article-日式卡通渲染笔记(罪恶装备 碧蓝幻想 原神 战双)

目录

日式卡通渲染的特性(Feature)

亮部

暗部

高光

边缘光

视角光

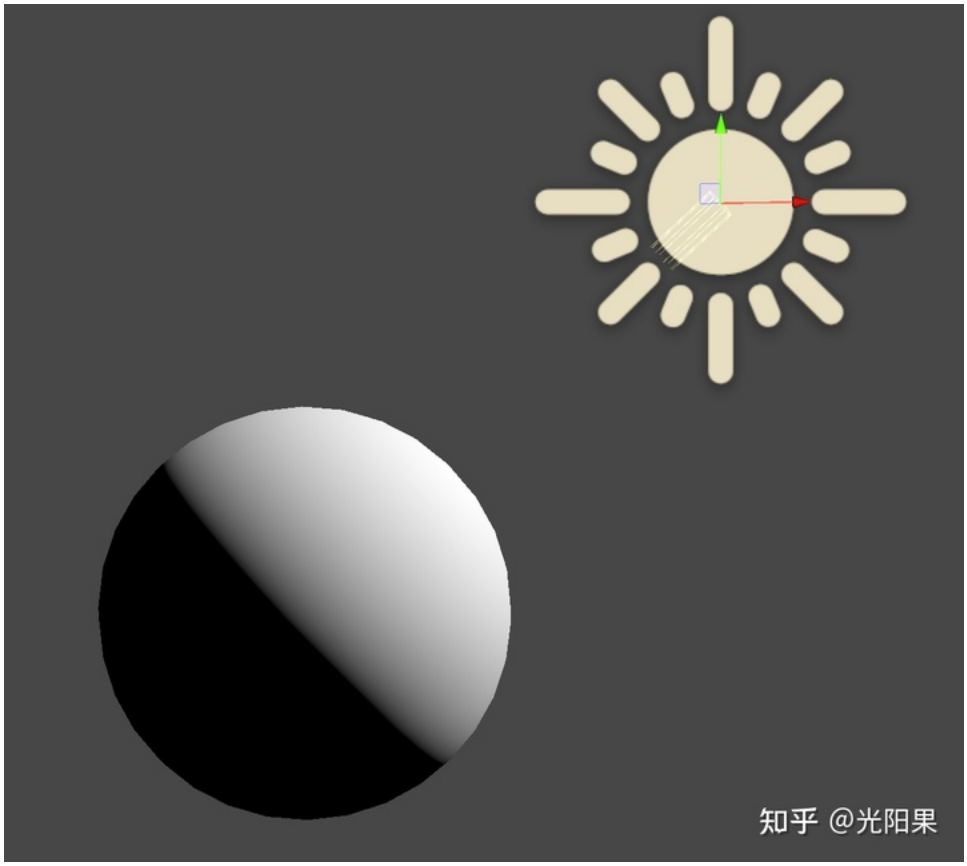
描边

高光形变

裁边漫反射(StepDiffuse)

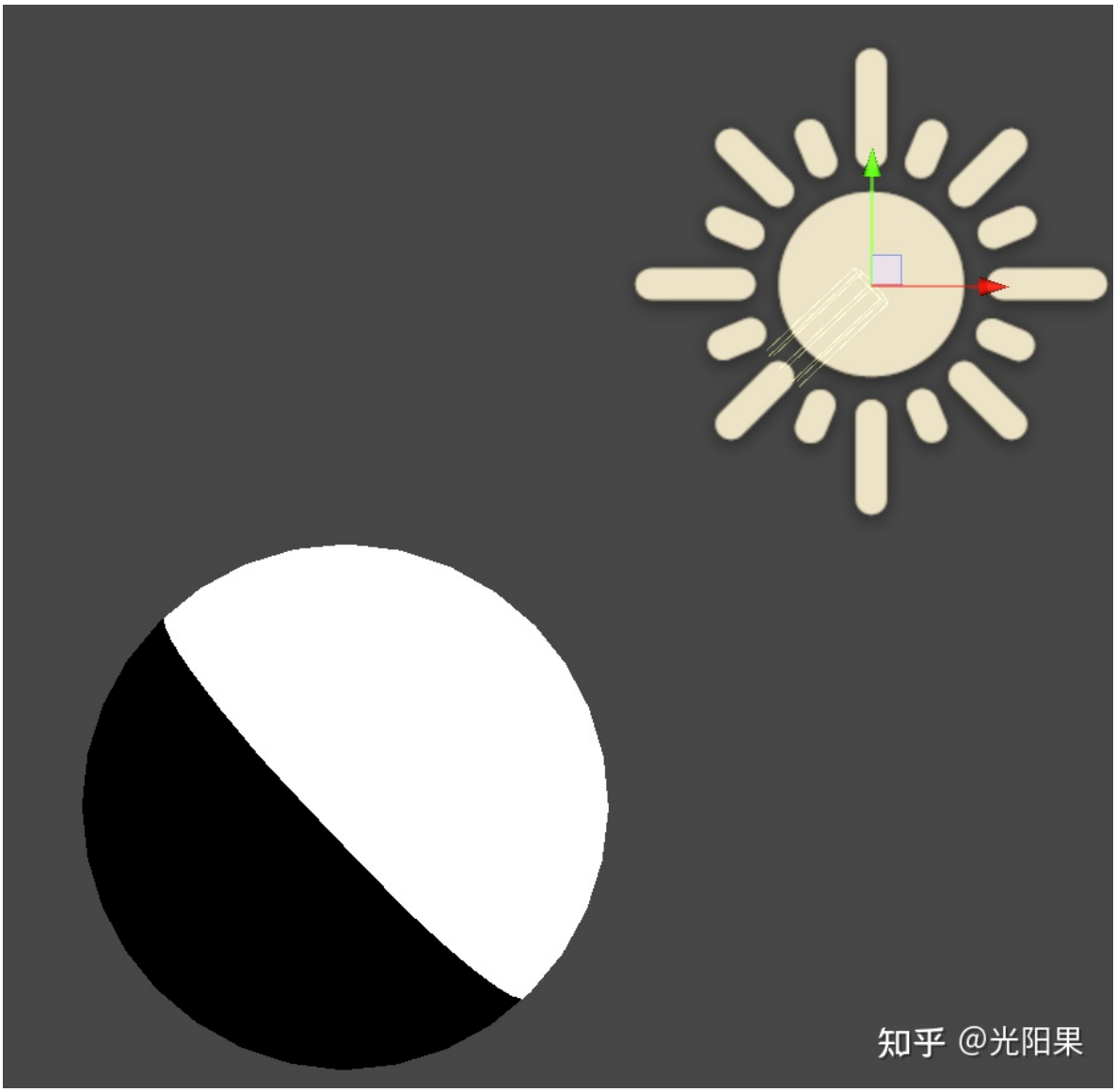
传统的漫反射是这样

```
float NL = dot(N,L);
float3 Diffuse = max(0,NL);//也可以除以PI
```

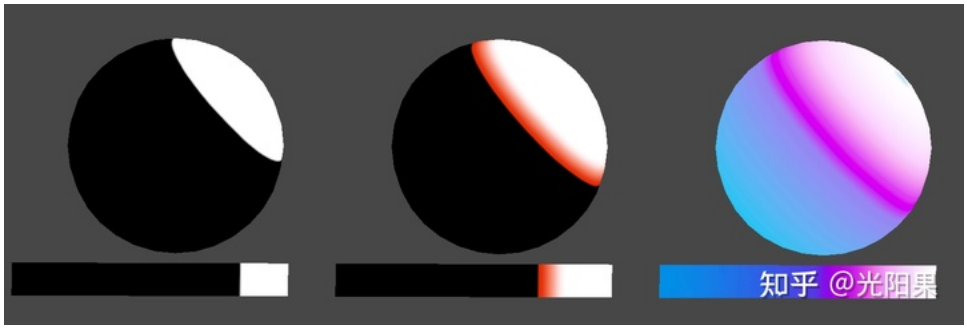


而卡通渲染里希望明快的色调对比，不希望有额外的过渡光照信息，可以对NL加一个Step做二分，表现出光照的 亮部 与 暗部。

```
float NL01 = NL*0.5+0.5;//将NL从(-1,1)映射到(0,1) 方便对贴图进行操作
float Threshold = step(_LightThreshold,NL01);
//也可以对明暗交界先进行光滑过渡处理: NL01 = smoothstep(0,_Smooth,NL01-_SmoothRange);
float3 Diffuse  = lerp(DarkSide,BrightSide,Threshold);
```


知乎 @光阳果

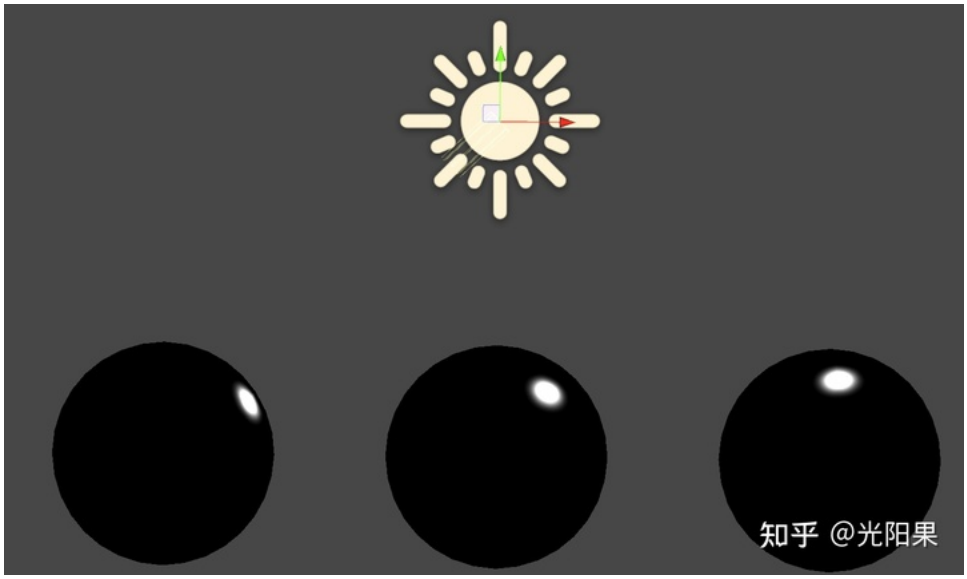
```
float NL01 = NL*0.5+0.5;//将NL从(-1,1)映射到(0,1) 方便对贴图进行采样
// float3 Threshold = step(_LightThreshold,NL01); //对明暗交界先进行光滑过渡处理: NL01 = smoothstep(0,_Smooth,NL01-_SmoothRange);
float3 Ramp = tex2D(_RampTex,NL01);
float3 Diffuse1  = lerp(DarkSide,BrightSide,Ramp);//将Ramp当做一个衰减映射Lut
float3 Diffuse2 = Ramp*BaseColor;//将Ramp当做明暗部色调与原色相乘
```



裁边高光(StepSpecular)

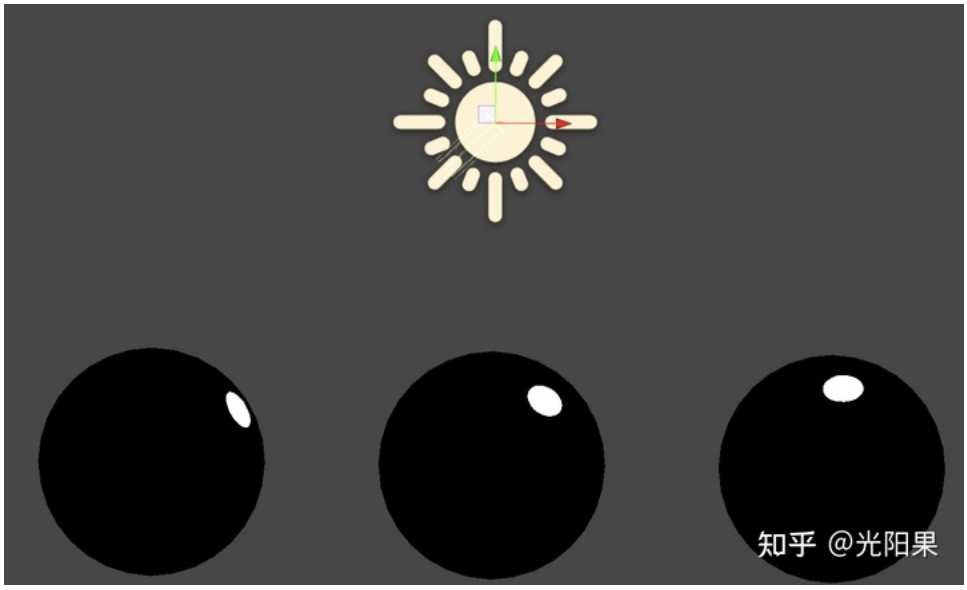
传统的BlinPhong高光带有明暗过度

```
float3 SpecularBlinPhong = pow(NH,_BPExp)*_BPScale;
```



可以对高光进行裁边处理，即 **裁边高光(StepSpecualr)，**可以裁BlinPhong，也可以裁GGX，但卡通渲染里追求简单，裁BlinPhong更快。

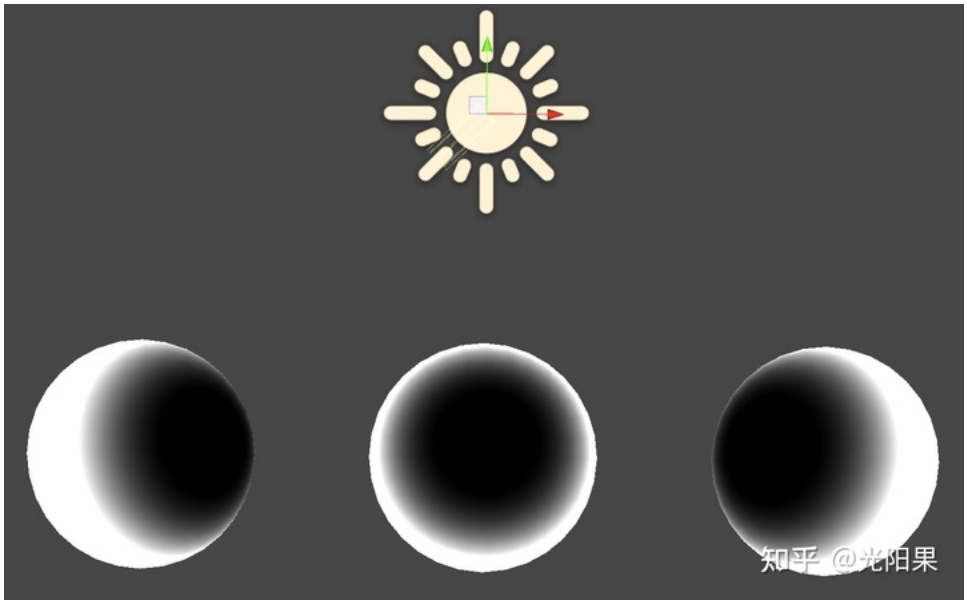
```
float3 StepSpecular = step(1-StepSpecularWidth*0.01,NH)*StepSpecularIntensity;
```



裁边边缘光(StepRim)

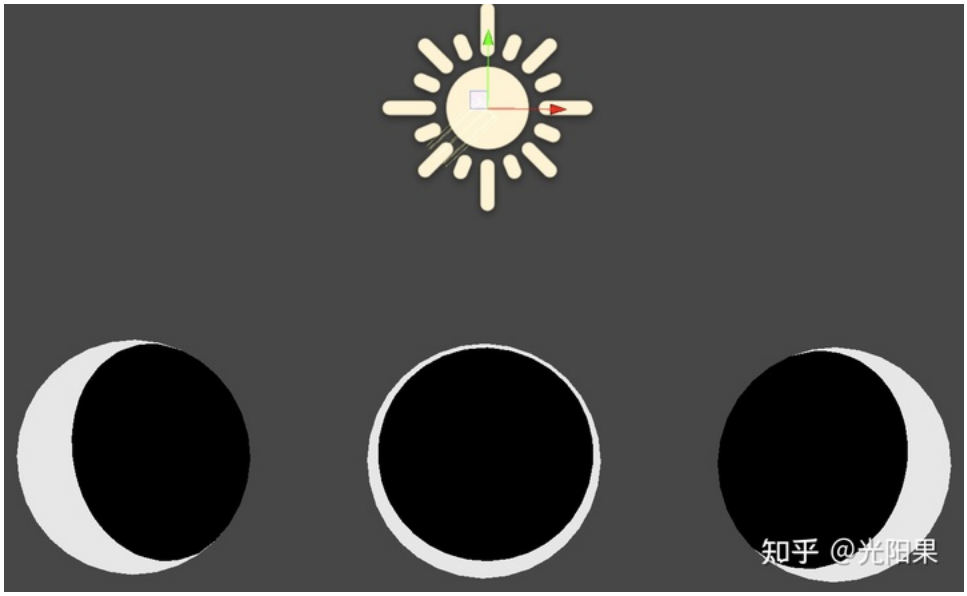
传统的边缘光有一个明显的明暗过渡

```
float3 Rim = pow(1-NV,RimExp)*RimIntensity;
```



卡通渲染里需要裁边，即 **裁边边缘光**

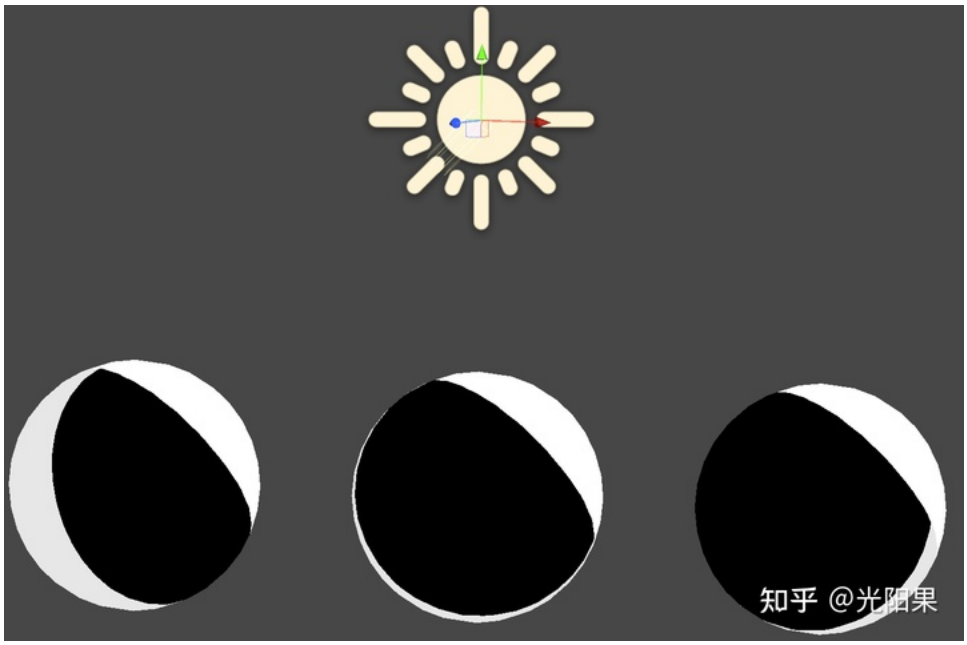
```
float3 Rim = step(1-RimWidth,1-NV )*RimIntensity;
```



