Lecture12 屏幕后处理效果

> Unity 的 Image Effect 包中有更多特效的实现
>
> GPU Gems 系列介绍了很多图像处理的渲染技术《GPU Gems 3》

1. 基本屏幕后处理脚本系统

屏幕后处理效果 screen post-processing effects 是指在渲染完整场景得到屏幕图像后，再对这个图像进行一系列操作，实现各种屏幕特效（景深 Depth of Field、运动模糊 Motion Blur）

OnRenderImage 接口

Unity 为我们提供了一个 **OnRenderImage** 接口

```
MonoBehaviour.OnRenderImage(RenderTexture src, RenderTexture dest)
```

- `src`：当前渲染得到的图像
- `dest`：目标渲染纹理，将会显示在屏幕上

在这个函数中，我们通常利用 **Graphics.Blit** 函数完成对渲染纹理的处理

```
public static void Blit(Texture src, RenderTexture dest);
public static void Blit(Texture src, RenderTexture dest, Material mat, int pass = -1);
public static void Blit(Texture src, Material mat, int pass = -1);
```

- `src`：源纹理，当前屏幕的渲染纹理
- `dest`：目标渲染纹理，如果它的值为 null 就会直接将结果显示在屏幕上
- `mat`：使用的材质，进行各种屏幕后处理操作
- `pass`：默认 -1 表示依次会调用 Shader 内所有的 Pass，否则，只会调用给给定索引的 Pass

默认情况下，OnRenderImage 会在所有的透明和不透明的 Pass 执行完毕后都会调用，有时候我们希望它只对不透明的物体产生影响，那么我们可以再 OnRenderImage 函数前添加 ImageEffectOpaque 属性来实现这样的目的

屏幕后处理脚本 Base

- 在摄像机中添加一个用于屏幕后处理的脚本
- 在这个脚本中实现 OnRenderImage 函数来获得当前屏幕的渲染纹理
- 调用 Graphics.Blit 函数使用特定的 Unity Shader 对当前图像进行处理
- 把返回的渲染纹理显示到屏幕上
- （对于一些复杂的屏幕特效，我们可能多次调用 Graphics.Blit 函数对上一步输出结果进行下一步处理）

```
using UnityEngine;
using System.Collections;

[ExecuteInEditMode] // 编辑器状态下也可以执行脚本
[RequireComponent (typeof(Camera))]
public class PostEffectsBase : MonoBehaviour {

    // Called when start 检查各种资源和条件是否满足
    protected void CheckResources() {
        bool isSupported = CheckSupport();

        if (isSupported == false) {
            NotSupported();
        }
    }

    // Called in CheckResources to check support on this platform
    protected bool CheckSupport() {
        if (SystemInfo.supportsImageEffects == false) {
            Debug.LogWarning("This platform does not support image effects.");
            return false;
        }
```

```
        return true;
    }

    // Called when the platform doesn't support this effect
    protected void NotSupported() {
        enabled = false;
    }

    protected void Start() {
        CheckResources();
    }

    // Called when need to create the material used by this effect
    protected Material CheckShaderAndCreateMaterial(Shader shader, Material material) {
        if (shader == null) {
            return null;
        }

        if (shader.isSupported && material && material.shader == shader)
            return material;

        if (!shader.isSupported) {
            return null;
        }
        else {
            material = new Material(shader);
            material.hideFlags = HideFlags.DontSave;
            if (material)
                return material;
            else
                return null;
        }
    }
}
```

2. 调整屏幕亮度、饱和度和对比度

处理脚本

```
using UnityEngine;
using System.Collections;

public class BrightnessSaturationAndContrast : PostEffectsBase {

    public Shader briSatConShader; // 指定的Shader
    private Material briSatConMaterial; // 创建的材质
    public Material material {
        get {
            briSatConMaterial = CheckShaderAndCreateMaterial(briSatConShader,
briSatConMaterial);
            return briSatConMaterial;
        }
    }

    [Range(0.0f, 3.0f)]
    public float brightness = 1.0f; // 亮度

    [Range(0.0f, 3.0f)]
    public float saturation = 1.0f; // 饱和度

    [Range(0.0f, 3.0f)]
    public float contrast = 1.0f; // 对比度

    void OnRenderImage(RenderTexture src, RenderTexture dest) {
```

```
        if (material != null) {
            material.SetFloat("_Brightness", brightness);
            material.SetFloat("_Saturation", saturation);
            material.SetFloat("_Contrast", contrast);

            Graphics.Blit(src, dest, material);
        } else {
            Graphics.Blit(src, dest);
        }
    }
}
```

