

Week6

Blending & Stenciling

Hooman Salamat

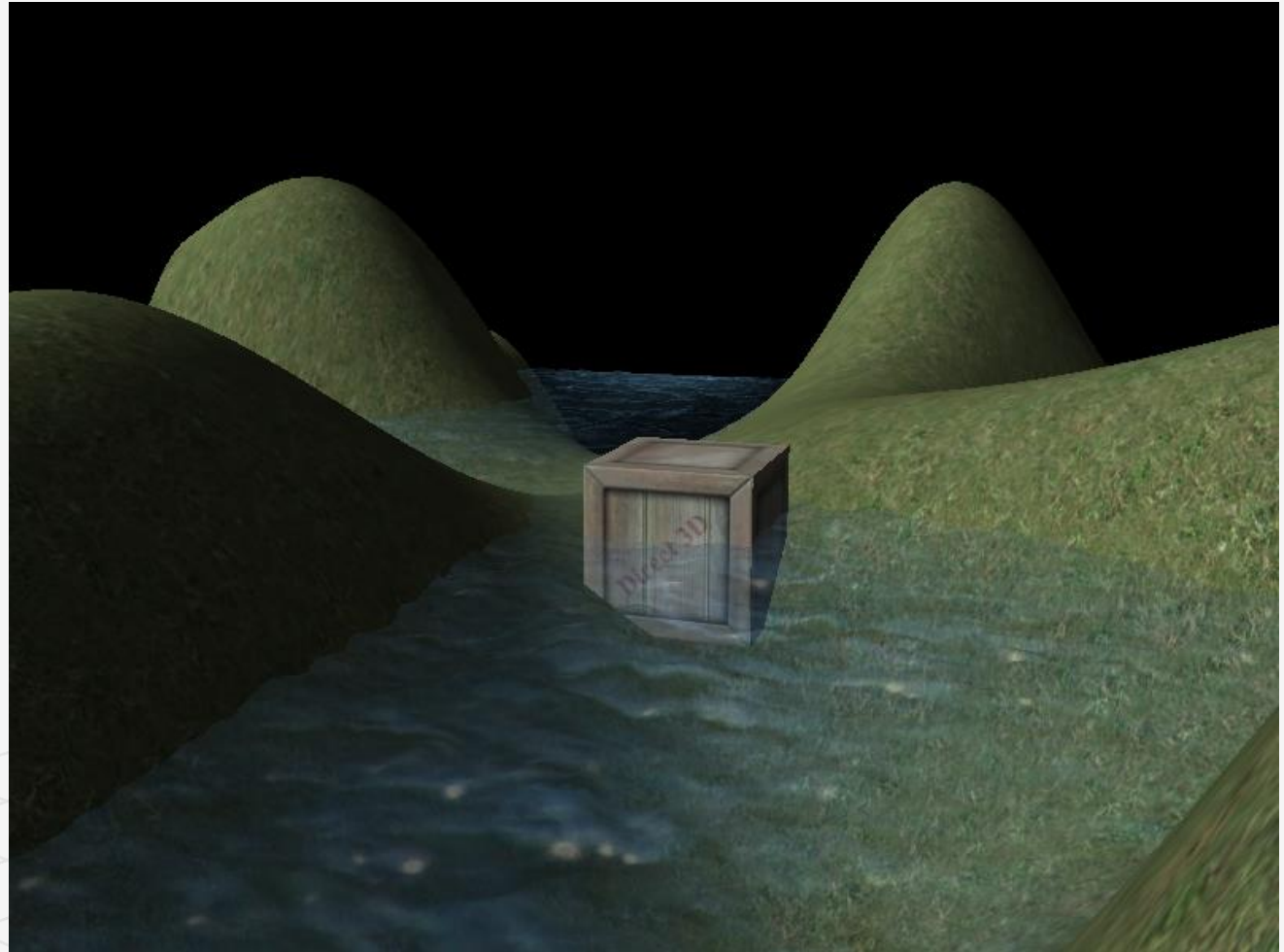
Objectives

- Understand how blending works, its 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 be 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 i^{th} pixel we are currently rasterizing (source pixel)
- Let C_{dst} be the color of the i^{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 i^{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

$$C = C_{src} \otimes F_{src} \boxplus C_{dst} \otimes F_{dst}$$

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,           $C = C_{src} \otimes F_{src} + C_{dst} \otimes F_{dst}$ 
    D3D12_BLEND_OP_SUBTRACT = 2,      $C = C_{dst} \otimes F_{dst} - C_{src} \otimes F_{src}$ 
    D3D12_BLEND_OP_REV_SUBTRACT = 3,  $C = C_{src} \otimes F_{src} - C_{dst} \otimes F_{dst}$ 
    D3D12_BLEND_OP_MIN = 4,           $C = \min(C_{src}, C_{dst})$ 
    D3D12_BLEND_OP_MAX = 5,           $C = \max(C_{src}, C_{dst})$ 
} D3D12_BLEND_OP;
```

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:

$$\begin{aligned} \mathbf{C} &= \mathbf{C}_{src} \otimes \mathbf{F}_{src} + \mathbf{C}_{dst} \otimes \mathbf{F}_{dst} \\ A &= A_{dst} F_{dst} - A_{src} F_{src} \end{aligned}$$

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 \mathbf{F}_{src} and \mathbf{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), \mathbf{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).

D3D12_BLEND_BLEND_FACTOR: $\mathbf{F} = (r, g, b)$ 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,

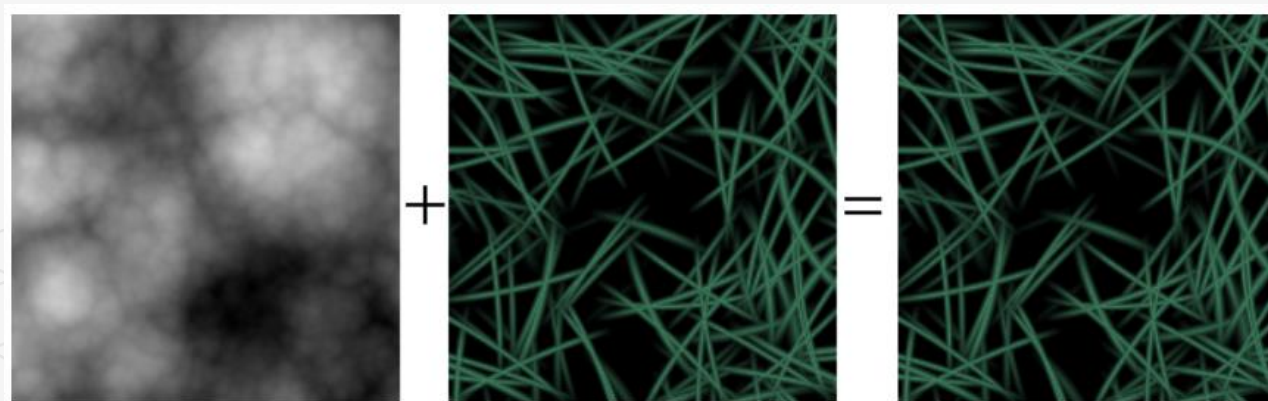
the blend operator: D3D12_BLEND_OP_ADD.

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

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

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

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

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.

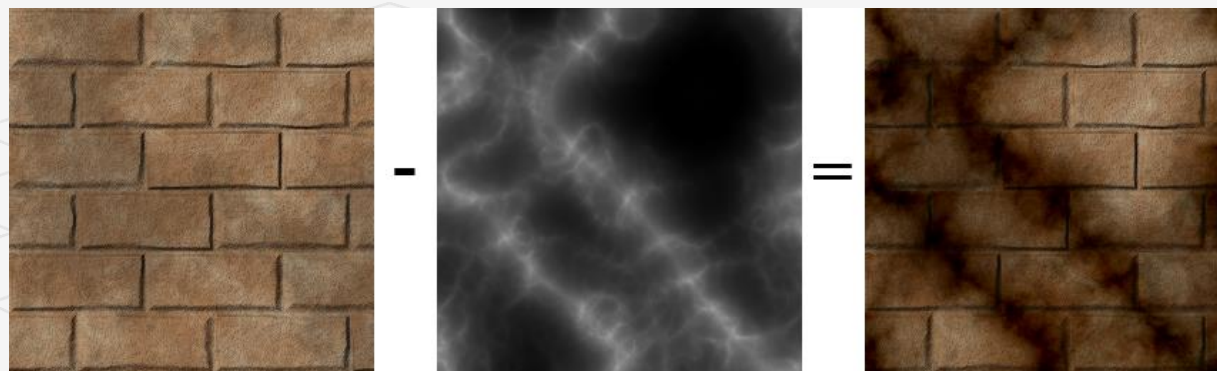
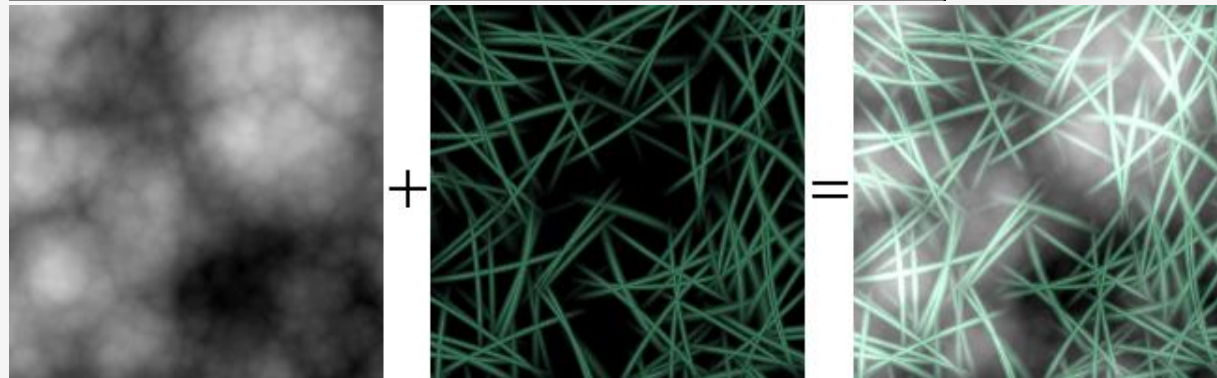
The following adds source and destination color. Adding
creates a brighter image since color is being added.

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

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

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

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



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 $a_s = 0.25$, 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

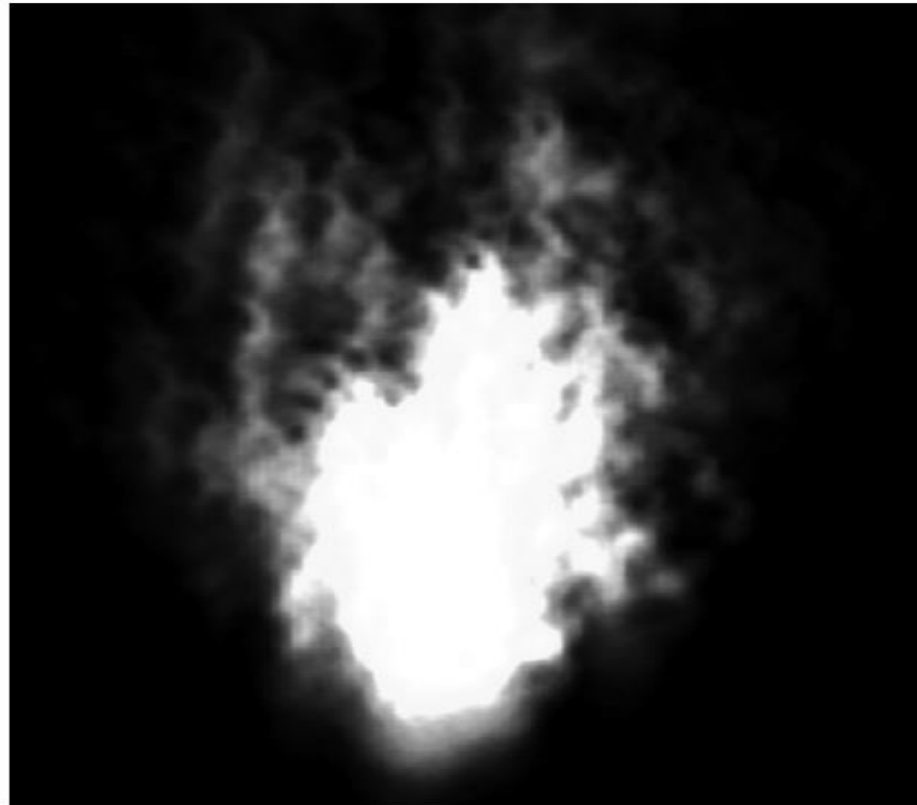
- We do not want to perform the depth test between objects in S

- 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

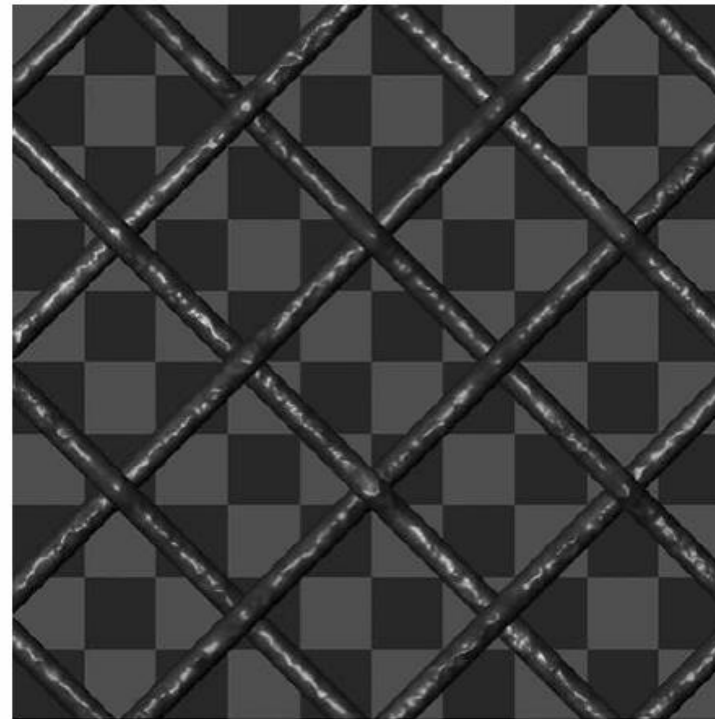
    albedo.litColor.a = diffuseAlbedo.a;

    return litColor;
}
```

