XIAMEN UNIVERSITY

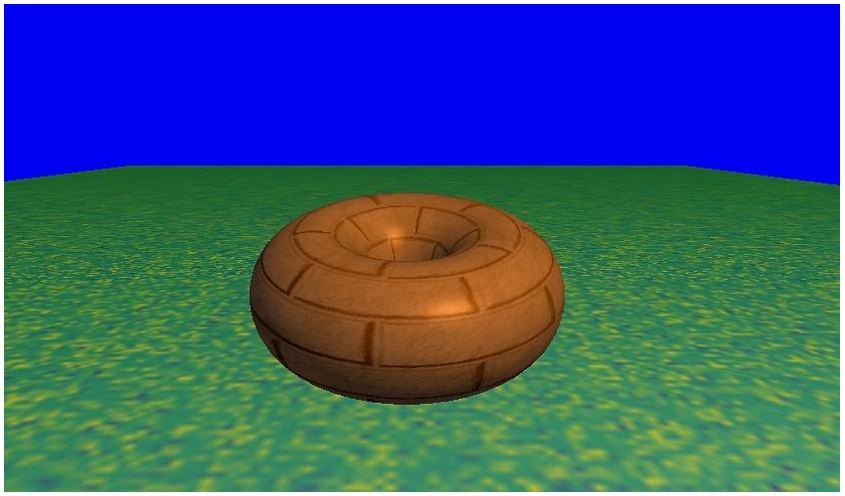
COMPUTER GRAPHICS

第八章 阴影

陈中贵
厦门大学信息学院
http://graphics.xmu.edu.cn

阴影的重要性

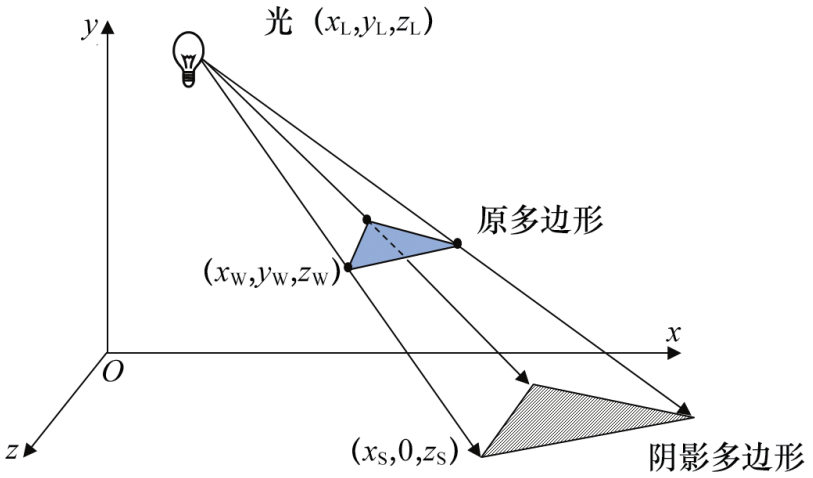
???

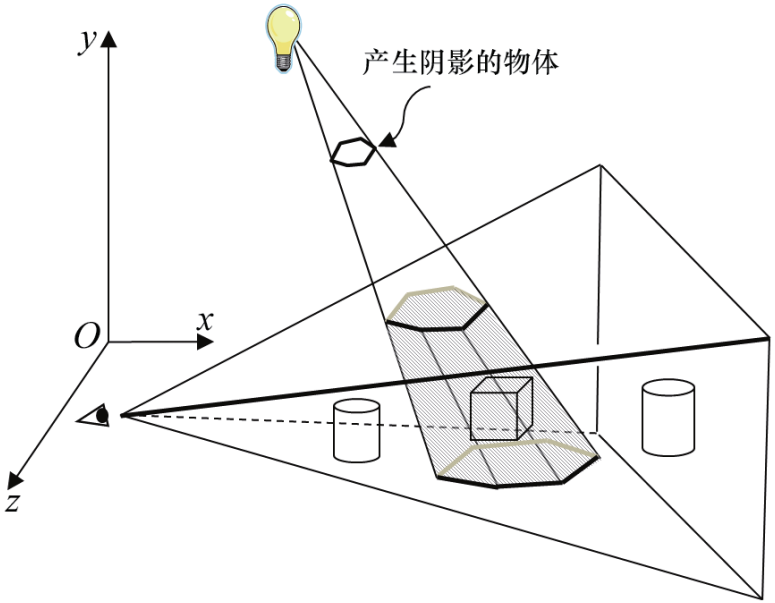


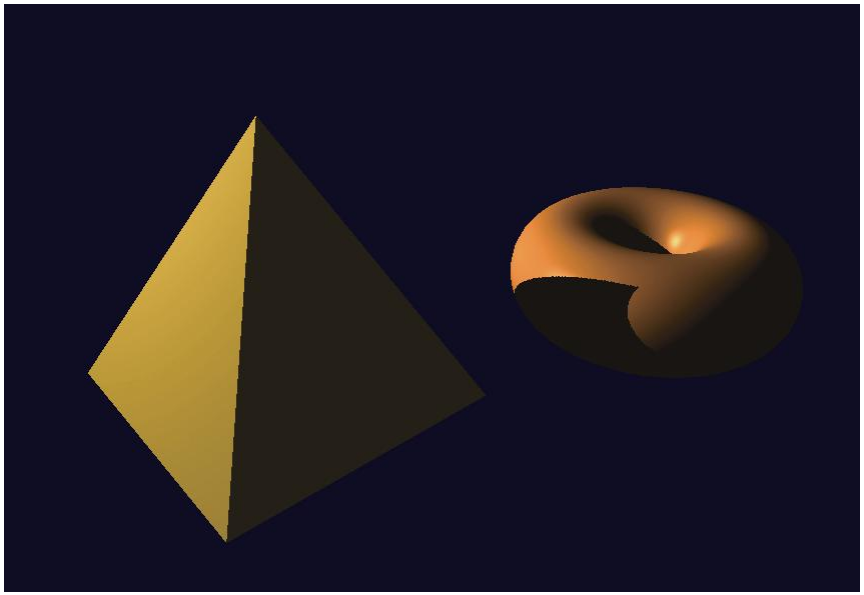
???

阴影计算方法

- 投影阴影
- 阴影体
- 阴影贴图



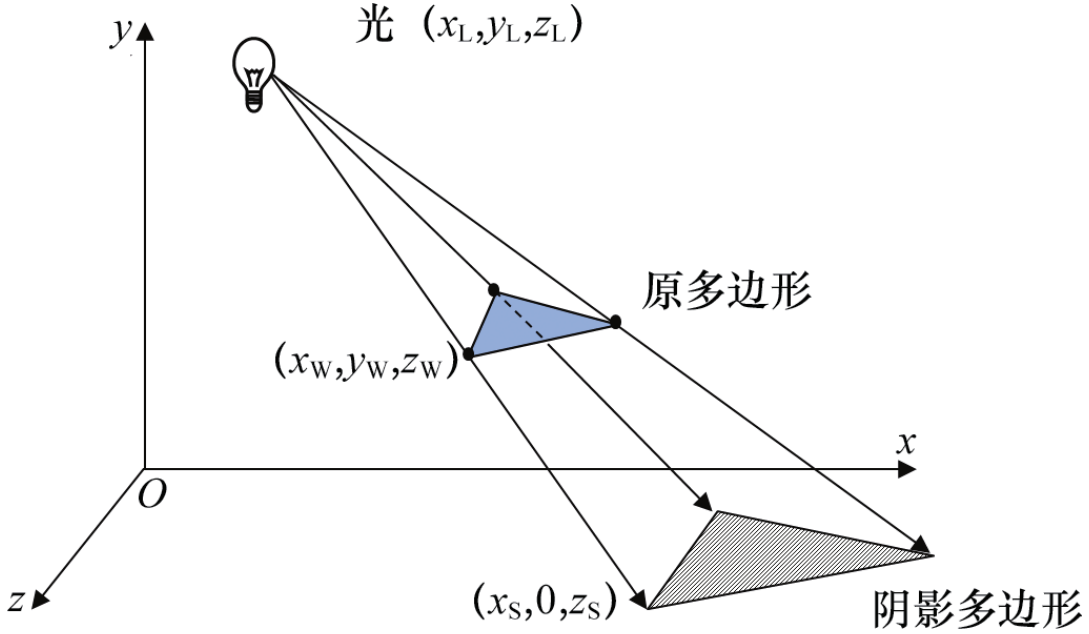




投影阴影

- 给定一个位于(x_L, y_L, z_L)的点光源、一个需要渲染的物体，以及一个投射阴影的平面
- 将物体上的点(x_W, y_W, z_W)变换为相应阴影在平面上的点$(x_S, 0, z_S)$，生成阴影多边形
- 使用暗色物体与地平面纹理混合作为其纹理

