上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

游戏设计与开发 平时作业5

作业报告

学生姓名：潘峰立

学生学号：518021910835

授课教师：杨旭波

学院(系)：电子信息与电气工程学院

软件工程专业

目录

一、基于物理的 shader

1.1 原理学习

BRDF——Bidirectional Reflectance Distribution Function，双向反射分布函数

$$L_r(v) = \int_\Omega L_i(l) f_r(l,v)(n \cdot l) dl$$

L 和 v 分别是 light 入射光的方向，以及视角（view）的方向

所以 fr（l，v）的意义就很明确了，不同方向入射的光在不同视角看起来也不一样。对应着类似这样的不均匀分布图

BRDF 只有漫反射和镜面反射两部分，环境光当然不是考虑范围。

在上一次作业中，我们其实已经实现了简单的 BRDF ，其中我们对于漫反射使用了 Lambert 的模型，而对高光使用了 Blinn-Phong 模型。在当时的计算中有

$$f_r(l,v) = \begin{cases} k_d f_{Lambert} + k_s f_{Blinn-Phong}, & \theta < 90^\circ \\ 0, & else \end{cases}$$

这里的 else，就是我们之前作业的的 dotclamp 所切割的部分。

我们真正需要做的，只是 specular 的部分，即实现下面的 DFG 三部分

含三部分，分别是 Facet slope distribution term D、Fresnel term F 以及 Geometrical attenuation term G。其中 D 描述了微面元上的朝向分布，表现了局部的反射效果；而 F 描述了菲涅尔效应的程度； G 则描述了微面元之间的几何遮挡因素。

$$f(l,v) = \begin{cases} k_d f_d + k_s \dfrac{D(h)F(v,h)G(l,v,h)}{4(n \cdot l)(n \cdot v)}, & \theta < 90^\circ \\ 0, & else \end{cases}$$

1.2 实现过程

打开 Package 之后发现很多东西都已经做好了，我们只需要实现 DFG 对应的三个函数即可。

1.2.1 D——GGX 算法

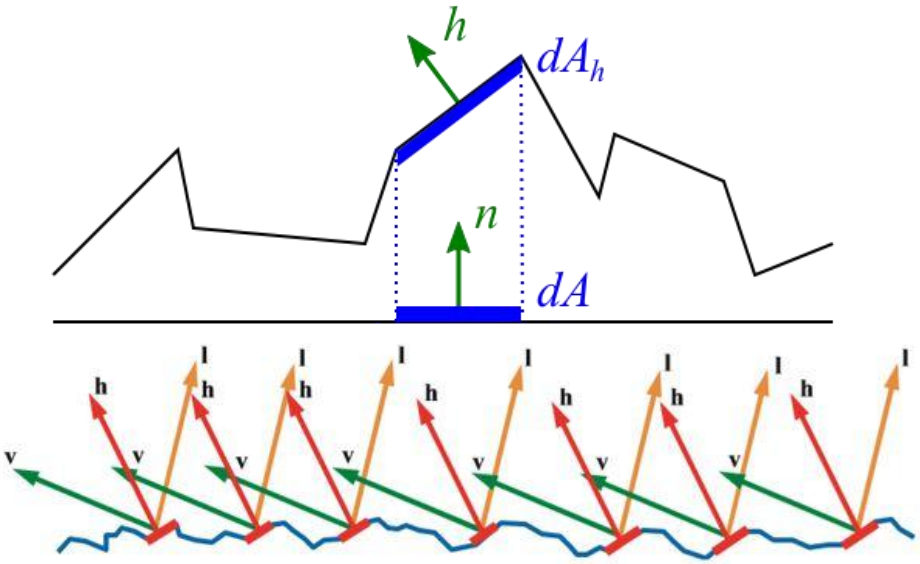
函数 D(h)被解释为微观角度下微小镜面法线的分布函数 NDF——normal distribution function。

函数 D(h)的物理含义为每单位立体角所有法向为 h 的微平面的面积，所以会有π

的存在。

> 知道了概率密度函数p，我们就可以根据它进行采样。在这里我们以GGX的NDF为例。
> GGX的NDF形式如下：
> $$NDF-GGX(n,h,\alpha)=\frac{\alpha^2}{\pi(cos\theta^2(\alpha^2-1)+1)^2}$$

只有当确定了观察方向 v 和入射方向 l 之后，我们所真正在乎的能对镜面反射做出贡献的只有法线为 h 的微平面，如下图所示



下面的公式用 w_h 代替法向 h。

You Found a Secret Area!