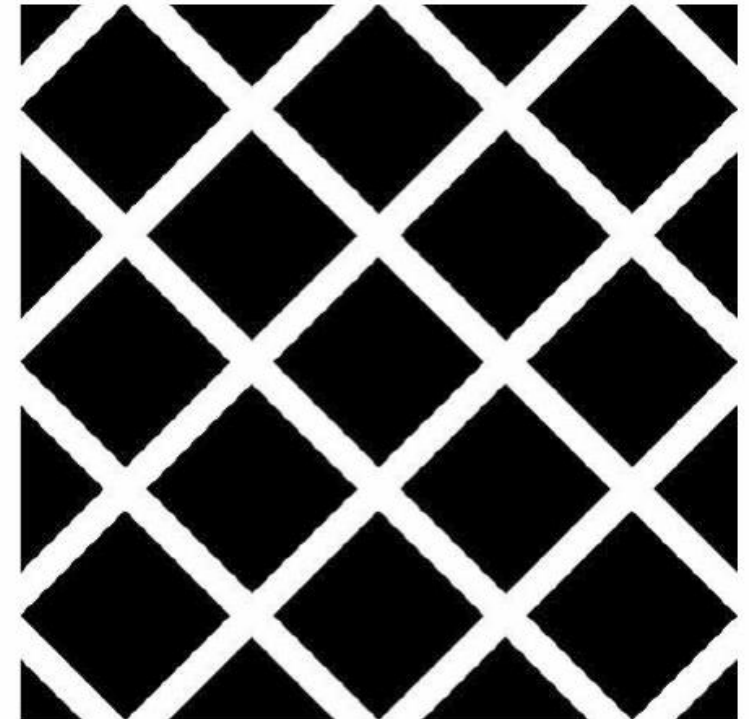
CLIPPING PIXELS

- 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.



RGB Channels



Alpha Channel

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;

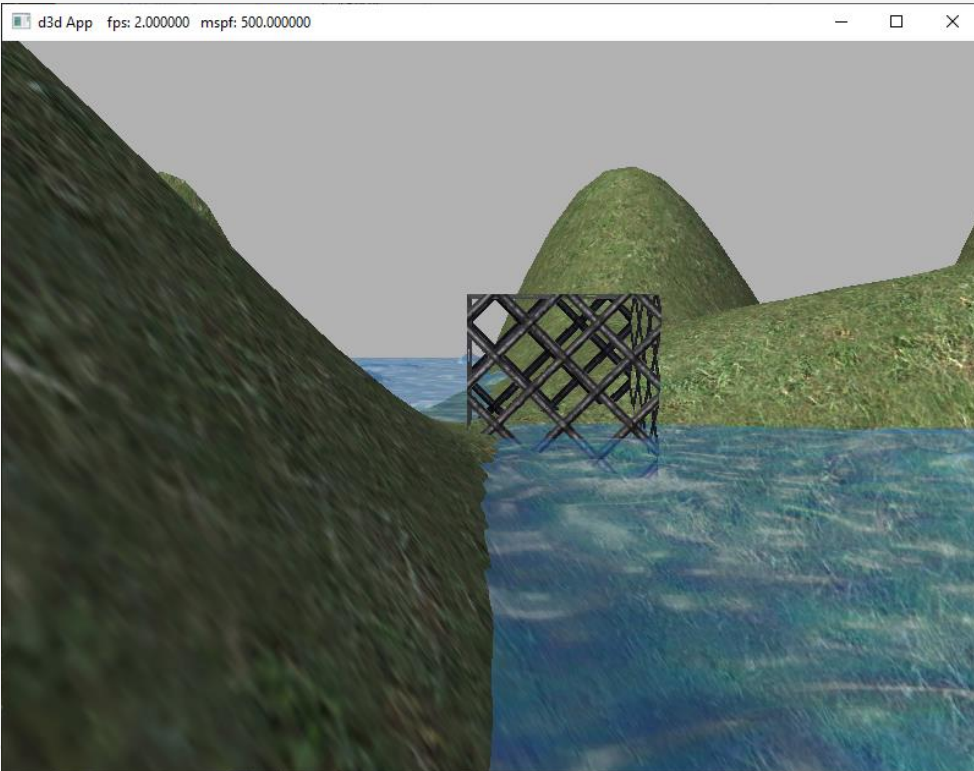
#ifdef ALPHA_TEST
    // Discard pixel if texture alpha < 0.1. We do this test as soon
    // as possible in the shader so that we can potentially exit the
    // shader early, thereby skipping the rest of the shader code.
    clip(diffuseAlbedo.a - 0.1f);
#endif
    .....
    // Common convention to 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

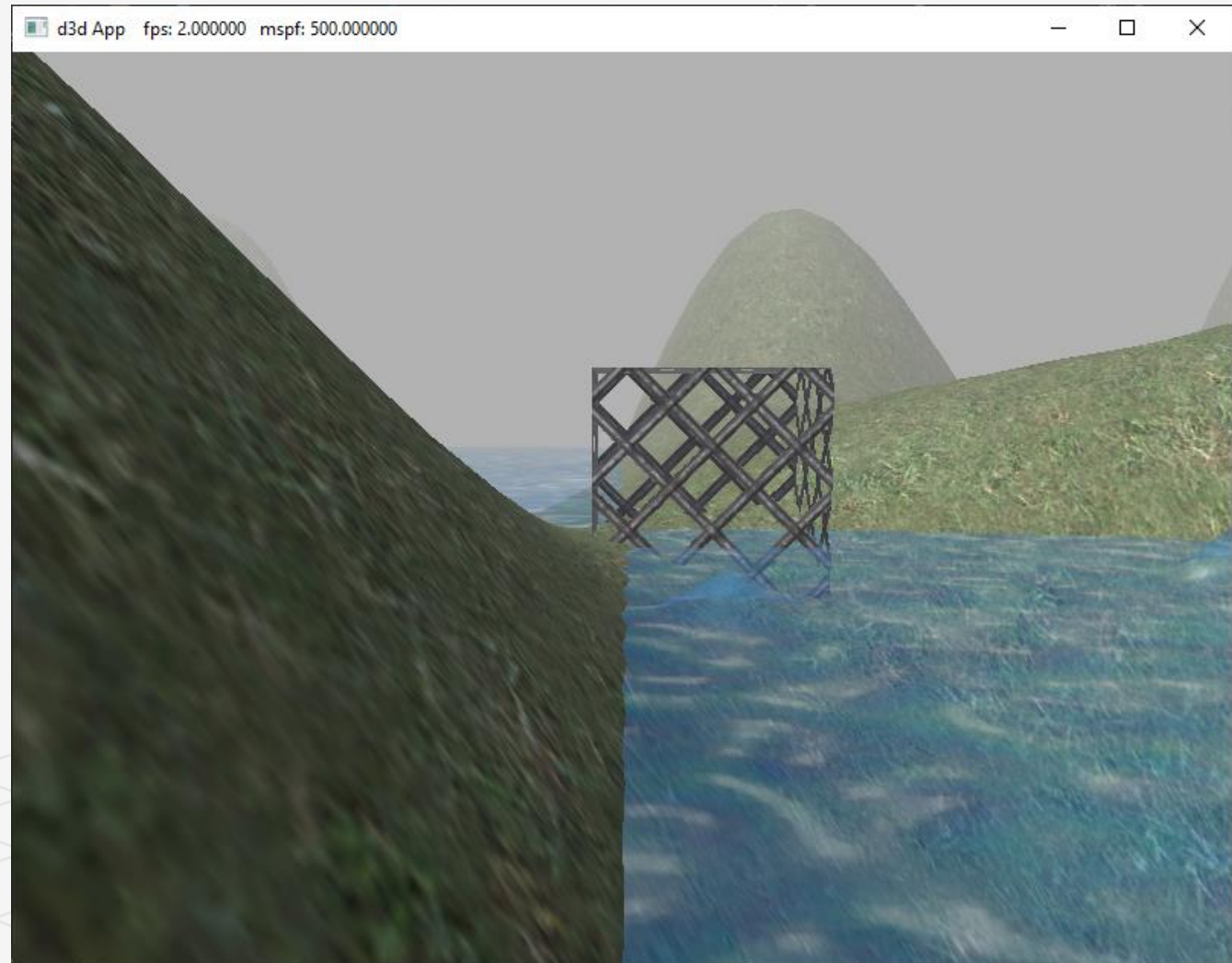
    D3D12_GRAPHICS_PIPELINE_STATE_DESC alphaTestedPsoDesc =
    opaquePsoDesc;
    alphaTestedPsoDesc.PS =
    {
        reinterpret_cast<BYTE*>(mShaders["alphaTestedPS"]-
        >GetBufferPointer()),
        mShaders["alphaTestedPS"]->GetBufferSize()
    };
    alphaTestedPsoDesc.RasterizerState.CullMode =
    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.

