

# 讲义五 - 光栅化管线

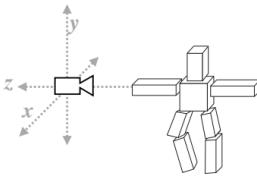
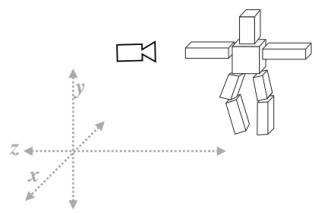
(及其在各种GPU上的实现)

---

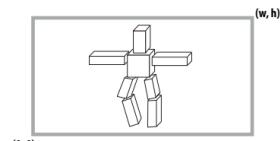
Computer Graphics: Rendering, Geometry, and Image Manipulation  
Stanford CS248A, Winter 2023

你所知道的关于如何实现的内容(本课程到目前为止)

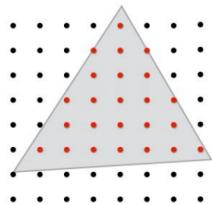
在世界坐标系中定位  
目标物体和相机



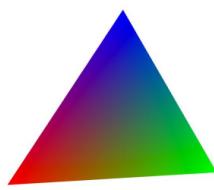
决定目标物体  
相对于相机的位置



投射目标物体到屏幕上



采样三角形覆盖率



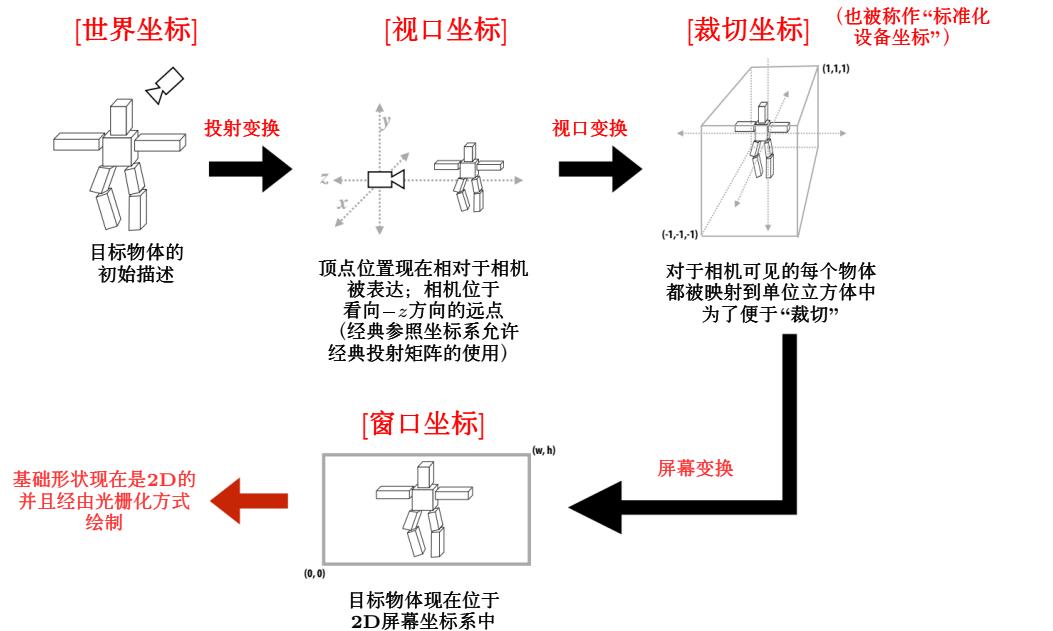
在覆盖的采样点  
计算三角形属性值  
(色彩, 纹理坐标, 深度)



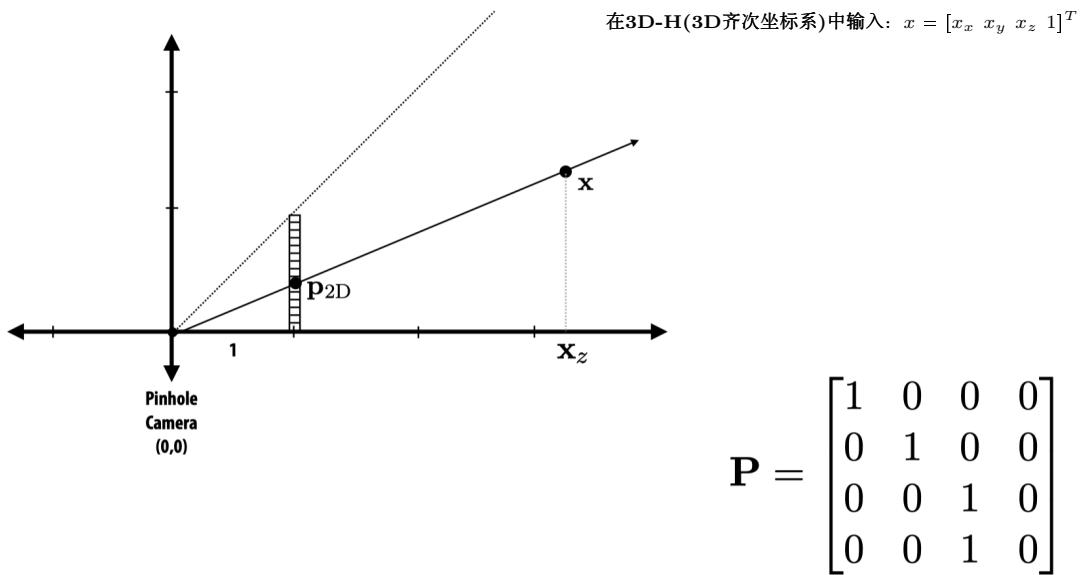
采样纹理贴图

关于透视投射将有更多细节

## 变换：从目标物体到屏幕



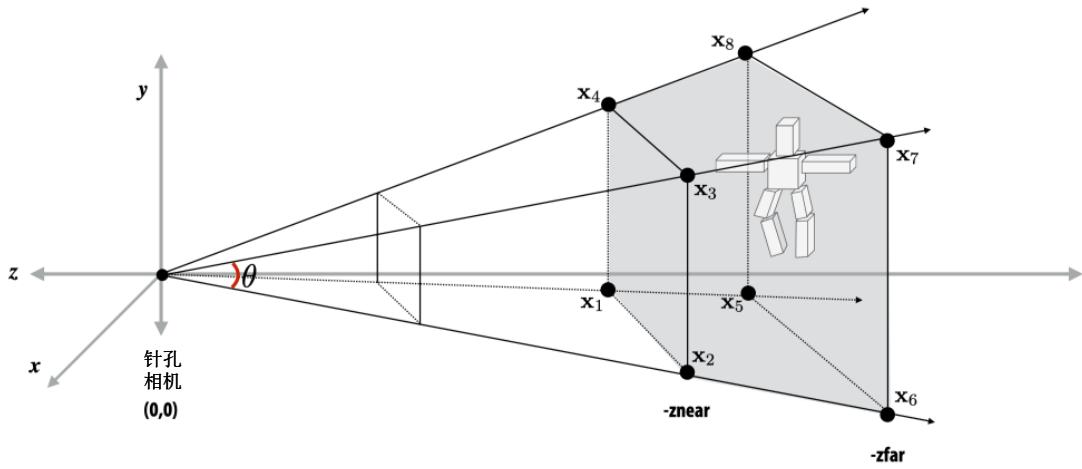
## 基本的透视投射



假设:

位于 $(0,0)$ 处的针孔相机看向 $z$ 轴方向

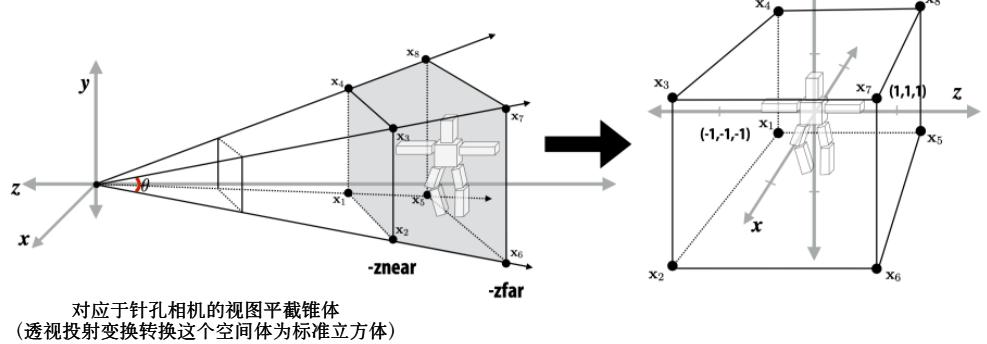
## 视图平截锥体



- 顶/底/左/右 平面对应于屏幕的边缘
- 近/远 平面对应于要绘制的最近/最远的物体

## 映射平截锥体(frustum)到标准立方体

在变换到2D空间之前，映射视图平截锥体的边角到立方体的边角：



为什么我们要映射平截锥体到单位立方体？

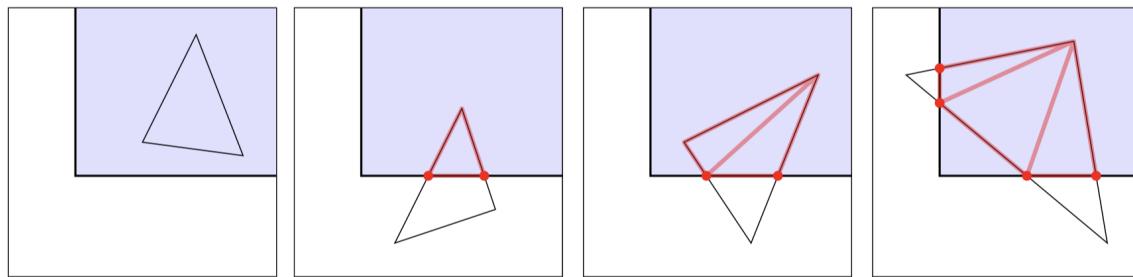
1. 让裁切更加容易！(参考下一页幻灯片)
  - 可以快速地废弃范围 $[-1, 1]$ 之外的几何体
2. 在标准立方体中以固定位数的数学方式表达所有的顶点

\*问题：正交相机(orthographic camera)视图平截锥体的形状看起来是什么样子？

## 裁切

- “裁切”是一种处理过程，其消除对相机不可见的三角形（它们位于视图平截锥体之外）

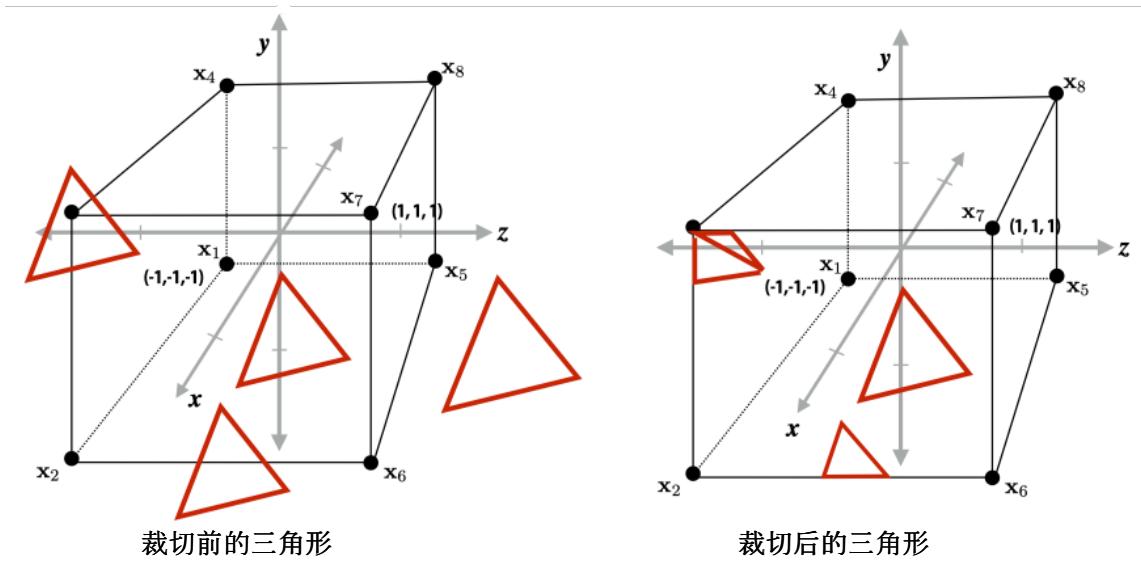
- 不要浪费时间计算相机不能看见的基本形状的外观
- 三角形内采样检测是昂贵的（“细粒度”可视性）
- 舍弃整个基础形状更有意义（“粗粒度”）
- 必须处理被部分裁切的基础形状



