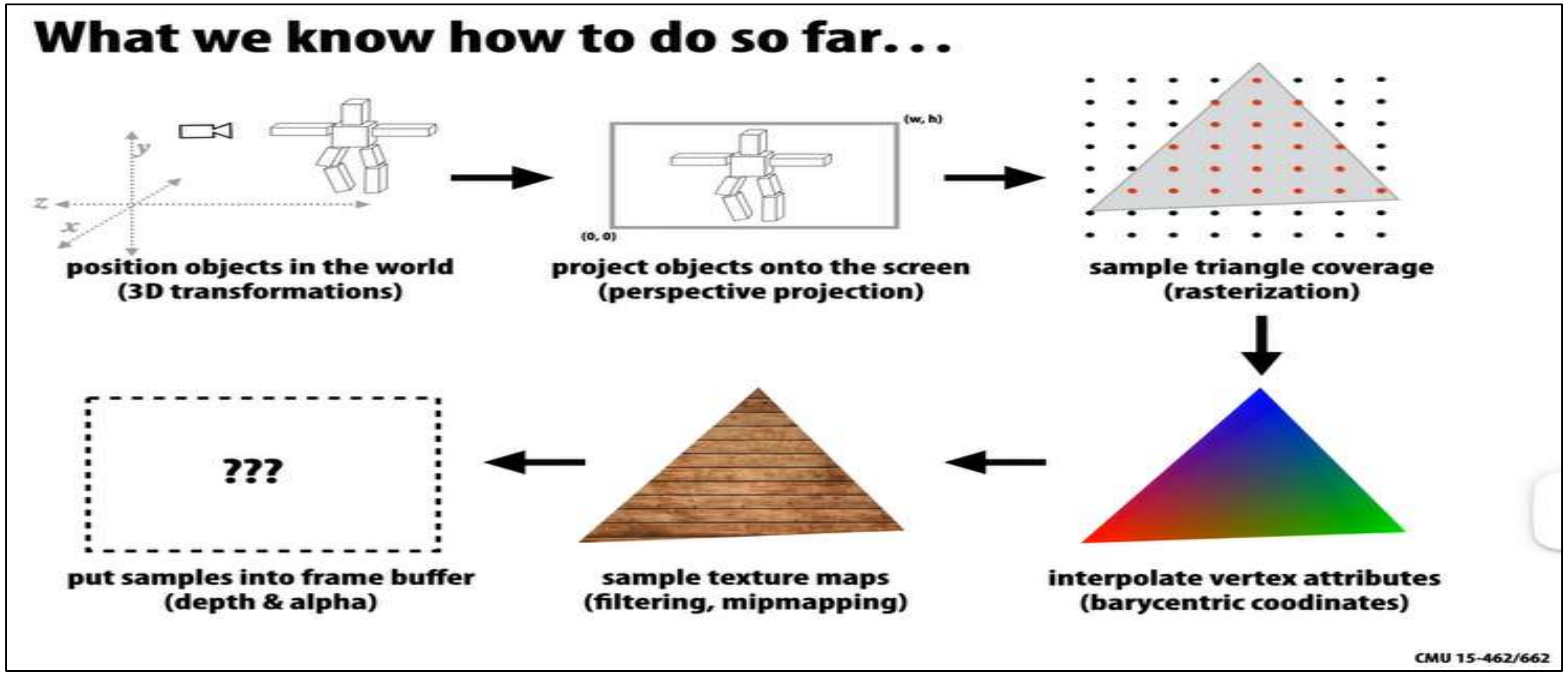
Recap

- ➢Geometry(Objects Modeling, set Scene)
- ➢Viewing(from MC to DC, set Camera and Norm Projection)
- ➢Rasterization(from Vertex to Fragment , set Light and Texture)

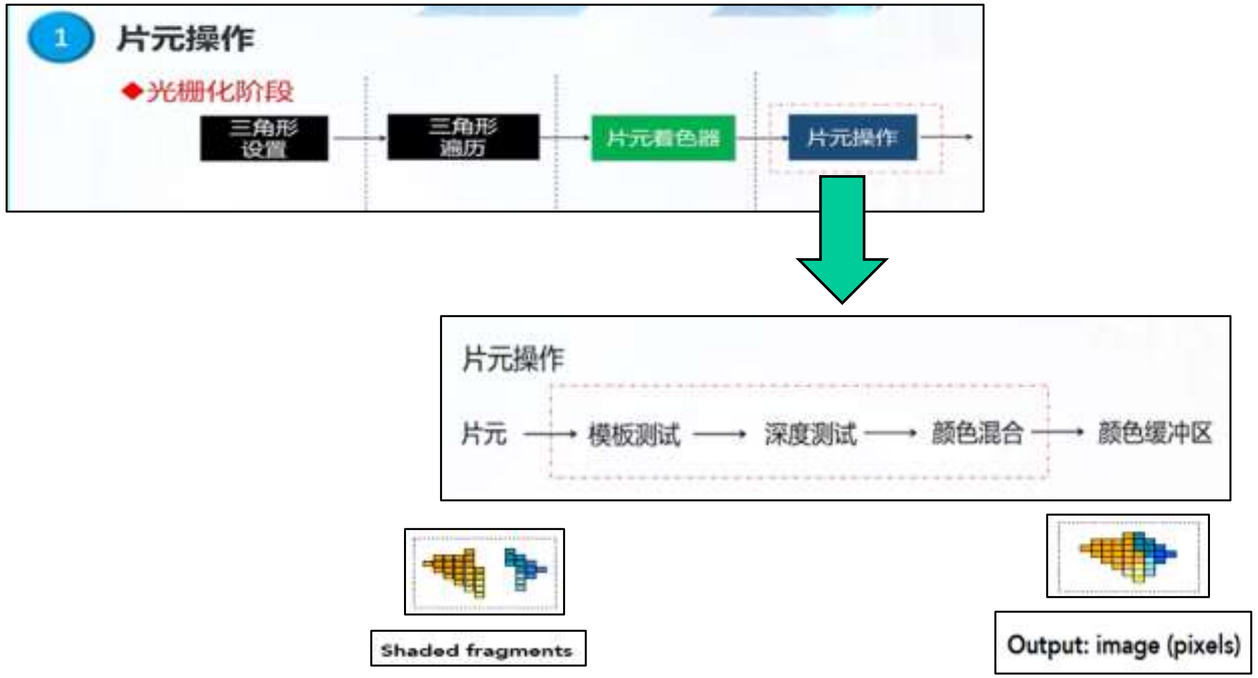

What we know how to do so far...
y
z
x
position objects in the world (3D transformations)
(w, h)
(0, 0)
project objects onto the screen (perspective projection)
sample triangle coverage (rasterization)
interpolate vertex attributes (barycentric coodinates)
sample texture maps (filtering, mipmapping)
???
put samples into frame buffer (depth & alpha)
CMU 15-462/662

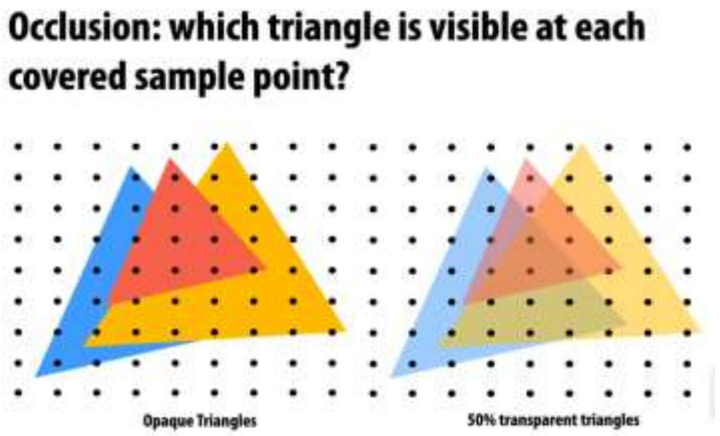
The University of New Mexico

Today's theme1

➢Fragment Operation“片元操作”

- 对每片元进行操作，只有通过了“片元操作”，片元颜色才会写入颜色缓冲区。
- 一般按顺序包含三种操作：“模板测试”，“深度测试”，“颜色混合/alpha测试”
- 一般由程序启动或关闭（enable/disable），若开启则渲染管线根据参数自动实现。




Occlusion: which triangle is visible at each covered sample point?
Opaque Triangles
50% transparent triangles

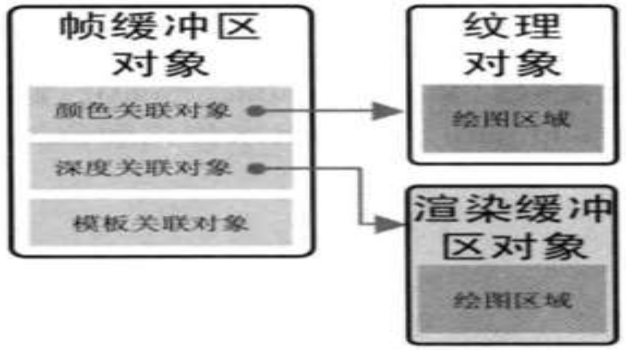
The University of New Mexico

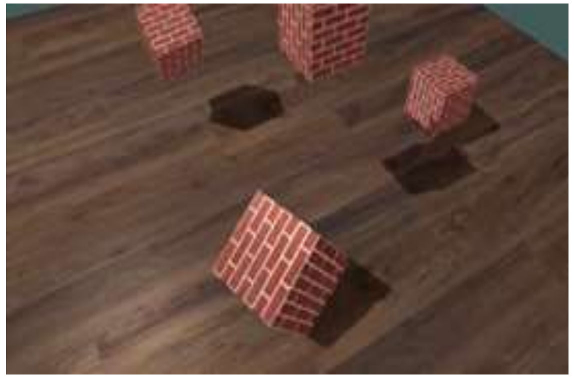
Today’s theme2

- 其它颜色合成技术(离散技术)
  - 其它alpha混合
    - **雾化，反走样、运动模糊、泛光效果等**
  - 图像合成技术
    - **图像增强，边缘提取等**
  - 离屏渲染OSR(Off-Screen Rendering)技术
    - **阴影贴图shadowmap**

original

enhanced





Outline

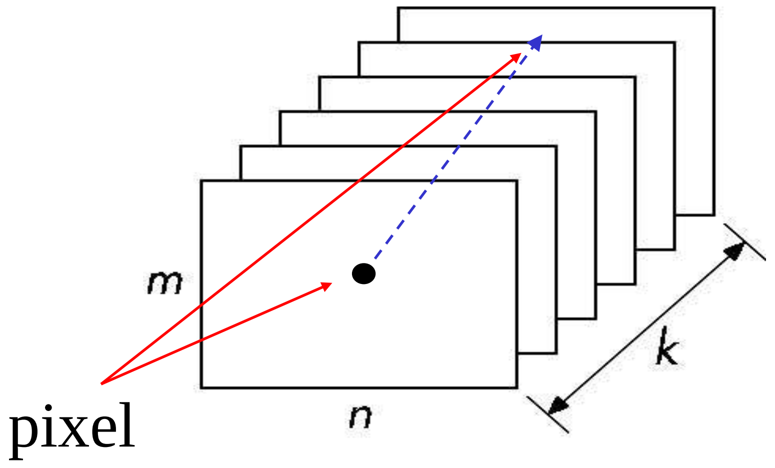
- 帧缓存和帧缓存操作Buffer and Buffer Operation

- 片元操作Fragment Operation
  - Stencil Test模板测试
  - Depth Test 深度测试
  - Color Blend 颜色混合
    - ➢半透明效果Translucence（Alpha Blending）

- 其它颜色合成技术Color Blending
  - Composite 合成技术
    - – Image processing图像处理
  - OSR离屏渲染技术
    - – Shadow Map 阴影贴图

Buffer and Buffer Operation

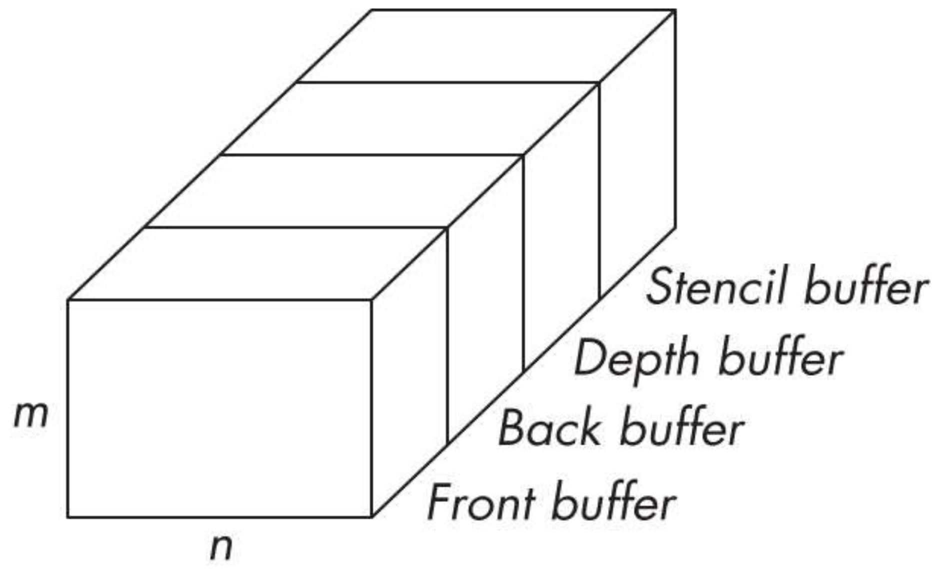
- 片元处理阶段主要是对片元fragment和缓存Buffer进行操作，不同于一般内存的存储和操作，所以首先应该了解缓存的存储和操作特点。
- Define a buffer by
  - its spatial resolution (n x m) and
  - its depth (or precision) k (the number of bits per pixel)
    - Most RGBA buffers 8 bits per component: K=**8bit * 4 component=32 bit,**
    - Latest are floating point (IEEE)



The University of New Mexico

Buffer and Buffer Operation(cont.)

- on graphics card, have different type buffers:
  - Default front and back color buffers(颜色缓存) Holds colors
  - Stencil buffer(模板缓冲) Holds masks
  - Depth buffer（深度缓存）Holds Deep Values (Z values)
  - ……

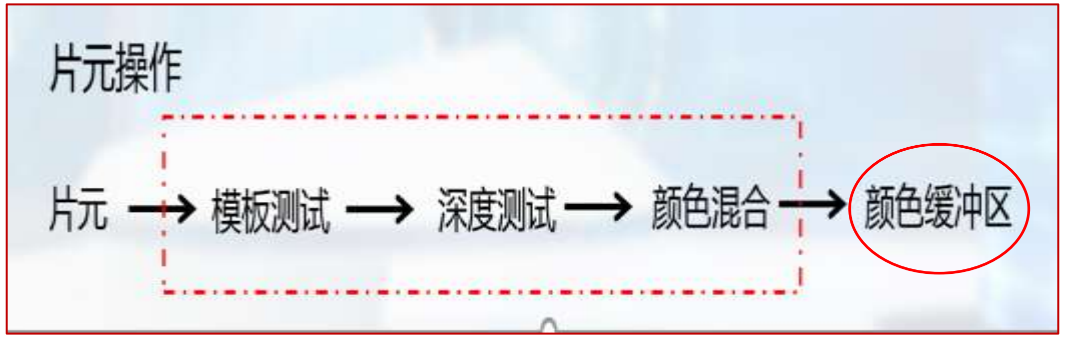


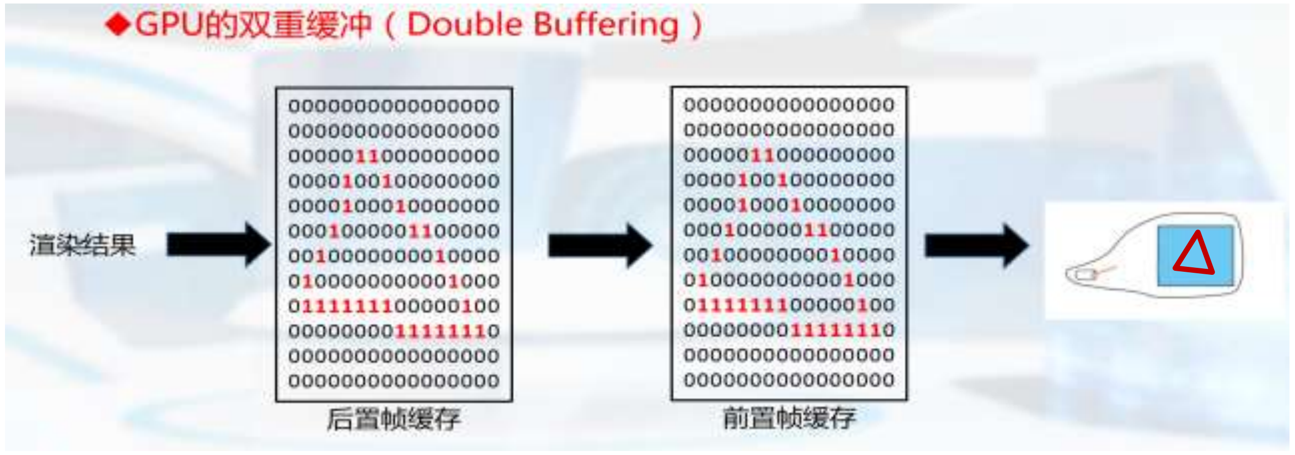
The University of New Mexico

Buffer and Buffer Operation(cont.)

