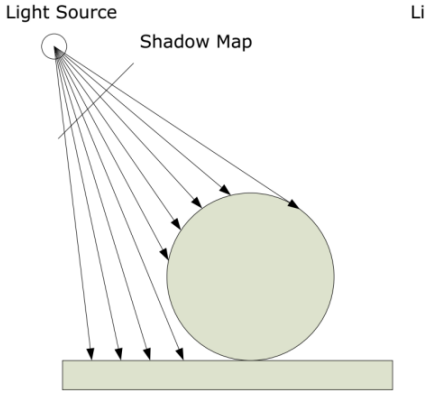
16340304 郑国林

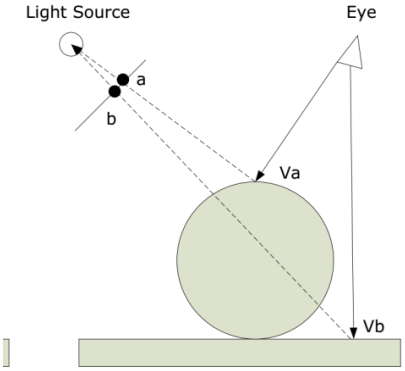
1. 实现方向光源的 Shadowing Mapping:
   要求场景中至少有一个 object 和一块平面(用于显示 shadow)
   光源的投影方式任选其一即可
   在报告里结合代码，解释 Shadowing Mapping 算法
2. 修改 GUI

Shadow mapping 基本原理：

Shadow map 的基本想法就是先在光源的位置放置一个摄像机（Light Space），那么这个相机进行渲染，会把场景投影到相应的投影平面上，在光源的位置看到的东西应该都是能被光照射到的，所以这时候记录的深度值就是在各个像素光源能照射到的最大深度值，而我们要用 Z-Buffer 来保存这个深度值。这就是第一步的渲染步骤。



然后就是根据这个保存的 shadow map 进行第二次的渲染。第二次的渲染是正常步骤的渲染操作，在观察者的位置进行渲染，这个时候在实际生成观察平面上的像素时，先将空间中的点通过第一步中相同的转换矩阵转换到光源所在位置的摄像机对应的 lights pace 中，然后将其得到的深度值与 shadow map 中对应的 z-buffer 的深度值进行比较，如果比 shadow map 中的值小，说明其能被光源位置的摄像机看到，也就是能够被光源照射到，如果比 shadow map 中的值大，说明照射不到，也就是在阴影中。



空间中的一点如果处于观察者 V 的视锥中，同时又位于 Light Space 的视锥之外，那么显然就无法通过上面的方法来判断它是否被阴影所覆盖。这也是 Shadow Map 的局限之处。

OpenGL 中的实现

参考 opengl 上的教程进行实现。

1. 第一步就是进行在光源位置的渲染，并得到 shadow map。
   a. 首先就是要生成用于保存 shadow map 的帧缓存 FBO 对象

```
//产生深度贴图缓冲对象
    unsigned int depthMapFBO;
    glGenFramebuffers(1, &depthMapFBO);
```

然后创建 2d 纹理作为 FBO 对象的附件，要注意创建纹理的时候属性的设置，`glTexParameteri`，并且要注意这里的纹理的大小长和宽，作为我们 shadow map 的一个像分辨率一样的东西。

```
//创建 2d 纹理
    const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
    unsigned int 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_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
    float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
    glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);

    //把纹理作为帧缓冲的附件
    glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2
D, depthMap, 0);
```

