Week6

Blending & Stenciling

Hooman Salamat

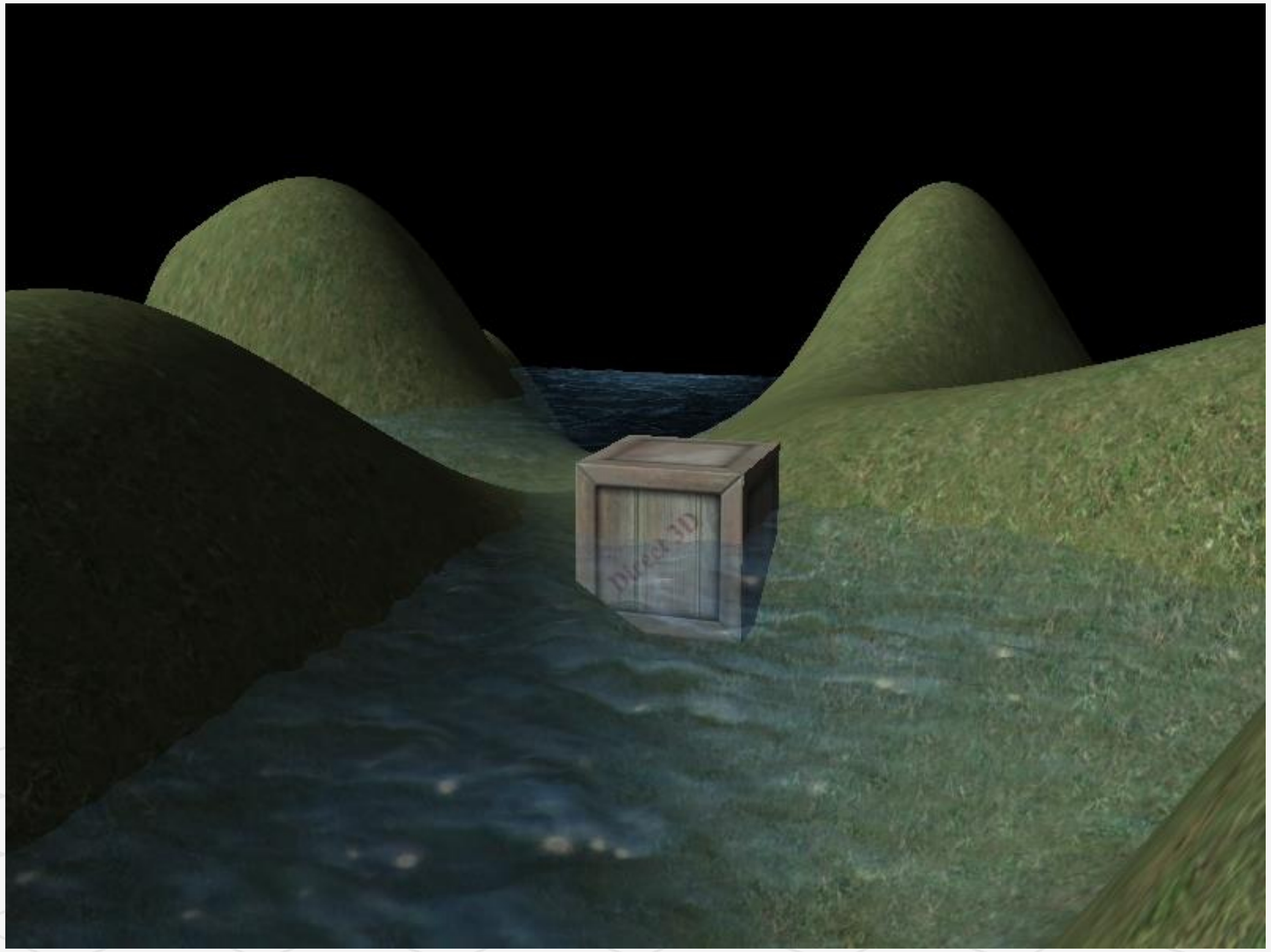
Objectives

- Understand how blending works, it’s different modes, and how to use it with Direct3D
- Prevent a pixel from being drawn to the back buffer altogether by employing the HLSL clip function
- Find out how to control the depth and stencil buffer state by filling out the D3D12_DEPTH_STENCIL_DESC field in a pipeline state object
- Implement mirrors by using the stencil buffer to prevent reflections from being drawn to non-mirror surfaces
- Identify double blending and understand how the stencil buffer can prevent it
- Explain depth complexity and describe two ways the depth complexity of a scene can to measured

Blending

- We start rendering the frame by first drawing the terrain followed by the wooden crate, so that the terrain and crate pixels are on the back buffer

- We then draw the water surface to the back buffer using blending, so that the water pixels get blended with the terrain and crate pixels on back buffer in such a way that the terrain and crate shows through the water



Blending Equation

- Let C_{src} be the color output from the pixel shader for the *ij*th pixel we are currently rasterizing (source pixel)
- Let C_{dst} be the color of the *ij*th pixel currently on the back buffer (destination pixel)
- Without blending, C_{src} would overwrite C_{dst} (assuming it passes the depth/stencil test) and become the new color of the *ij*th back buffer pixel
- With blending, C_{src} and C_{dst} are blended together to get the new color C that will overwrite C_{dst}

The colors F_{src} (source blend factor) and F_{dst} (destination blend factor) allow us modify the original source and destination pixels in a variety of ways, allowing for different effects to be achieved

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

The $\otimes$ operator means component wise multiplication for color vectors; the binary operator may be any of the following operators

```
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;
```

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} + \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$\mathbf{C} = \mathbf{C}_{dst} \otimes \mathbf{F}_{dst} - \mathbf{C}_{src} \otimes \mathbf{F}_{src}$$

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} - \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$\mathbf{C} = \min\left(\mathbf{C}_{src}, \mathbf{C}_{dst}\right)$$

$$\mathbf{C} = \max\left(\mathbf{C}_{src}, \mathbf{C}_{dst}\right)$$

The Alpha Component

The blending equation holds only for the RGB components of the colors. The alpha component is actually handled by a separate similar equation:

$$A = A_{src} F_{src} \boxplus A_{dst} F_{dst}$$

The equation is essentially the same, but it is possible that the blend factors and binary operation are different.

For example, it is possible to add the two RGB terms, but subtract the two alpha terms:

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} + \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$A = A_{dst} F_{dst} - A_{src} F_{src}$$

Blending the alpha component is needed much less frequently than blending the RGB components.

We do not care about the back buffer alpha values. Back buffer alpha values are only important if you have some algorithm that requires destination alpha values.

