Homework 8 Shader Programming

华南理工大学 曾亚军

一　实验目的

- 实现法向贴图和置换贴图。
- 实现 3D 网格模型去噪。
- 渲染阴影贴图。

二　法向贴图

法向贴图的目的时存储法线信息。通过设定对应点的法向信息，从而影响后续的光照计算等操作。使用法线贴图可以用较少的顶点实现更多的细节。

法向贴图中一般将法线信息存储在对应的切线空间中。原因如下：

- 如果存储世界坐标的对应法向，不能随物体移动保持有效。
- 如果存储模型坐标的对应法向，物体进行刚性变换，法相贴图依然有效。但法向贴图不能在不同物体上复用。

切线空间由 tagent 轴 (T),bitagent 轴 (B) 和法线轴 (N) 构成，即 TBN 空间。而 TBN 矩阵即可以将切线空间的法向转化到模型空间中。并通过如下公式可计算得到 TBN 矩阵：

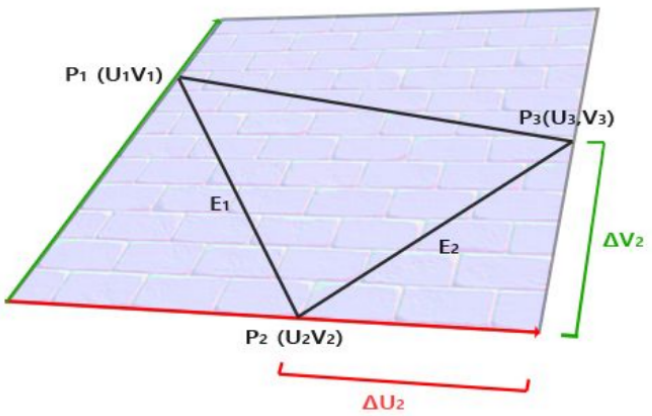


图 1: 切平面

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

图 2: 切平面

注意，N 向可直接用着色器传递的法向信息（转换成世界坐标后）。但法向不能简单用模型矩阵进行转换，否则可能会发生错误。如果模型矩阵 model 中仅包含刚性变换，则不会产生问题。但如果模型矩阵中还包含了局部的缩放，则产生的法向将会时错误的法向。因为需要采取法向矩阵 NormalMatrix:

$$NormalMatrix = transpose(inverse(view * model)) * vec4(aNormal, 0.0);$$

在顶点着色器中，TBN 矩阵计算如下

```
mat3 normalMatrix = transpose(inverse(mat3(model)));

vec3 N = normalize(normalMatrix * aNormal);
vec3 T = normalize(vec3(model * vec4(aTangent, 0.0)));
T = normalize(T - dot(T, N) * N);
vec3 B = cross(N,T);
vs_out.TBN = mat3(T, B, N);
```

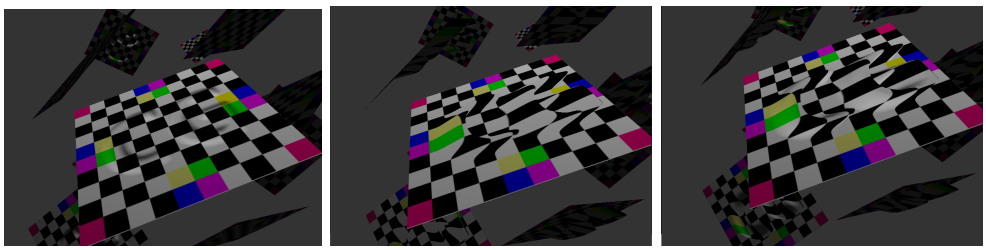
并在片元着色器中对应修改法向信息。

三　置换贴图

置换贴图中储存的是对应纹理坐标的点沿法向的偏移。实现步骤非常简单：

- 计算各点的偏移量。
- 投影到法向生成置换贴图。
- 通过顶点着色器取出置换贴图的信息。
- 在模型坐标中对各点进行偏移，再转换成世界坐标。

实现的结果如下图：



(a) 法向贴图结果　(b) 置换贴图结果　(c) 法向贴图 + 置换贴图

从图中的结果可以看出，法向贴图的效果看起来好像对物体做了偏移，但实际上没有，仅仅只是改变了法向信息。法相贴图的方法容易在某个角度中看出纰漏。置换贴图则是实际地对点做了便宜。两者结合会有较好的结果。

