

Homework 6 作业报告 by 76-朱雨田

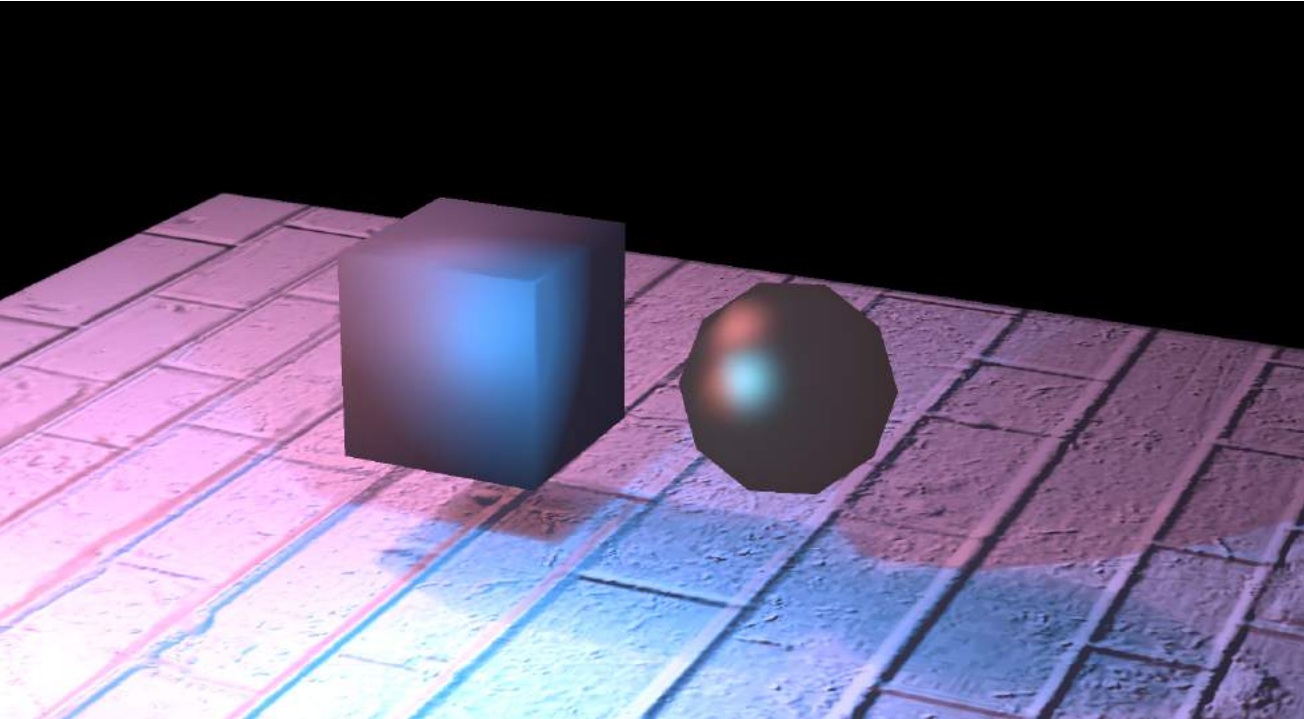
本次作业要求实现实现 Blinn-Phong 着色模型， Shadow Mapping 算法，以及 PCSS 和 SSAO 的阴影效果优化算法。

报告中图片如未特别标注，则都经过了 Gamma 校正以得到更好的视觉效果。

以下是部分文件的说明：

文件	说明
blinn_phong.fs	不带 ShadowMapping 的光照 Shader
blinn_phong_2.fs	带有朴素 ShadowMapping 的光照 Shader
blinn_phong_3.fs	带有 PCSS 的光照 Shader
pbr_metal_roughness.fs	仿照 Unity 中的金属度/粗糙度工作流制作的光照 Shader，但似乎并不成功
node_add_light.cpp	对 AddLight 节点中的颜色范围、半径范围做了调整，以更适合我这次实现的结果

以下是展示效果，这两张图片都启用了 PCSS，SSAO 以及 Gamma 校正：





上图 Sponza 模型中两条较硬的阴影是光源视角导致的。

1. Blinn-Phong 着色模型

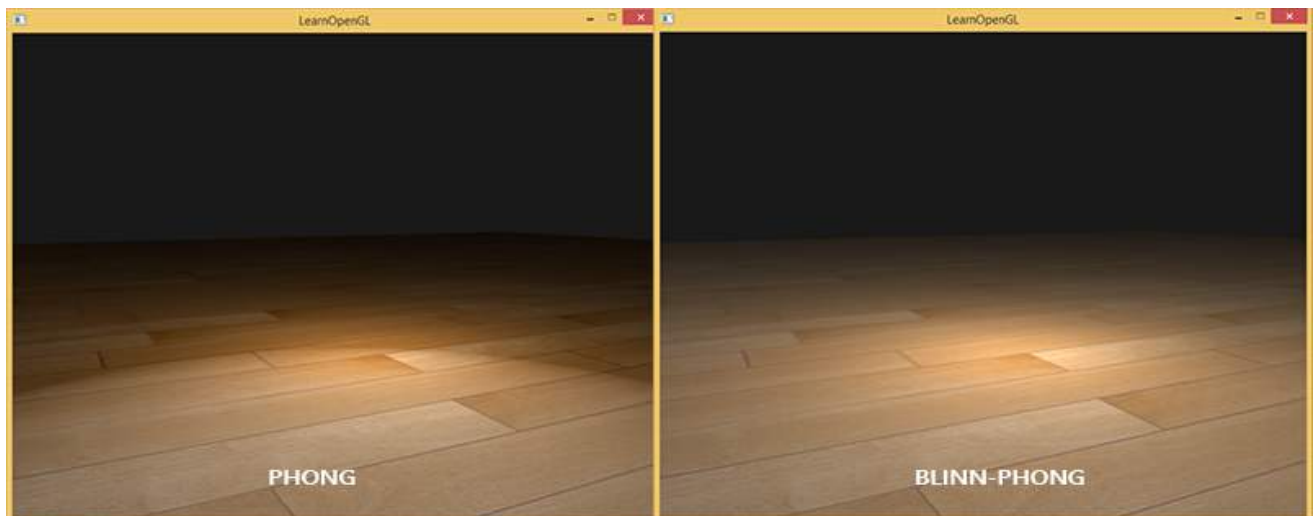
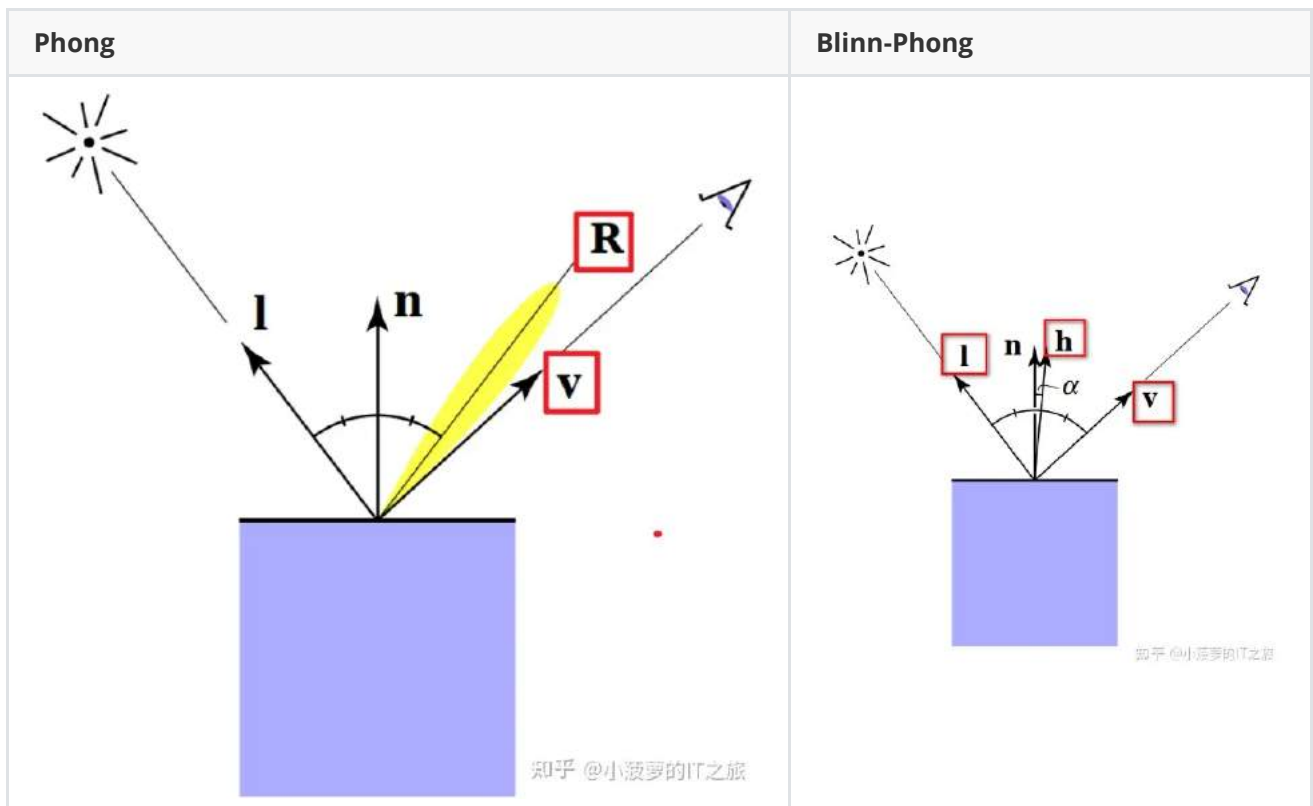
`blinn-phong.fs` 是实现了不含阴影映射的 Blinn-Phong 着色模型的 Shader。