Logic operators

A feature recently added to Direct3D is the ability to blend the source color and destination color using a logic operator

you cannot use traditional blending and logic operator blending at the same time; you pick one or the other. Note also that in order to use logic operator blending the render target format must support—it should be a format of the UINT variety, otherwise you will get errors like the following:

D3D12 ERROR: ID3D12Device::CreateGraphicsPipelineState: The render target format at slot 0 is format (R8G8B8A8_UNORM).

```
enum D3D12_LOGIC_OP
    {
        D3D12_LOGIC_OP_CLEAR= 0,
        D3D12_LOGIC_OP_SET= ( D3D12_LOGIC_OP_CLEAR + 1 ) ,
        D3D12_LOGIC_OP_COPY= ( D3D12_LOGIC_OP_SET + 1 ) ,
        D3D12_LOGIC_OP_COPY_INVERTED= ( D3D12_LOGIC_OP_COPY + 1 ) ,
        D3D12_LOGIC_OP_NOOP= ( D3D12_LOGIC_OP_COPY_INVERTED + 1 ) ,
        D3D12_LOGIC_OP_INVERT= ( D3D12_LOGIC_OP_NOOP + 1 ) ,
        D3D12_LOGIC_OP_AND= ( D3D12_LOGIC_OP_INVERT + 1 ) ,
        D3D12_LOGIC_OP_NAND= ( D3D12_LOGIC_OP_AND + 1 ) ,
        D3D12_LOGIC_OP_OR= ( D3D12_LOGIC_OP_NAND + 1 ) ,
        D3D12_LOGIC_OP_NOR= ( D3D12_LOGIC_OP_OR + 1 ) ,
        D3D12_LOGIC_OP_XOR= ( D3D12_LOGIC_OP_NOR + 1 ) ,
        D3D12_LOGIC_OP_EQUIV= ( D3D12_LOGIC_OP_XOR + 1 ) ,
        D3D12_LOGIC_OP_AND_REVERSE= ( D3D12_LOGIC_OP_EQUIV + 1 ) ,
        D3D12_LOGIC_OP_AND_INVERTED= ( D3D12_LOGIC_OP_AND_REVERSE + 1 ) ,
        D3D12_LOGIC_OP_OR_REVERSE= ( D3D12_LOGIC_OP_AND_INVERTED + 1 ) ,
        D3D12_LOGIC_OP_OR_INVERTED= ( D3D12_LOGIC_OP_OR_REVERSE + 1 )
    } D3D12_LOGIC_OP;
```

BLEND FACTORS

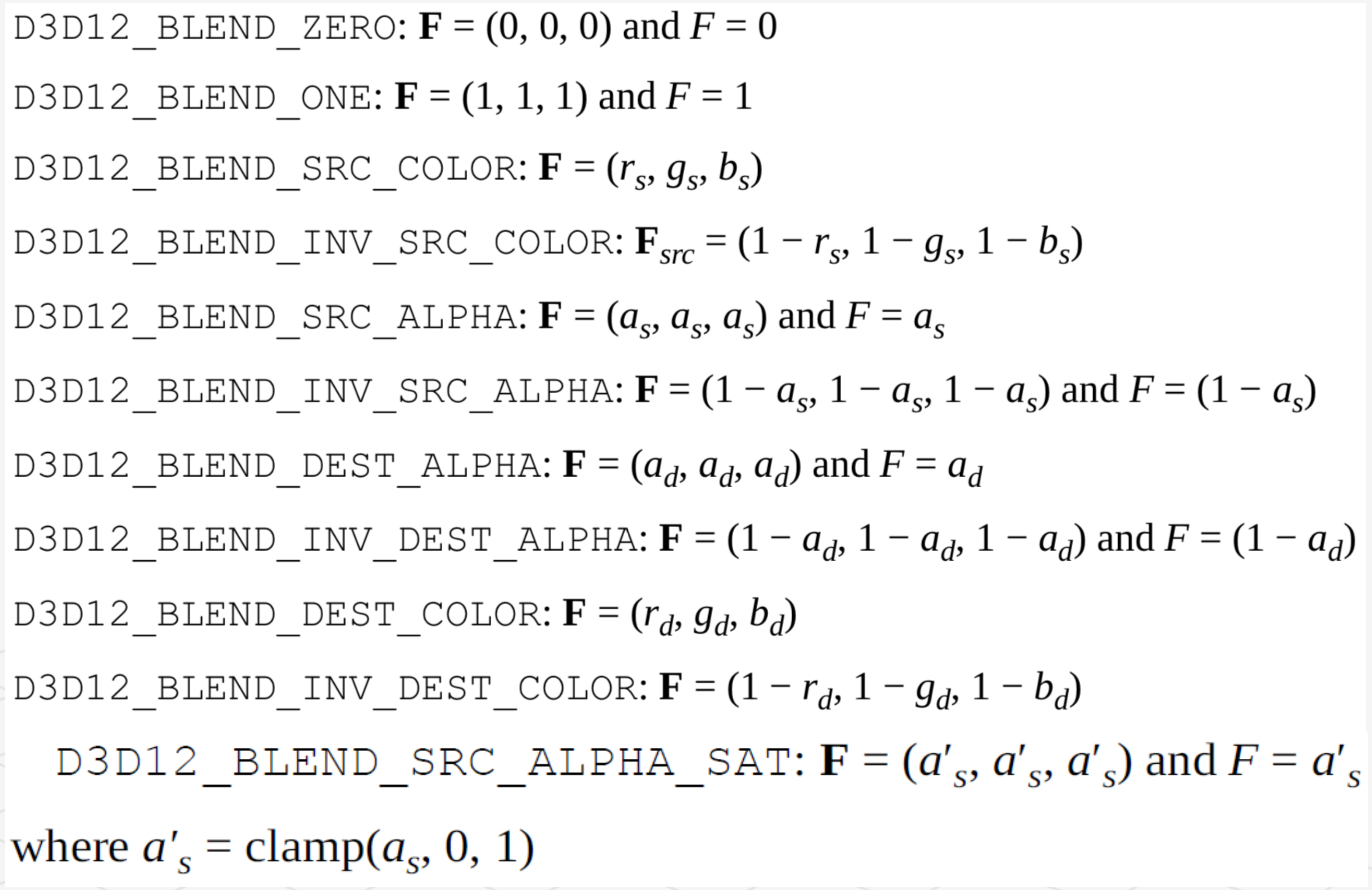
The following list describes the basic blend factors, which apply to both **F***src* and **F***dst*.

