

Game Programming with DirectX

출력 병합 단계

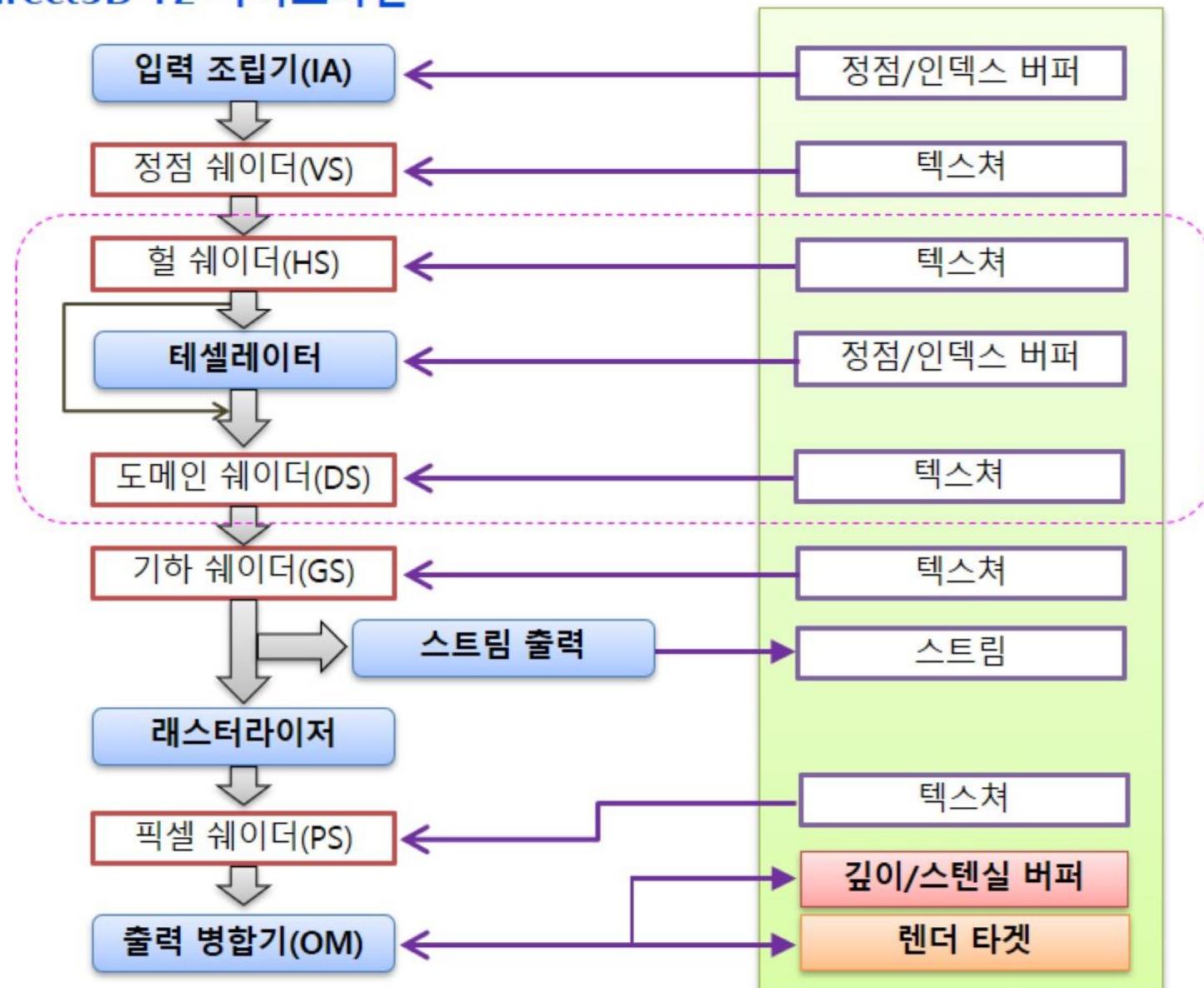
(Output Merger Stage)

블렌딩 (Blending)

스텐실 (Stencil)

Direct3D 파이프라인

- Direct3D 12 파이프라인



Direct3D 파이프라인

- **출력-병합 단계(Output-Merger Stage)**

- 최종적으로 픽셀의 색상을 생성하여 렌더 타겟으로 출력하는 단계
파이프라인 상태 정보, 픽셀-쉐이더가 생성한 픽셀 데이터, 렌더 타겟의 내용,
그리고 깊이/스텐실 버퍼의 내용을 조합하여 출력할 픽셀의 최종 색상을 결정
- 출력-병합 단계는 깊이/스텐실 테스트를 통하여 픽셀이 출력되는 가를 결정
출력되는 경우에 최종 픽셀 색상을 결정(블렌딩)하는 파이프라인의 마지막 단계임
- **깊이-스텐실 검사(Depth-Stencil Testing)**
 - 하나의 깊이/스텐실 버퍼만 활성화됨
 - 깊이 검사(Depth Test)
출력할 픽셀의 깊이 값과 깊이 버퍼의 값에 대한 검사가 성공하면 픽셀을 출력
$$z = \min(\text{Viewport.MaxDepth}, \max(\text{Viewport.MinDepth}, z))$$
 - 스템실 검사(Stencil Test)
스텐실 값을 사용하여 픽셀이 렌더 타겟으로 출력되는 여부를 결정
깊이 버퍼에 스템실 요소가 없으면 스템실 검사는 항상 성공
깊이 버퍼가 연결되지 않으면 스템실 검사는 항상 성공
- **블렌딩(Blending)**
 - 여러 개의 픽셀 값(색상)을 결합하여 하나의 최종 픽셀 색상을 생성하는 과정
 - 렌더 타겟의 색상과 픽셀 쉐이더의 출력 색상을 결합하여 최종 색상을 결정



블렌딩(Blending)

- 텍스쳐 블렌딩(Texture Blending)

```
VS_TERRAIN_OUTPUT VSTerrain(VS_TERRAIN_INPUT input)
{
    VS_TERRAIN_OUTPUT output = (VS_TERRAIN_OUTPUT)0;
    output.normalW = mul(input.normal, (float3x3)gmtxWorld);
    output.normalW = normalize(output.normalW);
    output.positionW = mul(input.position, (float3x3)gmtxWorld);
    output.positionW += float3(gmtxWorld._41, gmtxWorld._42, gmtxWorld._43);
    matrix mtxWorldViewProjection = mul(gmtxWorld, gmtxView);
    mtxWorldViewProjection = mul(mtxWorldViewProjection, gmtxProjection);
    output.position = mul(float4(input.position, 1.0f), mtxWorldViewProjection);
    output.uv0 = input.uv0;
    output.uv1 = input.uv1;
    return(output);
}
```

```
Texture2D gtxtBaseTexture : register(t1);
Texture2D gtxtDetailTexture : register(t2);
Texture2D gtxtAlphaTexture : register(t3);
```

```
output.uv1 = mul(float4(input.uv0, 0.0f, 1.0f), gmtxTexture);
```

```
struct VS_TERRAIN_INPUT {
    float3 position : POSITION;
    float3 normal: NORMAL;
    float2 uv0: TEXCOORD0;
    float2 uv1: TEXCOORD1;
};
```

```
struct VS_TERRAIN_OUTPUT {
    float4 position: SV_POSITION;
    float3 positionW: POSITION;
    float3 normalW: NORMAL;
    float2 uv0: TEXCOORD0;
    float2 uv1: TEXCOORD1;
};
```

```
float4 PSTerrain(VS_TERRAIN_OUTPUT input) : SV_Target
```

```
};
```

```
{
    float4 cillumination = Lighting(input.positionW, input.normalW);
    float4 cBaseTexColor = gtxtBaseTexture.Sample(gTerrainSamplerState, input.uv0);
    float4 cDetailTexColor = gtxtDetailTexture.Sample(gTerrainSamplerState, input.uv1);
    float4 cColor = cillumination * (cBaseTexColor + cDetailTexColor);
    return(saturate(cColor));
}
```

```
float4 cAlphaTexColor = gtxtAlphaTexture.Sample(gTerrainSamplerState, input.uv0);
float4 cColor = cillumination * lerp(cBaseTexColor, cDetailTexColor, cAlphaTexColor.r);
```

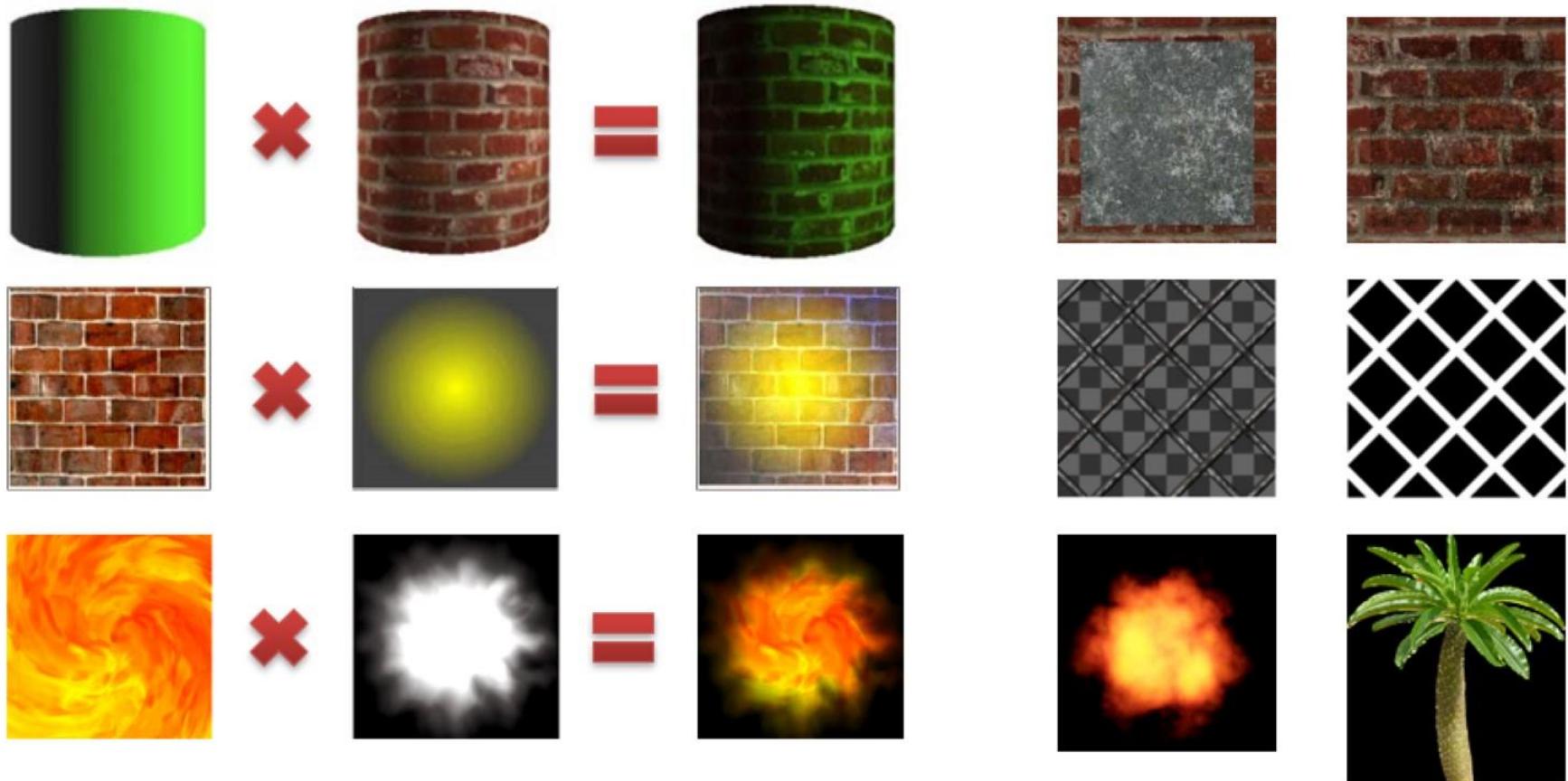
블렌딩(Blending)

- **텍스쳐 블렌딩(Texture Blending)**

- 블렌딩: 색상(Color) 블렌딩, 알파(Alpha) 블렌딩, **다중-패스(Multi-Pass) 렌더링**
- **다중 텍스쳐링(Multi-Texturing)**

