



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

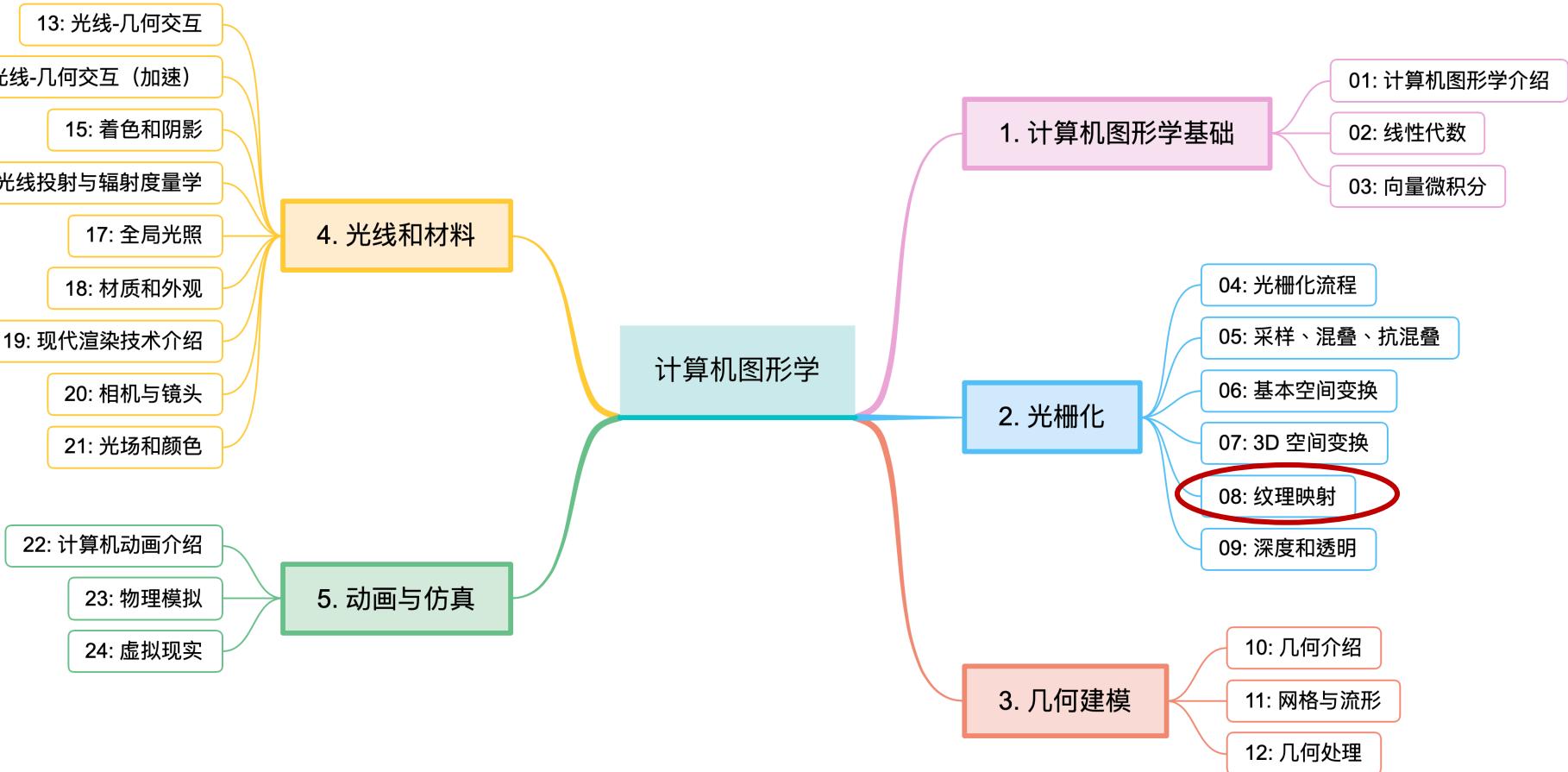
Lecture 08: 透视投影和纹理映射

SSE315: 计算机图形学
Computer Graphics

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn



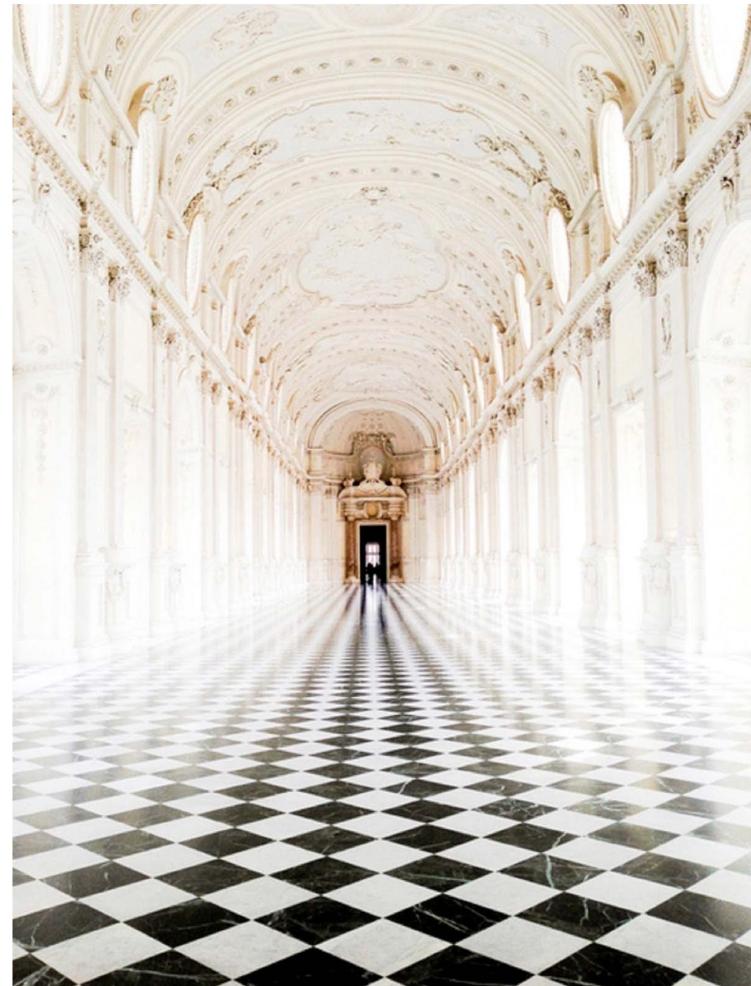
透视与纹理 Perspective & Texture

□ Previously

- 光栅化 (rasterization)
如何把图形基元转换成像素
- 变换 (transformations)
如何在空间中操纵图形基元

□ Today

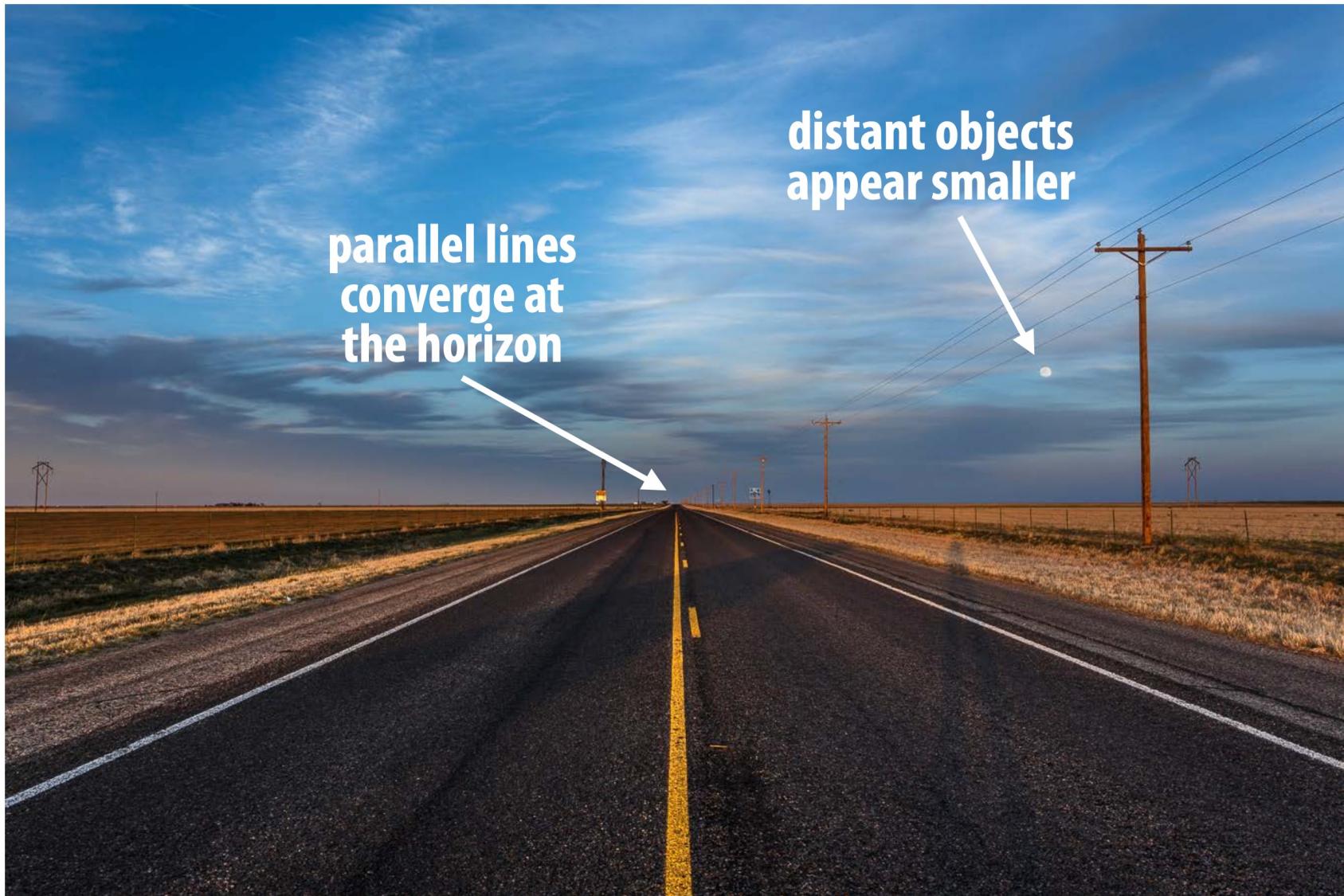
- 学习它们如何结合在一起
- 回顾透视变换 (perspective transformations)
- 讨论如何将纹理映射到基元上以获得更多细节
- 以及透视给纹理映射带来哪些挑战



透视投影

Perspective Projection

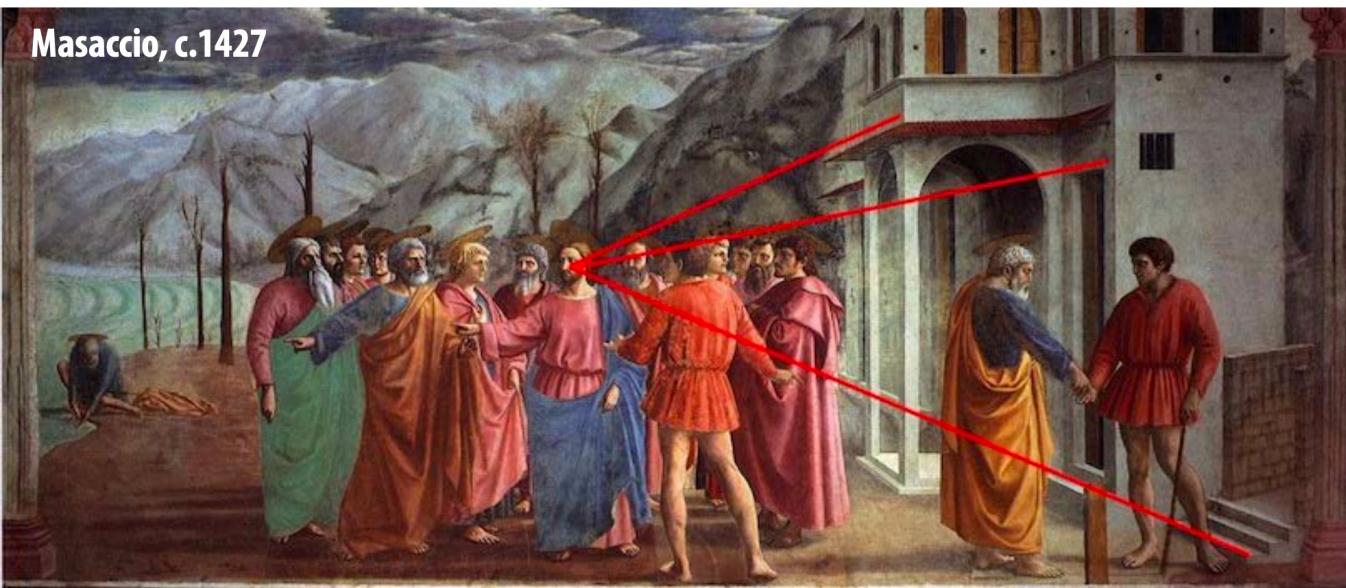
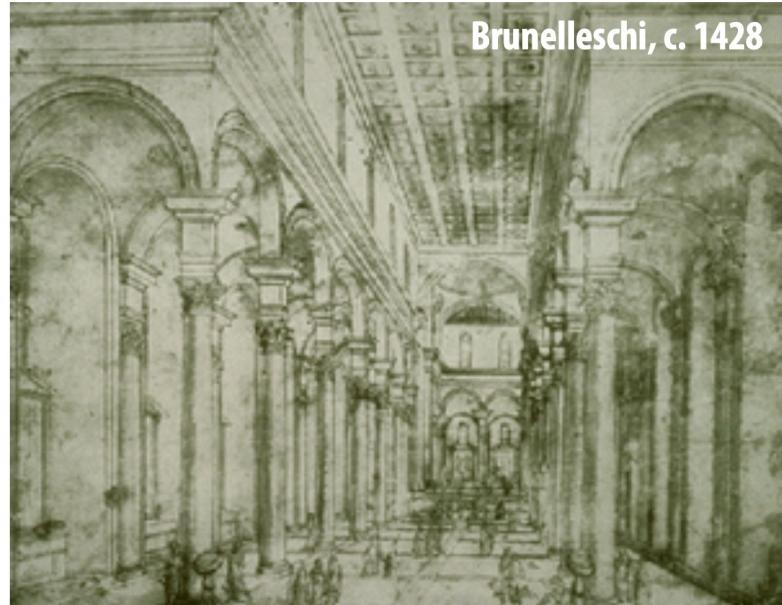
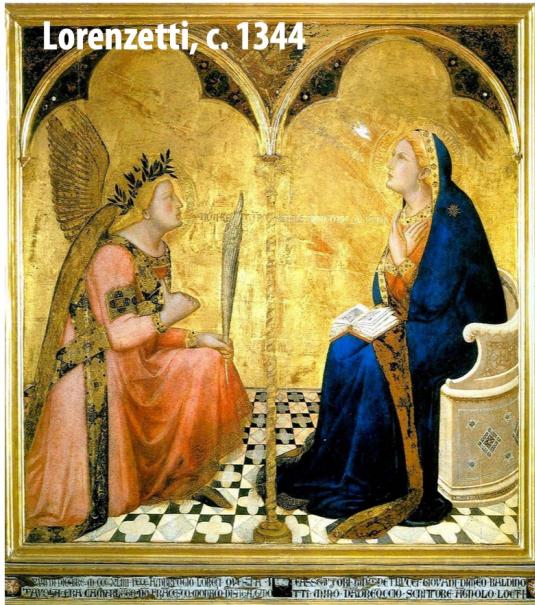
透视投影



早期绘画：不正确的透视



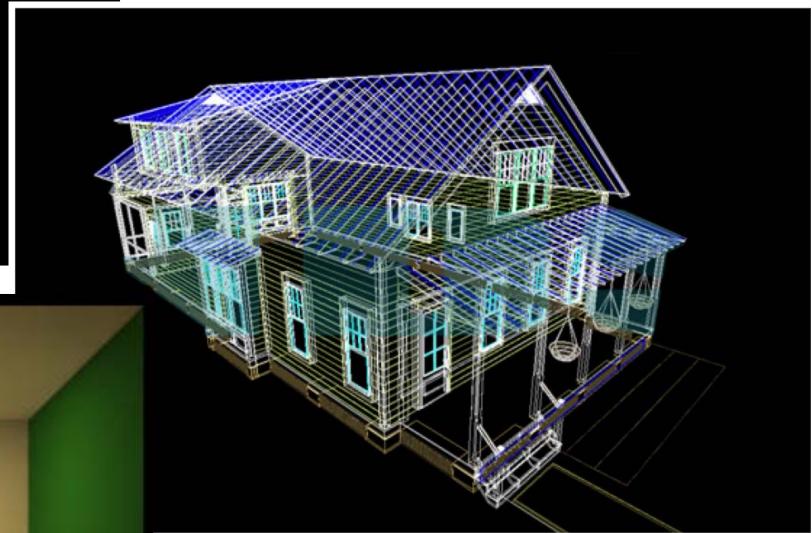
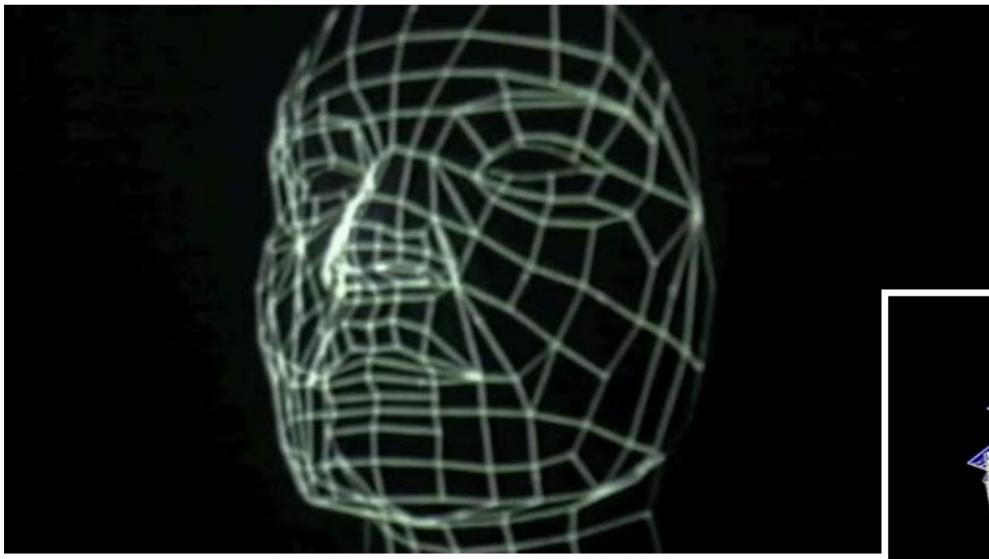
逐渐过渡到正确的透视



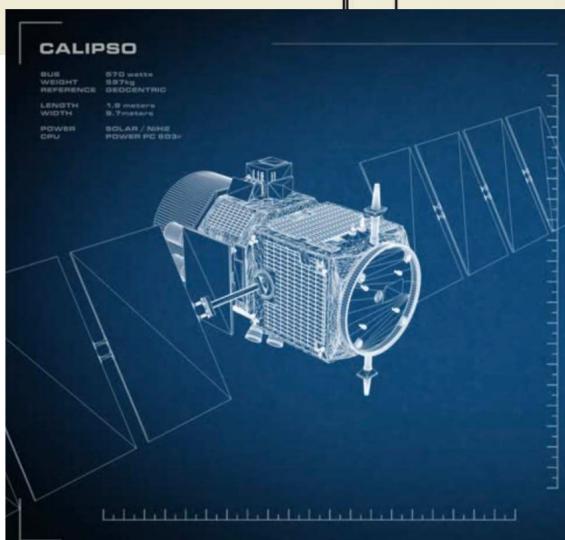
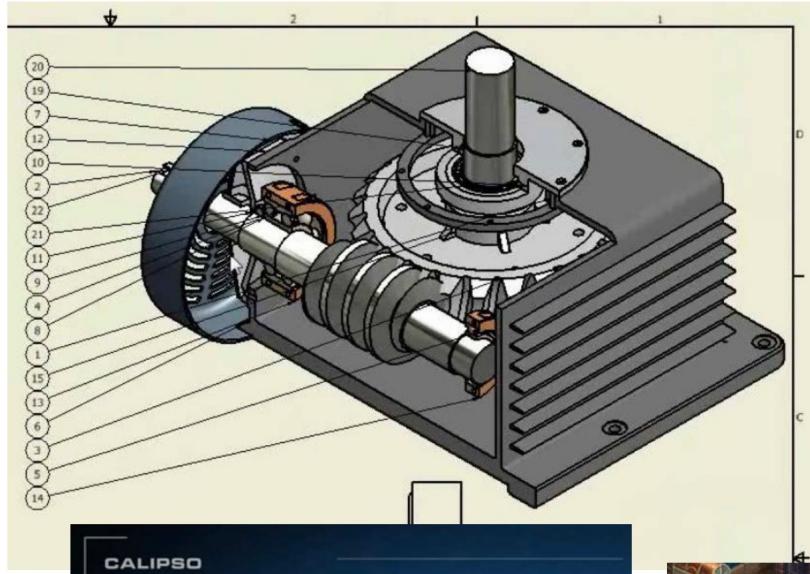
后来又拒绝使用正确的透视投影



回到计算机图形学



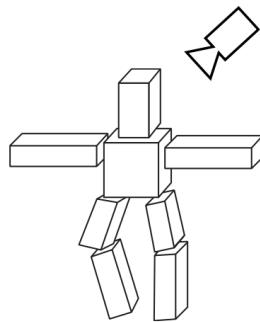
有些场景不能使用透视



Orthographic projection 正交投影

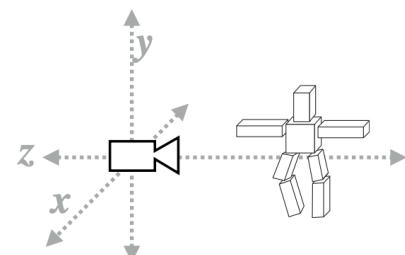
变换：从物体到显示器

[WORLD COORDINATES]



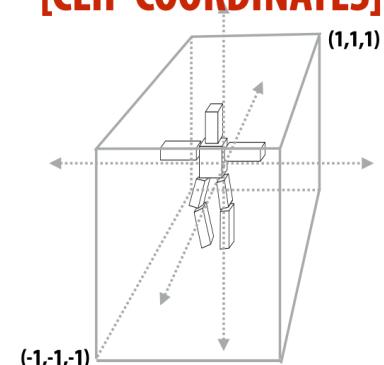
original description
of objects

[VIEW COORDINATES]



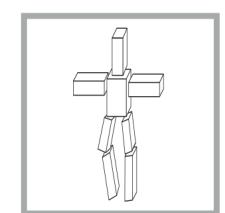
all positions now expressed
relative to camera; camera
is sitting at origin looking
down -z direction

[CLIP COORDINATES]



everything visible to the
camera is mapped to unit
cube for easy “clipping”

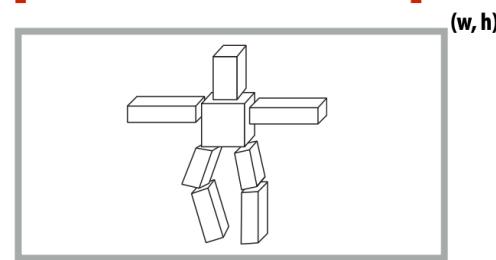
[NORMALIZED COORDINATES]



unit cube mapped to unit
square via perspective divide

2D primitives can
now be drawn via
rasterization

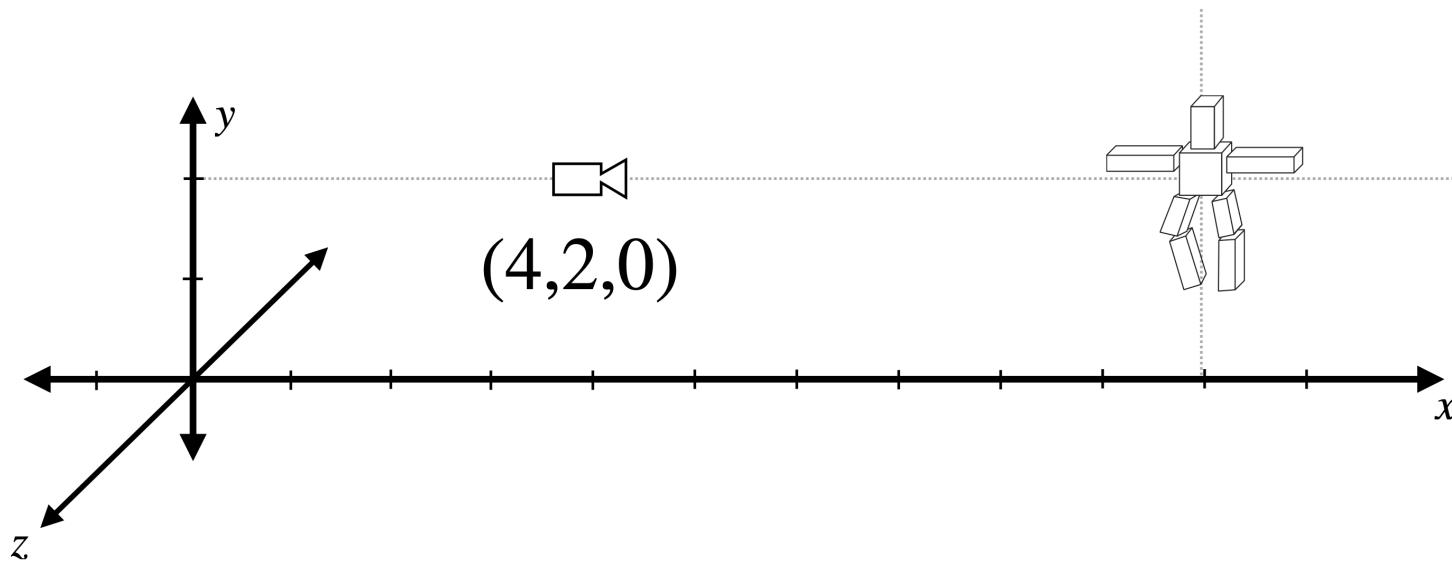
[IMAGE COORDINATES]



coordinates stretched to match image
dimensions (and flipped upside-down)

回顾：简单的相机变换

□ 考虑一个位于 $(4, 2, 0)$ ，面向 x 轴的相机，物体在世界坐标中给出

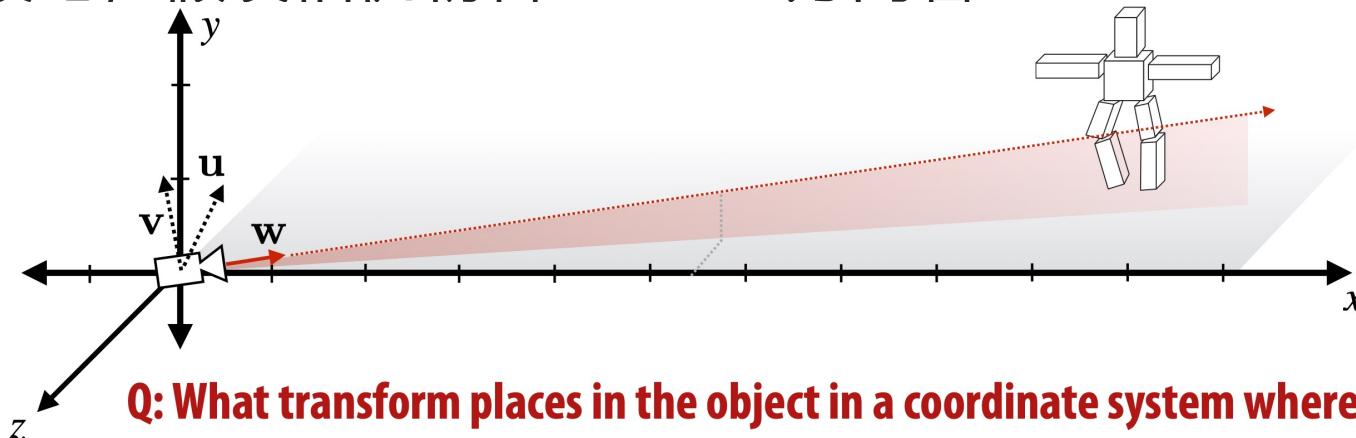


□ Q：要应用什么 3D 变换，才能使物体处于相机在原点，且面向 $-z$ 轴的（标准）坐标系中？

- 平移 $(-4, -2, 0)$ 让物体处于相对相机的位置
- 绕 y 逆时针旋转 $\pi/2$ 使物体处在新坐标系中，在该坐标系中相机的视角方向与 $-z$ 轴对齐

回顾：简单的相机变换

□ 更一般地，假设相机朝着 $w \in \mathbb{R}^3$ 方向看



Q: What transform places the object in a coordinate system where the camera is at the origin and the camera is looking directly down the -z axis?

