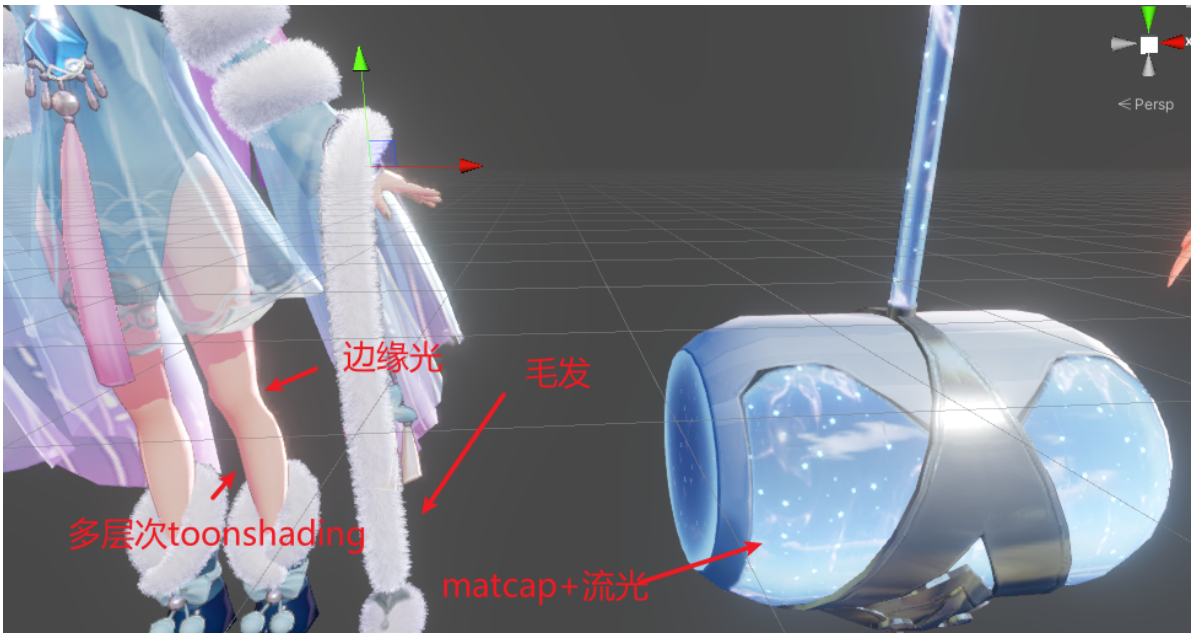
Character

背景

据美术说，我们角色的设计是武侠风带一部分的二次元风格。因此角色上的渲染基于toonshading，使用了GGX高光(不包含IBL环境反射)和边缘光，部分角色材质使用matcap模拟。且增添一些额外效果，如染色、流光。

