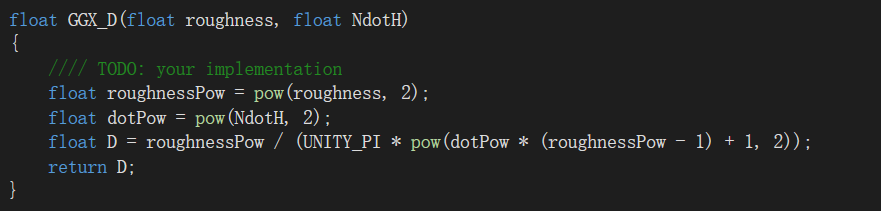
# Homework5报告

## 基础物理的shader实现

1. 实现过程：

参照作业文档依次实现BRDF中直接高光反射中的D、F、G项，代码依次如下：

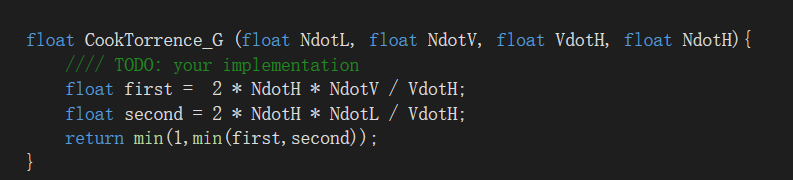
Distribution term：



Fresnel term：

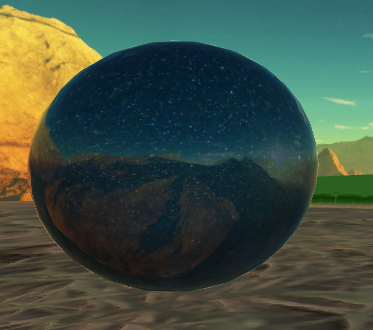


Geometry term：



实现结果如下：





## 非真实感渲染shader

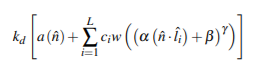
1. 卡通风格渲染

参考论文

<http://dalab.se.sjtu.edu.cn/gp2019/reference/NPAR07_IllustrativeRenderingInTeamFortress2.pdf>

卡通风格的渲染将光照分为View Independent光照和View Dependent光照。

View Independent光照的公式如下：



其中kd是从纹理图采样得到的反射率



a（n）项为环境光，因为没有找到合适的贴图，所以我在实现中简单地使用如下式子



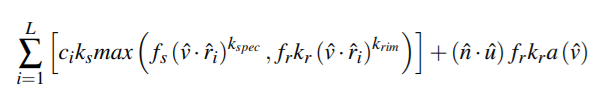
我的实现中只考虑了单光源，所以公式中的L为1。Ci为光源的颜色，实现中采用白光，可以在shaderGUI中调节。

α取0.5，β取0.5，γ取1，n向量为世界坐标系下的法向量，l向量为光照方向。通过这些计算出来的值（设为u）作为一个二维坐标（u，u）从如下纹理进行采样



采样得到的结果即w函数的值，然后代入公式，加上环境光，就得到了View Independent的值

View Dependent的光照公式如下：



仍然考虑单光照，L为1。Ci为光源颜色，ks为高光mask，用于控制高光的强度，实现中采用纯白贴图。

Fs是general高光的Fresnel系数，实现中取1，可以在shaderGUI中调节。Kspec是Phong光照模型中的高光系数，实现中取1，可以在shaderGUI中调节。

V向量为视线方向向量，r向量为光线关于法向量的反射向量。

Fr是用于控制rim light的fresnel系数，用于保证rim light只出现在边缘，实现中取，其中n向量为世界坐标系下的法向量。Krim是用于控制rim高光的高光系数，实现中取0.1，可以在shaderGUI中调节。Kr为rim light的mask，实现中采用纯白贴图。

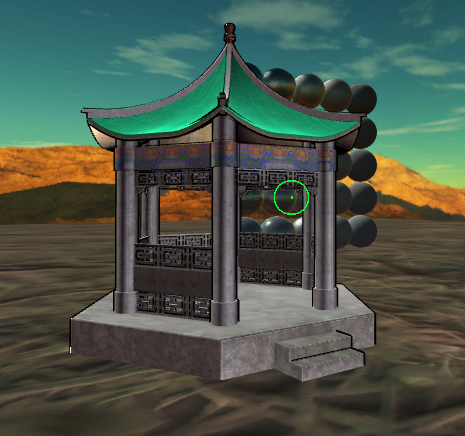
U向量为世界坐标系下的up向量。a为环境光，实现中采用白光。

将View Independent光照和View Dependent光照相加就得到了最终的渲染结果，我的实现中还利用了上次作业实现的outline描边shader来给物体描边：