□ 构建与 w 正交的向量 u, v

- 比如，取一个大致朝上且垂直 w 的 v ，然后 $u := w \times v$

□ 其对应的矩阵为

$$R = \begin{bmatrix} u_x & v_x & -w_x \\ u_y & v_y & -w_y \\ u_z & v_z & -w_z \end{bmatrix}$$

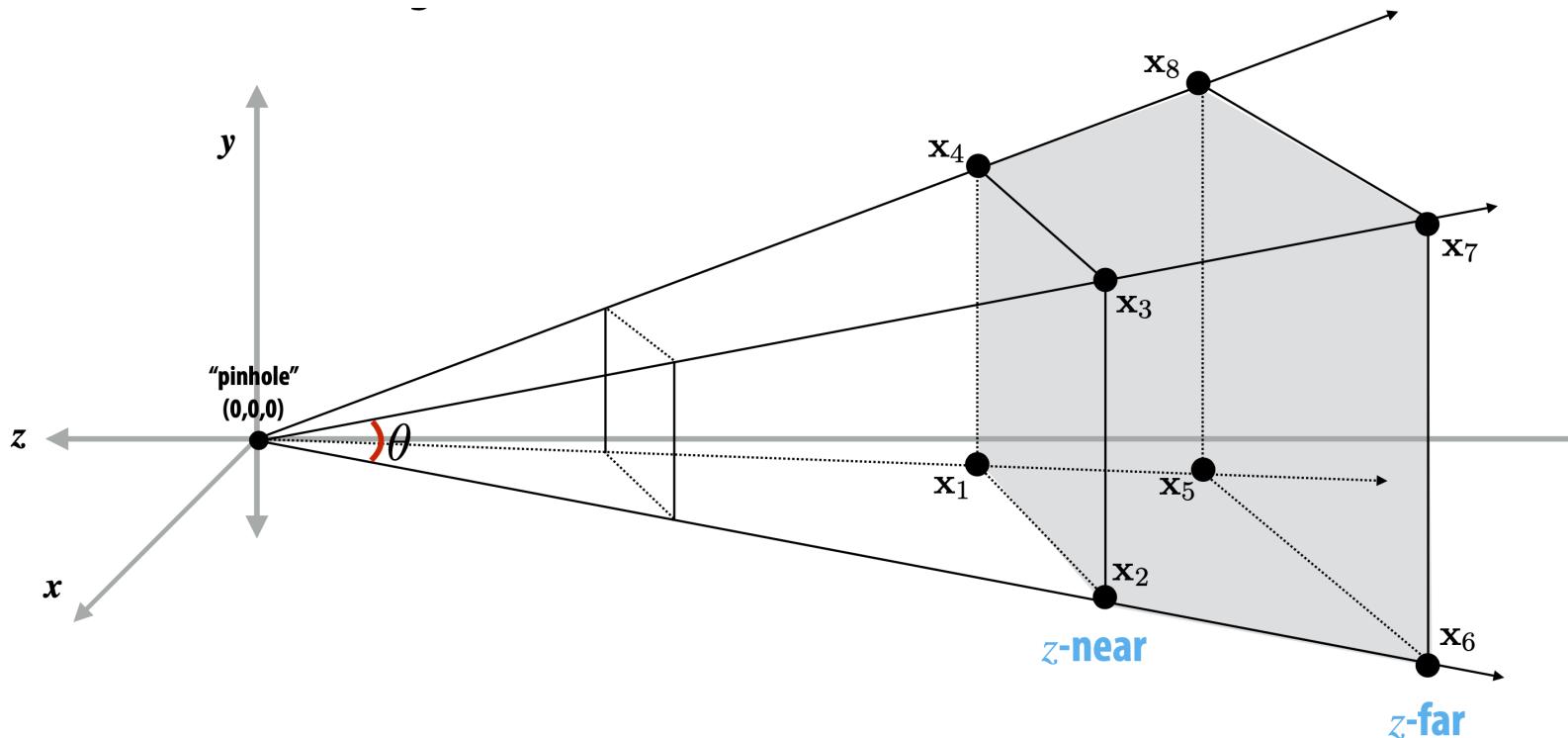
R maps x -axis to u , y -axis to v , z -axis to $-w$

Now invert (How do we do that?)

$$R^{-1} = R^T = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ -w_x & -w_y & -w_z \end{bmatrix}$$

视锥 View frustum

口视锥是另一个在光栅化中重要的概念，是相机可以看到的区域（灰色的盒子），视锥外的部分不会被画到



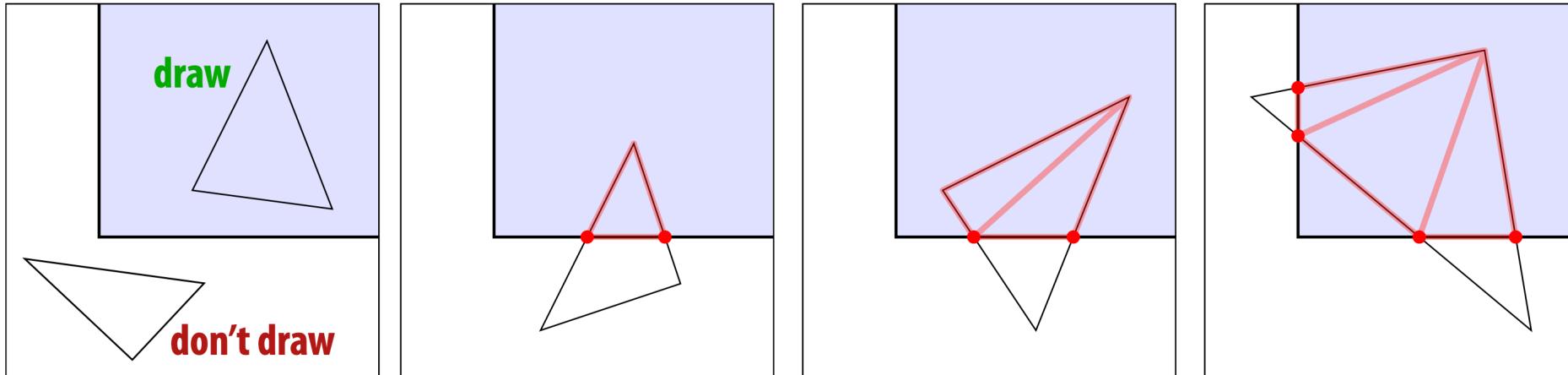
口上/下/左/右的平面对应于图像的四边

口近/远平面对应于我们想要绘制的最近/最远的距离

裁剪 Clipping

口裁剪去除相机/视锥中不可见的三角形

- 不要浪费时间光栅化你看不到的图像基元，例如三角形
- 去除看不见的像素成本高昂（“细粒度”）
- 去除整个图像基元更高效（“粗粒度”）
- 仍然需要处理部分裁剪的基元（转换成三角形）

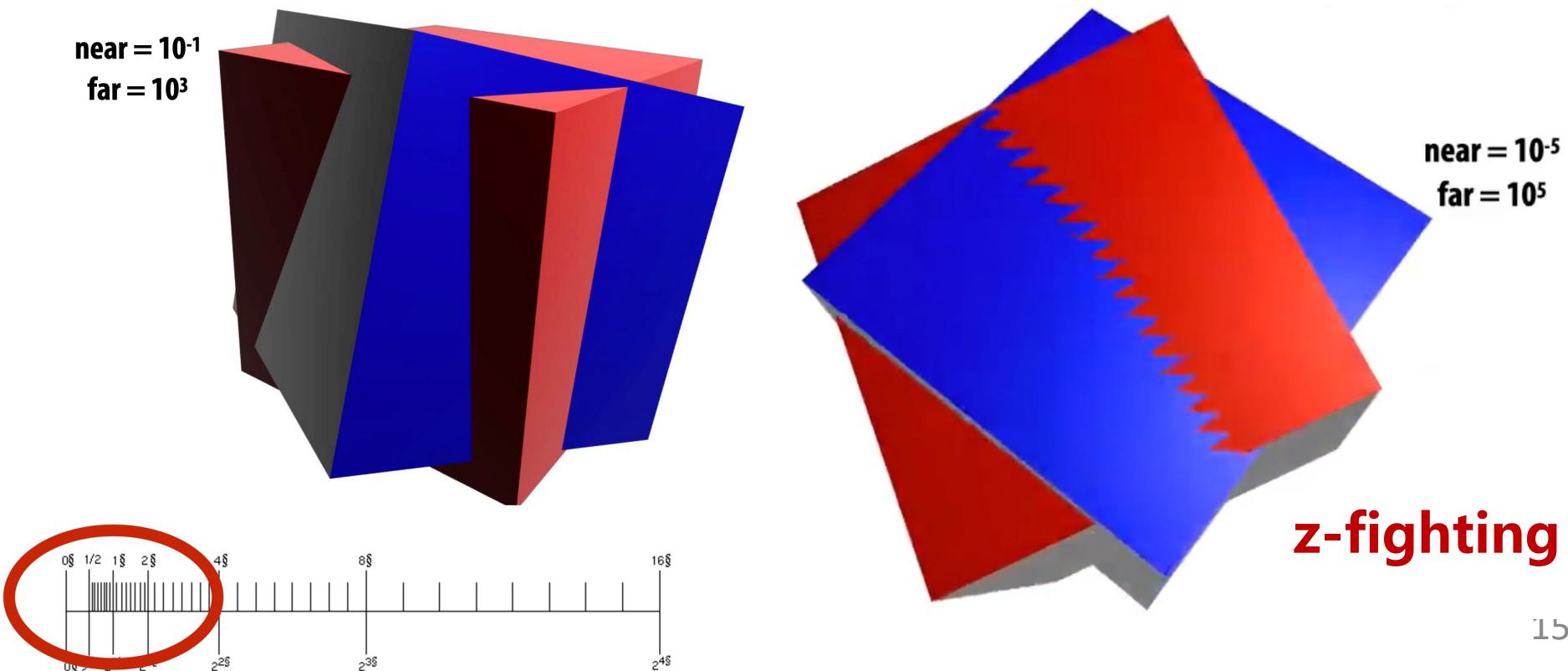


= in frustum

近/远裁剪 Near/Far Clipping

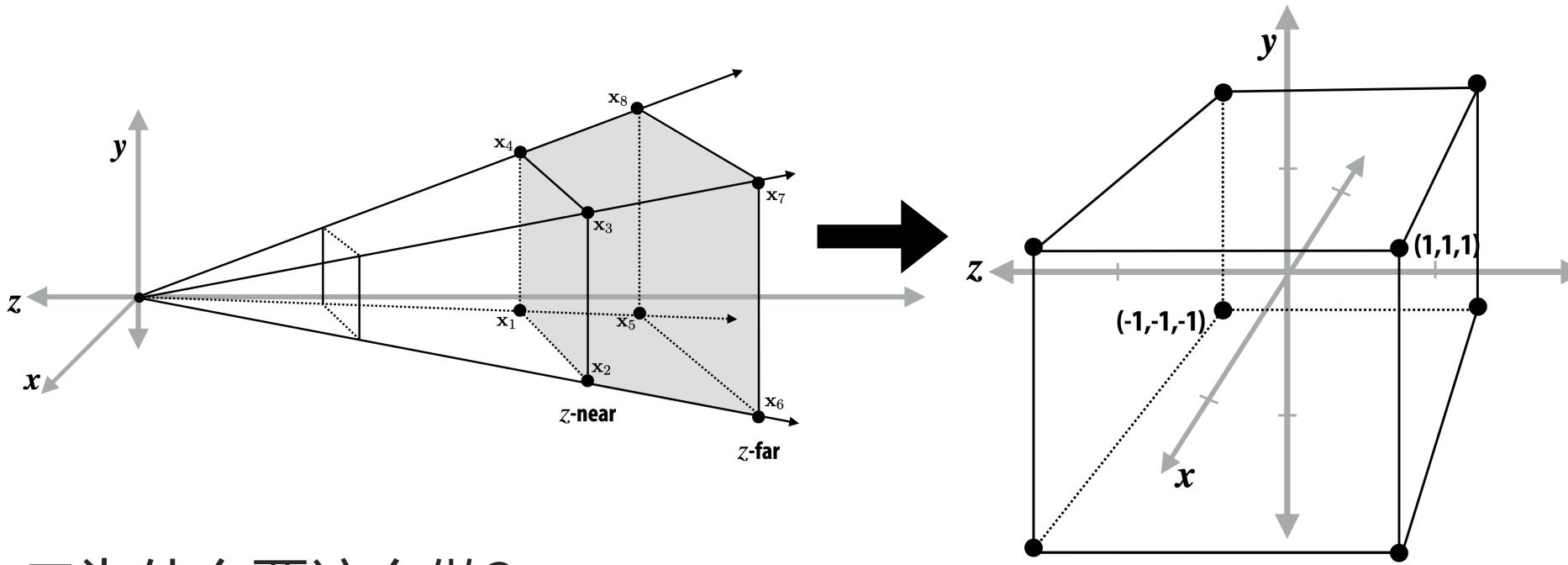
为什么需要有近/远裁剪平面？

- 太近的对象可能会失真，而太远的对象可能对观看者不可见或不相关
- 对于处理深度缓冲区 (depth buffer) 的有限精度也很重要



将视锥映射到单位立方体

在投影到 2D 之前，先将视锥投影到立方体 $[-1, 1]^3$

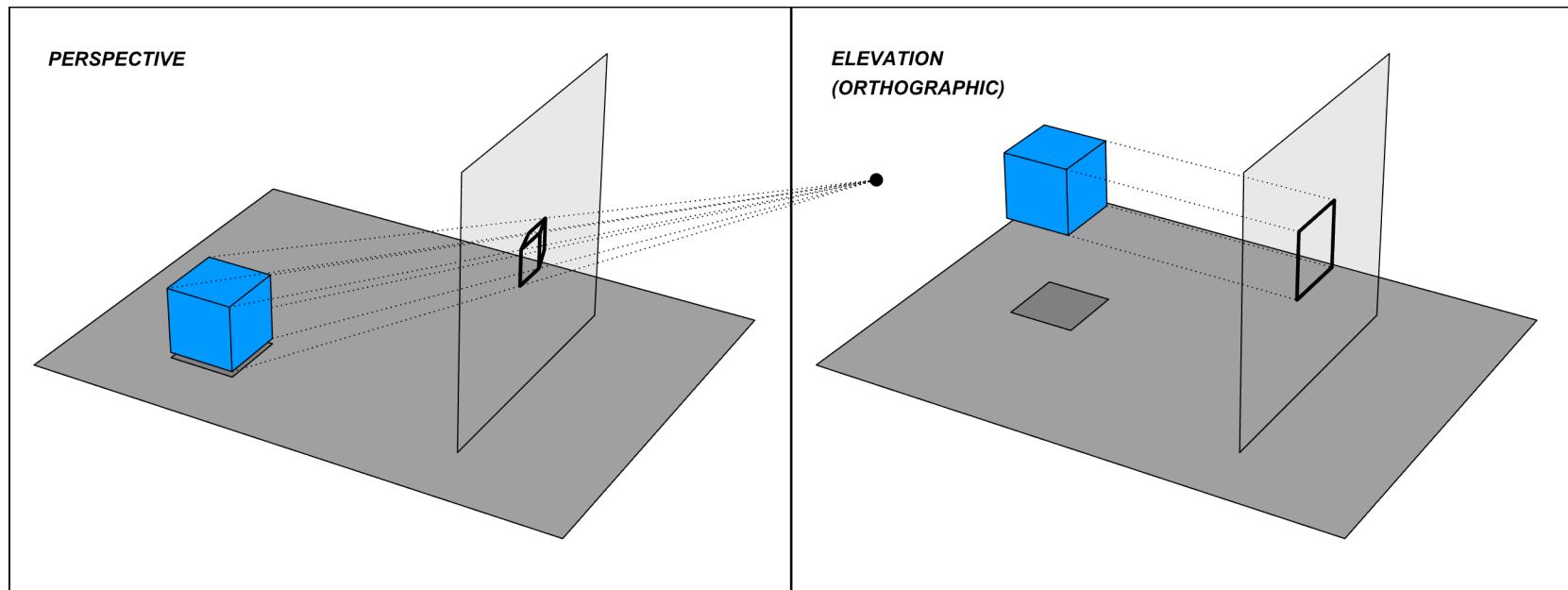


为什么要这么做？

- 简化了裁剪流程：丢弃 $[-1, 1]$ 之外的点（仍需剪裁三角形）
- 统一了不同显示设备的输入表示

透视投影与正交投影

- 口 透视投影：需要模拟真实世界的视角
- 口 正交投影：需要精确测量和不变形的视角（物体不会随距离增大而变小）

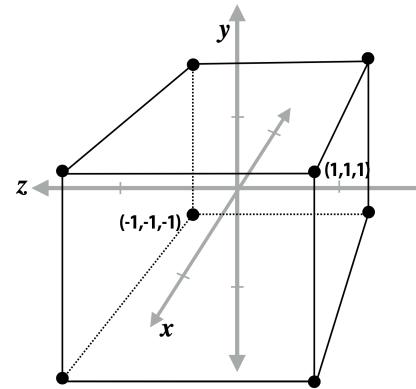
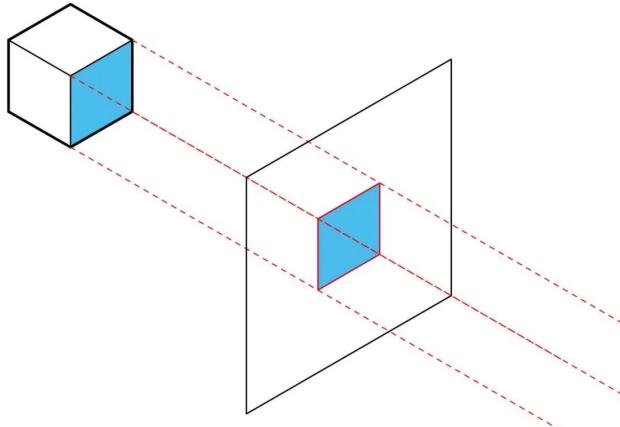


透视投影与正交投影

特征	正交投影 Orthographic Projection	透视投影 Perspective Projection
视觉效果	物体的大小与距离无关，保持不变	物体会“近大远小”，距离越远显得越小
平行线	保持平行	最终汇聚于一个消失点 (vanishing point)
类比	蓝图、建筑图纸	人眼、相机镜头
视景体	长方体 (Rectangular Box)	视锥体 (Frustum)
变换	线性变换 (缩放 + 平移)	非线性变换 (将视锥体“扭曲”成一个长方体)
应用场景	CAD、技术制图、2.5D 游戏、UI 界面	3D 游戏、模拟应用、电影

正交投影的映射矩阵

□ 在投影到 2D 之前，先将视锥投影到立方体 $[-1, 1]^3$



□ Q: 如何将此映射表达为矩阵?