Shader

```
Shader "Unity Shaders Book/Chapter 12/Brightness Saturation And Contrast" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _Brightness ("Brightness", Float) = 1
        _Saturation("Saturation", Float) = 1
        _Contrast("Contrast", Float) = 1
    }
    SubShader {
        Pass {
            ZTest Always Cull Off ZWrite Off // 屏幕后处理Shader标配

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            sampler2D _MainTex;
            half _Brightness;
            half _Saturation;
            half _Contrast;

            struct v2f {
                float4 pos : SV_POSITION;
                half2 uv: TEXCOORD0;
            };

            // Unity内置的appdata_img结构体
            v2f vert(appdata_img v) {
                v2f o;

                o.pos = UnityObjectToClipPos(v.vertex);

                o.uv = v.texcoord;

                return o;
            }

            fixed4 frag(v2f i) : SV_Target {
                fixed4 renderTex = tex2D(_MainTex, i.uv);

                // Apply brightness
                fixed3 finalColor = renderTex.rgb * _Brightness;

                // Apply saturation
                fixed luminance = 0.2125 * renderTex.r + 0.7154 * renderTex.g + 0.0721 *
renderTex.b;
                fixed3 luminanceColor = fixed3(luminance, luminance, luminance);
                finalColor = lerp(luminanceColor, finalColor, _Saturation);
```

```
            // Apply contrast
            fixed3 avgColor = fixed3(0.5, 0.5, 0.5);
            finalColor = lerp(avgColor, finalColor, _Contrast);

            return fixed4(finalColor, renderTex.a);
        }

        ENDCG
    }
  }

  Fallback Off // 关闭 Unity Shader的Fallback
}
```

- **亮度**：原颜色 * 亮度系数 _Brightness
- **饱和度**
  - 发光性 Luminance：对颜色的每个分量乘以一个特定的系数再相加
  - 为该亮度创建了一个饱和度为 0 的颜色，并使用 _Saturation 属性和上一步得到的颜色进行插值
- **对比度**
  - 创建一个对比度为 0 的颜色值 (0.5, 0.5, 0.5)
  - 使用 _Contrast 属性在其和上一步得到的颜色之间进行插值