Blinn-Phong

未引入金属度/粗糙度的 Blinn-Phong 着色模型关键代码如下：

```
1  vec3 lightDir = normalize(lights[i].position - frag_pos);
2  vec3 viewDir = normalize(iCameraPos - frag_pos);
3  vec3 halfwayDir = normalize(lightDir + viewDir);
4  float NdotL = max(dot(norm, lightDir), 0.0);
5  float NdotH = max(dot(norm, halfwayDir), 0.0);
6
7  vec3 ambient = lights[i].color * 0.05 * diff_color;
8  vec3 diff = lights[i].color * NdotL * diff_color;
9  vec3 spec = lights[i].color * pow(NdotH, 32.0);
10
11 result += (ambient + diff + spec) / dist_sq;
```

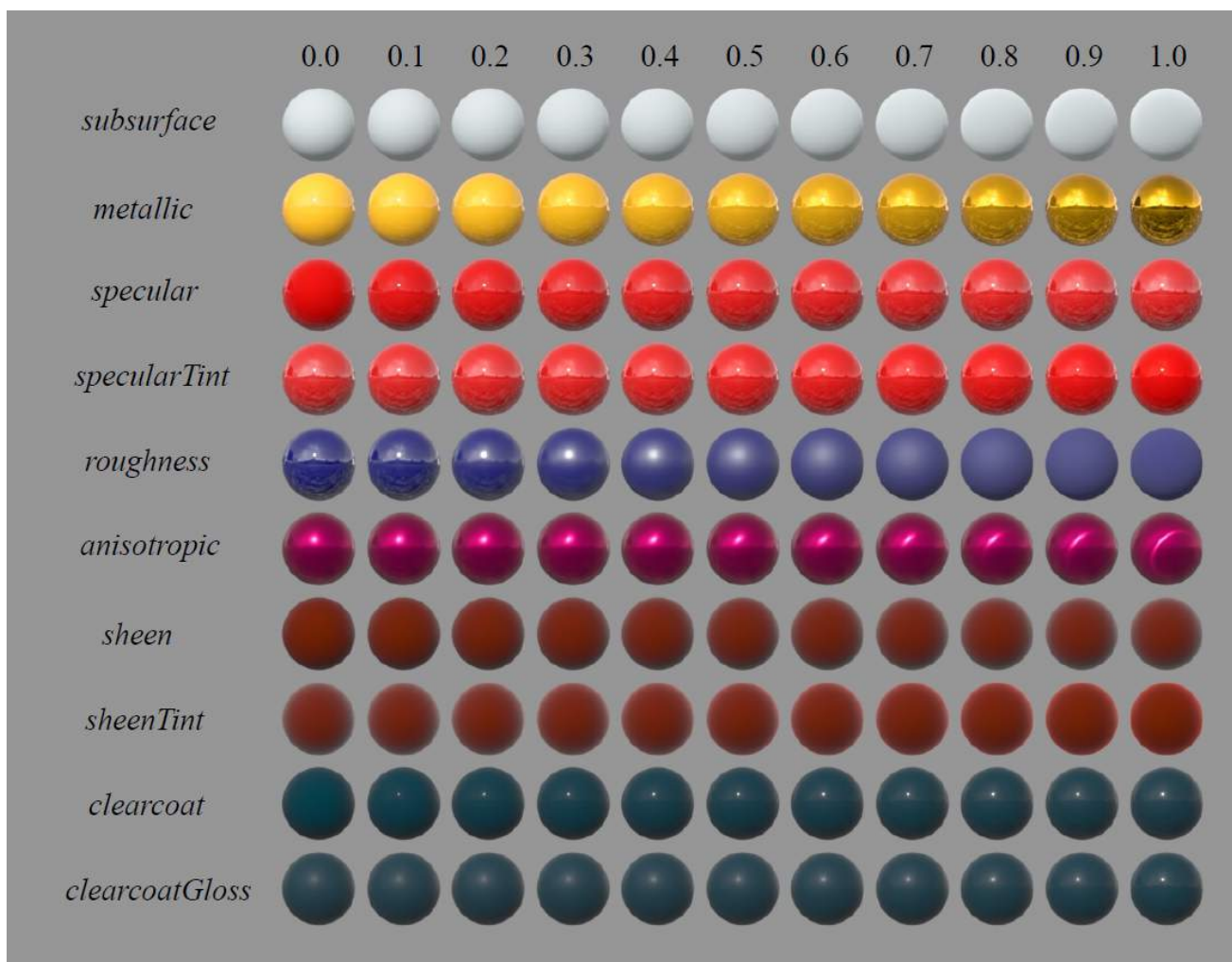
Blinn-Phong 和 Phong 模型都是对现实世界光照的一种简化建模。Blinn-Phong 模型相对 Phong 模型的变化是，将计算镜面反射项的底数从 $\vec{R} \cdot \vec{r}$ 改为了 $\vec{n} \cdot \vec{h}$ ，如下图（摘自 [入门Shading，详解Blinn-Phong和Phong光照模型 - 知乎\(zhihu.com\)](#)）。这不仅减少了计算机的计算量，还能够改善在某些角度下不自然的光照效果。



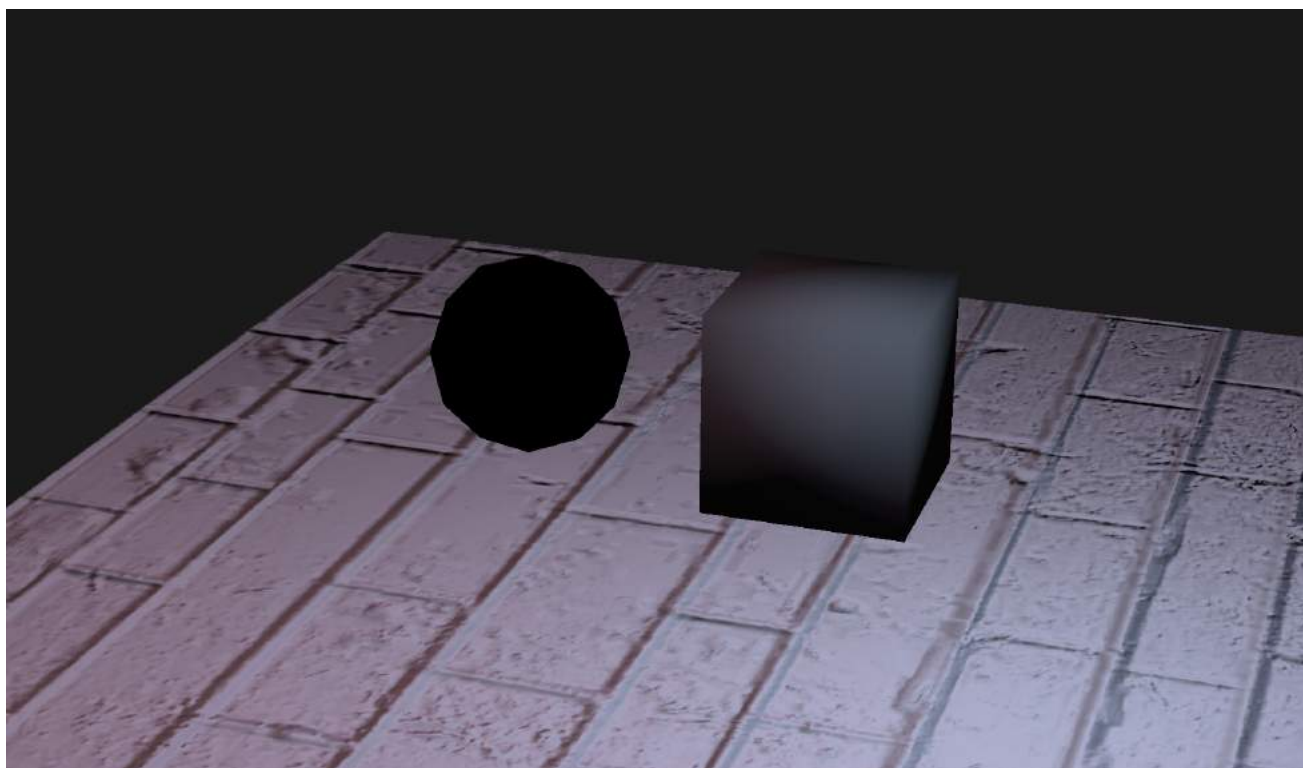
代码中还引入了光照的平方反比项，使得光照效果更为自然。虽然这会使得图像偏暗，但后文的 Gamma 校正能很好地使图片重新变亮。

金属度与粗糙度

在 Unity 和 Blender 中调整材质的时候，我们就能接触到金属度、粗糙度等概念。这些概念来自于 2012 年 Disney 提出的 BRDF 原则，允许美术用易于调整的参数控制材质以达到理想的美术效果。关于这些可以从毛星云老师的知乎文章中了解：[【基于物理的渲染（PBR）白皮书】（一）开篇：PBR核心知识体系总结与概览 - 知乎\(zhihu.com\)](https://zhuanlan.zhihu.com/p/100000000)



对于不同的渲染模型，我们有不同的近似实现以上参数的方法。在助教更新文档前，我仿照知乎上介绍 Unity 中实现的方法实现了一个自己的金属度/粗糙度模型，但最终效果并不理想，并且似乎改着改着就和 Blinn-Phong 关系不大了？如图，金属度为 1 的球体在下图中被错误地渲染成了纯黑色。