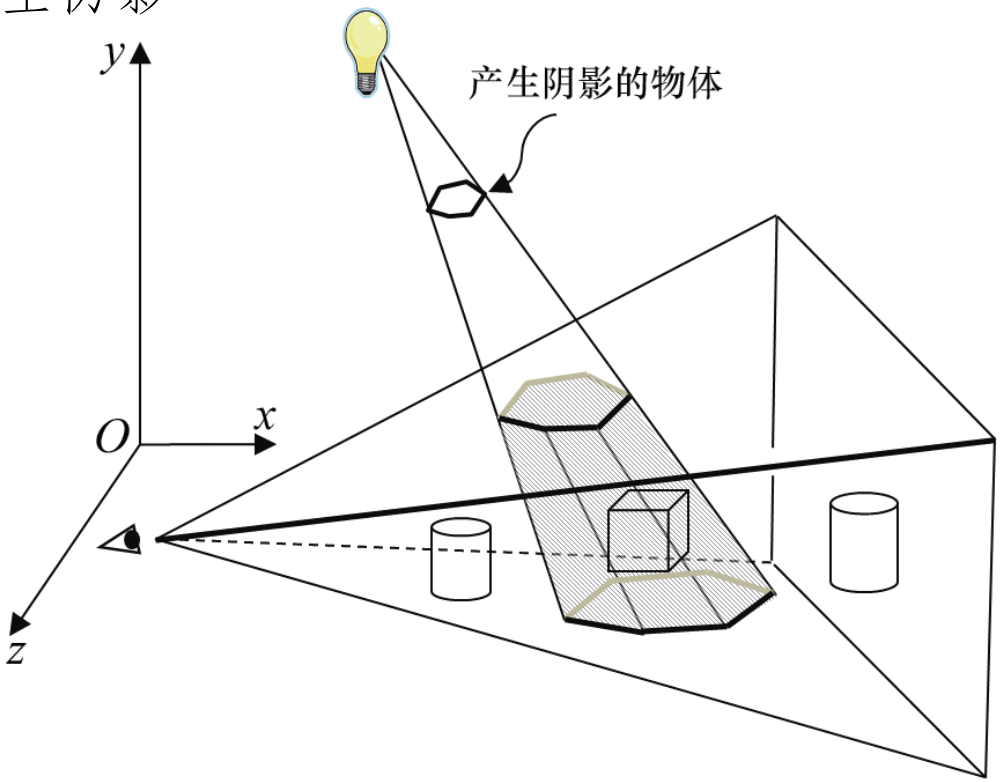
- **优点**：简单、高效
- **缺点**：仅适用于平面



阴影体

- 找到被物体阴影覆盖的阴影体
- 减少视体与阴影体相交部分中的多边形的颜色强度

- **优点**：准确，不容易产生伪影
- **缺点**：计算成本高



阴影贴图

- 想法：光线无法“看到”的任何东西都在阴影中。如果对象 1 阻挡光线到达对象 2，等同于光线不能“看到”对象 2。

- 策略：暂时将相机移动到光的位置，应用 Z-buffer算法，然后使用生成的深度信息来计算。

- 算法：两轮渲染场景
  - （第 1 轮）从光源的位置渲染场景。对于每个像素，深度缓冲区包含光源与最近的对象之间的距离。将深度缓冲区复制到单独的“阴影缓冲区”（阴影纹理或阴影贴图）。
  - （第 2 轮）从相机角度正常渲染场景。对于每个像素，在阴影缓冲区中查找相应的位置。如果光源到渲染点的距离大于从阴影缓冲区检索到的值，则在该像素处绘制的对象离光源的距离比当前离光源最近的对象离光源更远，因此该像素处于阴影中。

阴影贴图（第 1 轮）——从光源位置“绘制”物体

- 将相机移动到光源的位置，然后渲染场景以正确填充深度缓冲区
- 没有必要为像素生成颜色，仅使用顶点着色器，片段着色器不执行任何操作。

- 重要细节：
  - 配置缓冲区和阴影纹理。
  - 禁用颜色输出。
  - 在光源处为视野中的物体构建一个 LookAt 矩阵。
  - 启用 GLSL 第 1 轮着色器程序，准备接收MVP 矩阵。 MVP 包括模型矩阵 M、 LookAt 矩阵（作为观察矩阵 V），以及透视矩阵 P
  - 调用glDrawArrays()

阴影贴图（第 1 轮）——从光源位置“绘制”物体

- 第 1 轮中不需要包含纹理或光照，因为对象不会渲染到屏幕上

```
#version 430                    //顶点着色器
layout (location=0) in vec3 vertPos;
uniform mat4 shadowMVP;

void main(void)
{    gl_Position = shadowMVP * vec4(vertPos,1.0);
}
```

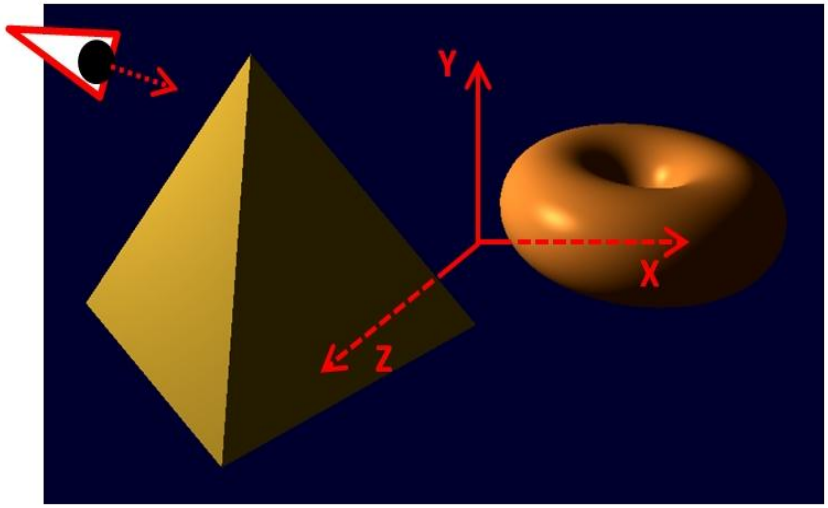
```
#version 430                    //片段着色器
void main(void) { }
```