然后由于我们希望保存在纹理中的中的只有深度值，而不需要颜色值，所以要通过调用 glDrawBuffer 和 glReadBuffer 把读和绘制缓冲设置为 GL_NONE。

```
//告诉 OpenGL 我们不适用任何颜色数据进行渲染
    glDrawBuffer(GL_NONE);
    glReadBuffer(GL_NONE);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

   b. 接着就是定义第一次在光源位置渲染的顶点着色器和片段着色器
   着色器的主要工作是在顶点着色器里面将坐标变换到以光源位置为视点的 light space，片段着色器的话由于没有颜色缓冲，不需要任何处理。这个 lightSpaceMatrix 我们在第二次渲染的时候还要用到的，将世界坐标转换到 lightspace 的坐标进行比较。

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

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

c. 然后就可以进行第一次渲染了，首先就是定义我们要用到的 view 矩阵和 projection 矩阵，在选投影的方式的时候，无论是正交投影还是透视投影，我们都要注意投影矩阵间接决定可视区域的范围，以及哪些东西不会被裁切，要保证投影视锥体的大小可以包含打算在深度贴图中包含的物体。当物体不在深度贴图中时，它们就不会产生阴影。View 矩阵的话使用 glm::lookAt 函数；从光源的位置看向场景中央。这两个矩阵乘起来就是在着色器中要用到的 lightspaceMatrix 了。

除此之外，我们还要显示的告诉 opengl 我们想将东西渲染到我们之前定义的帧缓冲对象中，并且暂时先将视口修改成我们定义 shadow map 的纹理的大小。因为阴影贴图经常和我们原来渲染的场景有着不同的解析度，我们需要改变视口的参数以适应阴影贴图的尺寸。如果我们忘了更新视口参数，最后的深度贴图要么太小要么就不完整。

这里中的 renderScene 函数是仿照教程上的将渲染平面和 cube 的过程抽象了出来，否则的话因为要渲染好几次代码太冗余了。

```
//从光源视角进行渲染，得到深度贴图
        //先获取对应的 view 和 projection 矩阵
        glm::mat4 lightProjection, lightView;
        glm::mat4 lightSpaceMatrix;
        GLfloat near_plane = 1.0f, far_plane = 7.5f;
        if(perspective)
            lightProjection = glm::perspective(glm::radians(45.0f), (GLfloat)SHADOW_WIDTH / (GLfloat)SHADOW_HEIGHT, near_plane, far_plane);
        else
            lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
        lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
        lightSpaceMatrix = lightProjection * lightView;
        //从光源的视角进行渲染
        simpleDepthShader.use();
        simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);

```

```
        //改变此时渲染的 viewport，作为深度贴图的分辨率？
        glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
        glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
        glClear(GL_DEPTH_BUFFER_BIT);
        renderScene(simpleDepthShader);
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

2. 第二步就是进行观察位置正常的渲染步骤。
   通过上面的在光源位置的渲染就得到了我们的 shadow map 了，存储在一开始定义的二维纹理中。接下来就是正常的渲染步骤，只不过是在着色器中利用 light space 的变换，和 shadow map 的深度值比较来对 shading 得到光强进行阴影修改。

   a. 首先就是把视口改回我们正常大小的视口，并且清除缓存，更改使用的着色器。

```
//先把视口设置回来，清楚缓存
        glViewport(0, 0, WIDTH, HEIGHT);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        shader.use();
```

   b.
   然后就是定义我们正常要用到的着色器了，大部分和之前的光照用到的着色器是一样的，不过增加了计算阴影的步骤。
   在顶点着色器中的操作首先是通过 model 和前面用到的 lightSpaceMatrix 来将坐标转换到 light space，然后还要计算顶点的法向量，这里的法向量是不能直接乘转换矩阵的，而是先求逆在转置，否则的话转换后法向量可能就不垂直了。还有一个步骤就是通过 model，view，projection 来正常将坐标转换到当今视角的 camera 坐标中。

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec4 FragPosLightSpace;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform mat4 lightSpaceMatrix;

```

```
void main()
{
    vs_out.FragPos = vec3(model * vec4(aPos, 1.0));
    vs_out.Normal = transpose(inverse(mat3(model))) * aNormal;
    vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0);

