OpenGL 入门

陈柯

2022 年 7 月 28 日

1 目的

利用 UGL 框架实现：

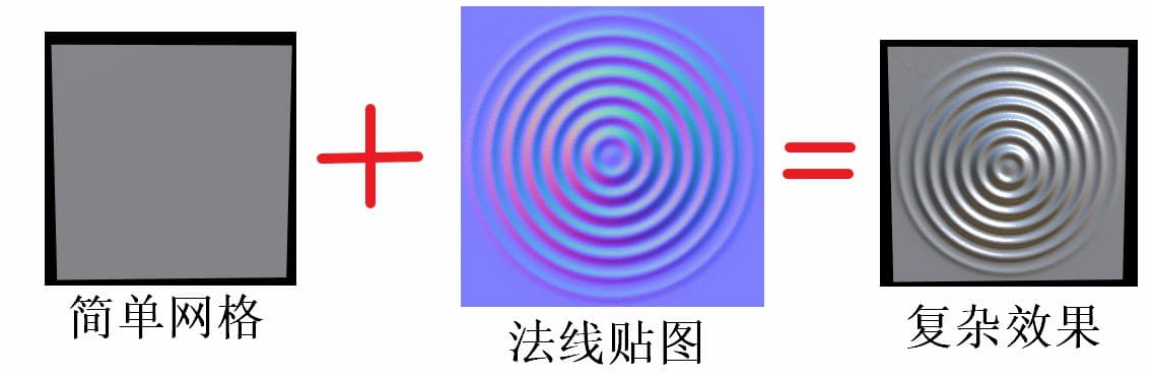
1. 实现 normal map 和 displacement map;
2. 利用 displacement map 在 vertex shader 中进行简单降噪;
3. 实现阴影映射

2 方法

2.1 法向贴图

典型使用方式如下：

图 1: 法向贴图原理

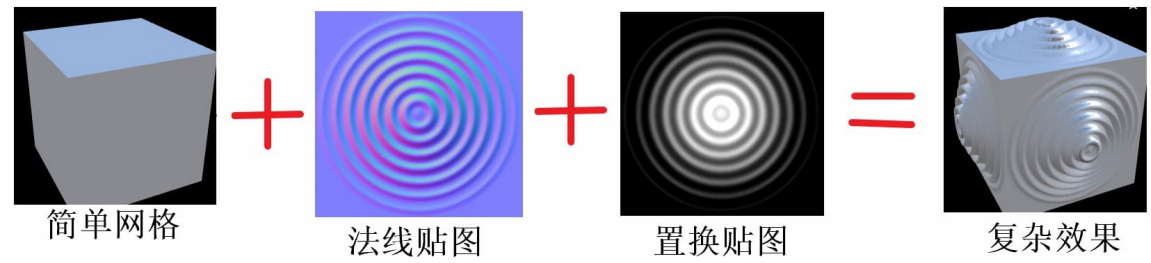


其中法线贴图在渲染中用于改变原法向，从而影响着色效果。法线贴图一般为蓝紫色，这是因为贴图中的法向是定义在切空间中的，上方向为 z 方向，对应于 RGB 的 B 通道。