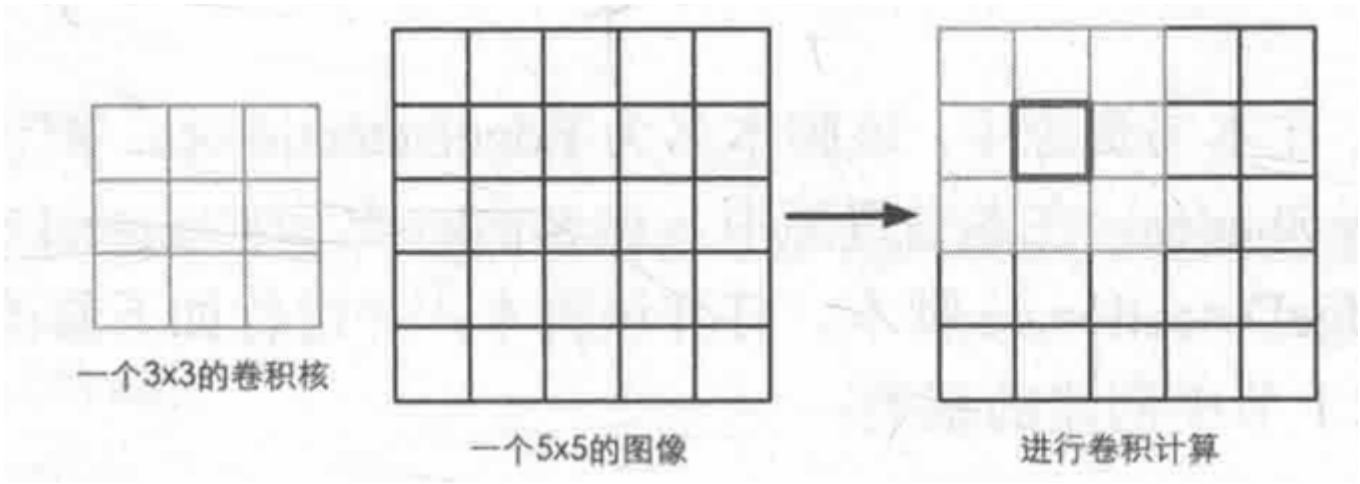
3. 边缘检测

边缘检测的原理是利用了边缘检测算子来进行卷积操作

卷积

使用一个**卷积核 kerenl** 对一张图像中的每个像素进行一系列操作

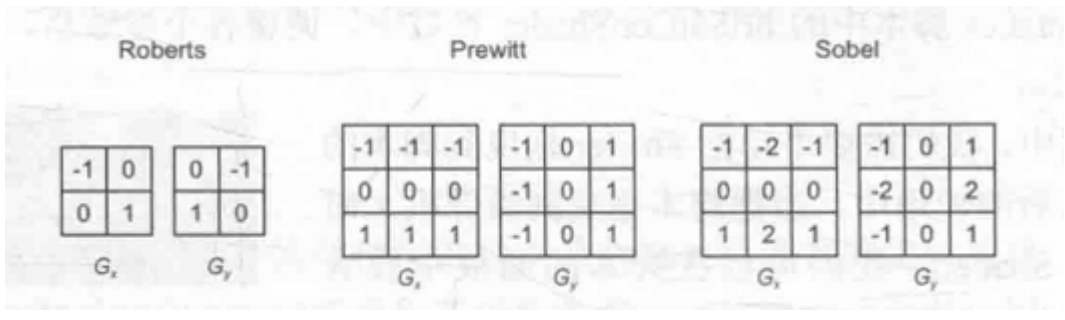
卷积核通常是一个四方形网格结构，该区域内每个方格都有一个权重值，当对图像中的某个像素进行卷积时，我们会把卷积核的中心放在该像素上



- 使用一张 3x3 大小的卷积核对一张 5x5 大小的图像进行卷积操作

常见边缘检测算子

- 相邻像素之间存在差别明显的颜色、纹理、亮度等属性时候，我们认为它们之间应该有一条边界
- 相邻像素之间的差值可以用**梯度 gradient** 表示出来，边缘处的梯度绝对值会比较大



- 三个算子都包含了两个方向的卷积核，分别检测水平和垂直方向的边缘信息

进行边缘检测时候，我们需要对每个像素分别进行一次卷积计算，得到两个方向上的梯度值 G_x 和 G_y，整体梯度可以按照下面的公式计算得到

$$G = \sqrt{G_x^2 + G_y^2}$$

上述计算包含了开根号，为了性能，有时候会使用绝对值操作代替开根号

$$G = |G_x| + |G_y|$$

处理脚本

```
using UnityEngine;
using System.Collections;

public class EdgeDetection : PostEffectsBase {

   public Shader edgeDetectShader;
   private Material edgeDetectMaterial = null;
   public Material material {
      get {
         edgeDetectMaterial = CheckShaderAndCreateMaterial(edgeDetectShader,
edgeDetectMaterial);
         return edgeDetectMaterial;
      }
   }

   [Range(0.0f, 1.0f)]
   public float edgesOnly = 0.0f; // 边缘线强度

   public Color edgeColor = Color.black; // 描边颜色

   public Color backgroundColor = Color.white; // 背景颜色

   void OnRenderImage (RenderTexture src, RenderTexture dest) {
      if (material != null) {
         material.SetFloat("_EdgeOnly", edgesOnly);
         material.SetColor("_EdgeColor", edgeColor);
         material.SetColor("_BackgroundColor", backgroundColor);

         Graphics.Blit(src, dest, material);
      } else {
         Graphics.Blit(src, dest);
      }
   }
}
```

Shader

```
Shader "Unity Shaders Book/Chapter 12/Edge Detection" {
   Properties {
      _MainTex ("Base (RGB)", 2D) = "white" {}
      _EdgeOnly ("Edge Only", Float) = 1.0
      _EdgeColor ("Edge Color", Color) = (0, 0, 0, 1)
      _BackgroundColor ("Background Color", Color) = (1, 1, 1, 1)
   }
   SubShader {
      Pass {
         ZTest Always Cull Off ZWrite Off

         CGPROGRAM

         #include "UnityCG.cginc"

         #pragma vertex vert
         #pragma fragment fragSobel

         sampler2D _MainTex;
