[2024.01.30]内置着色器UnityCG.cginc源码阅读

前言

函数源码版本为Unity2022.3.15。

圆周率PI相关

```
#define UNITY_PI            3.14159265359f    //圆周率
#define UNITY_TWO_PI        6.28318530718f    //2倍圆周率
#define UNITY_FOUR_PI       12.56637061436f    //4倍圆周率
#define UNITY_INV_PI        0.31830988618f    //圆周率的倒数
#define UNITY_INV_TWO_PI    0.15915494309f    //2倍圆周率的倒数
#define UNITY_INV_FOUR_PI   0.07957747155f    //4倍圆周率的倒数
#define UNITY_HALF_PI       1.57079632679f    //半圆周率
#define UNITY_INV_HALF_PI   0.636619772367f    //半圆周率的倒数
```

颜色空间(Gamma 和 Linear)的常量

```
#ifdef UNITY_COLORSPACE_GAMMA
#define unity_ColorSpaceGrey fixed4(0.5, 0.5, 0.5, 0.5)
#define unity_ColorSpaceDouble fixed4(2.0, 2.0, 2.0, 2.0)
#define unity_ColorSpaceDielectricSpec half4(0.220916301, 0.220916301,
0.220916301, 1.0 - 0.220916301)
#define unity_ColorSpaceLuminance half4(0.22, 0.707, 0.071, 0.0) // Legacy:
alpha is set to 0.0 to specify gamma mode
#else // Linear values
#define unity_ColorSpaceGrey fixed4(0.214041144, 0.214041144, 0.214041144, 0.5)
#define unity_ColorSpaceDouble fixed4(4.59479380, 4.59479380, 4.59479380, 2.0)
#define unity_ColorSpaceDielectricSpec half4(0.04, 0.04, 0.04, 1.0 - 0.04) //
standard dielectric reflectivity coef at incident angle (= 4%)
#define unity_ColorSpaceLuminance half4(0.0396819152, 0.458021790,
0.00609653955, 1.0) // Legacy: alpha is set to 1.0 to specify linear mode
#endif
```

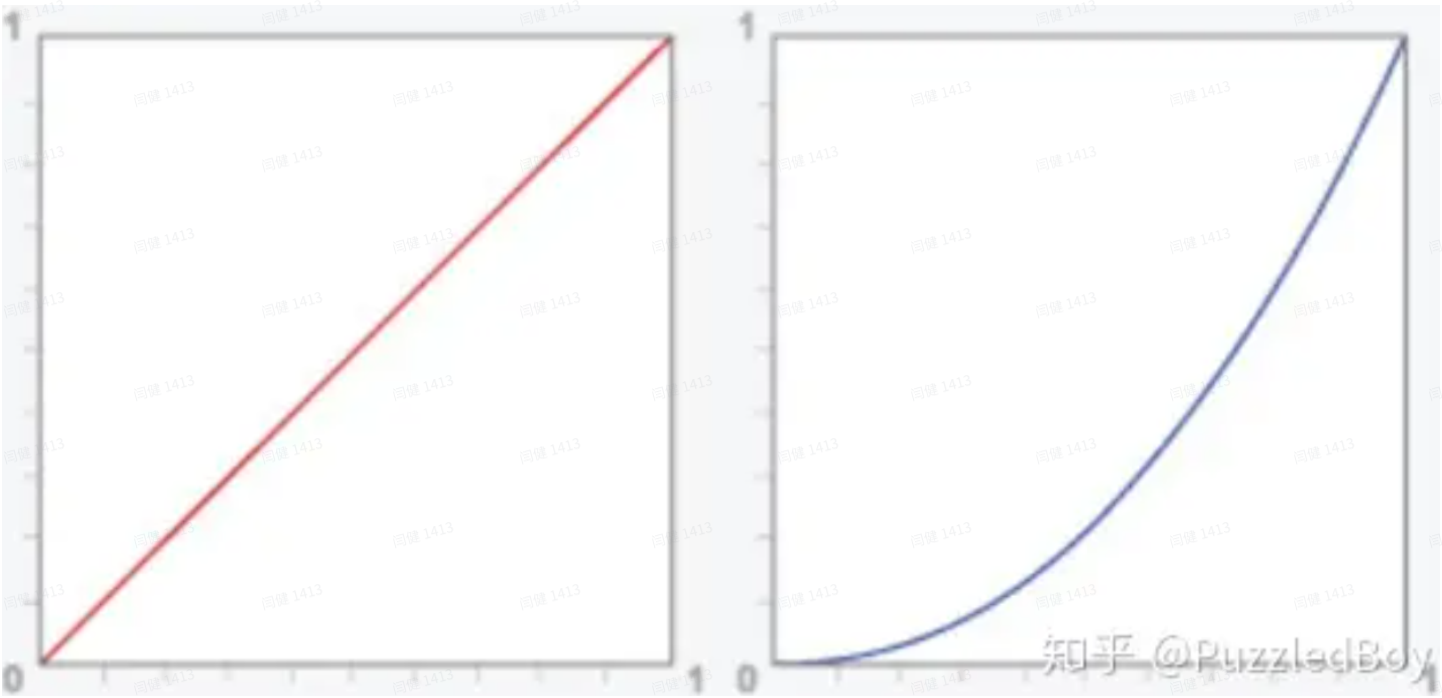
这些是针对不同的颜色空间下定义的不同常量，这里我们需要引入一些概念来辅助理解Gamma空间和Linear空间。

Gamma、Linear、sRGB和伽马校正

在物理世界中，如果光的强度增加一倍，则亮度也会增加一倍，这是线性关系，而历史上最早的显示器(阴极射线管)显示图像的时候，电压增加一倍，亮度并不跟着增加一倍，即输出亮度和电压并不是线性关系的，而是等于电压增加量的2.2次幂的非线性关系：

$$l = u^{2.2} \qquad (l \in [0,1], u \in [0,1])$$

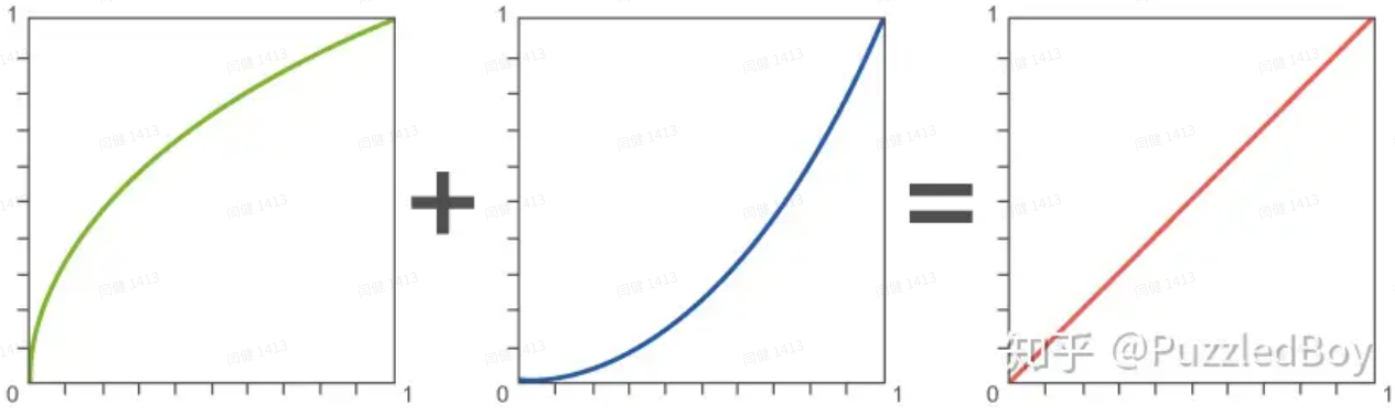
而2.2也称作该显示器的Gamma值，现代显示器的Gamma值也大都是2.2。这意味着当电压线性变化时，相对于人眼的视觉，亮度的变化在暗处较慢，暗色占据的数据范围更广，整体颜色会偏暗，下图分别是代表物理世界的线性空间（Linear Space）和代表显示器输出的Gamma2.2空间（Gamma Space）：



正常情况下，人眼看物理世界的亮度是正常亮度，此时你在看到显示器输出的颜色后，相当于走了一次Gamma2.2曲线的调整，颜色就变暗了。所以在显示器输出之前，做一次操作把显示器的Gamma2.2的影响平衡掉，哪看到的颜色就和人眼直接观察物理世界一样了；而这个平衡的操作就叫做伽马校正。

在数学上，伽马校正是一个约0.45的幂运算（和上面的2.2次幂互为逆运算）：

$$c_o = c_i^{\frac{1}{2.2}}$$



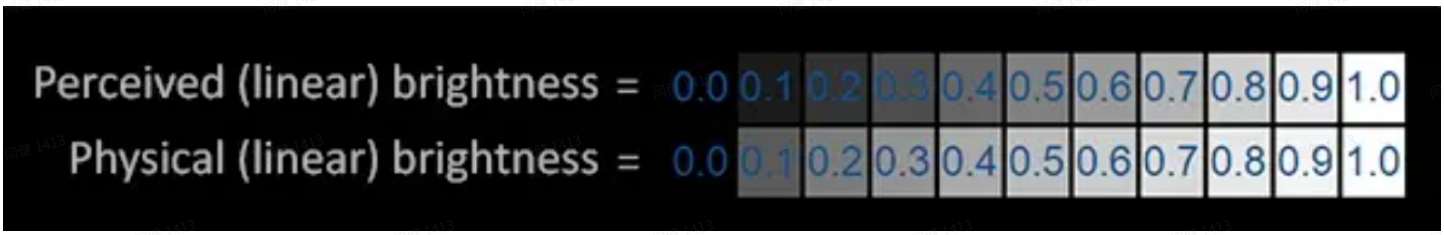
左(Gamma0.45) 中(Gamma2.2) 右(线性物理空间)

经过0.45幂运算，再由显示器经过2.2次幂输出，最后的颜色就和实际物理空间的一致了。

最后，sRGB是1996年，微软和惠普一起开发了一种标准sRGB色彩空间。这种标准得到许多业界厂商的支持。sRGB对应的是Gamma0.45所在的空间，为啥sRGB在Gamma0.45空间呢?

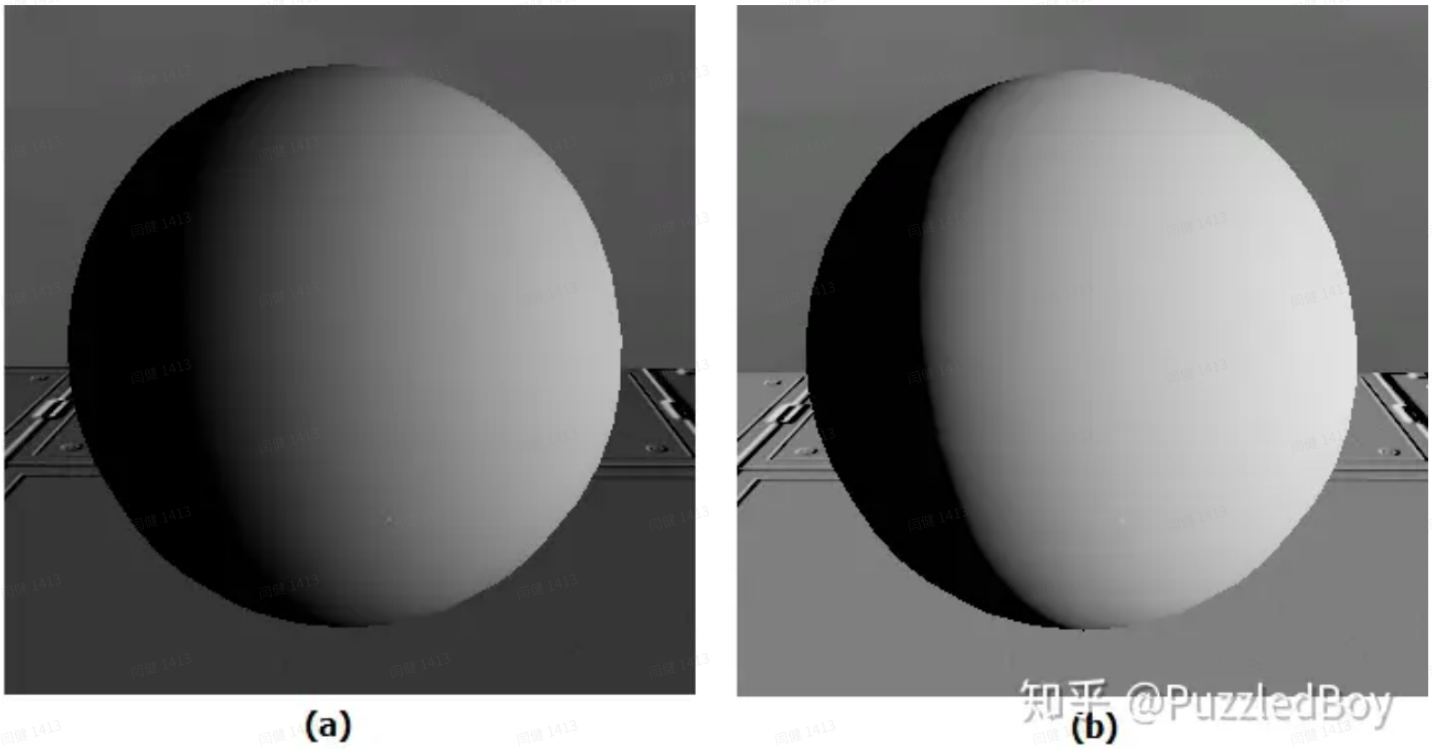
假设现在有一张图片，在你看到的图片和物理世界没有区别，因为此时屏幕已经对图片做了一次Gamma2.2处理，那图片必须要保存在经过Gamma2.2处理之前的空间里，才能被正确的显式颜色，所以反推一下，照片只能保存在Gamma0.45空间，由此经过Gamma2.2的矫正后，才能和人眼看到的一样；换言之，**sRGB相当于对物理空间的颜色做了一次伽马校正**。

另一种解释是，在真实世界中，如果广德强度从0.0逐步增加到1.0，那么亮度应该是线性增加的，如图：



但是对于人眼来说，感知到的亮度变化确不是线性的，而是在暗的地方有更多的细节，换句话说，**我们应该用更大的数据范围来存储暗色，用较小的数据范围来存储亮色**。这就是sRGB格式做的事情；此外，图片定义在Gamma0.45空间，由于显示器自带Gamma2.2，所以我们不需要额外计算，显示器就能显示正确的颜色。