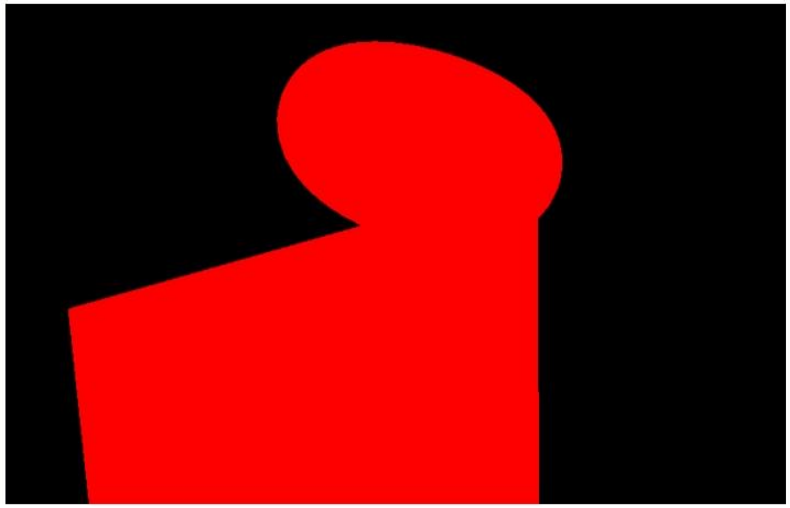
图 8.5 阴影贴图第 1 轮的顶点着色器和片段着色器

阴影贴图（第 1 轮）
——从光源位置“绘制”物体

- （可选）为了测试，可以使用片段着色器

```
#version 430
out vec4 fragColor;
void main(void)
{    fragColor = vec4(1.0, 0.0, 0.0, 0.0);
}
```





阴影贴图（中间步骤）——将深度缓冲区复制到纹理

- 第一种方法
  - 生成空阴影纹理
  - 使用命令 glCopyTexImage2D()将活动的深度缓冲区复制到阴影纹理中
- 第二种方法
  - 构建一个“自定义帧缓冲区”（而不是使用默认的深度缓冲区）
  - 使用命令 glFrameBufferTexture()将阴影纹理附加到它上面
  - 注：OpenGL 在 3.0 版中引入该命令，以进一步支持阴影纹理。使用这种方法时，无须将深度缓冲区“复制”到纹理中，因为缓冲区已经附加了纹理，深度信息由 OpenGL 自动放入纹理中。

阴影贴图（第 2 轮）——渲染带阴影的场景

- OpenGL 相机使用[−1,+1]坐标空间，而纹理贴图使用[0,1]空间
- 构建一个额外的变换矩阵 B，从相机空间转换到纹理空间

$$B = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{shadowMVP2} = [B]\,[\text{shadowMVP}_{(\text{pass1})}]$$

B translates by ½ and scales by ½ to change the light coordinates from a range of (-1..1) to a range of (0..1).

阴影贴图（第 2 轮）——渲染带阴影的场景

- 构建变换矩阵 B，用于从光照空间转换到纹理空间。
- 启用阴影纹理以进行查找。
- 启用颜色输出。
- 启用 GLSL 第 2 轮渲染程序，包含顶点着色器和片段着色器。
- 根据相机位置（正常）为正在绘制的对象构建 MVP 矩阵。
- 构建 shadowMVP2 矩阵（包含矩阵 B，如前所述）——着色器将用它查找阴影纹理中的像素坐标。
- 将生成的变换矩阵发送到着色器统一变量。
- 像往常一样启用包含顶点、法向量和纹理坐标（如果使用）的缓冲区。
- 调用 glDrawArrays()

阴影贴图（第 2 轮）——渲染带阴影的场景

- 除了渲染任务外，顶点和片段着色器还需要额外承担一些任务
  - 顶点着色器将顶点位置从相机空间转换为光照空间，并将结果坐标发送到顶点属性中的片段着色器，以便对它们进行插值。这样片段着色器可以从阴影纹理中检索正确的值。
  - 片段着色器调用 textureProj()函数，该函数返回 0 或 1，指示像素是否处于阴影中。如果像素处于阴影中，则着色器通过剔除其漫反射和镜面反射分量来输出更暗的像素。

阴影贴图是一种常见任务，因此 GLSL 为其提供了一种特殊类型的采样器变量，称为sampler2DShadow，可以附加到 C++ / OpenGL 应用程序中的阴影纹理中。

C++/OpenGL application

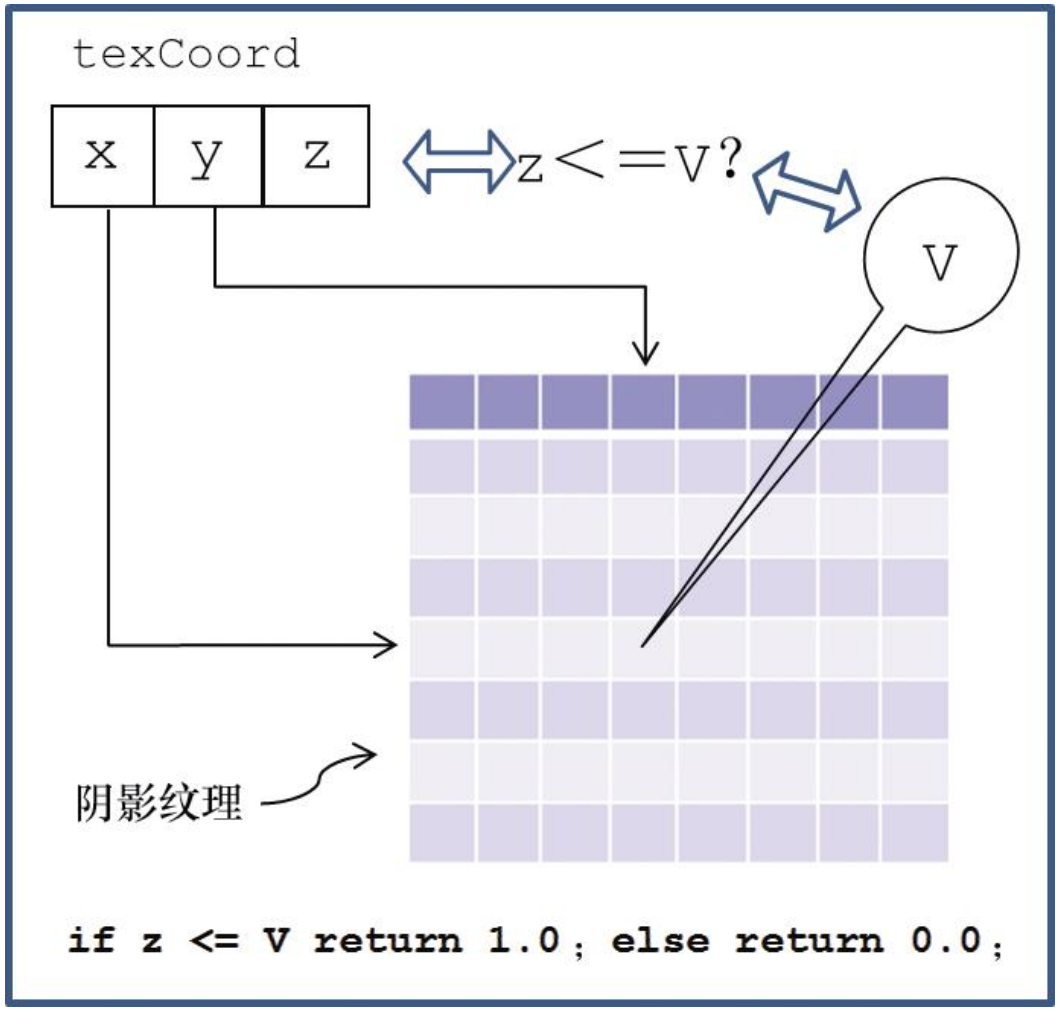
```
. . .
// shadow-related variables
nt screenSizeX, screenSizeY;
GLUint shadowTex, shadowBuffer;
glm::mat4 lightVmatrix;
glm::mat4 lightPmatrix;
glm::mat4 shadowMVP1;
glm::mat4 shadowMVP2;
glm::mat4 b;

void init(GLFWwindow* window)  {
    . . .
    setupShadowBuffers();
    // create "B" matrix
    b = glm::mat4(
            0.5f, 0.0f, 0.0f, 0.0f,
            0.0f, 0.5f, 0.0f, 0.0f,
            0.0f, 0.0f, 0.5f, 0.0f,
            0.5f, 0.5f, 0.5f, 1.0f );
}
```

continued . . .

