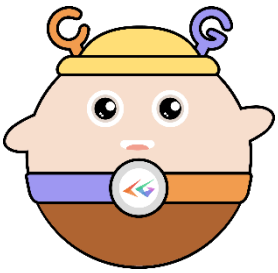
華中科技大學



计算机图形学课程

实验：帧缓冲&&阴影映射

目录

1.帧缓冲 1
1.1 创建一个帧缓冲 1
1.2 纹理附件 2
1.3 渲染缓冲对象附件 3
1.4 总结 4
2.阴影映射 4
2.1 算法思想 4
2.2 深度贴图 5
2.3 渲染阴影 8
2.4 抗锯齿 11

1.帧缓冲

1.1 创建一个帧缓冲

和 OpenGL 其他对象一样，我们使用 glGenFramebuffers 的函数来创建一个帧缓冲对象：

```
unsigned int fbo;
glGenFramebuffers(1, &fbo);
```

它的使用函数也和其它的对象类似。首先我们创建一个帧缓冲对象，将它绑定为激活的(Active)帧缓冲，做一些操作，之后解绑帧缓冲。我们使用 glBindFramebuffer 来绑定帧缓冲。

```
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```

在完成上面的操作后，我们还是不能直接使用帧缓冲，因为一个完整的帧缓冲至少需要满足以下条件：

- 附加至少一个缓冲（颜色、深度或模板缓冲）。
- 至少有一个附件(Attachment)。
- 所有的附件都必须是完整的（保留了内存）。

由于我们的帧缓冲不是默认帧缓冲，渲染指令将不会对窗口的视觉输出有任何影响。出于这个原因，渲染到一个不同的帧缓冲被叫做离屏渲染(Off-screen Rendering)。要保证所有的渲染操作在主窗口中有视觉效果，我们需要再次激活默认帧缓冲，将它绑定到 0：

```
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```

当然，在完成所有的帧缓冲操作之后，不要忘记删除这个帧缓冲对象：

```
glDeleteFramebuffers(1, &fbo);
```

1.2 纹理附件

当把一个纹理附加到帧缓冲的时候，所有的渲染指令将会写入到这个纹理中，就想它是一个普通的颜色/深度或模板缓冲一样。使用纹理的优点是，所有渲染操作的结果将会被储存在一个纹理图像中，我们之后可以在着色器中很方便地使用它。

为帧缓冲创建一个纹理和创建一个普通的纹理步骤类似：

```
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 800, 600, 0, GL_RGB,
GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

创建好纹理后，接下来我们把它附加到帧缓冲上：

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, texture, 0);
```

glFrameBufferTexture2D 有以下的参数：

target：帧缓冲的目标（绘制、读取或者两者皆有）

attachment：我们想要附加的附件类型。当前我们正在附加一个颜色附件。注意最后的 0 意味着我们可以附加多个颜色附件。如 ATTACHMENT1,2…

textarget：你希望附加的纹理类型

texture：要附加的纹理本身

level：多级渐远纹理的级别。我们将它保留为 0。

除了颜色附件之外，我们还可以附加一个深度和模板缓冲纹理到帧缓冲对象中。要附加深度缓冲的话，我们将附件类型设置为 GL_DEPTH_ATTACHMENT。注意纹理的格式(Format)和内部格式(Internalformat)类型将变为

GL_DEPTH_COMPONENT，来反映深度缓冲的储存格式。要附加模板缓冲的话，你要将第二个参数设置为 GL_STENCIL_ATTACHMENT，并将纹理的格式设定为 GL_STENCIL_INDEX。

1.3 渲染缓冲对象附件

渲染缓冲对象(Renderbuffer Object)是在纹理之后引入到 OpenGL 中，作为一个可用的帧缓冲附件类型的，所以在过去纹理是唯一可用的附件。和纹理图像一样，渲染缓冲对象是一个真正的缓冲，即一系列的字节、整数、像素等。渲染缓冲对象附加的好处是，它会将数据储存为 OpenGL 原生的渲染格式，它是为离屏渲染到帧缓冲优化过的。