效果展示



参数说明

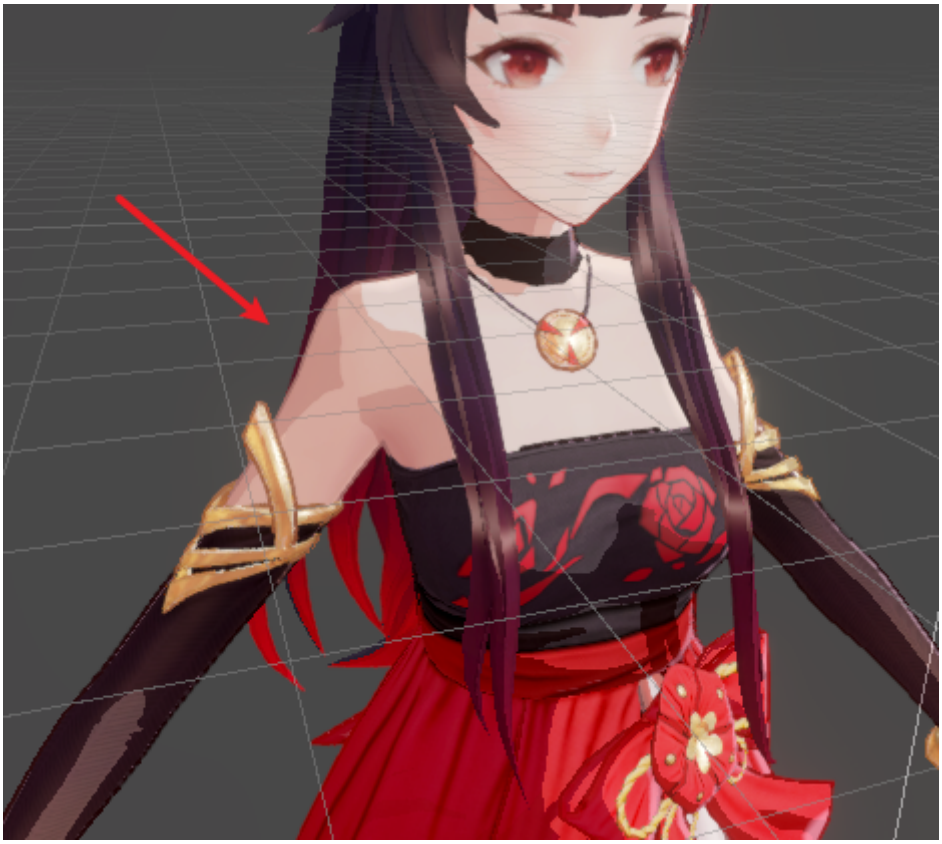
1. Basics
    1. 渲染类型 不透明或透明
    2. 背面消隐 单面模型双面显示选择off
    3. 叠加色
    4. 固有色 亮部颜色
    5. 暗面叠加色 暗面颜色
    6. 暗部使用贴图 使用5中的贴图或直接乘上Color值
2. Shading 光照相关
    1. 明暗线位置 第一层色阶边界
    2. 羽化 过渡柔和程度控制
    3. 暗部颜色 暗部乘上该颜色
    4. 第二层暗部 开关
    5. 明暗线位置2
    6. 羽化2
    7. 暗部颜色2
    8. 皮肤光照开关 皮肤光照使用dotNV代替dotNL，使皮肤光照从中间往两侧边缘柔和过渡
3. Specular 高光
    1. 启用高光 开关
    2. 强度
4. normals 法线