阴影贴图（第 2 轮）
——渲染带阴影的场景

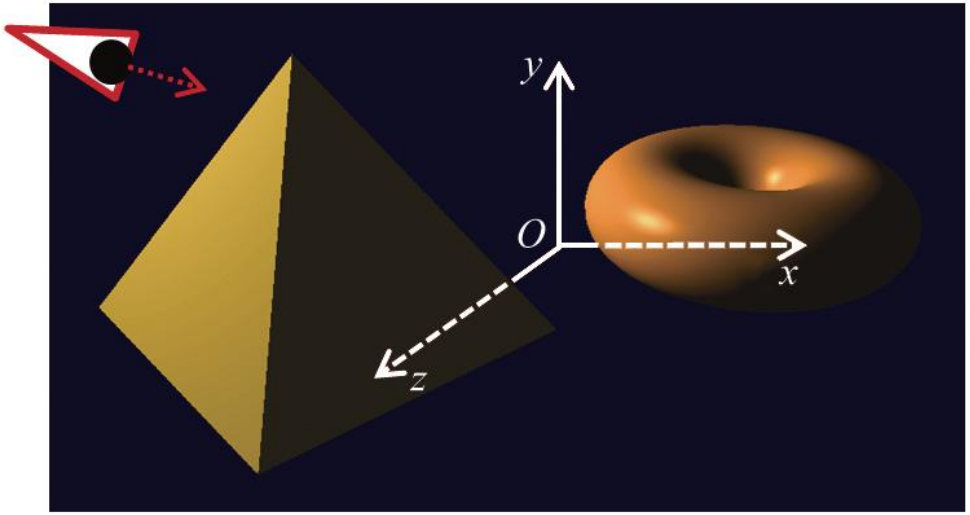
- sampler2DShadow 中深度比较



阴影贴图示例

- 为了阐明示例的开发，我们的第一步是将第 1 轮渲染到屏幕以确保它正常工作
- 临时添加一个简单的片段着色器（不会包含在最终版本中）并在第 1 轮中仅输出一种固定颜色（红色），

```
#version 430
out vec4 fragColor;
void main(void)
{ fragColor = vec4(1.0, 0.0, 0.0, 0.0);
}
```



Vertex Shader

```
. . .
out vec4 shadow_coord;
. . .
uniform mat4 shadowMVP2;
void main(void)
{   . . .
    shadow_coord = shadowMVP2 * vec4(vertPos,1.0);
}
```

Fragment Shader

```
. . .
in vec4 shadow_coord;
layout (binding=0) uniform sampler2DShadow shTex;
. . .
void main(void)
{   . . .
    float notInShadow = textureProj(shTex, shadow_coord);
    fragColor = globalAmbient * material.ambient +  light.ambient * material.ambient;
    If (notInShadow == 1.0)
    {     fragColor += light.diffuse * material.diffuse * max(dot(L,N),0.0)
            + light.specular * material.specular
            * pow(max(dot(H,N),0.0),material.shininess*3.0);
    }
}
```

```
void setupShadowBuffers(GLFWwindow* window)  {
    glfwGetFramebufferSize(window, &width, &height);
    screenSizeX = myCanvas.getWidth();
     screenSizeY = myCanvas.getHeight();
    // create the custom frame buffer
     glGenFramebuffers(1, &shadowBuffer);
    //    create the shadow texture and configure it to hold depth information.
    //    these steps are similar to those in Program 5.2
     glGenTextures(1, &shadowTex);
     glBindTexture(GL_TEXTURE_2D, shadowTex);
     glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
                                        screenSizeX, screenSizeY, 0,
                                        GL_DEPTH_COMPONENT, GL_FLOAT, 0);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                                        GL_COMPARE_REF_TO_TEXTURE);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC,
                                        GL_LEQUAL);
}
```

continued . . .

```
void display(GLFWwindow* window, double currentTime)  {
    . . .
    // vector from light to origin
    lightVmatrix = glm::lookAt(currentLightPos, origin, up);
    lightPmatrix = glm::perspective(toRadians(60.0f), aspect, 0.1f, 1000.0f);

    //  make the custom frame buffer current, and associate it with the shadow texture
    glBindFramebuffer(GL_FRAMEBUFFER, shadowBuffer);
    gl.glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                                                    shadowTex, 0);

    //  disable drawing colors, but enable the depth computation
    glDrawBuffer(GL_NONE);
    glEnable(GL_DEPTH_TEST);

    passOne();

    //  restore the default display buffer, and re-enable drawing
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, shadowTex);
    glDrawBuffer(GL_FRONT);  // re-enables drawing colors

    passTwo();
}
```

continued . . .

```
void passOne(void)  {
    // rendering_program1 contains only the pass one vertex shader
    glUseProgram(renderingProgram1);

    // build the torus object's Model matrix
    mMat = glm::translate(glm::mat4(1.0f), torusLoc);

    // we are drawing from the light's point of view, so we use the light's P and V matrices
    shadowMVP1 = lightPmatrix * lightVmatrix * mMat;
    sLoc = glGetUniformLocation(renderingProgram1, "shadowMVP");
    glUniformMatrix4fv(sLoc, 1, GL_FALSE, glm::value_ptr(shadowMVP1));
    . . .
    glDrawElements(…)

    //  repeat for the pyramid (but don't clear the GL_DEPTH_BUFFER_BIT)
}
```

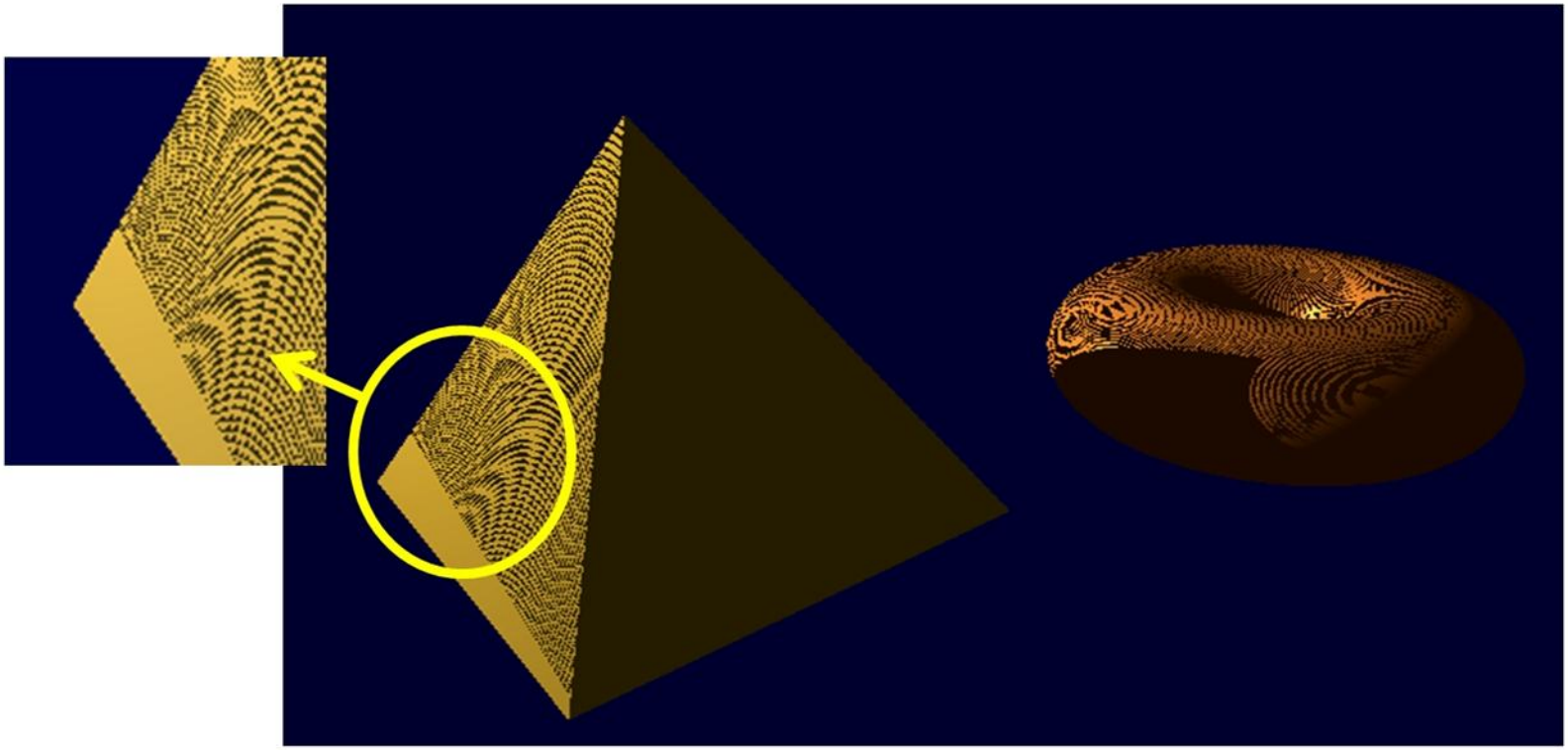
continued . . .

```
void passTwo(void)  {
    . . .
    glUseProgram(renderingProgram2);
    . . .
    sLoc = glGetUniformLocation(renderingProgram2, "shadowMVP2");
    . . .
    shadowMVP2 = b * lightPmatrix * lightVmatrix * mMat;
    . . .
    glUniformMatrix4fv(sLoc, 1, GL_FALSE, glm::value_ptr(shadowMVP2));
    . . .
    glDrawElements( . . . )

    // repeat for each object drawn in the scene
    . . .
}
```

