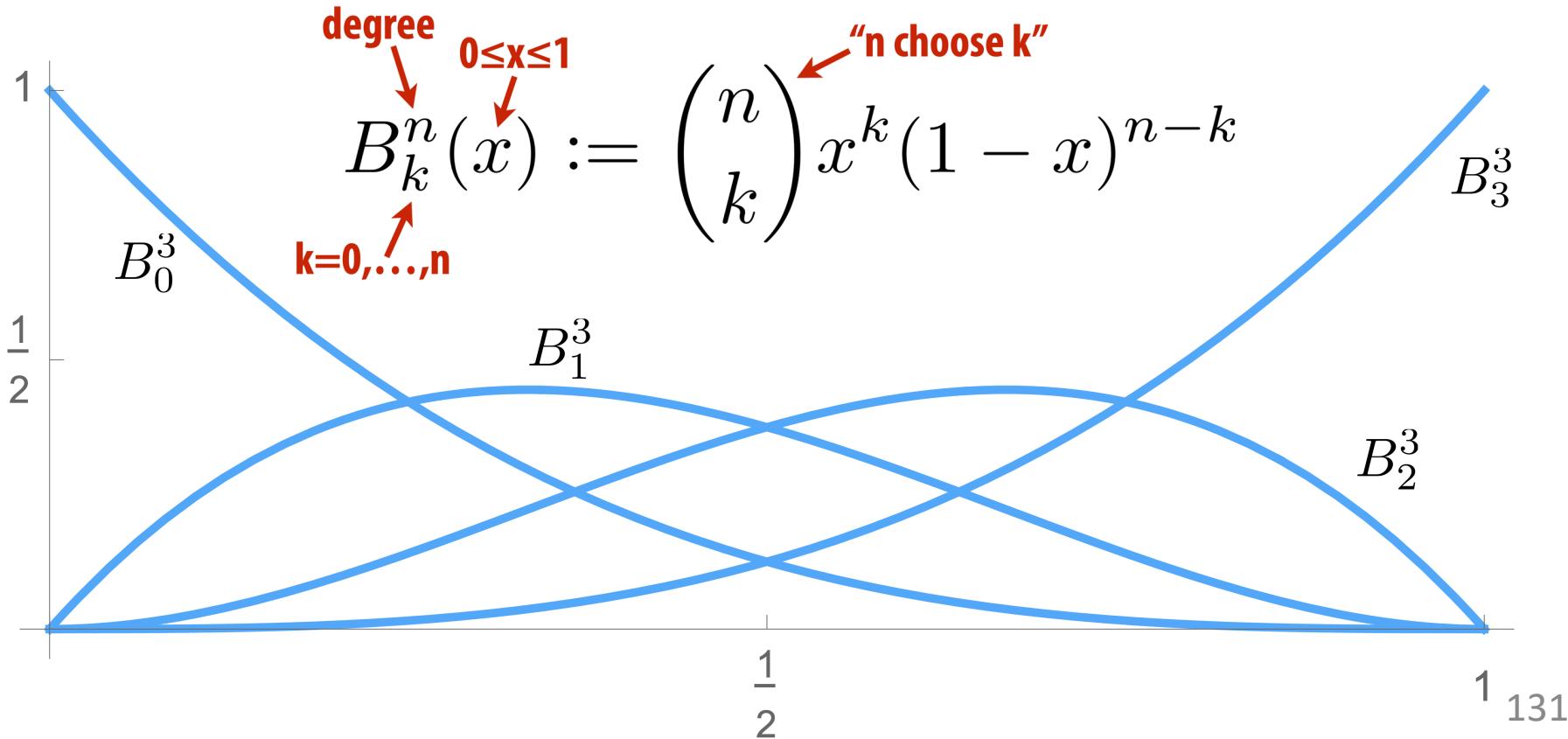


口每种选择适合不同的任务/几何体类型

口有时可能需要在不同表示方法之间来回转换

伯恩斯坦基 (Bernstein Basis)

- 线性插值本质上使用一阶多项式 (1st-order polynomials)
- 通过使用高阶多项式提供更大的灵活性
- 使用 Bernstein 基, 而不是通常的基 ($1, x, x^2, x^3, \dots$)



贝塞尔曲线 Bézier Curves (显式表达)

□ 贝塞尔曲线是利用 Bernstein 基表示的曲线

$$\gamma(s) := \sum_{k=0}^n B_{n,k}(s)p_k$$

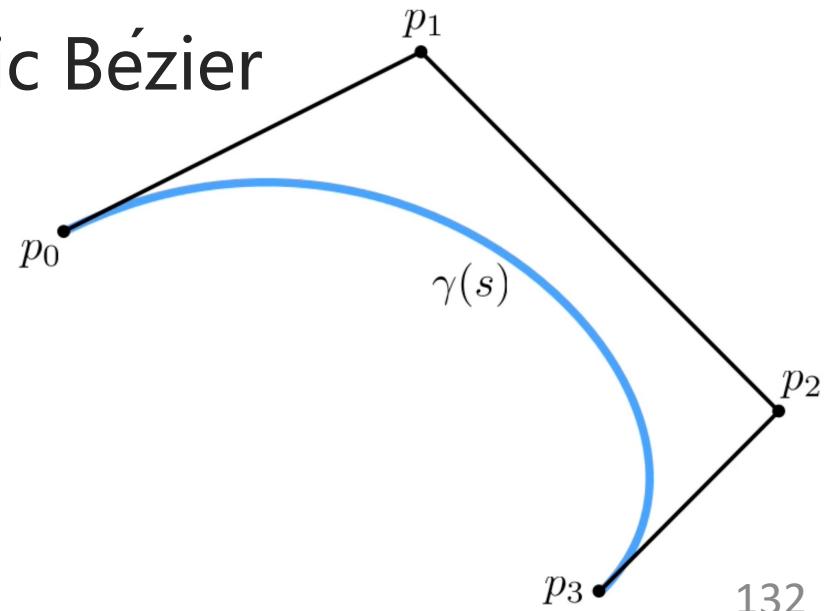
control points

□ $n = 1$, 为一条线段

□ $n = 3$, 为立方贝塞尔曲线 cubic Bézier

□ 重要特征:

- 1. 穿过端点
- 2. 与端段相切



贝塞尔曲线 Bézier Curves (显式表达)

□ 贝塞尔曲线是利用 Bernstein 基表示的曲线

$$\gamma(s) := \sum_{k=0}^n B_{n,k}(s)p_k$$

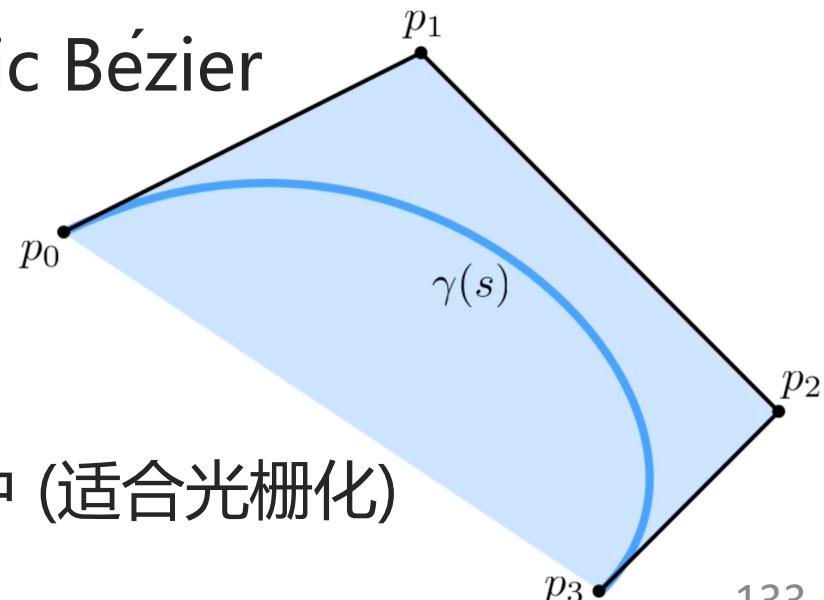
control points

□ $n = 1$, 为一条线段

□ $n = 3$, 为立方贝塞尔曲线 cubic Bézier

□ 重要特征:

- 1. 穿过端点
- 2. 与端段相切
- 3. 包含在凸包 (convex hull) 中 (适合光栅化)

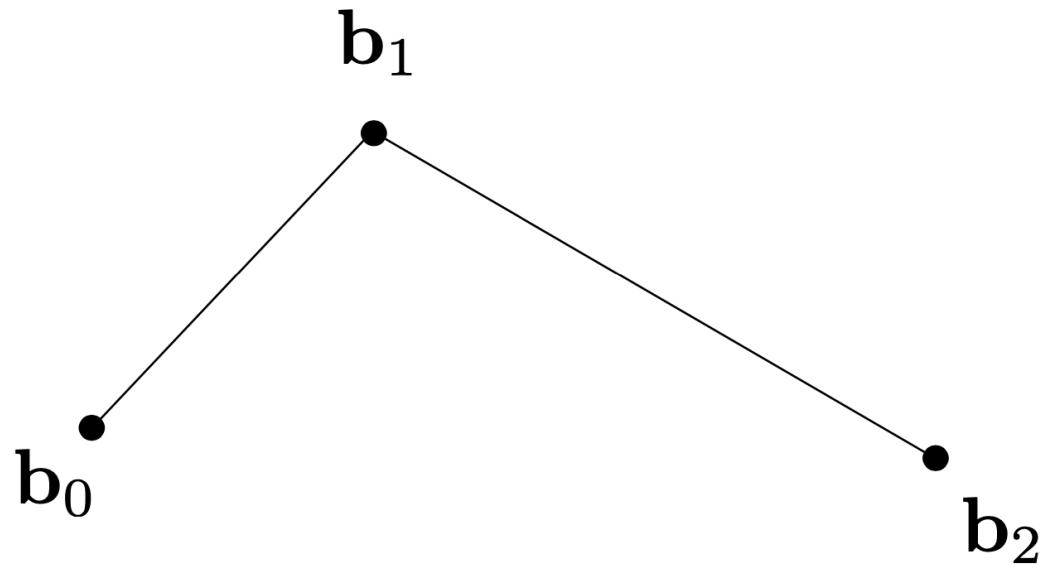


为什么要用点来控制曲线？

- 更灵活、精确地创建和修改曲线，不需要记录每一个点
- 更容易实现光滑、符合物理特性点曲线
- 更容易实现平滑的动画效果
- 更容易存储数据
- ...

Bézier Curves – de Casteljau Algorithm

考虑三个点的贝塞尔曲线 (quadratic Bezier)