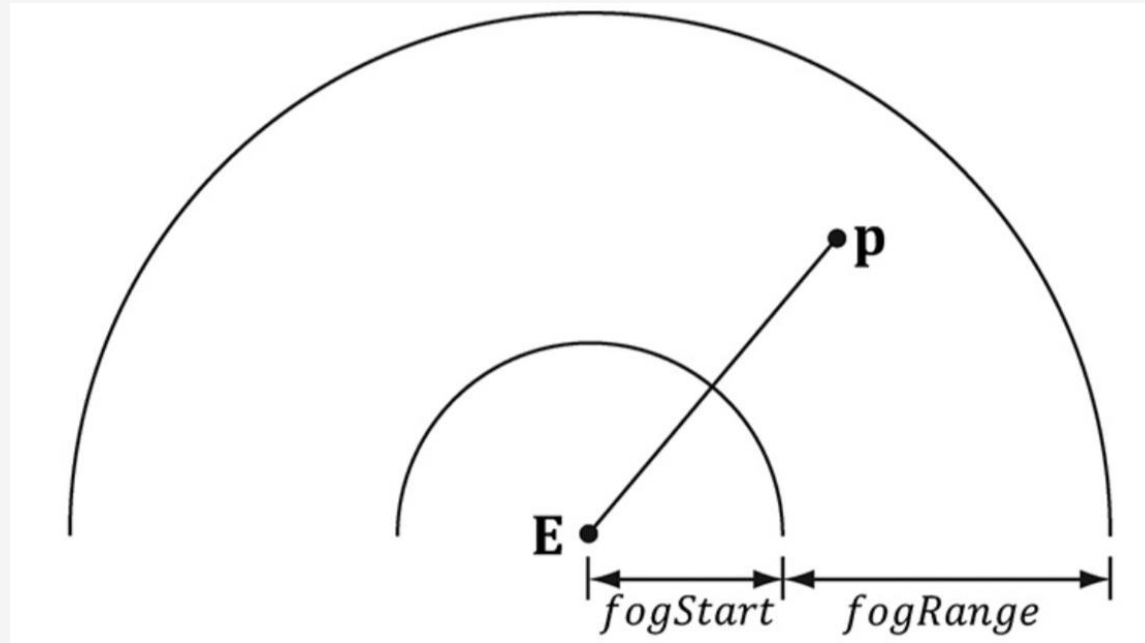


Implementing fog

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 $\text{dist}(\mathbf{p}, \mathbf{E})$ is the distance between the surface point \mathbf{p} and the camera position \mathbf{E} . The saturate function clamps the argument to the range $[0, 1]$:



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

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

Fog

Figure shows a plot of s as a function of distance. We see that when $\text{dist}(\mathbf{p}, \mathbf{E}) \leq \text{fogStart}$, $s = 0$ and the fogged color is given by: $\text{foggedColor} = \text{litColor}$

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

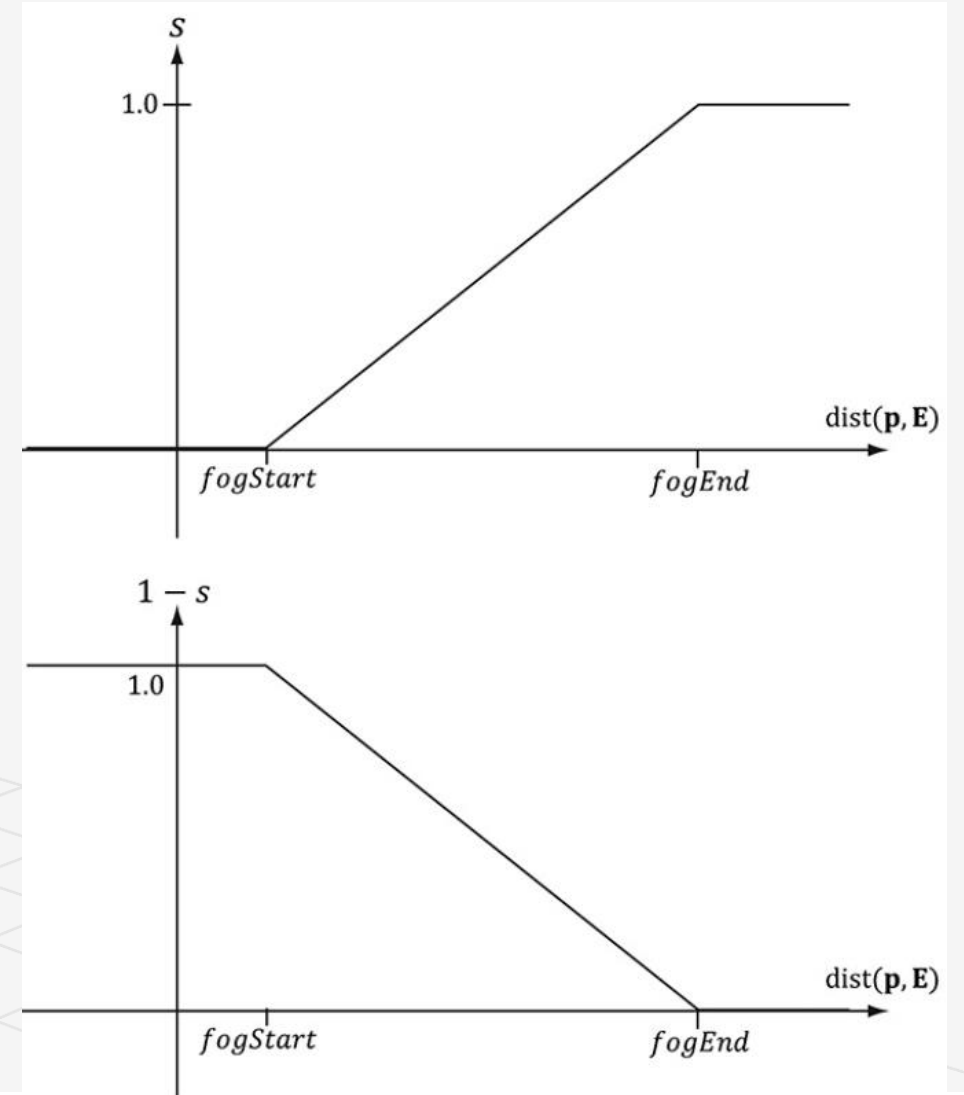
(Bottom): A plot of $1 - s$ (the lit color weight) as a function of distance. As s increases, $(1 - s)$ decreases the same amount.]

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

Let $\text{fogEnd} = \text{fogStart} + \text{fogRange}$. When $\text{dist}(\mathbf{p}, \mathbf{E}) \geq \text{fogEnd}$, $s = 1$ and the fogged color is given by:
 $\text{foggedColor} = \text{fogColor}$

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:*

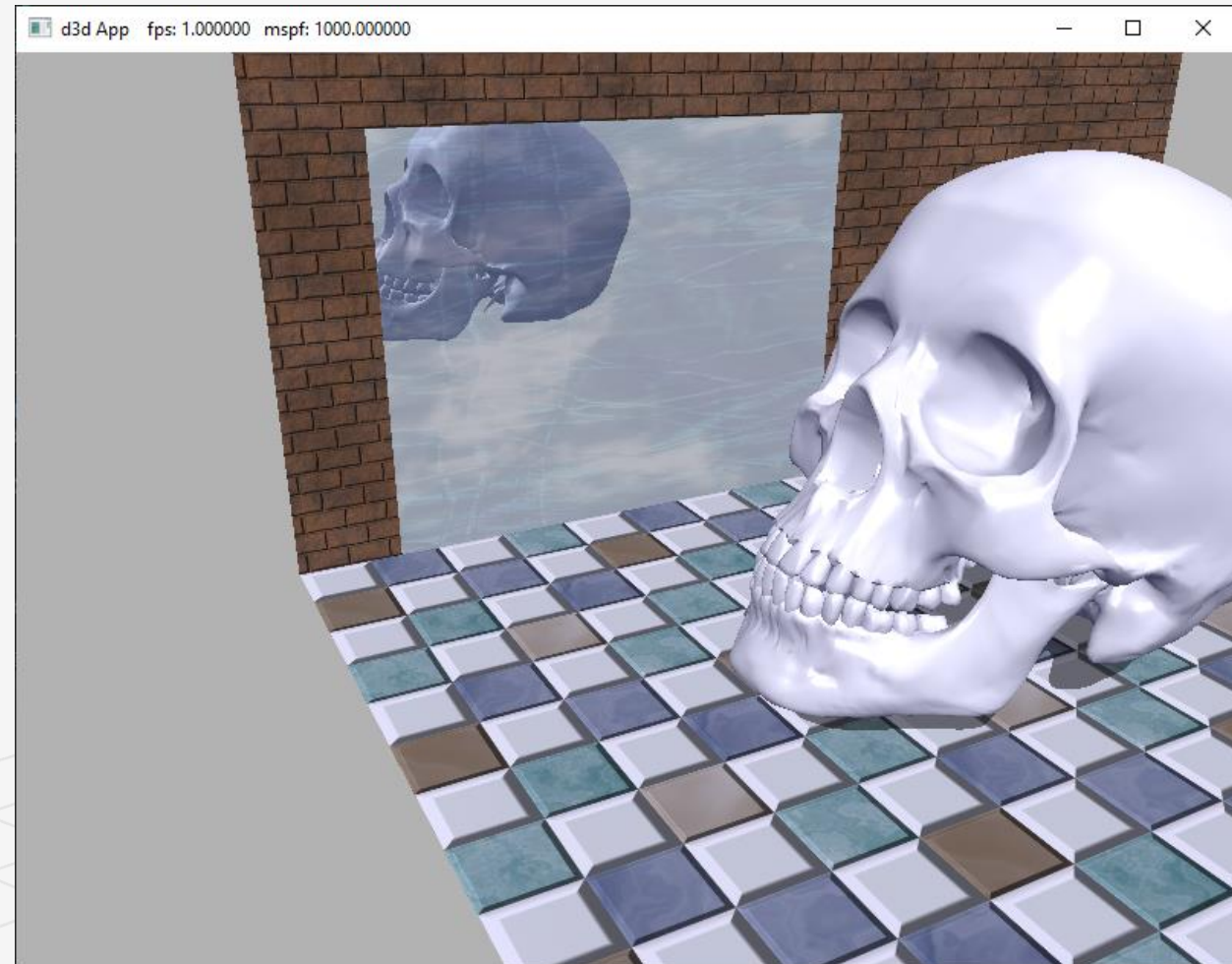
```
// Vector from point being lit to eye.  
  
float3 toEyeW = gEyePosW - pin.PosW;  
  
float distToEye = length(toEyeW);  
  
toEyeW /= distToEye; // normalize
```

```
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");  
}
```


Stenciling

Objectives:

1. To 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.
2. To learn how to implement mirrors by using the stencil buffer to prevent reflections from being drawn to non-mirror surfaces.
3. To be able to identify double blending and understand how the stencil buffer can prevent it.
4. To explain depth complexity and describe two ways the depth complexity of a scene can be measured.