➢**Color Buffer（颜色缓存）**

- 片元处理后的最终屏幕画面每个像素点的颜色值的存放地
- 系统“默认”的用于显示一帧画面用到的帧缓存
- 双帧结构又分为**后帧**(用于存放渲染结果)和**前帧**(用于屏幕显示输出)



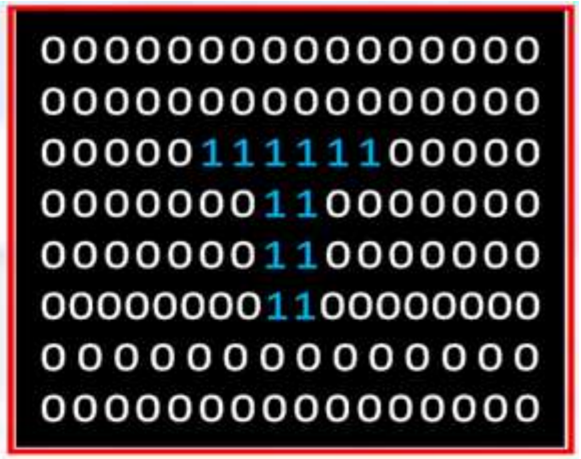


Buffer and Buffer Operation(cont.)

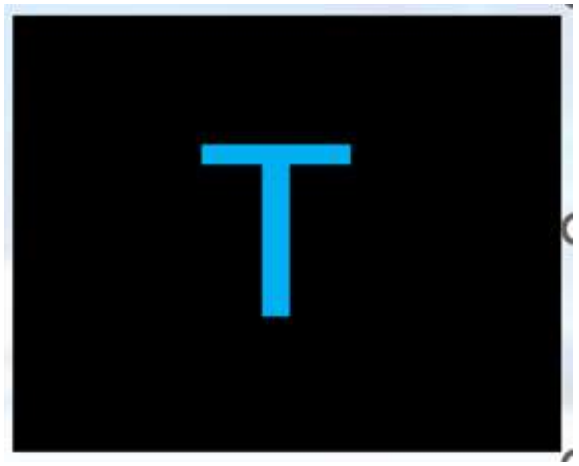
- Stencil Buffer 模板缓存
  - Stencil buffer(模板缓冲) Holds masks
  - 模板分辨率和屏幕分辨率一致

片元

模板缓存



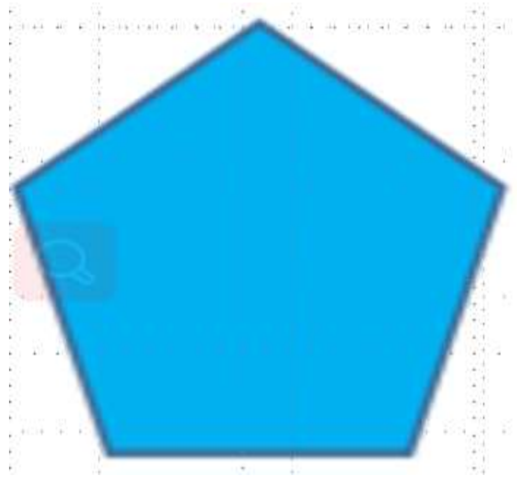
屏幕显示



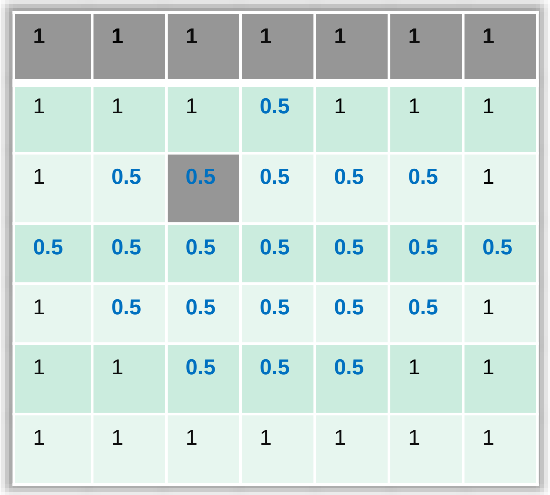
Buffer and Buffer Operation(cont.)

- Depth buffer（深度缓存）
  - Depth buffer（深度缓存）Holds Deep Values (Z values)
  - 深度缓存分辨率和屏幕分辨率一致

片元

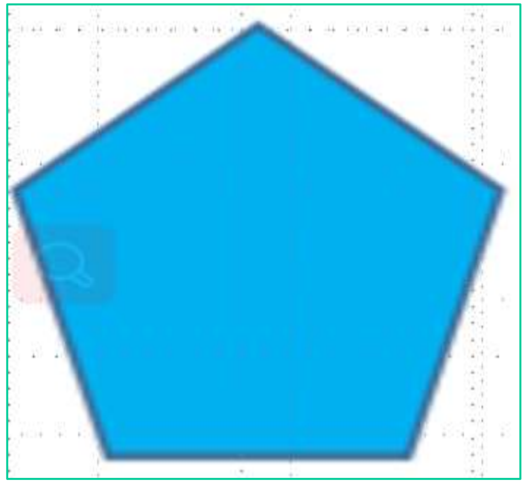


深度缓存



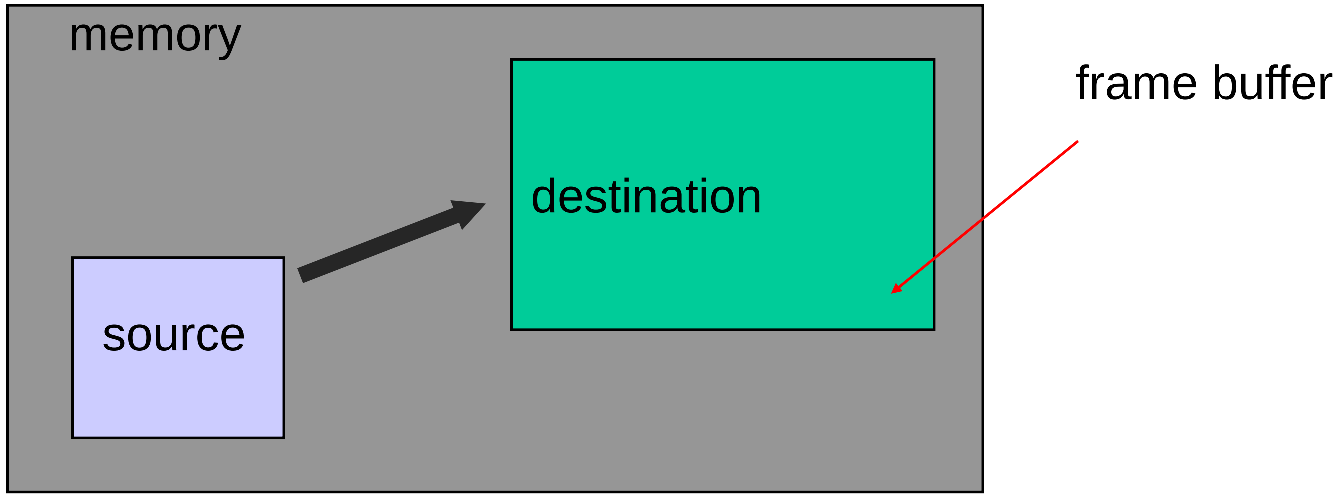
1	1	1	1	1	1	1
1	1	1	0.5	1	1	1
1	0.5	0.5	0.5	0.5	0.5	1
0.5	0.5	0.5	0.5	0.5	0.5	0.5
1	0.5	0.5	0.5	0.5	0.5	1
1	1	0.5	0.5	0.5	1	1
1	1	1	1	1	1	1

屏幕显示



Buffer and Buffer Operation(cont.)

- Buffer Operations：
  - Bit Block Transfer (BitBlt)(位块转移/位块传输)
    - consider all of memory as a large two-dimensional array of pixels（二维像素矩阵）
    - read and write rectangular block of pixels(读写矩形像素块)



writing into the frame buffer

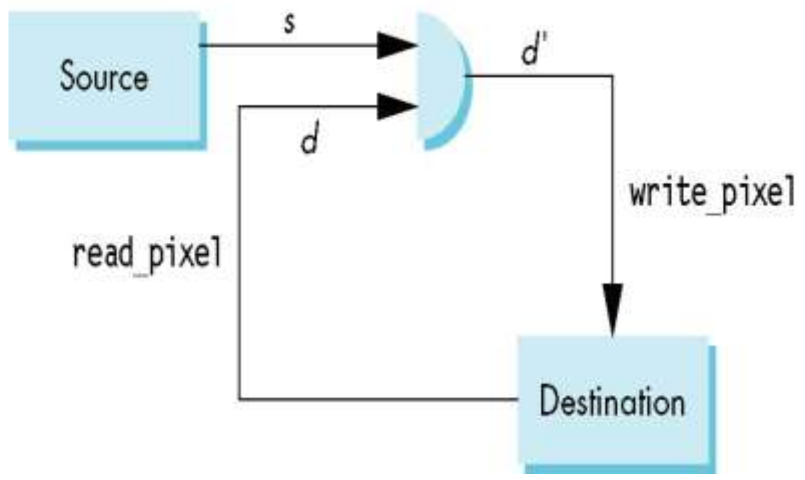
The University of New Mexico

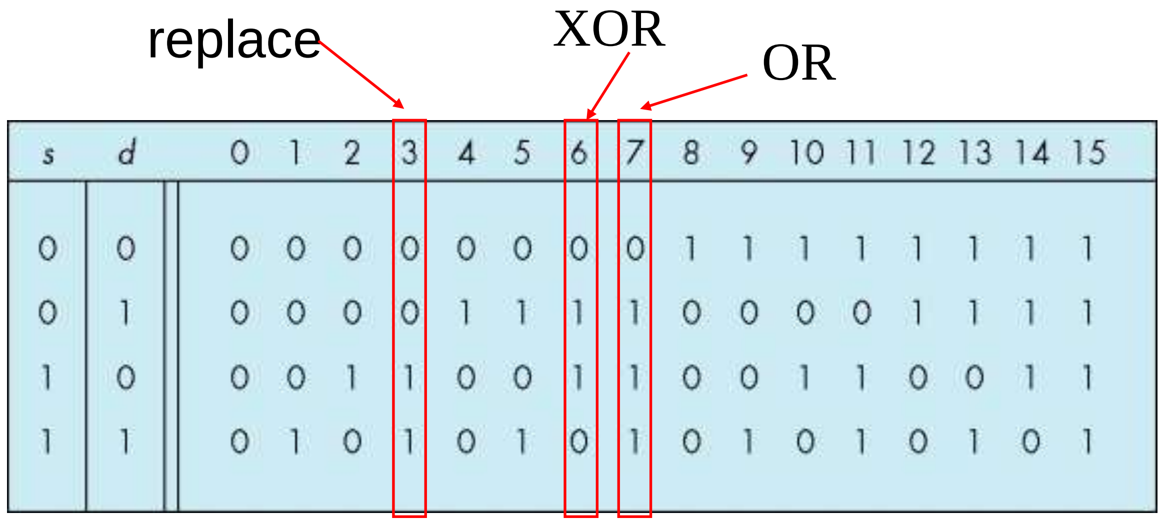
Buffer and Buffer Operation(cont.)

- Buffer Operations(cont.)
  - Bit operations: act on blocks of bits with <u>single instruction</u>

    writing Modes：d’=**function**(s, d)

    - ➢Note: Writing Model is Read destination pixel before Writing
    - ➢Source and destination bits are combined bitwise(源位和目标位是按位组合的)
    - ➢16 possible functions (one per column in table)，such as **Replace**, XOR, OR,……





replace XOR OR

s	d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The University of New Mexico

Outline

- 帧缓存和帧缓存操作Buffer and Buffer Operation

- 片元操作Fragment Operation
  - Stencil Test模板测试
  - Depth Test 深度测试
  - Color Blend 颜色混合
    - ➢半透明效果Translucence（Alpha Blending）

- 其它颜色合成技术Color Blending
  - Composite 合成技术
    - – **Image processing图像处理**
  - OSR离屏渲染技术
    - – **Shadow Map 阴影贴图**

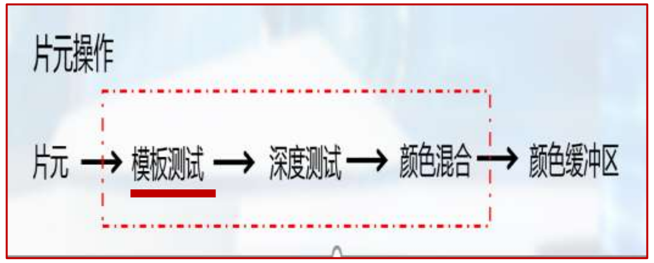
The University of New Mexico

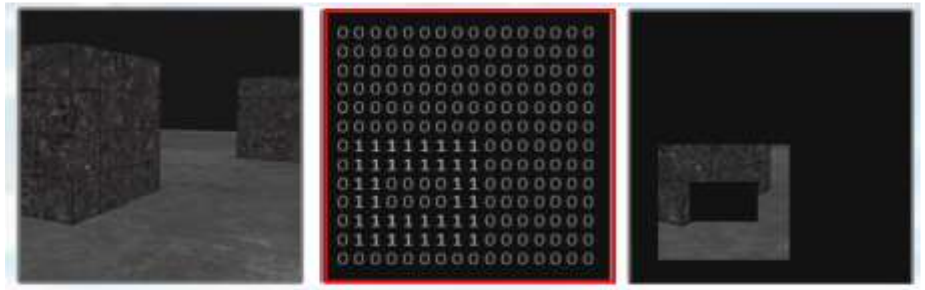
片元操作Fragment Operation

1. 模板测试（stencil test）

- 模板测试：主要用于根据模板缓冲区中的值来决定某些像素片段是否可见。模板测试在深度测试之前进行，提供了一种强大的机制来实现各种复杂的视觉效果，如遮罩、镜像、轮廓渲染等
- 默认是关闭，一般需要在程序中开启和设置相关的参数，由系统自动实现。

```
（openGL中的设置命令）
// 启用模板测试
glEnable(GL_STENCIL_TEST);

// 设置模板测试参数
glStencilFunc(GL_EQUAL, 1, 0xFF); // 只有当模板值等于1时才通过测试
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE); // 测试失败时保持原值，
测试通过时替换为新值

// 清除模板缓冲区
glClearStencil(0);
glClear(GL_STENCIL_BUFFER_BIT);

// 提交绘制调用
```





```
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0111111110000000
0111111110000000
0110000110000000
0110000110000000
0111111110000000
0111111110000000
0000000000000000
```

片元
示　　模板缓存　　屏幕显

The University of New Mexico

片元操作Fragment Operation（cont.)

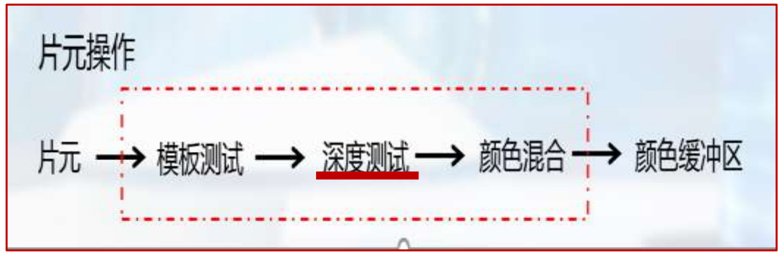
➢2. 深度测试（Depth Test）

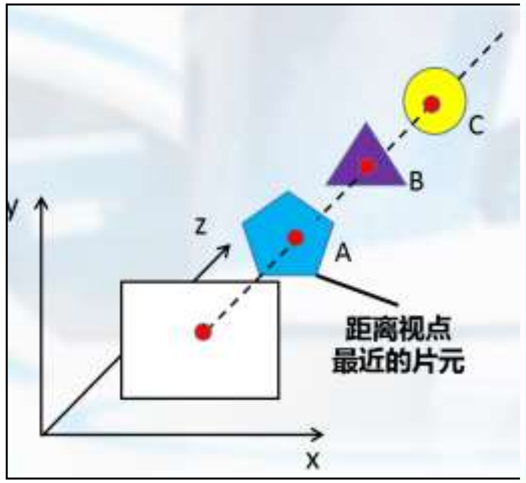
深度缓存（Z-buffer）:存储每个片元的深度信息（或Z值，一般Z值越大，深度越大)

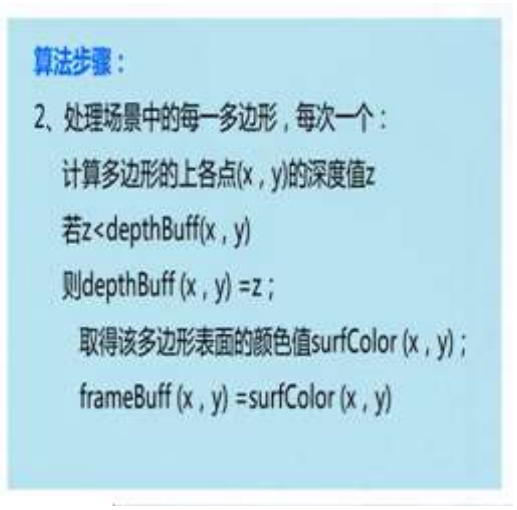
➢**深度测试:据**每片元的深度值与深度缓存中已有深度比较，判其颜色是否写入颜色缓存

➢默认是关闭，一般需要在程序中开启和设置相关的参数，由系统自动实现。

```
(webGL设置语句)
//设置Z-BUFFER初值
gl.clear(gl.DEPTH_BUFFER_BIT);

//开启Z-BUFFER算法
gl.enable(gl.DEPTH_TEST);

//关闭Z-BUFFER算法
 gl.disable(gl.DEPTH_TEST);
```






算法步骤:

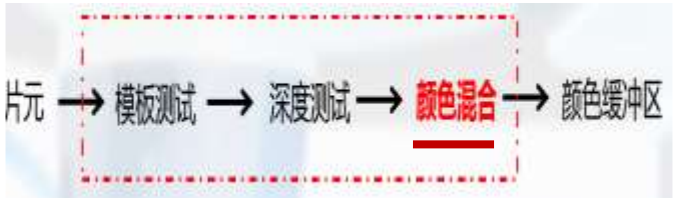
```
2、处理场景中的每一多边形，每次一个：
   计算多边形的上各点(x，y)的深度值z
   若z<depthBuff(x，y)
   则depthBuff (x，y) =z；
     取得该多边形表面的颜色值surfColor (x，y)；
     frameBuff (x，y) =surfColor (x，y)
```

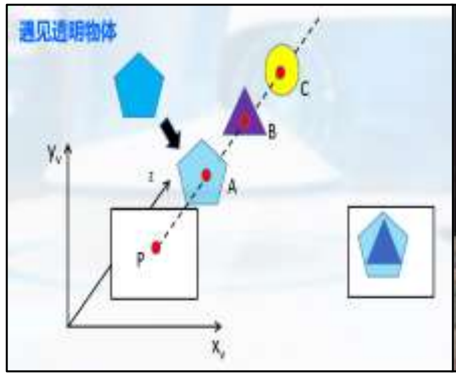
片元操作Fragment Operation(cont.)