图片来自: <https://paroj.github.io/gltut/>

## 在标准化设备坐标(NDC)中裁切

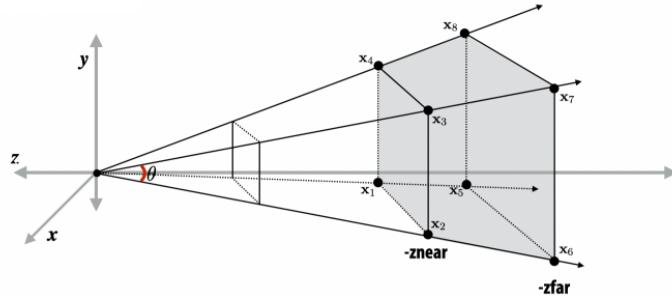
- 废弃那些完全位于标准立方体之外的三角形（剔除过程）
  - 这些三角形完全位于屏幕之外，不用困扰更深入一步处理
- 对于那些延伸出立方体的三角形...，裁切到立方体边缘
  - 注意：裁切可能生成更多三角形



\*这些图片没有问题: 用于OpenGL的标准设备坐标为左手性的坐标空间

## 用于透视变换的矩阵

考量视图平截锥体几何体:



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

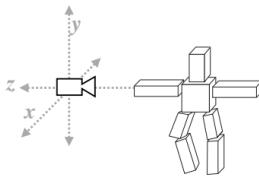
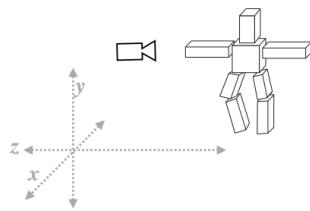
左(l),右(r),顶(t),底(b),近(n),远(f)

(左边的矩阵用于平截锥体frustum的透视投射,  
锥体关于x轴,y轴对称:l = -r, t = -b)

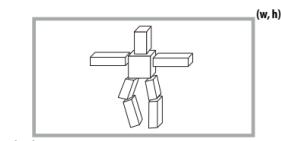
推导过程参考: [http://www.songho.ca/opengl/gl\\_projectionmatrix.html](http://www.songho.ca/opengl/gl_projectionmatrix.html)

## 你所知道的关于如何实现的内容(本课程到目前为止)

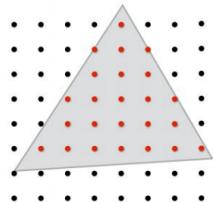
在世界坐标系中定位  
目标物体和相机



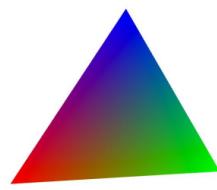
决定目标物体  
相对于相机的位置



投射目标物体到屏幕上



采样三角形覆盖率



在覆盖的采样点  
计算三角形属性值  
(色彩, 纹理坐标, 深度)



采样纹理贴图

要渲染下面这张图片，你还需要知道什么？

**表面描述**

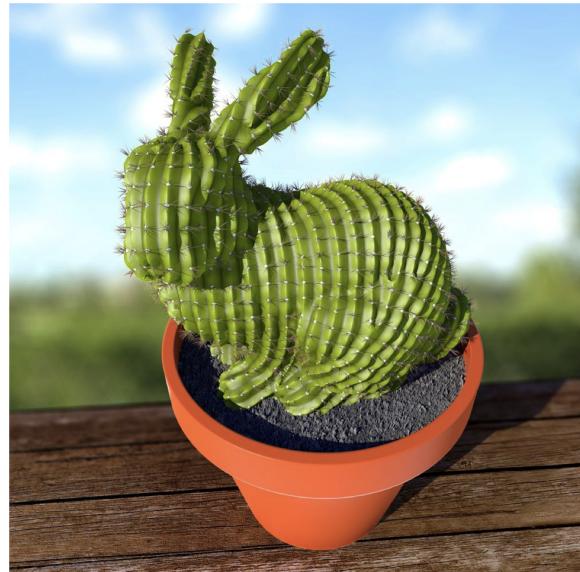
怎样表达复杂的表面？

**遮挡**

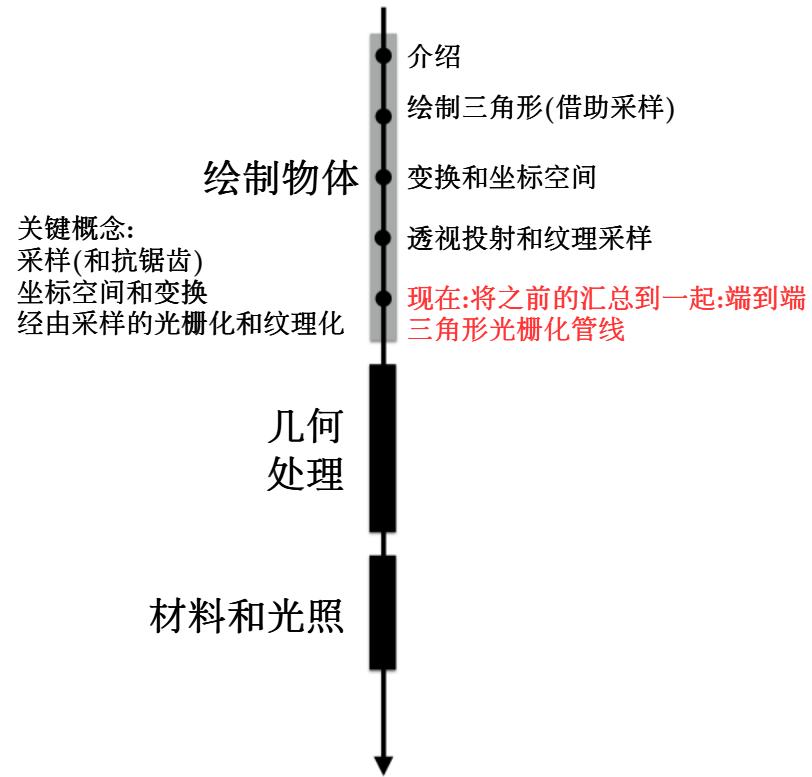
在每个采样点，决定对于相机那个表面可见。

**光照/材料**

描述场景中的光照和材料反射光的方式。

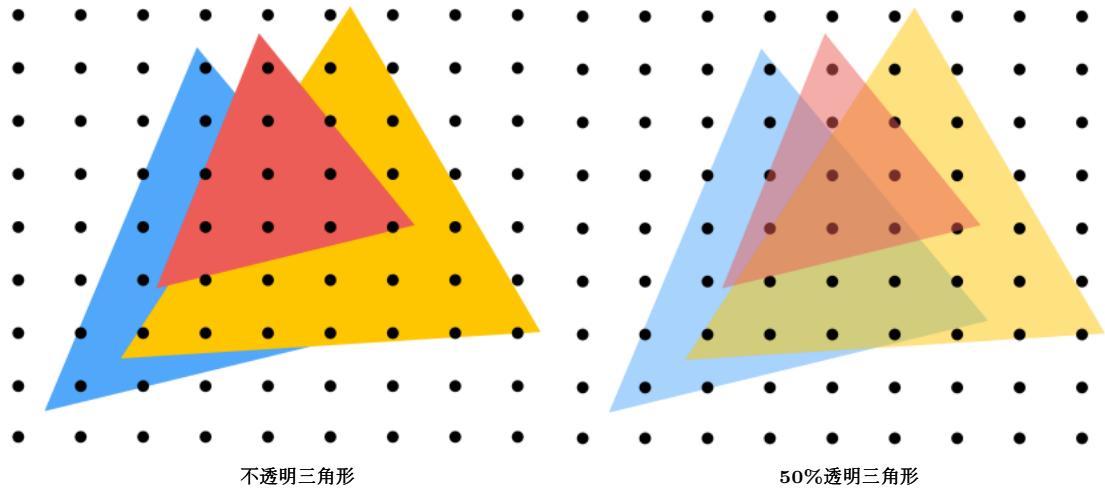


## 课程路线图，接下来的内容 …



使用深度缓存(Depth Buffer)的遮挡

遮挡：在每个被覆盖到的采样点哪个三角形是可见的？



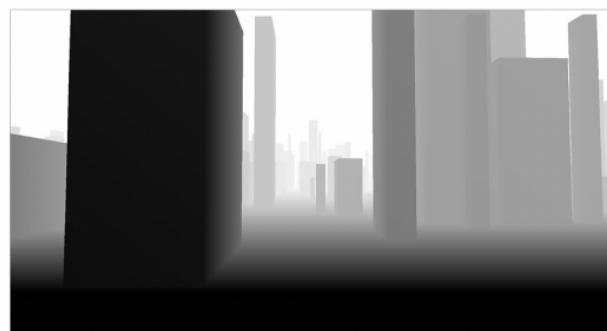
深度缓存(也被称为：“Z缓存”)

色彩缓存：  
存储每个样本的色彩/cdots,比如RGB值

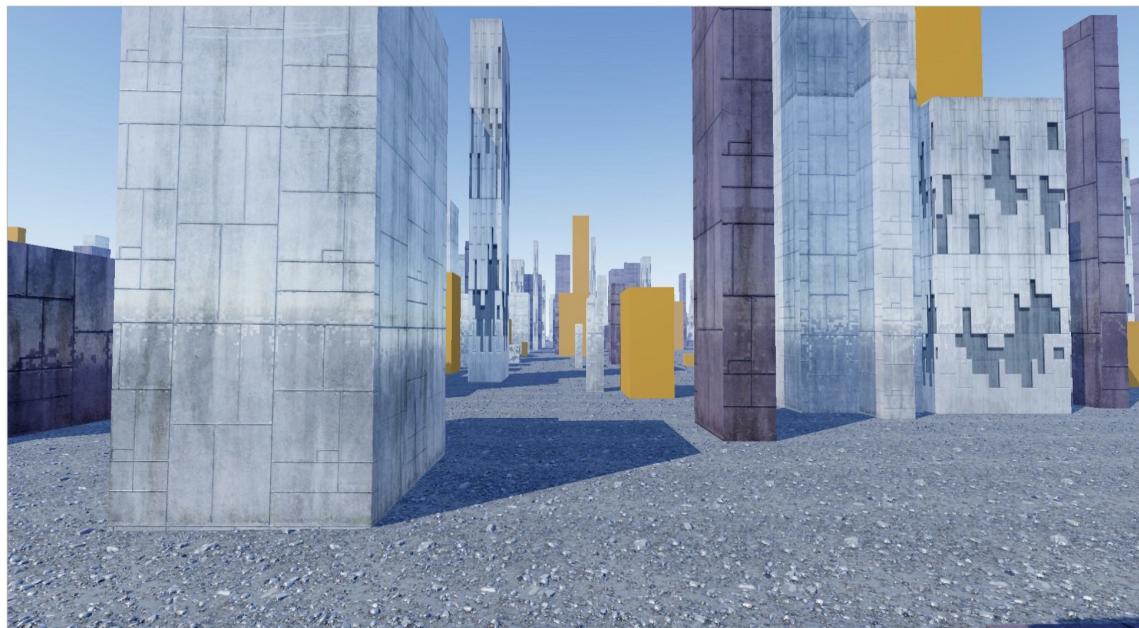


深度缓存：  
存储每个样本的深度值

存储当前绘制的最近表面的深度值。  
黑色=近的深度值  
白色=远的深度值



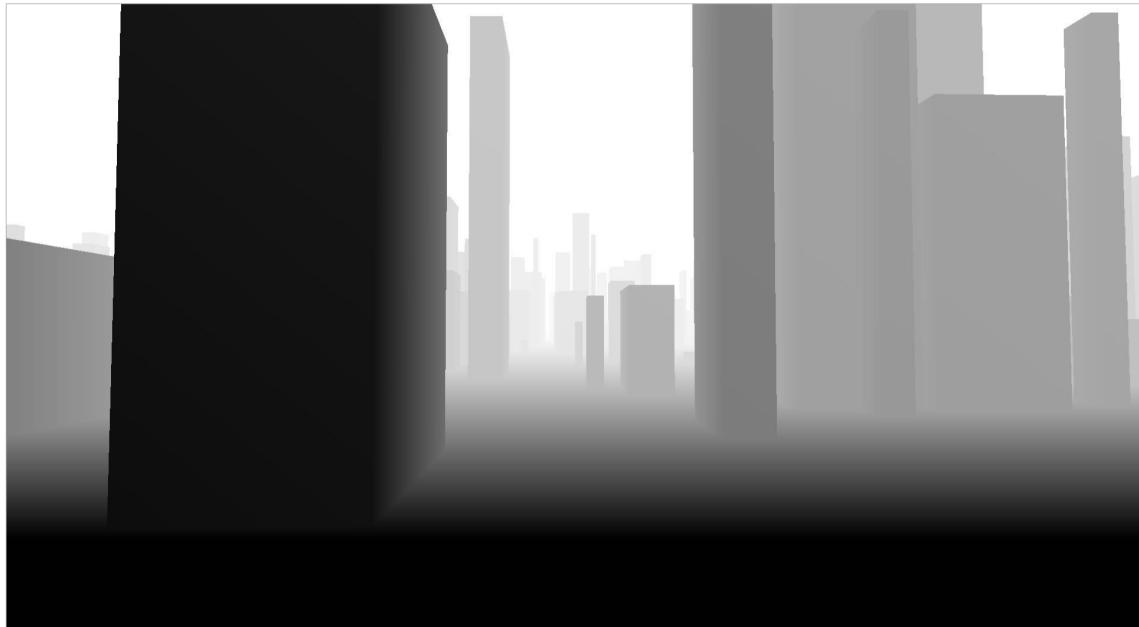
深度缓存(外观更好)