The GGX paper, Microfacet Models for Refraction through Rough Surfaces by Walter et al., has the answer. According to them, the NDF obeys the equation:

$$dA_h = D(h)\, d\omega_h\, A$$

> 知道了概率密度函数p，我们就可以根据它进行采样。在这里我们以GGX的NDF为例。
> GGX的NDF形式如下：
> $$NDF-GGX(n,h,\alpha)=\frac{\alpha^2}{\pi(cos\theta^2(\alpha^2-1)+1)^2}$$

实际上落实到代码的是这个公式

$$D_{GGX} = \frac{\alpha^2}{\pi((n\cdot h)^2(\alpha^2-1)+1)^2}$$

函数输入参数已经有了我们所需要的参数，直接翻译即可。

```
float GGX_D(float roughness, float NdotH)
{
    //// TODO: your implementation
    float a2 = roughness * roughness;
    float D = a2 / (UNITY_PI * pow((NdotH* NdotH*(a2-1)+1), 2));
    return D;
}
```

Unity_PI 是 unity 自带的π常数。

1.2.2 F——Fresnel 菲涅尔方程

菲涅尔方程是中学物理学到的折射光和折射率的关系，意义就是：入射光=反射光+折射光。实际上是描述了物体表面在不同入射光角度下反射光线所占的比率。回想一下当时的实验：观察角度与法线夹角越大反射程度一般越大。越是朝球面掠角的方向上看（此时视线和表面法线的夹角接近 90 度）菲涅尔现象就越明显，反光就越强（因为折射最弱）。

公式如下，C_{spec} 用来代替原来的反射系数 R

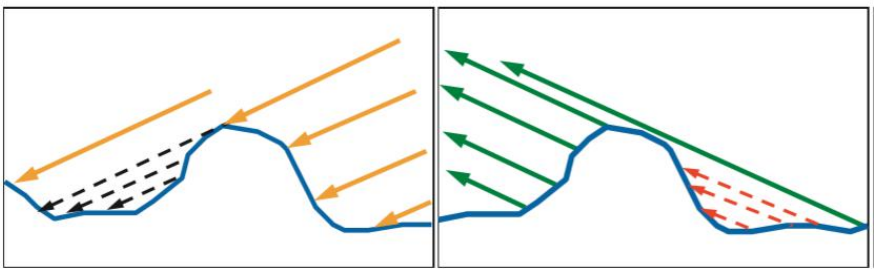
$$F_{Schlick} = C_{spec} + (1 - C_{spec})(1 - l \cdot h)^5$$

实际上已有的代码已经帮我们做了 C_{spec} 的转换了，所以直接用 R 即可。

```
float3 Schlick_F(half3 R, half cosA)
{
    //// TODO: your implementation
    float3 F = R + (1 - R) * pow((1 - cosA), 5);
    return F;
}
```

1.2.3 G——Cook-Torrance

几何函数 G 是为了表示微平面的自遮挡从而引起的光线损失，一般会出现如下两种的遮挡情况。



左边一幅图中是入射光线无法照射到一些微平面，这种情况称为 Shadowing，右边图中是反射光线无法正常到达人眼，称为 Masking，而几何函数 G 正是为了模拟出这两种情况所导致的光线所示。