    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

在片段着色器中参照教程，使用 Blinn-Phong 光照模型渲染场景。除此之外还要计算出一个 shadow 值来衡量像素在阴影中的程度，并且 diffuse 和 specular 效果会乘以这个阴影系数来进行阴影效果的渲染。由于阴影不会是全黑的（由于散射），我们把 ambient 分量从乘法中剔除。

片段着色器中的关于漫反射，镜面反射，环境光的计算和上次作业是差不多的，我同样是加入了参数的输入来对这些系数进行调整，并且教程用到了另一个纹理贴图，我就直接将其修改成了输入一个物体的颜色。

主要增加的步骤是计算 shadow 系数，首先当我们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，但是我们这个坐标是通过 lightSpaceMatrix 的转换之后传递过来的，不是通过 gl_Position 传递过来的，所以这一步我们要自己完成。另外，为了和 shadow map 的深度相比较，z 分量需要变换到[0,1]；为了作为从 shadow map 中采样的坐标，xy 分量也需要变换到[0,1]。所以整个 projCoords 向量都需要变换到[0,1]范围。接下来就是根据转换后的坐标从 shadowmap 中提取对应位置的深度值和当前深度值进行比较，判断是否在阴影中。然后根据这个阴影系数来对漫反射和镜面反射进行修改就行了。

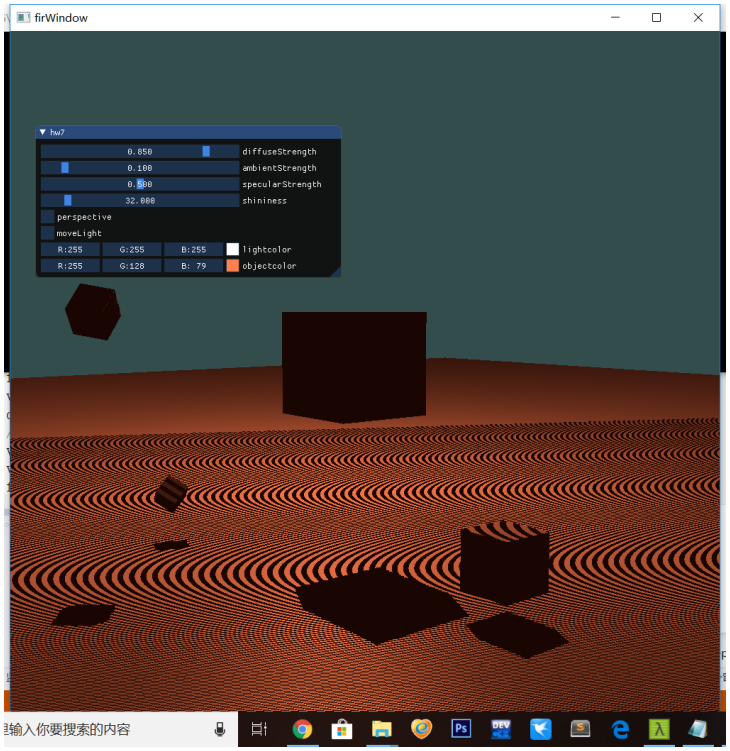
```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec4 FragPosLightSpace;
} fs_in;

uniform sampler2D shadowMap;

uniform vec3 lightPos;
uniform vec3 viewPos;

uniform vec3 objectColor;
uniform vec3 lightColor;
uniform float ambientStrength;
uniform float diffuseStrength;
```

```
uniform float specularStrength;
uniform float Shininess;

float ShadowCalculation(vec4 fragPosLightSpace)
{
    // 执行透视转换
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // 转换到 0-1 的范围
    projCoords = projCoords * 0.5 + 0.5;
    // 根据坐标从 shadowmap 中获取深度值，这个就是光源能照射到的最大的深度值
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // 然后获取现在这个坐标下的深度值
    float currentDepth = projCoords.z;

    float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;
    return shadow;
}

void main()
{
    vec3 color = objectColor;
    vec3 normal = normalize(fs_in.Normal);
    // ambient
    vec3 ambient = ambientStrength * color;
    // diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    diffuse = diffuseStrength * diffuse;
    // specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), Shininess);
    vec3 specular = specularStrength * spec * lightColor;
    // 增加了一个计算 shadow
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
    //shadow 只影响 duffuse 和 specular
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * colo
r;

    FragColor = vec4(lighting, 1.0);
}
```

