中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

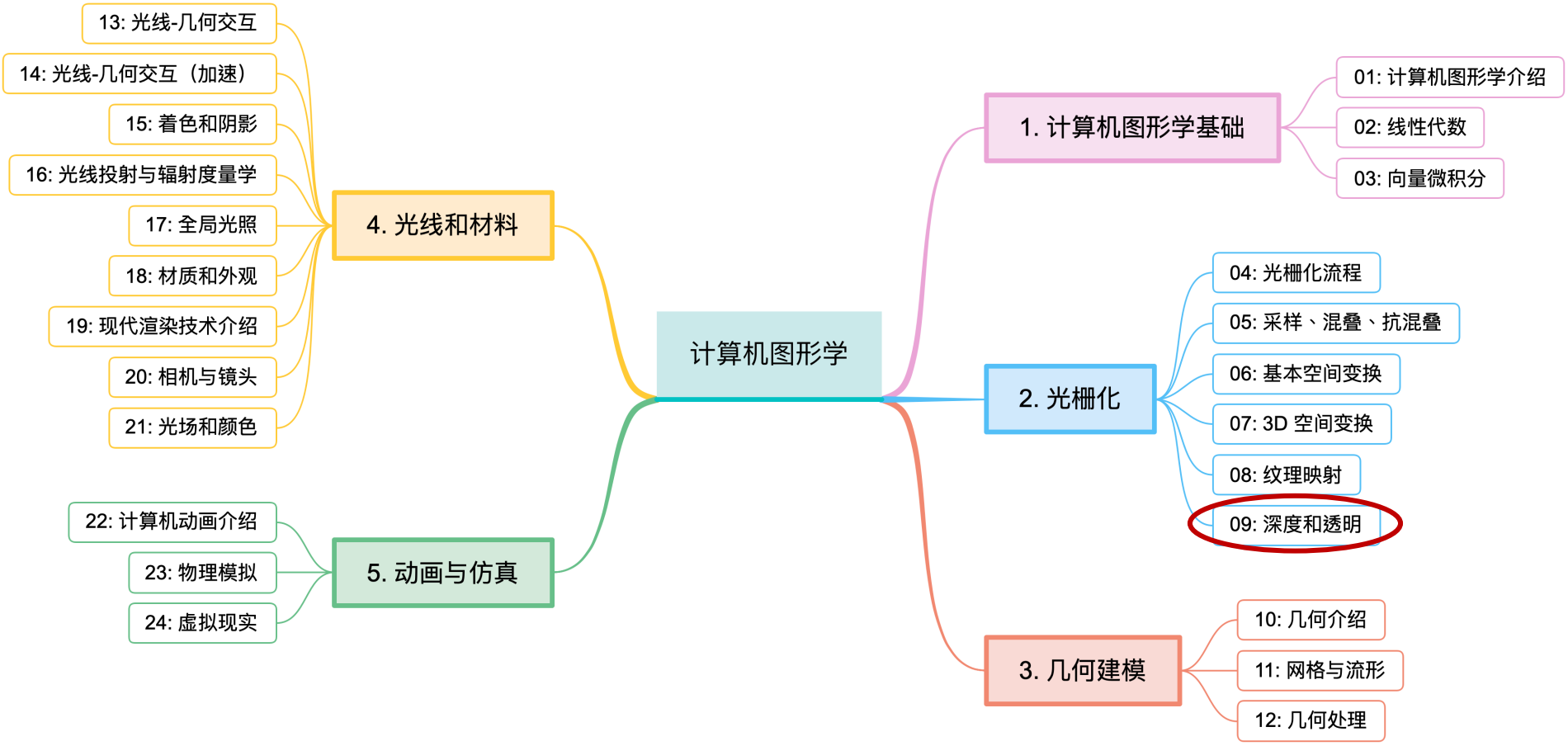
Lecture 09：深度和透明度

SSE315：计算机图形学
Computer Graphics

陈壮彬

软件工程学院

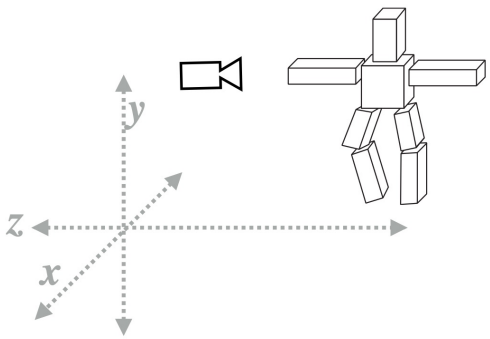
chenzhb36@mail.sysu.edu.cn


计算机图形学
1. 计算机图形学基础
01: 计算机图形学介绍
02: 线性代数
03: 向量微积分
2. 光栅化
04: 光栅化流程
05: 采样、混叠、抗混叠
06: 基本空间变换
07: 3D 空间变换
08: 纹理映射
09: 深度和透明
3. 几何建模
10: 几何介绍
11: 网格与流形
12: 几何处理
4. 光线和材料
13: 光线-几何交互
14: 光线-几何交互（加速）
15: 着色和阴影
16: 光线投射与辐射度量学
17: 全局光照
18: 材质和外观
19: 现代渲染技术介绍
20: 相机与镜头
21: 光场和颜色
5. 动画与仿真
22: 计算机动画介绍
23: 物理模拟
24: 虚拟现实

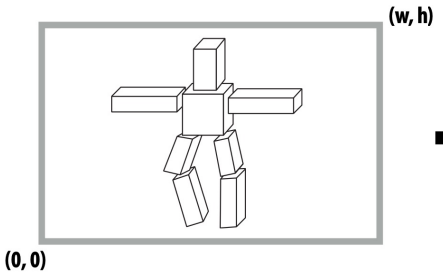
Today: 总结光栅话流程!

- 回顾我们的目标
  - 从一些输入 (INPUTS) 开始 (基本都是点和三角形)
    - 可能有一些其他属性 (比如**颜色**，**纹理**)
  - 应用一系列的变换：流程的不同阶段 (STAGES)
  - 生成输出 (OUTPUTS)，即最终图片

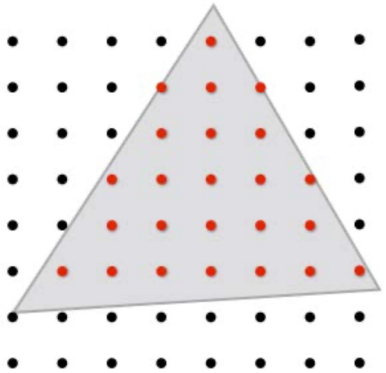
What we know how to do so far…



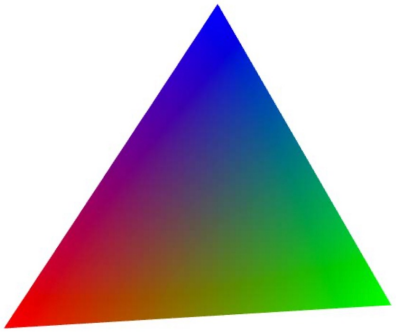
position objects in the world
(3D transformations)



project objects onto the screen
(perspective projection)

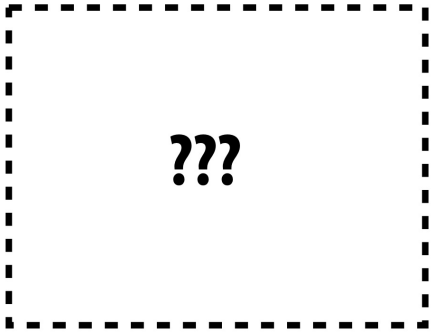


sample triangle coverage
(rasterization)



interpolate vertex attributes
(barycentric coodinates)

sample texture maps
(filtering, mipmapping)

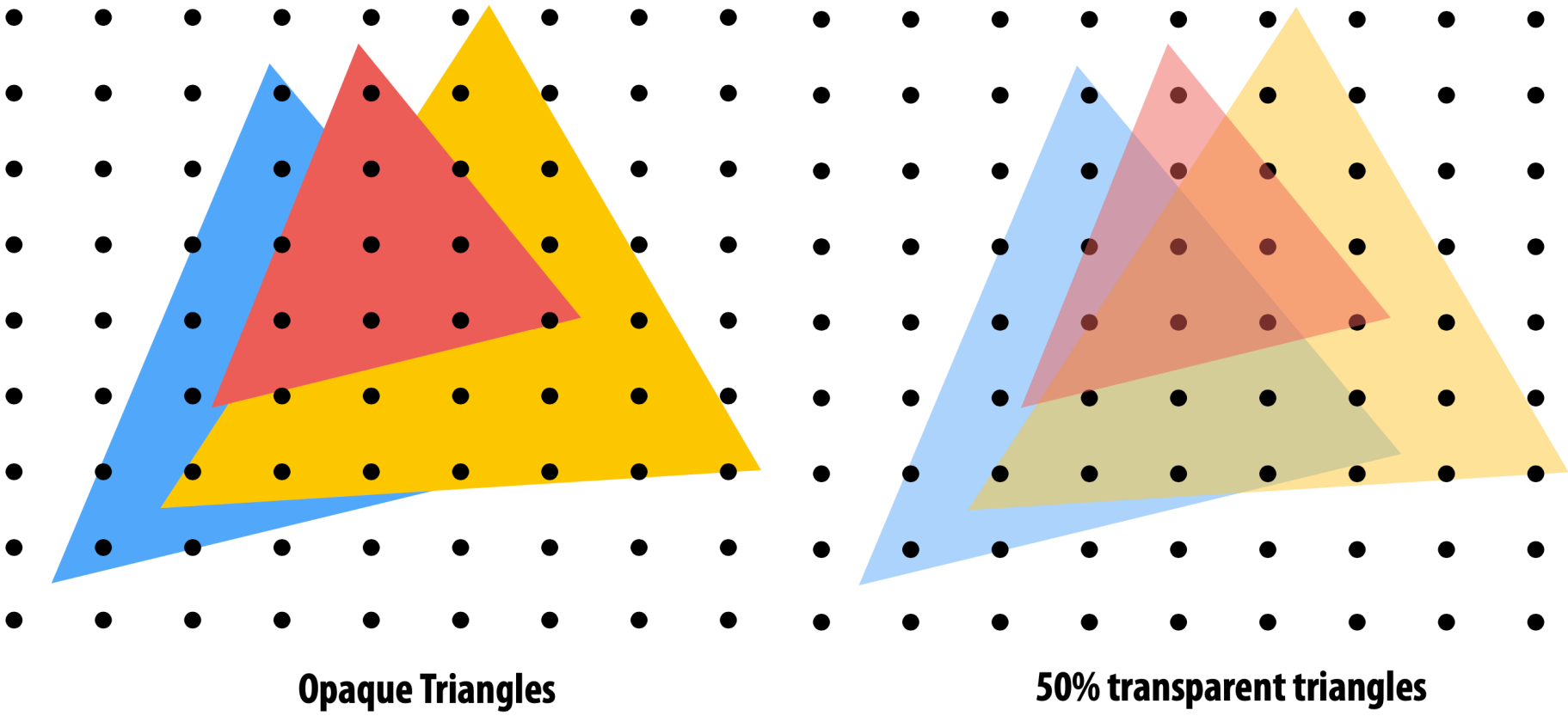


put samples into frame buffer
(depth & alpha)

遮挡
Occlusion

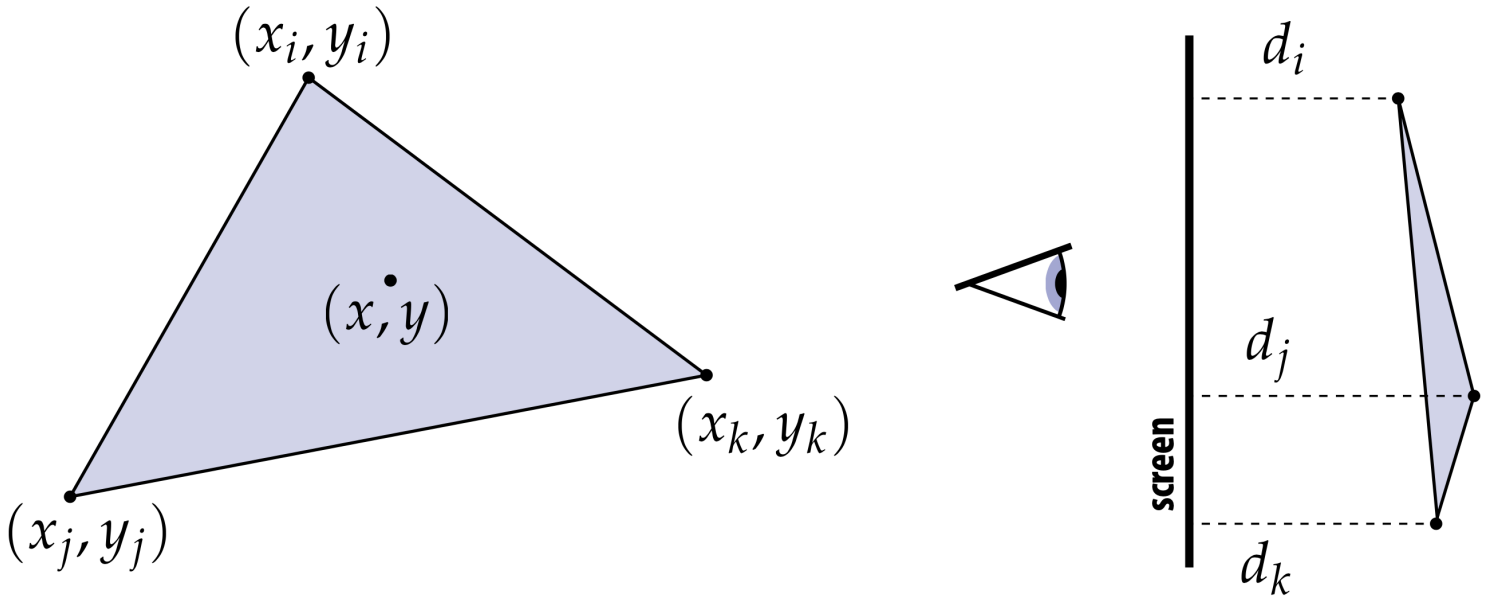
遮挡

- 在每个采样点上，哪个三角形可见?



采样深度 Depth

- 假设我们有如下三角形：
  - 每个顶点的 2D 投影坐标为 (x_i, y_i)
  - 每个顶点的深度 (点到观察者的距离) 为 d_i

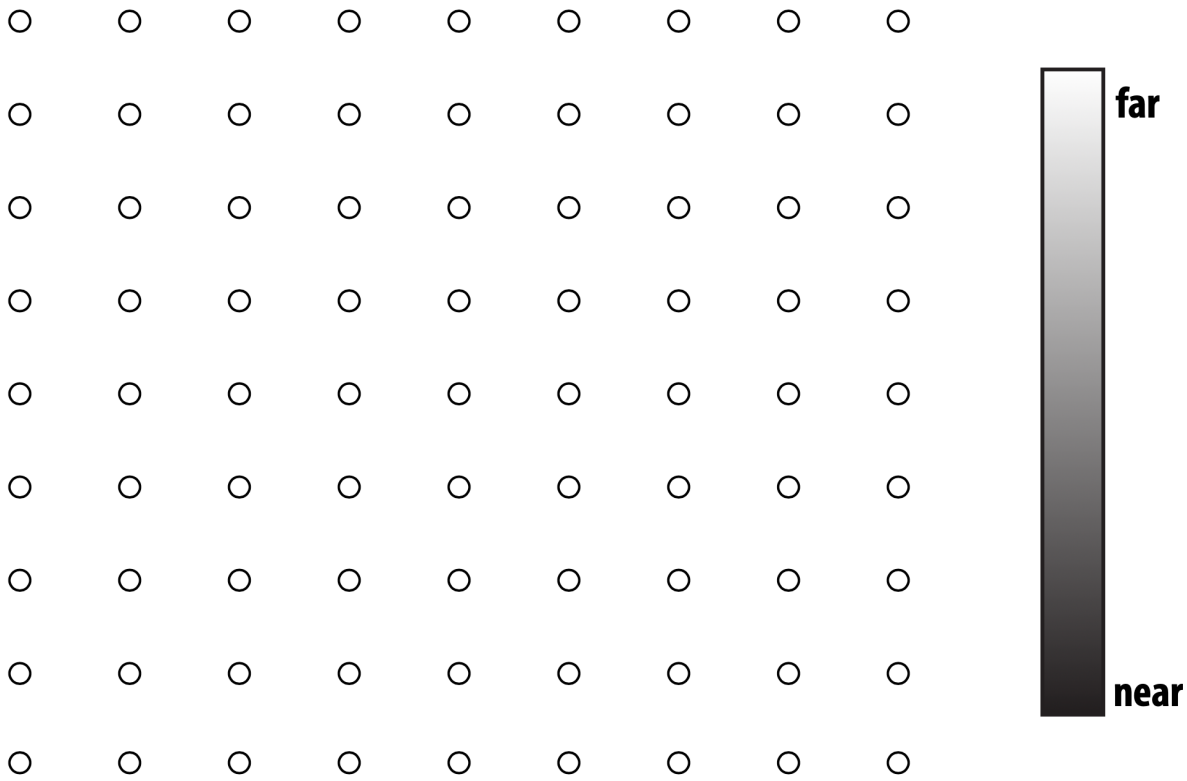


Q：如何计算任意给定点 (x, y) 的深度 d?