2.2 置换贴图

典型使用方式如下：

图 2: 置换贴图原理



其中置换贴图用于改变顶点的位置，0 （黑色）表示不动，1（白色）表示沿着法向偏移。

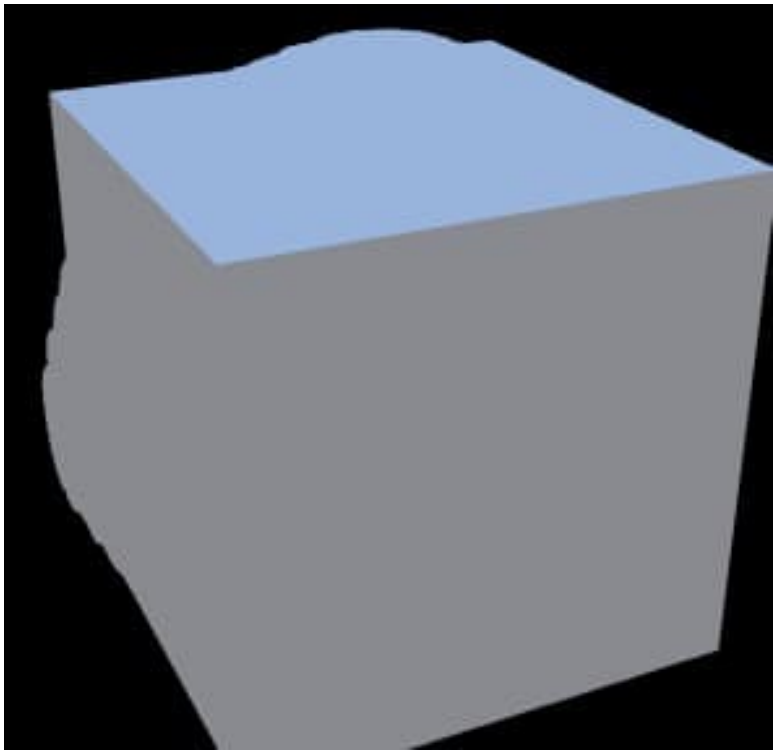
将置换贴图的值转换成顶点偏移量的方式是自定义的，示例如下：

- 0-0.5 为下沉，0.5-1.0 为凸起，变化量用一个系数决定
- 0-1.0 为凸起，变化量用一个系数决定
- 上边两种方式可范化成：displacement = lambda * (bias + scale * pixel value)
- displacement = lambda * (-1 + 2 * pixel value)
- displacement = lambda * (0 + 1 * pixel value)

由于要改变顶点坐标，在实时渲染中应在 vertex shader 中采样置换贴图来偏移顶点，因此简单网格应含有大量的内部顶点。

由于置换贴图只改变了顶点的位置，不改变顶点的法向，所以，如果不添加相应的法线贴图的话，渲染效果不太正确，如下：

图 3: 置换贴图不改变法线的情况

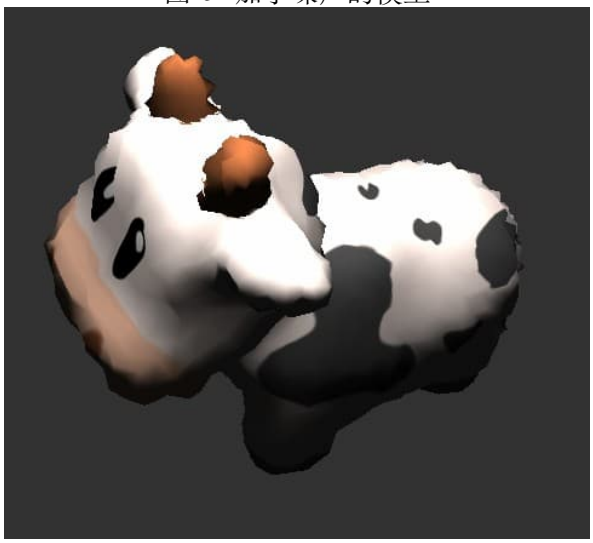


2.3 用置换贴图进行简单去噪

项目提供的模型和加了随机噪声之后的模型如下

图 4: 原模型

图 5: 加了噪声的模型



虽然给顶点加了噪声，但法线还是用了原本的，所以含噪声模型在渲染的不同主要体现在纹理的扭曲和边缘的凹凸不平上。我们只需将顶点进行合理的偏移就能达到不错的去噪效果。步骤如下：

1. 计算每个顶点的偏移量：

$$\delta_i = p_i - \frac{1}{|N(i)|}\sum_{j\in N(i)} p_j$$

2. 将偏移量投影到法向上：

$$\bar{\delta}_i = \langle \delta_i, \boldsymbol{n}_i \rangle \boldsymbol{n}_i$$

3. 对每一个顶点进行偏移：

$$\bar{p}_i = p_i - \lambda\bar{\delta}_i = p_i - \lambda\langle \delta_i, \boldsymbol{n}_i \rangle \boldsymbol{n}_i$$

4. 我们将 $\langle \delta_i, \boldsymbol{n}_i \rangle$ 存到置换贴图中，注意设置好 bias 和 scale 将值变换到 0 和 1 之间

