Lecture14 非真实渲染

《Real Time Rendering, third edition》

NPAR Non-Photorealistic Animation and Rendering

《艺术化绘制的图形学原理和方法》

Toon Shader Free，Toon Styles Shader Pack

Hand-Drawn Shader Pack （包含铅笔渲染、蜡笔渲染等多种手绘风格的非真实感渲染）

1. 卡通风格渲染

卡通渲染的实现方法之一是基于**色调的着色技术 tone-based shading**

- 在实现中，我们往往会使用漫反射系数对一张一维纹理进行采样，以控制漫反射的色调
- 卡通风格的高光效果也往往是一块分界明显的纯色区域

渲染轮廓线

实时渲染中，轮廓线的渲染是应用非常广泛的一种效果，有 5 种类型

基于观察角度和表面法线的轮廓线渲染

- 使用视角方向和表面法线点乘的结果来得到轮廓线的信息
- 简单快速，在一个 Pass 就可以得到渲染结果
- 局限性大，很多模型渲染出来的描边效果不尽人意

过程式几何轮廓线渲染

- 使用两个 Pass 渲染，第一个 Pass 渲染背面的面片，并使用某些技术使得它轮廓可见，第二个 Pass 再渲染正面的面片
- 快速有效，适用于大多数表面平滑的模型
- 不适合类似立方体的模型

基于图像处理的轮廓线渲染

- 边缘检测方法
- 可以适用于任何种类的模型
- 一些深度和法线变换很小的轮廓无法被检测出来，例如桌子上的纸张

基于轮廓边检测和轮廓线渲染

- 上面提到的方法无法控制轮廓线的风格渲染，对于一些情况，我们希望渲染出独特风格的轮廓线（水墨风格），为此，我们希望可以检测出精确的轮廓边，然后直接渲染它们

检测一条边是否是轮廓边的公式很简单，只需要检查和这条边相邻的两个三角面片是否满足一下条件

$$(\boldsymbol{n}_0 \cdot \boldsymbol{v} > 0) \neq (\boldsymbol{n}_1 \cdot \boldsymbol{v} > 0)$$

- $\boldsymbol{n}_0$ 和 $\boldsymbol{n}_1$：两个相邻三角面片的法向量
- $\boldsymbol{v}$ 从视角看向该边任意顶点的向量

该公式的本质在于检测两个相邻的三角面片是否一个朝正面，一个朝向背面

混合

- 混合上述几种渲染种类的方法
- 找到精确的轮廓边，把模型和轮廓边渲染到纹理中，再使用图像处理识别轮廓线，在图像空间下进行风格化渲染

实践 - 过程式几何轮廓线渲染

Shader

第一个 Pass

```
Pass {
    NAME "OUTLINE"

    Cull Front // 只渲染背面

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    #include "UnityCG.cginc"

```

```
    float _Outline;
    fixed4 _OutlineColor;

    struct a2v {
        float4 vertex : POSITION;
        float3 normal : NORMAL;
    };

    struct v2f {
         float4 pos : SV_POSITION;
    };

    v2f vert (a2v v) {
        v2f o;

        float4 pos = float4(UnityObjectToViewPos(v.vertex),
1.0);
        float3 normal = mul((float3x3)UNITY_MATRIX_IT_MV,
v.normal);
        normal.z = -0.5; // 对于内凹模型，可能发生背面面片遮挡正面面片的情况，所以我们首先对顶点法线z分量进行处理，使得扩展后的背面更加扁平化
        pos = pos + float4(normalize(normal), 0) * _Outline;
// 把模型的顶点沿着法线方向扩展一段距离，使得背部轮廓线可见
        o.pos = mul(UNITY_MATRIX_P, pos);

        return o;
    }

    float4 frag(v2f i) : SV_Target {
        return float4(_OutlineColor.rgb, 1);
    }

    ENDCG
}
```

第二个 Pass

```
Pass {
        Tags { "LightMode"="ForwardBase" }

        Cull Back