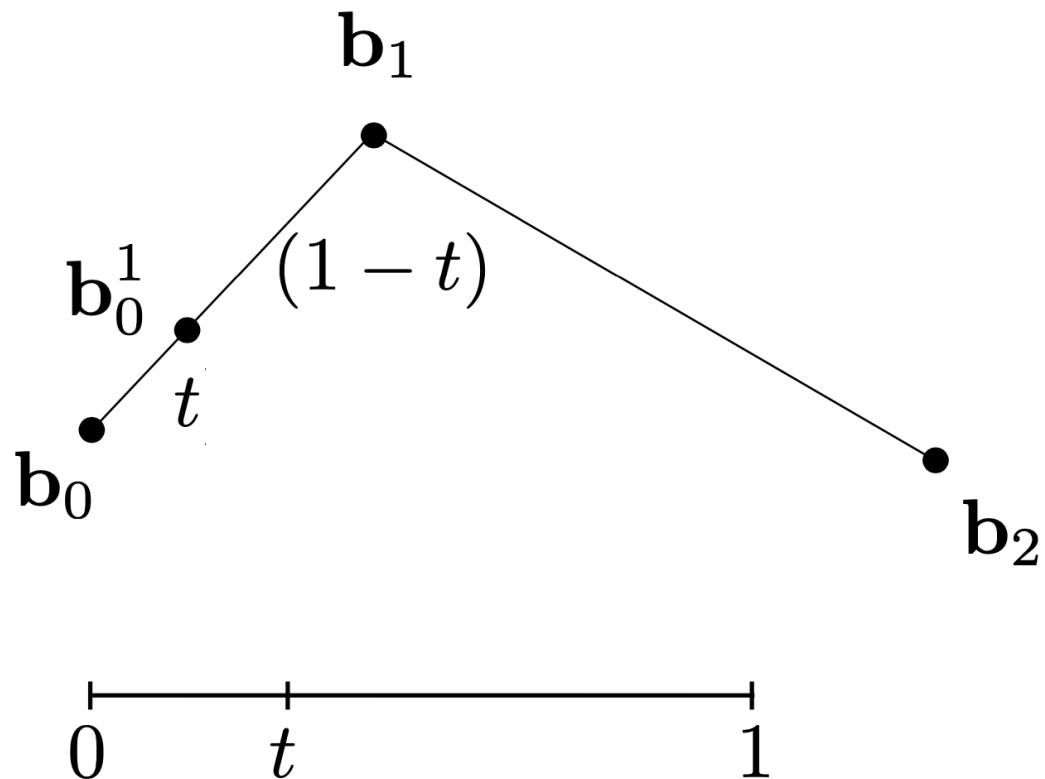
Pierre Bézier
1910 – 1999



Paul de Casteljau
b. 1930

Bézier Curves – de Casteljau Algorithm

口在直线 b_0, b_1 中利用线性插值插入一个点 b_0^1



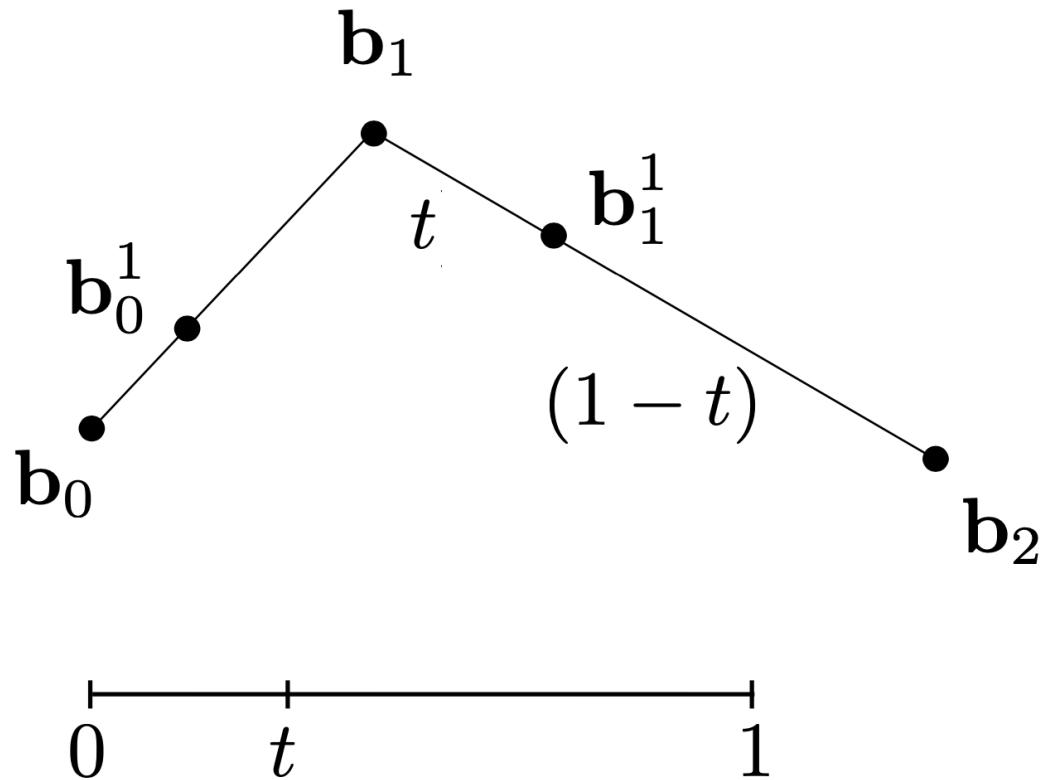
Pierre Bézier
1910 – 1999



Paul de Casteljau
b. 1930

Bézier Curves – de Casteljau Algorithm

□ 在直线 b_1, b_2 中同样插入一个点 b_1^1



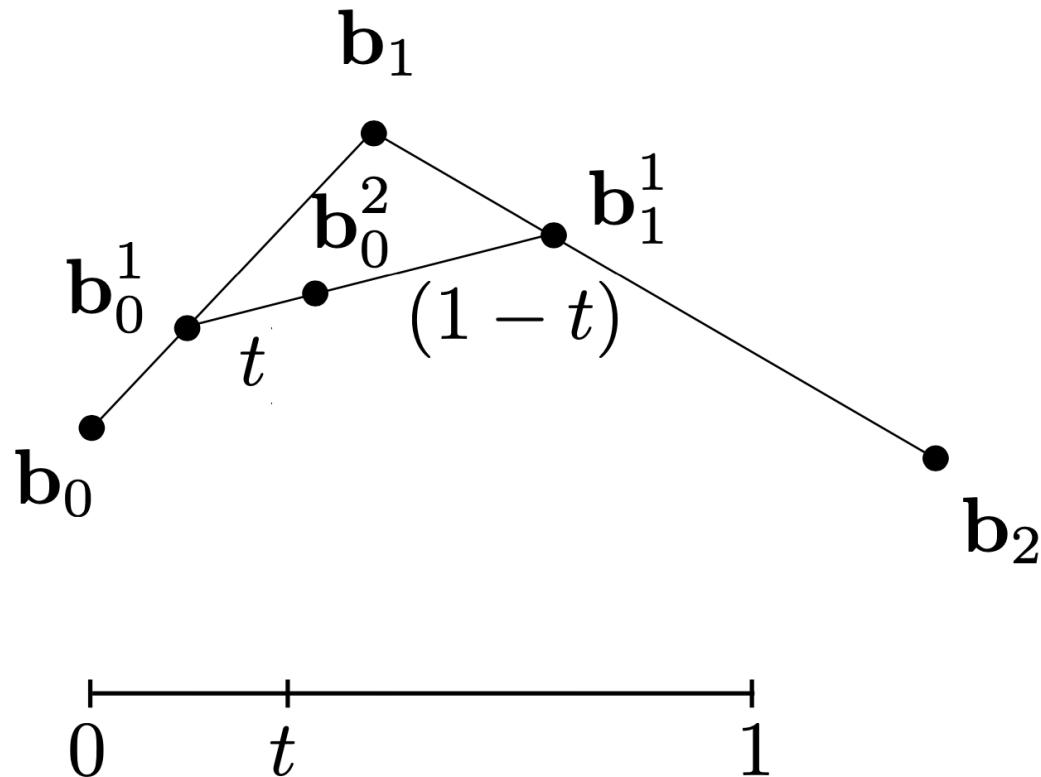
Pierre Bézier
1910 – 1999



Paul de Casteljau
b. 1930

Bézier Curves – de Casteljau Algorithm

□ 在新的直线 b_0^1, b_2 中插入一个点 b_1^1



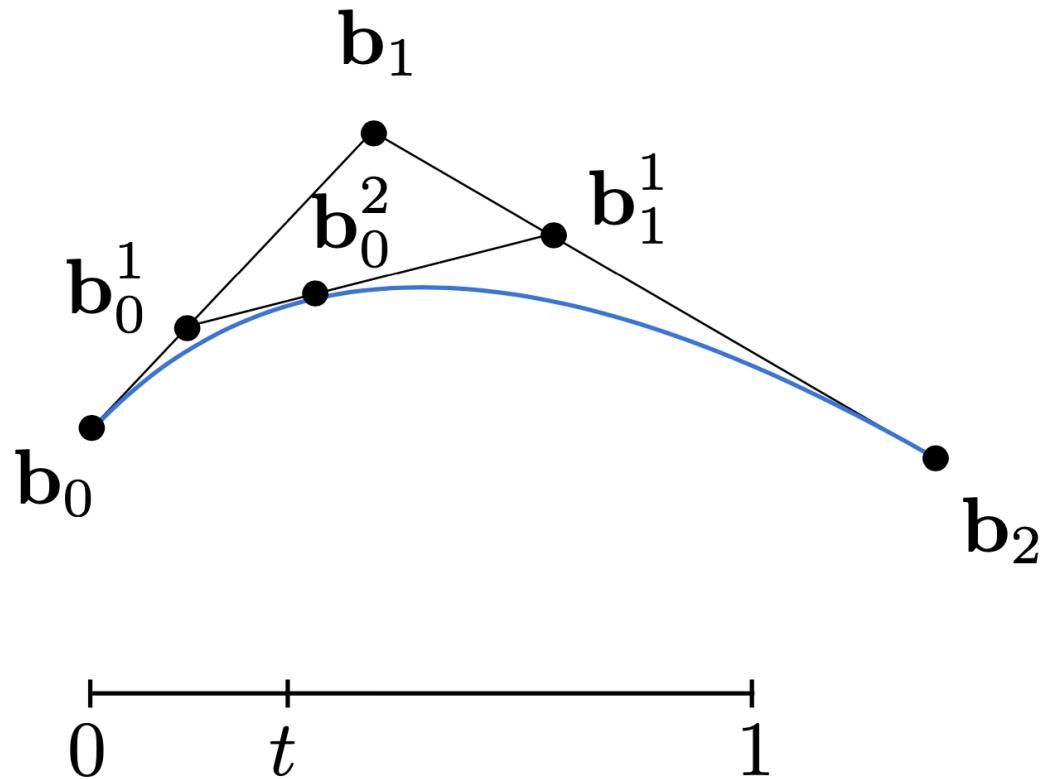
Pierre Bézier
1910 – 1999



Paul de Casteljau
b. 1930

Bézier Curves – de Casteljau Algorithm

对 $[0,1]$ 中的每个 t 运行相同的算法



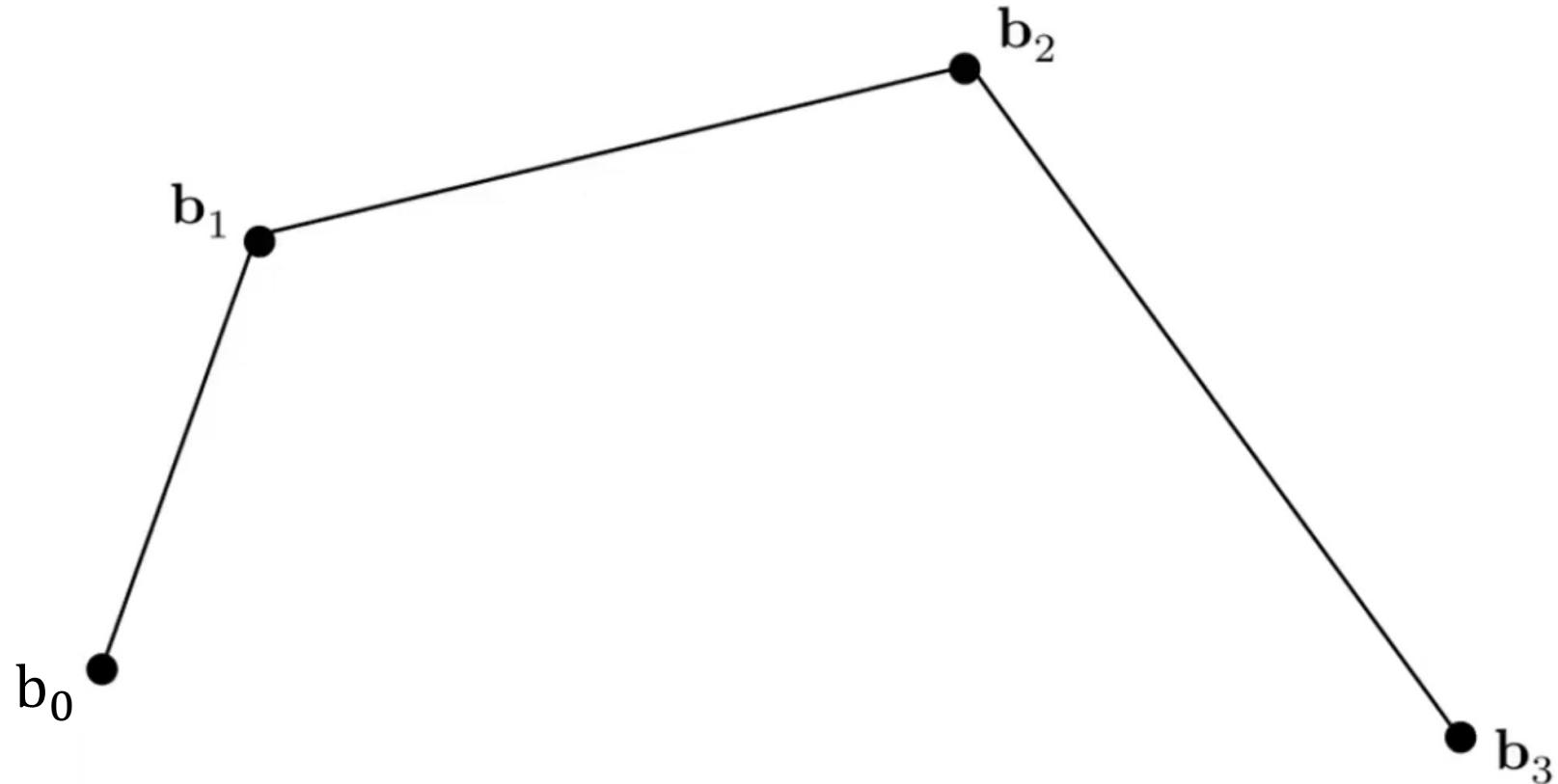
Pierre Bézier
1910 – 1999



Paul de Casteljau
b. 1930

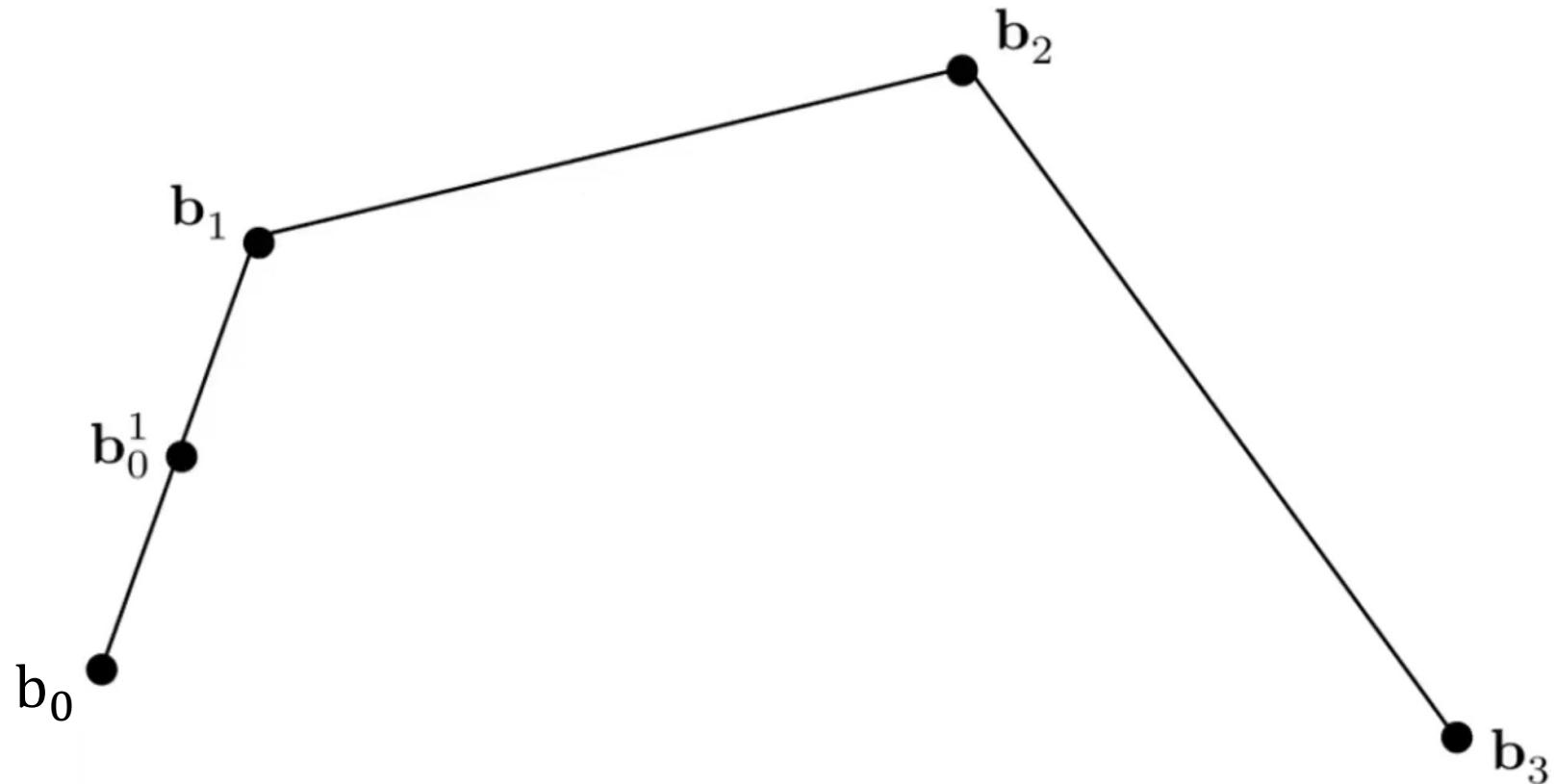
Cubic Bézier Curve – de Casteljau

- 共四个输入点
- 相同的递归线性插值



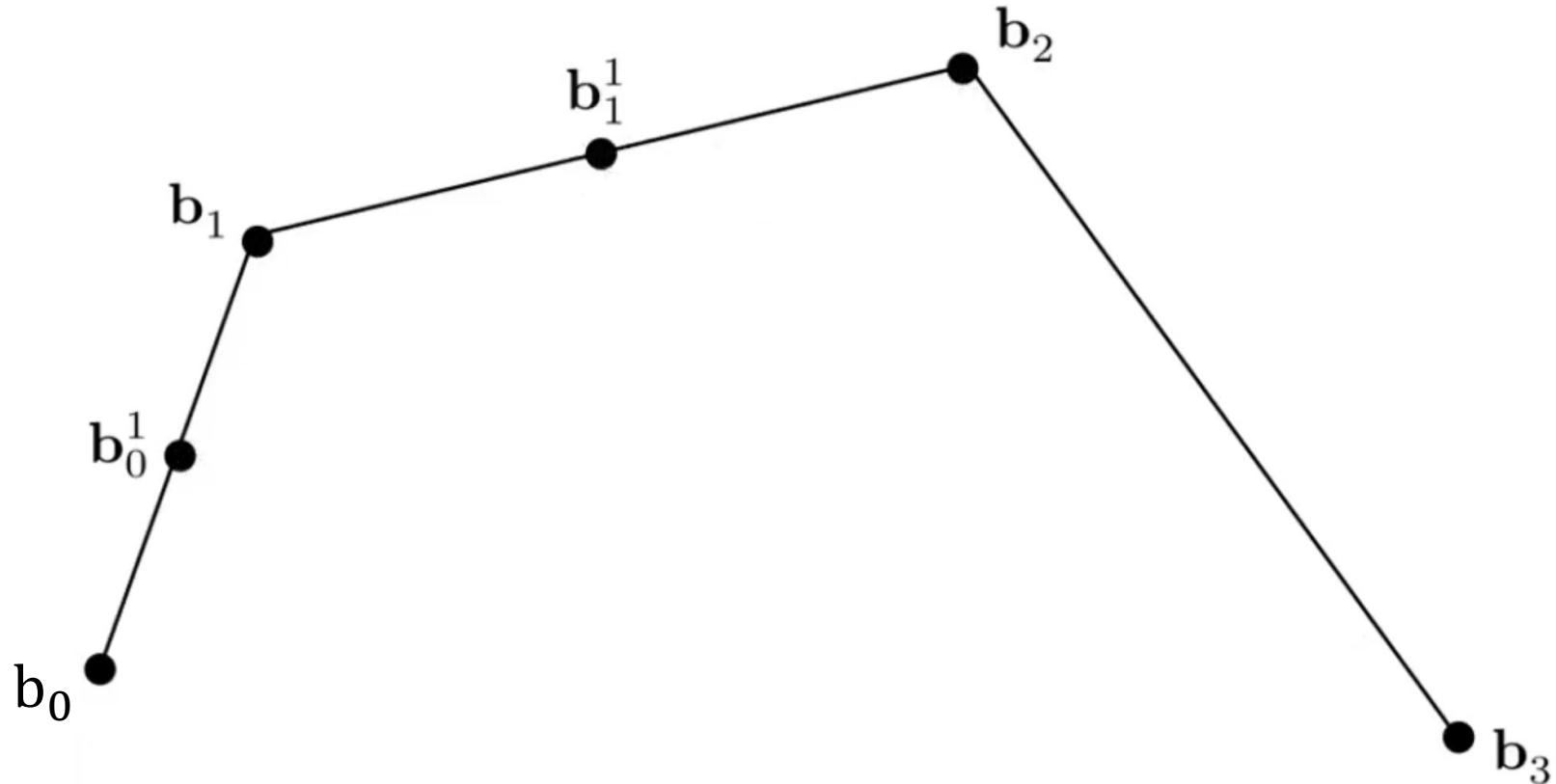
Cubic Bézier Curve – de Casteljau

- 共四个输入点
- 相同的递归线性插值



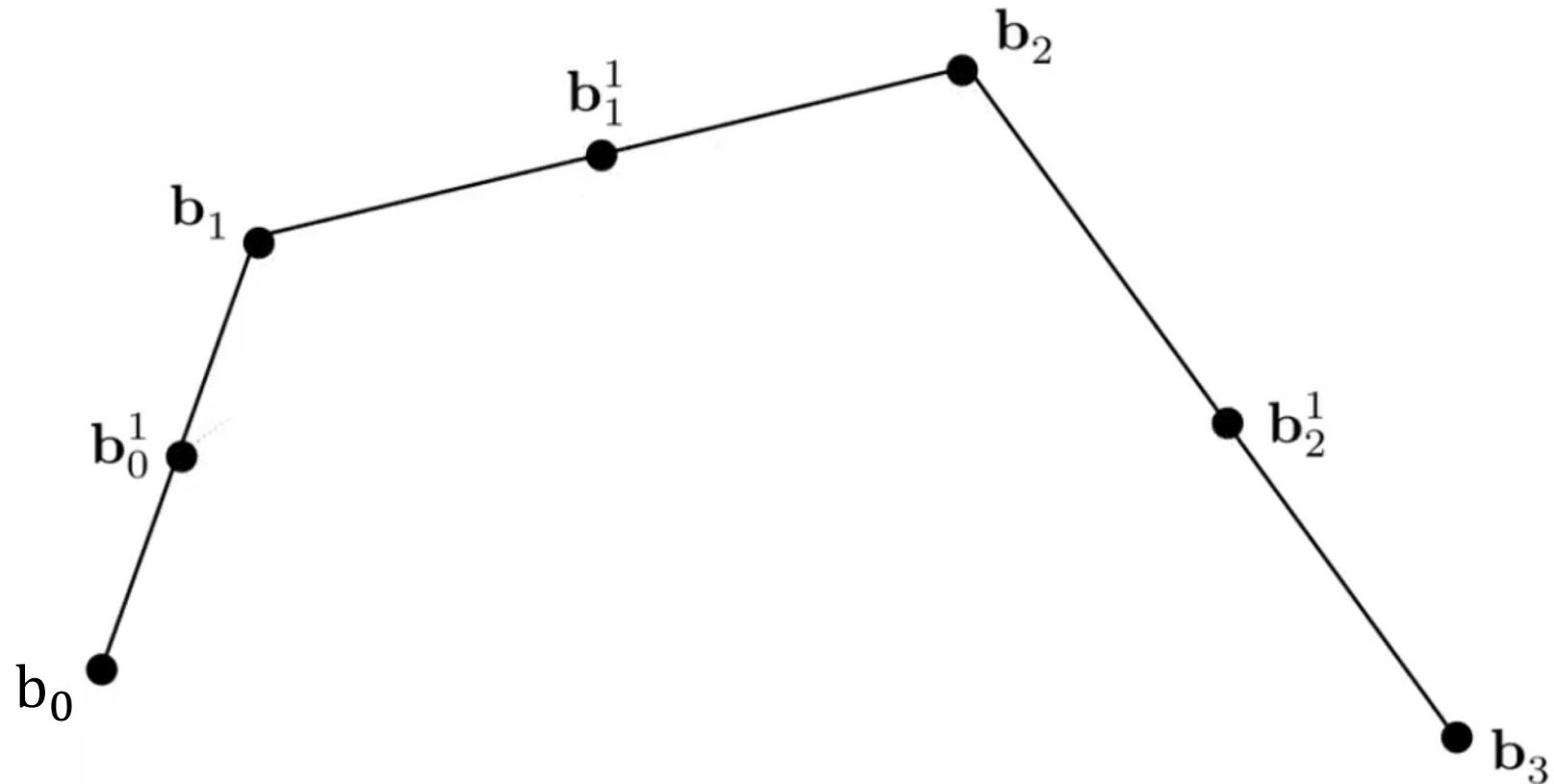
Cubic Bézier Curve – de Casteljau

- 共四个输入点
- 相同的递归线性插值



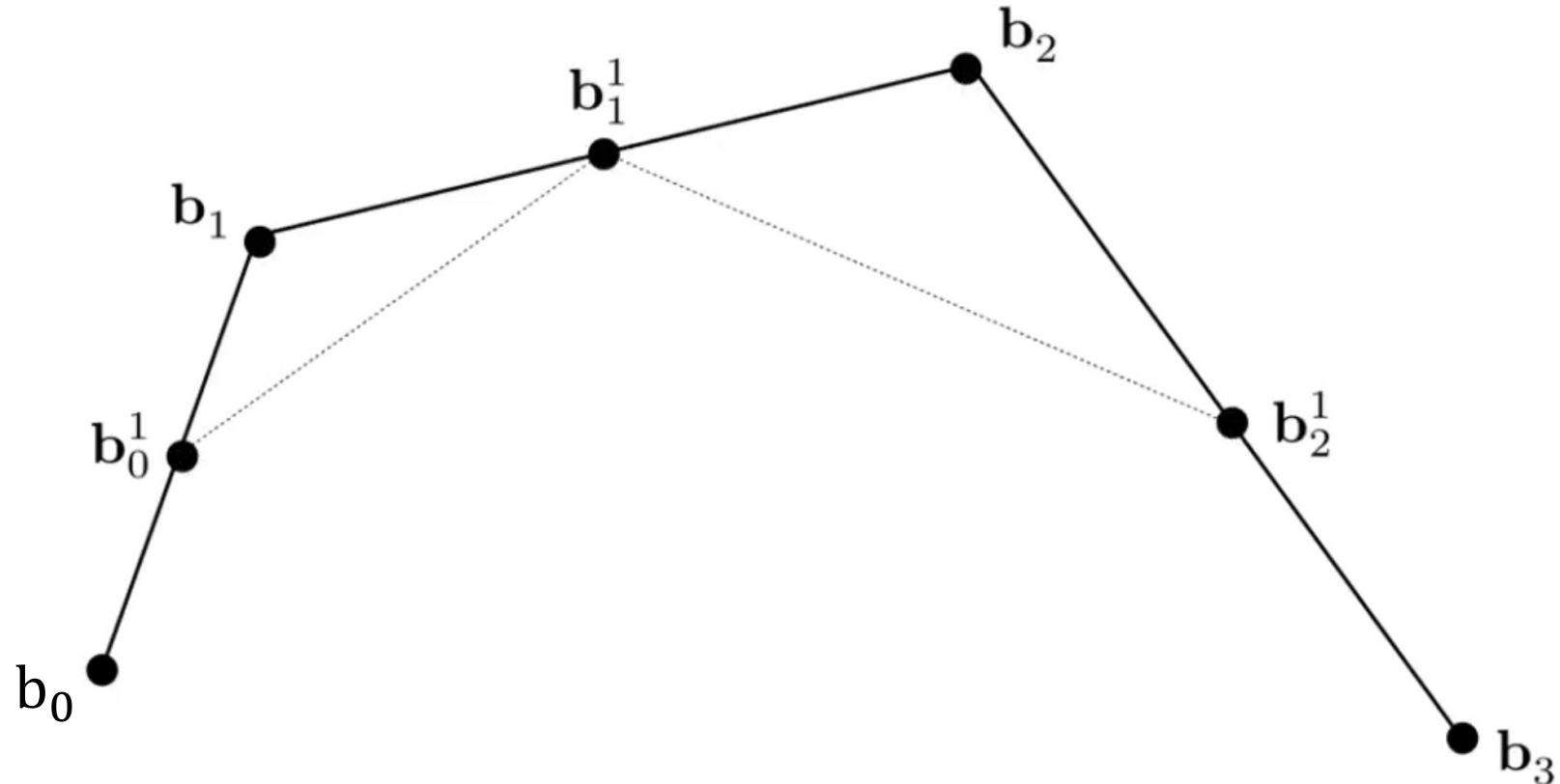
Cubic Bézier Curve – de Casteljau

- 共四个输入点
- 相同的递归线性插值



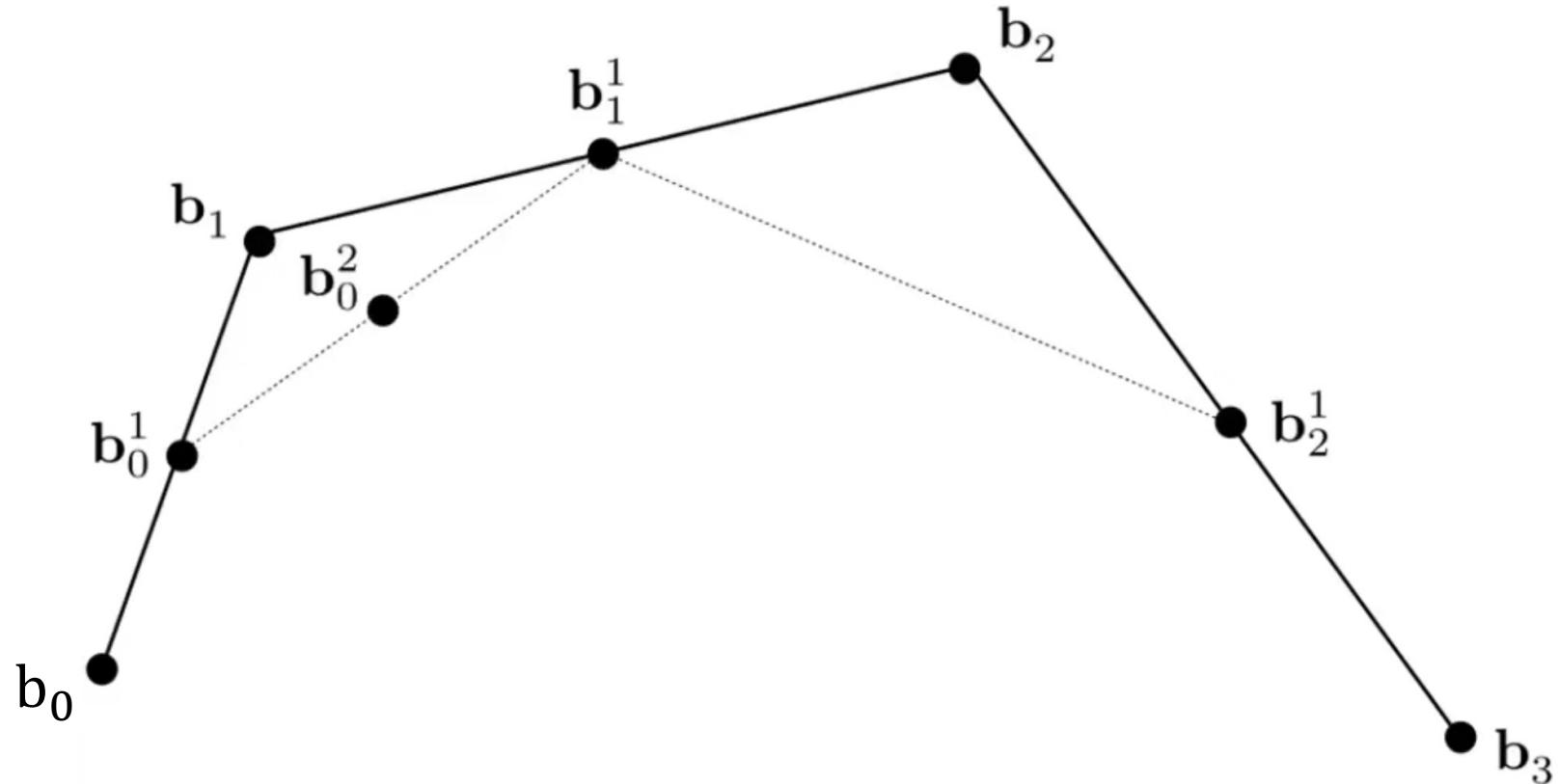
Cubic Bézier Curve – de Casteljau

- 共四个输入点
- 相同的递归线性插值



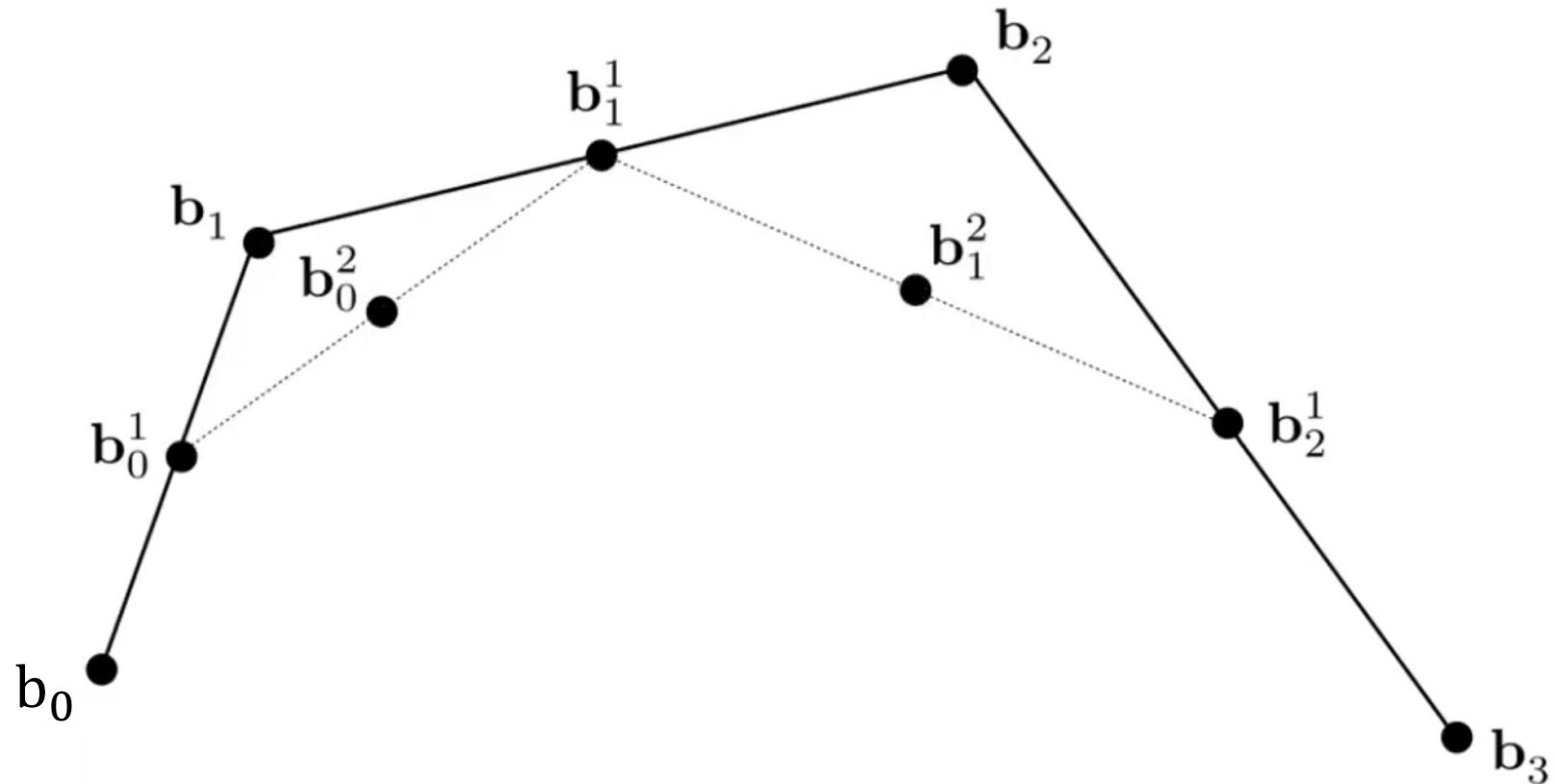
Cubic Bézier Curve – de Casteljau

- 共四个输入点
- 相同的递归线性插值



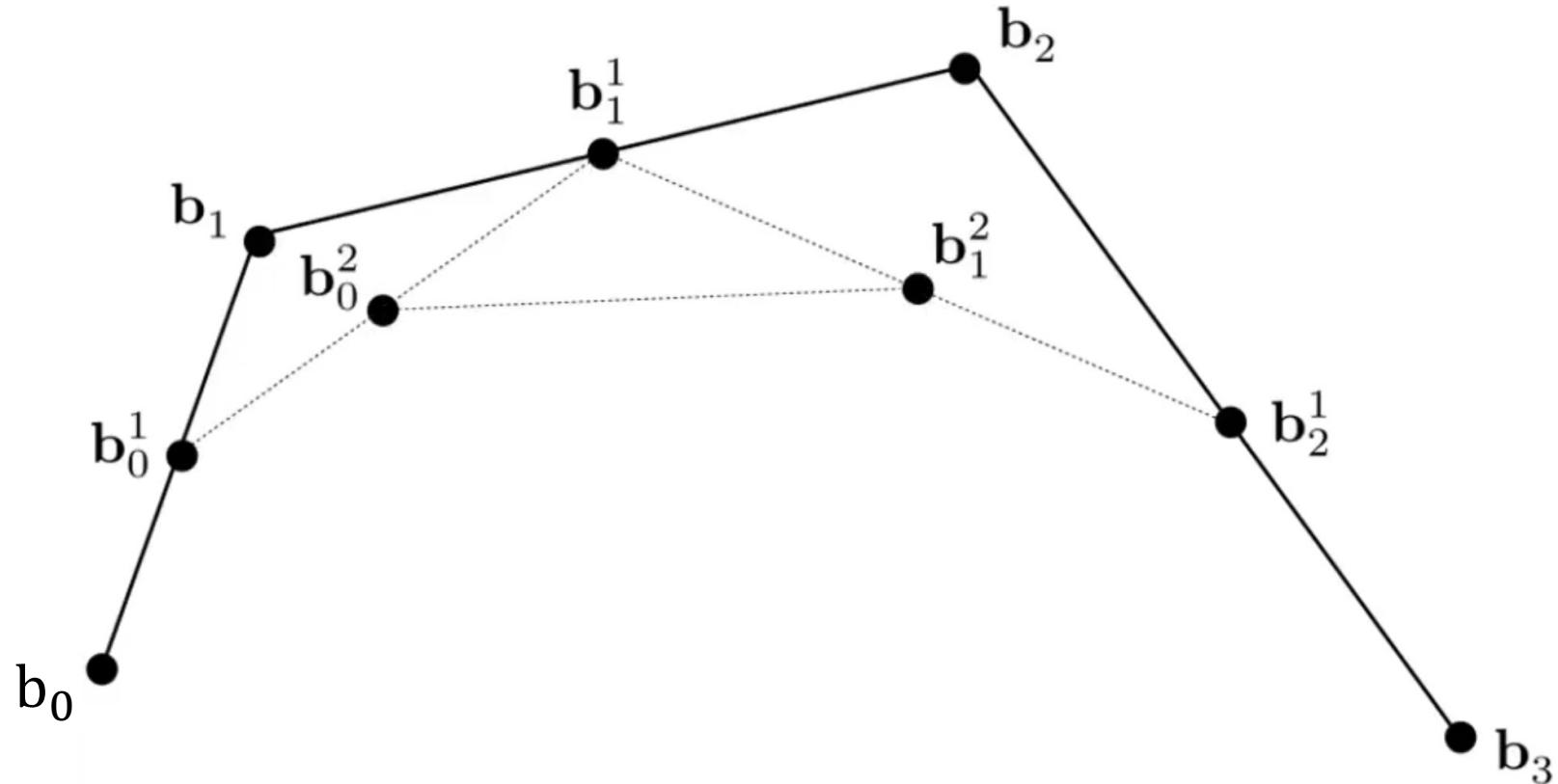
Cubic Bézier Curve – de Casteljau

- 共四个输入点
- 相同的递归线性插值



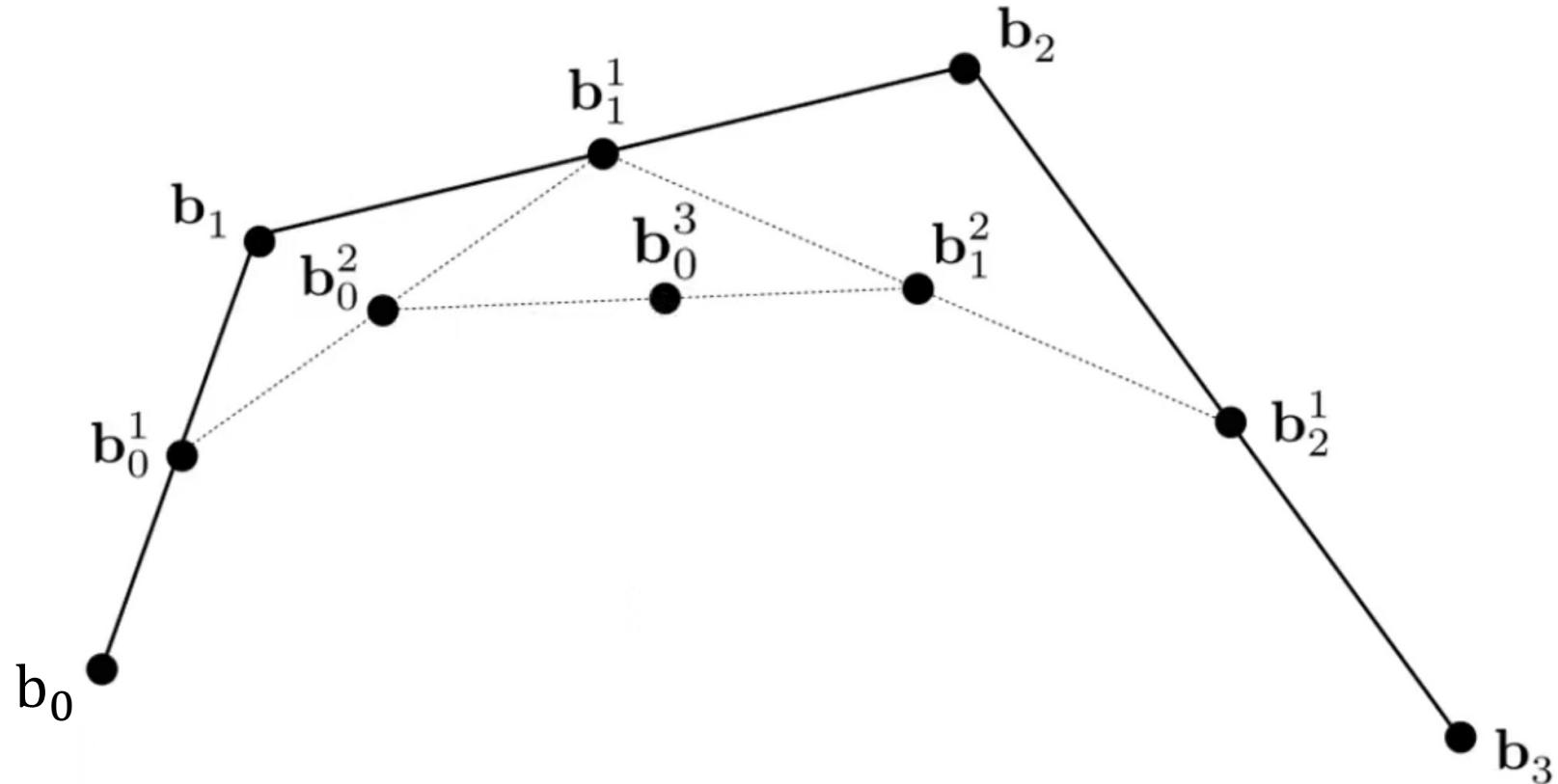
Cubic Bézier Curve – de Casteljau

- 共四个输入点
- 相同的递归线性插值



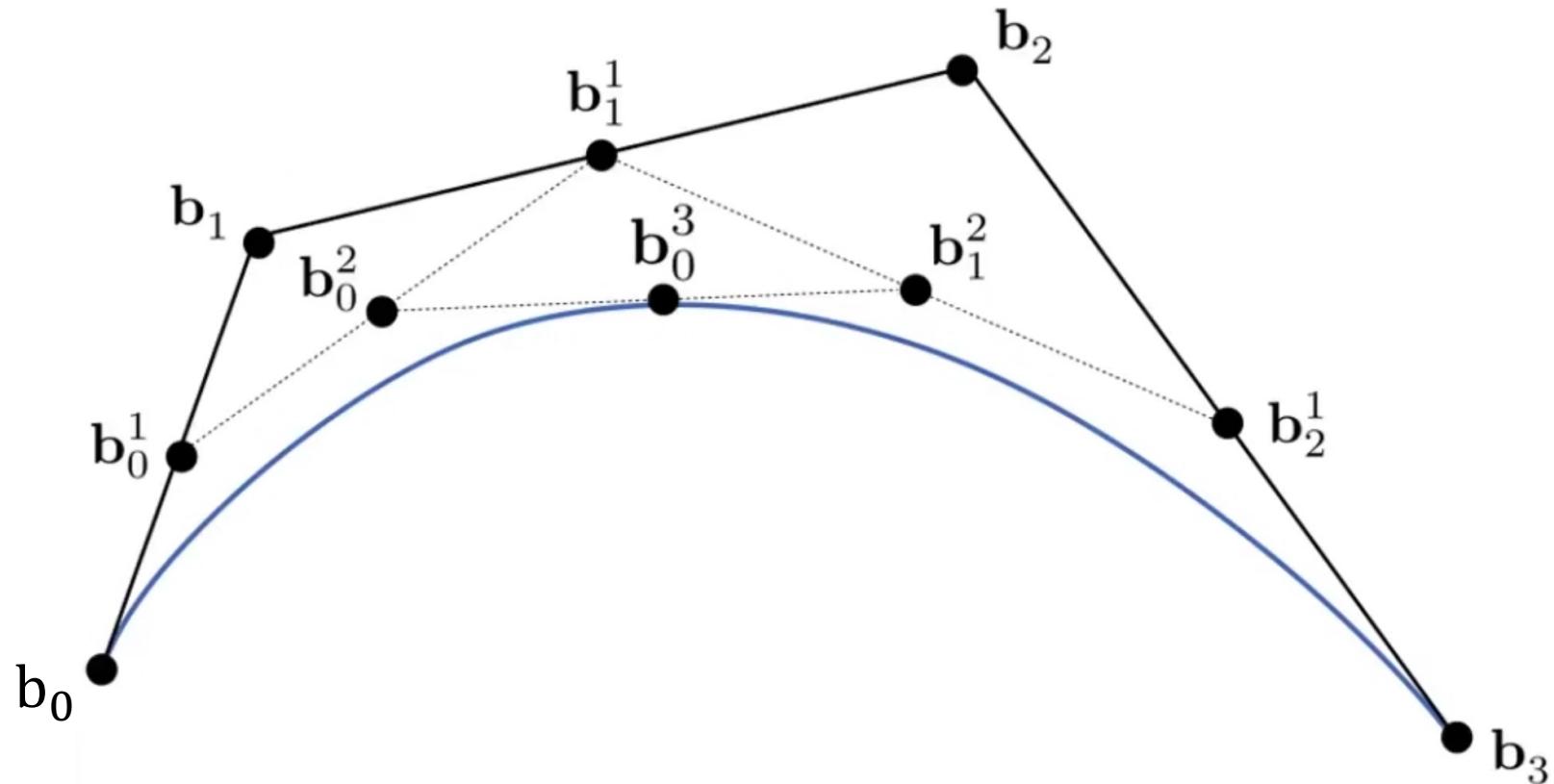
Cubic Bézier Curve – de Casteljau

- 共四个输入点
- 相同的递归线性插值



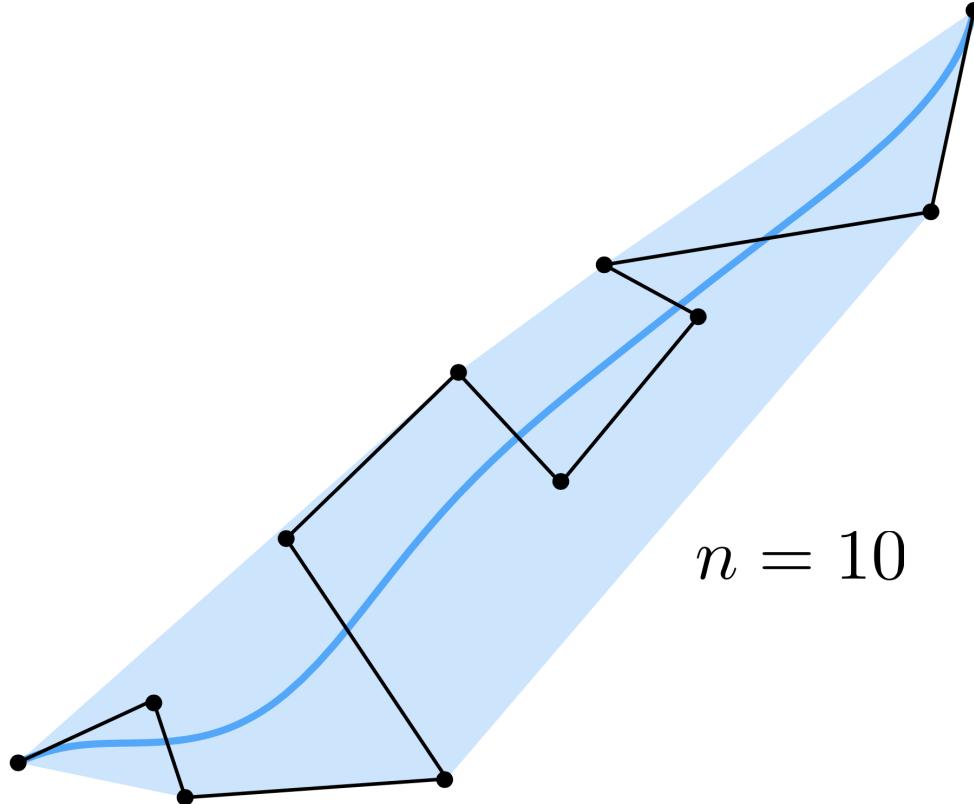
Cubic Bézier Curve – de Casteljau

- 共四个输入点
- 相同的递归线性插值



高阶贝塞尔曲线

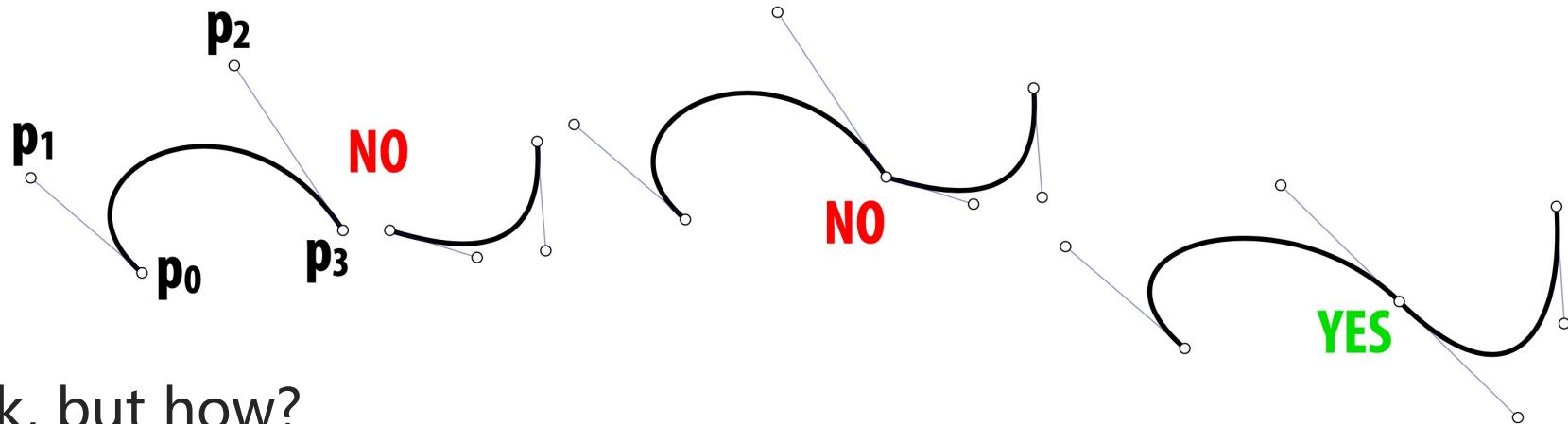
口高阶 Bernstein 多项式在插值上表现并不好



Very hard to control!

贝塞尔曲线 – 切线连续性 (tangent continuity)

□ 要获得无缝自然的曲线，需要点和切线对齐



□ Ok, but how?