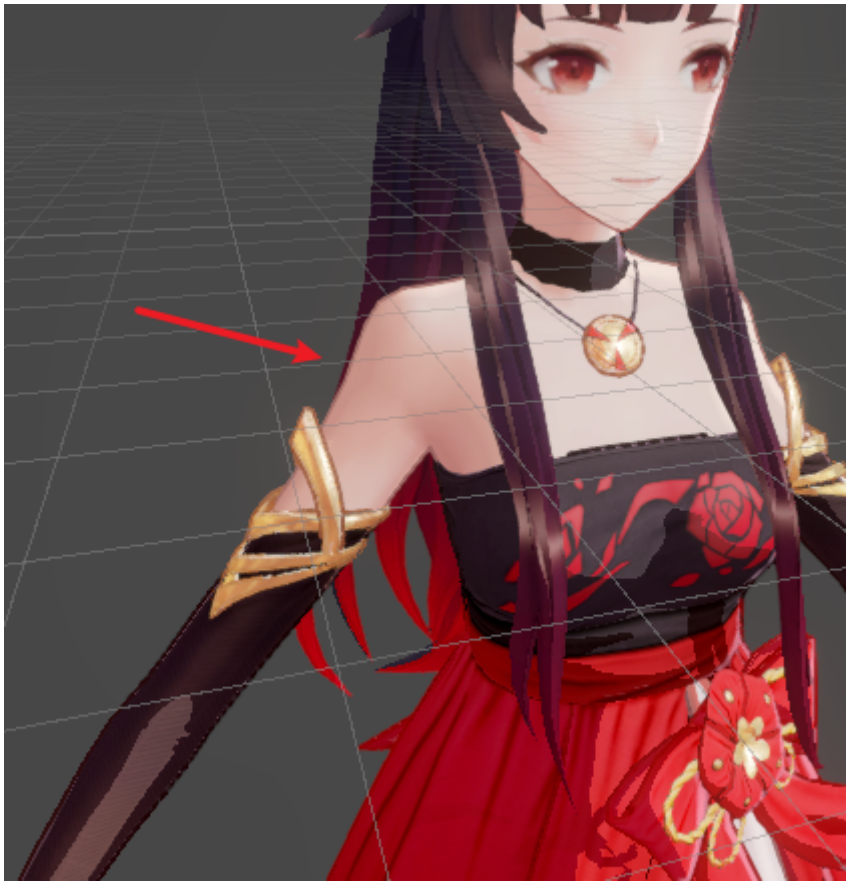
    1. 法线开关
    2. 法线贴图 (rg保存法线，ba为两层matcap遮罩)
    3. 强度
5. ILM Mixed Map 混合贴图
    1. ILM贴图 (R 粗糙度 G ao B 金属度 a 自发光)
    2. 金属度系数
    3. 粗糙度系数
    4. ao系数
6. Emission 自发光
    1. 启用自发光 开关
    2. 自发光颜色
    3. 强度
7. RimLight 边缘光
    1. 启用边缘光 开关
    2. 边缘光宽度
    3. 强度
    4. 边缘光向暗面沿伸
    5. 边缘光颜色
    6. 叠加固有色 固有色混合比例
    7. 卡通化边缘光 开关
    8. 阈值 色阶边界
    9. 羽化 边界过渡柔和程度
8. Outline
    1. 启用描边开关
    2. 描边叠加色
    3. 叠加固有色开关
    4. 描边宽度
    5. 描边z偏移 解决复杂多边形描边遮蔽物体的系数
    6. 描边消失距离 (已废弃)
9. Matcap
    1. 启用matcap
    2. 效果强度
    3. 叠加固有色
    4. 粗糙地方减弱
    5. 纹理叠加色
    6. 纹理强度
    7. matcap纹理是第几张 从4x4的matcap合集中选择当前matcap (0~15)
    8. matcap1区模式 加、混合、乘
    9. matcap 第二层相关，同上
10. 染色、流光遮罩（rgb染色、a流光）
11. 染色
    1. 染色模式
        1. 关闭
        2. 开启
        3. 过渡 两种染色过渡
    2. 染色目标
        1. 染色R (遮罩r通道控制的染色区域)
            1. 色相
            2. 饱和度

        3. 明度
    2. 染色G
    3. 染色B
3. 染色源 染色过渡模式才有效，控制从源染色到目标染色
    1. 染色R (遮罩r通道控制的染色区域)
        1. 色相
        2. 饱和度
        3. 明度
    2. 染色G
    3. 染色B
4. 染色过渡相关
    1. 染色渐变竖直偏移
    2. 1/染色渐变高度 染色渐变高度的倒数，简化shader计算
    3. 染色渐变百分比
    4. 染色渐变间隔
    5. 染色渐变火焰水平缩放
    6. 染色渐变内层颜色
    7. 染色渐变外层颜色
    8. 渐变噪声图
5. flow color 流光 两层流光 一层闪光
    1. 启用流光
    2. 流光贴图
    3. 流光颜色1
    4. 流光颜色2
    5. 流光参数 xy uv方向 y tile w 亮度
    6. 闪光颜色
    7. 闪光尺寸
    8. 闪光速度

技术介绍

1. toonshading：
    1. 卡通风格的角色往往不是柔和过渡而是只有几个色阶，所以我们要把lambert漫反射重新划分成几个色阶(多层次暗部)。色阶间可以有一些简单过渡。





2. 可参考[Unity-Chan Toon Shader 2.0 (UTS2)]