裁边边缘光 结合亮部 暗部，可以组合出 **亮部裁边边缘光** 与 **暗部裁边边缘光**

```
float3 Threshold = step(_LightThreshold,NL01);
float3 Rim = step(1-Value1,1-NV )*Value2;
float3 RimDarkSide = lerp(Rim,0,Threshold);
float3 RimBrightSide = lerp(0,Rim,Threshold);

float3 FinalColor = RimDarkSide + Diffuse;
```

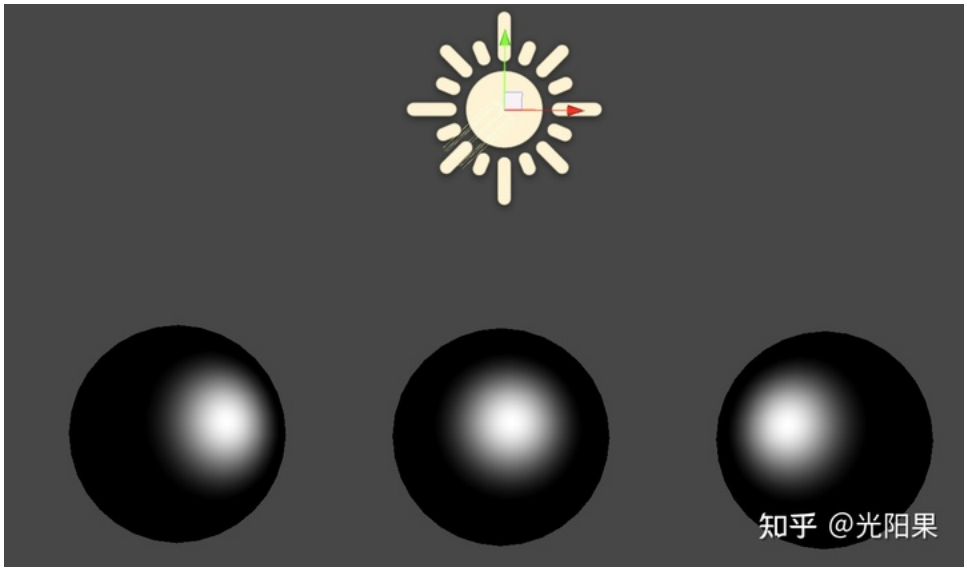
暗部裁边边缘光+裁边漫反射:



裁边视角光

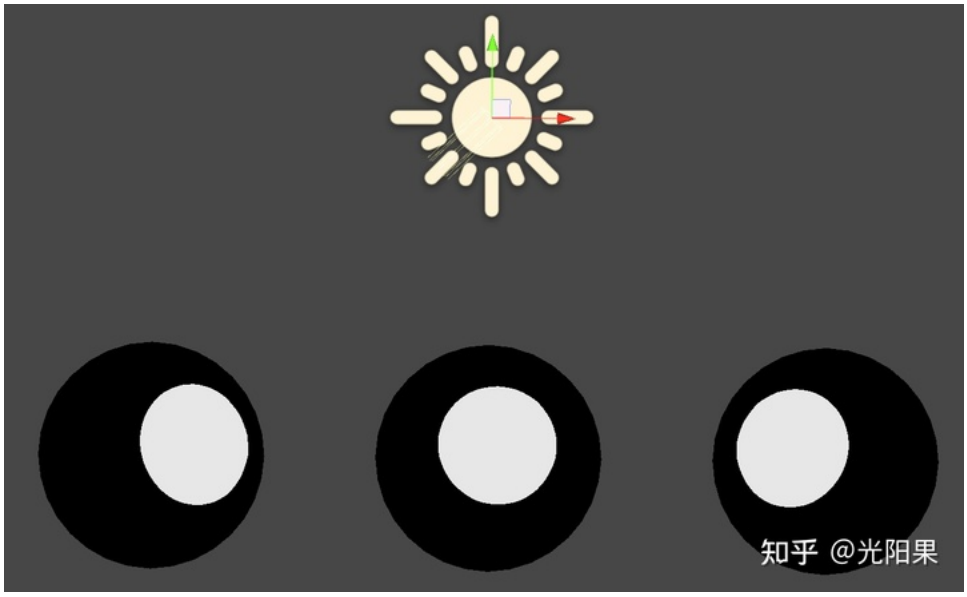
视角光，即是在眼睛看到部分更亮一些，传统的视角光:

```
float3 ViewLight = pow(NV,ViewLightExp)*ViewLightIntensity;
```



裁边视角光

```
float3 StepViewLight = step(1-StepViewLightWidth,NV)*StepViewLightIntensity;
```



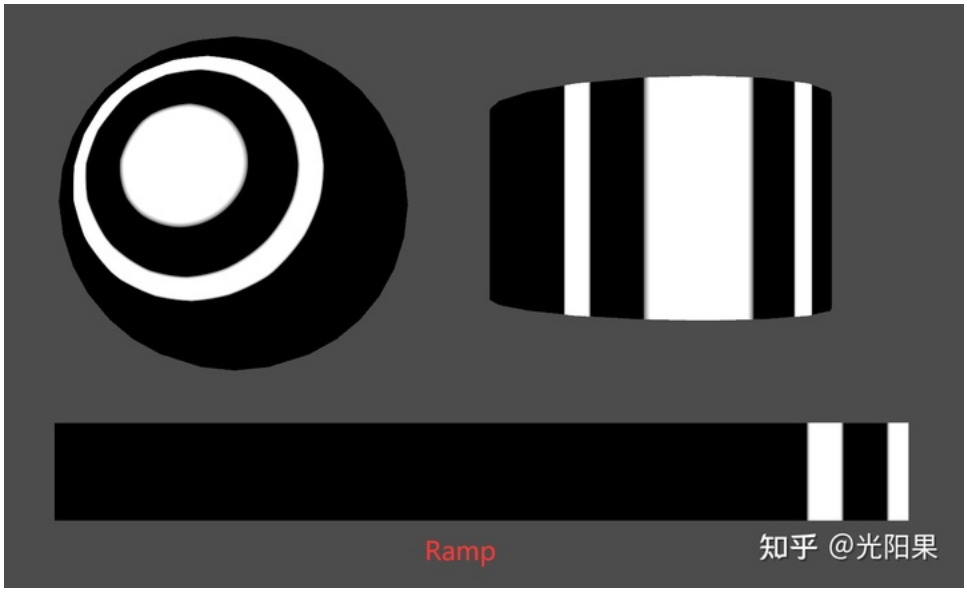
裁边光源光

裁边光源光即使对NL做一个Step，与视角无关，只与法线以及光源方向有关。本质上也可以将裁边视角光理解为裁边光源光，这光源方向只是与视角方向从重合了。

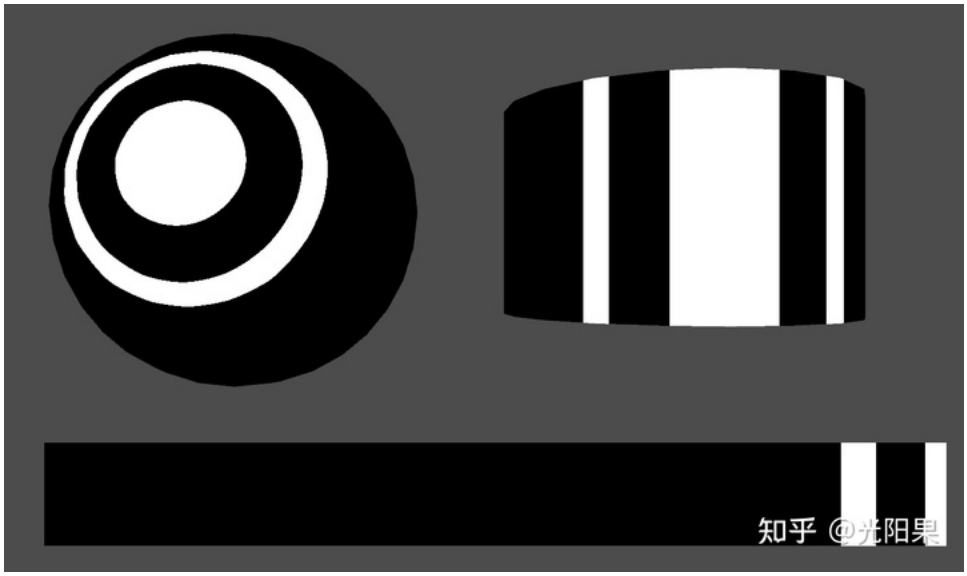
```
float StepLight = step(1-StepLightWidth,NL);
```

其实也可以对这个NL做一个Ramp，进行值域的重映射，以达到一些风格化的效果。

```
float StepLight = tex2D(_RampTex,NL01);
//如果不想要有中间灰度色,也可以用一个Step裁剪掉插值的部分
//float StepLight = step(0.9,tex2D(_RampTex,NL01));
//将贴图的滤波改为Point也能去掉这种过渡色
```



贴图的滤波模式为Point:



描边

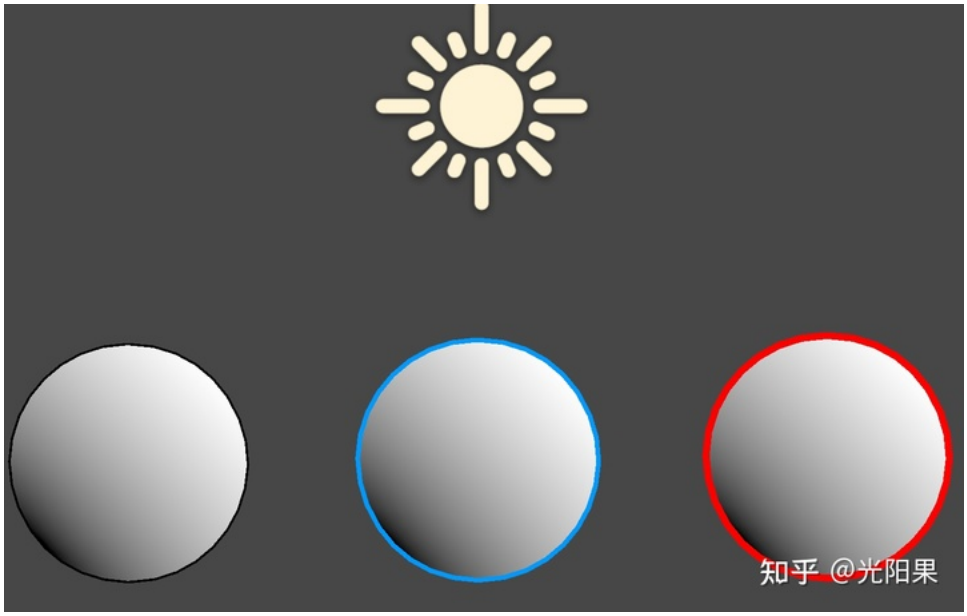
描边通常的做法是基于模型的，沿法先方向挤出外模型，可以提前对模型法线进行平滑处理，将平滑的法线存储到切线，或者顶点颜色，为了节省空间，可以只存储两个通道，因为法线是标准化的，在Shader里可以通过两个通道计算出第三个通道。描边的粗细可以用顶点色的一个通道来控制，(比如顶点色的Alpha)，卡通渲染里人物眼睛鼻子的地方通常不需要描边，那么就可以把这部分的Alpha通道填为零。

```
/// <summary>
/// 平滑法线，即是求出一个顶点 所在的所有三角面的法线的平均值
/// </summary>
/// <param name="mesh"></param>
private static void WriteSmoothNormalToTangent(Mesh mesh)
{
```

```
    Dictionary<Vector3, Vector3> vertexNormalDic = new Dictionary<Vector3, Vector3>();
    for (int i = 0; i < mesh.vertexCount; i++)
    {
        if (!vertexNormalDic.ContainsKey(mesh.vertices[i]))
        {
            vertexNormalDic.Add(mesh.vertices[i],mesh.normals[i]);
        }
        else
        {
            vertexNormalDic[mesh.vertices[i]] += mesh.normals[i];
        }
    }

    Vector4[] tangents = null;
    bool hasTangent = mesh.colors.Length == mesh.vertexCount;
    if (hasTangent)
    {
        tangents = mesh.tangents;
    }
    else
    {
        tangents = new Vector4[mesh.vertexCount];
    }

    for (int i = 0; i < mesh.vertexCount; i++)
    {
        Vector3 averageNormal = vertexNormalDic[mesh.vertices[i]].normalized;
        tangents[i] = new Vector4(averageNormal.x,averageNormal.y,averageNormal.z, 0f);//如果写入到顶点色需要将值映射到[0,1]，再在Shader中重新映
射到[-1,1]
    }
    mesh.tangents = tangents;
}
//有了平滑法线后再保存为Mesh，或者直接在DCC软件里做一个插件完成此功能
```

单独加一个描边Pass

```
//Outline
v2f vert(appdata v)
{
    v2f o;
    v.vertex.xyz += v.normal.xyz *_OulineScale*0.01*v.vertexColor.a;//用顶点色的Alpha通道控制描边粗细
    o.pos = UnityObjectToClipPos(v.vertex);
    return o;
}
float4 frag(v2f i) : SV_Target
{
    return _OutlineColor;
}
```



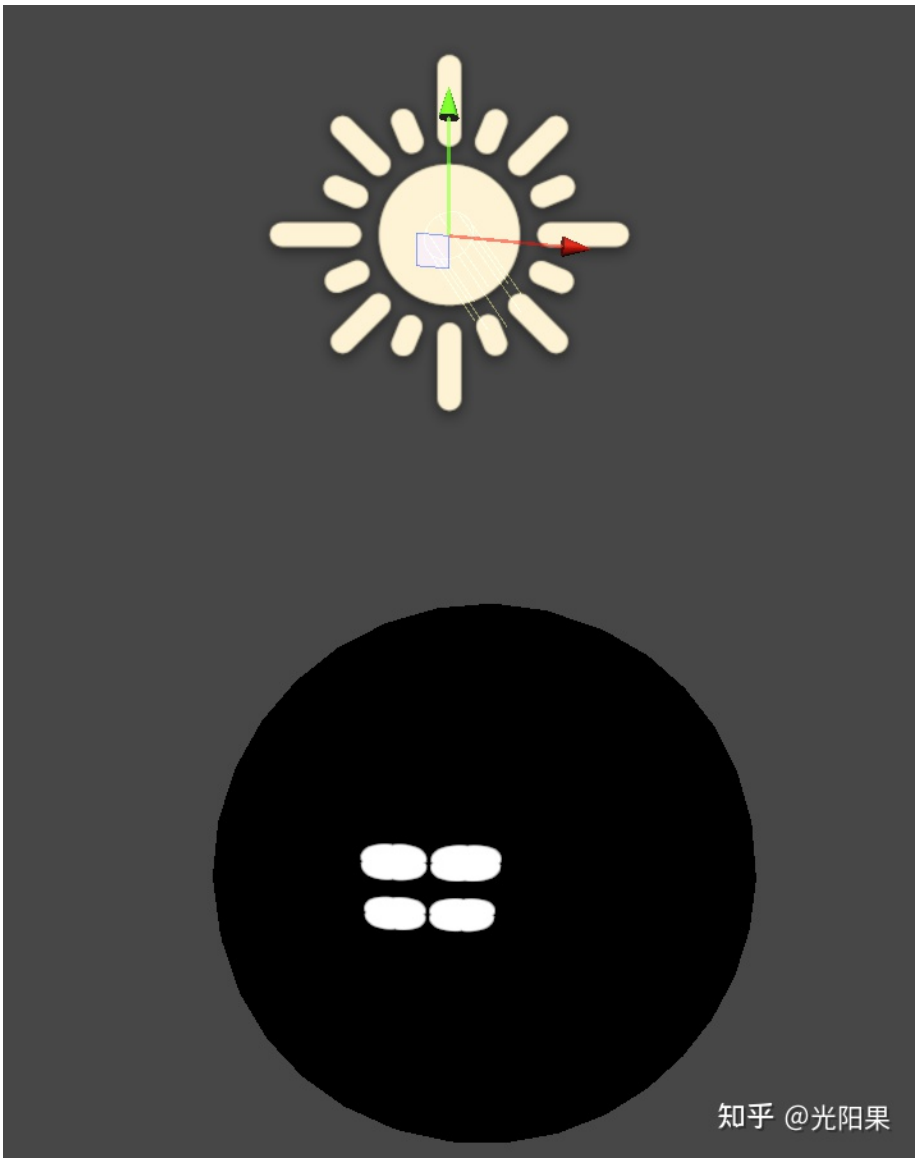
高光形变

传统的高光如BlinPhong 或者GGX都是一个圆形的光斑，卡通渲染里需要将其进行风格化处理，对高光进行形变可以达到此目的。

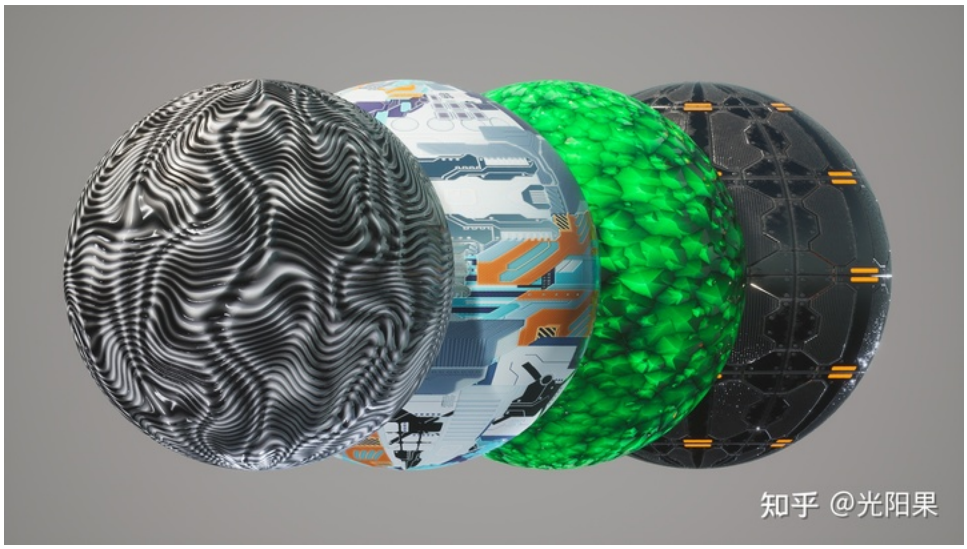
对BlinPhong进行高光形变：

```
StylizedSpecularParam Param;
Param.BaseColor =1;
Param.Normal = N;
Param.Shininess = _Shininess;
```