公式如下：

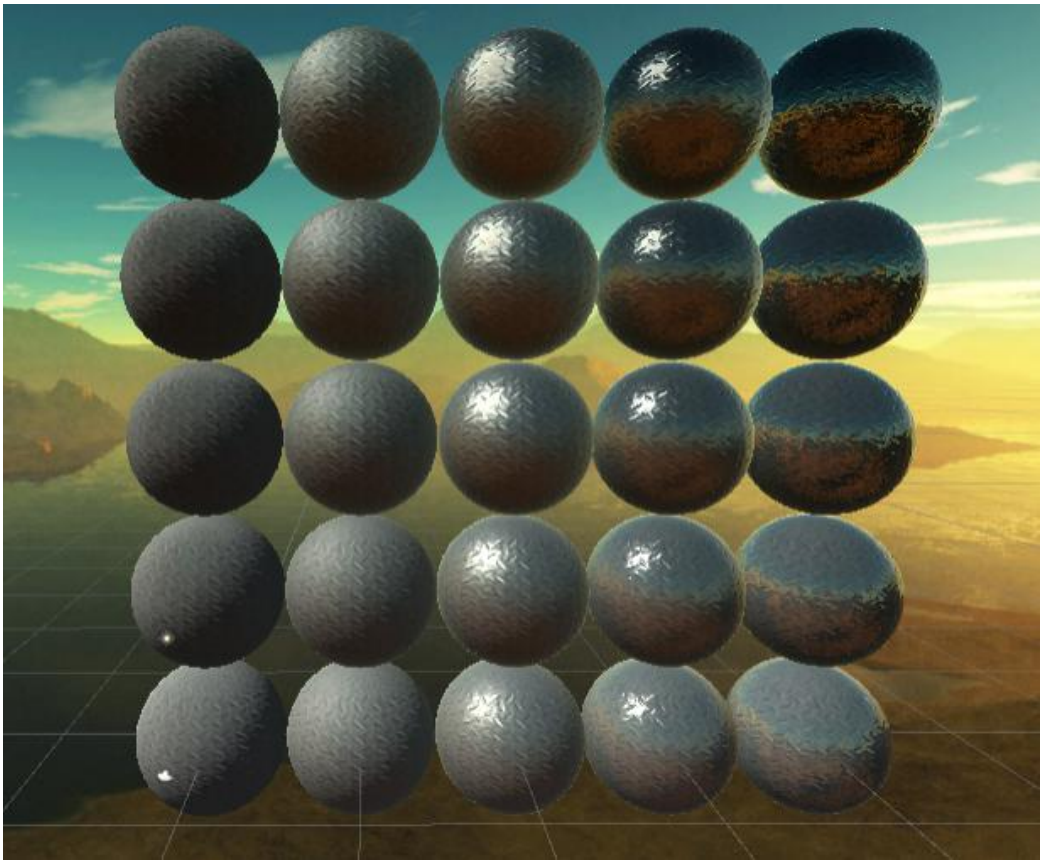
$$G_{Cook-Torrance} = \min\left(1, \frac{2(n\cdot h)(n\cdot v)}{v\cdot h}, \frac{2(n\cdot h)(n\cdot l)}{v\cdot h}\right)$$

代码翻译：

```
float CookTorrence_G (float NdotL, float NdotV, float VdotH, float NdotH){
    //// TODO: your implementation
    float G1 = min(1, 2 * NdotH * NdotV / VdotH);
    float G=min(G1, 2 * NdotH * NdotL / VdotH);
    return G;
}
```