3.color Blending 颜色混合

- 颜色混合: 将当前存储在颜色缓冲区中的颜色(目标颜色)与将要画上去的颜色(源颜色)通过某种方式混合在一起，再写入颜色缓冲区。
- 默认是关闭，一般需要在程序中开启和设置相关的参数，由系统根据参数自动实现

```
(webGL设置语句)
//打开混合功能
gl.enable(gl.BLEND)

//设置混合因子
gl.blendFunc(source_factor,destination_factor);

//关闭混合
gl.disable(gl.BLEND)
```





The University of New Mexico

片元操作Fragment Operation(cont.)

➢3.color Blending 颜色混合(cont.)

- **Blending Equation混合方程:** $C'_d = s\, C_s + d\ \ C_d$
  - ➢During rendering we can expand our writing model to use **RGBA** values

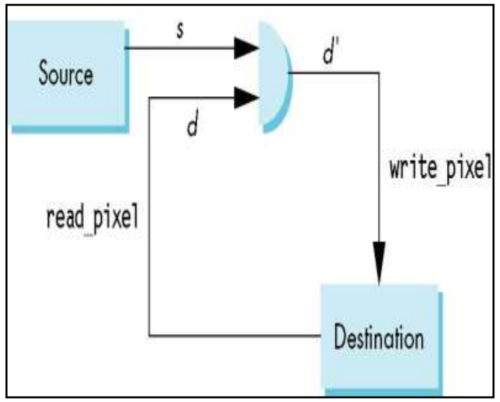
    source color: $C_{s}=[R_s, G_s, B_s, A_s]$,

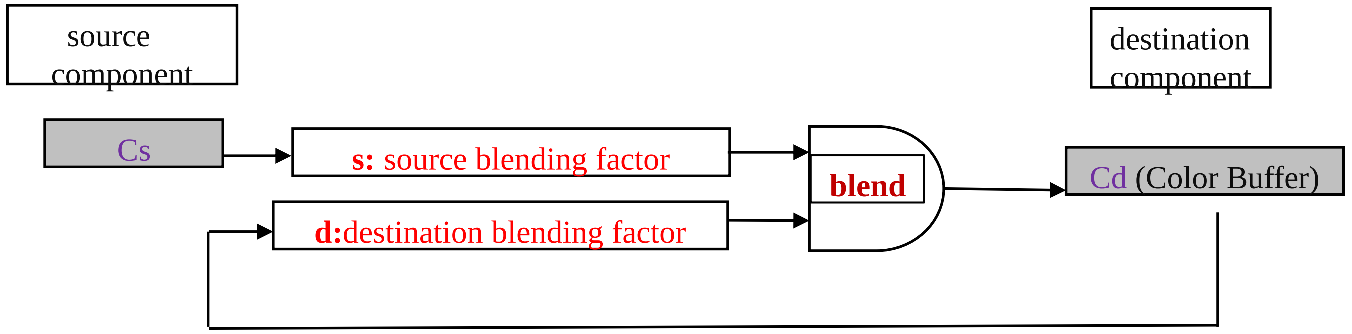
    destination color: $C_d=[R_d, G_d, B_d, A_d]$
  - ➢define source and destination blending factors s, d for each RGBA component

    sourse blending factor: $s = [s_r, s_g, s_b, s_\alpha]$,

    destination blending factor: $d = [d_r, d_g, d_b, d_\alpha]$

Blend Equation: $C'_d = [R_d, G_d, B_d, A_d] = [s_r R_s + d_r R_d,\ \ s_g G_s + d_g G_d,\ \ s_b B_s + d_b B_d,\ \ s_\alpha A_s + d_\alpha A_d]$





The University of New Mexico

Outline

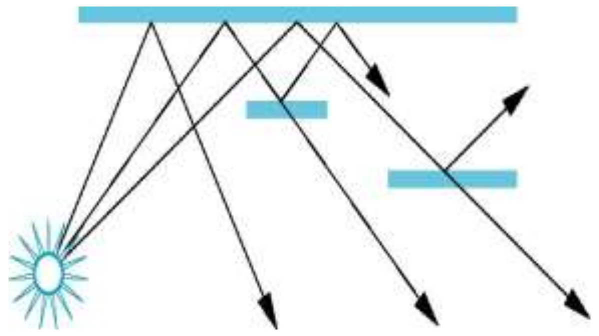
- 帧缓存和帧缓存操作Buffer and Buffer Operation

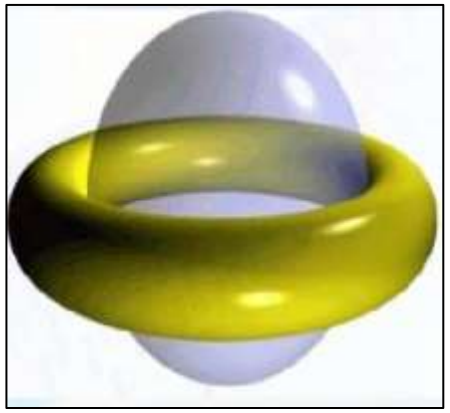
- 片元操作Fragment Operation
  - Stencil Test模板测试
  - Depth Test 深度测试
  - Color Blend 颜色混合
    - **半透明效果Translucence（Alpha Blending）**

- 其它颜色合成技术Color Blending
  - Composite 合成技术
    - **Image processing图像处理**
  - OSR离屏渲染技术
    - **Shadow Map 阴影贴图**

The University of New Mexico

Translucence（Alpha Blending）





- 光栅化渲染管线的局限：
  - 每个物体表面的颜色是单独计算的，后渲染的片元颜色直接覆盖先前的片元颜色即： $C'_d = C_s$
  - 若程序开启了“深度检测”，则离视点更近片元色会“替换**replace**”先前的像素颜色： $C'_d = C_{near}$

- 半透明Translucence（ α混合）:
  - 采用颜色混合方程“**blend equation** $C'_d = s\ C_s + d\ C_d$” 可显示被遮挡但能看见的表面颜色

Translucence（Alpha Blending）（cont.)

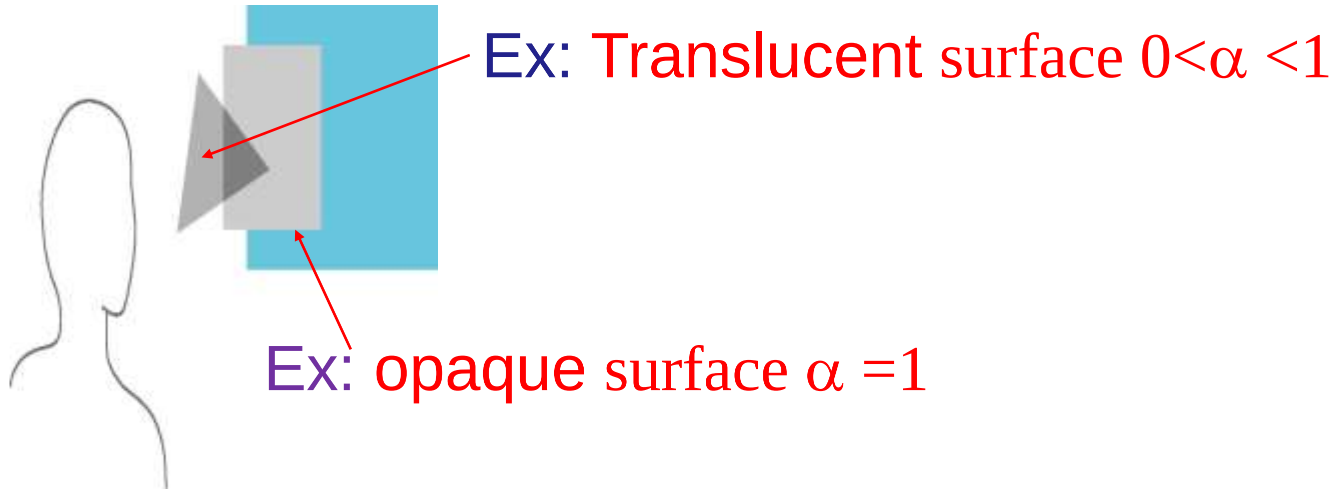
- **What is Alpha (α)? 不透明度**
  - **Use A component of RGBA（or α of RGBα） to store opacity不透明度**

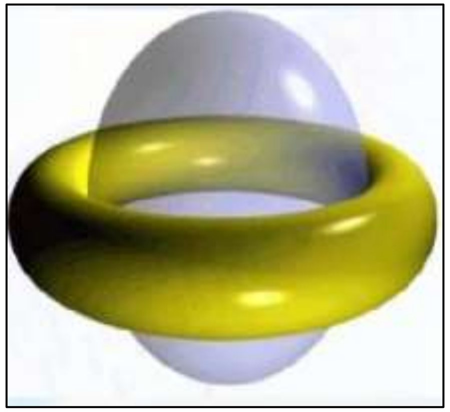
    (采用颜色通道RGBA中的A通道值(即α值)，表示该表面材质的不透明度）

α =1: Opaque不透明 surfaces permit no light to pass through表面"不允许光线通过"

α=0：Transparent透明surfaces permit all light to pass表面"允许所有光线通过"

0<α<1：Translucent半透明 surfaces pass some light 表面"允许部分光线通过"





Translucence（Alpha Blending）(cont.)

- Use "Alpha Blending" to Realize Translucence Effects
  - enable blending $C'_d = s\,C_s + d\,C_d$ and pick source and destination factors,
    - `gl.enable(gl.BLEND);`
    - `gl.blendFunc(source_factor,destination_factor)`   //通常s+d=1
    - Note: Only certain factors supported for "source_factor" and "destination_factor "
      - `gl.ZERO,gl.ONE,`   //注意两者之和等于1
      - `gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA`
      - `gl.DST_ALPHA, gl.ONE_MINUS_DST_ALPHA`
    - 透明采用:gl.blendFunc(`gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA`);
      - "`SRC_ALPHA`" as the source blending factors s, is the "A" of source RGBA
      - "`ONE_MINUS_SRC_ALPHA`" as the destination blending factors d

alpha Blending Equation: $C'_d = [R_d, G_d, B_d, A_d] = [A_s R_s + (1-A_s) R_d,\ A_s G_s + (1-A_s) G_d,\ A_s B_s + (1-A_s) B_d,\ A_s A_s + (1-A_s) A_d]$

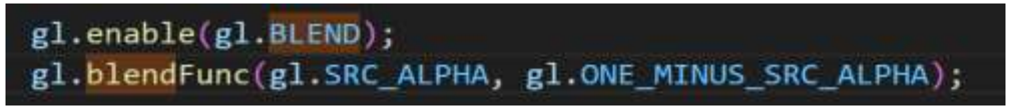
ref: *Blend Equation:* $C'_d = [R_d, G_d, B_d, A_d] = [s_r R_s + d_r R_d,\quad s_g G_s + d_g G_d,\quad s_b B_s + d_b B_d,\quad s_\alpha A_s + d_\alpha A_d]$

The University of New Mexico

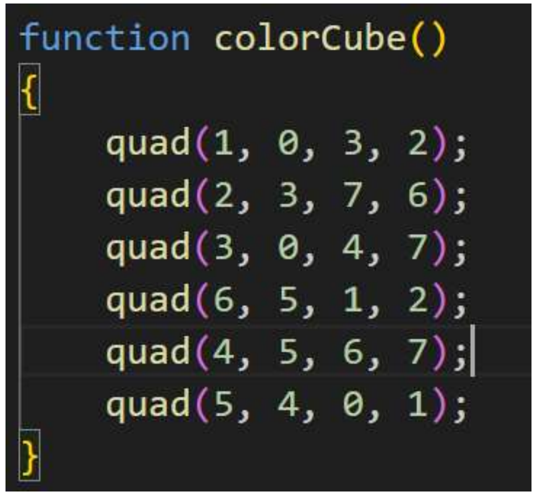
Translucence（Alpha Blending）(cont.)

➢Example:

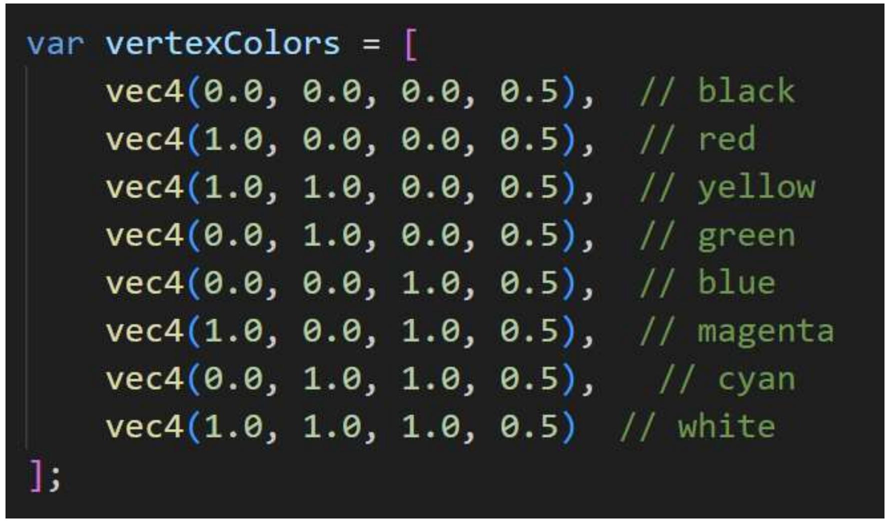
- ➢\AngelCode8E\08\cubit: rotating translucent cube. Hidden-surface removal can be toggled on and off
- ➢6个面绘制顺序：红，黄，绿， 兰，品红，青色



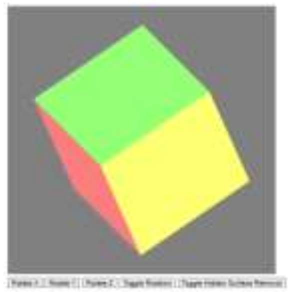
```
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

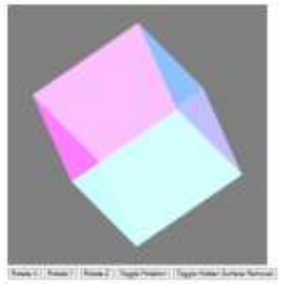


```
function colorCube()
{
    quad(1, 0, 3, 2);
    quad(2, 3, 7, 6);
    quad(3, 0, 4, 7);
    quad(6, 5, 1, 2);
    quad(4, 5, 6, 7);
    quad(5, 4, 0, 1);
}
```