创建一个渲染缓冲对象和帧缓冲类似：

```
unsigned int rbo;
glGenRenderbuffers(1, &rbo);
```

绑定操作：

```
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
```

由于渲染缓冲对象通常都是只写的，它们会经常用于深度和模板附件，因为大部分时间我们都不需要从深度和模板缓冲中读取值，只关心深度和模板测试。我们需要深度和模板值用于测试，但不需要对它们进行采样，所以渲染缓冲对象非常适合它们。当我们不需要从这些缓冲中采样的时候，通常都会选择渲染缓冲对象，因为它会更优化一点。

创建一个深度和模板渲染缓冲对象可以通过调用 glRenderbufferStorage 函数来完成：

```
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, 800, 600);
```

【GL_DEPTH24_STENCIL8 作为内部格式，它封装了 24 位的深度和 8 位的模板缓冲】

最后附加这个渲染缓冲对象

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER,GL_DEPTH_STENCIL_ATTACHMENT,
GL_RENDERBUFFER, rbo);
```

1.4 总结

在了解帧缓冲原理后，我们将会将场景渲染到一个附加到帧缓冲对象上的颜色纹理中，之后将在一个横跨整个屏幕的四边形上绘制这个纹理。我们将在第二部分的阴影实验中用到，接下来简单介绍下这个步骤：

1) 创建并绑定一个帧缓冲；

2) 创建一个纹理图像并把它作为附件绑定到帧缓冲上；

3) 将纹理附件（或渲染对象附件）附加到帧缓冲上；

4) 检查帧缓冲是否完整（检测是否出现异常）；

5) 在帧缓冲上进行绘制；

6) 解绑 && 删除工作；

2.阴影映射

2.1 算法思想

阴影映射(Shadow Mapping)背后的思路非常简单：具体过程是我们从光源视角出发（以光源所在位置作为摄像机位置），绘制一条射线到达场景上各个片元，得到射线第一次击中的那个物体，然后用这个最近点和射线上其他点进行对比。然后我们将测试一下看看射线上的其他点是否比最近点更远，如果是的话，这个点就在阴影中。我们使用深度缓冲来实现阴影。

深度缓冲里的一个值是摄像机视角下，对应于一个片元的一个 0 到 1 之间的深度值。我们从光源的透视图来渲染场景，并把深度值的结果储存到纹理中。通过这种方式，我们就能对光源的透视图所见的最近的深度值进行采样。最终，深度值就会显示从光源的透视图下见到的第一个片元了。我们管储存在纹理中的所有这些深度值，叫做深度贴图（depth map）或阴影贴图。

2.2 深度贴图

第一步我们需要生成一张深度贴图(Depth Map)。深度贴图是从光的透视图里渲染的深度纹理，用它计算阴影。因为我们需要将场景的渲染结果储存到一个纹理中，因此需要用到帧缓冲。按照第一节的帧缓冲使用步骤来使用：

首先，我们要为渲染的深度贴图创建一个帧缓冲对象：

```
GLuint depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
```

创建一个 2D 纹理，提供给帧缓冲的深度缓冲使用：

```
const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
GLuint depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,SHADOW_WIDTH,
SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

【我们只关心深度值，我们要把纹理格式指定为 GL_DEPTH_COMPONENT，1024 是我们预先设置的深度贴图的解析度】

绑定帧缓冲，并将纹理附件附加到帧缓冲上：

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_TEXTURE_2D, depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

【我们需要的只是在从光的透视图下渲染场景的时候深度信息，所以颜色缓冲没有用。然而帧缓冲对象不是完全不包含颜色缓冲的，所以我们需要显式告诉 OpenGL 我们不适用任何颜色数据进行渲染。调用 glDrawBuffer 和 glReadBuffer 把读和绘制缓冲设置为 GL_NONE 来做这件事】

接下来我们渲染深度贴图，主要分为两个步骤：

- 光源空间的变换
- 渲染至深度贴图

（1）光源空间的变换，这一部分的变换在着色器内完成

本次实验我们使用的是一个所有光线都平行的定向光。出于这个原因，我们将为光源使用正交投影矩阵：

```
GLfloat near_plane = 1.0f, far_plane = 7.5f;
glm::mat4 lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f,
near_plane, far_plane);
```

