

Homework 7 - Shadowing Mapping

16 级数媒 颜承轺 15322244

Basic:

1. 实现方向光源的 Shadowing Mapping: 要求场景中至少有一个 object 和一块平面(用于显示 shadow) 光源的投影方式任选其一即可。在报告里结合代码, 解释 Shadowing Mapping 算法

阴影映射 (Shadowing Mapping) 实现原理:

实现阴影映射的第一步是生成深度贴图, 使用一个来自光源的视图和投影矩阵来渲染场景就能创建一个深度贴图。首先, 要为渲染的深度贴图创建一个帧缓冲对象 (118~120 行)。然后, 创建一个 2D 纹理, 提供给帧缓冲的深度缓冲使用 (121~132) 行, 接着把生成的深度纹理作为帧缓冲的深度缓冲 (133~138 行)。

```

118 // 创建一个帧缓冲
119 unsigned int depthMapFBO;
120 glGenFramebuffers(1, &depthMapFBO);
121 // 创建一个2D纹理
122 const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
123 unsigned int depthMap;
124 glGenTextures(1, &depthMap);
125 glBindTexture(GL_TEXTURE_2D, depthMap);
126 glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
127 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
128 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
129 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
130 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
131 float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
132 glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
133 // 把生成的深度纹理作为帧缓冲的深度缓冲
134 glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
135 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
136 glDrawBuffer(GL_NONE);
137 glReadBuffer(GL_NONE);
138 glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

准备工作完成后, 就可以在循环渲染中渲染深度贴图。第 172 行代码提供了一个光空间的变换矩阵, 它能将每个世界空间坐标变换到光源处所见到的那个空间; 这正是渲染深度贴图所需要的。

```

166 //光源空间的变换
167 glm::mat4 lightProjection, lightView;
168 glm::mat4 lightSpaceMatrix;
169 float near_plane = 1.0f, far_plane = 7.5f;
170 lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
171 lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
172 lightSpaceMatrix = lightProjection * lightView;
173 // 渲染深度贴图
174 simpleDepthShader.use();
175 simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
176 glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
177 glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
178 glClear(GL_DEPTH_BUFFER_BIT);
179 renderScene(simpleDepthShader);
180 glBindFramebuffer(GL_FRAMEBUFFER, 0);
181 // 重置viewport
182 glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
183 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

```

上图中 RenderScene 函数是用于渲染场景的, 它的参数是对应的场景着色器。渲染深度贴图使用的着色器比较简单, simpleDepthShader 使用的顶点着色器和片段着色器如下:

```
#version 330 core
```

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

(片段着色器)

```
#version 330 core
```

```
layout (location = 0) in vec3 position;
```

```
uniform mat4 lightSpaceMatrix;
```

```
uniform mat4 model;
```

```
void main()
{
    gl_Position = lightSpaceMatrix * model * vec4(position, 1.0f);
}
```

(顶点着色器)

完成了深度贴图的生成，接下来的第二步就是使用深度贴图渲染阴影。先在顶点着色器中进行光空间的变换，然后在片段着色器中检验片元是否在阴影中。

顶点着色器传递一个普通的经变换的世界空间顶点位置 `vs_out.FragPos` 和一个光空间的 `vs_out.FragPosLightSpace` 给片段着色器。这两项对应的代码如下：

```
vs_out.FragPos = vec3(model * vec4(position, 1.0));
```

```
vs_out.FragPosLightSpace=lightSpaceMatrix*vec4(vs_out.FragPos, 1.0);
```

片段着色器使用 Blinn-Phong 光照模型渲染场景。通过声明一个 `shadowCalculation` 函数，来计算阴影 `shadow` 值，当 `fragment` 在阴影中时是 1.0，在阴影外是 0.0。

```
// calculate shadow
float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;
FragColor = vec4(lighting, 1.0);
```

要检查一个片元是否在阴影中，首先把光空间片元位置转换为裁切空间的标准化设备坐标。当在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。有了这些投影坐标，就能从深度贴图中采样得到 0 到 1 的结果，从第一个渲染阶段的 `projCoords` 坐标直接对应于变换过的 `NDC` 坐标。为了得到片元的当前深度，要获取投影向量的 `z` 坐标，它等于来自光的透视视角的片元的深度。实际的对比就是简单检查 `currentDepth` 是否高于 `closestDepth`，如果是，那么片元就在阴影中。代码如下：

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // 执行透视法
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // 变换到[0, 1]的范围
    projCoords = projCoords * 0.5 + 0.5;
    // 取得最近点的深度(使用[0, 1]范围下的fragPosLight当坐标)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // 取得当前片元在光源视角下的深度
    float currentDepth = projCoords.z;
    // 检查当前片元是否在阴影中
    float shadow = currentDepth > closestDepth ? 1.0 : 0.0;
}
```