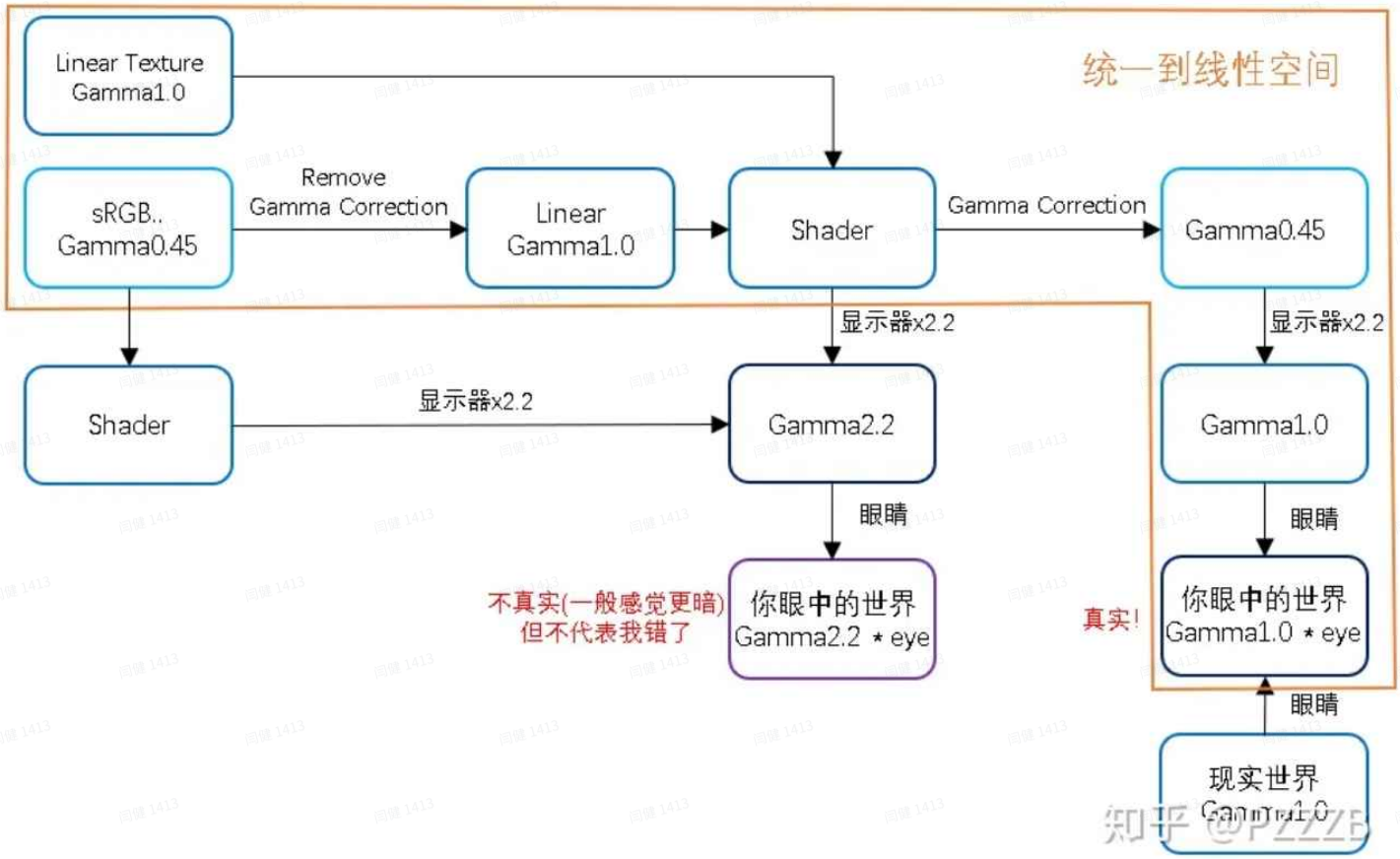
现在，统一回线性空间后，我们可以知道，对于同一张图片，它在Gamma空间和在Linear空间的渲染结果是不同的，在Gamma Space中会偏暗，在Linear Space中渲染会更接近物理世界。



左（Gamma Space），右（Linear Space）

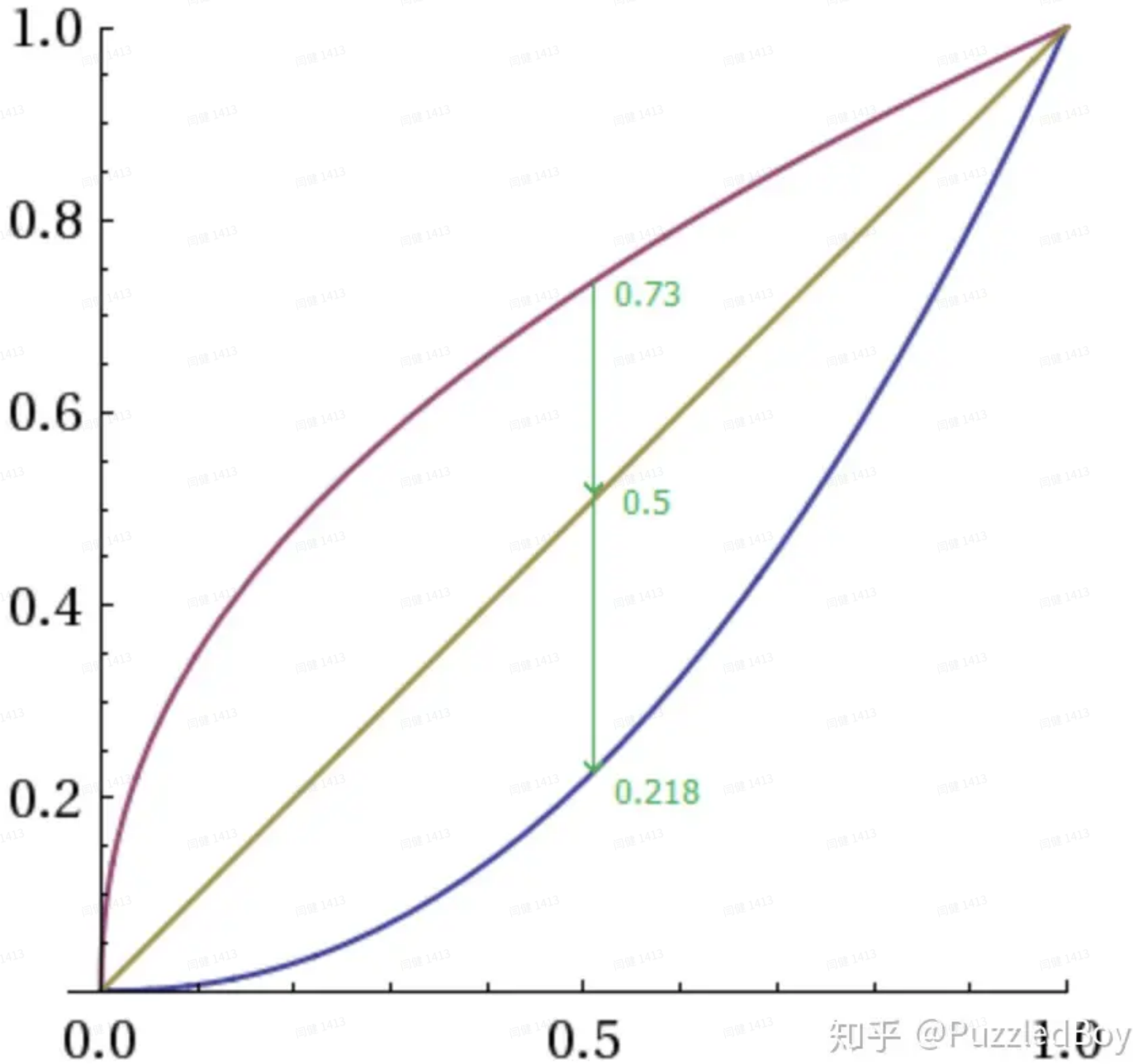
那为什么Linear Space更真实，可以这么想，我们在物理世界中的颜色和光照规律都是在线性空间描述的，而计算机图形学是用的物理世界视觉的数学模型，Shader中的颜色插值、光照计算也都是在线性空间描述的，当使用非线性空间的输入(sRGB-Gamma0.45)，又在线性空间进行计算，那结果当然不自然；

所以，如果所有的输入、计算、输出都能统一在线性空间里，那结果就是真实的；（比如是高画质要求的3A写实风大作，那就必须要统一，但如果追求非真实渲染，也未必要统一。）渲染统一到线性空间的过程看起来是这样的：



1. 第一步，输入的纹理如果是sRGB（Gamma0.45），那我们需要进行一个操作转换到线性空间，这个操作视角Remove Gamma Correction，在数学上是一个2.2的幂运算，如果驶入不是在sRGB而是已经在线性空间，就可以跳过Remove Gamma Correction了。（美术输出资源时都是在sRGB空间的，但Normal Map等其他电脑计算出来的纹理则一般在线性空间，即Linear Texture。）
2. 第二步，现在输入已经在线性空间了，那么进行Shader中光照、插值等计算后就是比较真实的结果了；如果不对sRGB进行Remove Gamma Correction就直接进入Shader计算，那结果就会不自然。
3. 第三步，Shader计算完成后，需要进行Gamma Correction，从线性空间变换到Gamma0.45空间，在数学上是一个约为0.45的幂运算，如果不进行Gamma Correction的输出，颜色就会从线性空间转换到Gamma2.2空间，再被人眼看到，结果就会更暗。
4. 第四步，经过了前面的Gamma Correction，显示器输出在了线性空间，这就和人眼看物理世界的过程是一样的了。