```
var vertexColors = [
    vec4(0.0, 0.0, 0.0, 0.5),  // black
    vec4(1.0, 0.0, 0.0, 0.5),  // red
    vec4(1.0, 1.0, 0.0, 0.5),  // yellow
    vec4(0.0, 1.0, 0.0, 0.5),  // green
    vec4(0.0, 0.0, 1.0, 0.5),  // blue
    vec4(1.0, 0.0, 1.0, 0.5),  // magenta
    vec4(0.0, 1.0, 1.0, 0.5),   // cyan
    vec4(1.0, 1.0, 1.0, 0.5)  // white
];
```





//顶点颜色RGBA的A=0.5,每个面都是半透明表面

The University of New Mexico

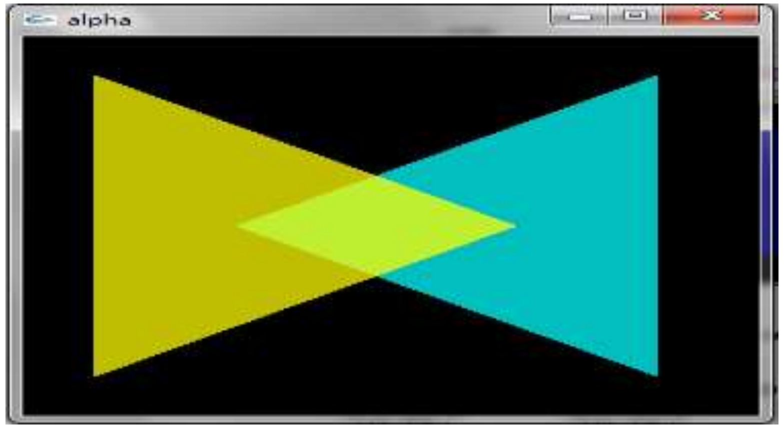
Translucence（Alpha Blending）（cont.)

➤问题：透明效果依赖于表面的渲染顺序！order dependency

gl.BlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)

$$R'_{buffer} = \alpha_{yellow}\ R_{yellow} + (1-\alpha_{yellow})\ R_{cyan}$$

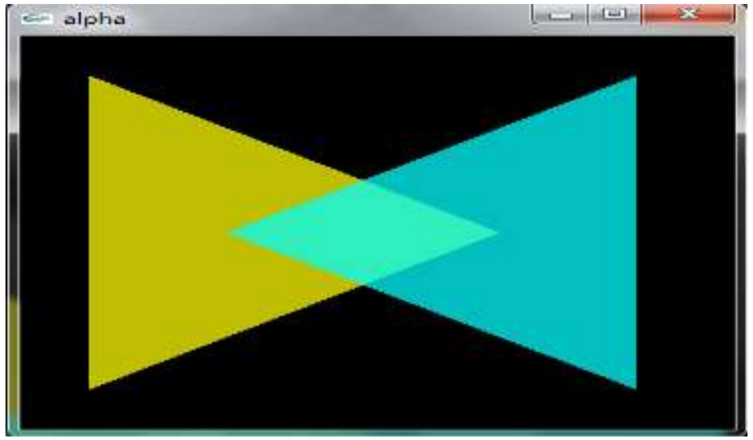
$$\begin{pmatrix}0.75\\1.0\\0.25\\1\end{pmatrix} = 0.75\begin{pmatrix}1.0\\1.0\\0.0\\0.75\end{pmatrix} + (1-0.75)\begin{pmatrix}0.0\\1.0\\1.0\\0.75\end{pmatrix}$$



gl.BlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)

$$R'_{buffer} = \alpha_{cyan}\ R_{cyan} + (1-\alpha_{cyan})\ R_{yellow}$$

$$\begin{pmatrix}0.25\\1.0\\0.75\\1\end{pmatrix} = 0.75\begin{pmatrix}0.0\\1.0\\1.0\\0.75\end{pmatrix} + (1-0.75)\begin{pmatrix}1.0\\1.0\\0.0\\0.75\end{pmatrix}$$



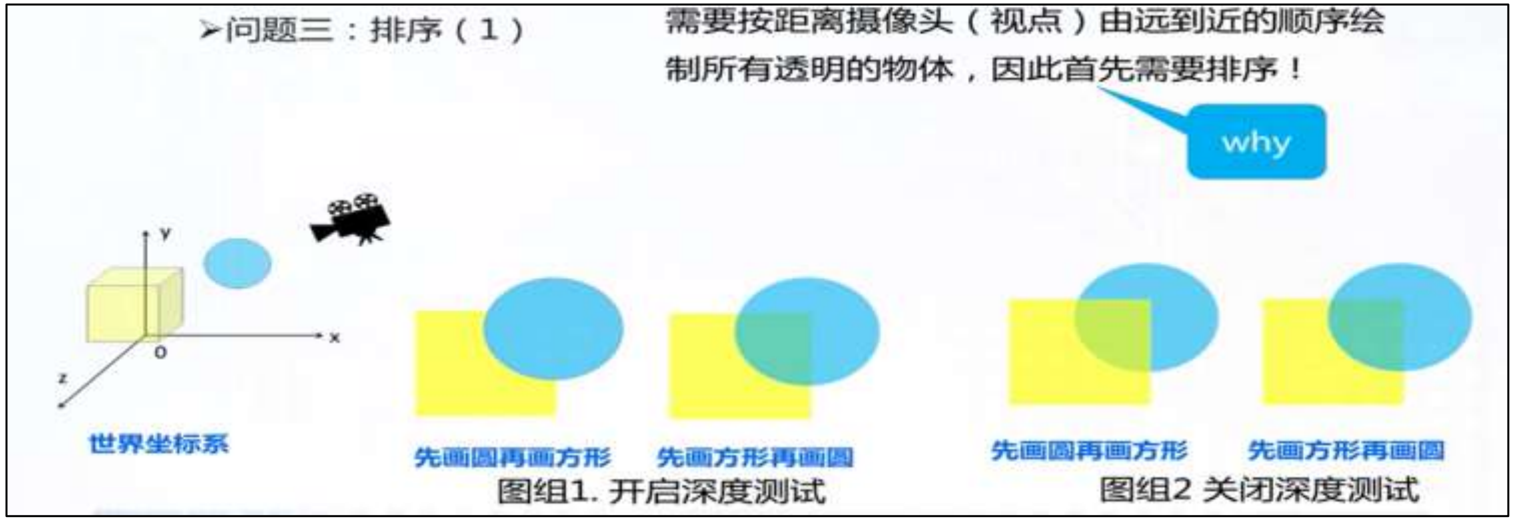
The University of New Mexico

Translucence（Alpha Blending）（cont.)

- When rendering Opaque Objects（当场景只有半透明对象时）

  解决方案：对物体按摄像机顺序排序,绘制时先远后近

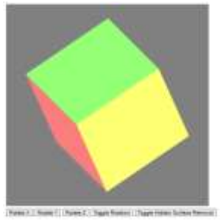
  （这时，无论是否开启了“深度测试”，都能够正确进行“颜色混合”实现透明效果！）



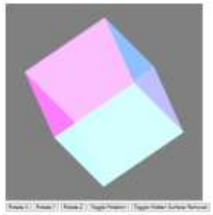
➢问题三：排序（1）

需要按距离摄像头（视点）由远到近的顺序绘制所有透明的物体，因此首先需要排序！

图组1. 开启深度测试　　图组2 关闭深度测试



gl.enable(gl.DEPTH_TEST);



gl.disenable(gl.DEPTH_TEST);

The University of New Mexico

Translucence（Alpha Blending）（cont.)

- When Both Opaque and Translucent Objects
  - **解决方案：**
    - 绘制不透明多边形：会挡住所有它背后的多边形，应该开启深度检测进行处理
    - 绘制半透明多边形：不能完全遮挡其背后的多边形，应该不开启深度检测，但应让其可读，并需要对它们按深度进行排序，按从远到近的顺序进行绘制。
      - **为什么"深度缓存只读"：因为若半透明片元的深度比深度缓存的值大，则表示其被不透明表面遮挡，应舍弃而不进行Alpha混合计算）**

- **在webGL中实现的步骤如下**（参见《webgl编程指南》）

1.开启隐藏面消除功能 gl.enable(gl.DEPTH_TEST);

2.绘制所有不透明的物体(A=1.0)

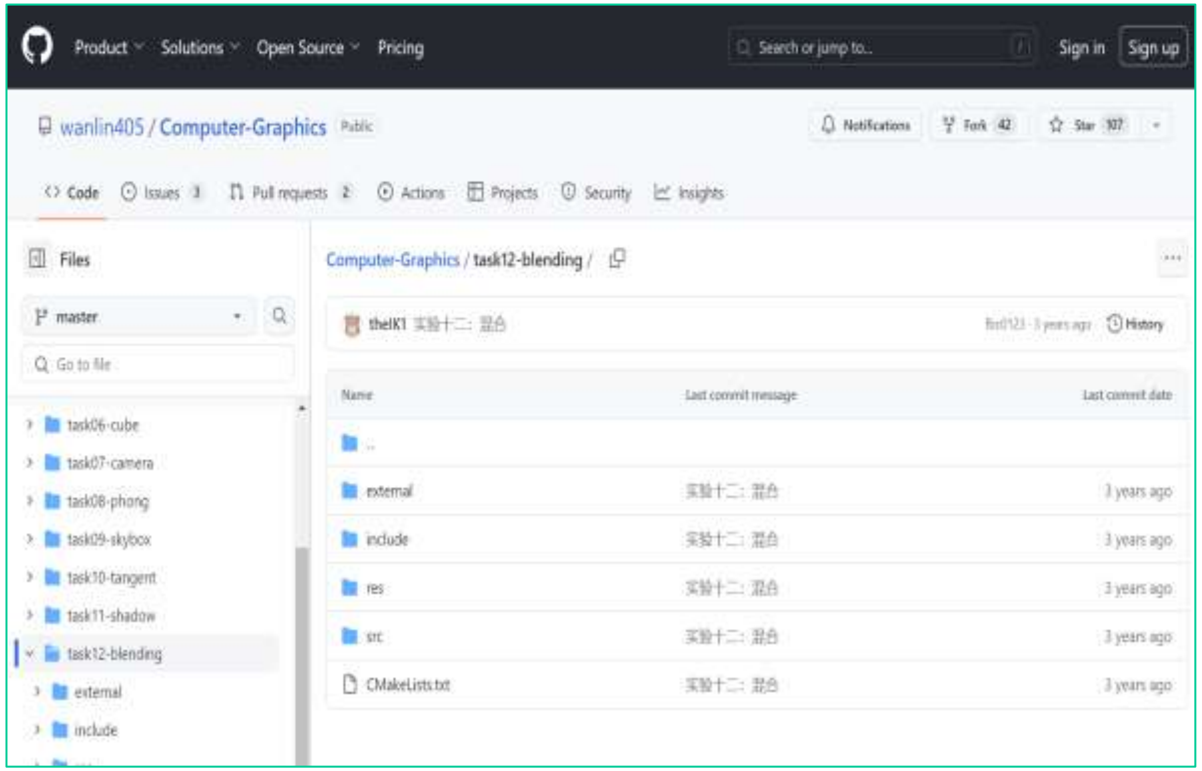
3.锁定用于进行隐藏面消除的深度缓冲区的写入操作，使深度缓冲区只读gl.depthMask(false);

4.绘制所有半透明的物体(A<1.0),并注意将它们进行深度排序，即从后向前顺序绘制。

5.释放深度缓冲区(恢复写入功能)gl.depthMask(true);

Translucence（Alpha Blending）(cont.)

- openGL实现透明效果的例子，参见万琳github代码
  - https://github.com/wanlin405/Computer-Graphics/tree/master/task12-blending



Product · Solutions · Open Source · Pricing · Search or jump to... · Sign in · Sign up

wanlin405 / Computer-Graphics Public · Notifications · Fork 42 · Star 107

Code · Issues 3 · Pull requests 2 · Actions · Projects · Security · Insights

Files · master · Go to file

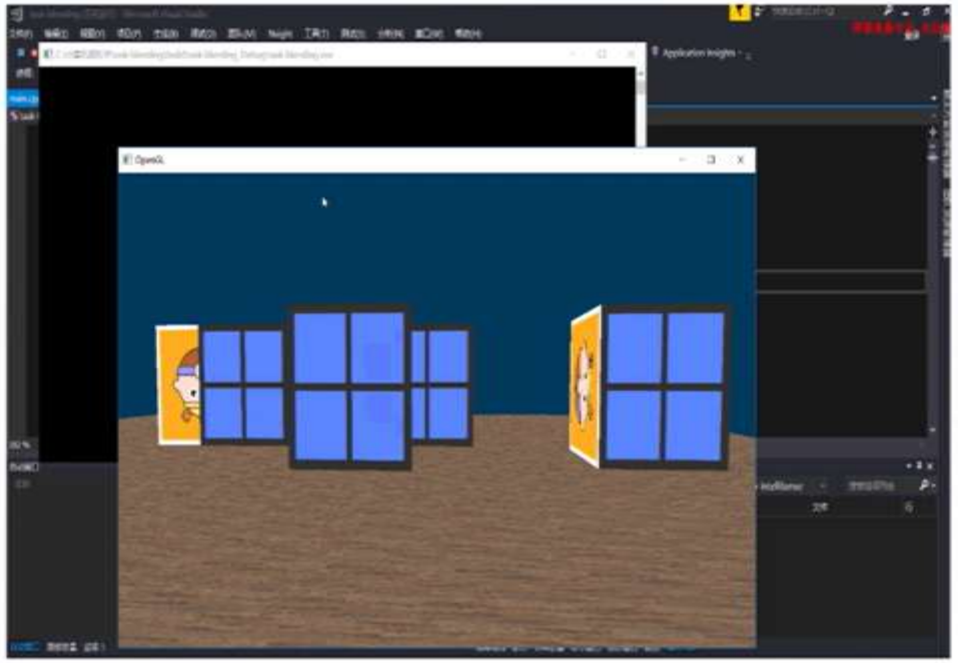
- task06-cube
- task07-camera
- task08-phong
- task09-skybox
- task10-tangent
- task11-shadow
- task12-blending
  - external
  - include

Computer-Graphics / task12-blending /

theIK1 实验十二：混合 · fcd123 · 3 years ago · History

Name	Last commit message	Last commit date
..		
external	实验十二：混合	3 years ago
include	实验十二：混合	3 years ago
res	实验十二：混合	3 years ago
src	实验十二：混合	3 years ago
CMakeLists.txt	实验十二：混合	3 years ago

在下面的视频中使用键盘按键Z（开启）、X（关闭）控制混合的开关，对比效果明显。



The University of New Mexico

Outline

- 帧缓存和帧缓存操作Buffer and Buffer Operation

- 片元操作Fragment Operation
  - Stencil Test模板测试
  - Depth Test 深度测试
  - Color Blend 颜色混合

- **其它颜色合成技术Color Blending**
  - Composite 合成技术
    - **Image processing图像处理**
  - OSR离屏渲染技术
    - **Shadow Map 阴影贴图**

The University of New Mexico

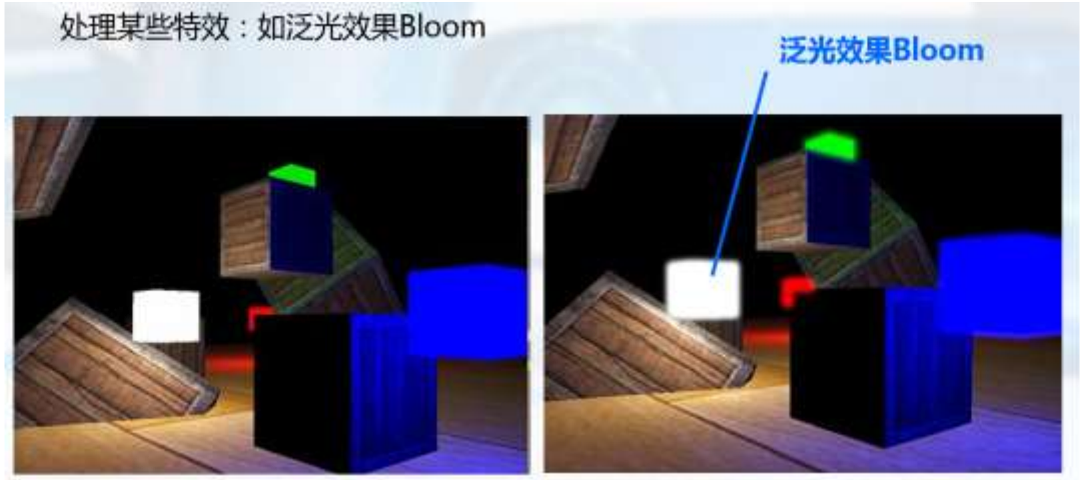
Composite

对多个“片元颜色”进行加权求和得到片元的颜色值，然后再写入帧缓存作像素色

- ➢片元可能来自多幅图像上的多个像素点，也可能是单幅图像上的多个像素点，
- ➢早期管线用累积缓存，现在采用“纹理对象”存储图像，在“片元着色器”里进行颜色合成

$$C' = \sum_{1}^{n} \frac{C_i}{n} \qquad C' = \sum_{1}^{n} p_i * C_i$$

处理某些特效：如运动模糊



Composite(cont.)

Composite Multiple Images 多幅图像合成

- Image Filtering (convolution卷积)图像滤波
  - add shifted and scaled versions of an image
- Whole scene antialiasing场景反走样
  - move primitives a little for each render
- Depth of Field深度模糊（深度滤波）
  - move viewer a little for each render keeping one plane unchanged
- Motion effects运动模糊（时间滤波）
  - ……

The University of New Mexico

Outline

- 帧缓存和帧缓存操作Buffer and Buffer Operation

- 片元操作Fragment Operation
  - Stencil Test模板测试
  - Depth Test 深度测试
  - Color Blend 颜色混合

- **其它颜色合成技术Color Blending**
  - Composite 合成技术
    - **Image processing图像处理**
  - OSR离屏渲染技术
    - **Shadow Map 阴影贴图**

The University of New Mexico

Image Processing

- Fragment Shaders and Images
  - Suppose that we send a rectangle (two triangles) to the vertex shader and render it with an n x m texture map
  - Suppose that in addition we use an n x m canvas
    - There is now a one-to-one correspondence between each texel and each fragment，Hence, can regard “fragment operations” as “imaging operations on the texture map” （当每个“纹素”和每个“片元”建立了一对一的关系时，可将“片元操作”看作在”纹理图上的图像操作”）

Image Processing(cont.)

➢Using Multiple Texels

◆Suppose we have a 256 x 256 texture in the texture object “image”

returns the the value of the texture at (x,y): sampler2D(image, vec2(x,y))

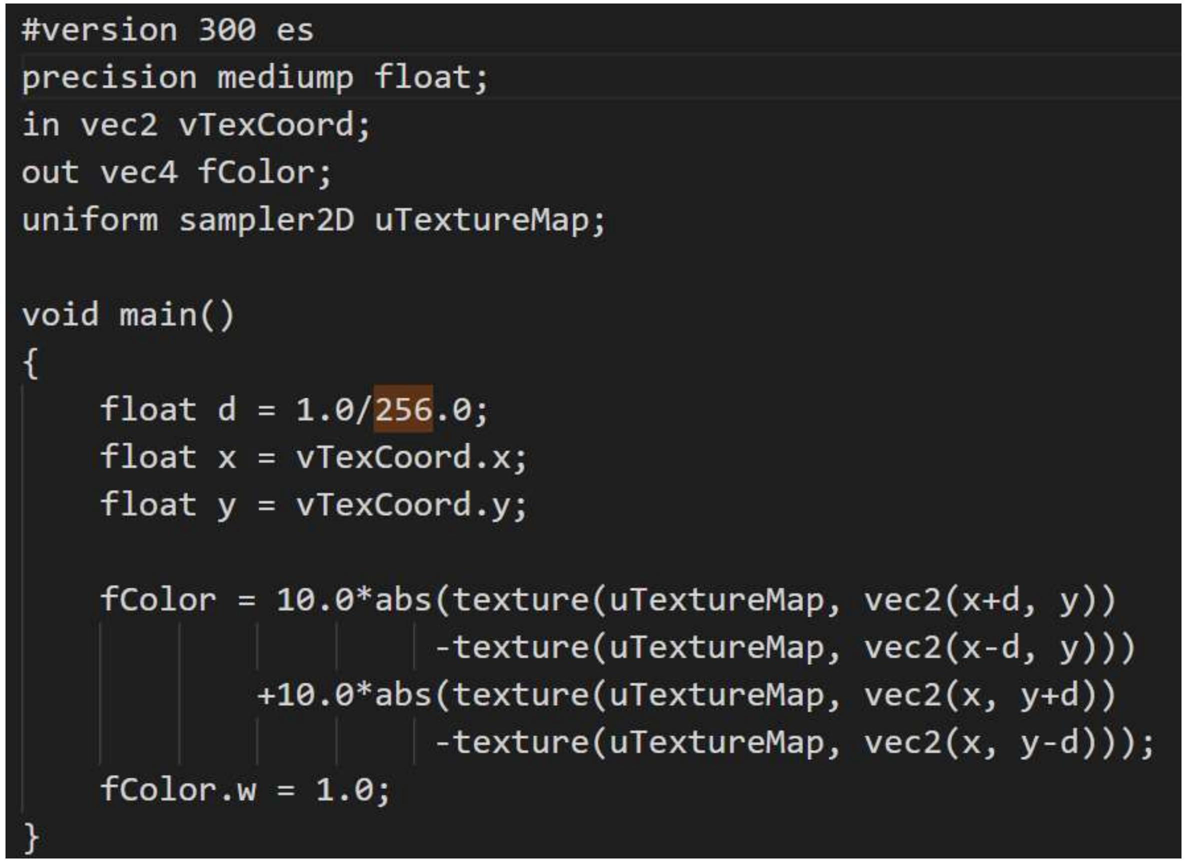
returns the value of the texel to the right of (x,y): sampler2D(image, vec2(x+1.0/256.0), y);

Examples in the following:

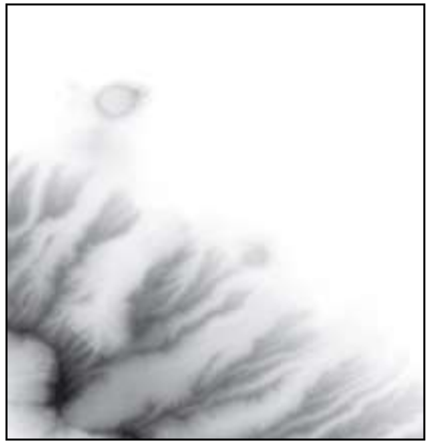
a) **Image Enhancer**
b) **Sobel Edge Detector**
c) **Using Multiple Textures(Matrix addition, subtract…)**

Image Processing(cont.)

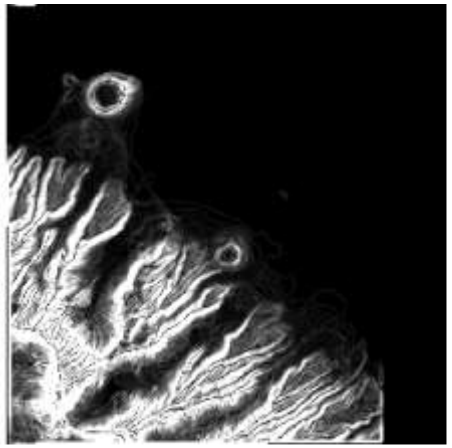
- a. Image Enhancer 图像增强 /2024AngelCode8E/08/hawaiiImage.html