原图：

渲染后：

渲染前：

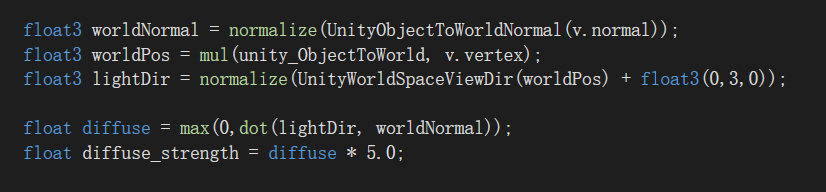
渲染后：

1. 素描风格渲染

参考论文

<http://dalab.se.sjtu.edu.cn/gp2019/reference/hatching.pdf>

我的实现原理比较简单，通过在顶点着色器中计算物体的diffuse光，将其与预设值好的阈值进行比较，然后在不同的hatch纹理图间进行插值。如下：



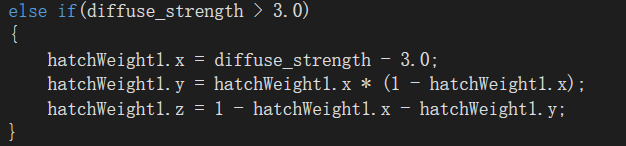
首先计算出diffuse光。如果将光源方向即lightDir设置为定值，那么物体背面的diffuse就全部为0，最终会出现如下效果

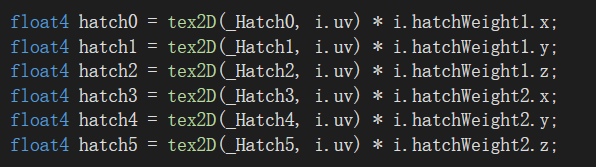


即背面diffuse\_strength全部为0，然后就缺少了阴影的变化。因此我在实现中将lightDir设置为视线向量加上一个偏移，即光源随着摄像机移动（相当于用户看到的物体所在面始终为被光照照射的那一面）。

因为总共有六张hatch贴图加上留白，每次在三张贴图之间进行插值，所以将diffuse乘以5映射到（0，5）的区间。

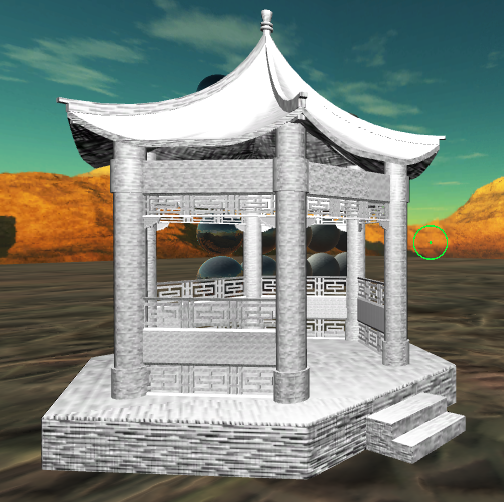
然后根据diffuse的值在不同的阴影纹理图间进行插值，六张阴影图的权重值通过两个float3向量进行存储。当diffuse\_strength大于4时，所有阴影纹理的权重都置为0，代表留白；其他情况下，按照阴影图笔触的疏密在三张相邻的阴影图之间进行插值





最终实现效果如下：





## 屏幕后处理效果shader

1. 运动模糊

参考资料：

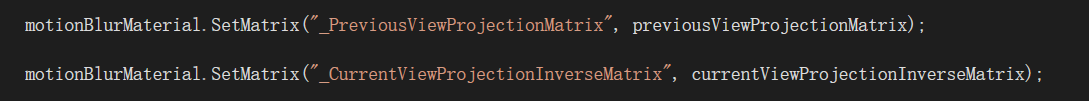
<https://developer.nvidia.com/gpugems/gpugems3/part-iv-image-effects/chapter-27-motion-blur-post-processing-effect>