3. 关键代码：

```
// 将lambert漫反射强度依据明暗部位颜色、分界线、羽化程度Remap成卡通化的光照，
half3 CalcToonColor(float3 albedo, float3 shadow, half lambert,half
step, half feather)
{
    float3 toonColor = Remap(
```

```
        saturate(step - feather/2.0),
        saturate(step + feather/2.0),
        shadow,
        albedo,
        lambert);
    return toonColor;
}

// 使用多层次暗部使皮肤过渡更柔和，衣物就不必有太柔和的过渡
inline half3 DiffuseShadow(float2 uv,half3 worldNormal, half3 worldView,
half3 worldLight,half3 albedo, half lightAtten)
{
    // 直接光部分
    float NdL = saturate(dot(worldNormal, worldLight))*0.5+0.5 ;
    // 美术希望视角方向的光影过渡，所以这里使用dotNV
    #ifdef _SKIN_ON
        NdL = saturate(dot(worldNormal,worldView));
    #endif
    NdL = NdL * lightAtten;
    // NdL = ao > 0.5 ? lerp(NdL, 1, Remap(0.5, 1, 0, 1, ao)) : ao < 0.5
? lerp(0, NdL, Remap(0, 0.5, 0, 1, ao)) : NdL;
    // 使用暗部贴图或固有色
    #ifdef _SHADOWTEX_ON
    half3 shadow = albedo * tex2D(_ShadowTex, uv).rgb;
    #else
    half3 shadow = albedo * albedo;
    #endif
    half3 shadow1 = _ShadowColor1 * shadow;
    // 第二层暗部
    #ifdef _SHADOW2_ON
    half3 shadow2 = shadow * _ShadowColor2;
    shadow1 = CalcToonColor(shadow1,shadow2, NdL,_ToonStep2,
_ToonFeather2);
    #endif
    half3 toonColor = CalcToonColor(albedo,shadow1, NdL,_ToonStep ,
_ToonFeather );
    half3 color = toonColor * _LightColor0;

    return color;
}
```

2. 描边
    1. 我们使用**基于几何生成方法的描边**。我们选择在模型空间外扩，这样能近大远小。如果在ndc空间下外扩会使法线宽度保持不变。
    2. 原本尝试平滑法线到顶点色，解决硬边描边断裂问题。因为种种原因作罢，这里给个链接。[Unity 环境下平滑法线（软化边）方案整理]
    3. 关键代码

```
v2f_outline vert_outline (appdata_outline v)
{
    v2f_outline o;
    UNITY_INITIALIZE_OUTPUT(v2f_outline, o);
    #if _OL_ON
        o.pos = UnityObjectToClipPos(v.vertex);
```

```
        o.color = v.color;
        // 在模型空间normalize保证外扩距离固定，如果需要描边宽度固定，要在ndc中
normalize
        // 模型空间到视线空间
        float3 expend = mul((float3x3)UNITY_MATRIX_IT_MV,
normalize(v.normal));
        float3 ndcNormal = (TransformViewToProjection(expend.xyz)); //将
法线变换到clip空间
        // 模型外扩(固定距离)
        o.pos.xy += ndcNormal.xy * _Outline * .04 * v.color.b ;
        // o.pos.z *= v.color.r;
        // Z偏移解决部分复杂形状，描边挡住物体
        // 不同平台下，z偏移方向比例不同
        #if UNITY_REVERSED_Z
            o.pos.z += 0.5 * _OutlineZBias * o.pos.w / 1000;
        #else
            o.pos.z -= _OutlineZBias * o.pos.w / 1000;
        #endif
        o.uv = TRANSFORM_TEX(v.uv, _MainTex);
        o.normal = v.normal;
    #else
        o.pos = half4(0, 0, 0, 1);
    #endif

    return o;
}
```