A： 使用重心坐标进行插值（与三角形其他线性变化的属性一样，如颜色）

深度缓冲 Depth-buffer (Z-buffer)

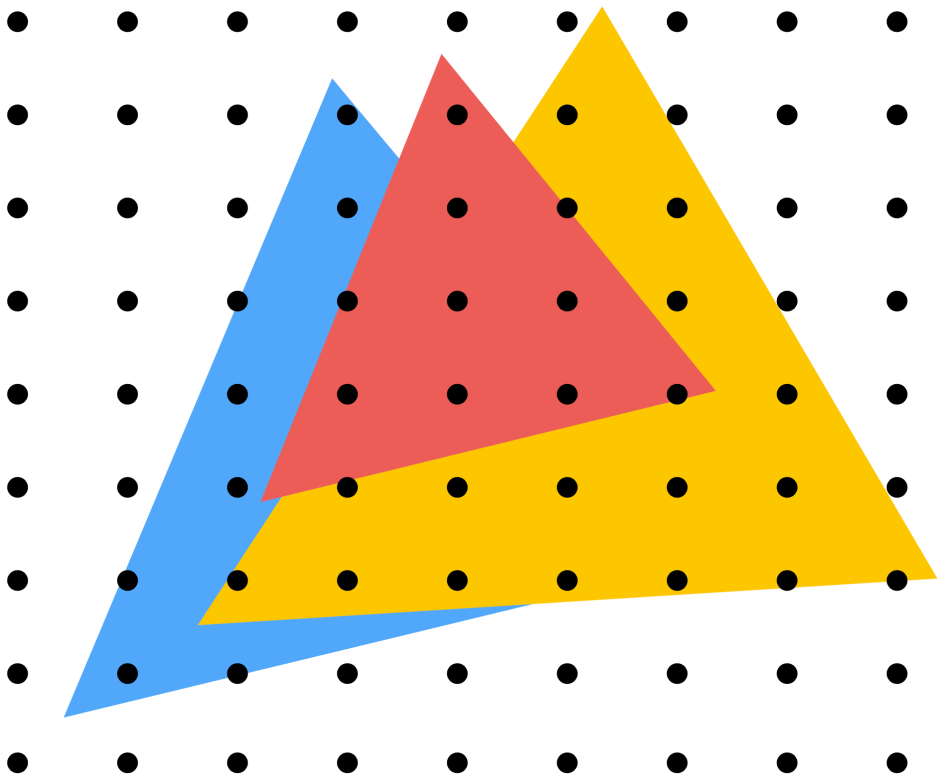
- 对于每个样本，深度缓冲存储到目前为止看到的最近三角形的深度



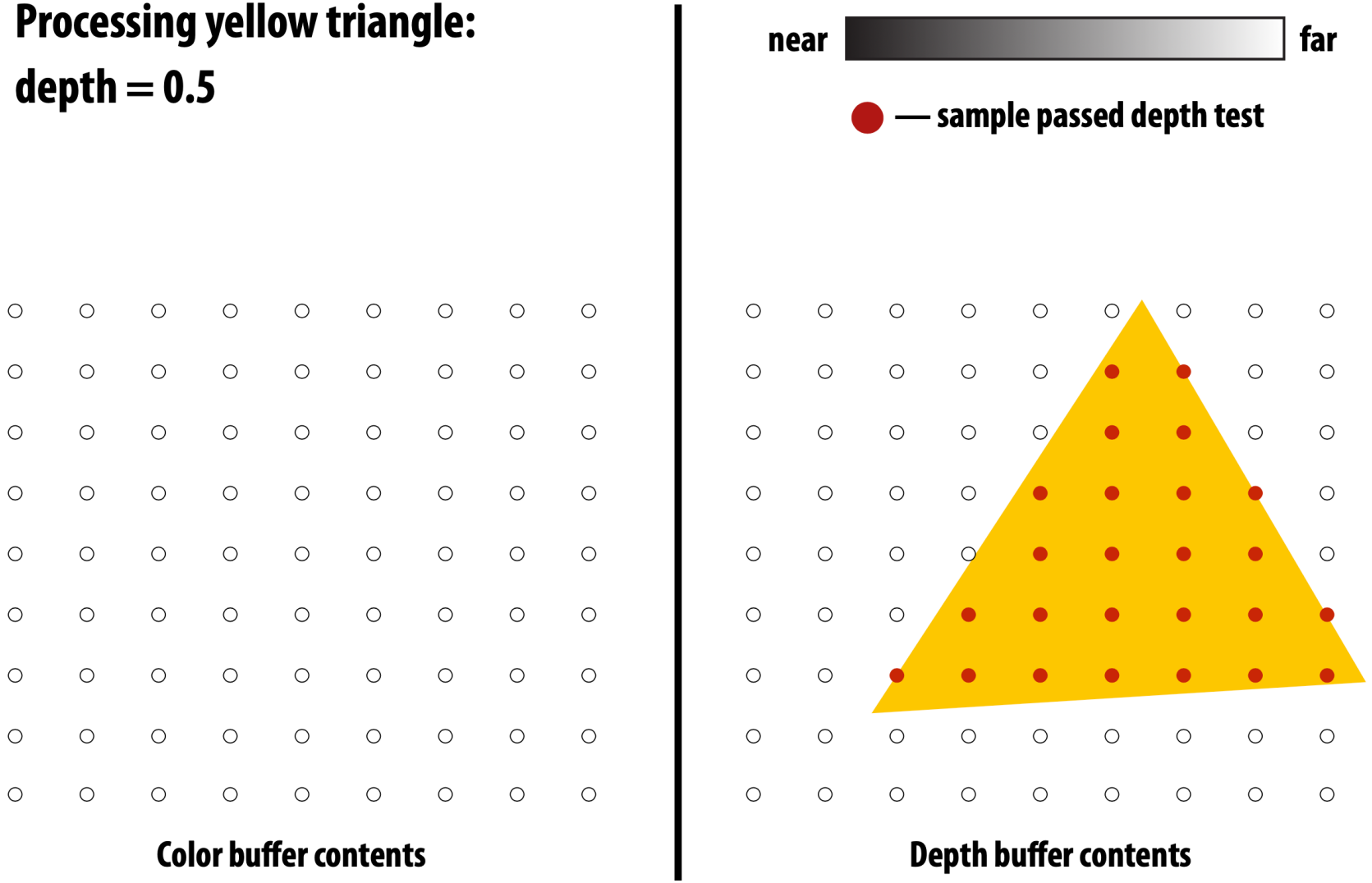
将所有深度缓冲值初始化为“无穷大” （最大值）

深度缓冲示例

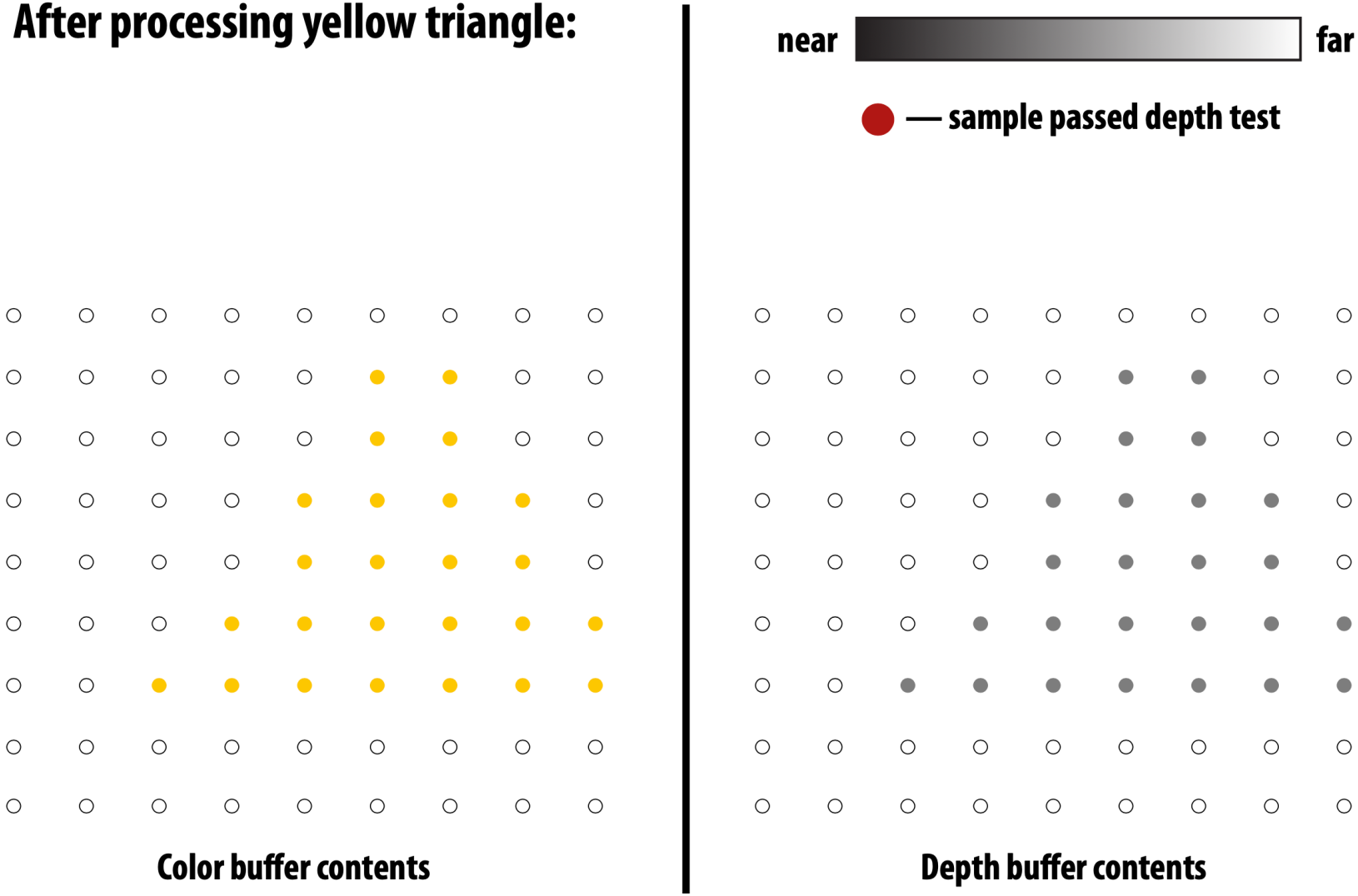
示例：渲染三个不透明三角形



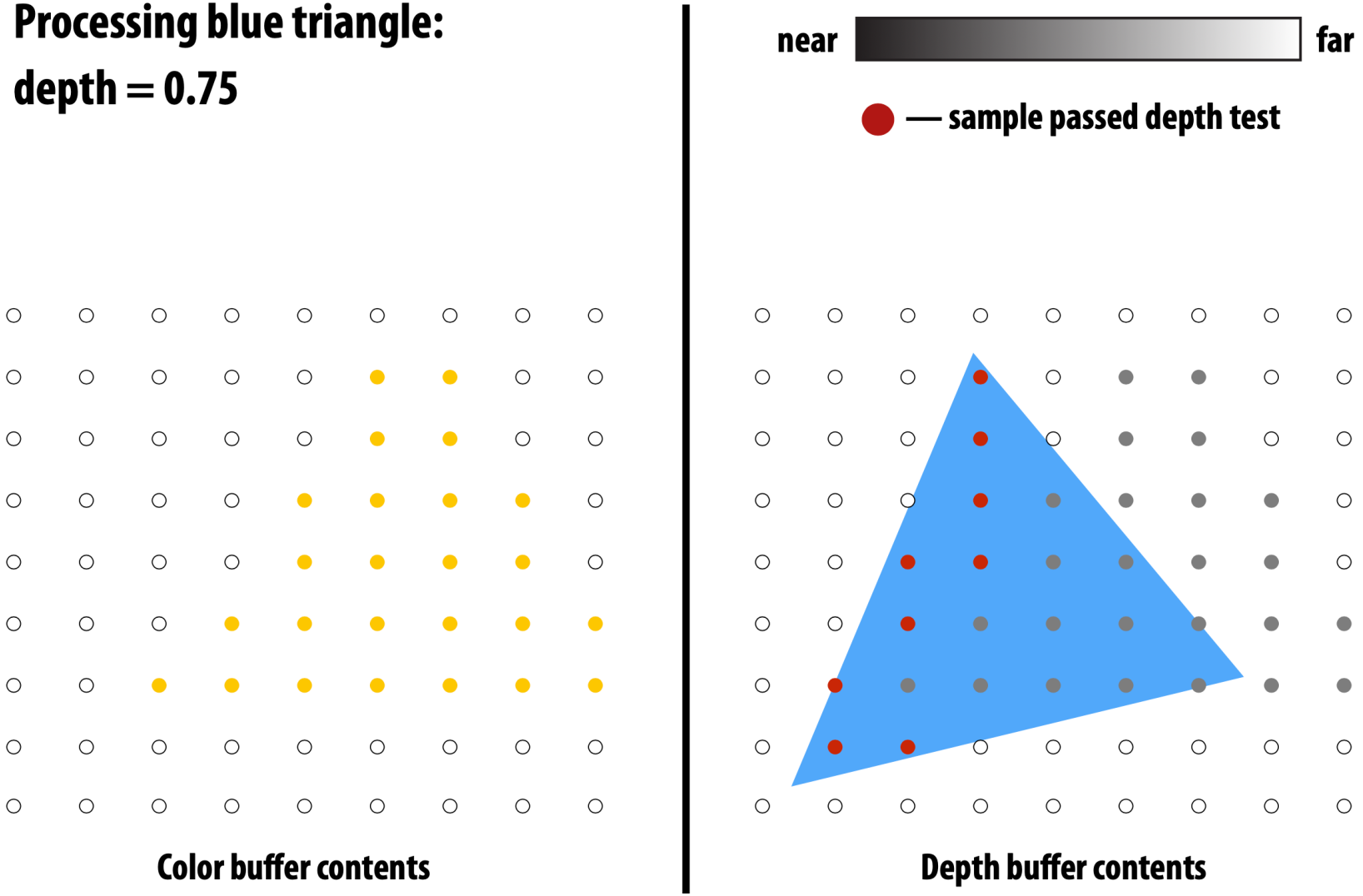
使用深度缓冲 (z-buffer) 进行遮挡



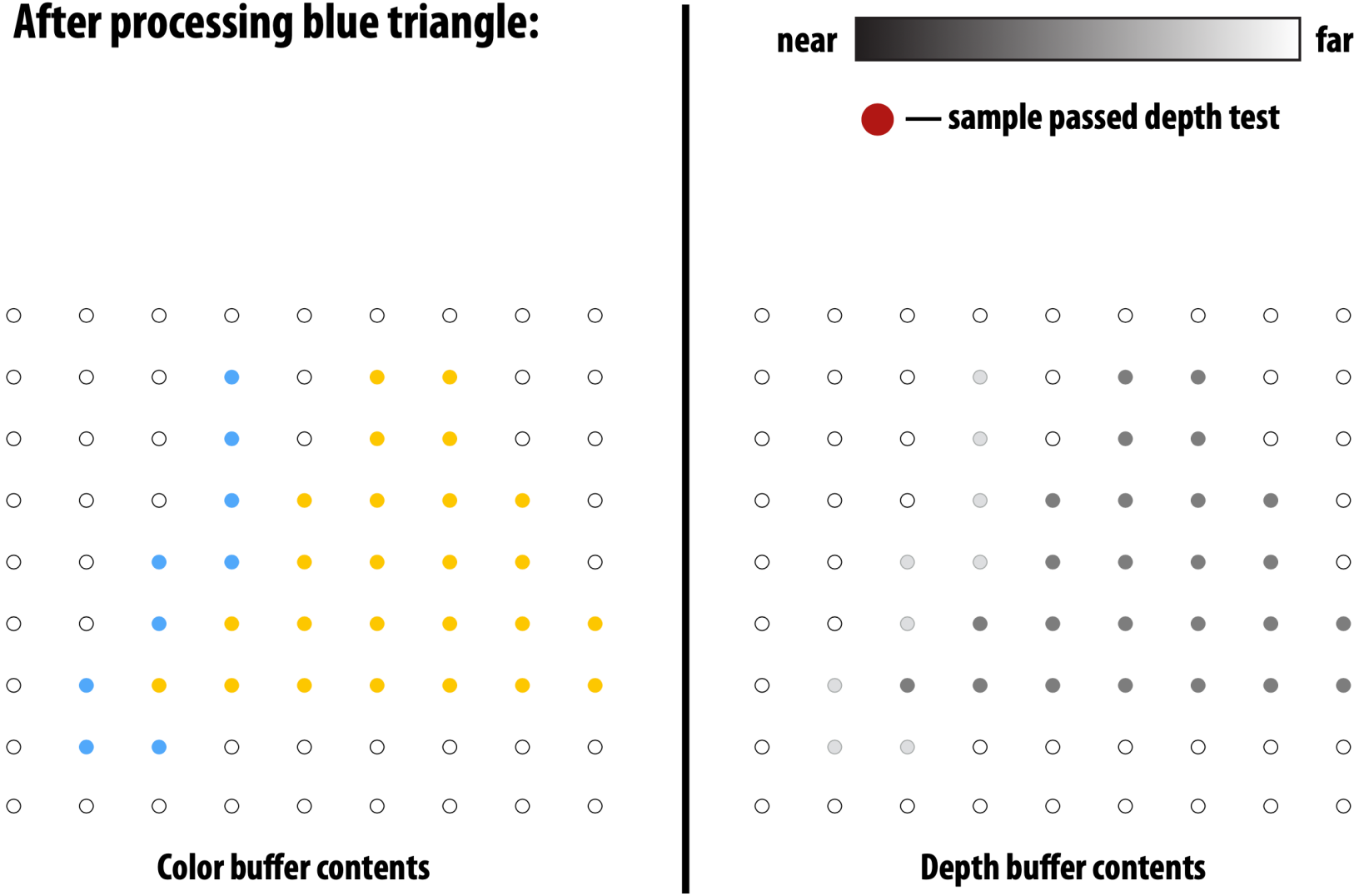
使用深度缓冲 (z-buffer) 进行遮挡



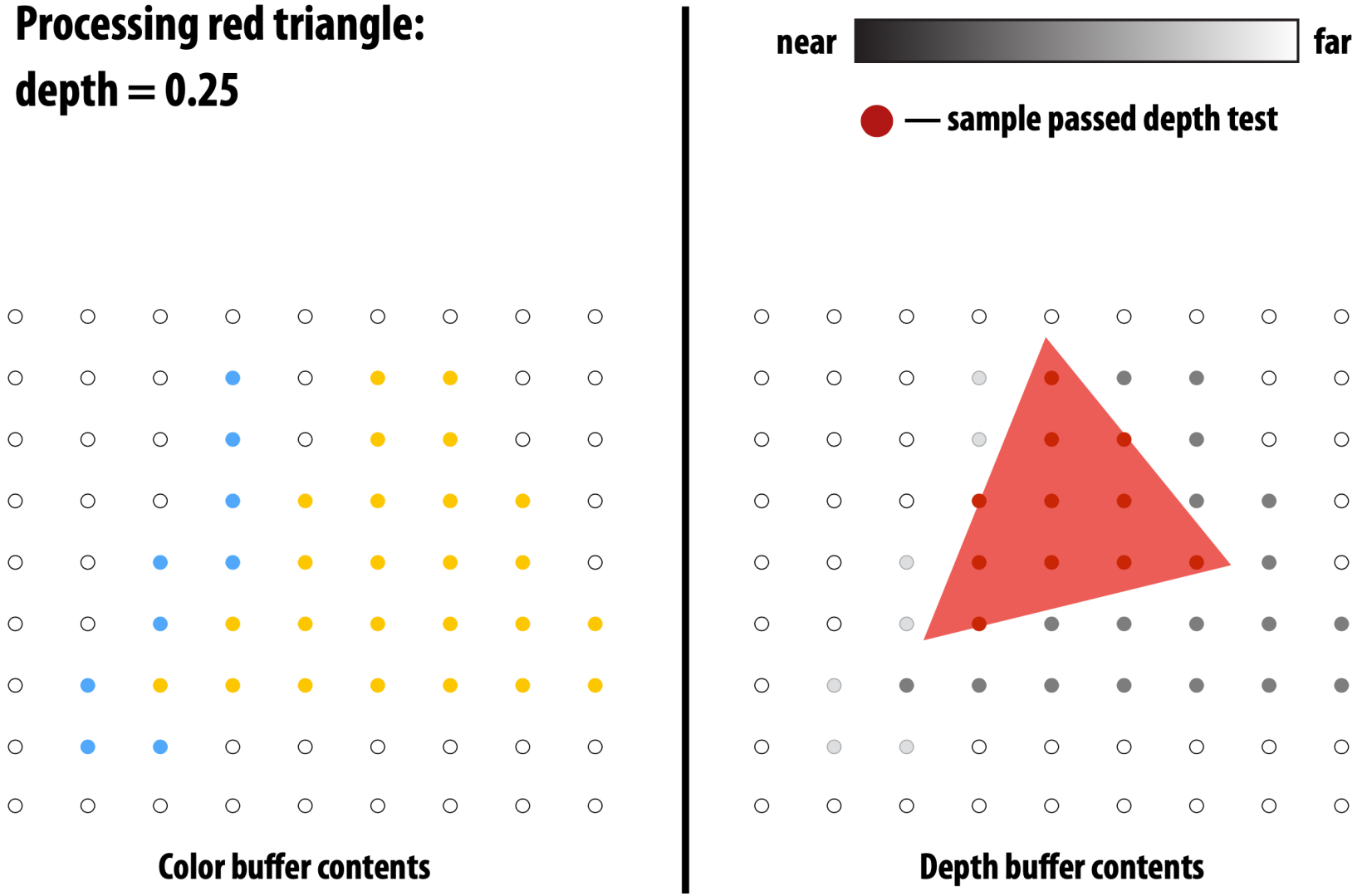
使用深度缓冲 (z-buffer) 进行遮挡



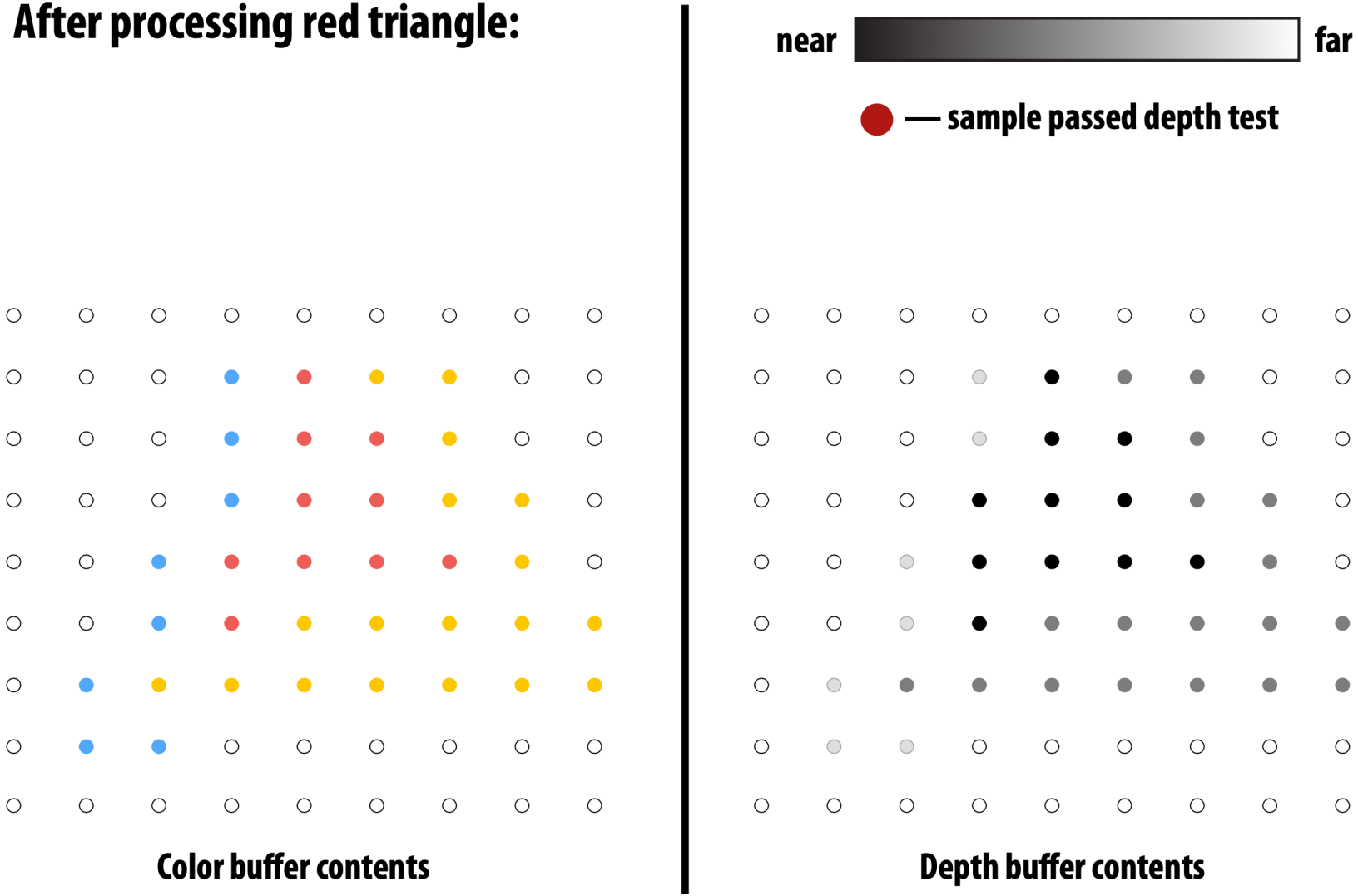
使用深度缓冲 (z-buffer) 进行遮挡



使用深度缓冲 (z-buffer) 进行遮挡



使用深度缓冲 (z-buffer) 进行遮挡



Does the order matter?

使用深度缓冲进行遮挡