四　使用置换贴图去噪

去噪简单思路就是利用拉普拉斯坐标，对物体的点进行更新。其流程如下：

- 计算各点的拉普拉斯坐标，并确定偏移参数 λ。
- 确定每个顶点的偏移量，并投影到法向。
- 生成置换贴图。
- 通过顶点着色器取出置换贴图的信息。
- 在模型坐标中对各点进行偏移，再转换成世界坐标。

在生成置换贴图方面，我首先考虑的是引入 ANN 库，用最近邻的方法插值那些没有发生置换的像素。但事实上，顶点着色器读取置换信息时，也仅仅只是读取对应顶点纹理坐标的信息，有很大一部分的纹理坐标的信息是没有用的，不需要可以去填补这部分的偏移信息。

但如果直接只生成对应点的像素，其他点的像素不予考虑，最终生成的置换贴图只是零散个像素点的集合。且如果不能精准地对应各个点的纹理坐标（因为小数点取整或其他问起没能取到对应像素），往往会造成问题。因而本作业最终采取了三角重心插值的方法。得到的置换贴图如下：

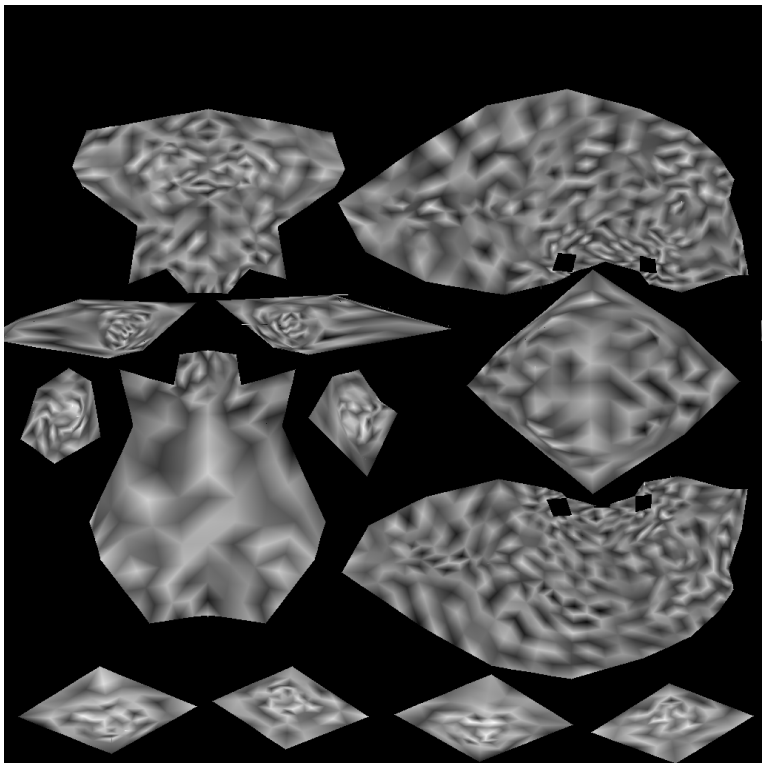
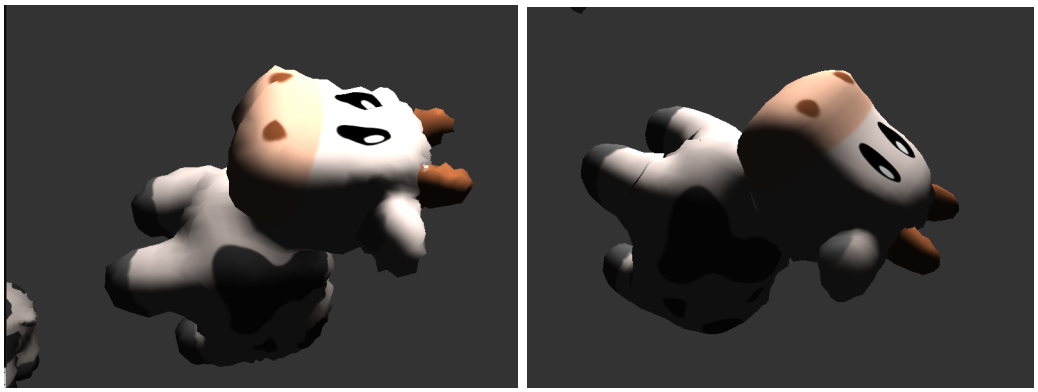