하나의 정점이 여러 개의 텍스쳐 좌표를 가진 경우 여러 개의 텍스쳐를 매핑 가능
쉐이더 프로그램에서 해결 가능

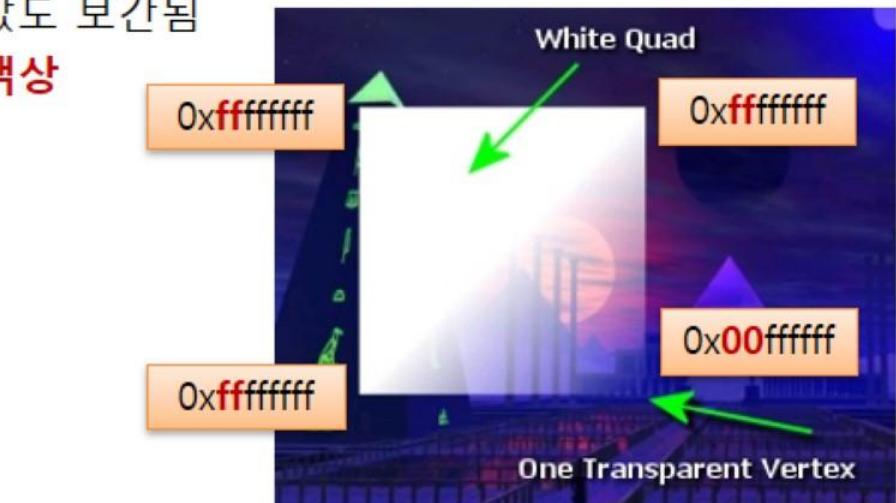
단일-패스(One-Pass) 렌더링



블렌딩(Blending)

- **알파 블렌딩(Alpha Blending)**

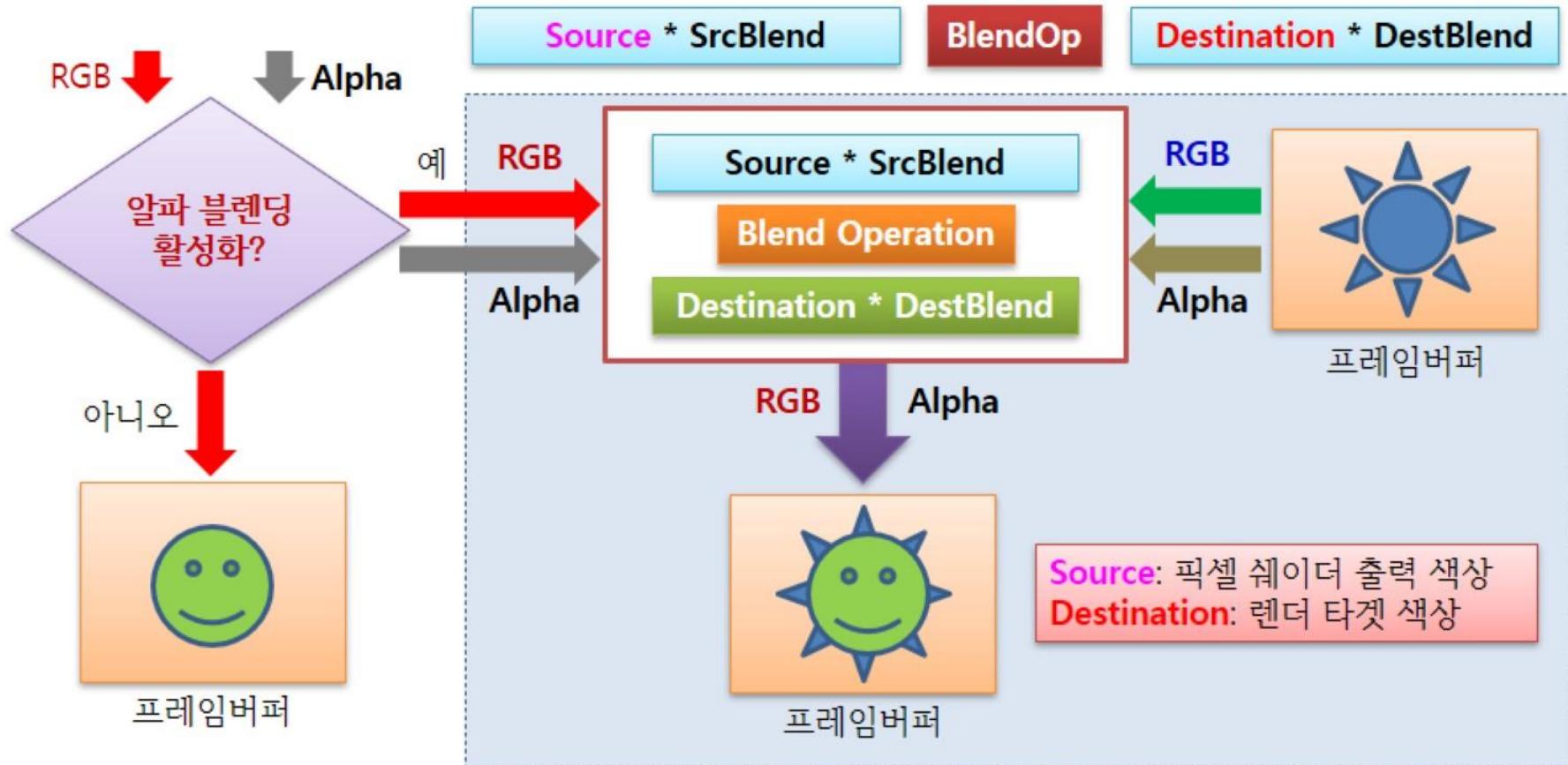
- 정점의 알파 값을 사용하여 블렌딩을 하는 경우
- 주로 투명 효과(Transparent Effect)를 위하여 사용
- RS 단계에서 색상 뿐 아니라 알파 값도 보간됨
- **정점의 색상, 재질의 색상, 텍스쳐 색상**



정점 색상의 알파: 0x40(64)
75% 투명 = 25% 불투명

블렌딩(Blending)

- 출력-병합 단계 블렌딩(Blending)



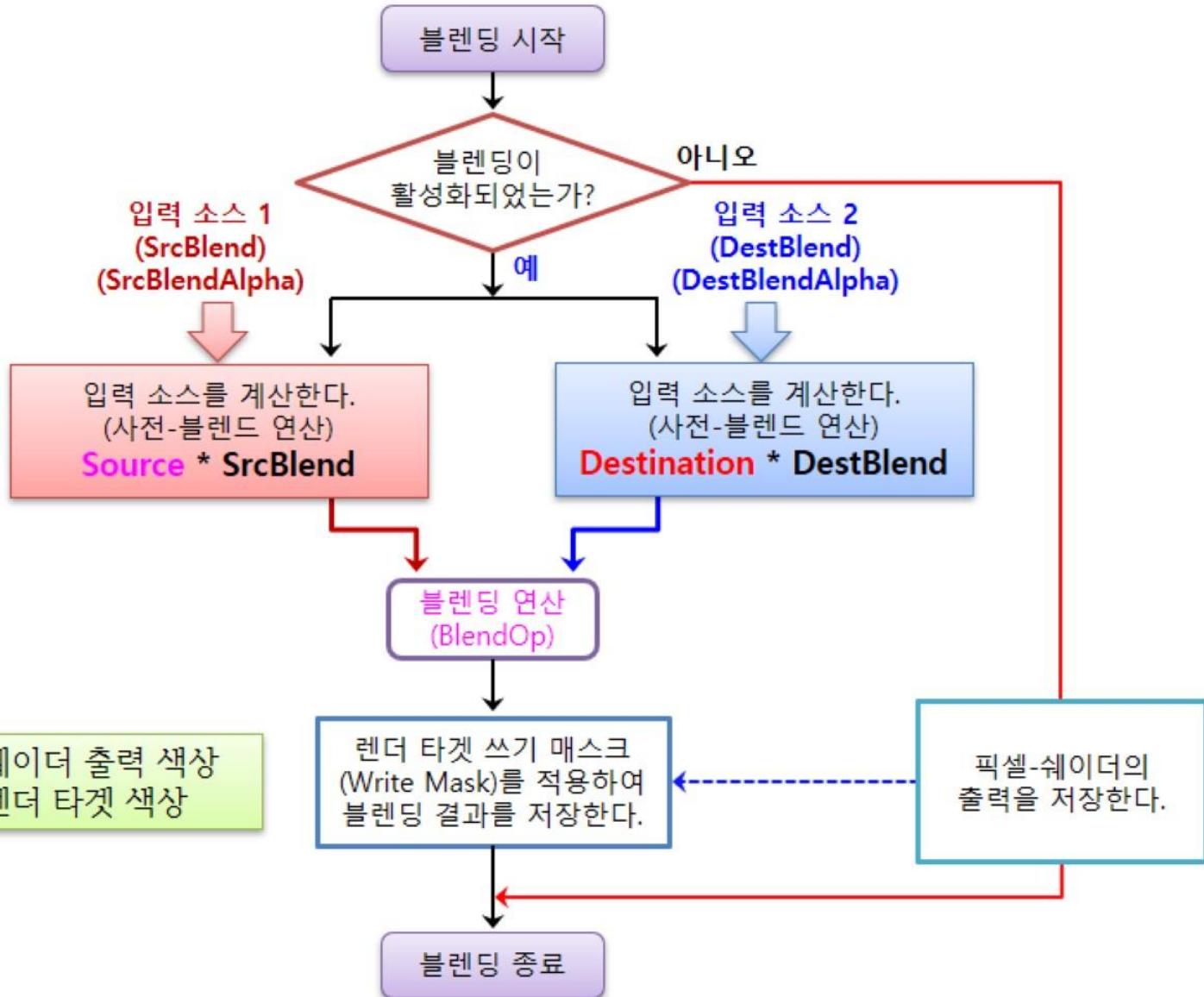
```
float4 PS(VS_OUTPUT input)
{
    ...
    return(cSourceColor);
}
```

블렌딩은 일반적으로 픽셀 쉐이더의 출력 값과 렌더 타겟의 값을 결합

Alpha	Red	Green	Blue
80	ff	40	40
128	255	64	64

블렌딩(Blending)

- 출력-병합 단계 블렌딩(Blending)



블렌딩(Blending)

- 그래픽 파이프라인 상태

```
typedef struct D3D12_GRAPHICS_PIPELINE_STATE_DESC {  
    ID3D12RootSignature *pRootSignature;  
    D3D12_SHADER_BYTECODE VS;  
    D3D12_SHADER_BYTECODE PS;  
    D3D12_SHADER_BYTECODE DS;  
    D3D12_SHADER_BYTECODE HS;  
    D3D12_SHADER_BYTECODE GS;  
    D3D12_STREAM_OUTPUT_DESC StreamOutput;  
    D3D12_BLEND_DESC BlendState;  
    UINT SampleMask;  
    D3D12_RASTERIZER_DESC RasterizerState;  
    D3D12_DEPTH_STENCIL_DESC DepthStencilState;  
    D3D12_INPUT_LAYOUT_DESC InputLayout;  
    D3D12_INDEX_BUFFER_STRIP_CUT_VALUE IBStripCutValue;  
    D3D12_PRIMITIVE_TOPOLOGY_TYPE PrimitiveTopologyType; //기하 셰이더와 헐 셰이더  
    UINT NumRenderTargets;  
    DXGI_FORMAT RTVFormats[8];  
    DXGI_FORMAT DSVFormat;  
    DXGI_SAMPLE_DESC SampleDesc;  
    UINT NodeMask;  
    D3D12_CACHED_PIPELINE_STATE CachedPSO;  
    D3D12_PIPELINE_STATE_FLAGS Flags;  
} D3D12_GRAPHICS_PIPELINE_STATE_DESC;
```

```
typedef struct D3D12_DEPTH_STENCIL_DESC {  
    BOOL DepthEnable;  
    D3D12_DEPTH_WRITE_MASK DepthWriteMask;  
    D3D12_COMPARISON_FUNC DepthFunc;  
    BOOL StencilEnable;  
    UINT8 StencilReadMask;  
    UINT8 StencilWriteMask;  
    D3D12_DEPTH_STENCILOP_DESC FrontFace;  
    D3D12_DEPTH_STENCILOP_DESC BackFace;  
} D3D12_DEPTH_STENCIL_DESC;
```

```
typedef struct D3D12_BLEND_DESC {  
    BOOL AlphaToCoverageEnable;  
    BOOL IndependentBlendEnable;  
    D3D12_RENDER_TARGET_BLEND_DESC RenderTarget[8];  
} D3D12_BLEND_DESC;
```

ID3D12Device::CreateGraphicsPipelineState(D3D12_GRAPHICS_PIPELINE_STATE_DESC *pDesc, ...);

블렌딩(Blending)

