备忘录_混合

笔记本: DirectX 12
创建时间: 2022/8/21 13:13 **更新时间:** 2022/8/21 15:13
作者: handsome小赞

- 混合：使我们可以将当前要光栅化的像素（源像素）与之前已光栅化至后台缓冲区的像素（目标像素）相融合。该技术可用于渲染如水与玻璃之类的半透明物体
- 传统混合方程：

> C_{src} ：像素着色器输出的当前正在光栅化的第 i 行、第 j 列像素（源像素）的颜色值
> C_{dst} ：目前在后台缓冲区中与之对应的第 i 行、第 j 列像素（目标像素）的颜色值
> F_{src} ：源混合因子
> F_{dst} ：目标混合因子
>
> RGB分量 : $\boldsymbol{C} = \boldsymbol{C}_{\text{src}} \otimes \boldsymbol{F}_{\text{src}} \boxplus \boldsymbol{C}_{\text{dst}} \otimes \boldsymbol{F}_{\text{dst}}$
>
> Alpha分量 : $\boldsymbol{A} = \boldsymbol{A}_{\text{src}} \boldsymbol{F}_{\text{src}} \boxplus \boldsymbol{A}_{\text{dst}} \boldsymbol{F}_{\text{dst}}$
>
> 下列枚举项成员将用作混合方程中的二元运算符 ⊞：
>
> ```
> typedef enum D3D12_BLEND_OP
> {
>   D3D12_BLEND_OP_ADD = 1,
>   D3D12_BLEND_OP_SUBTRACT = 2,
>   D3D12_BLEND_OP_REV_SUBTRACT = 3,
>   D3D12_BLEND_OP_MIN = 4,
>   D3D12_BLEND_OP_MAX = 5,
> } D3D12_BLEND_OP;
> ```
>
> $\boldsymbol{C} = \boldsymbol{C}_{\text{src}} \otimes \boldsymbol{F}_{\text{src}} + \boldsymbol{C}_{dst} \otimes \mathbf{F}_{dst}$
> $\boldsymbol{C} = \boldsymbol{C}_{\text{dst}} \otimes \boldsymbol{F}_{\text{dst}} - \boldsymbol{C}_{\text{src}} \otimes \boldsymbol{F}_{\text{src}}$
> $\boldsymbol{C} = \boldsymbol{C}_{\text{src}} \otimes \boldsymbol{F}_{\text{src}} - \boldsymbol{C}_{\text{dst}} \otimes \boldsymbol{F}_{\text{dst}}$
> $\boldsymbol{C} = \min(\boldsymbol{C}_{\text{src}}, \boldsymbol{C}_{\text{dst}})$
> $\boldsymbol{C} = \max(\boldsymbol{C}_{\text{src}}, \boldsymbol{C}_{\text{dst}})$

- 新增逻辑运算符，与传统混合方程不可同时使用，且逻辑运算符混合技术只支持 UINT（无符号整数）有关类型的渲染目标格式。
- 混合因子：

> 参考 D3D12_BLEND
> 以下仅列举数个

D3D12_BLEND_ZERO: $F = (0, 0, 0)$且 $F = 0$

D3D12_BLEND_ONE: $F = (1, 1, 1)$且 $F = 1$

D3D12_BLEND_SRC_COLOR: $F = (r_s, g_s, b_s)$

D3D12_BLEND_INV_SRC_COLOR: $F_{src} = (1 - r_s, 1 - g_s, 1 - b_s)$

D3D12_BLEND_SRC_ALPHA: $F = (a_s, a_s, a_s)$且 $F = a_s$

D3D12_BLEND_INV_SRC_ALPHA: $F = (1 - a_s, 1 - a_s, 1 - a_s)$且 $F = (1 - a_s)$

D3D12_BLEND_DEST_ALPHA: $F = (a_d, a_d, a_d)$且 $F = a_d$

D3D12_BLEND_INV_DEST_ALPHA: $F = (1 - a_d, 1 - a_d, 1 - a_d)$且 $F = (1 - a_d)$

D3D12_BLEND_DEST_COLOR: $F = (r_d, g_d, b_d)$

D3D12_BLEND_INV_DEST_COLOR: $F = (1 - r_d, 1 - g_d, 1 - b_d)$

D3D12_BLEND_SRC_ALPHA_SAT: $F = (a'_s, a'_s, a'_s)$ 且 $F = a'_s$，其中 $a'_s = \text{clamp}(a_s, 0, 1)$

特别注意，其中的颜色（r,g,b,a）可用作
ID3D12GraphicsCommandList::OMSetBlendFactor 的参数
这可以使我们直接指定所用的混合因子
但是在改变混合状态前，此值不会生效
对于alpha混合方程，不可用 _COLOR 作结尾的混合因子
clamp(x,a,b)，将x限制在 [a,b]

- 混合状态

混合状态也是PSO的一部分
transparentPsoDesc.BlendState
如同其他PSO一般，我们应该在应用程序初始化期间来创建它们，再根据需求以 ID3D12GraphicsCommandList::SetPipelineState 方法在不同的状态间切换