这是实现以上效果的参考资料：

[PBR中的金属度和粗糙度以及BRDF中的FDG项 - 知乎\(zhihu.com\)](#)

[【unity】Lit.shader解读 - 知乎\(zhihu.com\)](#)

以上实现在了 `pbr_metal_roughness.fs` 中。虽然没有得到正确的效果，但我仍然保留了这个 shader 文件，也许以后有空可以把它改的更合理一些？

助教更新文档后，我参考 [glTF PBR Example\(kcoley.github.io\)](#) 中的代码，实现了 Blinn-Phong 中的金属度-粗糙度模型。这是网页 Javascript 源代码中的转换：

```
1  const dielectricSpecular = new THREE.Color(0.04, 0.04, 0.04);
2      const epsilon = 1e-6;
3
4      PbrUtilities.DielectricSpecular = dielectricSpecular;
5
6      PbrUtilities.ConvertToSpecularGlossiness = function (metallicRoughness) {
7          var baseColor = metallicRoughness.baseColor;
8          var opacity = metallicRoughness.opacity;
9          var metallic = metallicRoughness.metallic;
10         var roughness = metallicRoughness.roughness;
11
12         var specular = dielectricSpecular.clone().lerp(baseColor, metallic);
13
14         var oneMinusSpecularStrength = 1 - specular.getMaxComponent();
15         var diffuse = baseColor.clone().multiplyScalar((1 -
dielectricSpecular.r) * (1 - metallic) / Math.max(oneMinusSpecularStrength,
epsilon));
16
17         var glossiness = 1 - roughness;
18
19         return {
20             specular: specular,
21             opacity: opacity,
22             diffuse: diffuse,
23             glossiness: glossiness
24         };
25     }
```

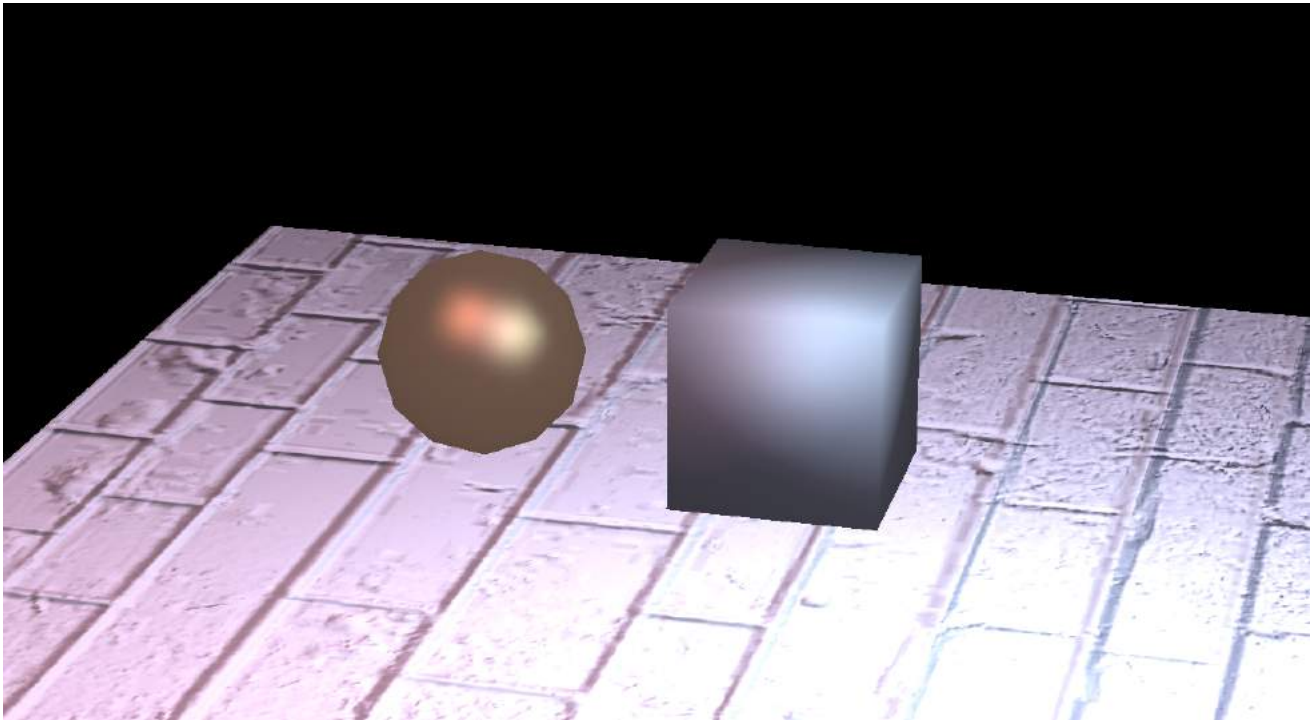
翻译为 GLSL 如下：

```
1  const vec3 dielectricSpecular = vec3(0.04, 0.04, 0.04);
2  const float epsilon = 1e-6;
3  vec3 specular = mix(dielectricSpecular, diff_color, metal);
4  vec3 oneMinusSpecularStrength = 1.0 - specular;
5  vec3 diffuse = diff_color * (1 - dielectricSpecular) * (1 - metal) /
max(oneMinusSpecularStrength, epsilon);
6  float glossiness = 1 - roughness;
```

因为不考虑透明度，我们默认了 Opacity 项为 1。以下是参数引入 Blinn-Phong 算法后的结果：

```
1 vec3 diff = lights[i].color * NdotL * diffuse;
2 vec3 spec = specular * pow(NdotH, glossiness * 31.0 + 1.0) * lights[i].color;
```

效果如下，可以看到小球很好的金属质感：



图中是修改过的 `box_on_plane2.usda` 模型。我通过修改源码对模型做了以下修改：

- 翻转地面的法线
- 给地面的材质赋予了一张砖纹法线贴图（贴图来自 Sponza 模型）
- 设置地面的金属度、粗糙度均为 0
- 设置 Cube 的漫反射颜色为 (0.2, 0.2, 0.2)，金属度为 0.5，粗糙度为 0.8
- 设置球体的漫反射颜色为 (0.76, 0.5, 0.25)，金属度为 1，粗糙度为 0.1