1.3 实现结果展示

按照 pdf 所说地把 prefab 代入到脚本中即可展示出如下的矩阵效果。



二、非真实感渲染

2.1 Hatching

2.1.1 实现原理和思路

论文讲到的东西比较多。关键词：Hatch——孵化，stroke——笔画，tonal art map——TAM 色调艺术图，mip-mapped hatch 图片

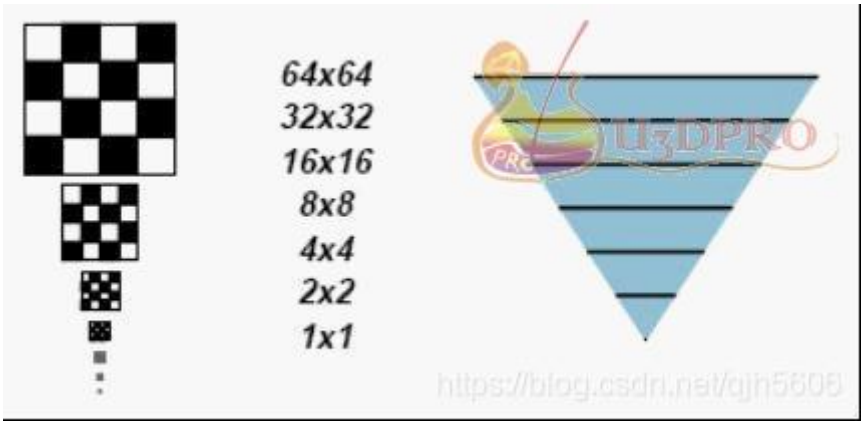
笔划被预先渲染成一组纹理图像，在运行时，通过适当地混合这些纹理来渲染曲面。所

以需要我们去预先载入那些图片。

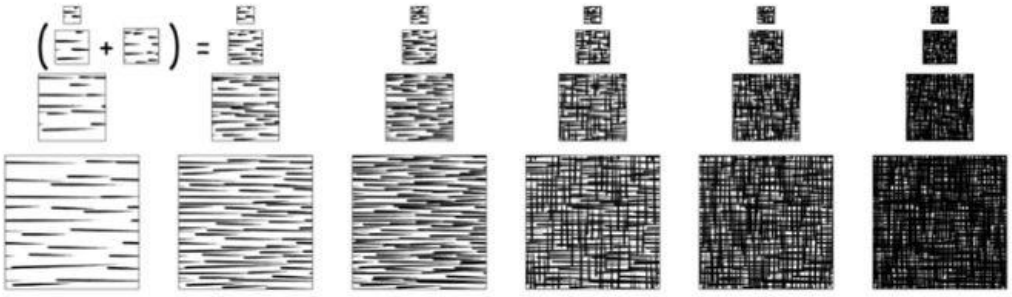
我们为与每个 tone 相关联的笔画图像执行 mip-map 构造。我们将由此得到的像素映射图像序列称为调性艺术地图(TAM)。

Mip-map 的意义在于，不同远近，不同大小，不同分辨率的情况下所展现出的笔画应该是不一样的。

Mip-map 原理大概是这样



Mip-map 原理图

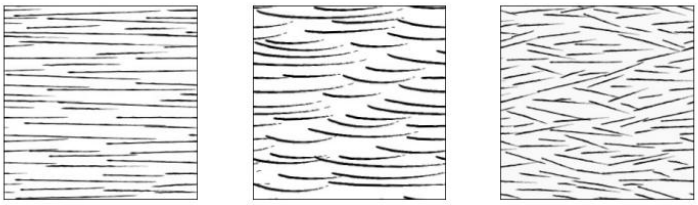


这些纹理组成色调艺术映射，纹理从左到右的笔画逐渐增多，用于模拟不同光照效果下的漫反射效果，从上到下对应每张纹理的多级渐远纹理。

在后面的实现中，我们不考虑多级渐远纹理的生成，直接使用 6 张纹理进行渲染。首先在顶点着色器计算逐顶点光照，根据光照结果决定纹理的混合权重，然后传递给片元着色器，片元着色器根据权重混合 6 张纹理的采样结果。

在生成笔画的过程中涉及到了随机化处理，但是为了均匀要取最拟合的一个：

The naïve approach of random stroke selection leads to a non-uniform distribution of strokes, as illustrated in the figure to the right. Typically an artist would space the strokes more evenly. To get even spacing, we generate multiple randomly-placed candidate strokes and select the "best-fitting" one. The number of candidates is set to, say, 1000 for



等等等等。

最关键的地方在于 6-way blend。

根据我们的颜色的灰度把笔画纹理分成了 6 个 level，镜面反射和漫反射（也就是反射光强）会决定这个比例大小，然后把这 6 个 level 的比例打包在两个纹理里面，然后把这些笔画纹理绑定到顶点上即可。这 6 种 tone 不包括纸的白色，但是需要保证他们所有的比例和是 1，所以实际上是分成了 7 个 level。