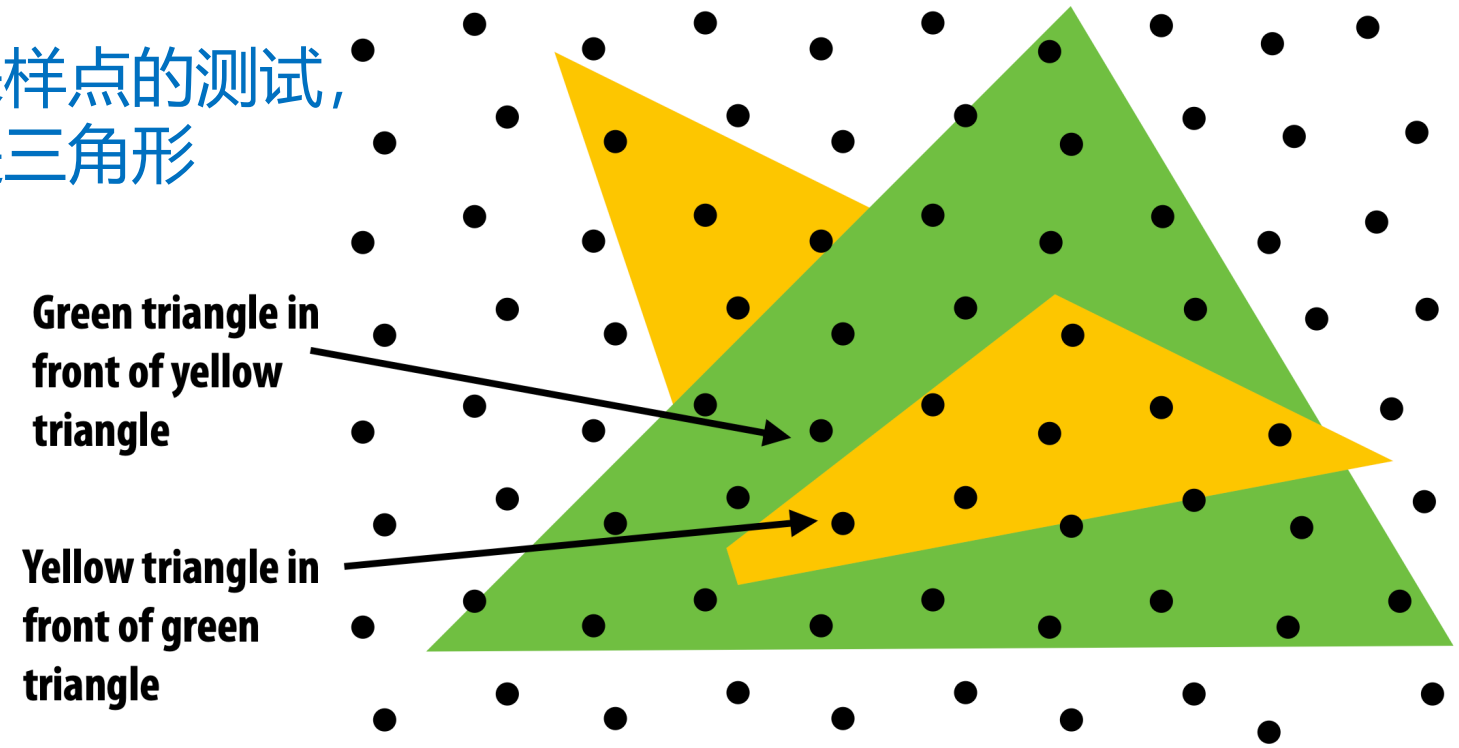
```
bool pass_depth_test(d1, d2)
{
   return d1 < d2;
}
```

```
draw_sample(x, y, d, c) //new depth d & color c at (x,y)
{
  if( pass_depth_test( d, zbuffer[x][y] ))
  {
    // triangle is closest object seen so far at this
    // sample point. Update depth and color buffers.
    zbuffer[x][y] = d;  // update zbuffer
    color[x][y] = c;   // update color buffer
  }
  // otherwise, we've seen something closer already;
  // don't update color or depth
}
```

深度与交叉

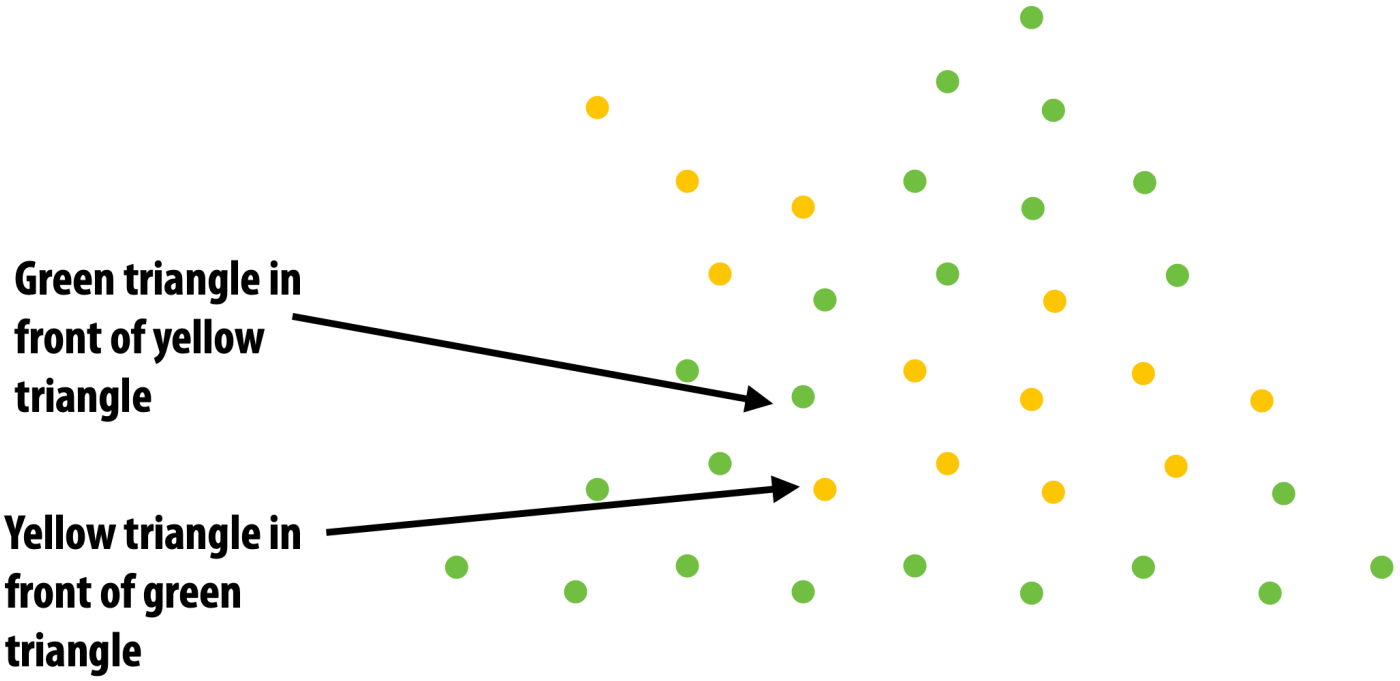
- Q：深度缓冲算法能处理相互穿插的三角形吗？
- 遮挡测试基于给定采样点的三角形深度
- 三角形的相对深度在不同的采样点可能不同