```
#version 300 es
precision mediump float;
in vec2 vTexCoord;
out vec4 fColor;
uniform sampler2D uTextureMap;

void main()
{
    float d = 1.0/256.0;
    float x = vTexCoord.x;
    float y = vTexCoord.y;

    fColor = 10.0*abs(texture(uTextureMap, vec2(x+d, y))
                     -texture(uTextureMap, vec2(x-d, y)))
            +10.0*abs(texture(uTextureMap, vec2(x, y+d))
                     -texture(uTextureMap, vec2(x, y-d)));
    fColor.w = 1.0;
}
```



original

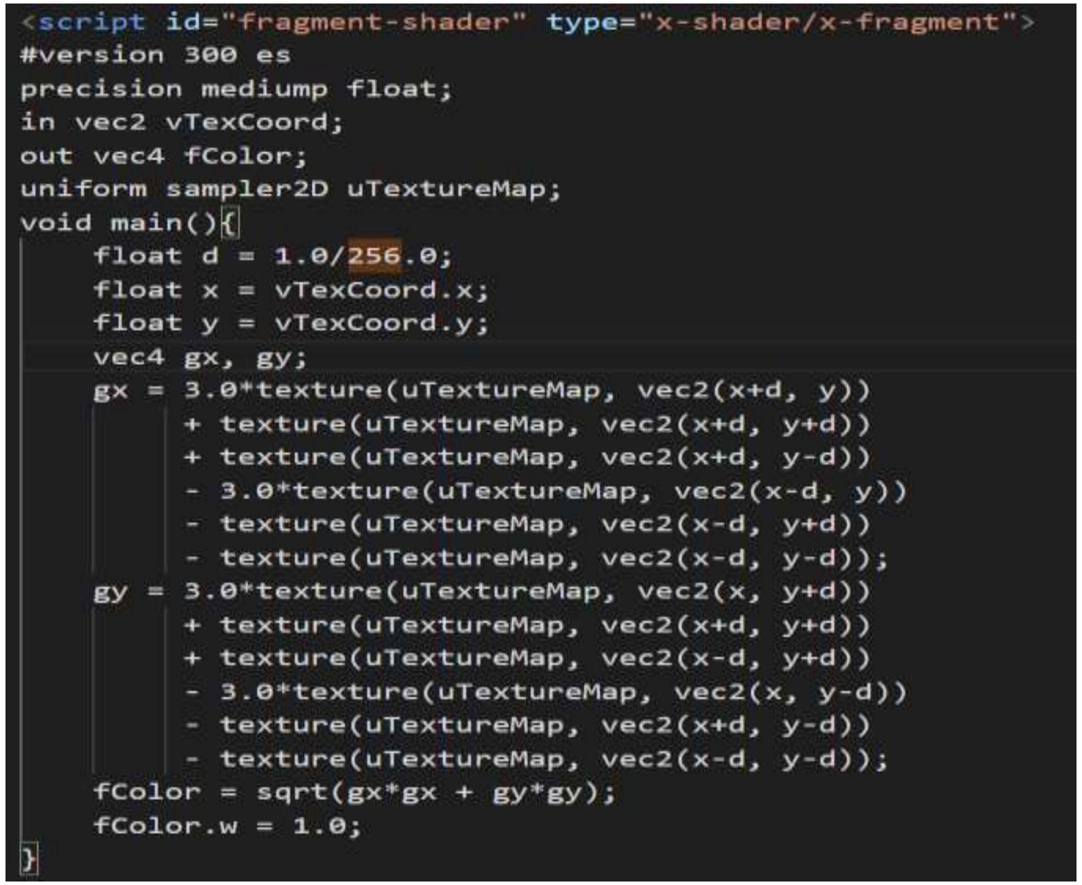


enhanced

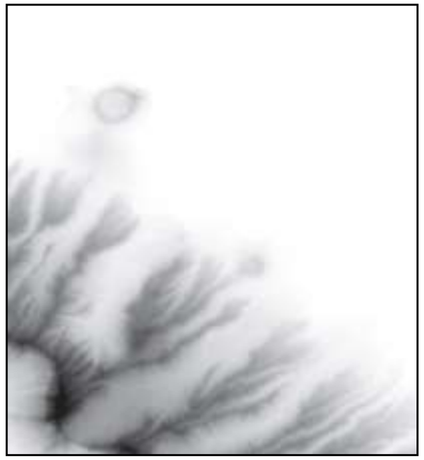
The University of New Mexico

Image Processing(cont.)

- b. Sobel Edge Detector边缘检测
  - Nonlinear, Find approximate gradient at each point
  - Compute smoothed finite difference approximations to x and y components separately
  - Simple with fragment shader, Display magnitude of approximate gradient



```
<script id="fragment-shader" type="x-shader/x-fragment">
#version 300 es
precision mediump float;
in vec2 vTexCoord;
out vec4 fColor;
uniform sampler2D uTextureMap;
void main(){
    float d = 1.0/256.0;
    float x = vTexCoord.x;
    float y = vTexCoord.y;
    vec4 gx, gy;
    gx = 3.0*texture(uTextureMap, vec2(x+d, y))
         + texture(uTextureMap, vec2(x+d, y+d))
         + texture(uTextureMap, vec2(x+d, y-d))
         - 3.0*texture(uTextureMap, vec2(x-d, y))
         - texture(uTextureMap, vec2(x-d, y+d))
         - texture(uTextureMap, vec2(x-d, y-d));
    gy = 3.0*texture(uTextureMap, vec2(x, y+d))
         + texture(uTextureMap, vec2(x+d, y+d))
         + texture(uTextureMap, vec2(x-d, y+d))
         - 3.0*texture(uTextureMap, vec2(x, y-d))
         - texture(uTextureMap, vec2(x+d, y-d))
         - texture(uTextureMap, vec2(x-d, y-d));
    fColor = sqrt(gx*gx + gy*gy);
    fColor.w = 1.0;
}
```


original

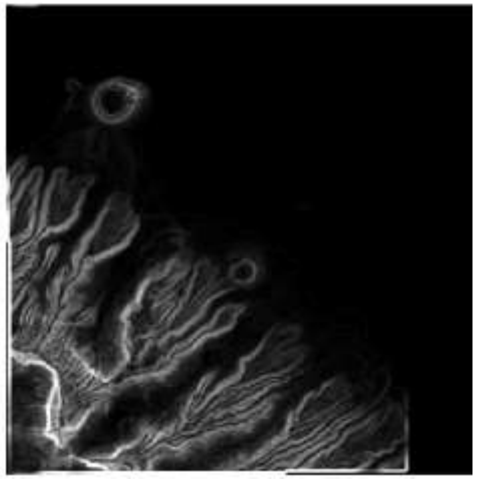

Sobel Edge Detect

Image Processing(cont.)

- c. Using Multiple Textures
  - matrix addition
    - Create two samplers, texture1 and texture2, that contain the data In fragment shader

      gl_FragColor = sampler2D(texture1, vec2(x, y)) +sampler2D(texture2, vec2(x,y));

  - Matrix subtract
    - Create two samplers, texture1 and texture2, that contain the data In fragment shader

      gl_FragColor = sampler2D(texture1, vec2(x, y)) - sampler2D(texture2, vec2(x,y));

Outline

- 帧缓存和帧缓存操作Buffer and Buffer Operation

- 片元操作Fragment Operation
  - Stencil Test模板测试
  - Depth Test 深度测试
  - Color Blend 颜色混合

- **其它颜色合成技术Color Blending**
  - Composite 合成技术
    - **Image processing图像处理**
  - OSR离屏渲染技术
    - **Shadow Map 阴影贴图**

Off-Screen Rendering

- What we have done as large matrix operations rather than graphics operations
- Leads to the field of General Purpose Computing with a GPU (GPGPU)
  - Add two matrices
  - Multiply two matrices
  - Fast Fourier Transform
  - Uses speed and parallelism of GPU
- But how do we get out results?
  - **Floating point frame buffers**
  - **OpenCL (WebCL)**
  - **Compute shaders**
- Need more storage for most GPGPU calculations，Need shared memory
  - ➢Solution: Use texture memory纹理存储 and off-screen rendering离屏渲染

The University of New Mexico

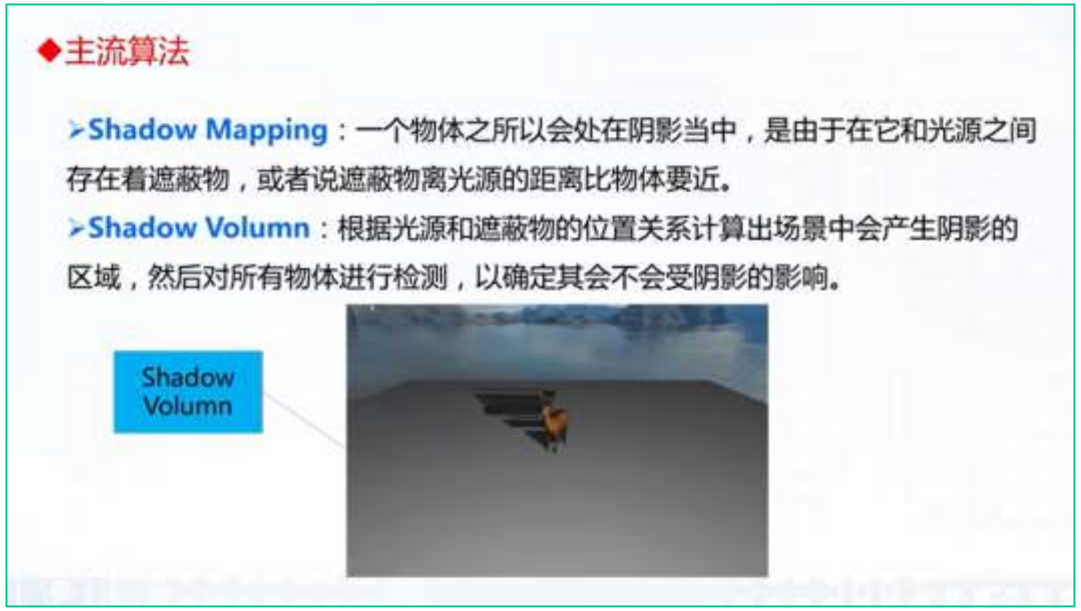
Off-Screen Rendering 离屏渲染技术

用多着色器渲染和离屏渲染技术,实现阴影绘制:

- 1.多着色器渲染Multi-Shaders Rendering
- 2.离屏渲染Off-Screen Rendering
- 3.阴影生成方法Shadow Mapping

1 阴影的概念

生活中的常识：有光就有影



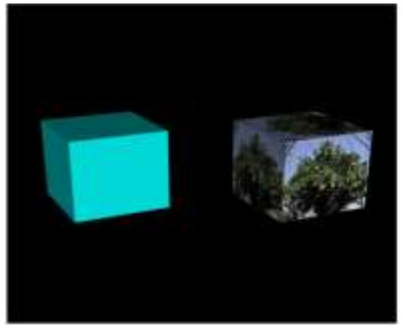
◆主流算法

➢Shadow Mapping：一个物体之所以会处在阴影当中，是由于在它和光源之间存在着遮蔽物，或者说遮蔽物离光源的距离比物体要近。

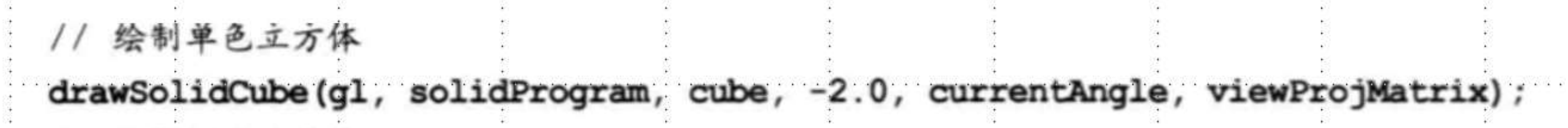
➢Shadow Volumn：根据光源和遮蔽物的位置关系计算出场景中会产生阴影的区域，然后对所有物体进行检测，以确定其会不会受阴影的影响。

The University of New Mexico

1. Multi-Shaders Rendering

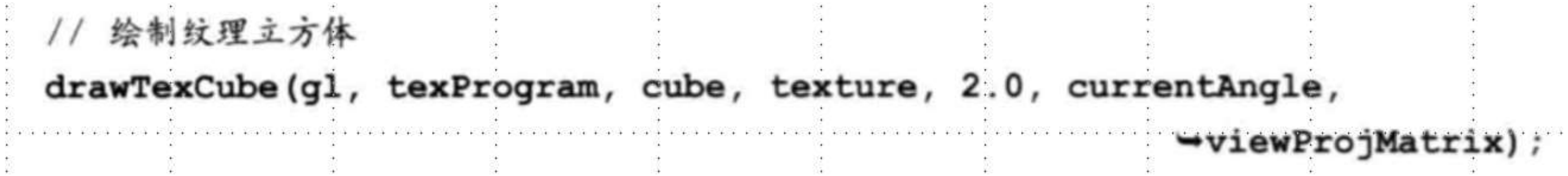


- 用不同的着色器绘制不同的物体
  - solidProgram里的vertex shader和fragment shader绘制solidCube



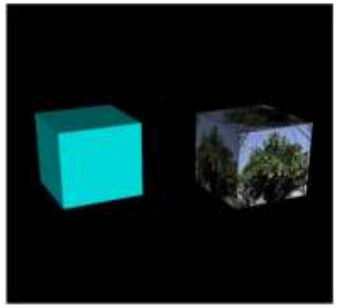
```
// 绘制单色立方体
drawSolidCube(gl, solidProgram, cube, -2.0, currentAngle, viewProjMatrix);
```

  - texProgram里的vertex shader和fragment shader绘制TexCube

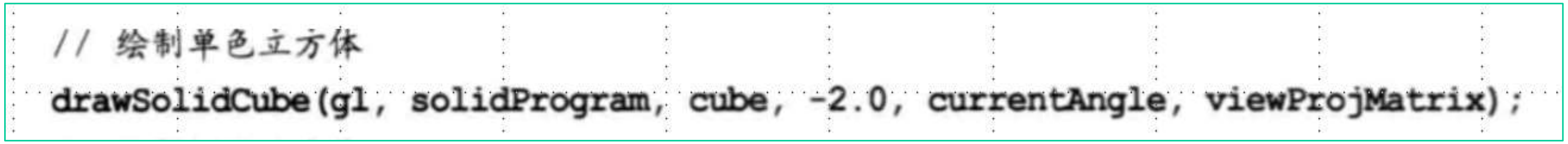