Stenciling

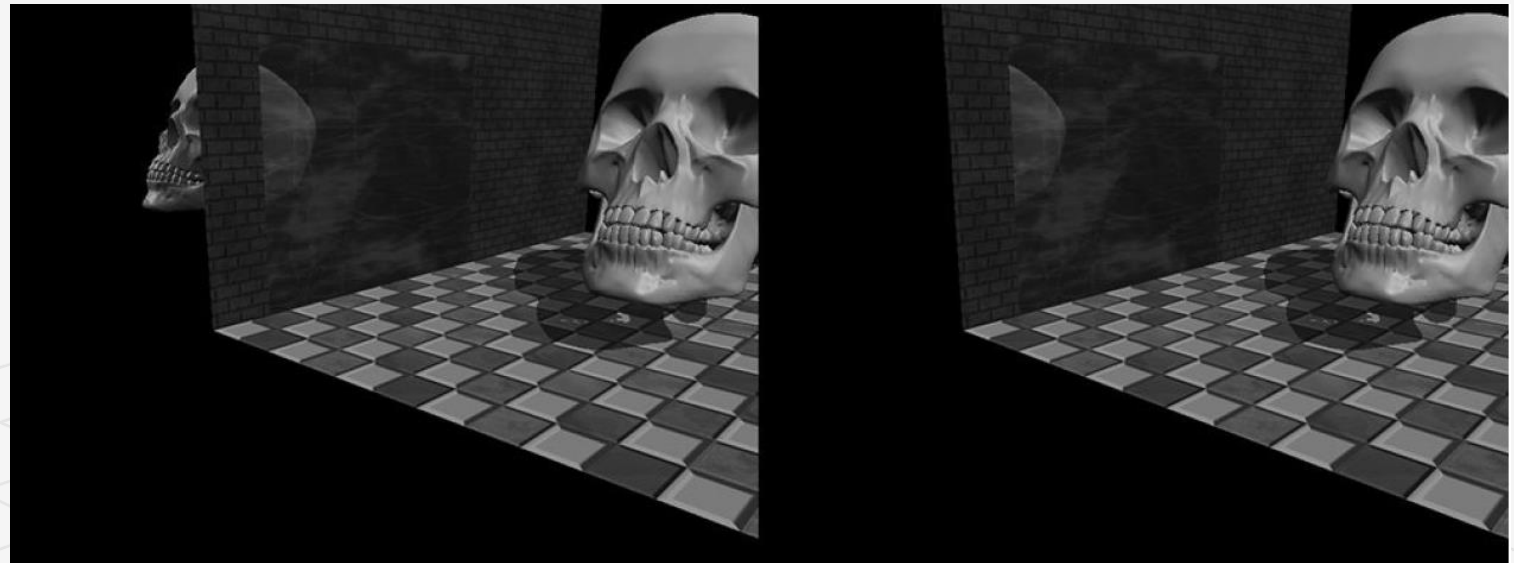
The stencil buffer is an off-screen buffer we can use to achieve some special effects.

The stencil buffer has the same resolution as the back buffer and depth buffer.

when a stencil buffer is specified, it comes attached to the depth buffer.

(Left) The reflected skull shows properly in the mirror. The reflection does not show through the wall bricks because it fails the depth test in this area. However, looking behind the wall we are able to see the reflection, thus breaking the illusion (the reflection should only show up through the mirror).

(Right) By using the stencil buffer, we can block the reflected skull from being rendered unless it is being drawn in the mirror.



D3D12_DEPTH_STENCIL_DESC

The stencil buffer (and also the depth buffer) state is configured by filling out a D3D12_DEPTH_STENCIL_DESC instance and assigning it to the D3D12_GRAPHICS_PIPELINE_STATE_DESC::DepthStencilState field of a pipeline state object (PSO).

1. DepthEnable: Specify true to enable the depth buffering
2. DepthWriteMask: This can be either D3D12_DEPTH_WRITE_MASK_ZERO (disable writes to depth buffer) or D3D12_DEPTH_WRITE_MASK_ALL
3. StencilReadMask: 0xff: does not mask any bits
4. StencilWriteMask: When the stencil buffer is being updated, we can mask off certain bits from being written to with the write mask. For example, if you wanted to prevent the top 4 bits from being written to, you could use the write mask of 0x0f.

```
typedef struct D3D12_DEPTH_STENCIL_DESC
{
    BOOL DepthEnable; // Default True

    D3D12_DEPTH_WRITE_MASK DepthWriteMask; // Default: D3D12_DEPTH_WRITE_MASK_ALL

    D3D12_COMPARISON_FUNC DepthFunc; //Default: D3D12_COMPARISON_LESS

    BOOL StencilEnable; // Default: False

    UINT8 StencilReadMask; // Default: 0xff

    UINT8 StencilWriteMask; // Default: 0xff

    D3D12_DEPTH_STENCIL_OP_DESC FrontFace;

    D3D12_DEPTH_STENCIL_OP_DESC BackFace;
} D3D12_DEPTH_STENCIL_DESC;
```

```
opaquePsoDesc.DepthStencilState = CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT);
```

DEPTH/STENCIL FORMATS AND CLEARING

The depth/stencil buffer is a texture, it must be created with certain data formats.

1. DXGI_FORMAT_D32_FLOAT_S8X24_UINT:

Specifies a 32-bit floating-point depth buffer, with 8-bits (unsigned integer) reserved for the stencil buffer mapped to the [0, 255] range and 24-bits not used for padding.

2. DXGI_FORMAT_D24_UNORM_S8_UINT: Specifies an unsigned 24-bit depth buffer mapped to the [0, 1] range with 8-bits (unsigned integer) reserved for the stencil buffer mapped to the [0, 255] range.

```
// d3dApp.h: In our D3DApp framework, when we create the depth buffer, we specify:
```

```
DXGI_FORMAT mDepthStencilFormat = DXGI_FORMAT_D24_UNORM_S8_UINT;
```

```
// d3dApp.cpp Create the depth/stencil buffer and view.
```

```
D3D12_RESOURCE_DESC depthStencilDesc;
```

```
depthStencilDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
```

```
depthStencilDesc.Alignment = 0;
```

```
depthStencilDesc.Width = mClientWidth;
```

```
depthStencilDesc.Height = mClientHeight;
```

```
depthStencilDesc.DepthOrArraySize = 1;
```

```
depthStencilDesc.MipLevels = 1;
```

```
// an SRV to the depth buffer to read from
```

```
// the depth buffer. Therefore, because we need to create two views to the same resource:
```

```
// 1. SRV format: DXGI_FORMAT_R24_UNORM_X8_TYPELESS
```

```
// 2. DSV Format: DXGI_FORMAT_D24_UNORM_S8_UINT
```

```
// we need to create the depth buffer resource with a typeless format.
```

```
depthStencilDesc.Format = DXGI_FORMAT_R24G8_TYPELESS;
```

```
depthStencilDesc.SampleDesc.Count = m4xMsaaState ? 4 : 1;
```

```
depthStencilDesc.SampleDesc.Quality = m4xMsaaState ? (m4xMsaaQuality - 1) : 0;
```

```
depthStencilDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
```

```
depthStencilDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;
```

ID3D12GraphicsCommandList::ClearDepthStencilView

Also, the stencil buffer should be reset to some value at the beginning of each frame. This is done with the following method (which also clears the depth buffer):


1. DepthStencilView: Descriptor to the view of the depth/stencil buffer we want to clear.
2. ClearFlags: Specify D3D12_CLEAR_FLAG_DEPTH to clear the depth buffer only; specify D3D12_CLEAR_FLAG_STENCIL to clear the stencil buffer only; specify D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL to clear both.
3. Depth: The float-value to set each pixel in the depth buffer to; it must be a floating point number x such that $0 \leq x \leq 1$.
4. Stencil: The integer-value to set each pixel of the stencil buffer to; it must be an integer n such that $0 \leq n \leq 255$.
5. NumRects: The number of rectangles in the array pRects points to.
6. pRects: An array of D3D12_RECTs marking rectangular regions on the depth/stencil buffer to clear; specify nullptr to clear the entire depth/stencil buffer.

```
void ClearDepthStencilView(  
    D3D12_CPU_DESCRIPTOR_HANDLE DepthStencilView,  
    D3D12_CLEAR_FLAGS ClearFlags,  
    FLOAT Depth,  
    UINT8 Stencil,  
    UINT NumRects,  
    const D3D12_RECT *pRects)  
{  
    mCommandList->ClearDepthStencilView(DepthStencilView(), D3D12_CLEAR_FLAG_DEPTH |  
        D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, nullptr);  
}
```

THE STENCIL TEST

We use the stencil buffer to block rendering to certain areas of the back buffer. The decision to block a particular pixel from being written is decided by the *stencil test*.

The stencil test is performed as pixels get rasterized (i.e., during the output-merger stage), assuming stenciling is enabled.

The stencil test then compares the LHS with the RHS as specified an application-chosen *comparison function*: 

if(StencilRef & StencilReadMask  Value & StencilReadMask)

accept pixel // write the pixel to the back buffer if test evaluates to true (and also passes the depth test!)

else

reject pixel // block the pixel from being written to the back buffer, therefore, won't go to depth buffer

1. A left-hand-side (LHS) operand that is determined by ANDing an application-defined *stencil reference value* (StencilRef) with an application-defined *masking value* (StencilReadMask).
2. A right-hand-side (RHS) operand that is determined by ANDing the entry already in the stencil buffer of the particular pixel we are testing (Value) with an application-defined masking value (same StencilReadMask).

The Comparison Operator

typedef

The comparison operator is any one of the functions defined:

1. D3D12_COMPARISON_NEVER: The function always returns false.
2. D3D12_COMPARISON_LESS: Replace with the < operator.
3. D3D12_COMPARISON_EQUAL: Replace with the == operator.
4. D3D12_COMPARISON_LESS_EQUAL: Replace with the ≤ operator.
5. D3D12_COMPARISON_GREATER: Replace with the > operator.
6. D3D12_COMPARISON_NOT_EQUAL: Replace with the != operator.
7. D3D12_COMPARISON_GREATER_EQUAL: Replace with the ≥ operator.
8. D3D12_COMPARISON_ALWAYS: The function always returns true.

enum D3D12_COMPARISON_FUNC

```
{  
  
    D3D12_COMPARISON_FUNC_NEVER= 1,  
  
    D3D12_COMPARISON_FUNC_LESS= 2,  
  
    D3D12_COMPARISON_FUNC_EQUAL= 3,  
  
    D3D12_COMPARISON_FUNC_LESS_EQUAL= 4,  
  
    D3D12_COMPARISON_FUNC_GREATER= 5,  
  
    D3D12_COMPARISON_FUNC_NOT_EQUAL= 6,  
  
    D3D12_COMPARISON_FUNC_GREATER_EQUAL= 7,  
  
    D3D12_COMPARISON_FUNC_ALWAYS= 8  
} D3D12_COMPARISON_FUNC;
```

D3D12_DEPTH_STENCIL_OP_DESC

```
typedef struct D3D12_DEPTH_STENCIL_OP_DESC
```

```
{
```

```
    D3D12_STENCIL_OP StencilFailOp;
```

```
    // Default: D3D12_STENCIL_OP_KEEP
```

```
    D3D12_STENCIL_OP StencilDepthFailOp;
```

```
    // Default: D3D12_STENCIL_OP_KEEP
```

```
    D3D12_STENCIL_OP StencilPassOp;
```

```
    // Default: D3D12_STENCIL_OP_KEEP
```

```
    D3D12_COMPARISON_FUNC StencilFunc;
```

```
    // Default: D3D12_COMPARISON_ALWAYS
```

```
} D3D12_DEPTH_STENCIL_OP_DESC;
```

1. StencilFailOp: A member of the D3D12_STENCIL_OP enumerated type describing how the stencil buffer should be updated when the stencil test fails for a pixel fragment.