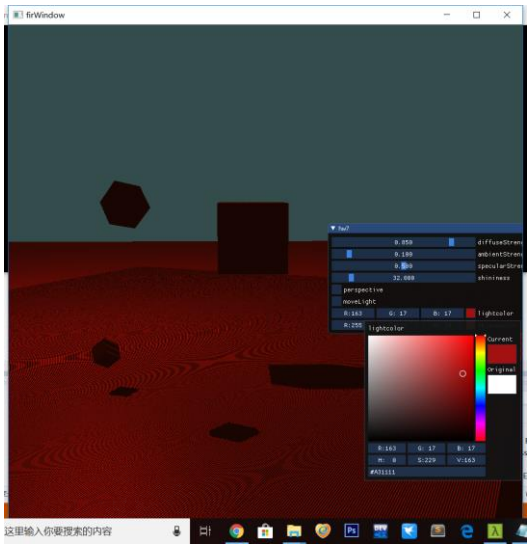
c. 定义完着色器之后，就是根据 camera 位置来得到 view 和 projection 矩阵就可以了。注意这里我们传递 shadowmap 的时候我们是通过传递那个纹理的 id 给着色器的。shader.setInt("shadowMap", 0); 要和 glActiveTexture(GL_TEXTURE0); 的编号对应上，shader 里用那个 texture 函数来进行读取。

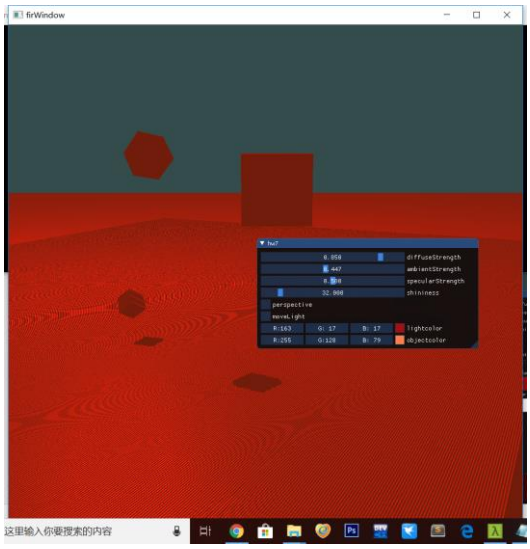
```
//view 矩阵和投影矩阵
        glm::mat4 projection = glm::perspective(glm::radians(45.f), (float)WIDTH / (float)HEIGHT, 0.1f, 100.0f);
        //获取对应的 lookat 矩阵，使得照相机可以更新方向那些
        glm::mat4 view = cam.GetViewMatrix();
        //给 shader 传递坐标转换矩阵
        shader.setMat4("projection", projection);
        shader.setMat4("view", view);
        // 给 shader 传递变量
        shader.setVec3("viewPos", cam.getPosition());
        shader.setVec3("lightPos", lightPos);
        shader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, depthMap);
        renderScene(shader);
```