□ A: 求解 $Ax_i = y_i$ 中 A 的未知变量

nonuniform scale to 2

$$A = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{l+r}{l-r} \\ 0 & \frac{2}{t-b} & 0 & \frac{b+t}{b-t} \\ 0 & 0 & \frac{2}{n-f} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translate to origin

$l = \text{left}$ $b = \text{bottom}$ $n = \text{near}$
 $r = \text{right}$ $t = \text{top}$ $f = \text{far}$

$\mathbf{x}_1 = \{l, b, n, 1\}$	$\mathbf{y}_1 = \{-1, -1, 1, 1\}$
$\mathbf{x}_2 = \{r, b, n, 1\}$	$\mathbf{y}_2 = \{1, -1, 1, 1\}$
$\mathbf{x}_3 = \{r, t, n, 1\}$	$\mathbf{y}_3 = \{1, 1, 1, 1\}$
$\mathbf{x}_4 = \{l, t, n, 1\}$	$\mathbf{y}_4 = \{-1, 1, 1, 1\}$
$\mathbf{x}_5 = \{l, b, f, 1\}$	$\mathbf{y}_5 = \{-1, -1, -1, 1\}$
$\mathbf{x}_6 = \{r, b, f, 1\}$	$\mathbf{y}_6 = \{1, -1, -1, 1\}$
$\mathbf{x}_7 = \{r, t, f, 1\}$	$\mathbf{y}_7 = \{1, 1, -1, 1\}$
$\mathbf{x}_8 = \{l, t, f, 1\}$	$\mathbf{y}_8 = \{-1, 1, -1, 1\}$

(orthographic projection)

透视投影的映射矩阵

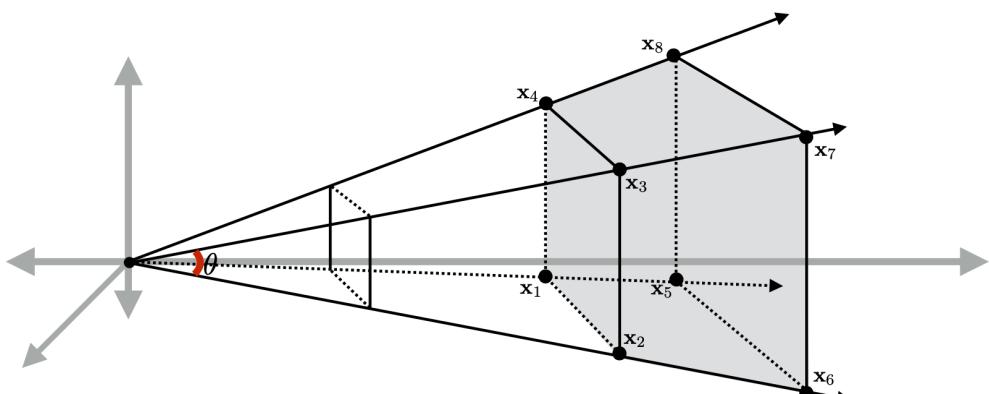
□回顾一下我们的基本透视投影矩阵

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} \rightarrow$$

objects shrink in distance

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

□透视矩阵考虑了视锥的几何形状



$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$l = \text{left}$

$r = \text{right}$

$b = \text{bottom}$

$t = \text{top}$

$n = \text{near}$

$f = \text{far}$

回顾：屏幕转换 Screen transformation

口光栅化中的最后一个变换：从 2D 观察平面到像素坐标

口将 $z = 1$ 平面上 $[-1,1] \times [-1,1]$ 中的点投影到 $W \times H$ 大小
到屏幕上

“normalized device coordinates”

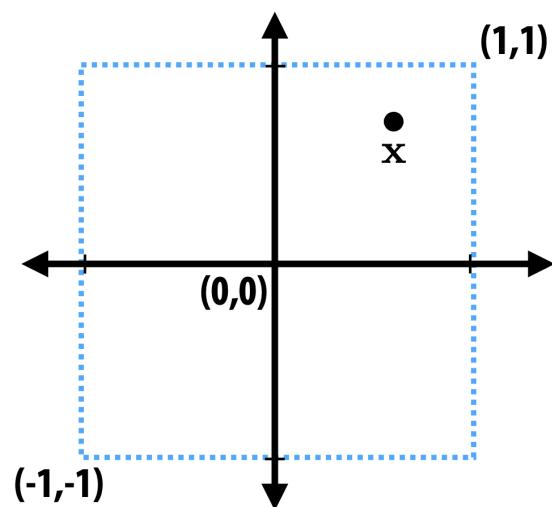
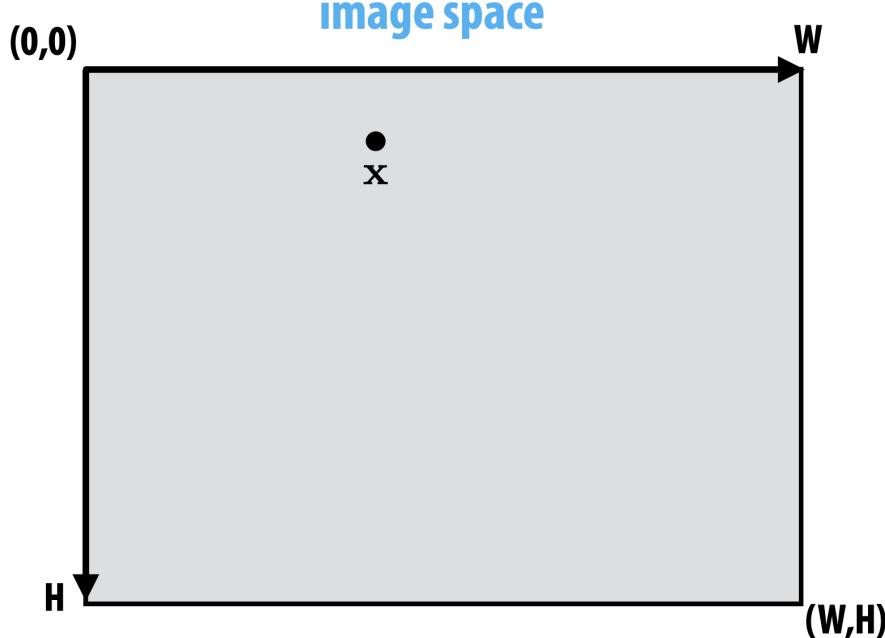


image space

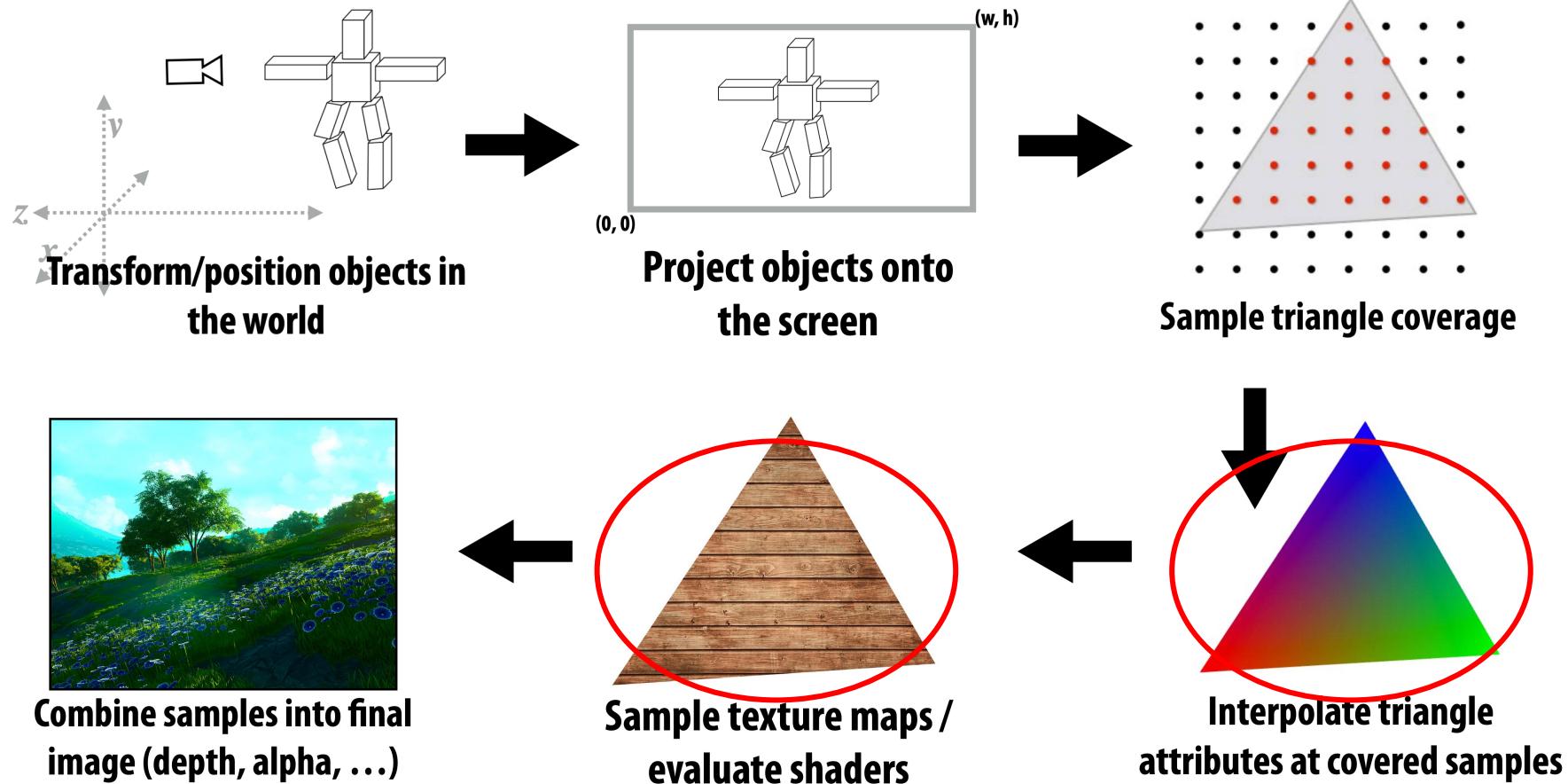


Step 1: reflect about x-axis

Step 2: translate by $(1,1)$

Step 3: scale by $(W/2, H/2)$

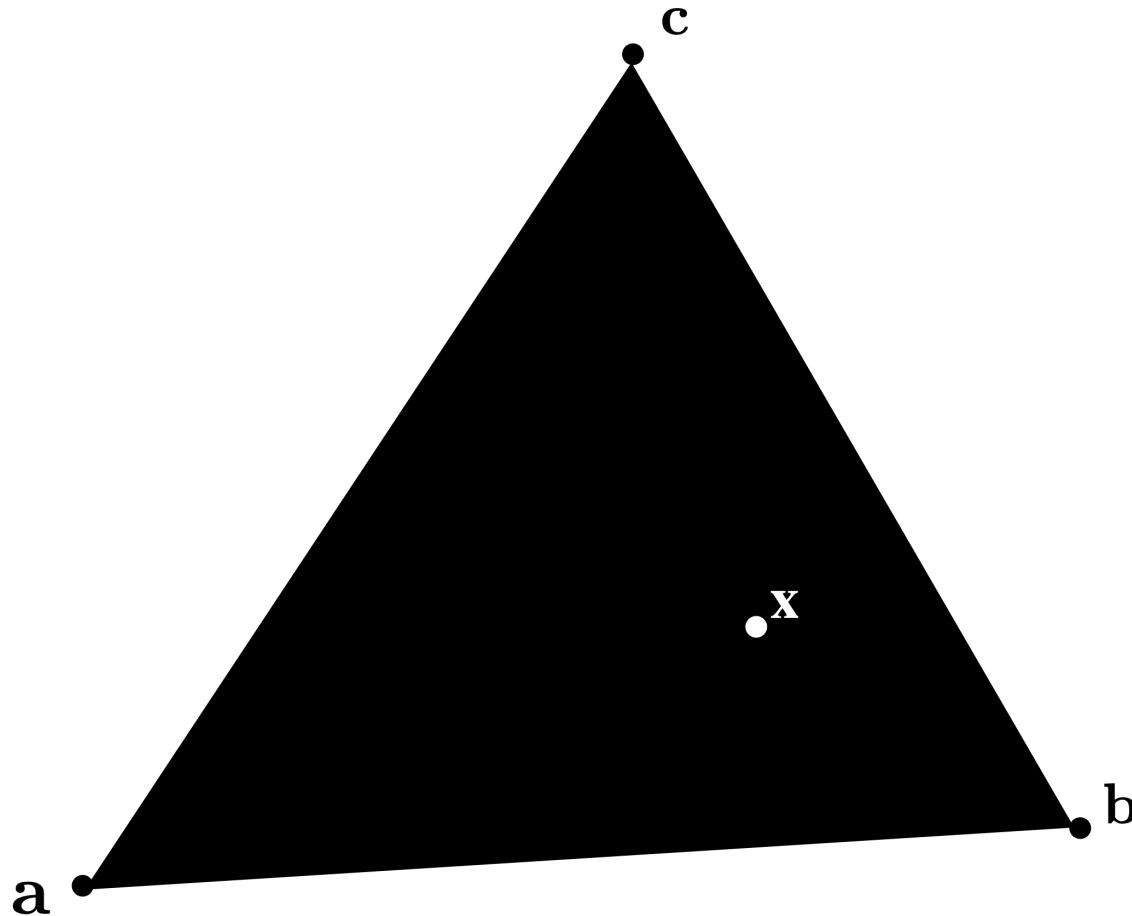
光栅化流程中的步骤



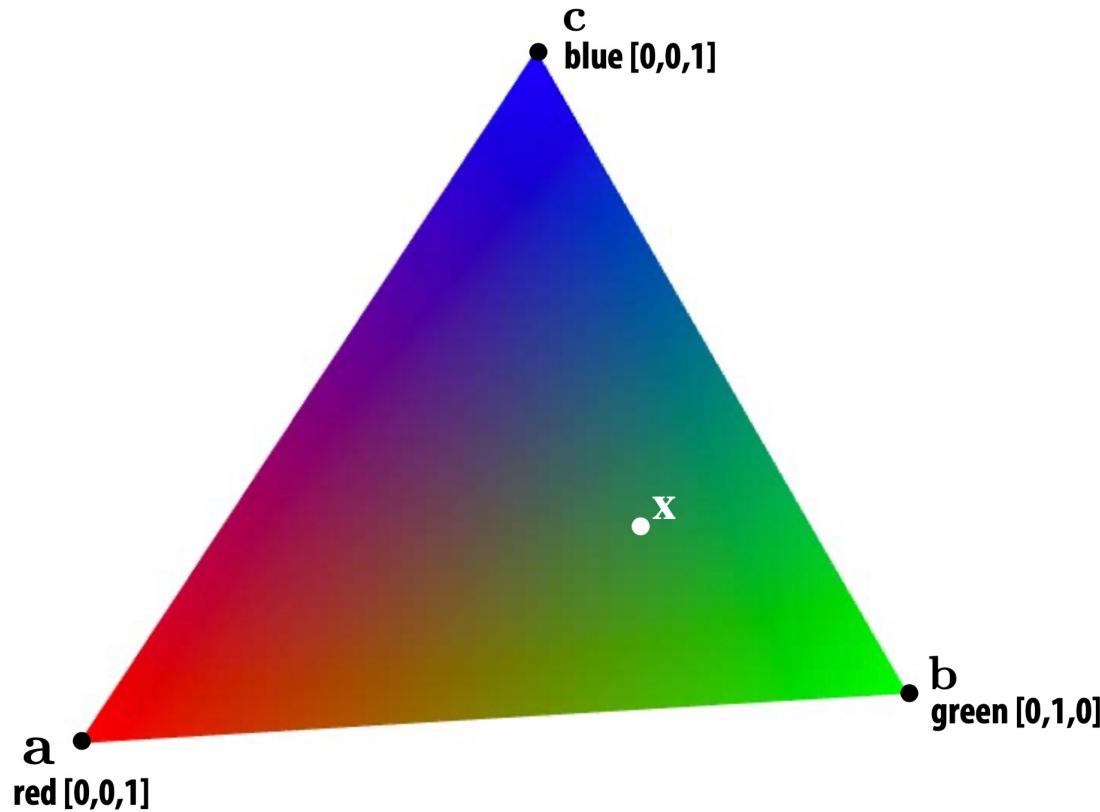
我们如何绘制漂亮的图像基元？

Coverage(x, y)

口前面讨论了如何在给定三角形顶点的
2D 位置的情况下对覆盖率进行采样



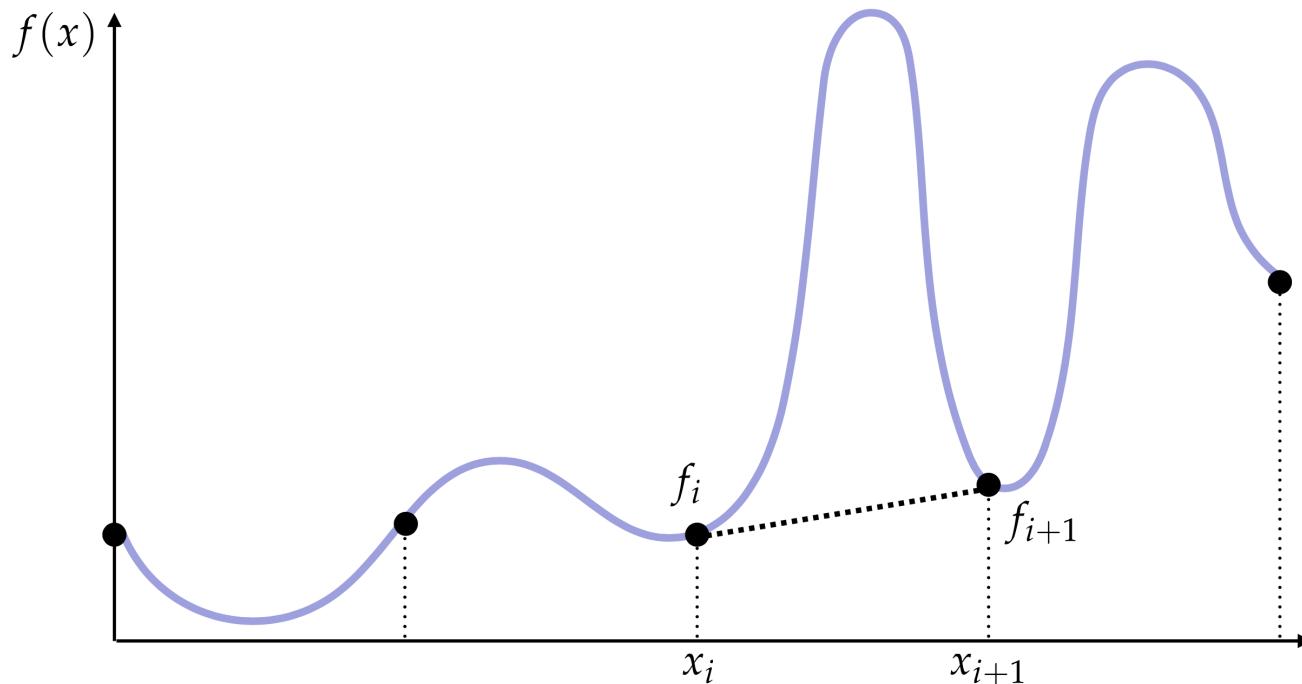
考虑采样 $\text{color}(x, y)$



- 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

1D 中的线性插值

- 假设我们已经在点 x_i 对函数 $f(x)$ 的值进行了采样，即 $f_i := f(x_i)$
- Q: 我们如何构造连接 x_i 和 x_{i+1} 之间的函数？

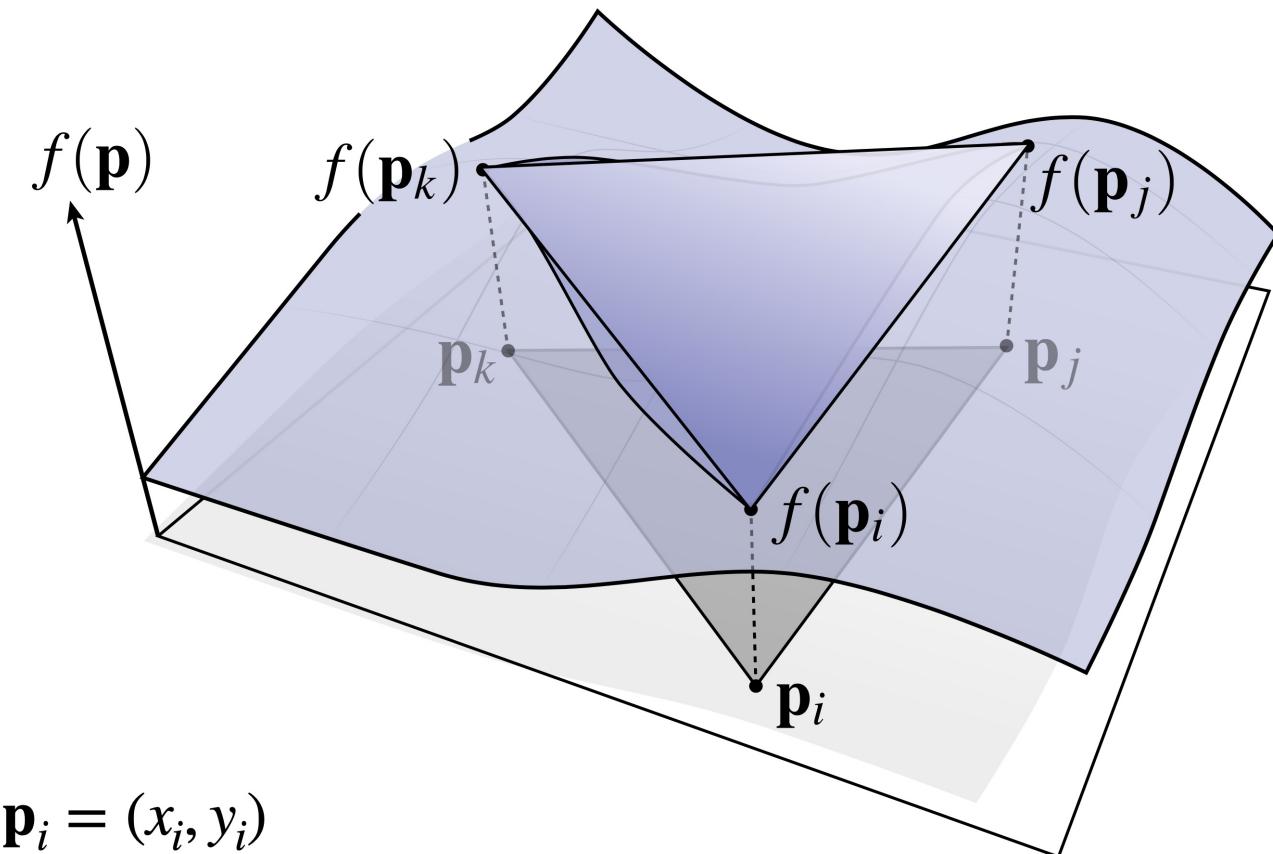


$$t := (x - x_i) / (x_{i+1} - x_i) \in [0, 1]$$

$$\hat{f}(t) = f_i + t(f_{i+1} - f_i) = (1 - t)f_i + tf_{i+1}$$

2D 中的线性插值

- 假设我们同样已经采样了 2D 中函数 $f(\mathbf{p})$ 在点 $\mathbf{p}_i, \mathbf{p}_j, \mathbf{p}_k$ 的值
- Q: 我们如何把点连起来? 即如何构造平面?



2D 中的线性插值

口如何用线性 (更准确地说, 是仿射) 函数拟合这三个值?

口任何这样的函数都有三个未知系数 a, b, c

$$\hat{f}(x, y) = ax + by + c$$

口要进行插值, 我们需要找到正确的系数, 使函数与采样点的采样值相匹配

$$\hat{f}(x_n, y_n) = f_n, n \in \{i, j, k\}$$

口在三个未知数中得到三个线性方程

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \frac{1}{(x_j y_i - x_i y_j) + (x_k y_j - x_j y_k) + (x_i y_k - x_k y_i)} \begin{bmatrix} f_i(y_k - y_j) + f_j(y_i - y_k) + f_k(y_j - y_i) \\ f_i(x_j - x_k) + f_j(x_k - x_i) + f_k(x_i - x_j) \\ f_i(x_k y_j - x_j y_k) + f_j(x_i y_k - x_k y_i) + f_k(x_j y_i - x_i y_j) \end{bmatrix}$$

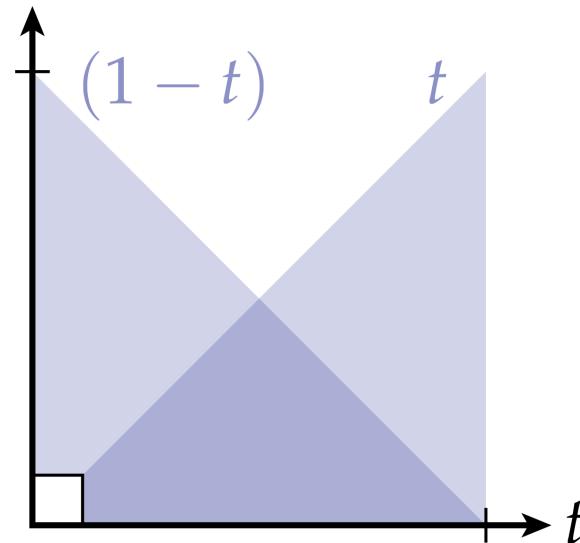
This is ugly. There has to be a better way to think about this...