         uniform half4 _MainTex_TexelSize; // 访问xxx纹理对应的每个纹素的大小，例如一张512x512大
小的纹理，该值为1/512
         fixed _EdgeOnly;
         fixed4 _EdgeColor;
```

```
fixed4 _BackgroundColor;

struct v2f {
    float4 pos : SV_POSITION;
    half2 uv[9] : TEXCOORD0; // 边缘检测时需要的纹理坐标
};

v2f vert(appdata_img v) {
    v2f o;
    o.pos = UnityObjectToClipPos(v.vertex);

    half2 uv = v.texcoord;

    o.uv[0] = uv + _MainTex_TexelSize.xy * half2(-1, -1);
    o.uv[1] = uv + _MainTex_TexelSize.xy * half2(0, -1);
    o.uv[2] = uv + _MainTex_TexelSize.xy * half2(1, -1);
    o.uv[3] = uv + _MainTex_TexelSize.xy * half2(-1, 0);
    o.uv[4] = uv + _MainTex_TexelSize.xy * half2(0, 0);
    o.uv[5] = uv + _MainTex_TexelSize.xy * half2(1, 0);
    o.uv[6] = uv + _MainTex_TexelSize.xy * half2(-1, 1);
    o.uv[7] = uv + _MainTex_TexelSize.xy * half2(0, 1);
    o.uv[8] = uv + _MainTex_TexelSize.xy * half2(1, 1);

    return o;
}

fixed luminance(fixed4 color) {
    return  0.2125 * color.r + 0.7154 * color.g + 0.0721 * color.b;
}

half Sobel(v2f i) {
    const half Gx[9] = {-1,  0,  1,
                        -2,  0,  2,
                        -1,  0,  1};
    const half Gy[9] = {-1, -2, -1,
                        0,  0,  0,
                        1,  2,  1};

    half texColor;
    half edgeX = 0;
    half edgeY = 0;
    for (int it = 0; it < 9; it++) {
        texColor = luminance(tex2D(_MainTex, i.uv[it]));
        edgeX += texColor * Gx[it];
        edgeY += texColor * Gy[it];
    }

    // 1-|Gx|-|Gy|得到edge，值越小，该位置越有可能是一个边缘点
    half edge = 1 - abs(edgeX) - abs(edgeY);

    return edge;
}

fixed4 fragSobel(v2f i) : SV_Target {
    half edge = Sobel(i); // 调用Sobel函数计算当前像素的梯度值

    // 利用该值分别计算了背景为原图和纯色下的颜色值
    fixed4 withEdgeColor = lerp(_EdgeColor, tex2D(_MainTex, i.uv[4]), edge);
    fixed4 onlyEdgeColor = lerp(_EdgeColor, _BackgroundColor, edge);
    // 两者间插值得到最终的像素值
    return lerp(withEdgeColor, onlyEdgeColor, _EdgeOnly);
}
```

```
            ENDCG
        }
    }
    FallBack Off // 关闭该Shader的Fallback
}
```