实现效果截图：



修改 GUI 后，可以设定反射系数以及光的颜色，物体颜色等



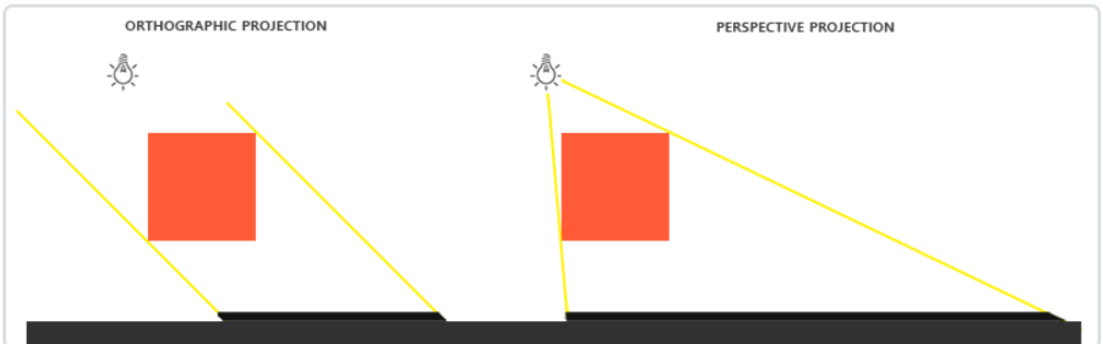


Bonus:

1. 实现光源在正交/透视两种投影下的 Shadowing Mapping
2. 优化 Shadowing Mapping

实现光源在正交/透视两种投影下的 Shadowing Mapping

正交投影更适用于平行光源，透视投影更适用与点光源和聚光灯



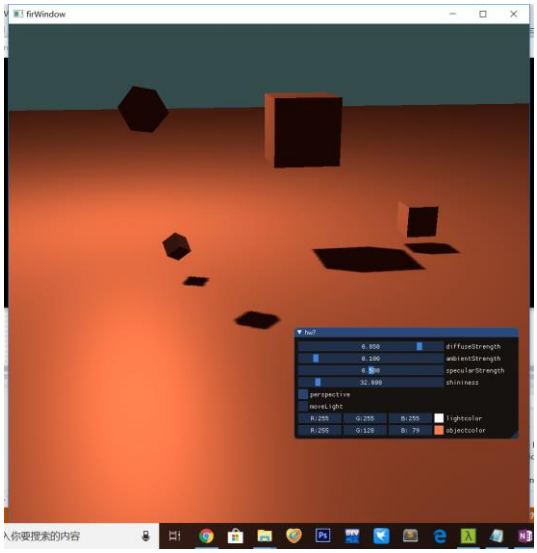
在第一次在光源位置进行渲染的时候，投影矩阵可以选择正交投影或者透视投影

```
1.  if(perspective)
2.              lightProjection = glm::perspective(glm::radians(45.0f), (GLfloat
    )SHADOW_WIDTH / (GLfloat)SHADOW_HEIGHT, near_plane, far_plane);
3.          else
4.              lightProjection = glm::ortho(-10.0f, 10.0f, -
    10.0f, 10.0f, near_plane, far_plane);
```

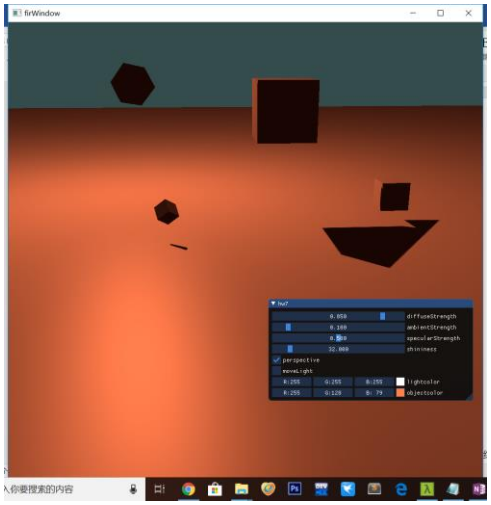
另外，正如前面所写的，在 lightspace 坐标在顶点着色器转移到片段着色器的过程中，我们要自己完成透视除法，将裁切空间坐标的范围-w 到 w 转为-1 到 1。并且根据教程