```

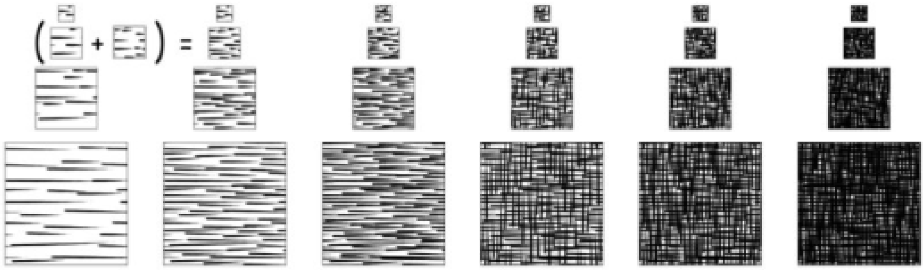
```
      CGPROGRAM

      #pragma vertex vert
      #pragma fragment frag

      #pragma multi_compile_fwdbase

      #include "UnityCG.cginc"
      #include "Lighting.cginc"
      #include "AutoLight.cginc"
      #include "UnityShaderVariables.cginc"

      fixed4 _Color;
      sampler2D _MainTex;
      float4 _MainTex_ST;
      sampler2D _Ramp;
      fixed4 _Specular;
      fixed _SpecularScale;

      struct a2v {
         float4 vertex : POSITION;
         float3 normal : NORMAL;
         float4 texcoord : TEXCOORD0;
         float4 tangent : TANGENT;
      };

      struct v2f {
         float4 pos : POSITION;
         float2 uv : TEXCOORD0;
         float3 worldNormal : TEXCOORD1;
         float3 worldPos : TEXCOORD2;
         SHADOW_COORDS(3)
      };

      v2f vert (a2v v) {
         v2f o;

         o.pos = UnityObjectToClipPos( v.vertex);
         o.uv = TRANSFORM_TEX (v.texcoord, _MainTex);
         o.worldNormal  =
UnityObjectToWorldNormal(v.normal);
         o.worldPos = mul(unity_ObjectToWorld,
v.vertex).xyz;
```

```
            TRANSFER_SHADOW(o);

            return o;
        }

        float4 frag(v2f i) : SV_Target {
            fixed3 worldNormal = normalize(i.worldNormal);
            fixed3 worldLightDir =
normalize(UnityWorldSpaceLightDir(i.worldPos));
            fixed3 worldViewDir =
normalize(UnityWorldSpaceViewDir(i.worldPos));
            fixed3 worldHalfDir = normalize(worldLightDir +
worldViewDir);

            fixed4 c = tex2D (_MainTex, i.uv);
            fixed3 albedo = c.rgb * _Color.rgb;

            fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz *
albedo;

            UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

            fixed diff =  dot(worldNormal, worldLightDir);
            diff = (diff * 0.5 + 0.5) * atten;

            fixed3 diffuse = _LightColor0.rgb * albedo *
tex2D(_Ramp, float2(diff, diff)).rgb;

            fixed spec = dot(worldNormal, worldHalfDir); // 卡
通渲染中，我们同样计算normal和halfDir的点乘结果
             // 在一个小的区域w内进行抗锯齿处理
            fixed w = fwidth(spec) * 2.0;
             // spec-<-w 返回0, spec
             // * step(0.0001, _SpecularScale): _SpecularScale
为0的时候，完全消除高光反射结果
            fixed3 specular = _Specular.rgb * lerp(0, 1,
smoothstep(-w, w, spec + _SpecularScale - 1)) *
step(0.0001, _SpecularScale);

            return fixed4(ambient + diffuse + specular, 1.0);
        }
```

```
        ENDCG
    }
}
```

2. 素描风格渲染

微软研究院 Praun 在 2001 年 SIGGRAPH 发表一篇著名的论文，他们使用提前生成素描纹理来实现实时的素描风格渲染，这些纹理组成了一个**色调艺术映射 Tonal Art Map，TAM**



- 从左到右纹理的笔触逐渐增多，用于模拟不同光照下的漫反射结果
- 从上到下对应了每张纹理的多级渐远纹理 mipmaps，保持笔触之间的间隔，以更真实地模拟素描效果