- 블렌드 상태(Blend State) 설정

```
typedef struct D3D12_BLEND_DESC {  
    BOOL AlphaToCoverageEnable; //다중 샘플링을 위하여 렌더 타겟 0의 알파 값을 커버리지 마스크로 변환  
    BOOL IndependentBlendEnable; //각 렌더 타겟에서 독립적인 블렌딩을 수행, FALSE(RenderTarget[0]만 사용)  
    D3D12_RENDER_TARGET_BLEND_DESC RenderTarget[8]; //각 렌더 타겟에 대한 블렌딩 설정  
} D3D12_BLEND_DESC;
```

```
typedef struct D3D12_RENDER_TARGET_BLEND_DESC {  
    BOOL BlendEnable; //블렌딩을 활성화  
    BOOL LogicOpEnable; //논리 연산을 활성화  
    D3D12_BLEND SrcBlend; //픽셀 색상에 곱하는 값(요소별 연산)  
    D3D12_BLEND DestBlend; //렌더 타겟 색상에 곱하는 값  
    D3D12_BLEND_OP BlendOp; //RGB 색상 블렌드 연산자  
    D3D12_BLEND SrcBlendAlpha;  
    D3D12_BLEND DestBlendAlpha;  
    D3D12_BLEND_OP BlendOpAlpha;  
    D3D12_LOGIC_OP LogicOp; //논리 연산자  
    UINT8 RenderTargetWriteMask; //블렌드 타겟에 적용할 마스크  
} D3D12_RENDER_TARGET_BLEND_DESC;
```

```
typedef enum D3D12_BLEND_OP {  
    D3D12_BLEND_OP_ADD, // DestBlend* + SrcBlend*  
    D3D12_BLEND_OP_SUBTRACT, //DestBlend* - SrcBlend*  
    D3D12_BLEND_OP_REV_SUBTRACT, //SrcBlend* - DestBlend*  
    D3D12_BLEND_OP_MIN, //min(DestBlend*, SrcBlend*)  
    D3D12_BLEND_OP_MAX //max(DestBlend*, SrcBlend*)  
} D3D12_BLEND_OP;
```

```
typedef enum D3D12_BLEND {  
    D3D12_BLEND_ZERO,  
    D3D12_BLEND_ONE,  
    D3D12_BLEND_SRC_COLOR,  
    D3D12_BLEND_INV_SRC_COLOR,  
    D3D12_BLEND_SRC_ALPHA,  
    D3D12_BLEND_INV_SRC_ALPHA,  
    D3D12_BLEND_DEST_ALPHA,  
    D3D12_BLEND_INV_DEST_ALPHA,  
    D3D12_BLEND_DEST_COLOR,  
    D3D12_BLEND_INV_DEST_COLOR,  
    D3D12_BLEND_SRC_ALPHA_SAT,  
    D3D12_BLEND_BLEND_FACTOR,  
    D3D12_BLEND_INV_BLEND_FACTOR,  
    D3D12_BLEND_SRC1_COLOR,  
    D3D12_BLEND_INV_SRC1_COLOR,  
    D3D12_BLEND_SRC1_ALPHA,  
    D3D12_BLEND_INV_SRC1_ALPHA  
} D3D12_BLEND;
```

블렌딩(Blending)

- **블렌드 상태(Blend State) 설정**

- 블렌드 연산 설정(D3D12_BLEND)

SrcColor * SrcBlend

BlendOp

DestColor * DestBlend

SrcAlpha * SrcBlendAlpha

BlendOp

DestAlpha * DestBlendAlpha

D3D12_BLEND_OP_ADD

D3D12_BLEND_OP_SUBTRACT

D3D12_BLEND_OP_REV_SUBTRACT

D3D12_BLEND_OP_MIN

D3D12_BLEND_OP_MAX

D3D12_BLEND_ZERO	검정색, (0, 0, 0, 0)
D3D12_BLEND_ONE	흰색, (1, 1, 1, 1)
D3D12_BLEND_SRC_COLOR	픽셀 쉐이더의 출력(SV_Target0) 색상(RGB), (R_s, G_s, B_s, A_s)
D3D12_BLEND_INV_SRC_COLOR	픽셀 쉐이더의 출력 색상(RGB)의 보색, ($1-R_s, 1-G_s, 1-B_s, 1-A_s$)
D3D12_BLEND_DEST_COLOR	렌더 타겟의 색상(RGB), (R_d, G_d, B_d, A_d)
D3D12_BLEND_INV_DEST_COLOR	렌더 타겟의 색상(RGB)의 보색, ($1-R_d, 1-G_d, 1-B_d, 1-A_d$)
D3D12_BLEND_SRC_ALPHA_SAT	픽셀 쉐이더의 출력 알파값(A_s), ($f, f, f, 1$), $f=\text{clamp}(A_s, 0, 1)$
D3D12_BLEND_SRC_ALPHA	픽셀 쉐이더의 출력 알파값(A_s), (A_s, A_s, A_s, A_s)
D3D12_BLEND_INV_SRC_ALPHA	픽셀 쉐이더의 출력 알파값(A_s)의 보수, ($1-A_s, 1-A_s, 1-A_s, 1-A_s$)
D3D12_BLEND_DEST_ALPHA	렌더 타겟의 알파값(A_d), (A_d, A_d, A_d, A_d)
D3D12_BLEND_INV_DEST_ALPHA	렌더 타겟의 알파값(A_d)의 보수, ($1-A_d, 1-A_d, 1-A_d, 1-A_d$)
D3D12_BLEND_BLEND_FACTOR	블렌딩 팩터(OMSetBlendFactor() 함수에서 설정하는 색상 값)
D3D12_BLEND_INV_BLEND_FACTOR	블렌딩 팩터(OMSetBlendFactor() 함수에서 설정하는 색상 값)의 보색
D3D12_BLEND_SRC1_COLOR	픽셀 쉐이더의 출력(SV_Target1)의 색상(RGB)
D3D12_BLEND_INV_SRC1_COLOR	픽셀 쉐이더의 출력(SV_Target1)의 색상의 보색(1-RGB)
D3D12_BLEND_SRC1_ALPHA	픽셀 쉐이더의 출력(SV_Target1)의 알파값(A)
D3D12_BLEND_INV_SRC1_ALPHA	픽셀 쉐이더의 출력(SV_Target1)의 알파값(A)의 보수

블렌딩(Blending)

- 블렌드 상태(Blend State) 설정

```
void ID3D12GraphicsCommandList::OMSetBlendFactor(FLOAT BlendFactor[4]); //NULL: (1, 1, 1, 1)
```

```
typedef struct D3D12_RENDER_TARGET_BLEND_DESC {  
    BOOL BlendEnable; //블렌딩을 활성화  
    BOOL LogicOpEnable; //논리 연산을 활성화  
    D3D12_BLEND SrcBlend; //픽셀 색상에 곱하는 값(요소별 연산)  
    D3D12_BLEND DestBlend; //렌더 타겟 색상에 곱하는 값  
    D3D12_BLEND_OP BlendOp; //RGB 색상 블렌드 연산자  
    D3D12_BLEND SrcBlendAlpha;  
    D3D12_BLEND DestBlendAlpha;  
    D3D12_BLEND_OP BlendOpAlpha;  
    D3D12_LOGIC_OP LogicOp; //논리 연산자  
    UINT8 RenderTargetWriteMask; //렌더 타겟에 적용할 마스크  
} D3D12_RENDER_TARGET_BLEND_DESC;
```

D3D12_COLOR_WRITE_ENABLE

D3D12_COLOR_WRITE_ENABLE_RED	R에 쓰기 허용
D3D12_COLOR_WRITE_ENABLE_GREEN	G에 쓰기 허용
D3D12_COLOR_WRITE_ENABLE_BLUE	B에 쓰기 허용
D3D12_COLOR_WRITE_ENABLE_ALPHA	A에 쓰기 허용
D3D12_COLOR_WRITE_ENABLE_ALL	RGBA에 쓰기 허용

블렌딩의 결과를 렌더 타겟에 적용(Write)할 색상 마스크(채널을 선택할 수 있음)
블렌딩이 활성화되지 않으면 픽셀 쉐이더의 출력 색상에 색상 마스크가 적용되지 않음

```
typedef enum D3D12_LOGIC_OP {  
    D3D12_LOGIC_OP_CLEAR,  
    D3D12_LOGIC_OP_SET,  
    D3D12_LOGIC_OP_COPY,  
    D3D12_LOGIC_OP_COPY_INVERTED,  
    D3D12_LOGIC_OP_NOOP,  
    D3D12_LOGIC_OP_INVERT,  
    D3D12_LOGIC_OP_AND,  
    D3D12_LOGIC_OP_NAND,  
    D3D12_LOGIC_OP_OR,  
    D3D12_LOGIC_OP_NOR,  
    D3D12_LOGIC_OP_XOR,  
    D3D12_LOGIC_OP_EQUIV,  
    D3D12_LOGIC_OP_AND_REVERSE,  
    D3D12_LOGIC_OP_AND_INVERTED,  
    D3D12_LOGIC_OP_OR_REVERSE,  
    D3D12_LOGIC_OP_OR_INVERTED  
} D3D12_LOGIC_OP;
```

블렌딩(Blending)

- **블렌드 상태(Blend State) 설정**

- 논리 연산 설정

픽셀 쉐이더의 출력 색상과 렌더 타겟의 색상에 대한 논리적 연산으로 블렌딩 논리 연산(Bitwise Boolean Operation)과 블렌딩 연산을 동시에 설정할 수 없음
렌더 타겟의 형식이 논리 연산을 지원할 수 있는 형식(UINT)이어야 함

BlendResult = **SrcColor** **LogicalOperation** **DestColor**

```
typedef enum D3D12_LOGIC_OP {
    D3D12_LOGIC_OP_CLEAR, //렌더 타겟을 0으로 지움(BlendResult = 0)
    D3D12_LOGIC_OP_SET, //렌더 타겟을 1로 설정(BlendResult = 1)
    D3D12_LOGIC_OP_COPY, //픽셀 쉐이더의 출력을 렌더 타겟으로 복사, BlendResult = SrcColor
    D3D12_LOGIC_OP_COPY_INVERTED, //BlendResult = ~SrcColor
    D3D12_LOGIC_OP_NOOP, //BlendResult = DestColor
    D3D12_LOGIC_OP_INVERT, //BlendResult = ~DestColor
    D3D12_LOGIC_OP_AND, //논리 AND 연산, BlendResult = SrcColor & DestColor
    D3D12_LOGIC_OP_NAND, //논리 NAND 연산, BlendResult = ~(SrcColor & DestColor)
    D3D12_LOGIC_OP_OR, //논리 OR 연산, BlendResult = SrcColor | DestColor
    D3D12_LOGIC_OP_NOR, //논리 NOR 연산, BlendResult = ~(SrcColor | DestColor)
    D3D12_LOGIC_OP_XOR, //논리 XOR 연산, BlendResult = SrcColor ^ DestColor
    D3D12_LOGIC_OP_EQUIV, //논리 비교(Equal) 연산을 수행, BlendResult = ~(SrcColor ^ DestColor)
    D3D12_LOGIC_OP_AND_REVERSE, //BlendResult = SrcColor & ~DestColor
    D3D12_LOGIC_OP_AND_INVERTED, //BlendResult = ~SrcColor & DestColor
    D3D12_LOGIC_OP_OR_REVERSE, //BlendResult = SrcColor | ~DestColor
    D3D12_LOGIC_OP_OR_INVERTED //BlendResult = ~SrcColor | DestColor
} D3D12_LOGIC_OP;
```

블렌딩(Blending)

- **블렌드 상태(Blend State) 설정**

- 기본 블렌드 상태(Default Blend State)

기본 블렌드-상태

AlphaToCoverageEnable	FALSE
IndependentBlendEnable	FALSE
RenderTarget[i].BlendEnable	FALSE
RenderTarget[i].LogicOpEnable	FALSE
RenderTarget[i].SrcBlend	D3D12_BLEND_ONE
RenderTarget[i].DestBlend	D3D12_BLEND_ZERO
RenderTarget[i].BlendOp	D3D12_BLEND_OP_ADD
RenderTarget[i].SrcBlendAlpha	D3D12_BLEND_ONE
RenderTarget[i].DestBlendAlpha	D3D12_BLEND_ZERO
RenderTarget[i].BlendOpAlpha	D3D12_BLEND_OP_ADD
RenderTarget[i].LogicOp	D3D12_LOGIC_OP_NOOP
RenderTarget[i].RenderTargetWriteMask	D3D12_COLOR_WRITE_ENABLE_ALL

- 듀얼-소스 색상 블렌딩(Dual-Source Color Blending)

픽셀 쉐이더 출력(슬롯 0와 슬롯 1)을 렌더 타겟 0와 동시에 블렌딩을 수행

D3D12_BLEND_OP_(ADD, SUBTRACT, REV_SUBTRACT) 연산만 유효

SrcBlend = D3D12_BLEND_ONE;

DestBlend = D3D12_BLEND_SRC1_COLOR;

SrcColor0 * (1, 1, 1, 1) □ DestColor * SrcColor1

SrcBlend = D3D12_BLEND_SRC1_COLOR;

DestBlend = D3D12_BLEND_INV_SRC1_COLOR;

SrcColor0 * SrcColor1 □ DestColor * (1 - SrcColor1)

블렌딩(Blending)