在以上的修改过程中，我也学到了不少 usda 的语法。

不过感觉金属质感更多的是来源于对环境的镜面反射，这在局部光照中是无法很好地做到的。

法线贴图

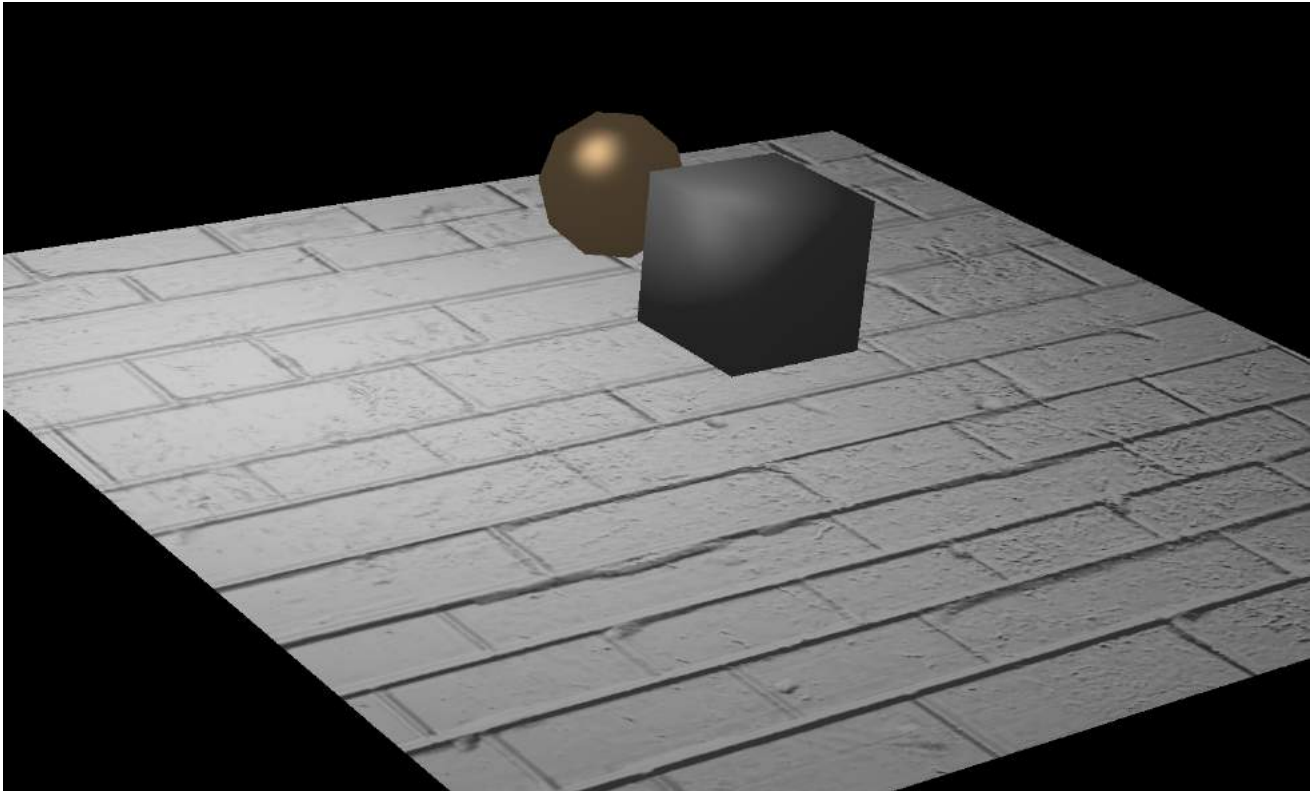
法线贴图是一种用较低的成本为材质增添光照纹理的方法。

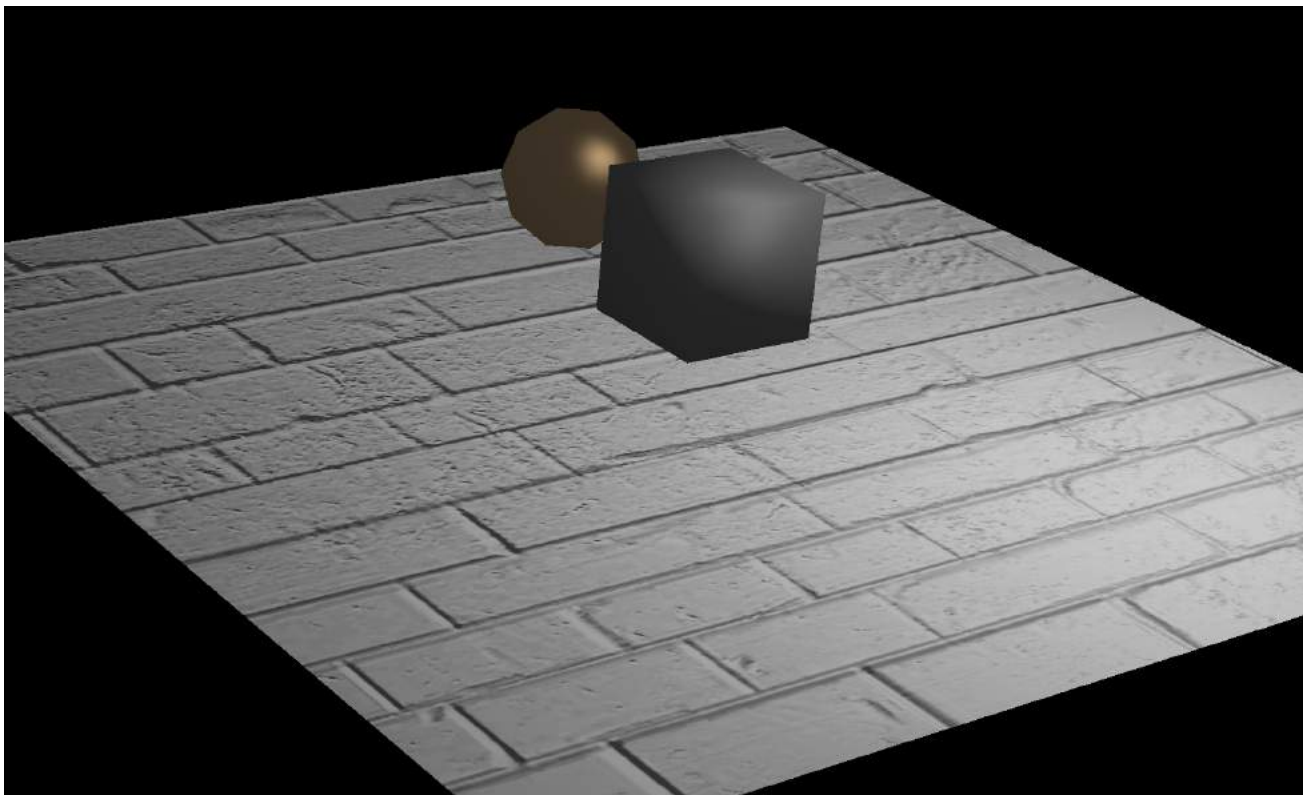
法线贴图使用缩放至 $[0, 1)$ 的 RGB 色彩分量表示材质在每个点处的法向。为了便于美术专注于材质表面的纹理，也为了法线贴图在不同物体上适用，法线贴图的色彩分量表示的法线是定义在切线空间内的，即 z 分量（或者缩放后的蓝色分量）总是指向法线的。如果知道切线方向，我们只需要将一条切线、垂直的另一条切线、法线作为新的基，对世界坐标进行变换即可得到切线空间下的法线。这一变换矩阵就是 TBN 矩阵。

在我们的代码中，只需在传入 normal 值前对 normal 做以下变换，即可启用法线贴图：

```
1 vec3 normalmap_value = texture(normalMapSampler, vTexCoord).xyz;
2 // 将 [0, 1) 的颜色值映射为 [-1, 1] 的法线向量坐标值
3 vec3 norm = normalmap_value * 2.0 - 1.0;
4 // ...
5 // 计算tangent, bitangent 的部分
6 mat3 TBN = mat3(tangent, bitangent, normal);
7 normal = normalize(TBN * norm);
```

在地板上贴上一张法线贴图，我们可以很清晰地看到光照位置对地板光照的影响：





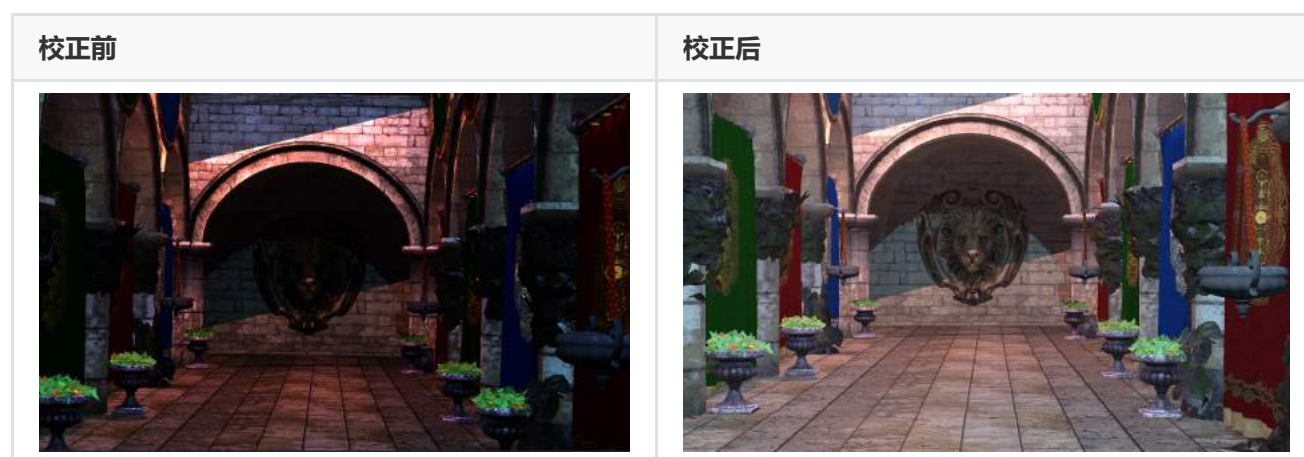
Gamma 校正

[Gamma校正 - LearnOpenGL CN \(learnopengl-cn.github.io\)](https://learnopengl-cn.github.io)

Gamma 校正能很好地改善画面的效果。其只需在 Fragment Shader 的最后加上简单的代码：

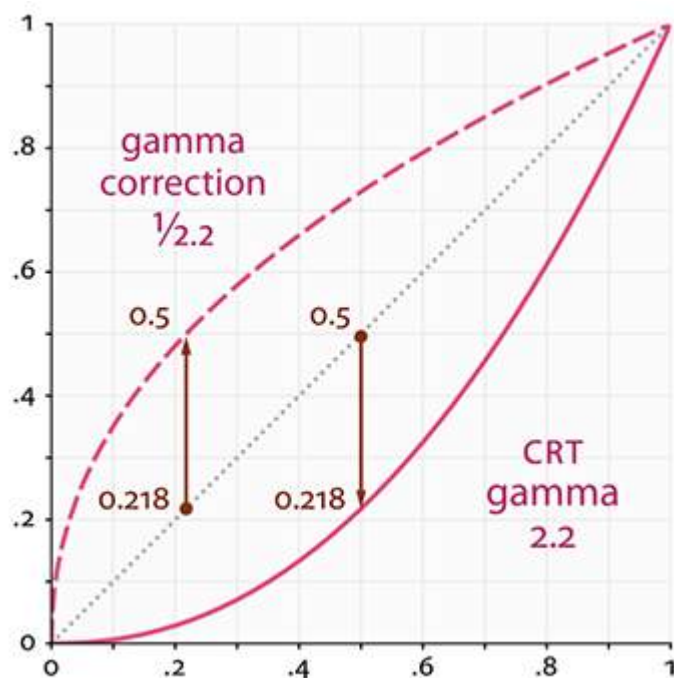
```
1 // Gamma correction
2 float gamma = 2.2;
3 color.rgb = pow(color.rgb, vec3(1.0 / gamma));
```

以下是效果对比：



其原理是用一条曲线调整了亮度，使本来用能量计算得出的颜色更接近人眼的观感。

Perceived (linear) brightness =	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Physical (linear) brightness =	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0



