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

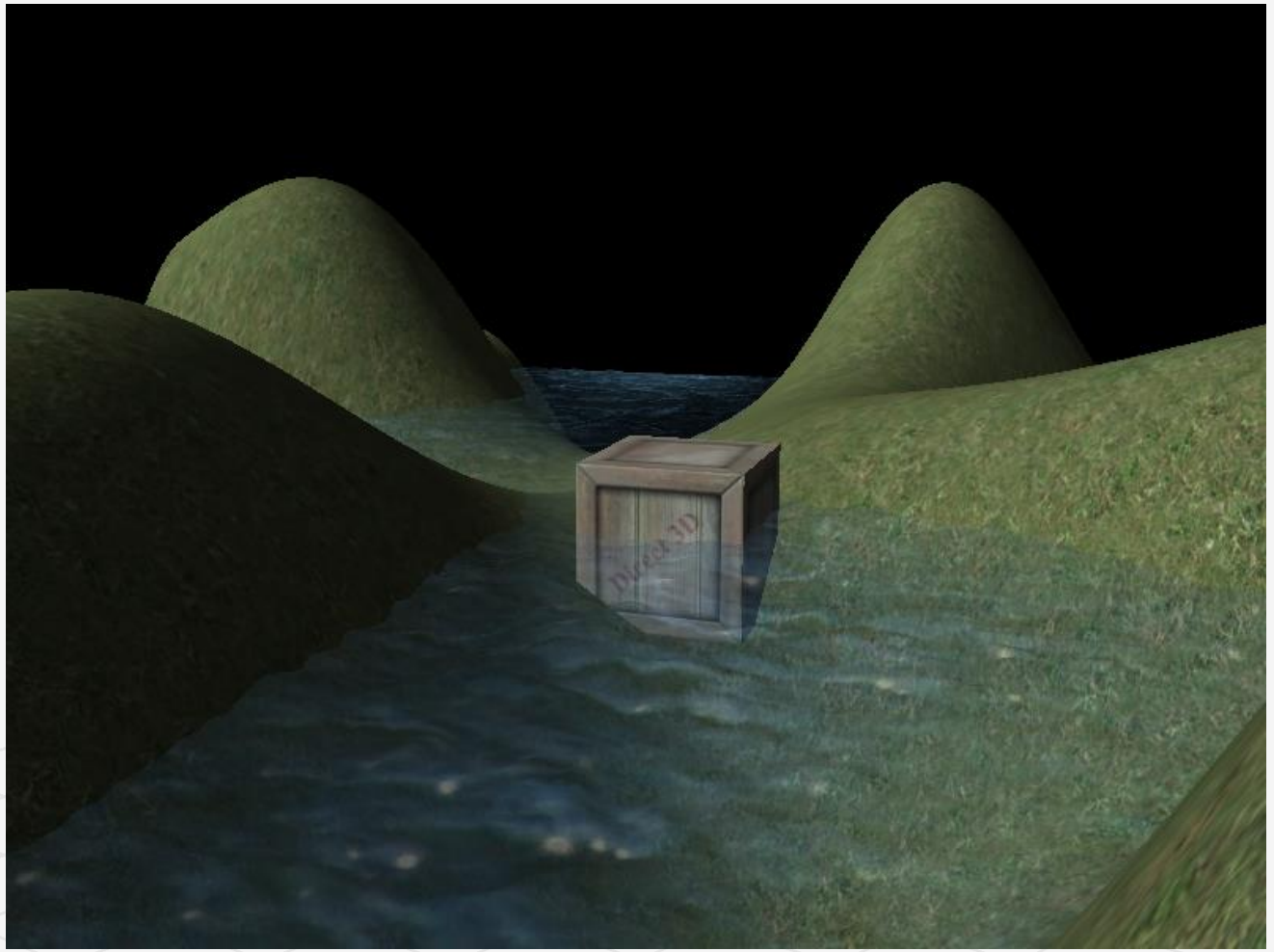
Blending

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 techniques allow us to blend (combine) the pixels that we are currently rasterizing (*source* pixels) with the pixels that were previously rasterized to the back buffer (*destination* pixels).

This technique enables us to render semi-transparent objects such as water and glass.



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.

The blend factors are ignored in the min/max operation.

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

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;
```

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

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

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

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

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

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.

The reason for separating RGB from alpha is simply so that we can process them independently which allows for a greater variety of possibilities.

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

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

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 instead of the traditional blending equations.

you can't 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

By setting different combinations for the source and destination blend factors along with different blend operators, dozens of different blending effects may be achieved.

You will need to experiment with others to get a feel of what they do.

See the D3D12_BLEND enumerated type in the SDK documentation for some additional advanced blend factors.

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

The following list describes the basic blend factors, which apply to both $\mathbf{F}_{src}$ and $\mathbf{F}_{dst}$.

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

ID3D12GraphicsCommandList::OMSetBlendFactor method sets the blend factor that modulate values for a pixel shader, render target, or both.

D3D12_BLEND_BLEND_FACTOR: **F** = (*r*, *g*, *b*) and *F* = *a*, where the color (*r*, *g*, *b*, *a*) is supplied to the first parameter of the ID3D12GraphicsCommandList::OMSetBlendFactor method.

This allows you to specify the blend factor color to use directly; however, it is constant until you change the blend state.

```
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).*

Blending is not free and does require additional per-pixel work, so only enable it if you need it, and turn it off when you are done.

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;
```

Blending operations are performed on every pixel shader output (RGBA value) before the output value is written to a render target. If multisampling is enabled, blending is done on each multisample; otherwise, blending is performed on each pixel.

1. AlphaToCoverageEnable: Specify true to enable alpha-to-coverage, which is a multisampling technique useful when rendering foliage or gate textures. Specify false to disable alpha-to-coverage. Alpha-to-coverage requires multisampling to be enabled (i.e., the back and depth buffer were created with multisampling).

2. IndependentBlendEnable: Direct3D supports rendering to up to eight render targets simultaneously. This flag specifies whether to enable independent blending in simultaneous render targets. Set to **TRUE** to enable independent blending. If set to **FALSE**, only the **RenderTarget**[0] members are used; **RenderTarget**[1..7] are ignored.

3. RenderTarget: An array of 8 D3D12_RENDER_TARGET_BLEND_DESC elements, where the *i*th element describes how blending is done for the *i*th simultaneous render target.

D3D12_RENDER_TARGET_BLEND_DESC

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

1. BlendEnable: Specify true to enable blending and false to disable it. Note that BlendEnable and LogicOpEnable cannot both be set to true
2. LogicOpEnable: Specify true to enable a logic blend operation.
3. SrcBlend: A member of the D3D12_BLEND
4. DestBlend: A member of the D3D12_BLEND
5. BlendOp: A member of the D3D12_BLEND_OP
6. SrcBlendAlpha: A member of the D3D12_BLEND
7. DestBlendAlpha: A member of the D3D12_BLEND
8. BlendOpAlpha: A member of the D3D12_BLEND_OP
9. LogicOp: A member of the D3D12_LOGIC_OP enumerated type

10. RenderTargetWriteMask: A combination of one or more of the RGBA flags. These flags control which color channels in the back buffer are written to after blending. For example, you could disable writes to the RGB channels, and only write to the alpha channel, by specifying D3D12_COLOR_WRITE_ENABLE_ALPHA.

```
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_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,

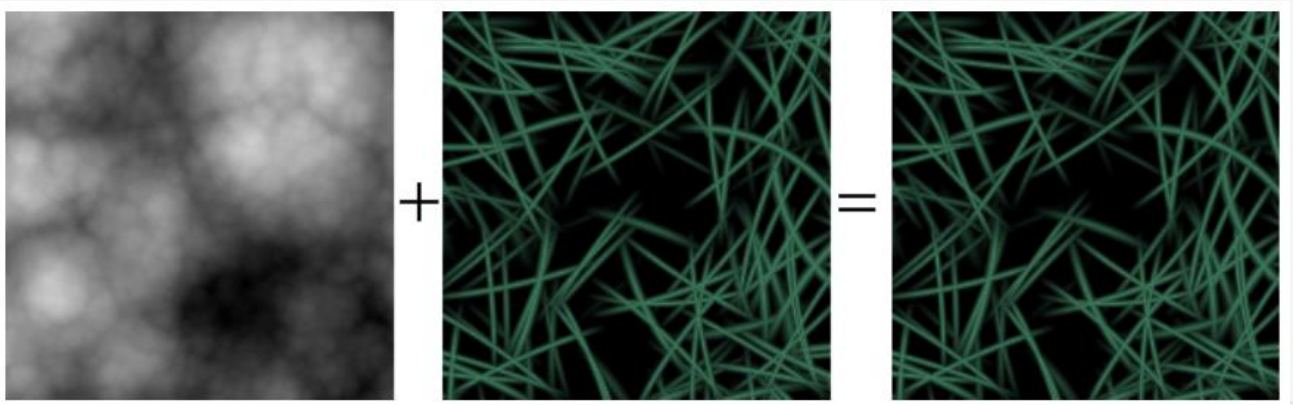
the blend operator: D3D12_BLEND_OP_ADD.

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

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

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

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 = D3D12_BLEND_OP_ADD.

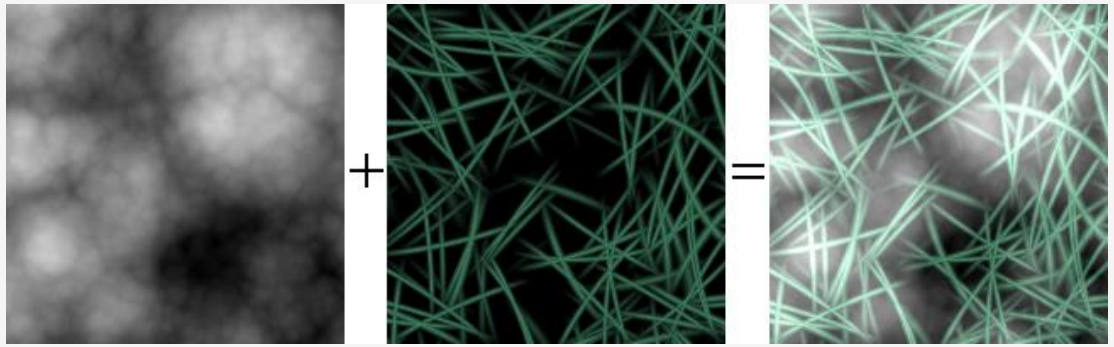
The following adds source and destination color. Adding creates a brighter image since color is being added.