4. 高斯模糊

模糊有很多种实现方法

- **均值模糊**：同样使用卷积操作，卷积核的个元素值相等，且相加等于 1，得到的像素值为邻域内各个像素的平均值
- **中值模糊**：对所有像素排序后的中值替换掉原颜色

高斯滤波

使用名为高斯核的卷积核，一个正方形大小的滤波核，每个元素计算基于下面的高斯方程

$$G(x,y)=\frac{1}{2\pi\sigma^2}e^{\frac{x^2+y^2}{2\sigma^2}}$$

- σ：标准方程（一般取值为 1）
- x,y：分别对应当前位置到卷积核中心的整数距离
- 为了保证滤波后的图像不会变暗，需要对高斯核中的权重进行归一化（每个权重除以所有权重的和），因此高斯函数 e 前面的系数实际不会对结果有任何影响
- 模拟每个像素对当前处理像素的影响程度，距离越近，印象越高，高斯核维数越高，模糊程度越高

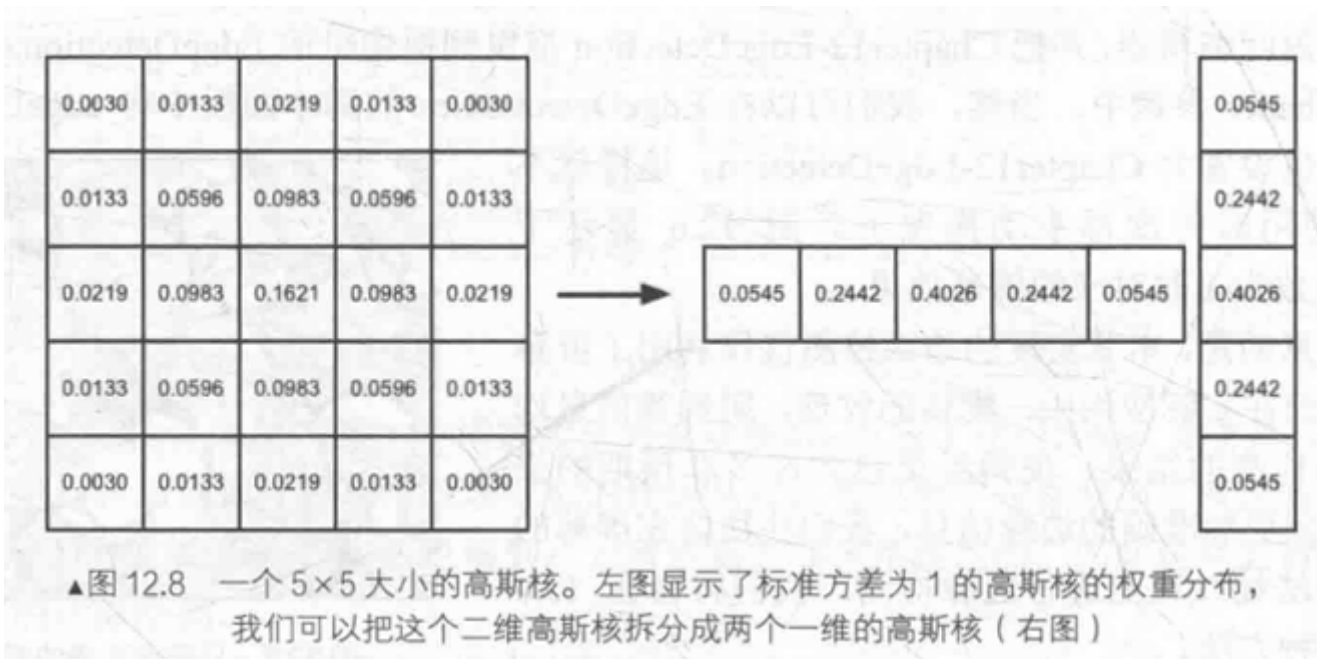
优化 - 快速高斯滤波

https://blog.csdn.net/HLW0522/article/details/100051564

当使用一个 NxN 的高斯核对图像进行卷积滤波的时候，需要 NxNxWxH 次纹理采样

二维高斯函数可以拆成两个一维函数，则采样次数只需要 2xNxWxH

$$G(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2}{2\sigma^2}-\frac{y2}{2\sigma^2}}$$
$$=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2}{2\sigma^2}}*\frac{1}{2\pi\sigma^2}e^{-\frac{y^2}{2\sigma^2}}=G(x)*G(y)$$



▲图 12.8　一个 5×5 大小的高斯核。左图显示了标准方差为 1 的高斯核的权重分布，我们可以把这个二维高斯核拆分成两个一维的高斯核（右图）

高斯滤波脚本

```
// Blur iterations - larger number means more blur.
[Range(0, 4)]
public int iterations = 3; // 迭代次数

// Blur spread for each iteration - larger value means more blur
[Range(0.2f, 3.0f)]
public float blurSpread = 0.6f; // 模糊范围

[Range(1, 8)]
public int downSample = 2; // 缩放系数
```

OnRenderImage 第一个实现

```
// 1st edition: just apply blur
void OnRenderImage(RenderTexture src, RenderTexture dest) {
    if (material != null) {
        int rtW = src.width;
        int rtH = src.height;
        RenderTexture buffer = RenderTexture.GetTemporary(rtW, rtH, 0);

         // 调用两个Pass，使用一块中间缓存来存储第一个Pass执行完毕后得到的模糊结果
        // Render the vertical pass
        Graphics.Blit(src, buffer, material, 0);
        // Render the horizontal pass
        Graphics.Blit(buffer, dest, material, 1);

         // 释放之前分配的内存
        RenderTexture.ReleaseTemporary(buffer);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

OnRenderImage 第二个实现（降采样提高性能）

```
/// 2nd edition: scale the render texture
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
         // 声明缓冲区大小的时候，使用了小于原屏幕分辨率大小的尺寸
        int rtW = src.width/downSample;
        int rtH = src.height/downSample;
        RenderTexture buffer = RenderTexture.GetTemporary(rtW, rtH, 0);
         // 滤波模式设置为算双线性
        buffer.filterMode = FilterMode.Bilinear;

        // Render the vertical pass
        Graphics.Blit(src, buffer, material, 0);
        // Render the horizontal pass
        Graphics.Blit(buffer, dest, material, 1);

        RenderTexture.ReleaseTemporary(buffer);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