色彩缓存(存储每样本色彩值, 比如, 每样本RGB值)

## 深度缓存(外观更好)

可视化:像素越远,到最近物体的距离越近



所有的三角形渲染完成后对应的深度缓存(存储每样本的最近场景深度)

## 借助深度缓存的遮挡(“Z缓存”)

针对每个覆盖采样点，深度缓存存储在这个采样点目前为止已经被渲染器处理过的最近三角形的深度。

在采样点 $(x, y)$ 处的距离最近的三角形是在 $(x, y)$ 处拥有最小深度的三角形



## 前一讲回顾

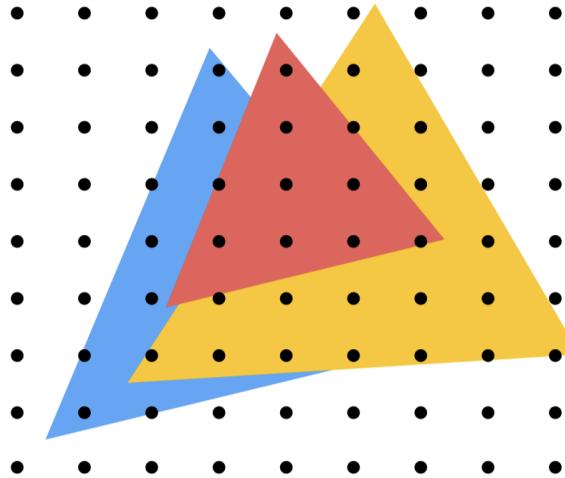
假设我们通过屏幕空间2D位置和相对每个顶点相机的距离(“深度”)定义了一个三角形

$$\begin{aligned} & [\mathbf{p}_{0x} \quad \mathbf{p}_{0y}]^T, \quad d_0 \\ & [\mathbf{p}_{1x} \quad \mathbf{p}_{1y}]^T, \quad d_1 \\ & [\mathbf{p}_{2x} \quad \mathbf{p}_{2y}]^T, \quad d_2 \end{aligned}$$

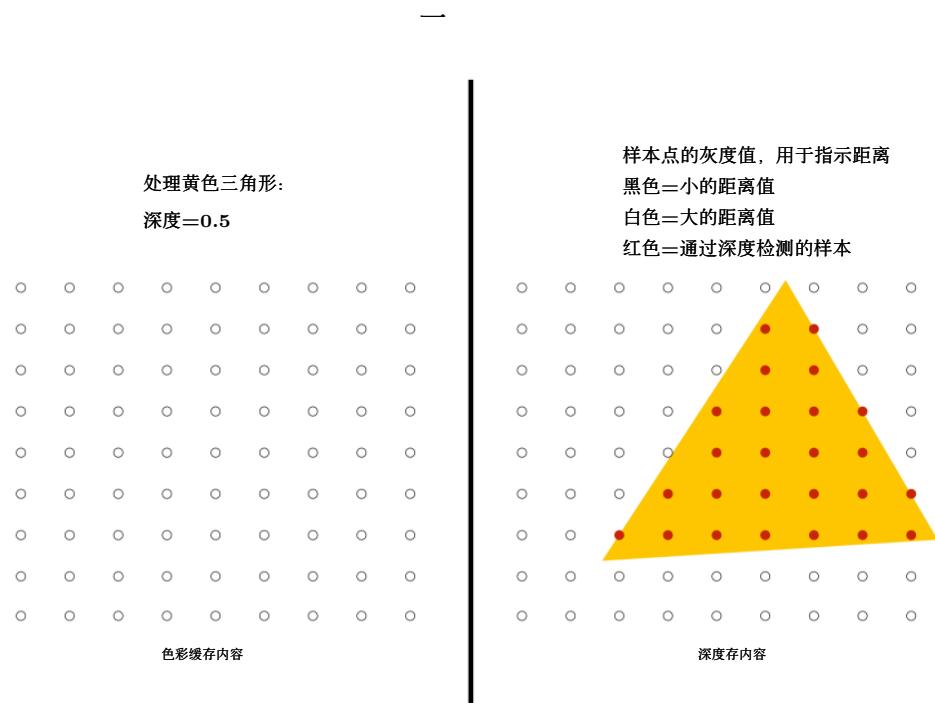
在覆盖的样本点 $(x, y)$ 如何计算三角形的深度?

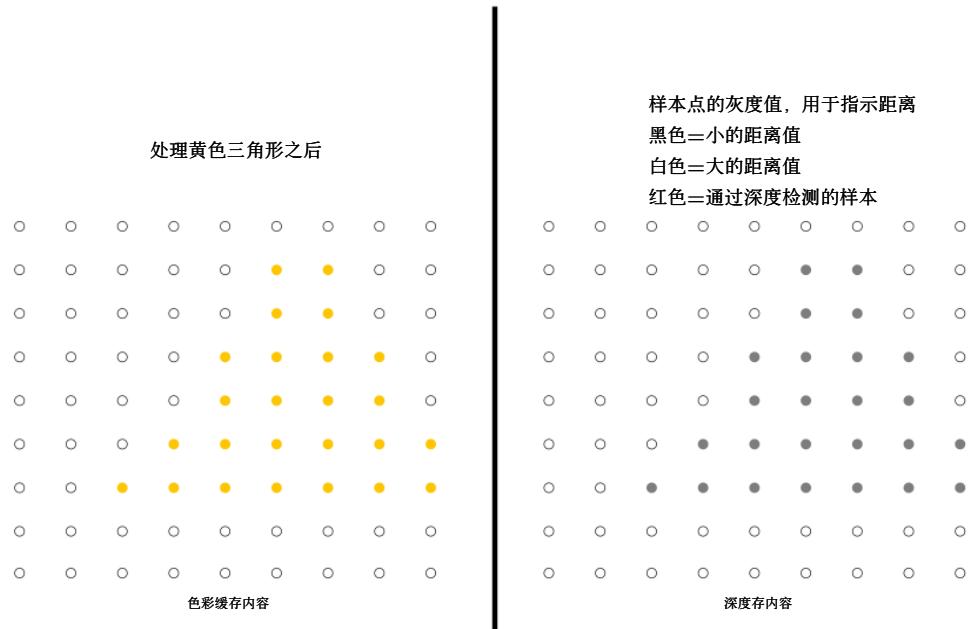
插值深度类似在三角形表面上线性变化的任何其它属性上的插值

例子：渲染3个不透明的三角形

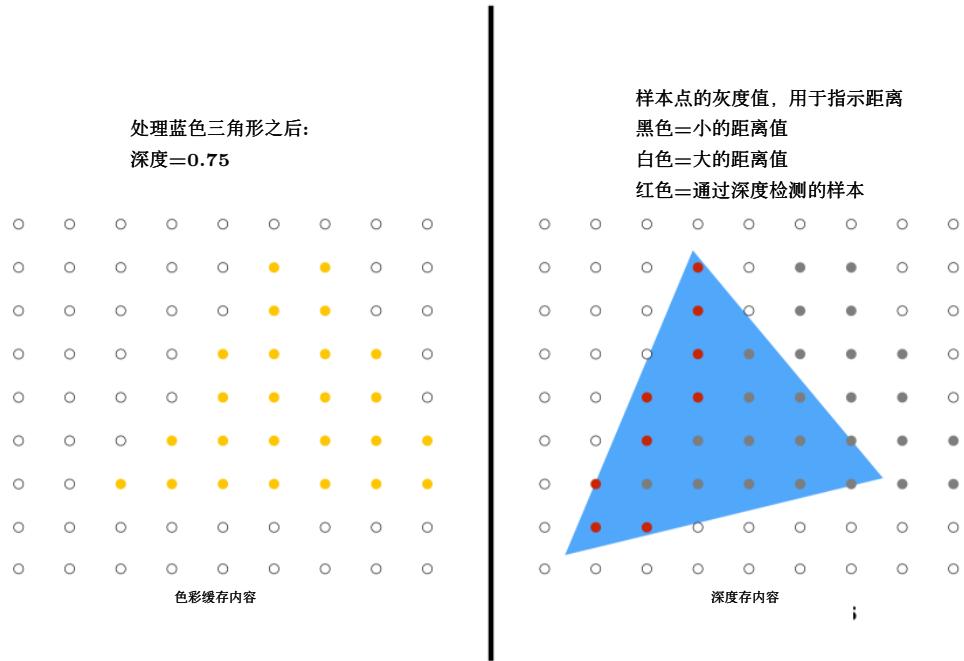


借助深度缓存的遮挡(“Z缓存”)



