# 混合(Blending)

# 混合方程

混合是将当前的光栅化像素（也称为源像素）与后台缓冲区中的像素（也称为目标像素）混合（融合）在一起。该技术通常用于渲染半透明物体，比如水和玻璃。

设**C**src为当前正在进行光栅化处理的第*ij*个像素（源像素）的颜色，**C**dst为后台缓冲区中的第*ij*个像素（目标像素）的颜色。当不使用混合时，**C**src会覆盖**C**dst的值（假设该值已通过深度/模板测试）并成为后台缓冲区中的第*ij*个像素的新颜色。但是当使用混合时，**C**src和**C**dst会被组合为一个新颜色**C**并覆盖**C**dst的值（即，混合颜色**C**会成为后台缓冲区中的第*ij*个像素的新颜色）。Direct3D使用如下混合方程混合源像素和目标像素颜色：

**C** = **C**src ⊗ **F**src ⊞ **C**dst ⊗ **F**dst

**F**src（源混合系数）和**F**dst（目标混合系数）可以是9.3节描述的任何一个值，它们按照各种不同的方式调整源像素和目标像素的比例，实现各种不同的混合效果。运算符⊗是定义在5.3.1节中定义的颜色的分量乘法；运算符⊞可以是9.2节描述的任何一个二进制运算符。

上述混合方程只负责处理颜色的RGB分量。alpha分量要由另外一个方程来处理：

*A* = *A*src*F*src ⊞*A*dst*F*dst

该方程与前一个方程基本相同，只是使用的混合系数和二进制运算符有些区别。我们将RGB和alpha分开的原因非常简单，就是为了独立地、按照不同的方式处理它们。

多数情况下，我们只是对RGB分量进行混合。

混合方程中的二进制运算符⊞可取以下各值之一：

typedef enum D3D11\_BLEND\_OP

{

    D3D11\_BLEND\_OP\_ADD = 1,            C = Csrc ⊗ Fsrc + Cdst ⊗ Fdst

    D3D11\_BLEND\_OP\_SUBTRACT = 2,      C = Cdst ⊗ Fdst - Csrc ⊗ Fsrc

    D3D11\_BLEND\_OP\_REV\_SUBTRACT = 3, C = Csrc ⊗ Fsrc - Cdst ⊗ Fdst

    D3D11\_BLEND\_OP\_MIN = 4,            C = min(Csrc , Cdst)

    D3D11\_BLEND\_OP\_MAX = 5,            C = max(Csrc , Cdst)

} D3D11\_BLEND\_OP;

混合系数取自于D3D11\_BLEND枚举类型，以下为基本的几种

    D3D11\_BLEND\_ZERO：F = (0, 0,0)且F = 0

    D3D11\_BLEND\_ONE：F = (1, 1,1) 且F = 1

    D3D11\_BLEND\_SRC\_COLOR：F = (rs ,gs ,bs )

    D3D11\_BLEND\_INV\_SRC\_COLOR：F = (1 - rs ,1 - gs ,1 - bs )

    D3D11\_BLEND\_SRC\_ALPHA：F = (as ,as ,as )且F = as

    D3D11\_BLEND\_INV\_SRC\_ALPHA：F = (1 − as,1 − as,1 − as)且F =1 − as

    D3D11\_BLEND\_DEST\_ALPHA：F = (ad ,ad ,ad )且F = ad

    D3D11\_BLEND\_INV\_DEST\_ALPHA：F = (1 – ad,1 − ad,1 – ad)且F =1 – ad

    D3D11\_BLEND\_DEST\_COLOR：F = (rd ,gd ,bd )

    D3D11\_BLEND\_INV\_DEST\_COLOR：F = (1 – rd ,1 – gd ,1 – bd )

    D3D11\_BLEND\_SRC\_ALPHA\_SAT：F = (asʹ, asʹ, asʹ)且F = asʹ，其中，asʹ = clamp(as , 0, 1)

    D3D11\_BLEND\_BLEND\_FACTOR：F = (r,g,b)且F = a，其中，颜色(r,g,b,a)由ID3D11DeviceContext::OMSetBlendState方法的第2个参数指定（9.4节）。也就是说，我们可以将一个颜色指定为混合系数；在修改混合状态前，该颜色一直有效。

    D3D11\_BLEND\_INV\_BLEND\_FACTOR：F = (1-r,1-g,1-b)且F = 1-a，其中，颜色(r,g,b,a)由ID3D11DeviceContext::OMSetBlendState方法的第2个参数指定（9.4节）。也就是说，我们可以将一个颜色指定为混合系数；在修改混合状态前，该颜色一直有效。

# Blend State

混合的各种参数通过ID3D11BlendState接口来控制。我们可以通过填充一个**D3D11\_BLEND\_DESC**结构体并调用**ID3D11Device::CreateBlendState**方法来创建该接口。