$\mathbf{C}_{src} = (r_s, g_s, b_s)$, $A_{src} = a_s$

(the RGBA values output from the pixel shader),

$\mathbf{C}_{dst} = (r_d, g_d, b_d)$, $A_{dst} = a_d$

(the RGBA values already stored in the render target), **F** being either $\mathbf{F}_{src}$ or $\mathbf{F}_{dst}$ and *F* being either F_{src} or F_{dst}, we have:



`D3D12_BLEND_ZERO`: $\mathbf{F} = (0, 0, 0)$ and $F = 0$

`D3D12_BLEND_ONE`: $\mathbf{F} = (1, 1, 1)$ and $F = 1$

`D3D12_BLEND_SRC_COLOR`: $\mathbf{F} = (r_s, g_s, b_s)$

`D3D12_BLEND_INV_SRC_COLOR`: $\mathbf{F}_{src} = (1 - r_s, 1 - g_s, 1 - b_s)$

`D3D12_BLEND_SRC_ALPHA`: $\mathbf{F} = (a_s, a_s, a_s)$ and $F = a_s$

`D3D12_BLEND_INV_SRC_ALPHA`: $\mathbf{F} = (1 - a_s, 1 - a_s, 1 - a_s)$ and $F = (1 - a_s)$

`D3D12_BLEND_DEST_ALPHA`: $\mathbf{F} = (a_d, a_d, a_d)$ and $F = a_d$

`D3D12_BLEND_INV_DEST_ALPHA`: $\mathbf{F} = (1 - a_d, 1 - a_d, 1 - a_d)$ and $F = (1 - a_d)$

`D3D12_BLEND_DEST_COLOR`: $\mathbf{F} = (r_d, g_d, b_d)$

`D3D12_BLEND_INV_DEST_COLOR`: $\mathbf{F} = (1 - r_d, 1 - g_d, 1 - b_d)$

`D3D12_BLEND_SRC_ALPHA_SAT`: $\mathbf{F} = (a'_s, a'_s, a'_s)$ and $F = a'_s$

where $a'_s = \text{clamp}(a_s, 0, 1)$

ID3D12GraphicsCommandList::OMSetBlendFactor

Sets the blend factor that modulate values for a pixel shader, render target, or both.

void OMSetBlendFactor(

//Array of blend factors, one for each RGBA component.

const FLOAT [4] BlendFactor

);

Passing a nullptr *restores the default blend factor of (1, 1, 1, 1).*

D3D12_BLEND_BLEND_FACTOR: **F** = (*r*, *g*, *b*) and *F* = *a*, where the color (*r*, *g*, *b*, *a*) is supplied to the second parameter

BLEND STATE

Where do we set blending operators and blend factors?

The blend state is part of the PSO.

To configure a non-default blend state we must fill out a D3D12_BLEND_DESC structure. The D3D12_BLEND_DESC structure is defined like so:

```
typedef struct D3D12_BLEND_DESC

    {

    BOOL AlphaToCoverageEnable;

    BOOL IndependentBlendEnable;

    D3D12_RENDER_TARGET_BLEND_DESC RenderTarget[ 8 ];

    } D3D12_BLEND_DESC;
```

```
void BlendApp::BuildPSOs()
{
    D3D12_GRAPHICS_PIPELINE_STATE_DESC opaquePsoDesc;

ZeroMemory(&opaquePsoDesc, sizeof(D3D12_GRAPHICS_PIPELINE_STATE_DESC));
………….rest of the code …..
    ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&opaquePsoDesc,
IID_PPV_ARGS(&mPSOs["opaque"])));


D3D12_GRAPHICS_PIPELINE_STATE_DESC transparentPsoDesc = opaquePsoDesc;
D3D12_GRAPHICS_PIPELINE_STATE_DESC transparentPsoDesc = opaquePsoDesc;

D3D12_RENDER_TARGET_BLEND_DESC transparencyBlendDesc;
transparencyBlendDesc.BlendEnable = true;
transparencyBlendDesc.LogicOpEnable = false;
transparencyBlendDesc.SrcBlend = D3D12_BLEND_SRC_ALPHA;
transparencyBlendDesc.DestBlend = D3D12_BLEND_INV_SRC_ALPHA;
transparencyBlendDesc.BlendOp = D3D12_BLEND_OP_ADD;
transparencyBlendDesc.SrcBlendAlpha = D3D12_BLEND_ONE;
transparencyBlendDesc.DestBlendAlpha = D3D12_BLEND_ZERO;
transparencyBlendDesc.BlendOpAlpha = D3D12_BLEND_OP_ADD;
transparencyBlendDesc.LogicOp = D3D12_LOGIC_OP_NOOP;
transparencyBlendDesc.RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;

transparentPsoDesc.BlendState.RenderTarget[0] = transparencyBlendDesc;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&transparentPsoDesc,
IID_PPV_ARGS(&mPSOs["transparent"])));
```

No Color Write

Suppose that we want to keep the original destination pixel exactly as it is and not overwrite it or blend it with the source pixel currently being rasterized. This can be useful, for example, if you just want to write to the depth/stencil buffer, and not the back buffer.

To do this:

the source pixel blend factor: D3D12_BLEND_ZERO,

the destination blend factor: D3D12_BLEND_ONE,

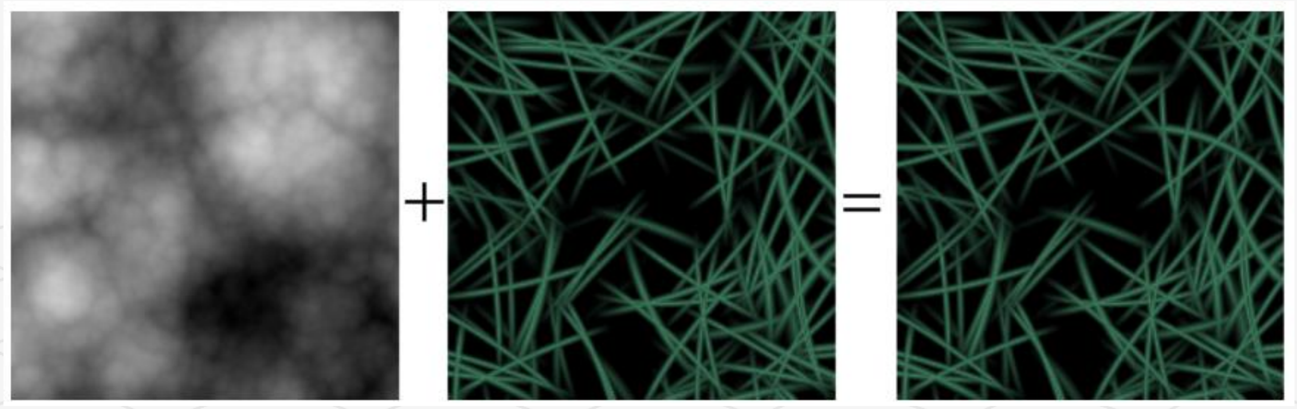
the blend operator: D3D12_BLEND_OP_ADD.