假设在sRGB纹理中有一个像素，其值为0.73，在统一线性空间的过程中，它的变化是：



第一步，0.73(上曲线) * [Remove Gamma Correction] = 0.5(直线)。（$0.73^{2.2} = 0.5$）

第二步，0.5(直线) * [Shader] = 0.5(直线)（假设我们的Shader啥也不干保持颜色不变）

第三步，0.5(直线) * [Gamma Correction] = 0.73(上曲线)。（$0.5^{\frac{1}{2.2}} = 0.73$）

第四步，0.73(上曲线) * [显示器] = 0.5(直线)。（$0.73^{2.2} = 0.5$）

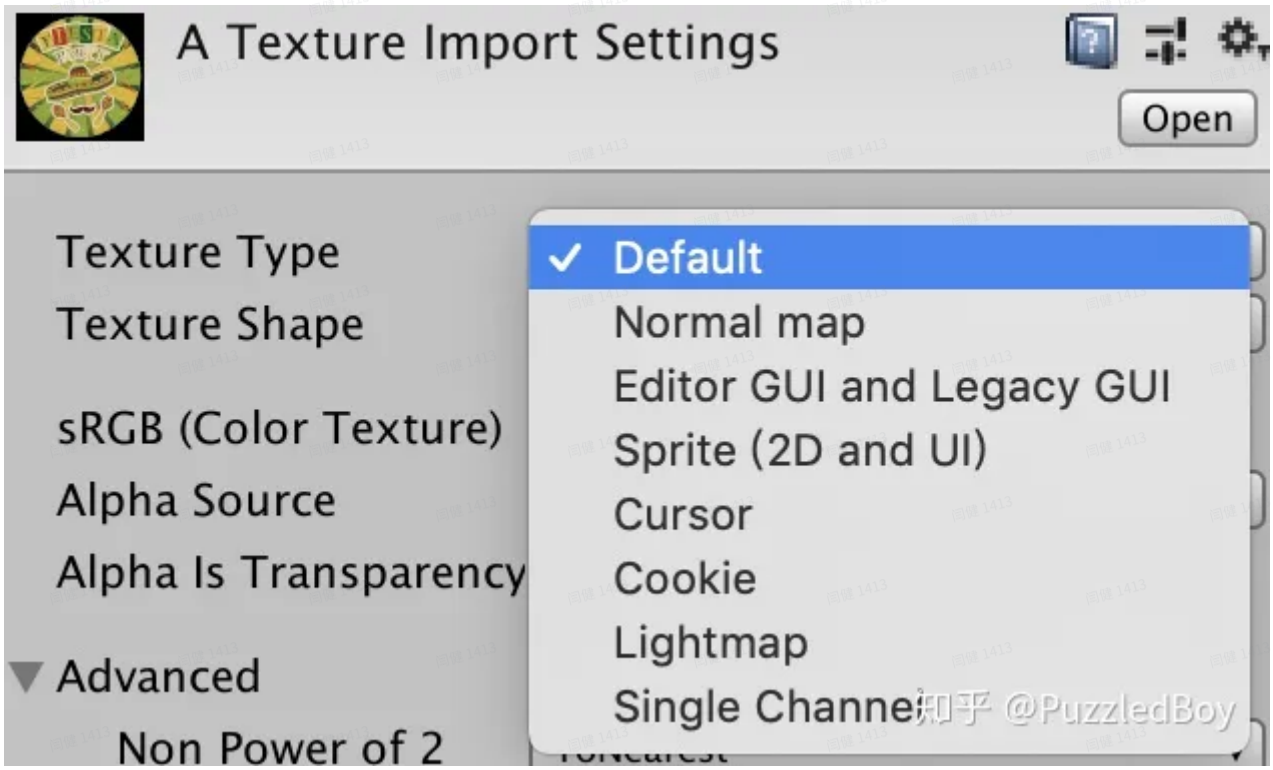
Unity中的Color Space

引申概念已经讲解清楚了，现在我们回到Unity，你可以在ProjectSetting中选择Gamma或者Linear作为Color Space：

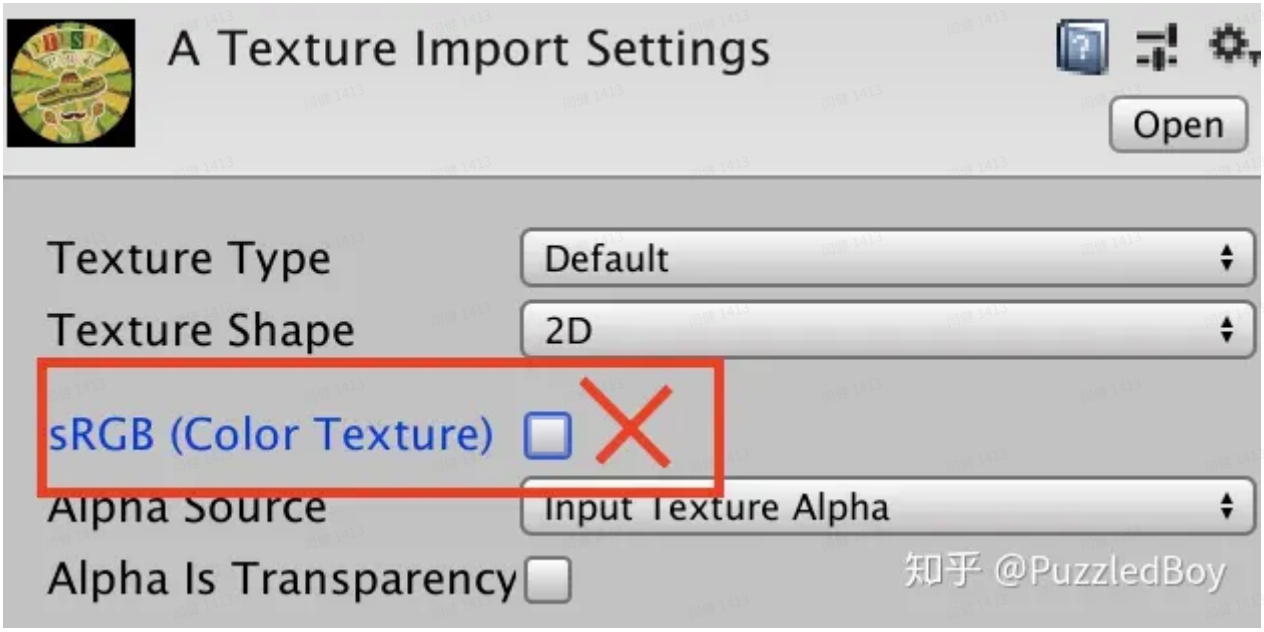
这两者的区别是：

1. 如果选择了Gamma，那Unity不会对输入和输出做任何处理，换句话说，Remove Gamma Correction和Gamma Correction都不会发生，除非你自己手动实现。
2. 如果选择了Linear，那就是之前提到的统一线性空间的流程了。对于sRGB纹理，Unity在进行纹理采样前会自动进行Remove Gamma Correction，对于Linear纹理则没有这一步；而在输出前，Unity会自动进行Gamma Correction再让显示器输出。

我们可以通过直接设置纹理所属的类型，来让Unity自己处理相关的计算，比如Normal Map、Light Map都是Linear；



再比如一些纹理均不属于上面的任何类型，但是它本身就在Linear Space（比如Mask纹理，噪声图），那需要我们取消sRGB选项，来跳过Remove Gamma Correction过程；



或者可以通过另一种统一的解释来看：所有需要人眼参与而被创作出来的纹理都应该是sRGB，而所有计算机计算生成的纹理，都应该是Linear。

此外，在Linear Space下，ShaderLab中的颜色输入也会被认为是sRGB颜色，会自动进行Gamma Correction Removed；或者想要一个Float变量也进行Gamma Correction Removed，那就需要在ShaderLab中使用 `[Gamma]` 前缀：

```
    [Gamma]_Metallic("Metallic",Range(0,1))=0
```

上面是来自于Standard Shader的源代码，其中_Metallic这一项就带了[Gamma]前缀，表示在Linear Space下，Unity要将其认为在sRGB空间，从而进行Remove Gamma Correction。

扩展：为什么官方源代码中_Metallic项需要加[Gamma]？这和底层的光照计算中考虑能量守恒的部分有关，Metallic代表了物体的“金属度”，如果值越大则反射(高光)越强，漫反射会越弱。在实际的计算中，这个强弱的计算和Color Space有关，所以需要加上[Gamma]项。

虽然Linear是最真实的，但是Gamma毕竟少了中间处理，渲染开销会更低，效率会更高。上文也说过不真实不代表是错的，毕竟图形学第一定律：如果它看上去是对的，那么它就是对的。

注：在Android上，Linear只在OpenGL ES 3.0和Android 4.3以上支持，iOS则只有Metal才支持。

在早期移动端上不支持Linear Space流程，所以需要考虑更多。不过随着现在手机游戏的发展，越来越多追求真实的项目出现，很多项目都选择直接在Linear Space下工作。

> 原文来源：Gamma、Linear、sRGB 和Unity Color Space，你真懂了吗？ - PZZZB的文章 - 知乎
> https://zhuanlan.zhihu.com/p/66558476

关于上面的内容，在UnityCG.cginc的Gamma和Linear空间转换相关的函数中也可见一斑：

```
// Legacy for compatibility with existing shaders
inline bool IsGammaSpace()
{
    #ifdef UNITY_COLORSPACE_GAMMA
        return true;
    #else
        return false;
    #endif
}

inline float GammaToLinearSpaceExact (float value)
{
    if (value <= 0.04045F)
        return value / 12.92F;
    else if (value < 1.0F)
        return pow((value + 0.055F)/1.055F, 2.4F);
    else
        return pow(value, 2.2F);
}

inline half3 GammaToLinearSpace (half3 sRGB)
{
    // Approximate version from http://chilliant.blogspot.com.au/2012/08/srgb-approximations-for-hlsl.html?m=1
    return sRGB * (sRGB * (sRGB * 0.305306011h + 0.682171111h) + 0.012522878h);

    // Precise version, useful for debugging.
    //return half3(GammaToLinearSpaceExact(sRGB.r), GammaToLinearSpaceExact(sRGB.g), GammaToLinearSpaceExact(sRGB.b));
}

inline float LinearToGammaSpaceExact (float value)
{
    if (value <= 0.0F)
        return 0.0F;
    else if (value <= 0.0031308F)
        return 12.92F * value;
    else if (value < 1.0F)
        return 1.055F * pow(value, 0.4166667F) - 0.055F;
    else
        return pow(value, 0.45454545F);
}

inline half3 LinearToGammaSpace (half3 linRGB)
{
    linRGB = max(linRGB, half3(0.h, 0.h, 0.h));
```

```
    // An almost-perfect approximation from
http://chilliant.blogspot.com.au/2012/08/srgb-approximations-for-hlsl.html?m=1
    return max(1.055h * pow(linRGB, 0.416666667h) - 0.055h, 0.h);

    // Exact version, useful for debugging.
    //return half3(LinearToGammaSpaceExact(linRGB.r),
LinearToGammaSpaceExact(linRGB.g), LinearToGammaSpaceExact(linRGB.b));
}
```

经过上面对Gamma和Linear空间的引申，相信这段代码已不用再赘述。

着色器的输入或输出结构

```
//顶点着色器的输入
struct appdata_base {
    //顶点坐标
    float4 vertex : POSITION;
    //顶点法线
    float3 normal : NORMAL;
    //第一套纹理坐标
    float4 texcoord : TEXCOORD0;
    //用于GPU Instance的SV_InstanceID
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
//顶点着色器的输入
struct appdata_tan {
    float4 vertex : POSITION;
    //顶点切线
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
//顶点着色器的输入
struct appdata_full {
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    //四套(或更多套)纹理坐标
    float4 texcoord1 : TEXCOORD1;
    float4 texcoord2 : TEXCOORD2;
    float4 texcoord3 : TEXCOORD3;
    //顶点颜色
```

```
    fixed4 color : COLOR;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
//顶点着色器的输入
struct appdata_img
{
    float4 vertex : POSITION;
    //用于纹理坐标，所以是half2即可
    half2 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
//顶点着色器的输出
struct v2f_img
{
    //裁剪空间中的位置
    float4 pos : SV_POSITION;
    //纹理坐标
    half2 uv : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
    //单通道实例化渲染，XR相关用的
    UNITY_VERTEX_OUTPUT_STEREO
};
//顶点着色器的输出
//该结构体用了一层纹理，另外指定了两种颜色，用来模拟漫反射颜色和镜面反射颜色
struct v2f_vertex_lit {
    float2 uv   : TEXCOORD0;
    fixed4 diff : COLOR0;
    fixed4 spec : COLOR1;
};
```

线性变换和仿射变换

```
//世界空间转裁剪空间
inline float4 UnityWorldToClipPos( in float3 pos )
{
    return mul(UNITY_MATRIX_VP, float4(pos, 1.0));
}

//视图(摄像机)空间转裁剪空间
inline float4 UnityViewToClipPos( in float3 pos )
{
    return mul(UNITY_MATRIX_P, float4(pos, 1.0));
}
```

```
//模型空间转视图空间
inline float3 UnityObjectToViewPos( in float3 pos )
{
    return mul(UNITY_MATRIX_V, mul(unity_ObjectToWorld, float4(pos, 1.0))).xyz;
}
//上面函数的重载
inline float3 UnityObjectToViewPos(float4 pos) // overload for float4; avoids
"implicit truncation" warning for existing shaders
{
    return UnityObjectToViewPos(pos.xyz);
}

//世界空间转视图空间
inline float3 UnityWorldToViewPos( in float3 pos )
{
    return mul(UNITY_MATRIX_V, float4(pos, 1.0)).xyz;
}

//模型空间中的某个向量转世界空间
inline float3 UnityObjectToWorldDir( in float3 dir )
{
    return normalize(mul((float3x3)unity_ObjectToWorld, dir));
}

//世界空间中的某个向量转模型空间
inline float3 UnityWorldToObjectDir( in float3 dir )
{
    return normalize(mul((float3x3)unity_WorldToObject, dir));
}

//模型空间的法线转世界空间
//之前我们已经推倒过，如果把顶点从模型空间转换到世界空间的变换矩阵是
unity_ObjectToWorld，
//则变换法线的矩阵应是unity_ObjectToWorld的逆转置矩阵，写法如下
inline float3 UnityObjectToWorldNormal( in float3 norm )
{
#ifdef UNITY_ASSUME_UNIFORM_SCALING
    return UnityObjectToWorldDir(norm);
#else
    // mul(IT_M, norm) => mul(norm, I_M) => {dot(norm, I_M.col0), dot(norm,
I_M.col1), dot(norm, I_M.col2)}
    return normalize(mul(norm, (float3x3)unity_WorldToObject));
#endif
}

//计算世界空间的坐标和同空间下的引擎内置光源的坐标_WorldSpaceLightPos0连线形成的向量
inline float3 UnityWorldSpaceLightDir( in float3 worldPos )
```

```
{
    #ifndef USING_LIGHT_MULTI_COMPILE
        return _WorldSpaceLightPos0.xyz - worldPos * _WorldSpaceLightPos0.w;
    #else
        #ifndef USING_DIRECTIONAL_LIGHT
        return _WorldSpaceLightPos0.xyz - worldPos;
        #else
        return _WorldSpaceLightPos0.xyz;
        #endif
    #endif
}

//先把localPos转换到世界坐标下，然后执行上面那个函数(该函数已禁用，保留只是为了兼容)
// *Legacy* Please use UnityWorldSpaceLightDir instead
inline float3 WorldSpaceLightDir( in float4 localPos )
{
    float3 worldPos = mul(unity_ObjectToWorld, localPos).xyz;
    return UnityWorldSpaceLightDir(worldPos);
}

//和UnityWorldSpaceLightDir相似，只是把_WorldSpaceLightPos0转到了模型空间，然后计算向量
inline float3 ObjSpaceLightDir( in float4 v )
{
    float3 objSpaceLightPos = mul(unity_WorldToObject, _WorldSpaceLightPos0).xyz;
    #ifndef USING_LIGHT_MULTI_COMPILE
        return objSpaceLightPos.xyz - v.xyz * _WorldSpaceLightPos0.w;
    #else
        #ifndef USING_DIRECTIONAL_LIGHT
        return objSpaceLightPos.xyz - v.xyz;
        #else
        return objSpaceLightPos.xyz;
        #endif
    #endif
}

//世界空间下，某点到摄像机_WorldSpaceCameraPos的方向向量
inline float3 UnityWorldSpaceViewDir( in float3 worldPos )
{
    return _WorldSpaceCameraPos.xyz - worldPos;
}

//同UnityWorldSpaceViewDir
// *Legacy* Please use UnityWorldSpaceViewDir instead
inline float3 WorldSpaceViewDir( in float4 localPos )
{
```

```
    float3 worldPos = mul(unity_ObjectToWorld, localPos).xyz;
    return UnityWorldSpaceViewDir(worldPos);
}

//把_WorldSpaceCameraPos变换到模型空间后计算向量
inline float3 ObjSpaceViewDir( in float4 v )
{
    float3 objSpaceCameraPos = mul(unity_WorldToObject,
float4(_WorldSpaceCameraPos.xyz, 1)).xyz;
    return objSpaceCameraPos - v.xyz;
}

// Declares 3x3 matrix 'rotation', filled with tangent space basis
//这个宏的作用公式定义一个类型为float3x3的矩阵;
//这个矩阵有顶点法线，切线、以及与顶点的法线和切线都互相垂直的副法线，构成了一个正交的切线
空间
#define TANGENT_SPACE_ROTATION \
    float3 binormal = cross( normalize(v.normal), normalize(v.tangent.xyz) ) *
v.tangent.w; \
    float3x3 rotation = float3x3( v.tangent.xyz, binormal, v.normal )
```

与光照相关的工具函数

```
// Used in ForwardBase pass: Calculates diffuse lighting from 4 point lights,
with data packed in a special way.
//函数将用在ForwardBase类型的渲染通道上
float3 Shade4PointLights (
    float4 lightPosX, float4 lightPosY, float4 lightPosZ,
    float3 lightColor0, float3 lightColor1, float3 lightColor2, float3
lightColor3,
    float4 lightAttenSq,
    float3 pos, float3 normal)
{
    // to light vectors
    //一次性计算顶点到每一个光源之间的x坐标差，y坐标差，z坐标差
    float4 toLightX = lightPosX - pos.x;
    float4 toLightY = lightPosY - pos.y;
    float4 toLightZ = lightPosZ - pos.z;
    // squared lengths
    //一次性计算顶点到每一个光源的距离的平方
    float4 lengthSq = 0;
    lengthSq += toLightX * toLightX;
    lengthSq += toLightY * toLightY;
    lengthSq += toLightZ * toLightZ;
```

```
    // don't produce NaNs if some vertex position overlaps with the light
    //如果顶点距离光源太近了，就微调一个很小的数作为他们的距离
    lengthSq = max(lengthSq, 0.000001);

    // NdotL
    //计算顶点到四个光源连线，以及顶点法线normal的夹角的余弦值，即顶点到四个光源在法线的投影
    float4 ndotl = 0;
    ndotl += toLightX * normal.x;
    ndotl += toLightY * normal.y;
    ndotl += toLightZ * normal.z;
    // correct NdotL
    //因为由toLightX、toLightY、toLightZ组成的顶点到是个光源连线的向量是没有经过单位化的
    //所以nodtl变量中的每一个分量必须除以lengthSq变量中的每一个分量，即顶点到每一个光源的距离的平方
    float4 corr = rsqrt(lengthSq);
    ndotl = max (float4(0,0,0,0), ndotl * corr);
    // attenuation
    //计算出从光源到顶点位置的光的衰减值
    float4 atten = 1.0 / (1.0 + lengthSq * lightAttenSq);
    //衰减值再乘以夹角余弦值
    float4 diff = ndotl * atten;
    // final color
    //计算出最终的颜色
    float3 col = 0;
    col += lightColor0 * diff.x;
    col += lightColor1 * diff.y;
    col += lightColor2 * diff.z;
    col += lightColor3 * diff.w;
    return col;
}
```