回顾 1D 中的线性插值

□ 让我们思考一下我们是如何在 1D 中进行线性插值的

$$\hat{f}(t) = (1 - t)f_i + tf_j$$

□ 可以将其视为两个(基)函数 $(1 - t)$ and t 的线性组合



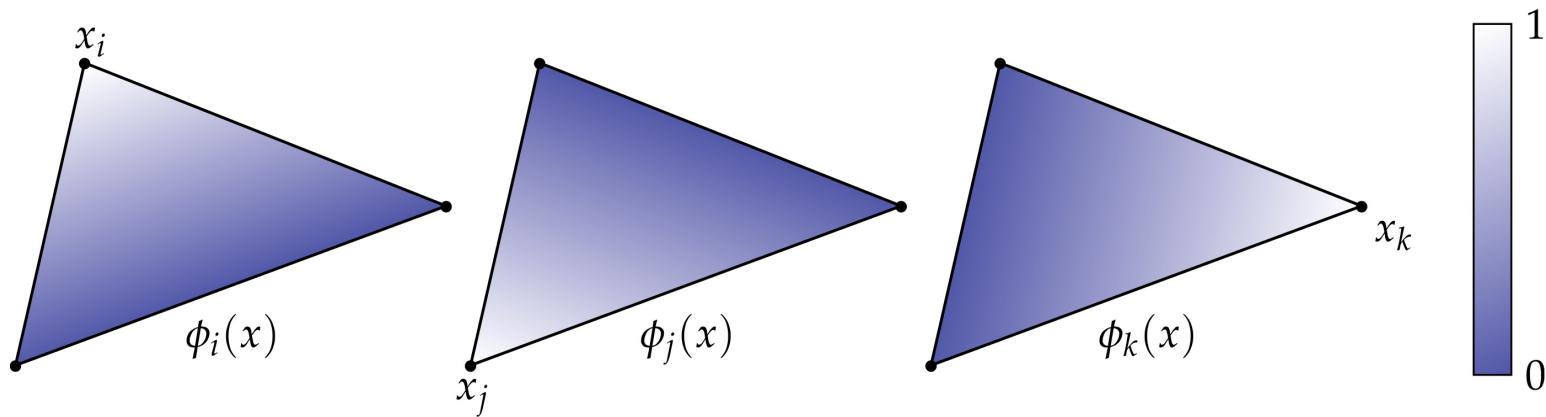
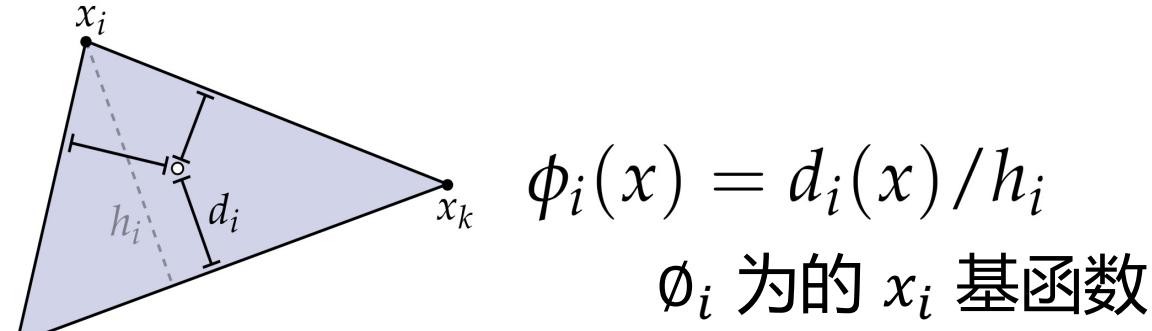
□ 当我们向 $t = 0$ 移动时，我们靠近 f 在 x_i 的值

□ 当我们向 $t = 1$ 移动时，我们靠近 f 在 x_j 的值

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:

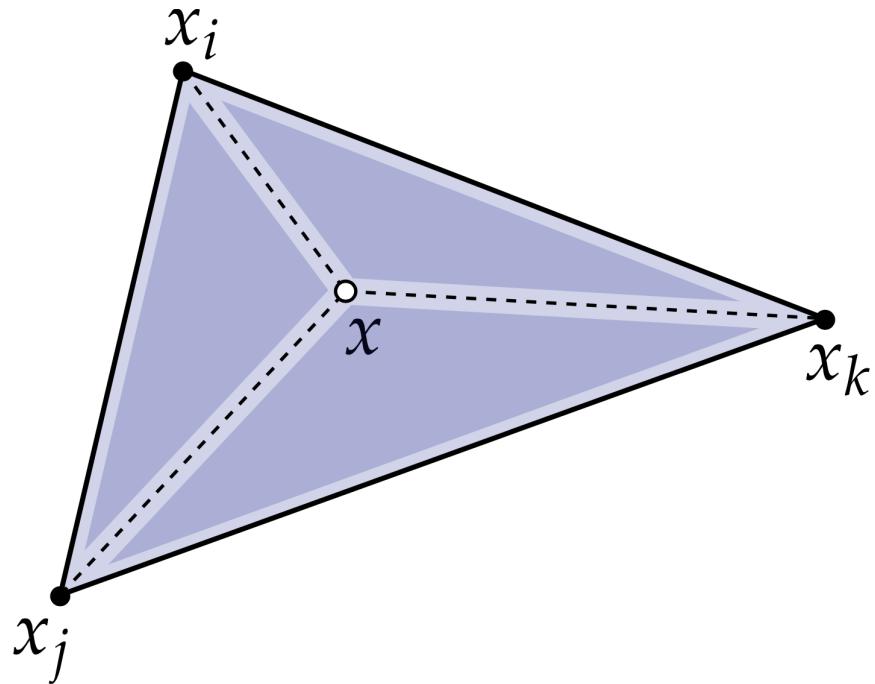


□ 通过线性组合得到插值函数: $\hat{f}(x) = f_i\phi_i + f_j\phi_j + f_k\phi_k$

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



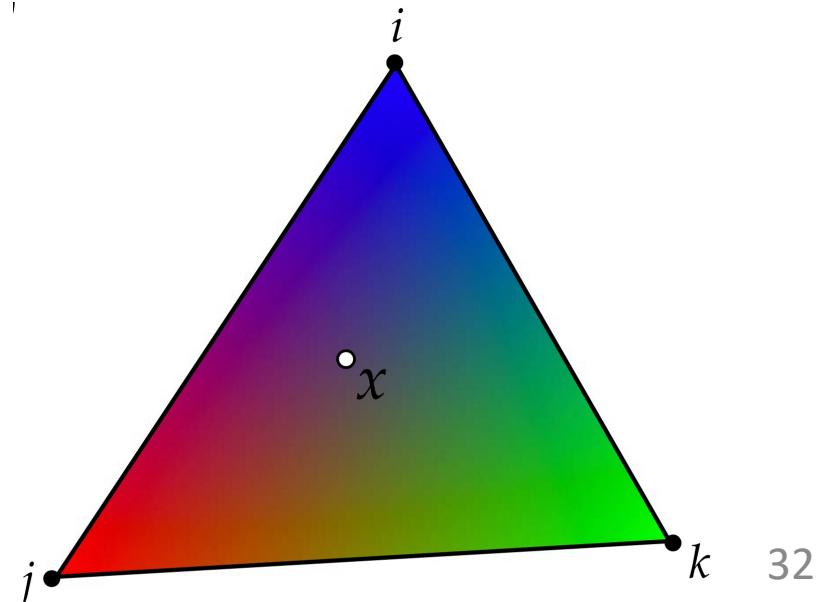
$$\phi_i(x) = \frac{\text{area}(x, x_j, x_k)}{\text{area}(x_i, x_j, x_k)}$$

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)

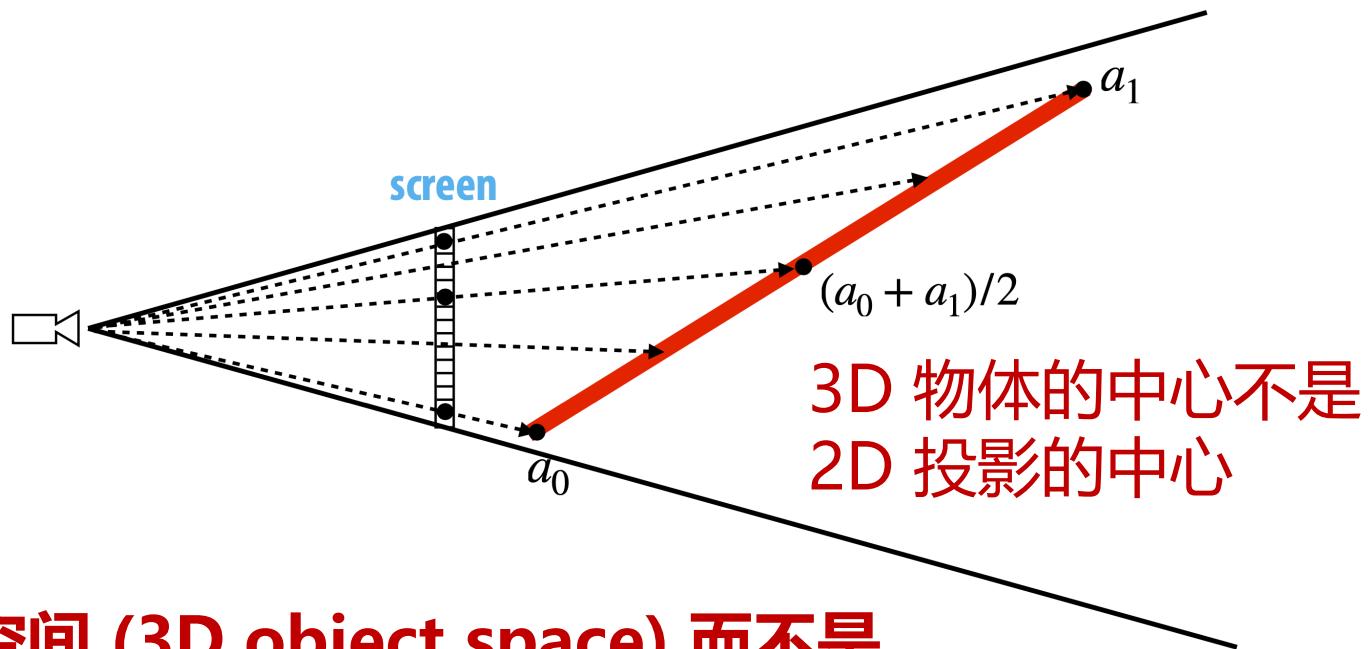
$$\begin{aligned}color(x) = & color(x_i)\phi_i + \\& color(x_j)\phi_j + color(x_k)\phi_k\end{aligned}$$



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

透视不正确的插值 Perspective-incorrect interpolation

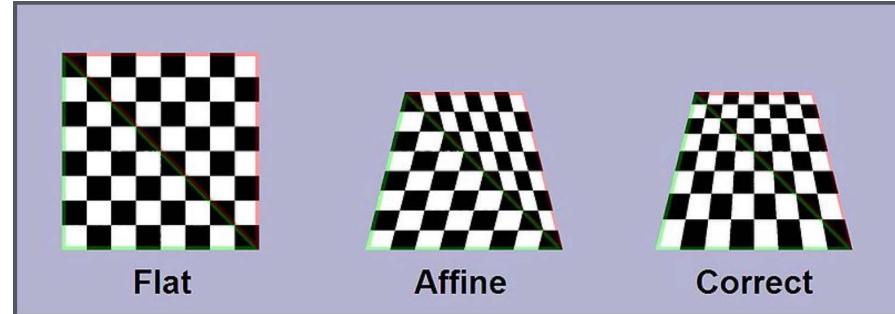
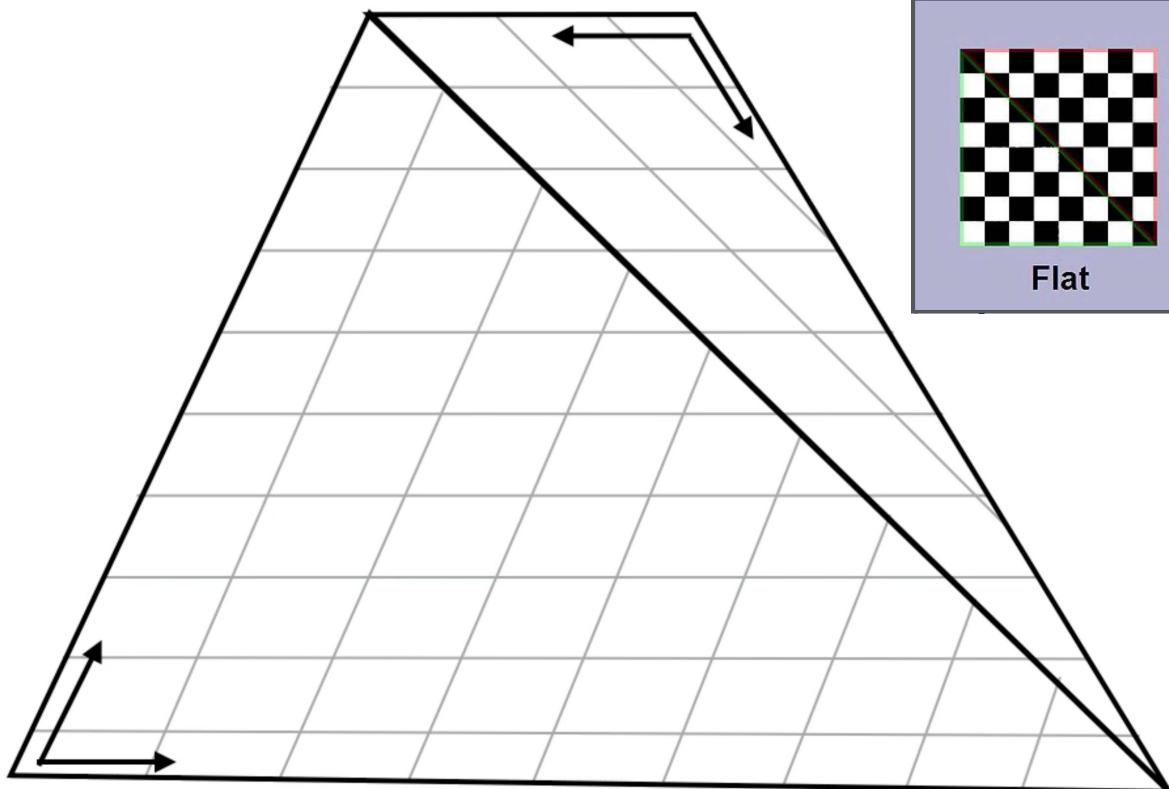
- 口 透视投影涉及齐次除法 (homogeneous divide) , 是非线性的 (除以 z), 意味着它可以扭曲线性或仿射特性
- 口 当三角形的顶点都位于同一平面上, 重心插值是屏幕 XY 坐标的仿射函数, 若三角形不在同一平面, 则这种仿射特性会被打破



应该在三维物体空间 (3D object space) 而不是 图像空间 (image space) 进行属性值的线性插值

透视不正确的插值 – 例子

□ 考虑将四边形拆分为两个三角形



如果我们在 2D (投影) 坐标上独立计算两个三角形的重心坐标，会导致插值中的 (导数) 不连续性，四边形被分割

透视正确的插值

□ 目标：在顶点对属性 ϕ 进行差值

□ 步骤：

- 计算每个顶点的深度 (depth): z
- 在每个顶点处计算 $Z := 1/z$ 和 $P := \phi/z$
- 使用标准 (2D) 重心坐标插值 Z 和 P
- 在每个像素上，将插值的 P 除以插值的 Z ，得到最终值

相当于在 3D 空间中做插值，然后再投影到 2D



For a derivation, see Low, “Perspective-correct interpolation”

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)



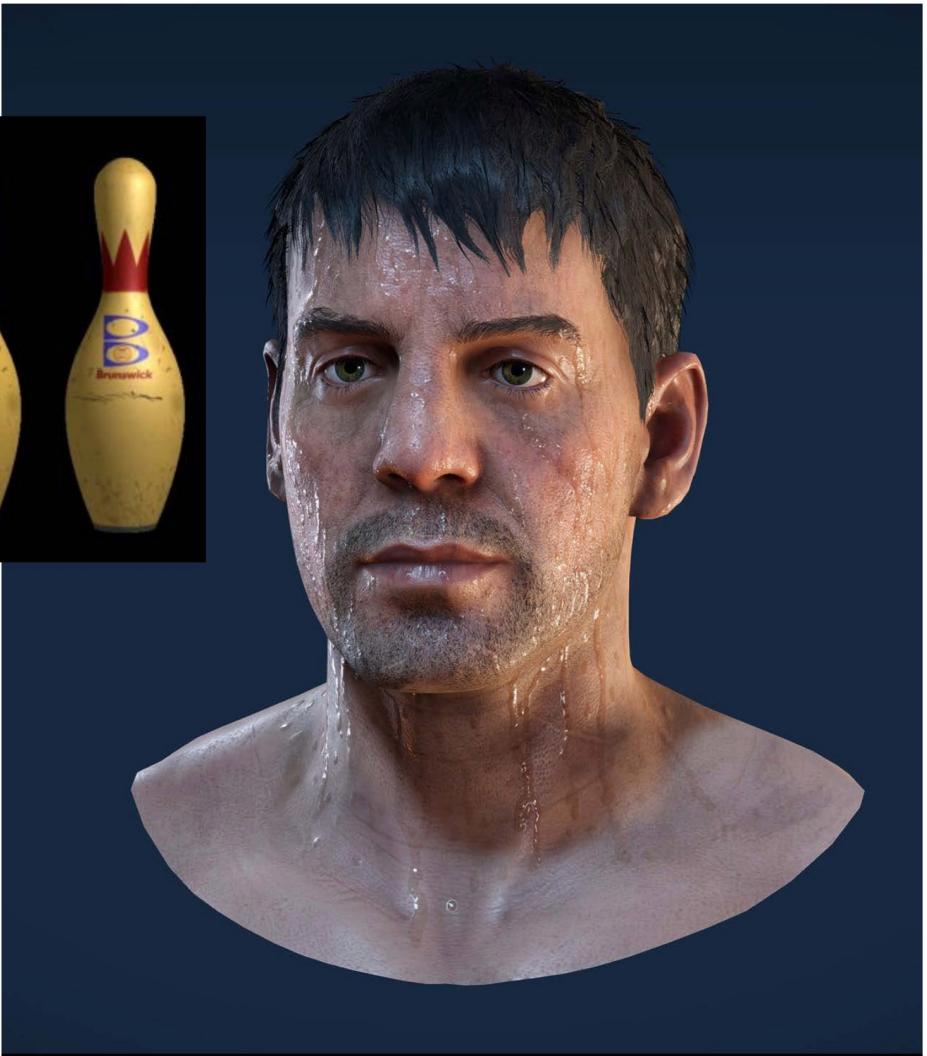
Pattern on ball

Wood grain on floor

描述表面材料特性



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



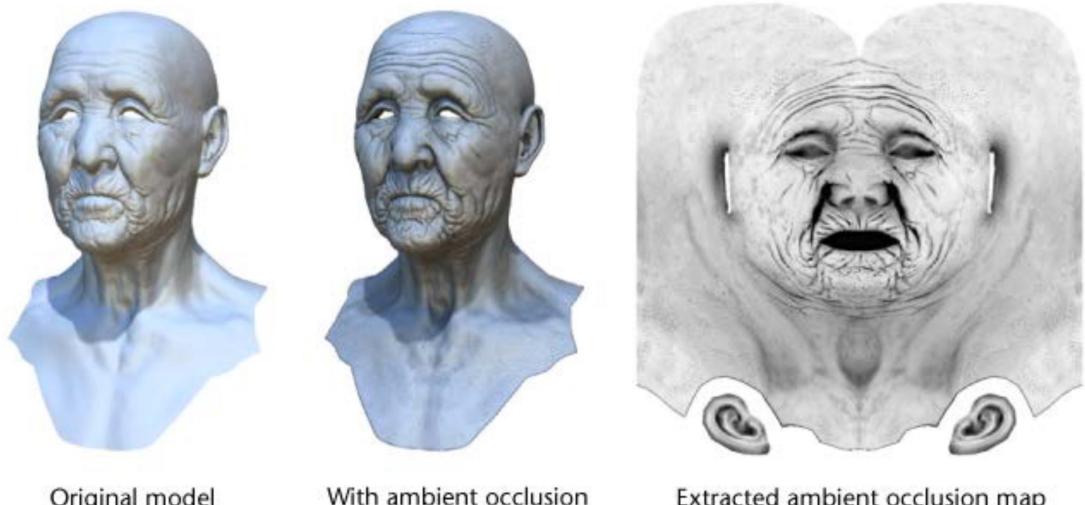
(C)2013 CRYTEK GMBH. ALL RIGHTS RESERVED. RYSE IS A REGISTERED TRADEMARK OF CRYTEK GMBH

RYSE
SON OF ROME

计算照明和阴影效果

计算环境遮挡 (ambient occlusion)

用于模拟由于物体表面之间的遮挡而导致的一种自然光照效果



Grace Cathedral environment map



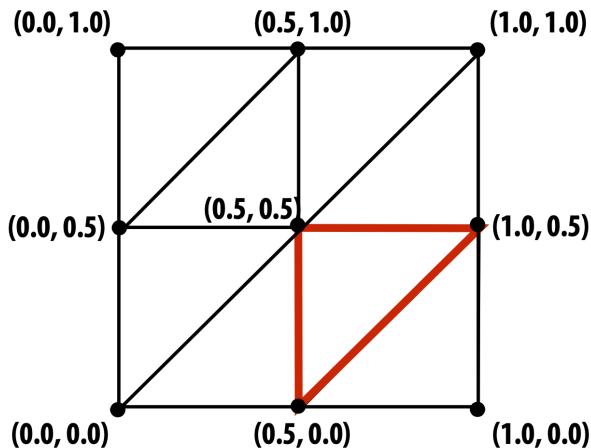
Environment map used in rendering

提前计算
光照效果

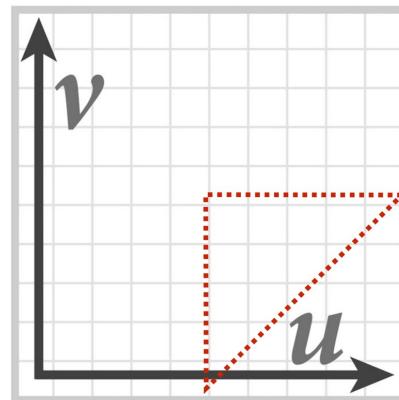
纹理坐标

- “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)