```
// 绘制纹理立方体
drawTexCube(gl, texProgram, cube, texture, 2.0, currentAngle,
                                                    ➥viewProjMatrix);
```

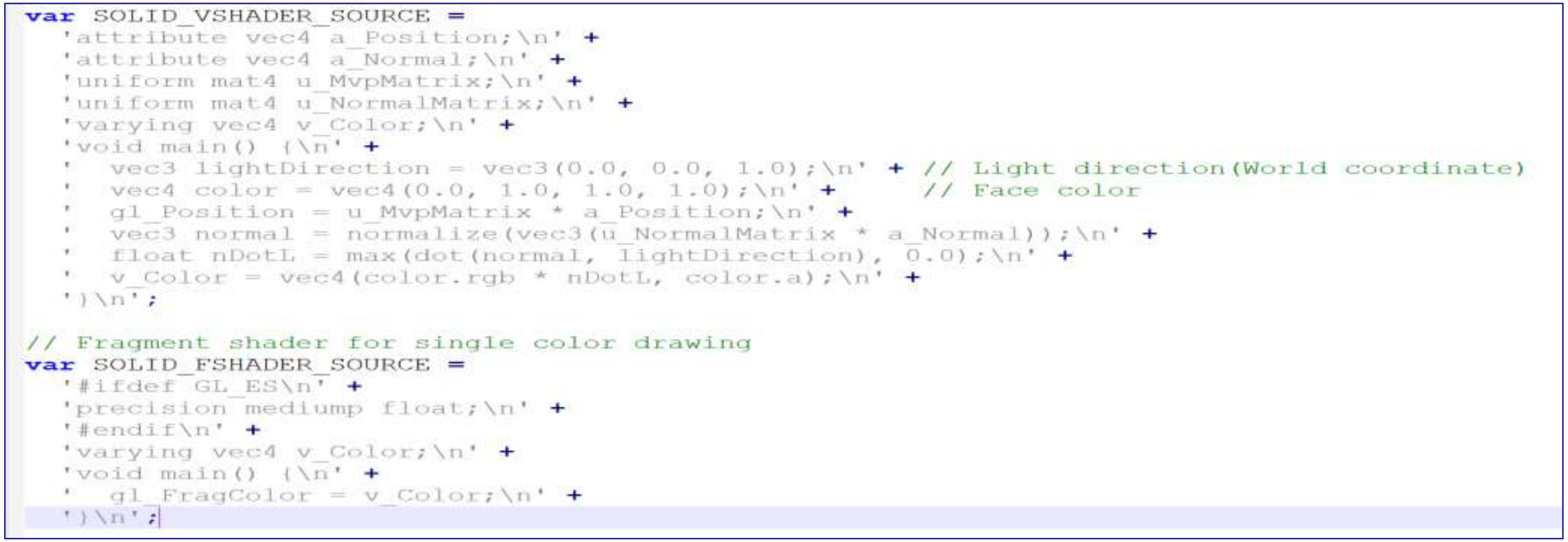
1. Multi-Shaders Rendering(cont.)



- drawsolidCube绘制左边立方体:



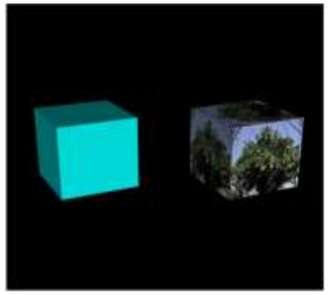
```
// 绘制单色立方体
drawSolidCube(gl, solidProgram, cube, -2.0, currentAngle, viewProjMatrix);
```



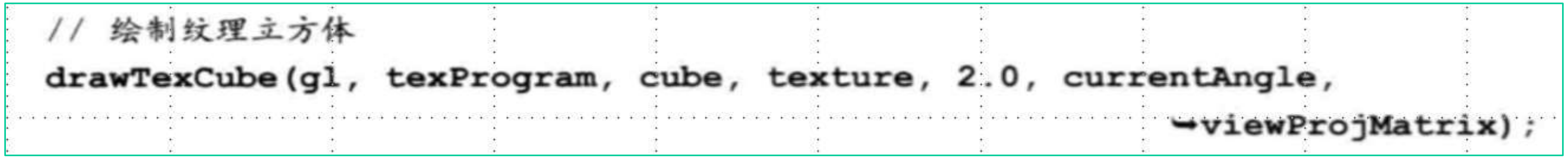
```
var SOLID_VSHADER_SOURCE =
    'attribute vec4 a_Position;\n' +
    'attribute vec4 a_Normal;\n' +
    'uniform mat4 u_MvpMatrix;\n' +
    'uniform mat4 u_NormalMatrix;\n' +
    'varying vec4 v_Color;\n' +
    'void main() {\n' +
    '  vec3 lightDirection = vec3(0.0, 0.0, 1.0);\n' + // Light direction(World coordinate)
    '  vec4 color = vec4(0.0, 1.0, 1.0, 1.0);\n' +      // Face color
    '  gl_Position = u_MvpMatrix * a_Position;\n' +
    '  vec3 normal = normalize(vec3(u_NormalMatrix * a_Normal));\n' +
    '  float nDotL = max(dot(normal, lightDirection), 0.0);\n' +
    '  v_Color = vec4(color.rgb * nDotL, color.a);\n' +
    '}\n';

// Fragment shader for single color drawing
var SOLID_FSHADER_SOURCE =
    '#ifdef GL_ES\n' +
    'precision mediump float;\n' +
    '#endif\n' +
    'varying vec4 v_Color;\n' +
    'void main() {\n' +
    '  gl_FragColor = v_Color;\n' +
    '}\n';
```

The University of New Mexico

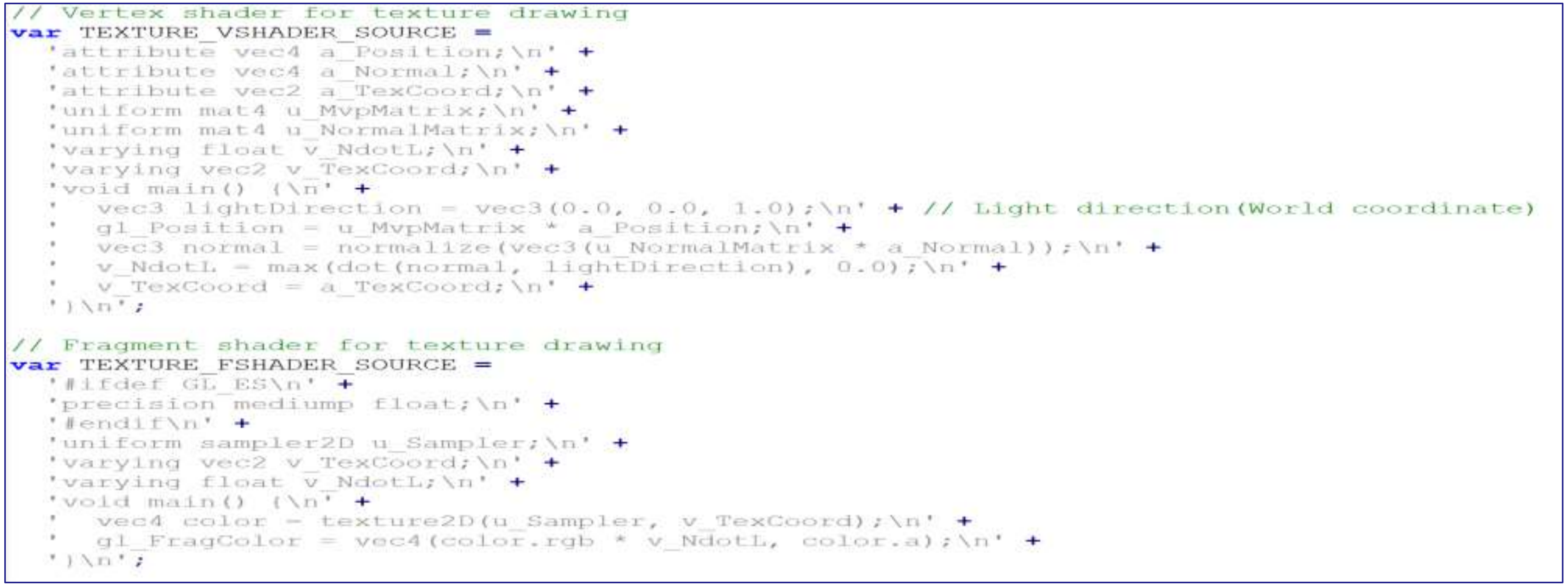
1. Multi-Shaders Rendering(cont.)



- drawTexCube绘制右边有纹理的立方体



```
// 绘制纹理立方体
drawTexCube(gl, texProgram, cube, texture, 2.0, currentAngle,
                                                          ➥viewProjMatrix);
```



```
// Vertex shader for texture drawing
var TEXTURE_VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'attribute vec4 a_Normal;\n' +
  'attribute vec2 a_TexCoord;\n' +
  'uniform mat4 u_MvpMatrix;\n' +
  'uniform mat4 u_NormalMatrix;\n' +
  'varying float v_NdotL;\n' +
  'varying vec2 v_TexCoord;\n' +
  'void main() {\n' +
  '  vec3 lightDirection = vec3(0.0, 0.0, 1.0);\n' + // Light direction(World coordinate)
  '  gl_Position = u_MvpMatrix * a_Position;\n' +
  '  vec3 normal = normalize(vec3(u_NormalMatrix * a_Normal));\n' +
  '  v_NdotL = max(dot(normal, lightDirection), 0.0);\n' +
  '  v_TexCoord = a_TexCoord;\n' +
  '}\n';

// Fragment shader for texture drawing
var TEXTURE_FSHADER_SOURCE =
  '#ifdef GL_ES\n' +
  'precision mediump float;\n' +
  '#endif\n' +
  'uniform sampler2D u_Sampler;\n' +
  'varying vec2 v_TexCoord;\n' +
  'varying float v_NdotL;\n' +
  'void main() {\n' +
  '  vec4 color = texture2D(u_Sampler, v_TexCoord);\n' +
  '  gl_FragColor = vec4(color.rgb * v_NdotL, color.a);\n' +
  '}\n';
```

The University of New Mexico

2.Off Screen Rendering离屏渲染

- OSR(Off Screen Rendering)
  - 把场景渲染保存到帧缓存对象FBO(Frame Buffer Object)且不显示输出。
  - 即渲染结果（一帧图像）不会渲染到默认的颜色帧缓存中进行屏幕显示。
  - 常用于实现：动态模糊，景深，阴影等效果

The University of New Mexico

2.Off Screen Rendering (cont.)

➢OSR(Off Screen Rendering)（cont.)

帧缓冲区对象FBO：自定义的帧缓冲

- 类似于显示用的帧缓存区对象，但FBO绘制的内容不直接显示在屏幕画布上。

- FBO包含三个“关联对象attachment object”

  - 颜色关联对象（color attachment）
  - 深度关联对象(depth attachment)
  - 模板关联对象(stencil attachment)

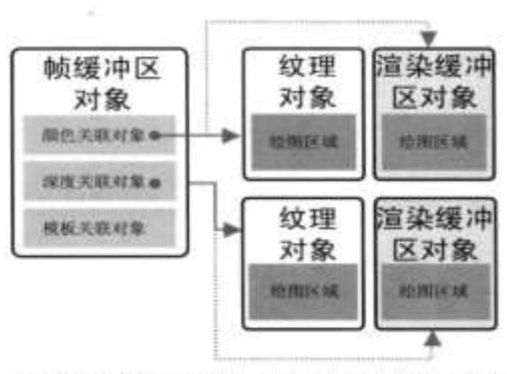


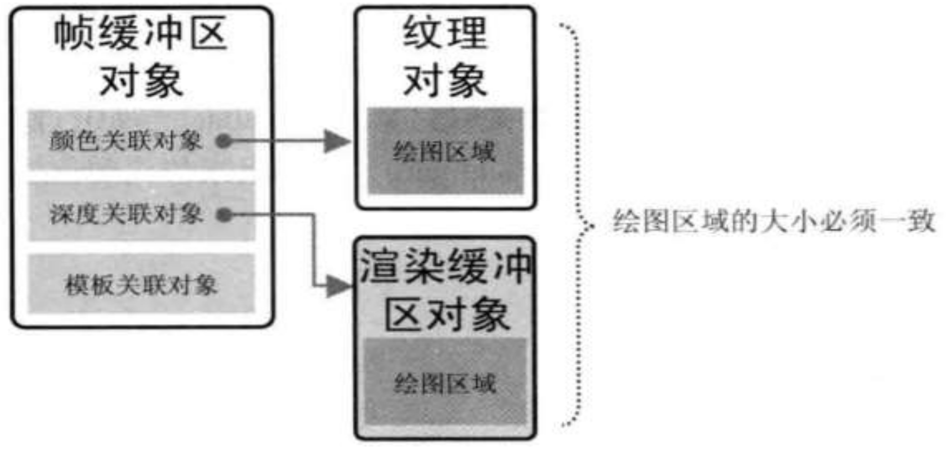
图 10.19 帧缓冲区对象、纹理对象和渲染缓冲区对象

  - 关联对象的类型：纹理对象TO或渲染缓冲区对象RBO
    - 纹理对象TO(texture object)
    - 渲染缓冲区对象RBO（render buffer object）
      - 比纹理对象更加通用的绘图区域，可以向其中写入多种类型的数据

The University of New Mexico

2.Off Screen Rendering (cont.)

- OSR技术操作步骤:
  - 首先，创建帧缓冲区对象FBO
  - 然后，将“颜色光联对象，深度关联对象，模板关联对象”等 关联 给 FBO
    - 如下图，颜色关联对象(用即纹理对象TO)替代了系统默认显示颜色缓冲区。
    - 如下图，深度关联对象(用渲染缓冲区对象RBO)替代了系统默认显示深度缓冲区。
  - 最后，在FBO中进行离屏渲染。
    - 常将纹理对象作为颜色缓存存放一帧的颜色内容。这种离屏幕渲染也称为: “渲染到纹理Render to Texture”，其中纹理对象中的纹理称为“动态纹理”。



The University of New Mexico

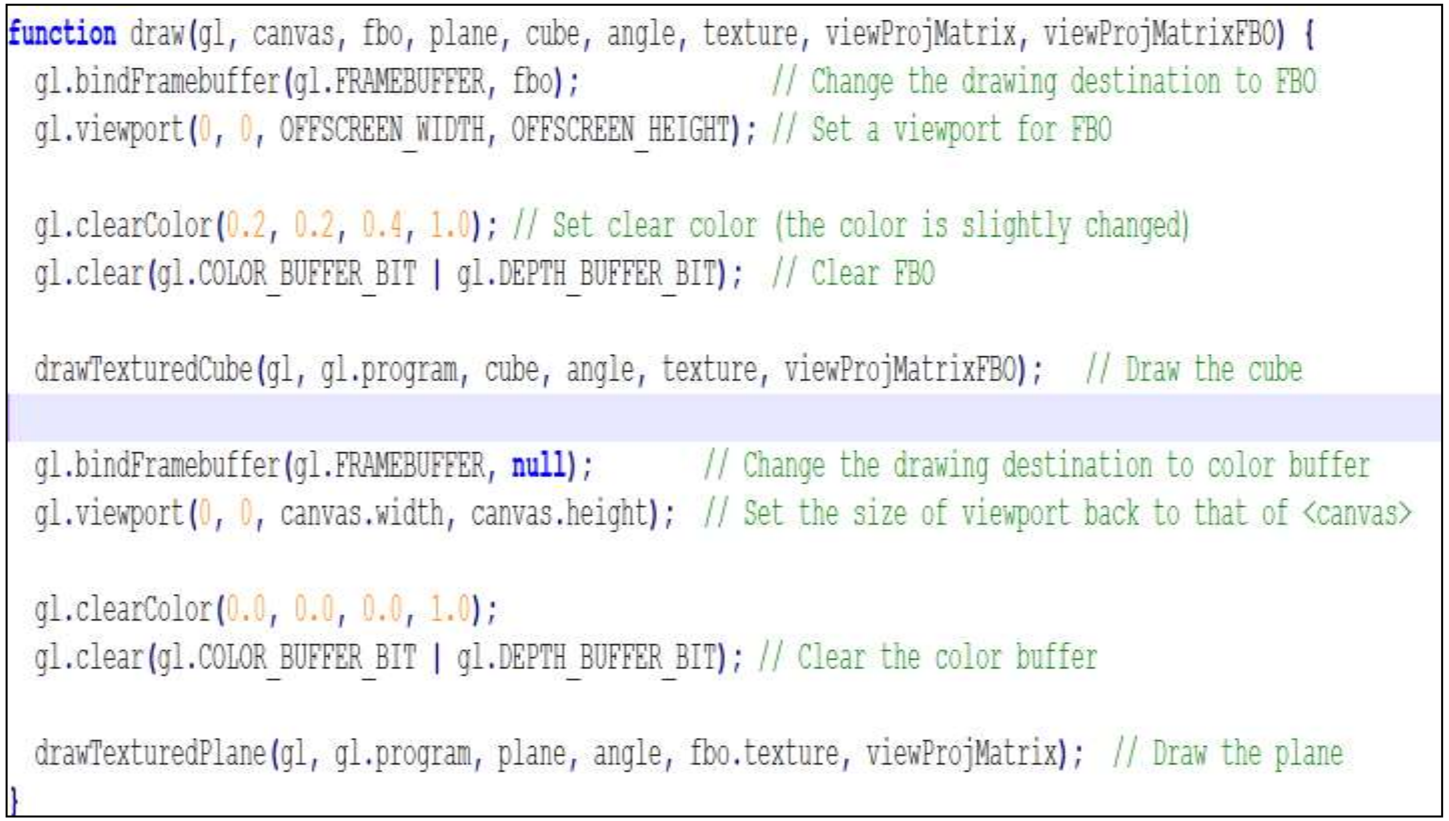
2.Off Screen Rendering (cont.)

Example： FramebufferObject.js， 参见右上图运行结果和代码

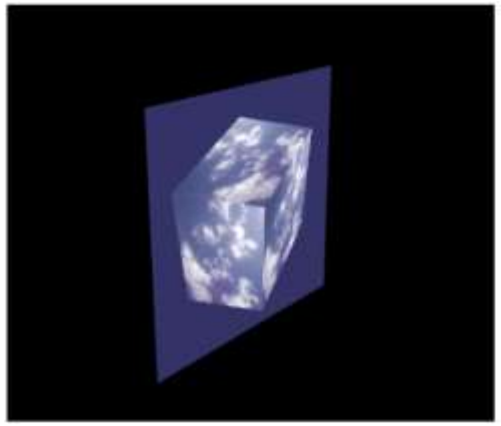
1.第一次渲染drawTextureCube是渲染到纹理（未显示）

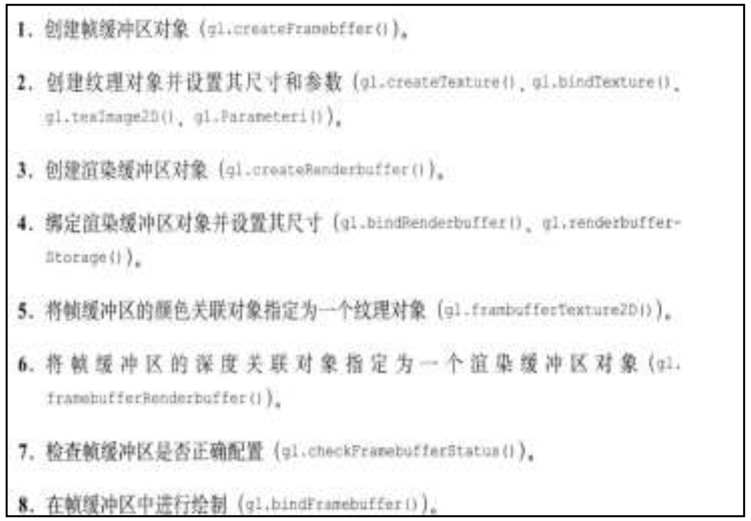
// 函数 function initFramebufferObject(gl)：包含1~7 步骤，参见右下图

2.第二次渲染drawTexturePlane是渲染到屏幕（进行显示输出）