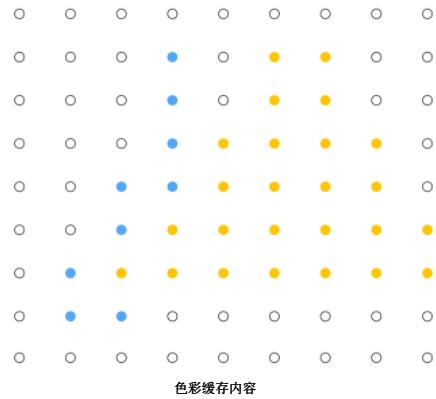


### 三



### 四

处理蓝色三角形之后

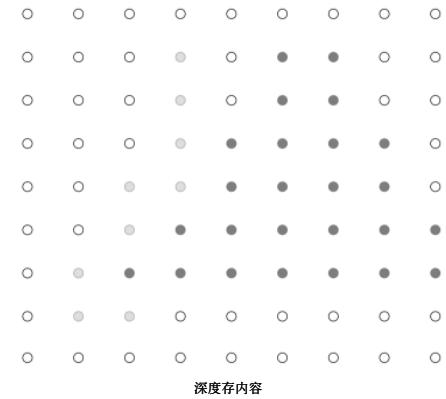


样本点的灰度值，用于指示距离

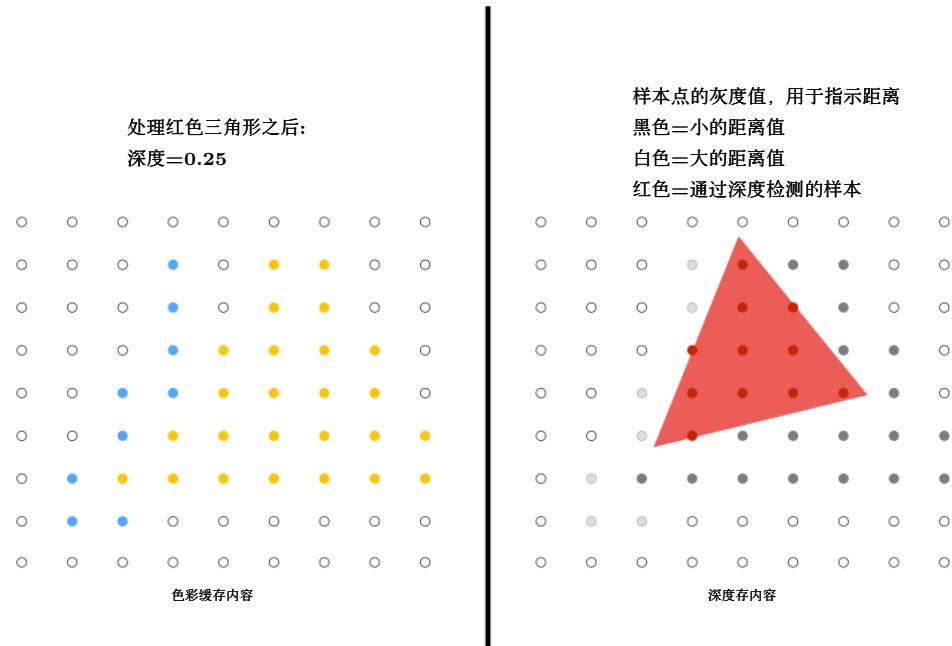
黑色=小的距离值

白色=大的距离值

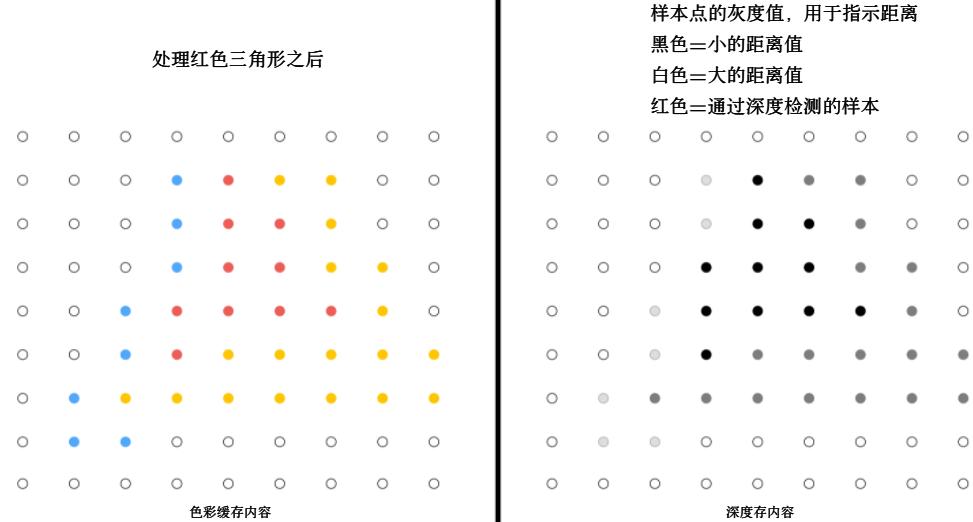
红色=通过深度检测的样本



## 五



## 六



## 借助深度缓存的遮挡(不透明表面)

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

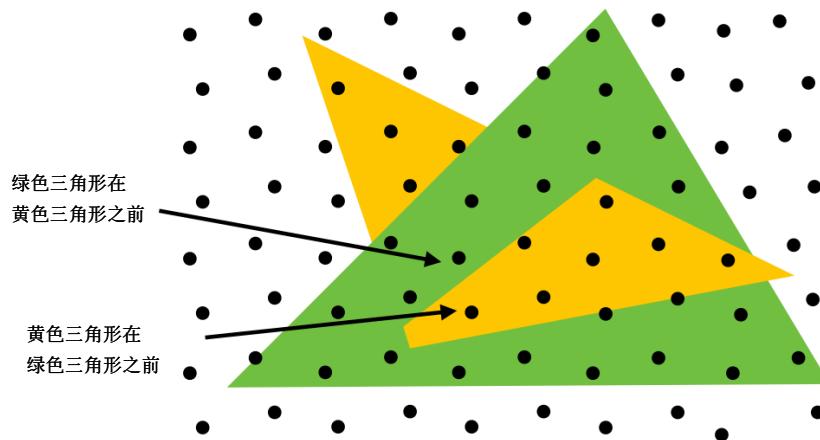
depth_test(tri_d, tri_color, x, y){
    if(pass_depth_test(tri_d, depth_buffer[x][y]))
    {
        // if triangle is closest object seen
        // so far at this sample point.
        // Update depth and color buffers.