2. StencilDepthFailOp: how the stencil buffer should be updated when the stencil test passes but the depth test fails for a pixel fragment.

3. StencilPassOp: how the stencil buffer should be updated when the stencil test and depth test both pass for a pixel fragment.

4. StencilFunc: A member of the D3D12_COMPARISON_FUNC enumerated type to define the stencil test comparison function.

```
typedef enum D3D12_STENCIL_OP
```

```
{
```

```
    D3D12_STENCIL_OP_KEEP= 1, //keep the value currently there
```

```
    D3D12_STENCIL_OP_ZERO= 2, //set the stencil buffer entry to zero
```

```
    D3D12_STENCIL_OP_REPLACE= 3, //replaces the stencil buffer entry with StencilRef value
```

```
    D3D12_STENCIL_OP_INCR_SAT= 4, //increment the stencil buffer entry & Clamp
```

```
    D3D12_STENCIL_OP_DECR_SAT= 5, //decrement the stencil buffer entry & Clamp
```

```
    D3D12_STENCIL_OP_INVERT= 6, //invert the bits of the stencil buffer entry
```

```
    D3D12_STENCIL_OP_INCR= 7, //increment the stencil buffer entry, wrap to 0
```

```
    D3D12_STENCIL_OP_DECR= 8 //decrement the stencil buffer entry, wrap to max(e.g. 255)
```

```
} D3D12_STENCIL_OP;
```

Creating and Binding a Depth/Stencil State

Once we have fully filled out a `D3D12_DEPTH_STENCIL_DESC` instance describing our depth/stencil state, we can assign it to the `D3D12_GRAPHICS_PIPELINE_STATE_DESC::DepthStencilState` field of a PSO. Any geometry drawn with this PSO will be rendered with the depth/stencil settings of the PSO.

```
opaquePsoDesc.DepthStencilState = CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT);
```

The stencil reference value is set with the `ID3D12GraphicsCommandList::OMSetStencilRef` method, which takes a single unsigned integer parameter; for example, the following sets the stencil reference value to 1:

```
mCommandList->OMSetStencilRef(1);
```

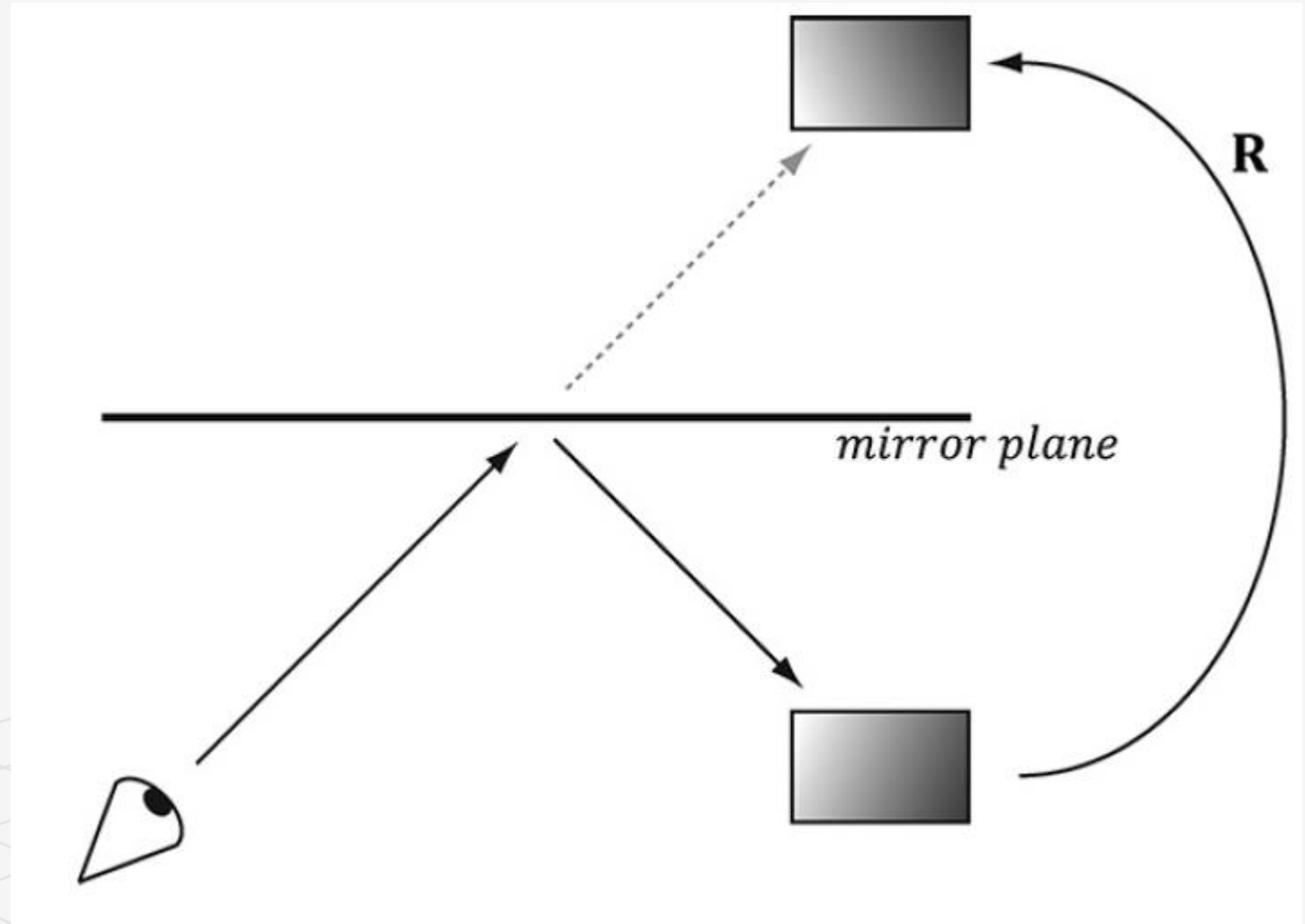


IMPLEMENTING PLANAR MIRRORS

Implementing mirrors programmatically requires us to solve three problems.

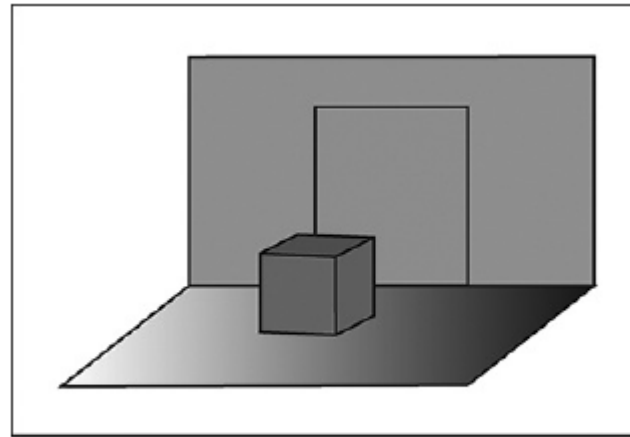
1. We must learn how to reflect an object about an arbitrary plane so that we can draw the reflection correctly.
2. *When we draw the reflection, we also need to reflect the light source across the mirror plane*
3. We must only display the reflection in a mirror, that is, we must somehow "mark" a surface as a mirror and then, as we are rendering, only draw the reflected object if it is in a mirror.

The eye sees the box reflection through the mirror. To simulate this, we reflect the box across the mirror plane and render the reflected box as usual.

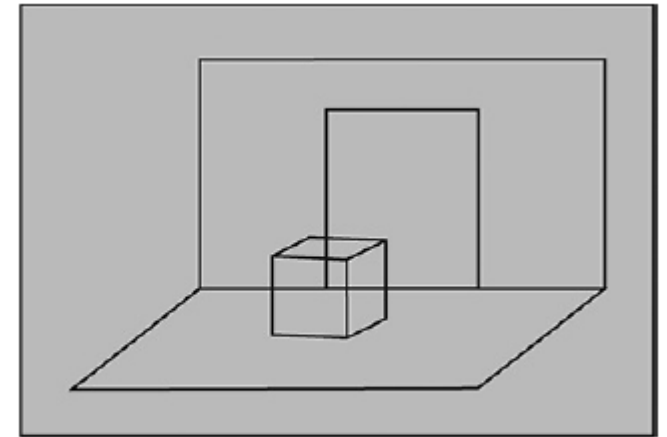


IMPLEMENTING PLANAR MIRRORS

The floor, walls, and skull to the back buffer and the stencil buffer cleared to 0 (denoted by light gray color). The black outlines drawn on the stencil buffer illustrate the relationship between the back buffer pixels and the stencil buffer pixels—they do not indicate any data drawn on the stencil buffer.



Back buffer



Stencil buffer

PLANAR MIRRORS Algorithm

1. Render the floor, walls, and skull to the back buffer as normal (but not the mirror). Note that this step does not modify the stencil buffer.

2. Clear the stencil buffer to 0. Figure in the last slide shows the back buffer and stencil buffer at this point (where we substitute a box for the skull to make the drawing simpler).

3. Render the mirror only to the stencil buffer.

Since we are only rendering the mirror to the stencil buffer, it follows that all the pixels in the stencil buffer will be 0 except for the pixels that correspond to the visible part of the mirror—they will have a 1.

We can disable color writes to the back buffer by creating a blend state that sets

```
D3D12_RENDER_TARGET_BLEND_DESC::RenderTargetWriteMask= 0;
```

and we can disable writes to the depth buffer by setting

```
D3D12_DEPTH_STENCIL_DESC::DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ZERO;
```

When rendering the mirror to the stencil buffer, we set the stencil test to always succeed

(D3D12_COMPARISON_ALWAYS) and specify that the stencil buffer entry should be replaced

(D3D12_STENCIL_OP_REPLACE) with 1 (StencilRef) if the test passes. If the depth test fails, we specify

D3D12_STENCIL_OP_KEEP so that the stencil buffer is not changed if the depth test fails

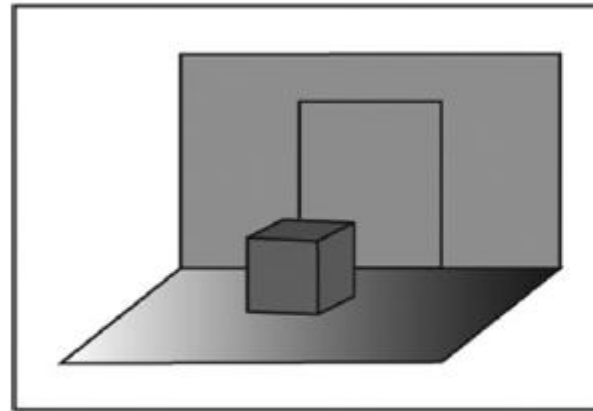
PLANAR MIRRORS Algorithm

Rendering the mirror to the stencil buffer, essentially marking the pixels in the stencil buffer that correspond to the visible parts of the mirror.