- 차이점

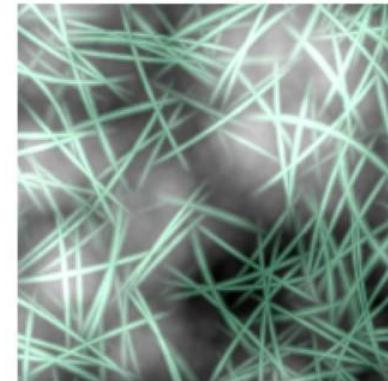
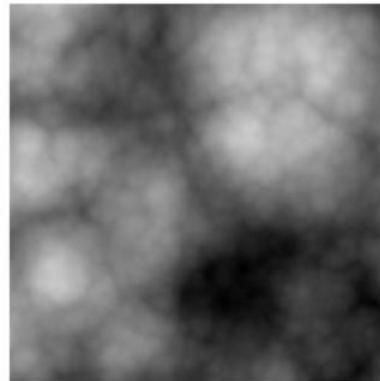
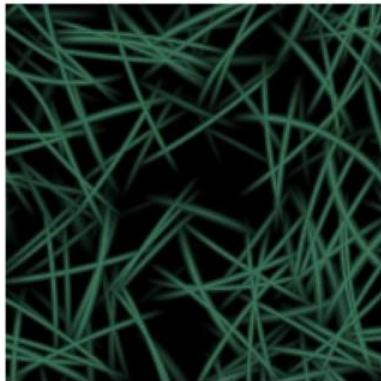


단일-패스 렌더링(Single-Pass Rendering)

다중-패스 렌더링(Multi-Pass Rendering)

블렌딩(Blending)

- 색상 블렌딩(Color Blending)

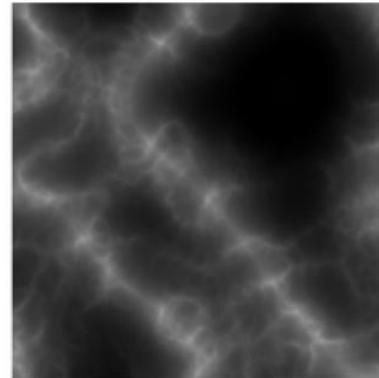


```
D3D12_BLEND_DESC d3dBlendStateDesc;  
::ZeroMemory(&d3dBlendStateDesc, sizeof(D3D12_BLEND_DESC));  
d3dBlendStateDesc.AlphaToCoverageEnable = false;  
d3dBlendStateDesc.IndependentBlendEnable = false;  
d3dBlendStateDesc.RenderTarget[0].BlendEnable = true;  
d3dBlendStateDesc.RenderTarget[0].LogicOpEnable = false;  
d3dBlendStateDesc.RenderTarget[0].SrcBlend = D3D12_BLEND_ONE; //D3D12_BLEND_ZERO ?  
d3dBlendStateDesc.RenderTarget[0].DestBlend = D3D12_BLEND_ONE;  
d3dBlendStateDesc.RenderTarget[0].BlendOp = D3D12_BLEND_OP_ADD;  
d3dBlendStateDesc.RenderTarget[0].SrcBlendAlpha = D3D12_BLEND_ONE;  
d3dBlendStateDesc.RenderTarget[0].DestBlendAlpha = D3D12_BLEND_ZERO;  
d3dBlendStateDesc.RenderTarget[0].BlendOpAlpha = D3D12_BLEND_OP_ADD;  
d3dBlendStateDesc.RenderTarget[0].LogicOp = D3D12_LOGIC_OP_NOOP;  
d3dBlendStateDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;
```

Color = SrcColor * **(1, 1, 1, 1)** + DestColor * **(1, 1, 1, 1)** = SrcColor + DestColor

블렌딩(Blending)

- 색상 블렌딩(Color Blending)

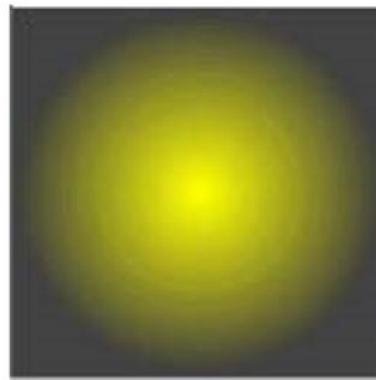


```
D3D12_BLEND_DESC d3dBlendStateDesc;  
::ZeroMemory(&d3dBlendStateDesc, sizeof(D3D12_BLEND_DESC));  
d3dBlendStateDesc.AlphaToCoverageEnable = false;  
d3dBlendStateDesc.IndependentBlendEnable = false;  
d3dBlendStateDesc.RenderTarget[0].BlendEnable = true;  
d3dBlendStateDesc.RenderTarget[0].LogicOpEnable = false;  
d3dBlendStateDesc.RenderTarget[0].SrcBlend = D3D12_BLEND_ONE;  
d3dBlendStateDesc.RenderTarget[0].DestBlend = D3D12_BLEND_ONE;  
d3dBlendStateDesc.RenderTarget[0].BlendOp = D3D12_BLEND_OP_REV_SUBTRACT;  
d3dBlendStateDesc.RenderTarget[0].SrcBlendAlpha = D3D12_BLEND_ONE;  
d3dBlendStateDesc.RenderTarget[0].DestBlendAlpha = D3D12_BLEND_ZERO;  
d3dBlendStateDesc.RenderTarget[0].BlendOpAlpha = D3D12_BLEND_OP_ADD;  
d3dBlendStateDesc.RenderTarget[0].LogicOp = D3D12_LOGIC_OP_NOOP;  
d3dBlendStateDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;
```

Color = SrcColor * **(1, 1, 1, 1)** - DestColor * **(1, 1, 1, 1)** = SrcColor - DestColor

블렌딩(Blending)

- 색상 블렌딩(Color Blending)

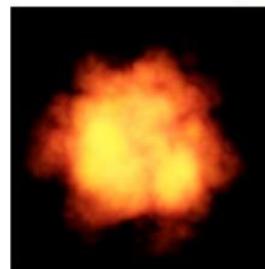


```
D3D12_BLEND_DESC d3dBlendStateDesc;  
::ZeroMemory(&d3dBlendStateDesc, sizeof(D3D12_BLEND_DESC));  
d3dBlendStateDesc.AlphaToCoverageEnable = false;  
d3dBlendStateDesc.IndependentBlendEnable = false;  
d3dBlendStateDesc.RenderTarget[0].BlendEnable = true;  
d3dBlendStateDesc.RenderTarget[0].LogicOpEnable = false;  
d3dBlendStateDesc.RenderTarget[0].SrcBlend = D3D12_BLEND_DEST_COLOR;  
d3dBlendStateDesc.RenderTarget[0].DestBlend = D3D12_BLEND_ZERO;  
d3dBlendStateDesc.RenderTarget[0].BlendOp = D3D12_BLEND_OP_ADD;  
d3dBlendStateDesc.RenderTarget[0].SrcBlendAlpha = D3D12_BLEND_ONE;  
d3dBlendStateDesc.RenderTarget[0].DestBlendAlpha = D3D12_BLEND_ZERO;  
d3dBlendStateDesc.RenderTarget[0].BlendOpAlpha = D3D12_BLEND_OP_ADD;  
d3dBlendStateDesc.RenderTarget[0].LogicOp = D3D12_LOGIC_OP_NOOP;  
d3dBlendStateDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;
```

Color = SrcColor * **DestColor** + DestColor * **(0, 0, 0, 0)** = SrcColor * DestColor

블렌딩(Blending)

- 알파 블렌딩(Alpha Blending, Transparent Blending)



Alpha	Red	Green	Blue
0	Red	Green	Blue
1	Red	Green	Blue



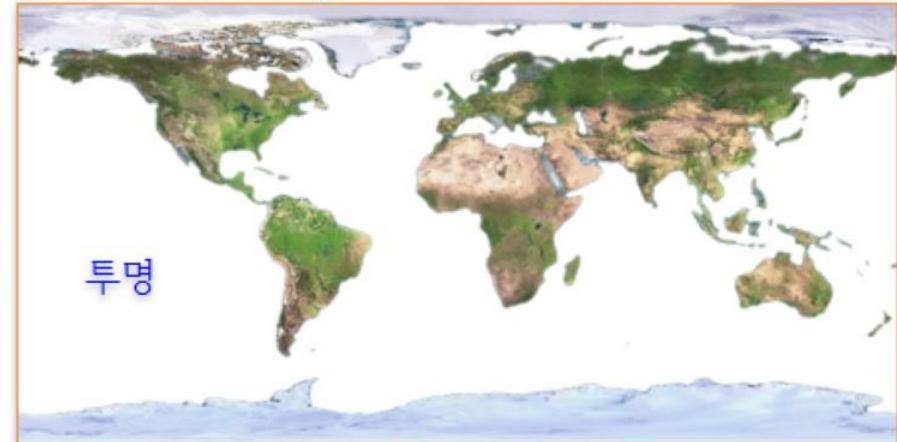
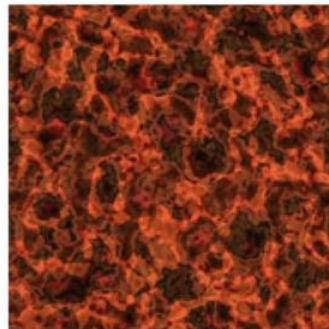
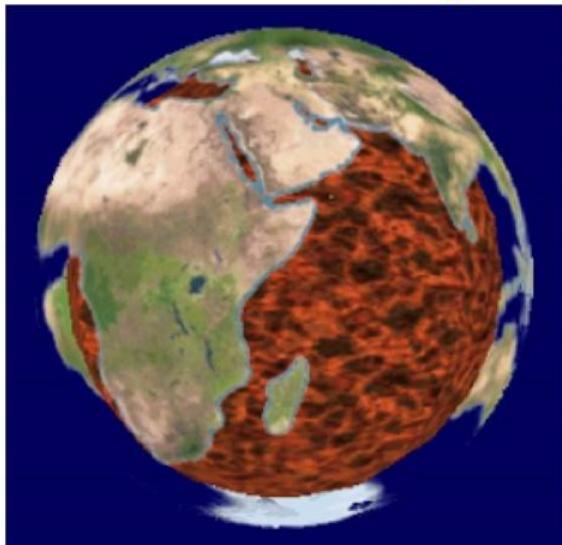
```
d3dBlendStateDesc.RenderTarget[0].BlendEnable = true;  
d3dBlendStateDesc.RenderTarget[0].SrcBlend = D3D12_BLEND_SRC_ALPHA;  
d3dBlendStateDesc.RenderTarget[0].DestBlend = D3D12_BLEND_INV_SRC_ALPHA;  
d3dBlendStateDesc.RenderTarget[0].BlendOp = D3D12_BLEND_OP_ADD;  
d3dBlendStateDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;
```

$$C = \text{SrcColor} * \text{SrcAlpha} + \text{DestColor} * (1 - \text{SrcAlpha})$$

```
float4 PSTransparent(VS_OUTPUT input) : SV_Target  
{  
    float4 cColor = gtxtTexture.Sample(gSamplerState, input.uv);  
    clip(cColor.a - 0.15f);  
    return(cColor);  
}
```

블렌딩(Blending)

- 알파 블렌딩(Alpha Blending, Transparent Blending)



투명

$(A, R, G, B) = (1, 1, 1, 1)$

```
d3dBlendStateDesc.RenderTarget[0].BlendEnable = true;  
d3dBlendStateDesc.RenderTarget[0].SrcBlend = D3D12_BLEND_INV_SRC_ALPHA;  
d3dBlendStateDesc.RenderTarget[0].DestBlend = D3D12_BLEND_SRC_ALPHA;  
d3dBlendStateDesc.RenderTarget[0].BlendOp = D3D12_BLEND_OP_ADD;  
d3dBlendStateDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;
```