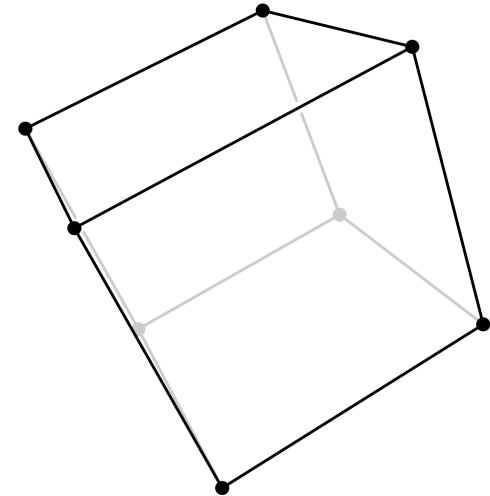


在域 $[0,1]^2$ 中的纹理可以被一个 2048×2048 的图像指定

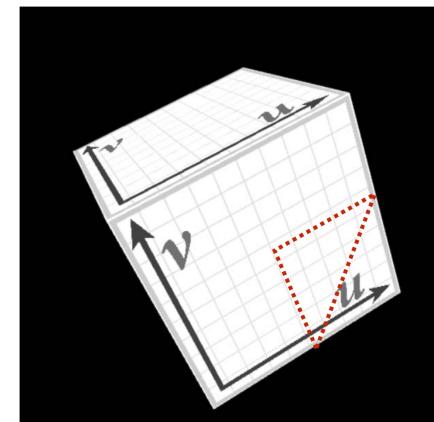


红色表示高亮三角形的位置

example: texture this cube

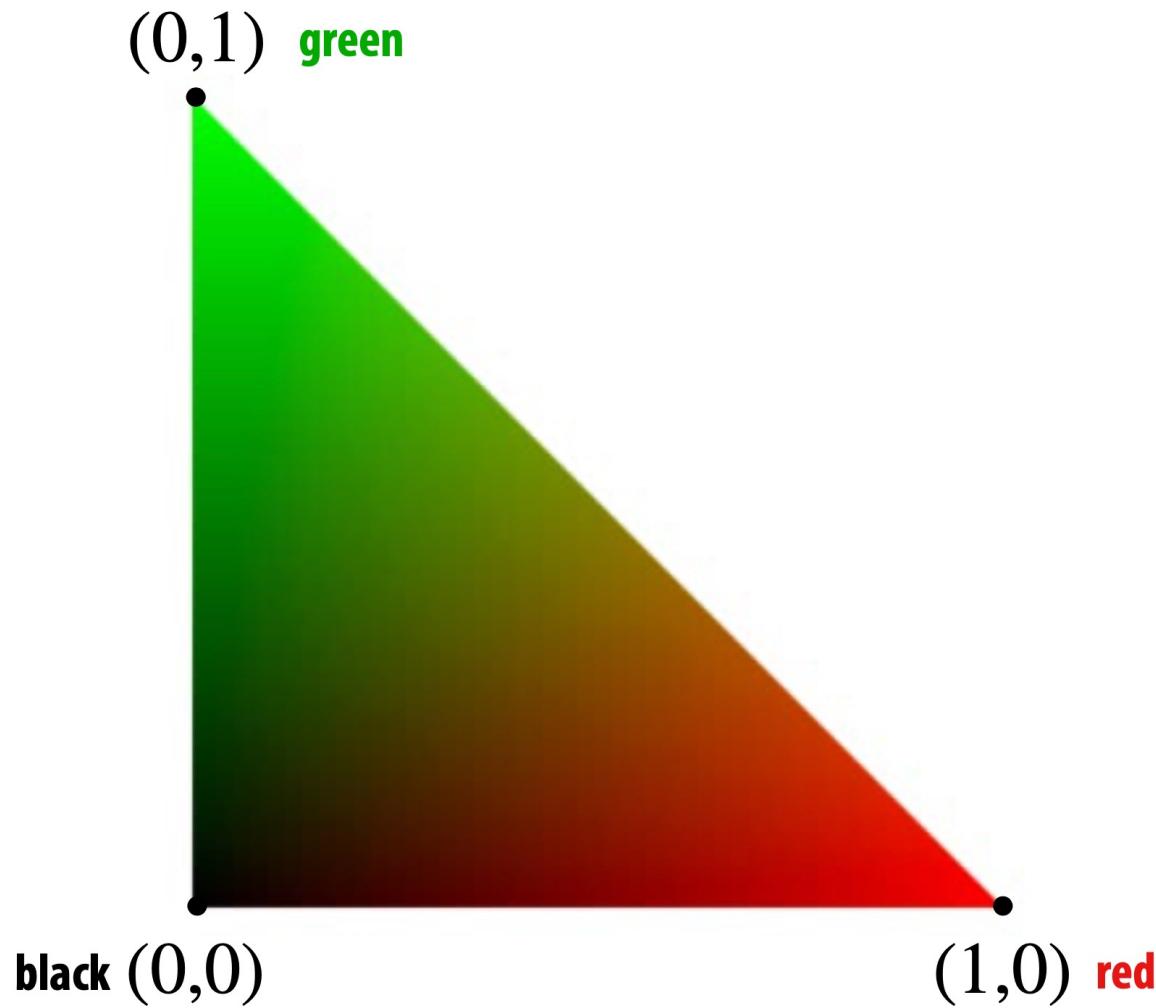


线性插值纹理坐标 & 在纹理中“查找”颜色可得到此图像



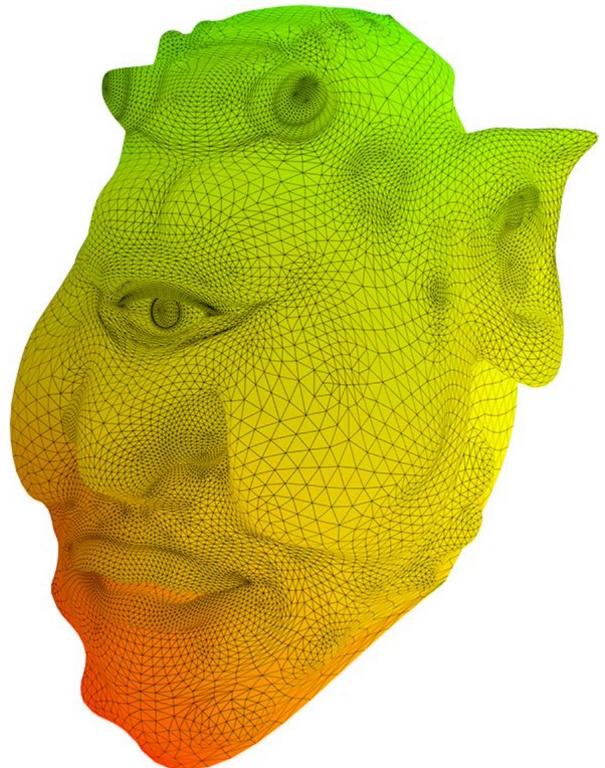
纹理坐标可视化

口将纹理坐标 (u, v) 与颜色关联有助于可视化映射

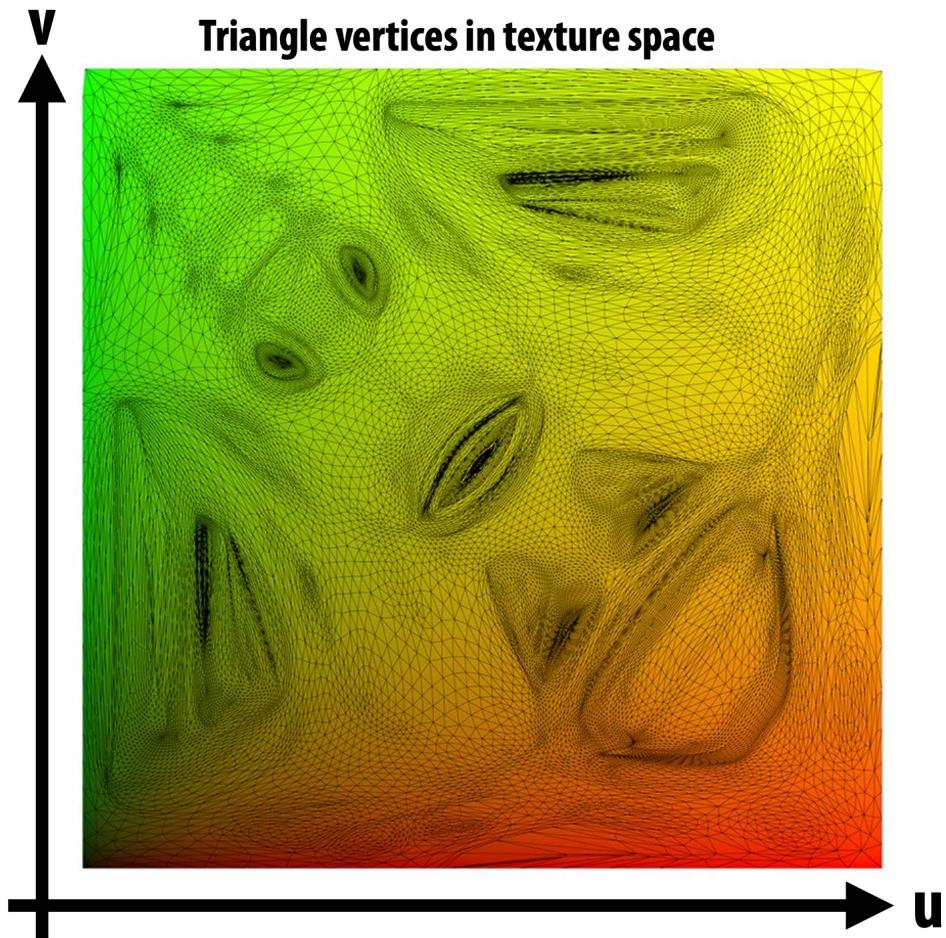


更复杂的映射

Visualization of texture coordinates

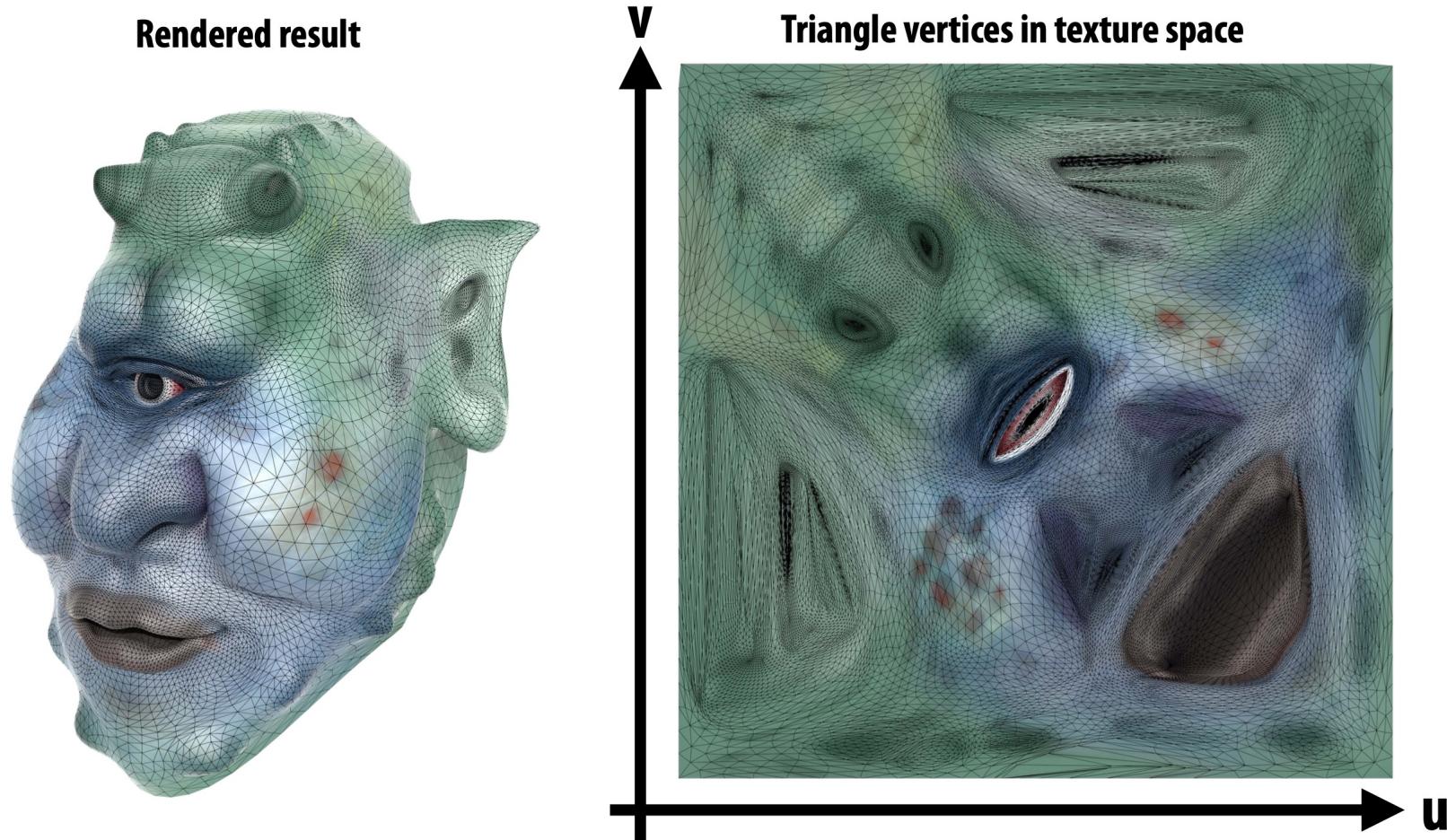


Triangle vertices in texture space



- 每个顶点在纹理空间中都有一个坐标 (u, v)
- 如何得出这些坐标是另一回事！

纹理贴图添加细节



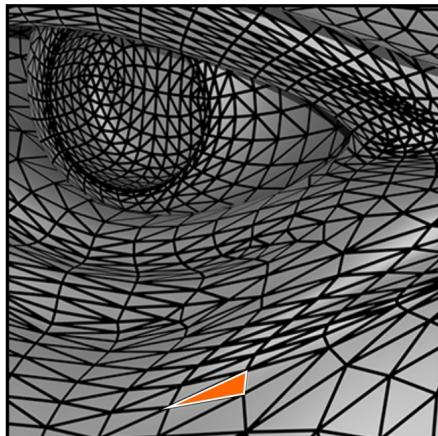
□ 每个三角形都将图像的一部分“复制”回曲面

纹理贴图添加细节

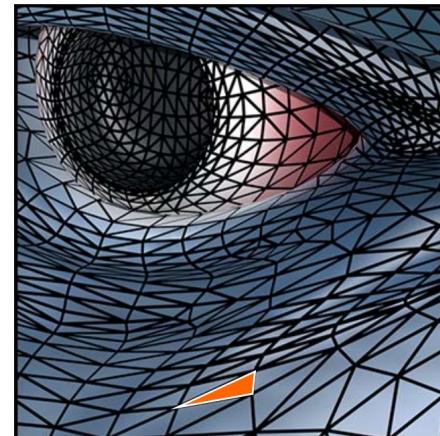
rendering without texture



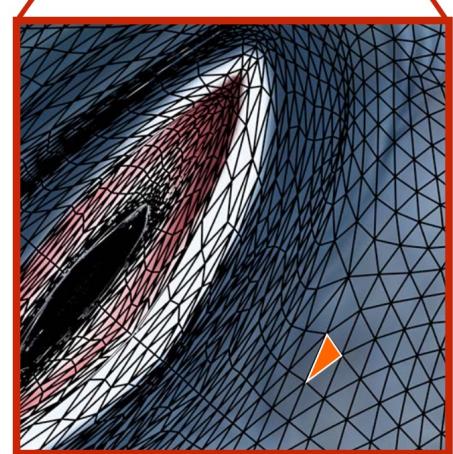
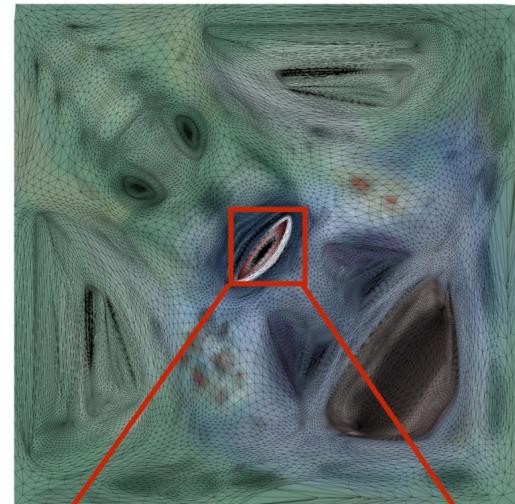
zoom



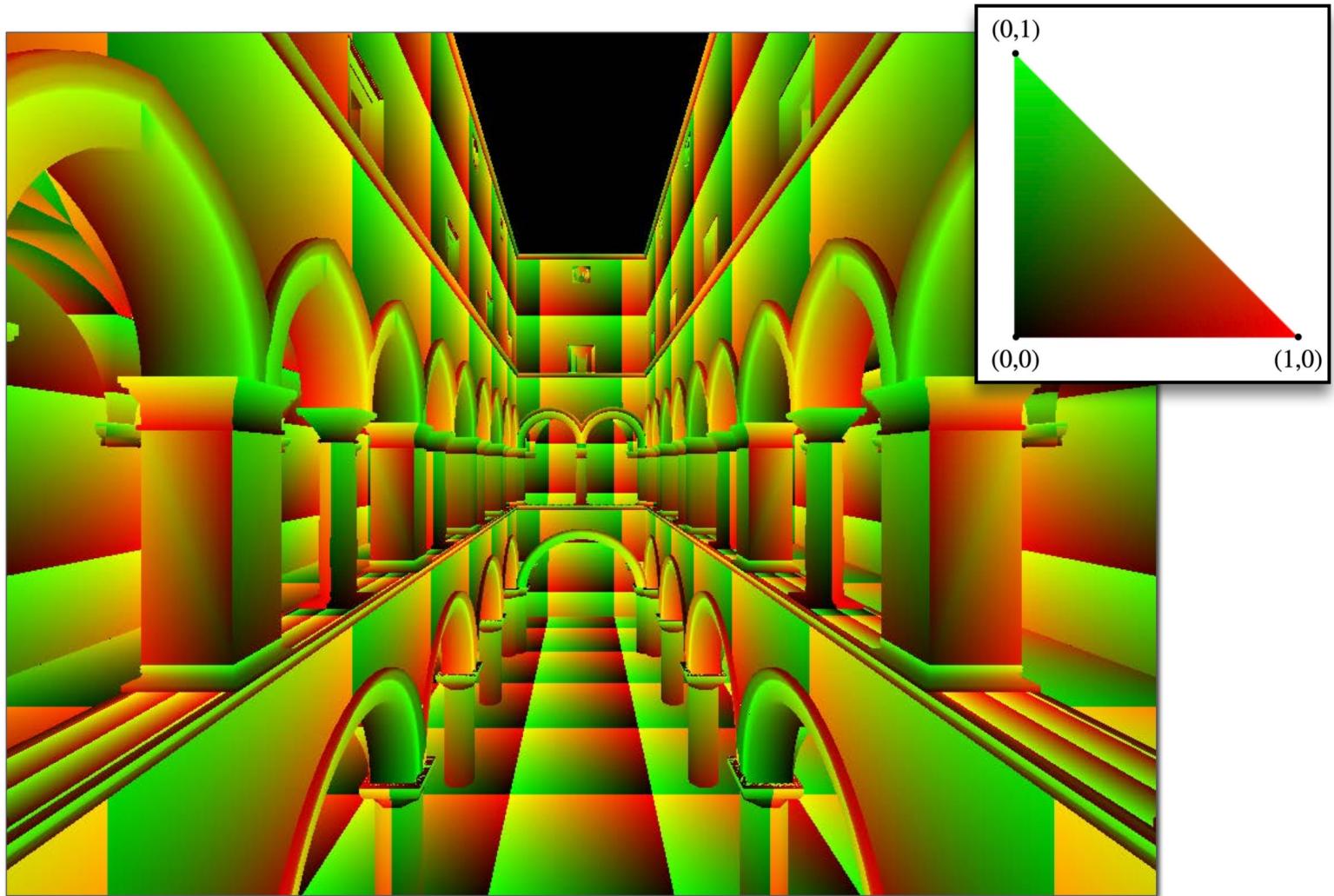
rendering with texture



texture image

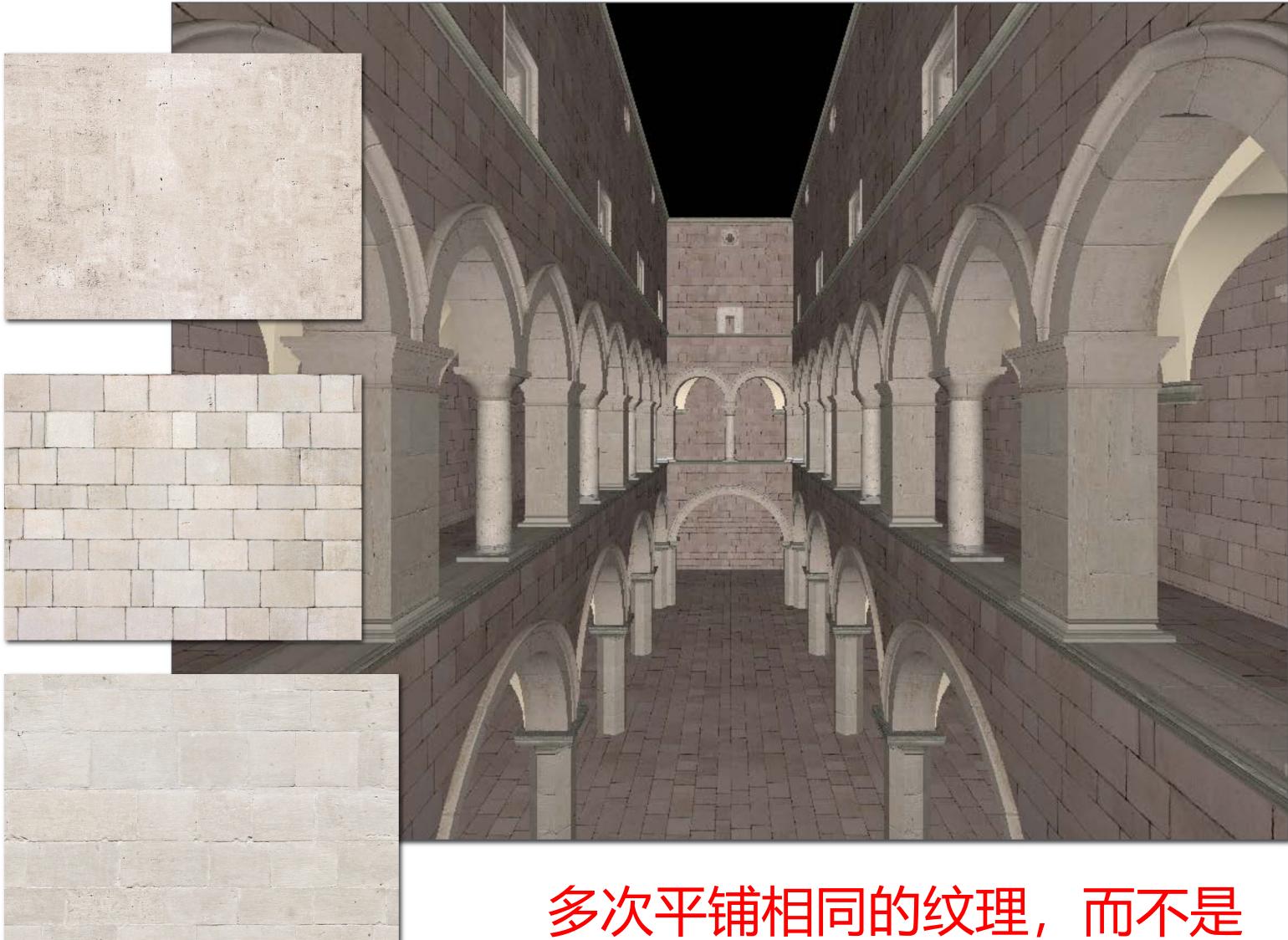


另一个例子：周期坐标



Q: 什么情况下纹理坐标会在表面上重复?

纹理海绵 Textured Sponza



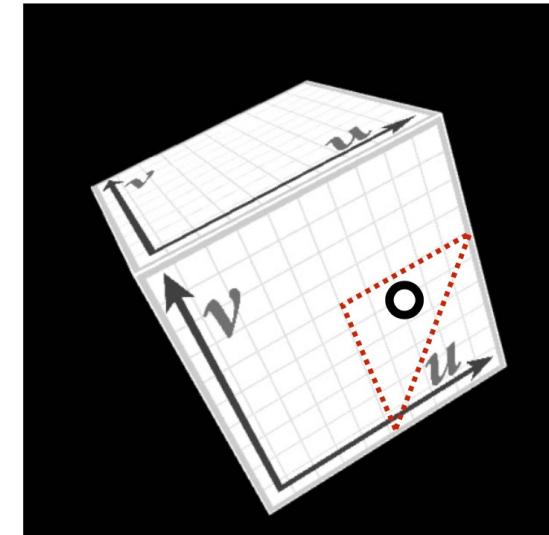
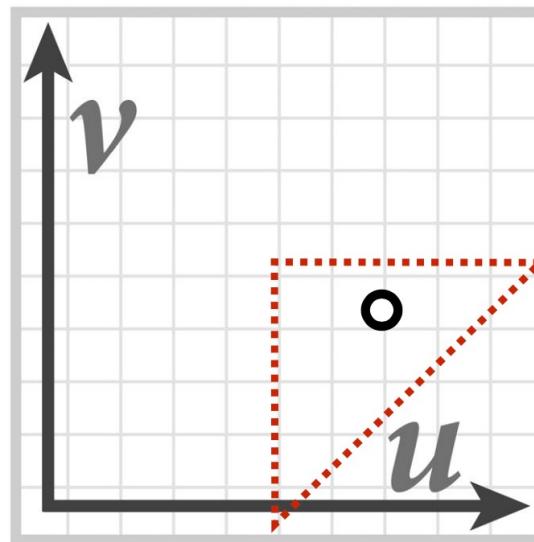
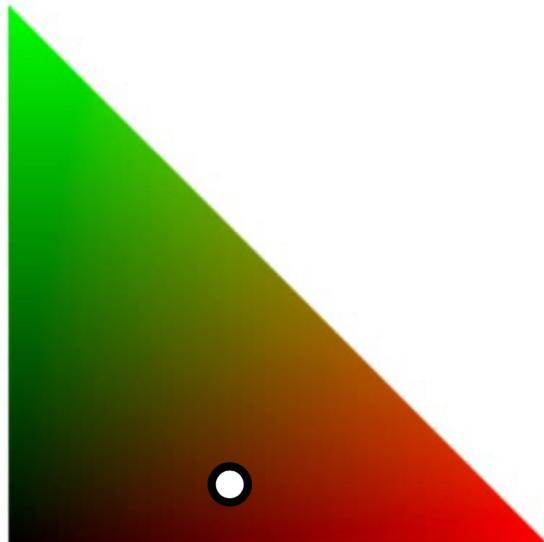
多次平铺相同的纹理，而不是
存储巨大的图像

纹理采样 101

如何绘制纹理映射的三角形基元？

口纹理映射的基本算法

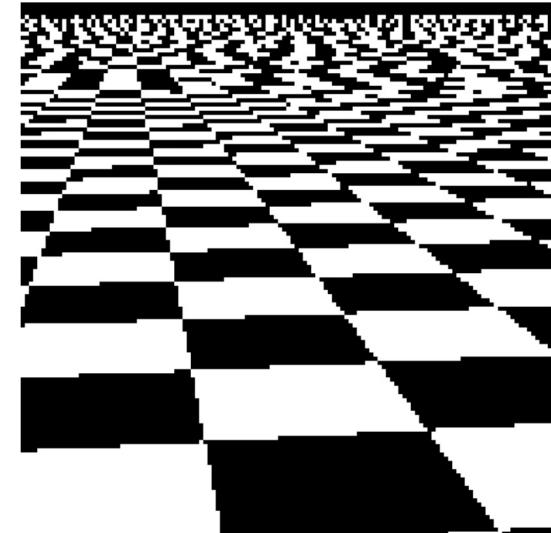
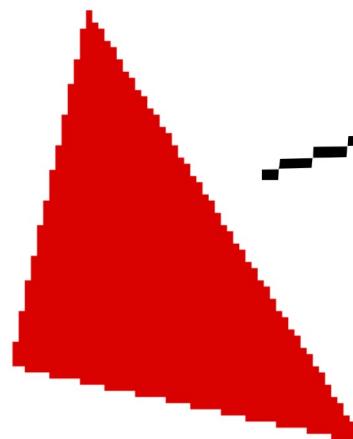
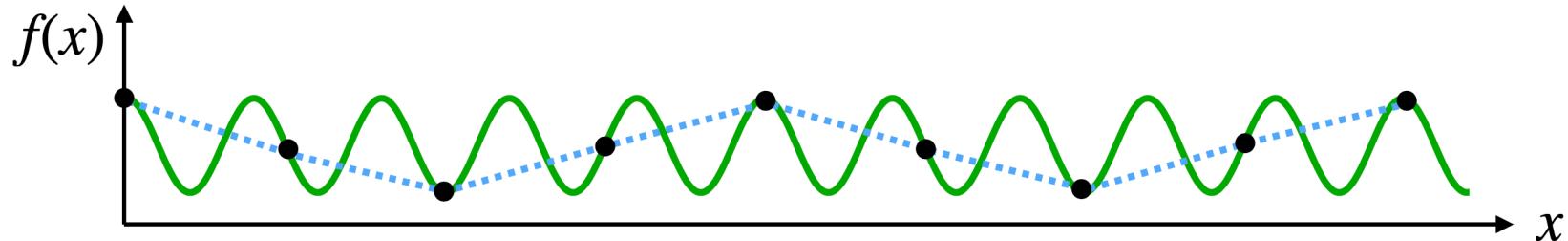
- 对于光栅化图像中的每个像素
 - 在三角形上插值坐标 (u, v)
 - 在插值处 (u, v) 采样 (计算) 纹理
 - 将对应像素的颜色设置为采样的纹理值



...sadly not this easy in general!