- **D3D12_BLEND_DESC**
  {
  BOOL AlphaToCoverageEnable ,
  BOOL IndependentBlendEnable ,
  D3D12_RENDER_TARGET_BLEND_DESC RenderTarget[8] ;
  }

1. AlphaToCoverageEnable：指定为 true，则启用 alpha-to-coverage 功能，这是一种在渲染叶片或门等纹理时极其有用的一种多重采样技术。若指定为 false，则禁用 alpha-to-coverage 功能。另外，要使用此技术还需开启多重采样（即创建后台缓冲区与深度缓冲区时要启用多重采样）。
2. IndependentBlendEnable：Direct3D 最多可同时支持 8 个渲染目标。若此标志被置为 true，即表明可以向每一个渲染目标执行不同的混合操作（不同的混合因子、不同的混合运算以及设置不同的混合禁用或开启状态等）。如果将此标志设为 false，则意味着所有的渲染目标均使用 D3D12_BLEND_DESC::RenderTarget 数组中第一个元素所描述的方式进行混合。多渲染目标技术常用于高级算法，而现在我们只假设每次仅向一个渲染目标进行绘制。
3. RenderTarget：具有 8 个 D3D12_RENDER_TARGET_BLEND_DESC 元素的数组，其中的第 i 个元素描述了如何针对第 i 个渲染目标进行混合处理。如果 IndependentBlendEnable 被设置为 false，则所有的渲染目标都将根据 RenderTarget[0]的设置进行混合运算。

- **D3D12_RENDER_TARGET_BLEND_DESC**

```
typedef struct D3D12_RENDER_TARGET_BLEND_DESC
{
  BOOL BlendEnable; // 默认值为 False
  BOOL LogicOpEnable; // 默认值为 False
  D3D12_BLEND SrcBlend; // 默认值为 D3D12_BLEND_ONE
  D3D12_BLEND DestBlend; // 默认值为 D3D12_BLEND_ZERO
  D3D12_BLEND_OP BlendOp; // 默认值为 D3D12_BLEND_OP_ADD
  D3D12_BLEND SrcBlendAlpha; // 默认值为 D3D12_BLEND_ONE
  D3D12_BLEND DestBlendAlpha; // 默认值为 D3D12_BLEND_ZERO
  D3D12_BLEND_OP BlendOpAlpha; // 默认值为 D3D12_BLEND_OP_ADD
  D3D12_LOGIC_OP LogicOp;    // 默认值为 D3D12_LOGIC_OP_NOOP
  UINT8 RenderTargetWriteMask; // 默认值为 D3D12_COLOR_WRITE_ENABLE_ALL
} D3D12_RENDER_TARGET_BLEND_DESC;
```

1. BlendEnable：指定为 true，则启用常规混合功能；指定为 false，则禁用常规混合功能。注意，不能将 BlendEnable 与 LogicOpEnable 同时置为 true，只能从常规混合与逻辑运算符混合两种方式中选择一种。
2. LogicOpEnable：指定为 true，则启用逻辑混合运算，反之则反。注意，不能将 BlendEnable 和 LogicOpEnable 同时设置为 true，只能从常规混合与逻辑运算混合中选择一种。
3. SrcBlend：枚举类型 D3D12_BLEND 中的成员之一，用于指定 RGB 混合中的源混合因子 $\mathbf{F}_{src}$。
4. DestBlend：枚举类型 D3D12_BLEND 中的成员之一，用于指定 RGB 混合中的目标混合因子 $\mathbf{F}_{dst}$。
5. BlendOp：枚举类型 D3D12_BLEND_OP 中的成员之一，用于指定 RGB 混合运算符。
6. SrcBlendAlpha：枚举类型 D3D12_BLEND 中的一个成员，指定了 alpha 混合中的源混合因子 $\mathbf{F}_{src}$。
7. DestBlendAlpha：枚举类型 D3D12_BLEND 中的一个成员，指定了 alpha 混合中的目标混合因子 $\mathbf{F}_{dst}$。
8. BlendOpAlpha：枚举类型 D3D12_BLEND_OP 中的一个成员，指定了 alpha 混合运算符。
9. LogicOp：枚举类型 D3D12_LOGIC_OP 中的成员之一，指定了源颜色与目标颜色在混合时所用的逻辑运算符。
10. RenderTargetWriteMask：下列标志中一种或多种的组合。

```
typedef enum D3D12_COLOR_WRITE_ENABLE {
 D3D12_COLOR_WRITE_ENABLE_RED   = 1,
 D3D12_COLOR_WRITE_ENABLE_GREEN  = 2,
 D3D12_COLOR_WRITE_ENABLE_BLUE  = 4,
 D3D12_COLOR_WRITE_ENABLE_ALPHA  = 8,
 D3D12_COLOR_WRITE_ENABLE_ALL   =
   ( D3D12_COLOR_WRITE_ENABLE_RED | D3D12_COLOR_WRITE_ENABLE_GREEN |
    D3D12_COLOR_WRITE_ENABLE_BLUE | D3D12_COLOR_WRITE_ENABLE_ALPHA )
} D3D12_COLOR_WRITE_ENABLE;
```