基于采样点的测试，而不是三角形



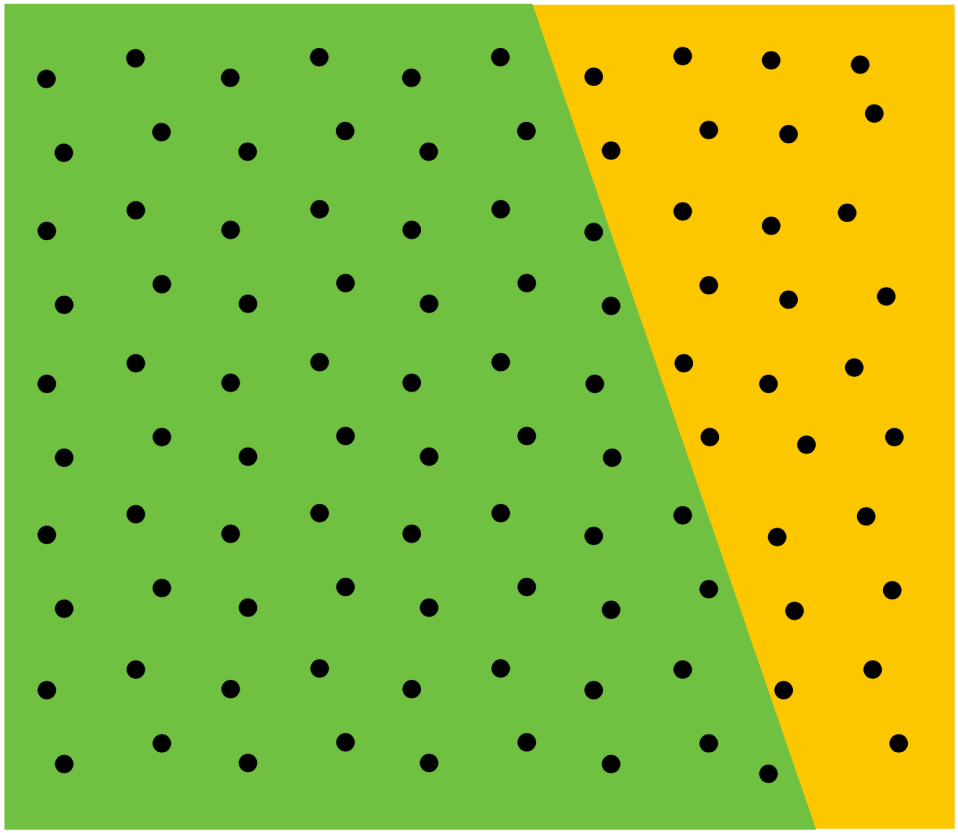
交叉 Intersection

- Q：深度缓冲算法能处理相互穿插的三角形吗？
- 遮挡测试基于给定采样点的三角形深度
- 三角形的相对深度在不同的采样点可能不同



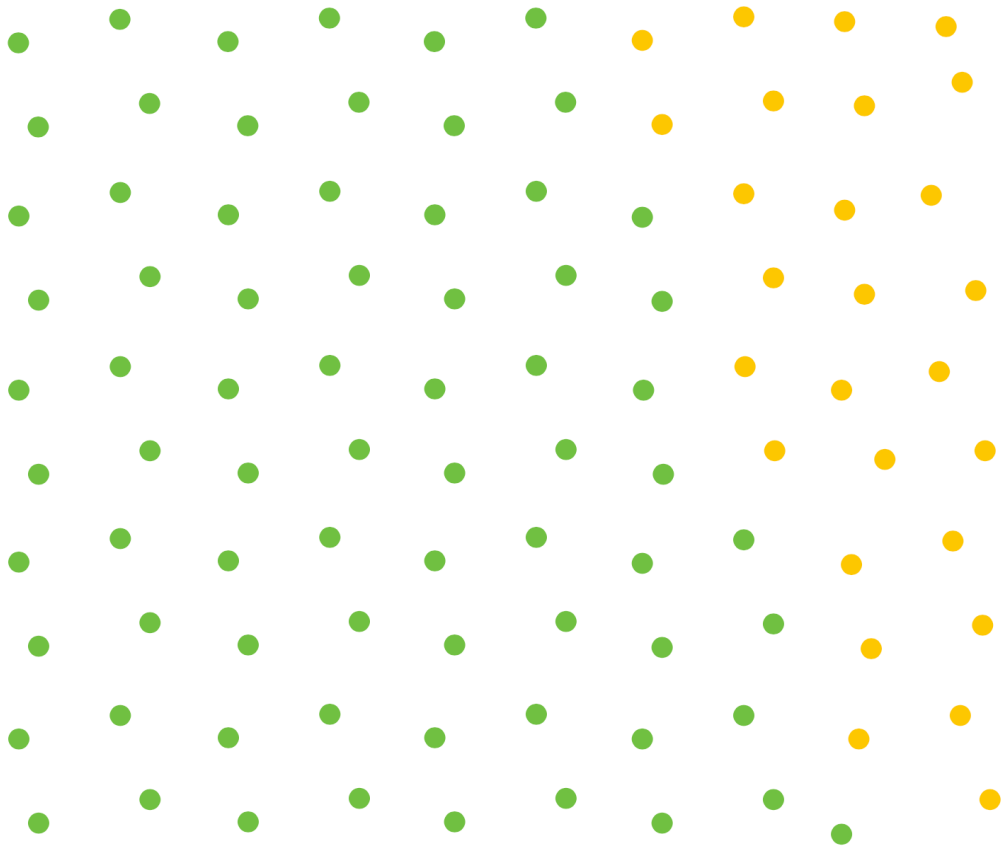
深度与超采样

- Q：深度缓冲能配合超采样 (supersampling) 一起工作吗?
- A：可以，每个超采样点有一个深度的采样



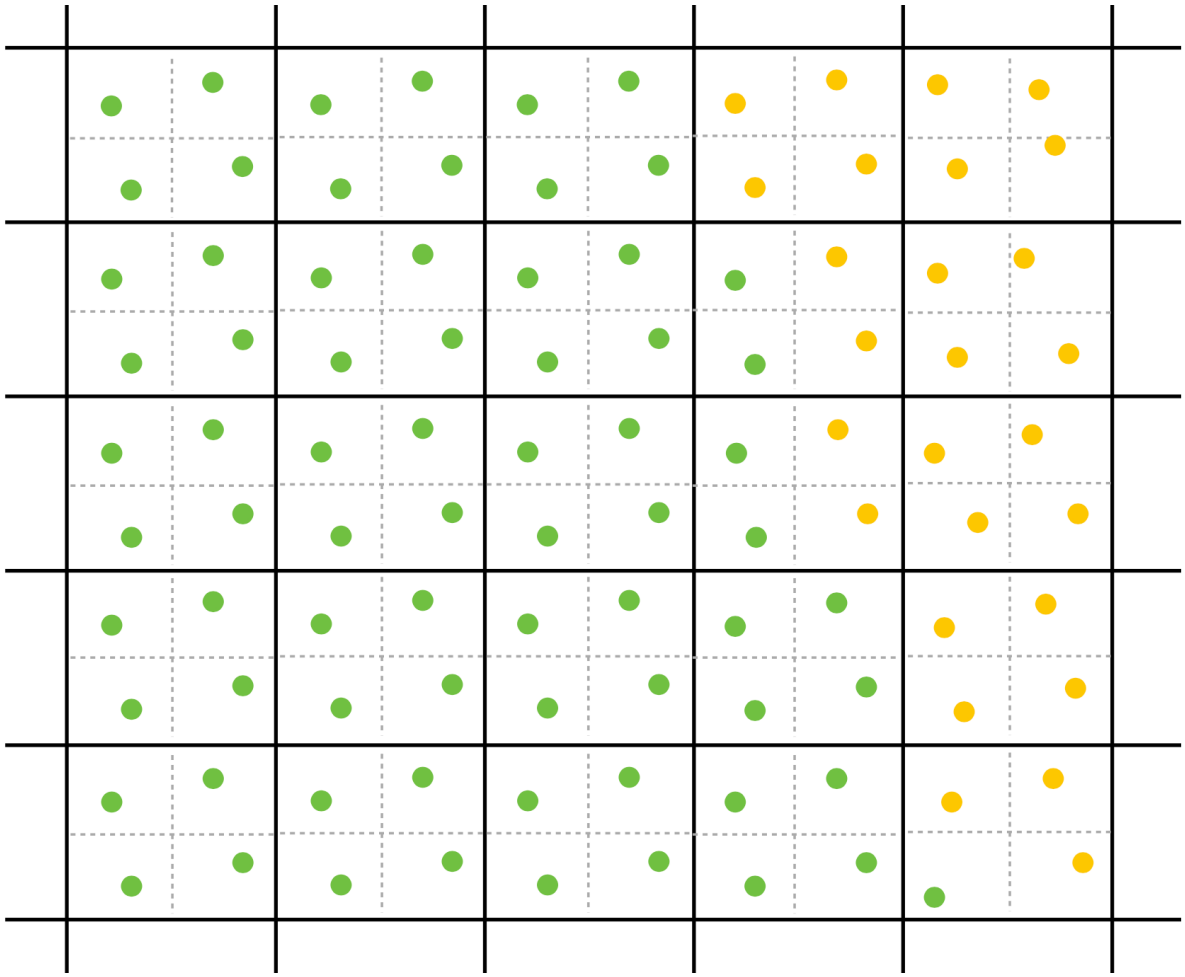
(Here: green triangle occludes yellow triangle)

深度与超采样

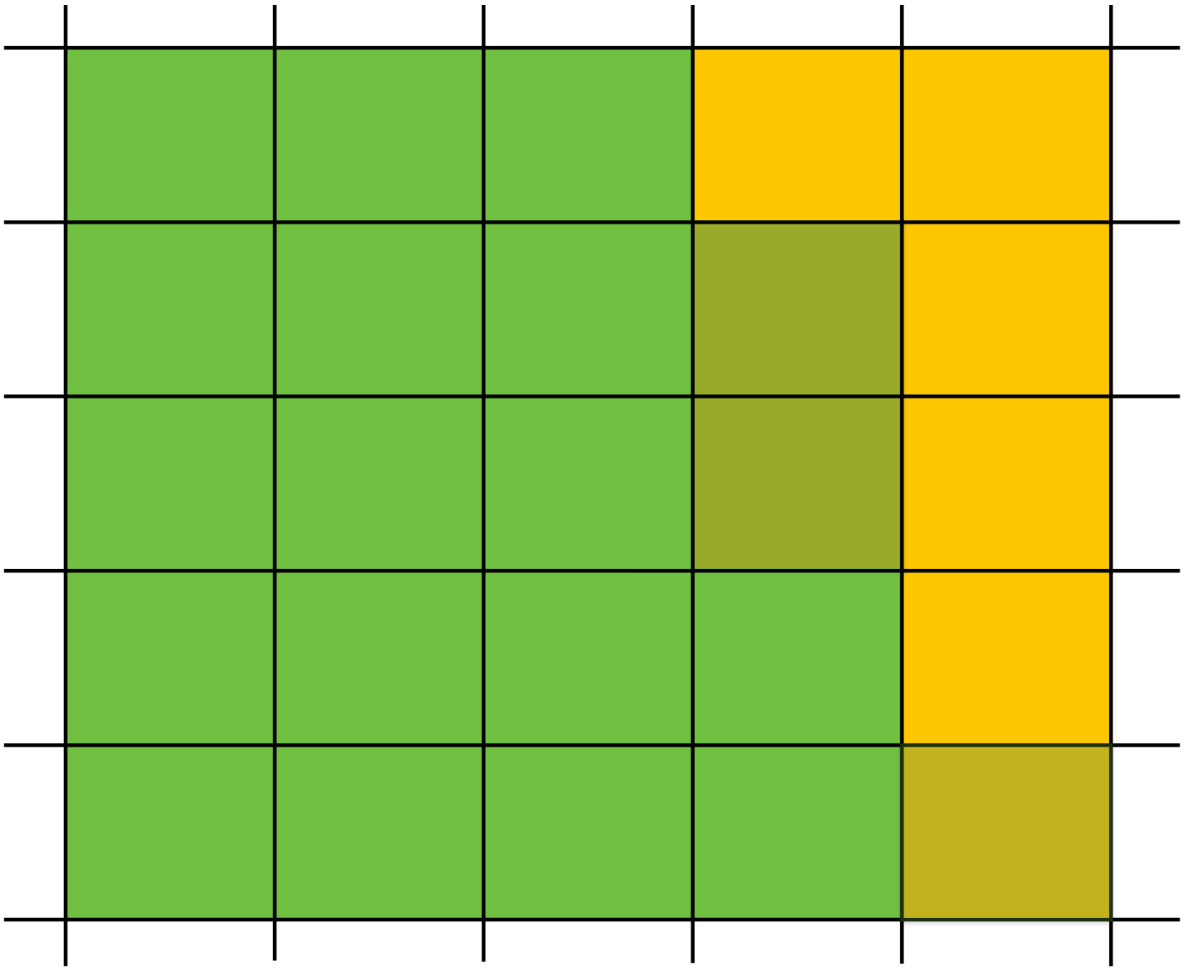


Color of super samples after rasterizing w/ depth buffer

深度与超采样



深度与超采样



Note anti-aliasing of edge due to filtering of green and yellow samples

小结：使用深度缓冲的遮挡

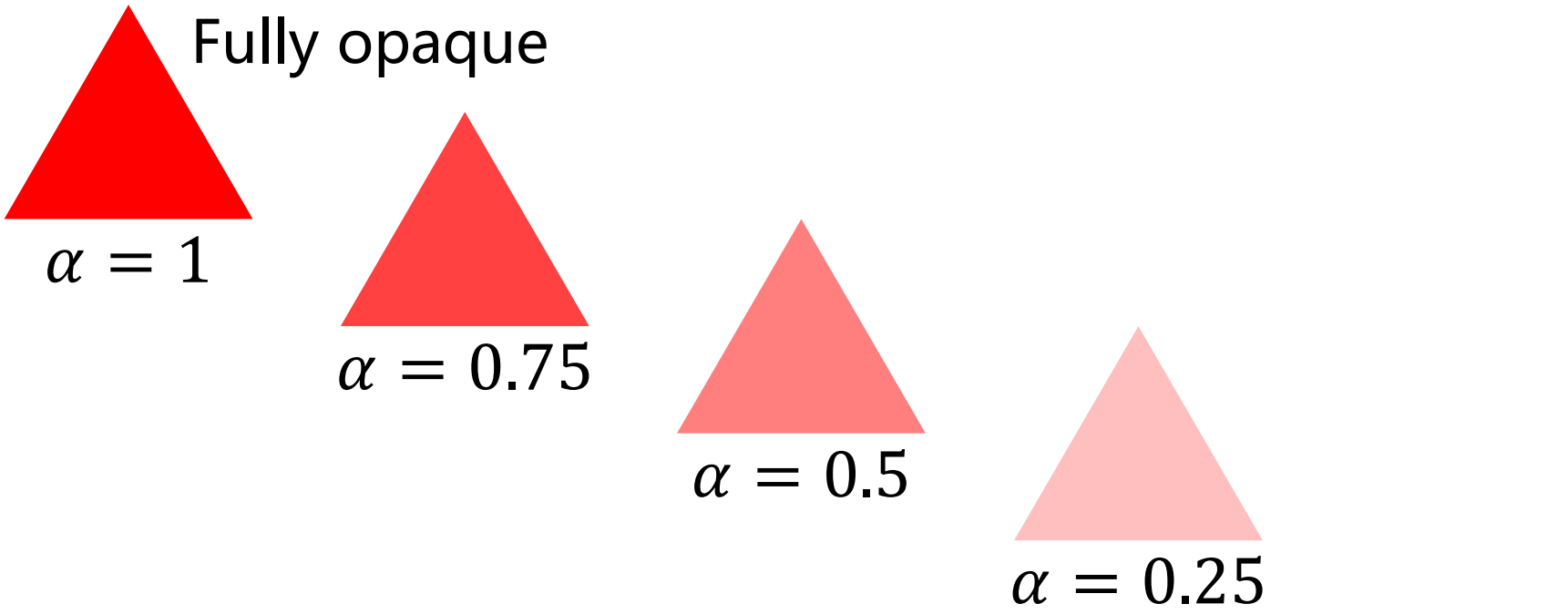
- 每个超样本存储一个深度值，而不是每个像素存储一个！
- 每个样本具有恒定的额外存储空间 (RGB + **Depth**)
  - 因此，深度缓冲所需的空间恒定
  - 不依赖于重叠图形基元的数量 (仅存储一个深度值)！
- 每个样本具有恒定的深度测试时间
  - 如果“通过”深度测试，读取-修改-写入深度缓冲区
  - 如果“失败”，只需读取
- 不特定于三角形：只要求可以在屏幕采样点评估表面深度