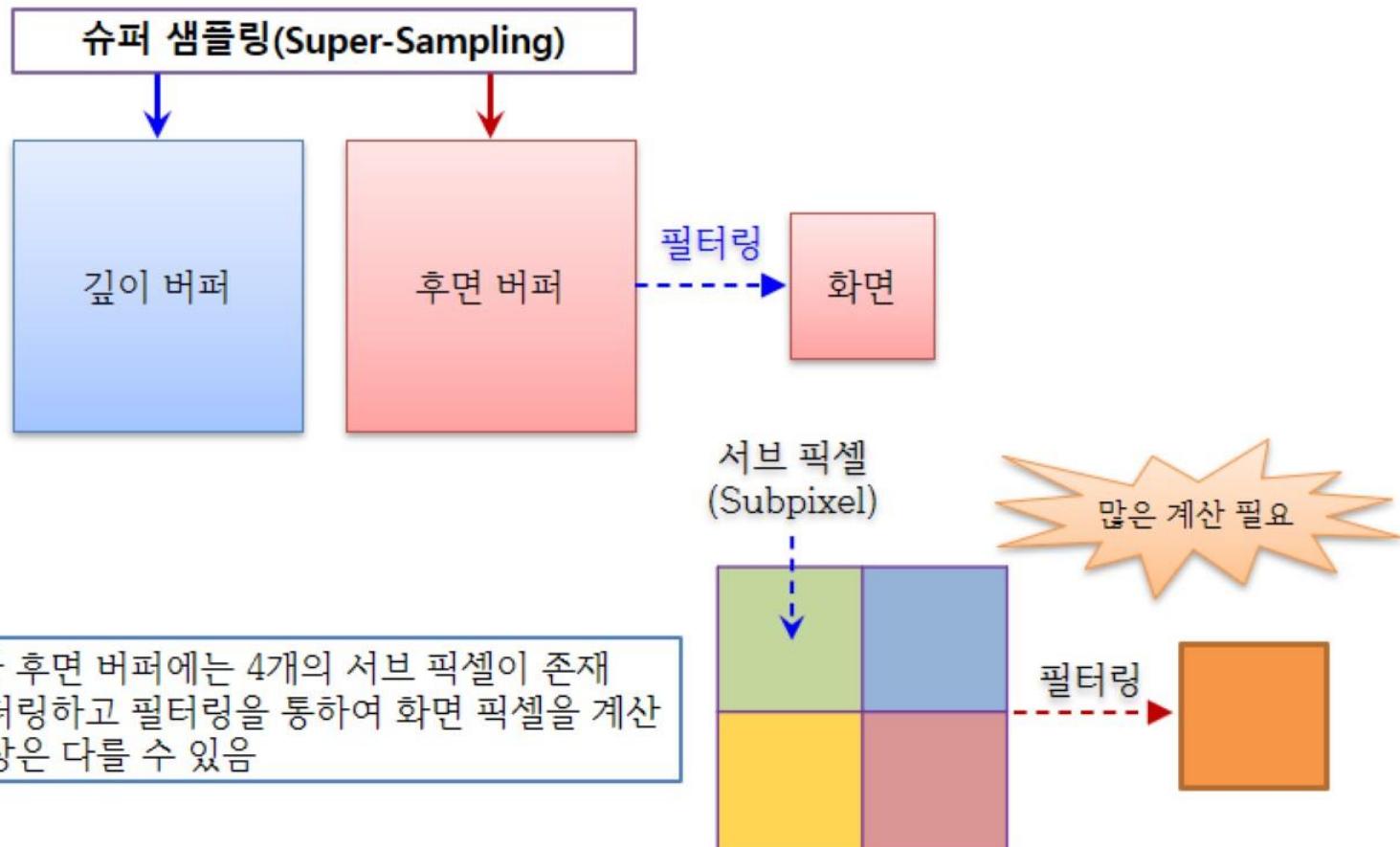
Color = SrcColor * $(1 - \text{SrcAlpha})$ + DestColor * SrcAlpha

```
float4 PSTransparent(VS_OUTPUT input) : SV_Target  
{  
    float4 cColor = gtxtTexture2.Sample(gSamplerState, input.uv0);  
    if (cColor.a > 0.95f) cColor = gtxtTexture1.Sample(gSamplerState, input.uv0);  
    return(cColor);  
}
```

Direct3D 12 디바이스

- **다중 샘플링(Multi-Sampling)**

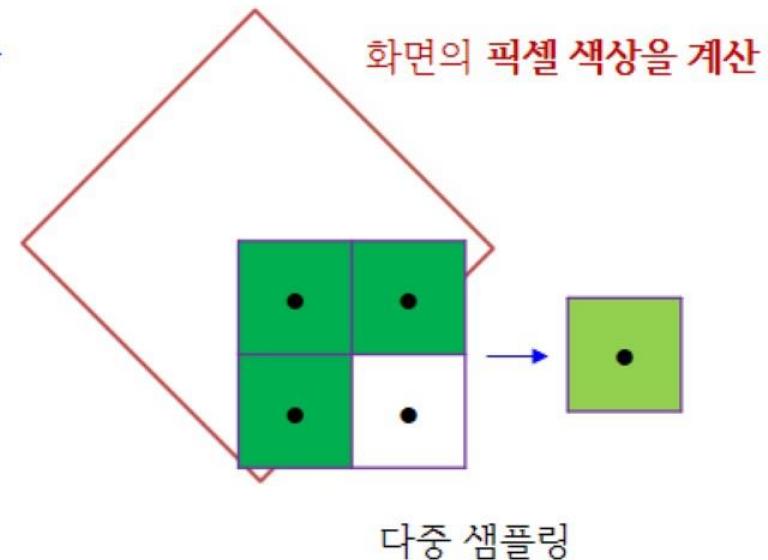
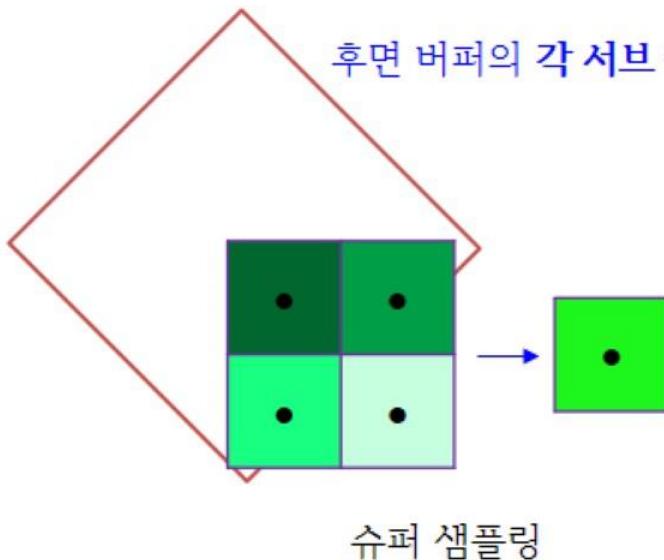
- 슈퍼 샘플링(Super-Sampling): 4X
 - 후면 버퍼와 깊이 버퍼 해상도를 화면 크기보다 4배(2×2) 크게 만들어 렌더링 렌더링할 후면 버퍼 픽셀의 개수가 4배가 됨(렌더링 시간 4배)
깊이 테스트를 할 픽셀의 개수가 4배가 됨
 - 프리젠테이션할 때 후면 버퍼를 샘플링(예: 4개 픽셀을 평균)하여 픽셀 색상을 구함



Direct3D 12 디바이스

• 다중 샘플링(Multi-Sampling)

- MSAA(Multi-Sample Anti-Aliasing): 4X
슈퍼 샘플링처럼 후면 버퍼와 깊이 버퍼를 화면보다 4배 더 크게 생성
슈퍼 샘플링보다 각 픽셀 색상을 계산하는 시간이 더 적게 걸리는 알고리즘
- 다중 샘플링은 화면의 **픽셀마다 한 번만 색상을 계산(화면 픽셀의 중심에서)**
계산된 픽셀의 색상으로 후면 버퍼의 서브 픽셀의 색상을 결정함
서브 픽셀의 색상은 서브 픽셀의 가시성과 포함 여부에 따라 결정됨
 - **가시성(Visibility):** 각 서브 픽셀이 보이는가?
가시성 판단을 위하여 깊이/스텐실 검사는 각 서브 픽셀마다 수행됨
 - **포함 여부(Coverage):** 각 서브 픽셀의 중심이 다각형의 내부에 존재하는가?
포함 여부 판단은 각 서브 픽셀마다 수행됨
계산된 픽셀의 색상을 다각형에 포함되고 보이는 모든 서브 픽셀의 색상에 복사



블렌딩(Blending)

- **알파-커버리지 다중 샘플링(Alpha-To-Coverage Multi-Sampling)**

- 나무와 풀과 같은 빌보드 객체를 그릴 때 투명한 부분을 그릴 필요가 없음
이러한 텍스쳐는 이미 **알파 값으로 투명도를 충분히 잘 표현**하고 있음
쉐이더에서 clip() 또는 discard 명령을 사용할 수 있음
경계(Edge)에서 계단 현상(Blocky Pixels)이 일어날 수 있음
- 투명 블렌딩(Transparent Blending)을 사용하면 계단 현상을 줄일 수 있음
투명 블렌딩의 문제점: 정렬(Back-to-Front Rendering)
- 다중 샘플링: 계단 현상을 없애기 위한 방법(계단 현상은 경계에서 주로 발생)
다중 샘플링에서 포함 여부는 다각형 수준에서 계산됨(알파 값을 사용하지 않음)
- 포함 여부를 계산할 때 알파 값을 사용하도록 설정(**AlphaToCoverageEnable**)
렌더 타겟과 깊이 버퍼가 다중 샘플링을 지원하도록 생성되어야 함
픽셀 쉐이더의 출력(SV_Target0)의 알파 값을 서브 픽셀의 포함 여부 계산에 사용함

```
D3D12_BLEND_DESC d3dBlendStateDesc;  
::ZeroMemory(&d3dBlendStateDesc, sizeof(D3D12_BLEND_DESC));  
d3dBlendStateDesc.AlphaToCoverageEnable = true;  
d3dBlendStateDesc.IndependentBlendEnable = false;  
d3dBlendStateDesc.RenderTarget[0].BlendEnable = false;  
d3dBlendStateDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;
```



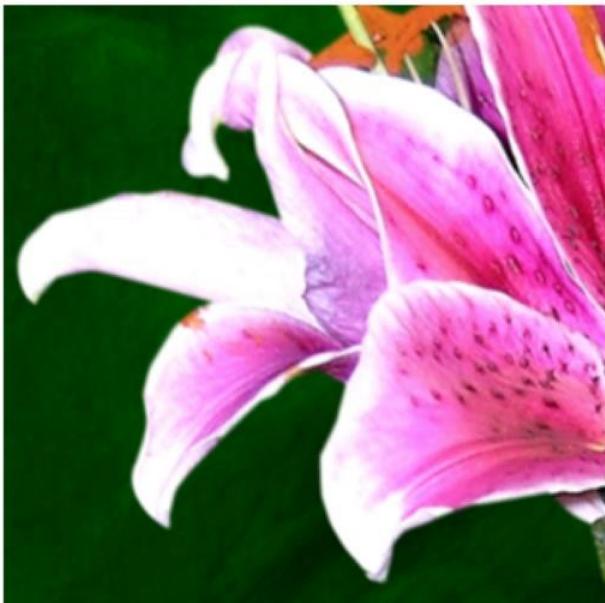
블렌딩(Blending)

- 알파-커버리지 다중 샘플링(Alpha-To-Coverage Multi-Sampling)

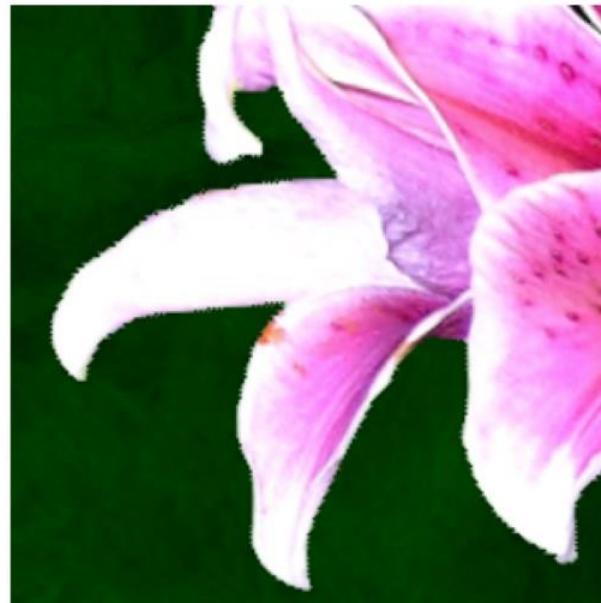
텍스쳐 이미지



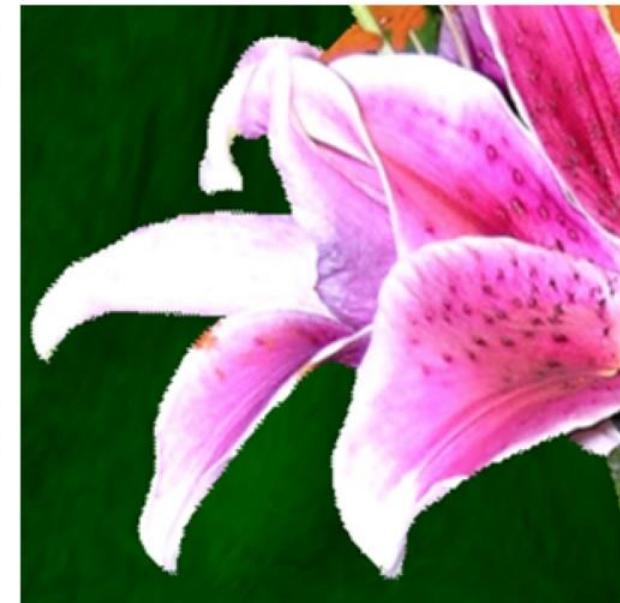
알파 이미지



블렌딩



블렌딩/Alpah-to-Coverage



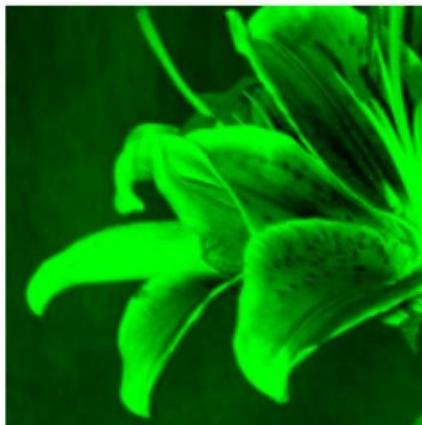
Alpah-to-Coverage

블렌딩(Blending)