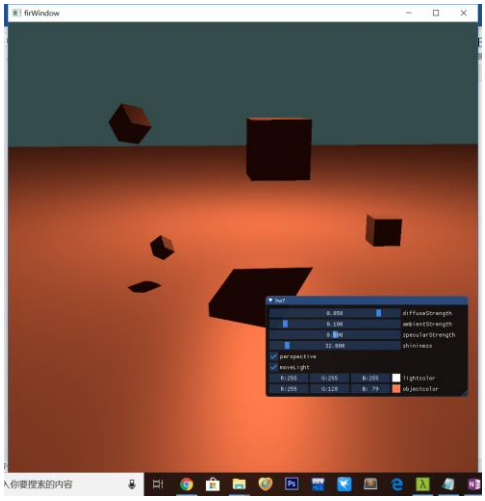
上所说，透视投影矩阵，将深度缓冲视觉化经常会得到一个几乎全白的结果。发生这个是因为透视投影下，深度变成了非线性的深度值，它的大多数可辨范围接近于近平面。为了可以像使用正交投影一样合适的观察到深度值，你必须先讲过非线性深度值转变为线性的。

效果对比：
正交投影：



透视投影，增加了一个移动光源位置的功能来更好的进行观察



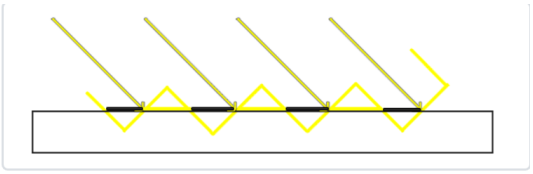


优化 Shadowing Mapping

在上面的效果图中，我们可以看到地板四边形渲染出很大一块交替黑线。这种阴影贴图的不真实感叫做阴影失真，根本原因就是 shadow depth map 的分辨率不够，因此多个 pixel 会对应 map 上的同一个点。

图中黄色箭头是照射的光线，黑色长方形是实际物体表面，黄色的波浪线是 shadow map 中的对应值的情况。因为 shadow mapping 受限于纹理的解析度，在距离光源比较远的情况下，多个 fragment 可能从 shadow map 的同一个值中去采样，可以看到，由于 map 是对场景的离散取样，所以黄色的线段呈阶梯状的波浪变化，相对于实际场景中的情况，就

有一部分比实际场景中的深度要大，这部分不会产生阴影；一部分比实际场景中的深度要小，这部分会产生阴影，所以就出现了条纹状的阴影。



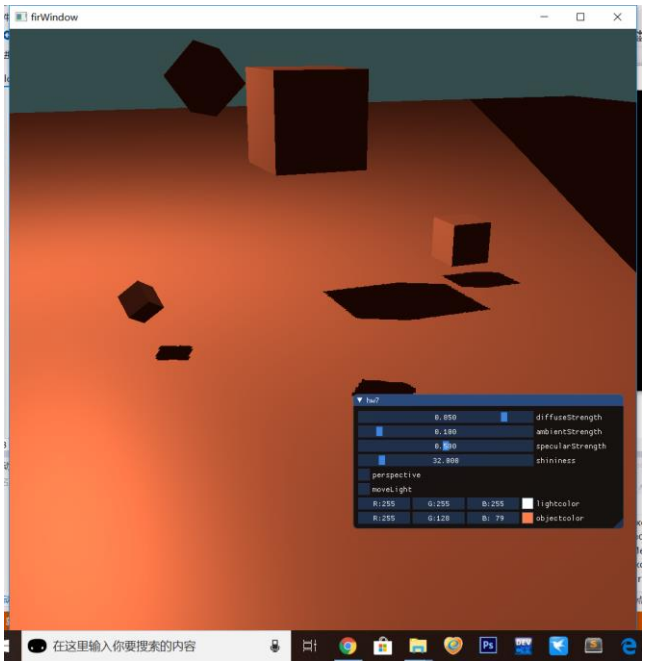
可以用一个阴影偏移的方法来解决这个问题，对表面的深度应用一个偏移量，这样 fragment 就不会被错误地认为在表面之下了。而这个偏移量可以是根据光线的角度来计算的，基于表面法向量和光照方向。这样像地板这样的表面几乎与光源垂直，得到的偏移就很小，而比如立方体的侧面这种表面得到的偏移就更大。