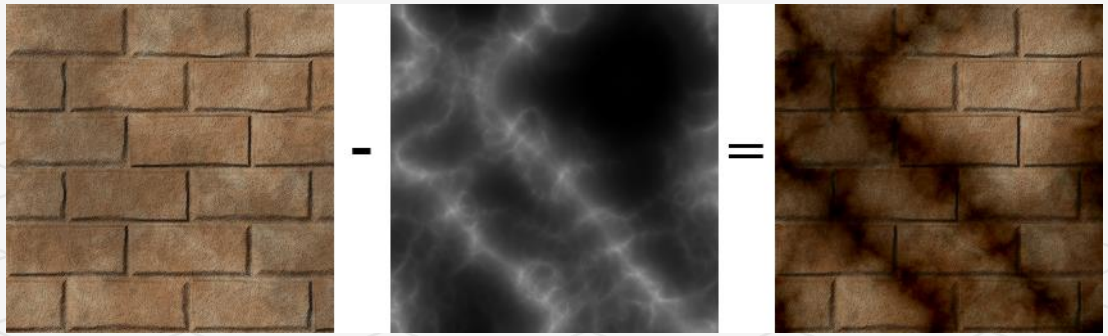
We can subtract source pixels from destination pixels by using the above blend factors and replacing the blend operation with D3D12_BLEND_OP_SUBTRACT

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

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

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





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,

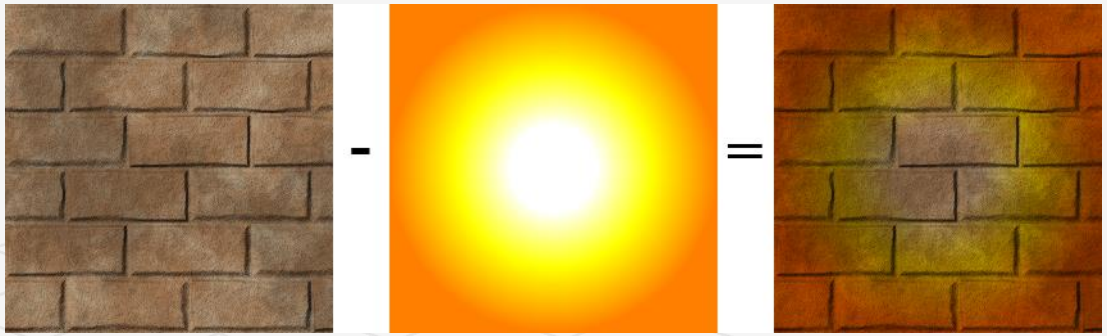
the blend operator: D3D12_BLEND_OP_ADD.

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

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

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

Orders in drawing

Using the blending methods, we can draw transparent objects like the one in Land and Waves figure.

It should be noted that with this blending method, the order that you draw the objects matters. We use the following rule:

1. Draw objects that do not use blending first.

2. Sort the objects that use blending by their distance from the camera.

3. Draw the objects that use blending in a back-to front order. The reason for the back-to-front draw order is so that objects are blended with the objects spatially behind them.

If an object is transparent, we can see through it to see the scene behind it. So it is necessary that all the pixels behind the transparent object have already been written to the back buffer, so that we can blend the transparent source pixels with the destination pixels of the scene behind it.

For the “No Color White” blending method, draw order does not matter since it simply prevents source pixel writes to the back buffer.

For the “additive/subtractive/multiplicative” blending methods, we still draw non-blended objects first and blended objects last; this is because we want to first lay all the non-blended geometry onto the back buffer before we start blending. However, we do not need to sort the objects that use blending. This is because the operations are commutative.

If you start with a back buffer pixel color **B**, and then do *n* additive/subtractive/multiplicative blends to that pixel,

the order does not matter:

$$\mathbf{B}' = \mathbf{B} + \mathbf{C}_0 + \mathbf{C}_1 + \ldots + \mathbf{C}_{n-1}$$
$$\mathbf{B}' = \mathbf{B} - \mathbf{C}_0 - \mathbf{C}_1 - \ldots - \mathbf{C}_{n-1}$$
$$\mathbf{B}' = \mathbf{B} \otimes \mathbf{C}_0 \otimes \mathbf{C}_1 \otimes \ldots \otimes \mathbf{C}_{n-1}$$

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.

Note that we only disable depth writes while drawing the objects in *S* (the set of objects drawn with additive blending).

Depth reads and the depth test are still enabled.

This is so that non-blended geometry (which is drawn before blended geometry) will still obscure blended geometry behind it.

ALPHA CHANNELS

- Source alpha components can be used in RGB blending to control transparency.
  - The source color used in the blending equation comes from the pixel shader.
- We return the diffuse material's alpha value as the alpha output of the pixel shader
- Therefore 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

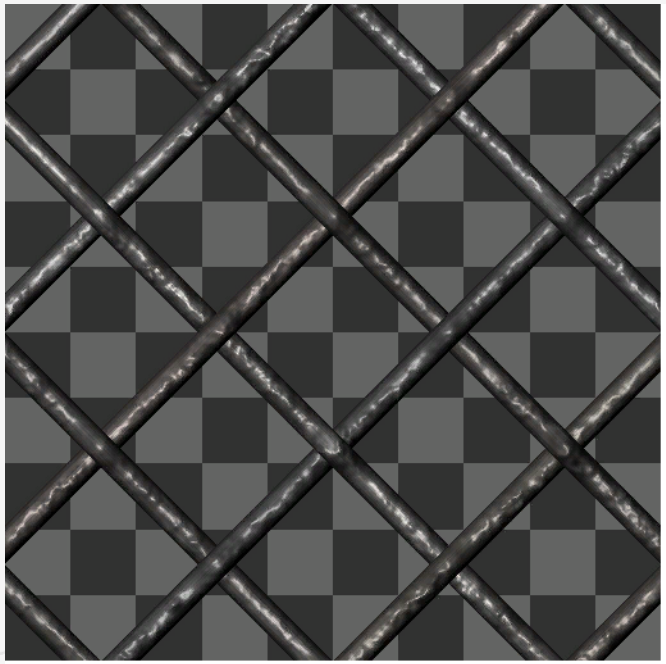
▪ Sometimes we want to completely reject a source pixel from being further processed.

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

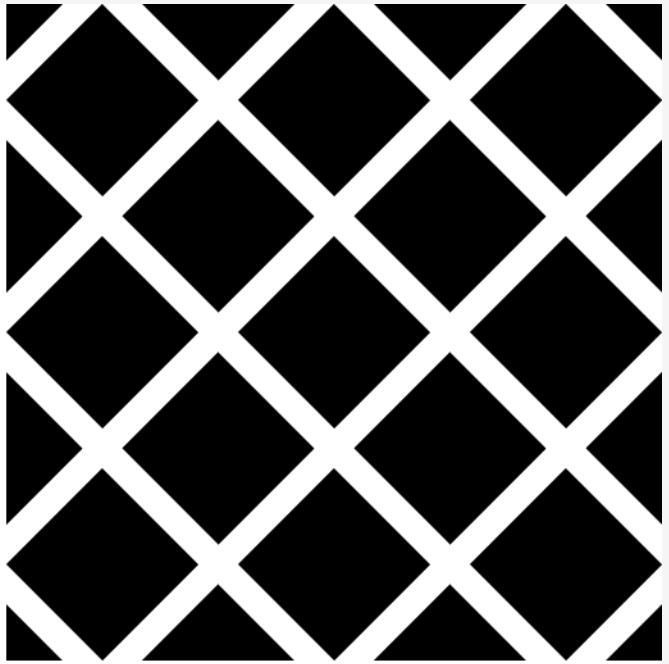
▪ This function can only be called in a pixel shader, and it discards the current pixel from further processing if x < 0

▪ 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.

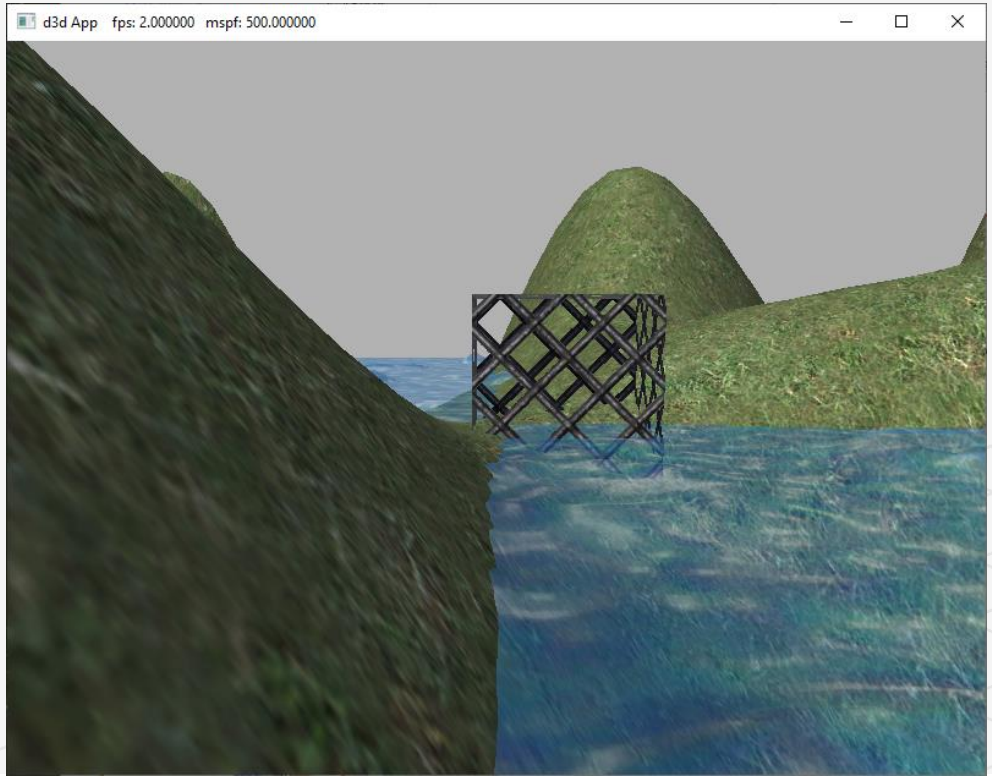
Note that the same result can be obtained using blending, but this is more efficient by discarding a pixel early from the pixel shader, the remaining pixel shader instructions can be skipped (no point in doing the calculations for a discarded pixel).

```
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.



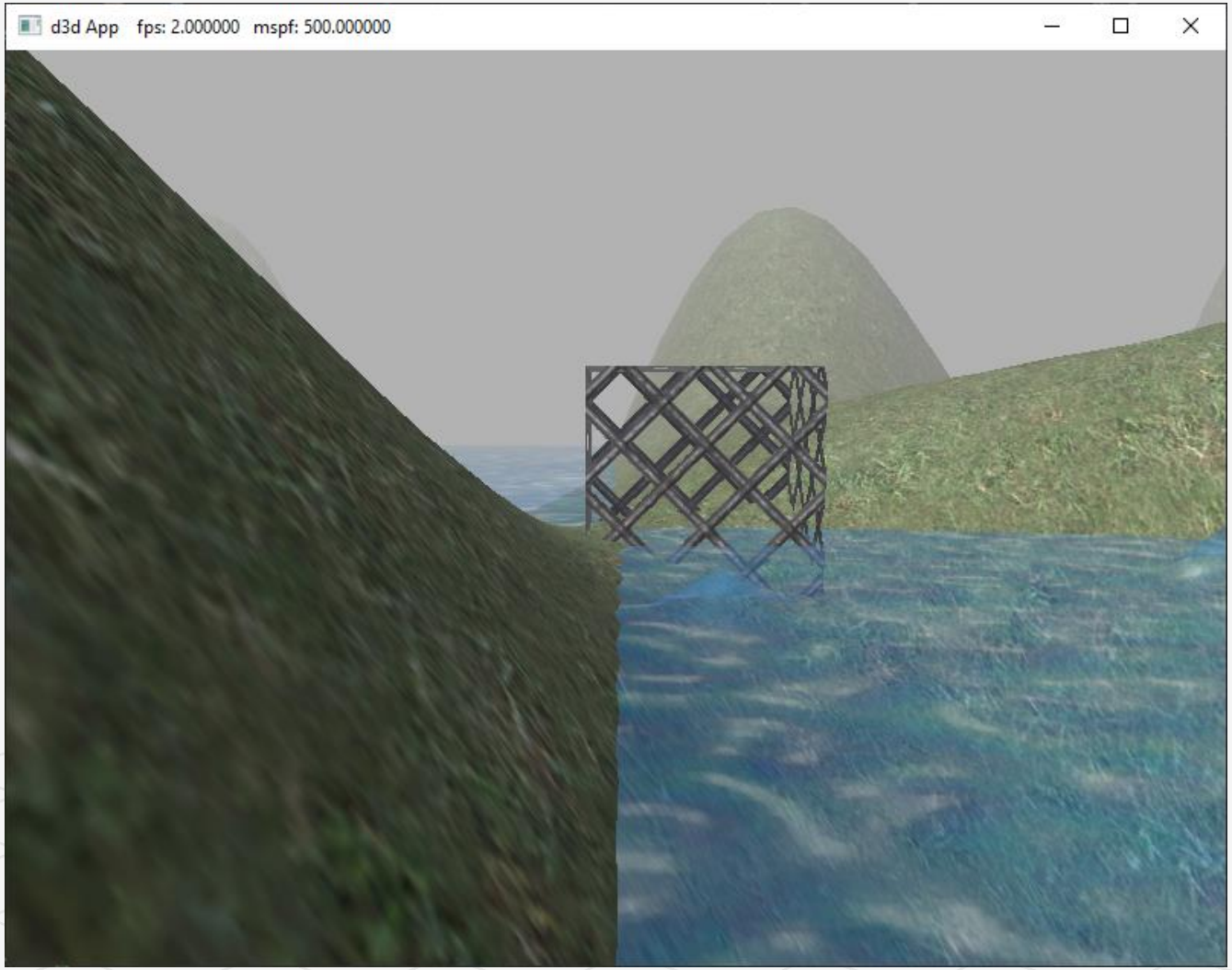