Shade4PointLights函数用在顶点着色器的ForwardBase渲染通道上，本函数在每一个顶点被4个点光源照亮时，利用兰伯特光照模型计算出光照的漫反射效果。

```
// Used in Vertex pass: Calculates diffuse lighting from lightCount lights.
Specifying true to spotLight is more expensive
// to calculate but lights are treated as spot lights otherwise they are
treated as point lights.
//本函数用在顶点着色器中，计算出光源产生的漫反射光照效果
//float4 vertex 顶点的位置坐标
//float4 normal 顶点的法线
//float4 lightCount 参与光照计算的光源数量
//bool spotLight 光源是不是聚光灯光源
```

```
float3 ShadeVertexLightsFull (float4 vertex, float3 normal, int lightCount,
bool spotLight)
{
    //Unity3D提供的光源位置和光线传播方向在当前摄像机所构成的视图空间中
    //所以先把传递进来的顶点坐标变换到视图空间，顶点的法线也乘以model-view的逆转置矩阵到
视图空间
    float3 viewpos = UnityObjectToViewPos (vertex.xyz);
    float3 viewN = normalize (mul ((float3x3)UNITY_MATRIX_IT_MV, normal));
    //UNITY_LIGHTMODEL_AMBIENT 在UnityShaderVariables.cginc文件中定义
    //#define UNITY_LIGHTMODEL_AMBIENT (glstate_lightmodel_ambient * 2)
    float3 lightColor = UNITY_LIGHTMODEL_AMBIENT.xyz;
    for (int i = 0; i < lightCount; i++) {
        //如果unity_LightPosition[i]对应的光源是有向平行光，
        //则unity_LightPosition[i].w的值为0，unity_LightPosition[i].xyz就是光的方
向;
        //如果不是有向平行光，则w值为1，unity_LightPosition[i].xyz是光源在观察空间中
的位置坐标。
        //总之，toLight就是顶点位置到光源位置的连线的方向向量。
        float3 toLight = unity_LightPosition[i].xyz - viewpos.xyz *
unity_LightPosition[i].w;
        //toLight自身的点积实际上就是顶点到光源的距离的平方
        float lengthSq = dot(toLight, toLight);

        // don't produce NaNs if some vertex position overlaps with the light
        lengthSq = max(lengthSq, 0.000001);
        //求出距离的倒数
        toLight *= rsqrt(lengthSq);
        //光源的衰减计算
        float atten = 1.0 / (1.0 + lengthSq * unity_LightAtten[i].z);
        if (spotLight)
        {
            float rho = max (0, dot(toLight, unity_SpotDirection[i].xyz));
            float spotAtt = (rho - unity_LightAtten[i].x) *
unity_LightAtten[i].y;
            atten *= saturate(spotAtt);
        }

        float diff = max (0, dot (viewN, toLight));
        lightColor += unity_LightColor[i].rgb * (diff * atten);
    }
    return lightColor;
}
```

上述代码中，变量rho由顶点到光源连线方向toLight与聚光灯光源的正前照射方向unity_SpotDirection求点积而得。也就是说，rho为toLight方向与unity_SpotDirection方向的夹角余弦值cos(p)。

```
//用4个非聚光灯光源进行光照计算
float3 ShadeVertexLights (float4 vertex, float3 normal)
{
    return ShadeVertexLightsFull (vertex, normal, 4, false);
}


// Transforms 2D UV by scale/bias property
//用顶点中的纹理映射坐标tex与目标纹理name##_ST的xy(Tiling平铺值)和zw(Offset偏移值)做运算操作。
//Tiling的默认值为(1,1)，Offset的默认值为(0,0)
//也就是说，如果有一张名为Sample的纹理，要是用文理的Tiling和Offset属性
//就还必须定义一个Sample_ST的四维向量(float4、half4、fixed4)才可以。
#define TRANSFORM_TEX(tex,name) (tex.xy * name##_ST.xy + name##_ST.zw)

// Deprecated. Used to transform 4D UV by a fixed function texture matrix. Now just returns the passed UV.
#define TRANSFORM_UV(idx) v.texcoord.xy

//VertexLight是一个简单的顶点光照计算函数，其颜色计算方式就是用顶点漫反射颜色乘以纹理颜色
//然后加上纹素的Alpha值与顶点镜面反射颜色，两者之和就是最终的颜色
inline fixed4 VertexLight(v2f_vertex_lit i, sampler2D mainTex)
{
    fixed4 texcol = tex2D(mainTex, i.uv);
    fixed4 c;
    c.xyz = ( texcol.xyz * i.diff.xyz + i.spec.xyz * texcol.a );
    c.w = texcol.w * i.diff.w;
    return c;
}

// Calculates UV offset for parallax bump mapping
//函数根据当前片元对应的高度图中的高度值h，以及高度缩放系数height和切线空间中片元到摄像机的向量
//计算到当前片元实际上要使用外观纹理的哪一点纹理
inline float2 ParallaxOffset( half h, half height, half3 viewDir )
{
    h = h * height - height/2.0;
    float3 v = normalize(viewDir);
    v.z += 0.42;
    return h * (v.xy / v.z);
}

// Converts color to luminance (grayscale)
//该函数把一个RGB颜色值转化成亮度值，当前的RGB颜色值基于伽马空间或者线性空间，得到的亮度值有不同的结果
inline half Luminance(half3 rgb)
```

```
{
    return dot(rgb, unity_ColorSpaceLuminance.rgb);
}

// Convert rgb to luminance
// with rgb in linear space with sRGB primaries and D65 white point
//把在线性空间中的颜色RGB值转换成亮度值
//它实质上就是把一个基于RGB颜色空间的颜色值变换到CIE1931-Yxy颜色空间中得到对应的亮度值y。
half LinearRgbToLuminance(half3 linearRgb)
{
    return dot(linearRgb, half3(0.2126729f,  0.7151522f, 0.0721750f));
}
```

与HDR及光照贴图颜色编解码相关的工具函数

高动态范围(HDR)光照是一种用来实现超过了显示器所能表现的亮度范围的渲染技术。如果采用8位通道存储每一个颜色的RGB分量，则每个分量的亮度级别只有256种。显然256个亮度级别是不足以描述自然界中的亮度差别的情况的，如太阳的亮度可能是一个白炽灯亮度的数千倍，将远远超出当前显示器的亮度表示能力。

假如房间中刺眼的阳光从窗外照射进来，普通渲染方法是把阳光和白色墙的颜色都视为白色，RGB(255,255,255)，尽管都是白色，但阳光肯定比白墙刺眼的多。所以应用HDR技术对亮度进行处理，使得他们的亮度能够体现出明显的差异。HDR技术就是把尽可能大的亮度值范围编码到尽可能小的存储空间中。

把大数字范围编码到小数字范围的简单方式，就是把大范围中的数字乘以一个缩小系数。线性映射到小范围上，这种方法虽然能表示的亮度范围扩大了，但却导致了颜色带状阶跃的问题。

所以实际上HDR实现一般遵循以下几步：

1. 在每个颜色通道是16位或者32位的浮点纹理或者渲染模板上渲染当前的场景。
2. 使用RGBM、LogLuv等编码方式来节省所需的内存和带宽。
3. 通过降采样计算场景的亮度。
4. 根据场景亮度值对场景做一个色调映射，将最终颜色值输出到一个每通道8位的RBG格式的渲染目标上。

RGBM(M: shared multiplier)，是为了解决精度不足以存储亮度范围信息的问题，可以创建一个精度更高的浮点值渲染目标；但使用高精度的浮点渲染目标会带来另一个问题，即需要更高的内存存储空间和更高的带宽，并且有些渲染硬件无法以操作8位精度的渲染目标的速度去操作16位浮点渲染目标。

为了解决这个问题，需要采用一种编码方式将这些颜色数据编码成一个能以8位颜色分量存储的数据。编码方式有多种，如RGBM编码，LogLuv编码等。加入有一个给定的包含了RGB颜色分量的颜色值

color，定义了一个编码后取值"最大范围值"的MaxRGBM，将其编码成一个含有R、G、B、M四个分量的颜色值的步骤，就如UnityCG.cginc中提供的UnityEncodeRGBM所示：

```
half4 UnityEncodeRGBM (half3 color, float maxRGBM)
{
    float kOneOverRGBMMaxRange = 1.0 / maxRGBM;
    const float kMinMultiplier = 2.0 * 1e-2;
    //将color的RGB分量各自除以MaxRGBM，然后取得最大商
    float3 rgb = color * kOneOverRGBMMaxRange;
    float alpha = max(max(rgb.r, rgb.g), max(rgb.b, kMinMultiplier));
    //用最大商，乘255得到结果值后，向上取整，再将这个数除以255后，再赋值给alpha
    alpha = ceil(alpha * 255.0) / 255.0;
    //最小的multiplier控制在0.02
    // Division-by-zero warning from d3d9, so make compiler happy.
    alpha = max(alpha, kMinMultiplier);

    return half4(rgb / alpha, alpha);
}



// Decodes HDR textures
// handles dLDR, RGBM formats
inline half3 DecodeHDR(half4 data, half4 decodeInstructions, int
colorspaceIsGamma)
{
    // Take into account texture alpha if decodeInstructions.w is true(the
alpha value affects the RGB channels)
    //当w分量的值为1(true)，则需要考虑HDR纹理中的alpha值对纹理的RGB值的影响，此时alpha
变量的值就是纹理的aplha值
    //当w分量的值为0(false)，则alpha的值始终为1
    half alpha = decodeInstructions.w * (data.a - 1.0) + 1.0;

    // If Linear mode is not supported we can skip exponent part
    if(colorspaceIsGamma)
        return (decodeInstructions.x * alpha) * data.rgb;
    //如果使用线性工作流，则使用decodeInstructions.x乘以alpha的decodeInstructions.y
次方
    return (decodeInstructions.x * pow(alpha, decodeInstructions.y)) *
data.rgb;
}

// Decodes HDR textures
// handles dLDR, RGBM formats
inline half3 DecodeHDR (half4 data, half4 decodeInstructions)
{
    #if defined(UNITY_COLORSPACE_GAMMA)
```

```
    return DecodeHDR(data, decodeInstructions, 1);
    #else
    return DecodeHDR(data, decodeInstructions, 0);
    #endif
}


// Decodes HDR textures
// handles dLDR, RGBM formats
//该函数把一个RGBM的颜色值解码成一个每个通道8位的RGB颜色值
inline half3 DecodeLightmapRGBM (half4 data, half4 decodeInstructions)
{
    // If Linear mode is not supported we can skip exponent part
    #if defined(UNITY_COLORSPACE_GAMMA)
    # if defined(UNITY_FORCE_LINEAR_READ_FOR_RGBM)
    //如果在Gamma空间下强行使用线性空间的值来解码RGBM，则乘以alpha值，再乘以rgb的平方根
    //还不清楚这里平方根的作用，我推测是和颜色聚类分析或者主色提取相关的内容
        return (decodeInstructions.x * data.a) * sqrt(data.rgb);
    # else
        return (decodeInstructions.x * data.a) * data.rgb;
    # endif
    #else
    //在Linear Space中，这里其实和DecodeHDR比较相似
        return (decodeInstructions.x * pow(data.a, decodeInstructions.y)) *
data.rgb;
    #endif
}


// 解码一个用dLDR编码的光照贴图
inline half3 DecodeLightmapDoubleLDR( fixed4 color, half4 decodeInstructions)
{
    // decodeInstructions.x contains 2.0 when gamma color space is used or
pow(2.0, 2.2) = 4.59 when linear color space is used on mobile platforms
    return decodeInstructions.x * color.rgb;
}

//通过宏控制如何解码一张Lightmap
inline half3 DecodeLightmap( fixed4 color, half4 decodeInstructions)
{
#if defined(UNITY_LIGHTMAP_DLDR_ENCODING)
    return DecodeLightmapDoubleLDR(color, decodeInstructions);
#elif defined(UNITY_LIGHTMAP_RGBM_ENCODING)
    return DecodeLightmapRGBM(color, decodeInstructions);
#else //defined(UNITY_LIGHTMAP_FULL_HDR)
    return color.rgb;
#endif
```

```
}


half4 unity_Lightmap_HDR;

inline half3 DecodeLightmap( fixed4 color )
{
    return DecodeLightmap( color, unity_Lightmap_HDR );
}

half4 unity_DynamicLightmap_HDR;

//解码启发式RGBM编码的光照贴图?
//动态HDR的格式与标准HDR纹理是不同的，如烘焙光照贴图，反射探针和图像照明
//二者在线性空间中，使用不同的幂次方
//此外，三次平方操作，在移动平台上耗的一批
inline half3 DecodeRealtimeLightmap( fixed4 color )
{
    //@TODO: Temporary until Geomerics gives us an API to convert lightmaps to
RGBM in gamma space on the enlighten thread before we upload the textures.
#if defined(UNITY_FORCE_LINEAR_READ_FOR_RGBM)
    return pow ((unity_DynamicLightmap_HDR.x * color.a) * sqrt(color.rgb),
unity_DynamicLightmap_HDR.y);
#else
    return pow ((unity_DynamicLightmap_HDR.x * color.a) * color.rgb,
unity_DynamicLightmap_HDR.y);
#endif
}
```