代码实现就在正常渲染的片段着色器中计算 shadow 值的时候进行修改就行了

```
// 根据法向量和光线方向向量来计算偏移值，加上偏移值之后在比较大小
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
    float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
```

实现效果图：

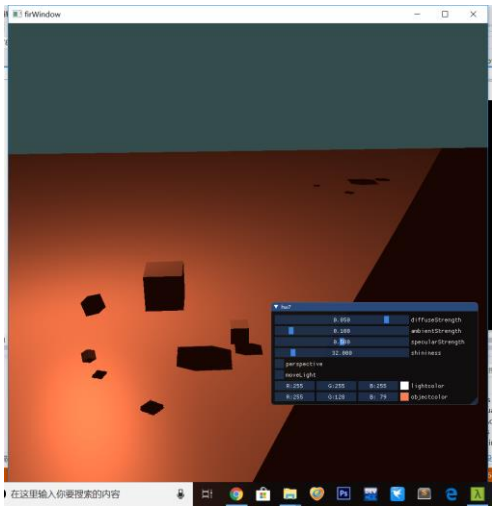
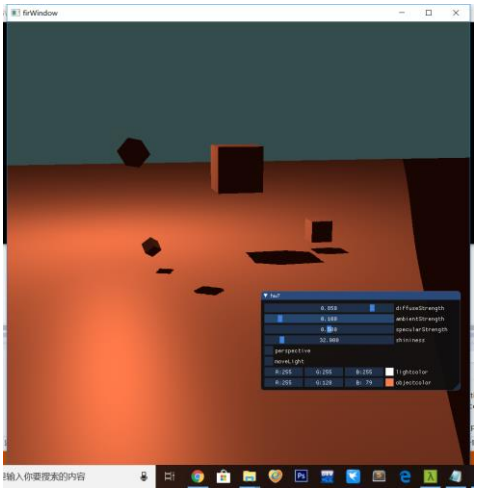
那些横线都没有了



但是仔细看远处会出现一些奇怪的阴影光照有一个区域，超出该区域就成为了阴影；这个区域实际上代表着深度贴图的大小，这个贴图投影到了地板上。发生这种情况的原因是我们之前将深度贴图的环绕方式设置成了 GL_REPEAT。我们可以储存一个边框颜色，然后把深度贴图的纹理环绕选项设置为 GL_CLAMP_TO_BORDER，这样超出的坐标不会在阴影中，显得更加真实。

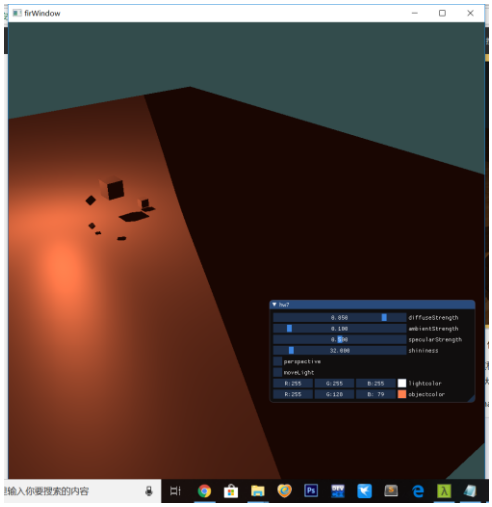
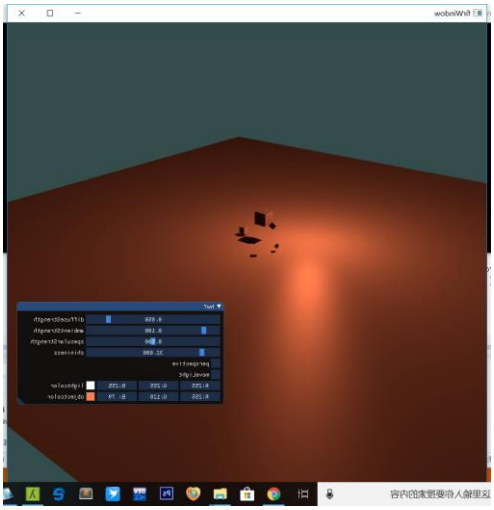
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
```