实践 - 素描风格 Shader

```
///
///  Reference:    Praun E, Hoppe H, Webb M, et al. Real-time hatching[C]
///                    Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM, 2001: 581.
///
Shader "Unity Shaders Book/Chapter 14/Hatching" {
    Properties {
        _Color ("Color Tint", Color) = (1, 1, 1, 1) // 模型颜色
        _TileFactor ("Tile Factor", Float) = 1 // 纹理的平铺系数，越大模型上的素描线条越密
        _Outline ("Outline", Range(0, 1)) = 0.1
        _Hatch0 ("Hatch 0", 2D) = "white" {}
        _Hatch1 ("Hatch 1", 2D) = "white" {}
        _Hatch2 ("Hatch 2", 2D) = "white" {}
        _Hatch3 ("Hatch 3", 2D) = "white" {}
        _Hatch4 ("Hatch 4", 2D) = "white" {}
        _Hatch5 ("Hatch 5", 2D) = "white" {}
    }
```

```
    SubShader {
        Tags { "RenderType"="Opaque" "Queue"="Geometry"}

        UsePass "Unity Shaders Book/Chapter 14/Toon
Shading/OUTLINE" // 直接使用卡通渲染中的渲染轮廓的Pass

        Pass {
            Tags { "LightMode"="ForwardBase" }

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #pragma multi_compile_fwdbase

            #include "UnityCG.cginc"
            #include "Lighting.cginc"
            #include "AutoLight.cginc"
            #include "UnityShaderVariables.cginc"

            fixed4 _Color;
            float _TileFactor;
            sampler2D _Hatch0;
            sampler2D _Hatch1;
            sampler2D _Hatch2;
            sampler2D _Hatch3;
            sampler2D _Hatch4;
            sampler2D _Hatch5;

            struct a2v {
                float4 vertex : POSITION;
                float4 tangent : TANGENT;
                float3 normal : NORMAL;
                float2 texcoord : TEXCOORD0;
            };

            struct v2f {
                float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
                // 需要6个混合权重，把它们存储在两个fixed3类型的变量
中
```

```
        fixed3 hatchWeights0 : TEXCOORD1;
        fixed3 hatchWeights1 : TEXCOORD2;
        float3 worldPos : TEXCOORD3;
        SHADOW_COORDS(4)
    };

    v2f vert(a2v v) {
        v2f o;

        o.pos = UnityObjectToClipPos(v.vertex);

        o.uv = v.texcoord.xy * _TileFactor; // 纹理采样坐
标

        // 计算逐顶点光照
        fixed3 worldLightDir =
normalize(WorldSpaceLightDir(v.vertex));
        fixed3 worldNormal =
UnityObjectToWorldNormal(v.normal);
        fixed diff = max(0, dot(worldLightDir,
worldNormal));

        o.hatchWeights0 = fixed3(0, 0, 0);
        o.hatchWeights1 = fixed3(0, 0, 0);

        // diff缩放到[0,7]的范围内
        float hatchFactor = diff * 7.0;

        // 把区间均匀划分成7个子区间，判断hatchFactor所处的
子区间来计算纹理的混合权重
        if (hatchFactor > 6.0) {
            // Pure white, do nothing
        } else if (hatchFactor > 5.0) {
            o.hatchWeights0.x = hatchFactor - 5.0;
        } else if (hatchFactor > 4.0) {
            o.hatchWeights0.x = hatchFactor - 4.0;
            o.hatchWeights0.y = 1.0 - o.hatchWeights0.x;
        } else if (hatchFactor > 3.0) {
            o.hatchWeights0.y = hatchFactor - 3.0;
            o.hatchWeights0.z = 1.0 - o.hatchWeights0.y;
        } else if (hatchFactor > 2.0) {
            o.hatchWeights0.z = hatchFactor - 2.0;
            o.hatchWeights1.x = 1.0 - o.hatchWeights0.z;
```

```
                } else if (hatchFactor > 1.0) {
                    o.hatchWeights1.x = hatchFactor - 1.0;
                    o.hatchWeights1.y = 1.0 - o.hatchWeights1.x;
                } else {
                    o.hatchWeights1.y = hatchFactor;
                    o.hatchWeights1.z = 1.0 - o.hatchWeights1.y;
                }

                o.worldPos = mul(unity_ObjectToWorld,
v.vertex).xyz;

                TRANSFER_SHADOW(o);

                return o;
            }

            fixed4 frag(v2f i) : SV_Target {
                fixed4 hatchTex0 = tex2D(_Hatch0, i.uv) *
i.hatchWeights0.x;
                fixed4 hatchTex1 = tex2D(_Hatch1, i.uv) *
i.hatchWeights0.y;
                fixed4 hatchTex2 = tex2D(_Hatch2, i.uv) *
i.hatchWeights0.z;
                fixed4 hatchTex3 = tex2D(_Hatch3, i.uv) *
i.hatchWeights1.x;
                fixed4 hatchTex4 = tex2D(_Hatch4, i.uv) *
i.hatchWeights1.y;
                fixed4 hatchTex5 = tex2D(_Hatch5, i.uv) *
i.hatchWeights1.z;
                 // 计算纯白在渲染中的贡献度，通过1减去所有6张纹理的权
重得到
                fixed4 whiteColor = fixed4(1, 1, 1, 1) * (1 -
i.hatchWeights0.x - i.hatchWeights0.y - i.hatchWeights0.z -
                        i.hatchWeights1.x - i.hatchWeights1.y
- i.hatchWeights1.z);

                fixed4 hatchColor = hatchTex0 + hatchTex1 +
hatchTex2 + hatchTex3 + hatchTex4 + hatchTex5 + whiteColor;

                UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

                return fixed4(hatchColor.rgb * _Color.rgb *
atten, 1.0);
```

```
            }


            ENDCG
        }
    }
    FallBack "Diffuse"
}
```