        depth_buffer[x][y] = tri_d; //update
        depth_buffer
        color[x][y] = tri_color; //update color
        buffer
    }
}
```

## 深度缓存算法是否可以处理互相穿透的表面?

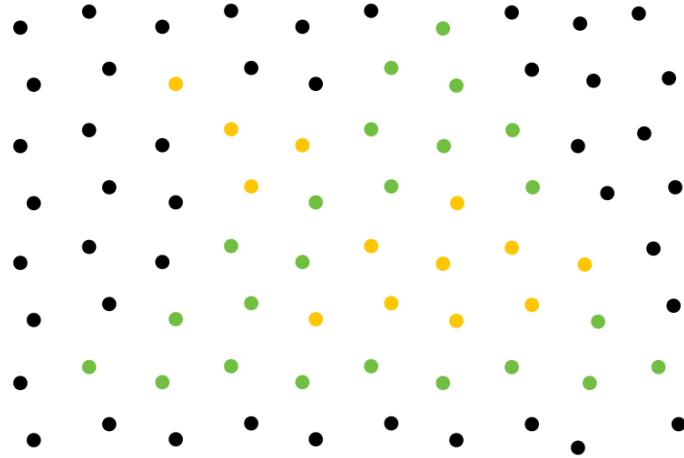
当然可以

遮挡检测基于指定采样点处三角形的深度。三角形的相对深度在不同采样点处可能是不同的。



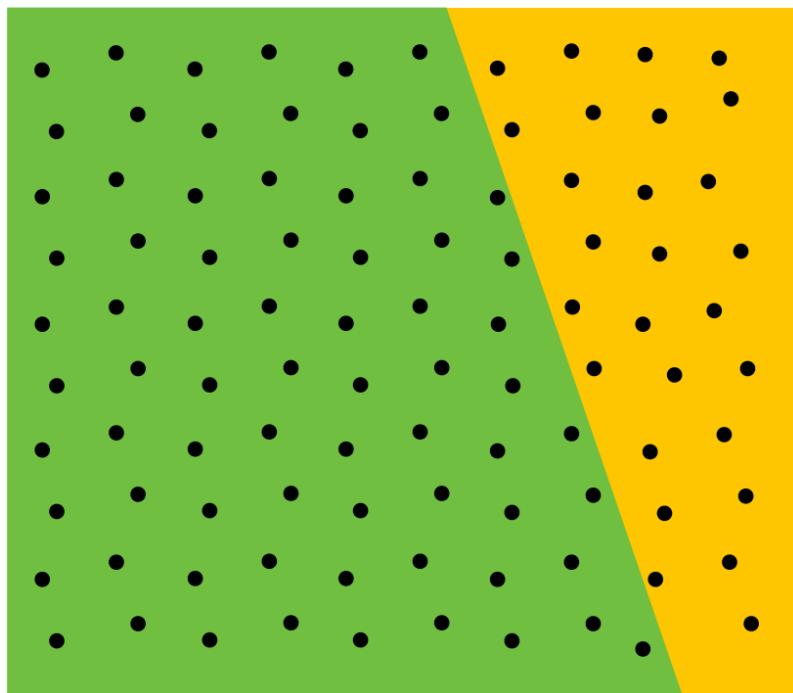
二

现在只展示  
已经着色的样本



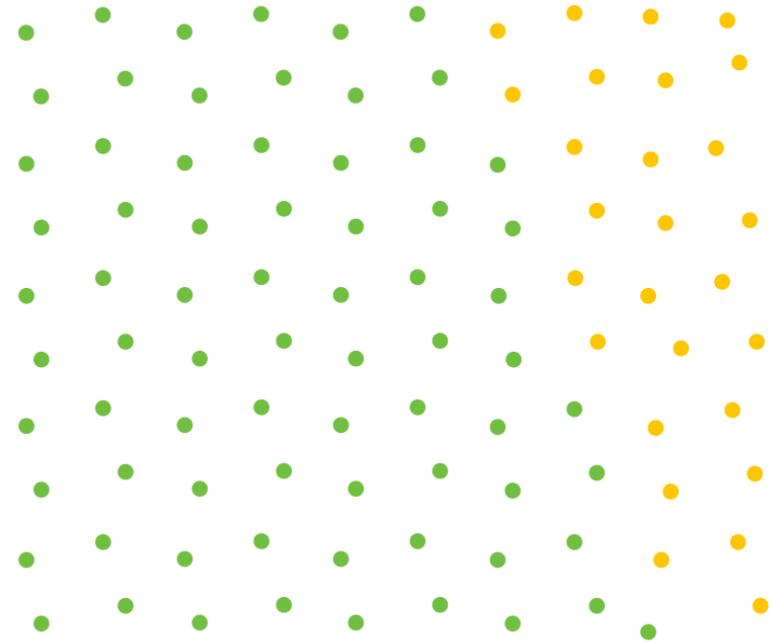
深度缓存可以和超采样(super sampling)一起生效吗?

当然可以! 深度检测是基于每样本的, 而不是每像素的。

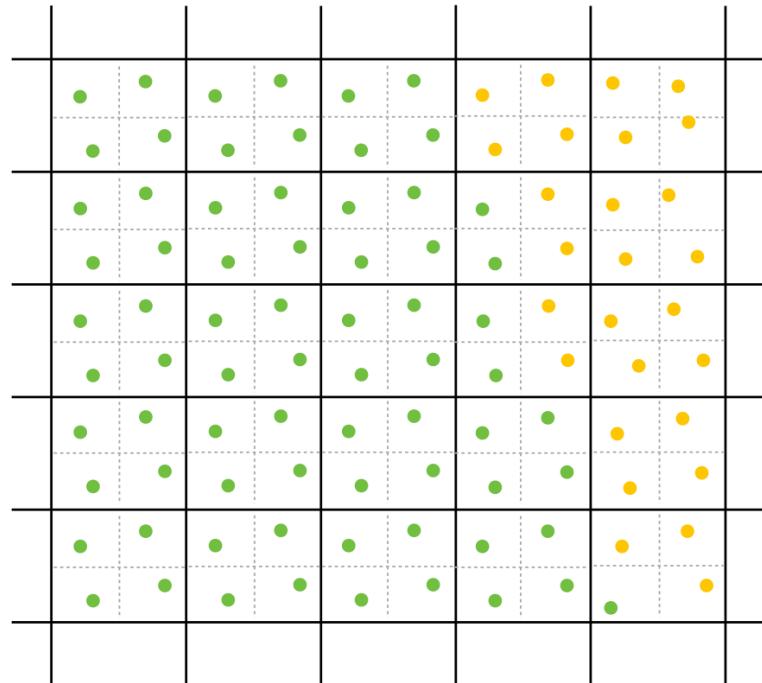


本例：绿色三角形遮挡了黄色三角形

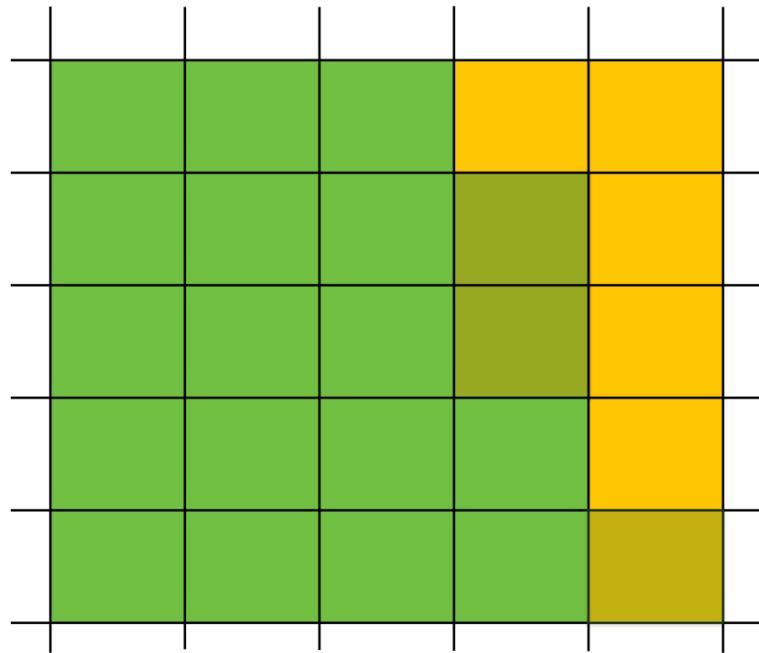
## 色彩缓存内容



色彩缓存内容(每像素4个样本)



## 最终重采样结果



由于对黄色和绿色样本进行了过滤，请注意边缘的抗锯齿效果

## 总结：借助深度缓存的遮挡

- 每一个被覆盖的样本存储一个深度值
- 每样本常量空间
  - 暗示：深度缓存使用常量空间
- 每被覆盖样本常量遮挡检测时间
  - 如果“通过”深度检测，要对深度缓存进行 读取-修改写入 操作
  - 如果“失败”，则只进行深度缓存的读取操作
- 不是三角形所专有的：仅仅要求在屏幕采样点表面深度可以被评估。

但是半透明表面要怎么处理？

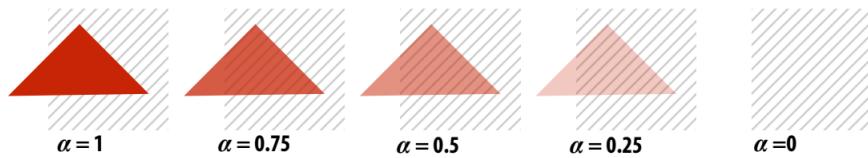
# 合成

## 表达不透明性为alpha

Alpha值描述了一个物体的不透明度

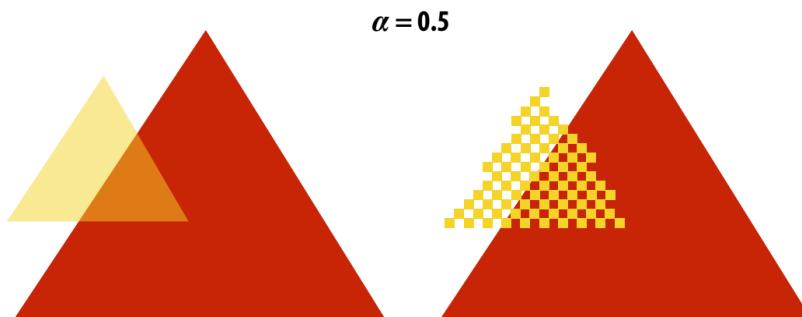
- 完全不透明表面:  $\alpha = 1$
- 50%透明表面:  $\alpha = 0.5$
- 完全透明表面:  $\alpha = 0$

红色三角形，不断减小不透明值



## Alpha:覆盖范围类比

- 可以把alpha当作描述半透明表面的不透明度
- 或…作为完全不透明物体的部分覆盖范围
  - 考虑纱窗效应(screen door)

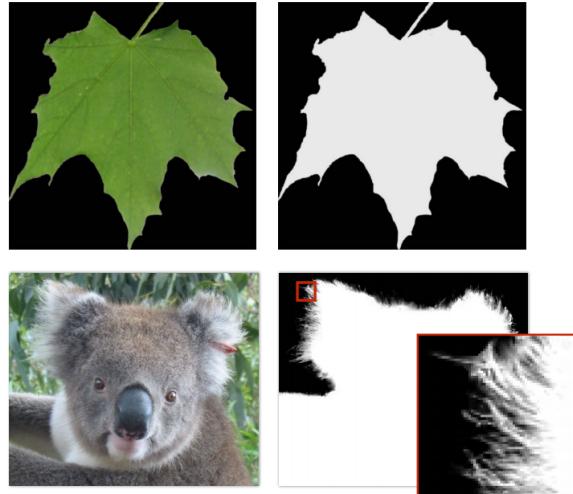


在左侧和右侧斜视本页幻灯片和屏幕将显示相似性

## Alpha:额外的图像通道(rgba)

Alpha描述了物体的不透明度

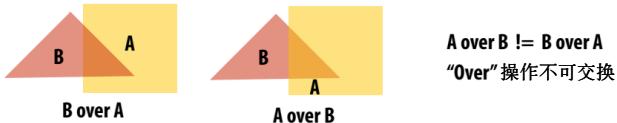
- 完全不透明表面:  $\alpha = 1$
- 50%透明表面:  $\alpha = 0.5$
- 完全透明表面:  $\alpha = 0$



前景物体的 $\alpha$ 值

## Over操作符

以拥有不透明值 $\alpha_B$ 的图像B Over 拥有不透明值 $\alpha_A$ 的图像A的方式进行合成



考拉 Over 纽约市

## Over操作符:非预乘的alpha

以拥有不透明值 $\alpha_B$ 的图像B Over 拥有不透明值 $\alpha_A$ 的图像A的方式进行合成

第一次尝试: (把色彩表示为3元矢量-3-vector,alpha分开表示)

$$A = [A_r \ A_g \ A_b]^T$$

$$B = [B_r \ B_g \ B_b]^T$$

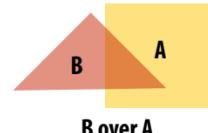
合成的色彩:

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

半透明图像  
B的外观

半透明图像  
A的外观

B让什么色彩通过



**A over B != B over A**

“Over”操作不可交换

合成alpha:

$$\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$$

## 预乘的alpha表达

- 表达(可能透明)的色彩为4元矢量(4-vectors), 其中RGB值已经预乘alpha值。

$$A' = [\alpha_A A_r \quad \alpha_A A_g \quad \alpha_A A_b \quad \alpha_A]^T$$

样例：50%不透明红色

**[0.5, 0.0, 0.0, 0.5]**



样例：75%不透明紫红色

**[0.75, 0.0, 0.75, 0.75]**

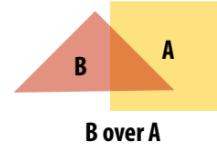


## Over操作符:借助预乘的alpha

以拥有不透明值 $\alpha_B$ 的图像B Over 拥有不透明值 $\alpha_A$ 的图像A的方式进行合成

非预乘的alpha表达:

$$\begin{aligned} A &= [A_r \ A_g \ A_b]^T \\ B &= [B_r \ B_g \ B_b]^T \\ C = \alpha_B B + (1 - \alpha_B) \alpha_A A &\leftarrow \begin{array}{l} 2\text{个乘法, 1个加法} \\ (\text{指色彩上的矢量操作}) \end{array} \end{aligned}$$



合成alpha:

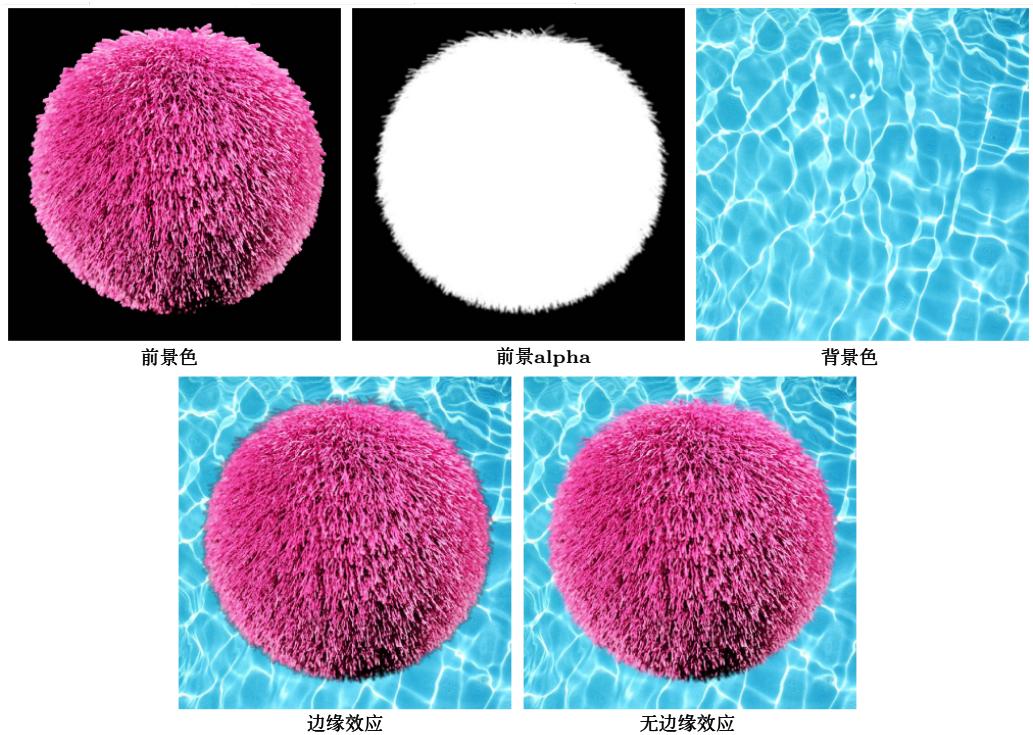
$$\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$$

预乘的alpha表达:

$$\begin{aligned} A' &= [\alpha_A A_r \ \alpha_A A_g \ \alpha_A A_b \ \alpha_A]^T \\ B' &= [\alpha_B B_r \ \alpha_B B_g \ \alpha_B B_b \ \alpha_B]^T \\ C' = B' + (1 - \alpha_B)A' &\leftarrow \begin{array}{l} \text{注意:预乘alpha方式合成alpha值} \\ \text{正如RGB色彩值的合成方式} \\ 1\text{个乘法, 1个加法} \end{array} \end{aligned}$$

## 边缘效应

糟糕的色彩/alpha处理可能导致暗色的“边缘效应”:



无边缘效应

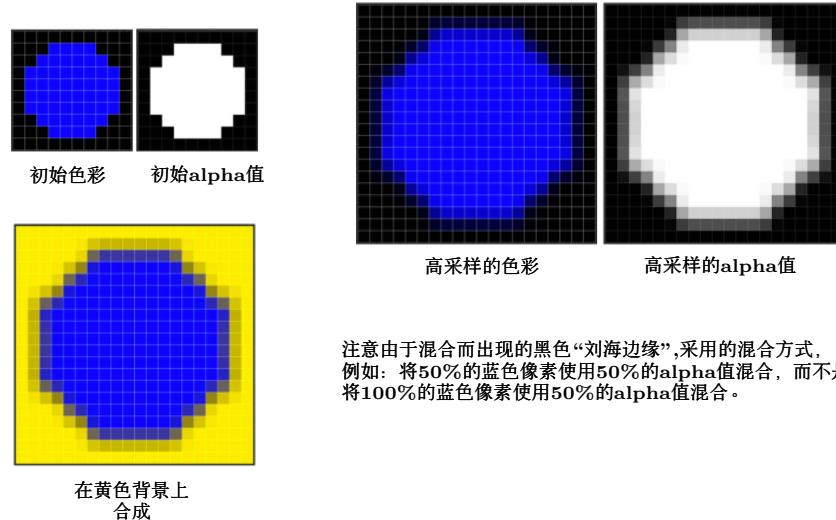


边缘效应(…为什么出现这种情形? )



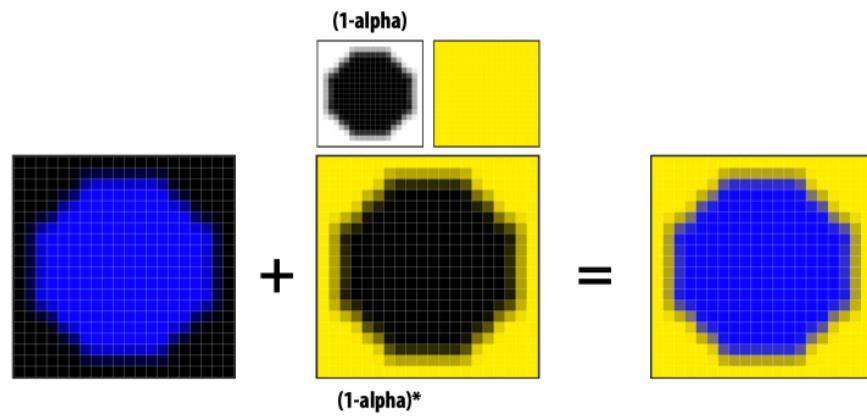
## 非预乘alpha的问题

- 假设我们上采样(upsampling)图像附带一个alpha遮罩(mask),随后合成在一个背景上
- 我们要如何计算插值的 色彩/alpha值
- 如果我们独立插值色彩和alpha值, 随后借助非预乘的“Over”操作符混合,下面是混合过程:



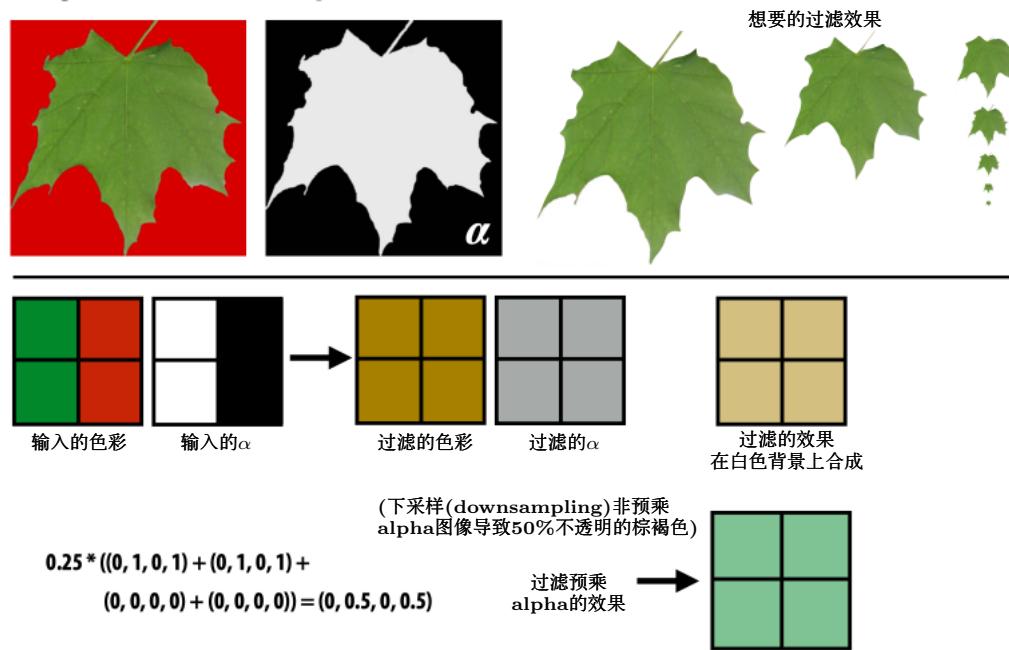
## 消除刘海, 借助预乘的“over”操作

相反, 如果我们使用预乘的“over”操作, 将会得到正确的alpha效果:

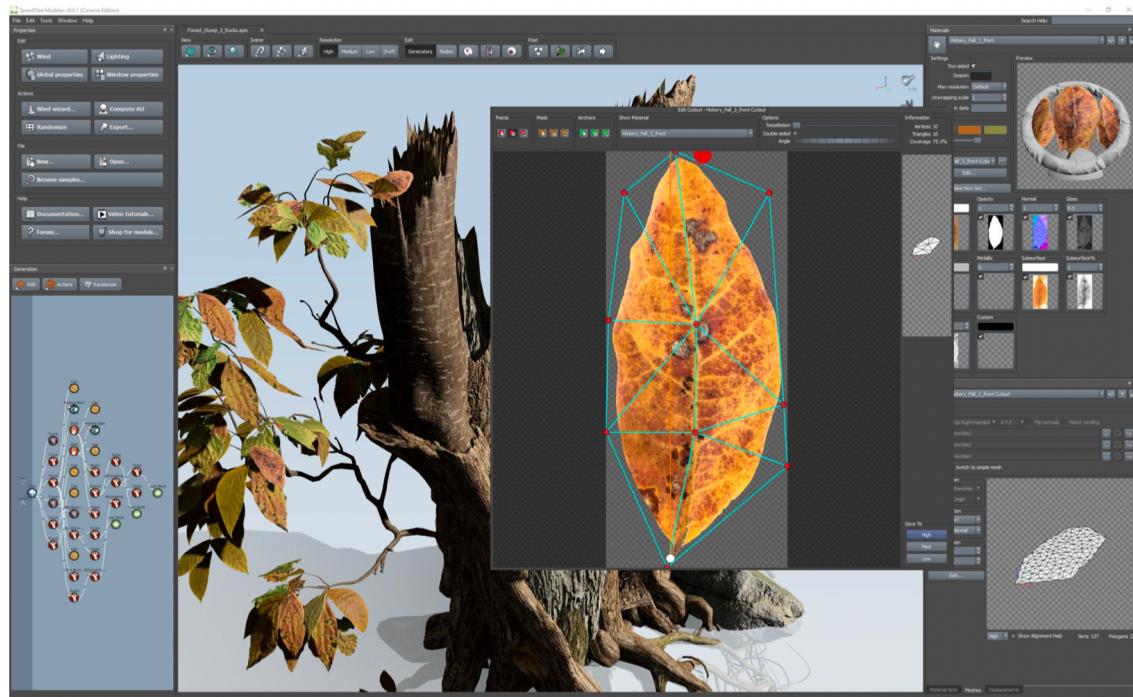


## 使用非预乘alpha的另一个问题

考慮借助alpha遮片预过滤纹理

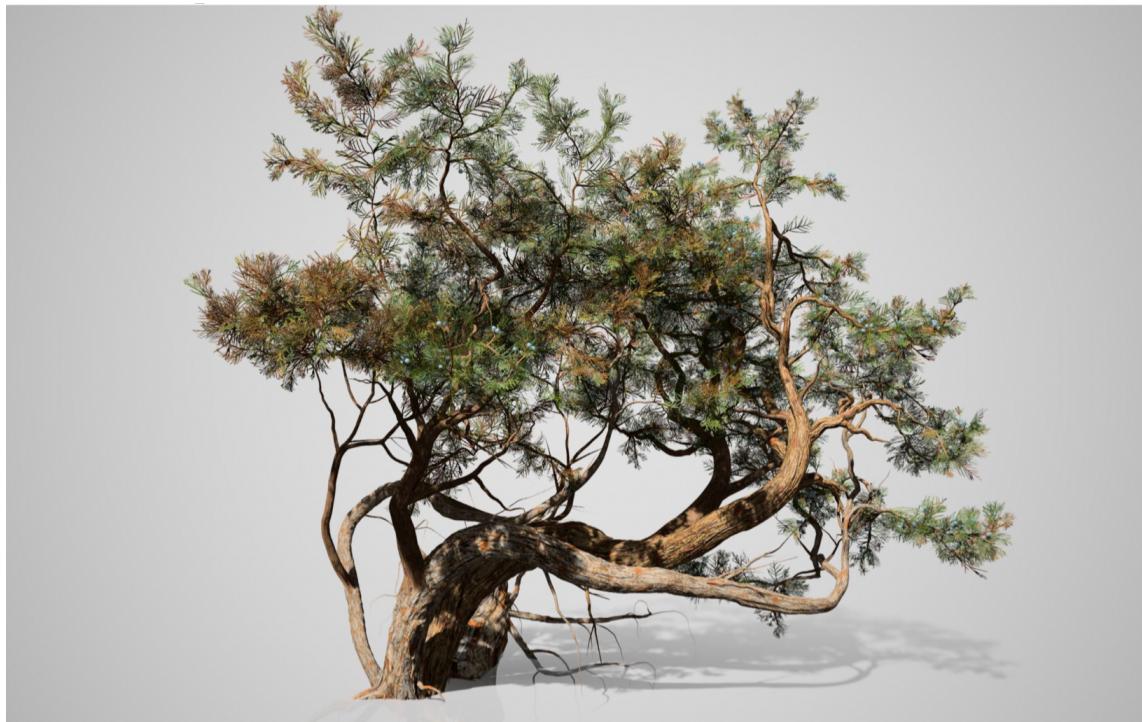


## 带有alpha通道的纹理的通常用处:植物叶子渲染



[Image credit: SpeedTree Cinema 8]

## 植物叶子渲染实例



[Image credit: SpeedTree Cinema 8]

## 另一个问题：当重复应用“over”操作时

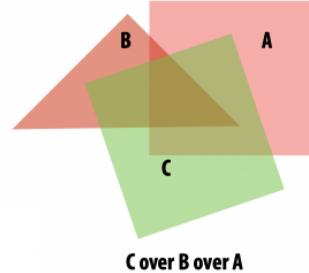
考虑这样的合成：拥有不透明值 $\alpha_C$ 的图像C over 拥有不透明值 $\alpha_B$ 的图像B over 拥有不透明值 $\alpha_A$ 的图像A

$$\begin{aligned} A &= [A_r \ A_g \ A_b]^T \\ B &= [B_r \ B_g \ B_b]^T \\ C &= \alpha_B B + (1 - \alpha_B)\alpha_A A \\ \alpha_C &= \alpha_B + (1 - \alpha_B)\alpha_A \end{aligned}$$

第一步考虑以50%红 over 50%红的方式合成：

$$C = [0.75 \ 0 \ 0]^T \text{ 等一下... 结果为预乘的色彩}$$

所以“over”操作针对非预乘的alpha和非预乘的色彩  
获得预乘的色彩(所以“over”操作不是闭合的)



不能在非预乘的值上合成“over”操作：  
**over(C, over(B, A))**

对于非预乘的alpha存在预测的形式：

$$C = \frac{1}{\alpha_C}(\alpha_B B + (1 - \alpha_B)\alpha_A A)$$

## 总结：预乘alpha操作的优点

- 简洁：合成操作以相同方式处理所有的通道(rgb和 $\alpha$ )
- 合成操作是闭合的
- 过滤有alpha通道的纹理时有更好的表达
- 比非预乘的表达更高效：“over”操作要求更少的数学计算操作

## 色彩缓存更新：半透明表面

假设：色彩缓存值和tri\_color变量使用预乘alpha表达