□ 每条曲线是一个立方曲线

$$u^3 p_0 + 3u^2(1-u)p_1 + 3u(1-u)^2 p_2 + (1-u)^3 p_3$$

□ 希望每个分段的端点相交

□ 希望端点处的切线对齐

□ Q: 有多少约束 constraints 与自由度 degrees of freedom?

□ Q: 二次贝塞尔曲线呢？线性贝塞尔曲线呢？

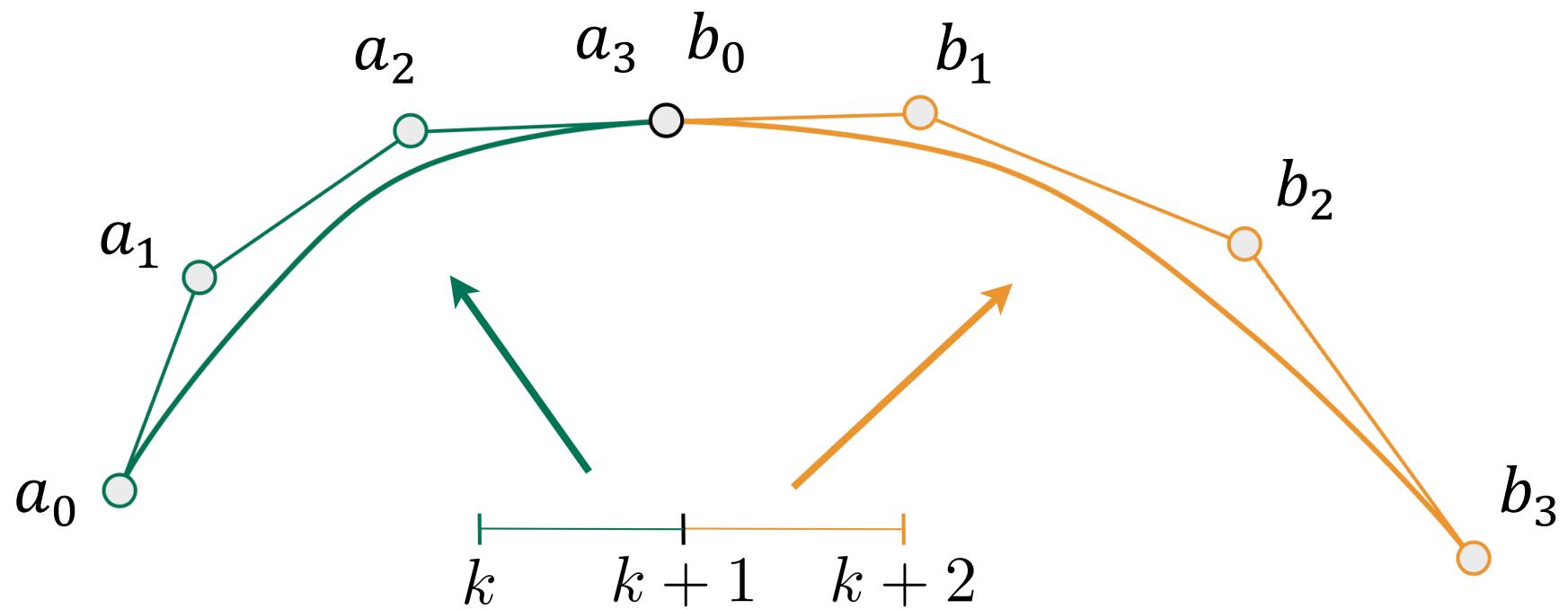
贝塞尔曲线 – 连续性 (Continuity)

两条贝塞尔曲线

$$\mathbf{a} : [k, k + 1] \rightarrow \mathbb{R}^N$$

$$\mathbf{b} : [k + 1, k + 2] \rightarrow \mathbb{R}^N$$

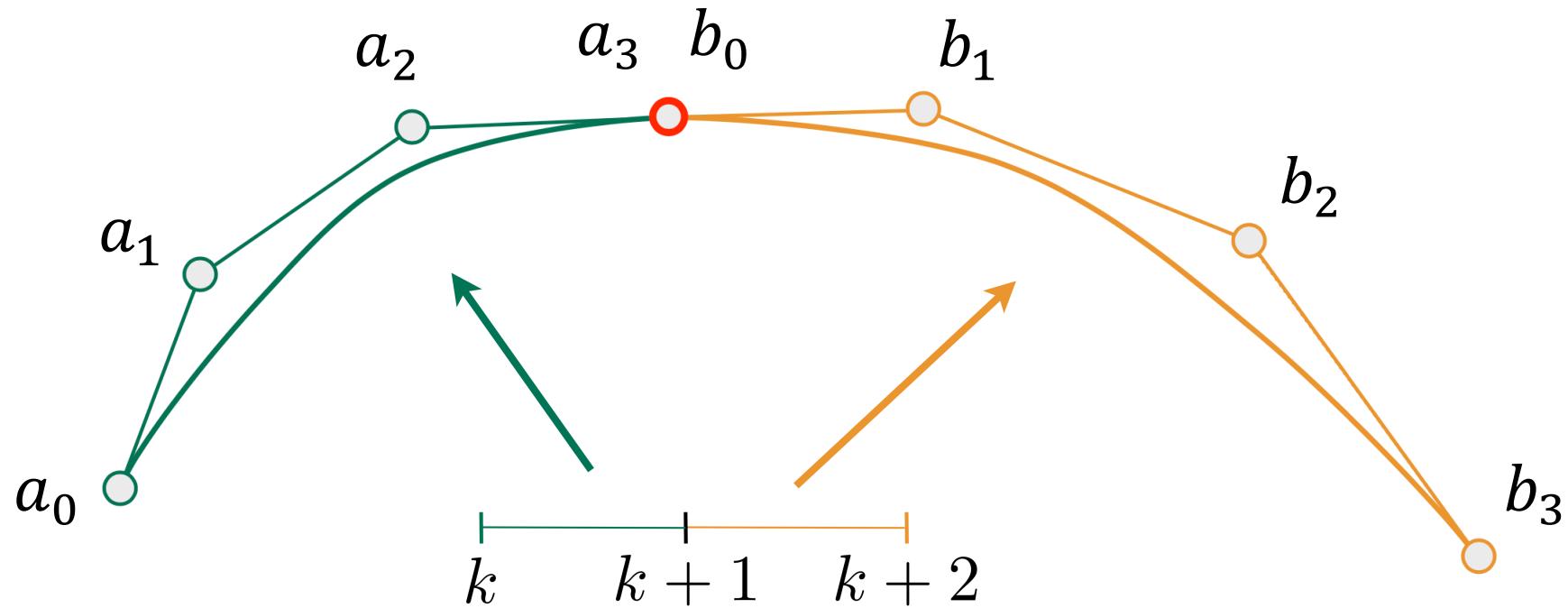
Assuming integer partitions here,
can generalize



贝塞尔曲线 – 连续性 (Continuity)

□ C^0 连续性 (仅要求首尾相连)

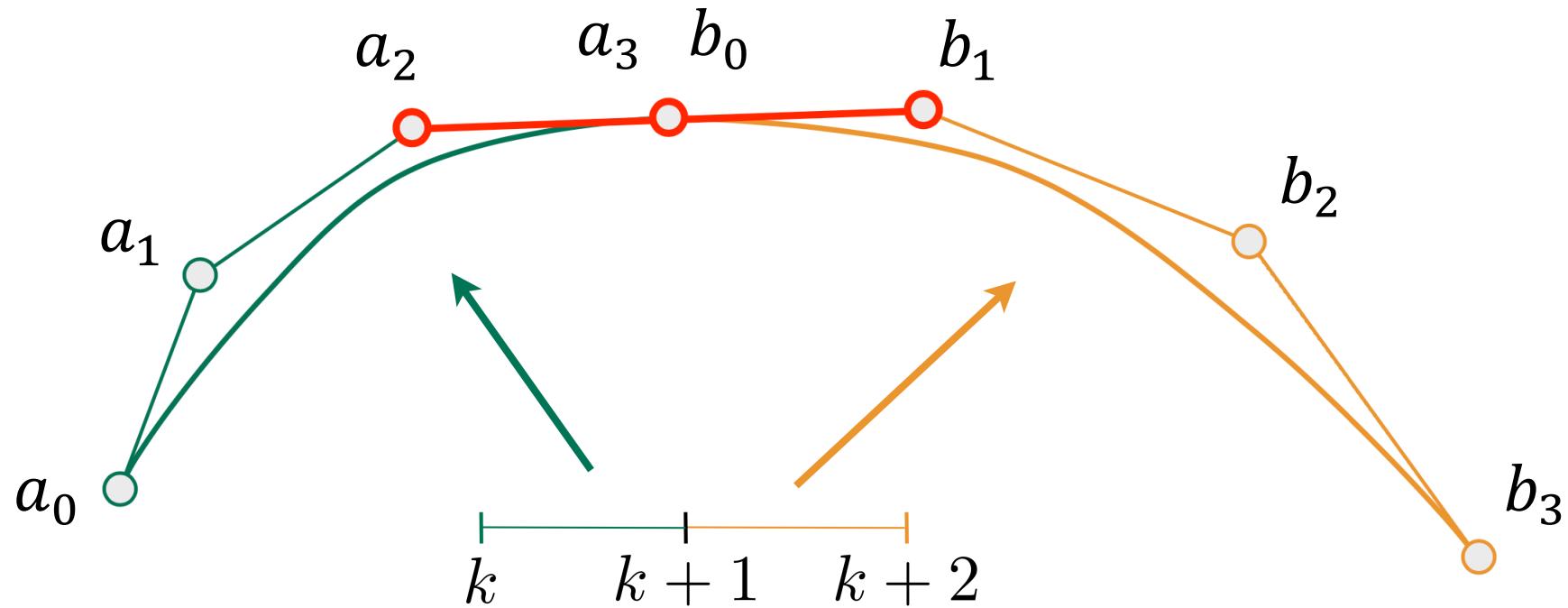
$$a_3 = b_0$$

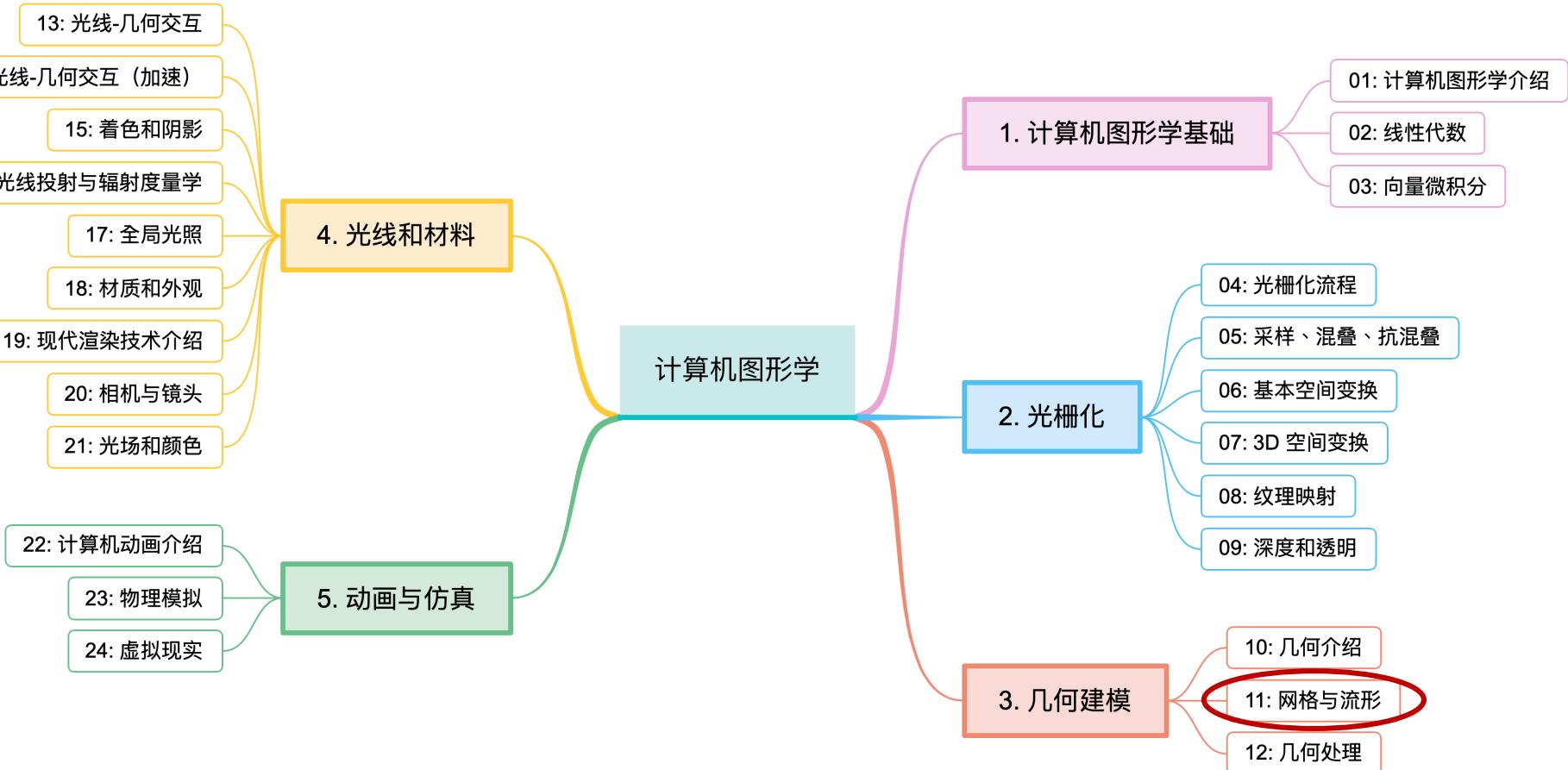


贝塞尔曲线 – 连续性 (Continuity)

□ C^1 连续性 (仅要求首尾相连)

$a_3 = b_0$ 且线段 (a_2, a_3) 和 (b_0, b_1) 在一条直线上

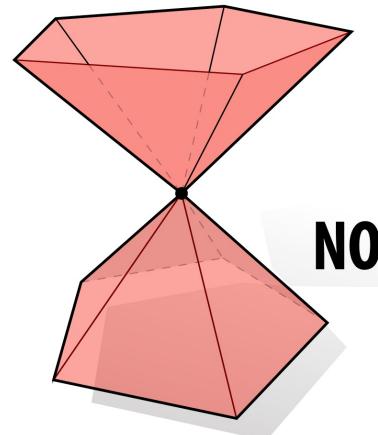
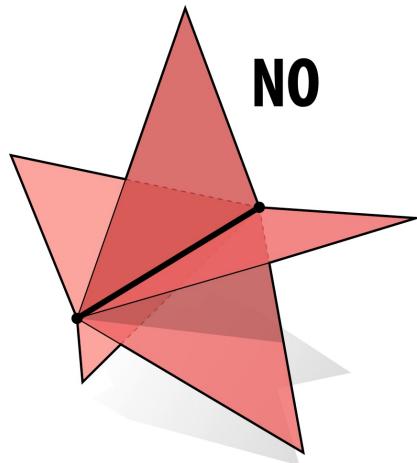
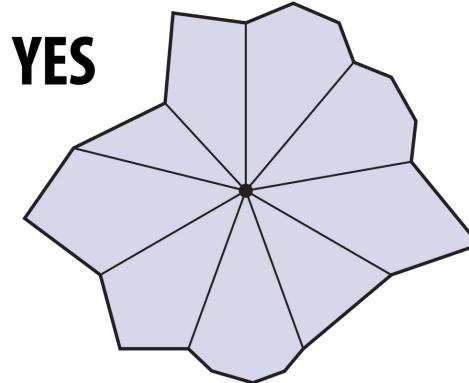
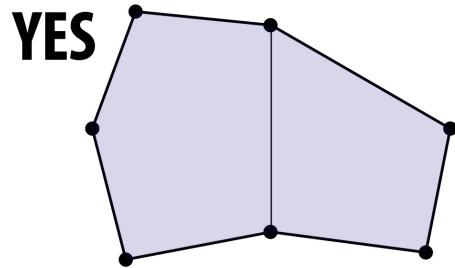




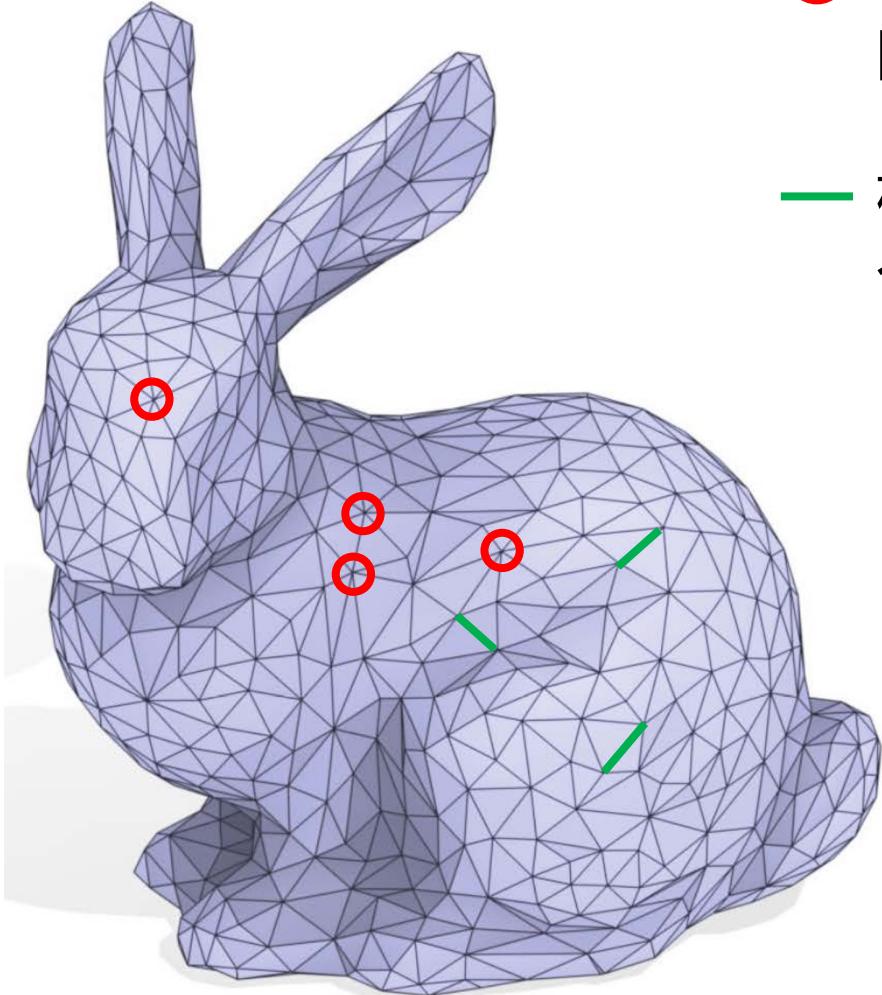
流形多边形网格 (manifold polygon mesh)

判断多边形曲面是否为流形，只需检查两个简单的条件：

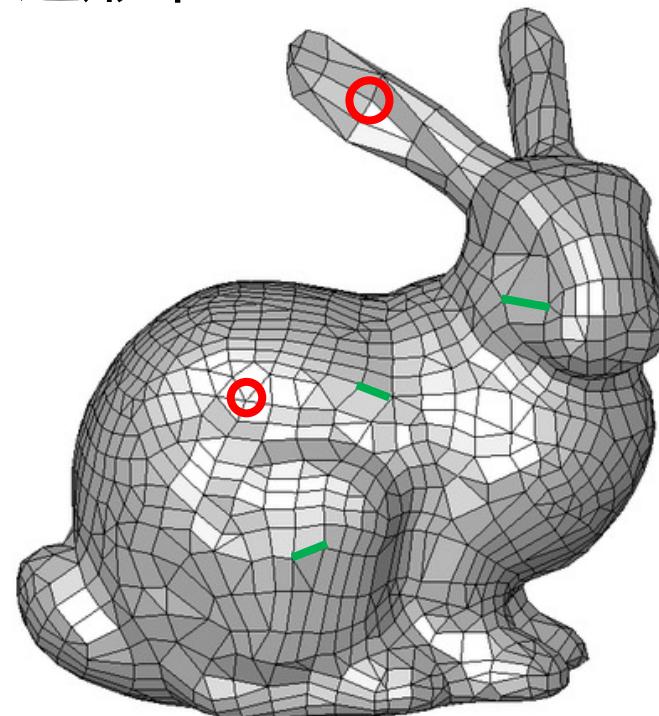
1. 每条“内部”边只包含在两个多边形中
2. 每个“内部”点周围包含一圈循环的多边形



流形多边形网格



- 标识“内部”点，被与其相连的多边形环绕一圈
- 标识“内部”边，仅包含在两个多边形中



其他多边形网格也一样

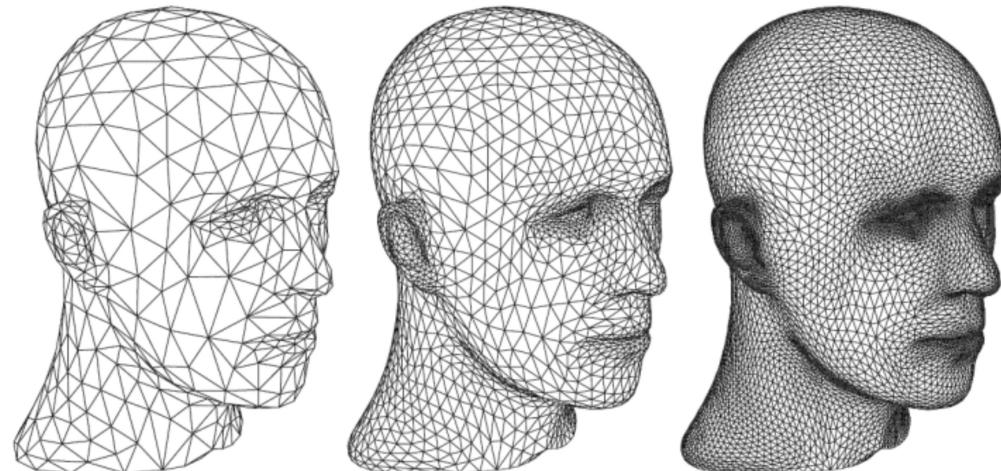
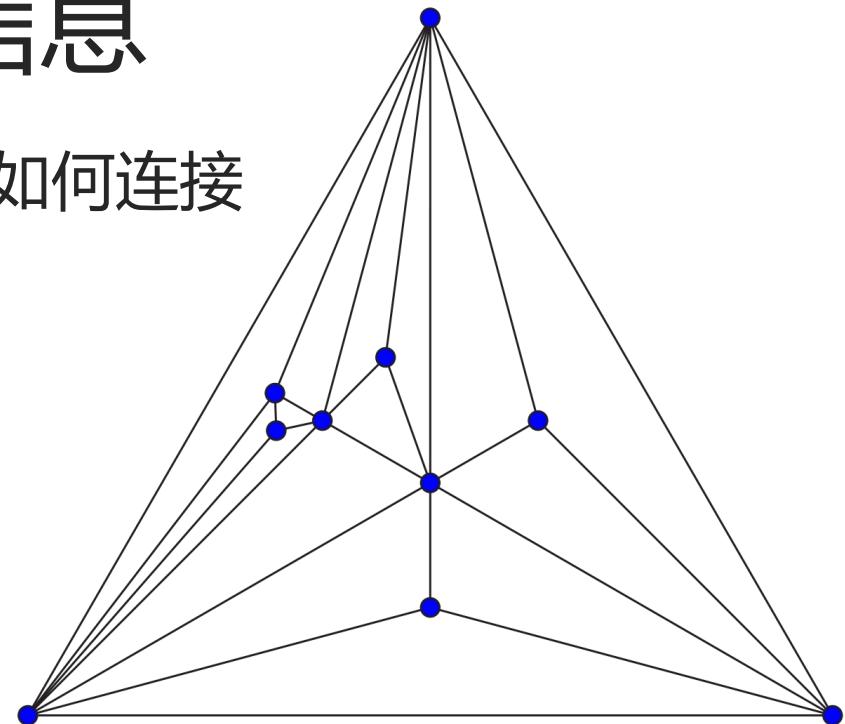
几何的连接性/邻居信息

□ 连接信息是指节点、边、面之间如何连接

- 点和点的连接
- 点和边的连接
- 边和面的连接
- ...

□ 连接信息在几何处理中很重要

- 网格细分
- 表面简化
- 表面规则化
- 纹理映射
- 导航
- 拓扑操作
- ...

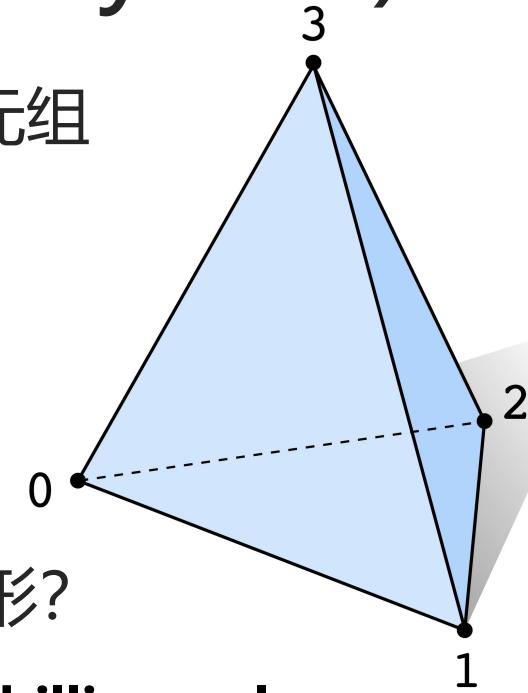


邻接表 Adjacency list (Array-like)

□ 存储坐标的三元组 (x, y, z) 以及索引的三元组

□ 比如四面体：

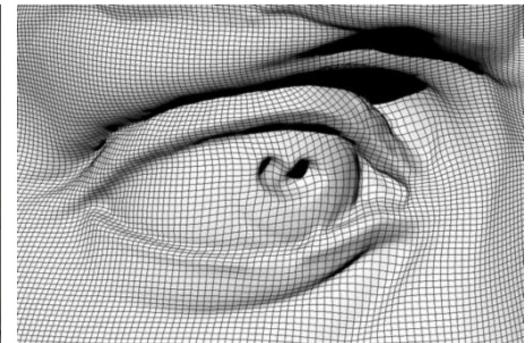
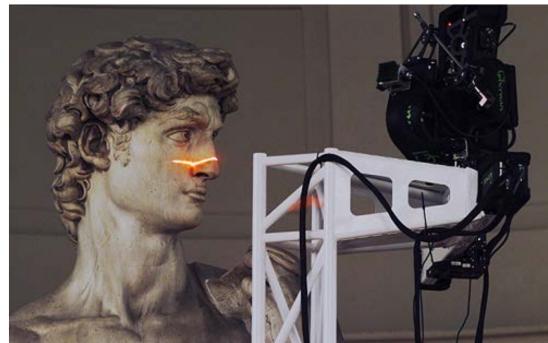
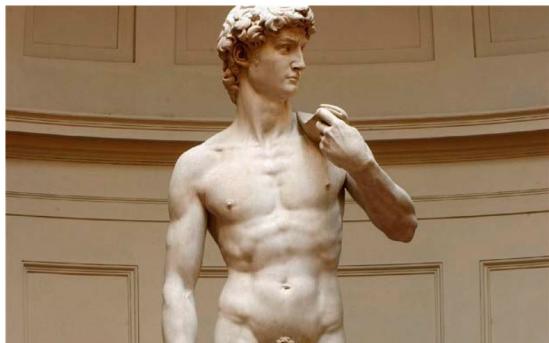
	VERTICES			POLYGONS		
	x	y	z	i	j	k
0:	-1	-1	-1	0	2	1
1:	1	-1	1	0	3	2
2:	1	1	-1	3	0	1
3:	-1	1	1	3	1	2



□ Q：我们如何找到所有接触顶点 2 的多边形？

□ 现在考虑一个更复杂的网格：

~1 billion polygons



□ 查找相邻多边形非常昂贵！

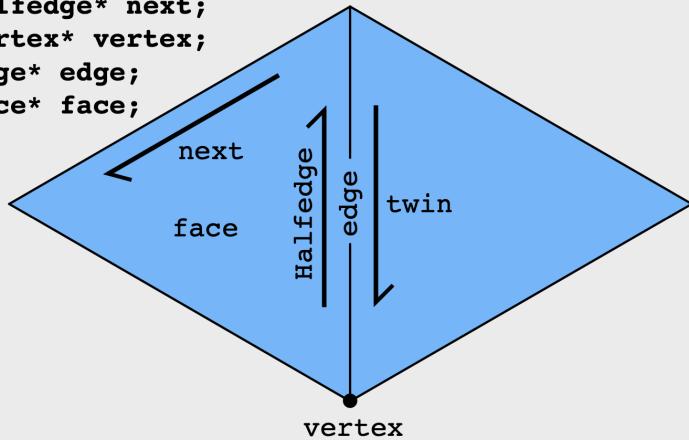
半边 Halfedge 数据结构 (类似链表)

□ 基本理念：存储一些邻居的信息

□ 不需要详尽的清单；只需要几个关键指针 pointer (**就能让我们存储/计算所有网格信息**)

□ 关键思想：两个半边缘充当网格元素之间的“粘合剂”

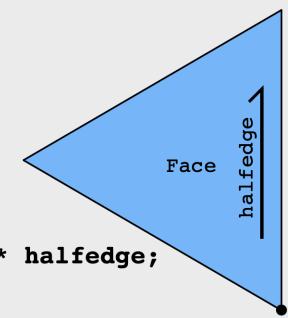
```
struct Halfedge
{
    Halfedge* twin;
    Halfedge* next;
    Vertex* vertex;
    Edge* edge;
    Face* face;
};
```



```
struct Edge
{
    Halfedge* halfedge;
};
```

```
struct Face
{
    Halfedge* halfedge;
};
```

```
struct Vertex
{
    Halfedge* halfedge;
};
```



每个 halfedge 存储很多信息

□ 每个顶点、边和面仅指向它的其中一个半边

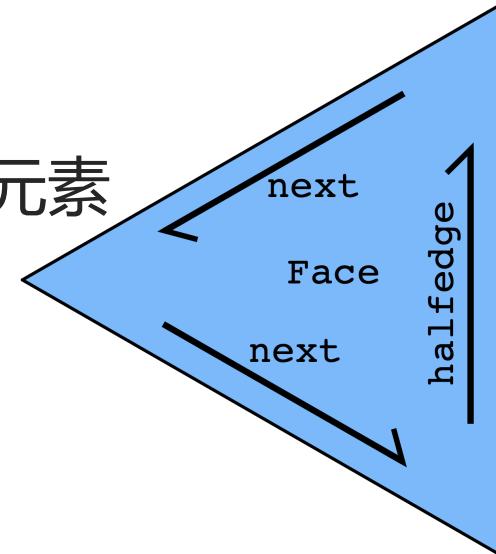
半边简化了网格遍历 mesh traversal

□ 使用 twin 和 next 指针在网格中移动

□ 使用 vertex、edge 和 face 指针来抓取元素

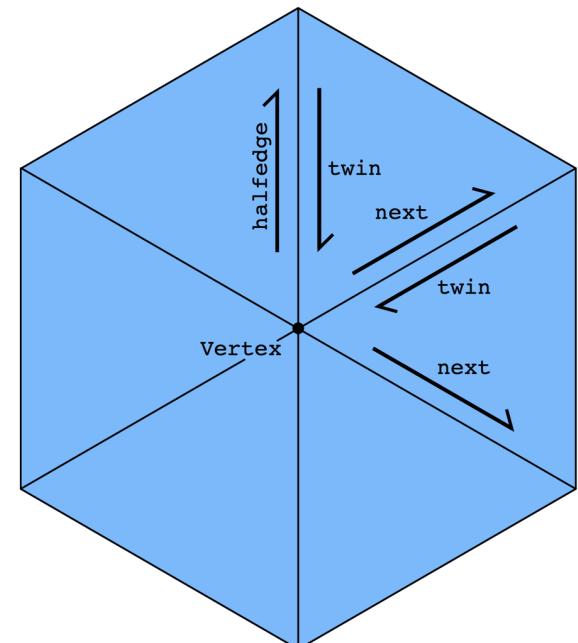
□ **示例1：访问一个面的所有顶点**

```
Halfedge* h = f->halfedge;
do {
    h = h->next;
    // do something w/ h->vertex
}
while( h != f->halfedge );
```



□ **示例2：访问一个顶点接触的所有邻居**

```
Halfedge* h = v->halfedge;
do {
    h = h->twin->next;
}
while( h != v->halfedge );
```