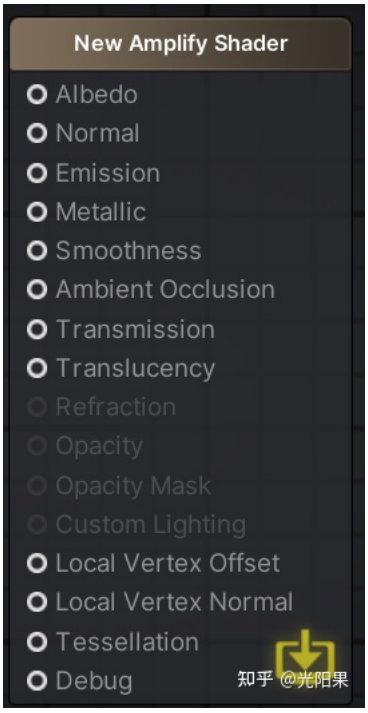
```
Param.Gloss = _Gloss;
Param.Threshold = _Threshold;
Param.dv = T;
Param.du = B;
float3 StylizedSpecular = StylizedSpecularLight_BlinPhong(Param,H);
```



卡通渲染中的材质表达



PBR渲染的材质



PBR参数

在PBR渲染中，可以使用BaseColor Roughness Metallic Normal等参数表达出大多数材质，但是卡通渲染中也想有各种特性的材质表达，因为没有一个完整的算法能将所有的卡通渲染将这些特性全部表达，这有两个解决方案：

A：每种特性都单独做一个Shader

B：做一个UberShader将所有特性都包含，然后在通过Mask进行材质分层

对于方案A 优点是Shader功能相对确定，GPU计算效率快，缺点是会增加DrawCall。

对于方案B 优点是DrawCall少，基本上一个通用的Shader可以满足大多数功能，减少贴图数量，缺点是会产生许多无用的GPU计算。

产生无用计算的原因在于：

GPU Shader中If语句都是先 计算出括号里面的内容 再根据条件值判断是否接受这个值。

这种运行方式与CPU不同，CPU是先根据条件值是否为真 再去判断，是否要执行括号里的内容。

一种成熟的解决方案是，方案A与B的结合，在通用的Shader中做出通用特性，每个特性不能消耗太多的计算，特殊的特性用单独的Shader。

罪恶装备Strive

后文中“罪恶装备”都是指 “罪恶装备Strive”，不同的版本做法会有一些差别。

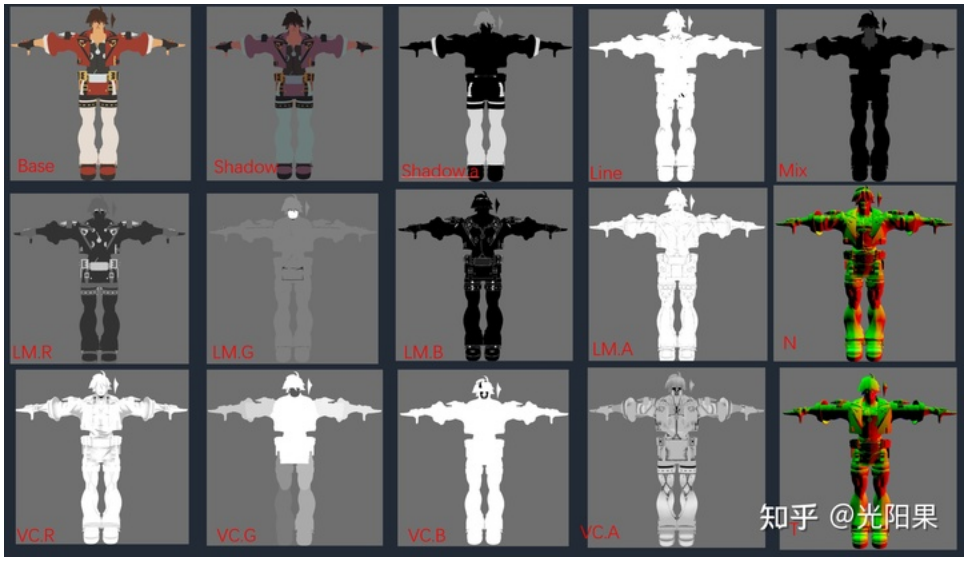
卡通渲染中最核心的明暗分割的公式就是

```
step(Threshold,dot(Light,Normal))
```

而罪恶装备就是这条公式上做文章。罪恶装备的人物模型面数很多，5W面起步，很多信息存储在模型模型中，如 顶点色、法线、UV，可以减少贴图带来的精度损失。

罪恶装备的模型UV是垂直水平展开的，使贴图采样可以减少锯齿，这种做法又叫做“本寸线”。

以下是罪恶装备模型与贴图的所有信息:

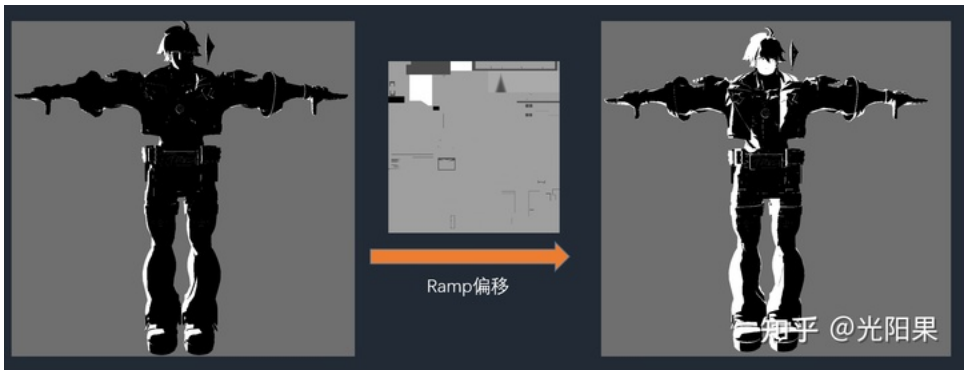


各个贴图以及模型信息的含义:

BaseMap :基础色 ShadowMap :暗部衰减色，与BaseMap相乘构成暗部 Line :磨损线条 Mix :标记的皮肤Mask LightMap.r :高光类型 LightMap.g :Ramp偏移值 LightMap.b :高光强度mask LightMap.a :内勾线Mask VertexColor.r:AO 常暗部分 VertexColor.g:用来区分身体的部位, 比如 脸部=88 VertexColor.b:渲染无用 VertexColor.a:描边粗细

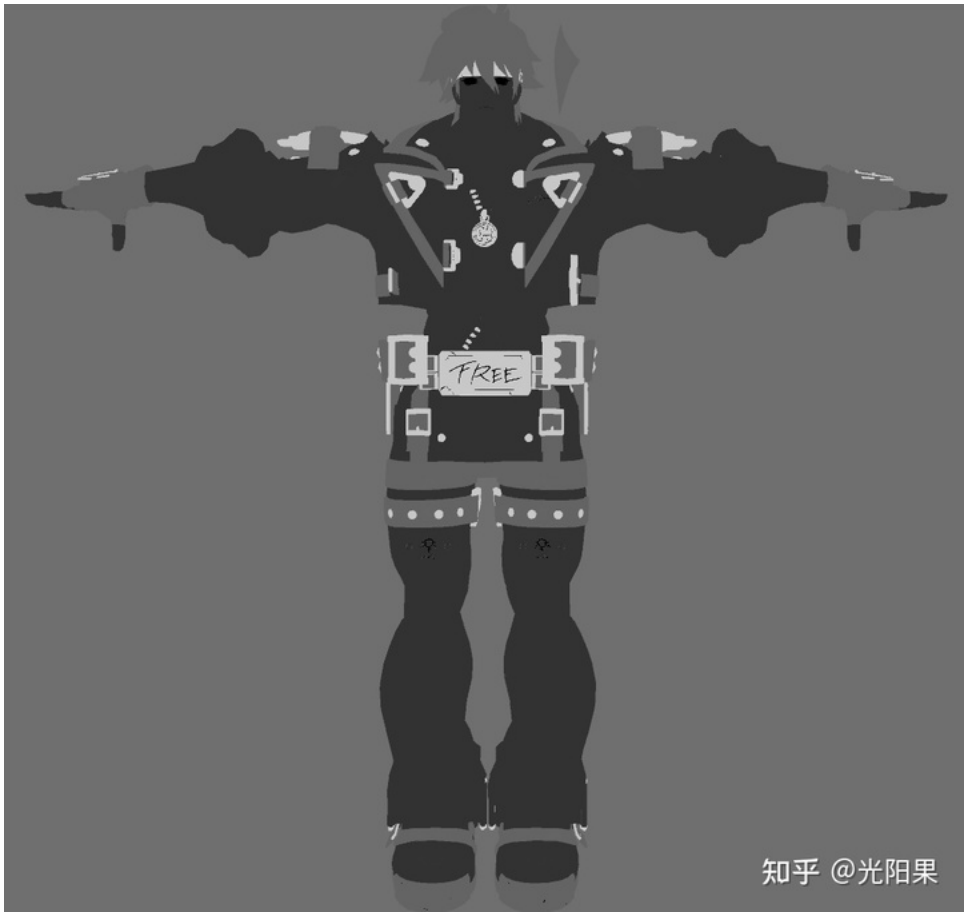
需要注意的点:

1. T :切线中存储的是编辑后的法线，艺术家可以跟好地控制明暗变化，修正后的平滑法线 也可使外勾边连续。
2. Ramp偏移值(LightMap.g)用来控制 在特定的光照角度下，那些区域更易感光，即更容易处于亮部，越不易感光的区域值越小，反之则越大，这张贴图很艺术家。
   以下是在固定光照角度下，是否加Ramp偏移值的区别:

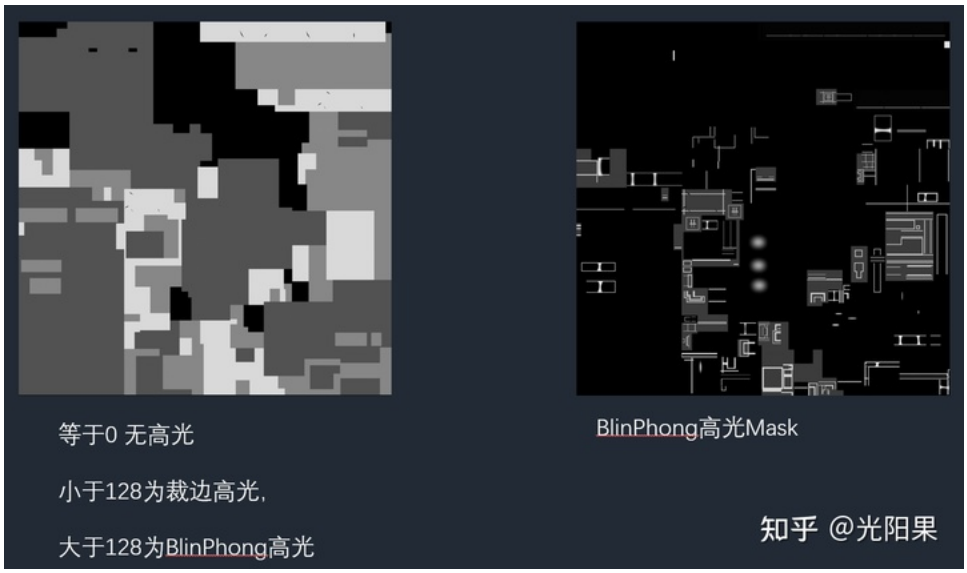


1. 材质类型(LightMap.r)用一张贴图控制材质的类型，根据贴图值得不同分为以下几种材质
   Layer<60 普通材质 无高光 暗部有边缘光 Layer>60&&Layer<190 皮革材质 有视角裁边高光(StepNV) 暗部有边缘光 Layer>190 金属材质 BlinPhong高光 光源裁边高光(StepNL)

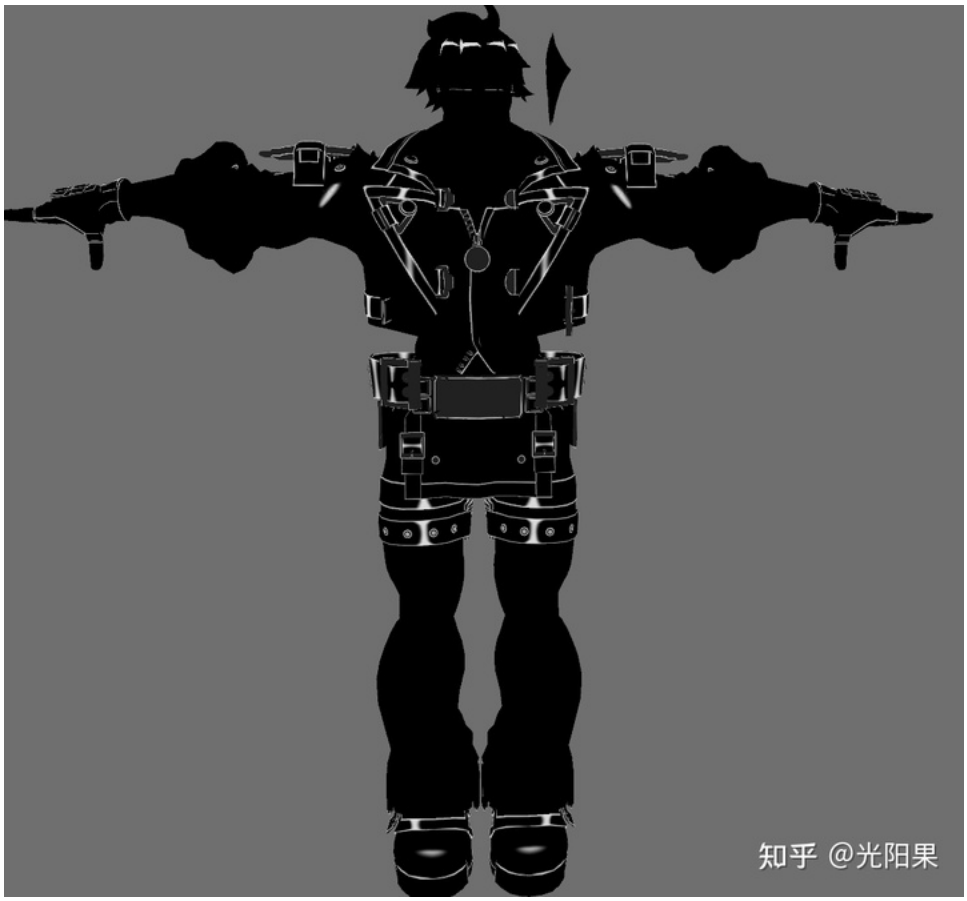
用贴图来区别材质类型:


FREE
知乎 @光阳果

1. 高光类型(LightMap.b) 用一张贴图控制高光的类型，等于0 无高光 小于128为裁边高光(裁边高光仅出现在皮革位置)，大于128为BlinPhong高光强度Mask

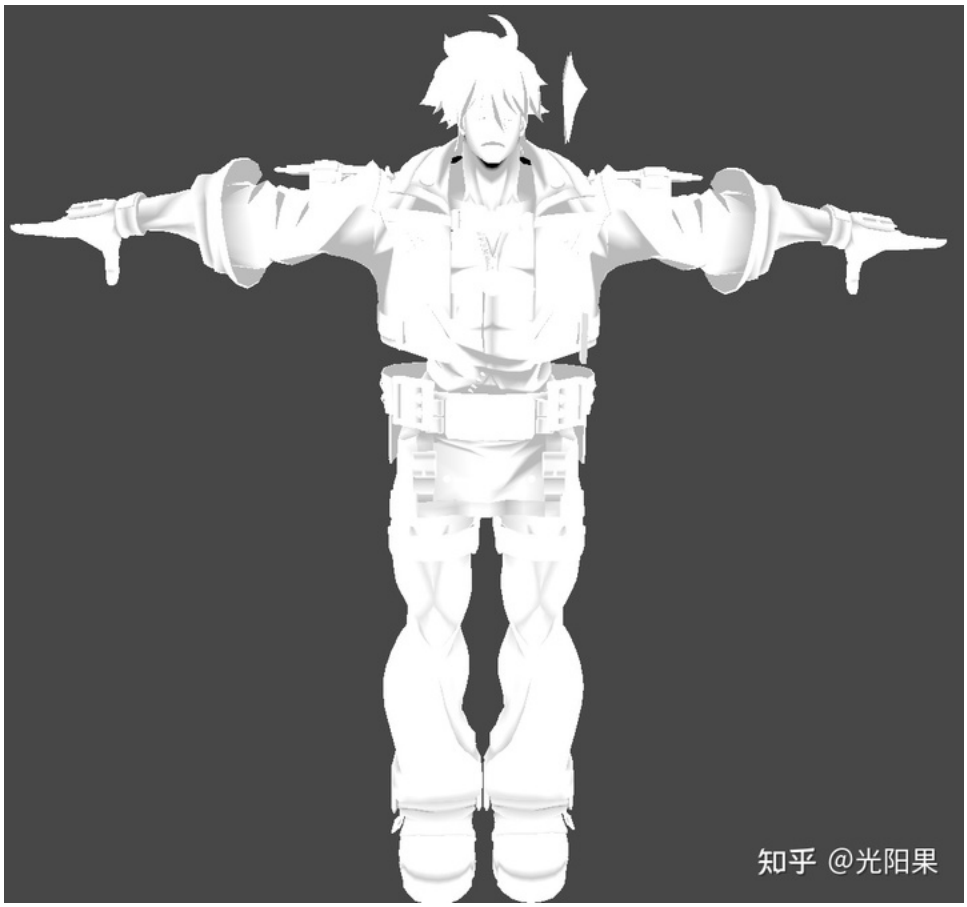


高光强度Mask在模型上的显示:

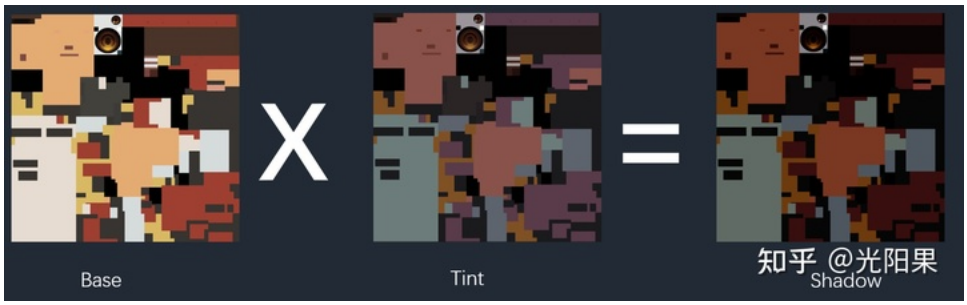

知乎 @光阳果

5.ShadowAO(VertexColor.r)用控制区域常暗，即这部分区域永远处于暗部。

区域场暗的部分:


知乎 @光阳果

5.暗部颜色，由BaseMap乘上一个颜色通透图来得到，相当于PhotoShop的正片叠底，其实也可以直接画暗部贴图出来，没必要乘一次(碧蓝幻想就是直接画出暗部贴图)



身体

根据前面的信息可以做出漫反射部分代码如下

```
//获取信息
float SpecularLayerMask         = LightMap.r;//高光类型
float RampOffsetMask            = LightMap.g;//Ramp偏移值
float SpecularIntensityMask     = LightMap.b;//高光强度mask
float InnerLineMask             = LightMap.a;//内勾线Mask
float ShadowAOMask                = VertexColor.r;//AO 常暗部分
                                  // VertexColor.g;//用来区分身体的部位, 比如 脸部=88
                                  // VertexColor.b;//渲染无用
float OutlineIntensity            = VertexColor.a;//描边粗细

//裁边漫反射
float NL01 = 0.5*NL+0.5;
float Threshold = step(_LightThreshold,(NL01 + _RampOffset +RampOffsetMask )*ShadowAOMask);
BaseMap*= InnerLineMask;//磨损线条
BaseMap = lerp(BaseMap,BaseMap*LineMap,_LineIntensity); //控制磨损线条强度
float3 Diffuse = lerp( lerp(  ShadowMap*BaseMap,BaseMap,_DarkIntensity),BaseMap,Threshold);
```

边缘光可以使用传统的NV做一个Step，但这种做法在某些角度下，会出现大面积的边缘光，比如角色蹲下时的腿部，可以使用一张额外的边缘光Mask来控制哪些区域出现。


知乎 @光阳果

裁边边缘光，将法线转到视角空间下，直接取X分量做为边缘光的Mask，只出现在暗部。

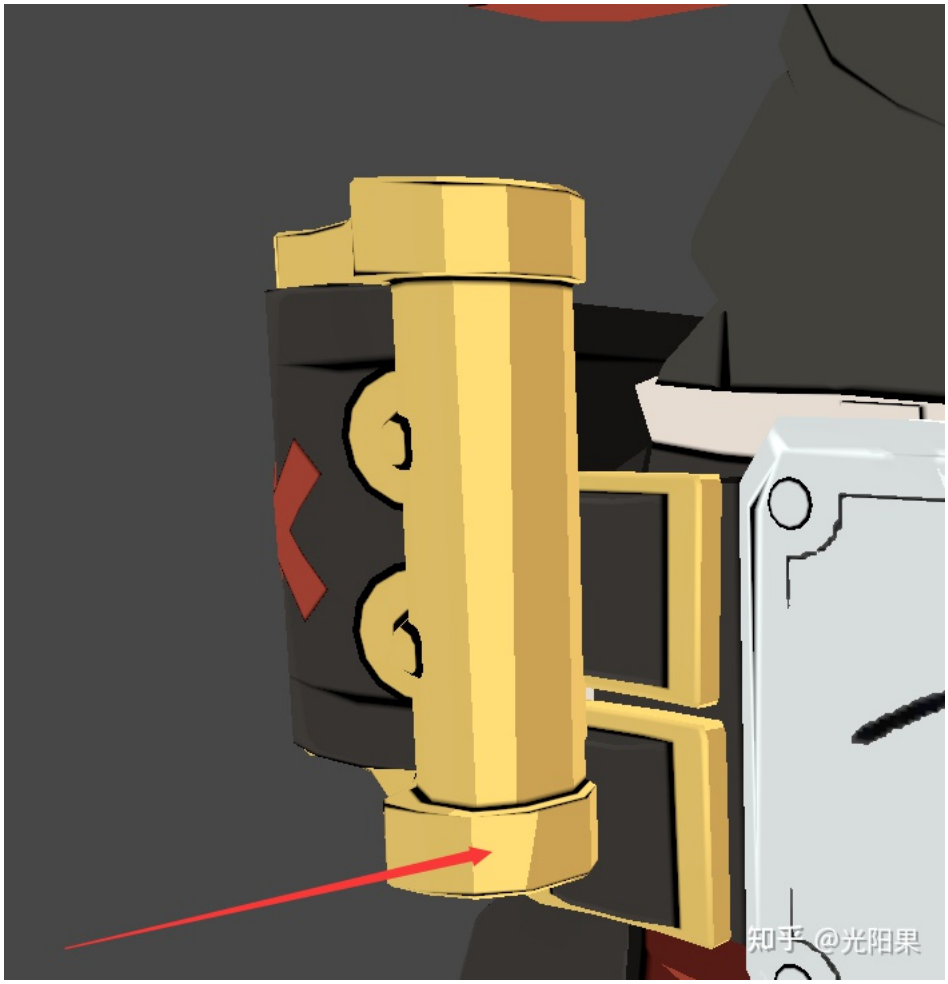
```
/*=========================边缘光 =========================*/
float3 N_VS = mul((float3x3)UNITY_MATRIX_V, T);
float3 Rim = step(1-_RimWidth,abs( N_VS.x))*_RimIntensity*BaseMap;
Rim = lerp(Rim,0,Threshold);
Rim = max(0,Rim);
```

高光

金属除了有BlinPhong高光外，还额外有一个与视角无关的 光源裁边高光，这可以使用NL加一个Step来模拟。

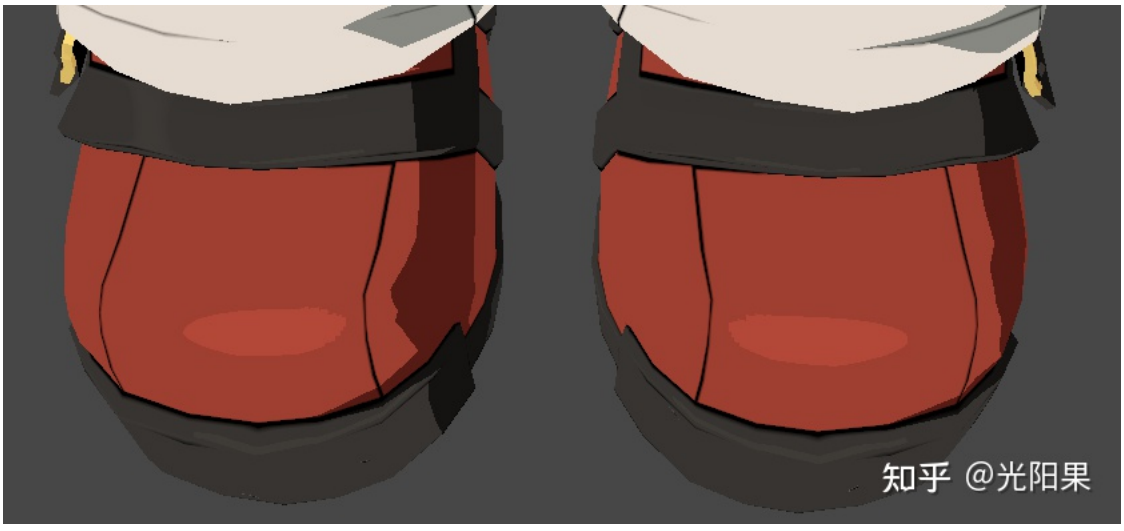
```
float3 MetallicStepSpecular = step(NL,_MetallicStepSpecularWidth)*_MetallicStepSpecularIntensity*BaseMap;
```

金属的光源裁边高光:


知乎 @光阳果

裁边高光仅出现在皮革上，有的特殊材质上也有裁边高光，但那是单独做的，不在通用材质的考虑范围。

鞋子上的裁边高光:



高光代码:

```
float3 Specular =0;
Specular = pow(saturate(NH),_SpecularPowerValue)*_SpecularIntensity * SpecularIntensityMask*BaseMap ;
Specular = max(Specular,0);

//LayerMask
// 0            : 普通  无高光
// 50           : 普通  无高光 有边缘光
// 100          : 皮革  高光   有边缘光
// >=200        ：金属  有裁边高光

float LinearMask = pow(LightMap.r, 1 / 2.2);
float Layer = LinearMask * 255;

//金属
if(Layer>190)
{
    float3 MetallicStepSpecular = step(NL,_MetallicStepSpecularWidth)*_MetallicStepSpecularIntensity*BaseMap;
    MetallicStepSpecular = max(0,MetallicStepSpecular);
    Specular += MetallicStepSpecular;
    // return Red;
}

//普通 无高光 暗部有边缘光
if(Layer<=60 && Layer>0)
{
   float SpecularIntensity = pow(SpecularIntensityMask,1/2.2)*255;
   float StepSpecularMask = float(SpecularIntensity<180 && SpecularIntensity>0);// step(128,SpecularIntensity)* step(0,SpecularIntensity)
   float3 LeatherSpecular = step(1-_LeatherStepSpecularWidth,NV)*_LeatherStepSpecularIntensity*BaseMap * StepSpecularMask;
   LeatherSpecular = max(0,LeatherSpecular);
   Specular = lerp(Specular, LeatherSpecular,StepSpecularMask);
}

//皮革 LightMap.r<128 && LightMap.r>0 的部分 是裁边高光的Mask
if(Layer>60 && Layer<190)
{
   float SpecularIntensity = pow(SpecularIntensityMask,1/2.2)*255;
   float StepSpecularMask = float(SpecularIntensity<128 && SpecularIntensity>0);// step(128,SpecularIntensity)* step(0,SpecularIntensity)
   float3 LeatherSpecular = step(1-_LeatherStepSpecularWidth,NV)*_LeatherStepSpecularIntensity*BaseMap * StepSpecularMask;
   LeatherSpecular = max(0,LeatherSpecular);
   Specular = lerp(Specular, LeatherSpecular,StepSpecularMask);
}
```

头发

头发也分明暗部，头发高光仅显示在亮部，暗部的高光可以使用一个数值来调整:

```
//头发 漫反射
float NL01 = 0.5*NL+0.5;
float Threshold = step(_LightThreshold,(NL01 + _RampOffset +RampOffsetMask )*ShadowAOMask);
BaseMap*= InnerLineMask;
BaseMap = lerp(BaseMap,BaseMap*LineMap,_LineIntensity);
float3 Diffuse = lerp( lerp(  ShadowMap*BaseMap,BaseMap,_DarkIntensity),BaseMap,Threshold);
//头发 高光
float3 Specular =0;
Specular = SpecularIntensityMask*BaseMap*lerp(_HairSpecularDarkIntensity,_HairSpecularBrightIntensity,Threshold);
//高光仅在亮部显示
FinalColor = Diffuse + Specular;
```

头发 裁边漫反射与高光:

脸部

脸部通过在DCC软件编辑法线之后，在光照旋转时可以获得以下的明暗过度效果:


知乎 @光阳果

罪恶装备中，脸部和头发使用同一个材质，并用VertexColor.g通道来区分，暗部有的模型用贴图控制，有的直接用颜色，因此代码中考虑到了这种情况。

```
//脸部
if (VertexColor.g<0.9)
{
    NL = dot(T.xz,L.xz);
    float halfLambert = 0.5*NL+0.5;
    float Threshold = step(_LightThreshold,(halfLambert + _RampOffset +0 )*ShadowAOMask);
    float3 FaceShadowSide = 0;
    #ifdef FACESHADOWMODE_TEX
        FaceShadowSide =  ShadowMap*BaseMap;
    #endif

    #ifdef FACESHADOWMODE_COLOR
        FaceShadowSide =  BaseMap*_FaceShadowColor;
    #endif

    float3 Diffuse = lerp( lerp( FaceShadowSide ,BaseMap,_DarkIntensity),BaseMap,Threshold);

    return Diffuse.xyzz;
}
```

贴花

罪恶装备的贴花，使用了单独的Mesh。贴画中可以画一些图案，而不规则的图案是会产生锯齿，因此罪恶装备的贴花使用了单独的Mesh 与 额外的贴花贴图，已减少锯齿。

贴花贴图:

单独Mesh的贴花:

代码中使用clip裁剪掉没有图案的像素，也可以使用透明混合的方式。

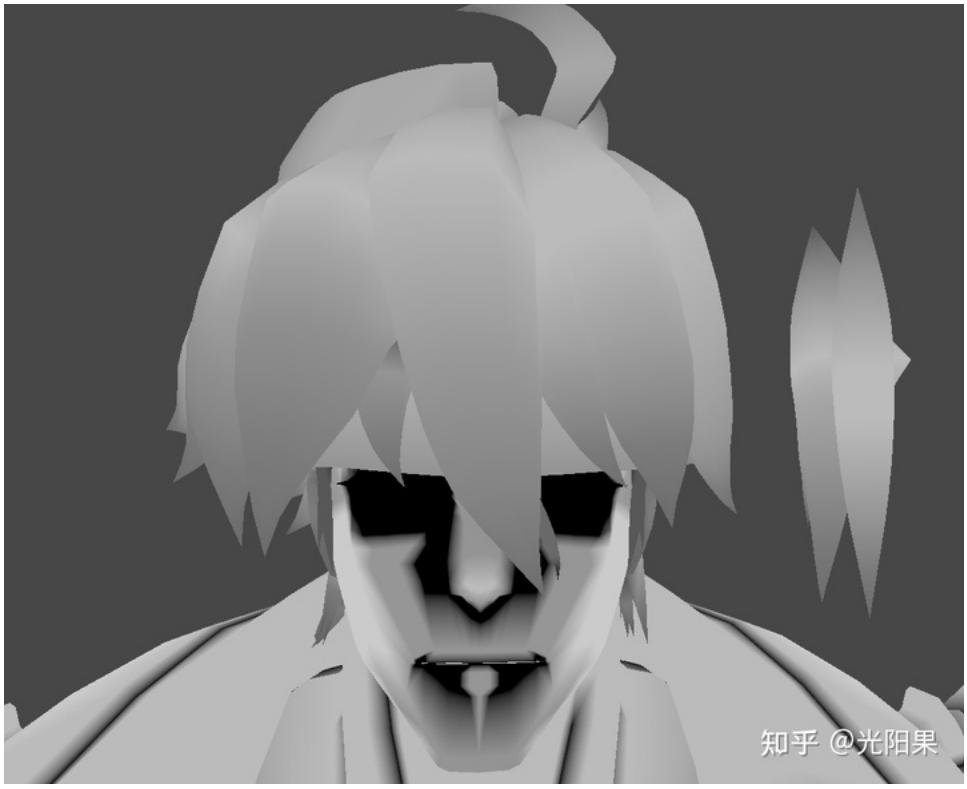
```
clip (0.1-DecalMap.r );
Decal = DecalMap;
```

自发光

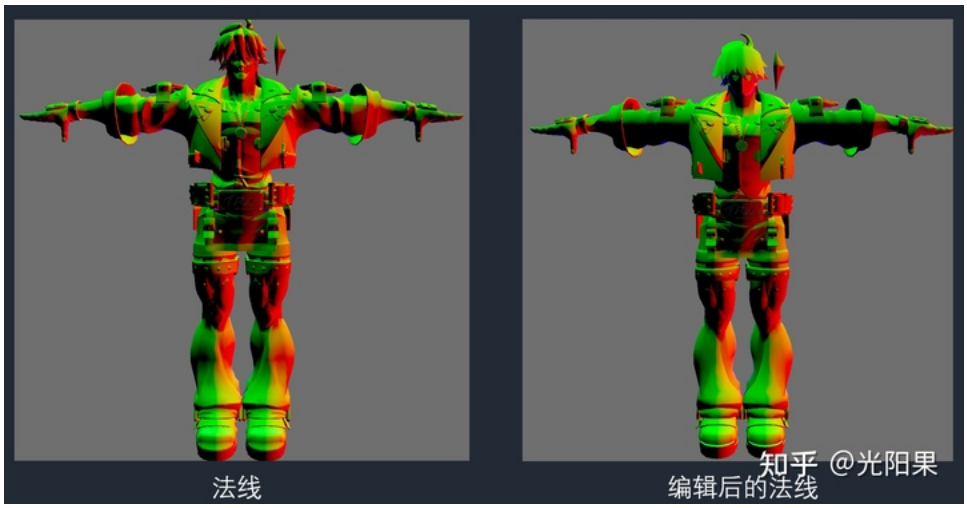
罪恶装备模型的自发光也是做了单独的Mesh。

描边

传统的按法线基础模型，会导致硬边描出的线产生“破裂”的效果，使用平滑后的法线可以有效的避免这个问题。罪恶装备处理过的法线存储在tangent通道，用顶点色的alpha值控制描边的粗细，比如鼻子眼睛出不需要描边，这里的顶底色Alpha通道直接填黑；比如头发上的顶点Alpha通道还有个过渡处理，可以实现描边从上倒下的粗细渐变。


知乎 @光阳果

将法线颜色输出可以看出编辑后的法线，显示更平滑。



描边代码:

```
//需要Cull Front，仅显示背面
v2f vert(appdata v)
{
    v2f o;
     v.vertex.xyz += v.tangent.xyz *_OulineScale*0.01*v.vertexColor.a;
    o.pos = UnityObjectToClipPos(v.vertex);
    return o;
}
float4 frag(v2f i) : SV_Target
{
    return _OutlineColor;
}
```

描边Pass仅显示背面，如果去掉模型正常显示，那么效果就是一个纯黑，为了方便观察，将模型显示为纯白:

完整渲染图:

SOL

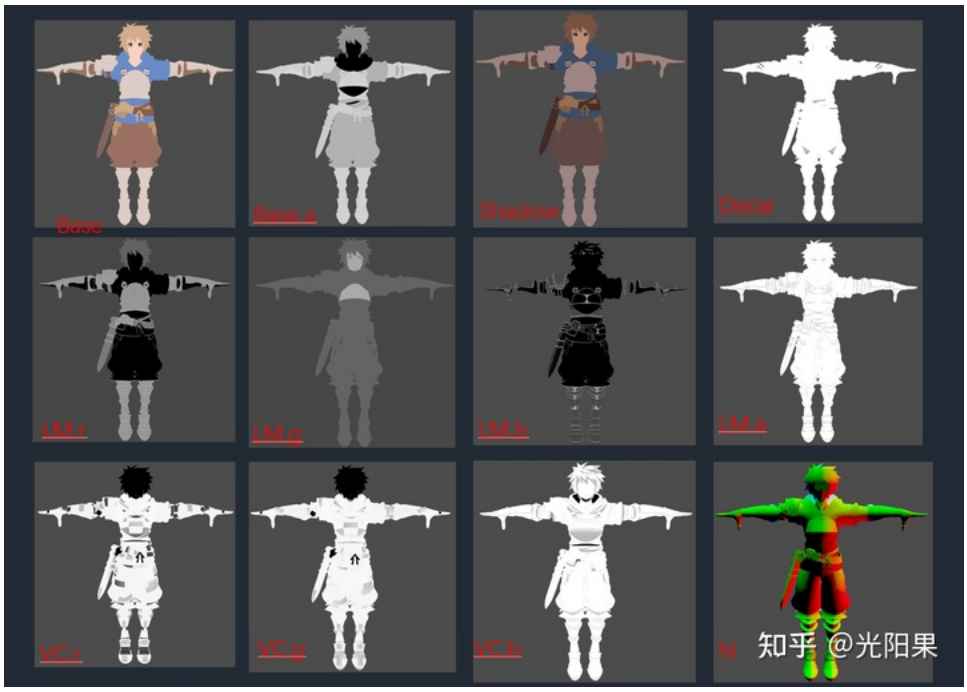
AXL


知乎 @光阳果

碧蓝幻想

碧蓝幻想与罪恶装备都是同一个公司ARC(Arc System Works) 开发，大体做法基本一致，只是有一些改进，因此相同的部分不再重复说明。

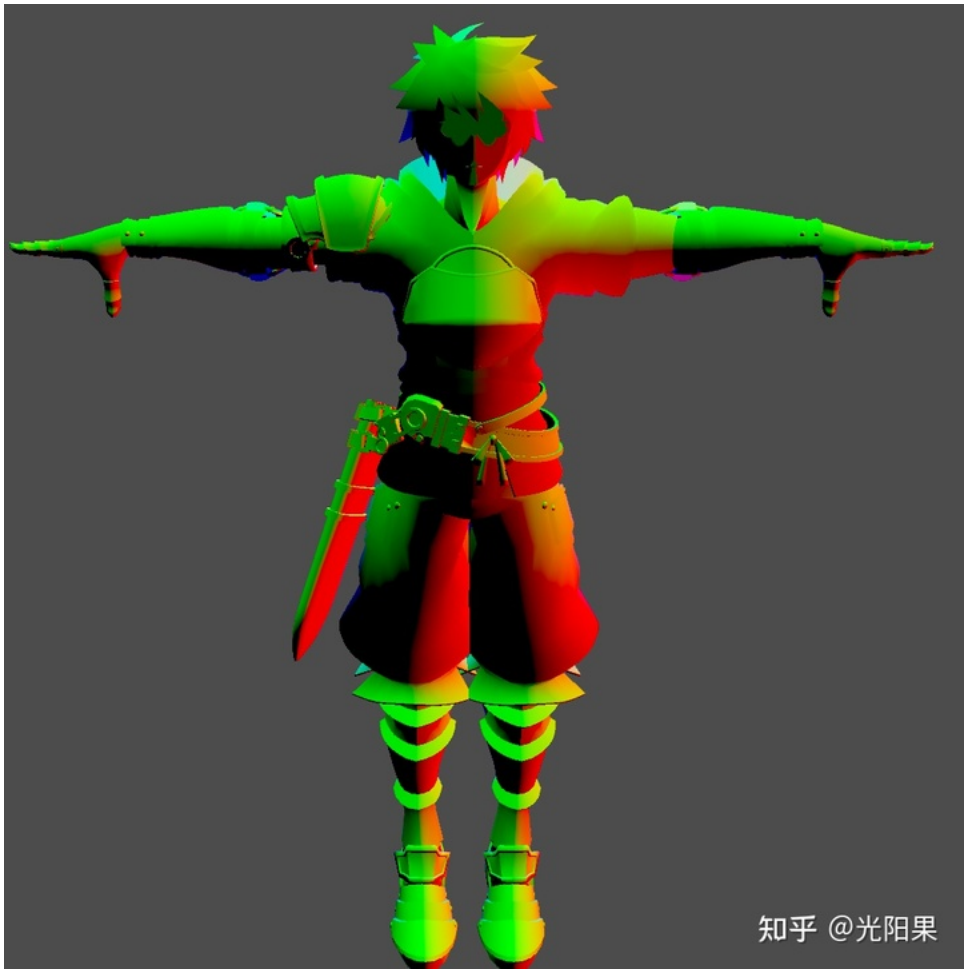
碧蓝幻想贴图与模型信息:



需要注意的点:

1. UV垂直水平展开，与罪恶装备一致，有“本村线”

1. 法线N是编辑平滑后的法线，可直接进行光照计算与平滑描边



3.暗部颜色贴图是直接画出来的，而不是像罪恶装备那样需要，用Base乘上Tint色

4.ShadowAO(VertexColor.b)是常暗区域


知乎 @光阳果

5.Ramp偏移值(LightMap.g)用来控制 在特定的光照角度下，那些区域更易感光，即更容易处于亮部，越不易感光的区域值越小，反之则越大。


知乎 @光阳果

6.Decal存储着磨损线条，可以用一个Lerp来控制磨损的程度


知乎 @光阳果

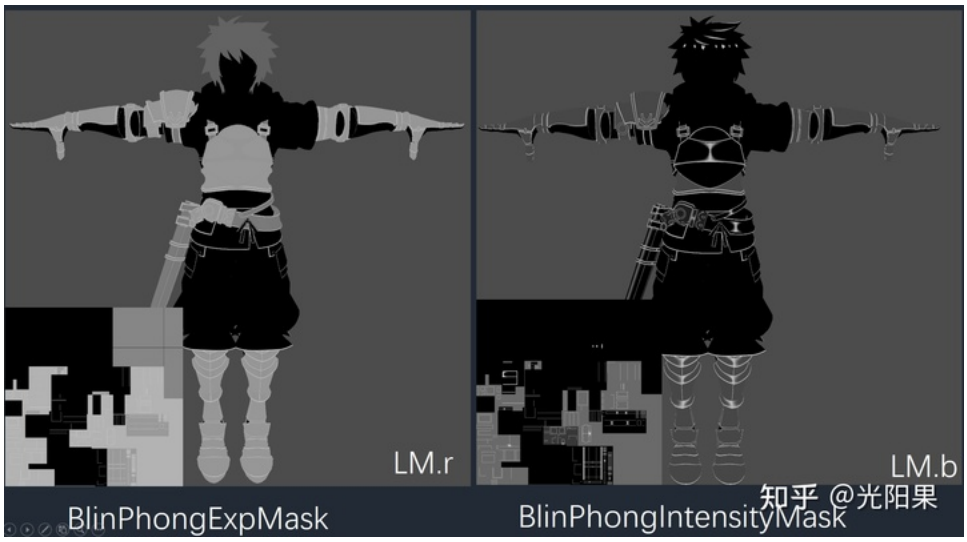
7.材质的分层信息存储在Base.a，总共有11个层，每个层都有属于自己的特性，特性比较多


知乎 @光阳果

1. 相比于罪恶装备多了一张BlinPhong的指数参数贴图(LightMap.r)，有这张图会使光泽度更细腻,但这更多地是一种美术风格上的选择。

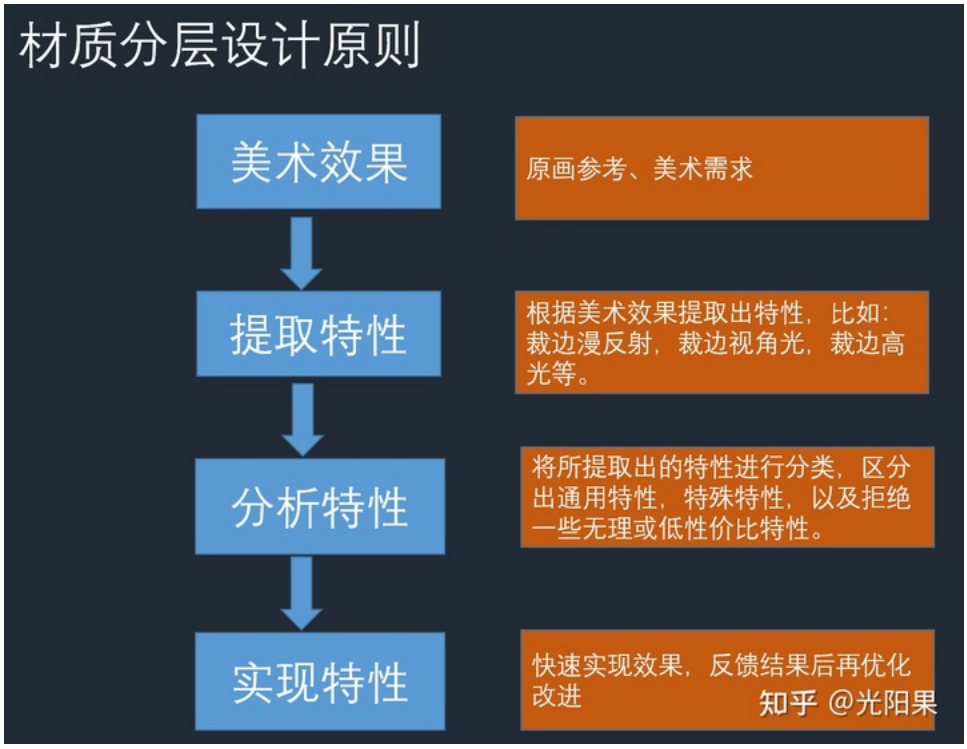
具体的用法是直接乘在BlinPhong的指数里:

```
float3 Specular = pow(NH,SpecularExp * SpecularExpMask) * SpecularIntensityMask*SpecularIntensity;
```



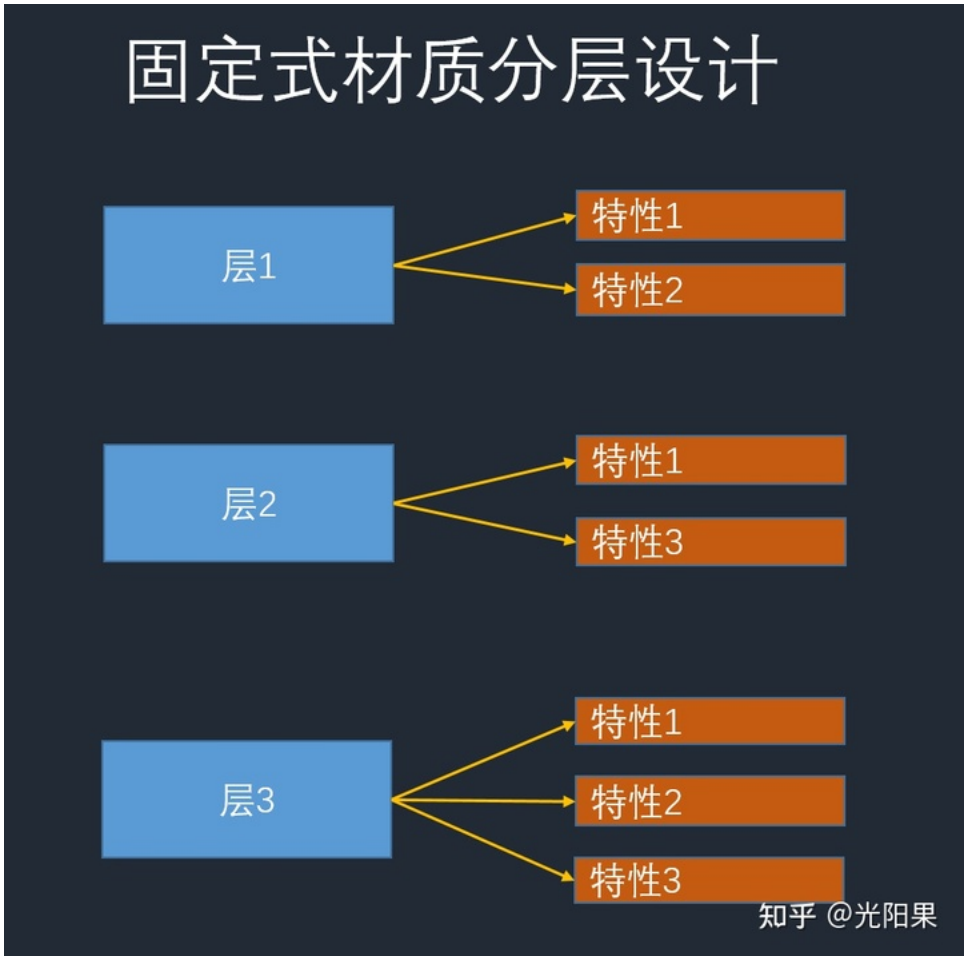
材质分层的设计

材质分层设计原则:


材质分层设计原则
美术效果
原画参考、美术需求
提取特性
根据美术效果提取出特性，比如：裁边漫反射，裁边视角光，裁边高光等。
分析特性
将所提取出的特性进行分类，区分出通用特性，特殊特性，以及拒绝一些无理或低性价比特性。
实现特性
快速实现效果，反馈结果后再优化改进
知乎 @光阳果

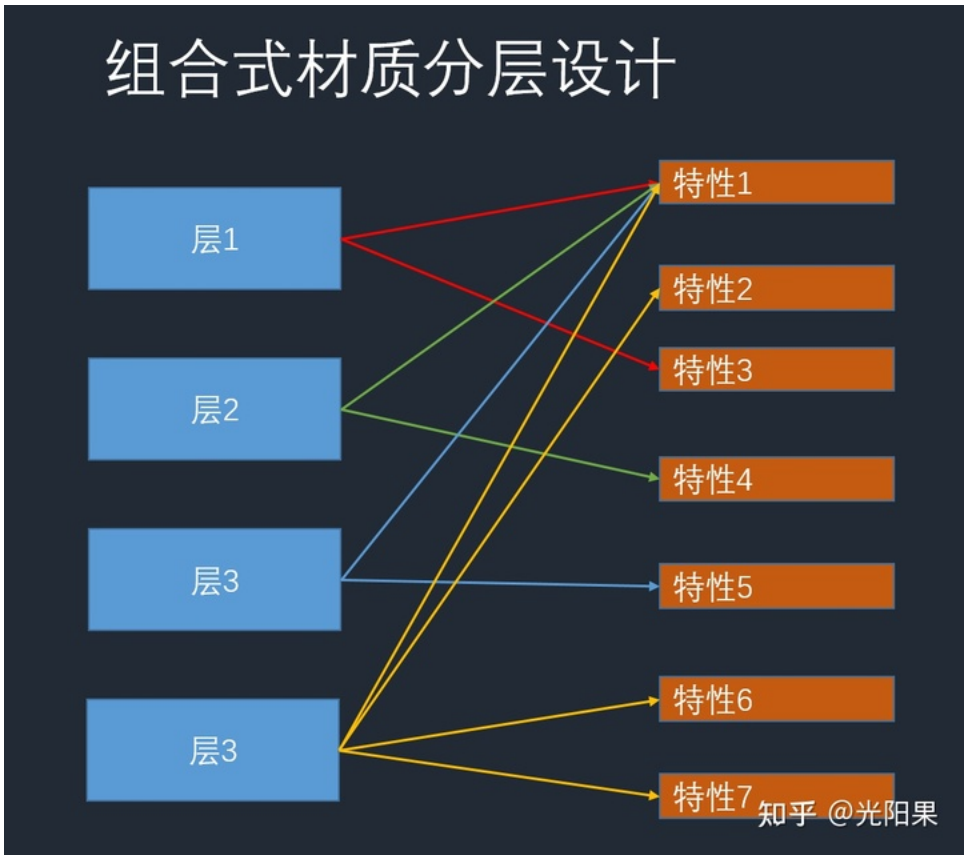
固定式材质分层设计

碧蓝幻想相比于罪恶装备有了更多材质分层的概念，而每个层中都有属于自己的特性。碧蓝幻想有11个材质分层，Shader的设计思路如下:


固定式材质分层设计
层1
特性1
特性2
层2
特性1
特性3
层3
特性1
特性2
特性3
知乎 @光阳果

组合式材质分层设计

另外一种材质分成的设计思路是组合式的，即每个层都可以自由地选择所需要的特性。艺术家在使用时，只需在相应的特性上打上勾，那么就能使用该特性,可以为艺术家提供更高的自由度！


组合式材质分层设计
层1
层2
层3
层3
特性1
特性2
特性3
特性4
特性5
特性6
特性7
知乎 @光阳果

碧蓝幻想的材质特性

这两个角色中，包含的特性如图所示，其他英雄中还有一些其他特性，要全部还原需要大量观察与检验，但按照这个思路去开发一套自己的渲染方案是简单明了的。

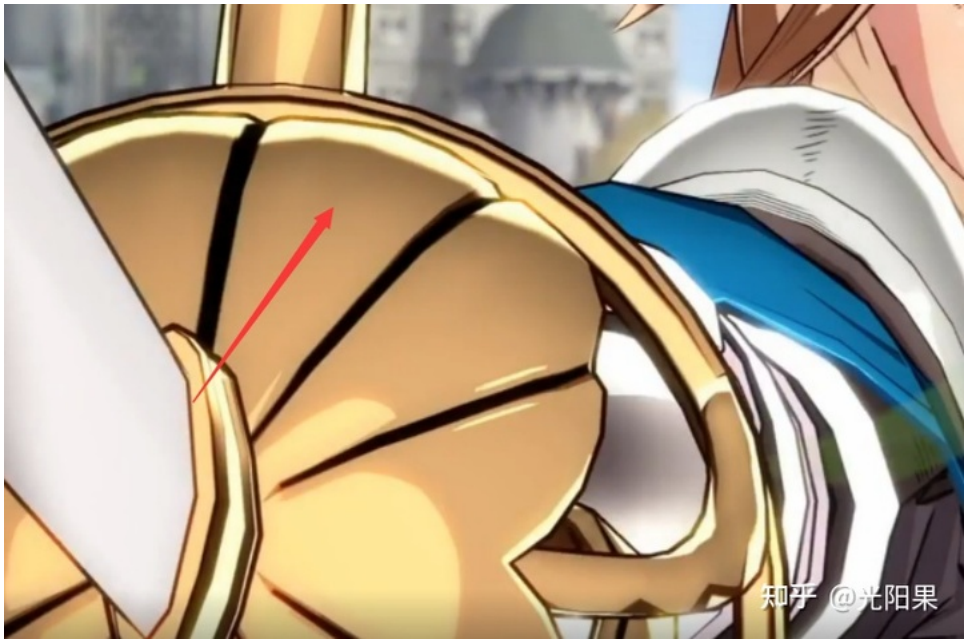
根据吐血观察与测试，碧蓝幻想的材质分层与特性如下:

```
//BaseMap.a是材质分层的信息
//0.0  - 0.2    =>基础材质:一个明暗面 无边缘光 无高光Mask无裁边高光 无视角光 无裁边视角光  (可能包含皮肤 布料 金属)
//0.21 - 0.25   =>布料1  :两层明暗面 裁边缘光 无高光Mask无裁边高光 无视角光 无裁边视角光
//0.26 - 0.30   =>布料2  :两层明暗面 裁边缘光 有高光Mask无裁边高光 无视角光 无裁边视角光
//0.31 - 0.46   =>布料3  :两层明暗面 裁边缘光 无高光Mask无裁边高光 无视角光 无裁边视角光
//0.47 - 0.55   =>皮革1  :一层明暗面 无边缘光 有高光Mask无裁边高光 无视角光 无裁边视角光
//0.56 - 0.57   =>皮革2  :两层明暗面 裁边缘光 有高光Mask有裁边高光 无视角光 无裁边视角光
//0.58 - 0.61   =>皮革3  :两层明暗面 裁边缘光 有高光Mask无裁边高光 无视角光 无裁边视角光    (可能包含 皮革 金属)
//0.62 - 0.66   =>金属1  :两层明暗面 裁边缘光 有高光Mask有裁边高光 无视角光 无裁边视角光
//0.67 - 0.71   =>金属2  :两层明暗面 裁边缘光 有高光Mask无裁边高光 无视角光 无裁边视角光
//0.72 - 0.87   =>金属3  :一层明暗面 裁边缘光 有高光Mask无裁边高光 无视角光 无裁边视角光
//0.88 - 1.0    =>金属4  :一层明暗面 裁边缘光 有高光Mask无裁边高光 无视角光 无裁边视角光  (布料 金属 英雄就一层基础材质+金属4材质)
```