冯乐乐《Unity Shader入门精要》13.2

运动模糊的实现原理：在摄像机上挂载脚本，脚本通过如下代码开启摄像机的深度值存储

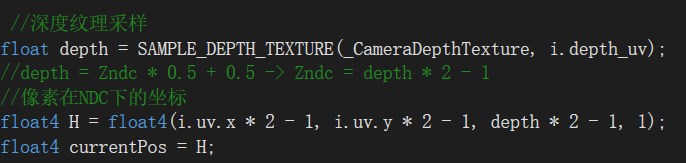


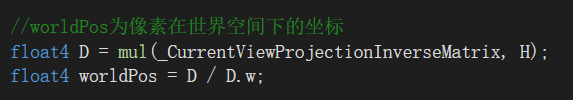
在OnRenderImage函数中向着色器传递两个重要参数



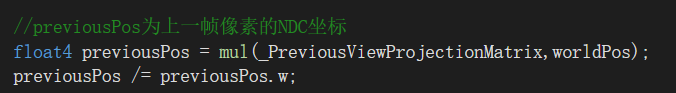
其中\_PreviousViewProjectionMatrix是渲染上一帧所使用的VP矩阵，\_CurrentViewProjectionInverseMatrix是当前渲染所使用的VP矩阵的逆矩阵。

在着色器中，首先通过相机的深度纹理计算当前片元的NDC坐标（记为currentPos），

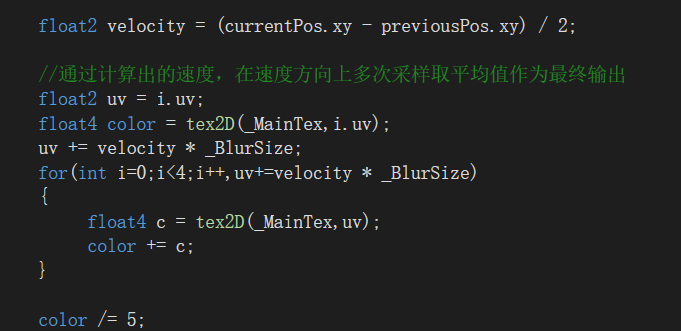
然后和传入的当前VP矩阵逆矩阵相乘计算出其世界空间的坐标（记为worldPos），



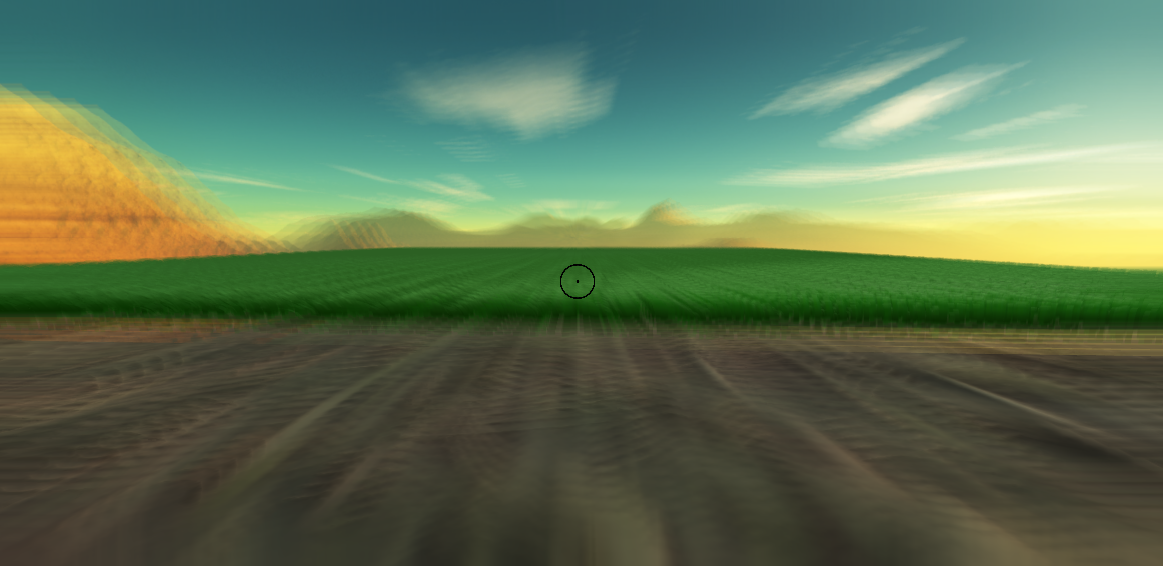
然后通过P和传入的上一帧VP矩阵构建该位置在上一帧中的NDC坐标（记为previousPos），