□ **注意：只有当网格是流形时才有意义！**

半边连通性总是流形的

□ 考虑一个简化的半边缘数据结构

□ 仅需要以下条件，就能得到一个流形多边形网格

```
struct Halfedge {  
    Halfedge *next, *twin;  
};
```

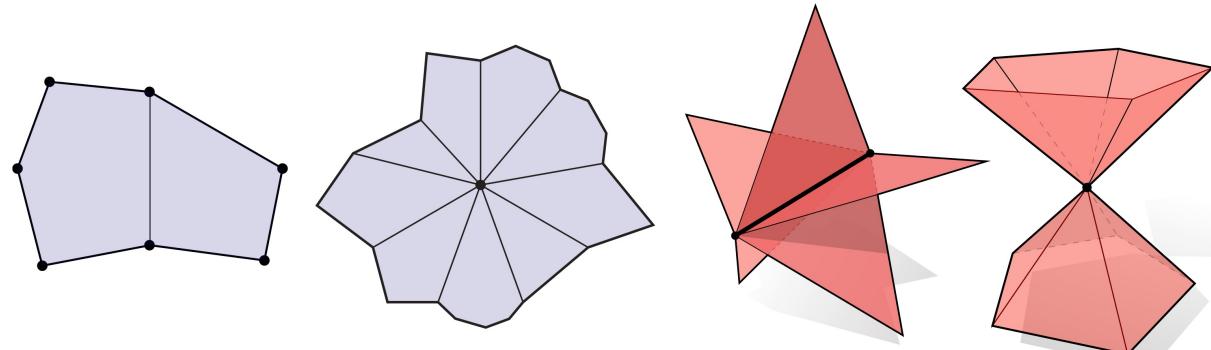
twin->twin == this
twin != this
every he is someone's "next"

指向自己的
的指针

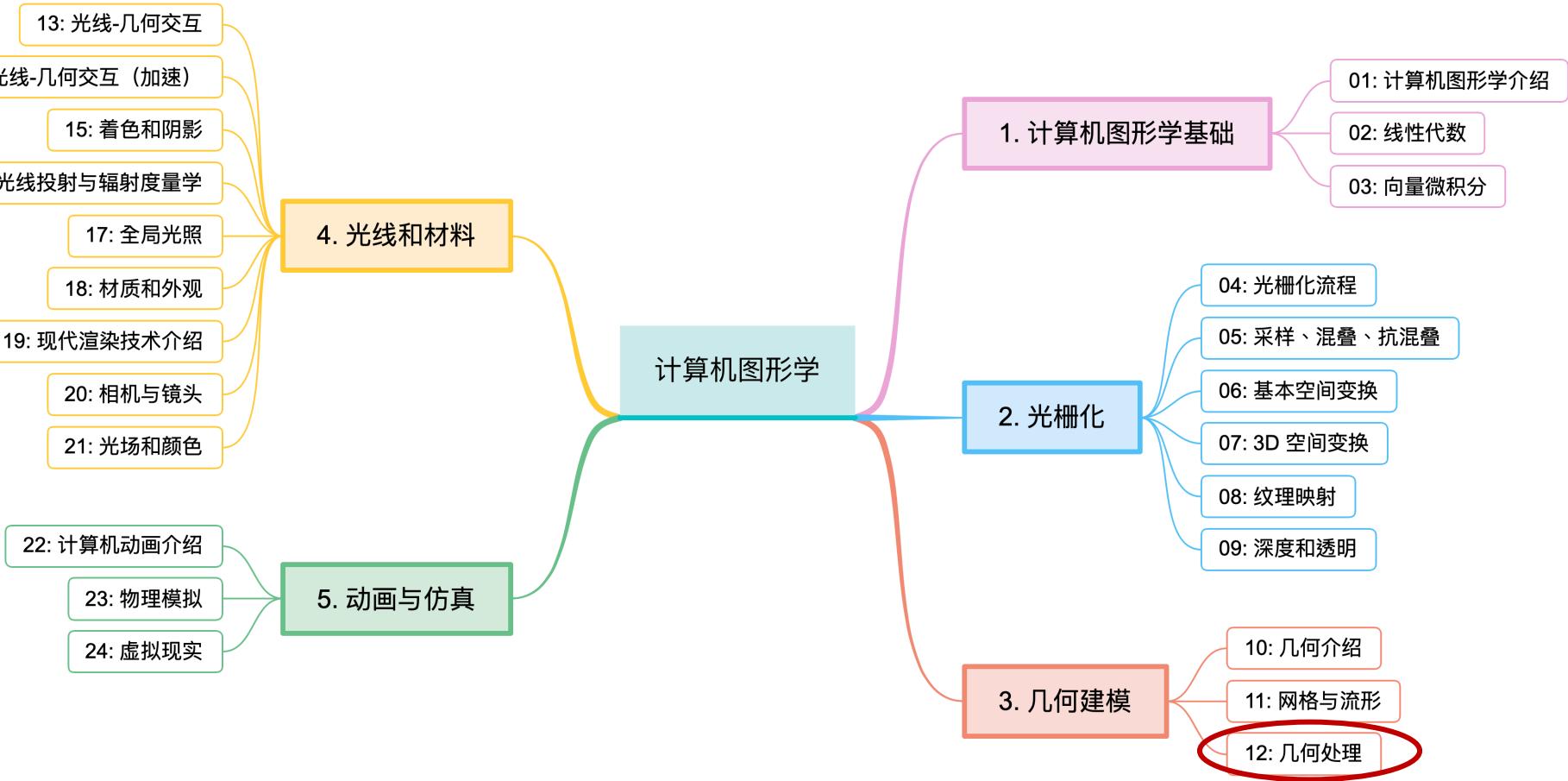
□ Keep following next, and you'll get faces

□ Keep following twin and you'll get edges

□ Keep following next->twin and you'll get vertices



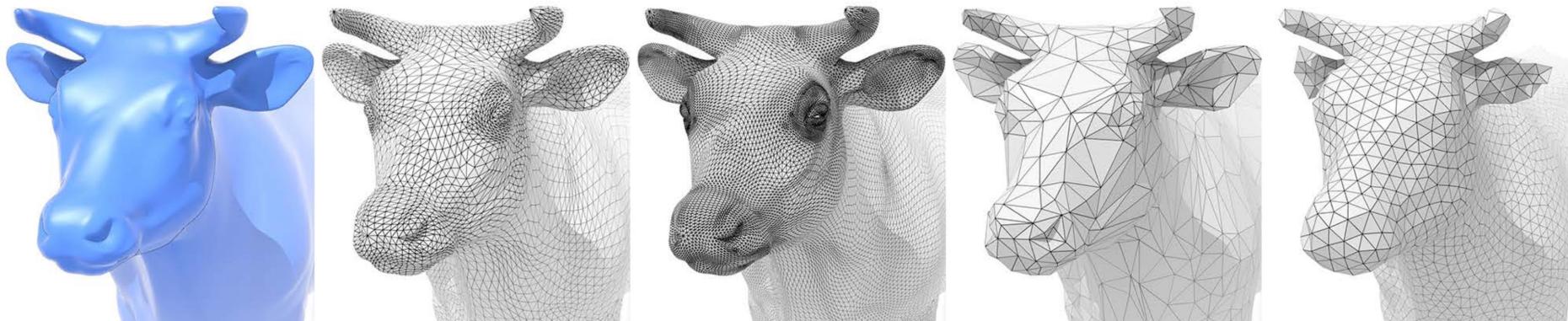
□ 因此，从半边的构造来看，不能编码红色的非流形几何图形



几何网格处理

扩展传统的数字信号处理 (音频/视频等) 以处理几何信号:

- 上采样 / 下采样 / 重采样 / 滤波 ...
- 走样 (重建的表面给出 “错误印象”)



原始网格

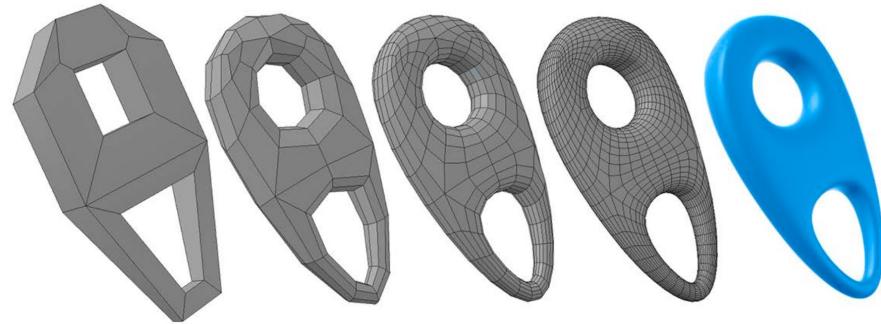
网格细分

网格简化

网格规则化

如何对网格进行上采样？

通过细分进行上采样



□不断地将每个元素分割成更小的部分

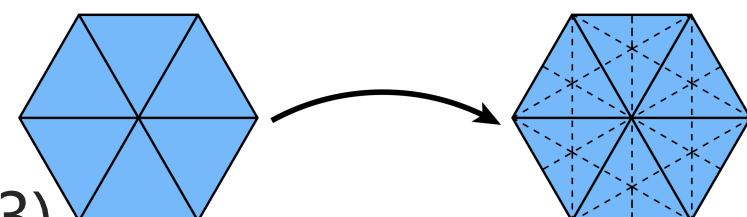
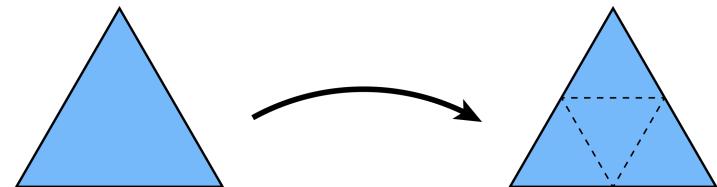
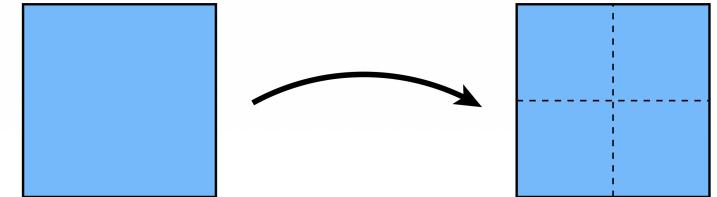
□将顶点位置替换为邻居的加权平均值

□主要的考虑因素：

- 插值与近似
- 限制曲面的连续性 (C^1, C^2, \dots)
- 在不规则顶点处的表现

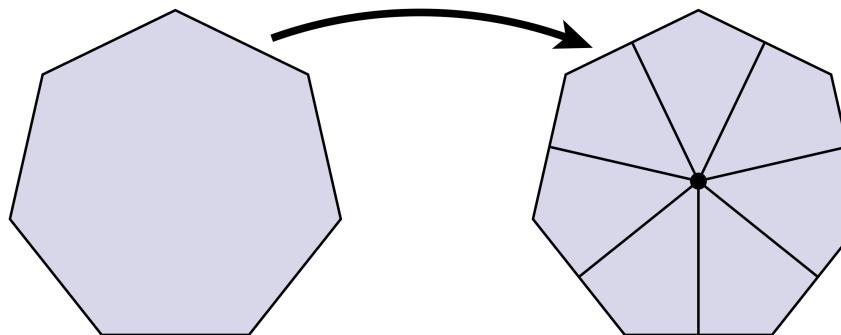
□有许多选择：

- 四边形：Catmull-Clark
- 三角形：Loop, Butterfly, Sqrt(3)

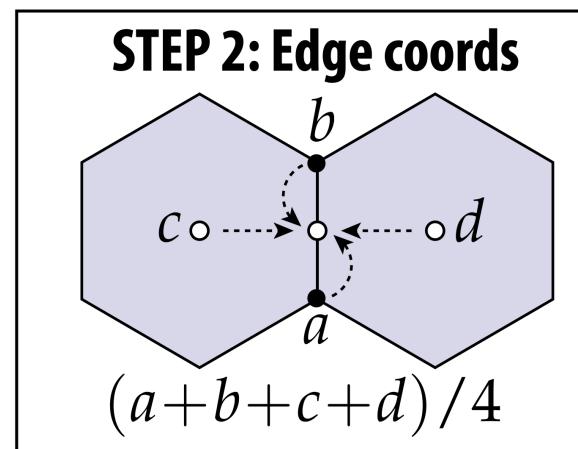
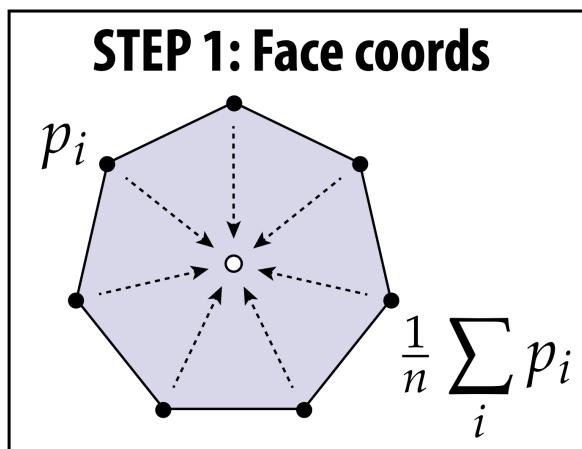


Catmull-Clark 细分 (任意网格)

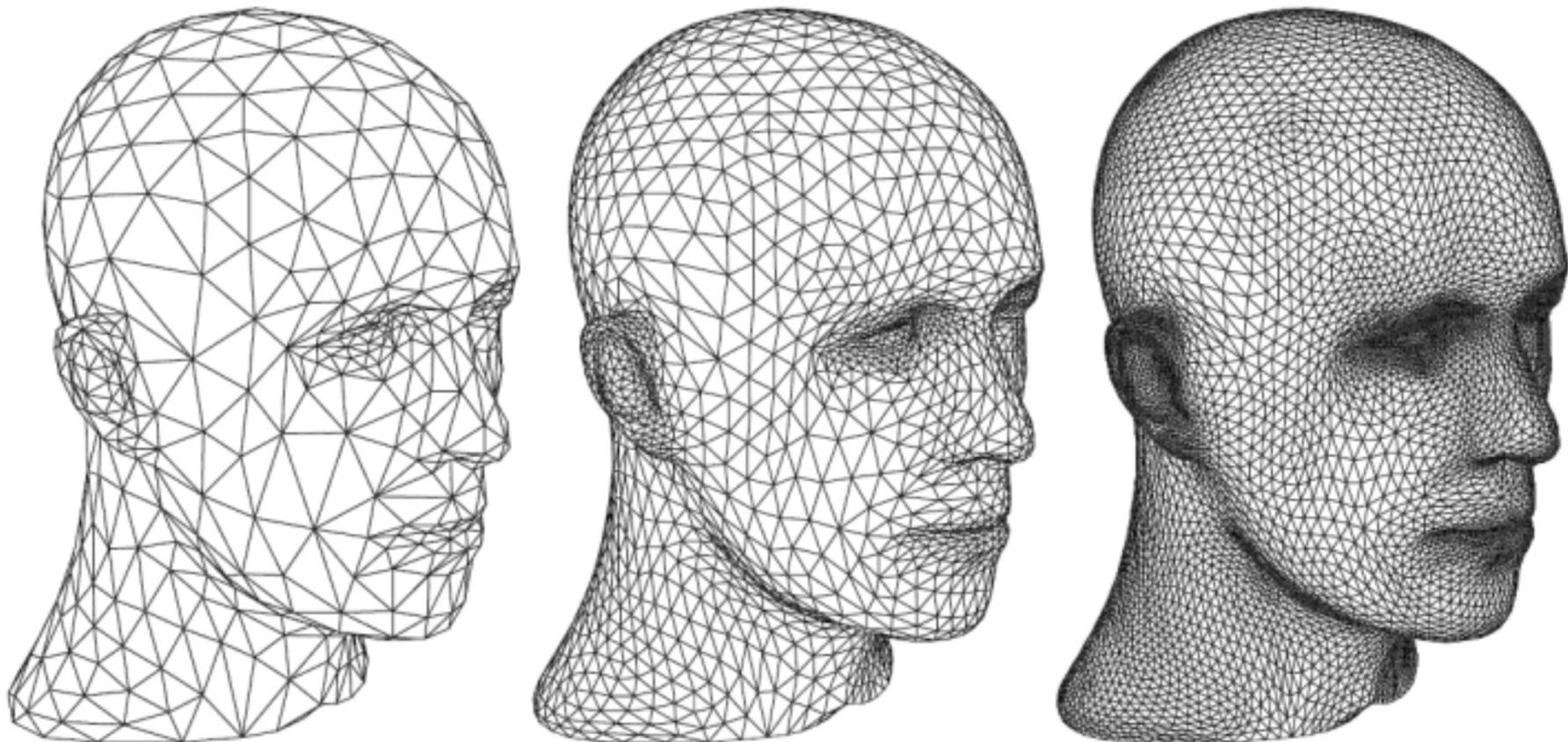
□ Step 0: 将每个多边形 (任意数量的边) 分成四边形



□ 新顶点位置是旧顶点位置的加权组合



Loop 细分 – 结果



如果我们想要更少的三角形
呢？

网格简化 (Mesh Simplification)

□ 目标：减少多边形的数量

□ 同时尽量保持整体的形状和外观



30,000 triangles



3,000



300



30



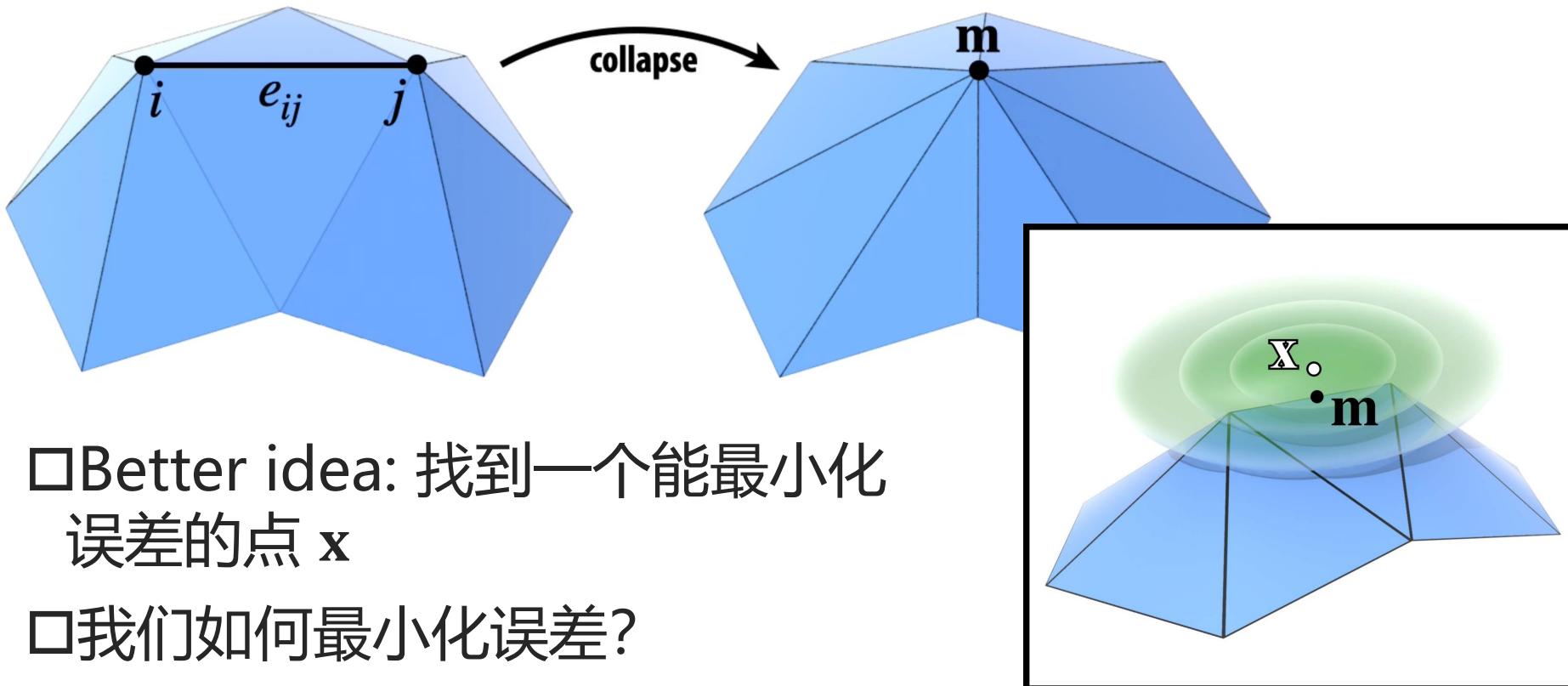
怎么进行计算？

边折叠的二次误差

□ 折叠一条边 e_{ij} 的代价是多少？

□ Idea: 计算中点 \mathbf{m} , 测量误差 $Q(\mathbf{m}) = \mathbf{m}^T(K_i + K_j)\mathbf{m}$

□ 误差作为边 e_{ij} 的“分数”，决定其优先级



基于二次误差的网格简化

□ 为每个三角形计算 K (到平面的平方距离)

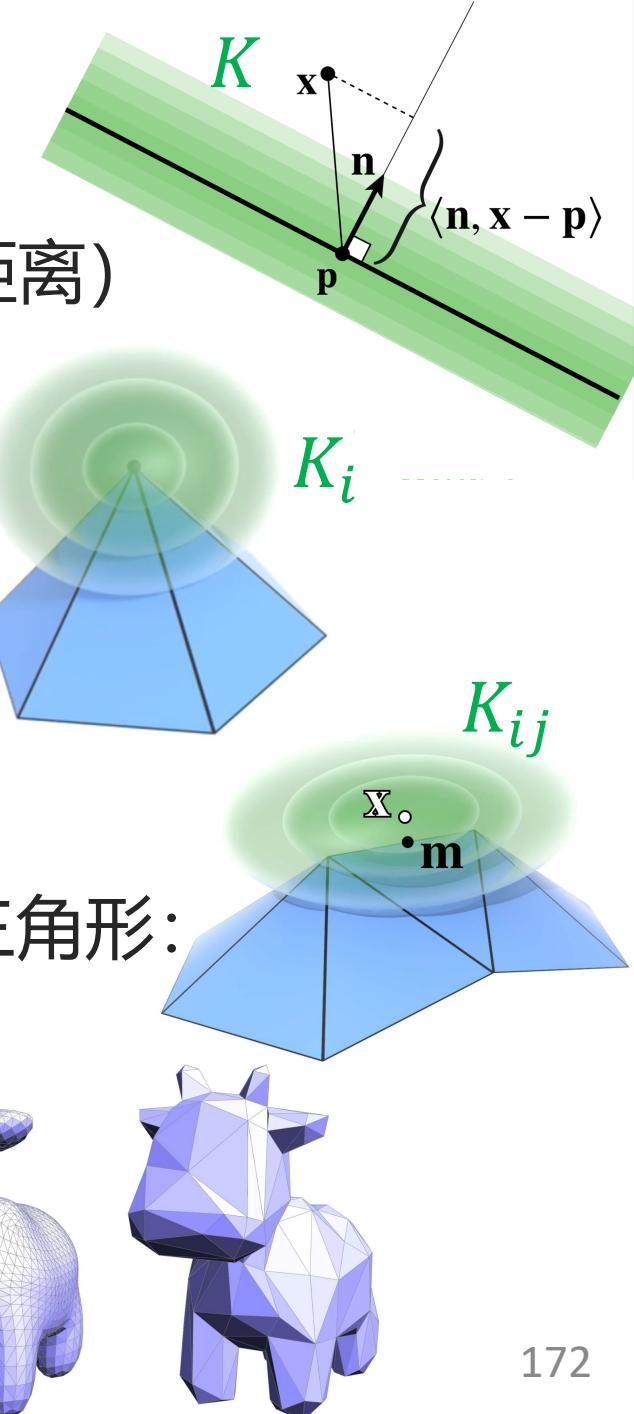
□ 在每个顶点处，将 K_i 设为相邻三角形的 K 之和

□ 对于每一条边 e_{ij} ：

- 设置 $K_{ij} = K_i + K_j$
- 找到最小化误差的点 x ，将误差设置为 $K_{ij}(x)$

□ 重复下列步骤，直到达到目标数量的三角形：

- 将边 e_{ij} 折叠到最优点 (即具有最小代价的点)
- 在新顶点处设置二次曲面为 K_{ij}
- 更新接触新顶点的边的误差

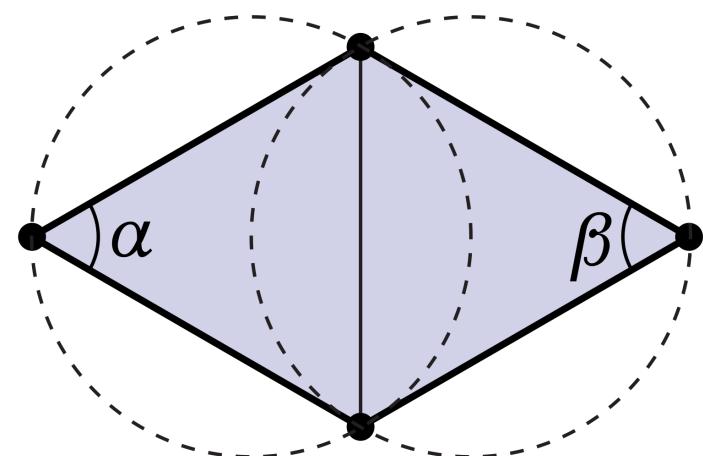
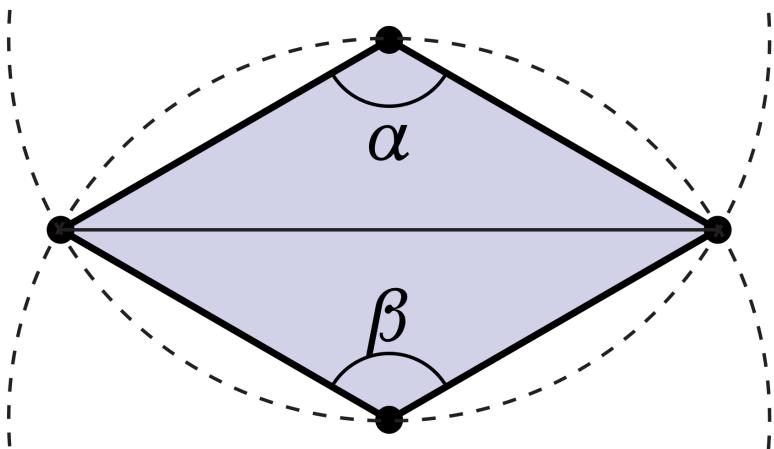


如果我们满意三角形的数量，
但想提高网格质量，该怎么办？

我们如何使一个网格"更 Delaunay"?

□ 我们已经有一个好工具：翻转边缘！

□ 如果 $\alpha + \beta > \pi$, 则反转



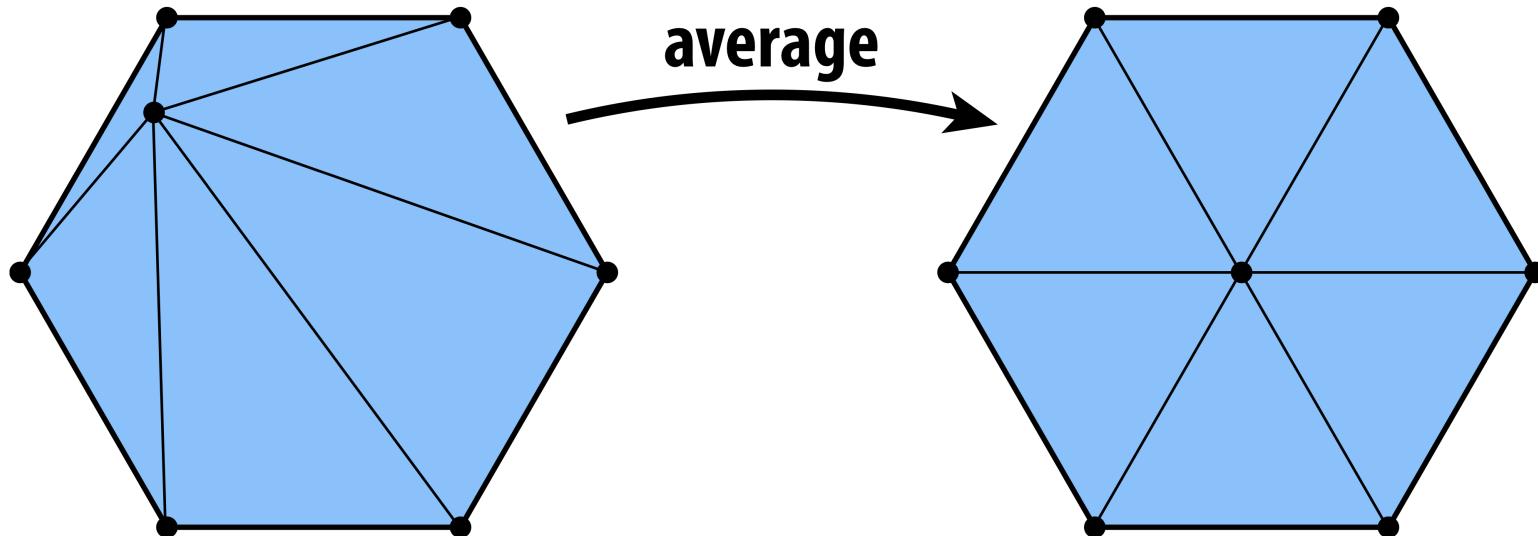
□ 事实：在2D中，翻转边缘最终会产生 Delaunay 网格

□ 理论：最坏情况 $O(n^2)$ ；对于 3D 表面并不总是有效

□ 实践：简单，有效的方法来提高网格质量

我们如何使三角形"更圆润"?