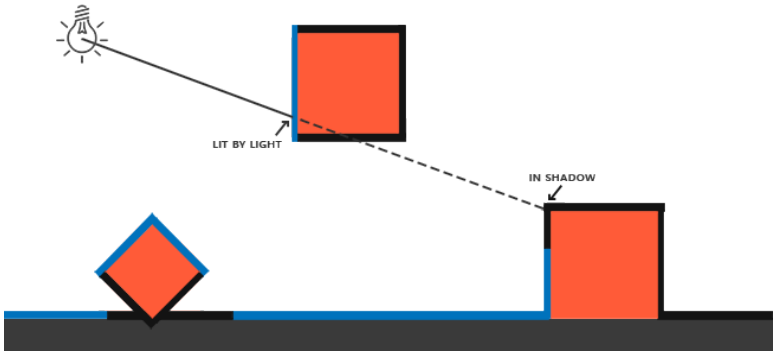
简单来说，每个顶点有纹理坐标，将图像中该位置设为 bias 和 scale 后的 $\langle \delta_i, \boldsymbol{n}_i \rangle$。

我们需要注意的是，图像是离散的，只按上述做法难免出现重合、缺漏等，因此要根据每个顶点的偏移量，合理插值出整个置换贴图（比如 K 近邻加权均值，最近邻等）。

2.4 阴影映射

阴影映射 (Shadow Mapping) 背后的思路非常简单：我们以光的位置为视角进行渲染，我们能看到的东西都将被点亮，看不见的一定是在阴影之中了。假设有一个地板，在光源和它之间有一个大盒子。由于光源处向光线方向看去，可以看到这个盒子，但看不到地板的一部分，这部分就应该

图 6: 阴影映射原理



在阴影中了。这里的所有蓝线代表光源可以看到的 fragment。黑线代表被遮挡的 fragment：它们应该渲染为带阴影的。如果我们绘制一条从光源出发，到达最右边盒子上的一个片段上的线段或射线，那么射线将先击中悬浮的盒子，随后才会到达最右侧的盒子。结果就是悬浮的盒子被照亮，而最右侧的盒子将处于阴影之中。