为此只挑一些有差别的特性分析:

1.金属明暗过度。有些金属，会有明暗过渡的特性，这种特性可以用菲尼尔来模拟，只不过这个菲尼尔不是用来点亮边缘，而是加黑边缘，只需要将MetallicFresnelIntensity调成一个0到1之间的小数值，那么就能有明暗过渡的效果。

```
Diffuse = Diffuse*  pow(1-saturate(NV),MetallicFresnelExp)*MetallicFresnelIntensity;
```



2.裁边光种类丰富，有 StepNV StepNL StepNH，每种特性开放，艺术家自己可选。


知乎 @光阳果

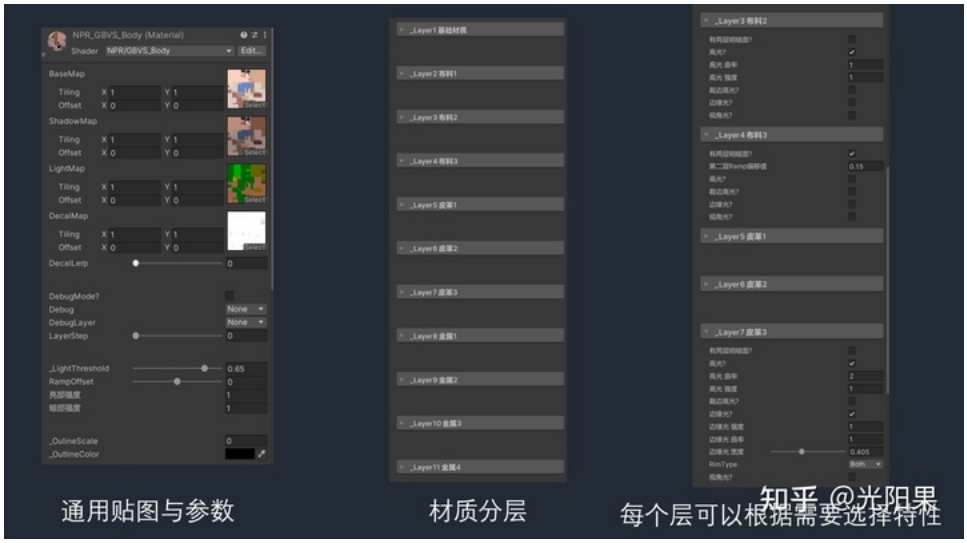
实现

考虑到碧蓝幻想特性较多的情况，我使用了，组合式分层材质设计。主要分为三部分

1.通用贴图与参数

2.材质分层

3.特性选择开关



首先，定义出每层所需参数的Struct:

```
struct NPRData
{
    //每层不一样的参数
    bool HasTwoSide,HasRim,HasSpecular,HasStepSpecular,HasViewLight;
    float SpecularExp,SpecularIntensity,StepSpecular_Intensity,StepSpecular_Exp,StepSpecular_Width,Rim_Intensity,Rim_Exp,Rim_Width,ViewLi
ght_Intensity,ViewLight_Exp,RampOffset2,RimType,StepSpecularType;
    //每层通用参数
    float3 BaseMap,ShadowMap;
    float ShadowMask, RampOffsetMask, SpecularIntensityMask, SpecularExpMask , HalfLambert, NH, NV,NL;
};
```

其次实现出所有特性：

```
float3 NPRLighting(in NPRData nprData)
{
    float stepValue = 0;
    float3 Final =0;

    float3 DarkSide = nprData.ShadowMap * _DarkIntensity;
    float3 BrightSide = nprData.BaseMap * _BrightIntensity;

    //边缘光
    if(nprData.HasRim)
    {
        float RimType = nprData.RimType; //0:两边 1:亮部 2:暗部
        bool BothSideRim = RimType == 0;
        bool OnlyBrightSideRim = RimType == 1;
        bool OnlyDarkSideRim = RimType == 2;

        float3 Rim = pow(step(1-nprData.Rim_Width, 1 - nprData.NV), nprData.Rim_Exp) * nprData.Rim_Intensity*nprData.ShadowMask;

        if(BothSideRim) Final += Rim*nprData.BaseMap;
        if(OnlyBrightSideRim) BrightSide += Rim*nprData.BaseMap;
        if(OnlyDarkSideRim) DarkSide += Rim*nprData.BaseMap;

        // Final += Rim*nprData.BaseMap;
    }

    //代码可优化，但为了直观易读性 省略这一步
    //漫反射
    if (nprData.HasTwoSide)
    {
        if (_LightThreshold < nprData.ShadowMask * (nprData.HalfLambert + _RampOffset +  nprData.RampOffset2 + nprData.RampOffsetMask))
        {
            stepValue = 0.5;
        }
    }

    if (_LightThreshold < nprData.ShadowMask * (nprData.HalfLambert + _RampOffset + 0 + nprData.RampOffsetMask))
    {
        stepValue = 1;
    }

    if (nprData.HasViewLight)
    {
        float3 ViewLight = saturate( pow(saturate(nprData.NV), nprData.ViewLight_Exp) * nprData.ViewLight_Intensity);
        BrightSide *= ViewLight;
        DarkSide *= ViewLight;
    }

    float3 Diffuse = lerp(DarkSide, BrightSide, saturate(stepValue));
    Final+=Diffuse;

    float3 FinalSpecular =0;
    //高光
    if(nprData.HasSpecular)
    {
        float3 Specular = max(0, pow((nprData.NH), nprData.SpecularExp * nprData.SpecularExpMask) * nprData.SpecularIntensity * nprData.S
pecularIntensityMask);
        FinalSpecular += Specular*nprData.BaseMap;
        // Final += Specular;
    }
    //无边高光
    if(nprData.HasStepSpecular)
    {
        float stepSpecularTypeValue = nprData.StepSpecularType ==0 ? nprData.NH:nprData.NV;
        float3 StepSpecular = step(1 - nprData.StepSpecular_Width, pow(stepSpecularTypeValue, nprData.StepSpecular_Exp)) * nprData.StepSp
ecular_Intensity;// * nprData.SpecularIntensityMask;
        FinalSpecular = max(FinalSpecular,  StepSpecular* Diffuse);//两部亮一点 暗部暗一点
        // Final += StepSpecular;
    }
    Final += FinalSpecular;

    return Final;
}
```

每一层可以设置NprData参数，选择所需的特性。

```
//第一层
//0.0  - 0.2    => 基础材质: 一个明暗面 无边缘光 无高光Mask 无边高光 无视角光    (可能包含皮肤 布料 金属)
if(Layer>=0.0 && Layer <=0.2 )
{
    // return  1;
     //每层不一样的参数
    nprData.HasTwoSide              = _Layer1_HasTwoSide;
    nprData.HasRim                  = _Layer1_HasRim;
    nprData.HasSpecular             = _Layer1_HasSpecular;
    nprData.HasStepSpecular         = _Layer1_HasStepSpecular;
    nprData.HasViewLight            = _Layer1_HasViewLight;

    nprData.RampOffset2             = _Layer1_RampOffset2;
    nprData.SpecularExp             = _Layer1_SpecularExp;
    nprData.SpecularIntensity       = _Layer1_SpecularIntensity;
    nprData.StepSpecular_Intensity  = _Layer1_StepSpecular_Intensity;
    nprData.StepSpecular_Exp        = _Layer1_StepSpecular_Exp;
    nprData.StepSpecular_Width      = _Layer1_StepSpecular_Width;
    nprData.Rim_Intensity           = _Layer1_Rim_Intensity;
    nprData.Rim_Exp                 = _Layer1_Rim_Exp;
    nprData.Rim_Width               = _Layer1_Rim_Width;
    nprData.ViewLight_Intensity     = _Layer1_ViewLight_Intensity;
    nprData.ViewLight_Exp           = _Layer1_ViewLight_Exp;
    nprData.RimType                 = _Layer1_RimType;
    nprData.StepSpecularType        = _Layer1_StepSpecularType;

    float3 Light = NPRLighting(nprData);

    return Light.xyzz;
}
//第二层
//0.21 - 0.25   => 布料1  : 两层明暗面 裁边缘光 无高光Mask 无边高光 无视角光
if(Layer>=0.21 && Layer <= 0.25)
{
     //每层不一样的参数
    nprData.HasTwoSide              = _Layer2_HasTwoSide;
    nprData.HasRim                  = _Layer2_HasRim;
    nprData.HasSpecular             = _Layer2_HasSpecular;
    nprData.HasStepSpecular         = _Layer2_HasStepSpecular;
    nprData.HasViewLight            = _Layer2_HasViewLight;

    nprData.RampOffset2             = _Layer2_RampOffset2;
    nprData.SpecularExp             = _Layer2_SpecularExp;
    nprData.SpecularIntensity       = _Layer2_SpecularIntensity;
    nprData.StepSpecular_Intensity  = _Layer2_StepSpecular_Intensity;
    nprData.StepSpecular_Exp        = _Layer2_StepSpecular_Exp;
    nprData.StepSpecular_Width      = _Layer2_StepSpecular_Width;
    nprData.Rim_Intensity           = _Layer2_Rim_Intensity;
    nprData.Rim_Exp                 = _Layer2_Rim_Exp;
    nprData.Rim_Width               = _Layer2_Rim_Width;
    nprData.ViewLight_Intensity     = _Layer2_ViewLight_Intensity;
    nprData.ViewLight_Exp           = _Layer2_ViewLight_Exp;
    nprData.RimType                 = _Layer2_RimType;
    nprData.StepSpecularType        = _Layer2_StepSpecularType;

    float3 Light = NPRLighting(nprData);

    return Light.xyzz;
}
//第N层
```

另外碧蓝幻想的头发、脸部、描边、实现基本与罪恶装备保持一致，不做过多解析。

完整渲染图:

格兰:


知乎 @光阳果

姬塔:


知乎 @光阳果

兰斯洛特:


知乎 @光阳果

原神

原神的贴图模型信息:

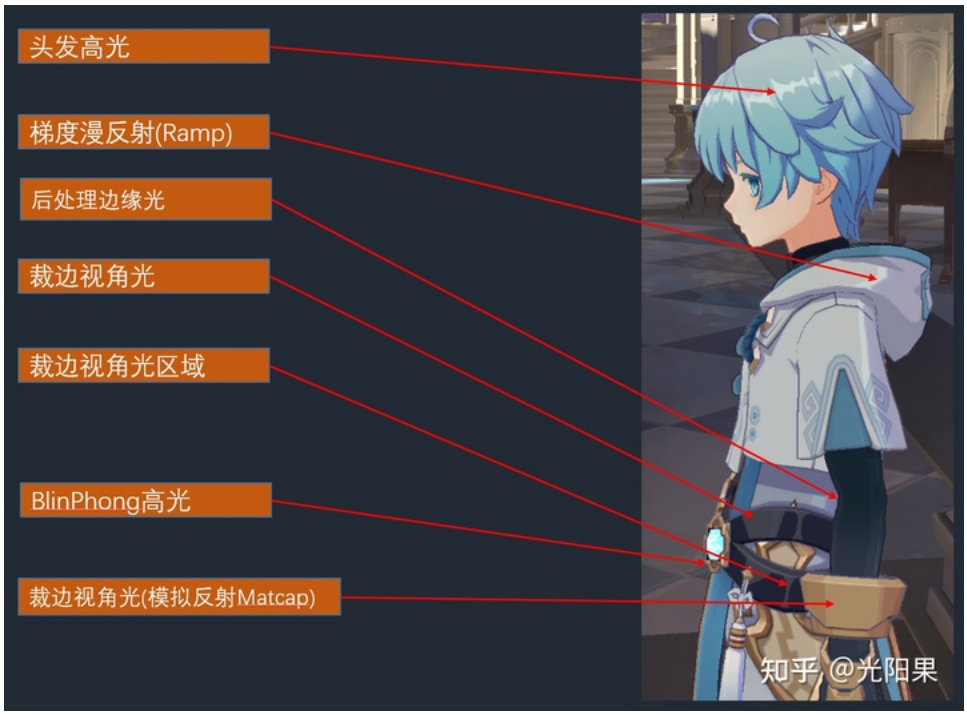
信息含义:

```
LightMap.r    :高光类型Layer,根据值域选择不同的高光类型(eg:BlinPhong 裁边视角光)
LightMap.g    :阴影AO ShadowAOMask
LightMap.b    :BlinPhong高光强度Mask SpecularIntensityMask
LightMap.a    :Ramp类型Layer，根据值域选择不同的Ramp
VertexColor.g :Ramp偏移值,值越大的区域 越容易"感光"(在一个特定的角度，偏移光照明暗)
VertexColor.a :描边粗细
```

渲染特性

原神的基本做法还是沿用碧蓝幻想，新增了 **漫反射分层** 与 **高光分层。**

部分特性:


头发高光
梯度漫反射(Ramp)
后处理边缘光
裁边视角光
裁边视角光区域
BlinPhong高光
裁边视角光(模拟反射Matcap)
知乎 @光阳果

1.漫反射分层

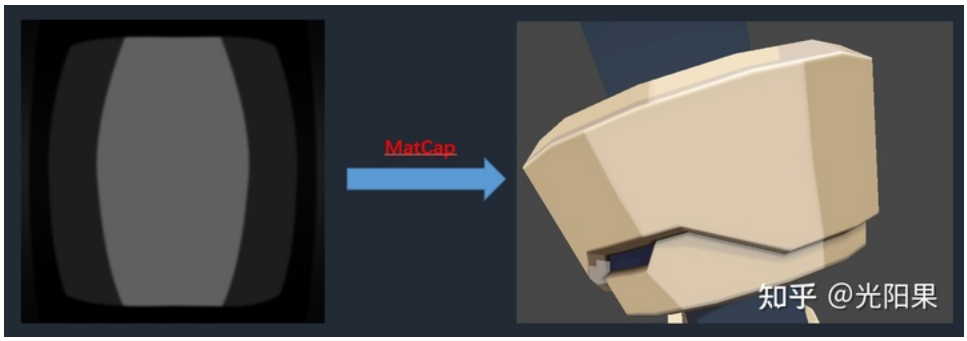
漫反射的DarkSide部分，由Base乘Ramp图得到，BrightSide则为Base。根据LightMap.a通道的不同值域，选择Ramp图中的不同层。Ramp共10层，分上下两部分，对应着夜晚与白天。

```
float SpecularLayerMask         = LightMap.r; // 高光类型Layer
float ShadowAOMask              = LightMap.g; //ShadowAOMask
float SpecularIntensityMask     = LightMap.b; //SpecularIntensityMask
float LayerMask                 = LightMap.a; //LayerMask Ramp类型Layer
// return VertexColor.a;//描边大小
float RampOffsetMask            = VertexColor.g; //Ramp偏移值,值越大的区域 越容易"感光"(在一个特定的角度，偏移光照明暗)
//Ramp图大小为256x20
float RampPixelY = 0.05; // 1.0/20.0;
float RampPixelX = 0.00390625; //1.0/256.0
float halfLambert = (NL * 0.5 + 0.5 + _RampOffset + RampOffsetMask);
halfLambert = clamp(halfLambert, RampPixelX, 1 - RampPixelX);

//头发Shader中,LightMap.A==1 为特殊材质
//根据LightMap.a选择Ramp中不同的层。
float RampIndex = 1;
if (LayerMask >= 0 && LayerMask <= 0.1)
{
    RampIndex = 6;
}

if (LayerMask >= 0.11 && LayerMask <= 0.33)
{
    RampIndex = 2;
}

if (LayerMask >= 0.34 && LayerMask <= 0.55)
{
    RampIndex = 3;
}

if (LayerMask >= 0.56 && LayerMask <= 0.9)
{
    RampIndex = 4;
}

if (LayerMask >= 0.95 && LayerMask <= 1.0)
{
    RampIndex = _RampIndex;
}

//漫反射分类 用于区别Ramp
//高光也分类 用于区别高光

float PixelInRamp = RampPixelY * (RampIndex * 2 - 1);

ShadowAOMask = 1 - smoothstep(saturate(ShadowAOMask), 0.2, 0.6); //平滑ShadowAOMask,减弱锯齿

//为了将ShadowAOMask区域常暗显示
float3 ramp = tex2D(_RampMap, saturate(float2(halfLambert * lerp(0.5, 1.0, ShadowAOMask), PixelInRamp)));
float3 BaseMapShadowed = lerp(BaseMap * ramp, BaseMap, ShadowAOMask);
BaseMapShadowed = lerp(BaseMap, BaseMapShadowed, _ShadowRampLerp);
float IsBrightSide = ShadowAOMask * step(_LightThreshold, halfLambert);
float3 Diffuse = lerp(lerp(BaseMapShadowed, BaseMap * ramp, _RampLerp) * _DarkIntensity,_BrightIntensity * BaseMapShadowed,IsBrightSide *
_RampIntensity * 1) * _CharacterIntensity;
```

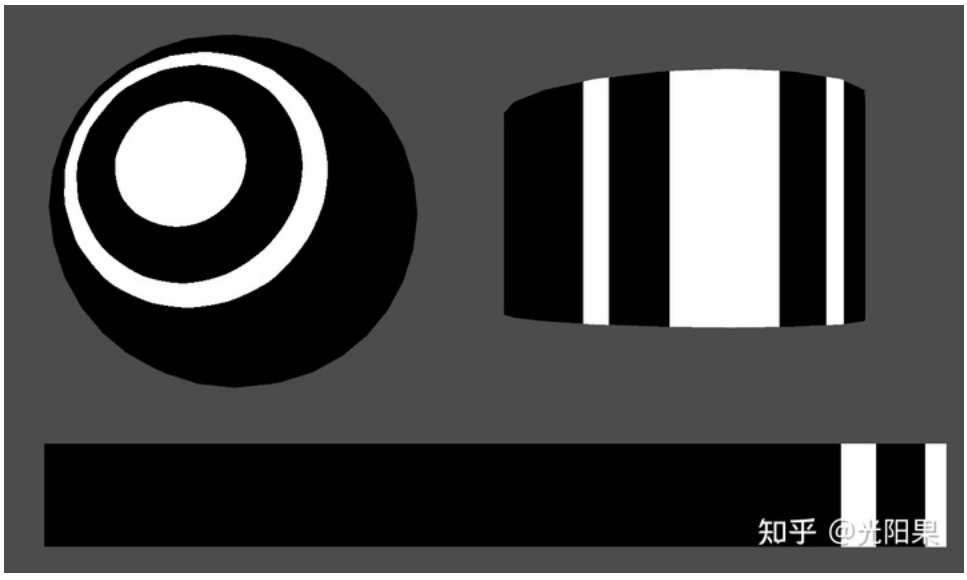
2.高光分层

原神的高光分层，是沿用碧蓝幻想的思路，使用LightMap.r通道，来控只各种高光表现。与前面重复的渲染特性就不再赘述。值得注意的点在于，原神的金属使用了一张MatCap图来做 金属的**裁边视角光**。

```
float MetalMap = saturate(tex2D(_MetalMap, mul((float3x3)UNITY_MATRIX_V, N).xy * 0.5f + 0.5f ).r);
MetalMap = step(_MetalMapV,MetalMap)*_MetalMapIntensity;
```