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$\mathbf{C} = \mathbf{C}_{src} \otimes (0,0,0) + \mathbf{C}_{dst} \otimes (1,1,1)$$

$$\mathbf{C} = \mathbf{C}_{dst}$$

another way to implement the same thing would be to set the D3D12_RENDER_TARGET_BLEND_DESC::RenderTargetWriteMask member to 0, so that none of the color channels are written to.



Adding/Subtracting

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$\mathbf{C} = \mathbf{C}_{src} \otimes (1,1,1) + \mathbf{C}_{dst} \otimes (1,1,1)$$

$$\mathbf{C} = \mathbf{C}_{src} + \mathbf{C}_{dst}$$

Suppose that we want to add the source pixels with the destination pixels.

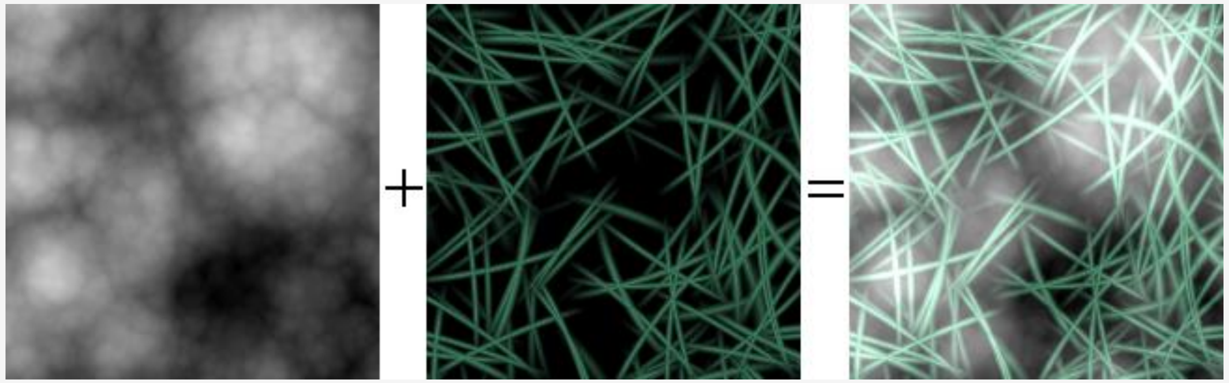
To do this,

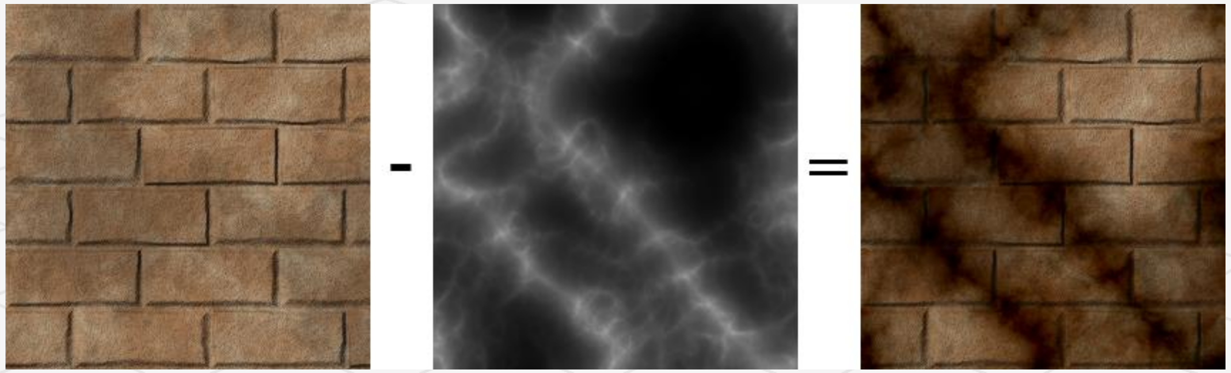
source blend factor: D3D12_BLEND_ONE, destination blend factor = D3D12_BLEND_ONE,

blend operator to D3D12_BLEND_OP_ADD.

The following adds source and destination color. Adding creates a brighter image since color is being added.

We can subtract source pixels from destination pixels by using the above blend factors and replacing the blend operation with D3D12_BLEND_OP_SUBTRACT





Multiplying

Suppose that we want to multiply a source pixel with its corresponding destination pixel. To do this, we set

source blend factor: D3D12_BLEND_ZERO,

destination blend factor: D3D12_BLEND_SRC_COLOR,

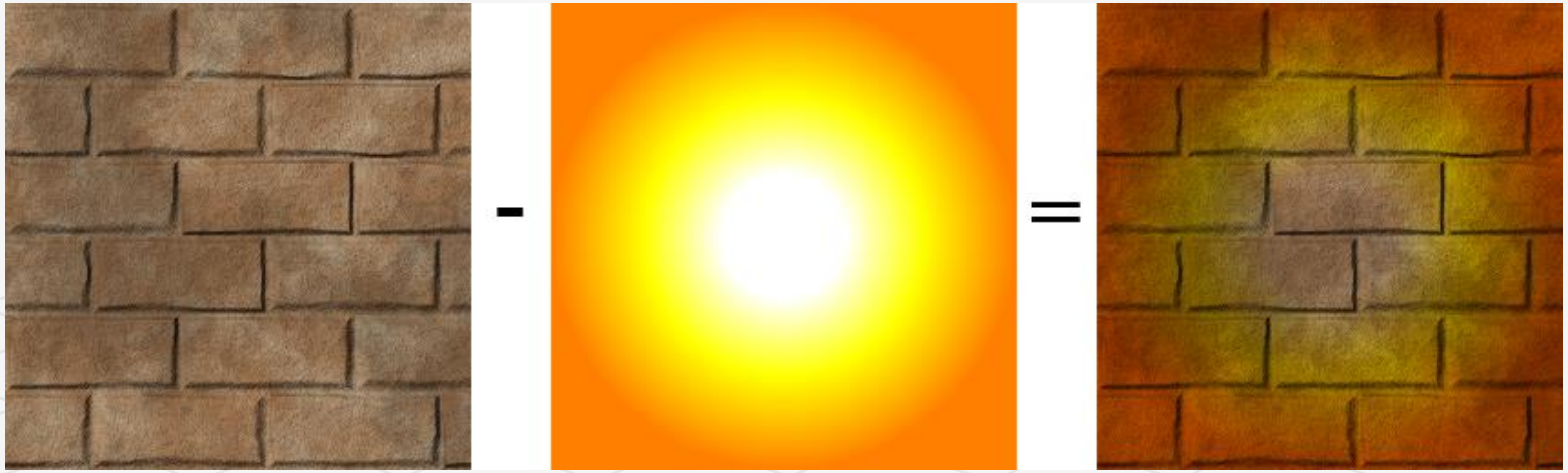
the blend operator: D3D12_BLEND_OP_ADD.