我们希望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。

3 实现结果

3.1 法向贴图

图 7: CG 字样的法向贴图

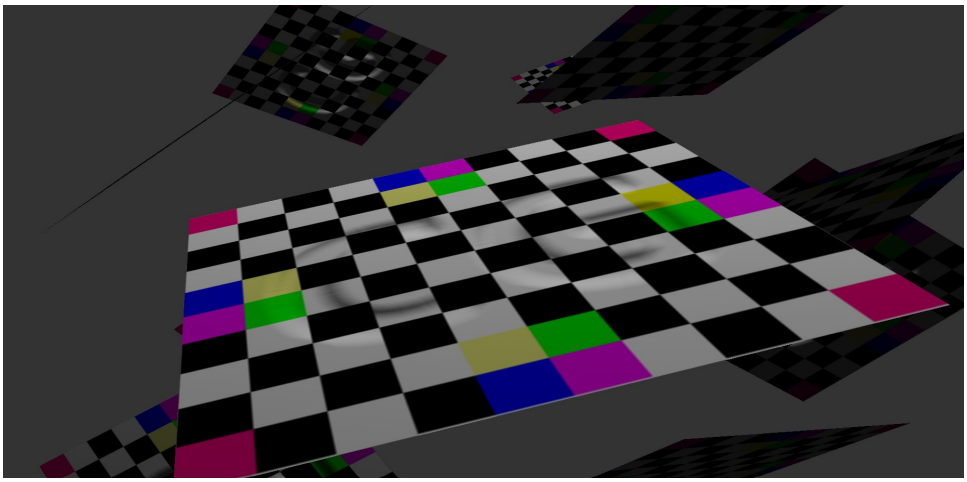


图 8: 人物图片的法向贴图

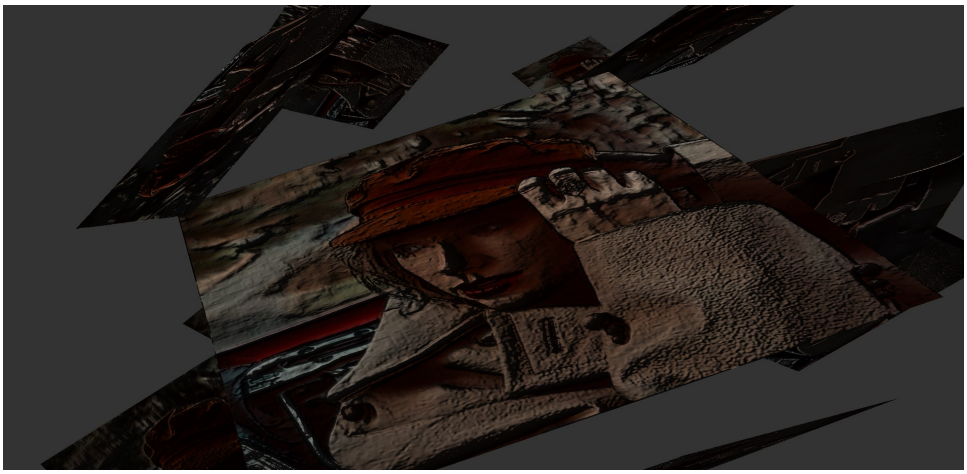
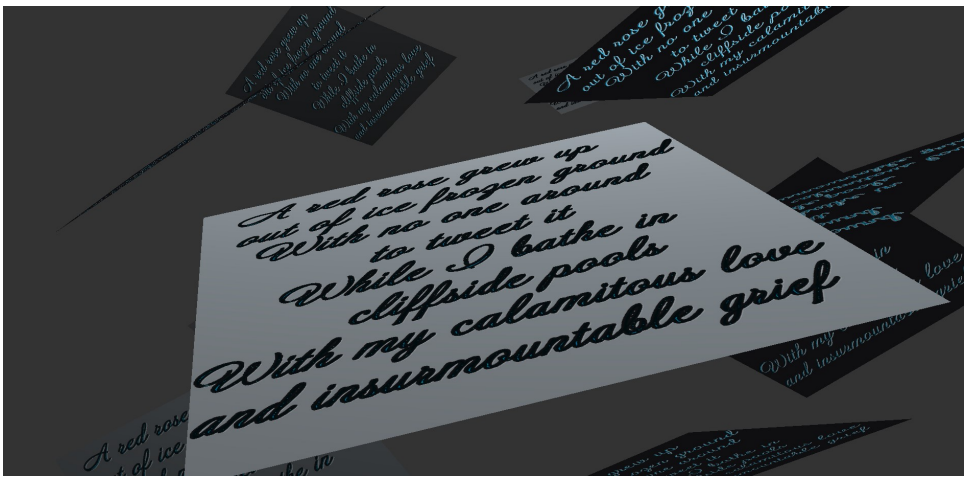
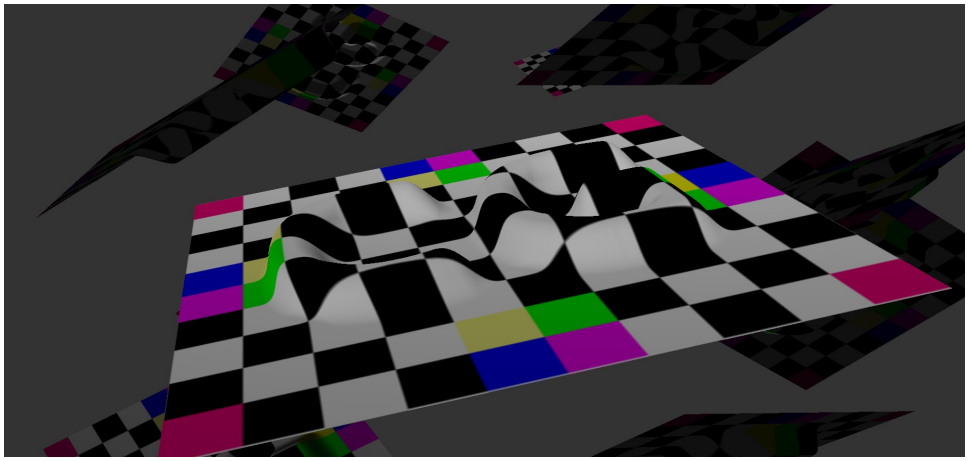