2. 阴影映射

含阴影映射的 Shader 在 `blinn_phong_2.fs` 中实现。

Shadow Mapping，就是以光源的视角对物体计算深度图，得到每个光源距离最近的点集，从而在着色步骤中将不属于这个点集中的点设为未被照到，实现阴影的效果。

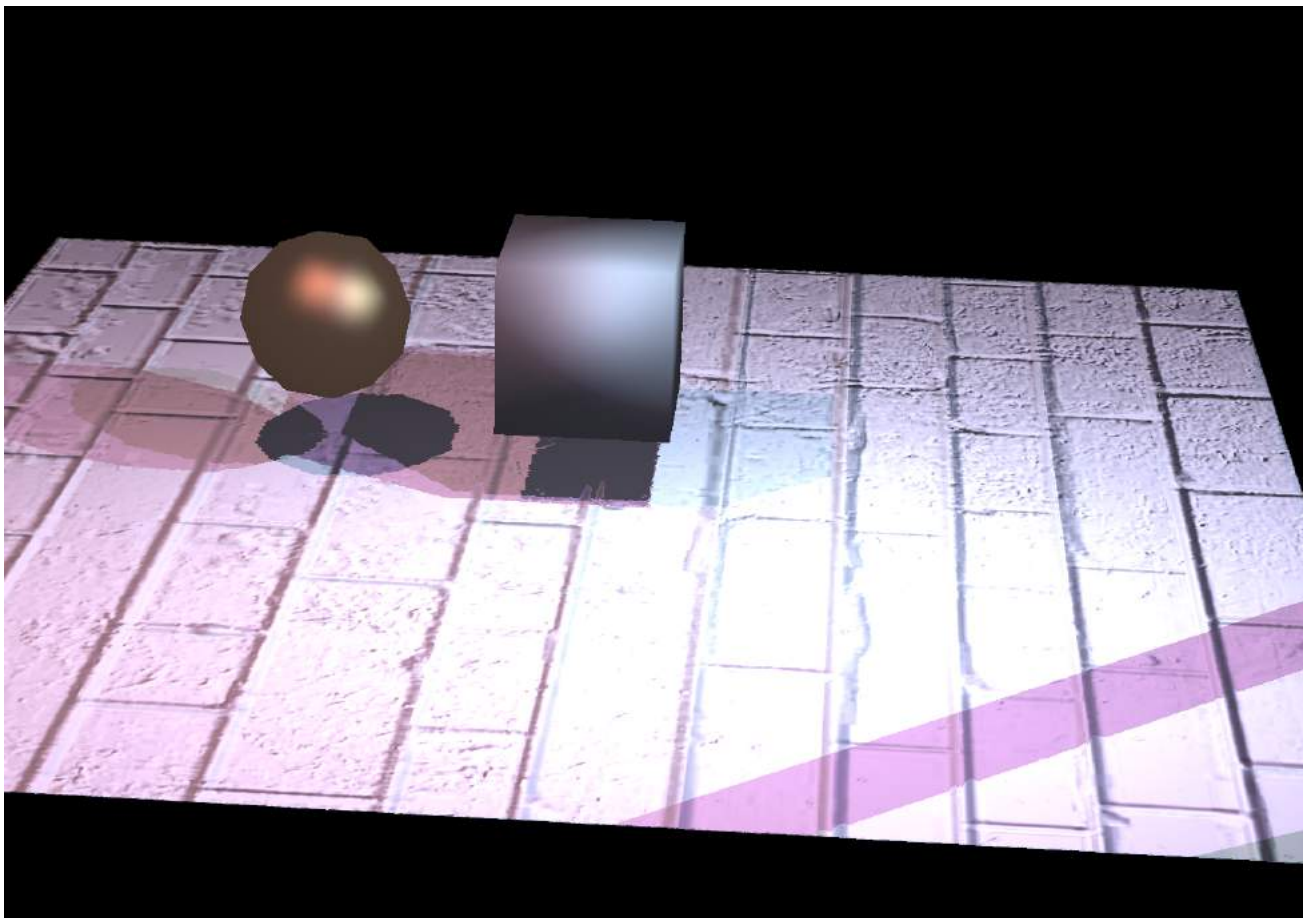
要验证一个位置是否属于离光源最近的点集，只需要在着色器中将该位置变换到该光源对应的阴影贴图的位置中，并比较深度即可。若深度大于阴影贴图中记录的最近深度，则视为阴影处理。

计算阴影贴图的代码，助教已经帮我们写好了。我将光源的视角调整为了 135° ，并将代码复制到了 Deferred Lighting 节点中，向 Shader 传入每个光源的阴影贴图与变换矩阵。以下是一个光源的阴影贴图：



这是计算阴影值的代码，也做了教程中的 bias 校正以避免摩尔纹。这里将阴影的颜色值全部设置为了 0，非阴影全部设置为了 1，这会导致一些锯齿的问题。

```
1 float ShadowCalc(vec4 frag_pos_light_space, int light_index, float NdotL)
2 {
3     const float bias = max(0.05 * (1.0 - NdotL), 0.005);
4     vec3 proj_coords = frag_pos_light_space.xyz / frag_pos_light_space.w;
5     proj_coords = proj_coords * 0.5 + 0.5;
6     float closest_depth = texture(shadow_maps, vec3(proj_coords.xy,
lights[light_index].shadow_map_id)).r;
7     float current_depth = frag_pos_light_space.z / frag_pos_light_space.w;
8     float shadow = current_depth - bias > closest_depth ? 1.0 : 0.0;
9     return shadow;
10 }
```



上图展示了 Shadow Mapping 的效果。右下角不正确的阴影是光源的视口导致的。调大光源的视角可以减小该阴影，但会导致物体阴影的锯齿增大（因为物体对应的阴影分辨率减小了）。

如果要实现不受视口限制的点光源阴影，就需要从六个角度都计算一遍阴影映射贴图。LearnOpenGL CN 中给出了介绍：[点阴影 - LearnOpenGL CN \(learnopengl-cn.github.io\)](https://learnopengl-cn.github.io/)

3. SSAO

在一些美术课程中，我们会接触到“闭塞”这一概念。闭塞是指任何角度的光线都很难到达的夹缝，这些区域对人眼来说也是判断物体形状的重要标准。LearnOpenGL CN 中对闭塞的表述是：这些区域很大程度上是被周围的几何体遮蔽的，光线会很难流失，所以这些地方看起来会更暗一些。

下图是抖抖村的插画课程中对“闭塞”的解释，与画出物体的轮廓和闭塞后的效果。可以看到，如果表达出了画面中闭塞的部分，就算不刻画其余地方的光影、颜色，也能非常好地表达物体的形状。因此闭塞对人眼来说是很重要的画面组成部分。



课程目录	
00	课程群
01	视频课 36分6秒
13	第8关-如何把眼睛画成“尺子”
02	视频课 8分27秒
14	第9关-靠比例临摹法，降维打击透视
03	视频课 7分50秒
15	第10关-靠角度临摹法，画仰视的福瑞小哥
04	视频课 9分54秒
16	第11关-丢掉“恶心”的辅助线，画泳装小姐姐
05	视频课 8分42秒
17	第12关-轮廓线条表现立体感
06	视频课 10分25秒
18	第13关-闭塞，提升画手的观察力
07	视频课 12分46秒
19	第14关-如何继续加入“亿点细节”
08	视频课 10分44秒
20	【奖励关卡】第14关课后原速跟画-速写机械潮玩
09	视频课 1小时18分
21	第15关-光影明暗练习法，暴增板绘实力
10	视频课 15分39秒



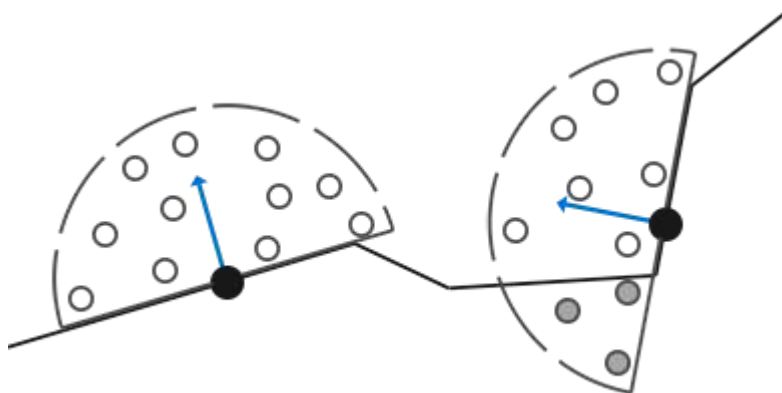
课程目录	
00	课程群
01	视频课 36分6秒
13	第8关-如何把眼睛画成“尺子”
02	视频课 8分27秒
14	第9关-靠比例临摹法，降维打击透视
03	视频课 7分50秒
15	第10关-靠角度临摹法，画仰视的福瑞小哥
04	视频课 9分54秒
16	第11关-丢掉“恶心”的辅助线，画泳装小姐姐
05	视频课 8分42秒
17	第12关-轮廓线条表现立体感
06	视频课 10分25秒
18	第13关-闭塞，提升画手的观察力
07	视频课 12分46秒
19	第14关-如何继续加入“亿点细节”
08	视频课 10分44秒
20	【奖励关卡】第14关课后原速跟画-速写机械潮玩
09	视频课 1小时18分
21	第15关-光影明暗练习法，暴增板绘实力
10	视频课 15分39秒

从原理上讲，局部光照模型并不能计算出光线难以达到的夹缝中的闭塞效果。因此，我们需要用一些 Trick 来模拟闭塞的光照效果，从而让画面更容易被人眼接受。这就是**环境光遮蔽**（AO）。

以下的 SSAO 算法实现主要参考了：[SSAO - LearnOpenGL CN \(learnopengl-cn.github.io\)](https://learnopengl-cn.github.io/)

SSAO 是一种在屏幕空间中进行的环境光遮蔽。要计算 SSAO，只需要传入片元中每个顶点中的位置、法线、深度等 G-Buffer 中已经计算好的信息，以及到相机坐标的变换矩阵。因此，SSAO 不需要知道场景本身的模型信息，从这个角度来讲，SSAO 是一种图像的后处理。