调用**ID3D11DeviceContext::OMSetBlendState**来将混合状态对象绑定到管线的输出合并器阶段。

与其他状态块相同，这里有一个默认的混合状态（禁用混合）；如果以空值来调用OMSetBlendState方法，它就会将混合状态恢复为默认值。注意，混合会在每个像素上执行额外的计算工作，所以我们只有在用到混合时才启用它，用完之后应该立即关闭。

Example:

|  |
| --- |
| CD3D11\_BLEND\_DESC bd(D3D11\_DEFAULT);  bd.RenderTarget[0] =  {  TRUE,  D3D11\_BLEND\_SRC\_ALPHA, D3D11\_BLEND\_INV\_SRC\_ALPHA, D3D11\_BLEND\_OP\_ADD,  D3D11\_BLEND\_ONE, D3D11\_BLEND\_ZERO, D3D11\_BLEND\_OP\_ADD,  D3D11\_COLOR\_WRITE\_ENABLE\_ALL  };  ComPtr<ID3D11BlendState> mBlendState;  m\_d3dDevice->CreateBlendState(&bd, mBlendState.GetAddressOf());  float blendFactors[] = { 0.f,0.f,0.f,0.f };  m\_d3dContext->OMSetBlendState(mBlendState.Get(), blendFactors, UINT\_MAX); |

# 使用场景

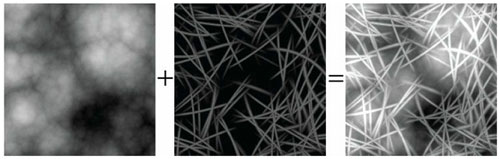
**屏蔽颜色写入**

当我们希望屏蔽后台缓冲区、只向深度/模板缓冲区写入数据时，该算法非常有用。要实现这一算法，可将源像素混合系数设为**D3D11\_BLEND\_ZERO**，目标混合系数设为**D3D11\_BLEND\_ONE**，混合运算符设为**D3D11\_BLEND\_OP\_ADD**。

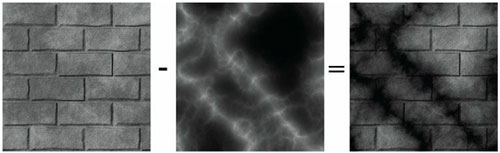
还有一种方法可以实现相同的结果，就是将**D3D11\_RENDER\_TARGET\_BLEND\_DESC::RenderTargetWriteMask**设置为0，这样就不会写入任何颜色通道。

**加法和减法**

假设我们希望将源像素和目标像素相加。要实现这一算法，可将源混合系数设为**D3D11\_BLEND\_ONE**，目标混合系数设为**D3D11\_BLEND\_ONE**，混合运算符设为**D3D11\_BLEND\_OP\_ADD**。



当把**D3D11\_BLEND\_OP\_ADD**替换为**D3D11\_BLEND\_OP\_SUBTRACT**或**D3D11\_BLEND\_OP\_REV\_SUBTRACT**时，可以实现源像素和目标像素的减法运算。



**乘法**

假设我们要将源像素和目标像素相乘。要实现这一算法，可将源混合系数设为**D3D11\_BLEND\_ZERO**，目标混合系数设为**D3D11\_BLEND\_SRC\_COLOR**，混合运算符设为**D3D11\_BLEND\_OP\_ADD**。



**透明度**

们可以使用源alpha分量as控制源像素的不透明度。例如，当alpha为0.0时表示0%不透明；为0.4时表示40%不透明，既60%透明；为1.0时表示100%不透明。

要实现一算法，可将源混合系数设为**D3D11\_BLEND\_SRC\_ALPHA**，目标混合系数设为**D3D11\_BLEND\_INV\_SRC\_ALPHA**，混合运算符设为**D3D11\_BLEND\_OP\_ADD**。

此时混合方程为C = asCsrc + (1 - as)Cdst

通过这一混合方式，我们可以绘制透明物体。

但需要注意的是，此时物体的绘制顺序非常重要。**我们需要首先绘制非透明物体，然后再绘制透明物体。**

**混合与深度缓冲**

当使用加/减/乘法混合时会出现一个与深度测试相关的问题。这里，我们仅以加法混合为例进行说明，同样的情况也适用于减法和乘法混合。

当我们使用加法混合来渲染一个粒子系统S时，每个粒子是否相互遮挡并不重要；我们只需要把粒子的颜色简单地累加起来 。我们并不希望对S中的每个粒子进行深度测试。在这一情景中，如果不按照从后向前的顺序绘图，那么当S中的某个粒子被另一个粒子遮挡时，该粒子将无法通过深度测试，它的像素片段将被丢弃，也就是说该粒子的像素颜色不会累加到最终的混合颜色中。