通过currentPos和previousPos的差即可得到当前像素的速度，最后在速度方向上进行多次纹理采样并取平均值作为最终输出



最终实现效果如下



1. 全局雾效

参考资料：

<https://www.derschmale.com/2014/01/26/reconstructing-positions-from-the-depth-buffer/>

冯乐乐《Unity Shader入门精要》13.1.2 13.3

全局雾效的实现原理为通过相机深度纹理图在片段着色器中重构出像素的世界坐标，然后利用世界坐标和相应的雾计算公式进行渲染。

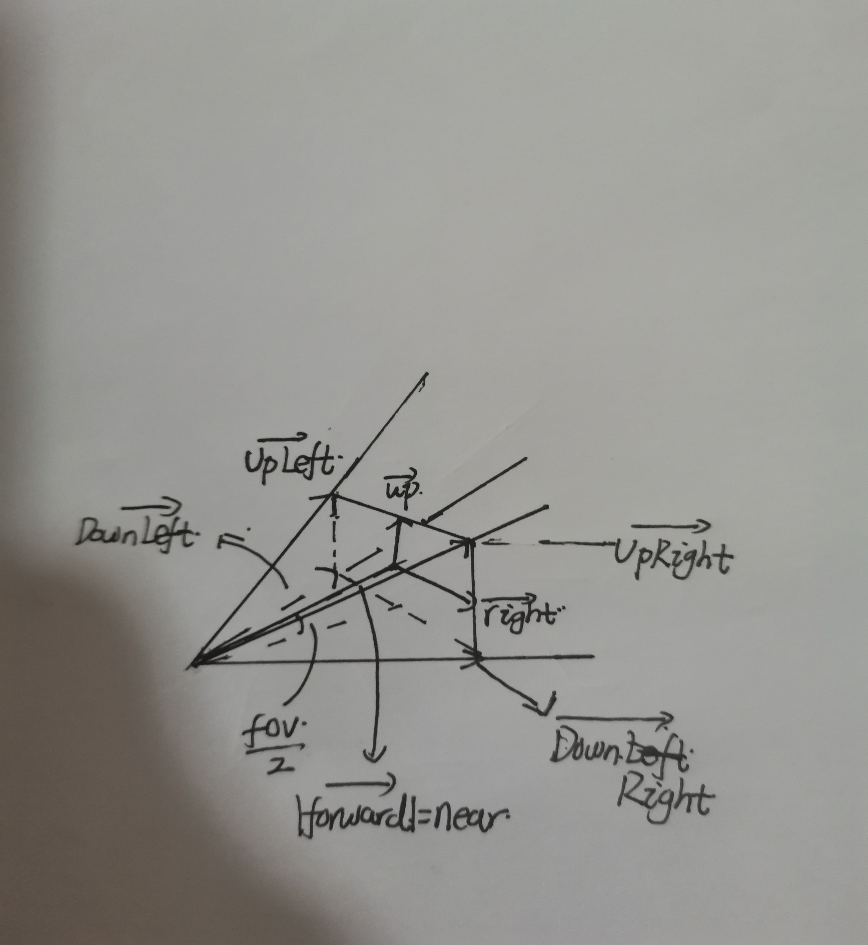
重构世界坐标：

采用的公式如下



其中\_WorldSpaceCameraPos是相机的世界坐标，可以在着色器中直接获取；linearDepth是由深度纹理得到的线性深度值，interpolatedRay是由顶点着色器插值输出的射线，它包含了像素到摄像机的方向和距离信息。