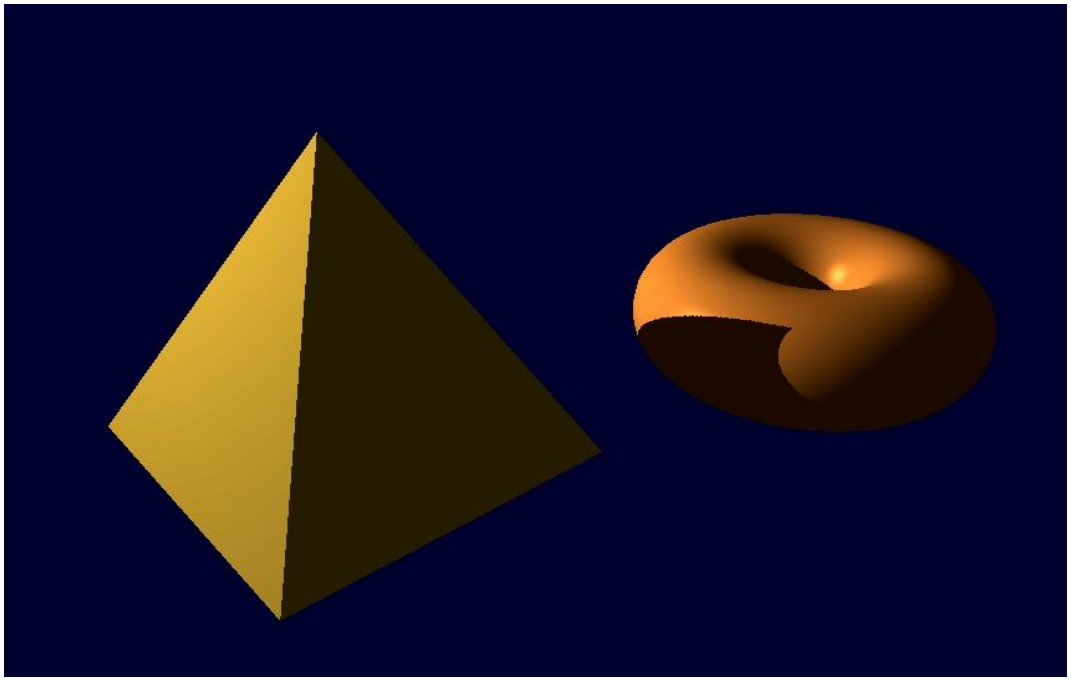
阴影贴图的伪影

- 阴影伪影是由深度测试期间的舍入误差引起的
- 解决方法：在第 1 轮中将每个像素稍微移离光源，之后在第 2 轮将它们移回原位。



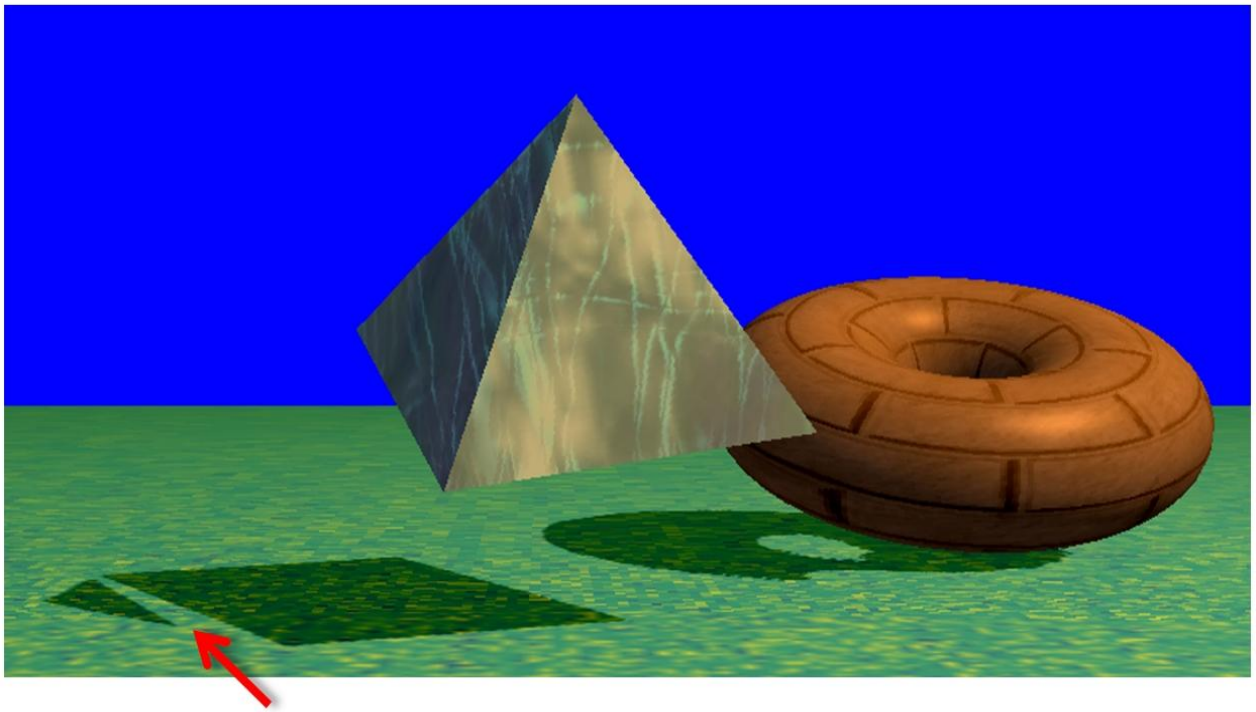
阴影贴图的伪影

```
void display(GLFWwindow* window double currentTime)  {
    . . .
    glEnable(GL_POLYGON_OFFSET_FILL);
    glPolygonOffset(2.0f, 4.0f);
    passOne();
    glDisable(GL_POLYGON_OFFSET_FILL);
    . . .
    passTwo();
}
```