对于半透明的表面呢？

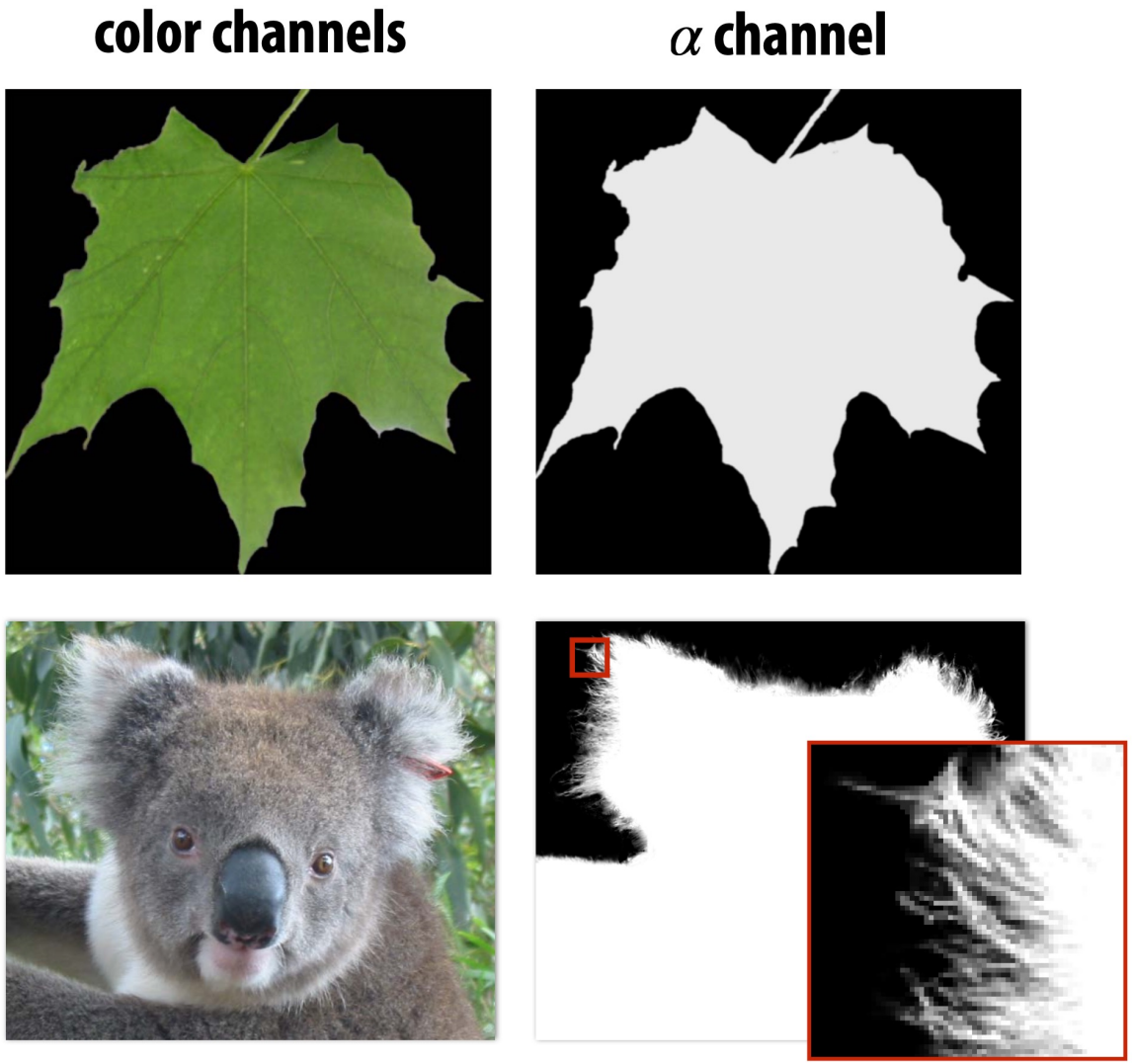
合成
Compositing

将不透明度 opacity 表示为 alpha

- $\alpha, 0 \leq \alpha \leq 1$ 表示对象的不透明度



图像的 alpha 通道

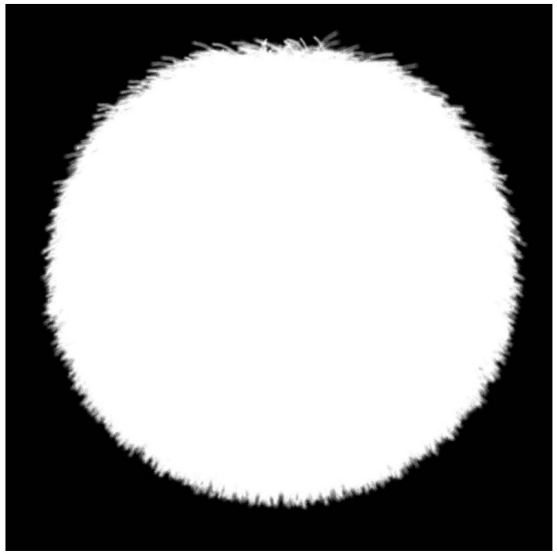


Key idea：我们可以利用图像的 alpha 通道将一张图片叠在另一张图片上以合成新的图片

边缘晕影 Fringing

- 不正确地处理颜色/alpha 值会导致边缘晕影

foreground color



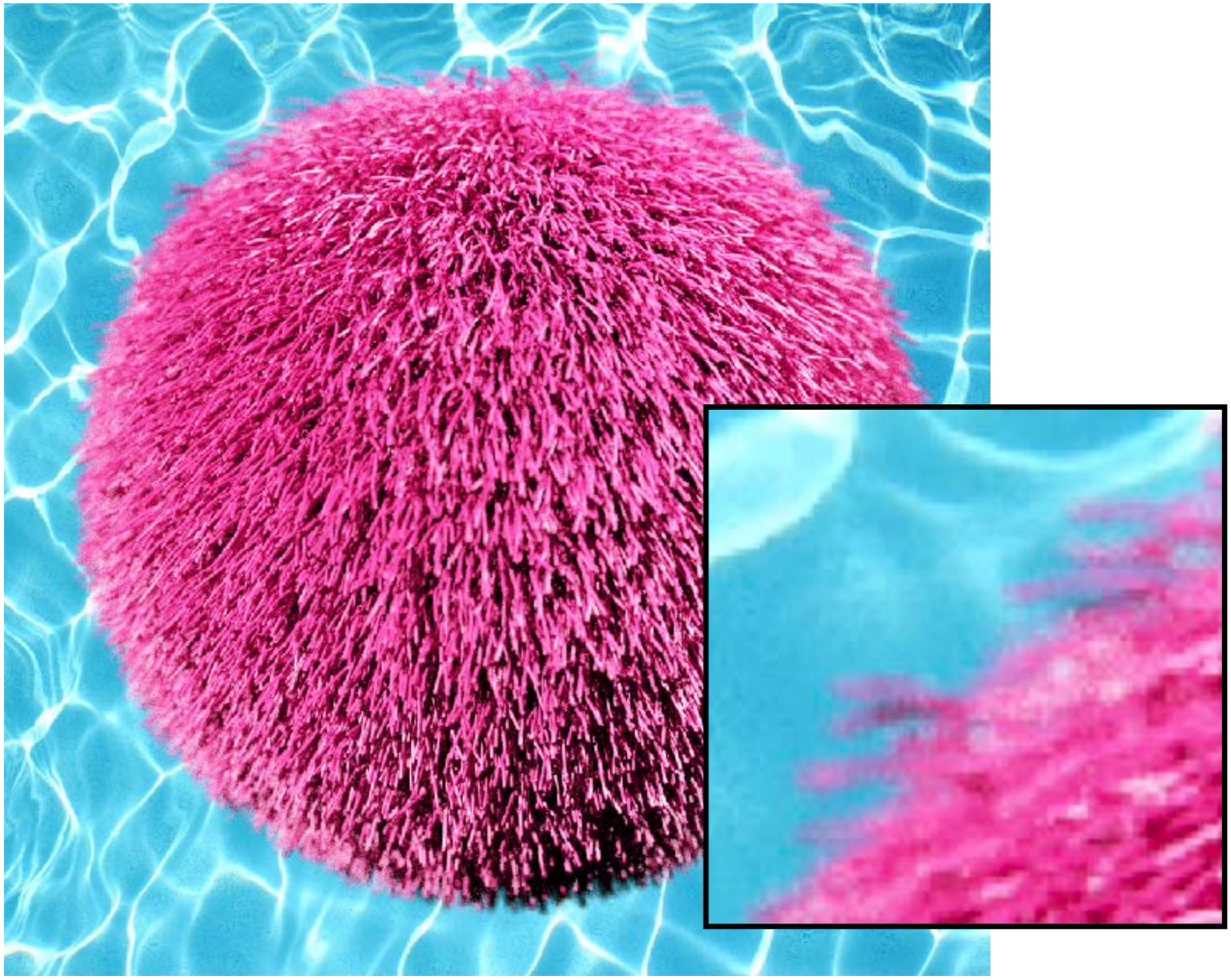
foreground alpha

background color

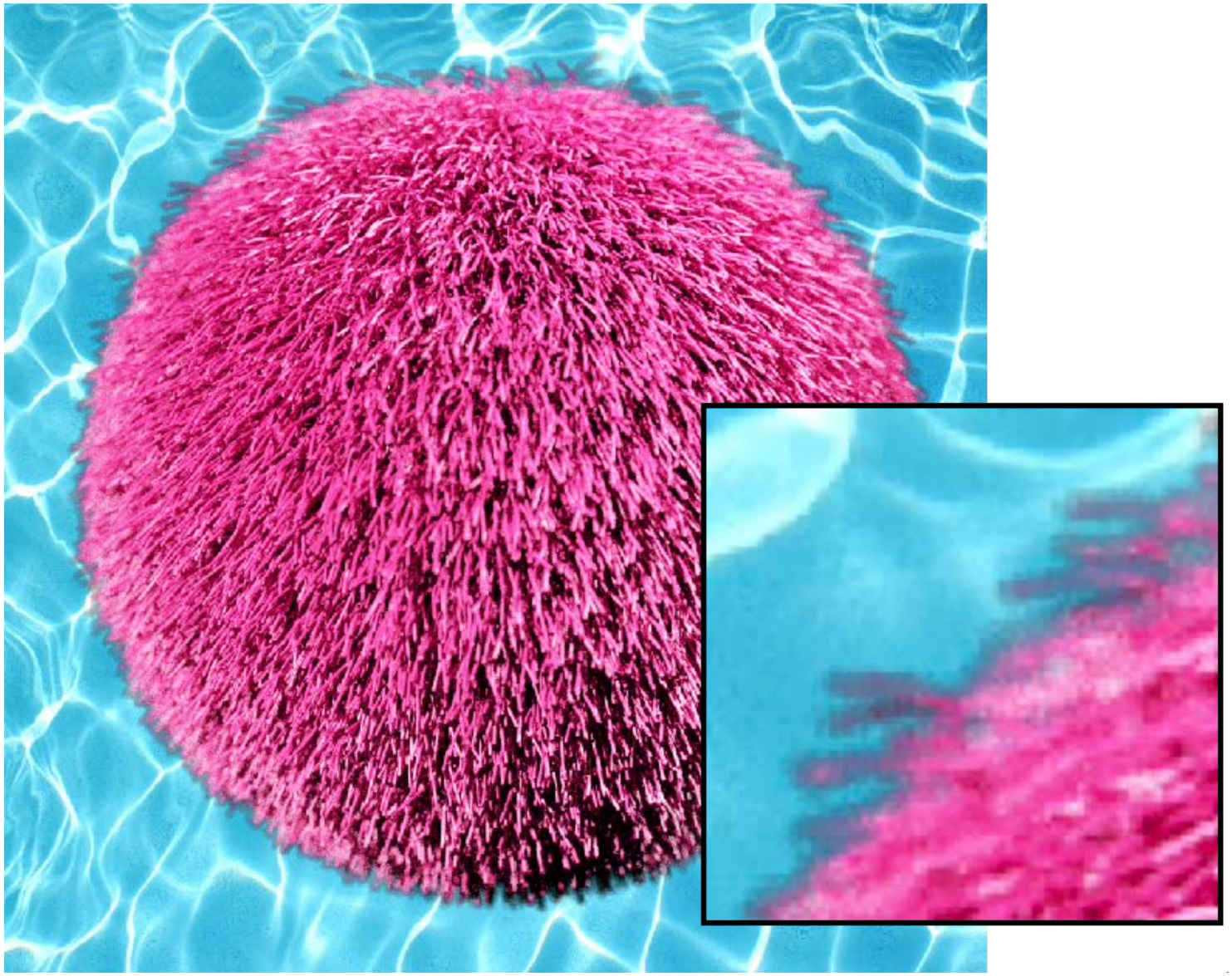
fringing

no fringing

No fringing

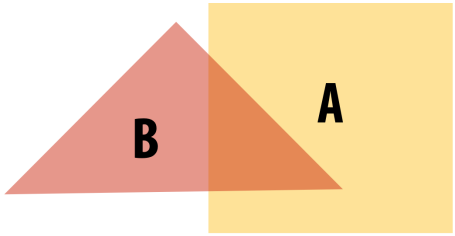


Fringing 为什么会出现?

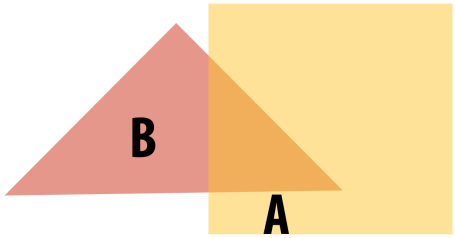


Over 算子 Operator

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



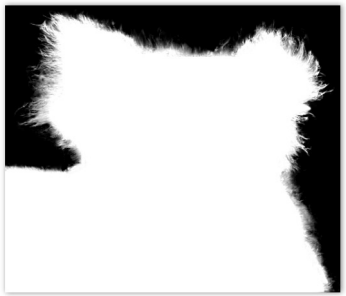
B over A



A over B

Notice: "over" is not commutative

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



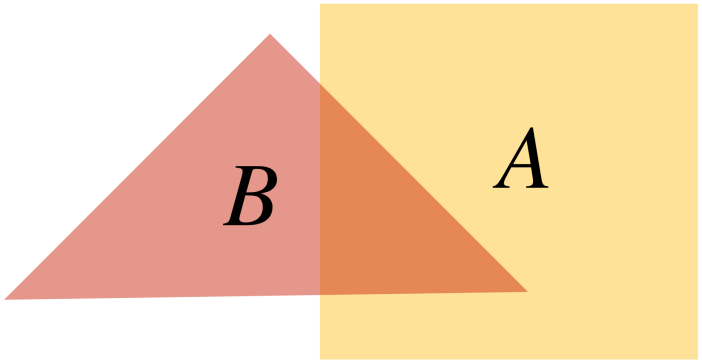
Koala over NYC

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

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

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



B over A

- 混合颜色

B 允许 A 通过的部分

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

半透明 B 的颜色

半透明 A 的颜色

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

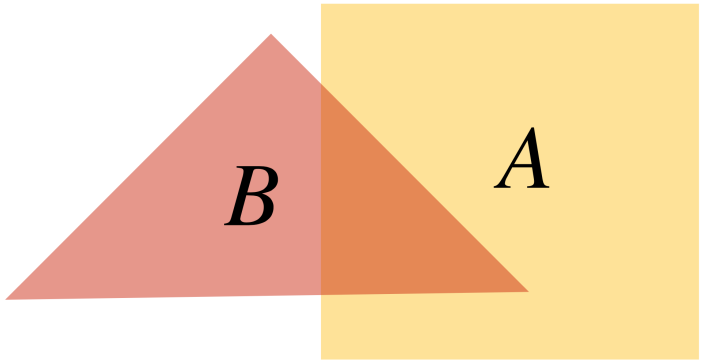
Over 算子：预乘 α (premultiplied α)

- 预乘 α：先将颜色乘以 α，再进行混合

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

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

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



B over 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)

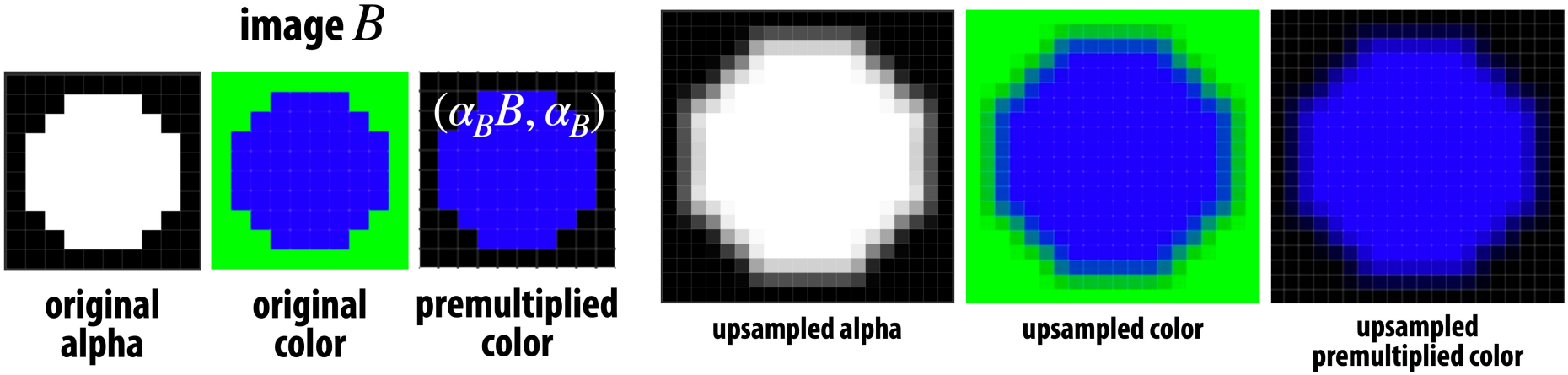
- 以“Un-premultiply”的方式得到最后的颜色 (除以 α_C)

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

Q: Does this division remind you of anything?

使用或不使用预乘 α 进行混合

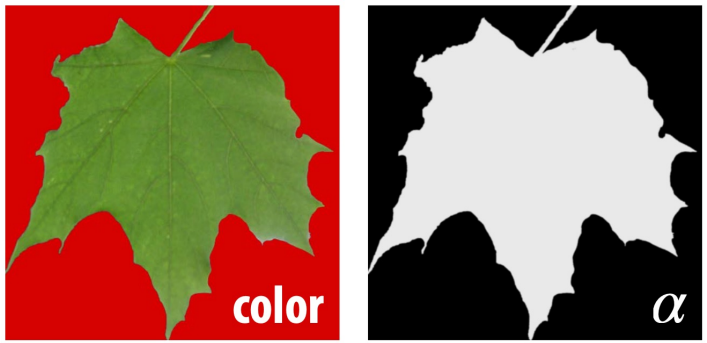
- 采样一张具有 α 通道的照片，再将其与一个背景混合

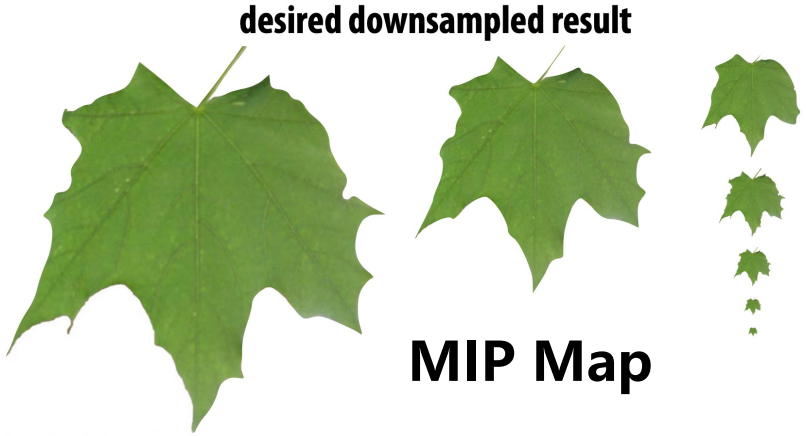


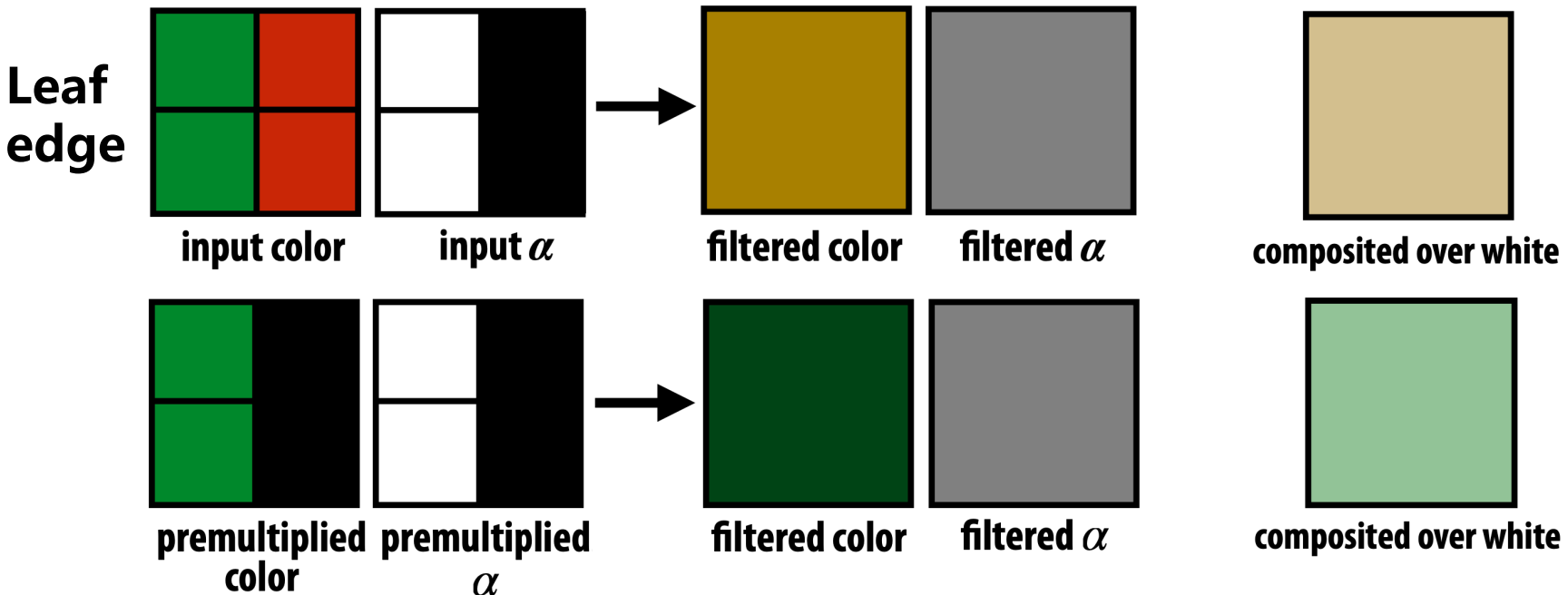
Q: Why do we get the "green fringe" when we don't premultiply?