```
function draw(gl, canvas, fbo, plane, cube, angle, texture, viewProjMatrix, viewProjMatrixFBO) {
  gl.bindFramebuffer(gl.FRAMEBUFFER, fbo);              // Change the drawing destination to FBO
  gl.viewport(0, 0, OFFSCREEN_WIDTH, OFFSCREEN_HEIGHT); // Set a viewport for FBO

  gl.clearColor(0.2, 0.2, 0.4, 1.0); // Set clear color (the color is slightly changed)
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);  // Clear FBO

  drawTexturedCube(gl, gl.program, cube, angle, texture, viewProjMatrixFBO);   // Draw the cube

  gl.bindFramebuffer(gl.FRAMEBUFFER, null);        // Change the drawing destination to color buffer
  gl.viewport(0, 0, canvas.width, canvas.height);  // Set the size of viewport back to that of <canvas>

  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT); // Clear the color buffer

  drawTexturedPlane(gl, gl.program, plane, angle, fbo.texture, viewProjMatrix);  // Draw the plane
}
```





1. 创建帧缓冲区对象（gl.createFramebffer()）。
2. 创建纹理对象并设置其尺寸和参数（gl.createTexture()、gl.bindTexture()、gl.texImage2D()、gl.Parameteri()）。
3. 创建渲染缓冲区对象（gl.createRenderbuffer()）。
4. 绑定渲染缓冲区对象并设置其尺寸（gl.bindRenderbuffer()、gl.renderbuffer-Storage()）。
5. 将帧缓冲区的颜色关联对象指定为一个纹理对象（gl.frambufferTexture2D()）。
6. 将帧缓冲区的深度关联对象指定为一个渲染缓冲区对象（gl.framebufferRenderbuffer()）。
7. 检查帧缓冲区是否正确配置（gl.checkFramebufferStatus()）。
8. 在帧缓冲区中进行绘制（gl.bindFramebuffer()）。

The University of New Mexico

3.Shadow Mapping(阴影映射/贴图)

- ➢Why Shadow
  - ➢局部光照模型并不会直接产生阴影，需要采用阴影生成技术来模拟。
- ➢What is Shadow
  - ➢一个物体表面之所以会处在阴影当中，是由于在它和光源之间存在着遮蔽物 (或者说遮蔽物离光源的距离比物体要近).
- ➢How to Shadow
  - 光栅化渲染常用阴影生成法：
    - Shadow Mapping(阴影映射) /深度贴图（depth map）
    - Shadow Volume（阴影体）

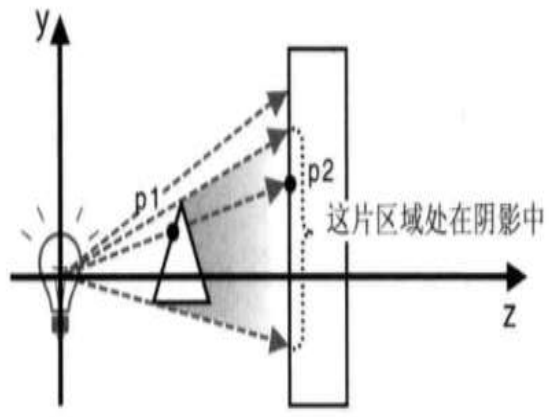
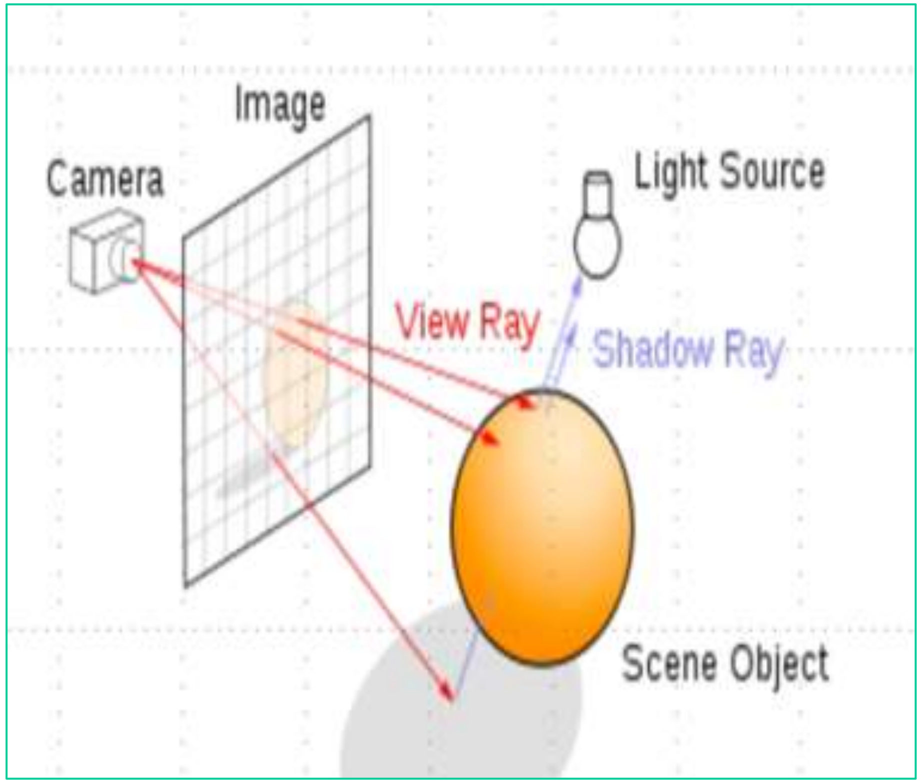


图 10.21 阴影贴图的原理

The University of New Mexico

3.Shadow Mapping（cont.)

- Shadow Mapping(阴影映射/阴影贴图)
  - 视点“可见的”，光源“可见的”的表面，则不在阴影中；
  - 视点“可见的”，光源“**不可见的**”的表面，则落在阴影中。



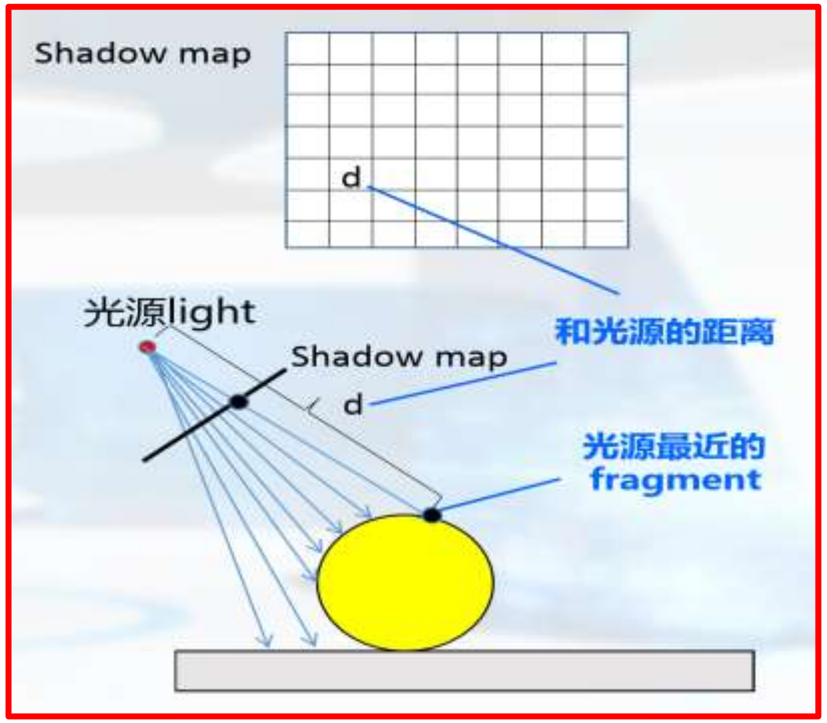
The University of New Mexico

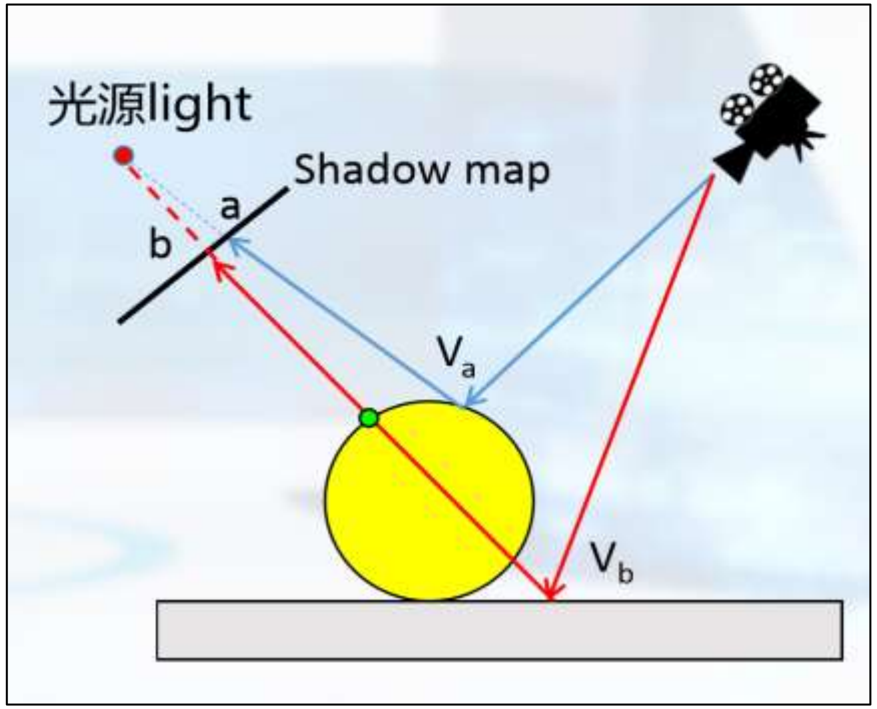
3.Shadow Mapping（cont.)

➢Shadow Mapping(阴影映射/阴影贴图)（cont.)

Step 1离屏渲染:以光源作为观察点，对场景进行第一次渲染（离屏渲染），建立shadow map

先把场景中所有的光照计算关掉，然后以“点光源”作为视点，在光源坐标系下，对整个场景进行第一次渲染，得“深度图”shadow map”（即光源可见面的像素深度d）。





3.Shadow Mapping（cont.)

➢Shadow Mapping(阴影映射/阴影贴图)（cont.)

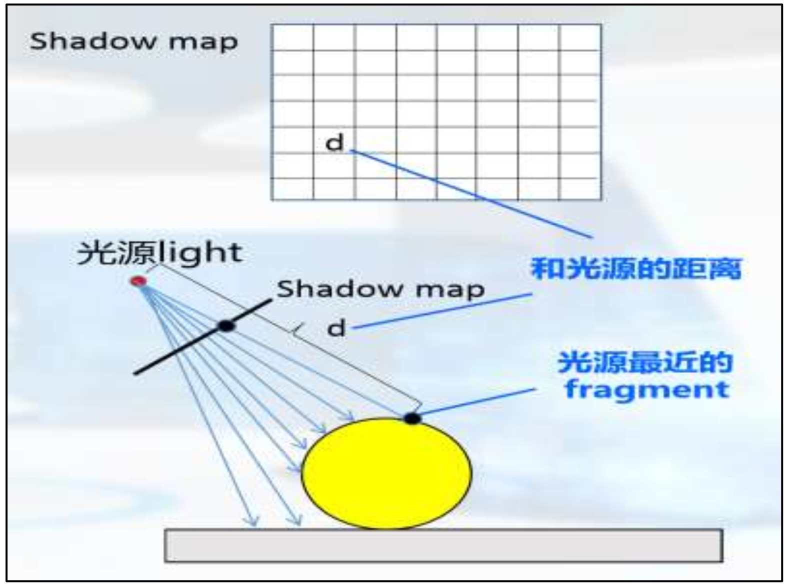
Step 1:以光源作为观察点，对场景进行第一次渲染（离屏渲染），建立shadow map

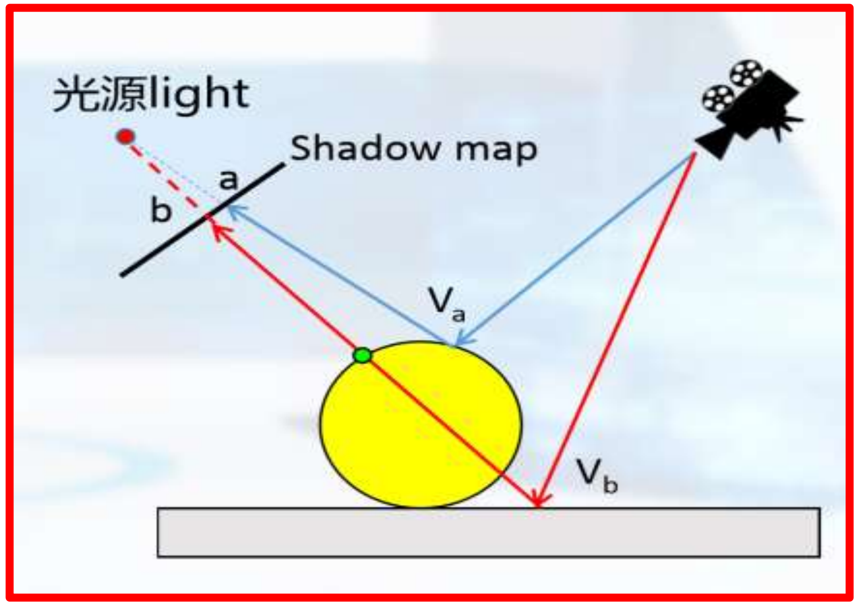
先把场景中所有的光照计算关掉，然后以“点光源”作为视点，在光源坐标系下，对整个场景进行第一次渲染，得“深度图”shadow map”（即光源可见面的像素深度）。

Step2屏幕渲染：以视点为观察点，对场景进行第二次渲染（屏幕渲染），且比较shadowmap中深度进行着色。

以“正常观察点”作为视点，在观察坐标下，对整个场景进行第二次渲染，即对每个“视点可见像素”计算它和光源的距离z，和第一次渲染得到的shadow map中该像素的可见深度值d，进行比较：

- 如果d= z，说明这个像素是“光源可见”并且“相机可见”，所以是非阴影，如Va.
- 如果d< z，说明这个像素是“光源不可见”但是“相机可见”，所以是阴影，如Vb.





The University of New Mexico

3.Shadow Map(cont.)

➢Shadow Mapping(阴影映射/阴影贴图) (cont.)

- 实例shadowmap ref /angelcode8E/08/shadowmap.*

✓第一遍渲染: 离屏渲染(渲染到纹理):以"点光源"作为观察点渲染到纹理

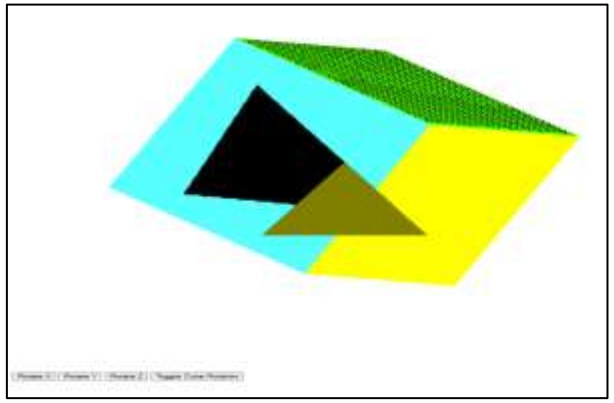
- 对光源可见的像素, 将其深度值Z值写入到FBO的"纹理缓存TO"中保存起来.
- 在片元着色器中: gl_FragColor = vec4(gl_FragCoord.z, 0.0, 0.0, 0.0); //可只用r保存深度

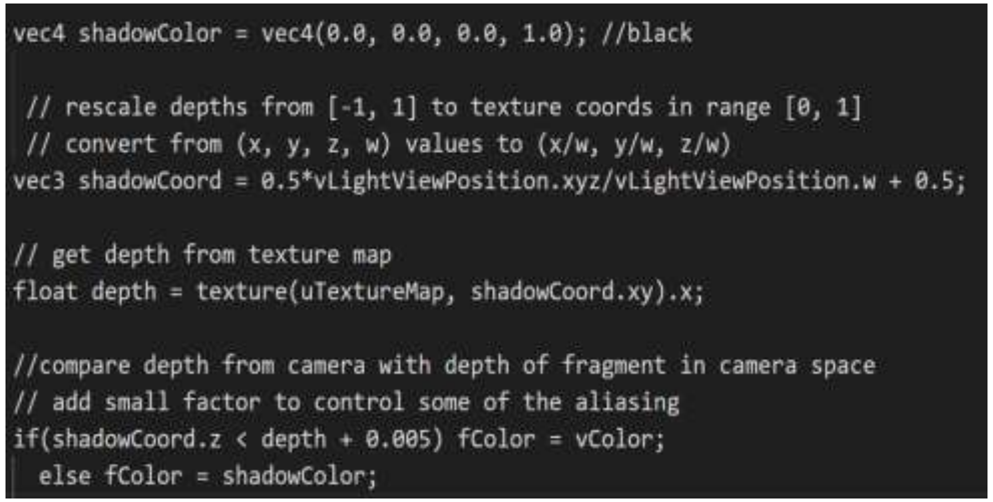
✓第二遍渲染: 屏幕渲染(渲染到屏幕):以"相机"作为观察点进行渲染

✓对每可见片元, 从纹理对象中取相应的"光源可见最小深度d"(即第一遍渲染中保存在r中的值)。

✓对每可见片元, 计算从光源观察时的裁剪坐标, 并且从[-1,1]规范化到[0,1]范围, 且除以W得到笛卡尔坐标, 得到其"光源观察时的深度值z",

✓若z<d, 则该片元是"光源不可见的", 应属于阴影。





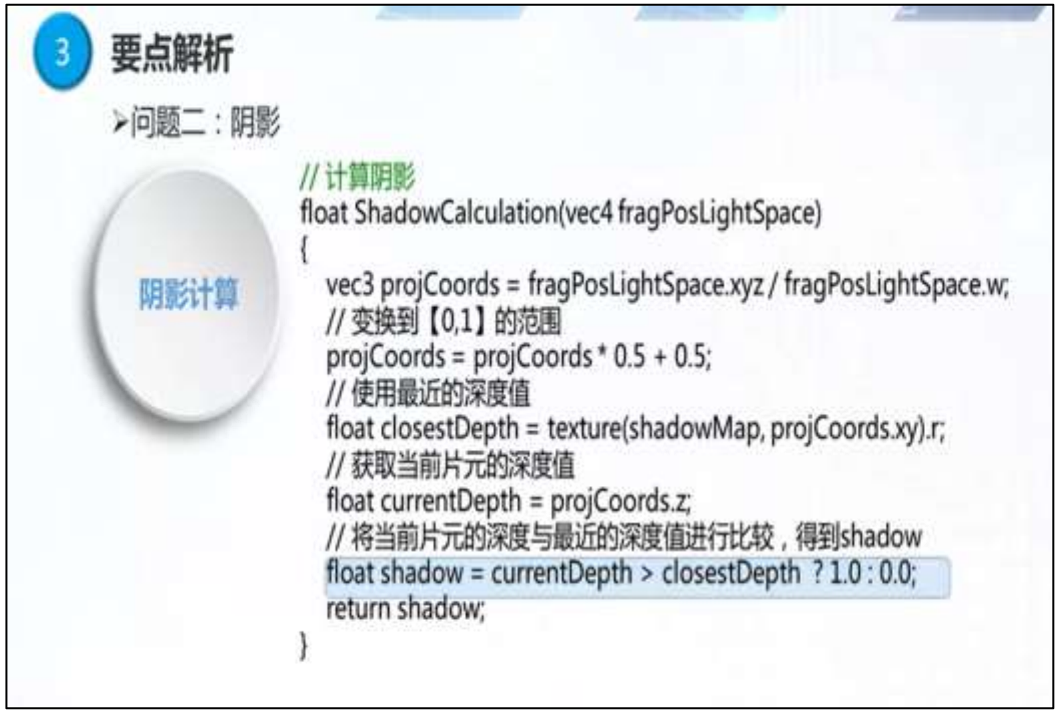
```
vec4 shadowColor = vec4(0.0, 0.0, 0.0, 1.0); //black