图 9: 文字的法向贴图



3.2 置换贴图

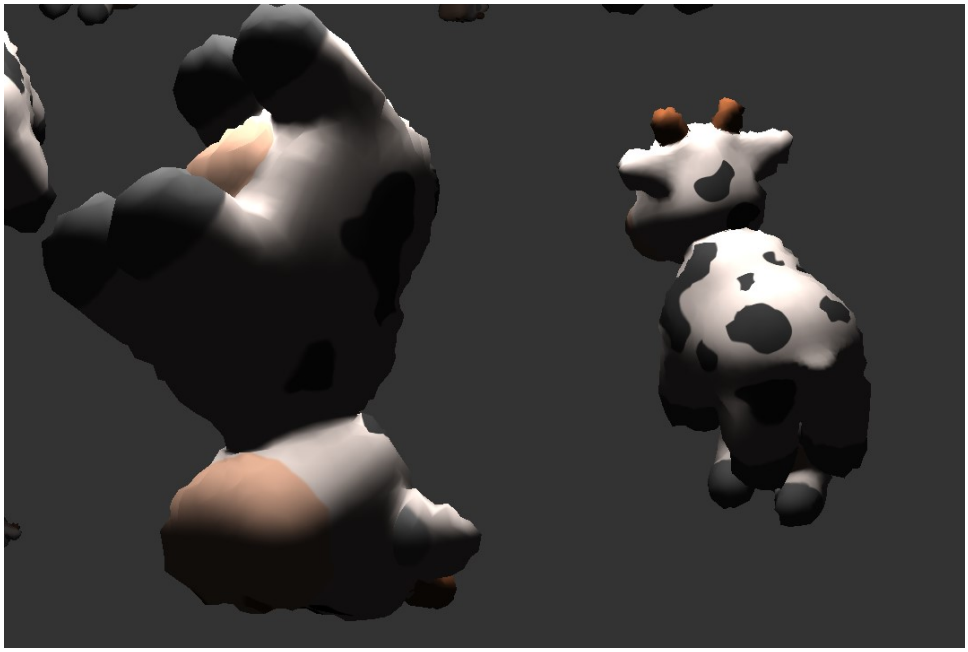
图 10: CG 字样的置换贴图



3.3 用置换贴图进行简单去噪

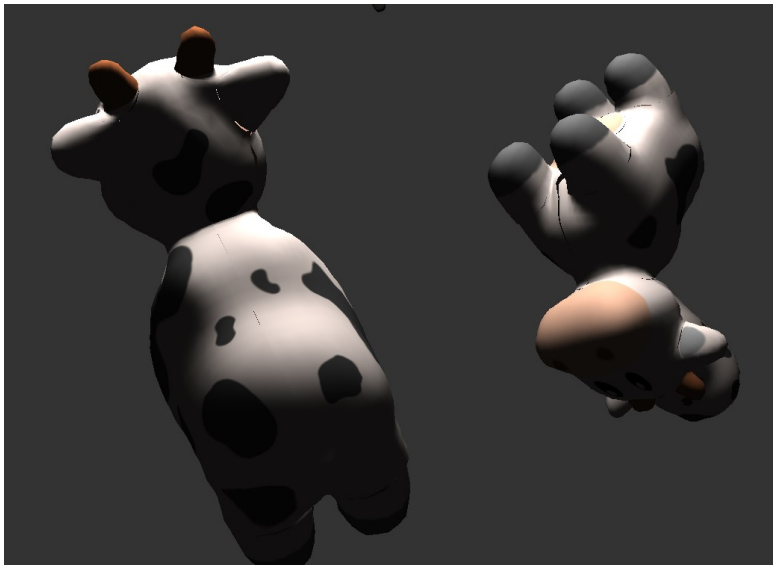
以下为原本的模型：

图 11: 有噪声的模型



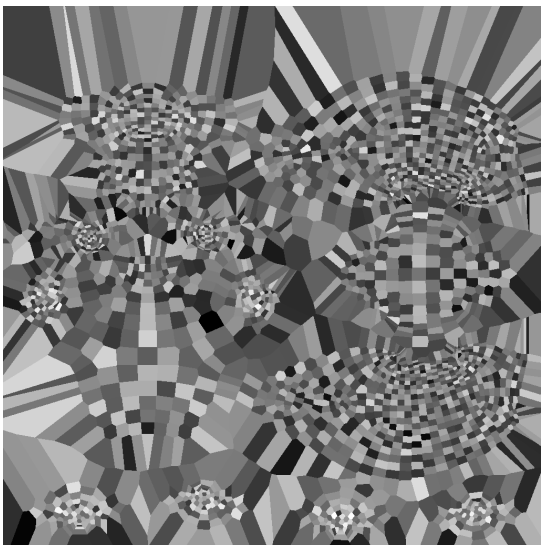
我调用 ANN 库来做图像插值：下面是我根据算法实现的去噪后的模型：

图 12: 去噪后的模型