3. GGX高光:
    1. PBR光照中的一部分，可参考LearnOpenGL CN 高级光照
    2. 源代码直接拷贝的Unity BRDF Specular
4. 各向异性高光
    1. 头发各向异性高光
        1. 光照模型Kajiya-kay Model
        2. 关键代码

```
half3 KKSpecular(float3 normal, float3 tangent, float3 view, float3
light)
{
float3 h = normalize(view + light);
float NdL = saturate((dot(normal, light)) * .5 + .5);
// 在光照方程中使用发丝切线T替代法线N
// 切线T和半角向量H之间夹角为θ,切线T和法线N垂直
// 反射光H = normalize(入射光方向L+视角方向V)
// cosθ = T·H
// cosθ^2 + sinθ^2 = 1
// 具体请看 Kajiya-kay Model
float TdH = dot(normalize(tangent + h * _AnisoSpecularPosition), h);
float sinTH2 = 1.0 - TdH * TdH;
// 两层各向异性高光模拟
float3 specular1 = pow(sinTH2, (_AnisoGlossiness + .05) * 250) *
_AnisoSpecColor.rgb;
float3 specular2 = pow(sinTH2, (_AnisoGlossiness *
_AnisoBaseGlossiness + .05) * 100) * _AnisoBaseSpecColor.rgb;
```

```
#if _TOON_ANISO
specular1 *= step(_ToonAnisoStep, Lumin(specular1));
specular2 *= step(_ToonAnisoStep, Lumin(specular2));
float3 specular = specular1 * _AnisoSpecPower + specular2 *
_AnisoSpecBasePower;
#else
float3 specular = specular1 * _AnisoSpecPower + specular2 *
_AnisoSpecBasePower;
#endif
return specular * _LightColor0.rgb * NdL;
}
```

2. 丝绸各向异性高光:
    1. 请参考丝绸效果的实现
    2. 原本打算使用ggx各向异性，后来觉得没必要那么复杂，直接按头发的各向异性方式实现

```
inline half3 SilkSpecular(float3 normal, float3 tangent, float3
view, float3 light,float3 albedo,float3 metallic, float smoothness )
{
// return metallic;
half3 specColor = lerp(unity_ColorSpaceDielectricSpec.rgb, albedo,
metallic);
// Kajiya-kay Model
float3 h = normalize(view + light);
float NdL = saturate((dot(normal, light)) * .5 + .5);
float TdH = dot(normalize(tangent + h * _AnisoSpecularPosition), h);
float sinTH2 = 1.0 - TdH * TdH;
float3 specular = pow(sinTH2, (smoothness + .05) * 250) * specColor
* _AnisoSpecColor.rgb;
return specular * _LightColor0.rgb * NdL;
}
```

    3. 使用flowmap描述各向异性方向

```
// flowmap转切线
inline half3 GetTangentFlow(in float2 uv, in float3 worldNormal, in
float3x3 worldTBN,out fixed specMask){
half3 tangent = tex2D(_TangentFlow, uv).rgb;
specMask = tangent.b;
tangent.xy = tangent.yx*2 - 1;
tangent.y = -tangent.y;
tangent = normalize(half3(tangent.xy,0));
tangent = mul(tangent,(float3x3)worldTBN);
tangent = tangent - worldNormal*dot(tangent, worldNormal);
return tangent;
}
```

5. 边缘光:
    1. 依照视线法线夹角，在物体边缘处发光
    2. 关键代码介绍:

```
half3 Rim(half mask, half3 albedo, half ao, half metallic, half3 normal,
half3 view, half3 light)
{
    // 视线与法线夹角
    half NdV = saturate(dot(normal, view));
    // 如果希望边缘光只在方向光下看到，计算NdL加强光照下的边缘光，减弱非光照下的边缘光
    #if _RIM_ON_LIGHT_DIRECTION
    half NdL = saturate((dot(normal, light) + _RimPermeation)/(1 +
_RimPermeation));
    #endif
    //
    float rim =
        #if _RIM_ON_LIGHT_DIRECTION
        NdL *
        #endif
        smoothstep(1 - _RimWidth, 1, 1 - NdV);
    rim = lerp(rim, .04, metallic);
    rim *= ao;//lerp(1, ao, _AOScale);
    // 卡通化边缘光
    #if _TOON_RIM
    rim = ToonStyleRim(rim);
    #endif
    half3 col = mask * rim * _RimPower * _RimColor;
    // 边缘光与漫反射混合
    #if _RIM_COLOR_DIFFUSE
    col = lerp(col, col * albedo, _RimDiffuseBlend);
    #endif
    // 边缘光x光照颜色
    #if _RIM_COLOR_LIGHT
    col *= _LightColor0;
    #endif
    return col;
}
```