SSAO 的原理是计算每个片元对应模型中点的周围半球区域中，可见区域占整个区域的比例，并将该比例低的区域压暗。如下图，右侧的半球体中因为有不可见区域，所以会被压暗：



要数值计算这一比例，自然的方法是对半球体区域进行采样。LearnOpenGL CN 中的做法是传入一系列切线坐标下的采样方向贴图，在片元着色器中再计算具体的采样点。这个方法有一些麻烦，我这里的采样方向是在片元着色器中用 Hash 函数实时计算的，代码如下：

```
1 vec3 dir = hash33_sphere(vec3(uv, i));
2 if(dot(dir, normal) < 0.0) dir = -dir;
3 vec3 samp = frag_pos + dir * radius;
```

要采样半球面区域，只需要将与法线点积小于 0 的方向反转即可，不需要像资料里一样做切线空间的变换。

`hash33_sphere` 是我手动实现的球体采样函数，在单位球中进行随机的均匀采样。（似乎不均匀的采样效果会更好？将 `pow` 指数改为 1，能做到越靠近原点，采样越多的效果）。代码如下：

```
1 //copied and adjusted from https://zhuanlan.zhihu.com/p/599263679
2 vec3 hash33(vec3 p3)
3 {
4     p3 = fract(p3 * vec3(1.4031, 2.1060, 2.0973) * vec3(50.0));
5     p3 += dot(p3, p3.yxz + 36.33);
6     return fract((p3.xxy + p3.yxx)*p3.zyx);
7 }
8
9 vec3 hash33_sphere(vec3 p3)
10 {
11     vec3 hash = hash33(p3);
12     hash.xy *= 2 * PI;
13     return vec3(vec2(cos(hash.x), sin(hash.x)) * sin(hash.y), cos(hash.y))
14                 * pow(hash.z, 0.333);
15 }
```

在确定了采样点在三维空间中的位置后，我们就能得到这个点的实际深度与对应的贴图位置（从而能确定相机看到的最远深度）了。确定采样点是否能被相机看到的代码如下：

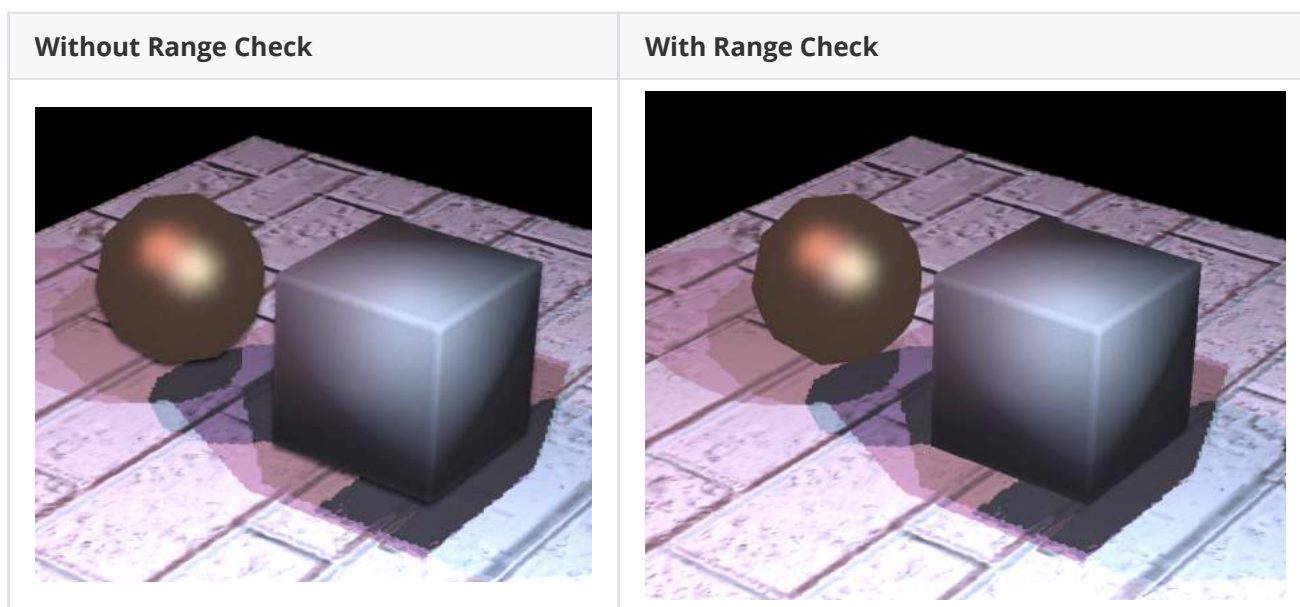
```
1 vec4 samp_pos_clip = projection * view * vec4(samp, 1.0);
2 vec3 samp_pos_camera_space = samp_pos_clip.xyz / samp_pos_clip.w;
3 float samp_depth = samp_pos_camera_space.z;
4 vec2 samp_uv = samp_pos_camera_space.xy * 0.5 + 0.5;
5 vec3 samp_pos = texture(positionSampler, samp_uv).xyz;
6
7 // 剔除背景
```

```

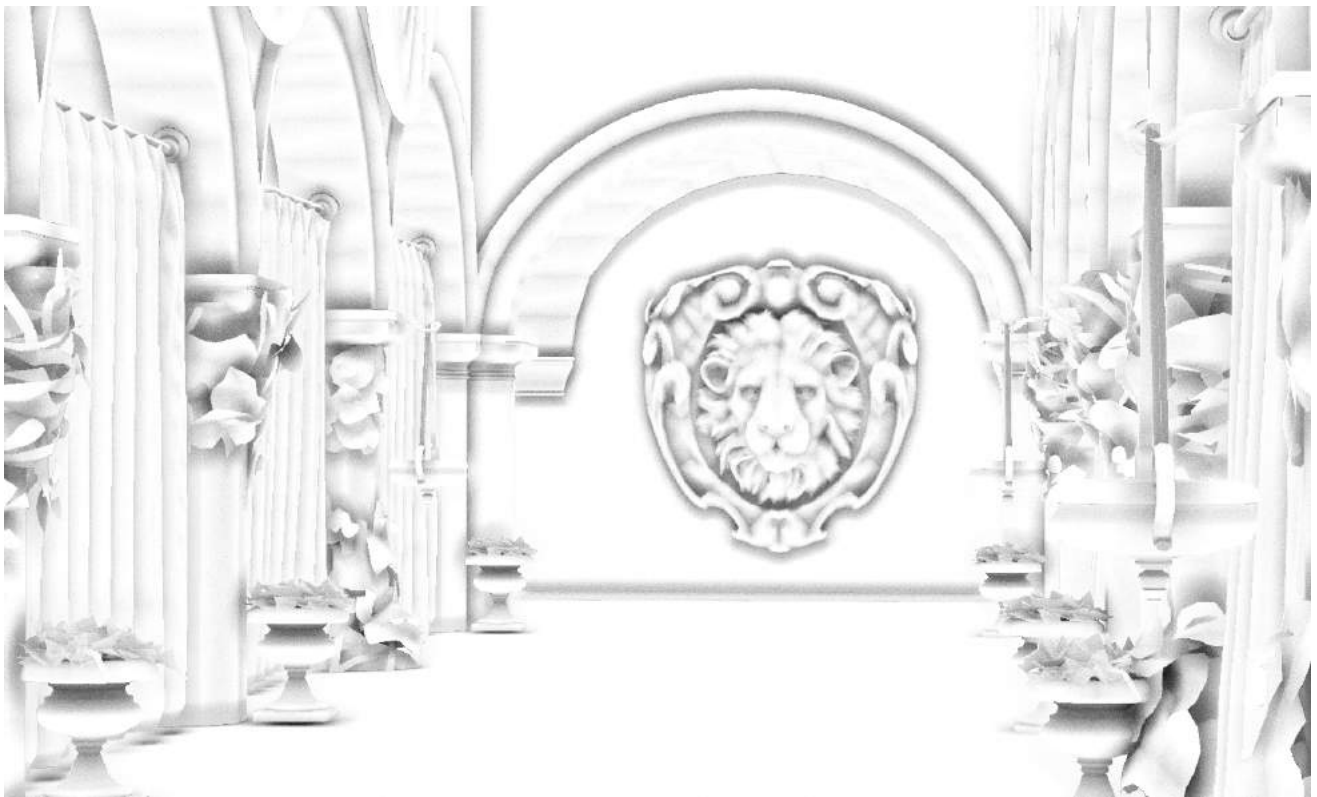
8  vec3 frag_norm = texture(normalSampler, samp_uv).rgb;
9  if(len(frag_norm) < 0.1)
10     continue;
11
12  float frag_depth = texture(depthSampler, samp_uv).r;
13
14  float rangeCheck = smoothstep(0.0, 1.0, radius / (len(frag_pos - samp_pos)));
15  occlusion += (samp_depth > frag_depth ? 1.0 : 0.0) * rangeCheck;

```

这里还实现了 RangeCheck，但实现的方式与 LearnOpenGL CN 中有所不同。（不知道为什么那个实现用处不大）。这是因为正常物体的边缘也会被 SSAO 判定为闭塞，而这会导致以下的错误效果，几何体的边缘出现了不理想的黑边：