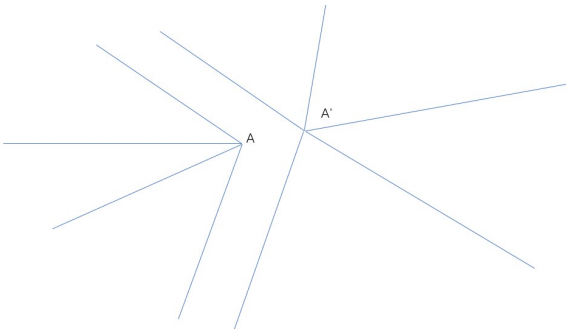
其中生成的 displacementmap 如下图：

图 13: displacementmap



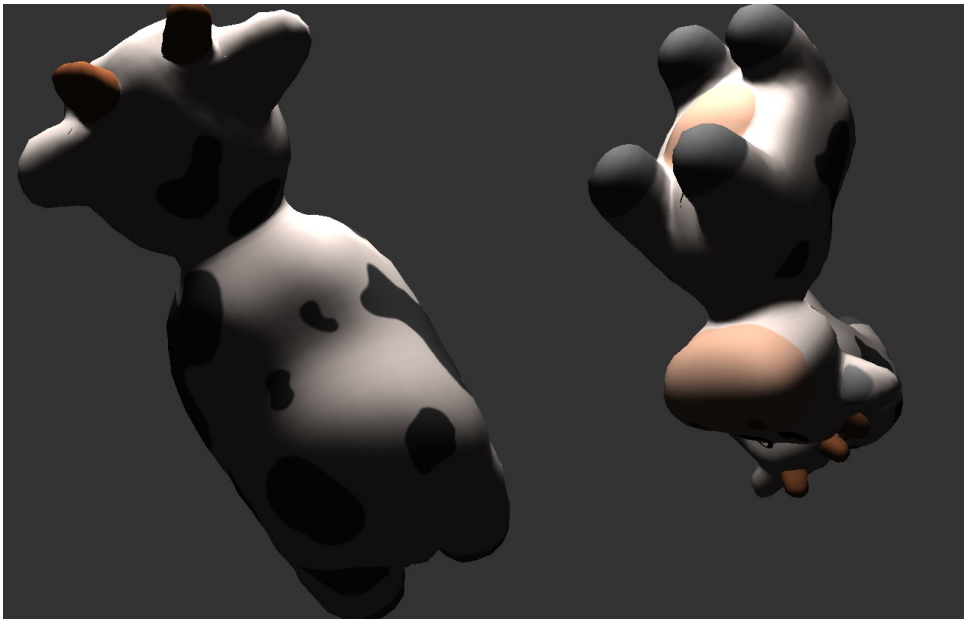
根据我的调试发现此时 displacement_lambda=0.9f 左右去噪效果最好。但是我们可以看到奶牛的背上，牛角处，肚子处等等都有裂缝。这里我们考虑可能是有许多顶点的世界坐标是相同的，法向也相同，纹理坐标不同。如下图：