6. matcap:
    1. **材质捕捉**（material capture）简称 MatCap，材质通过渲染一个球到纹理而被捕捉。我们可以将捕获下来的光照和材质效果的贴图，依据视线、法线，重新映射到指定模型上
    2. 简版原理

```
// vertex shader
uniform mat4 modelView;
uniform mat4 modelViewProjection;

in vec3 position;
in vec3 normal;

out vec3 _normal;

void main()
{
    gl_Position = modelViewProjection * vec4(position, 1.0);
    // 计算视空间法线
    _normal = transpose(inverse(mat3(modelView))) * normal;
}
```

```
// fragment shader
uniform sampler2D textureMatcap;

in vec3 _normal;

out vec4 fragmentColor;

void main()
{
    vec2 texcoord = _normal.xy * 0.5 + 0.5;
    // 采用matcap贴图
    vec3 color = texture(textureMatcap, texcoord).rgb;
    fragmentColor = vec4(color, 1.0);
}
```

3. 参考：[matcap 的制作与渲染]

4. 关键代码

```
v2f_main vert_main (appdata_main v)
{
    // ...
    #if _MATCAP_ON
        // 准备辅助参数，用于像素阶段计算视空间法线
        TANGENT_SPACE_ROTATION;
        o.TtoV0 = normalize(mul(rotation, UNITY_MATRIX_IT_MV[0].xyz));
        o.TtoV1 = normalize(mul(rotation, UNITY_MATRIX_IT_MV[1].xyz));
        #if _MATCAP_REFLECT_ON
            o.tangentView = normalize(mul(rotation,
ObjSpaceViewDir(v.vertex)));
        #endif

        // 4x4贴图选择
        half2 index = half2(_Matcap1Index, _Matcap2Index);
        half onePixel = _MatcapTex1_TexelSize.x;
        half4 offset = 0;
        offset.xz = (index % 4) ;
        offset.yw = (3 - floor(index / 4));
        offset = offset * 0.25 + onePixel;
        o.matcapOffset = offset;
    #endif
    // ...
}

fixed4 frag_main (v2f_main i,fixed facing : VFACE) : SV_Target
{
    // ...
    #if _MATCAP_ON
        // 使用view沿normal的反射作为法线。可以解决纯平面的matcap颜色一样的问题。
        #if _MATCAP_REFLECT_ON
            i.tangentView = normalize(i.tangentView);
            tangentNormal = reflect(-i.tangentView, tangentNormal);
        #endif
        // 切线空间法线转视线空间法线
```

```
        float2 viewNormal = ViewSpaceNormal(tangentNormal, i.TtoV0,
i.TtoV1);
        half3 matcap = Matcap(col, i.uv.xy, viewNormal,
roughness,i.matcapOffset);
        #if _DEBUG_MATCAP1 || _DEBUG_MATCAP2
            return half4(matcap, 1);
        #endif
    #endif
    // ...
}
half3 Matcap(half3 albedo, float2 maskUV, float2 viewSpaceNormal, half
roughness, half4 offset)
{
    // matcap 从4x4贴图选择matcap，我们最多有两张matcap
    half onePixel = _MatcapTex1_TexelSize.x;
    // matcap 遮罩
    fixed2 matcapMask = tex2D(_NormalTex, maskUV).ba;
    fixed mask = matcapMask.r;
    half scale = saturate(mask * _Matcap1Scale);
    // matcap1 依据视线空间法线与贴图选择偏移
    half3 matcap = tex2D(_MatcapTex1, viewSpaceNormal * (0.25 - 2 *
onePixel) + offset.xy);
    // matcap1 叠加固有色 然后使用add或blend或multiply作用固有色上
    #if _MATCAP1_COLOR_DIFFUSE
    matcap *= albedo * _MatcapColor1 * _Matcap1Power;
    #else
    matcap *= _MatcapColor1 * _Matcap1Power;
    #endif
    half3 matcap1 =
        #if _BLEND1_ADD
        lerp(albedo, albedo + matcap, scale);
    #elif _BLEND1_BLEND
    lerp(albedo, matcap, scale);
    #elif _BLEND1_MULTIPLY
    lerp(albedo, matcap * albedo, scale);
    #endif
    // 依据粗糙度插值，越光滑越接近matcap，粗糙接近固有色
    #if _MATCAP1_ROUGHNESS
    matcap1 = lerp(matcap1, albedo, roughness);
    #endif
    matcap1 -= albedo;
    #if _DEBUG_MATCAP1
    return matcap1;
    #endif
    // matcap2 计算同上
    #if _MATCAP2_ON
    mask = matcapMask.g;
    scale = saturate(mask * _Matcap2Scale);
    matcap = tex2D(_MatcapTex1, (viewSpaceNormal) * (0.25 - 2 *
onePixel) + offset.zw);
    // return matcap;
    #if _MATCAP2_COLOR_DIFFUSE
    matcap *= albedo * _MatcapColor2 * _Matcap2Power;
    #else
    matcap *= _MatcapColor2 * _Matcap2Power;
    #endif
    half3 matcap2 = mask <= 0 ? albedo :
    #if _BLEND2_ADD
```

```
    lerp(albedo, albedo + matcap, scale);
    #elif _BLEND2_BLEND
    lerp(albedo, matcap, scale);
    #elif _BLEND2_MULTIPLY
    lerp(albedo, matcap * albedo, scale);
    #endif
    #if _MATCAP2_ROUGHNESS
    matcap2 = lerp(matcap2, albedo, roughness);
    #endif
    matcap2 -= albedo;
    #if _DEBUG_MATCAP2
    return matcap2;
    #endif
    matcap1 += matcap2;
    #endif
    return matcap1;
}
```