图 3: 置换贴图

最终的去噪结果如下：



(a) 去噪前　　(b) 去噪后

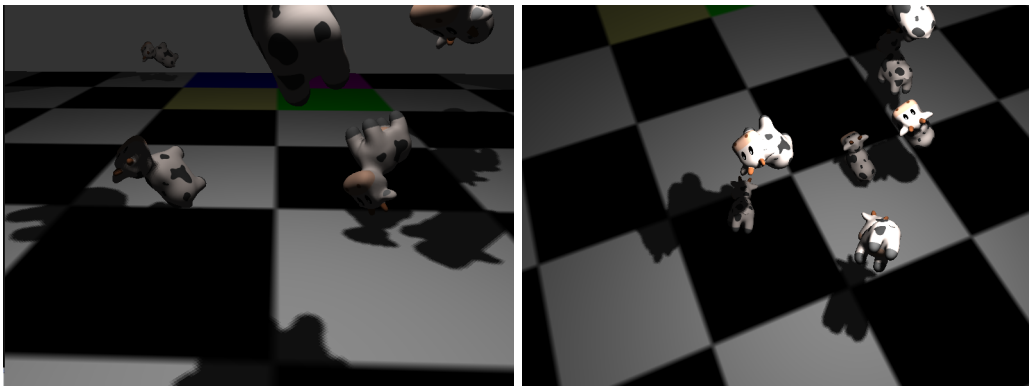
五　阴影贴图

阴影贴图的基本思想是通过以光源为视点渲染一张深度图，在光空间下确定哪些点可见哪些点不可见。再从照相机处重新渲染，光空间中的不可视点即为阴影部分。其具体流程如下：

- 以光源为视点，计算视口变换矩阵和模型矩阵，生成光空间的变换矩阵。
- 渲染深度贴图。
- 以照相机为视点，计算视口变换矩阵和模型矩阵。
- 在顶点着色器中分别计算点的世界坐标和光空间坐标，传递至片元着色器。
- 在片元着色器中计算像素可见度 Visiable。

改进：采取阴影偏移和 PCF（percentage-closer filtering）方法，避免锯齿状阴影，更加柔和的阴影边界。一个简单的 PCF 方案实现如下：

```
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;
```



(c) (d)

六　收获和感悟

本次作业相对简单，代码量较少，核心在于理解顶点着色器和片元着色器的功能。相应的 OpenGl 的教程也非常详细，整体而言并没有什么难度。但细节部分需要注意，在顶点着色器里面，有一个已经写好的 NormalMatrix 法向矩阵。事实上如果不是因为它已经写好在那，我估计会直接采取用 model 矩阵相乘得到世界坐标下的法向这种方法。然后我就去搜寻了一下法向矩阵的由来，才发现它存在的必要性。事实上，OpenGl 是一个非常好的工具，但它其实是实现了很多细节的数学功能，在我们调用这些函数和命令时，应当认真理解其背后的数学原理。