图 14: 有 2 个点世界坐标相同的情况



对于这种顶点，A 和 A’的邻结点应该是共有的。再把这种情况考虑进去我们可以得到：

图 15: 去裂缝后的模型



我们可以看到奶牛背上的裂缝没有了，但是肚子上和牛角上还有少量裂缝，我们猜测可能是网格取纹理的时候本身就是不连续的，所以裂缝难以避免。

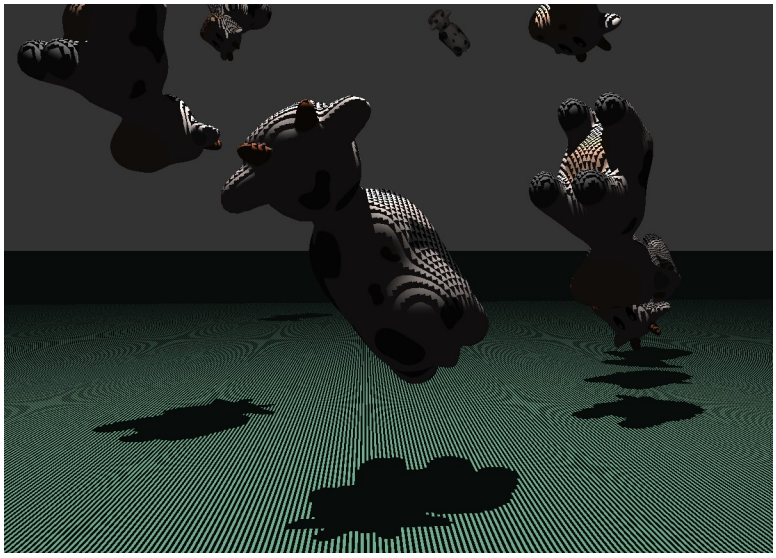
3.4 阴影映射

利用助教的框架，再按照教程：

https://learnopengl-cn.github.io/05%20Advanced%20Lighting/03%20Shadows/01 %20Shadow%20Mapping/

我们可以实现阴影的渲染，由于阴影贴图助教已经实现好了，我们只需要在 render loop 里计算好矩阵 lightProjection, lightView, 执行 shadow_program，再在片元着色器里渲染阴影就可以了。以下为实现的效果：

图 16: 初步的阴影渲染

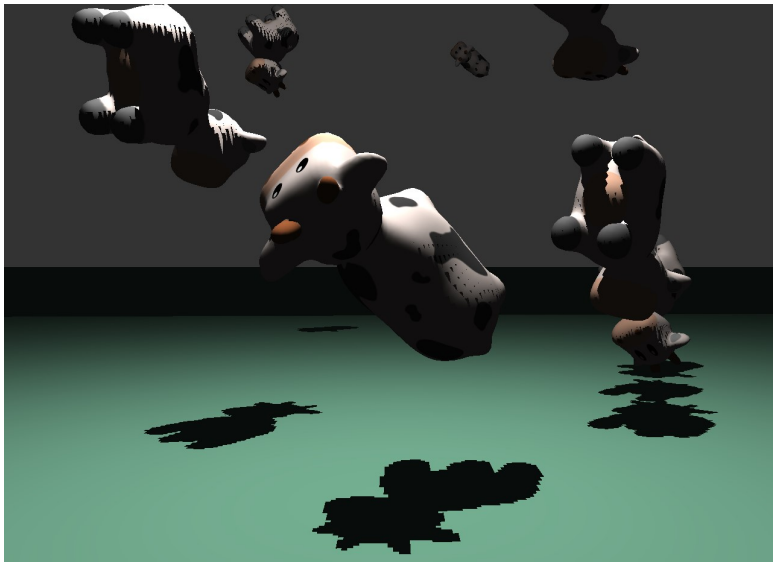


我们可以看到地板四边形渲染出很大一块交替黑线。这种阴影贴图的不真实感叫做阴影失真 (Shadow Acne)。教程里也提供如何优化算法来消除失真。我们可以用一个叫做阴影偏移（shadow bias）的技巧来解决这个问题，我们简单的对表面的深度（或深度贴图）应用一个偏移量，这样片段就不会被错误地认为在表面之下了。使用了偏移量后，所有采样点都获得了比表面深度更小的深度值，这样整个表面就正确地被照亮，没有任何阴影。我们可以这样实现这个偏移：