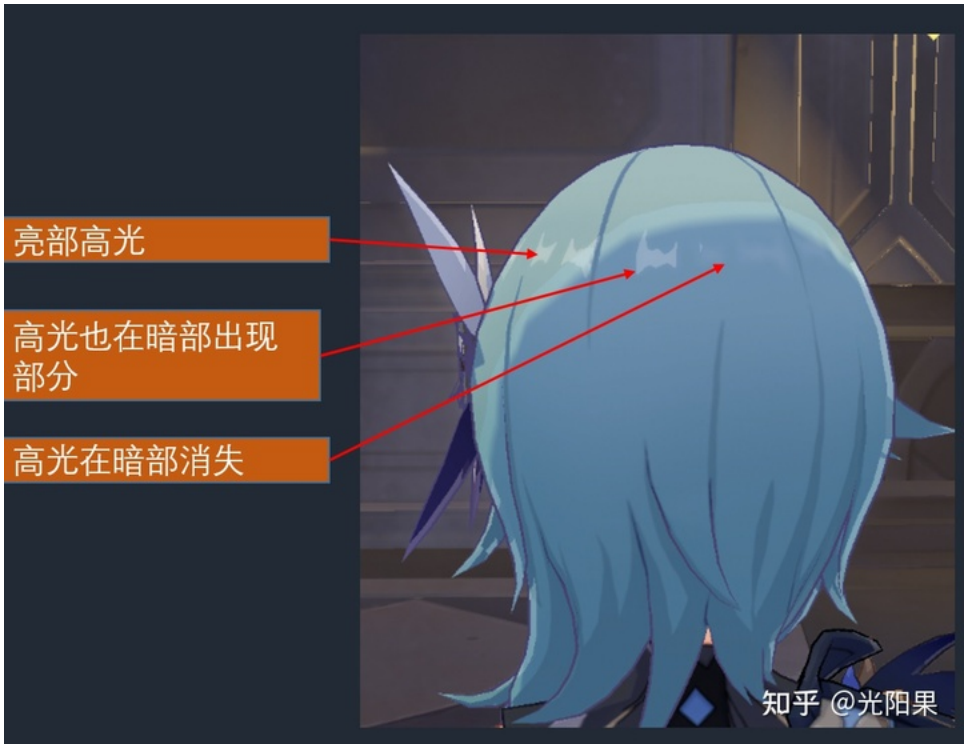
MatCap的本质就是Lut，其作用于Ramp图一致，只不过把NL换成了NV(相机空间下)。这和前面实现的裁边视角光效果一样(对值域进行映射):



```
float3 Specular = 0;
float3 StepSpecular = 0;
float3 StepSpecular2 = 0;

float LinearMask = pow(LightMap.r, 1 / 2.2); //图片格式全部去掉勾选SRGB
float SpecularLayer = LinearMask * 255;

//不同的高光层 LightMap.b 用途不一样
//裁边高光 (高光在暗部消失)
if (SpecularLayer > 100 && SpecularLayer < 150)
{
    StepSpecular = step(1 - _StepSpecularWidth, saturate(dot(N, V))) * 1 *_StepSpecularIntensity;
    StepSpecular *= BaseMap;
    // return Red;
}

//裁边高光 (StepSpecular2常亮 无视明暗部分)
if (SpecularLayer > 150 && SpecularLayer < 250)
{
    float StepSpecularMask = step(200, pow(SpecularIntensityMask, 1 / 2.2) * 255);
    StepSpecular = step(1 - _StepSpecularWidth2, saturate(dot(N, V))) * 1 *_StepSpecularIntensity2;
    StepSpecular2 = step(1 - _StepSpecularWidth3 * 5, saturate(dot(N, V))) *StepSpecularMask * _StepSpecularIntensity3;

    StepSpecular = lerp(StepSpecular, 0, StepSpecularMask);
    StepSpecular2 *= BaseMap;
    StepSpecular *= BaseMap;
}

//BlinPhong高光
if (SpecularLayer >= 250)
{
    Specular = pow(saturate(NH), 1 * _SpecularExp) * SpecularIntensityMask *_SpecularIntensity;
    Specular = max(0, Specular);
    Specular += MetalMap;
    Specular *= BaseMap;
}

Specular = lerp(StepSpecular, Specular, LinearMask);
Specular = lerp(0, Specular, LinearMask);

Specular = lerp(0, Specular, IsBrightSide) + StepSpecular2;

FinalColor.rgb = Diffuse + Specular;
```

3.头发高光

原神头发渲染中，高光在亮部出现，也在暗部出现，暗部的其他部分则消失。

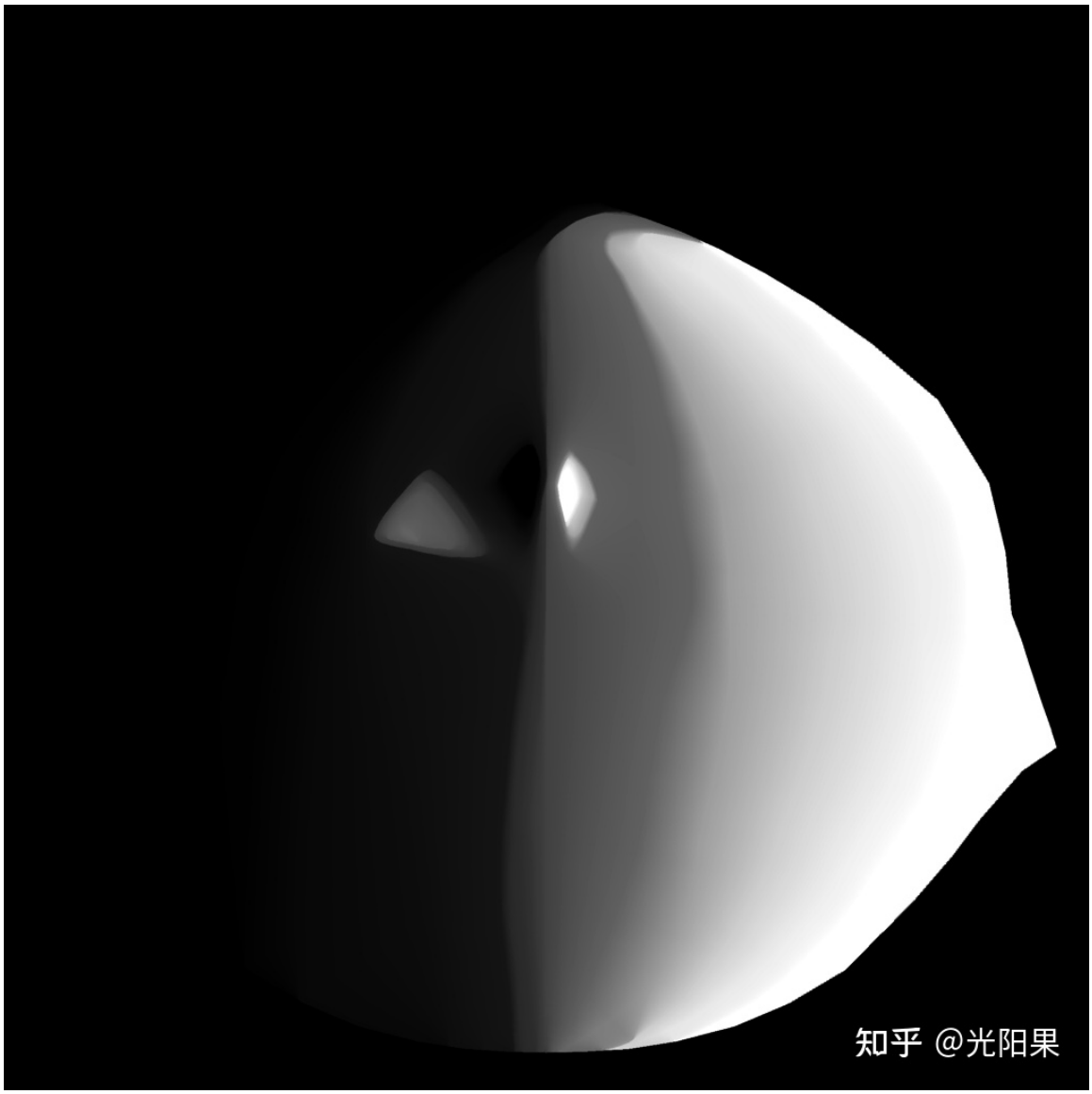


为了达到这种效果，除了用明暗分割来控制高光外，还有一个视角Mask也影响。因此计算高光的时候考虑到这种效果:

```
//头发高光
float SpecularRange = step(1 - _HairSpecularRange, saturate(NH));
float ViewRange = step(1 - _HairSpecularViewRange, saturate(NV));
HairSpecular = SpecularIntensityMask *_HairSpecularIntensity * SpecularRange * ViewRange;
HairSpecular = max(0, HairSpecular);
```

头发的漫反射部分和身体上的一致。

4.脸


知乎 @光阳果

使用SDF图来表现脸部的明暗过渡，相比于调整法线，使用SDF可以更快速地实现效果。

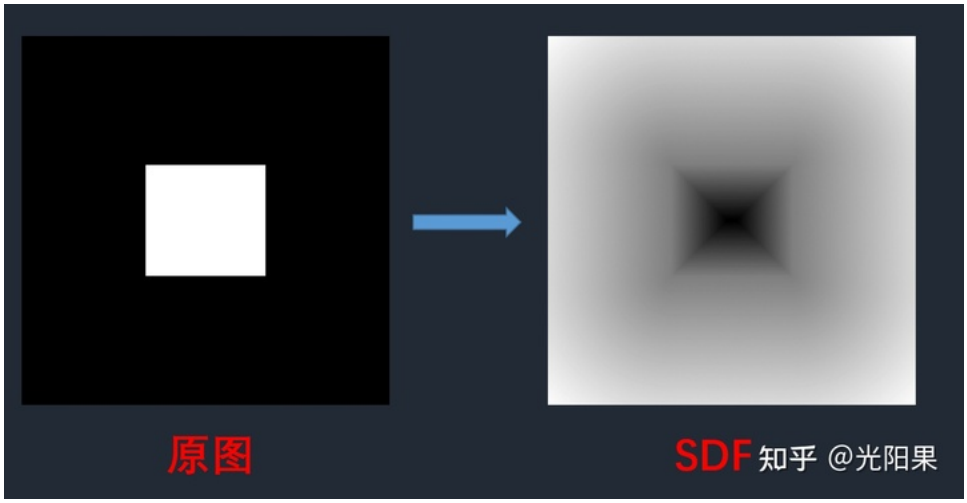
算法：将灯光方向转到局部坐标，求出变换后的XZ极坐标，去step 脸部SDF图做明暗过渡。

```
//将灯光方向转到局部坐标，用XZ求出极坐标，去Step 脸部SDF图做明暗过渡
#define InvHalfPi 0.159154943071114
float4 Left         = mul(unity_ObjectToWorld,float4(0,0,1,0));//左
float4 Up           = mul(unity_ObjectToWorld,float4(-1,0,0,0));//上
float4 Forward      = mul(unity_ObjectToWorld,float4(0,1,0,0));//前

float4x4 XYZ        = float4x4(Left,Up,Forward,float4(0,0,0,1));
float4 Light        = mul(XYZ,float4(-L.xyz,0));
Light.xz            = normalize(Light.xz);
float angle         = atan2(Light.x,Light.z);
float angle01       = angle*InvHalfPi+0.5;
float angle360      = angle01*360.0;

float value         = 0;
if(angle360>=0 && angle360<180)
{
    value = Remap(angle360,0,180,0.01,0.99);
}
else
```

```
{
    value = Remap(angle360,180,360,0.99,0.01);
}

float  NeedFlip    = angle360>180;
float4 FaceSDFMap  = tex2D(_FaceSDFMap, lerp(uv,float2(1-uv.x,uv.y),NeedFlip));
float  FaceLight   = step(value,FaceSDFMap.g)*FaceSDFMask;//FaceSDFMask是脸部需要做SDF过渡部分的Mask
float3 Diffuse     = lerp(_ShadowColor * BaseColor, BaseColor, FaceLight);
```

脸部SDF图的生成

SDF (SignedDistanceFunction)有向距离场记录的是，当前点到目标的距离。可以参考下图：



原图中白色表示内部，黑色表示外部。生成的SDF图，小于0.5的部分代表着 内部到方块边界的距离，大于0.5的部分代表着外部到边界的部分。

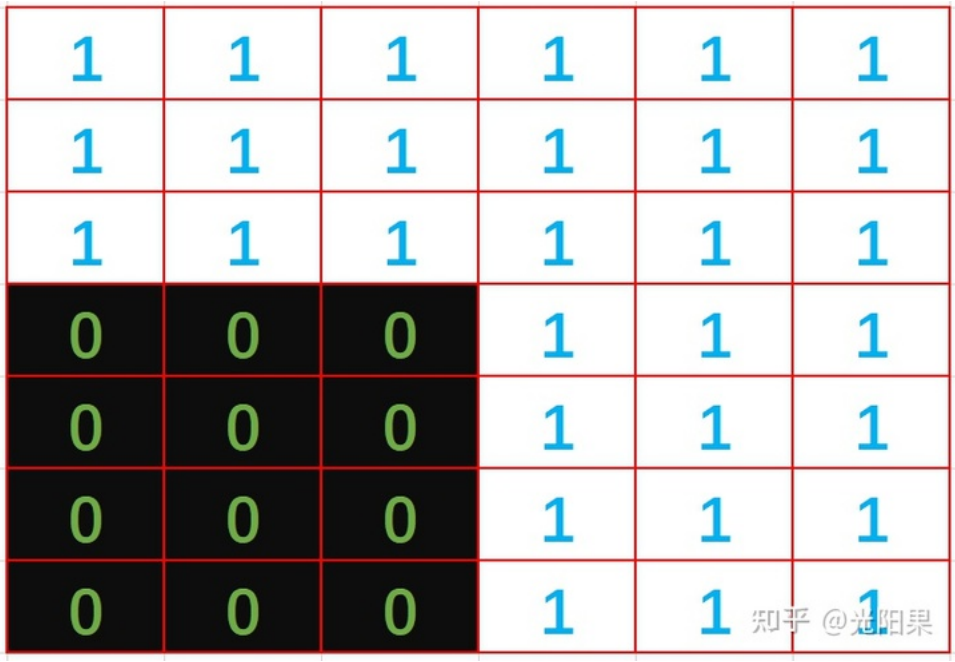
对生成后的SDF做Step操作，可以实现这种过渡效果。

8ssdet SDF生成算法

使用暴力算法，遍历所有像素点，求出最近距离，是最容易的思路，但是对于像素大小超过256x256以后会非常地慢，暴力算法的时间复杂度是O(n*n)。SDF的一种优化算法是8ssdet(8-points Signed Sequential Euclidean Distance Transform)，时间复杂度是O(n)。

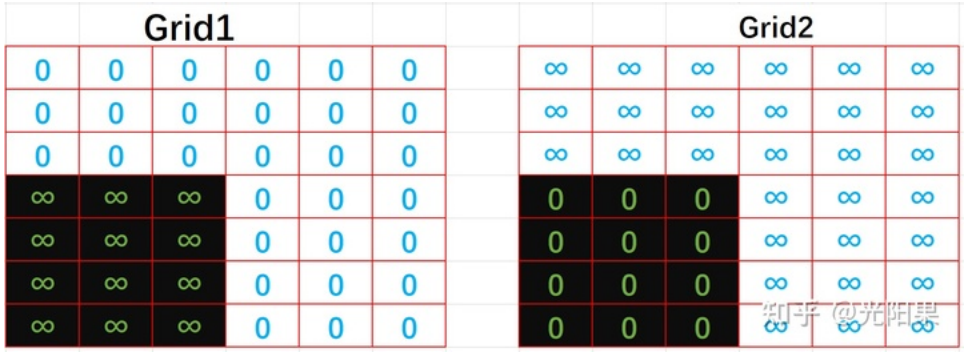
算法:

1.将图片中的像素看做是0与1，0代表黑色，1代表白色。那么一张6x7像素的图片就可以表示如下:



2.定义如下四个算子 A、B、C、D

3.将内部与外部分别定义为Grid1，Grid2。

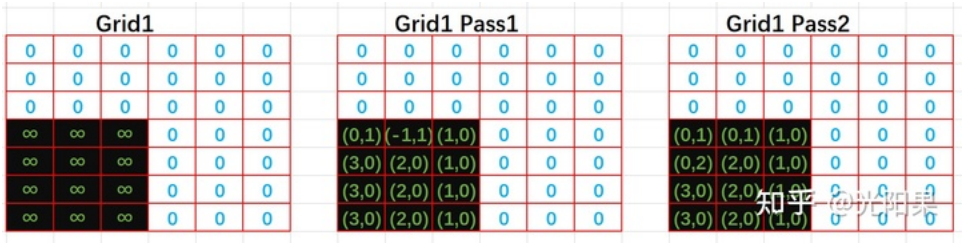


4.Grid1与Grid2分别进行以下计算

```
遍历1:
for(从下到上)
    for(从左到右)
        运用算子A
    for(从右到左)
        运用算子B

遍历2:
for(从上到下)
    for(从右到左)
        运用算子C
    for(从左到右)
        运用算子D
```

4.将最终的结果归一化

Grid1计算的过程(二维向量代表着该点到最近边界点的向量):