把高精度数据编码到低精度缓冲器的函数

```
// Encoding/decoding [0..1) floats into 8 bit/channel RGBA. Note that 1.0 will
not be encoded properly.
//如果v=1.0，那么编码将会不正确
inline float4 EncodeFloatRGBA( float v )
{
    //kEncodeMul中的各个分量分别是255的0、1、2、3次幂
    float4 kEncodeMul = float4(1.0, 255.0, 65025.0, 16581375.0);
    float kEncodeBit = 1.0/255.0;
    float4 enc = kEncodeMul * v;
    enc = frac (enc);
    enc -= enc.yzww * kEncodeBit;
    return enc;//返回的是每分量的浮点数数值都在区间[0, 1）内的浮点数
}
```

```
inline float DecodeFloatRGBA( float4 enc )
{
    float4 kDecodeDot = float4(1.0, 1/255.0, 1/65025.0, 1/16581375.0);
    return dot( enc, kDecodeDot );
}
```

来看一下它是如何编码的，后面的其实核心思想相似；

因为有些硬件不支持渲染到浮点纹理，只能将一个浮点数拆分成4个部分，分别存储到RGBA上。同时不支持浮点纹理的硬件往往也不支持整数指令和位运算操作，就只能采用传统的加减乘除四则运算了；

我们已经知道kEncodeMul $\mathrm{kEncodeMul} = (255^0, 255^1, 255^2, 255^3)$
其实就是：

而对于上面的函数，我们希望把传入的v表示成如下形式：

$$\mathrm{v} = \mathrm{v_R} + \frac{1}{255}\mathrm{v_G} + \frac{1}{255^2}\mathrm{v_B} + \frac{1}{255^3}\mathrm{v_A}$$

$$\frac{1}{255} \leq \mathrm{v_R} < 1$$

$$\frac{1}{255} \leq \mathrm{v_G} < 1$$

$$\frac{1}{255} \leq \mathrm{v_B} < 1$$

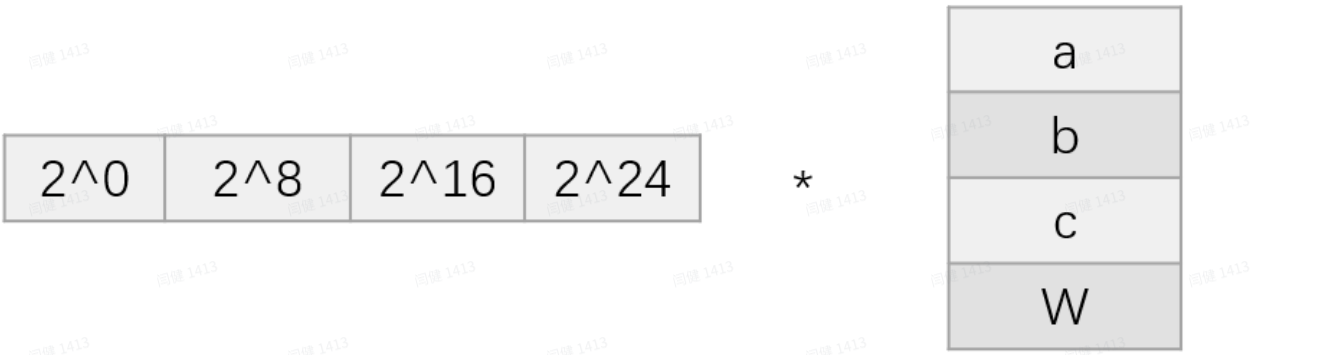
$$\frac{1}{255} \leq \mathrm{v_A} < 1$$

接下来一步步来计算，首先将float4看做是四部分8bit构成：

	25-32	17-24	9-16	1-8
float	a	b	c	d

假设此时选择的RenderTexture格式为R8G8B8A8，则天然地，每个通道可以存储8个bit的数据；四个通道是各自独立的，所以自然想到的办法是对float的四部分数据进行移位，然后分别存储；

首先，将四部分数据移位，可以看到，如果我们将b、c、d分别移位8、16、24位，这样abcd就出现在了同一个维度：



CSDN @manipu1a

这就对应了`float4 enc = float4(1.0, 255.0, 65025.0, 16581375.0) * v;`

最终得到的结果可以看做是：

X	0	0	0	a	b	c	d
Y	0	0	a	b	c	d	0
Z	0	a	b	c	d	0	0
W	a	b	c	d	0	0	0

CSDN @manipu1a

因为第一行是乘1的，所以可以看做是原始数据。那么最后只需要把黄色各自内的数据保存下来就好了。

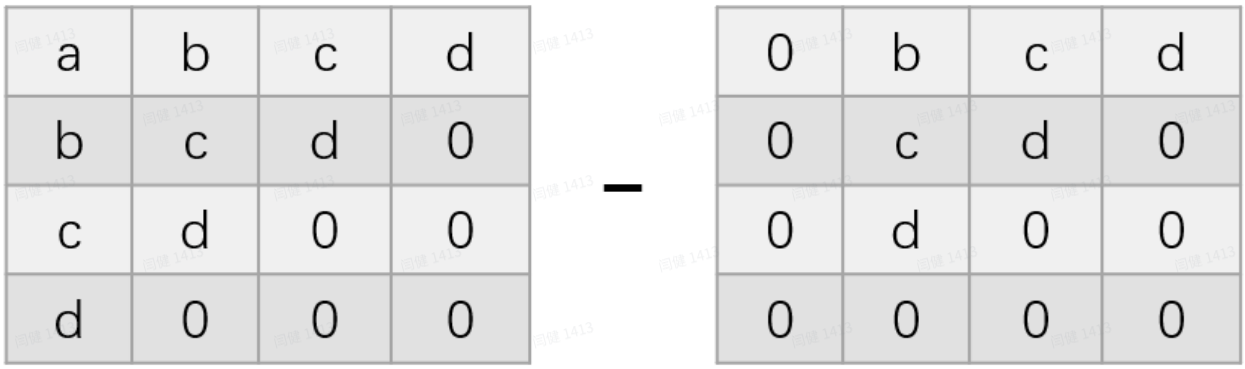
之后，因为我们要处理ShadowMap中的深度，float的取值范围为0~1，所以对于超出32位的数据，我们可以看做是大于1的部分，将这部分数据删除掉，对应**frac**函数：

然后我们要处理在0~24位的数据，有两部分的数据是相同的：

X	0	0	0	a	b	c	d
Y	0	0	0	b	c	d	0
Z	0	0	0	c	d	0	0
W	0	0	0	d	0	0	0

CSDN @manipu1a

所以我们可以继续进行移位，让两部分相减；之前讲float乘以2^n是左移，则除法就是右移了，所以提取出yzw向量除以对应位数即可；(因为之前的数据是xyzw，所以为了匹配，我们用yzww来计算，但是最后一个w乘以的是0)。



CSDN @manipu1a

则这一步对应代码：`enc -= enc.yzww * float4(1.0/255.0,1.0/255.0,1.0/255.0,0.0);`

经过这一步之后，数据就只剩下黄框的部分了，然后再将数据存储进R8G8B8A8就不会丢失精度了；而Decode的思想是类似的，只不过需要反过来。

```
// Encoding/decoding [0..1) floats into 8 bit/channel RG. Note that 1.0 will
not be encoded properly.
//和上面的函数相同，只不过这里用了RG两个通道去编码
inline float2 EncodeFloatRG( float v )
{
    float2 kEncodeMul = float2(1.0, 255.0);
    float kEncodeBit = 1.0/255.0;
    float2 enc = kEncodeMul * v;
    enc = frac (enc);
    enc.x -= enc.y * kEncodeBit;
    return enc;
}
inline float DecodeFloatRG( float2 enc )
{
    float2 kDecodeDot = float2(1.0, 1/255.0);
    return dot( enc, kDecodeDot );
}

// Encoding/decoding view space normals into 2D 0..1 vector
//把一个normal编码进一个flaot4类型的前两个分量
inline float2 EncodeViewNormalStereo( float3 n )
{
    float kScale = 1.7777;
    float2 enc;
    enc = n.xy / (n.z+1);
    enc /= kScale;
    enc = enc*0.5+0.5;
    return enc;
}
```

```
inline float3 DecodeViewNormalStereo( float4 enc4 )
{
    float kScale = 1.7777;
    float3 nn = enc4.xyz*float3(2*kScale,2*kScale,0) + float3(-kScale,-
kScale,1);
    float g = 2.0 / dot(nn.xyz,nn.xyz);
    float3 n;
    n.xy = g*nn.xy;
    n.z = g-1;
    return n;
}

inline float4 EncodeDepthNormal( float depth, float3 normal )
{
    float4 enc;
    //把法线编码到float4类型分量的前两个分量
    enc.xy = EncodeViewNormalStereo (normal);
    //把深度编码进float4类型分量的后两个分量
    enc.zw = EncodeFloatRG (depth);
    return enc;
}

inline void DecodeDepthNormal( float4 enc, out float depth, out float3 normal )
{
    depth = DecodeFloatRG (enc.zw);
    normal = DecodeViewNormalStereo (enc);
}
```

法线贴图及其编码操作的函数

法线贴图存储的信息是对模型顶点法线扰动方向向量，利用此扰动方向向量，在光照计算时对顶点原有的法线进行扰动，从而使法线方向排列有序的平滑表面产生法线方向杂乱无章，从而导致表面凹凸不平的效果。

在Unity3D中导入和使用法线贴图，要设置为Normal map类型，是因为在不同平台上，Unity3D可以利用该平台硬件加速的纹理格式对导入的法线贴图进行压缩，同时也因为法线贴图和普通纹理贴图在采样和解码时的方式也有所不同。

```
//根据宏做分支调用
inline fixed3 UnpackNormal(fixed4 packednormal)
{
#if defined(UNITY_NO_DXT5nm)
    return packednormal.xyz * 2 - 1;
#elif defined(UNITY_ASTC_NORMALMAP_ENCODING)
```

```
        return UnpackNormalDXT5nm(packednormal);
    #else
        return UnpackNormalmapRGorAG(packednormal);
    #endif
    }
```

如果UNITY_NO_DXT5nm启用了，表示引擎使用了DXT5nm压缩格式或者BC5压缩格式的法线贴图纹理，则调用UnpackNormalmapRGorAG函数去解码。

先看UNITY_NO_DXT5nm的情况，这里将法线通道的xyz*2-1的原理是：法线的每个分量，原来都在[-1,1]里，在制作法线贴图的时候，会先加上一个单位向量，变成[0,2]，再除以2变成[0,1]，最终乘以255，可以对应像素顶的[0,255]，从而映射成像素的值区间大小。

这里就是对公式进行逆操作；

DXT是一种纹理压缩格式，以前称为S3TC，当前很多图形硬件已经支持这种格式，即在显存中亦然保持着压缩格式，从而减少显存占用量。目前有DXT1~5这五种编码格式。

DXT系列压缩格式被很多格式的文件所使用的，如DDS文件格式就使用了DXT系列压缩格式。要使用DXT格式压缩图像，要求图像大小至少是4x4纹素，而且图像宽高的纹素个数是2的整数次幂，如32x32、64x128等。

```
    inline fixed3 UnpackNormalDXT5nm (fixed4 packednormal)
    {
        fixed3 normal;
        normal.xy = packednormal.wy * 2 - 1;
        normal.z = sqrt(1 - saturate(dot(normal.xy, normal.xy)));
        return normal;
    }
```

再看UNITY_ASTC_NORMALMAP_ENCODING的情况，该纹理生成的时候，有效值是放在DXT5nm，也就是(1,y,1,x)的形式去存储，将x和y放在纹理的w和y，所以`normal.xy = packednormal.wy * 2 - 1`，x和y的值是由这两个按照第一点的逆推公式得到。

至于z为了节省法线纹理的大小，可以使用公式得到：法线是一个单位向量，长度为1，只要知道了x和y，就能推出z的值：

`sqrt(dot(normal.xyz, normal.xyz)) = 1`；所以`z = sqrt(1 - saturate(dot(normal.xy, normal.xy)))`；这里saturate将点积值取在[0,1]区间。

至于为啥要放在y和w，这是因为这两个通道的bit位数最多；(**RGBA的四个通道的bit位数分别为5、6、5、8**)

```
// Unpack normal as DXT5nm (1, y, 1, x) or BC5 (x, y, 0, 1)
// Note neutral texture like "bump" is (0, 0, 1, 1) to work with both plain
RGB normal and DXT5nm/BC5
fixed3 UnpackNormalmapRGorAG(fixed4 packednormal)
{
    // This do the trick
    packednormal.x *= packednormal.w;

    fixed3 normal;
    normal.xy = packednormal.xy * 2 - 1;
    normal.z = sqrt(1 - saturate(dot(normal.xy, normal.xy)));
    return normal;
}
```

最后一种情况，这里的发现纹理的格式是BC5，即(x,y,0,1)，计算方式和上面的函数类似，不多赘述。

```
fixed3 UnpackNormalWithScale(fixed4 packednormal, float scale)
{
#if defined(UNITY_ASTC_NORMALMAP_ENCODING)
    // (y, y, y, x), preferred for ASTC
    packednormal.x = packednormal.w;
#elif !defined(UNITY_NO_DXT5nm)
    // Unpack normal as DXT5nm (1, y, 1, x) or BC5 (x, y, 0, 1)
    // Note neutral texture like "bump" is (0, 0, 1, 1) to work with both
plain RGB normal and DXT5nm/BC5
    packednormal.x *= packednormal.w;
#endif // UNITY_NO_DXT5nm
    fixed3 normal;
    normal.xy = (packednormal.xy * 2 - 1) * scale;
    normal.z = sqrt(1 - saturate(dot(normal.xy, normal.xy)));
    return normal;
}
```

这里的函数，运作机理和上面的两个函数是一样的，分别对DXT5nm和BC5，采样对应的值，只不过参数里额外传入了一个scale，在最后返回法线的时候，通过scale对法线进行了缩放。

线性化深度值的工具函数

```
// Values used to linearize the Z buffer
(http://www.humus.name/temp/Linearize%20depth.txt)
// x = 1-far/near
// y = far/near
// z = x/far
// w = y/far
// or in case of a reversed depth buffer (UNITY_REVERSED_Z is 1)
// x = -1+far/near
// y = 1
// z = x/far
// w = 1/far
float4 _ZBufferParams;

// Z buffer to linear 0..1 depth
//把从深度纹理中取得的顶点深度值z变换到观察空间中，然后映射到[0,1]区间内
//_ZBufferParams的x分量是 (1 - Far)/Near，其y分量是（Far/Near）
inline float Linear01Depth( float z )
{
    return 1.0 / (_ZBufferParams.x * z + _ZBufferParams.y);
}
//把从深度纹理中取得的顶点深度值z变换到视图空间中
//_ZBufferParams的z分量是 (x/Far)，其w分量是 (1/Near)
// Z buffer to linear depth
inline float LinearEyeDepth( float z )
{
    return 1.0 / (_ZBufferParams.z * z + _ZBufferParams.w);
}
```