With this setup, the blending equation reduces to:

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$\mathbf{C} = \mathbf{C}_{src} \otimes (0,0,0) + \mathbf{C}_{dst} \otimes \mathbf{C}_{src}$$

$$\mathbf{C} = \mathbf{C}_{dst} \otimes \mathbf{C}_{src}$$



Transparency

0 alpha means 0% opaque, 0.4 means 40% opaque, and 1.0 means 100% opaque

The relationship between opacity and transparency:

T = 1− *A*, where *A* is opacity and *T* is transparency

suppose that we want to blend the source and destination pixels based on the opacity of the source pixel:

source blend factor: D3D12_BLEND_SRC_ALPHA

destination blend factor : D3D12_BLEND_INV_SRC_ALPHA

blend operator: D3D12_BLEND_OP_ADD

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$\mathbf{C} = \mathbf{C}_{src} \otimes (a_s, a_s, a_s) + \mathbf{C}_{dst} \otimes (1-a_s, 1-a_s, 1-a_s)$$

$$\mathbf{C} = a_s \mathbf{C}_{src} + (1-a_s)\mathbf{C}_{dst}$$

For example, suppose *as* = 0.25, which is to say the source pixel is only 25% opaque. Then when the source and destination pixels are blended together, we expect the final color will be a combination of 25% of the source pixel and 75% of the destination pixel

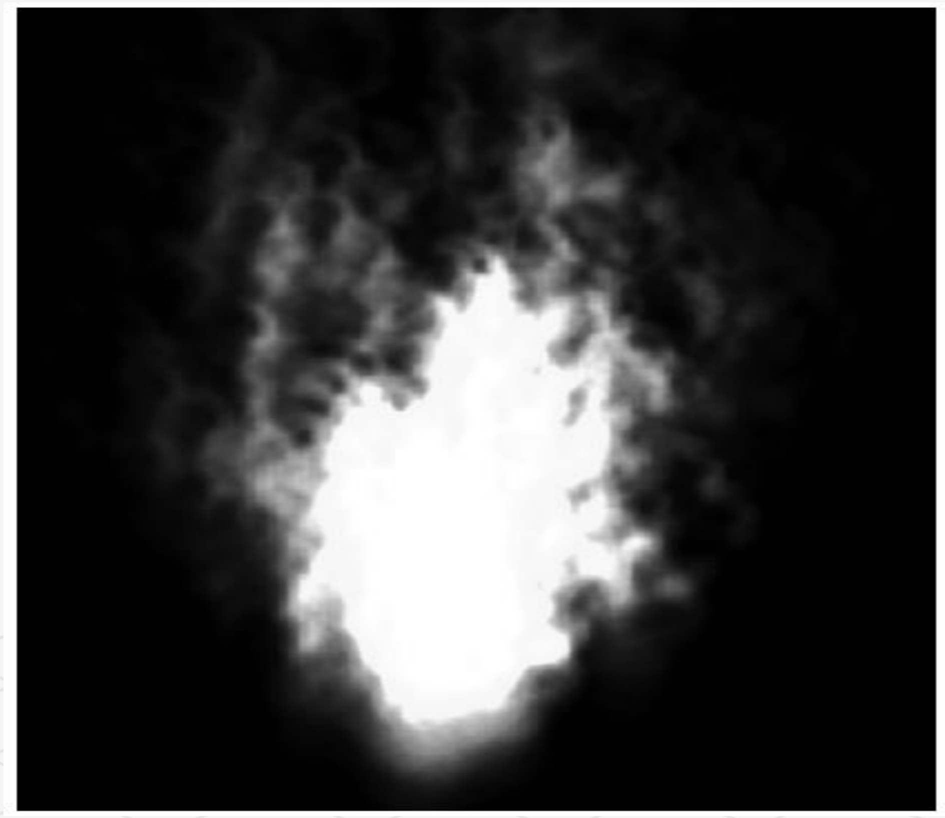
$$\mathbf{C} = a_s \mathbf{C}_{src} + (1-a_s)\mathbf{C}_{dst}$$

$$\mathbf{C} = 0.25\mathbf{C}_{src} + 0.75\mathbf{C}_{dst}$$

Blending and the Depth Buffer

- When blending with additive/subtractive/multiplicative blending, an issue arises with the depth test
- If we are rendering a set *S* of objects with additive blending, the idea is that the objects in *S* do not obscure each other
- Their colors are meant to simply accumulate
- We do not want to perform the depth test between objects in *S*
- If we did, one of the objects in *S* would obscure another object in *S*, causing the pixel fragments to be rejected due to the depth test
- This means that object's pixel colors would not be accumulated into the blend sum
- We can disable the depth test between objects in *S* by disabling writes to the depth buffer while rendering objects in *S*

With additive blending, the intensity is greater near the source point where more particles are overlapping and being added together. As the particles spread out, the intensity weakens because there are less particles overlapping and being added together.



ALPHA CHANNELS

- Source alpha components can be used in RGB blending to control transparency.
- We return the diffuse material's alpha value as the alpha output of the pixel shader
- Thus the alpha channel of the diffuse map is used to control transparency
- You can generally add an alpha channel in any popular image editing software, such as Adobe Photoshop, and then save the image to an image format that supports an alpha channel like DDS.

```
float4 PS(VertexOut pin) : SV_Target

{

float4 diffuseAlbedo = gDiffuseMap.Sample(gsamAnisotropicWrap, pin.TexC) *
gDiffuseAlbedo;

…

// Common convention to take alpha from diffuse

albedo.litColor.a = diffuseAlbedo.a;

return litColor;

}
```