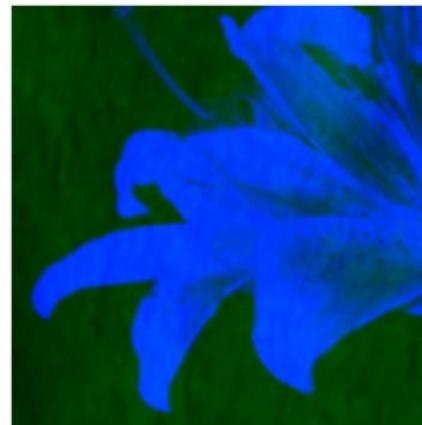
- 렌더 타겟 출력 마스크(RenderTargetWriteMask)



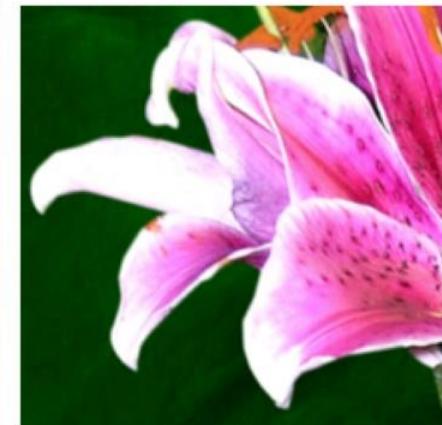
RED



GREEN



BLUE



ALL

```
d3dBlendStateDesc.RenderTarget[0].BlendEnable = true;  
d3dBlendStateDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_RED;
```

```
d3dBlendStateDesc.RenderTarget[0].BlendEnable = true;  
d3dBlendStateDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_GREEN;
```

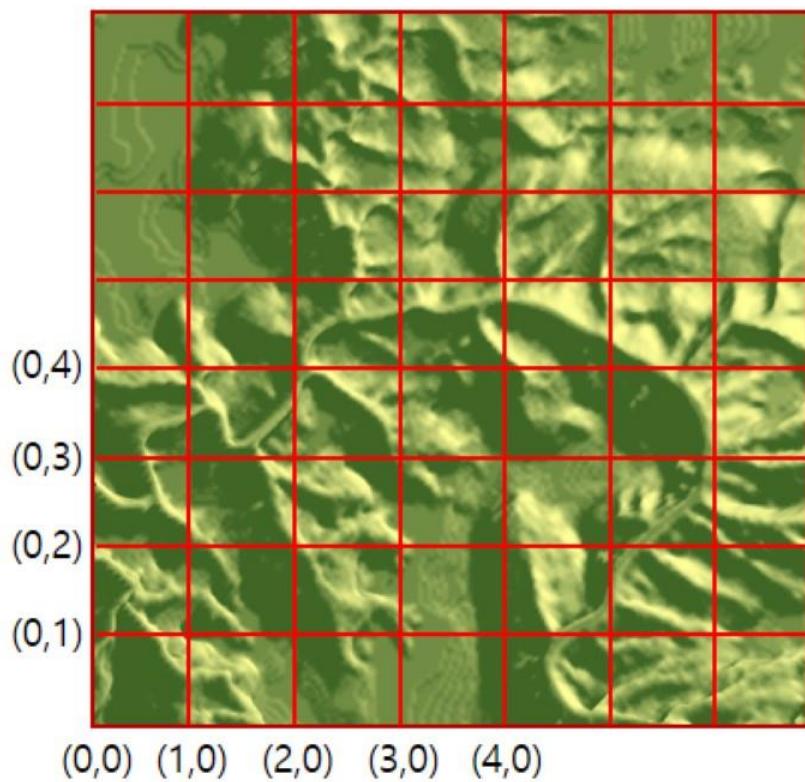
```
d3dBlendStateDesc.RenderTarget[0].BlendEnable = true;  
d3dBlendStateDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_BLUE;
```

```
d3dBlendStateDesc.RenderTarget[0].BlendEnable = true;  
d3dBlendStateDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;
```

블렌딩(Blending)

- **텍스쳐 스플래팅(Texture Splatting)**

- 타일 형태의 텍스쳐 레이어(Layer)를 사용하여 렌더링하는 것
- 레이어는 텍스쳐들과 텍스쳐 변환 행렬을 포함하는 구조체
텍스쳐 변환 행렬은 텍스쳐 좌표를 변환(크기 변환, 회전 변환)
(행렬을 사용하여 텍스쳐가 2×2 사각형 또는 8×8 사각형들에 매핑되게 할 수 있음)

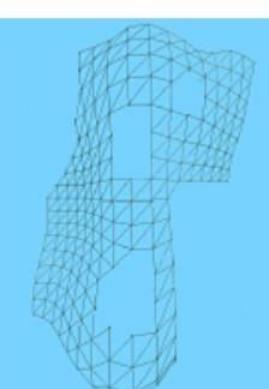
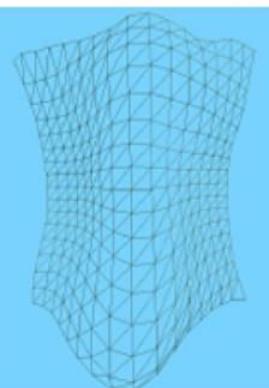


텍스쳐 좌표의 크기 변환, 0.5배, 0.1배, ...

블렌딩(Blending)

• 텍스쳐 스플래팅(Texture Splatting)

- 스플랫(Splat)은 레이어의 텍스쳐를 사용하는 사각형(인덱스)들의 집합
각 스플랫은 알파(Alpha) 텍스쳐를 포함(타일 형태로 매핑하지 않음)
- 레이어와 스플랫의 개수는 같음
- 색상은 기본 텍스쳐에서 샘플링, 알파값은 알파 텍스쳐에서 샘플링
- 알파 텍스쳐를 사용하여 각 스플랫을 렌더링하면 각 스플랫을 부드럽게 블렌딩



```
class CTerrain
{
    ID3D12Resource
    CTerrainLayer
    CTerrainSplat
};

class CTerrainLayer
{
    D3D12_GPU_DESCRIPTOR_HANDLE
    D3D12_GPU_DESCRIPTOR_HANDLE
    XMATRIX
};

class CTerrainSplat
{
    ID3D12Resource
    DWORD
    DWORD
};
```

*m_pd3dVertices;
*m_pLayers;
*m_pSplats;

정점은 2개의 텍스쳐 좌표를 가짐
(타일 텍스쳐 매핑, 알파 텍스쳐 매핑)

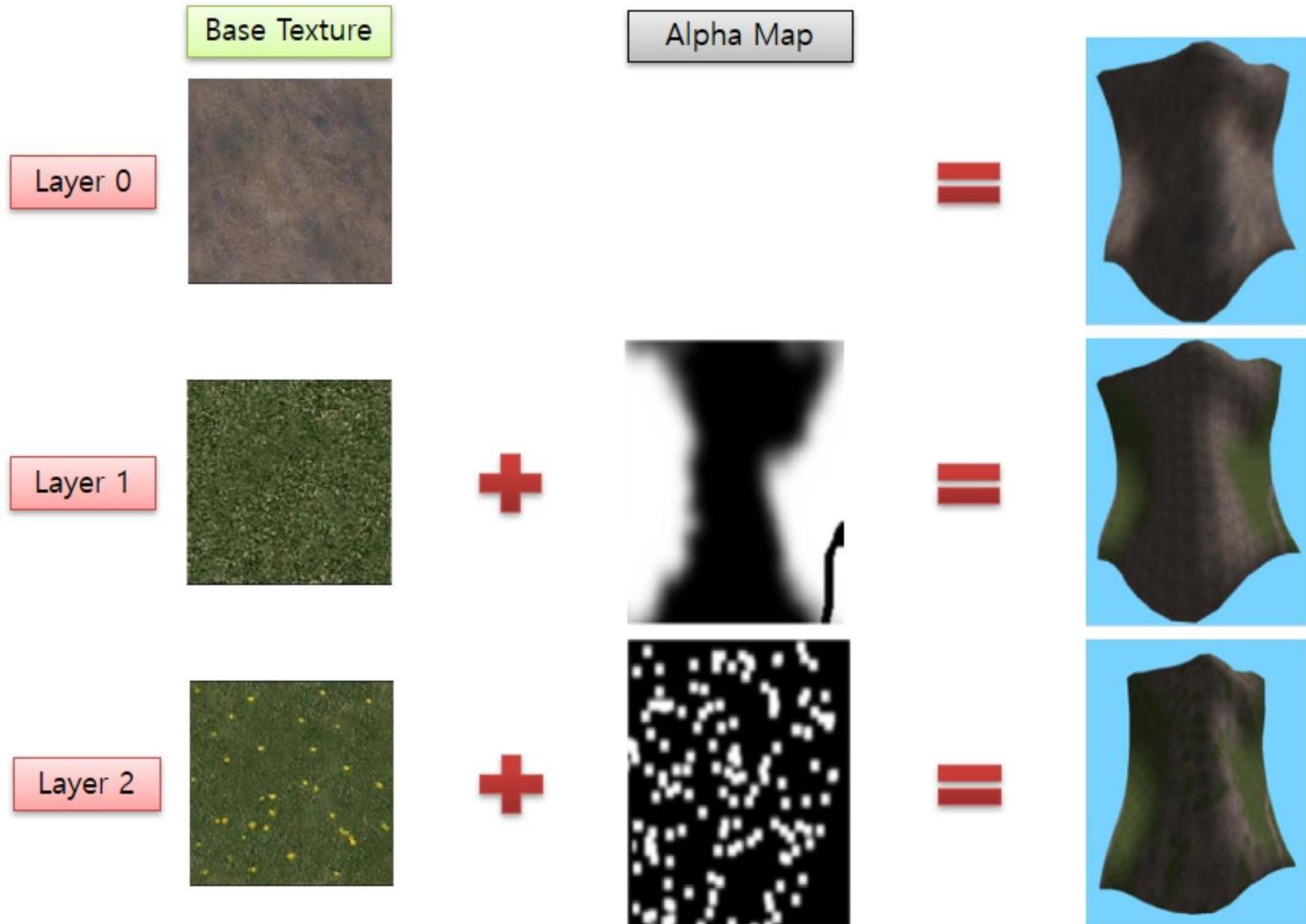
m_d3dSrvBaseTexture;
m_d3dSrvAlphaBlendTexture;
m_xmmtxTexture;

알파 텍스쳐는 지형 전체에 매핑

*m_pd3dxSplatFaces;
m_nIndices;
m_nPrimitives;

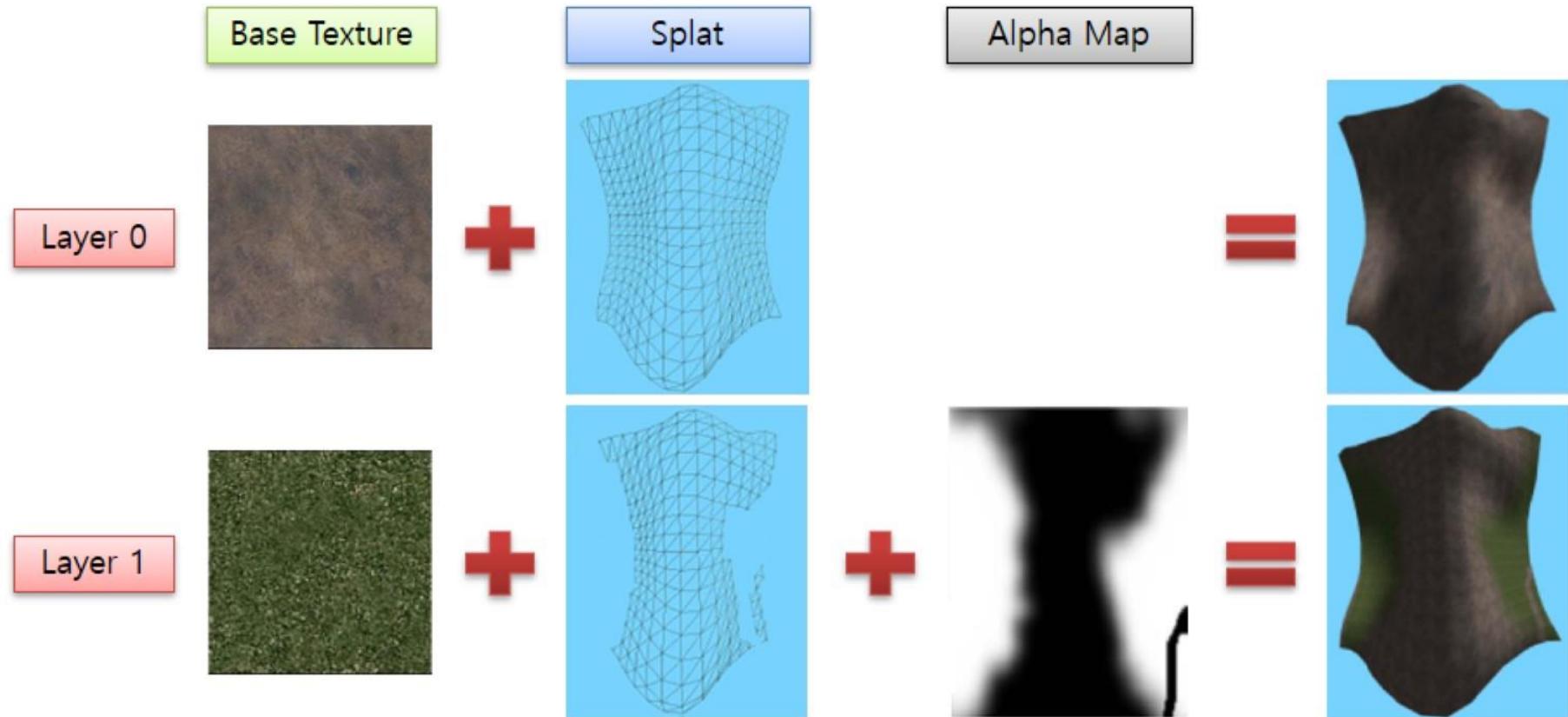
블렌딩(Blending)