阴影贴图的伪影

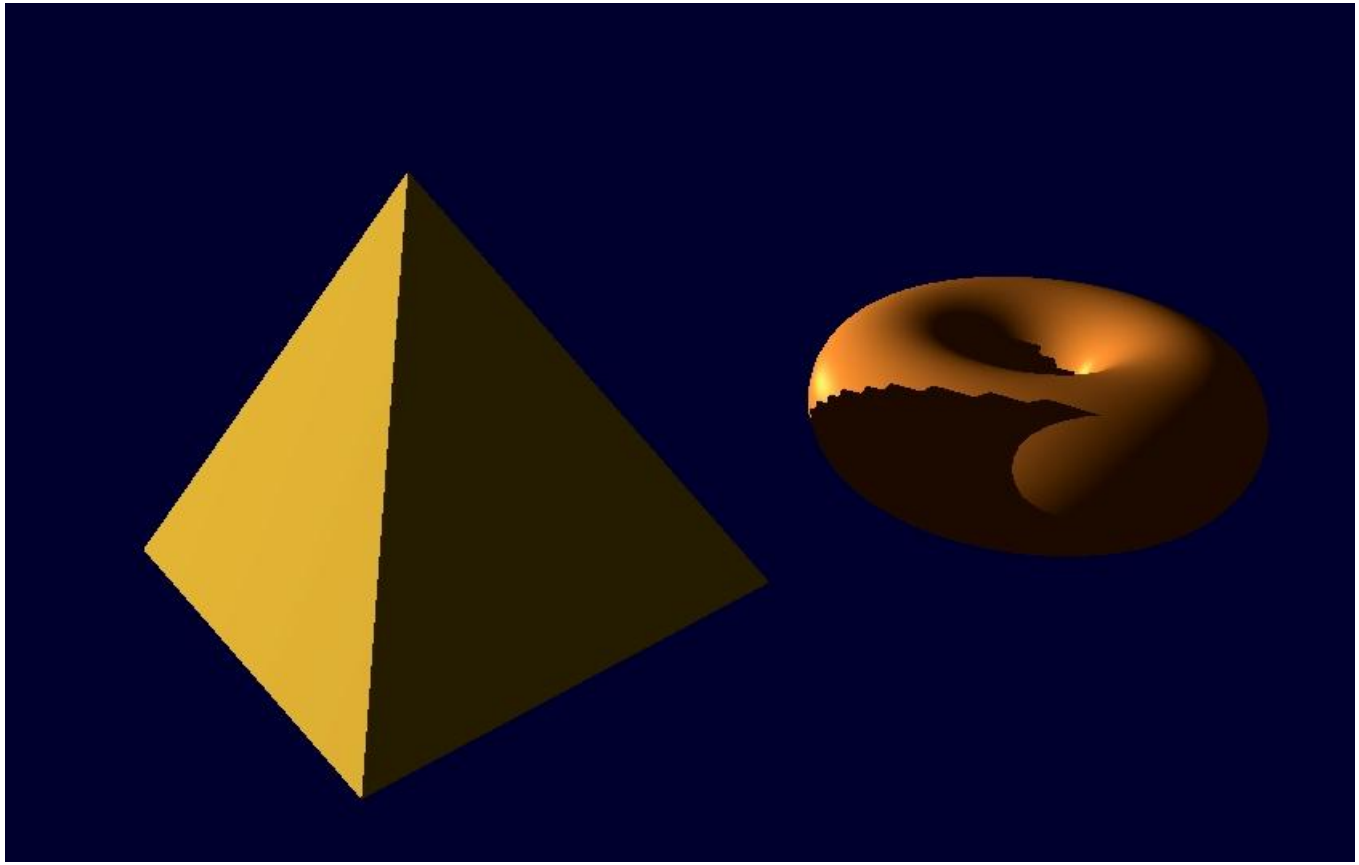
- 修复此伪影需要调整glPolygonOffset()的参数。
  - 如果参数太小，就会出现阴影痤疮;
  - 如果参数太大，则会出现 Peter Panning



“Peter Panning”

阴影贴图的伪影

- 锯齿状阴影边缘，当投射的阴影明显大于阴影缓冲区可以准确表示的阴影时，就有可能出现此问题



现实世界中的柔和阴影

图 8.16 现实世界中的柔和阴影示例

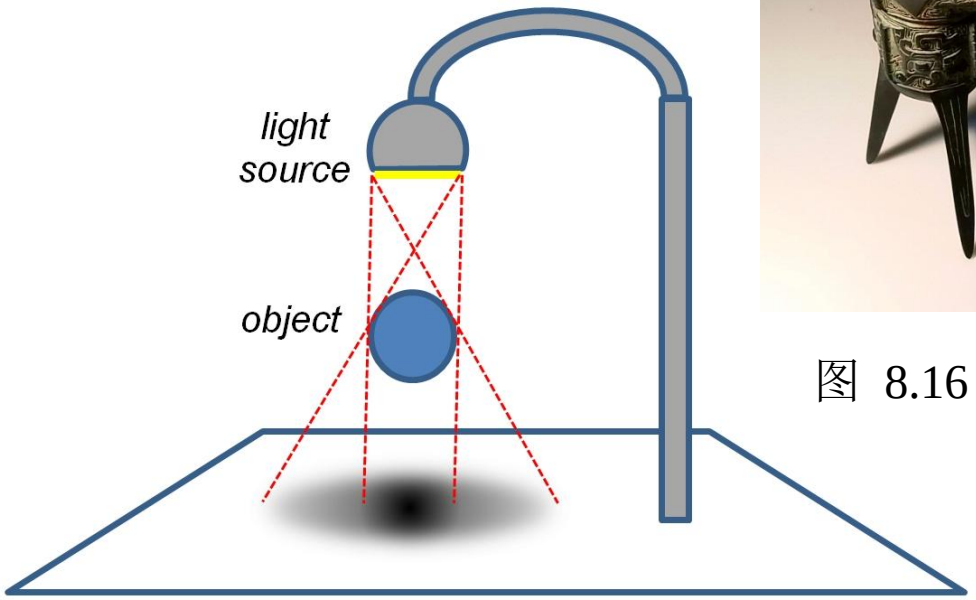
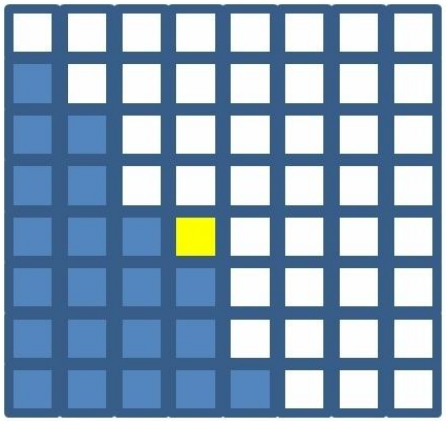
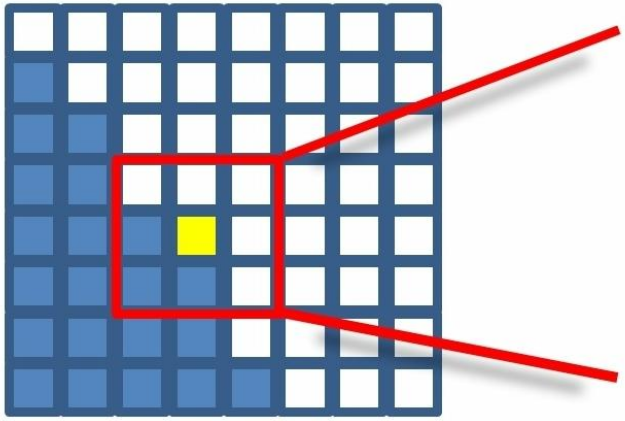
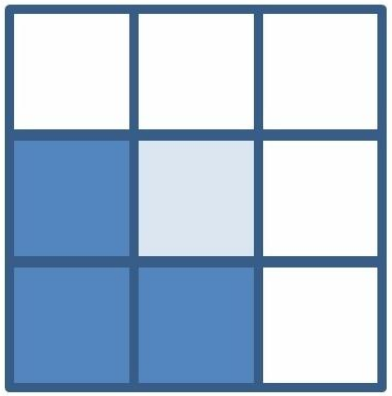