- Delaunay 不能保证三角形是"圆形"（角度接近60°）
- 通过将顶点居中，我们通常可以改善形状：

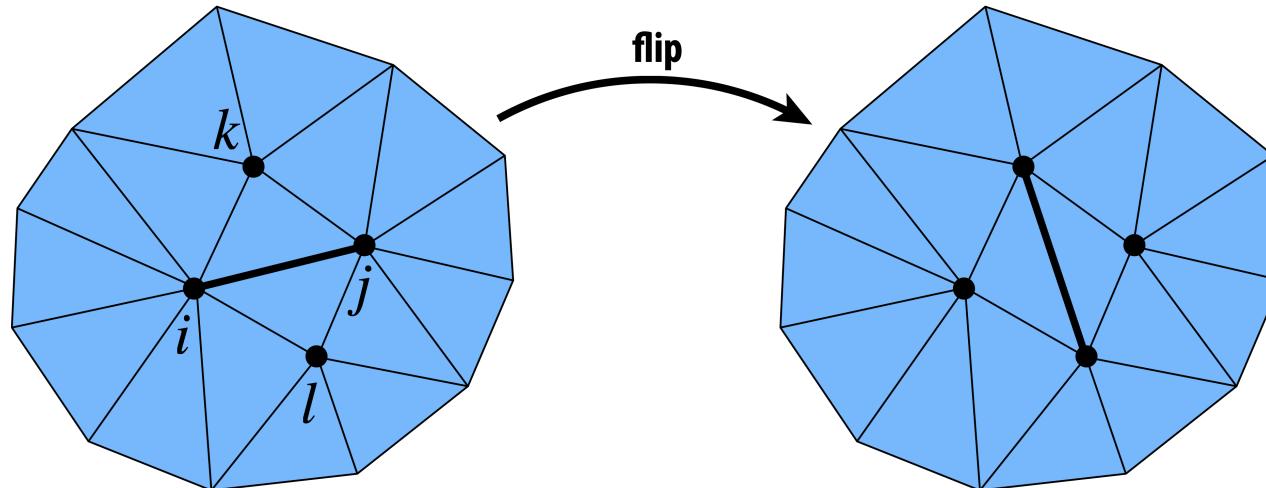


- 这是一种称为" Laplacian 平滑"的简单技术
- 在表面上：只在切线方向移动以居中顶点
- 如何做到？从“移动向量”中减去法向分量

或者，我们如何提高节点的度？

口同样的工具：翻转边缘！

口如果总的偏离度（三角形最优为 6）变小，就翻转它！

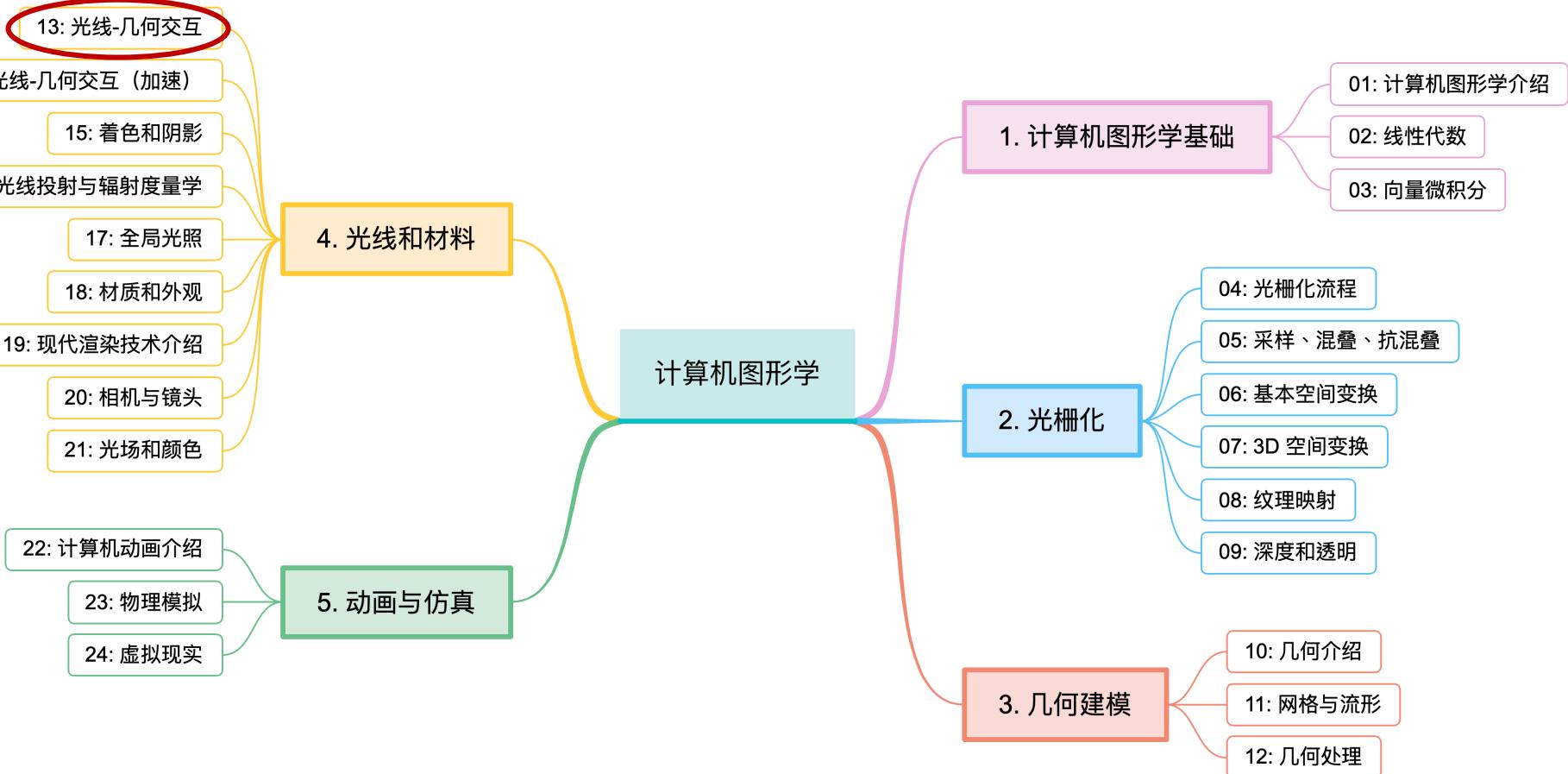


$$\text{总偏离度: } |d_i - 6| + |d_j - 6| + |d_k - 6| + |d_l - 6|$$

口事实：随着元素数量的增加，平均度数趋近于 6

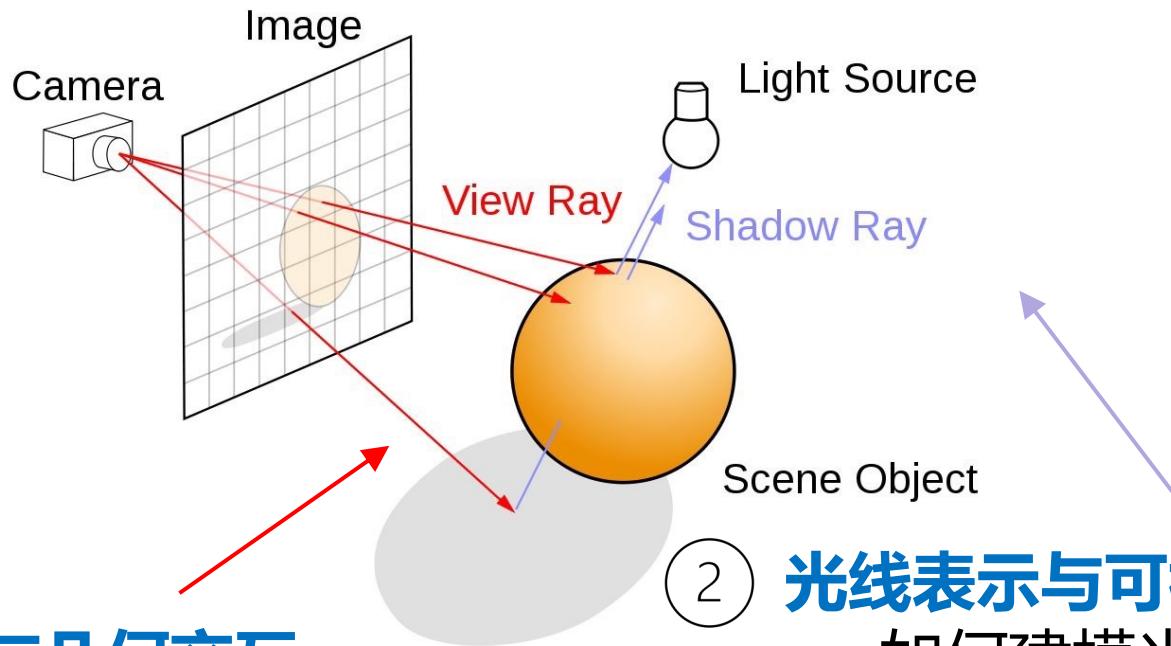
口迭代翻转边缘能把密集区域的度扩散到其他区域

口没有（已知的）理论保证；在实践中效果良好



光线追踪 Ray tracing

分成 **光线与几何交互** 和 **光线表示与可视化** 两部分



- ① **光线与几何交互**
- 查找光线照射在几何中的位置
 - 如何加速上述过程
- 看到哪里**

②

光线表示与可视化

- 如何建模光线
 - 光线与物体的相互作用，包括反射、折射和散射等
 - 全局光照
- 看到什么**

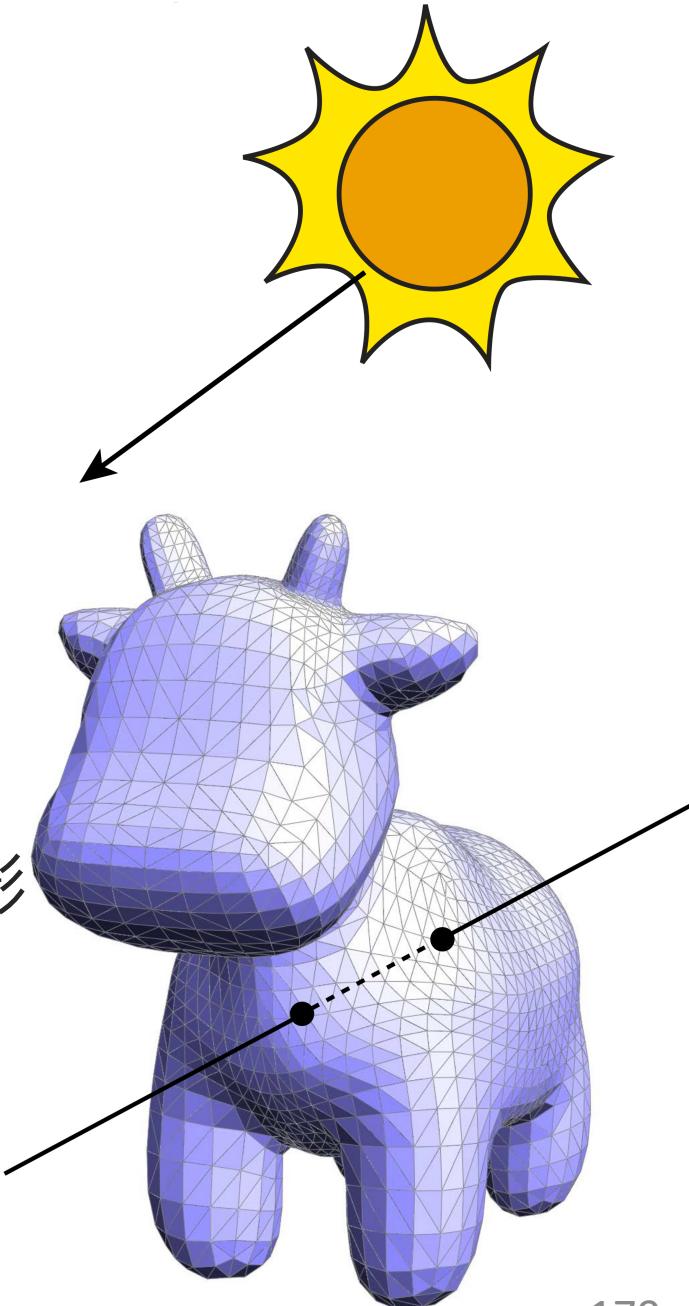
光线与网格交叉

- “光线” 是从一点开始的定向直线
- 想象一束来自太阳的光
- 我们想知道光线穿透表面的位置

□ 有什么作用？

- 几何 Geometry：内外测试
- 渲染 Rendering：可见性，生成阴影
- 动画 Animation：碰撞检测

□ 可能会在许多地方穿透表面！



光线方程 Ray equation

光线可用如下方程表示

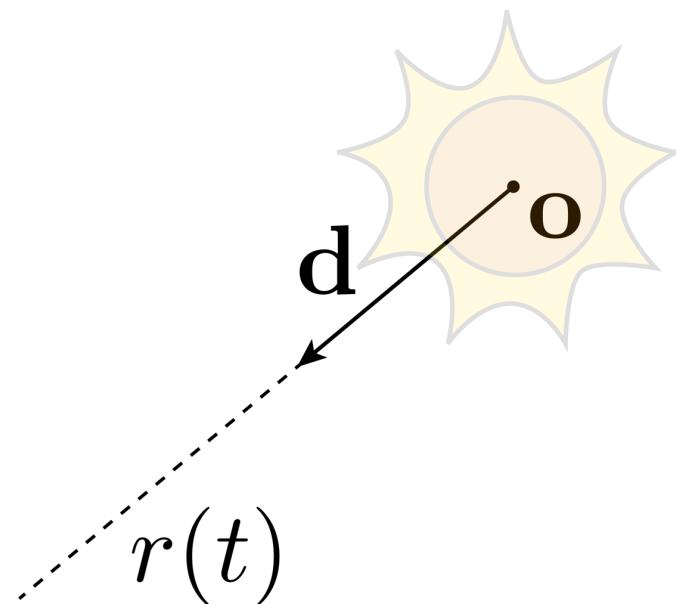
$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

沿着光线在 t 时间上的点
point along ray at t

时间
time

光线源点
origin

单位方向
unit direction



光线与平面相交

□ 假设我们有平面 $\mathbf{N}^T \mathbf{x} = c$

- \mathbf{N} 为单位法线
- c 为偏移量 (offset)

□ 我们如何找到平面与光线的交点?

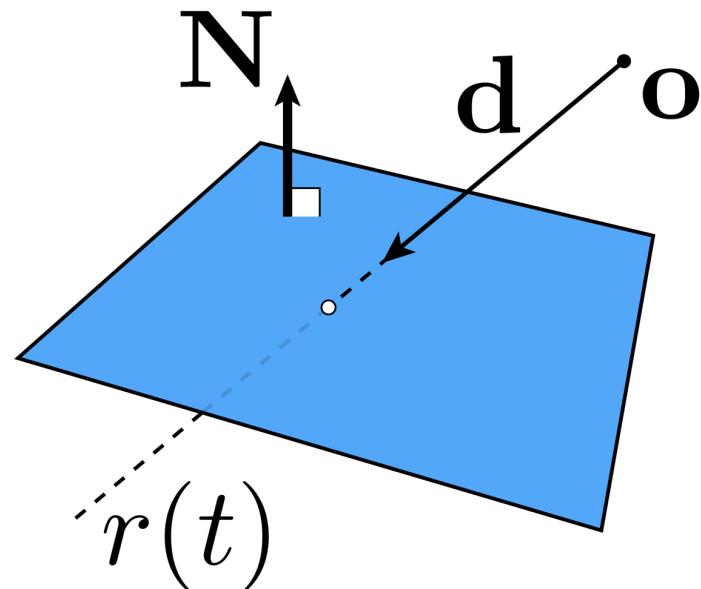
□ 算法: 同样地, 把点 x 替换为光线方程

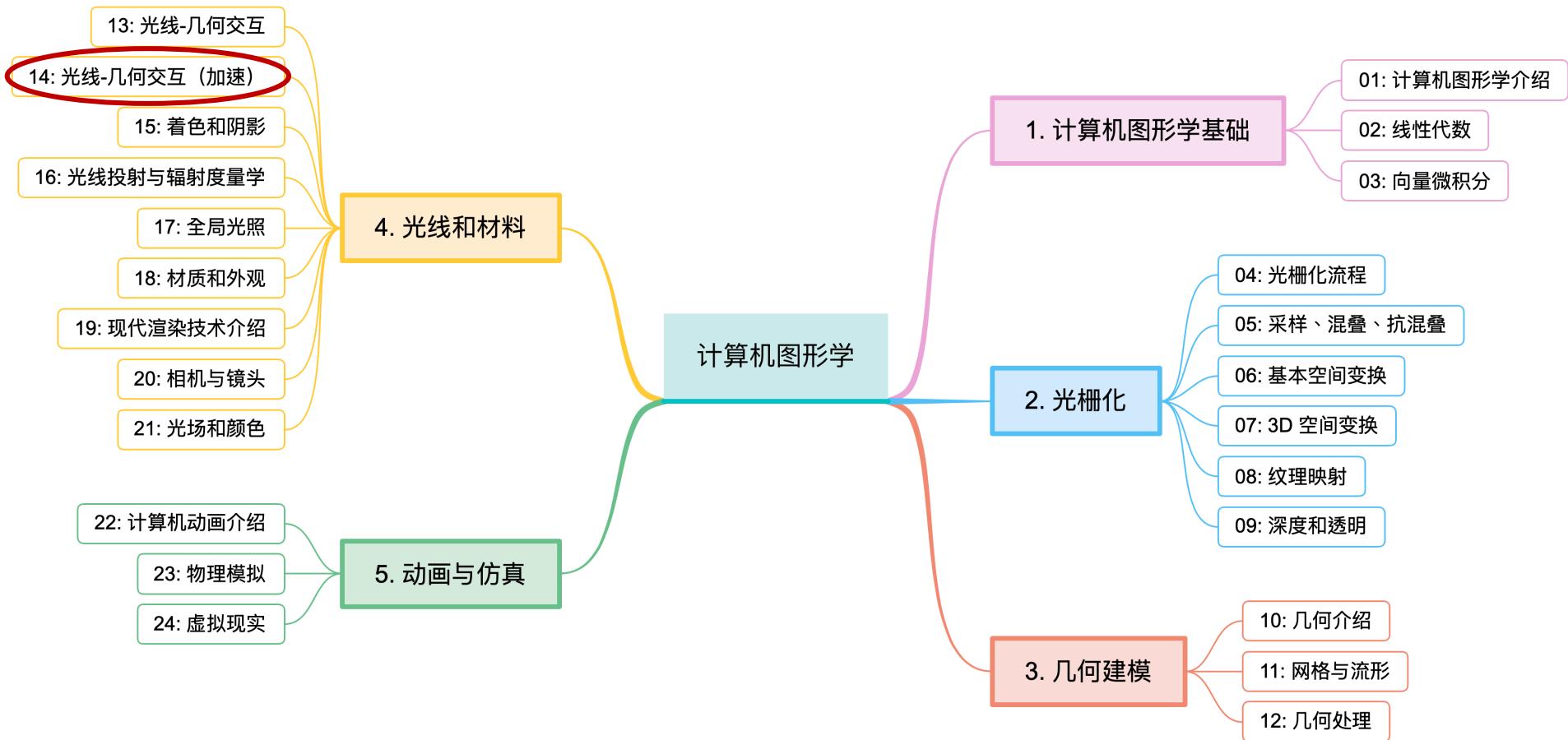
$$\mathbf{N}^T \mathbf{r}(t) = c$$

□ 求解 t :

$$\mathbf{N}^T (\mathbf{o} + t\mathbf{d}) = c \quad \Rightarrow t = \frac{c - \mathbf{N}^T \mathbf{o}}{\mathbf{N}^T \mathbf{d}}$$

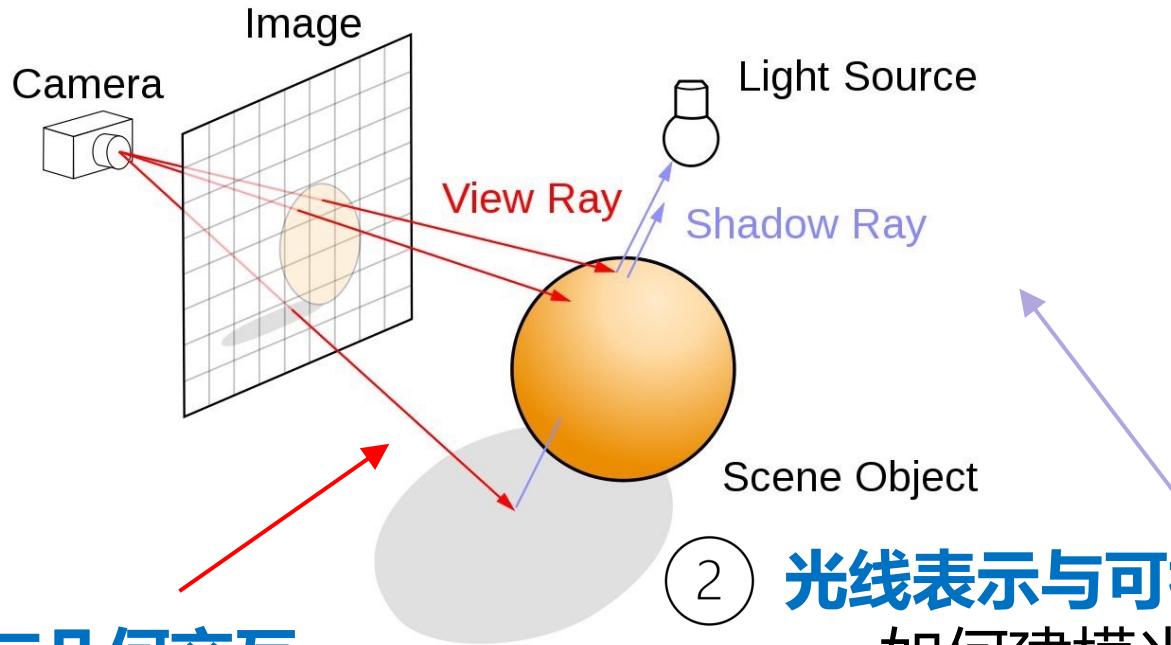
□ 将 t 代入光线方程中 $\mathbf{r}(t) = \mathbf{o} + \frac{c - \mathbf{N}^T \mathbf{o}}{\mathbf{N}^T \mathbf{d}} \mathbf{d}$





光线追踪 Ray tracing

分成 **光线与几何交互** 和 **光线表示与可视化** 两部分



① 光线与几何交互

- 查找光线照射在几何中的位置
- 如何加速上述过程

看到哪里

② 光线表示与可视化

- 如何建模光线
- 光线与物体的相互作用，包括反射、折射和散射等
- 全局光照

看到什么

包围盒

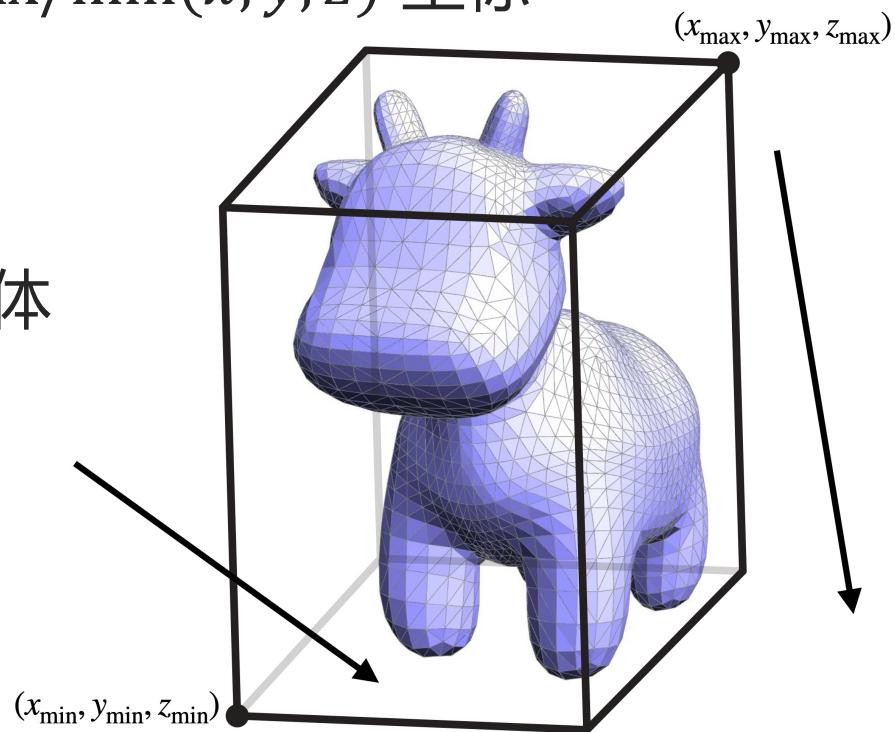
口一种快速(初步)判断交叉的方法：用最小的立方体包围复杂的几何物体(包含其所有图像基元)

- Q: 怎么构建包围盒?
- A: 循环所有的点, 记录 $\max/\min(x, y, z)$ 坐标
- Complexity: $O(N)$

口检查光线是否与包围盒相交

- 若没有, 则一定没有击中物体
- 若有, 检查所有三角形找到相交的点(也可能都不相交)

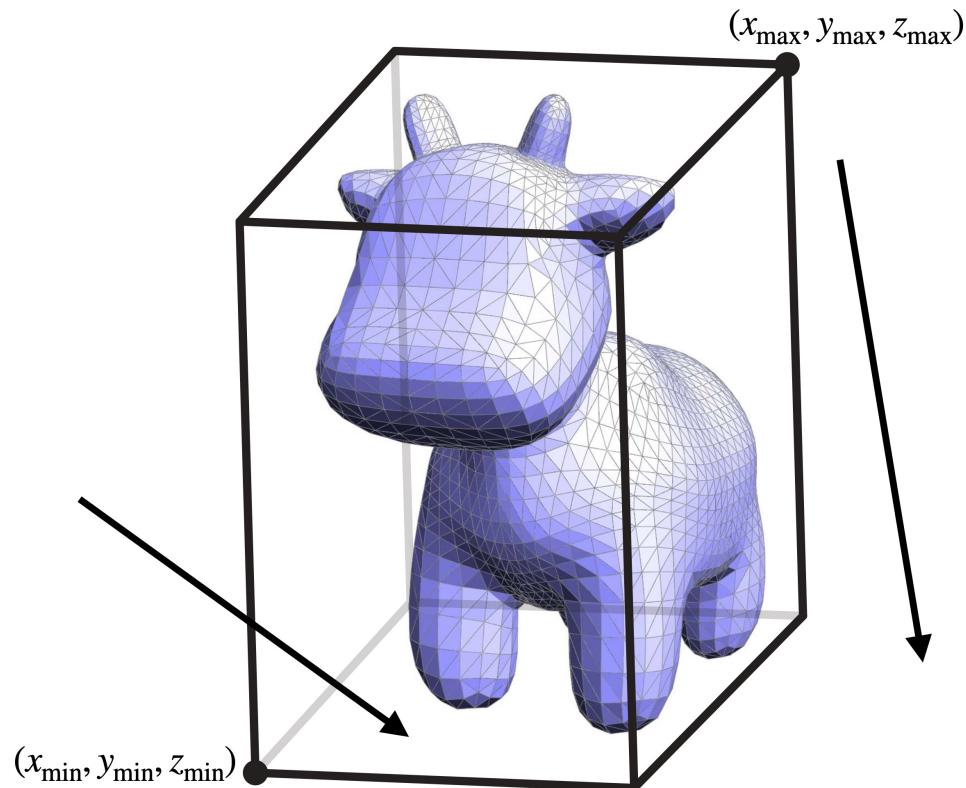
避免每一条光线都要检查所有图像基元!



包围盒性能

□ 我们是否做的更好?

- No, 算法最差情况还是 $O(N)$

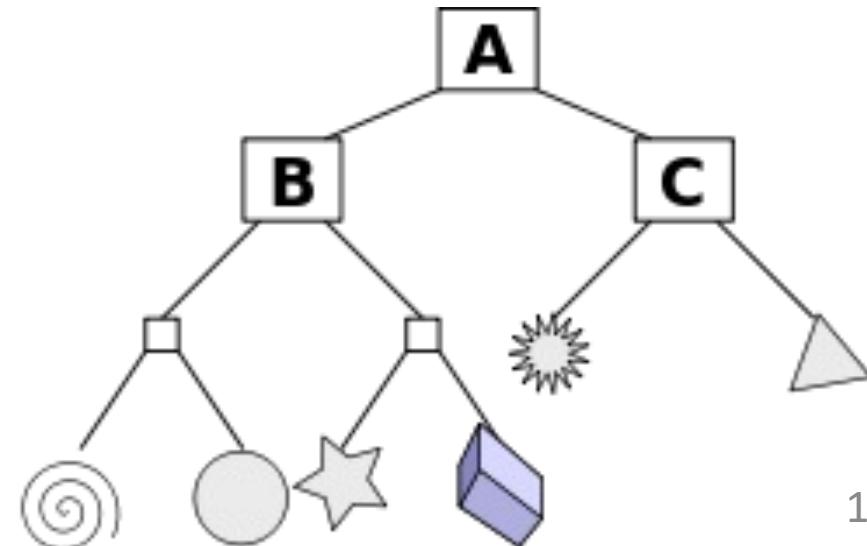
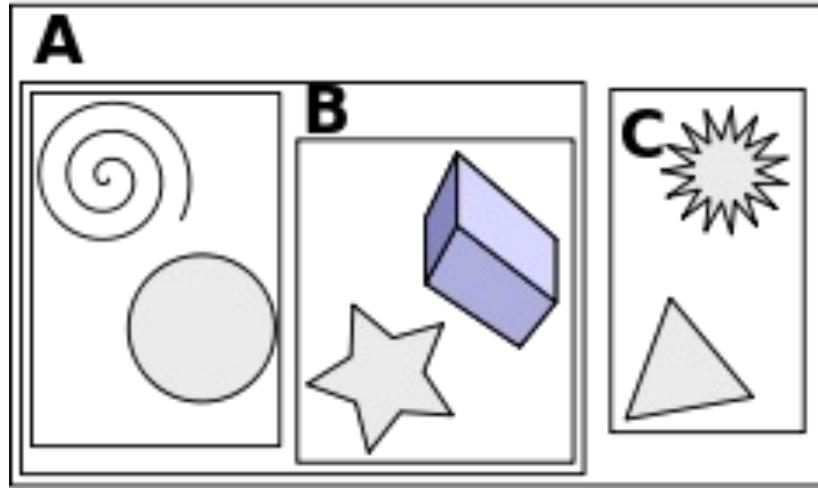


层次包围盒

□一种树形结构，对空间中的几何对象进行分层划分

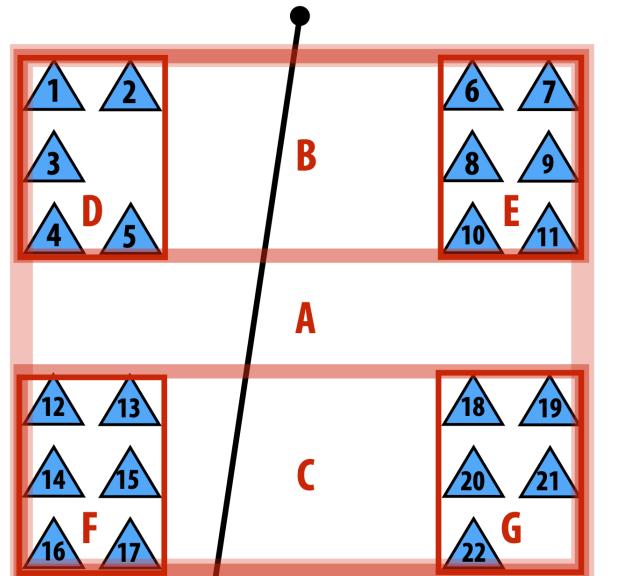
□查询时

- 从树的顶部开始，逐级检查光线是否与节点的包围盒相交
- 若不是，则跳过包围盒内的所有几何对象
- 大大减少需要进行**精确交叉检查**的几何对象数量

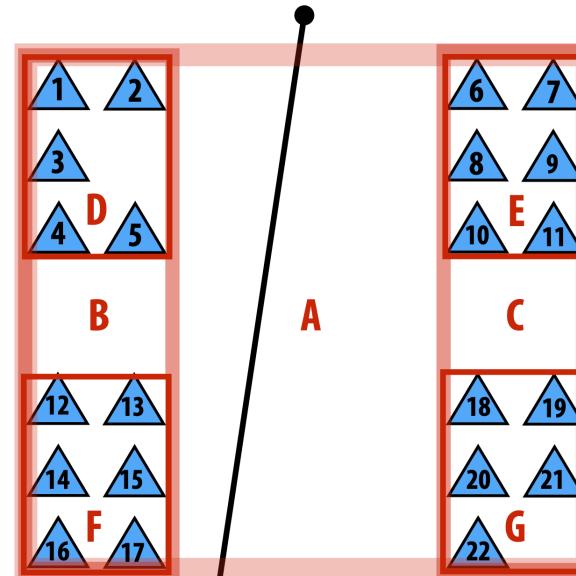


层次包围盒例子

- 两个不同结构的层次包围盒，包含相同的 22 个三角形
- 如何构造层次包围盒？



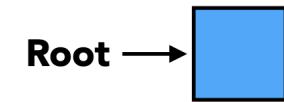
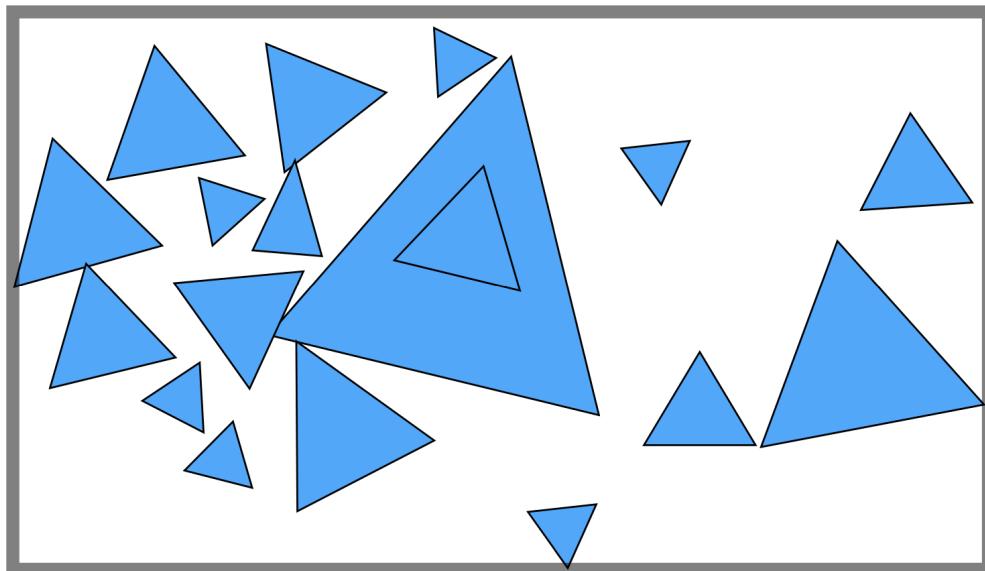
A
B
C
D
E
F
G
1,2,3
6,7,8,
9,10,11
12,13,14,
15,16,17
18,19,20,
21,22
4,5