- 텍스쳐 스플래팅(Texture Splatting)



블렌딩(Blending)

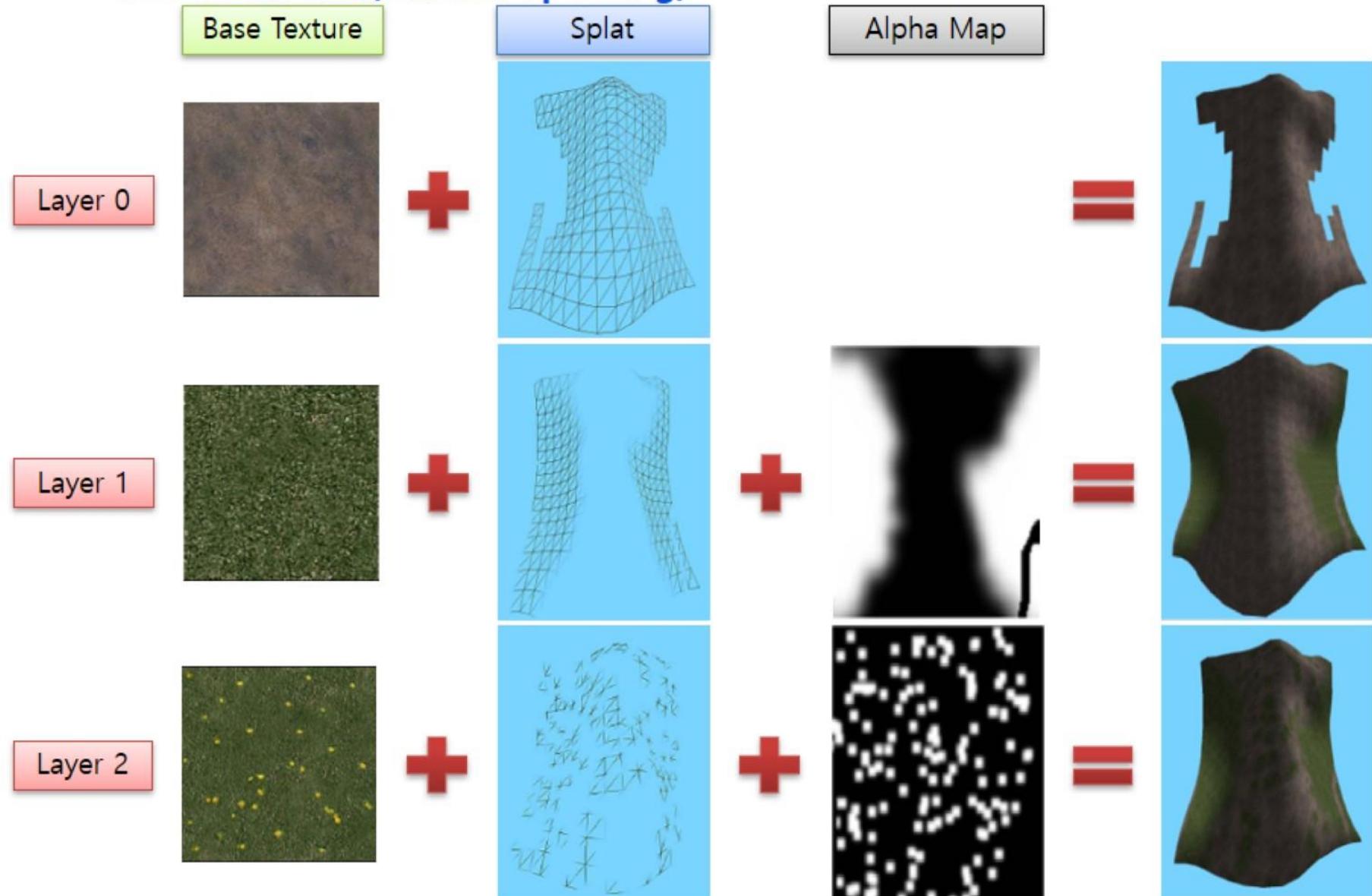
- 텍스쳐 스플래팅(Texture Splatting)



레이어 1의 불투명한 사각형 영역과 겹치는 레이어 0의 사각형 영역은 레이어 1을 렌더링하고 나면 보이지 않는다(Overwrite). 그러므로 이 영역은 레이어 0에서 그리지 않아도 된다. 즉, 레이어 0의 스플랫에서 이런 영역의 사각형들을 제거할 수 있다. 높은 번호의 레이어에서 불투명한 영역은 낮은 레이어에서 모두 제거할 수 있다.

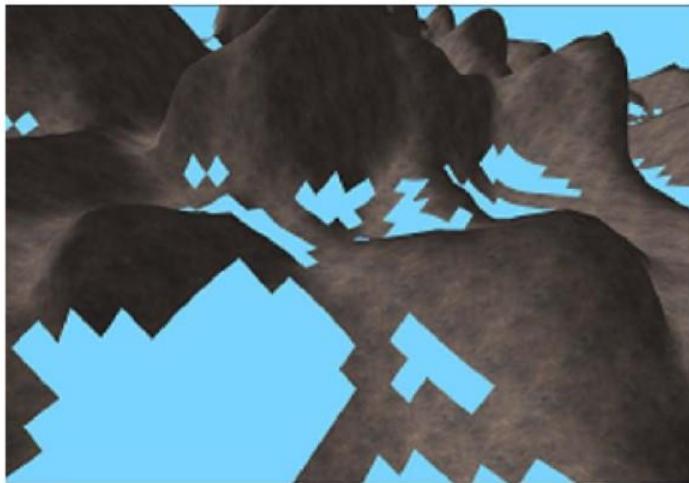
블렌딩(Blending)

- 텍스쳐 스플래팅(Texture Splatting)

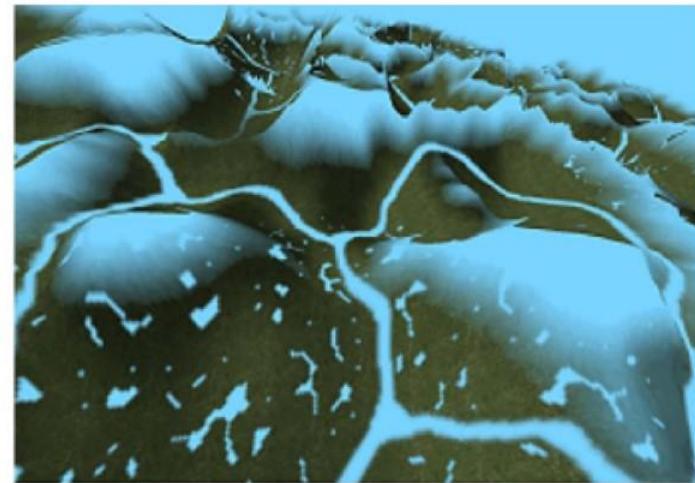


블렌딩(Blending)

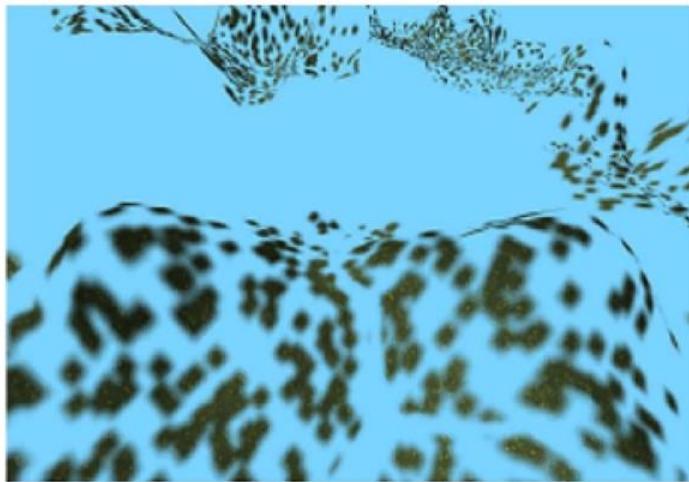
- 텍스쳐 스플래팅(Texture Splatting)



Base Layer – Concrete Splat



1st Layer – Grass Splat



2nd Layer – Flowery Splat

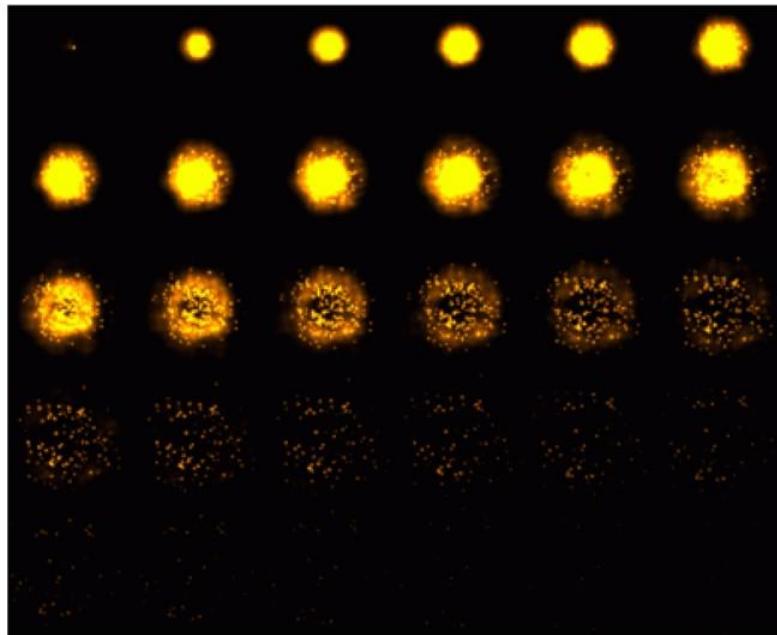


렌더링 결과

블렌딩(Blending)

- 텍스트(Text) 출력
- 2D 애니메이션
- 텍스쳐 이펙트(Effect)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

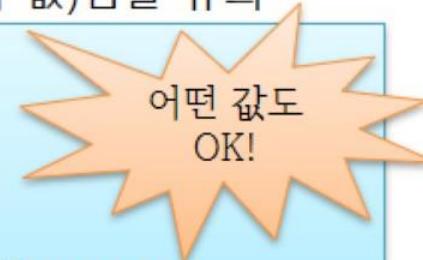


블렌딩(Blending)

• 고려 사항

- 블렌딩 연산의 소스(Source)는 픽셀 쉐이더의 출력 값(색상, 알파 값)임을 유의

```
float4 PSTransparent(VS_OUTPUT input) : SV_Target
{
    float4 cSourceColor;
    cSourceColor.rgb = ...
    float4 cTexture = gtxtTexture.Sample(gSamplerState, input.uv);
    cSourceColor.a = cTexture.a * gcMaterial.a; //cSourceColor.a = cTexture.r;
    return(cSourceColor);
}
```



어떤 값도
OK!

- 빌보드가 아닌 경우 은면 제거(Back Face Culling)를 하지 않아야 함
- 알파 블렌딩을 위한 텍스쳐 픽셀의 사전 계산

$$\text{Color} = \text{SrcColor} * (1 - \text{SrcAlpha}) + \text{DestColor} * \text{SrcAlpha}$$

텍스쳐의 각 텍셀에 대하여 $\text{SrcColor} * (1 - \text{SrcAlpha})$ 의 값은 미리 계산할 수 있음

텍스쳐 텍셀 값을 $\text{SrcColor} * (1 - \text{SrcAlpha})$ 로 저장하고 블렌딩 수식으로 다음을 사용

$$\text{Color} = \text{SrcColor} * 1 + \text{DestColor} * \text{SrcAlpha}$$

- 쉐이더 프로그램에서 같은 연산을 두 번 이상 계산하는 가를 점검
- 나무를 그릴 때 인스턴싱을 쓰면 많은 개수의 나무를 그릴 때 효율적일 수 있음
인스턴싱을 위해 각 나무 객체의 월드 변환 행렬을 파이프라인으로 보내야 함
나무가 1000 개라면 $(16 * 4\text{바이트}) * 1000 = 64,000\text{바이트}$ 필요
더 효율적인 방법이 있을까?
파이프라인으로 보내는 데이터의 양을 줄일 수 있는 방법이 있을까?