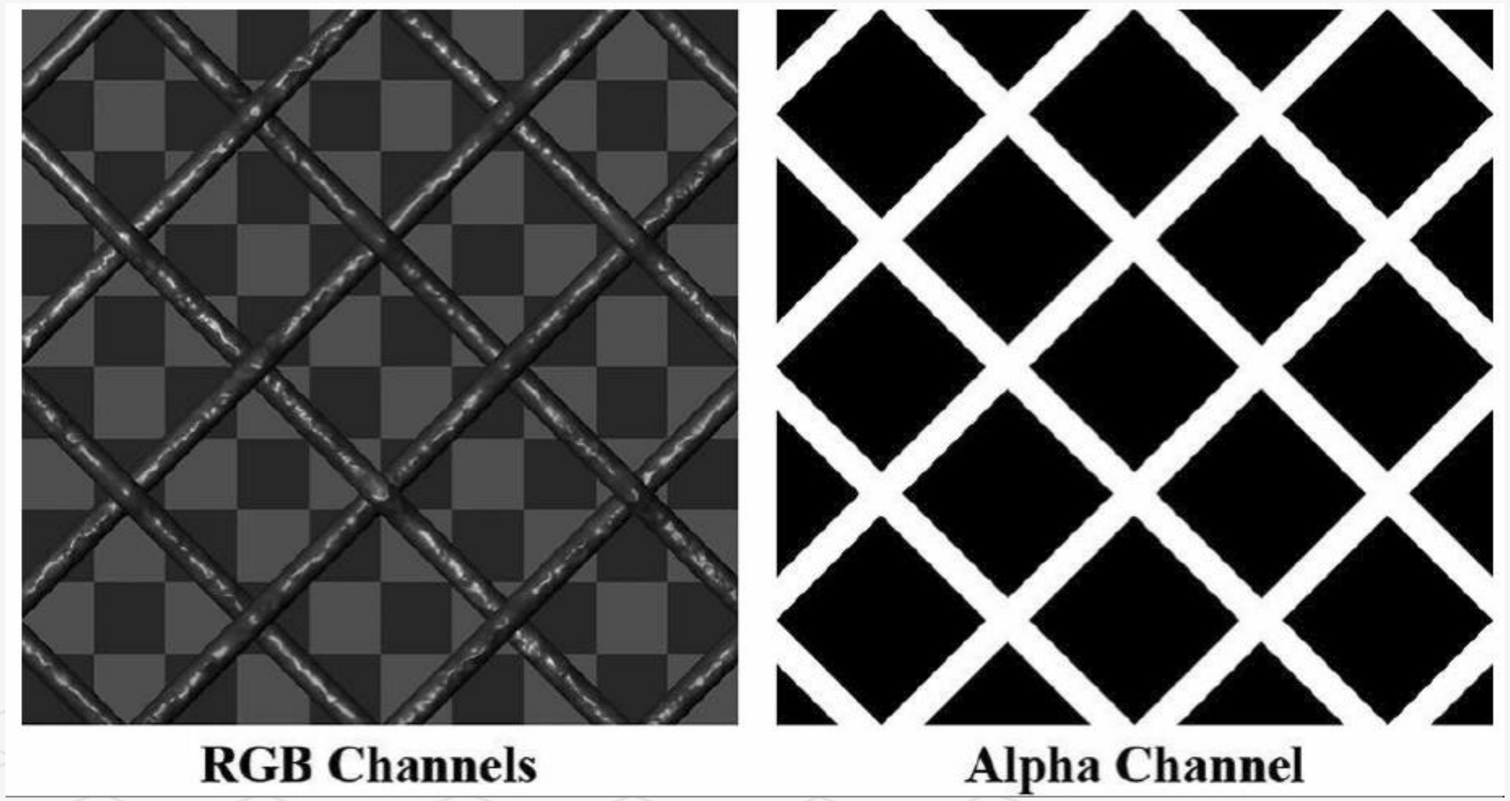
CLIPPING PIXELS

▪ Sometimes we want to completely reject a source pixel from being further processed

▪ This can be done with the intrinsic HLSL clip(x) function

▪ This function can only be called in a pixel shader, and it discards the current pixel from further processing if x < 0

▪ This function is useful to render wire fence textures, for example, like the one shown:, where a pixel is either completely opaque or completely transparent.



A wire fence texture with its alpha channel. The pixels with black alpha values will be rejected by the clip function and not drawn; hence, only the wire fence remains. Essentially, the alpha channel is used to mask out the non fence pixels from the texture.

Alpha Test

In the pixel shader, we grab the alpha component of the texture. If it is a small value close to 0, which indicates that the pixel is completely transparent, then we clip the pixel from further processing.

Observe that we only clip if ALPHA_TEST is defined; this is because we might not want to invoke clip for some render items, so we need to be able to switch it on/off by having specialized shaders. Moreover, there is a cost to using alpha testing, so we should only use it if we need it.

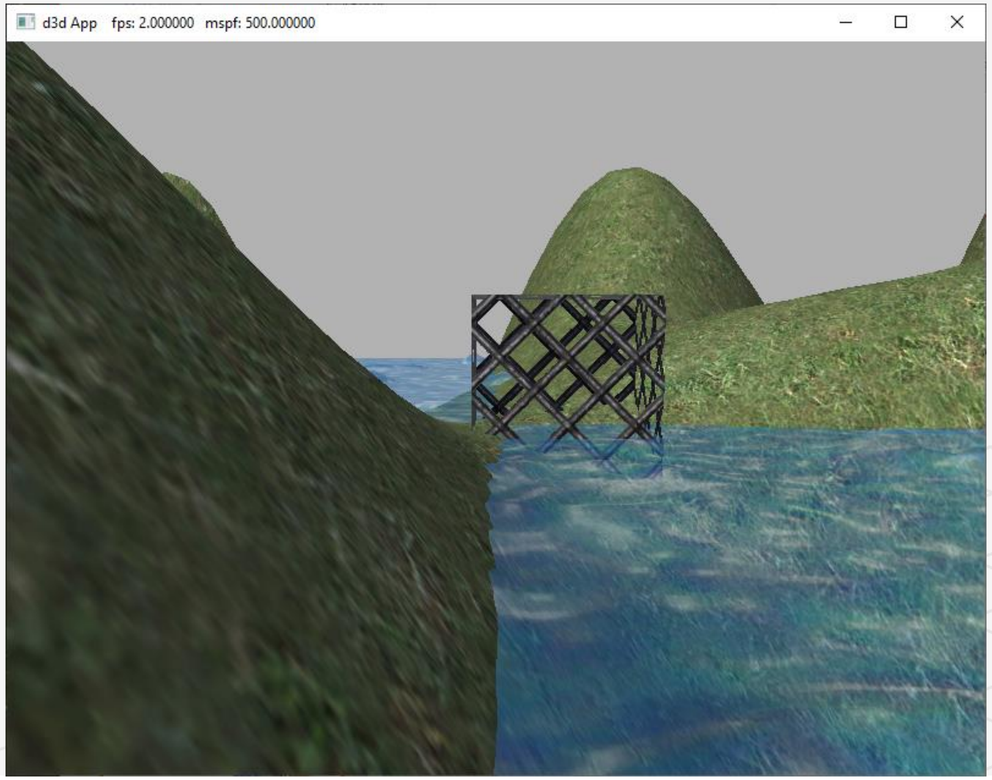
Note that the same result can be obtained using blending, but this is more efficient by discarding a pixel early from the pixel shader, the remaining pixel shader instructions can be skipped (no point in doing the calculations for a discarded pixel).