```

    over(c1, c2) {
        return c1 + (1 - c1.a) * c2;
    }
    update_color_buffer(tri_z, tri_color, x, y) {
        // Note: no depth check, no depth buffer
        update
        color[x][y] = over(tri_color, color[x][y]);
    }
}

```

这种实现的假设前提是什么？

三角形必须按照从后到前的顺序被渲染

假设三角形按照从前到后的顺序渲染怎么办？

需要改动代码: `over(color[x][y], tri_color)`

## 汇总所有内容

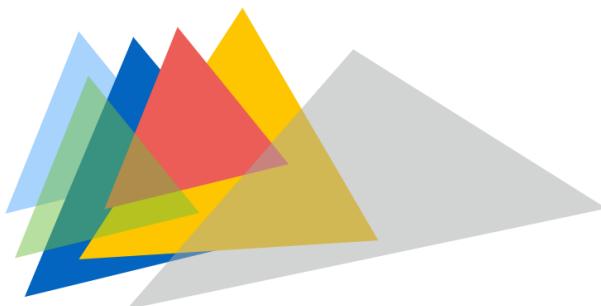
考虑渲染混合在一起的透明和不透明三角形

步骤1: 使用深度缓存遮挡(depth buffered occlusion)渲染不透明表面

如果通过深度缓存检测，三角形重写在样本处色彩缓存的值

步骤2: 禁用深度缓存更新,以从后到前的顺序渲染半透明表面。

如果通过深度缓存检测，三角形以OVER方式合成在样本处色彩缓存的内容。



\*如果这看起来有点复杂,那么渲染时你会喜欢光线追踪算法的简洁。

本科后面有关于光线追踪的更多内容，或者参考课程CS348B

## 合并不透明和半透明三角形的算法

假设:色彩缓存值和tri\_color变量使用预乘alpha表达

```
// phase 1: render opaque surfaces
update_color_buffer(tri_z, tri_color, x, y) {
    if (pass_depth_test(tri_z, zbuffer[x][y])) {
        color[x][y] = tri_color;
        zbuffer[x][y] = tri_z;
    }
}
// phase 2: render semi-transparent surfaces
update_color_buffer(tri_z, tri_color, x, y) {
    if (pass_depth_test(tri_z, zbuffer[x][y])) {
```

```
// Note: no depth buffer update  
color[x][y] = over(tri_color, color[x][y]);  
}
```

端到端光栅化管线

(“实时图形管线”)

End-to-end rasterization pipeline

(“real-time graphics pipeline”)

## 动作指令:绘制三角形

输入项:

```
list_of_positions = {      list_of_texcoords = {  
    v0x, v0y, v0z,          v0u, v0v,  
    v1x, v1y, v1z,          v1u, v1v,  
    v2x, v2y, v2z,          v2u, v2v,  
    v3x, v3y, v3z,          v3u, v3v,  
    v4x, v4y, v4z,          v4u, v4v,  
    v5x, v5y, v5z  };        v5u, v5v  };
```

物体空间到相机空间变换:

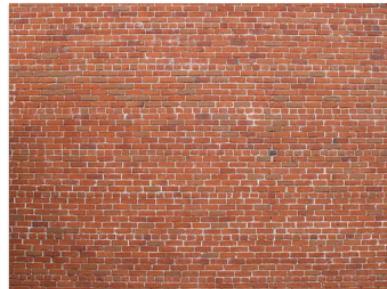
**T**

透视透视变换:

**P**

输出图像的尺寸( $W, H$ )

使用深度检测/更新深度缓存:是!

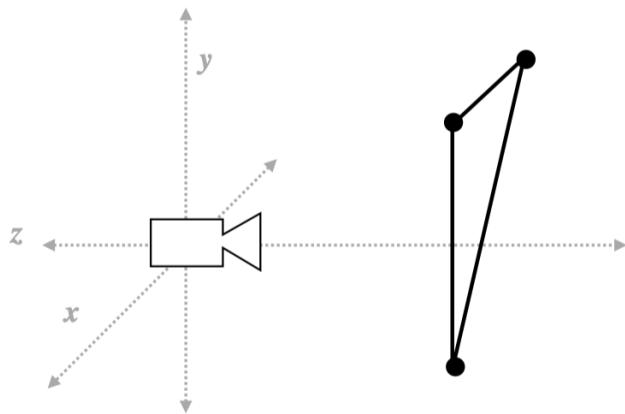


纹理贴图

## 第一步:

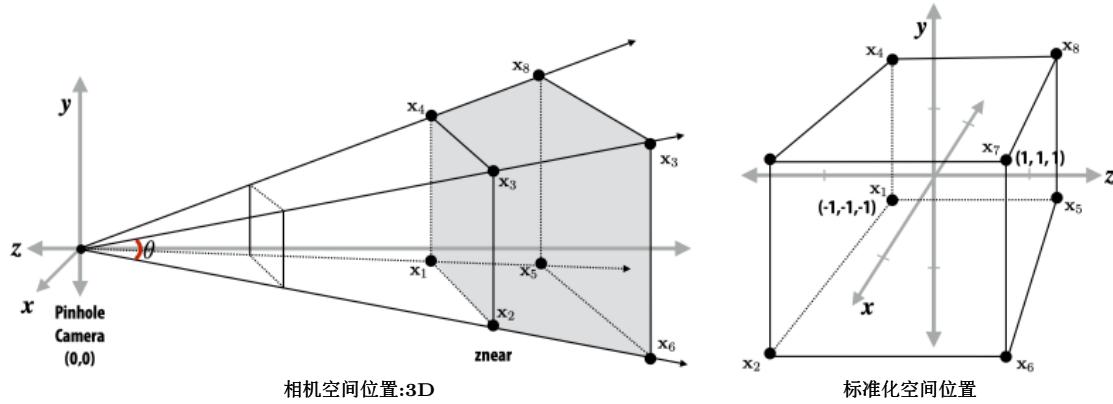
变换三角形顶点到相机空间

(应用模型和相机变换)



## 第二步:

应用透视投射变换转换三角形顶点到标准坐标空间



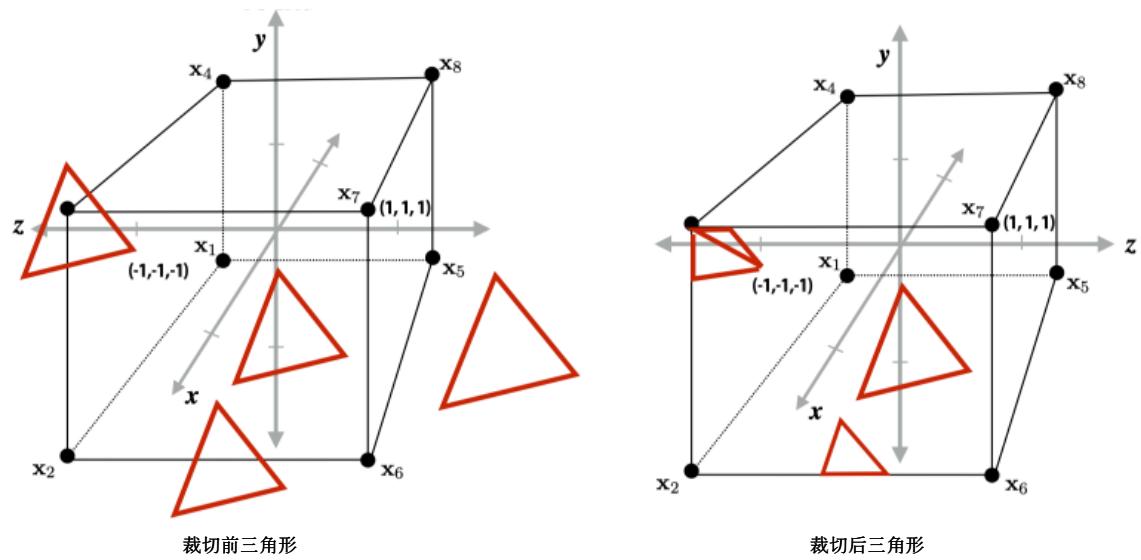
注意:我正在图示经过齐次除法后的标准化3D空间

更精确地考虑要将这个空间体积在3D-H空间中定义为:

$(-w, -w, -w, w)$  和  $(w, w, w, w)$

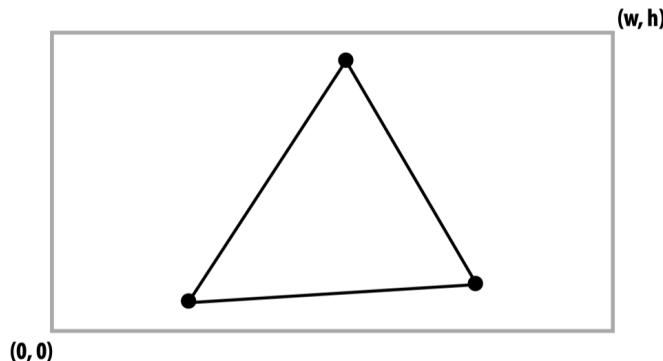
## 第三步:

- 废弃完全位于单位立方体外的三角形(剔除)
  - 这些三角形完全位于屏幕之外, 不用烦恼于进一步处理
- 裁切延伸出单位立方体外的三角形到立方体内
  - 注意:裁切可能产生更多的三角形



#### 第四步: 变换到屏幕坐标

变换顶点 $xy$ 坐标从标准化坐标到屏幕坐标(基于屏幕 $w, h$ )



## 第五步:设置三角形(三角形预处理)

计算三角形边缘方程式

计算三角形属性插值方程式

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

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

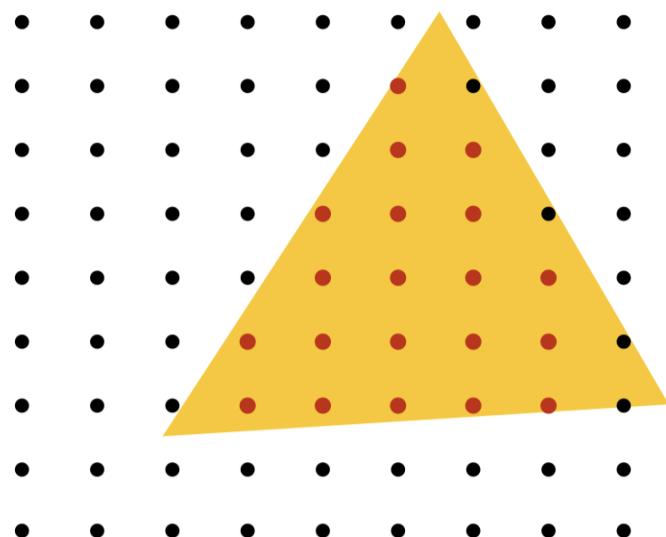
$$\mathbf{E}_{20}(x, y)$$

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

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

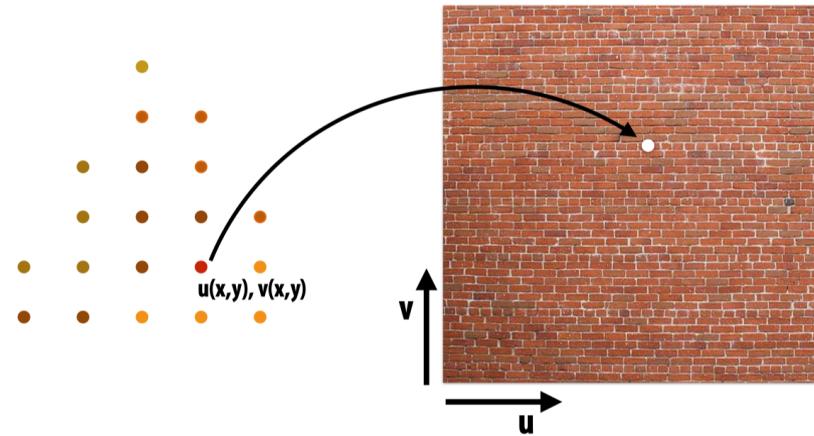
## 第六步:采样覆盖范围

在所有被覆盖的样本点评估属性 $z, u, v$



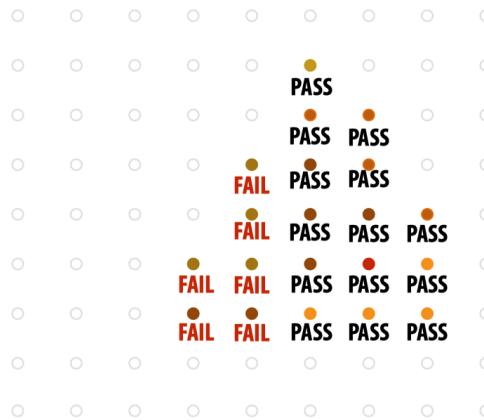
## 第六步:计算样本点的三角形色彩

即采样纹理贴图



## 第七步:执行深度检测(如果启用的话)

同时更新覆盖样本点的深度值(如果需要的话)



## 第八步:更新色彩画出(如果通过深度检测)

同时更新覆盖样本点的深度之(如果需要的话)



## 第九步:

- 针对场景中所有的三角形重复执行步骤一到八。

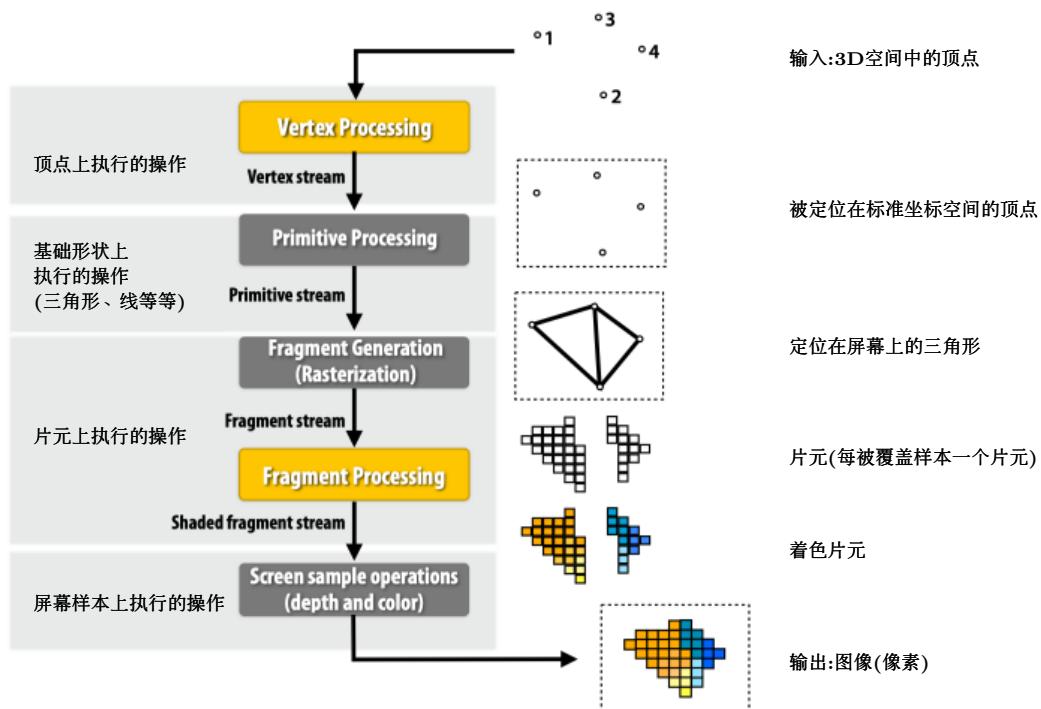
## 几种实时图形API

- OpenGL
- Microsoft Direct3D
- Apple Metal

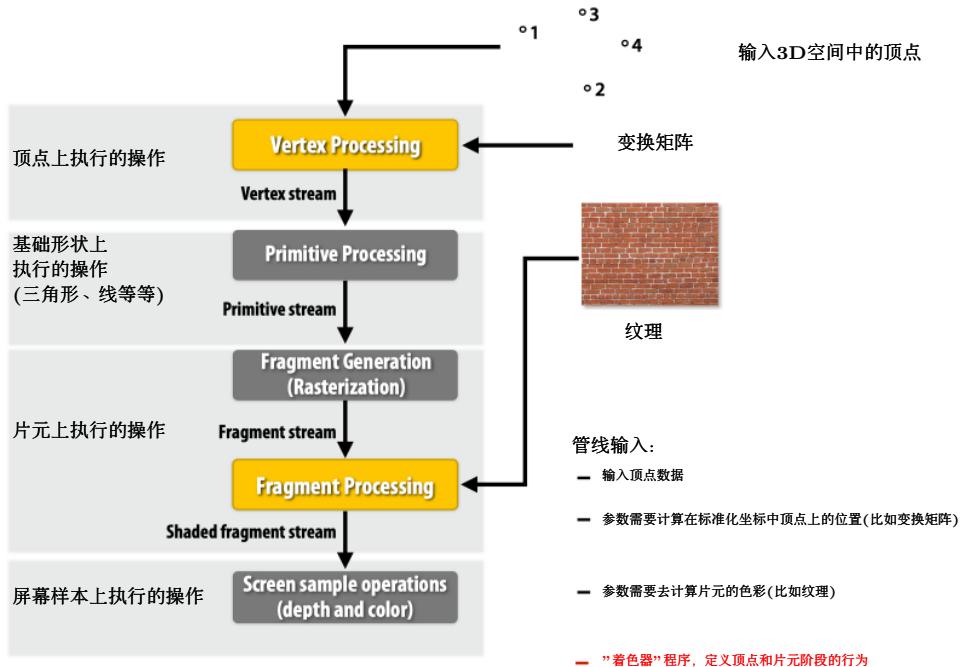
- 现在你已经知道很多这些api内部所实现的算法：绘制三角形（关键的变换和光栅化），纹理映射，经由超级采样的抗锯齿方法，等等。
- 网络中到处都是如何使用这些API变成的有用教程。

## OpenGL/Direct3D图形管线\*

结构化渲染计算为一系列操作，这些操作和顶点、基础形状、片元(fragment)和屏幕样本相关



## OpenGL/Direct3D图形管线\*



## 着色器程序

定义顶点处理和片元处理阶段的行为  
描述在单个顶点（或单个片元）上的操作。

## 片元着色器程序GLSL样例