OnRenderImage 第三个实现（加上迭代次数）

```
/// 3rd edition: use iterations for larger blur
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        int rtW = src.width/downSample;
        int rtH = src.height/downSample;

        RenderTexture buffer0 = RenderTexture.GetTemporary(rtW, rtH, 0);
        buffer0.filterMode = FilterMode.Bilinear;

        Graphics.Blit(src, buffer0);

        for (int i = 0; i < iterations; i++) {
            material.SetFloat("_BlurSize", 1.0f + i * blurSpread);

            RenderTexture buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);

            // Render the vertical pass
            Graphics.Blit(buffer0, buffer1, material, 0);
```

```
            RenderTexture.ReleaseTemporary(buffer0);
            buffer0 = buffer1;
            buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);

            // Render the horizontal pass
            Graphics.Blit(buffer0, buffer1, material, 1);

            RenderTexture.ReleaseTemporary(buffer0);
            buffer0 = buffer1;
        }

        Graphics.Blit(buffer0, dest);
        RenderTexture.ReleaseTemporary(buffer0);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

高斯滤波 Shader

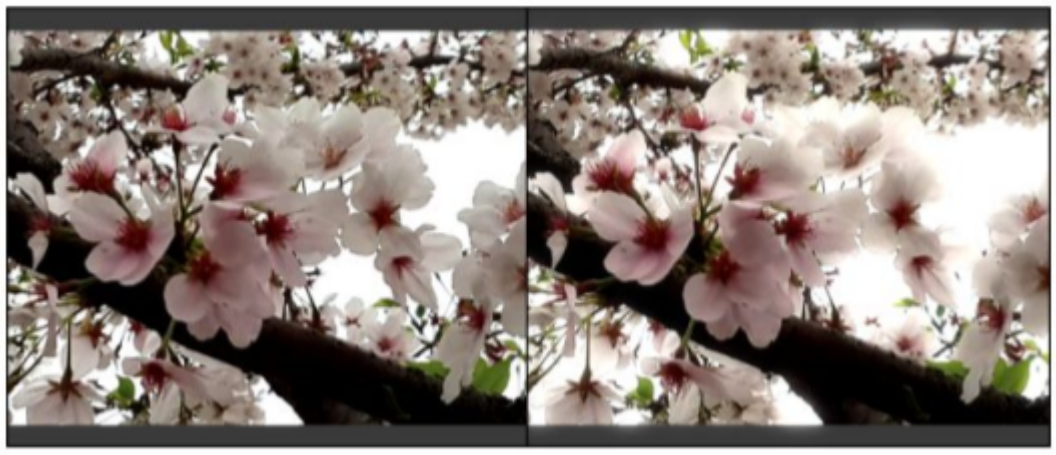
```
Shader "Unity Shaders Book/Chapter 12/Gaussian Blur" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _BlurSize ("Blur Size", Float) = 1.0
    }
    SubShader {
        CGINCLUDE // 注意这个

        #include "UnityCG.cginc"

        sampler2D _MainTex;
        half4 _MainTex_TexelSize;
        float _BlurSize; // 控制采样距离，在高斯核维数不变的情况下，_BlurSize越大，模糊程度越高，过高可能造成虚影

        struct v2f {
            float4 pos : SV_POSITION;
            half2 uv[5]: TEXCOORD0;
        };

        v2f vertBlurVertical(appdata_img v) {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex);

            half2 uv = v.texcoord;

            o.uv[0] = uv;
            o.uv[1] = uv + float2(0.0, _MainTex_TexelSize.y * 1.0) * _BlurSize;
            o.uv[2] = uv - float2(0.0, _MainTex_TexelSize.y * 1.0) * _BlurSize;
            o.uv[3] = uv + float2(0.0, _MainTex_TexelSize.y * 2.0) * _BlurSize;
            o.uv[4] = uv - float2(0.0, _MainTex_TexelSize.y * 2.0) * _BlurSize;

            return o;
        }

        v2f vertBlurHorizontal(appdata_img v) {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex);

            half2 uv = v.texcoord;

            o.uv[0] = uv;
            o.uv[1] = uv + float2(_MainTex_TexelSize.x * 1.0, 0.0) * _BlurSize;
```

```
        o.uv[2] = uv - float2(_MainTex_TexelSize.x * 1.0, 0.0) * _BlurSize;
        o.uv[3] = uv + float2(_MainTex_TexelSize.x * 2.0, 0.0) * _BlurSize;
        o.uv[4] = uv - float2(_MainTex_TexelSize.x * 2.0, 0.0) * _BlurSize;

        return o;
    }

    fixed4 fragBlur(v2f i) : SV_Target {
        float weight[3] = {0.4026, 0.2442, 0.0545};

        fixed3 sum = tex2D(_MainTex, i.uv[0]).rgb * weight[0];

        for (int it = 1; it < 3; it++) {
            sum += tex2D(_MainTex, i.uv[it*2-1]).rgb * weight[it];
            sum += tex2D(_MainTex, i.uv[it*2]).rgb * weight[it];
        }

        return fixed4(sum, 1.0);
    }

    ENDCG

    ZTest Always Cull Off ZWrite Off

    Pass {
        NAME "GAUSSIAN_BLUR_VERTICAL" // 使用Name语义定义它们的名字，这样可以再其它Shader直接通过它们的名字来使用该Pass

        CGPROGRAM

        #pragma vertex vertBlurVertical
        #pragma fragment fragBlur

        ENDCG
    }

    Pass {
        NAME "GAUSSIAN_BLUR_HORIZONTAL"

        CGPROGRAM

        #pragma vertex vertBlurHorizontal
        #pragma fragment fragBlur

        ENDCG
    }
    }
    FallBack "Diffuse"
}
```

- 我们第一次使用 CGINCLUDE-ENDCG 来组织代码，它不需要包含在任何 Pass 语义块中，只需要直接指定需要的顶点着色器核片元着色器函数名即可
- CGINCLUDE 类似 C++ 中的头文件功能，高斯模糊需要定义两个 Pass，但他们使用片元着色器的代码是完全相同的，使用 CGINCLUDE 可以避免我们编写两个完全一样的 frag 函数

5. Bloom 效果

模拟真实摄像机的一种图像效果，使得画面中较量的区域“扩散”到周围区域中，造成一种朦胧的效果



原理

根据一个阈值提取出图像较亮的区域，把它们存储在一张渲染纹理中，再利用高斯模糊对这张渲染纹理进行模糊处理，模拟光线扩散效果

处理脚本