其次创建一个视图矩阵来变换每个物体，把它们变换到从光源视角可见的空间中，我们将使用 glm::lookAt 函数；这次从光源的位置看向场景中央：

```
glm::mat4 lightView = glm::lookAt(glm::vec(-2.0f, 4.0f, -1.0f),
glm::vec3(0.0f), glm::vec3(1.0));
```

二者相结合为我们提供了一个光空间的变换矩阵，它将每个世界空间坐标变换到光源处所见到的那个空间：

```
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
```

（2）进行一次渲染，形成深度贴图

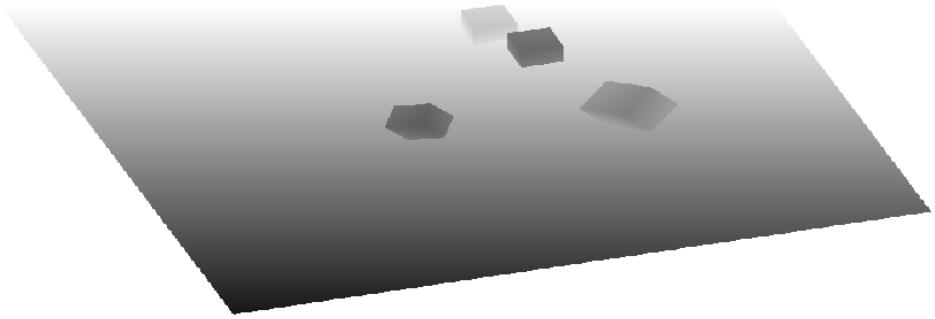
```
        // 渲染阴影深度贴图
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
    glClear(GL_DEPTH_BUFFER_BIT);
    shadowMap_shader.Use();
    shadowMap_shader.SetMat4("lightPV", lightPV);
    DrawScene(shadowMap_shader, current_frame);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

此时顶点着色器内容即为（1）中的空间变换，而片段着色器则什么也不做，如下所示：

```
void main()
{
    // gl_FragDepth = gl_FragCoord.z;
}
```

【因为其内置变量 gl_FragDepth 会自动更新深度信息】

在光的透视图视角下，很完美地用每个可见片元的最近深度填充了深度缓冲。通过将这个纹理投射到一个 2D 四边形上，就能在屏幕上显示出来，显示结果如下：



2.3 渲染阴影

生成深度贴图以后我们就可以开始生成阴影了。这段代码在像素着色器中执行，用来检验一个片元是否在阴影之中，不过我们在顶点着色器中进行光空间的变换：

```
void main()
{
    //...
    LightSpaceFragPos = lightPV * vec4(FragPos, 1.0f);
    gl_Position = projection * view * vec4(FragPos, 1.0f);
    //...
}
```

LightSpaceFragPos 这个输出向量。我们用同一个 lightSpaceMatrix，把世界空间顶点位置转换为光空间,将这个输出给片段着色器，进行深度的比较。

像素着色器使用 Blinn-Phong 光照模型渲染场景。我们接着计算出一个 shadow 值，当 fragment 在阴影中时是 1.0，在阴影外是 0.0

// 计算阴影

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    vec3 projCoords = fragPosLightSpace.xyz /fragPosLightSpace.w;
    projCoords = projCoords * 0.5 + 0.5;
   // 光的位置视野下最近的深度
    float closestDepth = texture(depthMap, projCoords.xy).r;
    // 片元的当前深度
    float currentDepth = projCoords.z;
    // 阴影偏移，防止阴影失真
     float bias = 0.005;
   float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
    return shadow;
}
```