脸部SDF

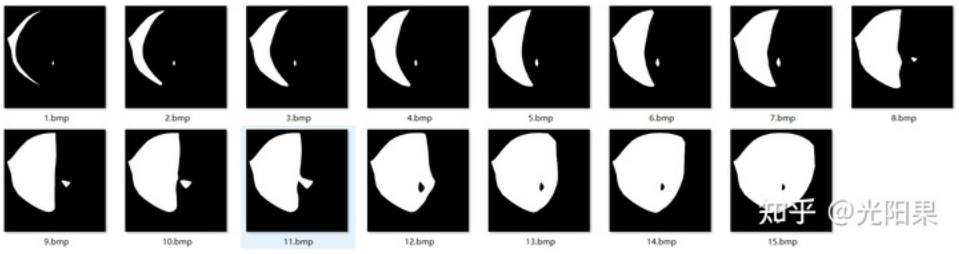
脸部为了做出阴影平滑过渡的效果，需要画多个过渡关键帧，将这些过渡关键帧分别生成SDF在进行融合，即可得到最终的脸部SDF图。

将两张SDF融合算法:

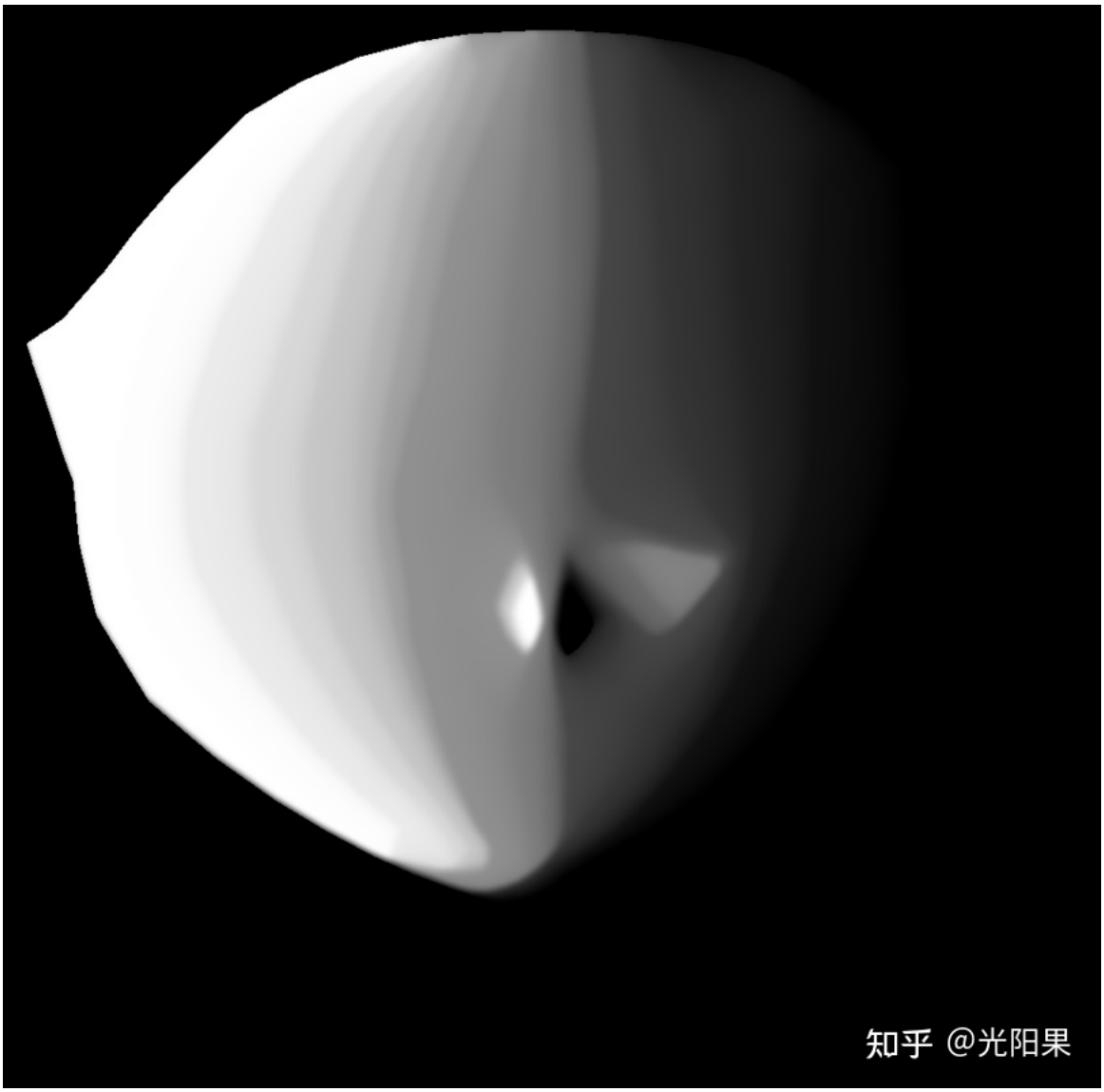
```
private Texture2D SDFBlend(Texture2D sdf1, Texture2D sdf2, int sampletimes)
{
    int WIDTH = sdf1.width;
    int HEIGHT = sdf1.height;
    Color[] pixels = new Color[WIDTH * HEIGHT];
    for (int y = 0; y < HEIGHT; y++)
    {
        for (int x = 0; x < WIDTH; x++)
        {
            var dis1 = sdf1.GetPixel(x, y);
            var dis2 = sdf2.GetPixel(x, y);
            var c = SDFLerp(sampletimes, dis1.r, dis2.r);
            pixels[y * WIDTH + x] = new Color(c, c, c);
        }
    }
    Texture2D outTex = new Texture2D(WIDTH, HEIGHT);
    outTex.SetPixels(pixels);
    //outTex.Apply();
    return outTex;
}

private float SDFLerp(int sampletimes, float dis1, float dis2)
{
    //float SampleTimes = 400;//400次效果比较好
    float res = 0f;
    if (dis1 < 0.5f && dis2 < 0.5f)
        return 1.0f;
    if (dis1 >= 0.5f && dis2 >= 0.5f)
        return 0f;
    for (int i = 0; i < sampletimes; i++)
    {
        float lerpValue = (float)i / sampletimes;
        res += Mathf.Lerp(dis1, dis2, lerpValue) < 0.5f ? 1 : 0;
    }
    return res / sampletimes;
}
```

将所有融合和的SDF，取平均值可以得到最终的脸部SDF图。但是会有锯齿，因此需要进行抗锯齿操作。

15张关键过渡关键帧:



最终融合生成的脸部SDF图:



其他

其他的一些生成SDF图的算法，比如MarchingParabolas(步进抛物线)也是挺有趣的算法，时间复杂度是O(n)。

https://prideout.net/blog/distance_fields/#min-erosion

http://cs.brown.edu/people/pfelzens/papers/dt-final.pdf

抗锯齿

1.增加关键帧

经过测试15帧以上效果为佳。

2.模糊SDF图

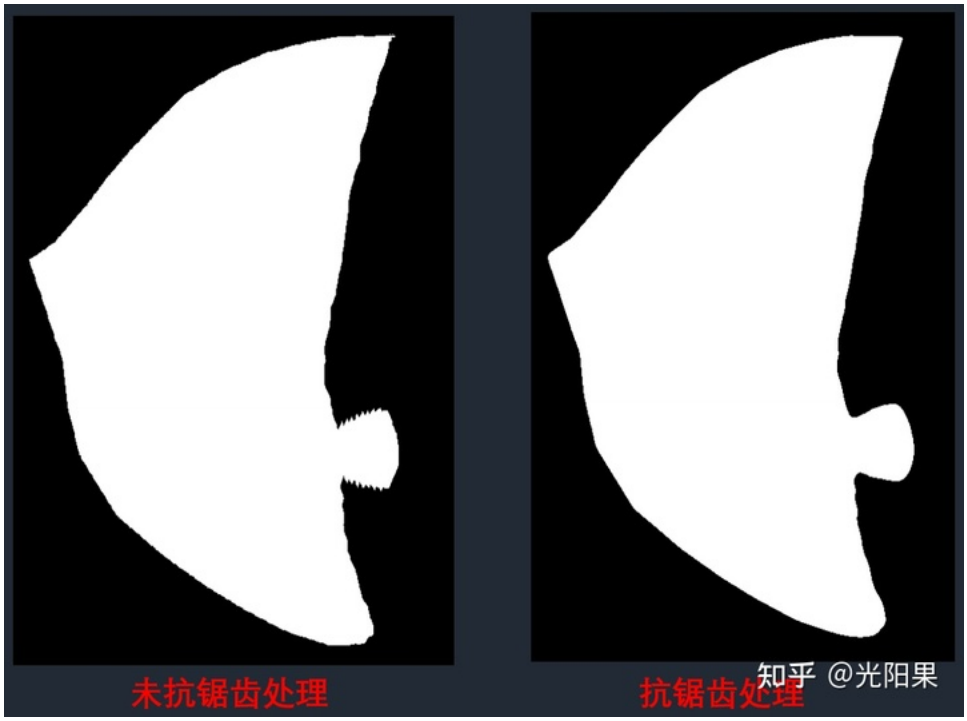
用半径为1、3、5的均值模糊，依次模糊处理SDF图(其他的模糊算法如高斯模糊、场景模糊，经测试效果不佳)。使用模糊算法后会导致SDF图范围扩大，可以使用Mask将模糊后的SDF图范围框住。

3.Shader中SDF抗锯齿

```
float GetFaceSDF(in float2 uv, in float Threshold)
{
```

```
    ///抗锯齿
    ///https://steamcdn-a.akamaihd.net/apps/valve/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf
    float dist = (tex2D(_FaceSDFTex, (uv)).r);
    float color = dist;
    // uv distance per pixel density for texture on screen
    float2 duv = fwidth(uv);
    // texel-per-pixel density for texture on screen (scalar)
    // nb: in unity, z and w of TexelSize are the texture dimensions
    float dtex = length(duv * _FaceSDFTex_TexelSize.zw);
    // distance to edge in pixels (scalar)
    float pixelDist = (Threshold - color) / _FaceSDFTex_TexelSize.x * 2 / dtex;
    return  step(pixelDist, 0.5);
}
```

以下是是否抗锯齿的对比：



在生成SDF图的时候进行抗锯齿处理 配合Shader中抗锯齿处理，最终可以得到一个比较理想的没有锯齿的效果。

5.后处理边缘光

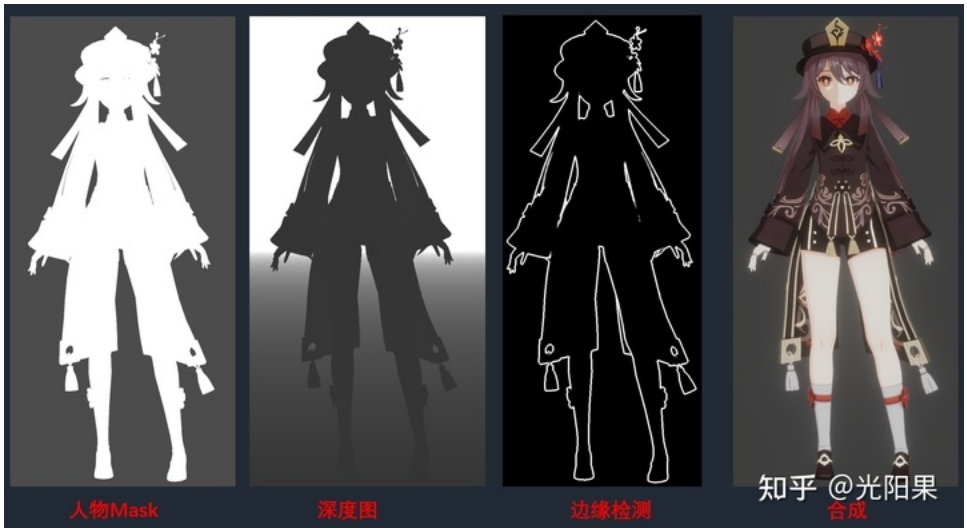
后处理边缘光分两部分

1.对深度图进行检测，某一个像素与周围深度差值大于一定阀值的则判定为是边缘。

2.这个边缘只作用在人物身上，因此需要单独渲染一张人物Mask来控制。

```
float pixel = tex2D(NPRMaskTex, i.uv).r;//NPRMaskTex为单独渲染的人物Mask
//左 右 上 下
float2 Op1[] = {float2(-1, 0), float2(1, 0), float2(0, 1), float2(0, -1)};
float4 infoRim = Black;
if (pixel > 0.01)
{
    float far = _ProjectionParams.z;
    //边缘检测
    float depthPixel = far* Linear01Depth(UNITY_SAMPLE_DEPTH(tex2Dproj(_CameraDepthTexture, i.screenPos )));
    for (int k = 0; k < 4; k++)//只检测左与右效果也差不多
    {
        float depth = far*Linear01Depth(UNITY_SAMPLE_DEPTH(tex2Dproj(_CameraDepthTexture, i.screenPos + float4(RimPixelOffset*Op1[k]* _Ca
meraDepthTexture_TexelSize.xy,0,0))));
        if (abs(depthPixel - depth) > DepthThreshold)
        {
            infoRim = RimColor;
            break;
        }
    }
}
FinalColor = MainTex + infoRim;
```

检测了周围的四个像素来判读深度插值，经过测试只检测左与右两个像素 效果也差不多；使用一个参数RimPixelOffset来控制边缘检测的像素差值，最终表现为边缘光的大小。



渲染图

胡桃:


知乎 @光阳果

刻晴:


知乎 @光阳果

重云:

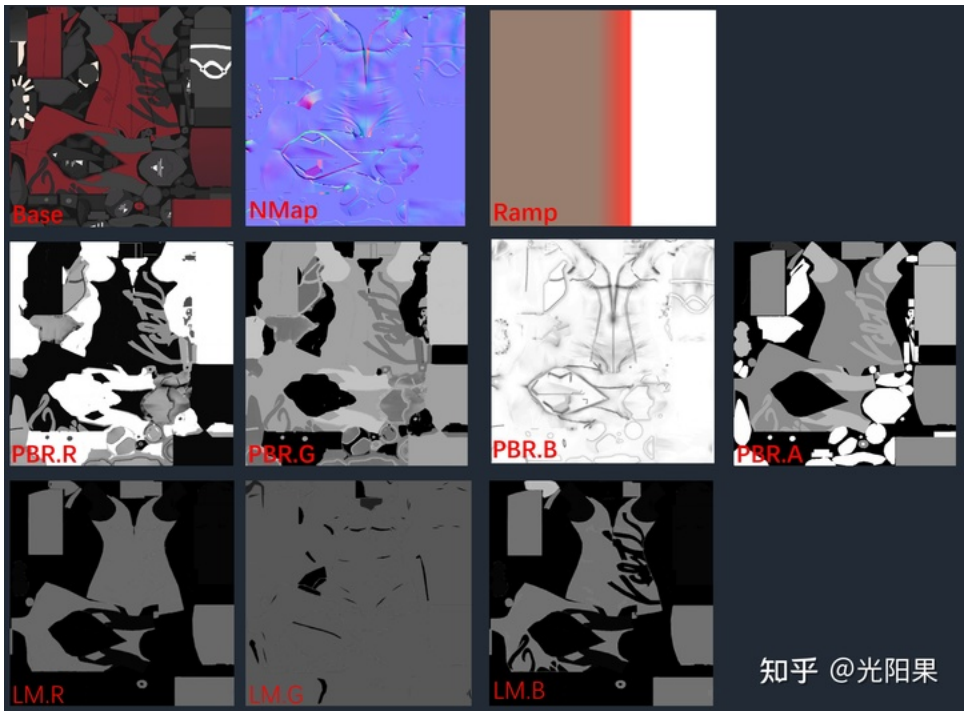

知乎 @光阳果

人是一根带着生殖器乱跑的芦苇！

战双帕弥什

战双的卡渲基本沿用了罪恶装备，这个基础上引入PBR的光照表现，要做PBR效果必然会使用到Roughness、Metallic、AO等贴图用来控制更细腻的光影表现。既然要混合PBR与卡通渲染，就必然会多出一张PBRMask来控制哪些地方显示PBR。

战双的模型贴图信息：



说明:

1.PBRMixMap

PBRMixMap.r Metallic 金属度 **PBRMixMap.g** Smoothness 光滑度(1-粗糙度) **PBRMixMap.b** AO 高光遮罩 **PBRMixMap.a** PBRMask PBR高光类型Mask

2.LightMap

LightMap.r RampOffset Ramp偏移值 **LightMap.g** ShadowAO 常暗区域Mask **LightMap.b** SpecularMask 高光类型Mask(决定是否是PBR)

其他

Ramp 模拟皮肤SSS

法线 结合NormalMap做PBR光影表现

切线 是平滑调整后的法线

顶点颜色 控制表变粗细

脸部 与罪恶装备一致，调整法线来做平滑的明暗过渡

头发 明暗高光

战双并不像原神或者碧蓝幻想那样有着统一的渲染特性，而是每个角色几乎都有自己的渲染特性，这在角色设计上会给艺术家更多的发挥空间，只要画风统一就行了！但由于渲染特性的不统一，导致每个角色基本都需要一个单独的Shader来表现，会导致多人协作不方便。

战双的渲染代码做法参考前文基本都能实现，由于特性过多只实现了部分。

部分代码:

```
//漫反射
float3 Diffuse = lerp( BaseColor*_DarkIntensity,BaseColor*_BrightIntensity,step( (_RampOffset + RampAdd),NL01) *ShadowAO);

//头发高光
float3 SpecularHair = BaseColor*lerp(_HairDarkIntensity,_HairBrightIntensity,IsBright)*SpecularMask;

//高光 有很多高光类型,这里只实现了两种，GGX 与 BlinPhong高光形变
float3 Specular = Specular_GGX(N,L,H,V,Roughness,F0) * AO*_SpecularIntensity * GGXMask *MatMask;

#ifdef _SPECSHIFT_GGX
StylizedSpecularParam param;
param.BaseColor = BaseColor;
param.Normal = N;
param.Shininess = _Shininess;
param.Gloss = _Gloss;
param.Threshold = _Threshold;
param.dv = T;
param.du = B;
Specular +=  StylizedSpecularLight_GGX(param,V,L,H,Roughness,Metallic)*_SpecShiftIntensity* GGXMask*MatMask;
#endif

#ifdef _SPECSHIFT_BLINPHONG
StylizedSpecularParam param;
param.BaseColor = BaseColor;
param.Normal = N;
param.Shininess = _Shininess;
param.Gloss = _Gloss;
param.Threshold = _Threshold;
param.dv = T;
param.du = B;
Specular +=  StylizedSpecularLight_BlinPhong( param, H)*_SpecShiftIntensity* GGXMask*MatMask;
#endif
```

渲染图

露西亚


知乎 @光阳果

卡列尼娜:


知乎 @光阳果

总结

日式卡通渲染漫反射公式：

```
Diffuse = lerp(DarkSide,BrightSide, step(_LightThreshold,(NL01+RampOffset)*ShadowAO ) );
```

源代码

文章中所有代码与高清渲染图连接:

https://github.com/ipud2/Unity-Basic-Shader/tree/master/%E6%97%A5%E5%BC%8F%E5%8D%A1%E9%80%9A%E6%B8%B2%E6%9F%93%E7%AC%94%E8%AE%B

最后欢迎大家加群交流学习：QQ群：688075253