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

#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(diffuseAlbedo.a - 0.1f);
#endif
.........
// Common convention to take alpha from diffuse albedo.
    litColor.a = diffuseAlbedo.a;

    return litColor;
}
```

Blend Demo

Blend Demo renders semi-transparent water using transparency blending, and renders the wire fenced box using the clip test.

We can now see through the box with the fence texture, we want to disable back face culling for alpha tested objects.



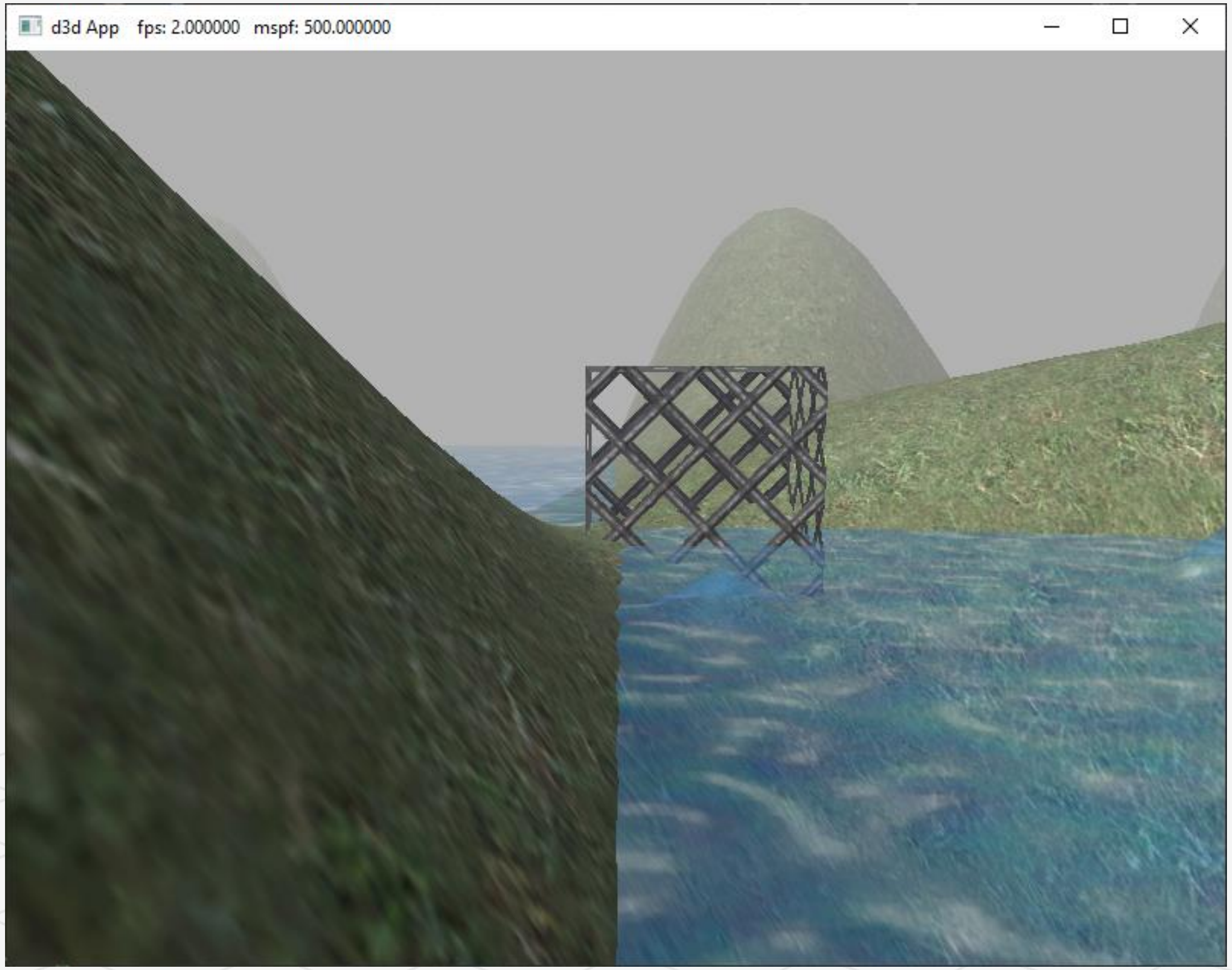
```
void BlendApp::BuildPSOs()
{
    D3D12_GRAPHICS_PIPELINE_STATE_DESC opaquePsoDesc;

// PSO for opaque objects.

…………………………………………………………….