```
using UnityEngine;
using System.Collections;

public class Bloom : PostEffectsBase {

    public Shader bloomShader;
    private Material bloomMaterial = null;
    public Material material {
        get {
            bloomMaterial = CheckShaderAndCreateMaterial(bloomShader, bloomMaterial);
            return bloomMaterial;
        }
    }

    // Blur iterations - larger number means more blur.
    [Range(0, 4)]
    public int iterations = 3;

    // Blur spread for each iteration - larger value means more blur
    [Range(0.2f, 3.0f)]
    public float blurSpread = 0.6f;

    [Range(1, 8)]
    public int downSample = 2;

     // 绝大多数情况，图像亮度值不会超过1，但是如果开启HDR，硬件允许我们把颜色存在更高精度的范围中，这之后亮度值可能超过1
    [Range(0.0f, 4.0f)]
    public float luminanceThreshold = 0.6f;

    void OnRenderImage (RenderTexture src, RenderTexture dest) {
        if (material != null) {
            material.SetFloat("_LuminanceThreshold", luminanceThreshold);

            int rtW = src.width/downSample;
            int rtH = src.height/downSample;

            RenderTexture buffer0 = RenderTexture.GetTemporary(rtW, rtH, 0);
            buffer0.filterMode = FilterMode.Bilinear;

             // 通过Pass1提取图像中较量的区域，将其存储在buffer0中
            Graphics.Blit(src, buffer0, material, 0);

             // 高斯模糊迭代处理
            for (int i = 0; i < iterations; i++) {
                material.SetFloat("_BlurSize", 1.0f + i * blurSpread);
```

```
        RenderTexture buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);

        //下面是高斯模糊的迭代处理
        // Render the vertical pass
        Graphics.Blit(buffer0, buffer1, material, 1);

        RenderTexture.ReleaseTemporary(buffer0);
        buffer0 = buffer1;
        buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);

        // Render the horizontal pass
        Graphics.Blit(buffer0, buffer1, material, 2);

        RenderTexture.ReleaseTemporary(buffer0);
        buffer0 = buffer1;
    }

     // 将buffer0传递给_Bloom材质属性
    material.SetTexture ("_Bloom", buffer0);
     // 使用Shader中的第4个Pass进行混合
    Graphics.Blit (src, dest, material, 3);

        RenderTexture.ReleaseTemporary(buffer0);
    } else {
        Graphics.Blit(src, dest);
    }
  }
}
```

Shader

```
Shader "Unity Shaders Book/Chapter 12/Bloom" {
  Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {} // 输入渲染纹理
    _Bloom ("Bloom (RGB)", 2D) = "black" {} // 高斯模糊后较量的区域
    _LuminanceThreshold ("Luminance Threshold", Float) = 0.5
    _BlurSize ("Blur Size", Float) = 1.0
  }
  SubShader {
    CGINCLUDE

    #include "UnityCG.cginc"

    sampler2D _MainTex;
    half4 _MainTex_TexelSize;
    sampler2D _Bloom;
    float _LuminanceThreshold;
    float _BlurSize;

    struct v2f {
      float4 pos : SV_POSITION;
      half2 uv : TEXCOORD0;
    };

    v2f vertExtractBright(appdata_img v) {
      v2f o;

      o.pos = UnityObjectToClipPos(v.vertex);

      o.uv = v.texcoord;

      return o;
    }
```

```
        fixed luminance(fixed4 color) {
            return  0.2125 * color.r + 0.7154 * color.g + 0.0721 * color.b;
        }

        fixed4 fragExtractBright(v2f i) : SV_Target {
            fixed4 c = tex2D(_MainTex, i.uv);
            fixed val = clamp(luminance(c) - _LuminanceThreshold, 0.0, 1.0);

            return c * val;
        }

        struct v2fBloom {
            float4 pos : SV_POSITION;
            half4 uv : TEXCOORD0;
        };

        v2fBloom vertBloom(appdata_img v) {
            v2fBloom o;

            o.pos = UnityObjectToClipPos (v.vertex);
            o.uv.xy = v.texcoord;
            o.uv.zw = v.texcoord;

            #if UNITY_UV_STARTS_AT_TOP
            if (_MainTex_TexelSize.y < 0.0)
                o.uv.w = 1.0 - o.uv.w;
            #endif

            return o;
        }

        fixed4 fragBloom(v2fBloom i) : SV_Target {
            return tex2D(_MainTex, i.uv.xy) + tex2D(_Bloom, i.uv.zw);
        }