一个使用非预乘 α 的相似问题

- 考虑下采样 (downsampling) 一个透明度为 α 的纹理

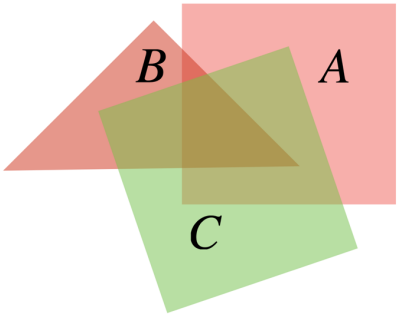






更多问题：重复使用 over 操作

- 将透明度分别为 $\alpha_A, \alpha_B, \alpha_C$ 的照片 A, B, C 进行混合
- 预乘 α 在混合操作下是闭合 (closed) 的，非预乘 α 不是
- 例子，将 50% 的红色与 50% 的红色混合
  (其中 红色=(1, 0, 0), $\alpha = 0.5$)



C over B over A

non-premultiplied

color

.5(1,0,0) + (1-.5).5(1,0,0)

⇓

(0.75,0,0) too dark!

alpha

.5 + (1-.5).5 = .75

premultiplied

color

(.5,0,0,.5)+(1-.5)(.5,0,0,.5)

⇓

(.75,0,0.75)

⇓ divide by α

bright red (1,0,0)

alpha

$\alpha = 0.75$

若不对非预乘α的结果进行调整，将越来越黑，预乘α的结果仍是合法的预乘形式(closed)，因此能正常趋近于红色

预乘 α 的优点

- 合成操作对所有通道 (颜色及 α) 一视同仁
- 与非预乘表示相比，over 运算的算术运算更少
- 在合成操作中是闭合的（重复 over 运算的结果是合法的）
- 是使用 α 通道过滤（上采样/下采样）图像更好的表示
- 无缝衔接光栅化流程（齐次坐标）

绘制半透明图形基元的策略

- 假设所有基元都是半透明的，并且颜色值已经预乘了 α，下面是光栅化图像的策略：

```
over(c1, c2)
{
   return c1.rgba + (1-c1.a) * c2.rgba;
}
```

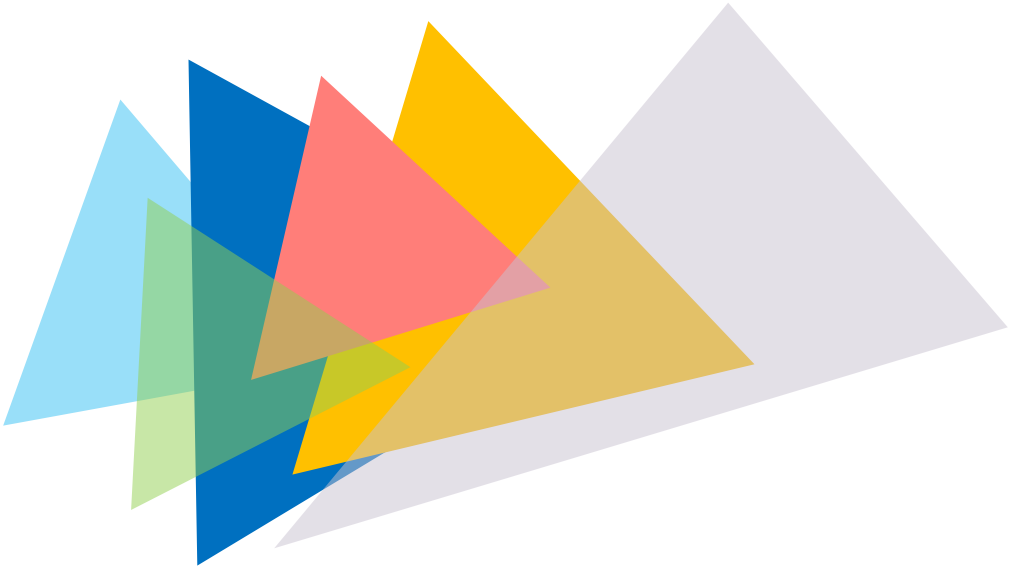
```
update_color_buffer( x, y, sample_color, sample_depth )
{
   if (pass_depth_test(sample_depth, zbuffer[x][y])
   {
       // (how) should we update depth buffer here??
       color[x][y] = over(sample_color, color[x][y]);
  }
}
```

Q：上述操作的隐含假设是什么？

A：三角形必须按从后往前的顺序进行渲染！

Putting it all together

- 如果我们同时有不透明和透明的三角形呢?
- 第一步
  - 使用深度缓冲遮挡渲染不透明的基元（按任何顺序）
  - 若通过深度测试，三角形将覆盖采样处颜色缓冲区中的值
- 第二步:
  - 禁用深度缓冲更新，按**从后往前**的顺序渲染半透明基元
  - 若通过深度测试，三角形将基于样本缓冲区中的颜色合成



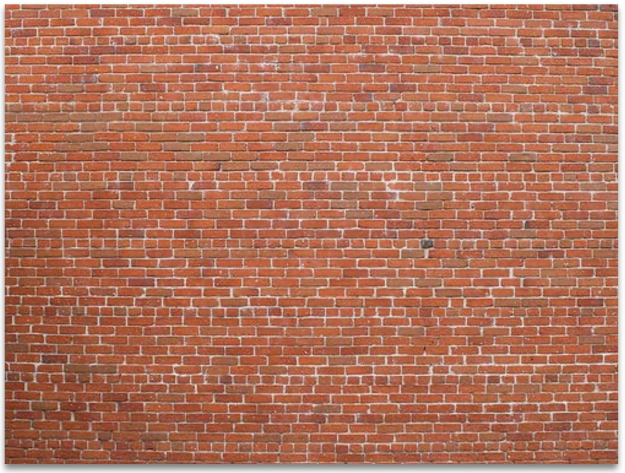
点到点的光栅化流程
End-to-end rasterization pipeline

目标：将输入变成图像

Inputs:

```
positions = {
    v0x, v0y, v0z,
    v1x, v1y, v1x,
    v2x, v2y, v2z,
    v3x, v3y, v3x,
    v4x, v4y, v4z,
    v5x, v5y, v5x
};
```

```
texcoords ={
    v0u, v0v,
    v1u, v1v,
    v2u, v2v,
    v3u, v3v,
    v4u, v4v,
    v5u, v5v
};
```



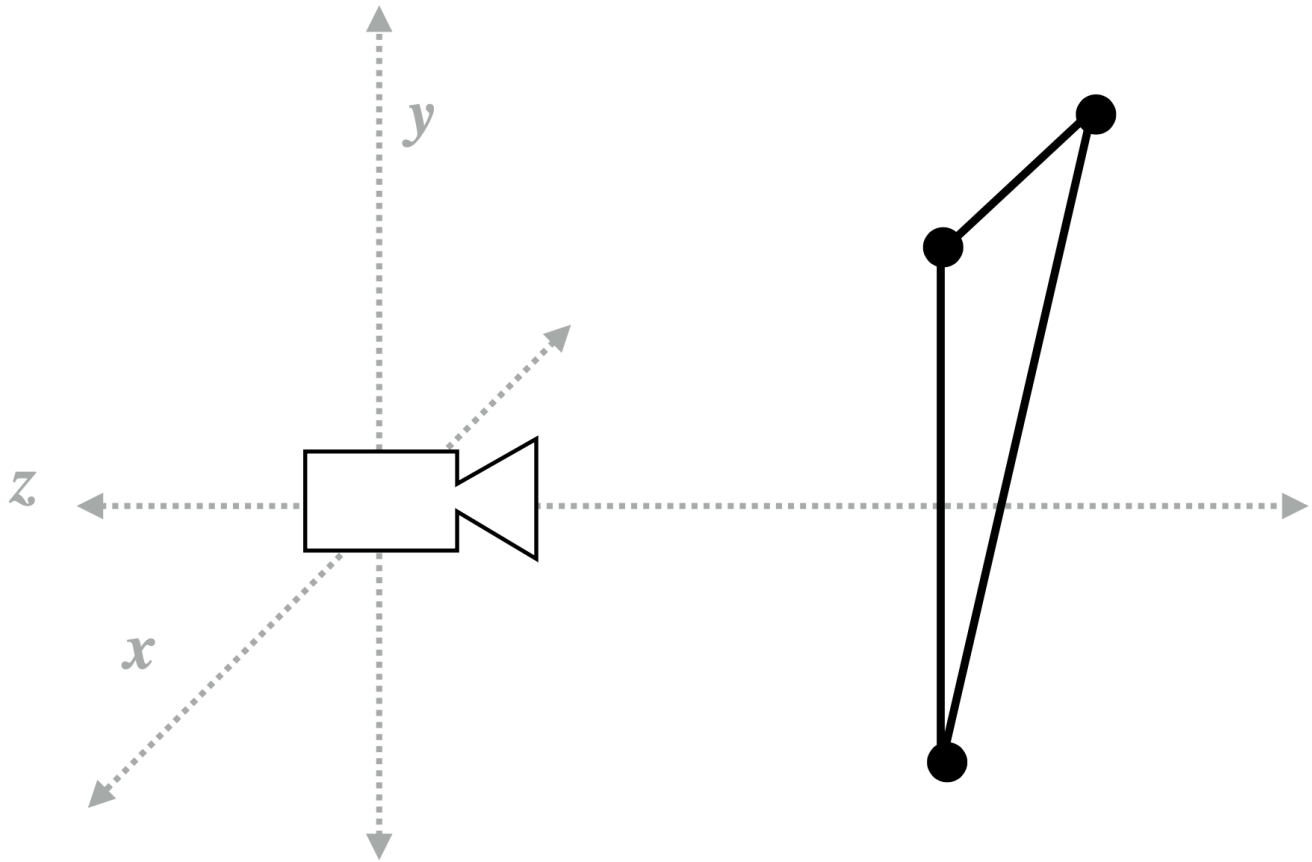
texture map

- 将物体转换到相机坐标：$T \in \mathbb{R}^{4\times4}$
- 透视投影转换： $P \in \mathbb{R}^{4\times4}$
- 显示器大小：$(W \times H)$

目前为止，我们已经有了制作图像所需的所有工具
让我们复习一下！

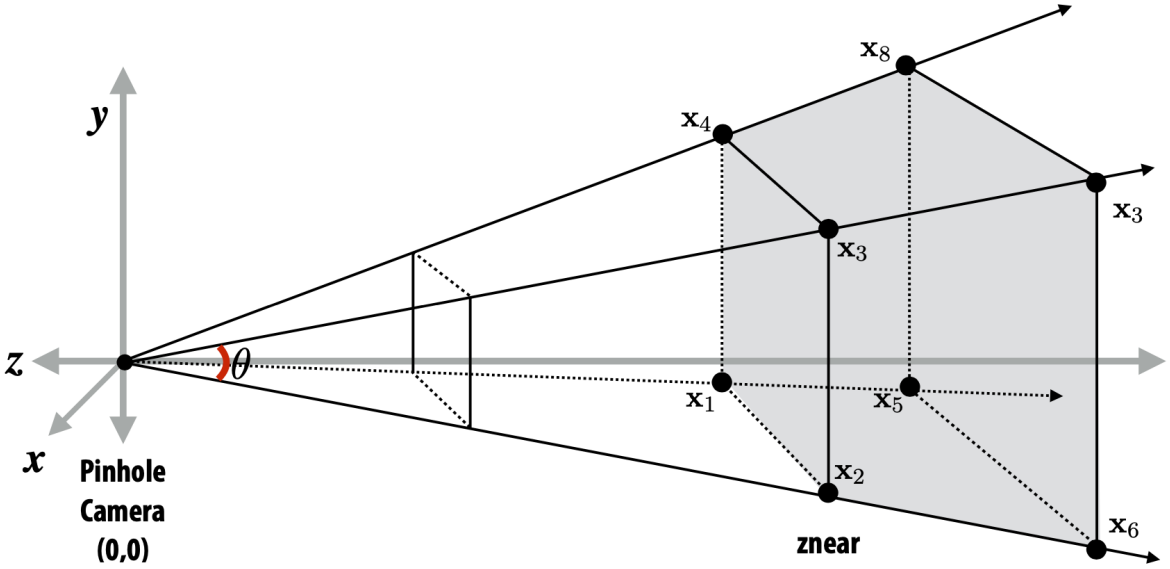
Step: 1

- 将三角形的顶点转换到相机坐标系中

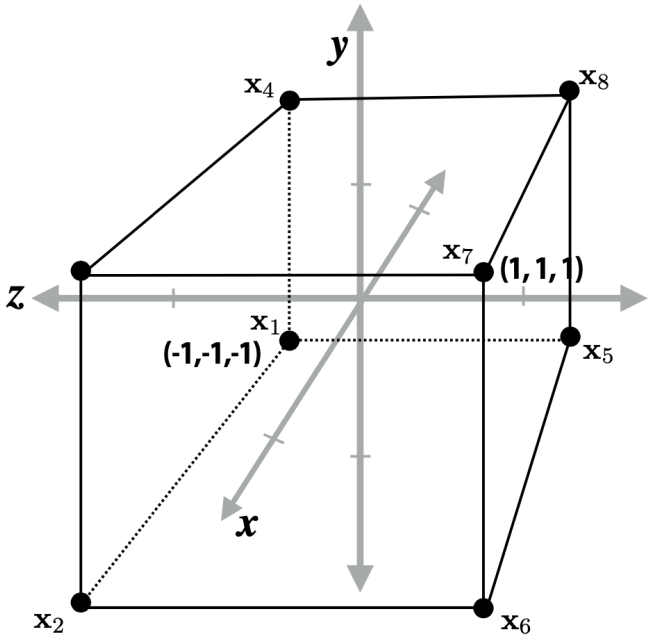


Step: 2

口应用透视投影转换将三角形顶点转换到归一化的坐标空间



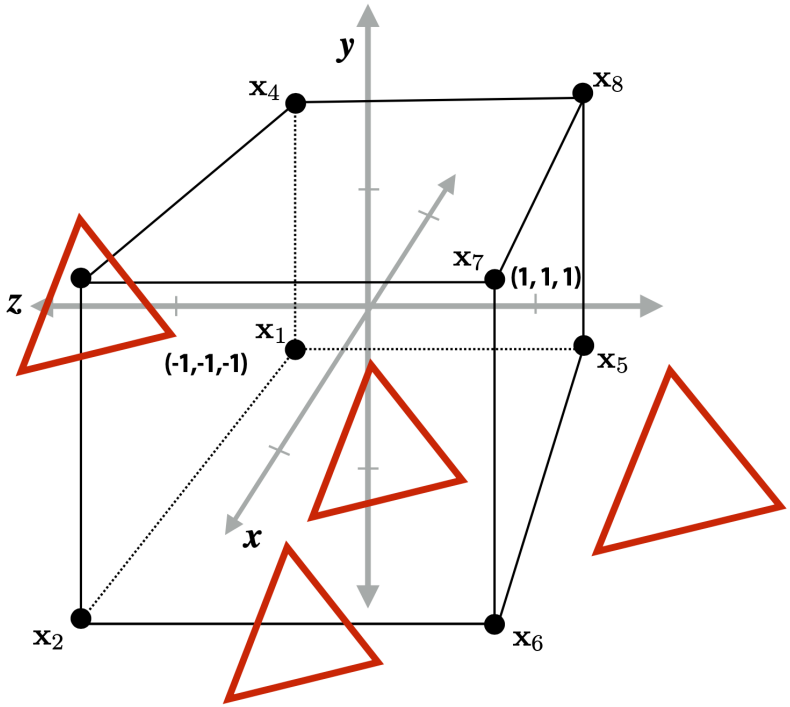
Camera-space positions: 3D



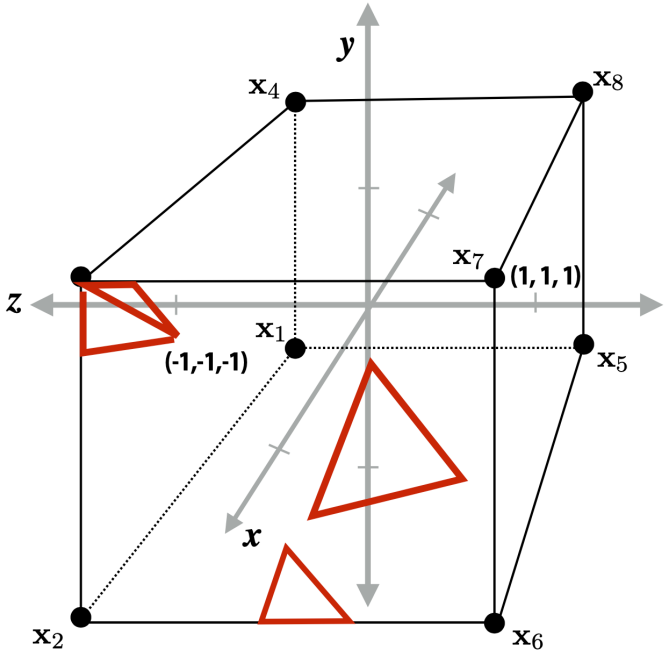
Normalized space positions

Step 3: 裁剪 Clipping

- 丢弃完全位于单位立方体之外的三角形（剔除 culling）
  - 它们不在屏幕上，不用再处理了
- 剪裁超出单位立方体的三角形
  - 可能会生成新的三角形