// PSO for alpha tested objects

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"])));
}
```

FOG

Fog can mask distant rendering artifacts and prevent *popping*.

Popping refers to when an object that was previously behind the far plane all of a sudden comes in front of the frustum, due to camera movement. By having a layer of fog in the distance, the popping is hidden.

Even on clear days, distant objects such as mountains appear hazier and lose contrast as a function of depth, and we can use fog to simulate this atmospheric perspective phenomenon.

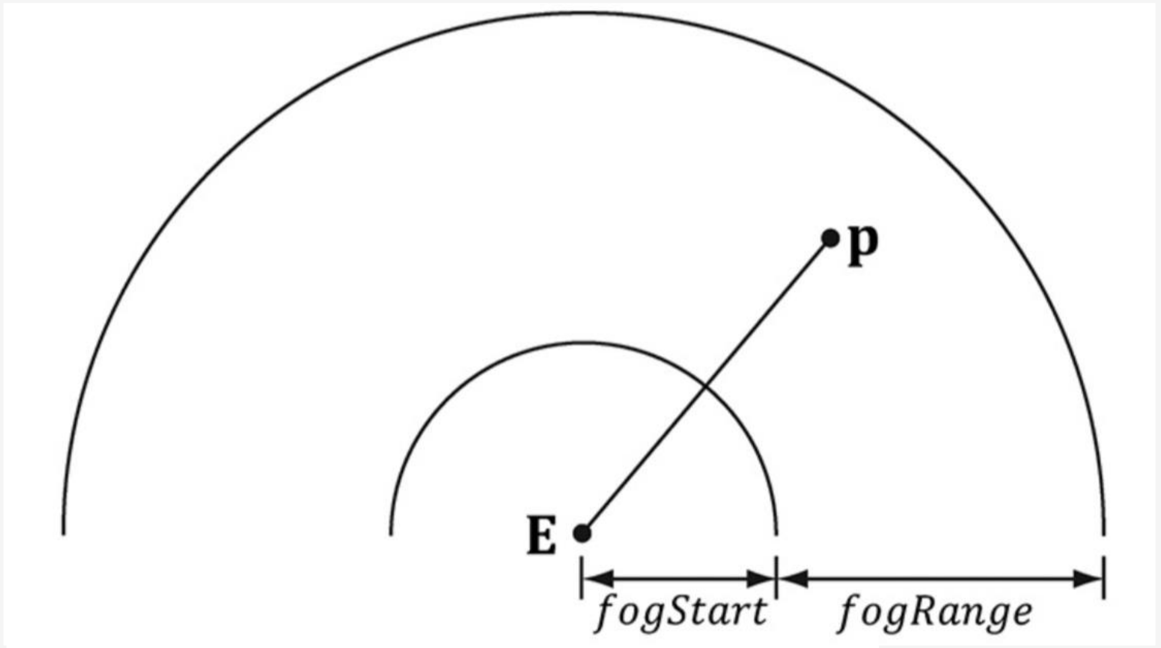


Implementing fog

We specify a fog color, a fog start distance from the camera and a fog range (i.e., the range from the fog start distance until the fog completely hides any objects). Then the color of a point on a triangle is a weighted average of its usual color and the fog color:

The parameter *s* ranges from 0 to 1 and is a function of the distance between the camera position and the surface point. As the distance between a surface point and the eye increases, the point becomes more and more obscured by the fog. The parameter s is defined as follows:

where dist (**p, E)** is the distance between the surface point p and the camera position **E**. The saturate function clamps the argument to the range [0, 1]:



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

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

Fog

Figure shows a plot of *s* as a function of distance. We see that when dist(**p, E)** ≤ *fogStart*, *s* = 0 and the fogged color is given by: *foggedColor* = *litColor*

(Top): A plot of *s* (the fog color weight) as a function of distance.

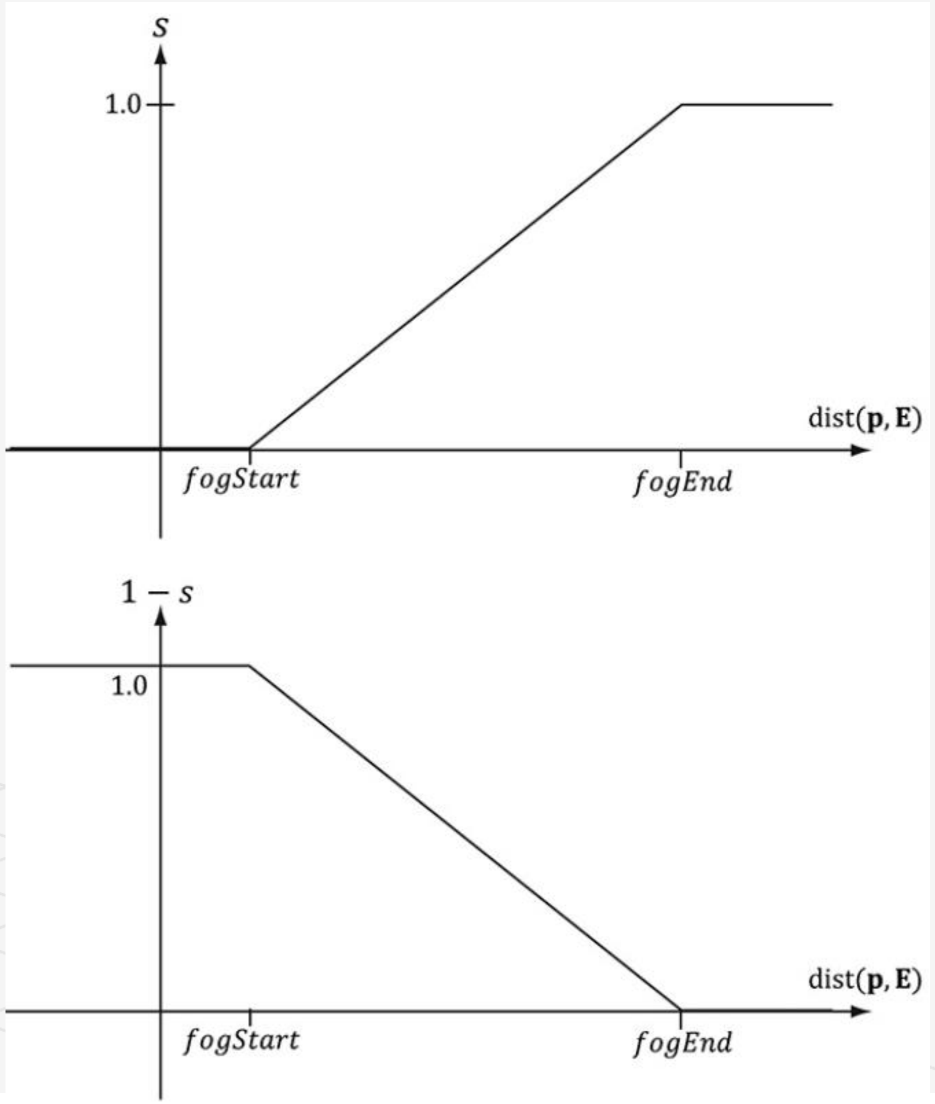
(Bottom): A plot of 1 – *s* (the lit color weight) as a function of distance. As *s* increases, (1 – *s*) decreases the same amount.]

*t*he fog does not start affecting the color until the distance from the camera is at least that of *fogStart*.

Let *fogEnd* = *fogStart* + *fogRange*. When dist(**p, E)** ≥ *fogEnd*, *s* = 1 and the fogged color is given by: *foggedColor* = *fogColor*

In other words, the fog completely hides the surface point at distances greater than or equal to *fogEnd*—so all you see is the fog color.

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



D3D_SHADER_MACRO

Some scenes may not want to use fog; therefore, we make fog optional by requiring FOG to be defined when compiling the shader.

Observe that in the fog calculation, we use the distToEye *value, that we also computed to normalize the* toEye *vector. A less optimal implementation would have been to write:*

```
 // Vector from point being lit to eye.

float3 toEyeW = gEyePosW - pin.PosW;

float distToEye = length(toEyeW);

toEyeW /= distToEye; // normalize
```

```
void BlendApp::BuildShadersAndInputLayout()
{
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");
```