 // rescale depths from [-1, 1] to texture coords in range [0, 1]
 // convert from (x, y, z, w) values to (x/w, y/w, z/w)
vec3 shadowCoord = 0.5*vLightViewPosition.xyz/vLightViewPosition.w + 0.5;

// get depth from texture map
float depth = texture(uTextureMap, shadowCoord.xy).x;

//compare depth from camera with depth of fragment in camera space
// add small factor to control some of the aliasing
if(shadowCoord.z < depth + 0.005) fColor = vColor;
  else fColor = shadowColor;
```

The University of New Mexico

3.Shadow Mapping（cont.)

➤Shadow Mapping(阴影映射/阴影贴图)（cont.)

- 实例（openGL)//参万琳“中国大学MOOC”讲解“实时动态阴影”
- https://github.com/wanlin405/Computer-Graphics/tree/master/

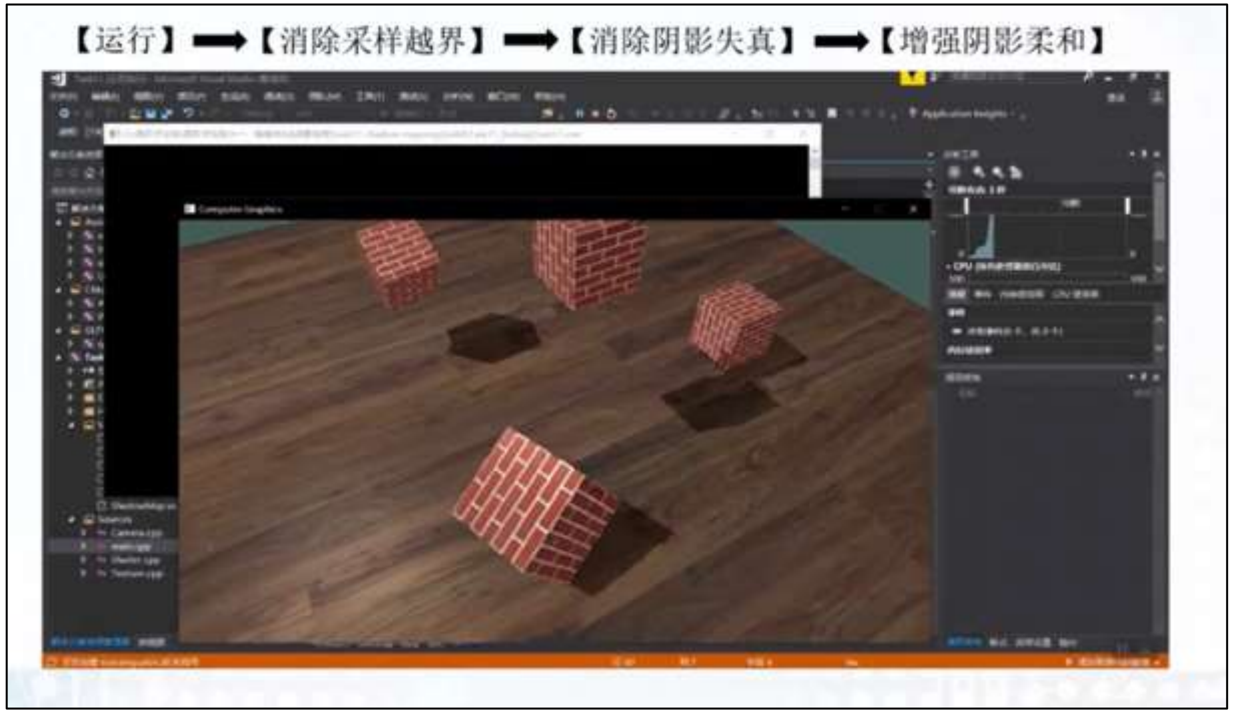


3 要点解析

➤问题二：阴影

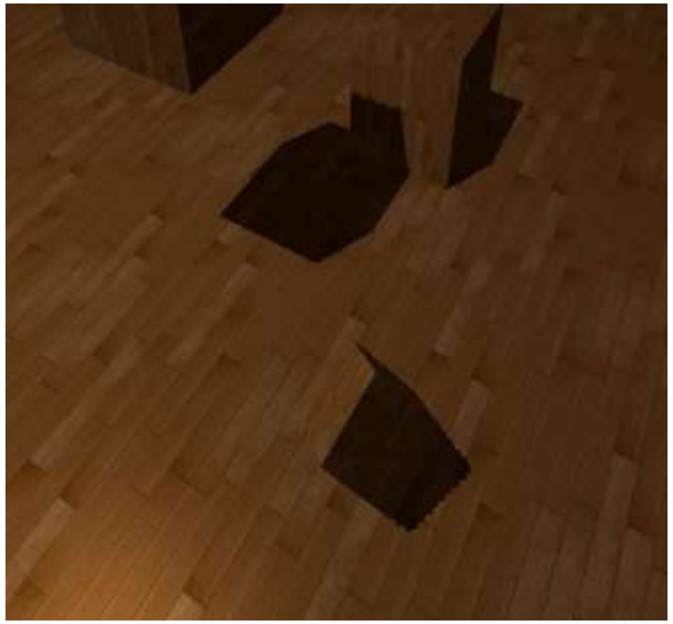
阴影计算

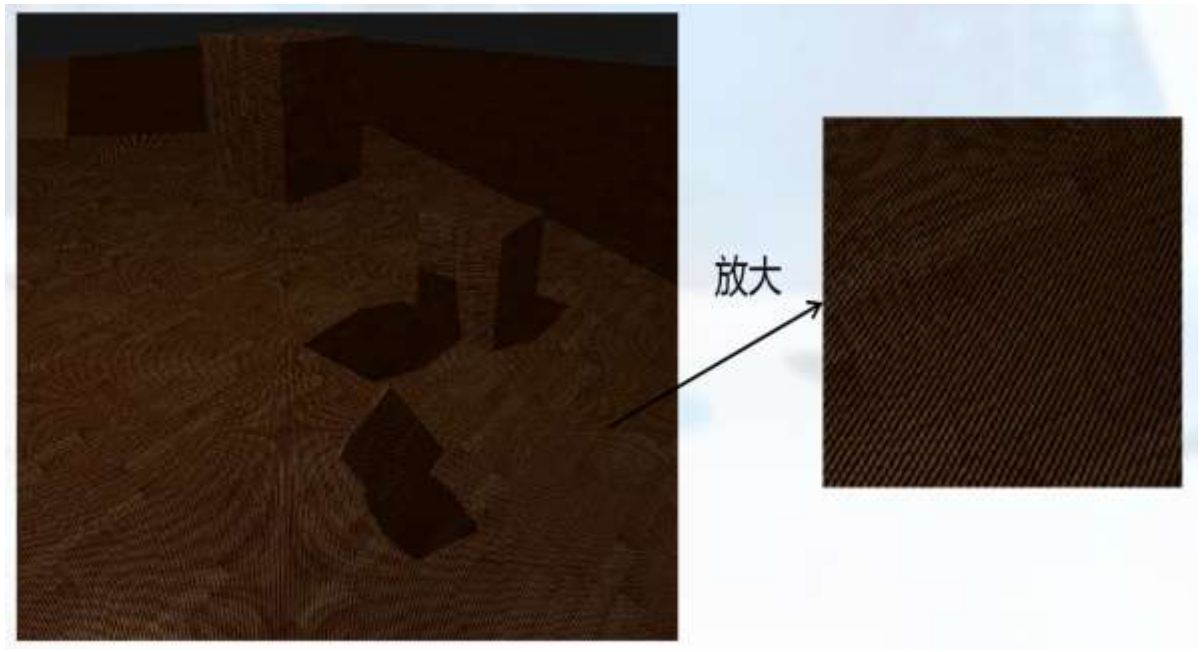
```
// 计算阴影
float ShadowCalculation(vec4 fragPosLightSpace)
{
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // 变换到【0,1】的范围
    projCoords = projCoords * 0.5 + 0.5;
    // 使用最近的深度值
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // 获取当前片元的深度值
    float currentDepth = projCoords.z;
    // 将当前片元的深度与最近的深度值进行比较，得到shadow
    float shadow = currentDepth > closestDepth ? 1.0 : 0.0;
    return shadow;
}
```



3.Shadow Mapping（cont.)

- Shadow Mapping (阴影映射/阴影贴图) 存在问题:
  - 硬阴影:发现无论遮挡物有多远，阴影的边界依然很明显
  - 阴影失真:距离光源比较远时，多个片元可能从深度贴图(shadow map)的同一个深度里取值。下图中，左边是希望效果，右边是失真效果(出现交替黑线)。





Summary

- **Buffer帧缓存**: 有一定分辨率m x n，每个像素按位面存储的存储区域
- **Buffer Operation**：像素位块操作 Cd’=f(Cs,Cd)
- **Fragment Process片元处理**: 模板检测，深度检测，**颜色混合**
  - **Blend混合**：Cd’=s*Cs + d*Cd
    - Translucence透明效果（ s取Cs或Cd的A值(不透明度）,且d=1-s）
    - Fog雾化效果（源混合因子s是和视点物体距离d相关的函数f(d)）
  - **Composite合成**：C’ =∑PiCi //多个像素颜色进行比例调和
    - Image Processing图像处理（采用纹理对象存储数据）
  - **Shadow Map 阴影**：离屏渲染技术OSR
    - 第一次渲染：采用OSR技术渲染到纹理。即渲染到FBO的TO，得到”光源可见最小深度d”
    - 第二次渲染：渲染到帧缓存。对每可见片元，从纹理对象TO中取相应的“光源可见最小深度d”，若小于“本次渲染的光源距离”，则该片元是阴影。

The University of New Mexico

Summary（cont.)

- **OSR（Off Screen Rendering)**
  - GPUs now include a large amount of “**texture memory**”（纹理存储）and use “**texture functions**”（纹理函数）can implement desired functionality in **fragment shaders**

- **OSR Advantages：**
  - Fast (not under control of window system)
  - Simple：Using frame buffer objects (FBOs帧缓冲对象)，can render into texture memory instead of the frame buffer（ (渲染到纹理而不是帧缓存）and then read from this memory.（读写都在GPU存储中，不在CPU中）