片元的深度往往是非线性的，从近切面到远切面之间的深度值精度分布不均匀，但有时候我们需要线性化的深度值，可以参考 [2023.11.30]深度图（Depth Texture）的介绍 ，里面也推导过为什么这个公式可以把深度转为线性。

此外，还封装了一些工具宏供纹理深度操作用：

```
// x = 1 or -1 (-1 if projection is flipped)
// y = near plane
// z = far plane
// w = 1/far plane
float4 _ProjectionParams;

// Depth render texture helpers
#define DECODE_EYEDEPTH(i) LinearEyeDepth(i)
//取得顶点从世界空间变换到视图空间后的z值，并取其相反数
#define COMPUTE_EYEDEPTH(o) o = -UnityObjectToViewPos( v.vertex ).z
```

```
//取得顶点从世界空间变换到视图空间后的z值，取得相反数后映射到[0,1]范围内
//_ProjectionParams的w分量就是1/Far 所以这里就是 -(z/far)，视图空间是右手系，所以本身变换后的z就是负数，这里取反就变回正数了
#define COMPUTE_DEPTH_01 -(UnityObjectToViewPos( v.vertex ).z * _ProjectionParams.w)
//把顶点法线从世界空间变换到视图空间，再标准化，这里乘上了UNITY_MATRIX_MV的逆转置矩阵
#define COMPUTE_VIEW_NORMAL normalize(mul((float3x3)UNITY_MATRIX_IT_MV, v.normal))
```

计算屏幕坐标的工具函数

```
inline float4 ComputeNonStereoScreenPos(float4 pos) {
    float4 o = pos * 0.5f;
#ifdef UNITY_PRETRANSFORM_TO_DISPLAY_ORIENTATION
    switch (UNITY_DISPLAY_ORIENTATION_PRETRANSFORM)
    {
    default: break;
    case UNITY_DISPLAY_ORIENTATION_PRETRANSFORM_90: o.xy = float2(-o.y, o.x);
break;
    case UNITY_DISPLAY_ORIENTATION_PRETRANSFORM_180: o.xy = -o.xy; break;
    case UNITY_DISPLAY_ORIENTATION_PRETRANSFORM_270: o.xy = float2(o.y, -o.x);
break;
    }
#endif
    o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w;
    o.zw = pos.zw;
    return o;
}

inline float4 ComputeScreenPos(float4 pos) {
    float4 o = ComputeNonStereoScreenPos(pos);
#if defined(UNITY_SINGLE_PASS_STEREO)
    o.xy = TransformStereoScreenSpaceTex(o.xy, pos.w);
#endif
    return o;
}
```

屏幕坐标的一部分内容，我们在之前的变换里面提到过；

UNITY_PRETRANSFORM_TO_DISPLAY_ORIENTATION看起来是给Valkan使用的，我们先不管，其次_ProjectionParams.x可以理解为DirectX和OpenGL的坐标差异，我们以OpenGL为例，令其=1，所以代码就被简化成了：

```
    float4 o = clipPos.xyzw;
    o.xy = float2(o.x, o.y) * 0.5f + o.w * 0.5f;
    o.zw = clipPos.zw;
    return o;
```

已知齐次空间坐标转屏幕坐标，需要做一次齐次除法，再映射到[0,1]的区间内，则公式为：

```
//(clipPos.x/clipPos.w) 就是做了齐次除法的齐次空间坐标
screenPos.x = ((clipPos.x/clipPos.w) * 0.5 + 0.5) * Width
screenPos.y = ((clipPos.y/clipPos.w) * 0.5 + 0.5) * Height

//等式前后都乘以clipPos.w得:
screenPos.x * clipPos.w = ((clipPos.x/clipPos.w)*0.5 + 0.5) * Width * clipPos.w
screenPos.y * clipPos.w = ((clipPos.y/clipPos.w)*0.5 + 0.5) * Height *
clipPos.w

//简化后得:
screenPos.x * clipPos.w = (clipPos.x * 0.5 + 0.5 * clipPos.w) * Width
screenPos.y * clipPos.w = (clipPos.y * 0.5 + 0.5 * clipPos.w) * Height
```

和上面的代码对比一下，可以发现只比我们的目标函数多出了渲染目标的宽和高，将这两项除去，就和我们的函数一样了。

```
(screenPos.x / Width) * clipPos.w = (clipPos.x * 0.5 + 0.5 * clipPos.w)
(screenPos.y / Height) * clipPos.w = (clipPos.y * 0.5 + 0.5 * clipPos.w)
```

不难发现，ComputeScreenPos系列函数，返回的值值齐次坐标系下的屏幕坐标值，其范围为[0, w]。Unity的本意是想把这个坐标值用于tex2Dproj指令的参数值，tex2Dproj会在对纹理采样前除以w分量，而这步操作会将正交投影转换为透视投影。

```
inline float4 ComputeGrabScreenPos (float4 pos) {
    #if UNITY_UV_STARTS_AT_TOP
    float scale = -1.0;
    #else
    float scale = 1.0;
    #endif
    float4 o = pos * 0.5f;
#ifdef UNITY_PRETRANSFORM_TO_DISPLAY_ORIENTATION
    switch (UNITY_DISPLAY_ORIENTATION_PRETRANSFORM)
    {
    default: break;
```

```
    case UNITY_DISPLAY_ORIENTATION_PRETRANSFORM_90: o.xy = float2(-o.y, o.x);
break;
    case UNITY_DISPLAY_ORIENTATION_PRETRANSFORM_180: o.xy = -o.xy; break;
    case UNITY_DISPLAY_ORIENTATION_PRETRANSFORM_270: o.xy = float2(o.y, -o.x);
break;
    }
#endif
    o.xy = float2(o.x, o.y*scale) + o.w;
#ifdef UNITY_SINGLE_PASS_STEREO
    o.xy = TransformStereoScreenSpaceTex(o.xy, pos.w);
#endif
    o.zw = pos.zw;
    return o;
}
```

函数用于把当前屏幕内容截图并保存在一个目标纹理中，它传入了裁剪空间中某点的齐次坐标值，返回目标纹理中的纹理贴图坐标。

```
// x = width
// y = height
// z = 1 + 1.0/width
// w = 1 + 1.0/height
float4 _ScreenParams;

// snaps post-transformed position to screen pixels
inline float4 UnityPixelSnap (float4 pos)
{
    float2 hpc = _ScreenParams.xy * 0.5f;
#if  SHADER_API_PSSL
// An old sdk used to implement round() as floor(x+0.5) current sdks use the
round to even method so we manually use the old method here for compatabilty.
    float2 temp = ((pos.xy / pos.w) * hpc) + float2(0.5f,0.5f);
    float2 pixelPos = float2(floor(temp.x), floor(temp.y));
#else
    float2 pixelPos = round ((pos.xy / pos.w) * hpc);
#endif
    pos.xy = pixelPos / hpc * pos.w;
    return pos;
}
```

该函数把视图坐标转换为屏幕空间的齐次坐标，可以看到它先做了其次除法，用屏幕的宽高比的做了映射之后，再变回齐次坐标的xy分量。

```
inline float2 TransformViewToProjection (float2 v) {
    return mul((float2x2)UNITY_MATRIX_P, v);
}

inline float3 TransformViewToProjection (float3 v) {
    return mul((float3x3)UNITY_MATRIX_P, v);
}
```

以及视图坐标转投影坐标的重载函数。

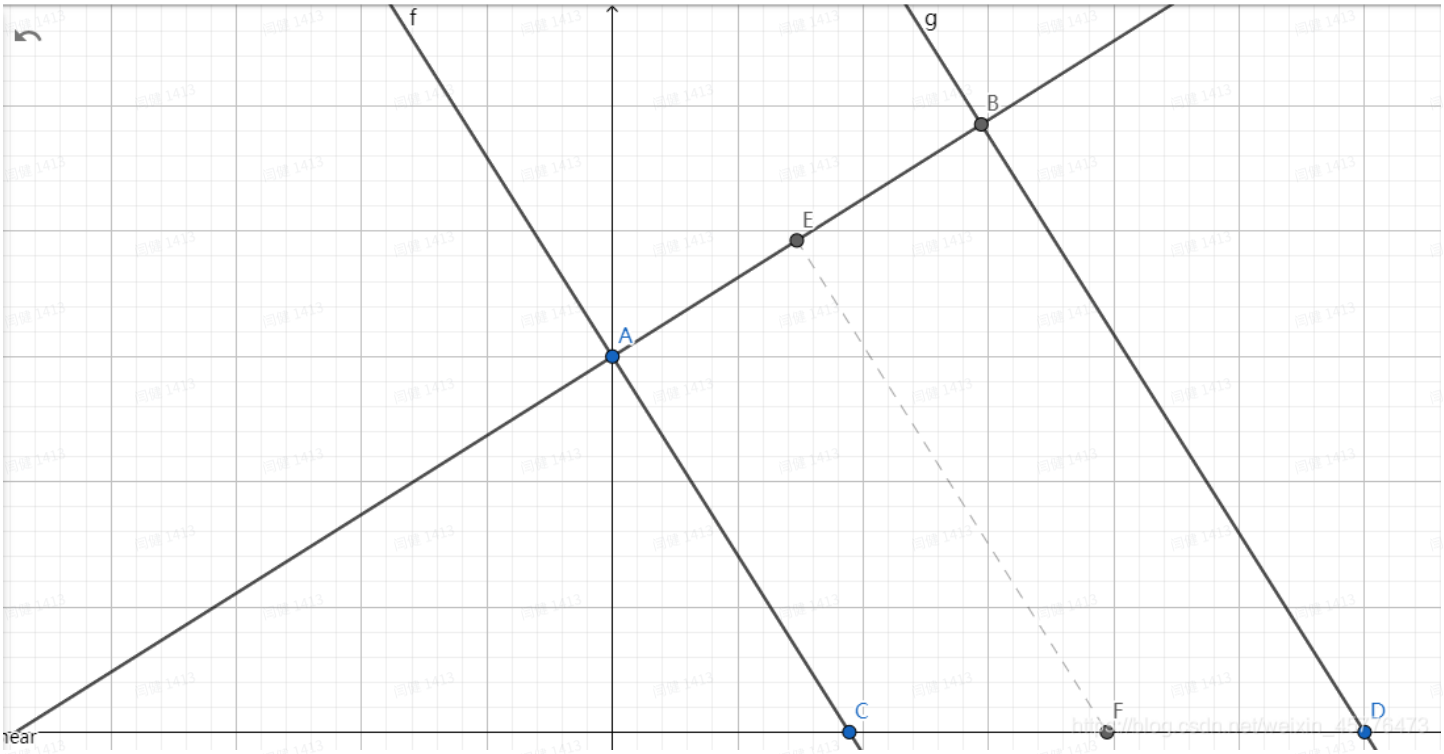
与阴影处理相关的工具函数

```
// Shadow caster pass helpers
//把一个float类型的阴影深度编码进一个float4类型的RGBA数值中。
float4 UnityEncodeCubeShadowDepth (float z)
{
    #ifdef UNITY_USE_RGBA_FOR_POINT_SHADOWS
    return EncodeFloatRGBA (min(z, 0.999));
    #else
    return z;
    #endif
}
//解码版本
float UnityDecodeCubeShadowDepth (float4 vals)
{
    #ifdef UNITY_USE_RGBA_FOR_POINT_SHADOWS
    return DecodeFloatRGBA (vals);
    #else
    return vals.r;
    #endif
}
```

Shadow bias

要理解下面这个函数，我们需要去了解一下阴影失真的概念是什么：

Shadow Acne（阴影失真）是由于ShadowMap的精度问题，阴影图上的一个pixel实际上是对应了场景上的一片区域，而这片区域中的所有点，在shadow caster阶段，都会去取shadow map同一个pixel的深度值，当这个深度值与区域中所有的点的深度值相近时，在进行深度比较后，就会出现一部分点在阴影中，一部分点不在阴影中，导致场景会出现明显的明暗条纹。

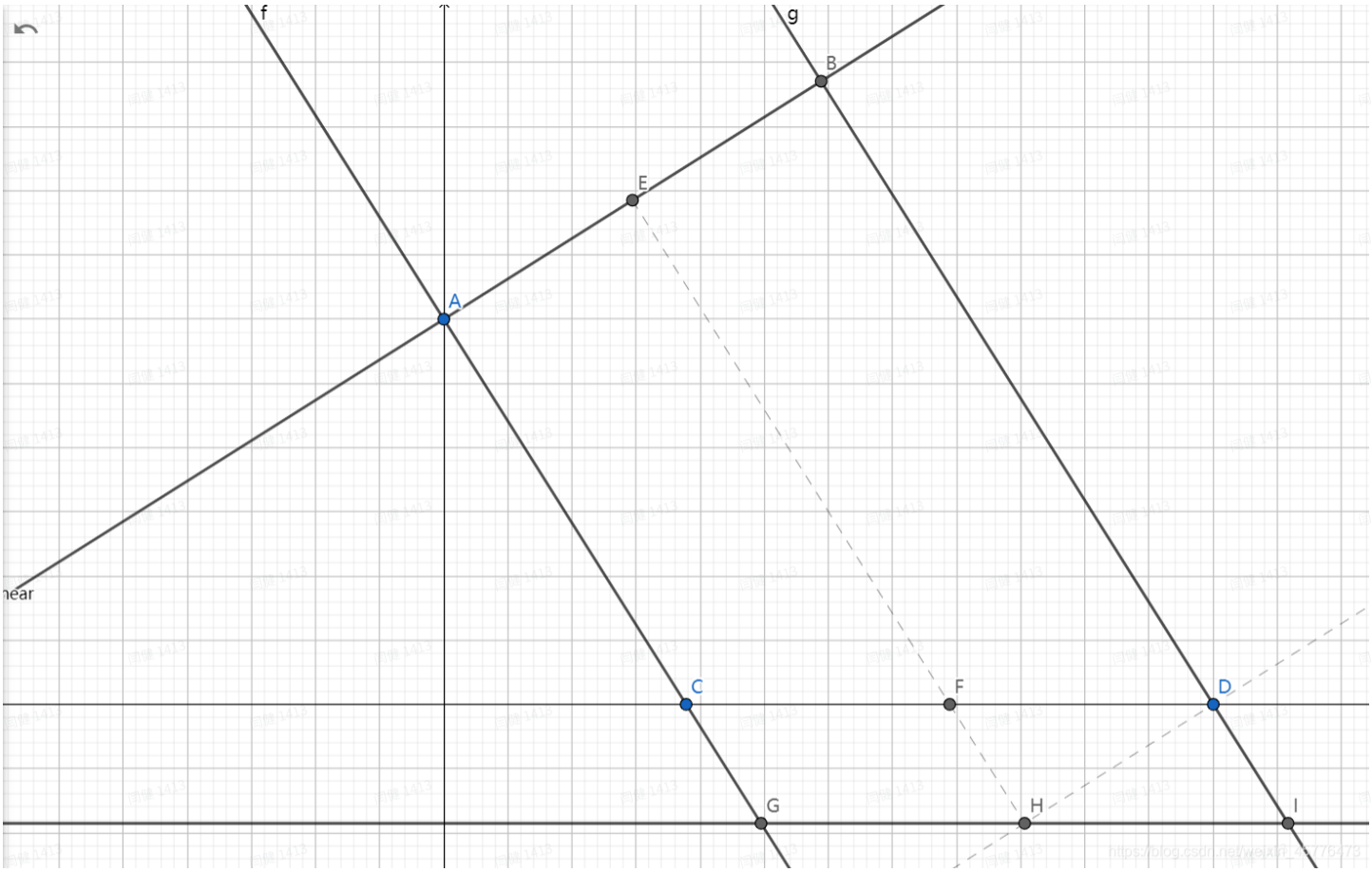


从数学模型上来看：首先**AB**所在的平面是平行光世界的近平面，记为L，这个近平面最终将映射为一张ShadowMap深度图。**CD**所在的平面就是接受光照的平面。