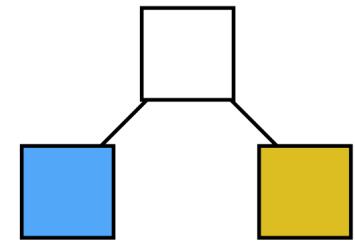
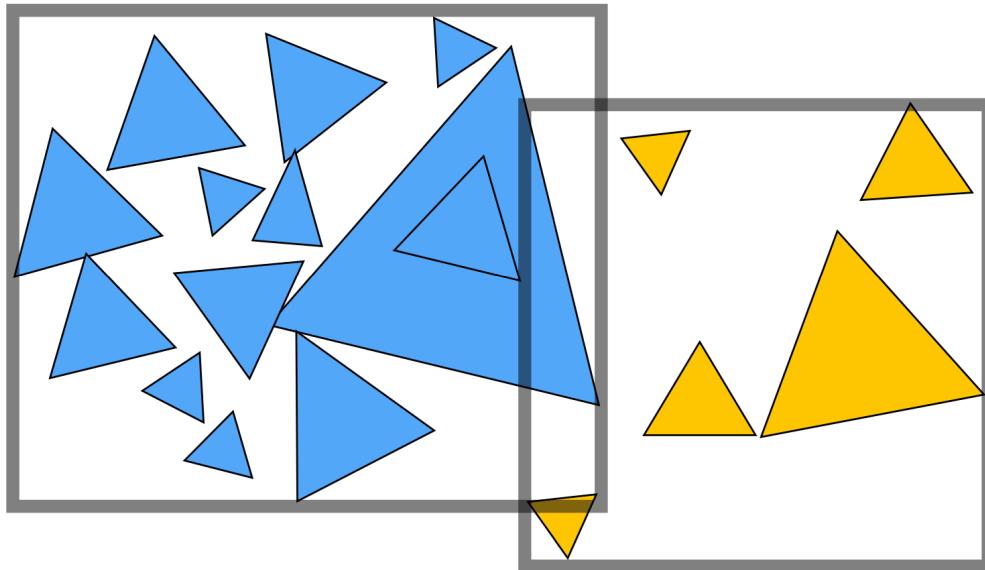
A
B
C
D
F
E
G
1,2,3
6,7,8,
9,10,11
12,13,14,
15,16,17
18,19,20,
21,22
4,5

哪一个更好？

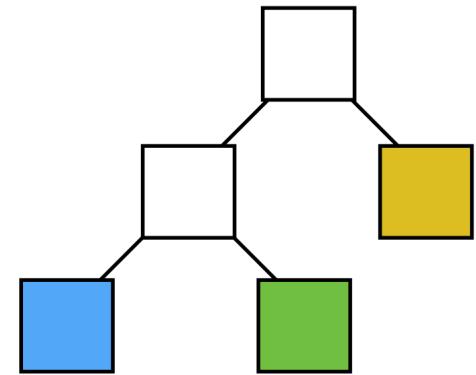
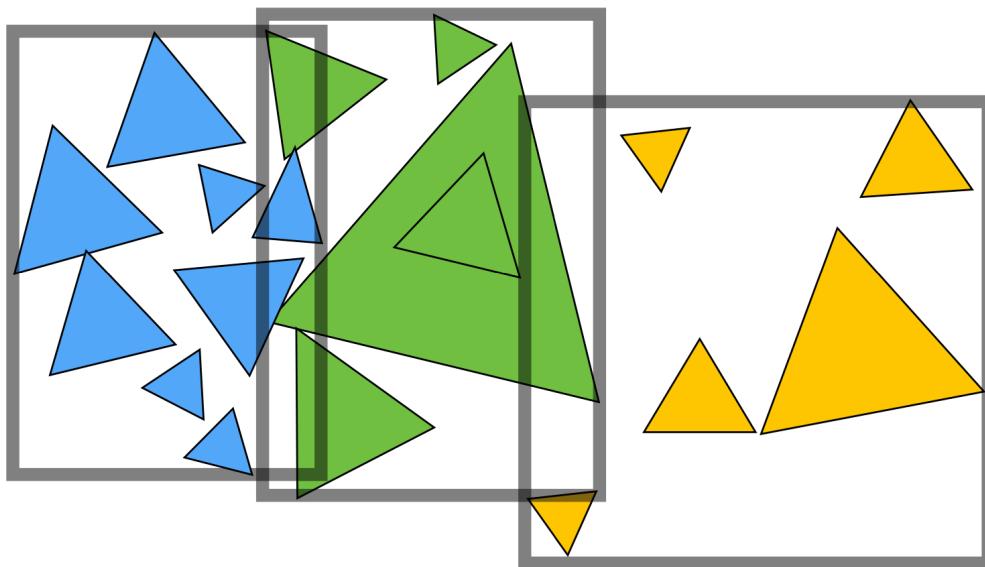
层次包围盒的构建



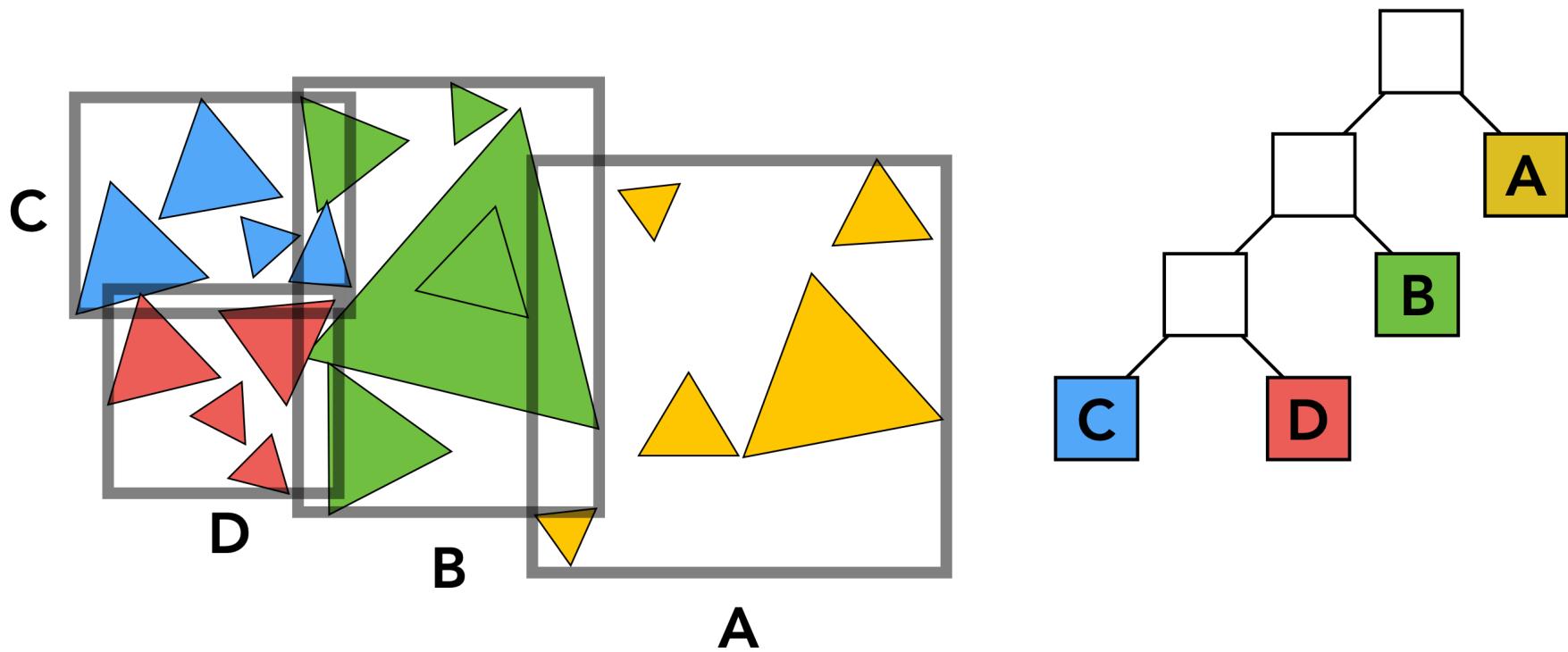
层次包围盒的构建



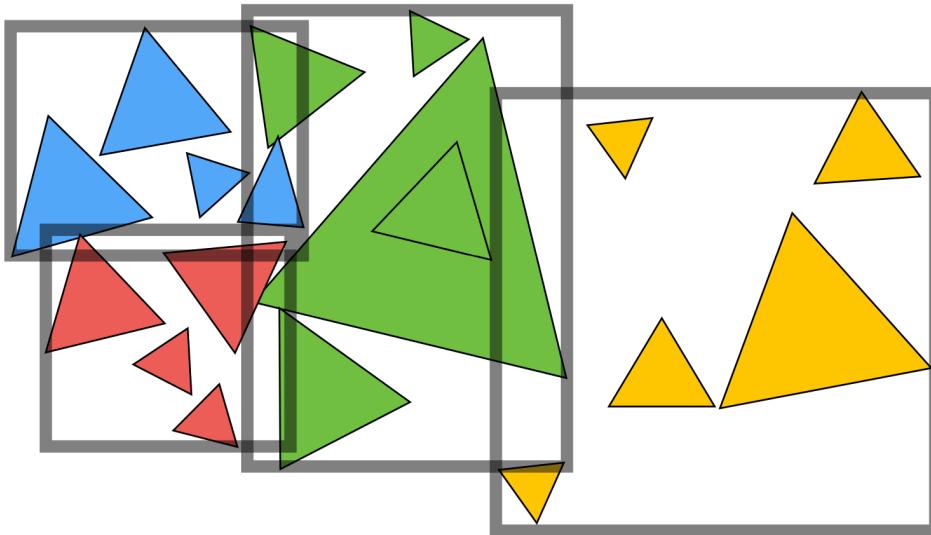
层次包围盒的构建



层次包围盒的构建



层次包围盒的构建



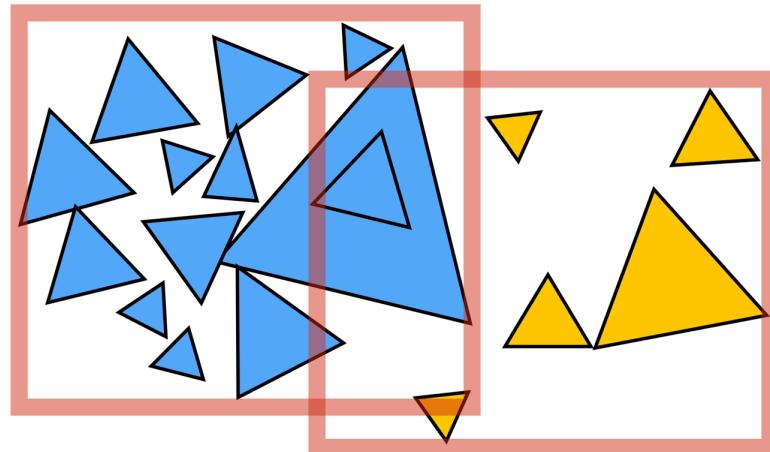
1. 找到包围盒
2. 迭代地把图像基元分成两个子集
3. 重新计算子集的包围盒
4. 在合适的时候停止
5. 字节点存储相应的图像基元列表

不同节点的包围盒可能会重叠，会导致什么问题？

其他图形基元划分方法

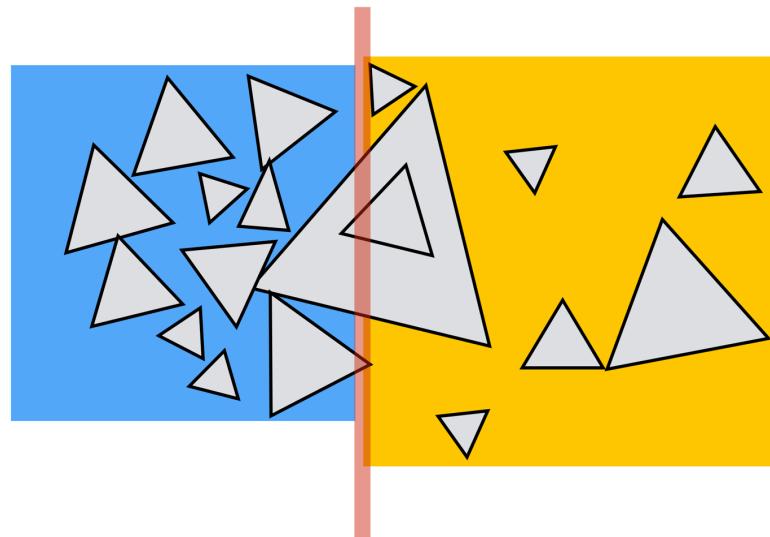
口基于图像基元的划分：

- 将节点的基元划分为不相交的包围盒 (包围盒在空间上可能重叠)
- 比如层次包围盒 (BVH)

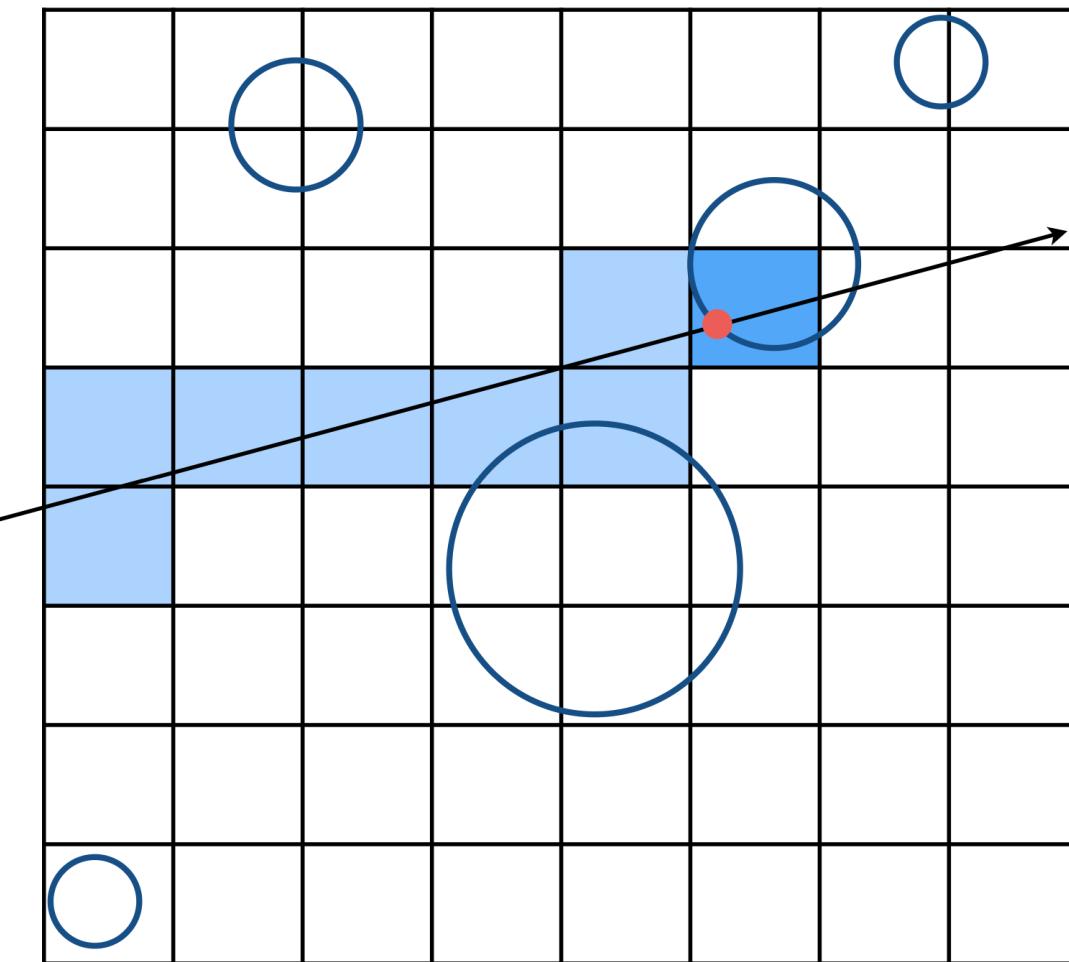


口基于空间的划分：

- 将空间划分为不相交的区域 (同一个基元可以包含在空间的多个区域中)
- 比如均匀空间划分, KD 树



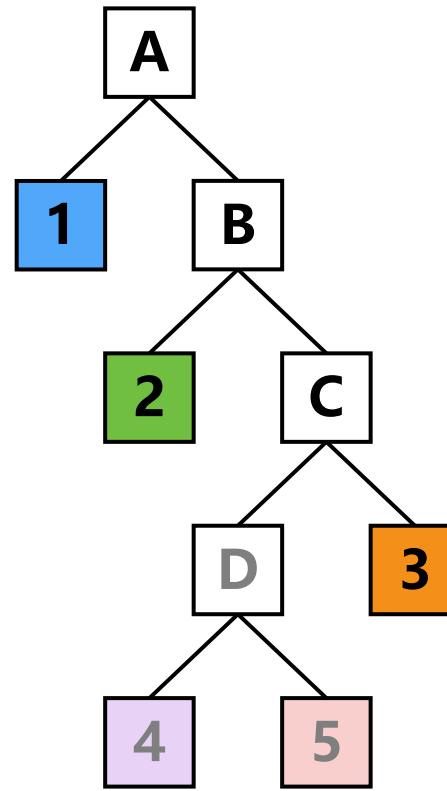
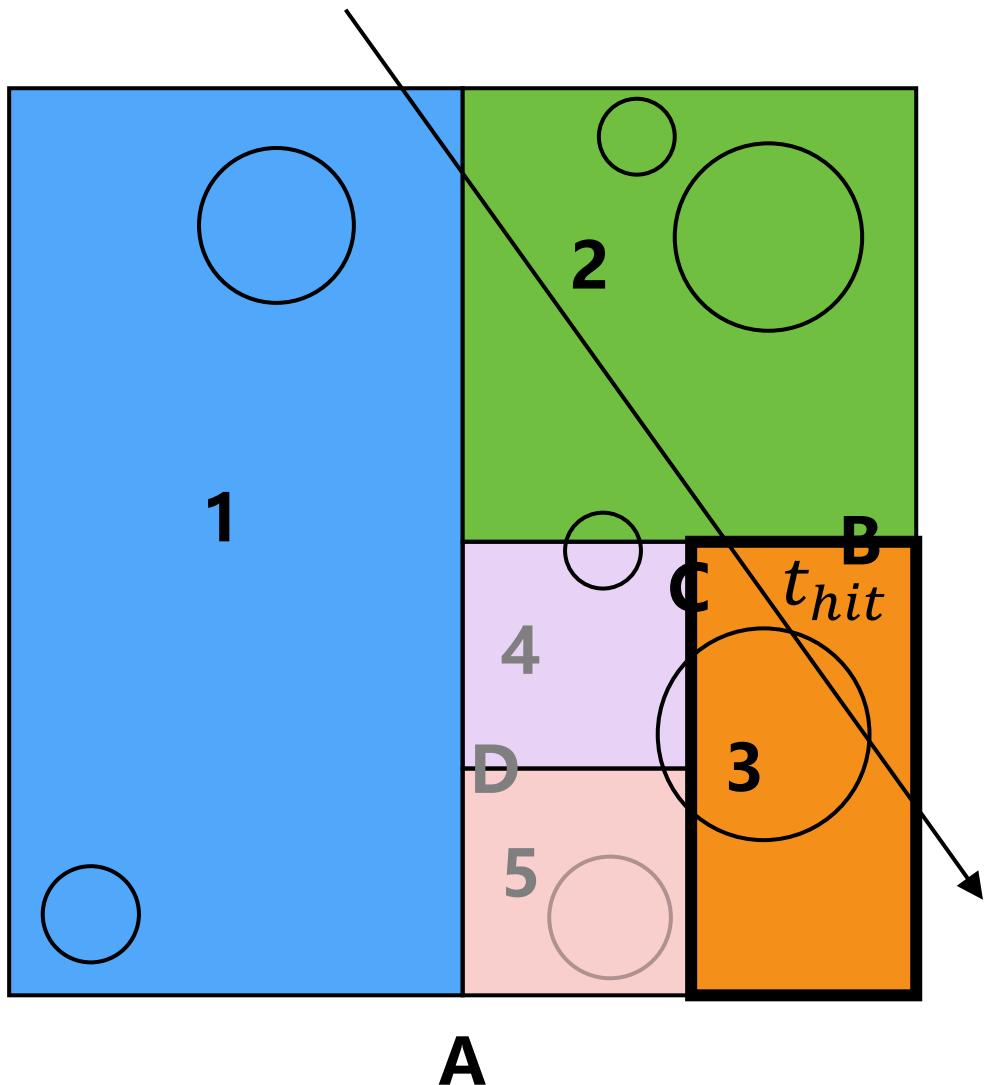
光线网格相交



按光线的走向依次
穿过网格内的格子

对于每一个格子
测试是否与格子
中保存的图像基
元相交

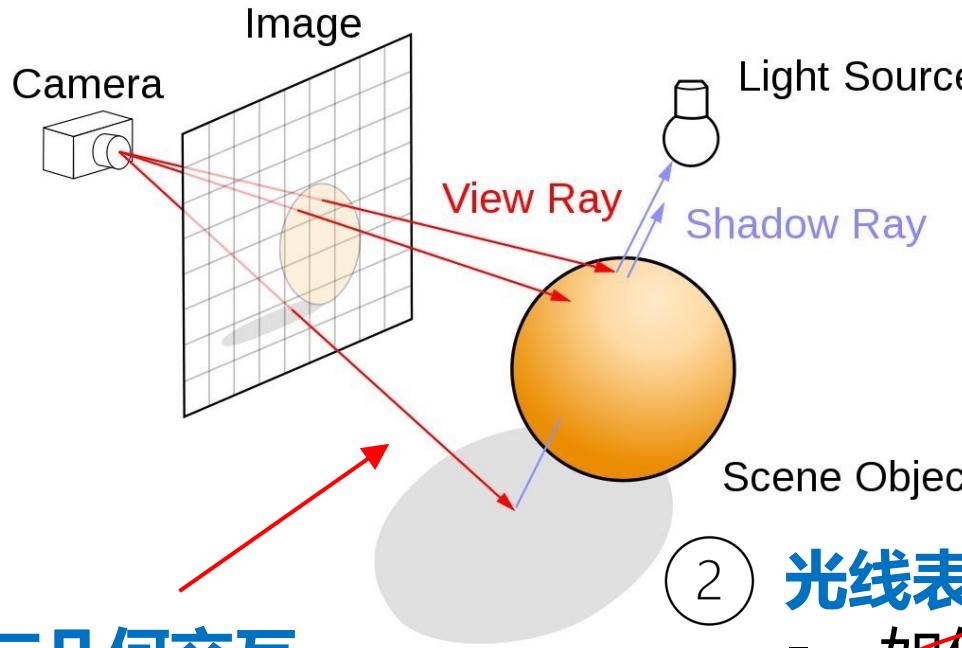
KD 树划分



找到与光线相交的
图像基元!

光线追踪 Ray tracing

分成 **光线与几何交互** 和 **光线表示与可视化** 两部分



① 光线与几何交互

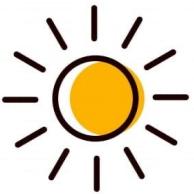
- 查找光线照射在几何中的位置
- 如何加速上述过程

看到哪里

② 光线表示与可视化

- 如何建模光线
- 光线与物体的相互作用，包括反射、折射和散射等
- 全局光照

看到什么



光线感知

高光 (Specular highlights)

大部分光线沿着特定方向散射出去，形成高亮的点

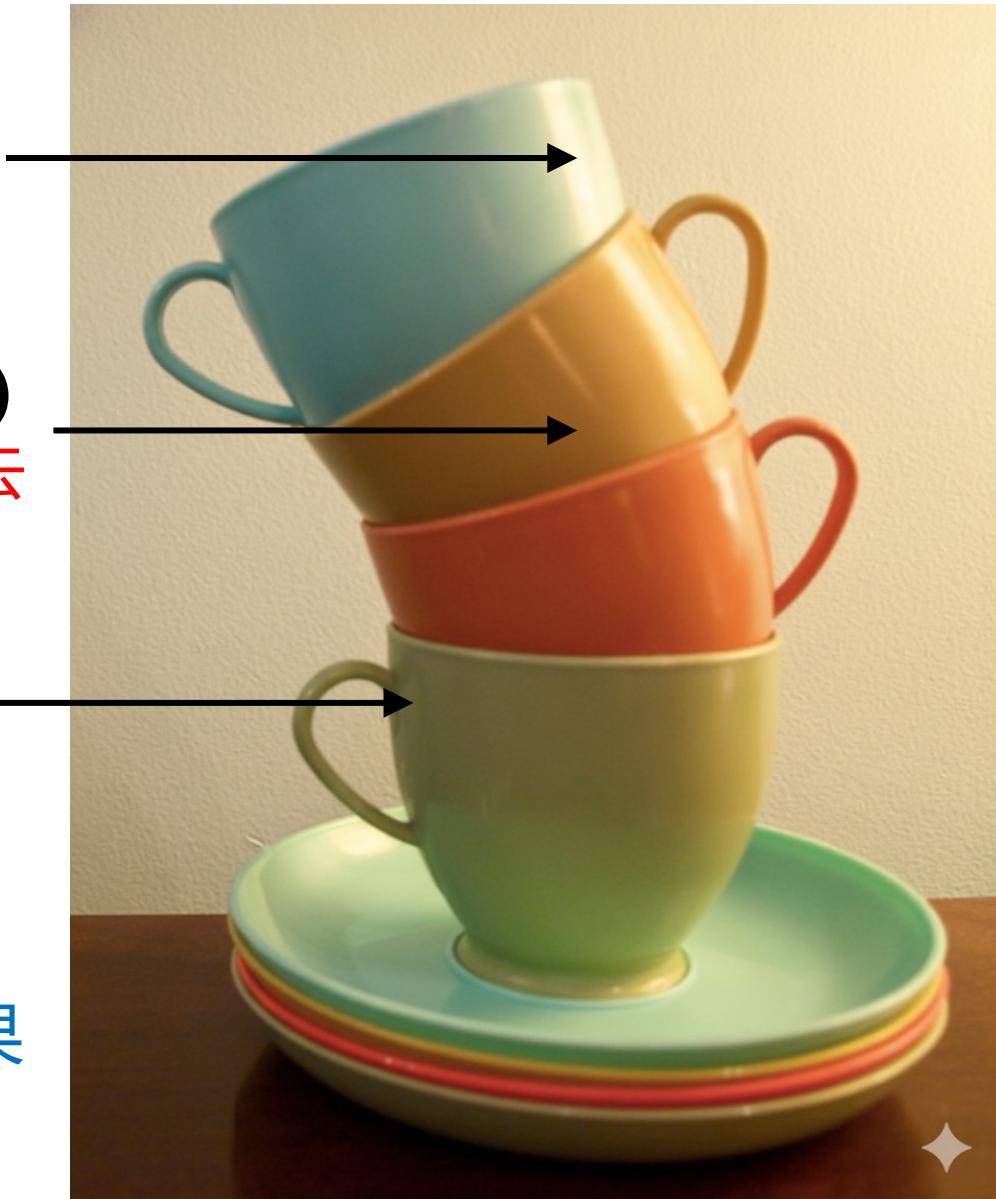
漫反射光 (Diffuse reflection)

光线向各个方向均匀地散射出去

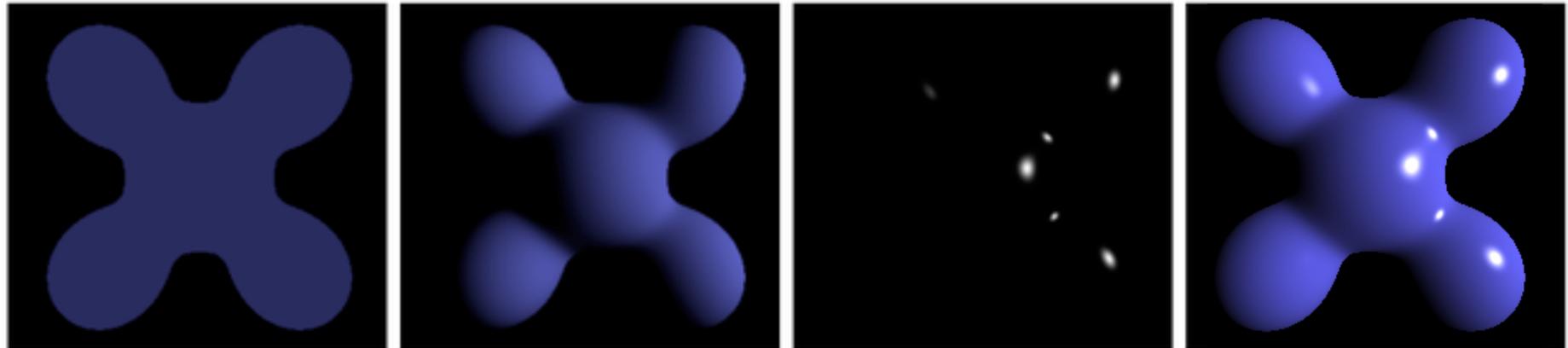
环境光 (Ambient lighting)

光线从所有方向均匀地照射在物体上，可简化为固定光照

BP 反射模型同时考虑了这三种类型的光以生成真实的效果



Blinn-Phong 反射模型



Ambient + Diffuse + Specular = Blinn-Phong
Reflection

$$L = L_a + L_d + L_s$$

$$= k_a I_a + k_d \left(\frac{I}{r^2} \right) \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s \left(\frac{I}{r^2} \right) \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

阴影贴图

- 如何在光栅化中制造阴影效果?
- 阴影贴图 (Shadow mapping)!