激活该着色器、第二个渲染阶段默认的投影以及视图矩阵，就实现了阴影渲染。

修改 GUI

在 GUI 中添加了一个“移动光源”选项，方便观察阴影的变化。可见演示的 gif。

Bonus:

1. 实现光源在正交/透视两种投影下的 Shadowing Mapping

在透视投影中，深度变成了非线性的深度值，它的大多数可辨范围接近于近平面。为了可以像使用正交投影一样合适的观察到深度值，必须先将非线性深度值转变为线性的。代码如下：

```
float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // Back to NDC
    return (2.0 * near_plane * far_plane) / (far_plane + near_plane - z * (far_plane - near_plane));
}

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;
    FragColor = vec4(vec3(LinearizeDepth(depthValue) / far_plane), 1.0); // perspective
    //FragColor = vec4(vec3(depthValue), 1.0); // orthographic
}
```

如代码所示，透视投影需要线性化，而正交投影则不需要。

最终显示出来的效果的区别不太明显。


2. 优化 Shadowing Mapping

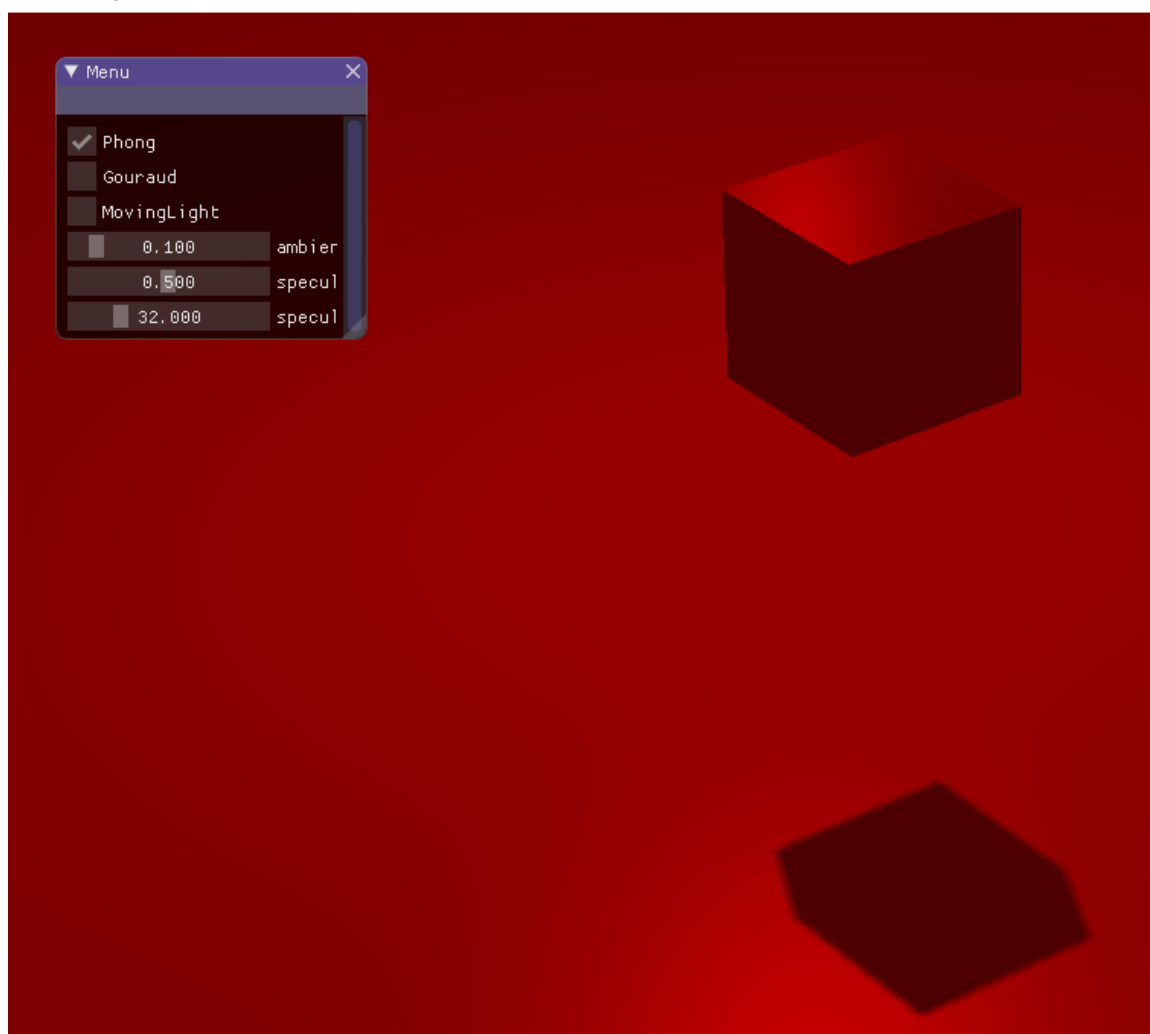
使用 PCF (percentage-closer filtering) 方法可以降低锯齿，它是一种多个不同过滤方式的组合，它产生柔和阴影，使它们出现更少的锯齿块和硬边。核心思想是从深度贴图中多次采样，每一次采样的纹理坐标都稍有不同。每个独立的样本可能在也可能不再阴影中。所有的次生结果接着结合在一起，进行平均化。简单的 PCF 的实现是简单的从纹理像素四周对深度贴图采样，然后把结果平均起来，代码如下：