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

以下为偏移 0.005 后的效果：

图 17: bias=0.005



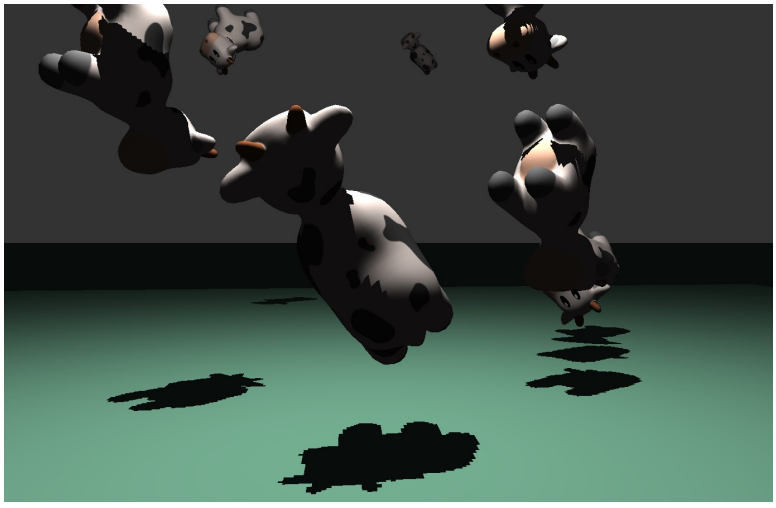
一个 0.005 的偏移就能帮到很大的忙，下面的平面已经没有阴影失真了。但是有些表面坡度很大，仍然会产生阴影失真。比如奶牛的身上坡度比较大，仍然有阴影失真。

有一个更加可靠的办法能够根据表面朝向光线的角度更改偏移量：使用点乘：

```
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
```

这里我们有一个偏移量的最大值 0.05，和一个最小值 0.005，它们是基于表面法线和光照方向的。下图展示了同一个场景，但使用了点乘的阴影偏移，效果的确更好：

图 18: bias 用点乘来计算



我们可以看到奶牛身上阴影就没有失真了。

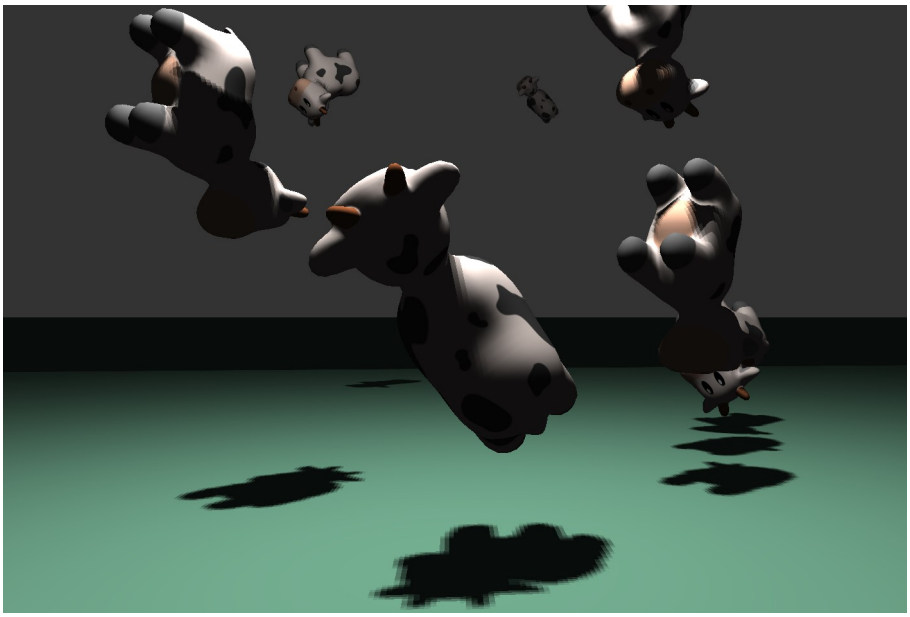
虽然阴影现在已经附着到场景中了，不过这仍不是我们想要的。如果你放大看阴影，阴影映射对分辨率的依赖很快变得很明显。因为深度贴图有一个固定的分辨率，多个片段对应于一个纹理像素。结果就是多个片段会从深度贴图的同一个深度值进行采样，这几个片段便得到的是同一个阴影，这就会产生锯齿边。

我们采用的解决方案叫做 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。一个简单的 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：

```
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;
```

以下为采用 PCF 后的效果：阴影的锯齿边确实好了很多。

图 19: 采用 PCF 方法



以下是在以上优化后的算法基础上，将正交投影改为透视投影的效果：

图 20: 透视投影

4 不足

1. 没有实现更多功能，只实现了最基本的交互和相关功能。
2. 去噪的任务中，不能将裂缝完全消除。