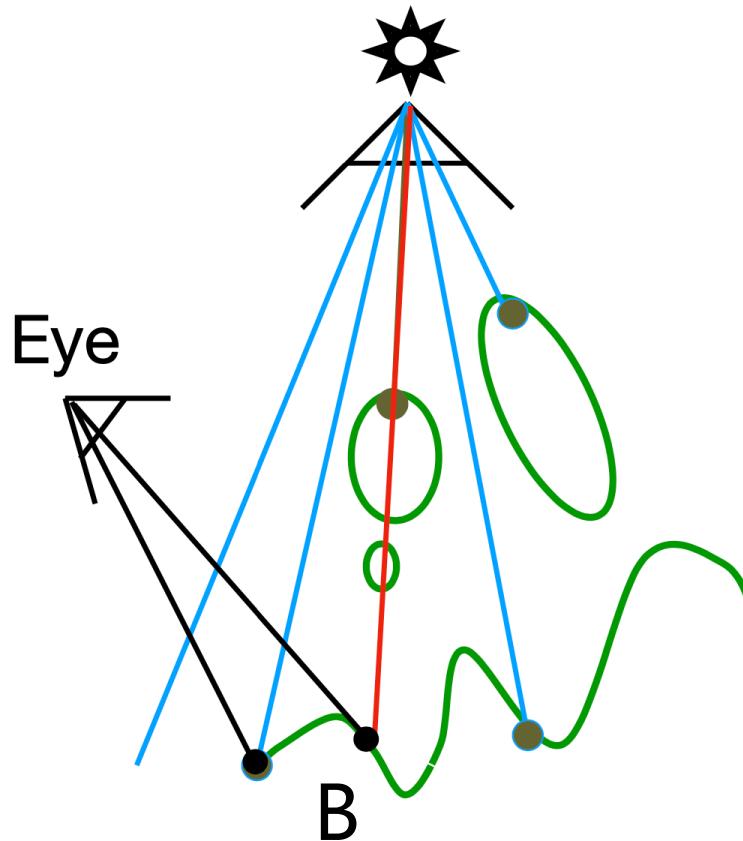


古墓丽影 2018 中的阴影效果

阶段二：阴影测试阶段

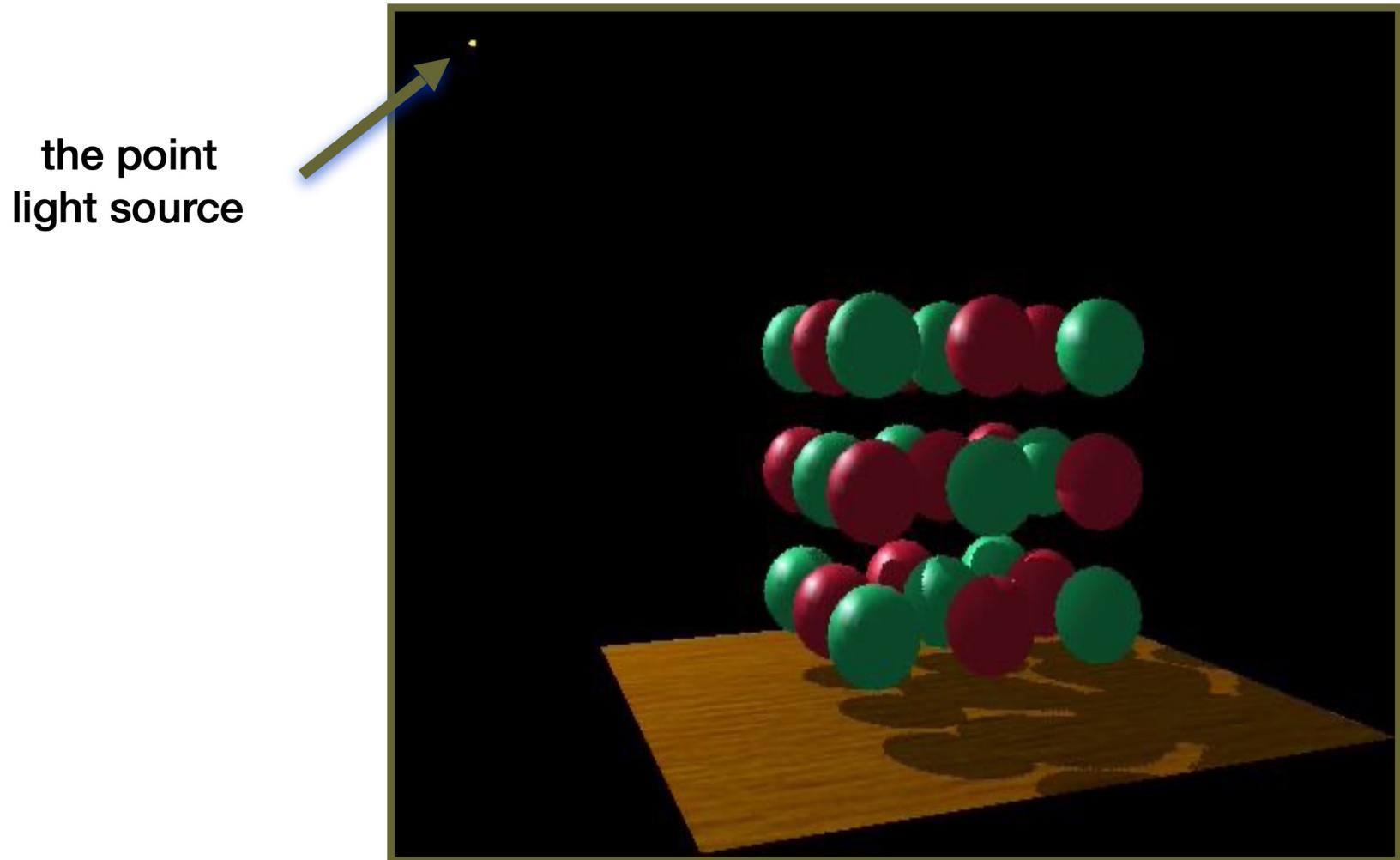
□ 从相机的视角渲染场景

- B 点投影的深度与深度图中记录的对应值不相等
- 因此在阴影中



可视化阴影贴图

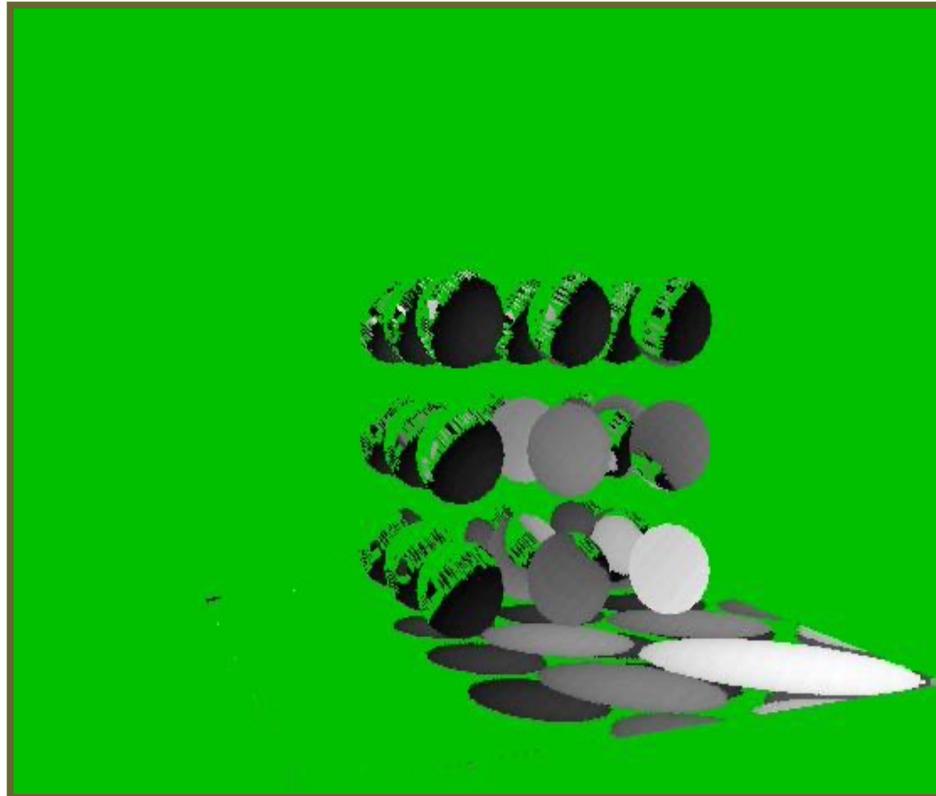
□一个包含阴影的复杂场景



可视化阴影贴图

口对比投影的深度与深度图中记录的深度

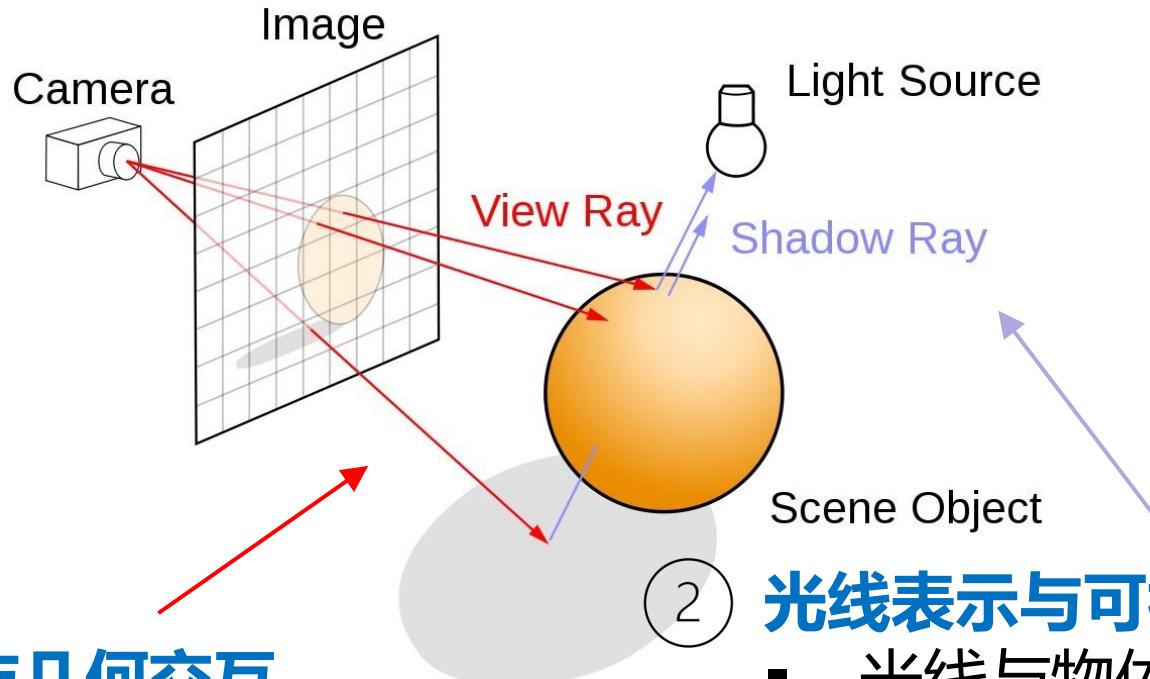
绿色表示二者的值大致相等



非绿色则是阴影所在的区域

光线追踪 Ray tracing

分成 **光线与几何交互** 和 **光线表示与可视化** 两部分



① 光线与几何交互

- 查找光线照射在几何中的位置
- 如何加速上述过程

看到哪里

② 光线表示与可视化

- 光线与物体的相互作用，包括反射、折射和散射等
- 如何建模光线
- 全局光照

看到什么

回顾目前为止的概念

辐射能量

Radiant energy

一段时间、某一面积的撞击数

辐射能量密度

Radiant energy density

一段时间、单位面积的撞击数

辐射功率

Radiant flux

单位时间、某一面积的撞击数

辐射功率密度

Radiant flux density

a.k.a. 辐照度 **Irradiance**

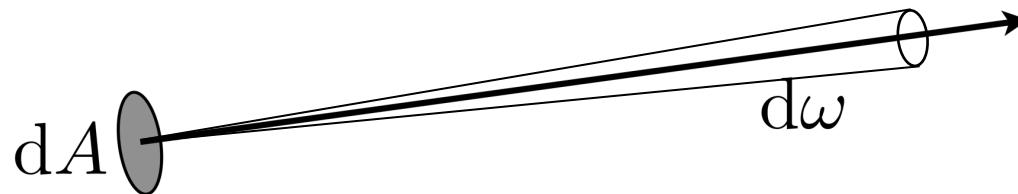
单位时间、单位面积的撞击数

辐(射)亮度 Radiance

口辐亮度是辐照度 Irradiance 的立体角密度 (即辐照度随着立体角的微小变化), 因此在点 p 方向为 ω 的辐亮度为:

$$L(p, \omega) = \lim_{\Delta \rightarrow 0} \frac{\Delta E_\omega(p)}{\Delta \omega} = \frac{dE_\omega(p)}{d\omega} \left[\frac{W}{sr \ m^2} \right] \left[\frac{cd}{m^2} = \frac{lm}{sr \ m^2} = nit \right]$$

其中 E_ω 表示微分的表面积是沿着 ω 方向的:

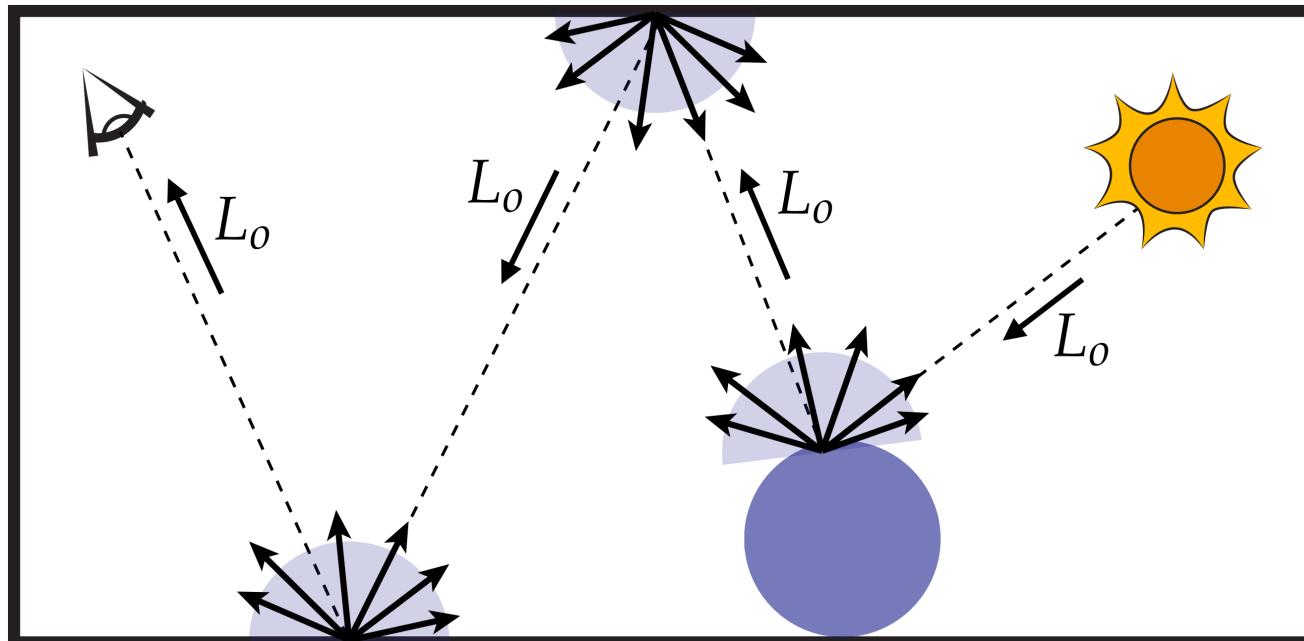


口亦即, 辐亮度是沿着由点 p 和方向 ω 定义的射线的能量

单位时间、单位面积、单位立体角的能量!

辐射度量学 (概念上的) 理解

但是，真实的物理世界不仅仅只有光源照亮物体
还有光线在物体之间形成的**复杂反射**！



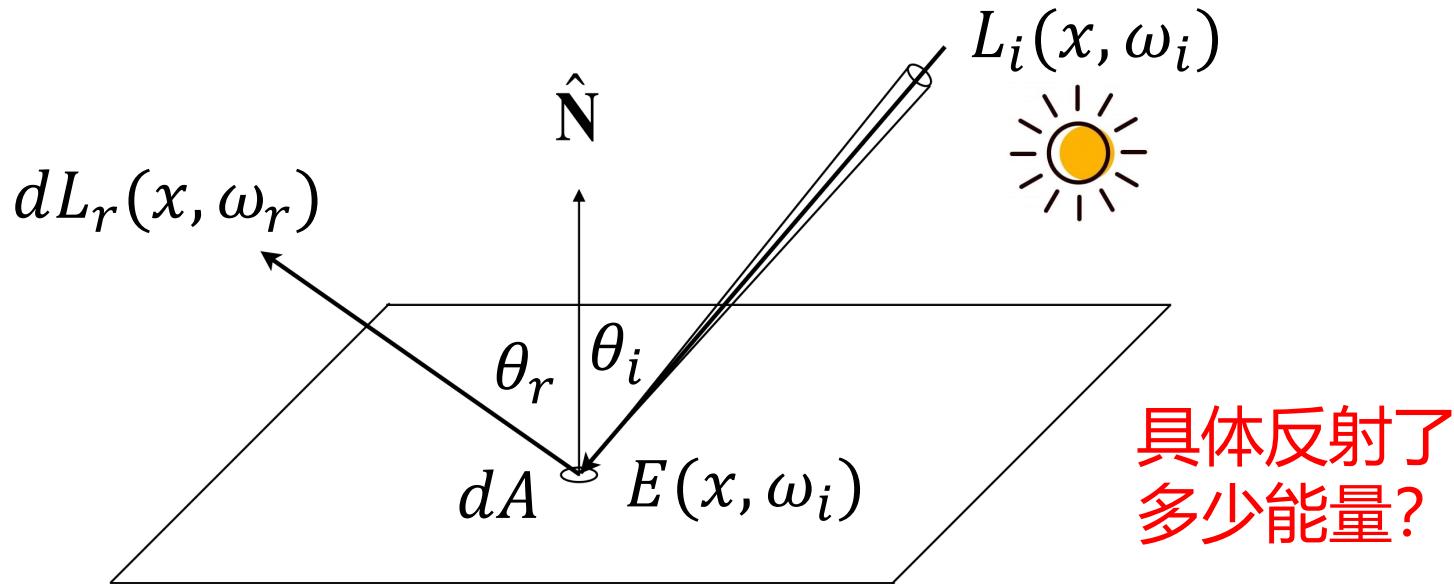
这也是需要光线追踪的原因，光栅化无法建模此过程

对于光在物体表面的**反射**，我们同样可以将物体视作“光源”，计算其如何“照亮”其他物体

一个点的反射

□ 通过计算辐照度，我们能知道一个小区域 dA 从方向 ω_i 接收的能量 E

□ 能量 E 随后会变成 dA 向另一个方向 ω_r 发射的辐亮度

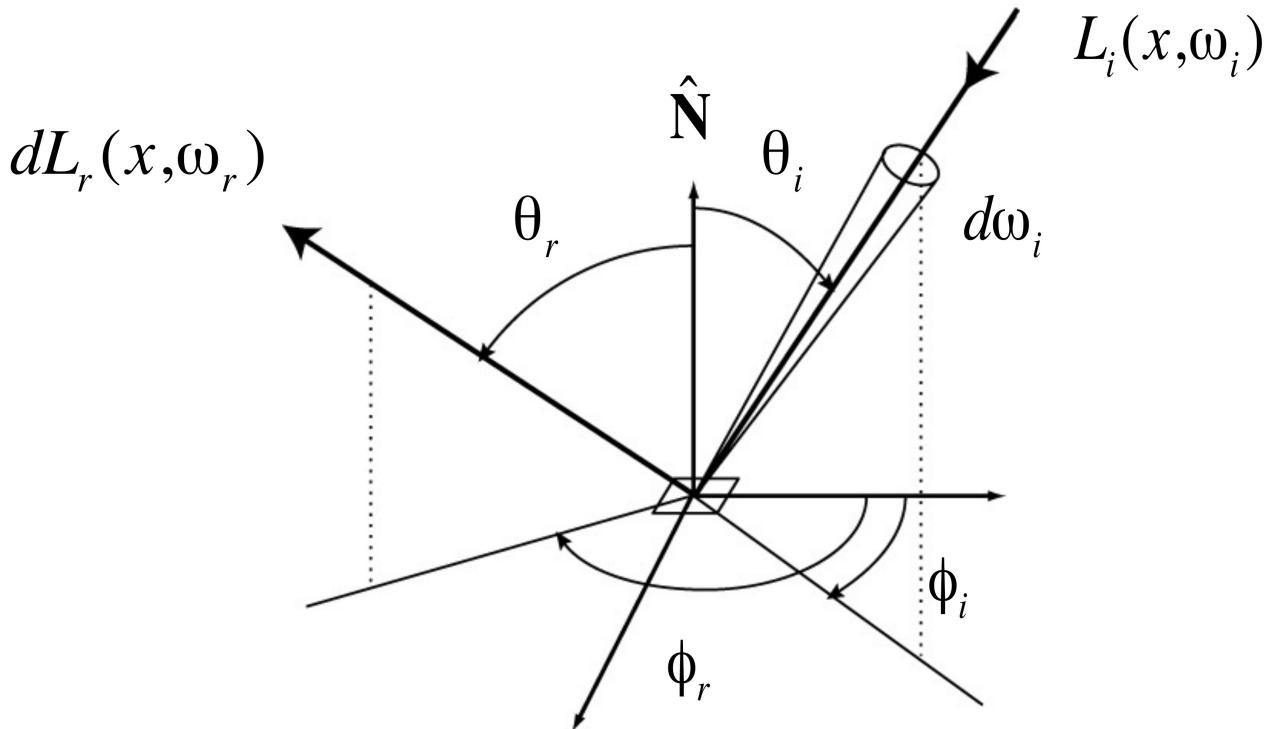


□ 微分入射辐照度： $dE(x, \omega_i) = L_i(x, \omega_i) \cos \theta_i d\omega_i$

□ 微分出射辐亮度 (由于 $dE(x, \omega_i)$)： $dL_r(x, \omega_r)$

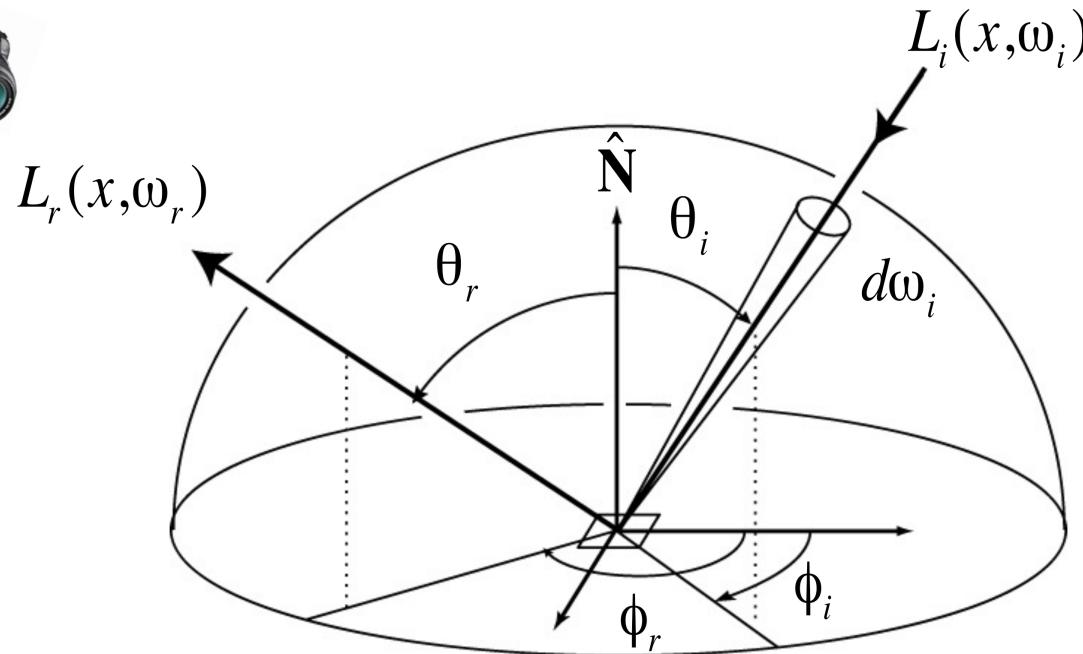
双向反射分布函数 $f_r(\omega_i \rightarrow \omega_r)$

□代表有多少光从入射方向 ω_i 反射到出射方向 ω_r



$$f_r(\omega_i \rightarrow \omega_r) = \frac{dL_r(\omega_r)}{dE_i(\omega_i)} = \frac{dL_r(\omega_r)}{L_i(\omega_i) \cos \theta_i d\omega_i} \left[\frac{1}{sr} \right]$$

反射方程 The reflection equation



$$L_r(x, \omega_r) = \int_{H^2} f_r(x, \omega_i \rightarrow \omega_r) L_i(x, \omega_i) \cos \theta_i d\omega_i$$

辐亮度 Radiance

BRDF

辐照度 irradiance

渲染方程 The rendering equation

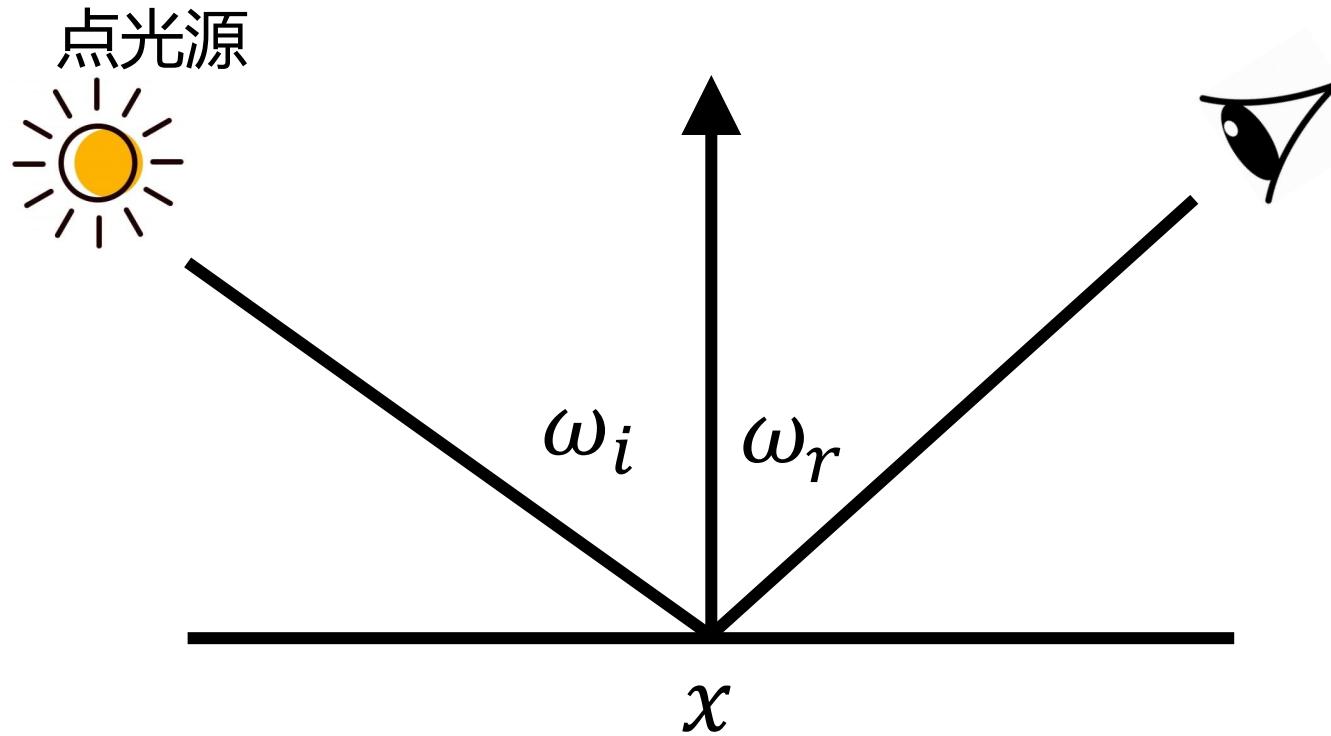
口在反射方程中添加一个 emission term 使其更具一般性

$$L_r(x, \omega_r) = \int_{H^2} f_r(x, \omega_i \rightarrow \omega_r) L_i(x, \omega_i) \cos \theta_i d\omega_i$$



$$L_r(x, \omega_r) = \boxed{L_e(x, \omega_r)} + \int_{\Omega^+} L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) (n \cdot \omega_i) d\omega_i$$

渲染方程



$$L_r(x, \omega_r) = L_e(x, \omega_r) + L_i(x, \omega_i)f_r(x, \omega_i, \omega_r)(n \cdot \omega_i)$$

反射的光线
(即生成的图像)

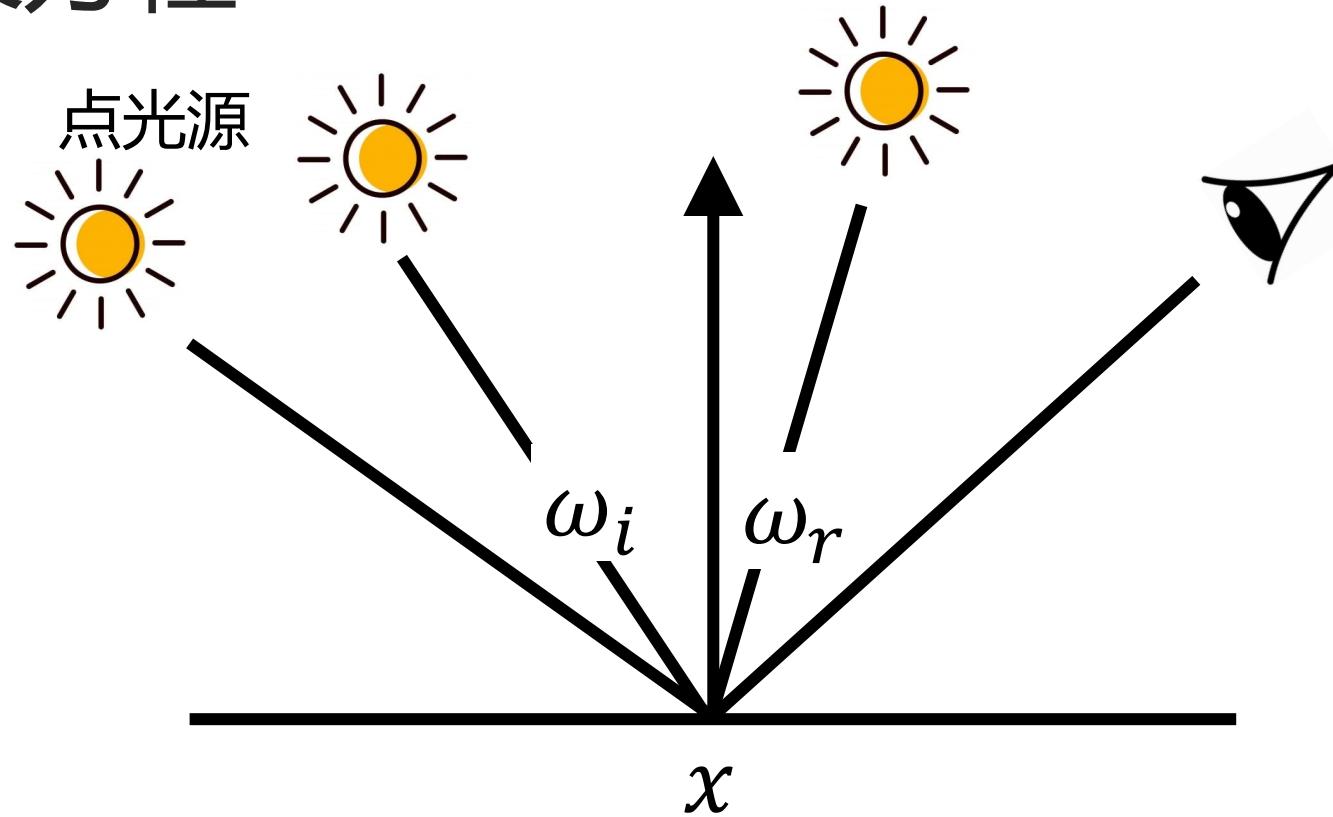
自身光源

来自光源
的入射光

BRDF

入射光
的角度

渲染方程



$$L_r(x, \omega_r) = L_e(x, \omega_r) + \sum L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) (n \cdot \omega_i)$$

反射的光线
(即生成的图像)

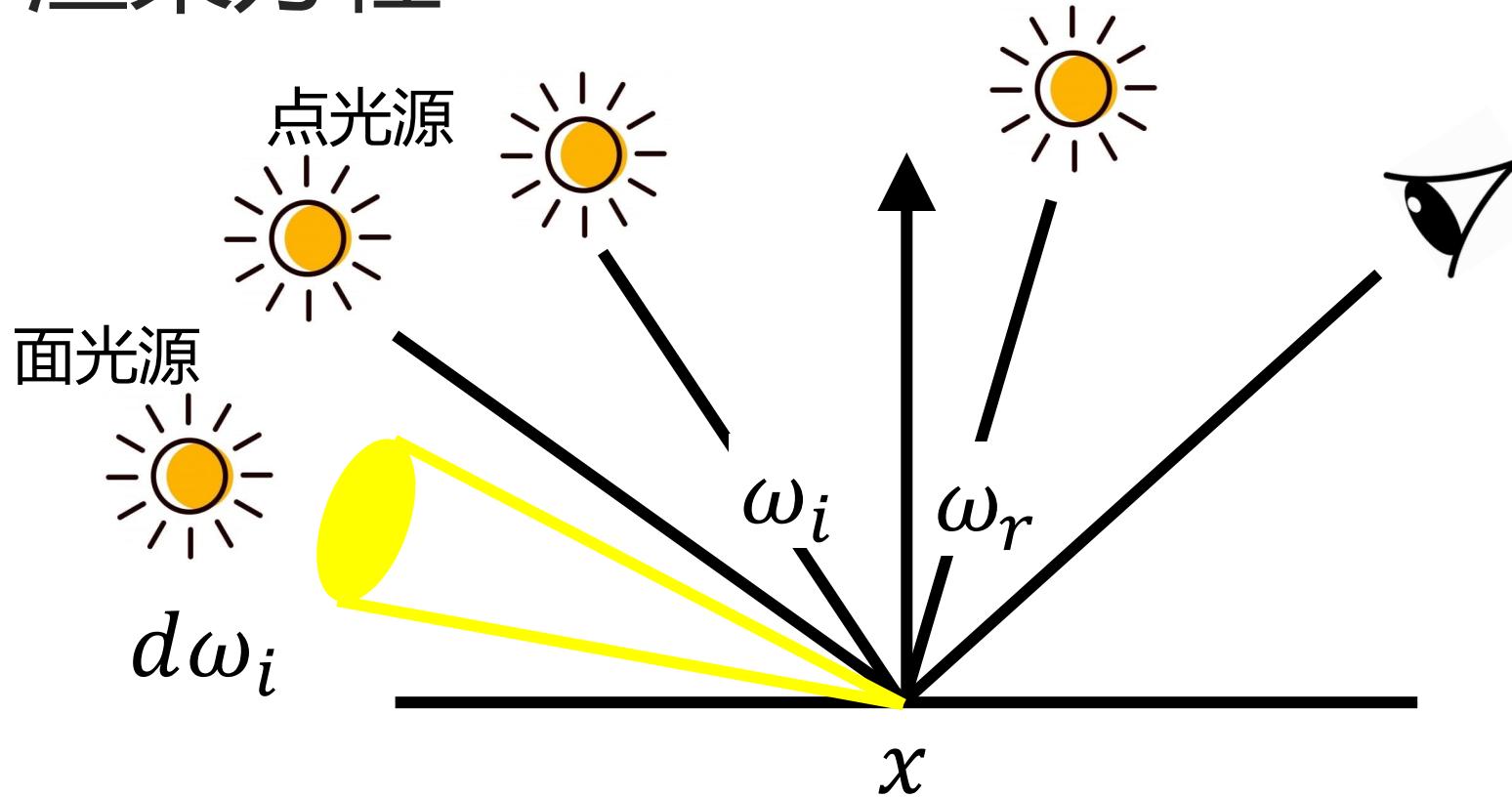
自身光源

来自光源
的入射光

BRDF

入射光
的角度

渲染方程



$$L_r(x, \omega_r) = L_e(x, \omega_r) + \int_{\Omega} L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) \cos \theta_i d\omega_i$$