生成的 SSAO 贴图如下（radius = 0.100, 采样数为 128, 均匀球体采样）：



将 SSAO 贴图与计算出来的光照颜色相乘，即可得到带有闭塞的画面。这里的画面是将 SSAO 贴图也做了 Gamma 校正后相乘得到的。



以下是 SSAO 前后的对比，可以更明显地看到 SSAO 消除了狮子浮雕的“悬浮”效果，从而优化了观感。

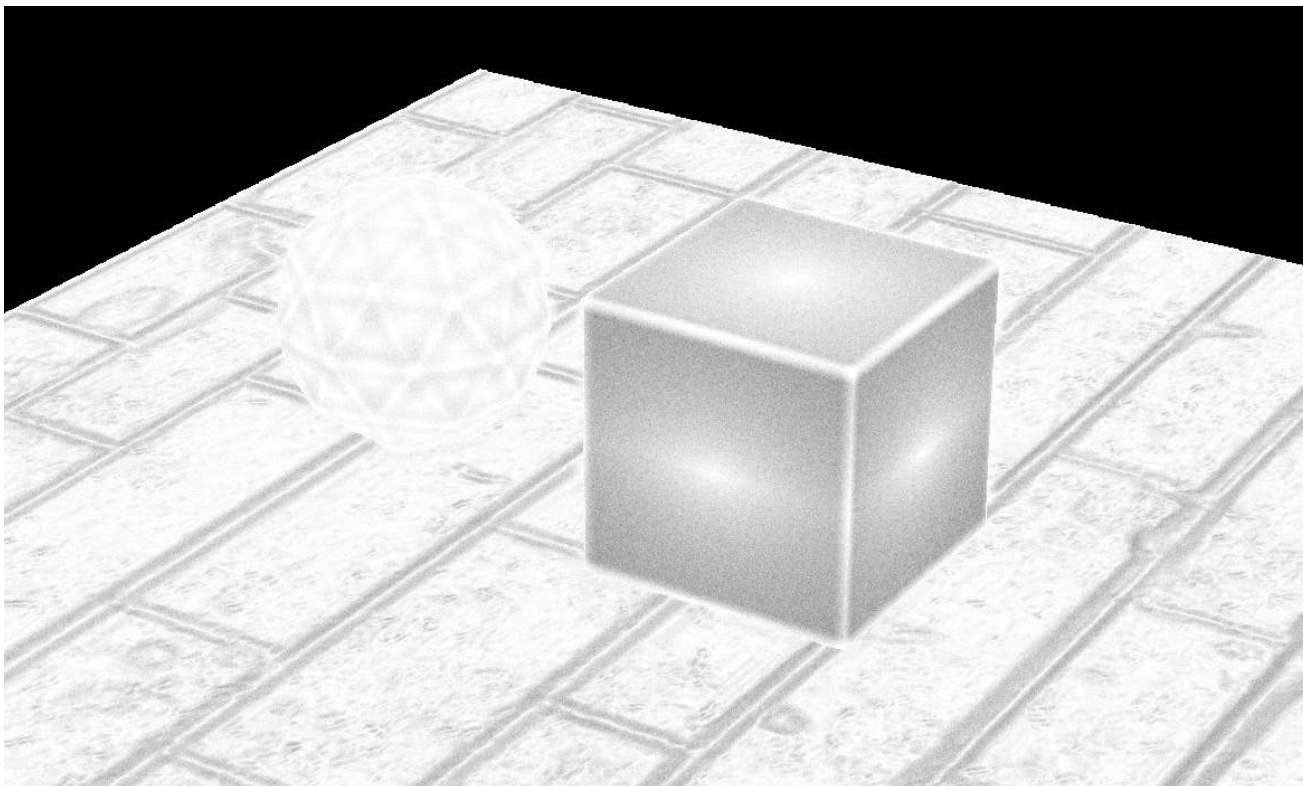
Without SSAO	With SSAO
	

目前实现的 SSAO 会因为采样不足而存在噪点问题。因为随机采样的依据是屏幕空间中的坐标，所以噪点会“蒙”在画面上，对于一些游戏来说是不太好的画面效果。（虽然也见过刻意为画面增添风格化噪点的画师和摄影师...）以下是降低采样率后的噪点效果（采样率为 8）：



这实际上可以通过再用一个 Pass 来低通滤波降噪处理。但再写一个 Pass 需要新建一个节点，这里没有再这样做了（躺）。

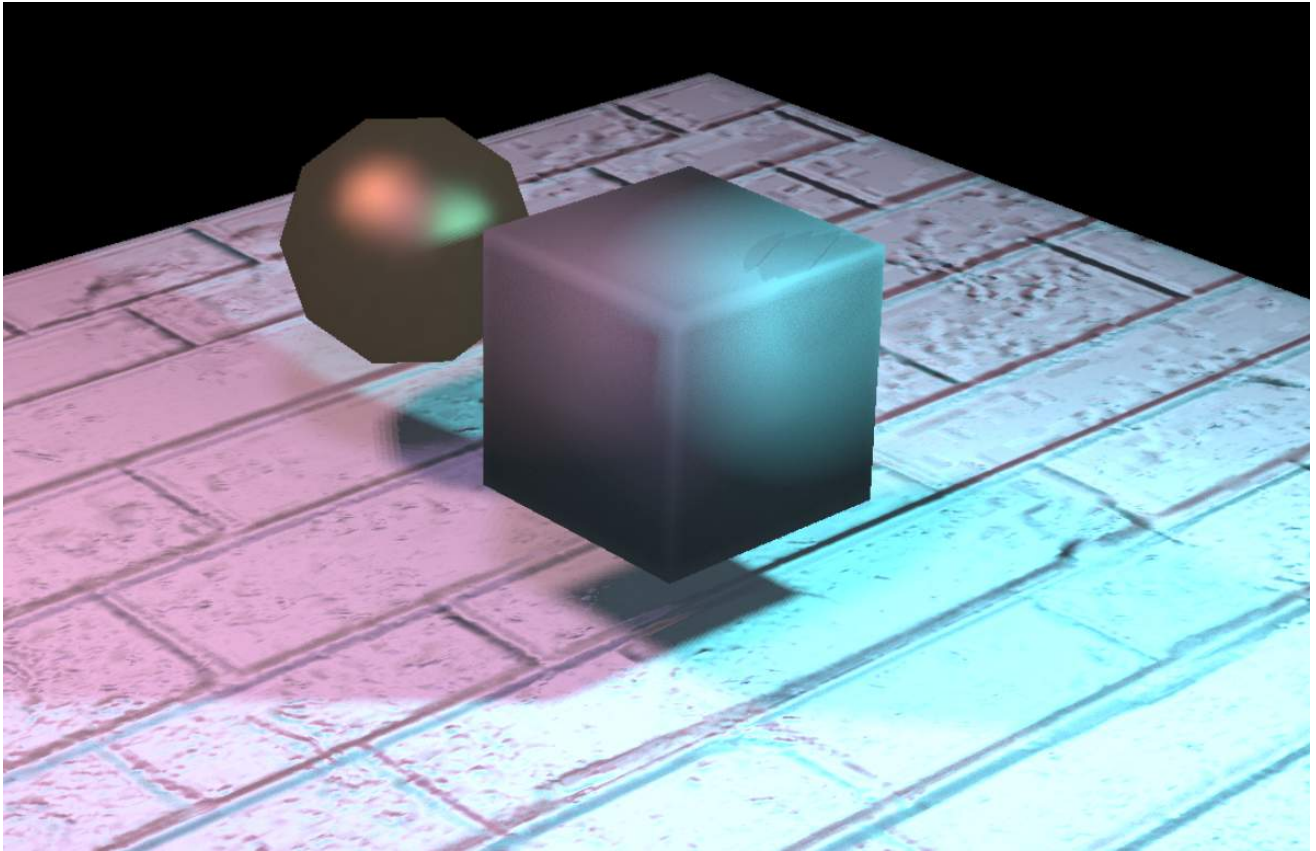
另一个问题是，Box on Plane 模型中似乎会因为立方体的法线设置问题而导致立方体表面出现不该出现的 SSAO，如下图。不过这似乎是模型本身对法线插值造成的问题？



在实现 SSAO 的过程中要向 Shader 中传入不少新的纹理，这些都要在 C++ 代码中实现。这又是一个学习 OpenGL 的机会。

4. PCSS

为了消除 Shadow Map 分辨率不足造成的阴影锯齿，我们要对阴影做一些处理来避免锯齿的产生。下图是 PCF 方法的结果：

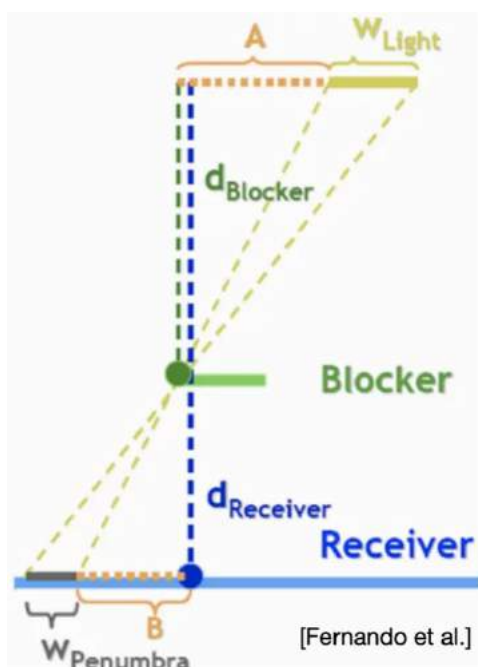


PCF 的思想，是用片元的深度对光照贴图中邻域的深度都比较一次后，对“有无阴影”做平均。如上图，PCF 能得到不错的抗锯齿效果。为了有效地对邻域深度求平均，我们需要将阴影贴图的分辨率也传入 Shader，从而将邻域设置为周边的像素。