上面即为片段着色器的相关代码，在计算阴影中我们要注意几个问题：

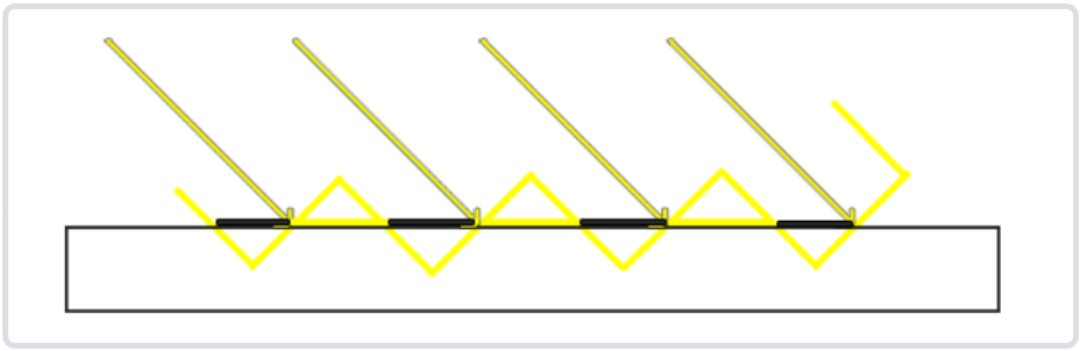
（1）顶点着色器输出一个裁切空间顶点位置到 gl_Position 时，OpenGL 自动进行一个透视除法，将裁切空间坐标的范围-w 到 w 转为-1 到 1，这要将 x、y、z 元素除以向量的 w 元素来实现。由于裁切空间的 FragPosLightSpace 并不会通过 gl_Position 传到像素着色器里，我们必须自己做透视除法

```
vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
```

来自深度贴图的深度在 0 到 1 的范围，我们也打算使用 projCoords 从深度贴图中去采样，所以我们将 NDC 坐标变换为 0 到 1 的范围：

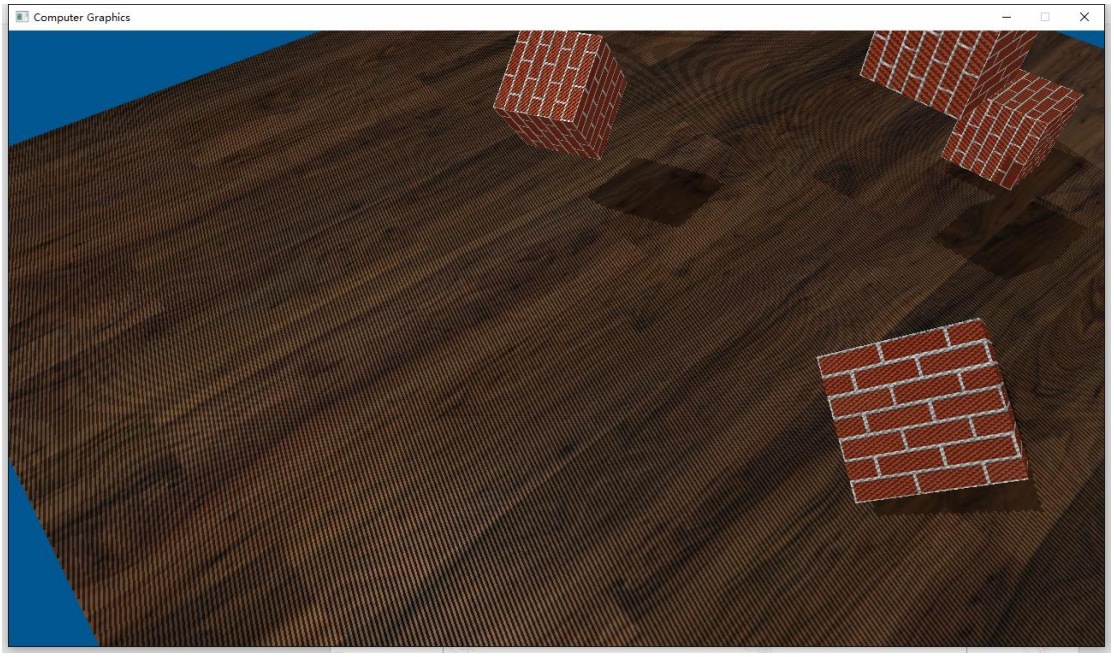
```
projCoords = projCoords * 0.5 + 0.5;
```

在渲染阴影时，很容易出现阴影失真的问题，表现为交替黑线，形成原因如下：



阴影贴图受限于解析度，在距离光源比较远的情况下，多个片元可能从深度贴图的同一个值中去采样。图片每个斜坡代表深度贴图一个单独的纹理像素。你可以看到，多个片元从同一个深度值进行采样。