```
void BlendApp::BuildPSOs()
{
    D3D12_GRAPHICS_PIPELINE_STATE_DESC opaquePsoDesc;

// 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

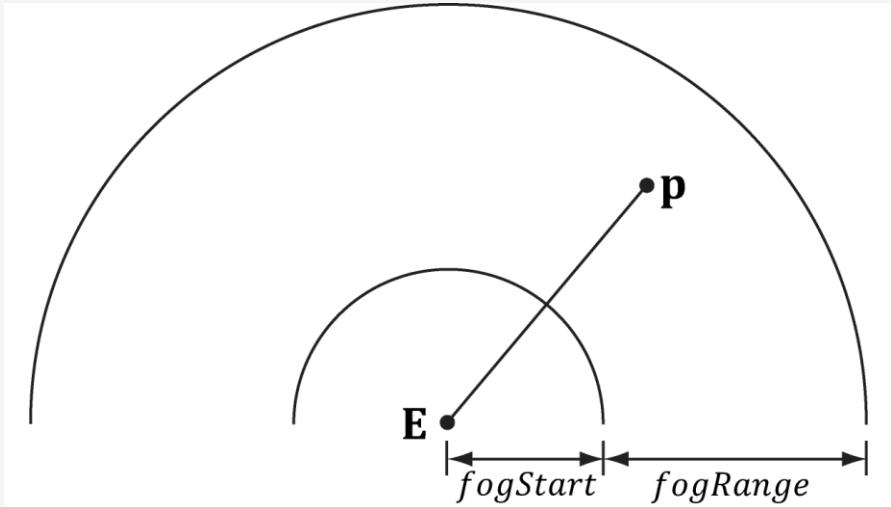
We specify a fog color, a fog start distance from the camera and a fog range (i.e., the range from the fog start distance until the fog completely hides any objects).

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:

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].



$$foggedColor = litColor + s\left(fogColor - litColor\right)$$
$$= \left(1 - s\right) \cdot litColor + s \cdot fogColor$$

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

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.

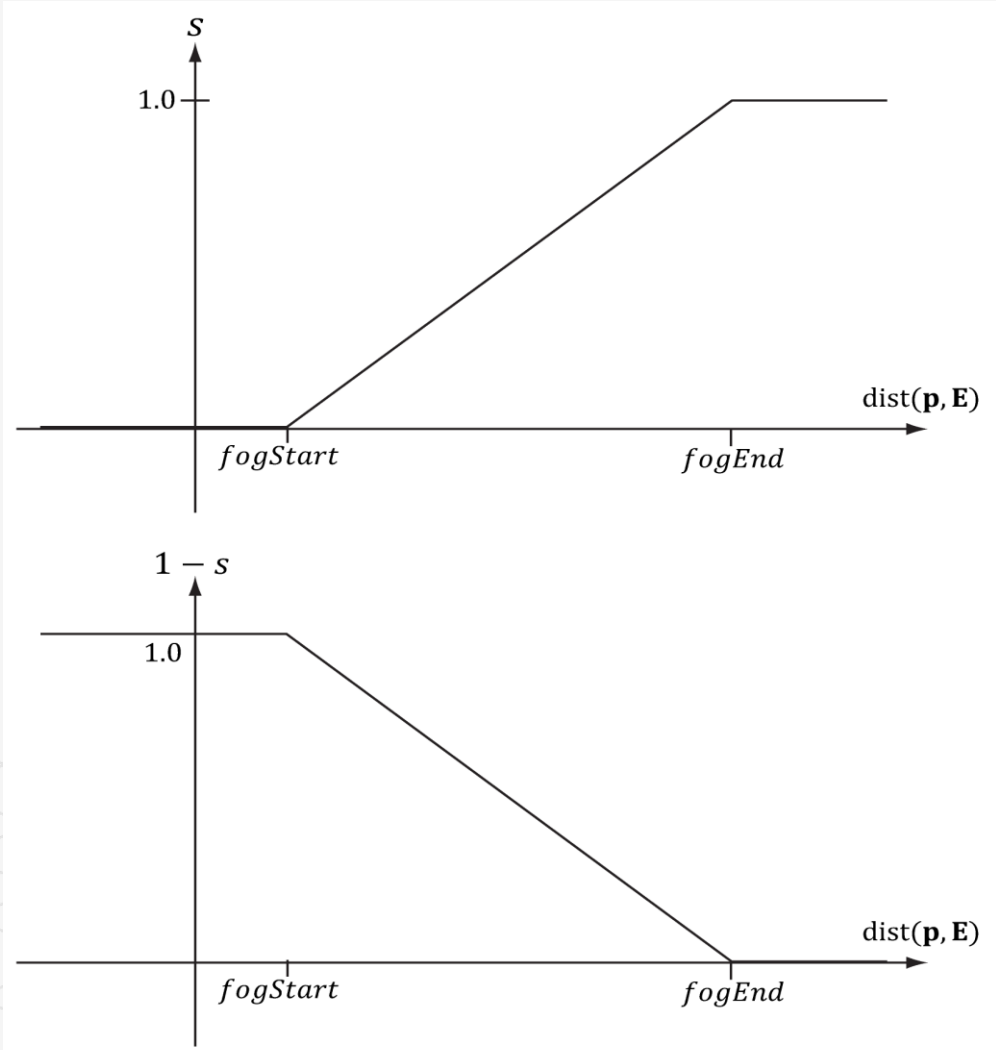
The fog does not start affecting the color until the distance from the camera is at least that of *fogStart.*

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
```

Linear interpolation **lerp(*x, y, s*)** is based on the following formula: x*(1-s) + y*s which can equivalently be written as x + s(y-x).



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_1");
mShaders["opaquePS"] = d3dUtil::CompileShader(L"Shaders\\Default.hlsl", defines,
"PS", "ps_5_1");
mShaders["alphaTestedPS"] = d3dUtil::CompileShader(L"Shaders\\Default.hlsl",
alphaTestDefines, "PS", "ps_5_1");
```

Stenciling

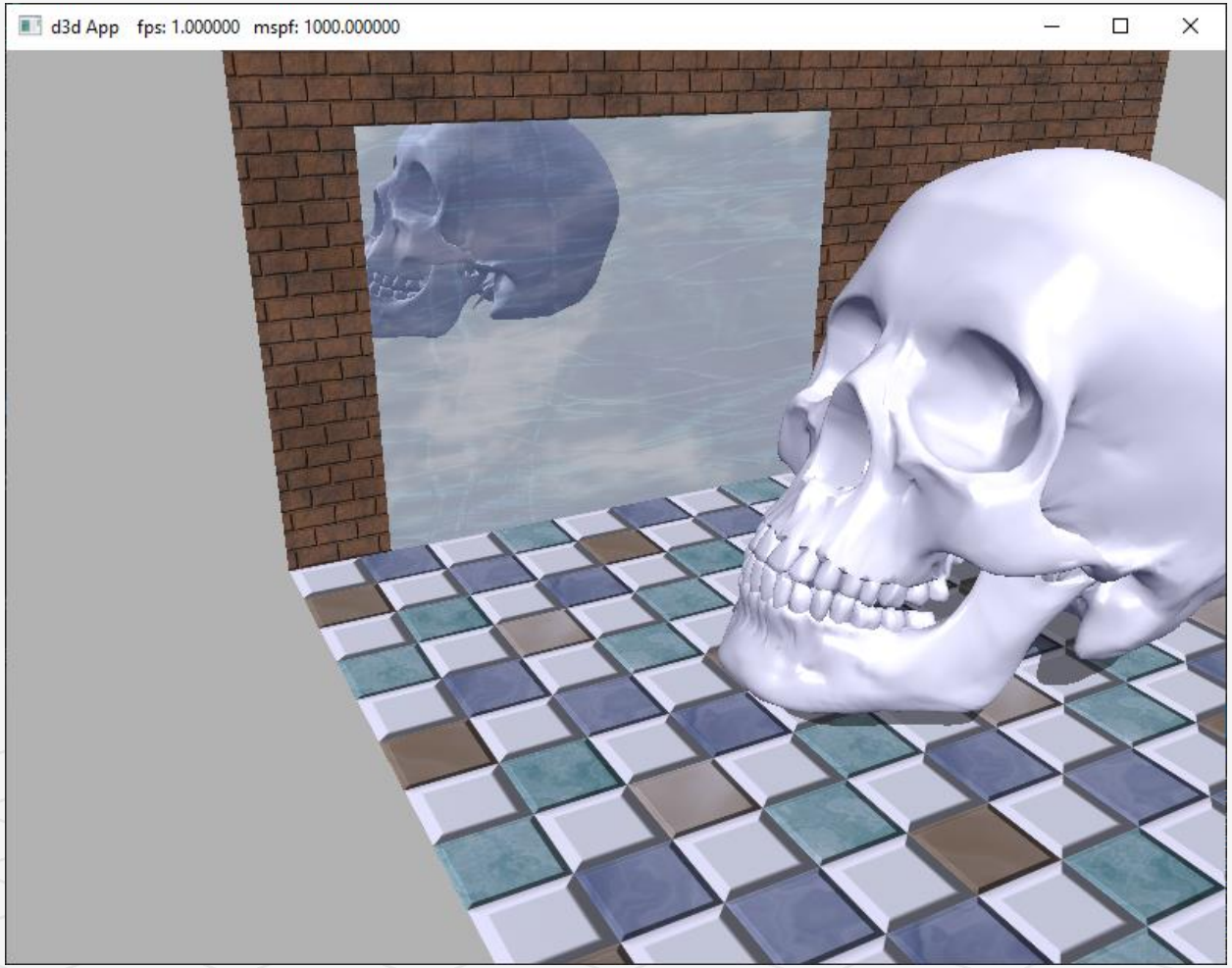
Objectives:

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

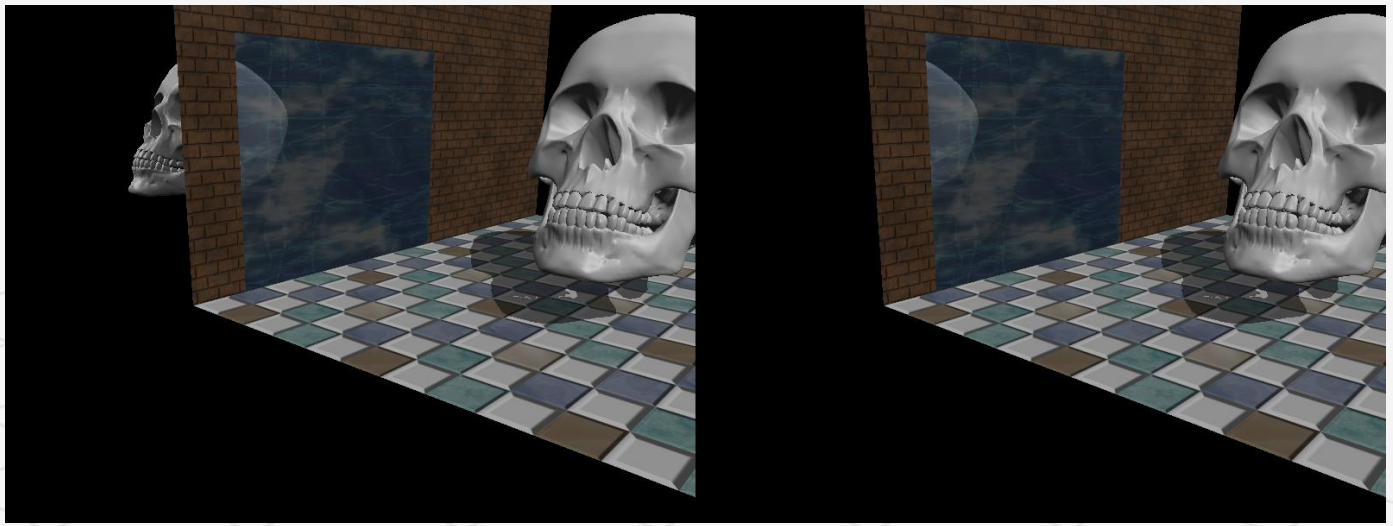
The stencil buffer is an off-screen buffer we can use to achieve some special effects.

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, therefore 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.



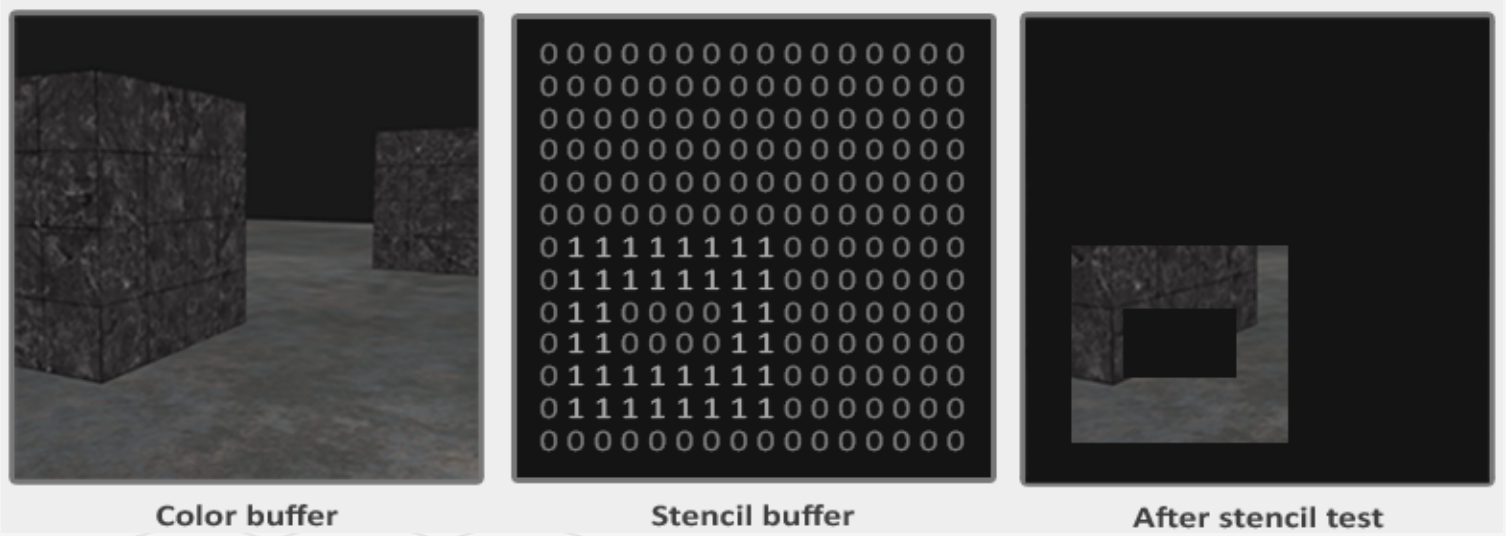
Stencil Testing

A depth/stencil buffer (usually) contains 8 bits per stencil value that amounts to a total of 256 different stencil values per pixel/fragment.

We can set these stencil and then we can discard or keep fragments whenever a particular fragment has a certain stencil value.

By changing the content of the stencil buffer while we're rendering, we're *writing* to the stencil buffer. In the same (or following) render iteration(s) we can then *read* these values to discard or pass certain fragments. When using stencil buffers, the general outline is usually as follows:

- Enable writing to the stencil buffer.
- Render objects, updating the content of the stencil buffer.
- Disable writing to the stencil buffer.
- Render (other) objects, this time discarding certain fragments based on the content of the stencil buffer.



D3D12_DEPTH_STENCIL_DESC

The depth/stencil state is described by filling out a D3D12_DEPTH_STENCIL_DESC instance.

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: Identify a portion of the depth-stencil buffer for reading stencil data. 0xff: does not mask any bits

4. StencilWriteMask: Identify a portion of the depth-stencil buffer for writing stencil data. 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_STENCILOP_DESC FrontFace; //discuss in next slides
    D3D12_DEPTH_STENCILOP_DESC BackFace;  //discuss in next slides
    } 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

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: Describes the CPU descriptor handle that represents the start of the heap for the depth stencil to be cleared.

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);
```

Create the Depth/Stencil Buffer and View

1. The depth/stencil buffer is just a 2D texture that stores the depth information of the nearest visible objects and stencil information if using stenciling).

A texture is a kind of GPU resource, so we create one by filling out a D3D12_RESOURCE_DESC structure describing the texture resource, and then calling the ID3D12Device::CreateCommittedResource method.

GPU resources live in heaps, which are essentially blocks of GPU memory with certain properties. The ID3D12Device::CreateCommittedResource method creates and commits a resource to a particular heap with the properties we specify.

```
md3dDevice->CreateCommittedResource(&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT), D3D12_HEAP_FLAG_NONE, &depthStencilDesc,
D3D12_RESOURCE_STATE_COMMON, &optClear, IID_PPV_ARGS(mDepthStencilBuffer.GetAddressOf()))
```

2. In addition, before using the depth/stencil buffer, we must create an associated depth/stencil view to be bound to the pipeline.

```
md3dDevice->CreateDepthStencilView(mDepthStencilBuffer.Get(), &dsvDesc, DepthStencilView());
```

3. The stencil buffer should be reset to some value at the beginning of each frame.

```
mCommandList->ClearDepthStencilView(DepthStencilView(), D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, nullptr);
```

4. We have to tell the OM what depth/stencil buffer to use. We set the depth/stencil buffer at the same time we set the render target. We do this by providing the OMSetRenderTargets command with a descriptor handle to our depth/stencil buffer.

```
mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(), true, &DepthStencilView());
```

Update the stencil buffer

```
// You might want to 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]);
```

PSO for marking stencil mirrors

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()
{
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"])));
```

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).

Example

```
DepthEnable = TRUE; // enable depth testing
DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ALL; // can write depth data to all of the depth/stencil buffer
DepthFunc = D3D12_COMPARISON_FUNC_LESS; // pixel fragment passes depth test if destination pixel's depth is less than pixel fragment’s
StencilEnable = FALSE; // disable stencil test
StencilReadMask = D3D12_DEFAULT_STENCIL_READ_MASK; // a default stencil read mask (doesn't matter at this point since stencil testing is turned off)
StencilWriteMask = D3D12_DEFAULT_STENCIL_WRITE_MASK; // a default stencil write mask (also doesn't matter)
const D3D12_DEPTH_STENCILOP_DESC defaultStencilOp = // a stencil operation structure, again does not really matter since stencil testing is turned off
 { D3D12_STENCIL_OP_KEEP, D3D12_STENCIL_OP_KEEP, D3D12_STENCIL_OP_KEEP, D3D12_COMPARISON_FUNC_ALWAYS };
 FrontFace = defaultStencilOp; // both front and back facing polygons get the same treatment
BackFace = defaultStencilOp;
```

The Comparison Operator ⊴

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.

```
typedef 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_STENCILOP_DESC

```
typedef struct D3D12_DEPTH_STENCILOP_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_STENCILOP_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.

When setting stencil by filling out a D3D12_DEPTH_STENCIL_DESC instance, you need to set both back face and front face parameters:

FrontFace: A D3D12_DEPTH_STENCILOP_DESC structure that describes how to use the results of the depth test and the stencil test for pixels whose surface normal is facing towards the camera.