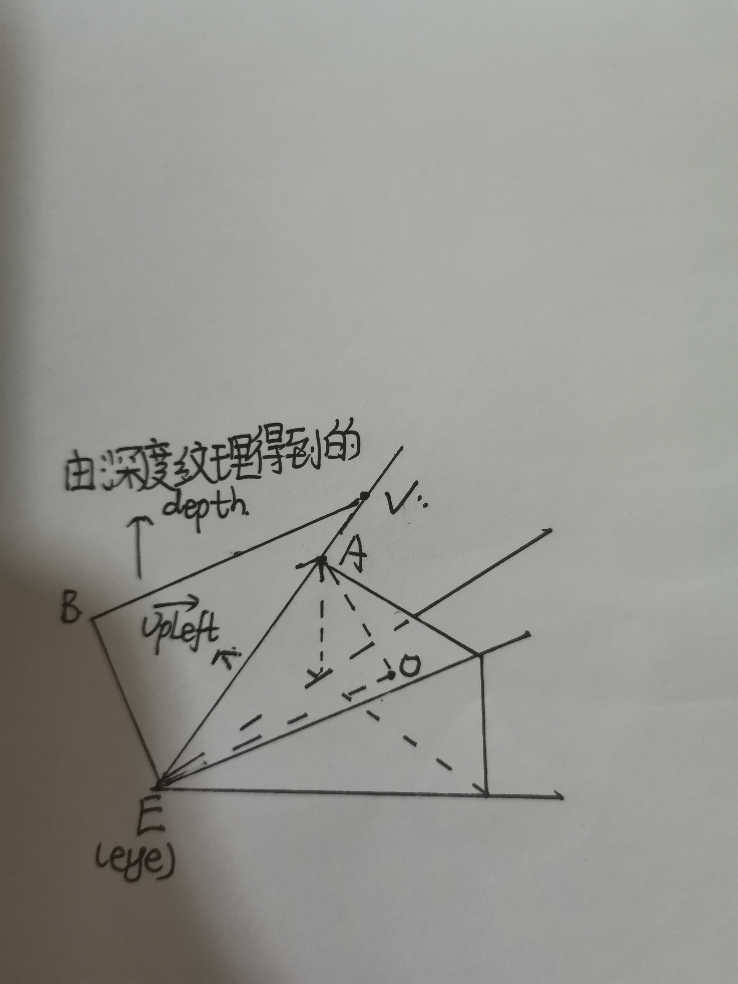
如下图



Near为相机近平面的距离，fov为视角，up向量和right向量可以由近平面的宽和长依次得到，UpLeft、UpRight、DownRight和DownLeft向量可以由forward、up和right向量计算得到，以UpLeft为例



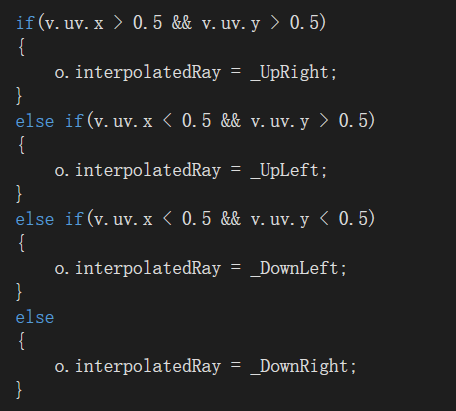
同时由于从深度纹理中得到的是线性的深度值，如下图



▲VBE相似于▲EOA，所以可以得到depth / near = |VE| / |UpLeft|，得到|VE| = depth \* |UpLeft| / near，对其他三个角来说，|UpLeft| / near是一样的，因此在将UpLeft、UpRight、DownRight和DownLeft向量传给着色器前，进行如下变换（以UpLeft为例）



在顶点着色器中，根据顶点的uv坐标选择使用哪条射线进行插值



在片段着色器中，首先通过



获得像素的线性深度（范围为[near，far]（near为近平面距离，far为远平面距离）），然后通过前面提到的公式



将相机坐标、线性深度和传入的interpolatedRay代入，即可计算得到像素的世界坐标。



我实现的是与高度相关的线性全局雾，



其中\_FogEndHeight是雾的最大高度，\_FogEndHeight是雾的最低高度，两者均可在shaderGUI中进行调节。将fogRate和传入的雾的最大浓度\_FogDensity相乘得到雾的实际浓度，最后利用该浓度在原色和雾色之间进行插值得到最终结果。

实现效果如下



可以看到，下面的雾要比上面浓。

## 四、游戏操作指南

WASD和鼠标控制移动；

按B键弹出背包，可以选择其中的模型进行放置；再按B键或者鼠标右键返回游戏；

鼠标中央的准心在碰到模型时会变绿：

按下鼠标左键，可以对模型的着色器进行选择，在着色器选择界面可以按下鼠标右键返回游戏；

按下鼠标中键，销毁模型；

按下鼠标右键，恢复模型默认的着色器。

按左shift键可以奔跑，屏幕会出现运动模糊效果

按F键可以开启/关闭全局雾效