这些标志控制着混合后的数据可被写入后台缓冲区中的哪些颜色通道。例如，通过指定 D3D12_COLOR_WRITE_ENABLE_ALPHA 可以禁止向 RGB 通道的写操作，而仅写入 alpha 通道的有关数据。对于一些高级技术而言，这种灵活性是极其实用的。当混合功能被禁止时，从像素着色器返回的颜色数据将按没有设置上述写掩码来进行处理（即不对目标像素执行任何操作）。

- 裁剪像素

使用HLSL内置函数 clip(x) 来禁止某个源像素参与后续的处理
当x<0时当前像素将从后面的处理阶段中丢弃

```
float4 PS(VertexOut pin) : SV_Target
{
  float4 diffuseAlbedo = gDiffuseMap.Sample(
    gsamAnisotropicWrap, pin.TexC) * gDiffuseAlbedo;



#ifdef ALPHA_TEST
  // 若 alpha < 0.1 则抛弃该像素。我们要在着色器中尽早执行此项测试，以尽快检测出满足条件的像素
  // 并退出着色器，从而跳过后续的相关处理过程
  clip(diffuseAlbedo.a - 0.1f);
#endif

  ...

  // 从漫反射反照率获取 alpha 值的常用手段
  litColor.a = diffuseAlbedo.a;

  return litColor;
}
```

- 由图可见，只有在定义了 ALPHA_TEST宏 的时候才会实行透明像素筛选。因为我们有时并不希望对某些渲染项执行 clip 方法
- 注意，为着色器定义宏 需要在编译着色器时指定 D3D_SHADER_MACRO

```
const D3D_SHADER_MACRO defines[] =
{
    "FOG", "1",
    NULL, NULL
};

const D3D_SHADER_MACRO alphaTestDefines[] =
{
    "FOG", "1",
    "ALPHA_TEST", "1",
    NULL, NULL
};

mShaders["standardVS"] = d3dUtil::CompileShader(L"Shaders\\Default.hlsl", nullptr, "VS", "vs_5_0");
mShaders["opaquePS"] = d3dUtil::CompileShader(L"Shaders\\Default.hlsl", defines, "PS", "ps_5_0");
mShaders["alphaTestedPS"] = d3dUtil::CompileShader(L"Shaders\\Default.hlsl", alphaTestDefines, "PS", "ps_5_0");
```

- 注意，通过混合操作也能实现相同效果，但是使用 clip 函数更为有效。针对 ALPHA_TEST 的物体禁用背面剔除，否则会穿帮

```
// 针对 alpha 测试物体所采用的 PSO
D3D12_GRAPHICS_PIPELINE_STATE_DESC alphaTestedPsoDesc = opaquePsoDesc;
alphaTestedPsoDesc.PS =
{
  reinterpret_cast<BYTE*>(mShaders["alphaTestedPS"]->GetBufferPointer()),
    mShaders["alphaTestedPS"]->GetBufferSize()
};
alphaTestedPsoDesc.RasterizerState.CullMode = D3D12_CULL_MODE_NONE;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(
  &alphaTestedPsoDesc, IID_PPV_ARGS(&mPSOs["alphaTested"])));
```

- 雾

可以使用雾效来模拟各种气象环境效果和大气透视现象，以此来掩饰远处场景渲染的失真以及物体突然出现于视椎体之中闯进用户视野内的情况。在我们所用的线性雾效模型中，需要指定雾气的颜色，从摄像机至雾效的最近距离以及雾的出现范围。此时，网格三角形上某点的颜色是其原色与雾色的加权平均值：

$$\begin{aligned}\texttt{foggedColor} &= \texttt{litColor} + \texttt{s}(\texttt{fogColor} - \texttt{litColor}) \\ &= (1-\texttt{s})\cdot\texttt{litColor} + \texttt{s}\cdot\texttt{fogColor}\end{aligned}$$

参数 s 的范围为[0, 1]，由一个以摄像机位置与被覆盖物体表面点之间距离作为参数的函数来确定。随着该表面点与观察点之间距离的增加，它会被雾气遮挡得愈加朦胧。参数 s 的定义如下：

$$s = \text{saturate}\left(\frac{\text{dist}(\boldsymbol{p},\boldsymbol{E}) - \text{fogStart}}{\text{fogRange}}\right)$$

其中，$\text{dist}(\boldsymbol{p}, \boldsymbol{E})$为表面点 $\boldsymbol{p}$ 与摄像机位置 $\boldsymbol{E}$ 之间的距离。而函数 `saturate` 会将其参数限制在区间[0, 1]内：

$$\text{saturate}(x) = \begin{cases} x, 0 \leqslant x \leqslant 1 \\ 0, x < 0 \\ 1, x > 1 \end{cases}$$

当 (p,E)<=fogStart 时，s = 0
foggedColor = litColor
当 (p,E)>=fogStatr 时，s = 1
foggedColor = fogColor

- **HLSL**:

```
#ifdef FOG
  float fogAmount = saturate((distToEye - gFogStart) / gFogRange);
  litColor = lerp(litColor, gFogColor, fogAmount);
#endif
```