7. 染色:
    1. HSV空间染色，先将RGB转成HSV，再通过色相(Hue)、饱和度(Saturation)、明度(Value)进行修改，最后转换回来。
    2. 我们不同染色部分用遮罩图划分
    3. 原本我们还是用过YIQ空间偏移HSV的方法,该方法计算更简单，不用转换空间，仅乘上转换矩阵即可。但效果有些区别，转不出纯白。因此被撤销。
    4. 还添加了两种染色切换的过渡效果。
    5. 关键代码：Unity中使用HSV颜色模型进行颜色的随机变化

```
// HSV -> RGB
float3 HSV2RGB(float3 c)
{
    float4 K = float4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    float3 p = abs(frac(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * lerp(K.xxx, saturate(p - K.xxx), c.y);
}
// RGB -> HSV
float3 RGB2HSV(float3 c)
{
    float4 K = float4(0.0, -1.0 / 3.0, 2.0 / 3.0, -1.0);
    float4 p = lerp(float4(c.bg, K.wz), float4(c.gb, K.xy), step(c.b,
c.g));
    float4 q = lerp(float4(p.xyw, c.r), float4(c.r, p.yzx), step(p.x,
c.r));
    float d = q.x - min(q.w, q.y);
    float e = 1.0e-10;
    return float3(abs(q.z + (q.w - q.y) / (6.0 * d + e)), d / (q.x + e),
q.x);
}

// 染色计算
inline half3 DyeColor(in half3 col, in float2 uv){
    half3 dyeMask = tex2D(_DyeFlowMask,uv).rgb;
    float3 offsetHSV = 0;
    // float3 offsetRow1 = float3(1,0,0);
    // float3 offsetRow2 = float3(0,1,0);
```

```
    // float3 offsetRow3 = float3(0,0,1);
    half intensity = 1.0;
    // 染色区域选择
    [branch]if (dyeMask.r > 0.01)
    {
        offsetHSV = _Offset1.xyz;
        intensity = dyeMask.r;
    }
    else [branch]if (dyeMask.g > 0.01)
    {
        offsetHSV = _Offset2.xyz;
        intensity = dyeMask.g;
    }
    else [branch]if (dyeMask.b > 0.01)
    {
        offsetHSV = _Offset3.xyz;
        intensity = dyeMask.b;
    }
    else
    {
        return col;
    }
    // HSV偏移
    #if 1 //
    // RGB2HSV->Apply Offset->HSV2RGB
    {
        float3 hsv = RGB2HSV(col.rgb);
        hsv.x = offsetHSV.x;
        hsv.y = saturate(hsv.y + offsetHSV.y);
        hsv.z = saturate(hsv.z + offsetHSV.z);
        return lerp(col,HSV2RGB(hsv),intensity);
    }
    #else // 仅保留，不调用
        // YIQ偏移HSV,在cpu端计算，已废弃
        // Apply YIQ_Matrix
    {
        half3 ret = 0;
        ret.x = dot(col, offsetRow1);
        ret.y = dot(col, offsetRow2);
        ret.z = dot(col, offsetRow3);
        return lerp(col, ret, intensity);
    }
    #endif
}
```