反射的光线
(即生成的图像)
自身光源

来自光源
的入射光
BRDF

入射光
的角度
213

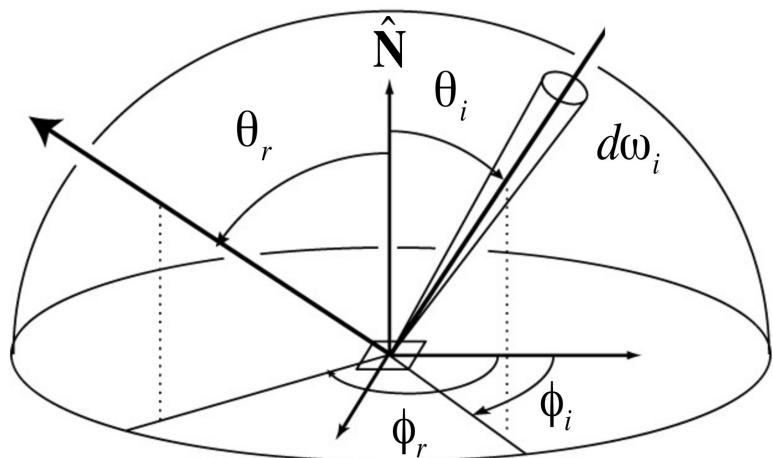
一个简单的模特卡罗解法

回到我们的渲染方程 (假设物体本身不发光)

$$L_r(x, \omega_r) = \int_{\Omega^+} L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) (\mathbf{n} \cdot \omega_i) d\omega_i$$

虽然看起来很复杂，但也只是对半球区域做立体角积分

因此，我们当然可以用蒙特卡罗积分法对其进行求解



一个简单的蒙特卡罗解法

口我们想计算在点 x , 面向相机方向 ω_r 的辐亮度 $L_r(x, \omega_r)$

$$L_r(x, \omega_r) = \int_{\Omega^+} L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) (n \cdot \omega_i) d\omega_i$$

口蒙特卡罗积分法
$$\int_a^b f(x) dx \approx \frac{1}{N} \sum_{k=1}^N \frac{f(X_k)}{p(X_k)} \quad X_k \sim p(x)$$

口我们的 $f(x)$ 是什么? $L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) (n \cdot \omega_i)$

口我们的是 $p(x)$ 什么? $p(\omega_i) = \frac{1}{2\pi}$

假设我们对半球面进行均匀采样

一个简单的蒙特卡罗解法

□ 蒙特卡罗积分结果

$$\begin{aligned} L_r(x, \omega_r) &= \int_{\Omega^+} L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) (\mathbf{n} \cdot \omega_i) d\omega_i \\ &\approx \frac{1}{N} \sum_{i=1}^N \frac{L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) (\mathbf{n} \cdot \omega_i)}{p(\omega_i)} \end{aligned}$$

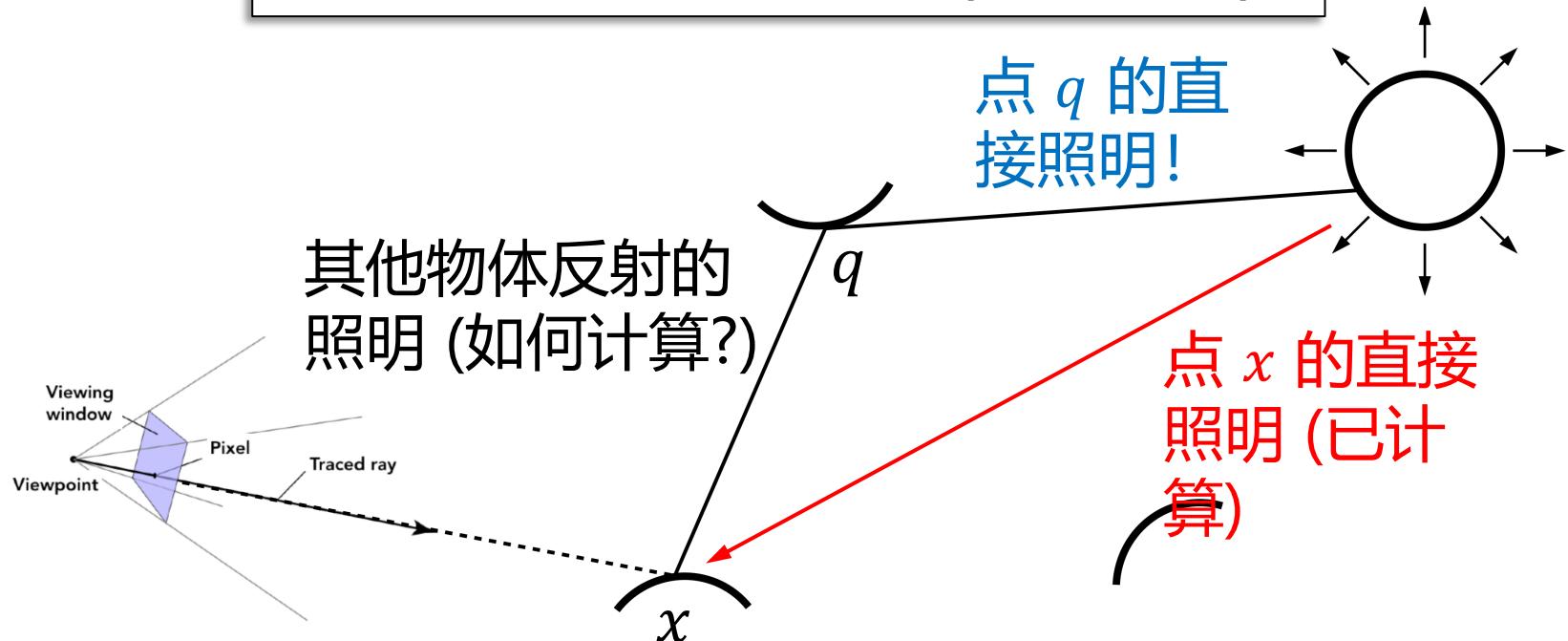
□ 这个公式表示什么？

□ 一种正确的、适用于**直接照明**的着色算法

全局光照 Global Illumination

- 口向前一步：刚刚只计算了点 x 来自直接照明的能量
- 口如果光线击中其他物体（点 q ）并反射到点 x ，如何计算此部分的能量？

点 x 来自点 q 的能量即为点 q 对来自直接照明显量的反射（形成递归）



全局光照

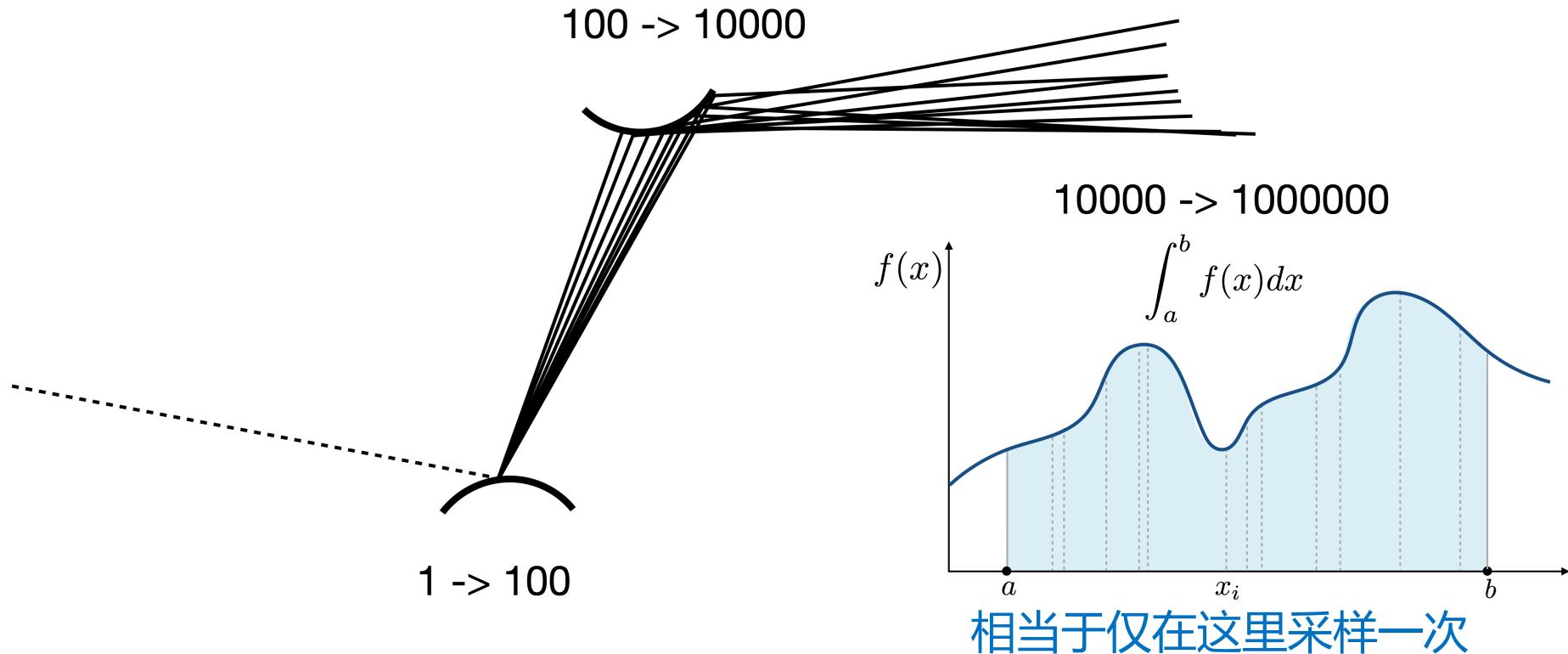
```
shade(p, wr )  
    Randomly choose N directions wi~pdf  
    Lo = 0.0  
    For each wi  
        Trace a ray r(p, wi)  
        If ray r hit the light  
            Lo += (1 / N) * L_i * f_r * cosine / pdf(wi)  
        Else If ray r hit an object at q  
            Lo += (1 / N) * shade(q, -wi) * f_r * cosine  
            / pdf(wi)  
    Return Lo
```

这样是否就完成了? No. 至少还有 2 个问题!

路径追踪

□问题 1：随着光线反弹，光线的数量将爆炸

$$\#rays = N^{\#bounces}$$



□为了避免光线爆炸，当且仅当 $N = 1$ ！

路径追踪

从现在起，我们假设在每个着色点仅跟踪一条反射光线：

shade(p, wr)

Randomly choose **ONE** direction $wi \sim pdf(w)$

Trace a ray $r(p, wi)$

If ray r hit the light

Return $L_i * f_r * cosine / pdf(wi)$

Else If ray r hit an object at q

Return $shade(q, -wi) * f_r * cosine / pdf(wi)$

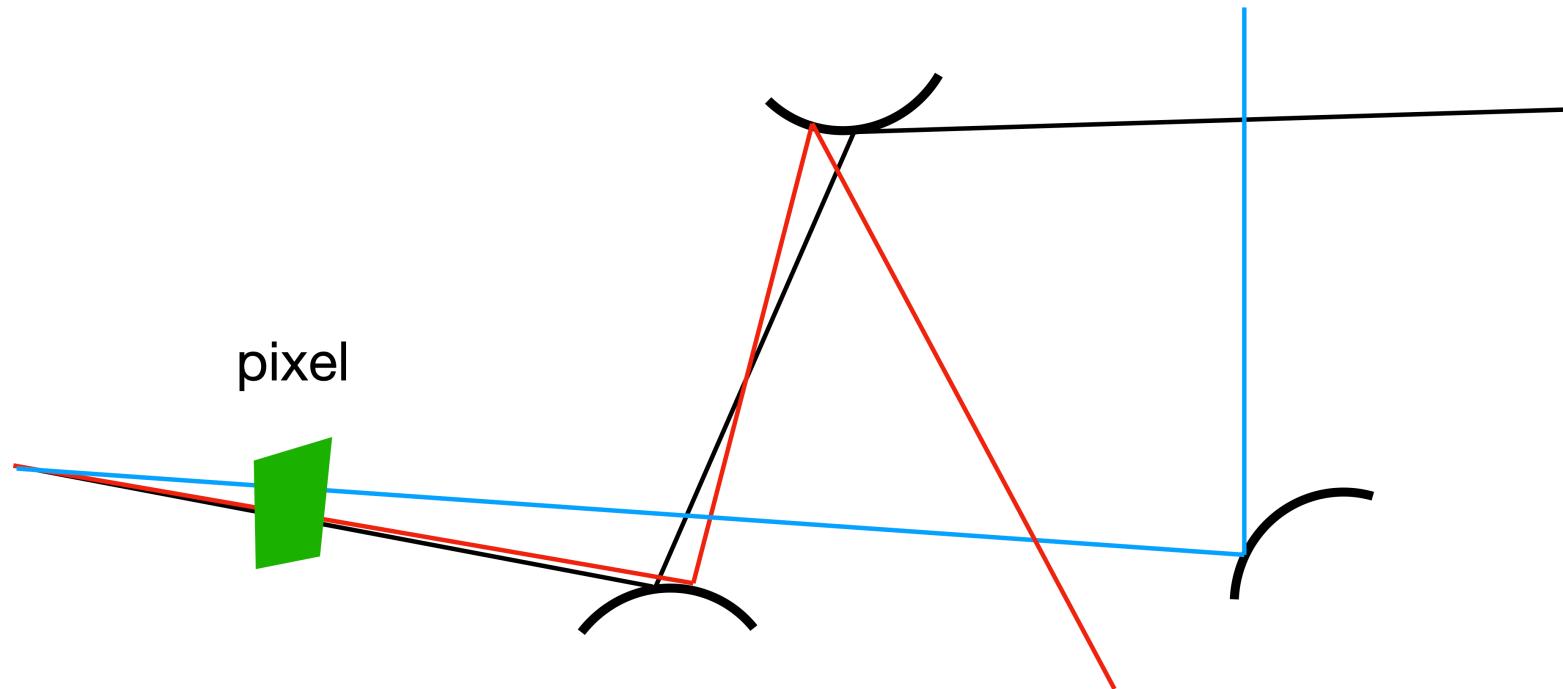
这就是**路径追踪 (Path tracing)**，即 $N = 1$

光线生成

□但是这种做法会充满噪声！

□没关系，只需在每个像素中追踪更多光线，并平均不同光线的辐亮度

□这里每条光线在每个着色点仅反射一次，区别于反射多次的情况（会形成光线爆炸）



光线生成算法

与光线投射很类似

```
ray_generation(camPos, pixel)
```

Uniformly choose N sample positions within the pixel

```
pixel_radiance = 0.0
```

For each sample in the pixel

```
    Shoot a ray r(camPos, cam_to_sample)
```

```
    If ray r hit the scene at p
```

```
        pixel_radiance += 1 / N * shade(p, sample_to_cam)
```

Return pixel_radiance

光线追踪

□ 将目前的情况结合起来

shade(p, wr)

Randomly choose ONE direction $w_i \sim \text{pdf}(w)$

Trace a ray $r(p, w_i)$

If ray r hit the light

 Return $L_i * f_r * \text{cosine} / \text{pdf}(w_i)$

Else If ray r hit an object at q

 Return **shade(q, -wi) * f_r * cosine / pdf(wi)**



递归!

□ 这样是否就完成了?

□ No, 这个递归算法永远不会结束!

解决方法：俄罗斯轮盘赌

口之前我们总是向着色点射出光线，并获得着色结果 L_r

口现在我们手动设置一个概率 p ($0 < p < 1$)

口在概率 p 下，我们射出光线并将着色结果除以 p : L_r/p

口在概率 $1 - p$ 下，**不发射光线，着色结果为 0**

口这样，着色结果的期望值仍然为 L_r ！

$$E = p * \left(\frac{L_r}{p}\right) + (1 - p) * 0 = L_r$$

解决方法：俄罗斯轮盘赌

shade(p, wr)

Manually specify a probability P_RR

Randomly select ksi in a uniform dist. in [0, 1]

If (ksi > P_RR) return 0.0;

Randomly choose ONE direction wi~pdf(w)

Trace a ray r(p, wi)

If ray r hit the light

 Return L_i * f_r * cosine / pdf(wi) / P_RR

Else If ray r hit an object at q

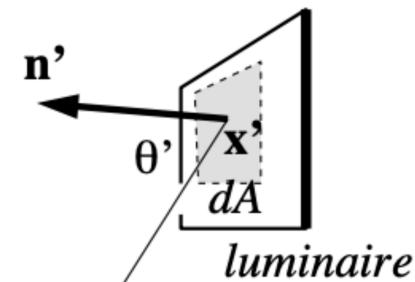
 Return shade(q, -wi) * f_r * cosine / pdf(wi) / P_RR

对光线进行采样

□ 先对光源采样，然后投影到半球上

$$L_r(x, \omega_r) = \int_{\Omega^+} L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) (\mathbf{n} \cdot \omega_i) d\omega_i$$

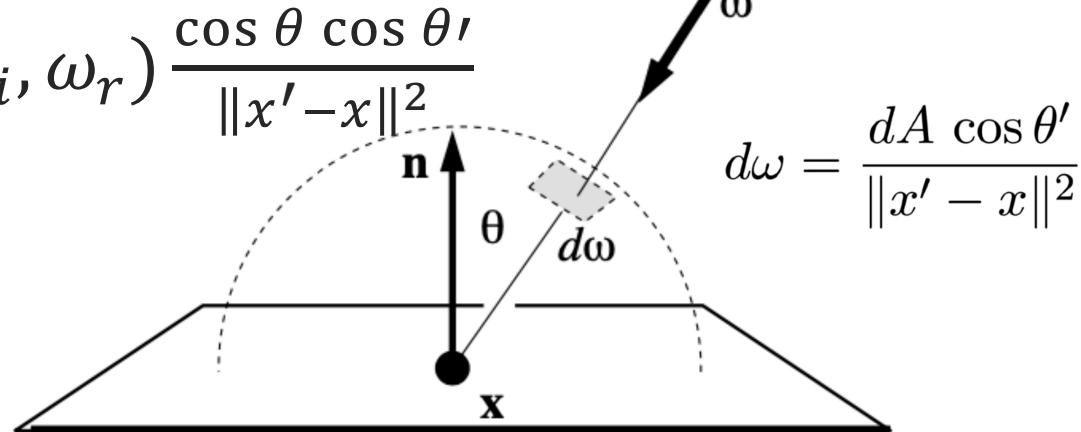
$$= \int_A L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) \frac{\cos \theta \cos \theta'}{\|x' - x\|^2} dA$$



□ 改变积分变量，现在是针对光源的积分

$$\square f(x): L_i(x, \omega_i) f_r(x, \omega_i, \omega_r) \frac{\cos \theta \cos \theta'}{\|x' - x\|^2}$$

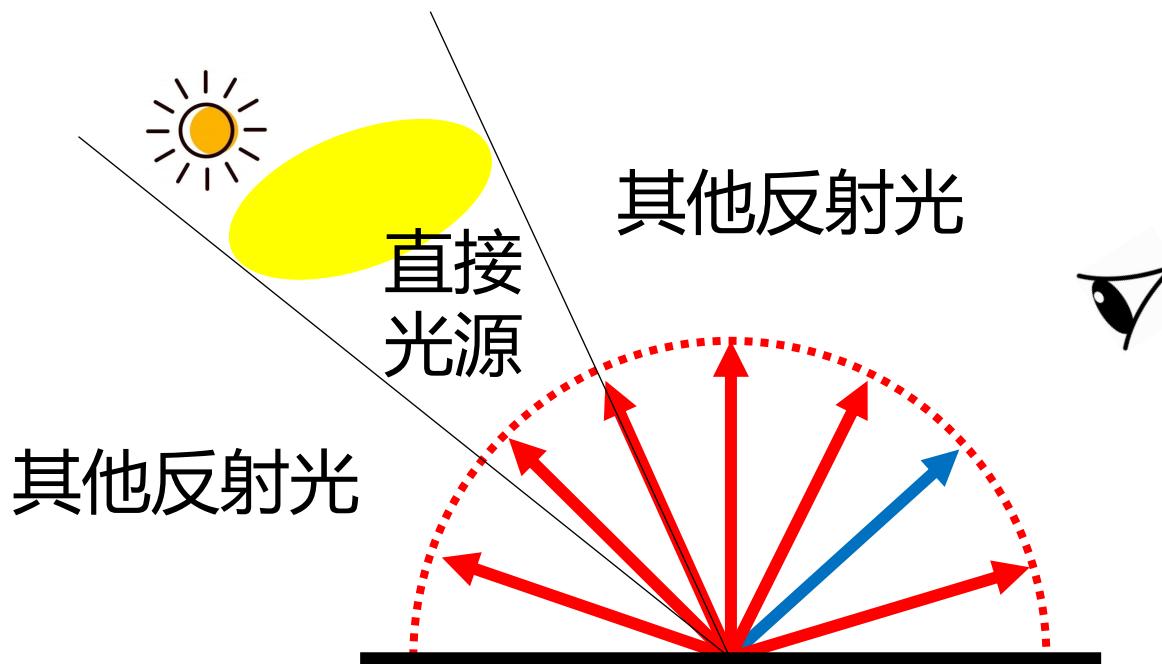
$$\square p(x): \frac{1}{A}$$



采样光源

口先前，我们假设光线是"偶然"通过均匀半球采样射出的
口现在，我们考虑来自两部分的辐亮度 Radiance：

1. 光源 (直接光照，不需要使用俄罗斯轮盘赌)
2. 其他反射光 (间接光照，需要使用俄罗斯轮盘赌)



采样光源

shade(p, wo)

Contribution from the light source.

Uniformly sample the light at x' ($\text{pdf_light} = 1 / A$)

$L_{\text{dir}} = L_i * f_r * \cos \theta * \cos \theta' / |x' - p|^2 / \text{pdf_light}$

Contribution from other reflectors.

$L_{\text{indir}} = 0.0$

Test Russian Roulette with probability P_{RR}

Uniformly sample the hemisphere toward w_i ($\text{pdf_hemi} = 1 / 2\pi$)

Trace a ray $r(p, wi)$

If ray r hit a **non-emitting** object at q

$L_{\text{indir}} = \text{shade}(q, -wi) * f_r * \cos \theta / \text{pdf_hemi} / P_{RR}$

Return $L_{\text{dir}} + L_{\text{indir}}$

对光线进行采样

口最后一件事是：我们如何知道着色点是否被遮挡？

Contribution from the light source.

L_dir = 0.0

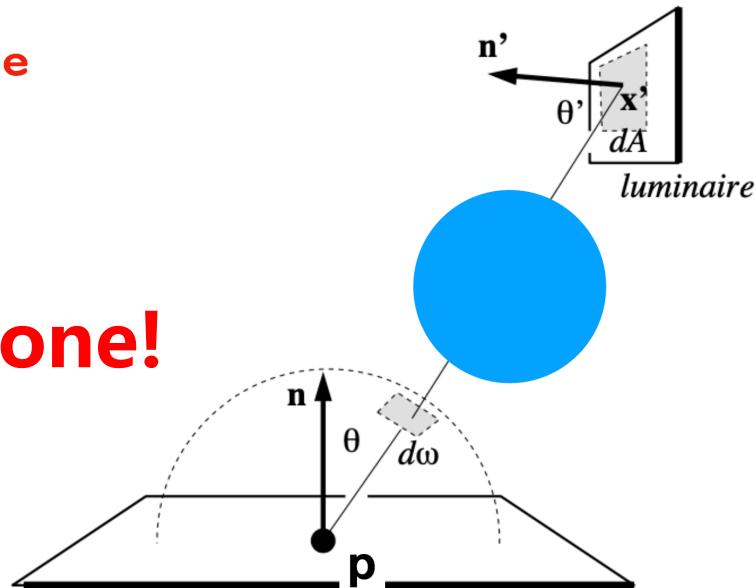
Uniformly sample the light at x' ($\text{pdf_light} = 1 / A$)

Shoot a ray from p to x'

If the ray is not blocked **in the middle**

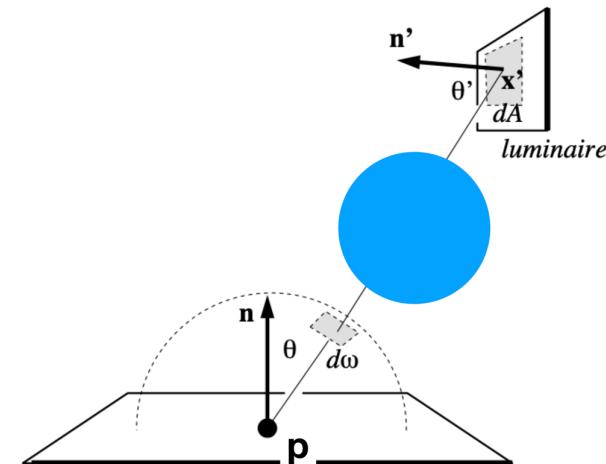
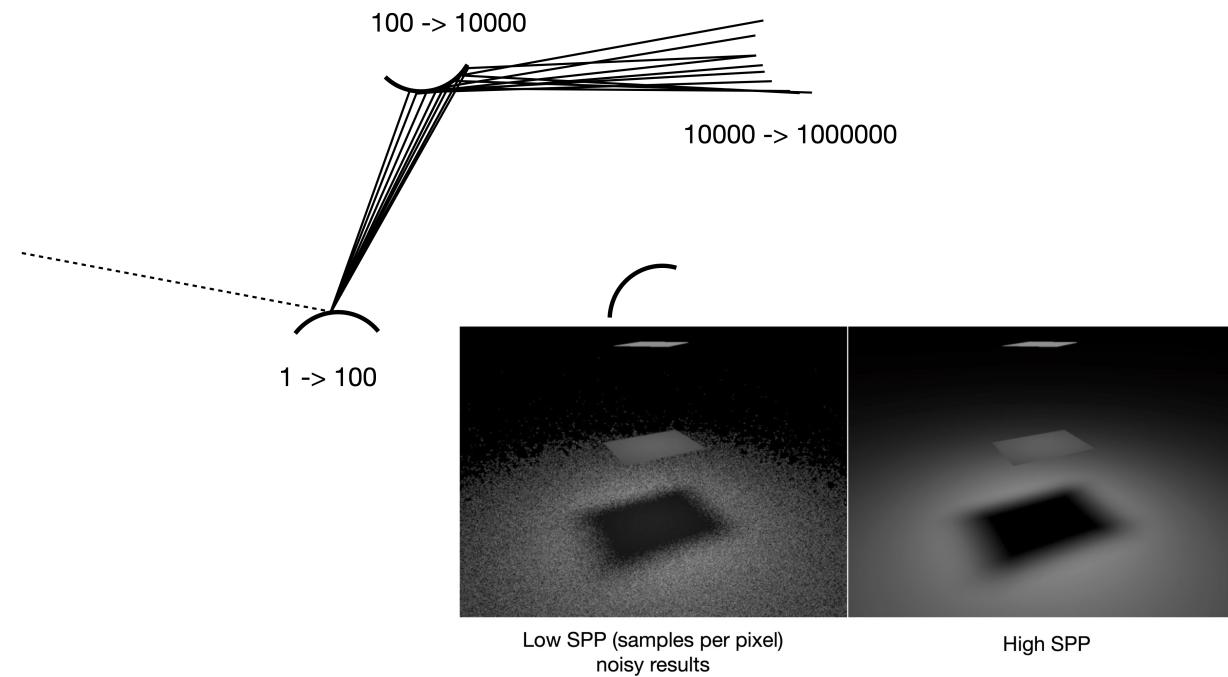
L_dir = ...

Now path tracing is finally done!



总结 – 路径追踪算法 (解法)

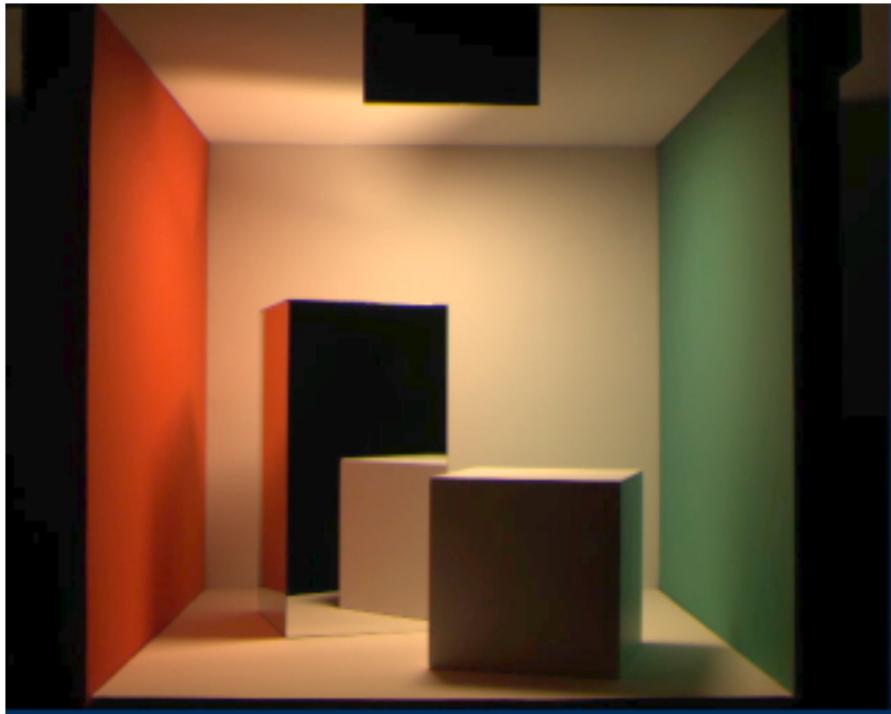
- 利用蒙特卡罗积分法近似求解
- 假定光线在物体表面仅反射一次 (即采样函数仅采样一次) 来避免光线爆炸，通过发射多条光线减少噪声
- 通过在光源处 (而不是半球体处) 采样来增加计算效率
- 通过碰撞检测判断着色点与光源之间是否存在遮挡物



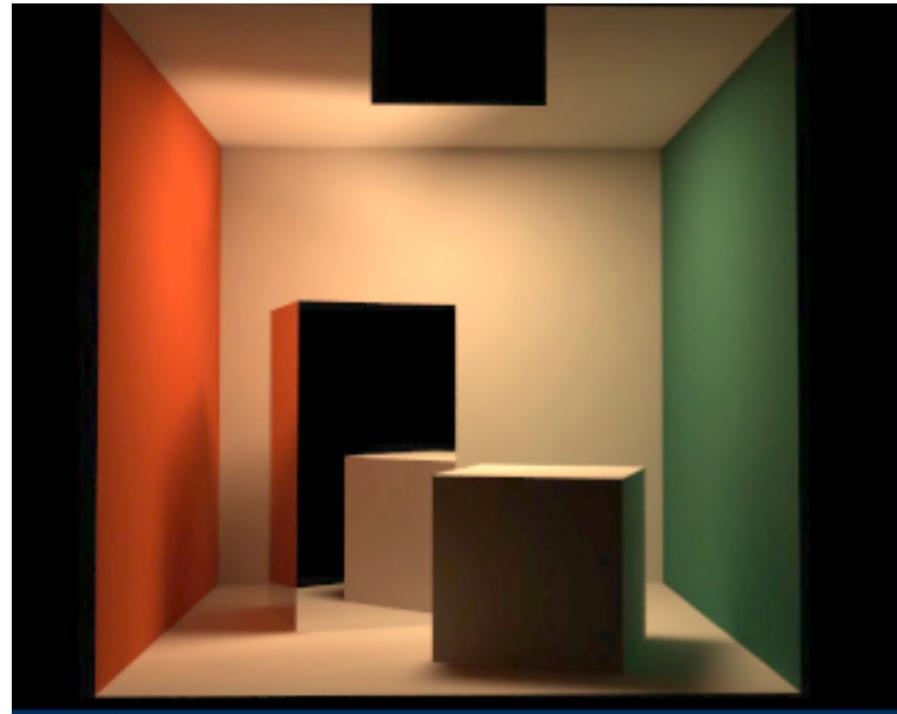
路径追踪是否正确？

口是的，几乎 100% 正确，能达到真实照片的水平

口下列哪张图片是照片，哪张是算法生成的？



照片



路径追踪算法生成的
全局光照



中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬
软件工程学院
chenzhb36@mail.sysu.edu.cn