EF就是shadow map采样的深度，那么**CF**区域中的点，由于深度都小于EF，因此不在阴影中；而**DF**区域中的点深度大于EF，所以会被判定在阴影中。

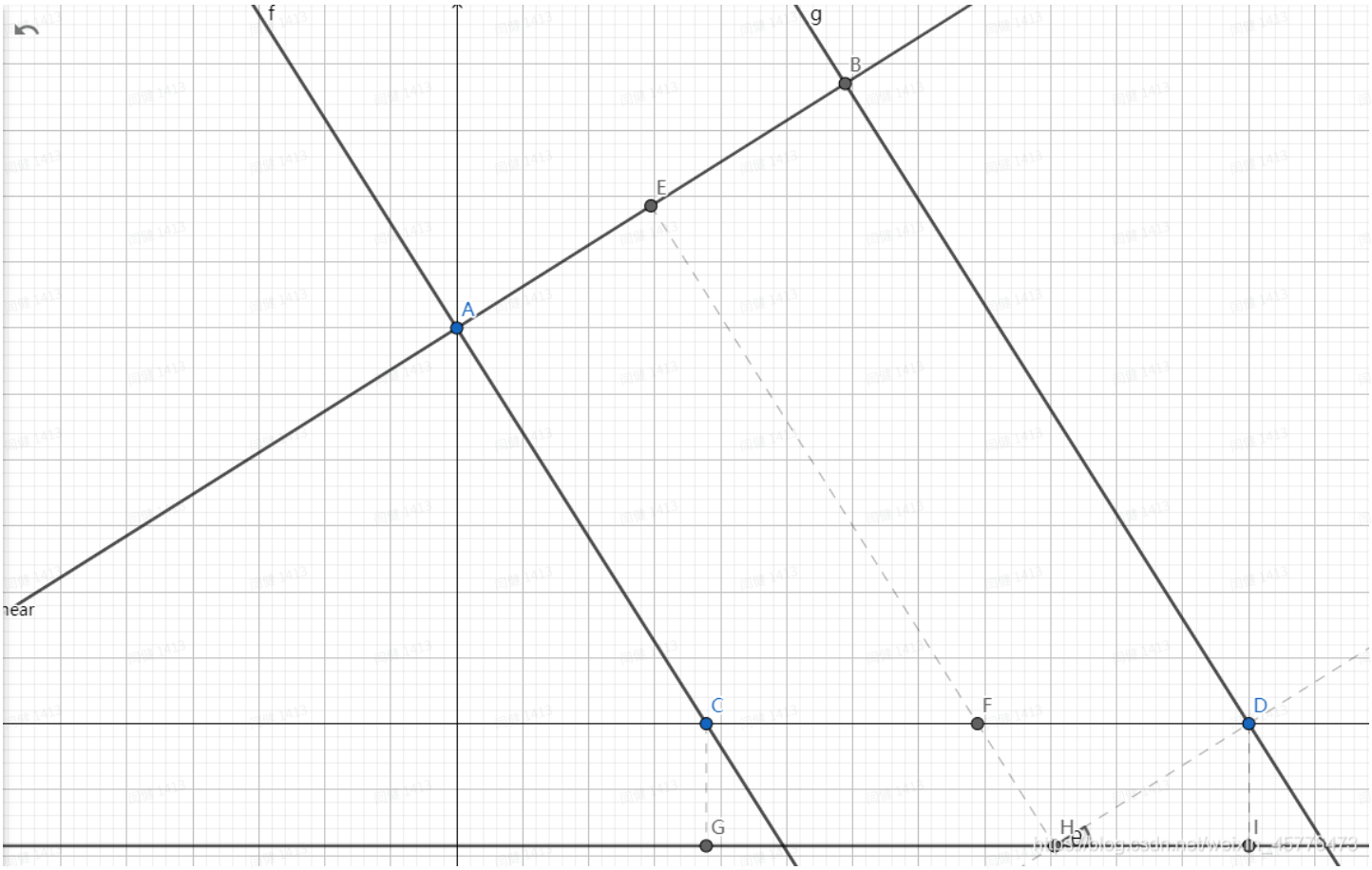
为了避免shadow acne，可以将渲染shadow map的时候人为加上偏移，让阴影位于物体之后，这个偏移就被称为shadow bias；常见的shadow bias有两种，一种是**depth bias**，另一种是**normal bias**；

我们先来看depth bias，顾名思义就是在shadow caster阶段，将物体沿着摄像机空间的z方向往后偏移：



可以看到，将**CD**区域沿着摄像机空间的z方向平移到了**GI**区域。此时，shadow map采样到的深度值为**EH**，它等于**BD**。这样，CD区域中的所有点的深度都不会大于**EH**(因为最大深度也就是**BD**，而**BD**等于**EH**)，从而保证了CD区域中所有的点都不会判定在阴影中，也就可以消除shadow acne；这里**depth bias的值就是CG的大小**。

而normal bias就是在shadow caster阶段，将物体沿着自身法线的方向往后偏移，也就是下面这张图，它的原理和depth bias是类似的，不再赘述；这里normal bias的值就是CG的大小。




```
//vertex是物体顶点在模型空间中的坐标值
//normal是物体法线在模型空间中的坐标值
float4 UnityClipSpaceShadowCasterPos(float4 vertex, float3 normal)
{
    //顶点从模型空间转换到世界空间
    float4 wPos = mul(unity_ObjectToWorld, vertex);

    if (unity_LightShadowBias.z != 0.0)
    {
        //法线从模型空间转换到世界空间
        float3 wNormal = UnityObjectToWorldNormal(normal);
        //UnityWorldSpaceLightDir计算世界坐标系下光源位置点_WorldSpaceLightPos0
        //与世界坐标系下的点wPos的连线的方向向量
        float3 wLight = normalize(UnityWorldSpaceLightDir(wPos.xyz));

        // apply normal offset bias (inset position along the normal)
        // bias needs to be scaled by sine between normal and light direction
        // (http://the-witness.net/news/2013/09/shadow-mapping-summary-part-1/)
        //
        // unity_LightShadowBias.z contains user-specified normal offset amount
        // scaled by world space texel size.
        //计算光线与法线夹角的余弦值
        float shadowCos = dot(wNormal, wLight);
        float shadowSine = sqrt(1-shadowCos*shadowCos);
        float normalBias = unity_LightShadowBias.z * shadowSine;
```

```
            wPos.xyz -= wNormal * normalBias;
        }

        return mul(UNITY_MATRIX_VP, wPos);//进行了偏移之后的值变换到裁剪空间
    }
```

回到我们的函数本身，`unity_LightShadowBias.z`存储了与normal bias相关的参数，这个值还与shadow map的贴图尺寸有关，且该值只在平行光源情况下有效，在其它光源下都是0。

我们要求的normal bias就是**CG**的长度，也就是**DI**：

$$CG = DI = DH \cdot \sin\theta = \frac{1}{2} AB \cdot \sin\theta$$

AB就是光源的视椎体在shadowmap的尺寸下的映射值，可以理解成是shadowmap的一个texel所能覆盖视椎体区域，这里也能看出，shadowmap的精度越高，覆盖的光源视椎体区域越小，引起shadow acne(阴影失真)的可能性也越小。

$$AB = \frac{\mathrm{frustumSize}}{\mathrm{shadowMapSize}}$$

再从图中可以看出，θ其实就是光源与平面的夹角，cosθ就是光线与法线夹角的余弦值；继而通过余弦值，可以求出正弦值是sqrt(1 - cosθ^2)；最后求得`normal bias = CG = DI = frustumSize/(2*shadowMapSize) * shadowSine(sinθ)`；

则`unity_LightShadowBias.z`中存储的应该就是`frustumSize/(2*shadowMapSize)`；

```
    float4 UnityApplyLinearShadowBias(float4 clipPos)
    {
        // For point lights that support depth cube map, the bias is applied in
    the fragment shader sampling the shadow map.
        // This is because the legacy behaviour for point light shadow map cannot
    be implemented by offseting the vertex position
        // in the vertex shader generating the shadow map.
    #if !(defined(SHADOWS_CUBE) && defined(SHADOWS_CUBE_IN_DEPTH_TEX))
        #if defined(UNITY_REVERSED_Z)
        //如果UNITY_REVERSED_Z，则z值的范围是[near,0]，所以此时偏移需要朝向近平面做偏移，
    此时的空间中，近偏移就是减法；
        //没有UNITY_REVERSED_Z的空间中，此时近偏移是加法；
            // We use max/min instead of clamp to ensure proper handling of the
    rare case
            // where both numerator and denominator are zero and the fraction
    becomes NaN.
            clipPos.z += max(-1, min(unity_LightShadowBias.x / clipPos.w, 0));
        #else
```

```
            clipPos.z += saturate(unity_LightShadowBias.x/clipPos.w);
        #endif
    #endif
        //UNITY_NEAR_CLIP_VALUE在dx中为1，opengl中是-1，是为了不让越界
        //接下来的操作就是为了不让计算出的z值越界
    #if defined(UNITY_REVERSED_Z)
        //z值范围为[near,0]，是递减的，如果clipPos.z大于clipPos.w*UNITY_NEAR_CLIP_VALUE
    的值，说明越界了
        float clamped = min(clipPos.z, clipPos.w*UNITY_NEAR_CLIP_VALUE);
    #else
        //z值范围为[near,1]，是递增的，如果clipPos.z小于clipPos.w*UNITY_NEAR_CLIP_VALUE
    的值，说明越界了
        float clamped = max(clipPos.z, clipPos.w*UNITY_NEAR_CLIP_VALUE);
    #endif
        //点光源不做处理，unity_LightShadowBias.y=0的时候为非平行光，为1的时候表示平行光
        clipPos.z = lerp(clipPos.z, clamped, unity_LightShadowBias.y);
        return clipPos;
    }
```

再来看UnityApplyLinearShadowBias，上面的函数是法线偏移，但我们真正做计算比较深度的地方是裁剪空间，所以这里才是真正意义上的阴影偏移，我们进行偏移的时候要注意避免裁剪空间中的z值发生越界。

代码一开始的宏是用来判断是否是点光源且当前的硬件平台支持depth cube map，如果是就跳过bias计算，因为这里无法实现。bias的计算和是否UNITY_REVERSED_Z相关；

`unity_LightShadowBias.x` 表示的是阴影在裁剪空间中的线性偏移bias。

UnityApplyLinearShadowBias是将调用了UnityClipSpaceShadowCasterPos得到的在裁剪空间坐标的z值再进行进一步的偏移，因为这个操作是在裁剪空间这样的齐次坐标系下进行的，所以要对透视投影产生的z值进行补偿，因为透视投影是近大远小嘛，我们要保证物体在光源空间的不同位置往后偏移时，都能偏移相同的一个量。

```
    #if defined(SHADOWS_CUBE) && !defined(SHADOWS_CUBE_IN_DEPTH_TEX)
        // Rendering into point light (cubemap) shadows
        //用来存储世界坐标系下，当前顶点到光源位置的连线向量
        #define V2F_SHADOW_CASTER_NOPOS float3 vec : TEXCOORD0;
        #define TRANSFER_SHADOW_CASTER_NOPOS_LEGACY(o,opos) o.vec =
    mul(unity_ObjectToWorld, v.vertex).xyz - _LightPositionRange.xyz; opos =
    UnityObjectToClipPos(v.vertex);
        //x、y、z分量为光源的位置，w分量为光源照射范围的倒数；
        //TRANSFER_SHADOW_CASTER_NOPOS的功能是计算在世界坐标系下，当前顶点到光源位置的连线
    向量，同时把顶点位置变换到裁剪空间
```

```
    #define TRANSFER_SHADOW_CASTER_NOPOS(o,opos) o.vec =
mul(unity_ObjectToWorld, v.vertex).xyz - _LightPositionRange.xyz; opos =
UnityObjectToClipPos(v.vertex);
    //把一个float类型的阴影深度值边骂道一个float4类型中并返回
    #define SHADOW_CASTER_FRAGMENT(i) return UnityEncodeCubeShadowDepth
((length(i.vec) + unity_LightShadowBias.x) * _LightPositionRange.w);

#else
    // Rendering into directional or spot light shadows
    //渲染由平行光或者聚光灯光源产生的阴影
    #define V2F_SHADOW_CASTER_NOPOS
    // Let embedding code know that V2F_SHADOW_CASTER_NOPOS is empty; so that
it can workaround
    // empty structs that could possibly be produced.
    #define V2F_SHADOW_CASTER_NOPOS_IS_EMPTY
    #define TRANSFER_SHADOW_CASTER_NOPOS_LEGACY(o,opos) \
        opos = UnityObjectToClipPos(v.vertex.xyz); \
        opos = UnityApplyLinearShadowBias(opos);
    #define TRANSFER_SHADOW_CASTER_NOPOS(o,opos) \
        opos = UnityClipSpaceShadowCasterPos(v.vertex, v.normal); \
        opos = UnityApplyLinearShadowBias(opos);
    #define SHADOW_CASTER_FRAGMENT(i) return 0;
#endif
```