        ENDCG

        ZTest Always Cull Off ZWrite Off

        Pass {
            CGPROGRAM
            #pragma vertex vertExtractBright
            #pragma fragment fragExtractBright

            ENDCG
        }

        UsePass "Unity Shaders Book/Chapter 12/Gaussian Blur/GAUSSIAN_BLUR_VERTICAL" //
UsePass 必须大写

        UsePass "Unity Shaders Book/Chapter 12/Gaussian Blur/GAUSSIAN_BLUR_HORIZONTAL"

        Pass {
            CGPROGRAM
            #pragma vertex vertBloom
            #pragma fragment fragBloom

            ENDCG
        }
    }
    FallBack Off
}
```

6. 运动模糊

运动模糊是真实世界摄像机的一种效果，当摄像机曝光时，拍摄场景发生了变化，就会产生模糊的画面

累计缓存 accumulation buffer

混合多张连续的图像，当物体快速移动产生多张图向后，取它们之间的平均值作为最后的运动模糊图像

这种暴力的方法对于性能消耗很大

速度缓存 velocity buffer

存储各个像素当前的运动速度，来决定模糊的方向和大小

处理脚本

```
using UnityEngine;
using System.Collections;

public class MotionBlur : PostEffectsBase {

    public Shader motionBlurShader;
    private Material motionBlurMaterial = null;

    public Material material {
        get {
            motionBlurMaterial = CheckShaderAndCreateMaterial(motionBlurShader,
motionBlurMaterial);
            return motionBlurMaterial;
        }
    }

    [Range(0.0f, 0.9f)]
    public float blurAmount = 0.5f; // 模糊参数，值越大，运动拖尾效果越明显

    private RenderTexture accumulationTexture; // 保存图像的叠加效果

     // 脚本不运行时，立即销毁accumulationTexture，下次开始应用运动模糊时重新叠加图像
    void OnDisable() {
        DestroyImmediate(accumulationTexture);
    }

    void OnRenderImage (RenderTexture src, RenderTexture dest) {
        if (material != null) {
            // Create the accumulation texture
            if (accumulationTexture == null || accumulationTexture.width != src.width ||
accumulationTexture.height != src.height) {
                DestroyImmediate(accumulationTexture);
                accumulationTexture = new RenderTexture(src.width, src.height, 0);
                accumulationTexture.hideFlags = HideFlags.HideAndDontSave; // 这个变量不会显示在
Hierarchy中，也不会保存到场景中
                Graphics.Blit(src, accumulationTexture); // 帧图像初始化
            }

            // We are accumulating motion over frames without clear/discard
            // by design, so silence any performance warnings from Unity
            accumulationTexture.MarkRestoreExpected(); // 需要进行一个渲染纹理恢复操作，发生在渲染到
纹理而该纹理又没有提前被清空或者销毁的情况下

            material.SetFloat("_BlurAmount", 1.0f - blurAmount);

            Graphics.Blit (src, accumulationTexture, material); // 把当前屏幕图像 src 叠加到
accumulationTexture中
            Graphics.Blit (accumulationTexture, dest); // 把结果显示在屏幕上
        } else {
            Graphics.Blit(src, dest);
```

```
        }
    }
}
```

Shader

```
Shader "Unity Shaders Book/Chapter 12/Motion Blur" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _BlurAmount ("Blur Amount", Float) = 1.0
    }
    SubShader {
        CGINCLUDE // 代码复用

        #include "UnityCG.cginc"

        sampler2D _MainTex;
        fixed _BlurAmount;

        struct v2f {
            float4 pos : SV_POSITION;
            half2 uv : TEXCOORD0;
        };

        v2f vert(appdata_img v) {
            v2f o;

            o.pos = UnityObjectToClipPos(v.vertex);

            o.uv = v.texcoord;

            return o;
        }

        // 定义两个片元着色器 为了维护渲染纹理的透明通道值，使得它不受混合时使用的透明度值影响
        // 更新渲染纹理的RGB通道，设置A通道为_BlurAmount
        fixed4 fragRGB (v2f i) : SV_Target {
            return fixed4(tex2D(_MainTex, i.uv).rgb, _BlurAmount);
        }

        // 更新渲染纹理的A通道部分，直接返回结果
        half4 fragA (v2f i) : SV_Target {
            return tex2D(_MainTex, i.uv);
        }

        ENDCG

        ZTest Always Cull Off ZWrite Off

        Pass {
            Blend SrcAlpha OneMinusSrcAlpha
            ColorMask RGB // 更新RGB通道

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment fragRGB

            ENDCG
        }

        Pass {
            Blend One Zero
```

```
            ColorMask A // 更新A通道

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment fragA

            ENDCG
        }
    }
    FallBack Off
}
```