```
uniform sampler2D myTexture;           程序参数
uniform vec3 lightDir;
varying vec2 uv;                      每片元属性
varying vec3 norm;                    (由光栅化器插值)

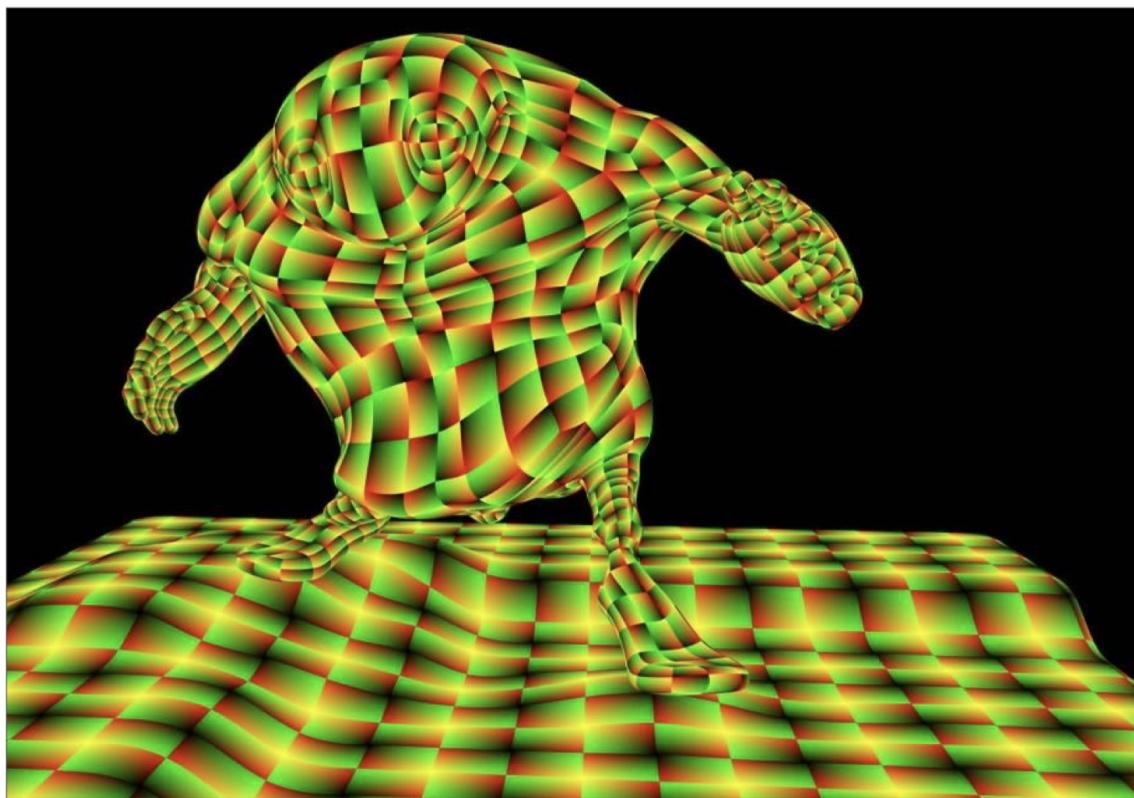
void diffuseShader()
{
    vec3 kd;                         采样表面反照率
    kd = texture2d(myTexture, uv);    (反射色彩)来自纹理
    kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);
    gl_FragColor = vec4(kd, 1.0);
}
```

着色器函数在每个片元上执行一次  
在对应于片元的样本点上的  
表面的输出色彩  
(这个着色器程序执行一次纹理查找以获取在这个点上的材料  
的颜色, 然后执行简单的光照计算)

着色器输出表面色彩  
通过入射辐照度(入射光)  
调整表面反照率

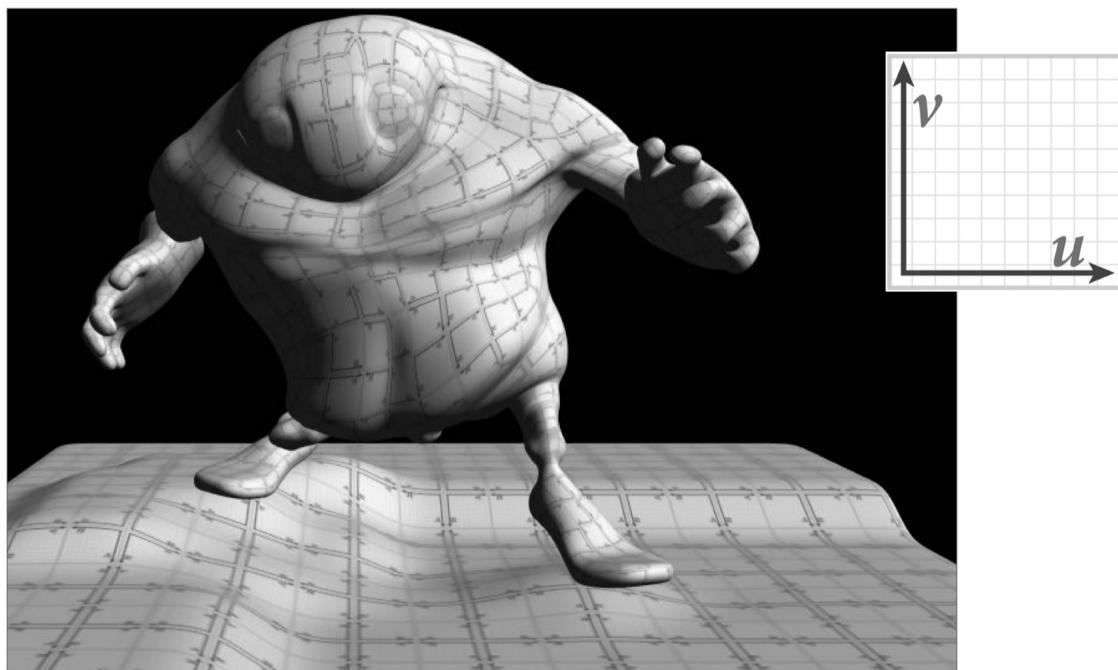
## 纹理坐标可视化

定义表面上的点到纹理域中的点(uv)之间的映射



红色通道 = u, 绿色通道 = v 所以  $uv=(0,0)$  为黑色,  $uv=(1,1)$  为黄色

渲染后的结果(针对每个像素评估了片元着色器之后)





## 目标:渲染高度复杂的3D场景

- 在一个场景中存在10万级到百万级数量的三角形
- 复杂的顶点和片元着色器计算
- 高分辨率屏幕输出(2-4百万像素+超采样)
- 30-60 fps(帧/秒)

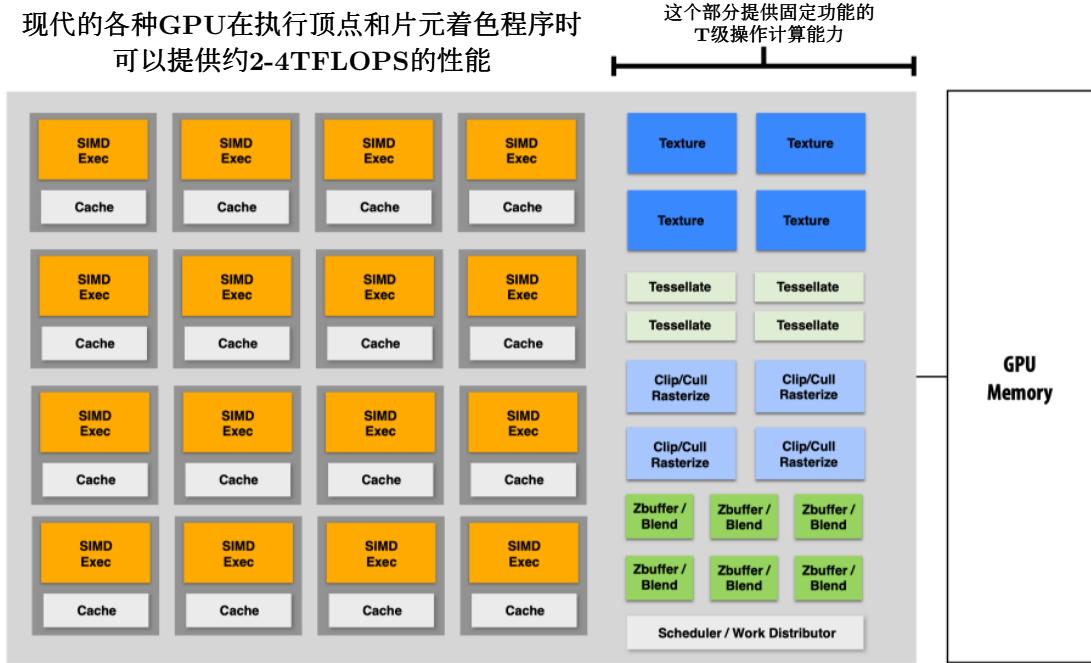
## 图形管线实现:GPUs

专有处理器用于执行图形管线计算



## GPU:多种类,多核处理器

现代的各种GPU在执行顶点和片元着色程序时  
可以提供约2-4TFLOPS的性能



选修Kayvon的Visual Computing System课程(CS348V)寻求更多细节

## 总结

- 借助深度缓存, 遮挡(occlusion)在每个屏幕样本上被单独解析
- 针对半透明表面的Alpha合成
  - 预乘alpha组成了简单重复的合成
  - “Over”合成操作不可互换: 要求三角形以从后到前 (或者从前到后) 的顺序被处理
- 图形管线:
  - 以一系列在顶点、基础形状 (三角形, 线条等) 、片元、屏幕样本上的操作进行结构化渲染操作
  - 图形管线的部分行为是借助着色器程序通过应用级别定义的
  - 图形管线操作通过高度和优化过地平行处理器以及固定功能硬件实现 (GPUs)