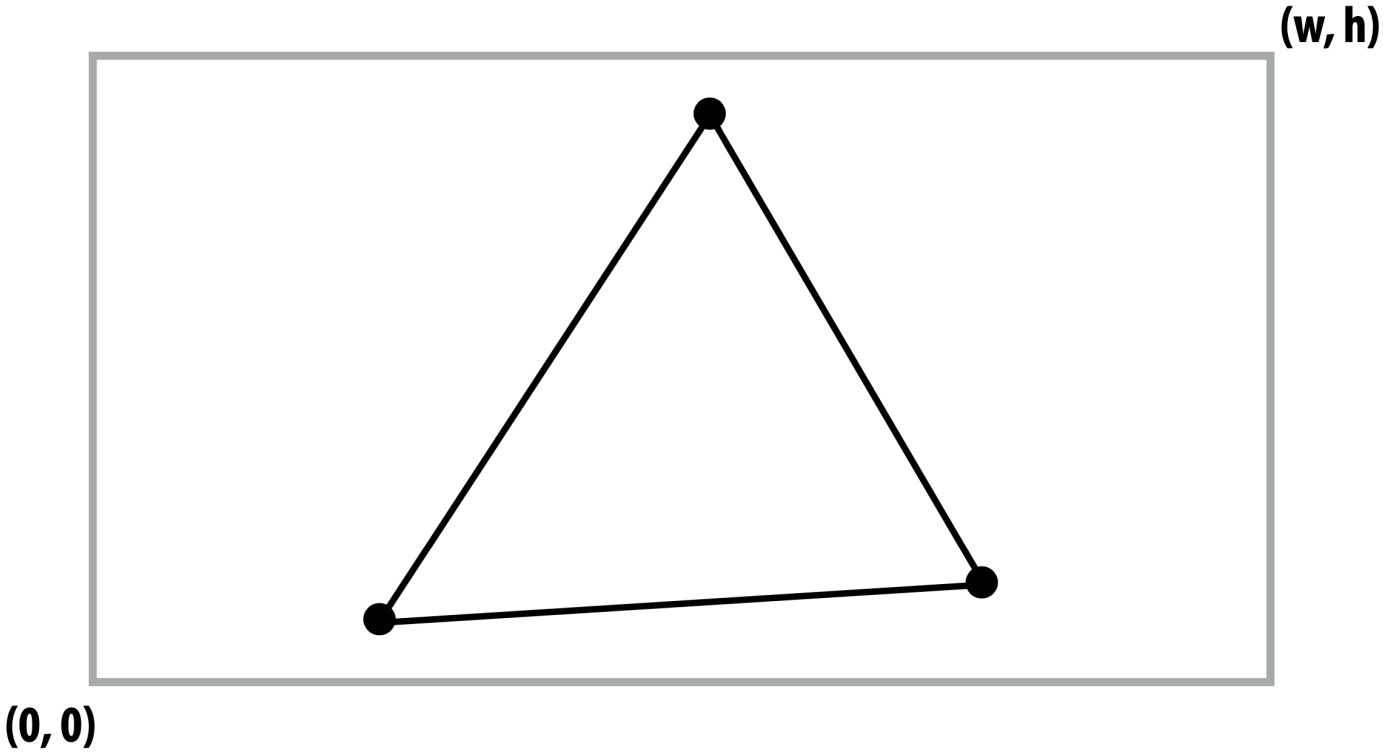
Triangles before clipping



Triangles after clipping

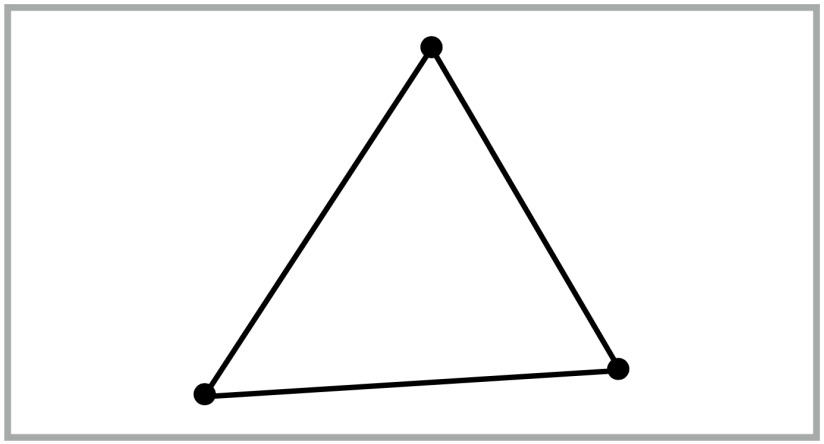
Step 4: 转换到屏幕坐标

- 执行齐次分割，将顶点 (x, y) 位置从归一化坐标转换为屏幕坐标 (屏幕大小为 $W \times H$)



Step 5: 三角形预处理

- 在光栅化三角形之前，可以预先计算一系列将被反复用到的函数，比如：
  - 三角形边的公式
  - 三角形属性的公式
  - …



$$\mathbf{E}_{01}(x,y) \qquad \mathbf{U}(x,y)$$

$$\mathbf{E}_{12}(x,y) \qquad \mathbf{V}(x,y)$$

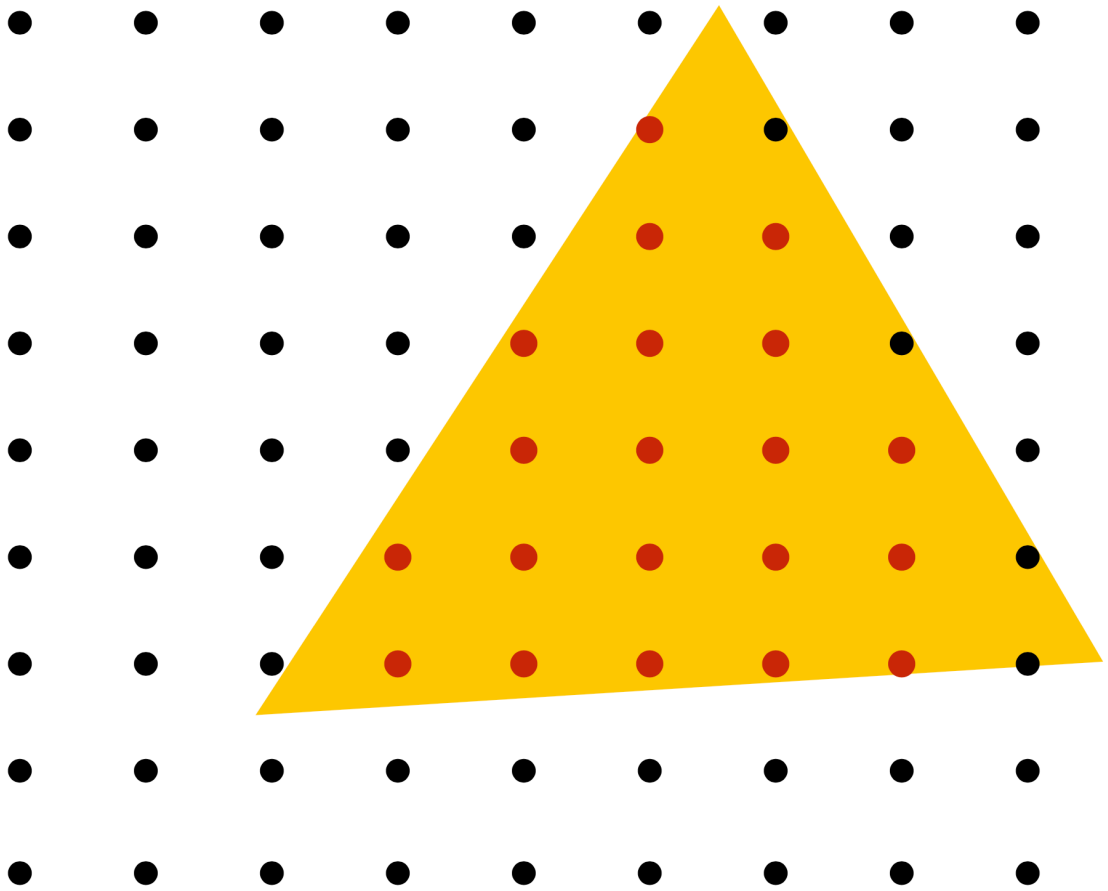
$$\mathbf{E}_{20}(x,y)$$

$$\frac{1}{\mathbf{w}}(x,y)$$

$$\mathbf{Z}(x,y)$$

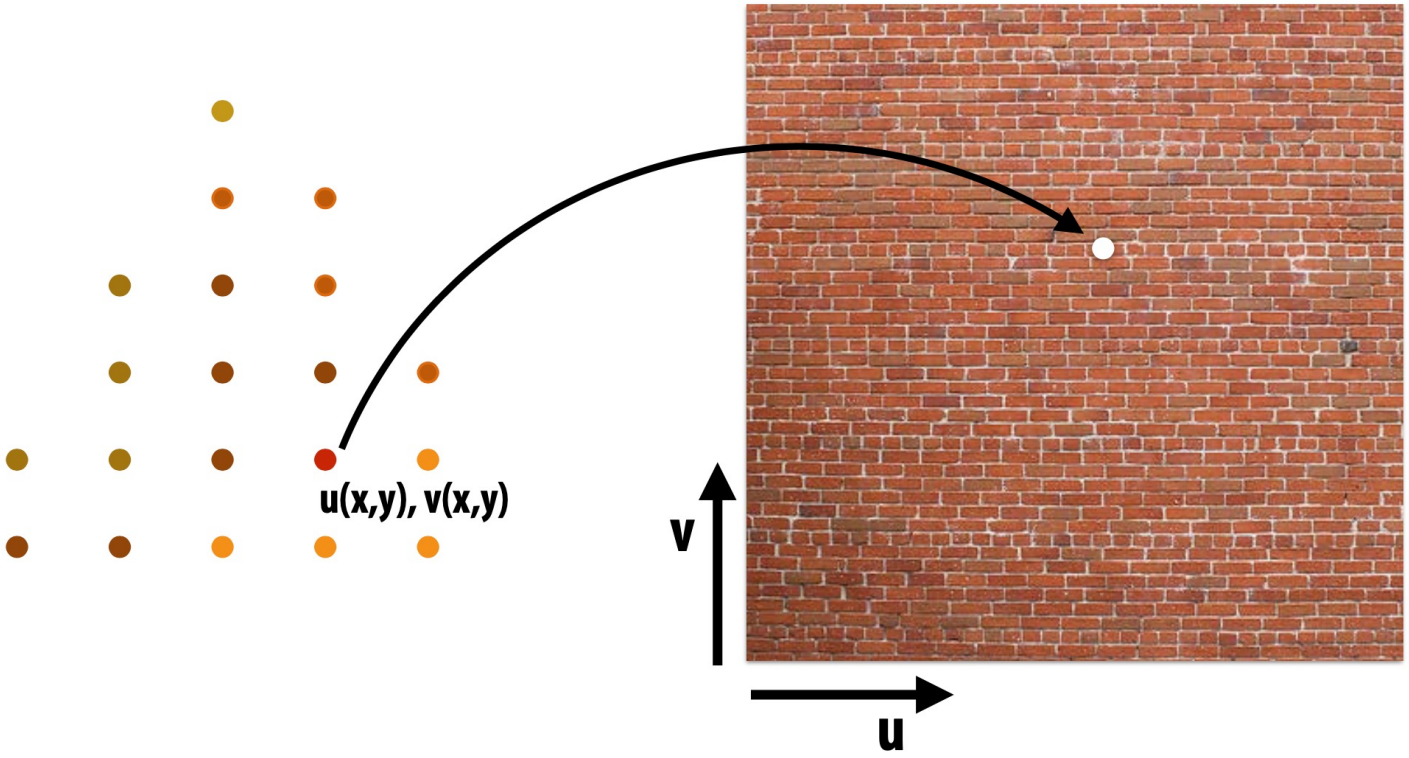
Step 6: 采样覆盖

- 在所有覆盖的样本评估属性



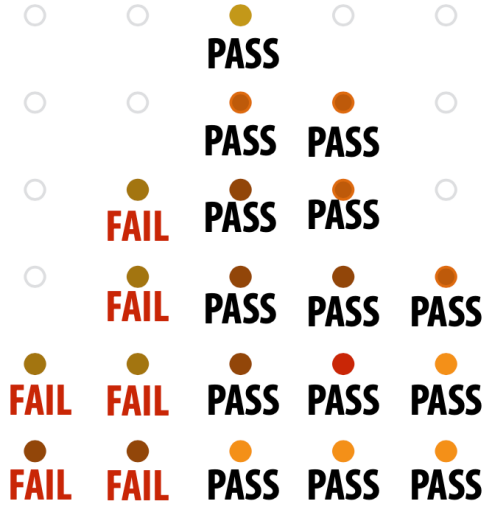
Step 6: 在样本点计算三角形的颜色

- 比如，采样纹理映射

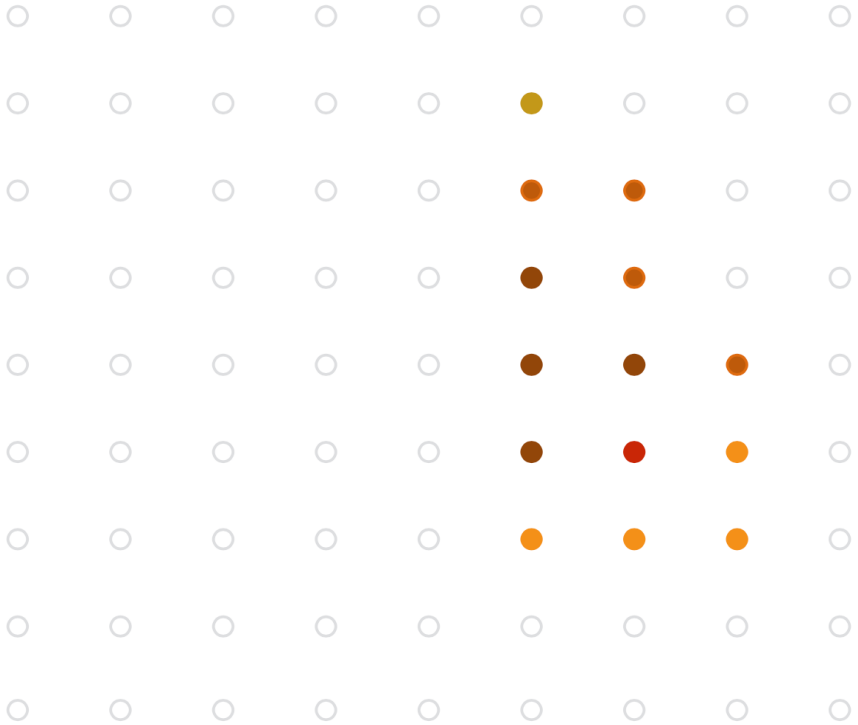


Step 7: 进行深度测试 (if enabled)

- 同时更新覆盖样本的深度值 (如有必要)