与雾效相关的工具函数和宏

```
#if defined(UNITY_REVERSED_Z)
    #if UNITY_REVERSED_Z == 1
        //D3d with reversed Z => z clip range is [near, 0] -> remapping to [0,
far]
        //max is required to protect ourselves from near plane not being
correct/meaningfull in case of oblique matrices.
        //不经过z轴反向操作的OpenGL平台上的裁剪空间坐标
        #define UNITY_Z_0_FAR_FROM_CLIPSPACE(coord) max(((1.0-
(coord)/_ProjectionParams.y)*_ProjectionParams.z),0)
    #else
        //GL with reversed z => z clip range is [near, -far] -> should remap
in theory but dont do it in practice to save some perf (range is close enough)
        //经过z轴反向操作的OpenGL平台上的裁剪空间坐标
        #define UNITY_Z_0_FAR_FROM_CLIPSPACE(coord) max(-(coord), 0)
    #endif
#elif UNITY_UV_STARTS_AT_TOP
    //D3d without reversed z => z clip range is [0, far] -> nothing to do
    #define UNITY_Z_0_FAR_FROM_CLIPSPACE(coord) (coord)
```

```
#else
    //Opengl => z clip range is [-near, far] -> should remap in theory but
dont do it in practice to save some perf (range is close enough)
    #define UNITY_Z_0_FAR_FROM_CLIPSPACE(coord) (coord)
#endif
```

在计算雾化因子时，需要取得当前片元的摄像机的距离的绝对值，并且离摄像机越远这个值要越大。而这个距离要通过片元在裁剪空间中的z值计算得到。不同平台下裁剪空间的z值取值范围有所不同，所以UNITY_Z_0_FAR_FROM_CLIPSPACE是把各个平台的差异化处理掉。

前面在很多地方，我们都见到了UNITY_REVERSED_Z这个宏，接着来看一下它的定义：

```
//可以看到在D3D、PSSL、METAL、VULKAN、SWITCH这些平台下，会逆转裁剪空间的near-far取值，
near为1，far为0
#if defined(SHADER_API_D3D11) || defined(SHADER_API_PSSL) ||
defined(SHADER_API_METAL) || defined(SHADER_API_VULKAN) ||
defined(SHADER_API_SWITCH)
// D3D style platforms where clip space z is [0, 1].
#define UNITY_REVERSED_Z 1
#endif

//根据是否UNITY_REVERSED_Z来定义UNITY_NEAR_CLIP_VALUE 的值
#if defined(UNITY_REVERSED_Z)
#define UNITY_NEAR_CLIP_VALUE (1.0)
#else
#define UNITY_NEAR_CLIP_VALUE (-1.0)
#endif
```

接着是雾化因子衰减的计算：

```
//雾化因子线性衰减
#if defined(FOG_LINEAR)
    // factor = (end-z)/(end-start) = z * (-1/(end-start)) + (end/(end-start))
    #define UNITY_CALC_FOG_FACTOR_RAW(coord) float unityFogFactor = (coord) *
unity_FogParams.z + unity_FogParams.w
//雾化因子指数衰减
#elif defined(FOG_EXP)
    // factor = exp(-density*z)
    #define UNITY_CALC_FOG_FACTOR_RAW(coord) float unityFogFactor =
unity_FogParams.y * (coord); unityFogFactor = exp2(-unityFogFactor)
//雾化因子指数平方级衰减
#elif defined(FOG_EXP2)
```

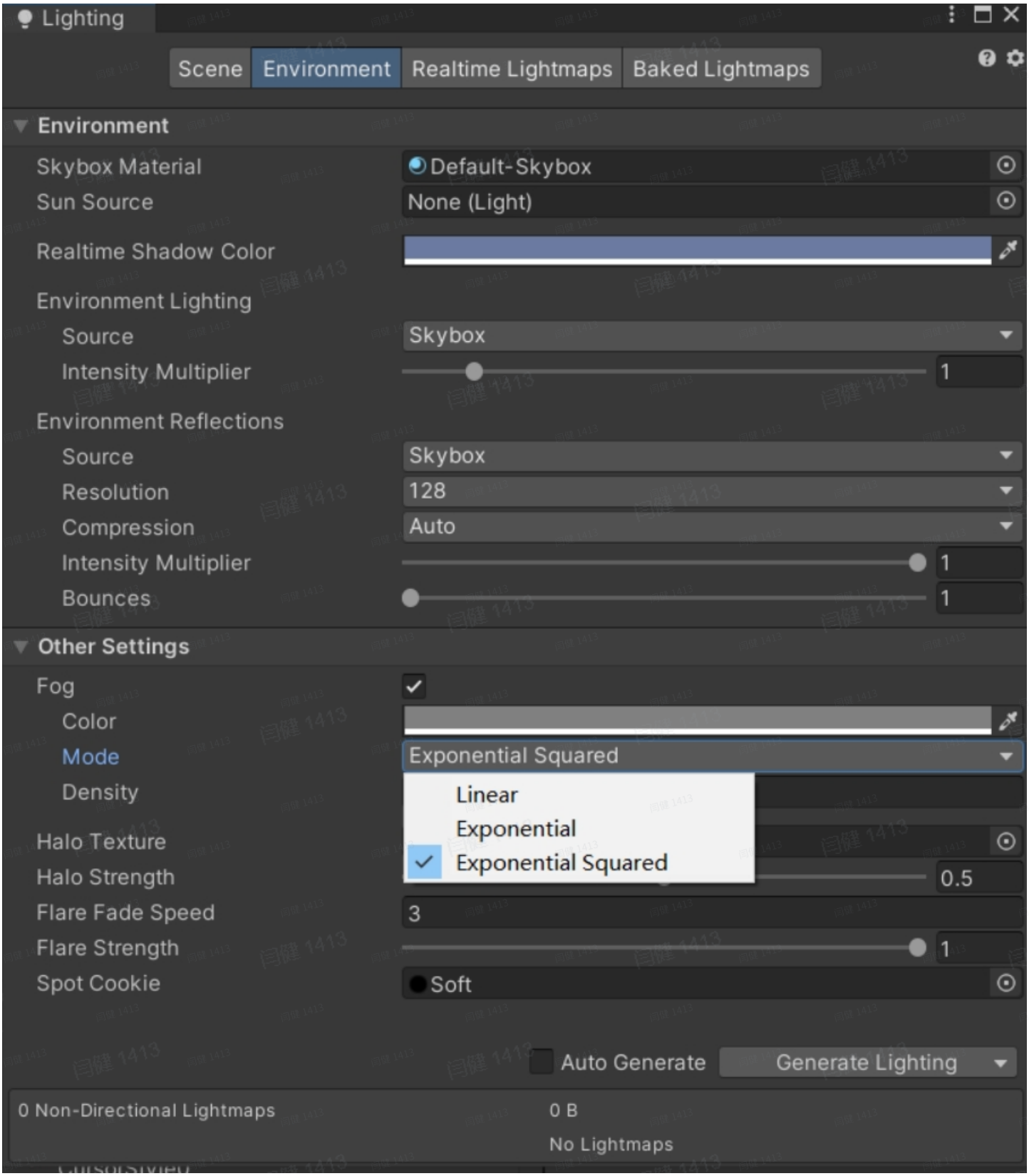
```
        // factor = exp(-(density*z)^2)
        #define UNITY_CALC_FOG_FACTOR_RAW(coord) float unityFogFactor =
    unity_FogParams.x * (coord); unityFogFactor = exp2(-
    unityFogFactor*unityFogFactor)
    #else
        #define UNITY_CALC_FOG_FACTOR_RAW(coord) float unityFogFactor = 0.0
    #endif

    #define UNITY_CALC_FOG_FACTOR(coord)
    UNITY_CALC_FOG_FACTOR_RAW(UNITY_Z_0_FAR_FROM_CLIPSPACE(coord))

    #define UNITY_FOG_COORDS_PACKED(idx, vectype) vectype fogCoord : TEXCOORD##idx;
```

如果不启用雾化效果，则雾化因子为0；Unity提供了控制雾化因子衰减选项的功能：

1. Color是雾的颜色；
2. Density是雾的浓度；
3. Mode中可以选择：
    a. `Linear → FOG_LINEAR`
    b. `Exponential → FOG_EXP`
    c. `Exponential Squared → FOG_EXP2`



接下来是不同平台和不同雾化因子计算方式下的UNITY_TRANSFER_FOG

```
        #if defined(FOG_LINEAR) || defined(FOG_EXP) || defined(FOG_EXP2)
            #define UNITY_FOG_COORDS(idx) UNITY_FOG_COORDS_PACKED(idx, float1)

            #if (SHADER_TARGET < 30) || defined(SHADER_API_MOBILE)
                // mobile or SM2.0: calculate fog factor per-vertex
```

```
        //如果移动平台或者是使用了Shader model2.0的平台，则在顶点中计算雾化效果
        #define UNITY_TRANSFER_FOG(o,outpos)
UNITY_CALC_FOG_FACTOR((outpos).z); o.fogCoord.x = unityFogFactor
        #define UNITY_TRANSFER_FOG_COMBINED_WITH_TSPACE(o,outpos)
UNITY_CALC_FOG_FACTOR((outpos).z); o.tSpace1.y = tangentSign; o.tSpace2.y =
unityFogFactor
        #define UNITY_TRANSFER_FOG_COMBINED_WITH_WORLD_POS(o,outpos)
UNITY_CALC_FOG_FACTOR((outpos).z); o.worldPos.w = unityFogFactor
        #define UNITY_TRANSFER_FOG_COMBINED_WITH_EYE_VEC(o,outpos)
UNITY_CALC_FOG_FACTOR((outpos).z); o.eyeVec.w = unityFogFactor
    #else
        // SM3.0 and PC/console: calculate fog distance per-vertex, and fog
factor per-pixel
        //如果是使用shader model3.0的平台，或者使用PC或一些游戏主机平台
        //就在顶点着色器中计算每个顶点离当前摄像机的距离。在片元着色器中计算雾化因子。
        #define UNITY_TRANSFER_FOG(o,outpos) o.fogCoord.x = (outpos).z
        #define UNITY_TRANSFER_FOG_COMBINED_WITH_TSPACE(o,outpos) o.tSpace2.y
= (outpos).z
        #define UNITY_TRANSFER_FOG_COMBINED_WITH_WORLD_POS(o,outpos)
o.worldPos.w = (outpos).z
        #define UNITY_TRANSFER_FOG_COMBINED_WITH_EYE_VEC(o,outpos) o.eyeVec.w
= (outpos).z
    #endif
#else
    #define UNITY_FOG_COORDS(idx)
    #define UNITY_TRANSFER_FOG(o,outpos)
    #define UNITY_TRANSFER_FOG_COMBINED_WITH_TSPACE(o,outpos)
    #define UNITY_TRANSFER_FOG_COMBINED_WITH_WORLD_POS(o,outpos)
    #define UNITY_TRANSFER_FOG_COMBINED_WITH_EYE_VEC(o,outpos)
#endif

//还有雾化颜色
//利用雾的颜色和当前像素的颜色，根据雾化因子进行线性插值运算，得到最终的雾化效果颜色
#define UNITY_FOG_LERP_COLOR(col,fogCol,fogFac) col.rgb = lerp((fogCol).rgb,
(col).rgb, saturate(fogFac))
```

```
#if defined(FOG_LINEAR) || defined(FOG_EXP) || defined(FOG_EXP2)
    #if (SHADER_TARGET < 30) || defined(SHADER_API_MOBILE)
        // mobile or SM2.0: fog factor was already calculated per-vertex, so
just lerp the color
        #define UNITY_APPLY_FOG_COLOR(coord,col,fogCol)
UNITY_FOG_LERP_COLOR(col,fogCol,(coord).x)
    #else
```

```
            // SM3.0 and PC/console: calculate fog factor and lerp fog color
            #define UNITY_APPLY_FOG_COLOR(coord,col,fogCol)
        UNITY_CALC_FOG_FACTOR((coord).x);
        UNITY_FOG_LERP_COLOR(col,fogCol,unityFogFactor)
        #endif
        #define UNITY_EXTRACT_FOG(name) float _unity_fogCoord = name.fogCoord
        #define UNITY_EXTRACT_FOG_FROM_TSPACE(name) float _unity_fogCoord =
        name.tSpace2.y
        #define UNITY_EXTRACT_FOG_FROM_WORLD_POS(name) float _unity_fogCoord =
        name.worldPos.w
        #define UNITY_EXTRACT_FOG_FROM_EYE_VEC(name) float _unity_fogCoord =
        name.eyeVec.w
        #else
        #define UNITY_APPLY_FOG_COLOR(coord,col,fogCol)
        #define UNITY_EXTRACT_FOG(name)
        #define UNITY_EXTRACT_FOG_FROM_TSPACE(name)
        #define UNITY_EXTRACT_FOG_FROM_WORLD_POS(name)
        #define UNITY_EXTRACT_FOG_FROM_EYE_VEC(name)
        #endif

        #ifdef UNITY_PASS_FORWARDADD
        #define UNITY_APPLY_FOG(coord,col)
        UNITY_APPLY_FOG_COLOR(coord,col,fixed4(0,0,0,0))
        #else
        #define UNITY_APPLY_FOG(coord,col)
        UNITY_APPLY_FOG_COLOR(coord,col,unity_FogColor)
        #endif
```

UNITY_APPLY_FOG_COLOR定义在不同平台上的最终雾化效果的颜色计算方法。

引用

(二)unity自带的着色器源码剖析之————————UnityCG.cginc文件(上篇:数学常数、颜色空间常数和函数、顶点布局格式结构体、进行空间变换的函数、HDR级光照贴图编解码相关函数等)_uni

(三)unity自带的着色器源码剖析之————————UnityCG.cginc文件(下篇:法线贴图及编解码操作函数、线性化深度值函数、计算屏幕坐标工具函数、阴影处理相关函数、雾相关函数等)__zbuff

理解EncodeFloatRGBA和DecodeFloatRGBA-CSDN博客

Unity中的shadows(一)_unity upr 点光源 阴影-CSDN博客

自适应Shadow Bias算法