2.1.2 实现过程

如上所说，我们的纹理需要预先导入，所以 properties 里面需要 0~5 六个纹理图片。此外，我们还会用到上次作业所实现的描边效果，来让素描更加真实。

```
Properties
{
    _Color("Color Tint", Color) = (1, 1, 1, 1)
    _TileFactor("Tile Factor", Float) = 1 //TileFactor为纹理的平铺系数，值越大，素描线条越密集
    _OutlineThickness("_OutlineThickness", Range(0, 1)) = 0.03
    _OutlineColor("Outline Color", Color) = (0, 0, 0, 1)
    _Hatch0("Hatch 0", 2D) = "white" {}
    _Hatch1("Hatch 1", 2D) = "white" {}
    _Hatch2("Hatch 2", 2D) = "white" {}
    _Hatch3("Hatch 3", 2D) = "white" {}
    _Hatch4("Hatch 4", 2D) = "white" {}
    _Hatch5("Hatch 5", 2D) = "white" {}
    _AlphaScale("Alpha Scale",Float) = 1            //自定义Alpha值
}
```

既然涉及到引用上次作业，那么可以用 UsePass

```
UsePass "Unlit/OutlineShader/OUTLINE"

Pass
{
    Name "OUTLINE"
    Cull Front
```

给上次的描边的 pass 命名

然后我们的片段着色器的结构体需要存储起来我们计算出的六个 level 的权重数据。SHADOW 只是一个阴影增强效果，无伤大雅。

```
struct v2f
{
    float4 pos : SV_POSITION;//这里需要注意，由于TRANSFER_SHADOW的要求，需要名字必须
    float2 uv : TEXCOORD0;
    fixed3 hatchWeights0 : TEXCOORD1;
    fixed3 hatchWeights1 : TEXCOORD2;//6个权重值分别存储在2个float3的TEXCOORD中
    float3 worldPos:TEXCOORD3;
    SHADOW_COORDS(4) // 相当于float4 _ShadowCoord : TEXCOORD4
};
```

然后通过光强来计算我们各个 level 对应的权重。这里的光强用漫反射系数来代替

```
float diff = DotClamped(worldLightDir, worldNormal);
//使用世界空间下的光照方向和法线方向得到漫反射系数
//通过计算漫反射系数来区分采样权重
```

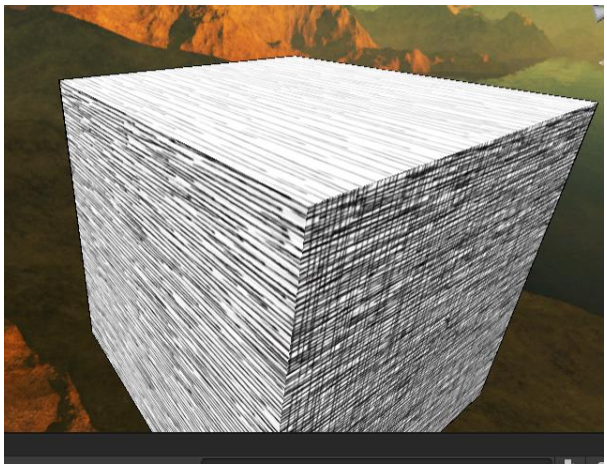
实际上是分成 7 个 level。论文中也提了到一次对应最多两个 level，并且比例和为 1.