BackFace: A D3D12_DEPTH_STENCILOP_DESC structure that describes how to use the results of the depth test and the stencil test for pixels whose surface normal is facing away from the camera.

```
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

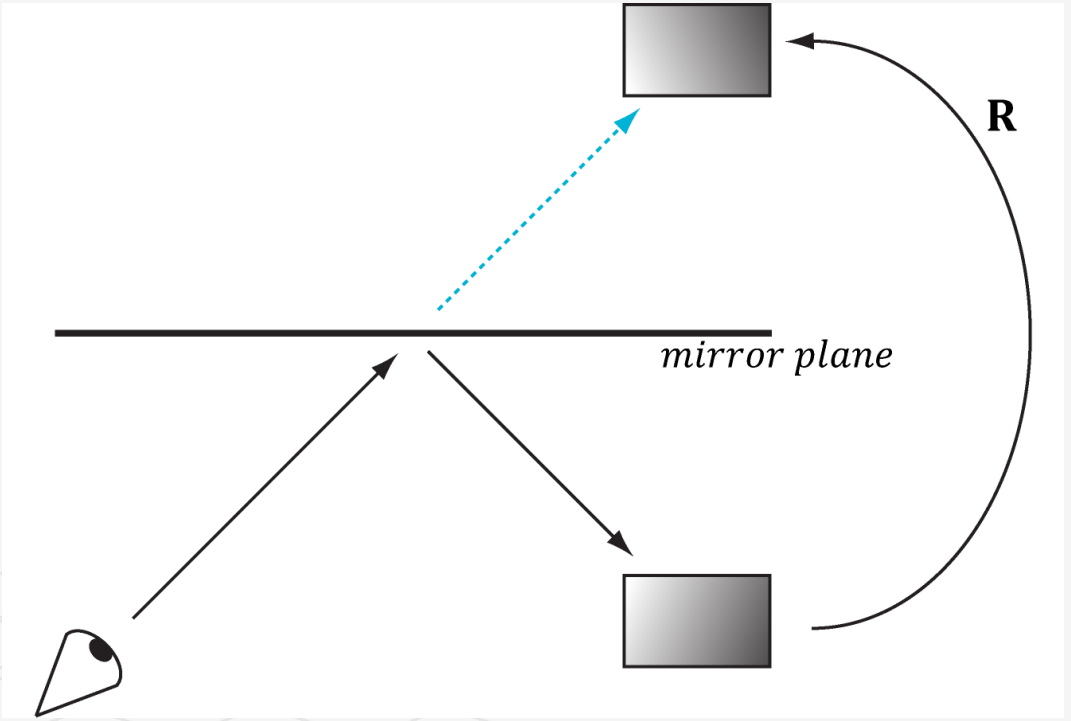
Implementing mirrors programmatically requires us to solve three problems.

1. We must learn how to reflect an object about an arbitrary plane so that we can draw the reflection correctly using some analytical geometry.

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 using stencil buffer. Because the stencil buffer allows us to block rendering to certain areas on the back buffer.

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.

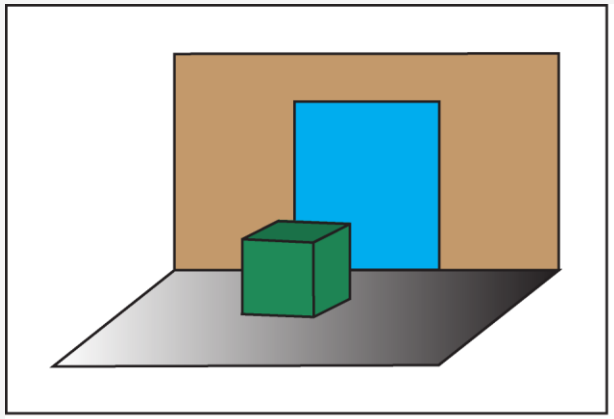


Mirror Overview

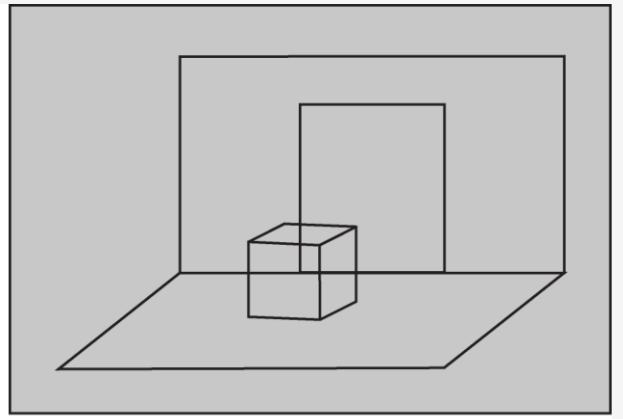
(Top) The floor, walls, and box to the back buffer and the stencil buffer cleared to 0 (denoted by light gray color). The black outlines drawn on the stencil buffer illustrate the relationship between the back buffer pixels and the stencil buffer pixels—they do not indicate any data drawn on the stencil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

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

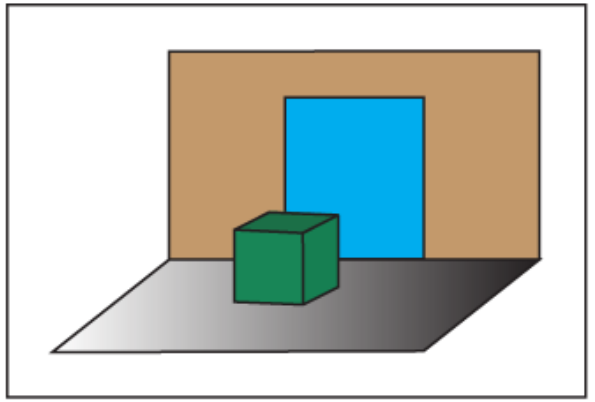
The solid black area on the stencil buffer denotes stencil entries set to 1. Note that the area on the stencil buffer occluded by the box does not get set to 1 since it fails the depth test (the box is in front of that part of the mirror).



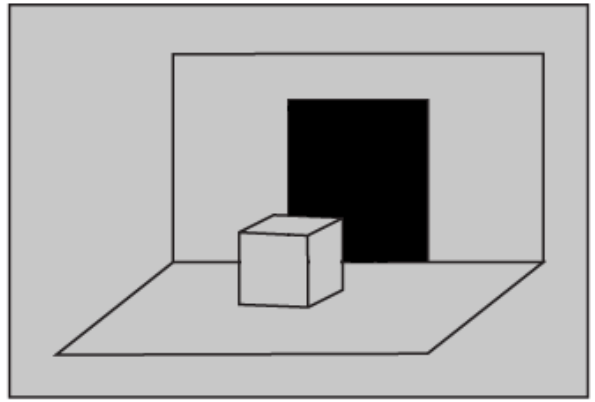
Back buffer



Stencil buffer



Back Buffer



Stencil Buffer

BuildRoomGeometry

For our demo, we draw a floor and a wall with a mirror on it. We put the floor, wall, and mirror geometry in one vertex buffer.

```
std::array<std::int16_t, 30> indices =
{
// Floor
0, 1, 2,
0, 2, 3,
// Walls
4, 5, 6,
4, 6, 7,
8, 9, 10,
8, 10, 11,
12, 13, 14,
12, 14, 15,
// Mirror
16, 17, 18,
16, 18, 19
};
```

```
std::array<Vertex, 20> vertices =
{
// Floor: Observe we tile texture coordinates.
Vertex(-3.5f, 0.0f, -10.0f, 0.0f, 1.0f, 0.0f, 0.0f, 4.0f), // 0
Vertex(-3.5f, 0.0f,   0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f),
Vertex(7.5f, 0.0f,   0.0f, 0.0f, 1.0f, 0.0f, 4.0f, 0.0f),
Vertex(7.5f, 0.0f, -10.0f, 0.0f, 1.0f, 0.0f, 4.0f, 4.0f),

// Wall: Observe we tile texture coordinates, and that we
// leave a gap in the middle for the mirror.
Vertex(-3.5f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 2.0f), // 4
Vertex(-3.5f, 4.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f),
Vertex(-2.5f, 4.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.5f, 0.0f),
Vertex(-2.5f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.5f, 2.0f),

Vertex(2.5f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 2.0f), // 8
Vertex(2.5f, 4.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f),
Vertex(7.5f, 4.0f, 0.0f, 0.0f, 0.0f, -1.0f, 2.0f, 0.0f),
Vertex(7.5f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 2.0f, 2.0f),

Vertex(-3.5f, 4.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f), // 12
Vertex(-3.5f, 6.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f),
Vertex(7.5f, 6.0f, 0.0f, 0.0f, 0.0f, -1.0f, 6.0f, 0.0f),
Vertex(7.5f, 4.0f, 0.0f, 0.0f, 0.0f, -1.0f, 6.0f, 1.0f),

// Mirror
Vertex(-2.5f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f), // 16
Vertex(-2.5f, 4.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f),
Vertex(2.5f, 4.0f, 0.0f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f),
Vertex(2.5f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f)
};
```

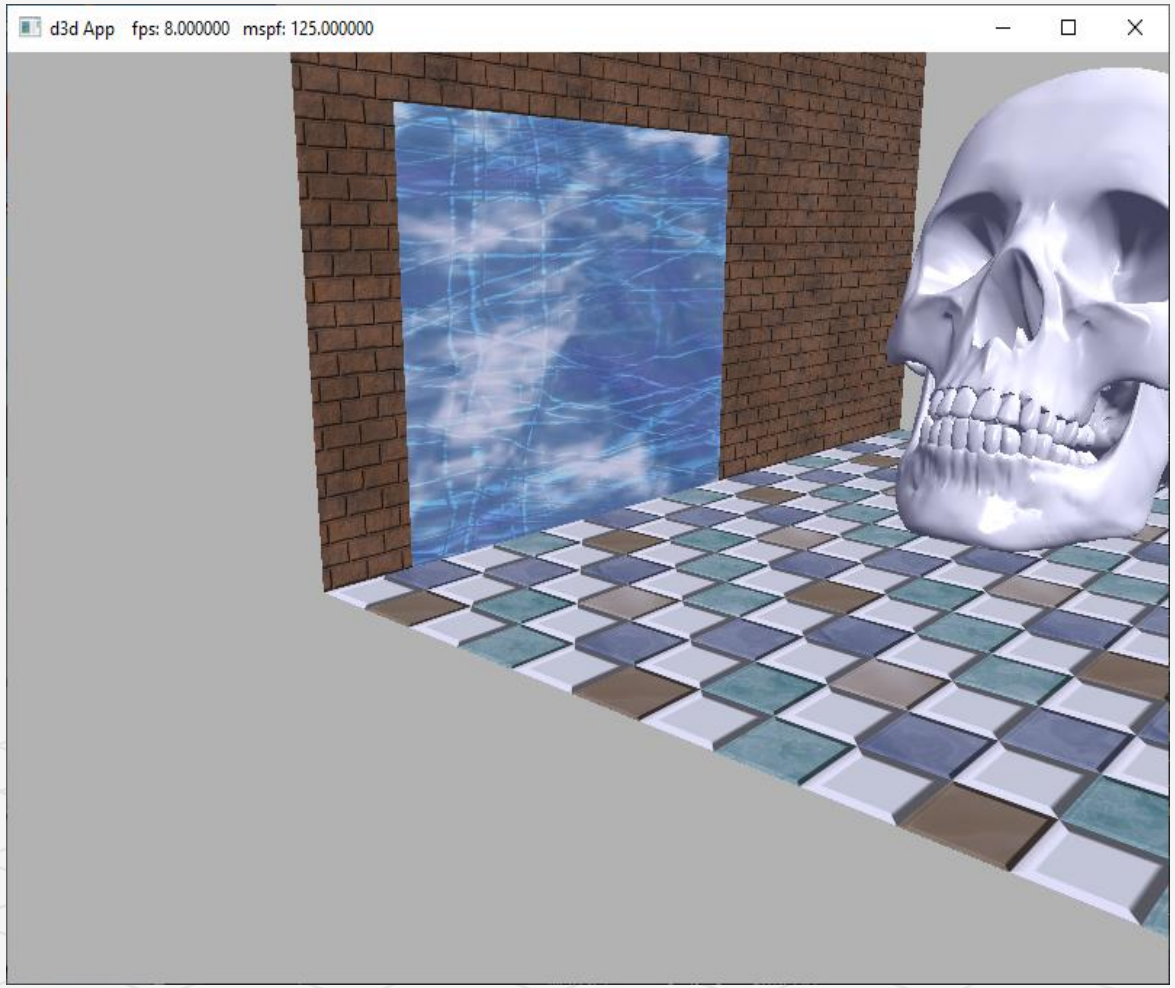
BuildRenderItems