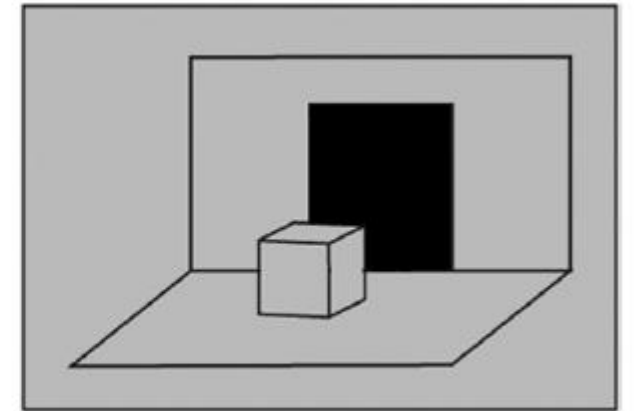
The solid black area on the stencil buffer denotes stencil entries set to 1.

Note that the area on the stencil buffer occluded by the box does not get set to 1 since it fails the depth test (the box is in front of that part of the mirror).

It is important to draw the mirror to the stencil buffer after we have drawn the skull so that pixels of the mirror occluded by the skull fail the depth test, and thus do not modify the stencil buffer. We do not want to turn on parts of the stencil buffer that are occluded; otherwise the reflection will show through the skull.



Back Buffer



Stencil Buffer

PLANAR MIRRORS Algorithm

Now we render the reflected skull to the back buffer and stencil buffer. But recall that we only will render to the back buffer if the stencil test passes. This time, we set the stencil test to only succeed if the value in the stencil buffer equals 1; this is done using a StencilRef of 1, and the stencil operator D3D12_COMPARISON_EQUAL.

5. Finally, we render the mirror to the back buffer as normal. However, in order for the skull reflection to show through (which lies behind the mirror), we need to render the mirror with transparency blending. If we did not render the mirror with transparency, the mirror would simply occlude the reflection since its depth is less than that of the reflection.

Assuming we have laid down the reflected skull pixels to the back buffer, we see 30% of the color comes from the mirror (source) and 70% of the color comes from the skull (destination).

To implement this, we simply need to define a new material instance for the mirror; we set the alpha channel of the diffuse component to 0.3 to make the mirror 30% opaque, and we render the mirror with the transparency blend state

```
auto icemirror = std::make_unique<Material>();

icemirror->Name = "icemirror";

icemirror->MatCBIndex = 2;

icemirror->DiffuseSrvHeapIndex = 2;

icemirror->DiffuseAlbedo = XMFLOAT4(1.0f, 1.0f, 1.0f, 0.3f);

icemirror->FresnelR0 = XMFLOAT3(0.1f, 0.1f, 0.1f);

icemirror->Roughness = 0.5f;
```

$$C = 0.3 \cdot C_{src} + 0.7 \cdot C_{dst}$$

Defining the Mirror Depth/Stencil States

we need two PSOs:

The first PSO `markMirrorsPsoDesc` is used when drawing the mirror to mark the mirror pixels on the stencil buffer.

We render the mirror only to the stencil buffer. We can disable color writes to the back buffer by creating a blend state that sets

`D3D12_RENDER_TARGET_BLEND_DESC::RenderTargetWriteMask = 0;`

and we can disable writes to the depth buffer by setting

`D3D12_DEPTH_STENCIL_DESC::DepthWriteMask =`

`D3D12_DEPTH_WRITE_MASK_ZERO;`

When rendering the mirror to the stencil buffer, we set the stencil test to always succeed (`D3D12_COMPARISON_ALWAYS`) and specify that the stencil buffer entry should be replaced (`D3D12_STENCIL_OP_REPLACE`) with 1 (`StencilRef`) if the test passes. If the depth test fails, we specify `D3D12_STENCIL_OP_KEEP` so that the stencil buffer is not changed if the depth test fails

```
void StencilApp::BuildPSOs()
{
    .....
    // PSO for marking stencil mirrors.

    CD3DX12_BLEND_DESC mirrorBlendState(D3D12_DEFAULT);
    mirrorBlendState.RenderTarget[0].RenderTargetWriteMask = 0;

    D3D12_DEPTH_STENCIL_DESC mirrorDSS;
    mirrorDSS.DepthEnable = true;
    mirrorDSS.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ZERO;
    mirrorDSS.DepthFunc = D3D12_COMPARISON_FUNC_LESS;
    mirrorDSS.StencilEnable = true;
    mirrorDSS.StencilReadMask = 0xff;
    mirrorDSS.StencilWriteMask = 0xff;

    mirrorDSS.FrontFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;
    mirrorDSS.FrontFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;
    mirrorDSS.FrontFace.StencilPassOp = D3D12_STENCIL_OP_REPLACE;
    mirrorDSS.FrontFace.StencilFunc = D3D12_COMPARISON_FUNC_ALWAYS;

    // We are not rendering backfacing polygons, so these settings do not matter.
    mirrorDSS.BackFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;
    mirrorDSS.BackFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;
    mirrorDSS.BackFace.StencilPassOp = D3D12_STENCIL_OP_REPLACE;
    mirrorDSS.BackFace.StencilFunc = D3D12_COMPARISON_FUNC_ALWAYS;

    D3D12_GRAPHICS_PIPELINE_STATE_DESC markMirrorsPsoDesc = opaquePsoDesc;
    markMirrorsPsoDesc.BlendState = mirrorBlendState;
    markMirrorsPsoDesc.DepthStencilState = mirrorDSS;
    ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&markMirrorsPsoDesc,
        IID_PPV_ARGS(&mPSOs["markStencilMirrors"])));
}
```

Defining the Mirror Depth/Stencil States

The second PSO:

`drawReflectionsPsoDesc` is used to draw the reflected skull so that it is only drawn into the visible parts of the mirror.

Now we render the reflected skull to the back buffer and stencil buffer. But recall that we only will render to the back buffer if the stencil test passes. This time, we set the stencil test to only succeed if the value in the stencil buffer equals 1; this is done using a `StencilRef` of 1, and the stencil operator `D3D12_COMPARISON_EQUAL`.

In this way, the reflected skull will only be rendered to areas that have a 1 in their corresponding stencil buffer entry.

```
// PSO for stencil reflections.  
//
```

```
D3D12_DEPTH_STENCIL_DESC reflectionsDSS;  
reflectionsDSS.DepthEnable = true;  
reflectionsDSS.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ALL;  
reflectionsDSS.DepthFunc = D3D12_COMPARISON_FUNC_LESS;  
reflectionsDSS.StencilEnable = true;  
reflectionsDSS.StencilReadMask = 0xff;  
reflectionsDSS.StencilWriteMask = 0xff;
```

```
reflectionsDSS.FrontFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;  
reflectionsDSS.FrontFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;  
reflectionsDSS.FrontFace.StencilPassOp = D3D12_STENCIL_OP_KEEP;  
reflectionsDSS.FrontFace.StencilFunc = D3D12_COMPARISON_FUNC_EQUAL;
```

```
// We are not rendering backfacing polygons, so these settings do not matter.  
reflectionsDSS.BackFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;  
reflectionsDSS.BackFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;  
reflectionsDSS.BackFace.StencilPassOp = D3D12_STENCIL_OP_KEEP;  
reflectionsDSS.BackFace.StencilFunc = D3D12_COMPARISON_FUNC_EQUAL;
```

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC drawReflectionsPsoDesc = opaquePsoDesc;  
drawReflectionsPsoDesc.DepthStencilState = reflectionsDSS;  
drawReflectionsPsoDesc.RasterizerState.CullMode = D3D12_CULL_MODE_BACK;  
drawReflectionsPsoDesc.RasterizerState.FrontCounterClockwise = true;  
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&drawReflectionsPsoDesc,  
IID_PPV_ARGS(&mPSOs["drawStencilReflections"])));
```

Drawing the Scene

The scene lighting also needs to get reflected when drawing the reflection. The lights are stored in a per-pass constant buffer, so we create an additional per-pass constant buffer that stores the reflected scene lighting.

```
PassConstants mMainPassCB;
```

```
PassConstants mReflectedPassCB;
```

```
void StencilApp::UpdateReflectedPassCB(const  
GameTimer& gt)
```

```
{  
    mReflectedPassCB = mMainPassCB;  
  
    XMVECTOR mirrorPlane = XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f); // xy plane  
    XMATRIX R = XMMatrixReflect(mirrorPlane);  
  
    // Reflect the lighting.  
    for(int i = 0; i < 3; ++i)  
    {  
        XMVECTOR lightDir = XMLoadFloat3(&mMainPassCB.Lights[i].Direction);  
        XMVECTOR reflectedLightDir = XMVector3TransformNormal(lightDir, R);  
        XMStoreFloat3(&mReflectedPassCB.Lights[i].Direction, reflectedLightDir);  
    }  
  
    // Reflected pass stored in index 1  
    auto currPassCB = mCurrFrameResource->PassCB.get();  
    currPassCB->CopyData(1, mReflectedPassCB);  
}
```

```
// Draw opaque items--floors, walls, skull.
```

```
auto passCB = mCurrFrameResource->PassCB->Resource();  
mCommandList->SetGraphicsRootConstantBufferView(2, passCB->GetGPUVirtualAddress());  
    DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Opaque]);
```

```
// Mark the visible mirror pixels in the stencil buffer with the value 1
```

```
mCommandList->OMSetStencilRef(1);  
mCommandList->SetPipelineState(mPSOs["markStencilMirrors"].Get());  
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Mirrors]);
```

```
// Draw the reflection into the mirror only (only for pixels where the stencil  
//buffer is 1). Note that we must supply a different per-pass constant buffer--one  
//with the lights reflected.
```

```
mCommandList->SetGraphicsRootConstantBufferView(2, passCB->GetGPUVirtualAddress() +  
1 * passCBByteSize);  
mCommandList->SetPipelineState(mPSOs["drawStencilReflections"].Get());  
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Reflected]);
```

```
// Restore main pass constants and stencil ref.
```

```
mCommandList->SetGraphicsRootConstantBufferView(2, passCB->GetGPUVirtualAddress());  
mCommandList->OMSetStencilRef(0);
```