但是，实际生活中的软阴影并不是处处均匀的。资料中给出了钢笔尖的例子，如下图：



距离钢笔尖越近的区域，阴影越硬；越远的区域，阴影越粗糙。这是因为光源本身是有面积的，如下图：

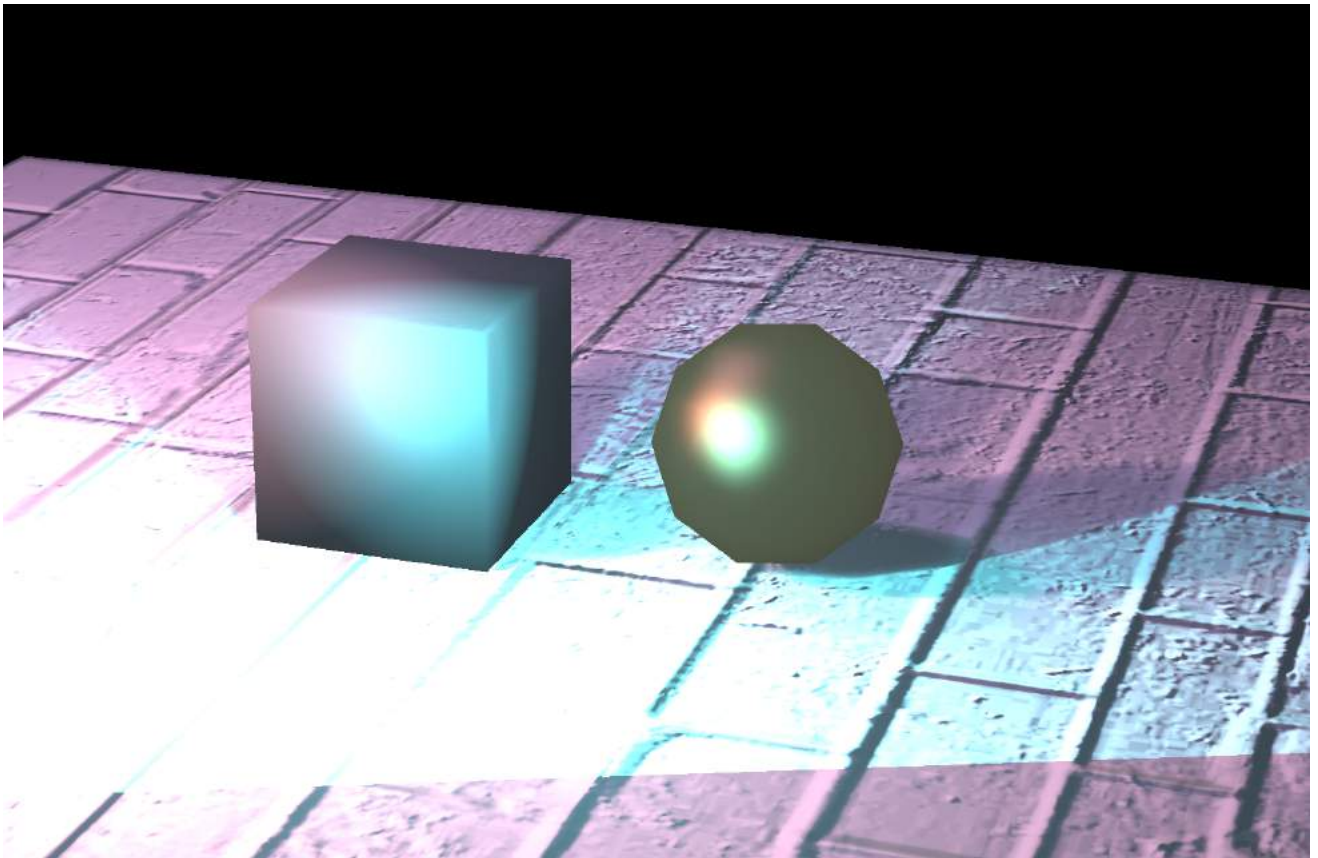
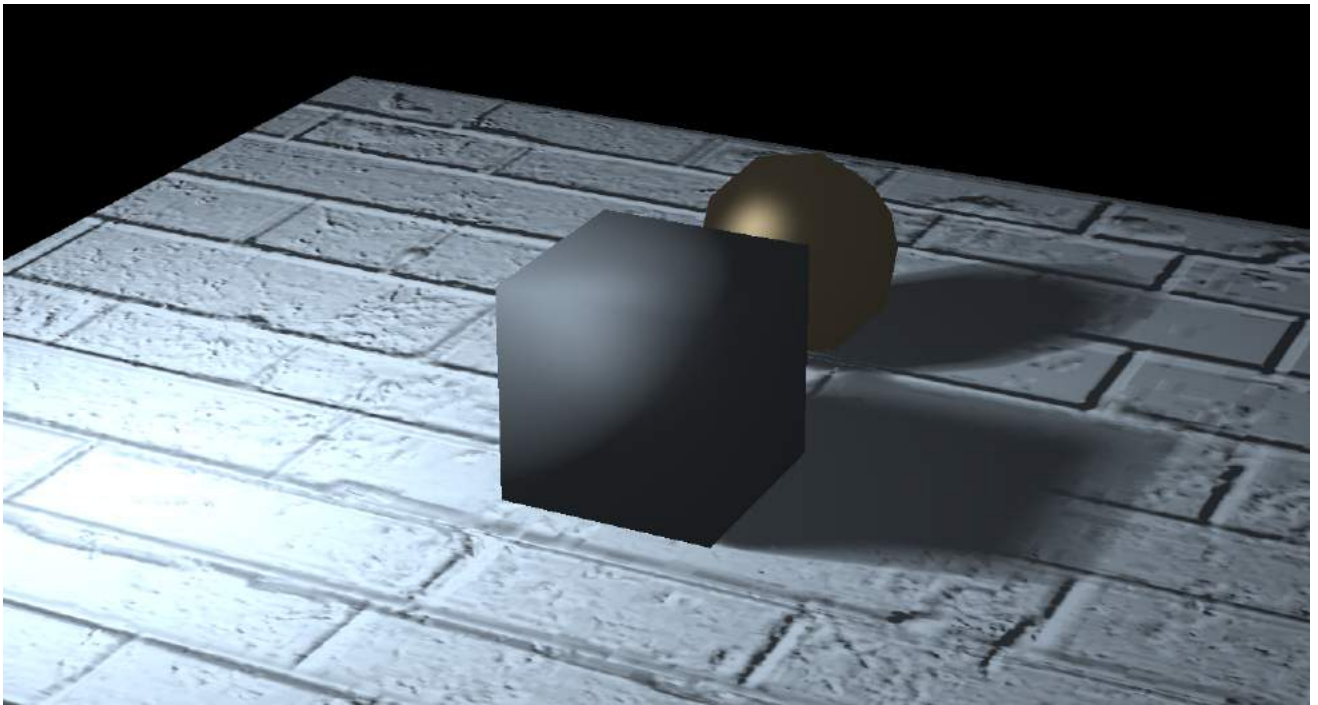


$$W_{Penumbra} = (d_{Receiver} - d_{Blocker}) \cdot W_{Light} / d_{Blocker}$$

知乎 @kakaroto

为了模拟光源面积对阴影软硬产生的影响，PCSS 算法提出利用相似三角形计算所需邻域的大小。为了求出这里的 $d_{Blocker}$ ，我们需要先遍历一次邻域的阴影贴图，将深度小于片元的所有点的深度做平均，以作为遮挡物的深度；从而，我们可以用相似三角形原理从已知的 W_{Light} 计算得到采样邻域面积 $W_{Penumbra}$ 。

以下是我实现的 PCSS 结果：





目前版本的 PCSS 还有一些没写太明白的地方。比如计算采样邻域阴影的代码：

```
1 float w = (current_depth - blocker) / blocker * lights[light_index].radius;
2
3 float shadow_sum = 0.0;
4
5 for(int i = -sample_range_shadow; i <= sample_range_shadow; i++)
6 {
7     for(int j = -sample_range_shadow; j <= sample_range_shadow; j++)
8     {
9         vec2 offset = vec2(i, j) / float(sample_range_shadow) * w;
10        float sample_depth = texture(shadow_maps, vec3(proj_coords.xy + offset,
11        lights[light_index].shadow_map_id)).r;
12        if(current_depth - bias > sample_depth)
13        {
14            shadow_sum += 1.0;
15        }
16    }
```

可以看到 Lights 的 Radius 并不是真实世界中的尺寸，而是反映了其大小在光照贴图上的占比。因此 Radius 值一般取在 (0, 0.1) 比较合适。目前还不太清楚怎样让 Radius 反映实际区域的大小。

同时，这里的采样做的也是一个正方形采样，而不是对圆形区域采样。这会导致阴影无法反映光源的球形形状。

这些问题在以后都可以改进，不过现在已经没有时间完成了（？）

心得体会

这是 USTC-CG 2024 中的第一个**全新**的作业。在 USTC-CG 2020 的 Homework8 中，我并没有完成选做的阴影贴图，所以这次作业的所有部分对我来说都是第一次做。不过好在之前有一些 Shader 基础，理解 GLSL 对我并不困难。

框架的一些 bug 耽误了不少时间。不过助教的反馈很积极，bug 的修改都很及时。如果以后能做充足的测试后再发布作业就好了！这样也许我就能不用赶着 ddl 写 Optional 了。

这次的实时渲染作业比起 2020 年的 Shader 作业，更加深入接触了一些实时渲染 Trick 的东西，很好玩。两个 Optional 似乎已经算是 GAMES202 的东西了，不过文档啃下来做起来也不难（比理解 ARAP 的数学原理简单不少（？））

做出了很酷的东西，图形学，好玩。