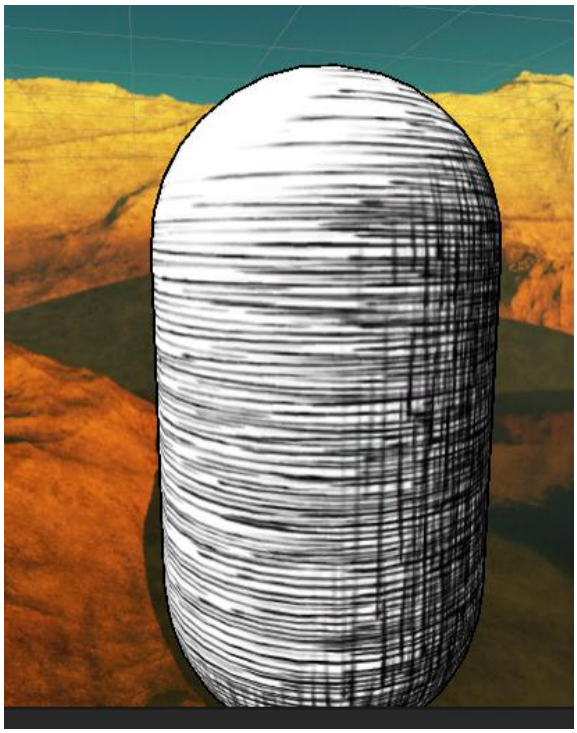
```
float hatchFactor = diff * 7.0;//化标，方便下面计算分类

if (hatchFactor > 6.0)
{
    //Pure white,do nothing
}
else if (hatchFactor > 5.0)
{
    o.hatchWeights0.x = hatchFactor - 5.0;
}
else if (hatchFactor > 4.0)
{
    o.hatchWeights0.x = hatchFactor - 4.0;
    o.hatchWeights0.y = 1.0- o.hatchWeights0.x;
```

最后在片段着色器中计算出各个纹理对应的颜色数据，并且计算出留白。

```
fixed4 frag(v2f i) : SV_Target
{
    //得到6张素描纹理采样结果，并乘以对应的权重
    float4 hatchTex0 = tex2D(_Hatch0, i.uv) * i.hatchWeights0.x;
    float4 hatchTex1 = tex2D(_Hatch1, i.uv) * i.hatchWeights0.y;
    float4 hatchTex2 = tex2D(_Hatch2, i.uv) * i.hatchWeights0.z;
    float4 hatchTex3 = tex2D(_Hatch3, i.uv) * i.hatchWeights1.x;
    float4 hatchTex4 = tex2D(_Hatch4, i.uv) * i.hatchWeights1.y;
    float4 hatchTex5 = tex2D(_Hatch5, i.uv) * i.hatchWeights1.z;
    //计算纯白的占比程度，素描风格中会有留白，并且高光部分也是白色
    float4 WhiteColor = float4(1, 1, 1, 1) * (1 - i.hatchWeights0.x - i.hatchWeights0.y - i.hatchWeights0.z
    float4 hatchColor = hatchTex0 + hatchTex1 + hatchTex2 + hatchTex3 + hatchTex4 + hatchTex5+ WhiteColor;
```