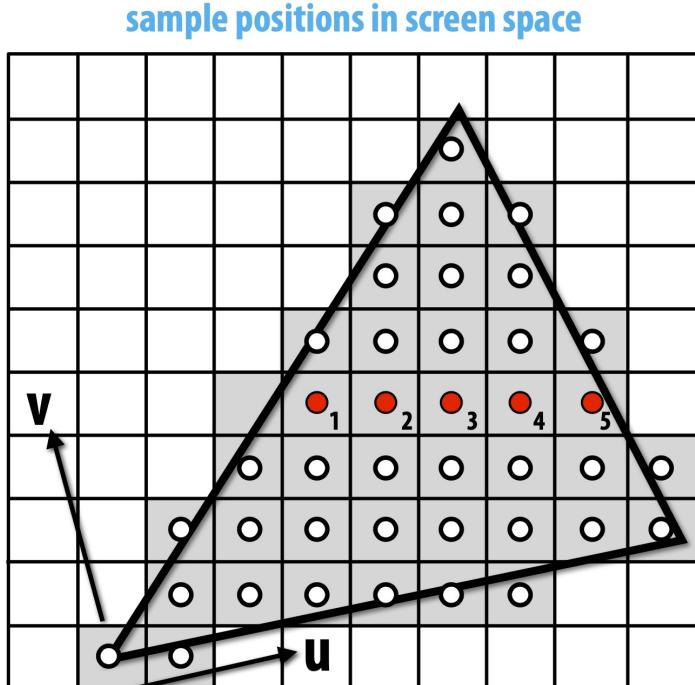
回顾：走样 Aliasing

口高频信号采样不足可能导致走样

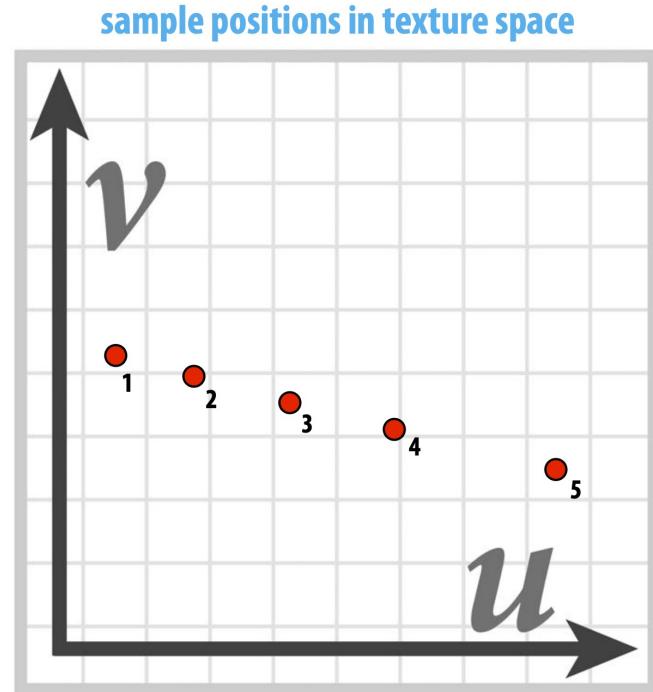


可视化纹理样本

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



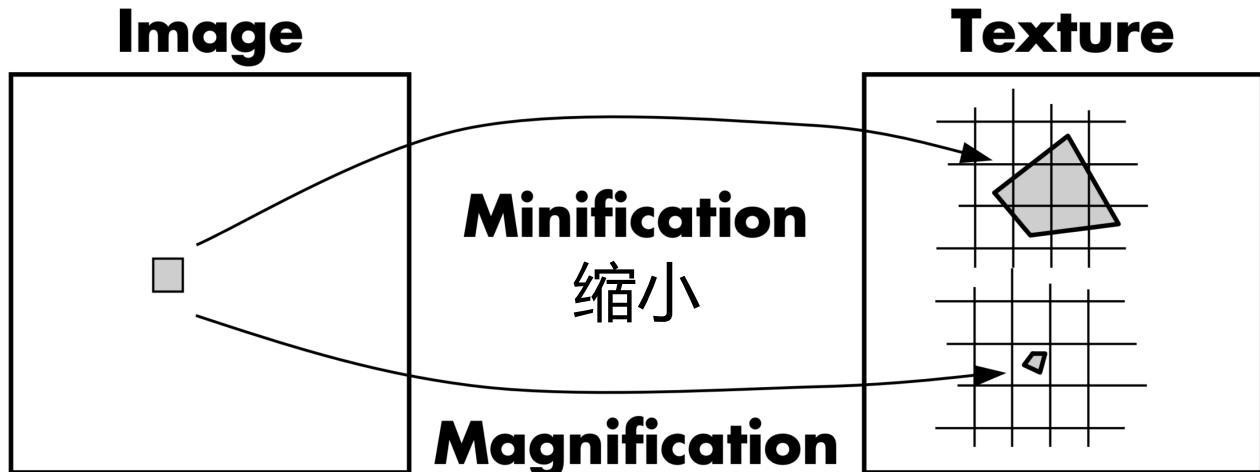
样本位置在屏幕空间中均匀分布
(光栅化器 rasterizer 在这些位置对三角形的外观进行采样)



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

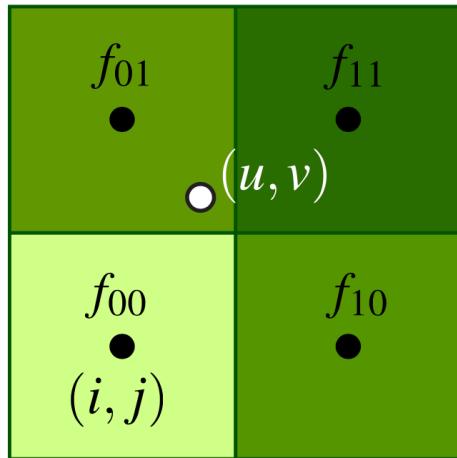
放大

□ 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

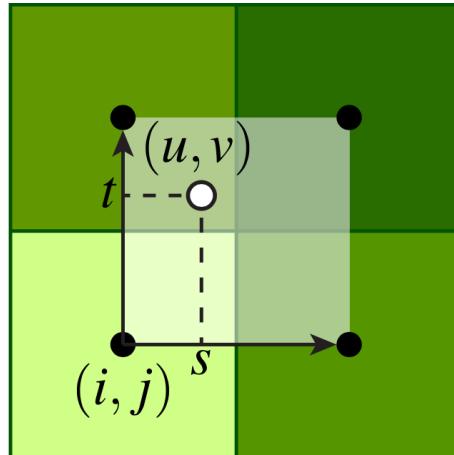
双线性插值 Bilinear interpolation (放大)

口我们如何在非整数位置 (u, v) 查找纹理值?



$$i = \lfloor u - \frac{1}{2} \rfloor$$

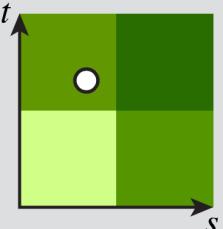
$$j = \lfloor v - \frac{1}{2} \rfloor$$



$$s = u - (i + \frac{1}{2}) \in [0, 1]$$

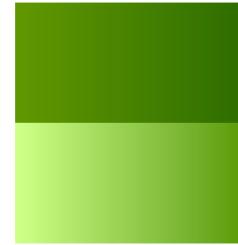
$$t = v - (j + \frac{1}{2}) \in [0, 1]$$

nearest neighbor



fast but ugly:
just grab value of nearest
“texel” (texture pixel)

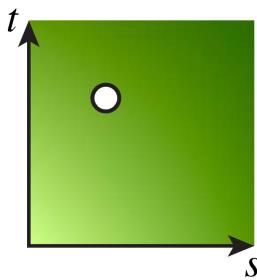
linear (each row)



$$(1 - s)f_{01} + sf_{11}$$

$$(1 - s)f_{00} + sf_{10}$$

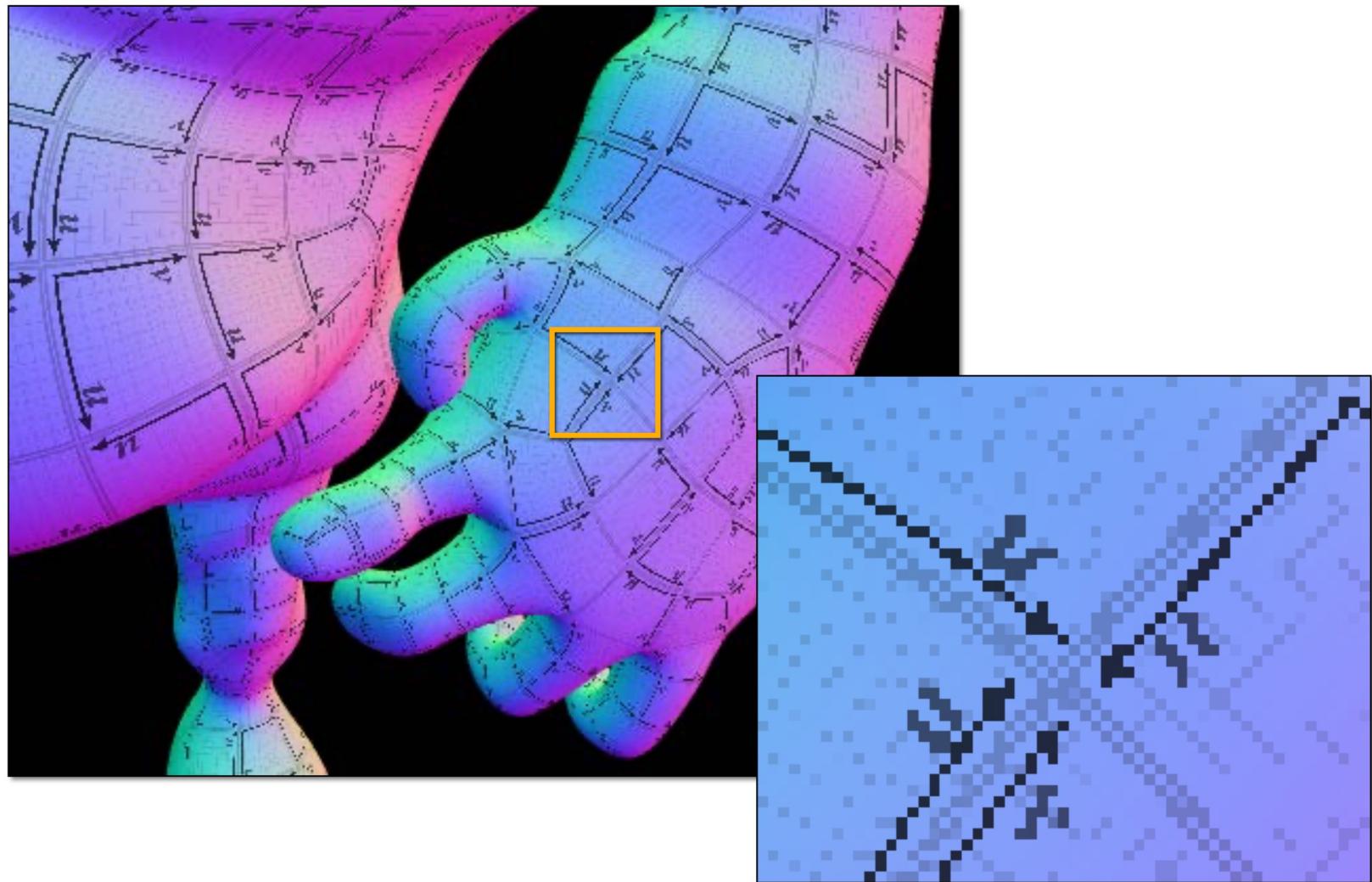
bilinear



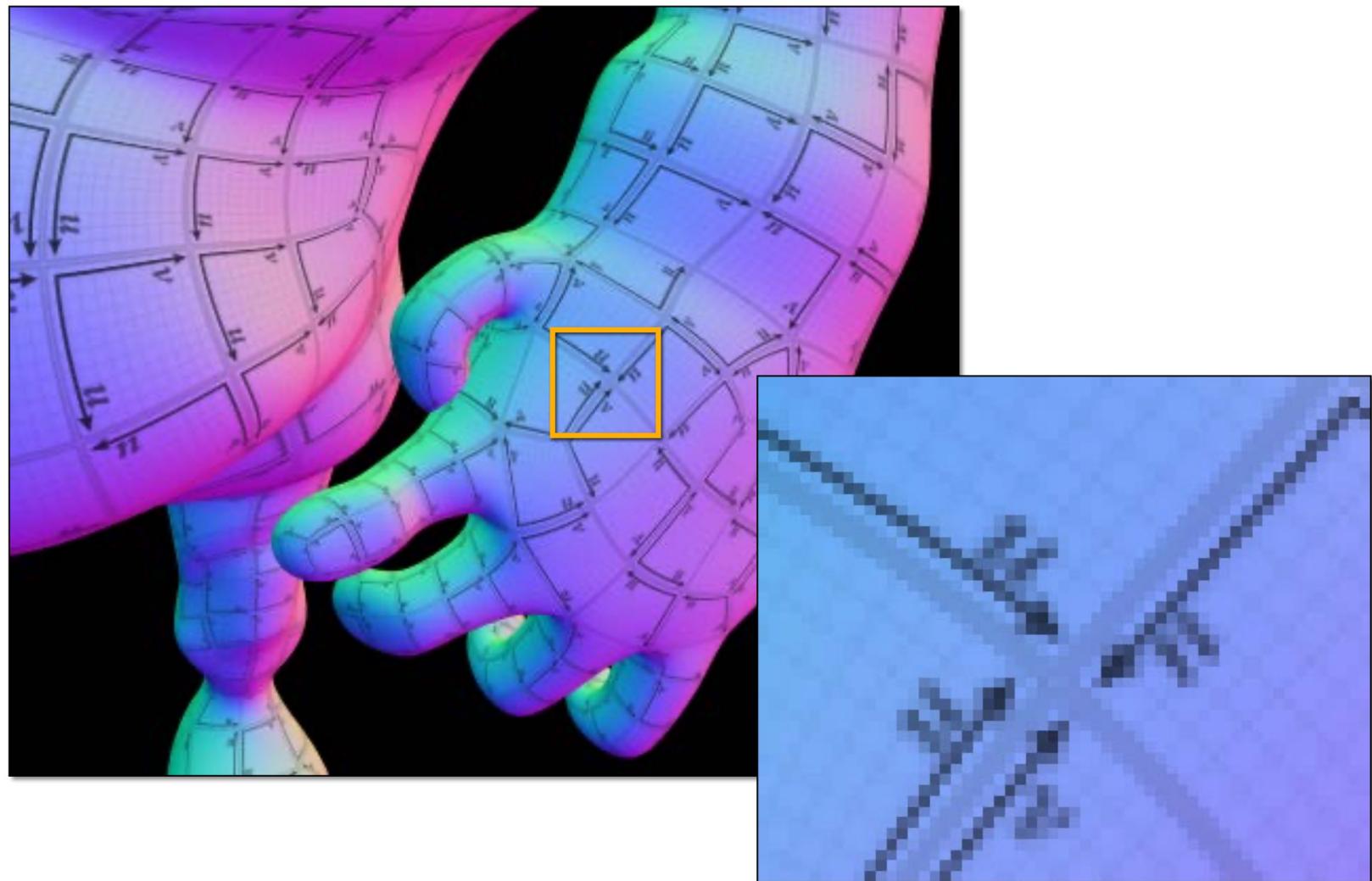
$$(1 - t)((1 - s)f_{00} + sf_{10})$$

$$+ t((1 - s)f_{01} + sf_{11})$$

缩小导致的走样



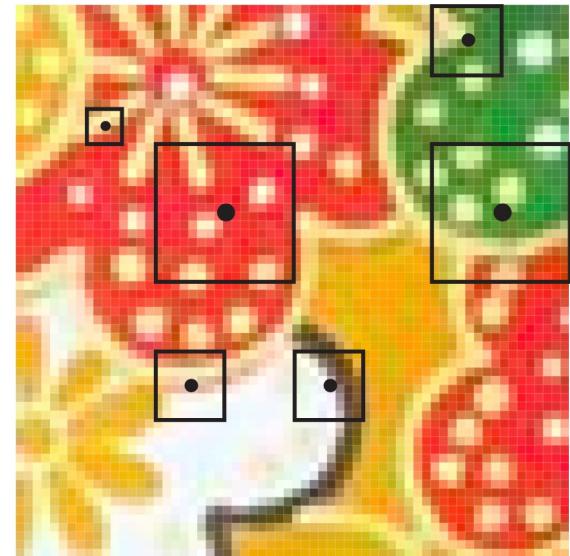
“预过滤”纹理（缩小）



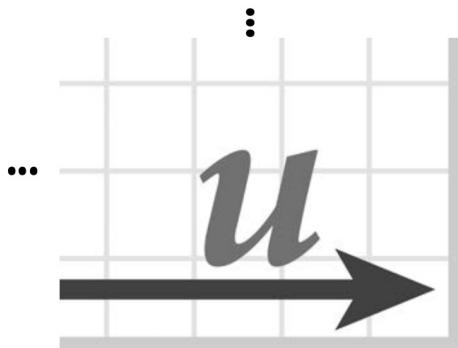
纹理预过滤 Texture prefiltering

- 口 纹理走样经常发生，因为屏幕上的单个像素覆盖了纹理的多个像素
- 口 如果我们只获取像素中心的纹理值，就会有走样现象（即便样本移动得很小，都可能得到“随机”的颜色）
- 口 理想情况下，将使用平均纹理值（就像超采样），但计算成本很高
- 口 相反，我们可以预先计算平均值（一次），然后在运行时查寻这些平均值（多次）

但我们应该存储哪些平均值？
不可能提前计算所有值



预过滤的纹理



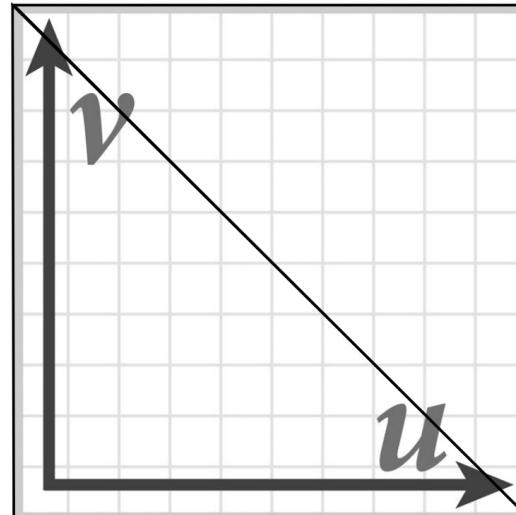
Actual texture: 700x700 image
(only a crop is shown)



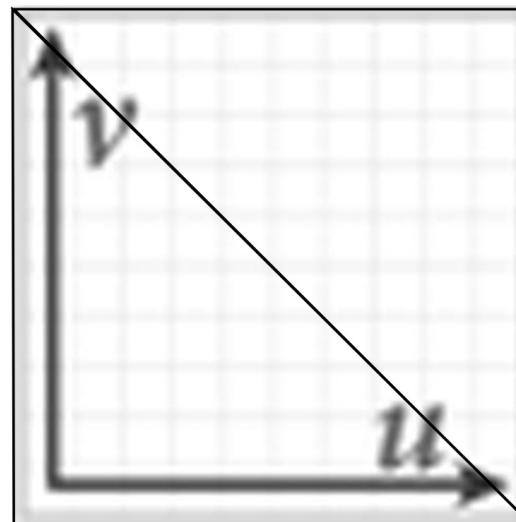
Actual texture: 64x64 image

Q: 两个分辨率就够吗?

A: No...

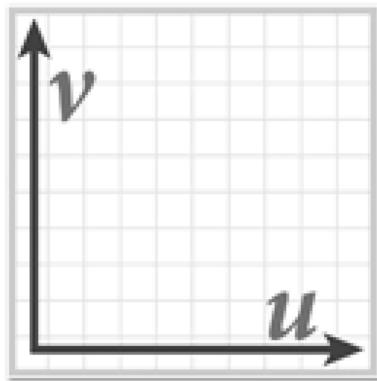


Texture minification

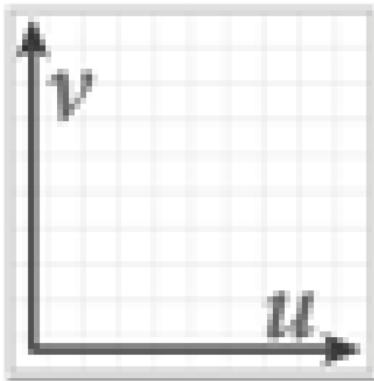


Texture magnification

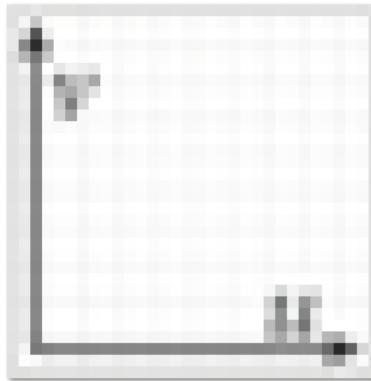
MIP map (L. Williams 83)



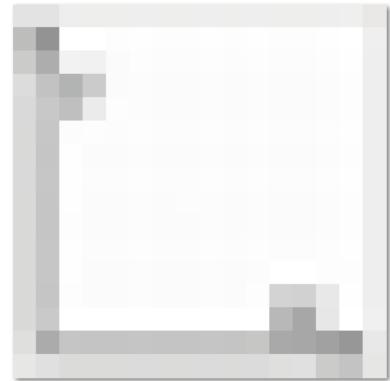
Level 0 = 128x128



Level 1 = 64x64



Level 2 = 32x32



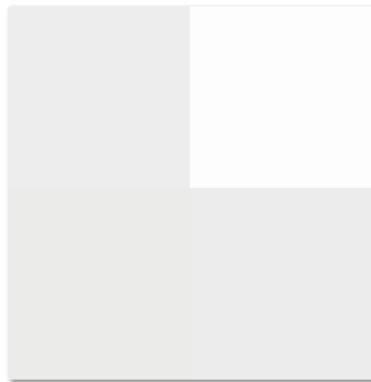
Level 3 = 16x16



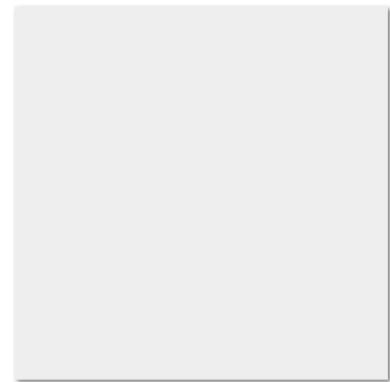
Level 4 = 8x8



Level 5 = 4x4



Level 6 = 2x2



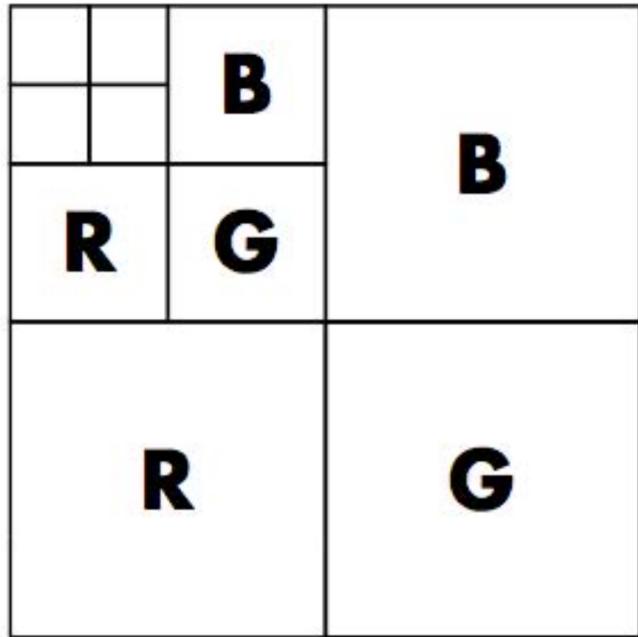
Level 7 = 1x1

□ 粗略想法：以“各种可能的大小”存储预过滤的图像

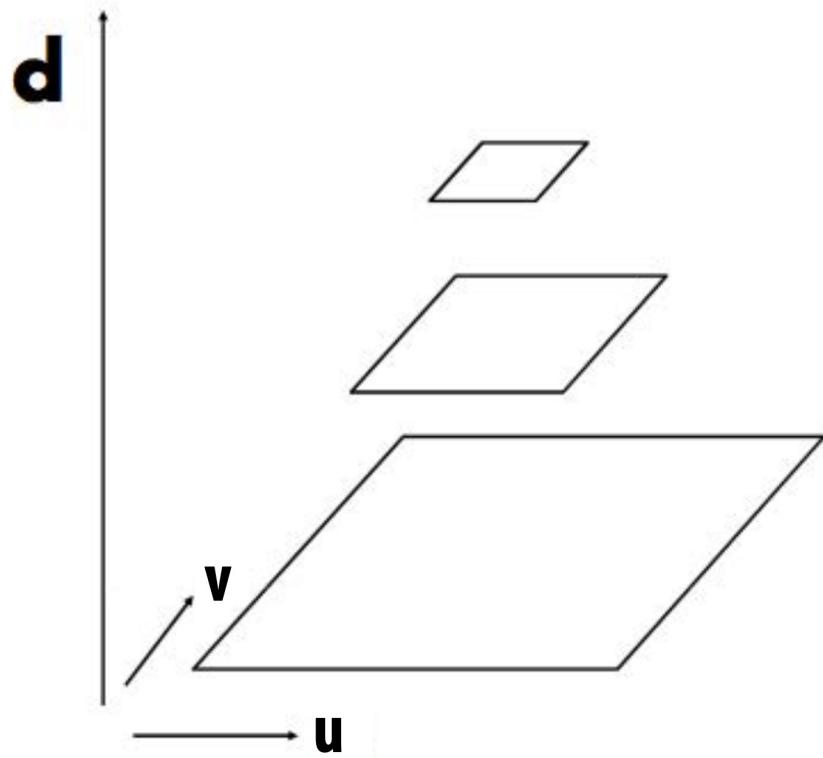
□ 较高级别存储纹理空间中较大区域的纹理平均值（下采样）

□ 从适当大小的 MIP 图中查找单个像素

Mipmap (L. Williams 83)



Williams' original proposed
mip-map layout



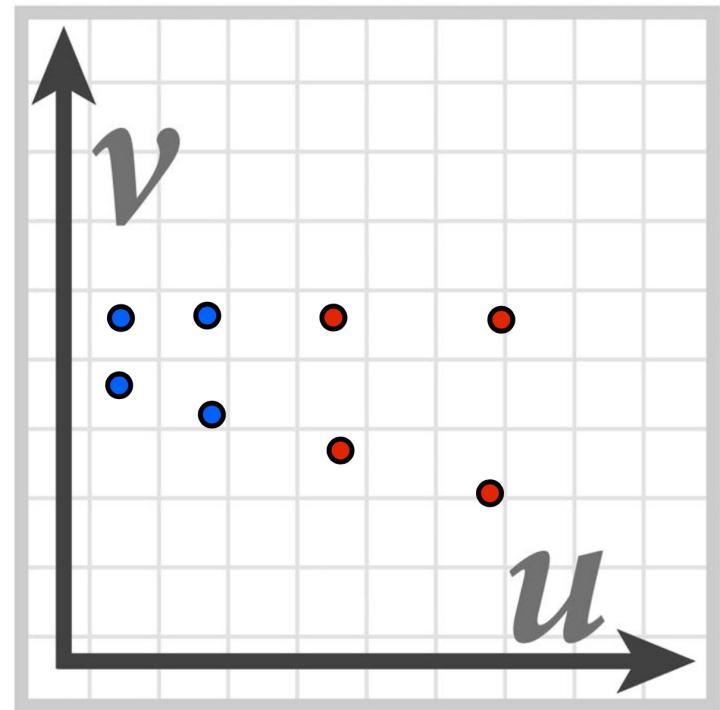
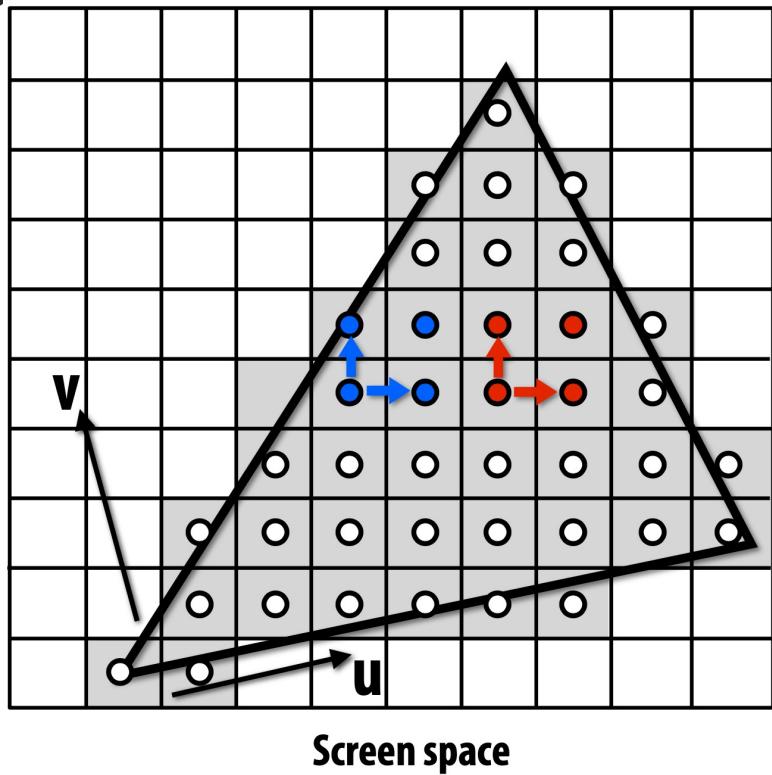
"Mip hierarchy"
 $\text{level} = d$

Q: Mipmap 的存储开销是多少?