图 8.17 柔和阴影的半影效果

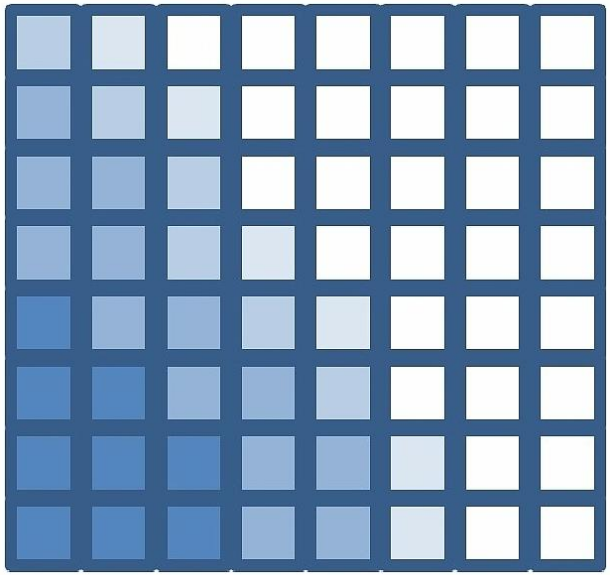
生成柔和阴影——百分比邻近滤波（PCF）







对单个点周围的几个位置的阴影纹理进行采样，以估计附近位置在阴影中的百分比，并根据附近位置在阴影中的百分比，对正在渲染的像素的光照分量进行修改



生成柔和阴影——百分比邻近滤波（PCF）

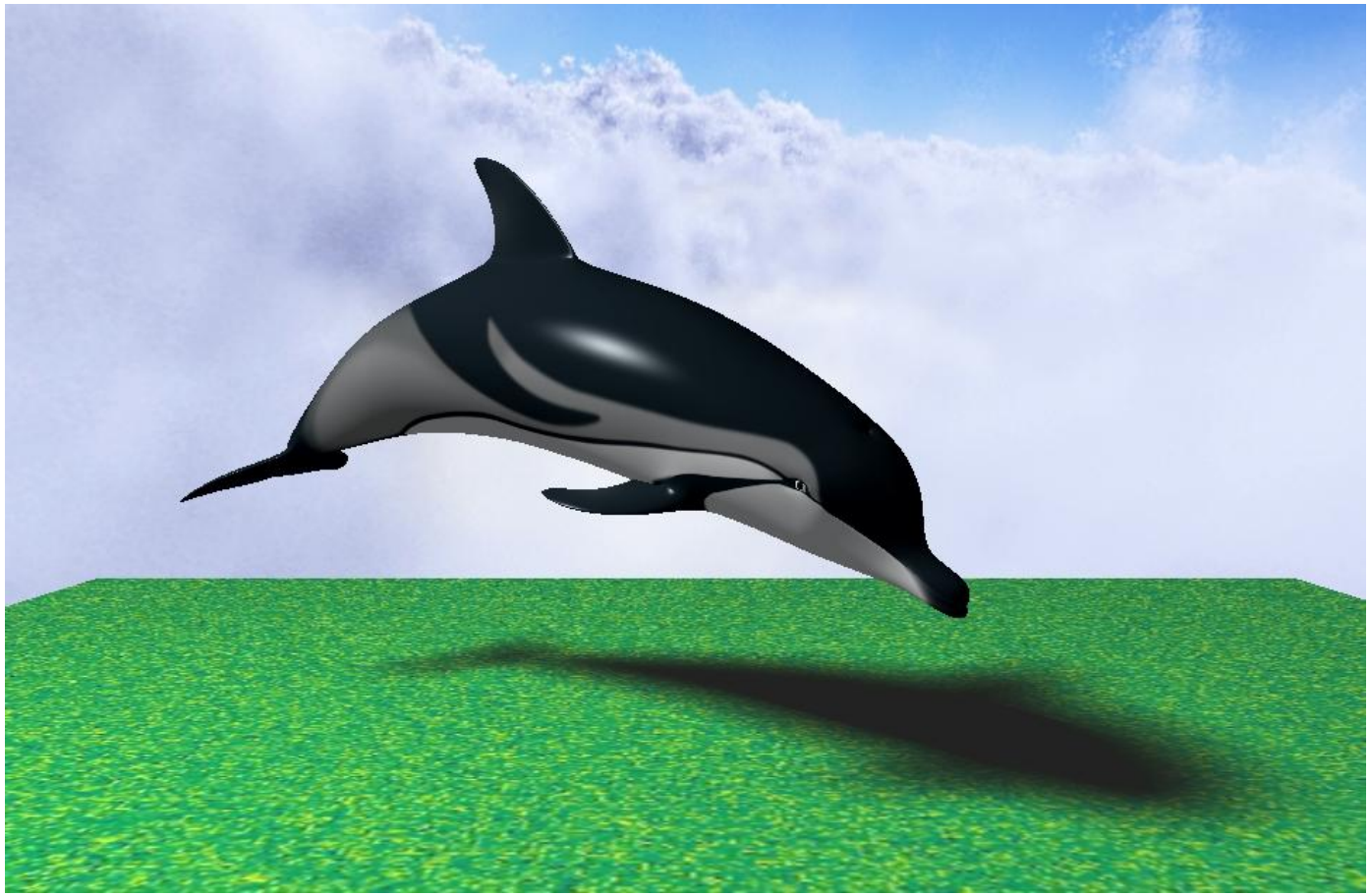


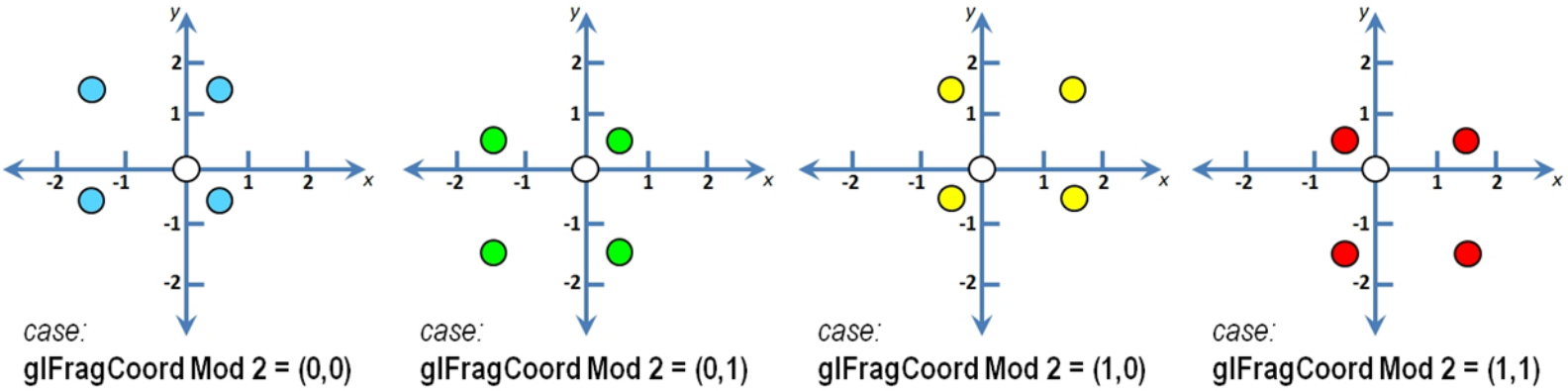
图 8.21 柔和阴影渲染——每像素 64 次采样

生成柔和阴影——百分比邻近滤波（PCF）

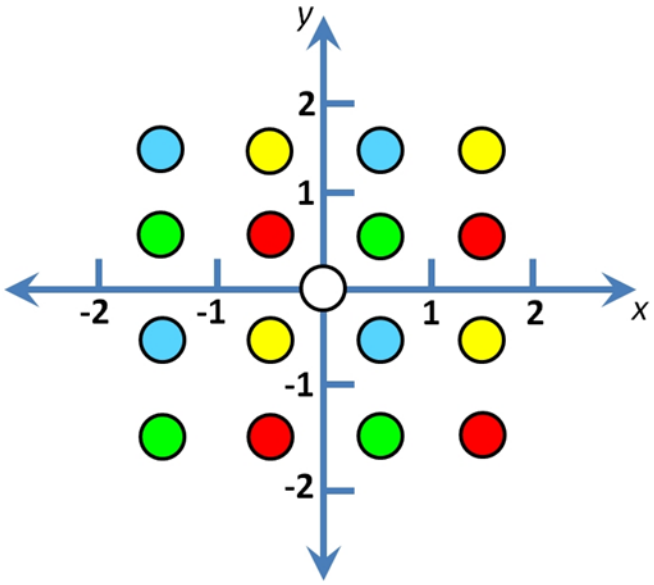
- 在 PCF 的实现中，不对被渲染像素邻域内的每个像素进行采样。这有两个原因：
  - （a）我们想在片段着色器中执行此计算，但片段着色器无法访问其他像素；
  - （b）要获得足够宽的半影效果（例如，10～20 像素宽）需要为每个被渲染的像素采样数百个附近的像素

- 通常的做法：
  - 在阴影贴图中对附近的纹元进行采样。片段着色器可以执行此操作，因为它虽然无法访问附近像素，但可以访问整个阴影贴图。
  - 其次，为了获得足够宽的半影效果，我们采取对附近一定数量的阴影贴图纹元进行采样的方法，每个被采样的纹元都与所渲染像素的纹元有一定距离。

生成柔和阴影——百分比邻近滤波（PCF）



一种常见的方法是假设有 4 种不同偏移模式，每次取其中一种——我们可以通过计算像素的glFragCoord mod 2 值来选择当前像素的偏移模式。使用交错的方式改变偏移量的方法被称为抖动。



生成柔和阴影——百分比邻近滤波（PCF）

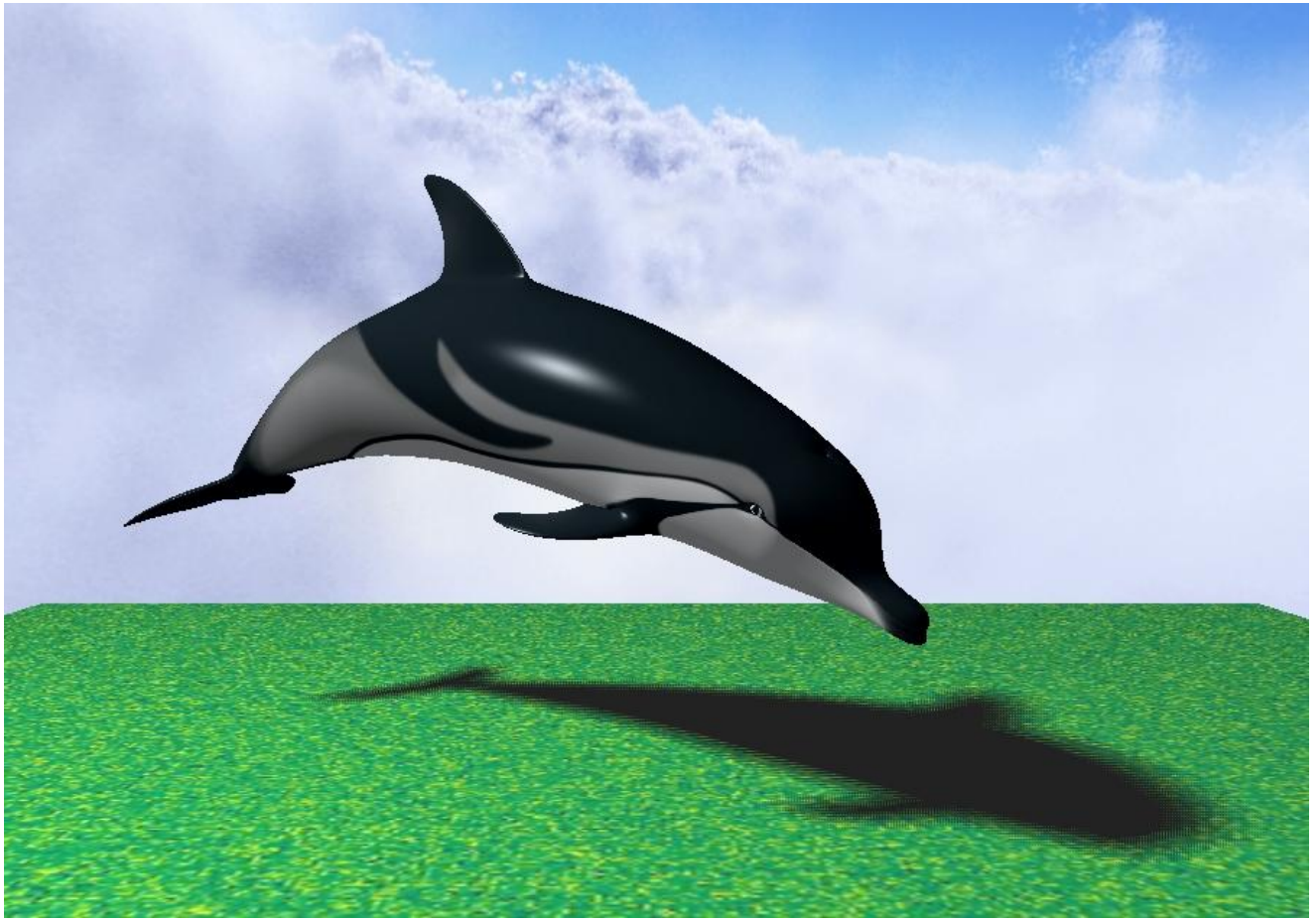


图 8.24 柔和阴影渲染——每像素 4 次

Fragment shader:

```
// Returns the shadow depth value for a texel at distance (x,y) from shadow_coord.
float lookup(float ox, float oy)
{     float t = textureProj(shadowTex,
          shadow_coord + vec4(ox * 0.001 * shadow_coord.w, oy * 0.001 * shadow_coord.w,
          -0.01, 0.0));      // the third parameter (-0.01) is an offset to counteract shadow acne
      return t;
}
void main(void)
{     . . .
      float shadowFactor = 0.0f;
      . . .
      // this section produces a 4-sample dithered soft shadow
      float swidth = 2.5;  // tunable amount of shadow spread
      // produces one of 4 sample patterns depending on glFragCoord mod 2
      vec2 offset = mod(floor(gl_FragCoord.xy), 2.0) * swidth;
      shadowFactor += lookup(-1.5*swidth + offset.x,  1.5*swidth - offset.y);
      shadowFactor += lookup(-1.5*swidth + offset.x, -0.5*swidth - offset.y);
      shadowFactor += lookup( 0.5*swidth + offset.x,  1.5*swidth - offset.y);
      shadowFactor += lookup( 0.5*swidth + offset.x, -0.5*swidth - offset.y);
      shadowFactor = shadowFactor / 4.0;        // average of the four sampled points
      . . .
      fragColor = vec4((shadowColor.xyz + shadowFactor*(lightedColor.xyz)),1.0);
}
```