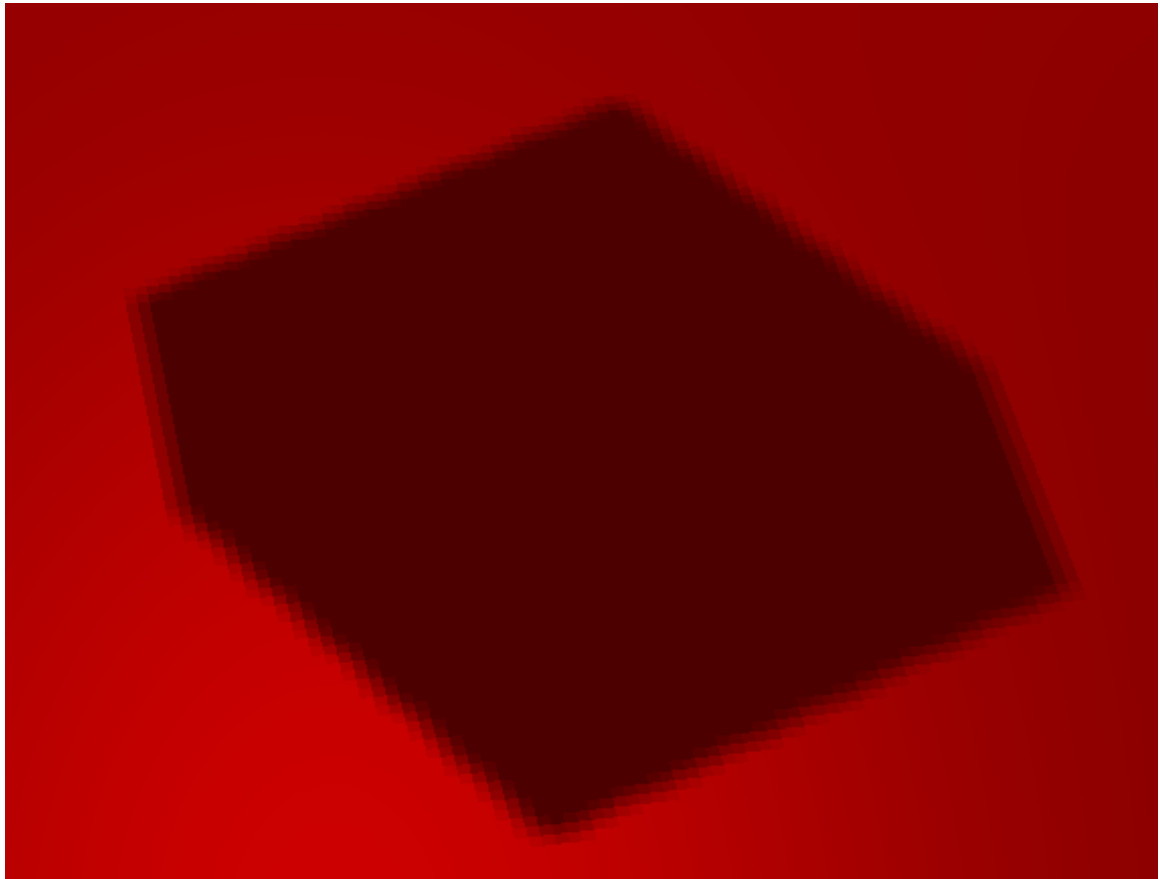
```
// PCF
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
for(int x = -1; x <= 1; ++x)
{
    for(int y = -1; y <= 1; ++y)
    {
        float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;
```

以上代码就是将采样的 9 个值求平均，类似于模糊处理。

得到的结果为：

 LearnOpenGL





能看到阴影的边缘的锯齿变得模糊。从上上张图来看，阴影效果还可以。