我们应该在渲染S时禁用深度写入功能，使粒子的深度信息不写入深度缓冲区。在禁用深度写入功能之后，由加法混合生成的粒子深度信息不会写入到深度缓冲区；所以，当S中的一个粒子被其他粒子遮挡时，该粒子依然可以通过深度测试并绘制到后台缓冲区中。注意，我们只在绘制S时禁用深度写入功能（以便于使用加法混合绘制该粒子系统），而深度读取和深度测试功能依然有效。非透明物体（在绘制透明物体之前）仍然可以遮挡位于它后面的透明物体。例如，当你在一堵墙的后面绘制一个透明物体时，该物体不会显示出来，因为它被墙体挡住了。

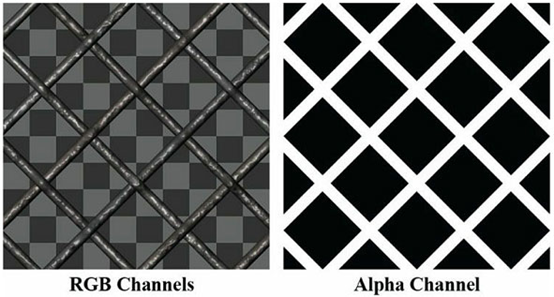
例如如下的火焰，中心的亮度高于边缘，因为中间粒子较多，它们彼此重叠并累加在一起。



# 裁剪像素

有时，我们希望完全丢弃某个源像素，使它不再接受后续处理。这一工作可以由HLSL的内置函数clip(x)来实现。该函数只能在像素着色器中使用，当x<0时丢弃当前像素，使之不再接受后续处理。

我们可以在像素着色器中，获取漫反射纹理的alpha分量。当它的值接近于0时，我们将该像素视为完全透明，则可以丢弃该像素，不再对它进行后续处理。



注意：由于过滤器的作用，alpha通道可能会变得有些模糊，所以当裁剪像素时，应该保留一些容差值。例如，裁剪alpha值接近于0的像素，而不必让alpha值精确为0。

Example:

|  |
| --- |
| float4 sampledDiffuse = gDiffuseMap.Sample(gSampler, pin.TexUV);    #ifdef ALPHA\_TEST  // Discard pixel if texture alpha < 0.1. We do this test as soon  // as possible in the shader so that we can potentially exit the  // shader early, thereby skipping the rest of the shader code.  clip(sampledDiffuse.a - 0.1f);  #endif |

# 雾

当我们在游戏中模拟某些类型的天气状况时，可能会用到雾效。

雾除了本身所具有的用途外，还具有一些附加效用。例如，雾可以用来掩盖渲染过程中出现的不自然的人工痕迹，避免蹿出问题的发生。蹿出（popping）是指由于摄像机的移动，使原本在远平面后面的物体突然进入平截头体内，从不可见变为可见；这看上去就像是突然“蹿”到场景里面一样。通过在一定距离内加入雾效，可以掩盖这一问题。

即使你的场景是在晴朗的白天，你也可以在较远的地方加入一些淡淡的雾气。因为就算是晴天，远处的物体（比如山岳）也会看上去有些模糊，就像是有一层薄薄的雾笼罩在上面一样，当深度增加时，物体的对比度会逐渐减小。我们可以用雾来模拟这种大气透视现象。

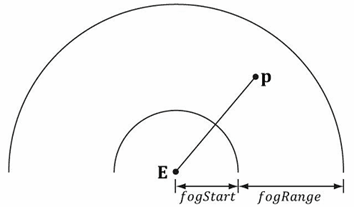
我们用如下方法来实现雾效：我们为雾指定一个颜色、一个相对于摄像机的起始位置和一个范围（即，该范围从雾的起始位置开始到完全遮隐任何物体为止）。那么，三角形表面点的颜色等于点的照颜色与雾颜色的加权平均值：

*foggedColor* = *litColor* + *s* (*fogColor* − *litColor*) = (1 − *s* ) ∙ *litColor* + *s* ∙ *fogColor*

参数s的取值范围是从0到1，它是一个以表面点和观察点之间的距离为自变量的函数。随着表面点和观察点之间的距离增大，雾在表面点颜色中所占的比例会越来越大。该参数的定义如下：

_scroll_external/attachments/image2021-1-6_18-21-53-81e89aa7f7c7486b499705260026385c27eeca5328cf24e97a000eca7ff276f5.png

其中，dist(p,E)是表面点p和观察点E之间的距离。saturate函数将自变量限定在[0,1]区间内。



Example:

|  |
| --- |
| // Vector from point being lit to eye.  float3 toEyeW = gEyePosW - pin.PosW;  float distToEye = length(toEyeW);  toEyeW /= distToEye; // normalize  ...  // Blend the fog color and the lit color.  float fogAmount = saturate((distToEye - gFogStart) / gFogRange);  litColor = lerp(litColor, gFogColor, fogAmount); |

在雾效计算中，我们使用了distToEye，这个值还用来normalize toEyeW矢量。下面的代码也可以达成这样的效果，但是会计算两次矢量的长度，一次在**normalize**函数中，一次在**distance**函数中，不够优化。

|  |
| --- |
| float3 toEye = normalize(gEyePosW - pin.PosW);  float distToEye = distance(gEyePosW, pin.PosW); |