```
2.      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
3.      float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
4.      glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

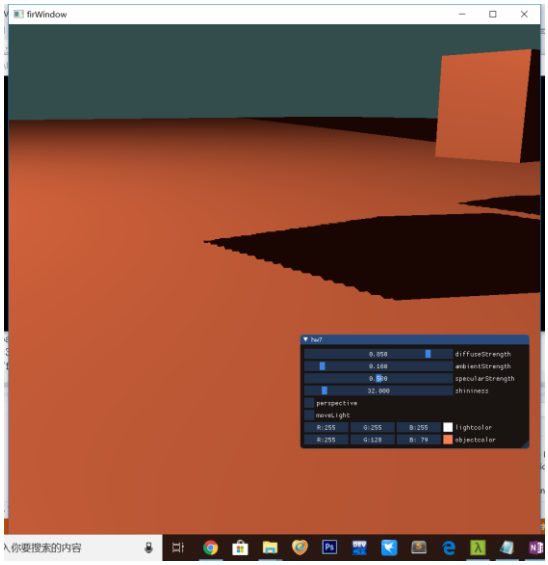
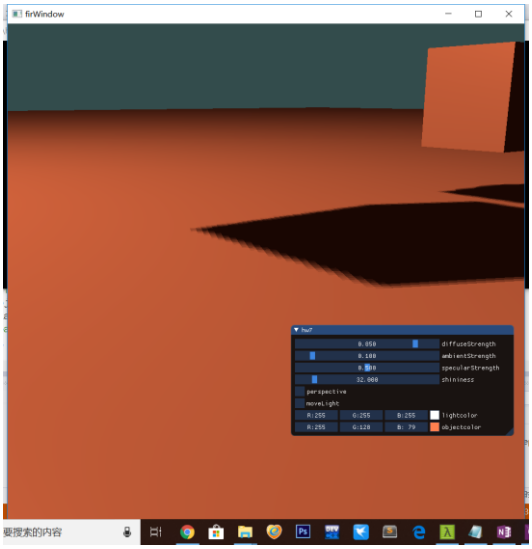
但是这个时候发现右边平面还是有一坨全部变成了阴影，那里的坐标超出了光的正交视锥的远平面，当一个点比光的远平面还要远时，它的投影坐标的 z 坐标大于 1.0。这种情况下，GL_CLAMP_TO_BORDER 环绕方式不起作用，因此可以把坐标的 z 元素和 shadow map 的值进行了对比，z 如果大于 1，就直接把 shadow 设为 0，假设他不在阴影中，这样效果更加真实。

```
1.  //超出视锥体的话直接假设不在阴影中
2.      if(projCoords.z > 1.0)
3.          shadow = 0.0;
```

仔细看阴影的话，会发现有很多锯齿状的边缘，因为 shadow map 的解析度是固定的，多个 fragment 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。
可以采取 PCF 方法，核心思想是从 shadow map 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。最简单就是直接从旁边 3*3 小邻域取平均。

```
// PCF，在 3*3 邻域取平均值
    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;
```

虽然近看还是有锯齿的，但是远一点看由于这个模糊的效果看上去阴影的效果就柔和了一点，类似于处理走样的时候的区域采样的方法。