```
auto floorRitem = std::make_unique<RenderItem>();
…
mRitemLayer[(int)RenderLayer::Opaque].push_back(floorRitem.get());
 auto wallsRitem = std::make_unique<RenderItem>();
…
mRitemLayer[(int)RenderLayer::Opaque].push_back(wallsRitem.get());
auto skullRitem = std::make_unique<RenderItem>();
…
mRitemLayer[(int)RenderLayer::Opaque].push_back(skullRitem.get());
// Reflected skull will have different world matrix, so it needs to be its own render item.
auto reflectedSkullRitem = std::make_unique<RenderItem>();
….
mRitemLayer[(int)RenderLayer::Reflected].push_back(reflectedSkullRitem.get());
auto mirrorRitem = std::make_unique<RenderItem>();
……
mRitemLayer[(int)RenderLayer::Mirrors].push_back(mirrorRitem.get());
mRitemLayer[(int)RenderLayer::Transparent].push_back(mirrorRitem.get());
```

PLANAR MIRRORS Algorithm

Step1. Render the floor, walls, and box to the back buffer as normal (but not the mirror). Note that this step does not modify the stencil buffer. This figure shows if we mirror was rendered as an opaque item.

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

auto passCB = mCurrFrameResource->PassCB->Resource();

mCommandList->SetGraphicsRootConstantBufferView(2,
passCB->GetGPUVirtualAddress());

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



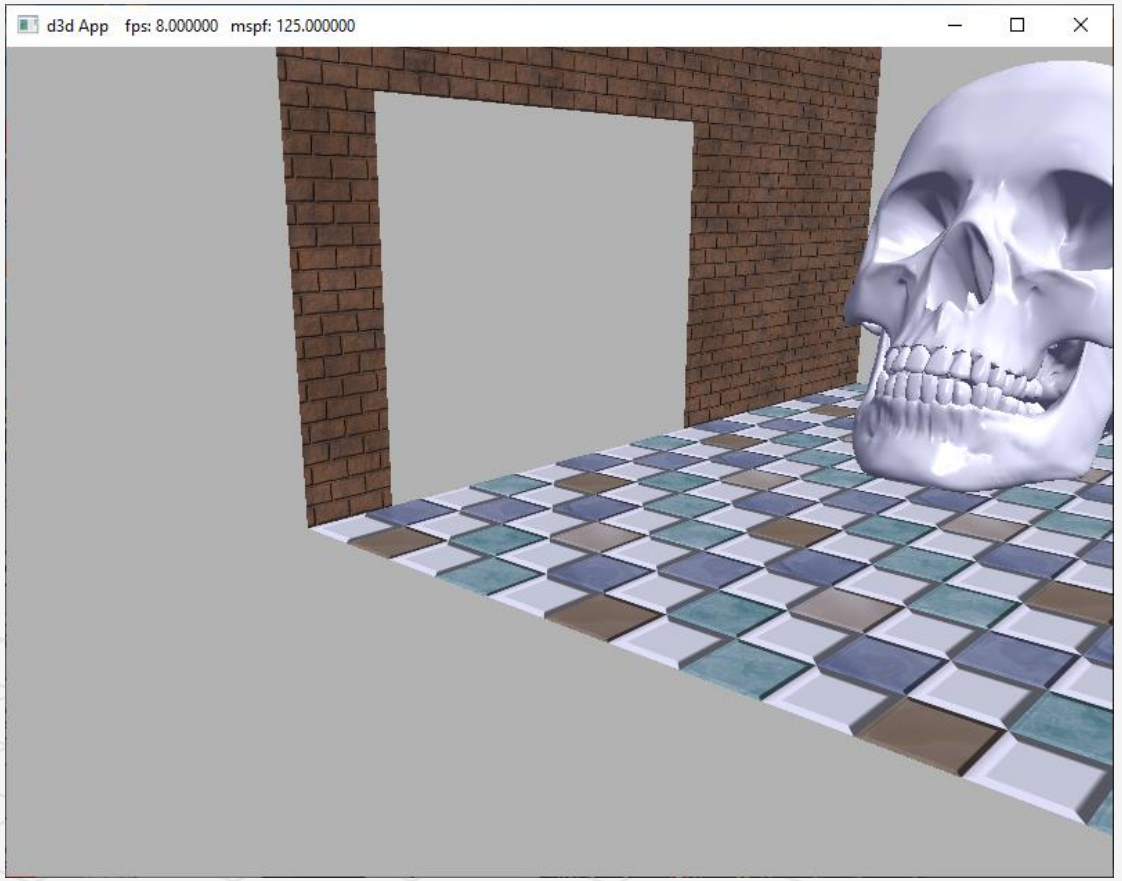
PLANAR MIRRORS Algorithm

Step 2. Render the mirror only to the stencil buffer.

Now we are drawing the mirror to mark the mirror pixels on 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.

```
mCommandList->OMSetStencilRef(1);

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

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



PLANAR MIRRORS Algorithm

We can disable color writes to the back buffer by creating a blend state that sets D3D12_RENDER_TARGET_BLEND_DESC::RenderTarget WriteMask= 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):

if the depth test passes, we specify that the stencil buffer entry should be replaced (D3D12_STENCIL_OP_REPLACE) with 1 (StencilRef).

If the depth test fails, we specify D3D12_STENCIL_OP_KEEP so that the stencil buffer is not changed.

```
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;
```

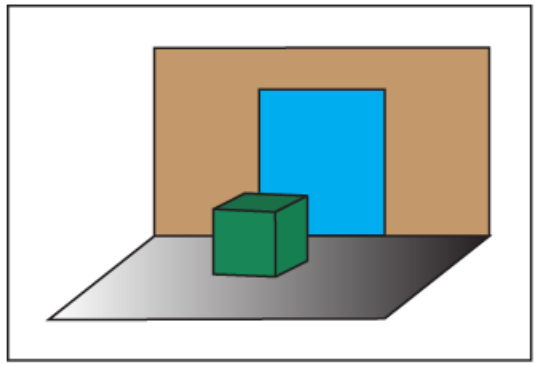
PLANAR MIRRORS Algorithm

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

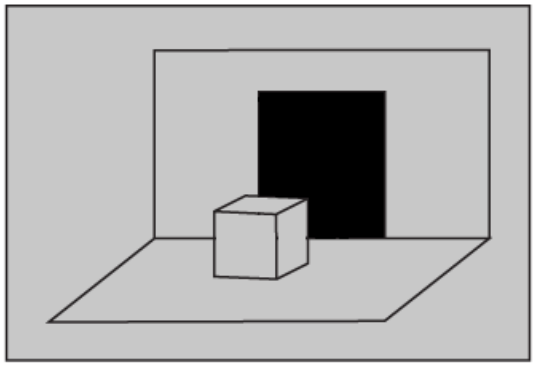
The solid black area on the stencil buffer denotes stencil entries set to 1.

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

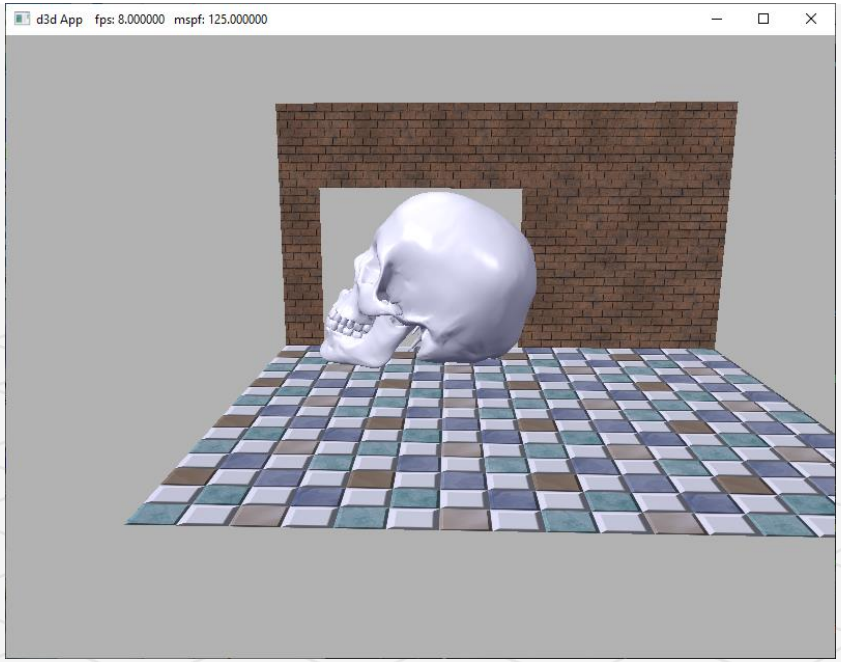
It is important to draw the mirror to the stencil buffer after we have drawn the skull(box) so that pixels of the mirror occluded by the skull fail the depth test, and therefore do not modify the stencil buffer. We do not want to turn on parts of the stencil buffer that are occluded; otherwise the reflection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



Back Buffer



Stencil Buffer

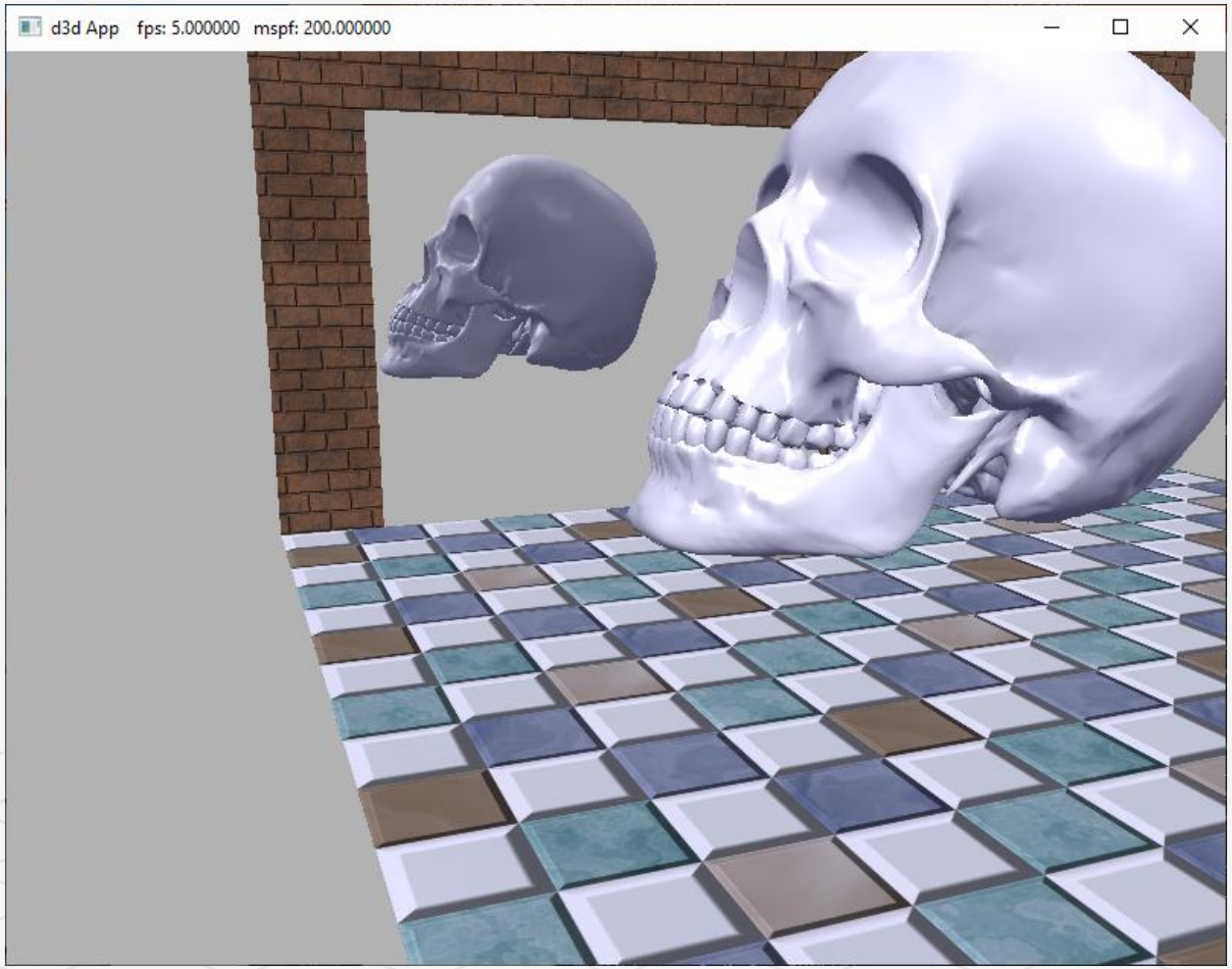


PLANAR MIRRORS Algorithm

Step3. 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]);
```



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.