8. 流光
    1. 参考天姬变的衣服流光制作
    2. 两层滚动UV一层闪光
    3. 关键代码

```
half3 FlowLight(half3 col, float2 maskUV, float4 flowUV)
{
// 对时间截断取余，避免过大在后续计算中精度不足
float tx = fmod(_Time.x,100);
// 遮罩
```

```
half maskA = tex2D(_DyeFlowMask, maskUV).a;
// 两层流光
half flowCol1 = tex2D(_FlowLightTex, flowUV.xy).r;
half flowCol2 = tex2D(_FlowLightTex, flowUV.zw).g;
// 闪光
half blinCol = tex2D(_FlowLightTex, maskUV * _BlinkTile * 0.97 + tx *
_BlinkSpeed).b;
blinCol = clamp(blinCol *tex2D(_FlowLightTex, maskUV * _BlinkTile - tx *
_BlinkSpeed).b * 200,0,2);
half3 tempCol = 0;
tempCol += flowCol1 * _FlowParam1.w * _FlowColor1;
tempCol += flowCol2 * _FlowParam2.w * _FlowColor2;
tempCol += blinCol * _BlinkColor;
col += tempCol.rgb * maskA;
return col;
}
```

9. 毛发
    1. 使用多层外扩(20层效果比较好),目前我们用了10层
    2. 使用噪声和外扩衰减做clip
    3. 参考[Unity 毛发渲染]
    4. 关键源代码

```
// BasicToonFur.cginc
    v2f_fur vert_fur (appdata_fur v)
    {
        // ...
        // 边缘光
        half Fresnel = 1 - max(0,dot(o.worldNormal,worldView));
        half RimLight = Fresnel * Occlusion;
        RimLight *= RimLight * _FresnelLV * (_LightColor0 + SH) ;
        color += RimLight;
        // 平行光
        half dotNL = dot(worldLight, o.worldNormal);
        // _LightFilter控制平行光扩散，模拟毛发特性
        half DirLight = saturate(dotNL + _LightFilter + FUR_STEP);
        half3 diffuse = DirLight  * _LightColor0 * _DirLightExposure;
        // ...
    }
    fixed4 frag_fur (v2f_fur i) : SV_Target
    {
        // sample the texture
        // 固有色
        fixed4 albedo = tex2D(_MainTex, i.uv.xy);
        // 毛发形状噪声
        fixed noise = tex2D(_LayerTex,i.uv.zw).r;
        // 外扩毛发粗细控制
        fixed alpha = clamp(noise - (FUR_STEP * FUR_STEP) * _FurDensity,
0, 1);
        clip(alpha - 0.001);
        // ...
        return half4( col  , alpha * _AlphaScale);
    }
// BasicToonFur.shader
```