当光源以一个角度朝向表面的时候就会出问题，这种情况下深度贴图也是从一个角度下进行渲染的。多个片元就会从同一个斜坡的深度纹理像素中采样，有些在地板上面，有些在地板下面；这样我们所得到的阴影就有了差异。因为这个，有些片元被认为是在阴影之中，有些不在，由此产生该现象。

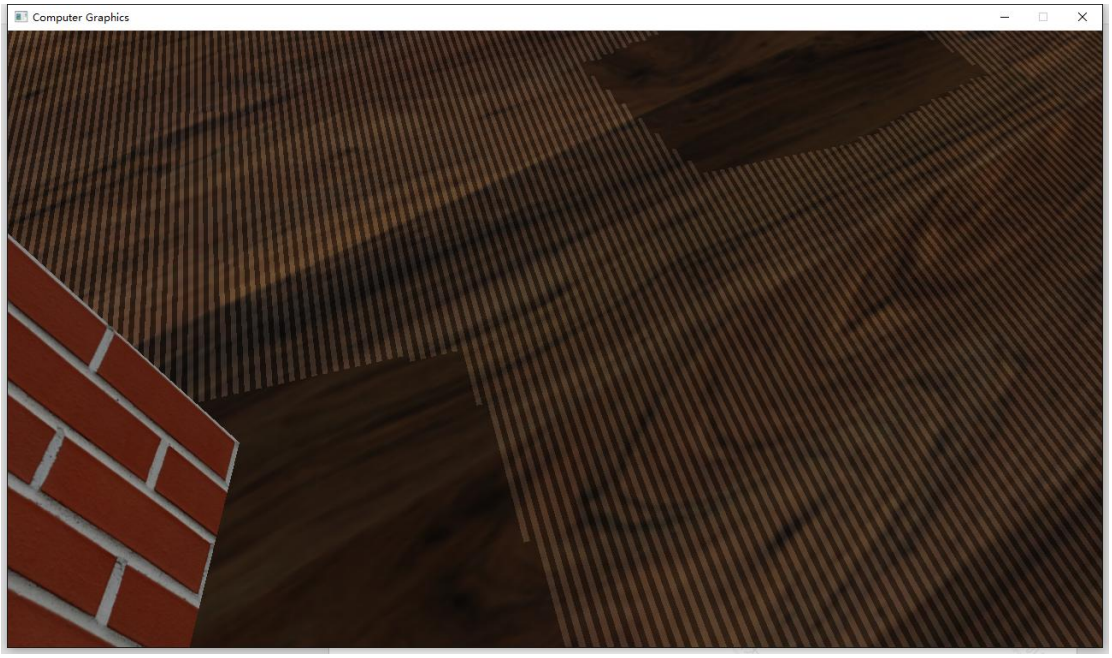


这时，我们用一个叫做阴影偏移（shadow bias）的技巧来解决这个问题，我们简单的对表面的深度（或深度贴图）应用一个偏移量，这样片元就不会被错误地认为在表面之下了：

```
float bias = 0.005;
float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
```

2.4 抗锯齿

如果你放大看阴影，在阴影映射边缘你可以看到很明显的锯齿：



因为深度贴图有一个固定的解析度，多个片元对应于一个纹理像素。结果就是多个片元会从深度贴图的同一个深度值进行采样，这几个片元便得到的是同一个阴影，这就会产生锯齿边；

【可以通过增加解析度来降低锯齿】

另一个（并不完整的）解决方案叫做 PCF（percentage-closer filtering），这是一种多个不同过滤方式的组合，它产生柔和阴影，使它们出现更少的锯齿块和硬边。核心思想是从深度贴图中多次采样，每一次采样的纹理坐标都稍有不同。每个独立的样本可能在也可能不再阴影中。所有的次生结果接着结合在一起，进行平均化，我们就得到了柔和阴影。

```
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
for(int x = -1; x <= 1; ++x)
```

```
    {
        for(int y = -1; y <= 1; ++y)
        {
            float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x,
y) * texelSize).r;
            shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
        }
    }
    shadow /= 9.0;
```

PCF 使用后效果如图：