2.1.3 实现效果展示



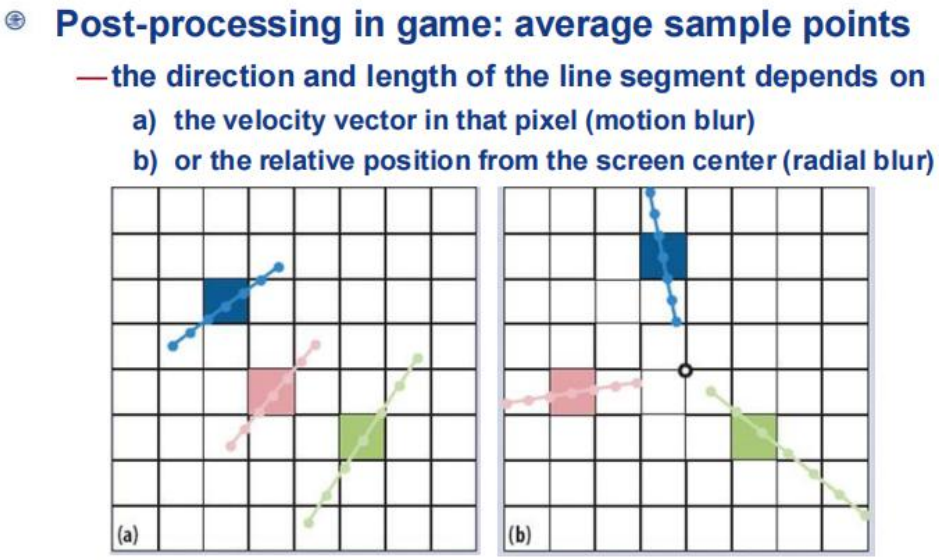


三、屏幕后处理

3.1 Motion Blur

3.1.1 实现原理和思路

上课的 ppt 里面讲到了基础的原理：



大致思路是在像素点的速度方向去采样求平均来展示出模糊的效果。

参考网站的意思大致相同，但是详细介绍了屏幕后处理的技术——通过使用存储在深度缓冲区中的深度值以及当前帧的视图投影矩阵来计算每个像素的世界空间位置。一旦确定了该像素处的世界空间位置，就可以使用上一帧的视图投影矩阵对其进行变换。然后，我们可以计算当前帧和前一帧之间的视口位置差异，以生成每个像素的速度值。

我们可以通过将一个给定像素的**视口空间位置**定义为 H 来说明这是如何实现的。设 M

为世界观投影矩阵，W 为该像素的世界空间位置。

$$H = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right)$$

$$H \times M^{-1} = \frac{wX}{wW}, \frac{wY}{wW}, \frac{wZ}{wW}, wW = D$$

$$W = \frac{D}{D.w}$$

从视口空间变换到了世界空间。

参考网站之后详细讲到了如何操作，但是最主要的问题就在于，如何获取到 depth texture——获取 depth buffer as texture 的方法因平台而异，并取决于所使用的图形 API——这就需要从 camera 获取到并且传入到 shader 中去——上面的代码都是 shader 的代码。

3.1.2 实现过程

首先明确我们需要获取到的数据有哪些：

（模糊程度）、前一个视角投影矩阵、当前视角投影矩阵的逆矩阵

```
float4x4 _PreviousViewProjectionMatrix;
float4x4 _currentViewProjectInverseMatrix;
float _BlurSize;
```

通过计算可以得到我们需要的矩阵

```
Matrix4x4 currentViewProjectMatrix = MainCamera.projectionMatrix * MainCamera.worldToCameraMatrix;//当前的矩阵，从世界到视角空间
Matrix4x4 currentViewProjectInverseMatrix = currentViewProjectMatrix.inverse;//逆矩阵
```

我们可以通过 SetFloat 函数来设置 shader 的参数，类似的方法在之后的游戏本体优化里面也有用到。

```
material.SetFloat("_BlurSize", blurSize);//把模糊的程度传入shader
material.SetMatrix("_PreviousViewProjectionMatrix", previoisViewProjectMatrix);//之前的视角空间的投影矩阵
```

由于这里是后处理效果，所以需要我们去临时创建一个 material，并且用于渲染。

```
Graphics.Blit(source, destination, material);//使用材质进行渲染
```

接下来我们得到了 shader 所需要的参数，就可以尽情地用来做处理了。

为了得到更好的效果有些代码需要微调

```
//_CameraDepthTexture深度纹理
//SAMPLE_DEPTH_TEXTURE:对深度采样
float zOverW = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture,i.uv_depth);
// H is the viewport position at this pixel in the range -1 to 1.
//float4 H = float4(i.uv.x * 2 - 1, (1 - i.uv.y) * 2 - 1, zOverW, 1);
float4 H = float4(i.uv.x * 2 - 1, i.uv.y * 2 - 1, 2*zOverW-1, 1);
//float2 velocity = (currentPos - previousPos) / 2.0f;//得到了速度
float2 velocity = -1*(currentPos - previousPos) / 2.0f;//速度小一点...
```

以及调节采样个数以及模糊程度来得到不至于过分模糊的效果。

```
uv += velocity*_BlurSize;//新的采样点
int g_numSamples = 5;//先用3试试?
for (int i = 1; i < g_numSamples; ++i, uv += velocity * _BlurSize)
{   // Sample the color buffer along the velocity vector.
    float4 currentColor = tex2D(_MainTex, uv);//偏移采样
    // Add the current color to our color sum.
    color += currentColor;
}
// Average all of the samples to get the final blur color.
float4 finalColor = color / g_numSamples; //平均采样
```