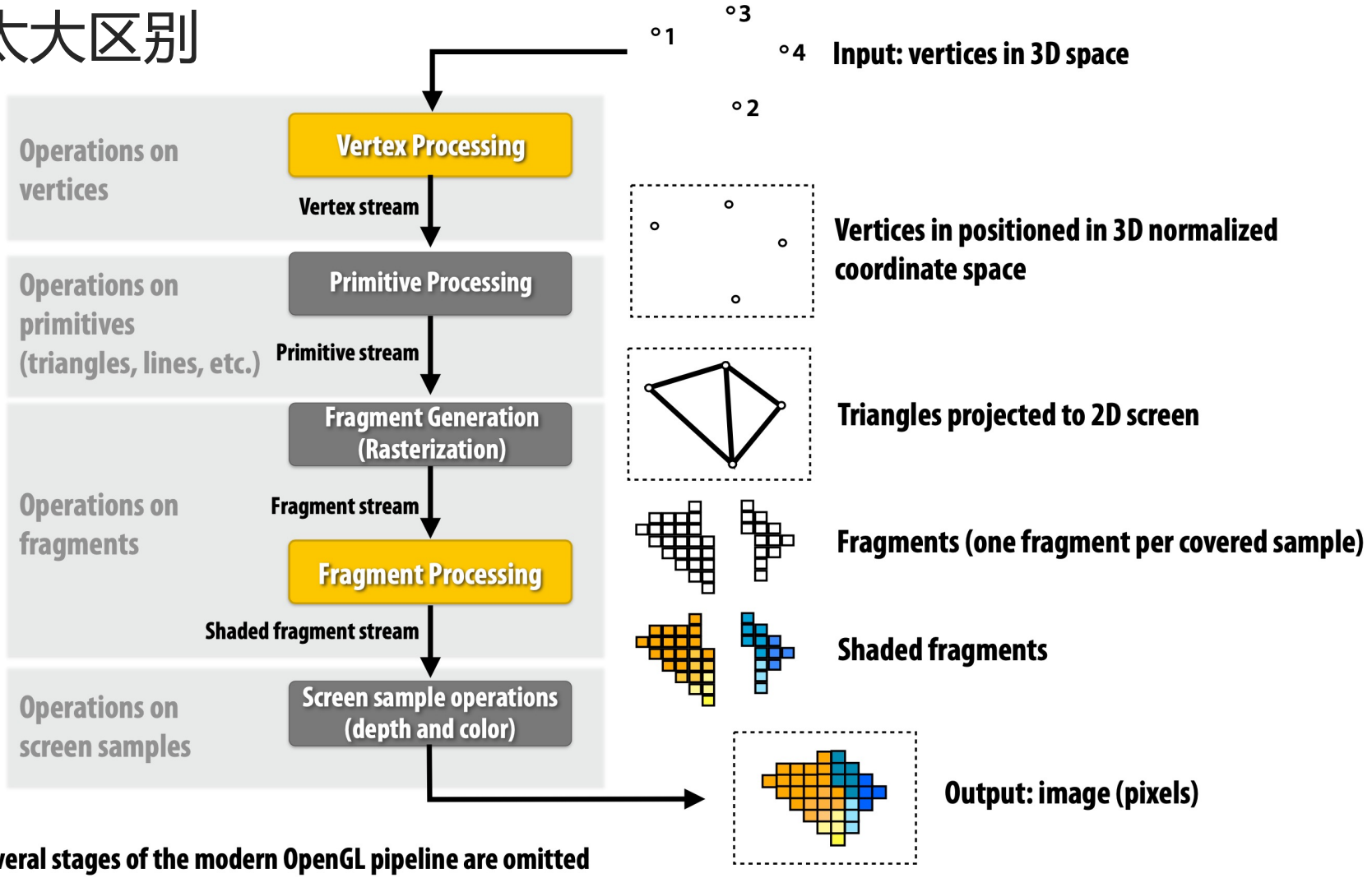
Step 8: 更新颜色缓存* (若通过深度测试)



*可能使用 over 操作来处理透明度

OpenGL/Direct3D graphics pipeline

- 我们的光栅化流程与现代API/图形硬件中的实际流程没有太大区别



* Several stages of the modern OpenGL pipeline are omitted

目标：渲染超高复杂度的 3D 场景

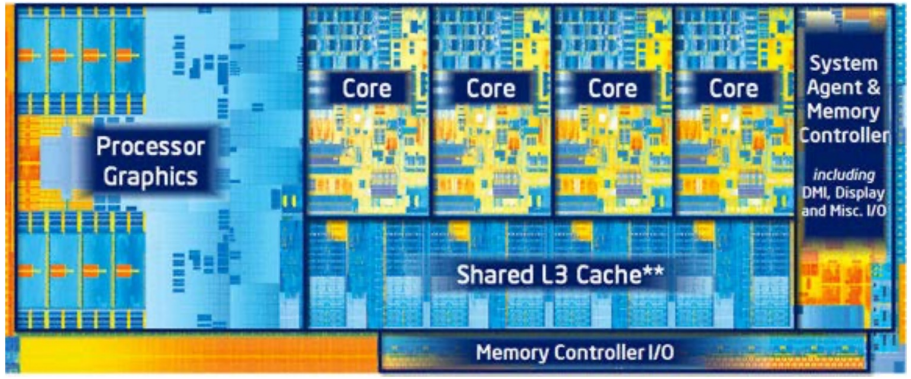
- 一个场景中可能包含百万个三角形
- 复杂的顶点和片段着色计算
- 高分辨率屏幕输出 (~10万像素+超采样)，30-120帧/秒

Unreal Engine Kite Demo (Epic Games 2015)

Graphics pipeline implementation: GPUs

- 用于执行图形计算的专用处理器

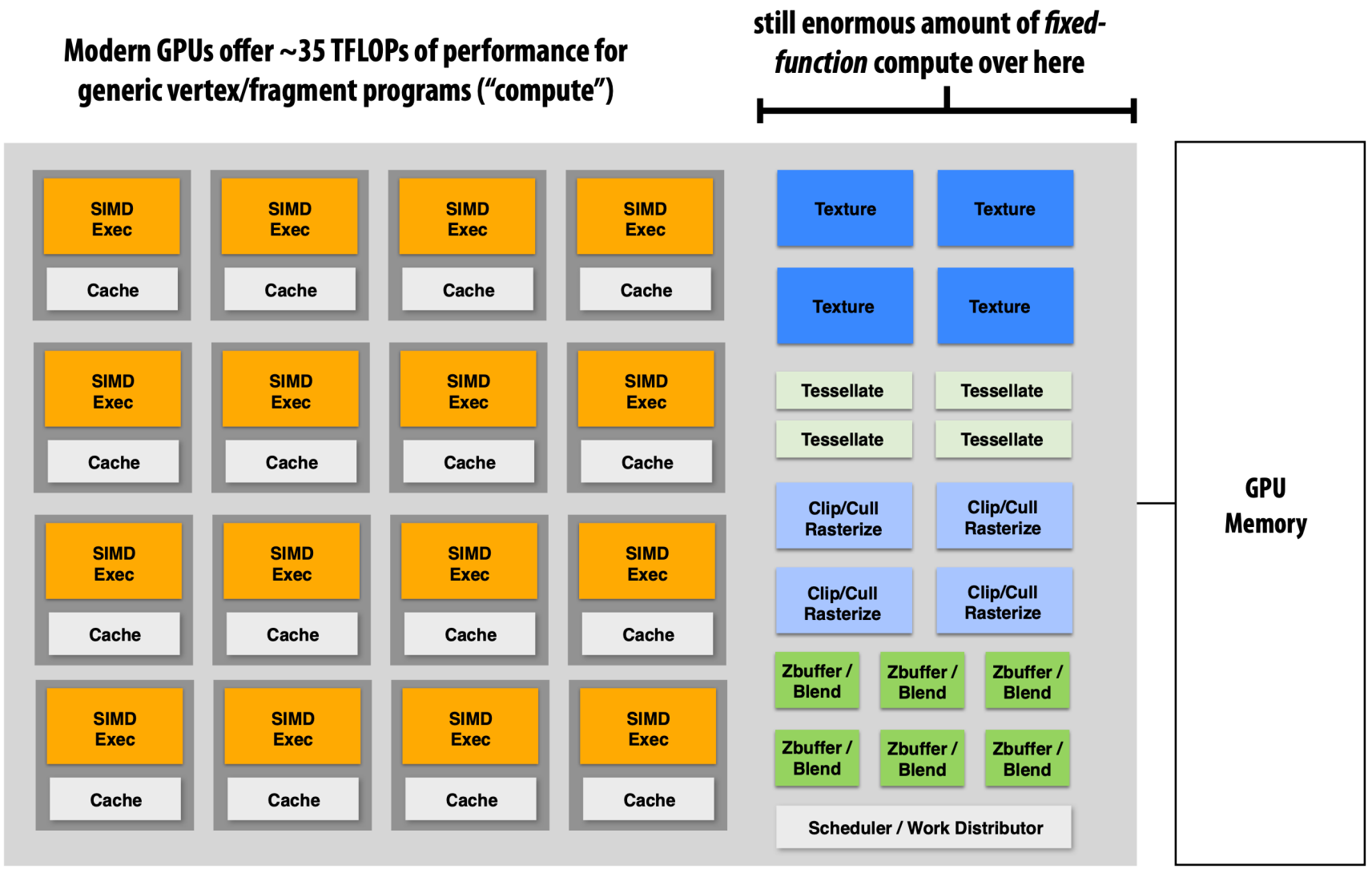
discrete GPU card



integrated GPU: part of modern CPU die

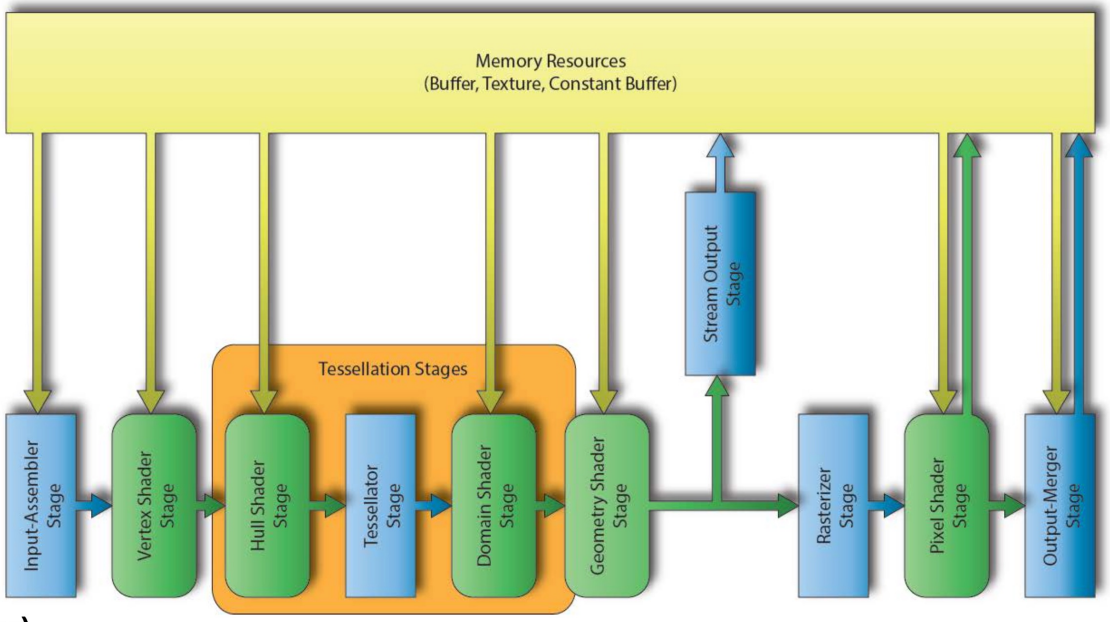
smartphone GPU (integrated)

GPU：异构多核处理器


Modern GPUs offer ~35 TFLOPs of performance for generic vertex/fragment programs ("compute")
still enormous amount of *fixed-function* compute over here
SIMD Exec
Cache
Texture
Tessellate
Clip/Cull Rasterize
Zbuffer / Blend
Scheduler / Work Distributor
GPU Memory
This part (mostly) not used by CUDA/OpenCL; raw graphics horsepower still greater than compute!

现代光栅化流程

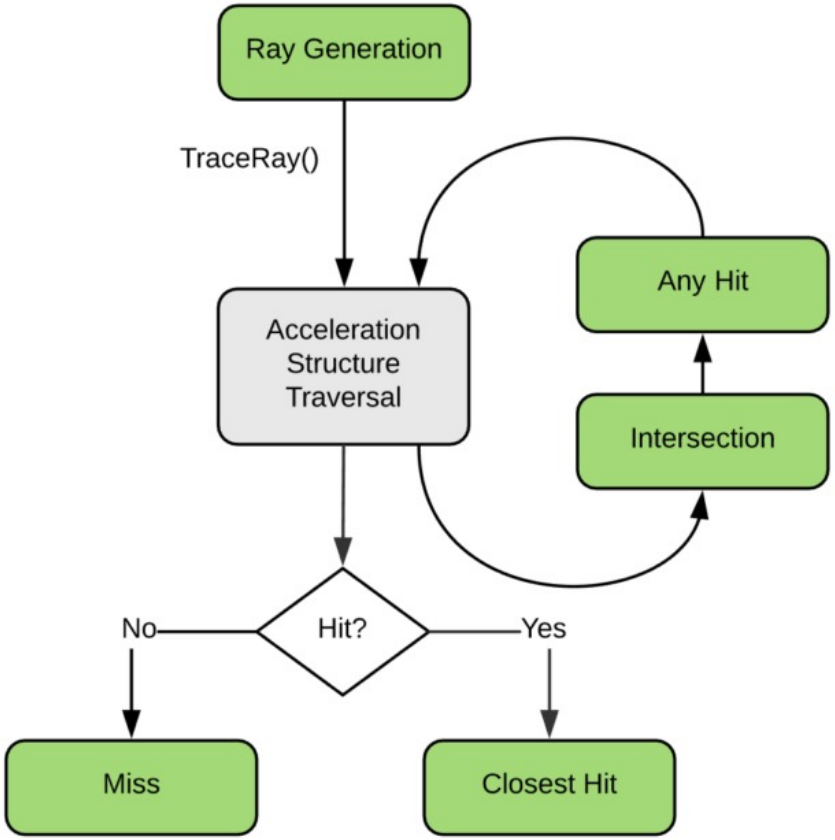
- 趋向于更通用（但仍然高度并行！）的计算：
  - 使阶段可编程
    - 替换固定的顶点，片段处理函数
    - 添加几何体，镶嵌着色器
    - 通用“计算”着色器
  - 更灵活的阶段调度



(DirectX 12 Pipeline)

图形流程中的光线跟踪 (ray tracing)

- 最近：光线跟踪的特定流程（NVIDIA RTX）



https://developer.nvidia.com/blog/introduction-nvidia-rtx-directx-ray-tracing/

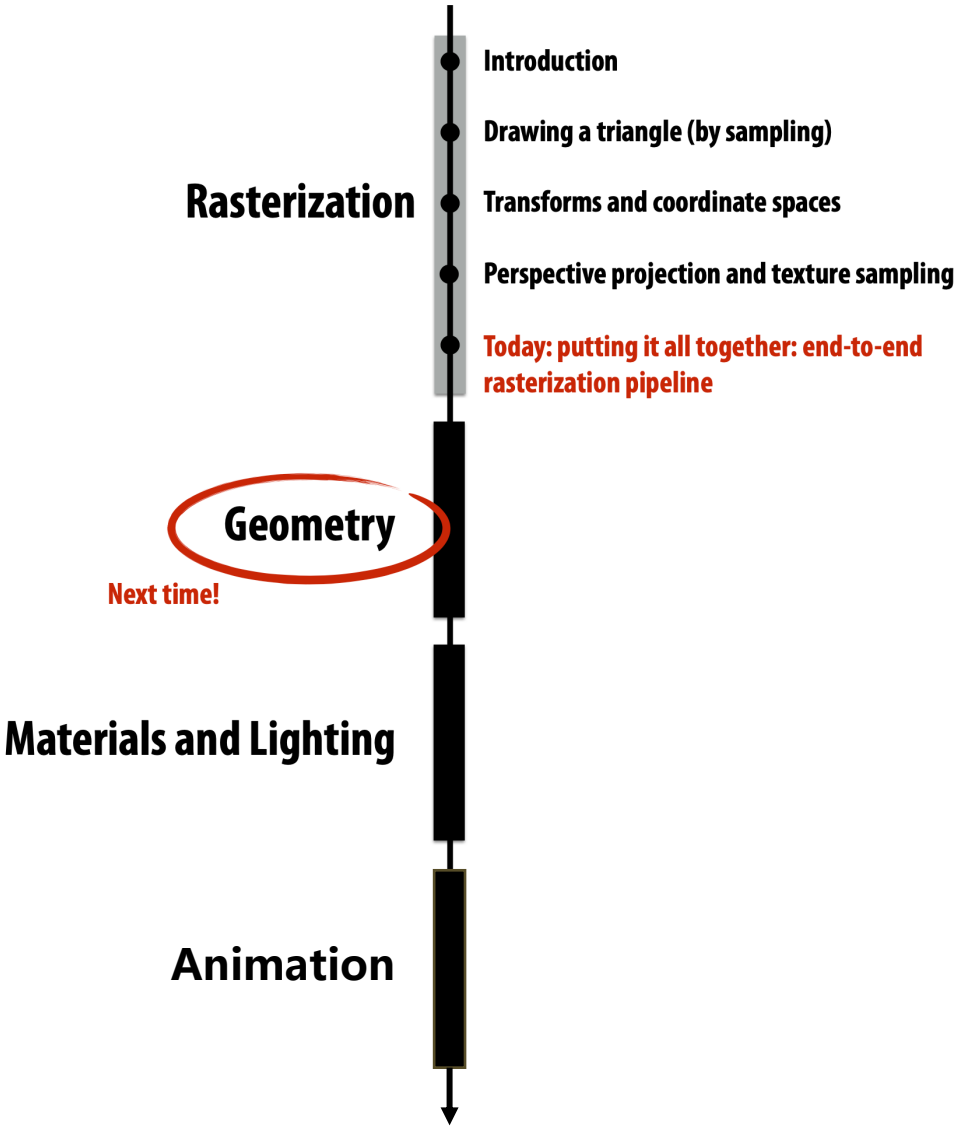
GPU Ray Tracing Demo (Marbles at night)

我们还要知道什么才能生成这样的图像?

- 几何形状 (Geometry)
  - 我们如何描述复杂的形状（到目前为止只有三角形...）
- 渲染 (Rendering)
  - 光是如何与材料相互作用产生颜色的?
- 动画 (Animation)
  - 我们如何描述事物的运动方式?

迪士尼“摩阿娜”

Course roadmap



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