A: 大约是原图像的 1/3

计算 MIP Map 级别

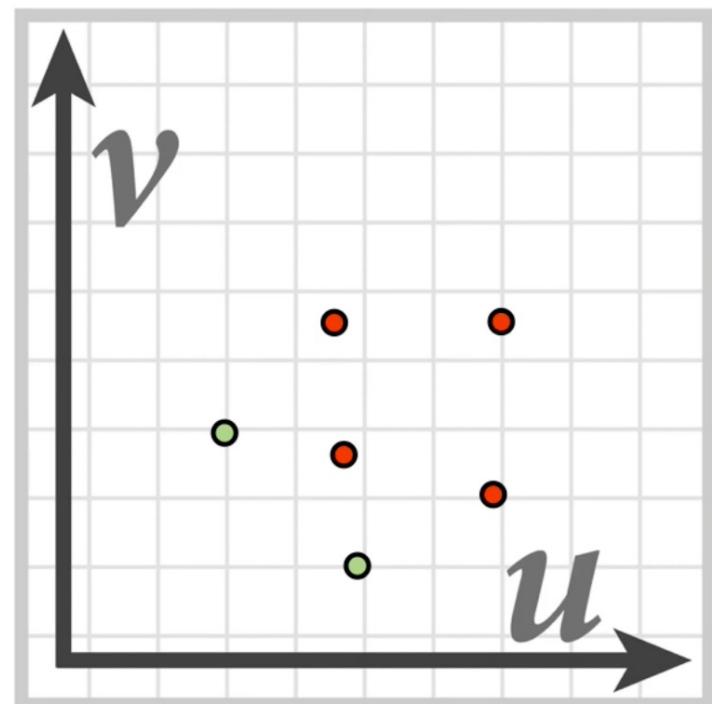
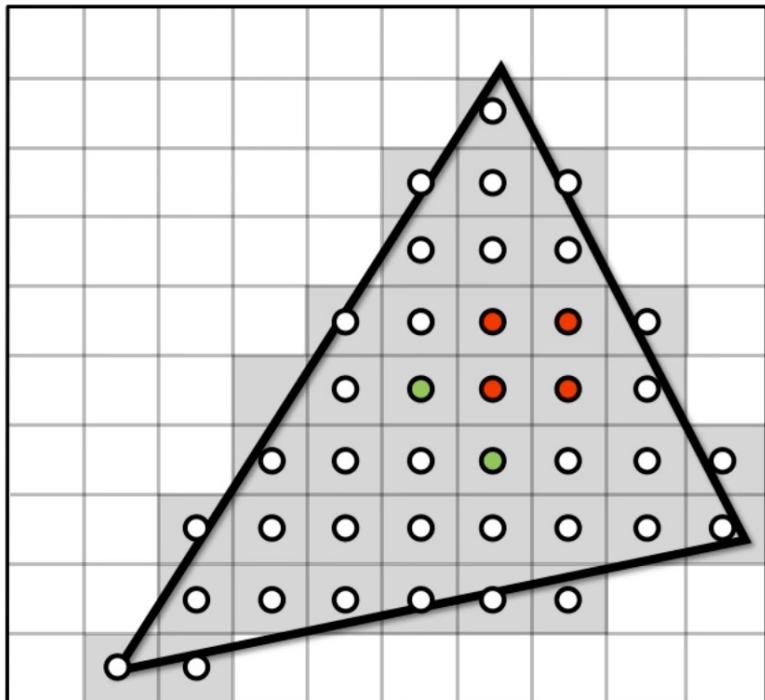
即使在单个三角形内，我们也希望从不同的 MIP Map 级别进行采样



Q: 哪个像素应该从较高的 MIP Map 级别采样：蓝色还是红色像素？

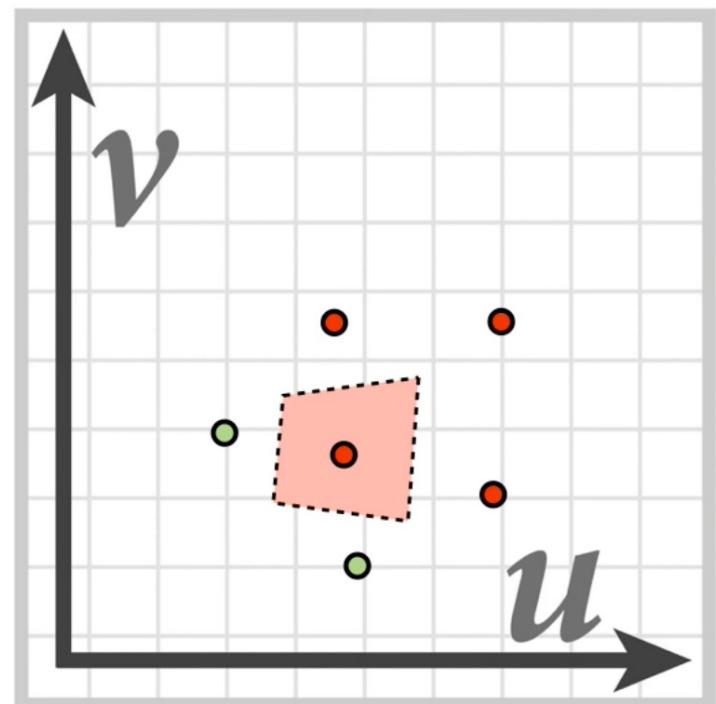
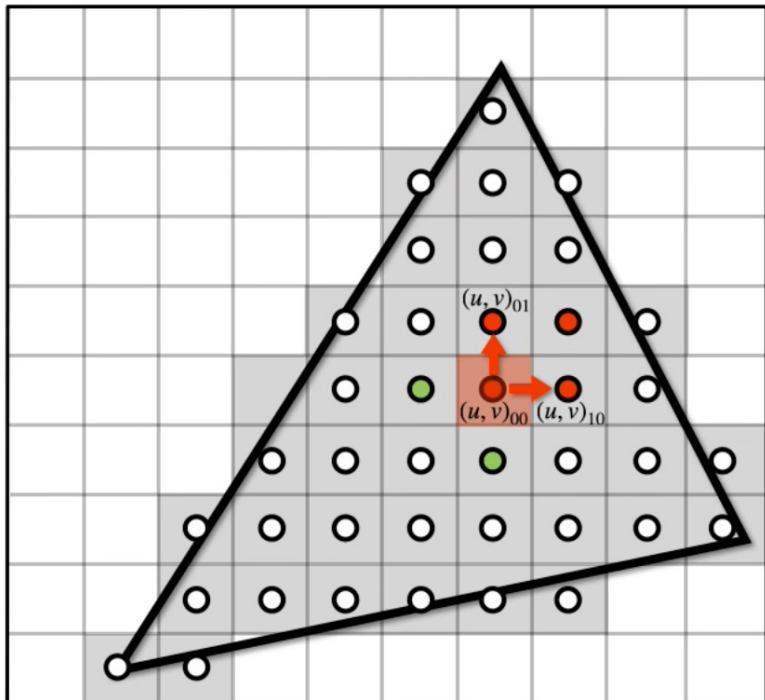
计算 MIP Map 级别

口计算相邻采样的纹理坐标值之间的差异



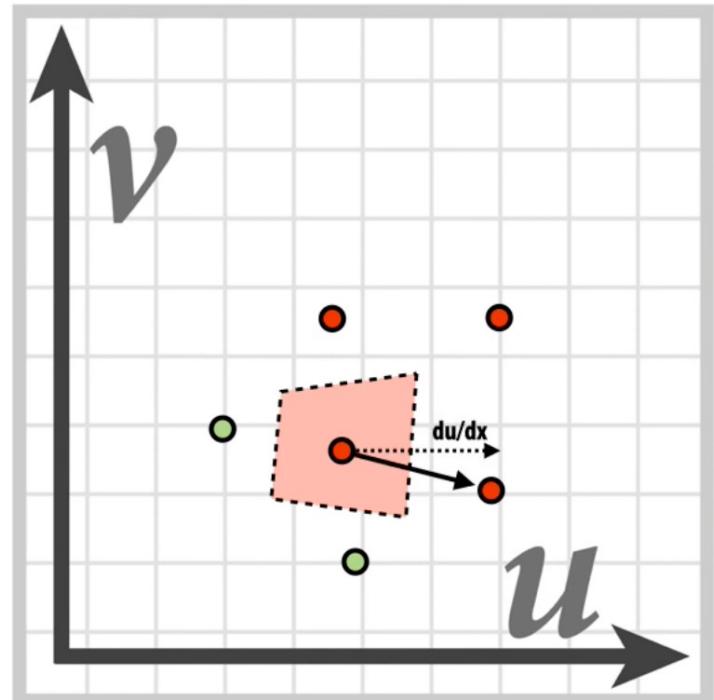
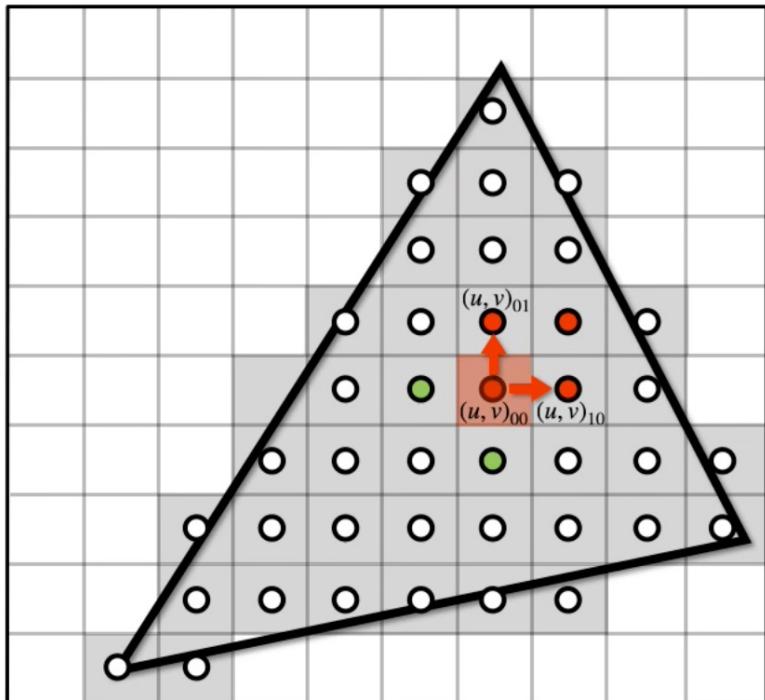
计算 MIP Map 级别

口计算相邻采样的纹理坐标值之间的差异



计算 MIP Map 级别

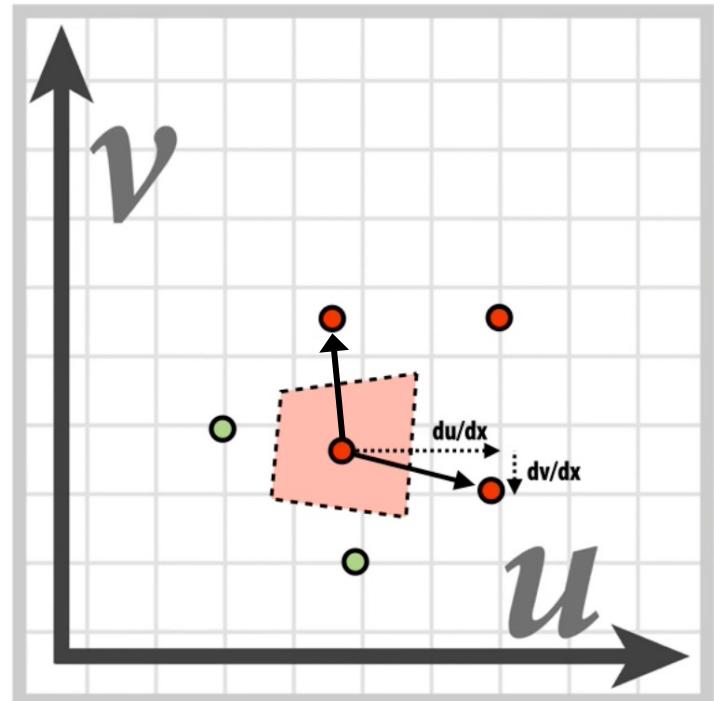
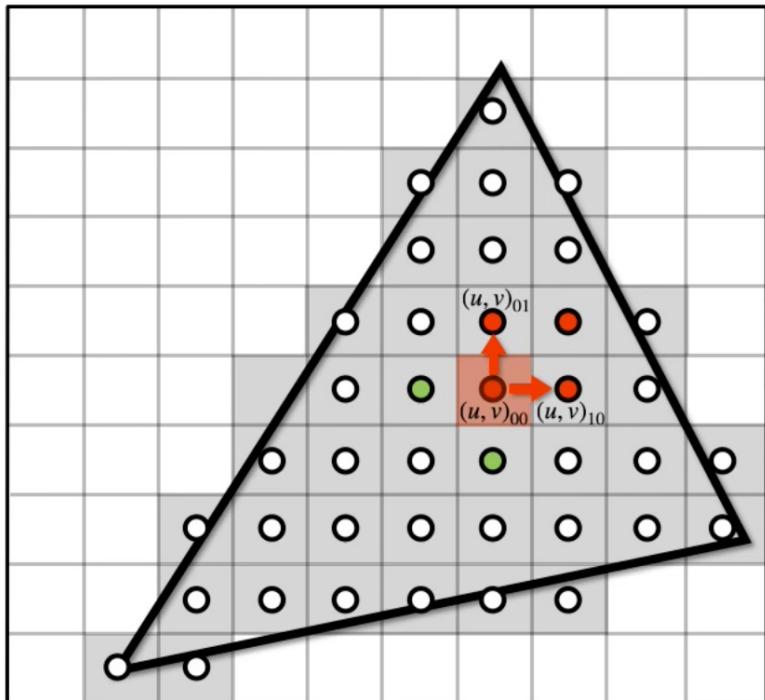
口计算相邻采样的纹理坐标值之间的差异



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

计算 MIP Map 级别

口计算相邻采样的纹理坐标值之间的差异

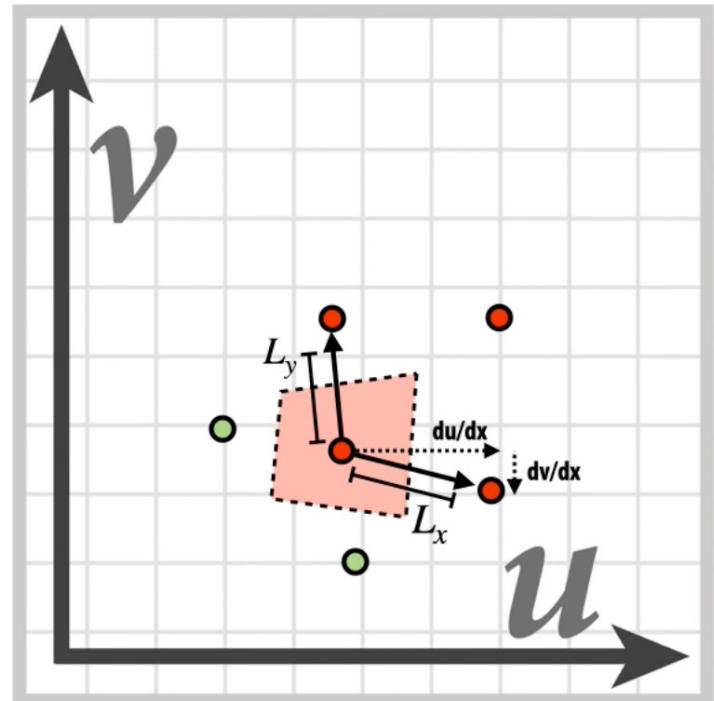
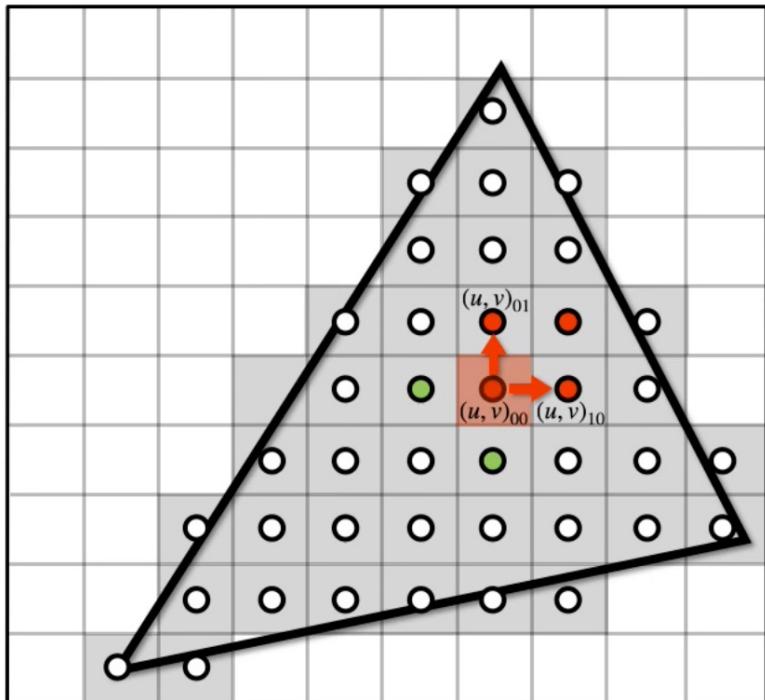


$$\frac{du}{dx} = u_{10} - u_{00} \quad \frac{dv}{dx} = v_{10} - v_{00}$$

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

计算 MIP Map 级别

口计算相邻采样的纹理坐标值之间的差异



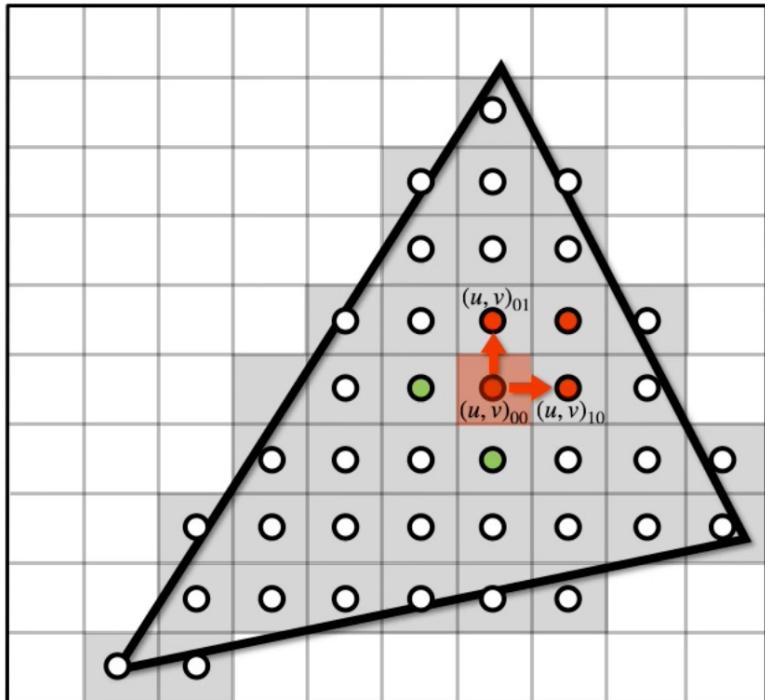
$$\frac{du}{dx} = u_{10} - u_{00} \quad \frac{dv}{dx} = v_{10} - v_{00}$$

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

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

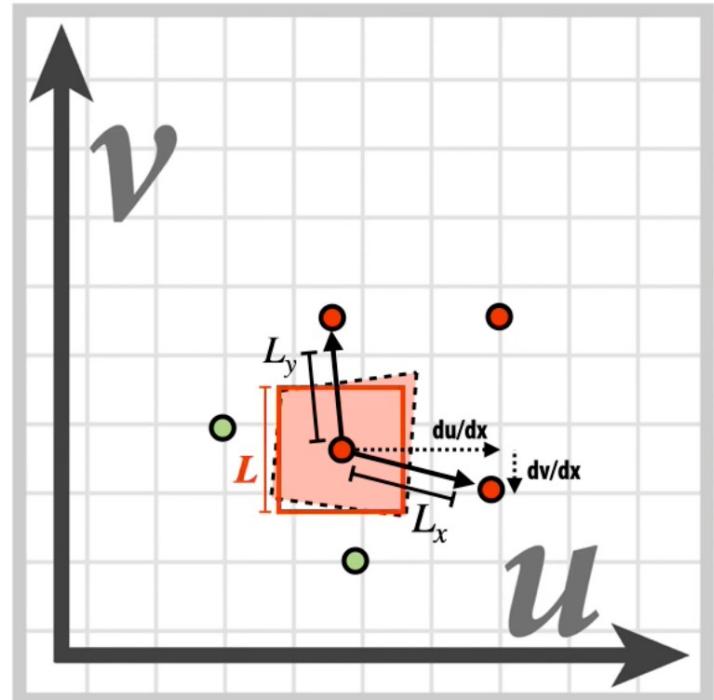
计算 MIP Map 级别

口计算相邻采样的纹理坐标值之间的差异



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

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



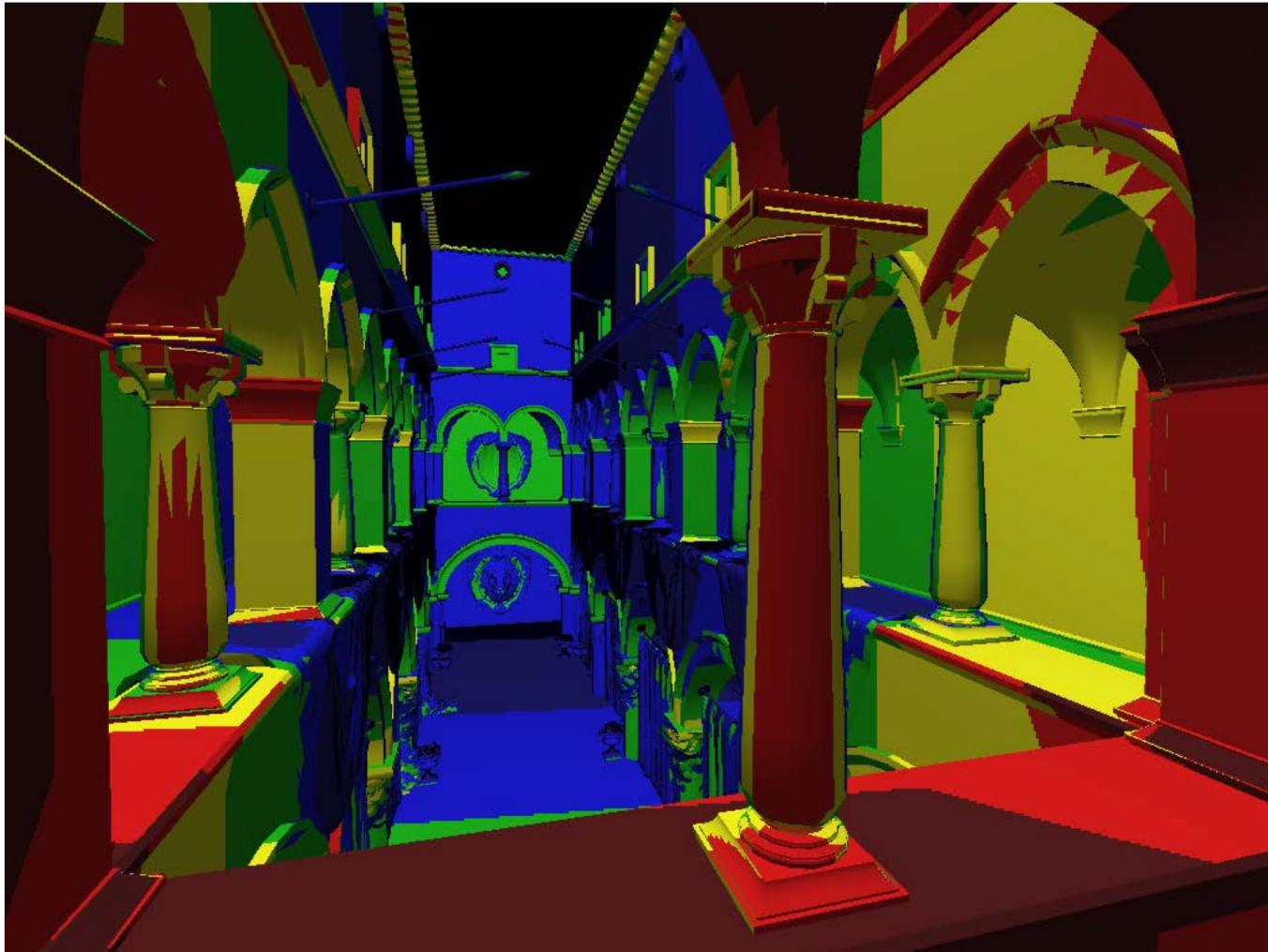
$$L_x^2 = \left(\frac{du}{dx} \right)^2 + \left(\frac{dv}{dx} \right)^2 \quad L_y^2 = \left(\frac{du}{dy} \right)^2 + \left(\frac{dv}{dy} \right)^2$$