3.1.3 实现效果展示

四、游戏操作指南

本次作业在前一次的基础上对游戏的 UI 和操作方式进行了优化

4.1 基础操作

整体是一个第一人称游戏，可以 wsad 前后左右走，空格跳跃。鼠标会固定成中间的准星，按 esc 键可以解除锁定，按鼠标中键可以继续锁定。

左上角可以暂停，暂停界面可以选择开启和关闭动态模糊。

左键点击可以创建物体，右键可以破坏物体。

4.2 更新内容

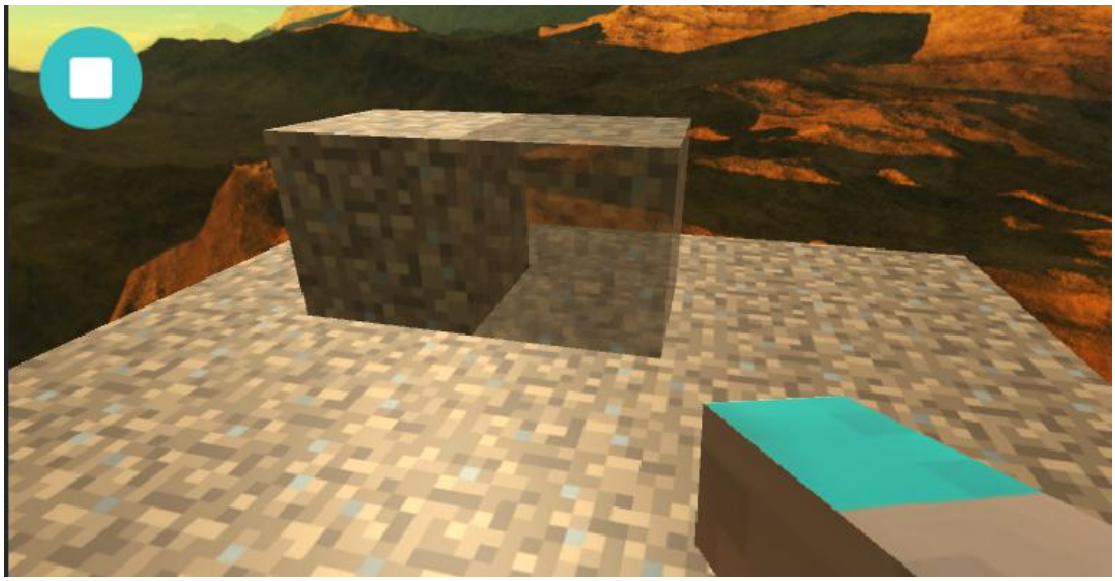
除了基础的前后左右行走，还设计了空格键可以上升，左 ctrl 可以下降的操作。

摒弃了之前的笨重的 UI 设计，优化成鼠标滚轮进行切换，提高了整个游戏的可拓展性。

并且加上了射线检测的操作，现在可以针对不同的面去创建物体了

4.3 效果展示

游戏会显示一个半透明的虚的方块来展示出当前选择的是哪一个材质。然后点击之后就会建立一个实体化的物体



五、参考文献

[1] GGX 讲解：https://www.cnblogs.com/wickedpriest/p/13788527.html
https://www.reedbeta.com/blog/hows-the-ndf-really-defined/
[2]cook-torrance 原理 https://zhuanlan.zhihu.com/p/152226698
[3]mip-map: https://blog.csdn.net/qjh5606/article/details/89040887
[4]motion-blur: https://blog.csdn.net/qq_26489679/article/details/108725098
[5]PostEffectBaseTest: https://blog.csdn.net/l773575310/article/details/78604913
[6]CGINCLUDE: https://blog.csdn.net/h5502637/article/details/84945261
[7]物理射线检测：https://blog.csdn.net/wanglining1987/article/details/74837945
[8]滚轮":https://www.cnblogs.com/yxwkf/p/5237654.html
[9]改变透明度: https://blog.csdn.net/qq_42459006/article/details/84253454