```
// Draw mirror with transparency so reflection blends through.
```

```
mCommandList->SetPipelineState(mPSOs["transparent"].Get());  
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Transparent]);
```

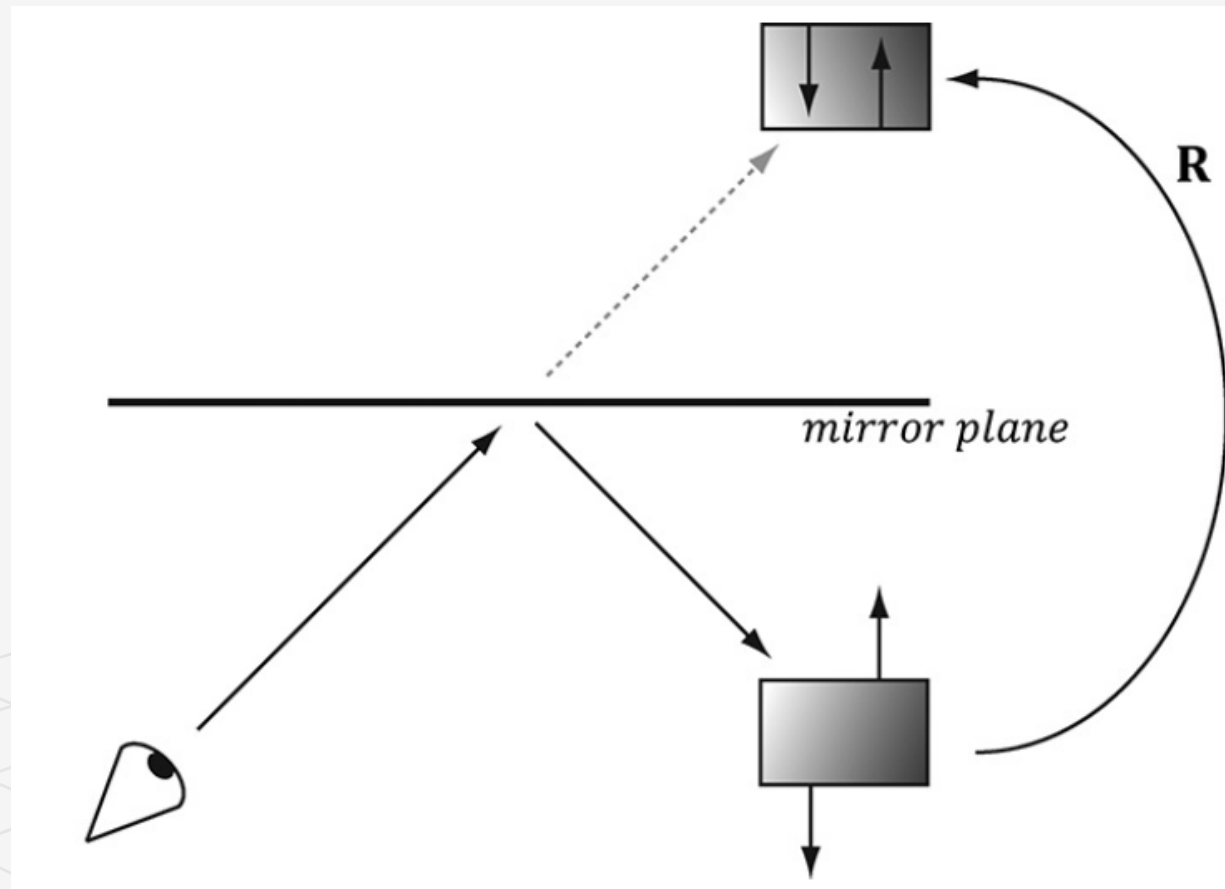
Winding Order and Reflections

When a triangle is reflected across a plane, its winding order does not reverse, and thus, its face normal does not reverse. Hence, outward facing normals become inward facing normals after reflection.

To correct this, we tell Direct3D to interpret triangles with a counterclockwise winding order as front-facing and triangles with a clockwise winding order as back-facing.

This effectively reflects the normal directions so that they are outward facing after reflection. We reverse the winding order convention by setting the following rasterizer properties in the PSO:

```
drawReflectionsPsoDesc.RasterizerState.FrontCounterClockwise = true;
```

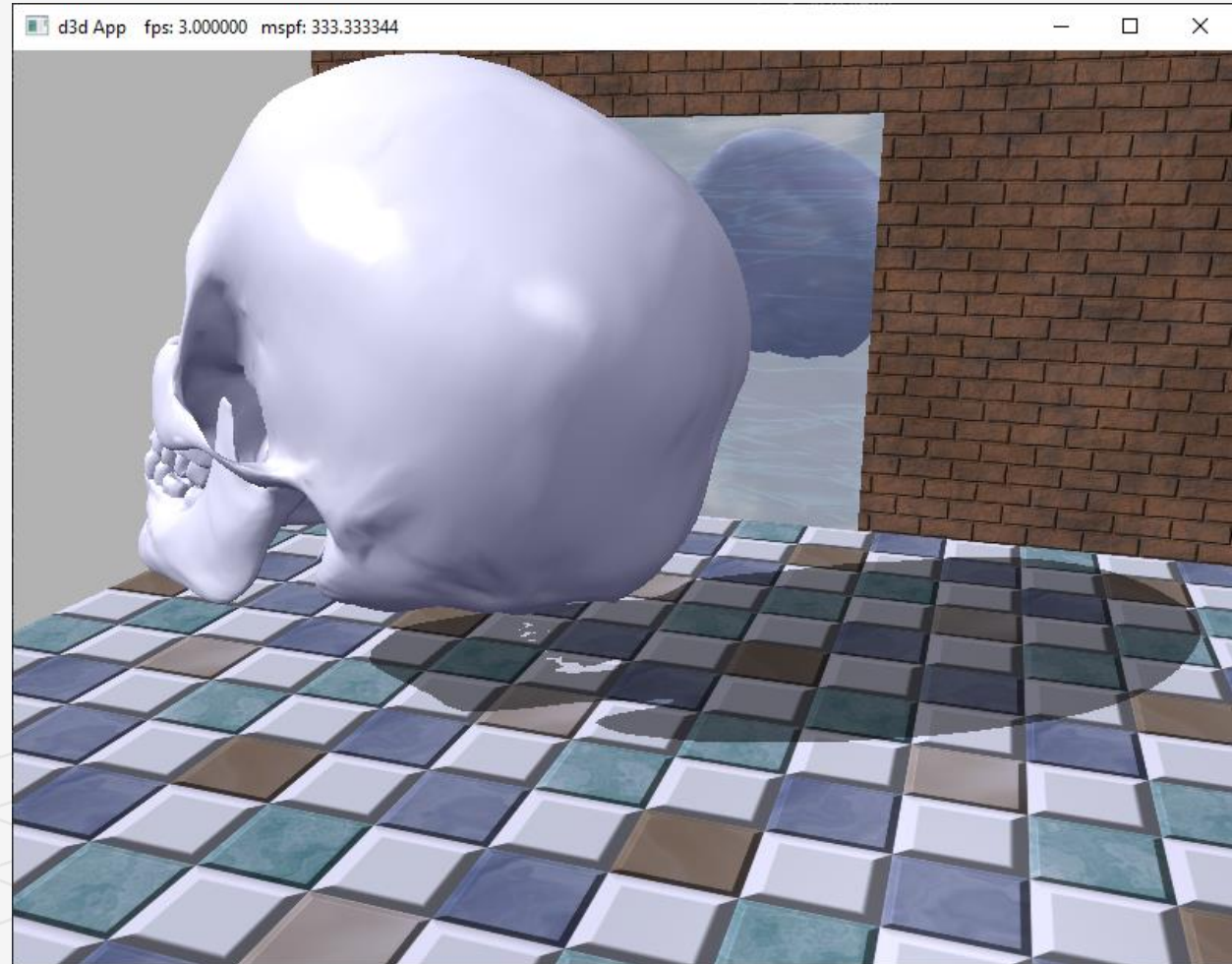


IMPLEMENTING PLANAR SHADOWS

Shadows aid in our perception of where light is being emitted in a scene and ultimately makes the scene more realistic. In this section, we will show how to implement planar shadows; that is, shadows that lie on a plane.

To implement planar shadows, we must first find the shadow an object casts to a plane and model it geometrically so that we can render it. This can easily be done with some 3D math.

We then render the triangles that describe the shadow with a black material at 50% transparency.



Parallel Light Shadows

The shadow cast with respect to a parallel light source.

Given a parallel light source with direction \mathbf{L} , the light ray that passes through a vertex \mathbf{p} is given by $\mathbf{r}(t) = \mathbf{p} + t\mathbf{L}$. The intersection of the ray $\mathbf{r}(t)$ with the shadow plane (\mathbf{n}, d) gives \mathbf{s} .

The set of intersection points found by shooting a ray through each of the object's vertices with the plane defines the projected geometry of the shadow. For a vertex \mathbf{p} , its shadow projection is given by

$$\mathbf{s} = \mathbf{r}(t_s) = \mathbf{p} - \frac{\mathbf{n} \cdot \mathbf{p} + d}{\mathbf{n} \cdot \mathbf{L}} \mathbf{L}$$

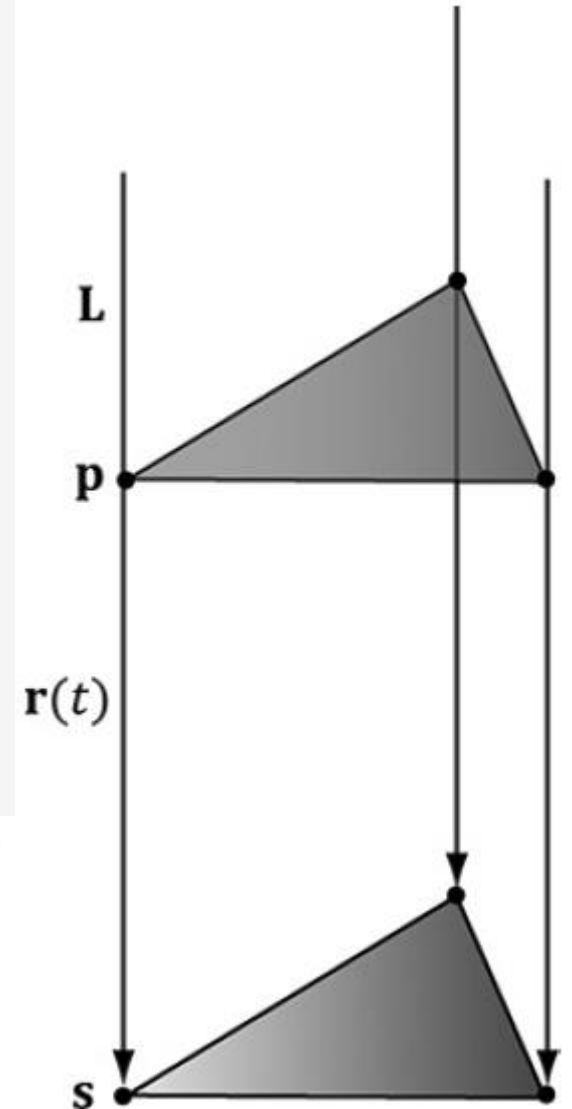
The details of the ray/plane intersection test are given in Appendix C.

Equation can be written in terms of matrices. We call

This 4×4 matrix the directional shadow matrix and

denote it by \mathbf{S}_{dir}

$$\mathbf{s}' = \begin{bmatrix} p_x & p_y & p_z & 1 \end{bmatrix} \begin{bmatrix} \mathbf{n} \cdot \mathbf{L} - L_x n_x & -L_y n_x & -L_z n_x & 0 \\ -L_x n_y & \mathbf{n} \cdot \mathbf{L} - L_y n_y & -L_z n_y & 0 \\ -L_x n_z & -L_y n_z & \mathbf{n} \cdot \mathbf{L} - L_z n_z & 0 \\ -L_x d & -L_y d & -L_z d & \mathbf{n} \cdot \mathbf{L} \end{bmatrix}$$



Point Light Shadows

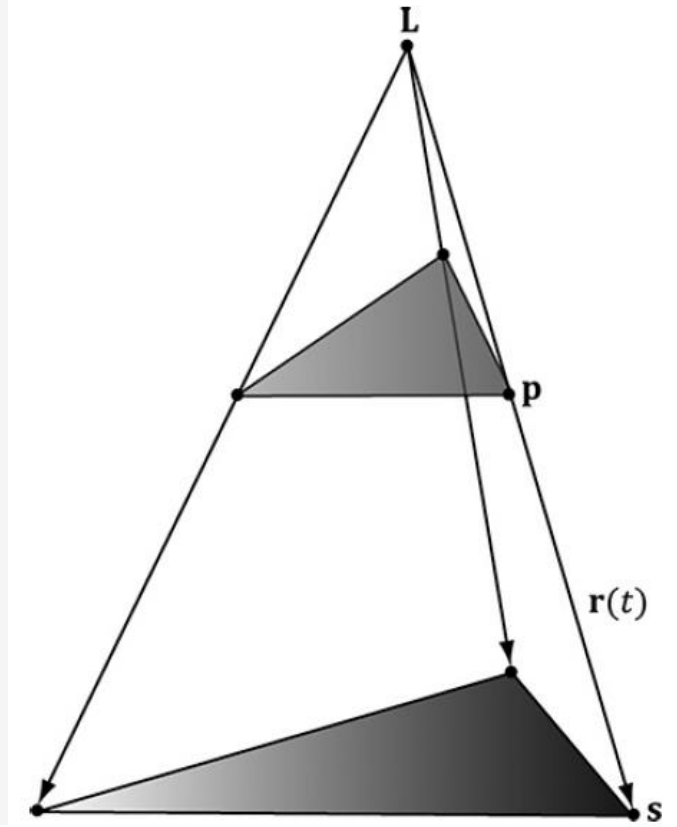
The shadow cast with respect to a point light source.

The light ray from a point light through any vertex \mathbf{p} is given by $\mathbf{r}(t) = \mathbf{p} + t(\mathbf{p} - \mathbf{L})$. The intersection of the ray $\mathbf{r}(t)$ with the shadow plane (\mathbf{n}, d) gives \mathbf{s} . The set of intersection points found by shooting a ray through each of the object's vertices with the plane defines the projected geometry of the shadow. For a vertex \mathbf{p} , its shadow projection is given by

$$\mathbf{s} = \mathbf{r}(t_s) = \mathbf{p} - \frac{\mathbf{n} \cdot \mathbf{p} + d}{\mathbf{n} \cdot (\mathbf{p} - \mathbf{L})}(\mathbf{p} - \mathbf{L})$$

Equation can also be written by a matrix equation:

$$\mathbf{S}_{point} = \begin{bmatrix} \mathbf{n} \cdot \mathbf{L} + d - L_x n_x & -L_y n_x & -L_z n_x & -n_x \\ -L_x n_y & \mathbf{n} \cdot \mathbf{L} + d - L_y n_y & -L_z n_y & -n_y \\ -L_x n_z & -L_y n_z & \mathbf{n} \cdot \mathbf{L} + d - L_z n_z & -n_z \\ -L_x d & -L_y d & -L_z d & \mathbf{n} \cdot \mathbf{L} \end{bmatrix}$$



General Shadow Matrix

Using homogeneous coordinates, it is possible to create a general shadow matrix that works for both point and directional lights.

1. If $L_w = 0$ then \mathbf{L} describes the direction towards the infinitely far away light source

(i.e., the opposite direction the parallel light rays travel).

2. If $L_w = 1$ then \mathbf{L} describes the location of the point light.

It is easy to see that \mathbf{S} reduced to \mathbf{S}_{dir} if $L_w = 0$ and \mathbf{S} reduces to \mathbf{S}_{point} for $L_w = 1$.

Then we represent the transformation from a vertex p to its projection s with the following *shadow matrix*:

$$\mathbf{S} = \begin{bmatrix} \mathbf{n} \cdot \mathbf{L} + dL_w - L_x n_x & -L_y n_x & -L_z n_x & -L_w n_x \\ -L_x n_y & \mathbf{n} \cdot \mathbf{L} + dL_w - L_y n_y & -L_z n_y & -L_w n_y \\ -L_x n_z & -L_y n_z & \mathbf{n} \cdot \mathbf{L} + dL_w - L_z n_z & -L_w n_z \\ -L_x d & -L_y d & -L_z d & \mathbf{n} \cdot \mathbf{L} \end{bmatrix}$$

The DirectX math library provides the following function to build the shadow matrix given the plane we wish to project the shadow into and a vector describing a parallel light if $w = 0$ or a point light if $w = 1$:

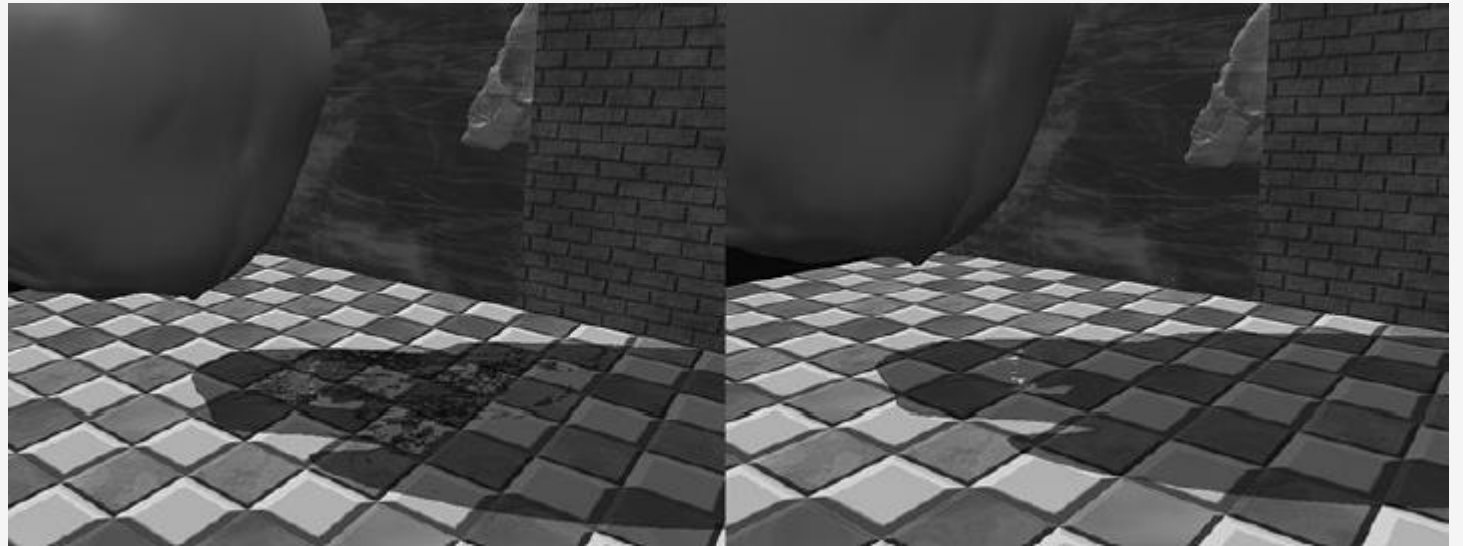
```
inline XMMATRIX XM_CALLCONV XMMatrixShadow
(
    FXMVECTOR ShadowPlane,
    FXMVECTOR LightPosition
)
```

Double Blending

When we flatten out the geometry of an object onto the plane to describe its shadow, it is possible (and in fact likely) that two or more of the flattened triangles will overlap.

When we render the shadow with transparency (using blending), these areas that have overlapping triangles will get blended multiple times and thus appear darker.

Notice the darker “acne” areas of the shadow in the left image; these correspond to areas where parts of the flattened skull overlapped, thus causing a “double blend.” The image on the right shows the shadow rendered correctly, without double blending.




Using the Stencil Buffer to Prevent Double Blending

We can solve this problem using the stencil buffer.

1. Assume the stencil buffer pixels where the shadow will be rendered have been cleared to 0. This is true in our mirror demo because we are only casting a shadow onto the ground plane, and we only modified the mirror stencil buffer pixels.
2. Set the stencil test to only accept pixels if the stencil buffer has an entry of 0. If the stencil test passes, then we increment the stencil buffer value to 1.

The first time we render a shadow pixel, the stencil test will pass because the stencil buffer entry is 0. However, when we render this pixel, we also increment the corresponding stencil buffer entry to 1. Thus, if we attempt to *overwrite to an area that has already been rendered to (marked in the stencil buffer with a value of 1), the stencil test will fail. This prevents drawing over the same pixel more than once, and thus prevents double blending.*



Shadow Code

We define a shadow material used to color the shadow that is just a 50% transparent black material:

```
auto shadowMat = std::make_unique<Material>();

shadowMat->Name = "shadowMat";

shadowMat->MatCBIndex = 4;

shadowMat->DiffuseSrvHeapIndex = 3;

shadowMat->DiffuseAlbedo = XMFLOAT4(0.0f, 0.0f, 0.0f, 0.5f);

shadowMat->FresnelR0 = XMFLOAT3(0.001f, 0.001f, 0.001f);

shadowMat->Roughness = 0.0f;
```

In order to prevent double blending we set up the following PSO with depth/stencil state:

```
// We are going to draw shadows with transparency, so base it off the transparency description.
D3D12_DEPTH_STENCIL_DESC shadowDSS;
shadowDSS.DepthEnable = true;
shadowDSS.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ALL;
shadowDSS.DepthFunc = D3D12_COMPARISON_FUNC_LESS;
shadowDSS.StencilEnable = true;
shadowDSS.StencilReadMask = 0xff;
shadowDSS.StencilWriteMask = 0xff;

shadowDSS.FrontFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;
shadowDSS.FrontFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;
shadowDSS.FrontFace.StencilPassOp = D3D12_STENCIL_OP_INCR;
shadowDSS.FrontFace.StencilFunc = D3D12_COMPARISON_FUNC_EQUAL;

// We are not rendering backfacing polygons, so these settings do not matter.
shadowDSS.BackFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;
shadowDSS.BackFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;
shadowDSS.BackFace.StencilPassOp = D3D12_STENCIL_OP_INCR;
shadowDSS.BackFace.StencilFunc = D3D12_COMPARISON_FUNC_EQUAL;

D3D12_GRAPHICS_PIPELINE_STATE_DESC shadowPsoDesc = transparentPsoDesc;
shadowPsoDesc.DepthStencilState = shadowDSS;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&shadowPsoDesc, IID_PPV_ARGS(&mPSOs["shadow"])));
```


Shadow Code

We then draw the skull shadow with the shadow PSO with a StencilRef value of 0:

```
mCommandList->OMSetStencilRef(0);

// Draw mirror with transparency so reflection
blends through.

mCommandList-
>SetPipelineState(mPSOs["transparent"].Get());

DrawRenderItems(mCommandList.Get(),
mRitemLayer[(int)RenderLayer::Transparent]);

// Draw shadows

mCommandList-
>SetPipelineState(mPSOs["shadow"].Get());

DrawRenderItems(mCommandList.Get(),
mRitemLayer[(int)RenderLayer::Shadow]);
```

where the skull shadow render-item's world matrix is computed like so:

```
void StencilApp::OnKeyboardInput(const GameTimer& gt)
{
    // Allow user to move skull.
    const float dt = gt.DeltaTime();

    .....
    // Update shadow world matrix.
    XMVECTOR shadowPlane = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f); // xz plane
    XMVECTOR toMainLight = -XMLoadFloat3(&mMainPassCB.Lights[0].Direction);
    XMMATRIX S = XMMatrixShadow(shadowPlane, toMainLight);
    XMMATRIX shadowOffsetY = XMMatrixTranslation(0.0f, 0.001f, 0.0f);
    XMStoreFloat4x4(&mShadowedSkullRitem->World, skullWorld * S * shadowOffsetY);
```