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

$$\text{mip-map level: } d = \log_2 L$$

Mip-map 等级可视化

□ d 被固定到最接近的等级



Sponza (bilinear resampling at level 0)



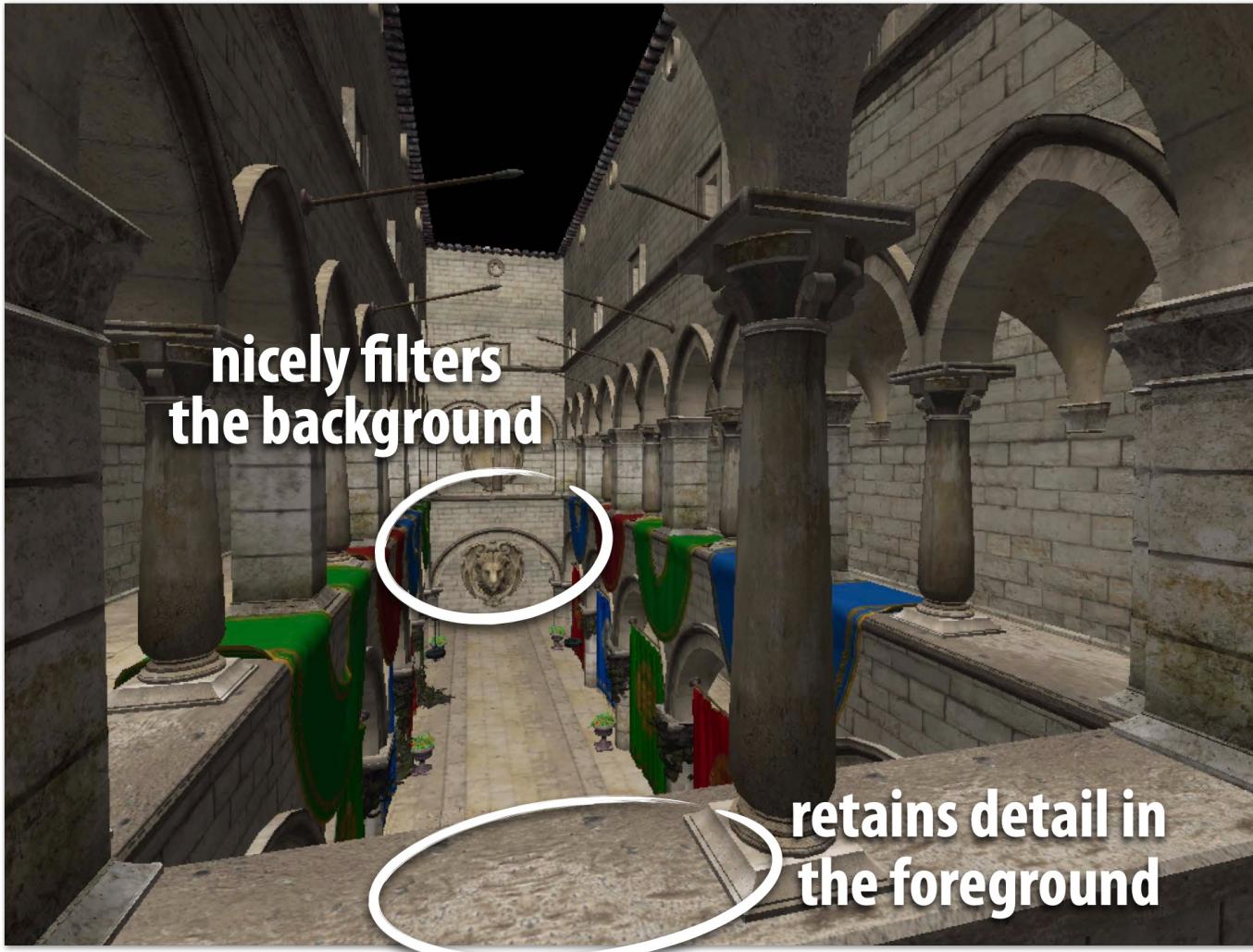
Sponza (bilinear resampling at level 2)



Sponza (bilinear resampling at level 4)

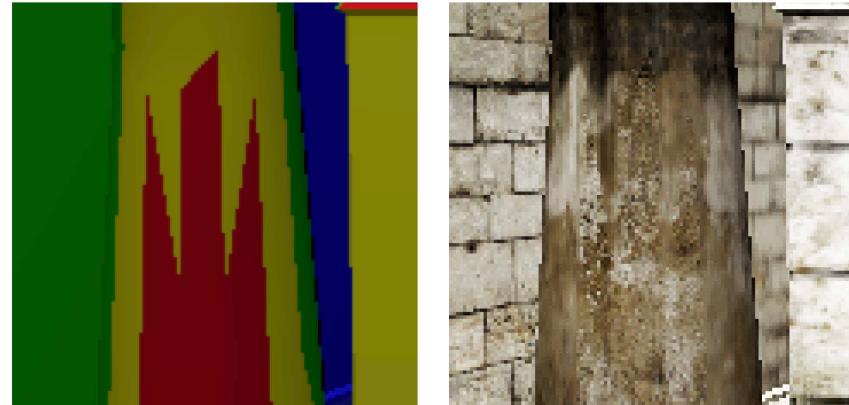


Sponza (MIP mapped)



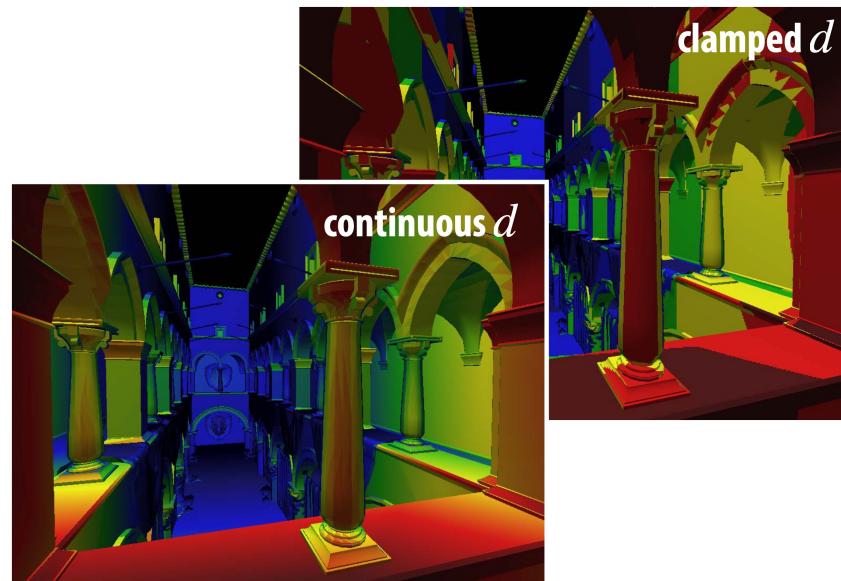
基本 MIP 映射的问题

□ 如果我们只使用最近的级别，就可能出现级别“跳跃”的伪影 — 外观从详细纹理急剧过渡到模糊纹理



□ IDEA：与其将MIP映射级别钳制为最接近的整数，不如使用原始(连续的)MIP映射等级

□ 问题：我们只计算了固定数量的MIP映射级别。如何在级别之间进行插值？



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}

● (u, v, w)

三线性过滤 Trilinear filtering

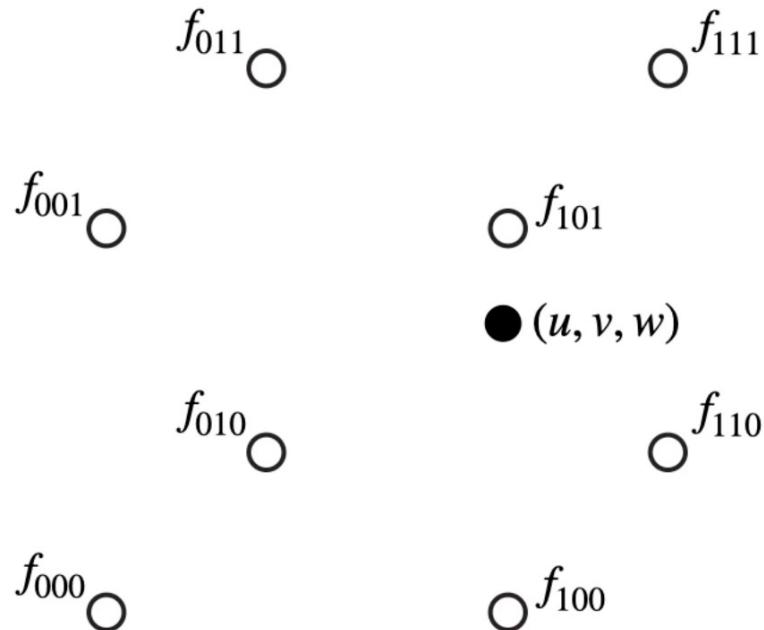
□ 对 2D 数据使用双线性滤波；

可以对 3D 数据使用三线性过滤

□ 给定一个点 $(u, v, w) \in [0,1]^3$ ，

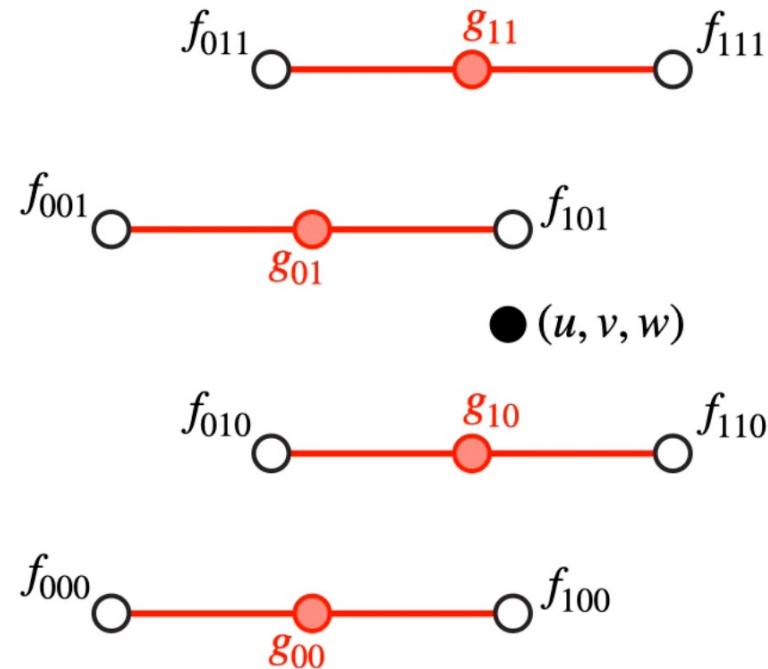
以及 8 个最近的值 f_{ijk}

□ 只需迭代线性过滤：



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



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

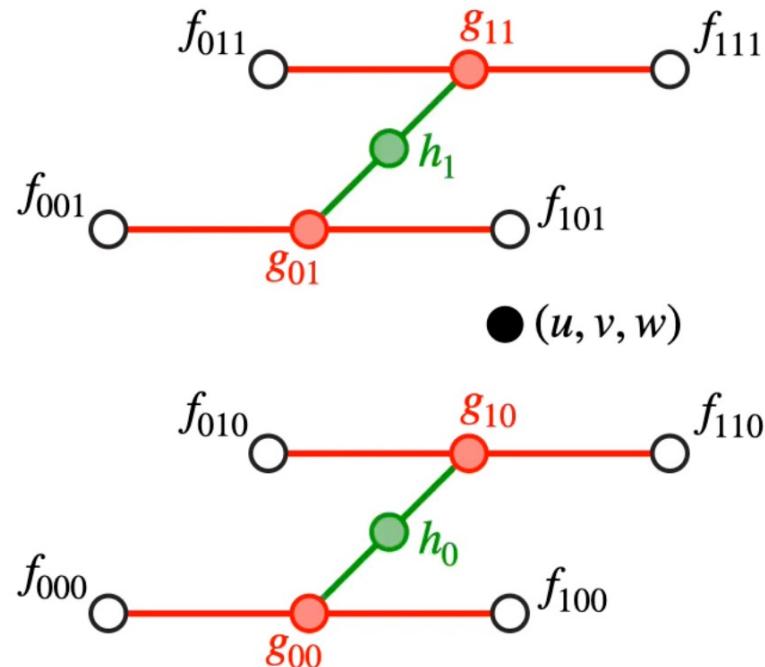
$$g_{10} = (1 - u)f_{010} + uf_{110}$$

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

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

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



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

$$\xrightarrow{\hspace{1cm}} h_0 = (1 - v)g_{00} + vg_{10}$$

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

$$\xrightarrow{\hspace{1cm}} h_1 = (1 - v)g_{01} + vg_{11}$$

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

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：

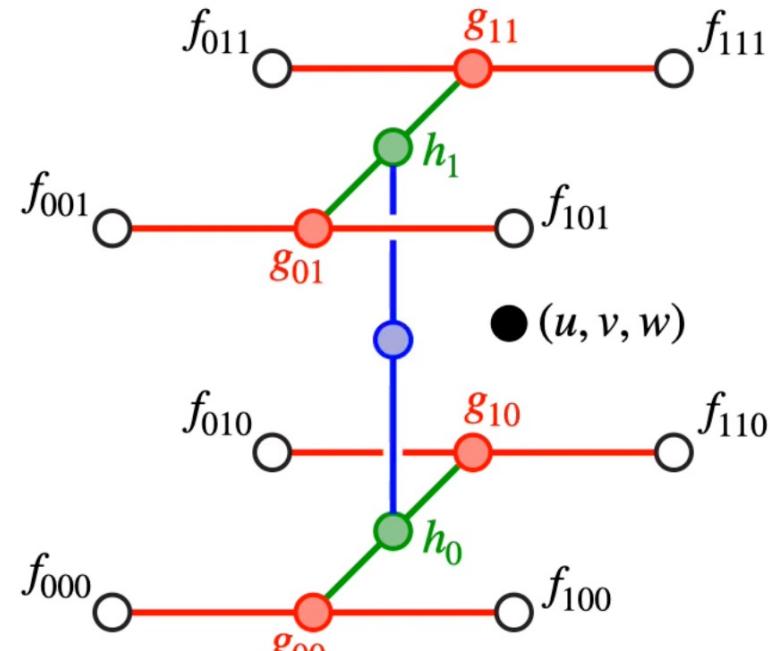
- 沿着 u 做加权平均
- 沿着 v 做加权平均
- 沿着 w 做加权平均

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

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

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

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



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

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

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

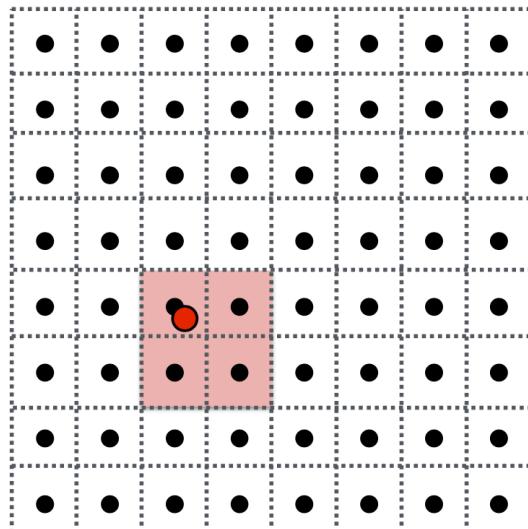
MIP Map 查找

□ MIP Map 插值的工作方式基本相同

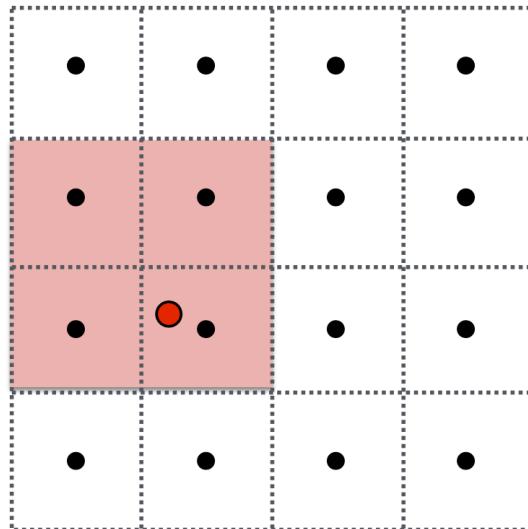
- 从最接近 $d \in \mathbb{R}$ 的两个 MIP Map 级别进行插值
- 在每个级别中独立执行双线性插值
- 使用 $w = d - \lfloor d \rfloor$ 在两个双线性值之间插值

□ 开始变得昂贵! (\rightarrow 专用硬件)

- 双线性插值:
 - 4 次纹素读 (texel read)
 - 3 次像素插值 (3 mul + 6 add)
- 三线性/MIP map 插值:
 - 8 次像素读
 - 7 次像素插值 (7 mul + 14 add)



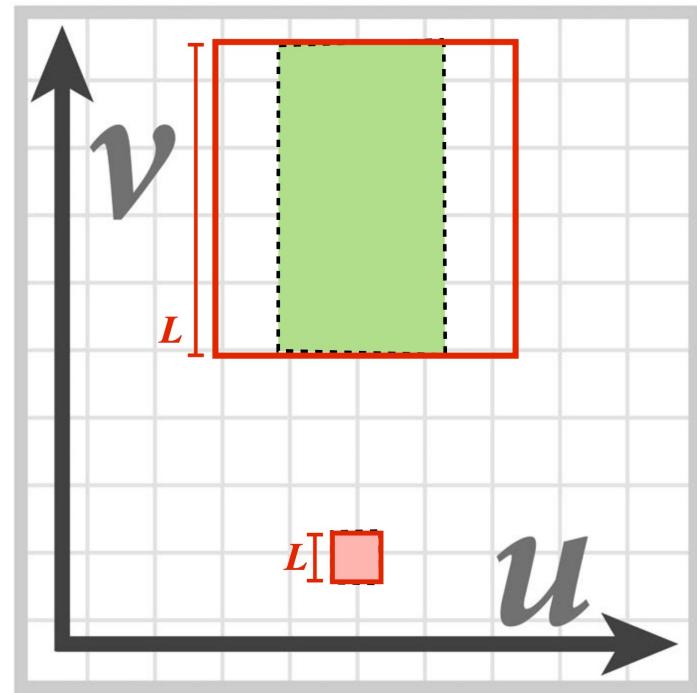
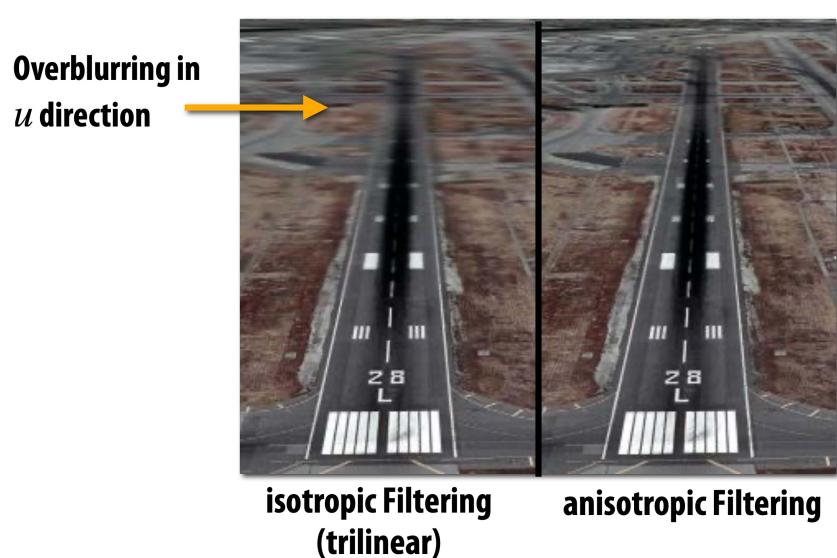
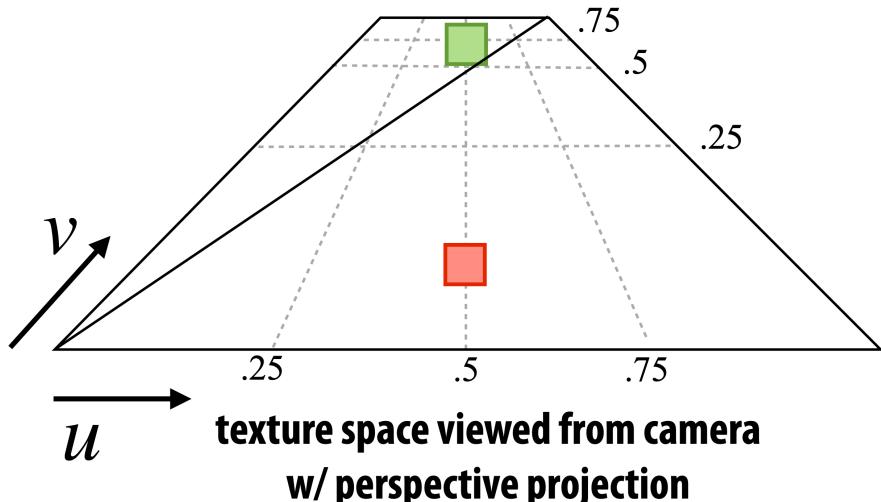
mip-map texels: level $[d]$



mip-map texels: level $[d] + 1$

各向异性过滤 Anisotropic Filtering

口样本可能沿着 u 和 v 拉伸 (非常) 不同的量



常见解决方案：组合多个 MIP map 样本（更多的计算/带宽! ）

纹理采样流程

- 通过重心插值从屏幕样本 (x, y) 计算 u 和 v
- 通过求屏幕相邻样本的积分近似 $\frac{du}{dx}, \frac{du}{dy}, \frac{dv}{dx}, \frac{dv}{dy}$
- 计算 MIP Map 级别 d
- 将归一化的 $[0,1]$ 纹理坐标 (u, v) 转换为纹理图像中的像素位置 $(U, V) \in [W, H]$
- 确定过滤器所需的纹素地址 (例如，三线性的八个邻居)
- 将纹素加载到本地寄存器
- 根据 (U, V, d) 执行三线性插值
- (...even more work for anisotropic filtering...)

要点：高质量的纹理处理需要比查找图像中的像素多得多的工作！每个样本都需要大量的运算和带宽

出于这个原因，图形处理单元 (GPU) 有专门的、固定功能的硬件支持来执行纹理采样操作

透视和纹理映射 – 总结

口 透视投影 将三维基元转换为可以光栅化的二维基元

- 用于管理裁剪、Z-fighting 的 视锥

口 一旦有了 2D 基元，就可以使用 重心坐标 在顶点之间插值属性

口 重要示例： 纹理坐标，用于将 2D 图像片段复制到 3D 曲面上

口 需要仔细的 纹理过滤 以避免走样

- 关键思想：像素覆盖的平均颜色是什么？
- 对于放大，只需进行 双线性查找
- 对于缩小，使用 预过滤 提前计算平均值
 - MIP map 存储不同级别的平均值
 - 使用 三线性滤波 在级别之间混合

口 在超大射角下，需要 各向异性滤波 来处理样本的“拉伸”

口 一般来说， 没有完美的走样解决方案！ 努力平衡质量和效率



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