```
mCommandList->OMSetStencilRef(1);

// Draw mirror with transparency so
reflection blends through.

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

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

```
// 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;
```

PLANAR MIRRORS Algorithm

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;
```

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

PSO for stencil reflections

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.

```
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"])));
```

PSO for transparent objects

```
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"])));
```

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
XMMATRIX 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]);
```

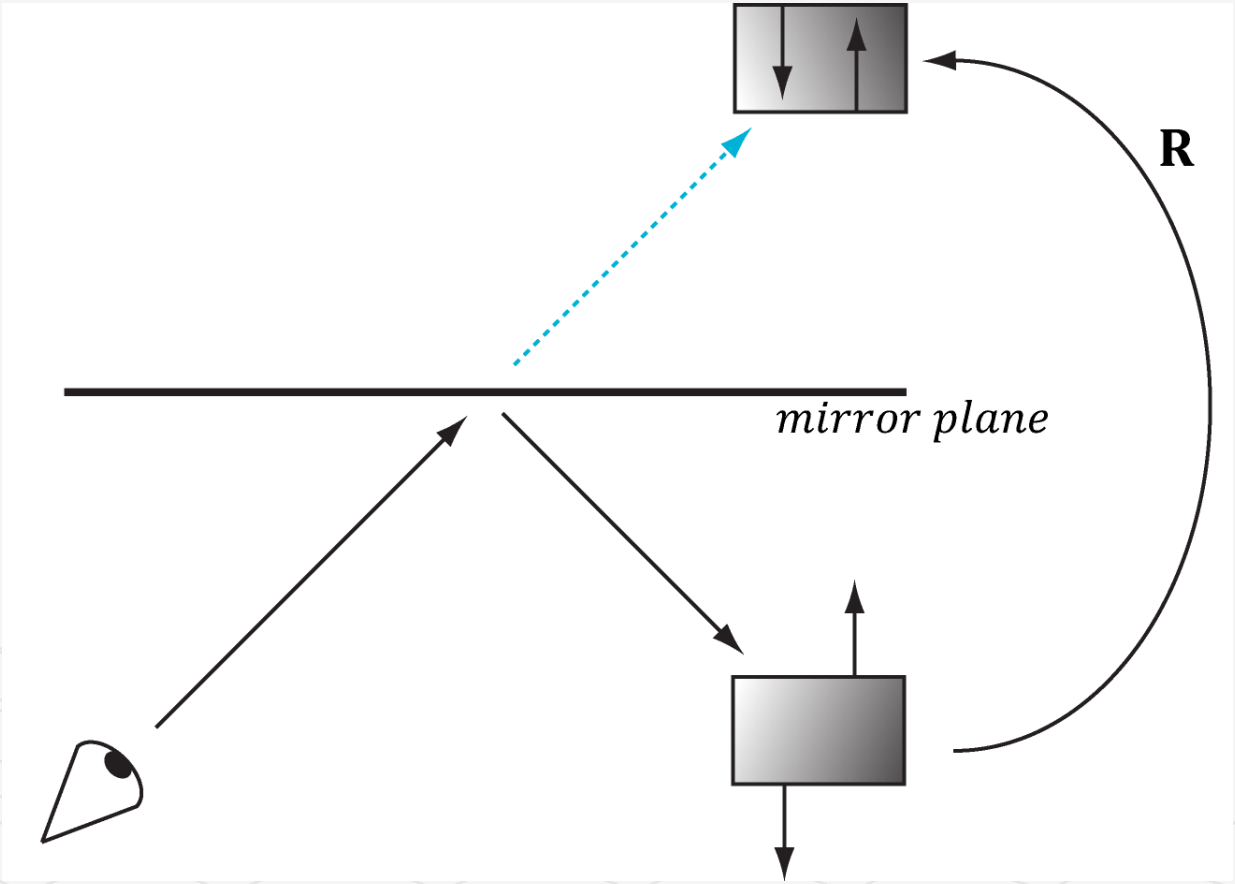
Winding Order and Reflections

When a triangle is reflected across a plane, its winding order does not reverse, and thus, its face normal does not reverse. Therefore, 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;
```



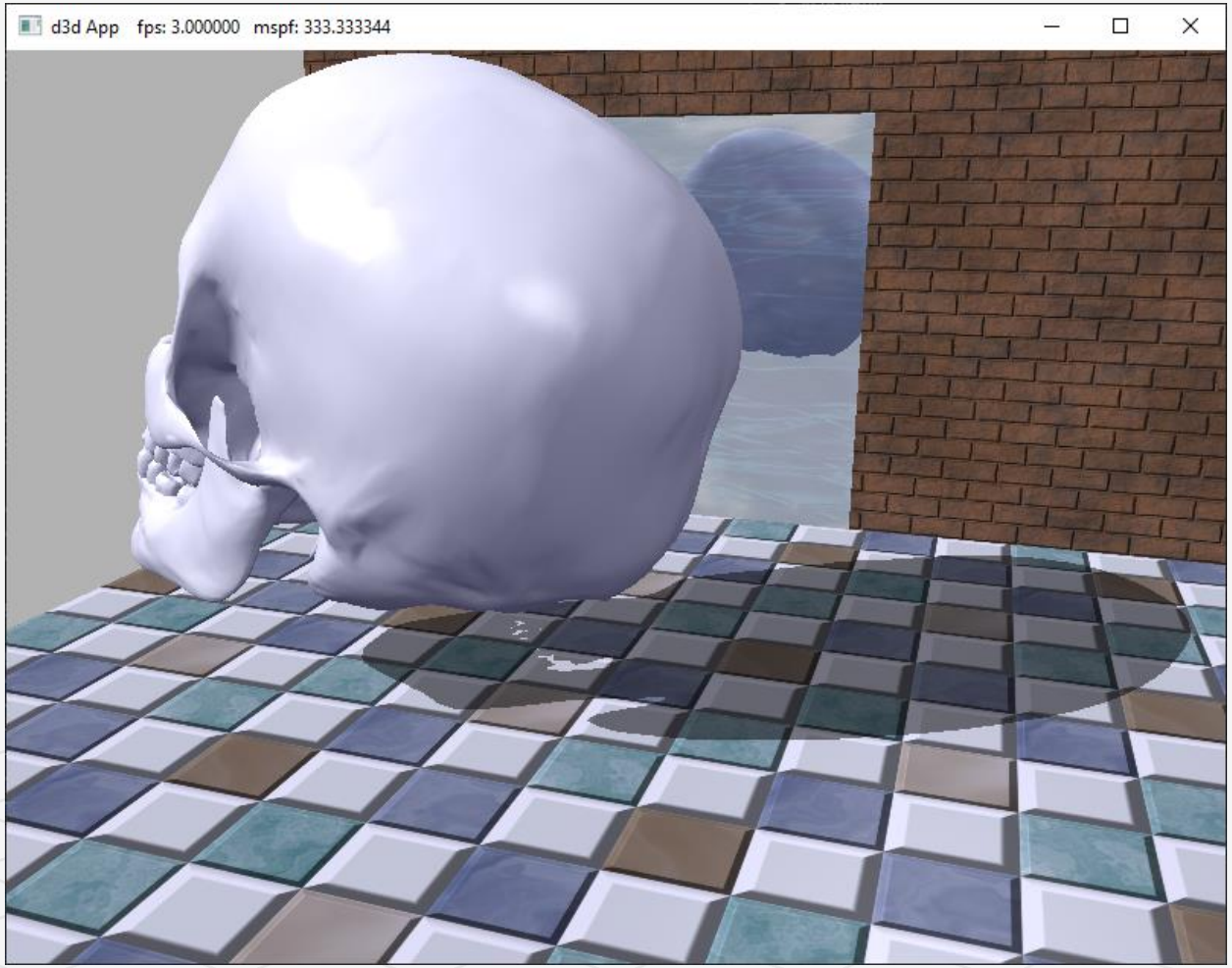
IMPLEMENTING PLANAR SHADOWS

Shadows aid in our perception of where light is being emitted in a scene and ultimately makes the scene more realistic.

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.



Planes

A plane can be specified with a vector ν and a point $\mathbf{p}_0$ on the plane.

The vector **n** is called the plane's normal vector. A plane defined by a normal vector n and a point $\mathbf{p}_0$ on the plane.

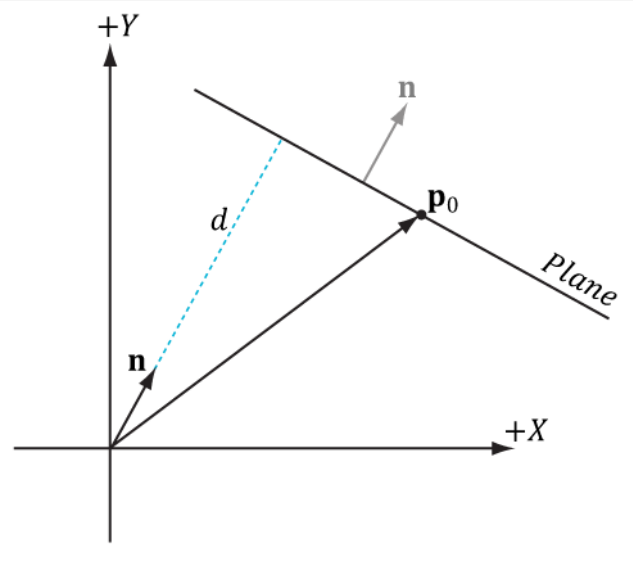
If $\mathbf{p}_0$ is a point on the plane, then the point **p** is also on the plane if and only if the vector $\mathbf{p} - \mathbf{p}_0$ is orthogonal to the plane's normal vector.

$n.(p - p_0) = n.p - np_0 = n.p + d = 0$ where $d = -\mathbf{n} \cdot \mathbf{p}_0$

The figure shows that ***d*** gives us the shortest distance from a plane to the origin.

A plane divides space into a positive half-space and a negative half-space.

A line, with a perpendicular normal, can be thought of as a 2D plane since the line divides the 2D space into a positive half space and negative half space.



DirectX Math Planes

When representing a plane in code, it suffices to store only the normal vector **n** and the constant *d*.

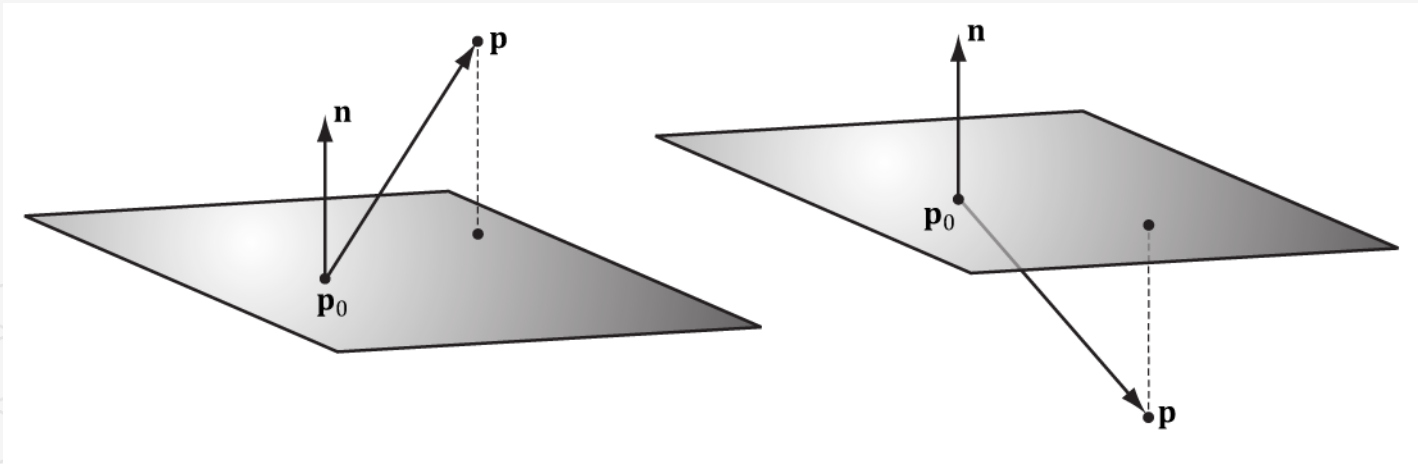
It is useful to think of this as a 4D vector, which we denote as (**n**, *d*) = (*a*, *b*, *c*, *d*).

Therefore, because the XMVECTOR type stores a 4-tuple of floating-point values, the DirectX Math library overloads the XMVECTOR type to also represent planes.

This DirectX Math function evaluates n · p = d for a particular plane and point:

```
XMVECTOR XMPlaneDotCoord(// Returns n · p = d replicated in each coordinate
XMVECTOR P, // plane
XMVECTOR V); // point with w = 1
// Test the locality of a point relative to a plane.
XMVECTOR p = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);
XMVECTOR v = XMVectorSet(3.0f, 5.0f, 2.0f);
float x = XMVectorGetX(XMPlaneDotCoord(p, v));
if (x approximately equals 0.0f) // v is coplanar to the plane.
if (x > 0) // v is in positive half-space.
if (x < 0) // v is in negative half-space.
```

A similar function is: XMVECTOR XMPlaneDotNormal(XMVECTOR Plane, XMVECTOR Vec); This returns the dot product of the plane normal vector and the given 3D vector.



XMPlaneFromPointNormal & XMPlaneFromPoints

The DirectX Math library provides the following function to construct a plane from a point and normal in this way:

XMVECTOR XMPlaneFromPointNormal(

XMVECTOR Point,

XMVECTOR Normal);

The DirectX Math library provides the following function to compute a plane given

three points on the plane:

XMVECTOR XMPlaneFromPoints(

XMVECTOR Point1,

XMVECTOR Point2,

XMVECTOR Point3);

Sometimes we might have a plane and would like to normalize the normal vector. At first thought, it would seem that we could just normalize the normal vector as we would any other vector. But recall that the d component also depends on the normal vector: $d = -\mathbf{n} \cdot \mathbf{p}_0$. Therefore, if we normalize the normal vector, we must also recalculate d.

We can use the following DirectX Math function to normalize a plane's normal vector:

XMVECTOR XMPlaneNormalize(XMVECTOR P);

Parallel Light Shadows

The figure shows a shadow cast with respect to a parallel light source.

Given a parallel light source with direction **L**, the light ray that passes through a vertex **p** is given by **r**(*t*) = **p** + *t***L**. The intersection of the ray *r*(*t*) with the shadow plane (**n**, *d*) gives **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 **p**, its shadow projection is given by:

The DirectX Math library provides the following function:

XMVECTOR XMPlaneIntersectLine(

XMVECTOR P,

XMVECTOR LinePoint1,

XMVECTOR LinePoint2);

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

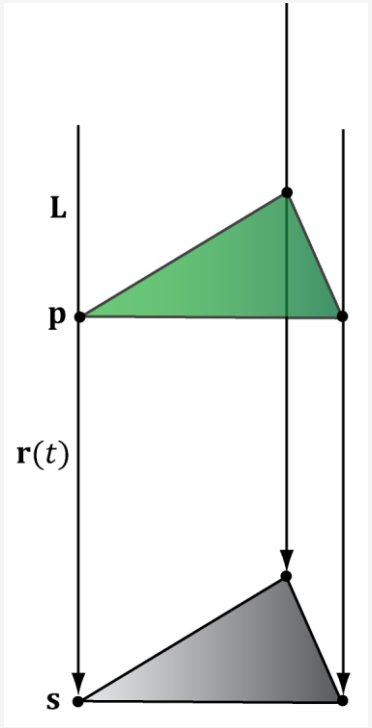
The details of the ray/plane intersection test are given in

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

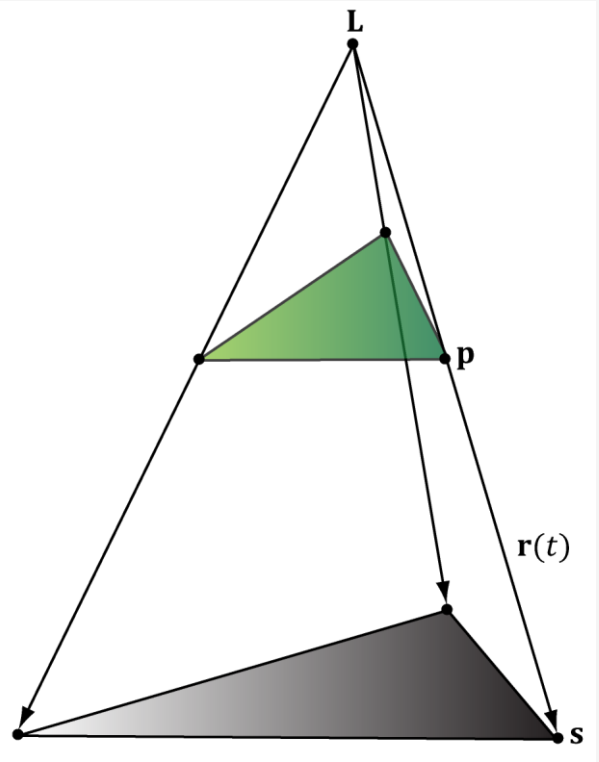
The shadow cast with respect to a point light source.

The light ray from a point light through any vertex **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 **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 **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 **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 **L** describes the location of the point light.

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

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
)
```

Shadowed Skull Item and Material

```
// Shadowed skull will have different world
matrix, so it needs to be its own render
item.

auto shadowedSkullRitem =
std::make_unique<RenderItem>();

*shadowedSkullRitem = *skullRitem;

shadowedSkullRitem->ObjCBIndex = 4;

shadowedSkullRitem->Mat =
mMaterials["shadowMat"].get();

mShadowedSkullRitem =
shadowedSkullRitem.get();

mRitemLayer[(int)RenderLayer::Shadow].push_ba
ck(shadowedSkullRitem.get());
```

```
auto skullMat = std::make_unique<Material>();
skullMat->Name = "skullMat";
skullMat->MatCBIndex = 3;
skullMat->DiffuseSrvHeapIndex = 3;
skullMat->DiffuseAlbedo = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
skullMat->FresnelR0 = XMFLOAT3(0.05f, 0.05f, 0.05f);
skullMat->Roughness = 0.3f;

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;
```

StencilApp::OnKeyboardInput

```
void StencilApp::Update(const GameTimer& gt)
{
//we are calling OnKeyboardInput to calculate the
shadow world skulled matrix for every frame or every
time user presses AWSD keys
    OnKeyboardInput(gt);

void StencilApp::OnKeyboardInput(const GameTimer& gt)
{
// Allow user to move skull.

const float dt = gt.DeltaTime();

if(GetAsyncKeyState('A') & 0x8000)
mSkullTranslation.x -= 1.0f*dt;

if(GetAsyncKeyState('D') & 0x8000)
mSkullTranslation.x += 1.0f*dt;

if(GetAsyncKeyState('W') & 0x8000)
mSkullTranslation.y += 1.0f*dt;

if(GetAsyncKeyState('S') & 0x8000)
mSkullTranslation.y -= 1.0f*dt;
```

```
// 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);
```

Note that we offset the projected shadow mesh along the y-axis by a small amount to prevent z-fighting so the shadow mesh does not intersect the floor mesh, but lies slightly above it. If the meshes did intersect, then due to limited precision of the depth buffer, we would see flickering artifacts as the floor and shadow mesh pixels compete to be visible.

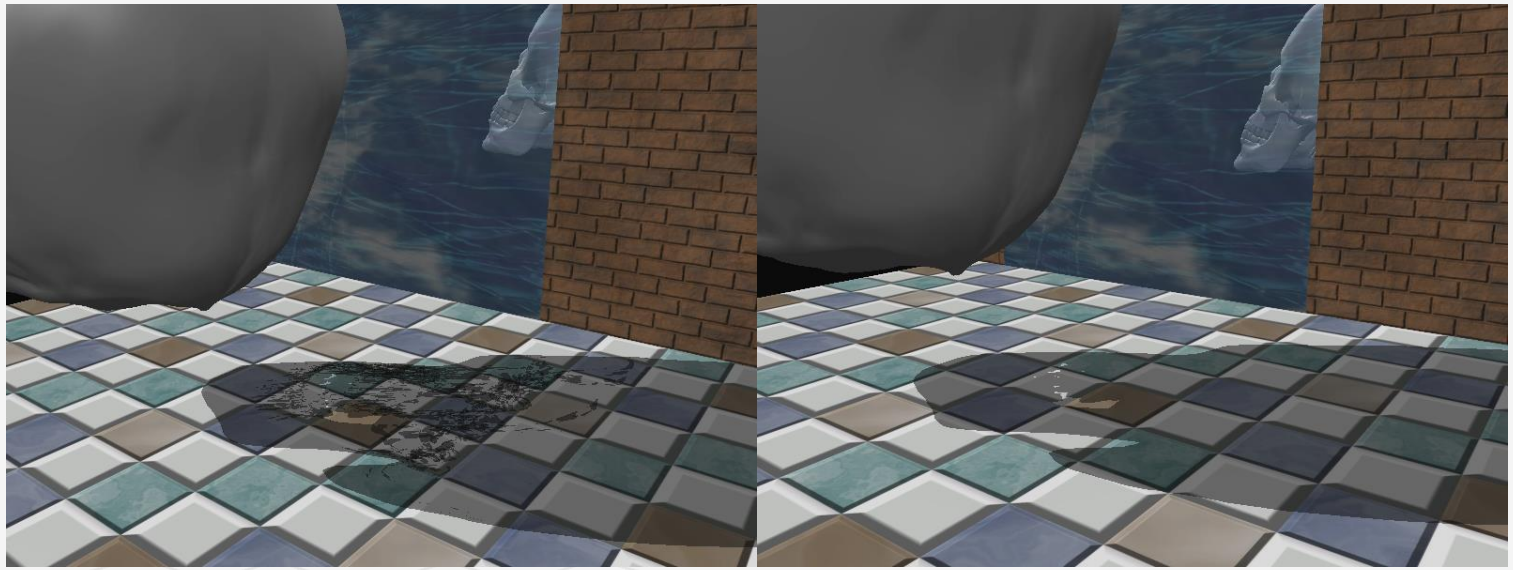
```
XMMATRIX shadowOffsetY = XMMatrixTranslation(0.0f, 0.001f, 0.0f);

XMStoreFloat4x4(&mShadowedSkullRitem->World, skullWorld * S * shadowOffsetY);
```

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.

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);
```