블렌딩(Blending)

- 렌더링 순서(Rendering Order)

- 블렌딩의 결과 색상은 투명(완전 투명, 부분 투명)하거나 불투명할 수 있음
- 블렌딩의 결과가 프레임 버퍼에 쓰여질 때 깊이 버퍼에도 깊이 값이 쓰여짐
- 깊이 검사(Depth Test)를 하는 경우 렌더링 순서에 따라 결과가 달라질 수 있음
- 깊이 검사 또는 깊이 버퍼를 사용하는 이유는?
- 불투명한 다각형(Non-alpha Polygon)만을 렌더링하는 경우
- 불투명한 다각형들과 투명한 다각형들을 섞어서 렌더링하는 경우



유리를 먼저 렌더링



유리를 나중에 렌더링

- 불투명한 다각형들을 먼저 렌더링하고 (부분)투명한 다각형들을 나중에 렌더링함
(부분)투명한 다각형들을 렌더링할 때 깊이 검사를 하면 문제가 발생(어떤 문제?)
- 해결책(Solution)**
 - 투명한 다각형들은 카메라까지의 거리에 따라 정렬(Sorting)
 - 카메라에서 거리가 먼 다각형을 먼저 렌더링(Back-to-Front Rendering)

블렌딩(Blending)

- 렌더링 순서(Rendering Order)

- 씬을 렌더링할 때 블렌딩 연산이 포함되면 항상 불투명 다각형을 먼저 렌더링 불투명한 다각형들로 렌더링된 프레임 버퍼에 (부분)투명한 다각형들을 블렌딩
- (부분)투명한 객체 또는 다각형들을 렌더링하는 순서를 어떻게 결정할 것인가?
블렌딩 연산이 더하기(+)로만 구성되면 렌더링하는 순서는 상관없음
블렌딩 연산이 빼기(-)로만 구성되면 렌더링하는 순서는 상관없음
블렌딩 연산이 곱하기(*)로만 구성되면 렌더링하는 순서는 상관없음

$$\text{Color} = \text{SrcColor} * (1, 1, 1, 1) + \text{DestColor} * (1, 1, 1, 1) = \text{SrcColor} + \text{DestColor}$$

$$\text{Color} = \text{SrcColor} * \text{DestColor} + \text{DestColor} * (0, 0, 0, 0) = \text{SrcColor} * \text{DestColor}$$

$$\text{Color} = \text{SrcColor} * (1, 1, 1, 1) - \text{DestColor} * (1, 1, 1, 1) = \text{SrcColor} - \text{DestColor}$$

- 불투명한 객체 또는 다각형을 렌더링한 프레임 버퍼: F
렌더링할 (부분)투명한 객체 또는 다각형: $B_1, B_2, B_3, \dots, B_n$
- 블렌딩 연산이 더하기, 빼기, 또는 곱하기로만 구성되면 투명한 다각형을 어떤 순서로 렌더링 하더라도 결과가 같으므로 **투명한 다각형을 정렬하지 않아도 됨**
- 깊이 검사(쓰기)를 사용하면 블렌딩의 결과가 달라질 수 있음
깊이 검사(쓰기)를 사용하지 않으면 다각형들의 가려지는 관계에 상관없이 블렌딩

블렌딩(Blending)

- 렌더링 순서(Rendering Order)

- 블렌딩 연산이 더하기, 빼기, 곱하기 중 하나가 아닌 경우

$$\text{Color} = \text{SrcColor} * \text{SrcAlpha} + \text{DestColor} * (1 - \text{SrcAlpha})$$

- 부분 투명한 다각형들을 렌더링하는 경우

부분 투명한 사각형(Blue, Green, Red) 3개를 렌더링하는 경우

예제 1: Blue * Green * Red

① Blue 사각형을 렌더링

Frame Buffer는 Blue = (0, 0, 1.0)

② Green 사각형을 렌더링(블렌딩)

$$\text{Color} = \text{Green} * 0.5 + \text{Blue} * (1 - 0.5)$$

$$\text{Color} = (0, 1.0, 0) * 0.5 + (0, 0, 1.0) * 0.5 = (0, 0.5, 0) + (0, 0, 0.5) = (0, 0.5, 0.5) = \text{Murky Blue}$$

③ Red 사각형을 렌더링(블렌딩)

$$\text{Color} = \text{Red} * 0.5 + \text{MurkyBlue} * (1 - 0.5) = (1.0, 0, 0) * 0.5 + (0.0, 0.5, 0.5) * 0.5 = (0.5, 0.25, 0.25)$$

결과 = (0.5, 0.25, 0.25) = Brownish Color

SrcAlpha = 0.5

예제 2: Blue * Red * Green

① Blue 사각형을 렌더링

Frame Buffer는 Blue = (0, 0, 1.0)

② Red 사각형을 렌더링(블렌딩)

$$\text{Color} = \text{Red} * 0.5 + \text{Blue} * (1 - 0.5)$$

$$\text{Color} = (1.0, 0, 0) * 0.5 + (0, 0, 1.0) * 0.5 = (0.5, 0, 0) + (0, 0, 0.5) = (0.5, 0, 0.5) = \text{Purple}$$

③ Green 사각형을 렌더링(블렌딩)

$$\text{Color} = \text{Green} * 0.5 + \text{Purple} * (1 - 0.5) = (0, 1.0, 0) * 0.5 + (0.5, 0, 0.5) * 0.5 = (0.25, 0.5, 0.25)$$

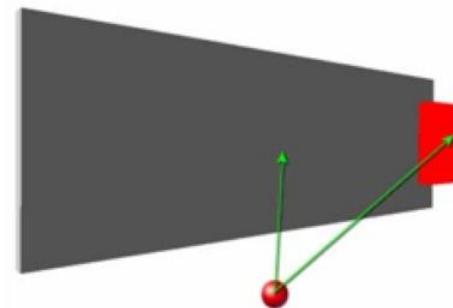
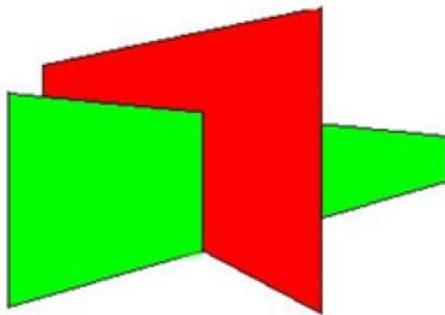
결과 = (0.25, 0.5, 0.25) = Grayish Green Color

정렬이 필요!

블렌딩(Blending)

- 렌더링 순서(Rendering Order)

- (부분)투명한 다각형들을 정렬(Sorting)할 때 고려할 사항
프레임마다 (부분)투명한 다각형들을 정렬하는 시간 때문에 비효율의 가능성
정렬을 정확하게 할 수 없는 상황(Sorting Paradox)이 발생
카메라와 거리를 측정하는 기준의 문제, 정렬 알고리즘(Quick/Bubble/...)의 선택



일반적으로 부정확하게 정렬을 해도(즉, 블렌딩이 부정확해도) 큰(?) 문제가 아님

- (부분)투명한 다각형을 렌더링할 때 깊이 값을 깊이 버퍼에 쓰지 않는 것이 해결책

```
typedef struct D3D12_DEPTH_STENCIL_DESC {  
    BOOL DepthEnable;  
    D3D12_DEPTH_WRITE_MASK DepthWriteMask;  
    D3D12_COMPARISON_FUNC DepthFunc;  
    BOOL StencilEnable;  
    UINT8 StencilReadMask;  
    UINT8 StencilWriteMask;  
    D3D12_DEPTH_STENCILOP_DESC FrontFace;  
    D3D12_DEPTH_STENCILOP_DESC BackFace;  
} D3D12_DEPTH_STENCIL_DESC;
```

```
typedef enum D3D12_DEPTH_WRITE_MASK {  
    D3D12_DEPTH_WRITE_MASK_ZERO,  
    D3D12_DEPTH_WRITE_MASK_ALL  
} D3D12_DEPTH_WRITE_MASK;
```

블렌딩(Blending)

- **안개(Fog)**

- 안개(공기에 포함된 물과 먼지 입자 때문에 빛이 산란되는 현상)
 - 렌더링의 최적화를 위해 사용
 - 절두체의 원평면(Far Plane) 바깥쪽에 있는 물체들은 보이지(그려지지) 않음
 - 카메라가 전진하면 바깥쪽 물체들이 갑자기 보이게 됨(Popping)
 - 안개를 사용하면 이러한 현상을 부드럽게 할 수 있음
 - 프레임 레이트를 유지하기 위하여 안개 효과를 사용할 수도 있음
 - 안개는 색상 블렌딩의 형태로 구현할 수 있음
 - 카메라와 물체까지의 거리에 따라 블렌딩 색상을 결정

$$C_{\text{final}} = F_{\text{factor}} \times C_{\text{original}} + (1 - F_{\text{factor}}) \times C_{\text{fog}}$$

C_{final} : 안개를 적용한 결과 색상

C_{original} : 안개를 적용하기 전의 색상

F_{factor} : 안개 인자(Fog Factor), 0.0 ~ 1.0, 카메라까지의 거리에 반비례

C_{fog} : 안개 색상



블렌딩(Blending)

• 안개(Fog)

- 안개 효과는 정점 또는 픽셀에 대하여 구현 가능
- 정점 안개(Vertex Fog)
각 정점에 대하여 안개 효과(안개 인자)를 계산(래스터라이저 단계에서 보간)
- 픽셀 안개(Pixel Fog)
각 픽셀에 대하여 안개 효과를 직접 계산
- 안개 계산식

$$C_{\text{final}} = F_{\text{factor}} \times C_{\text{original}} + (1 - F_{\text{factor}}) \times C_{\text{fog}}$$

① 선형 안개(Linear Fog)

$$F_{\text{factor}} = \frac{end - d}{(end - start)}$$

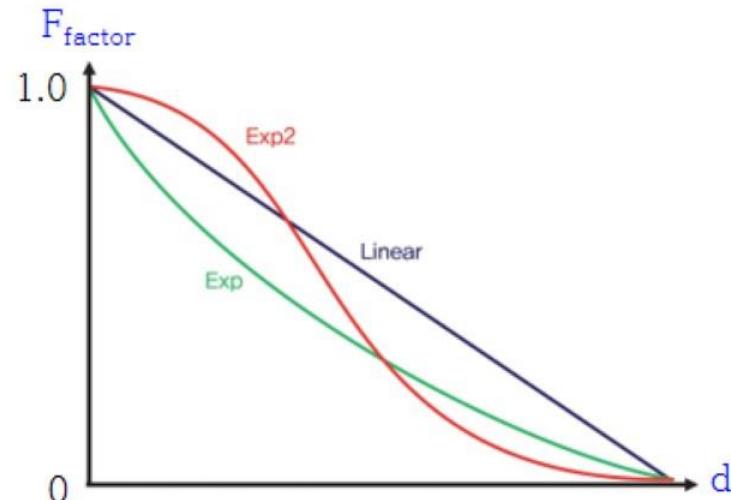
start : 안개 시작 거리, end = 안개 종료 거리
density : 안개 농도(0.0~1.0)
d : 카메라까지의 거리

② 비선형 안개(Exponential Fog)

$$F_{\text{factor}} = \frac{1}{e^{d * \text{density}}}$$

③ 비선형 안개(Exponential Squared Fog)

$$F_{\text{factor}} = \frac{1}{e^{(d * \text{density})^2}}$$



블렌딩(Blending)

- **안개(Fog)**

- **정점 안개(Vertex Fog)**

- 카메라와 정점 사이의 거리 계산을 직접 아니면 다른 방법으로 할 것인가?

- 카메라와 정점 사이의 거리 계산

- 각 정점과 카메라 사이의 거리를 직접 계산하면 시간이 많이 걸림

- 각 정점과 카메라 사이의 거리는 정점의 카메라 좌표계의 Z-값을 사용하여 근사
이 값은 카메라와 정점 사이의 실제 거리가 아님(카메라 좌표계의 Z-좌표)

- W-친화 투영 변환 행렬(W-Friendly Projection Matrix)

- 투영 변환 행렬의 W 열이 (0, 0, 1, 0)인 행렬

- 이 행렬을 사용하여 카메라 좌표 (x, y, z)를 변환하면 W 값이 z 값이 된다.
 $(x, y, z, 1) \Rightarrow (X, Y, Z=\text{depth buffer value}, W=z)$

- D3DXMatrixPerspectiveLH() 함수가 반환하는 행렬은 W-친화 투영 행렬이다.

- 투영 행렬 변환 행렬이 W-친화 투영 행렬이 아니면 m34로 나눈다.

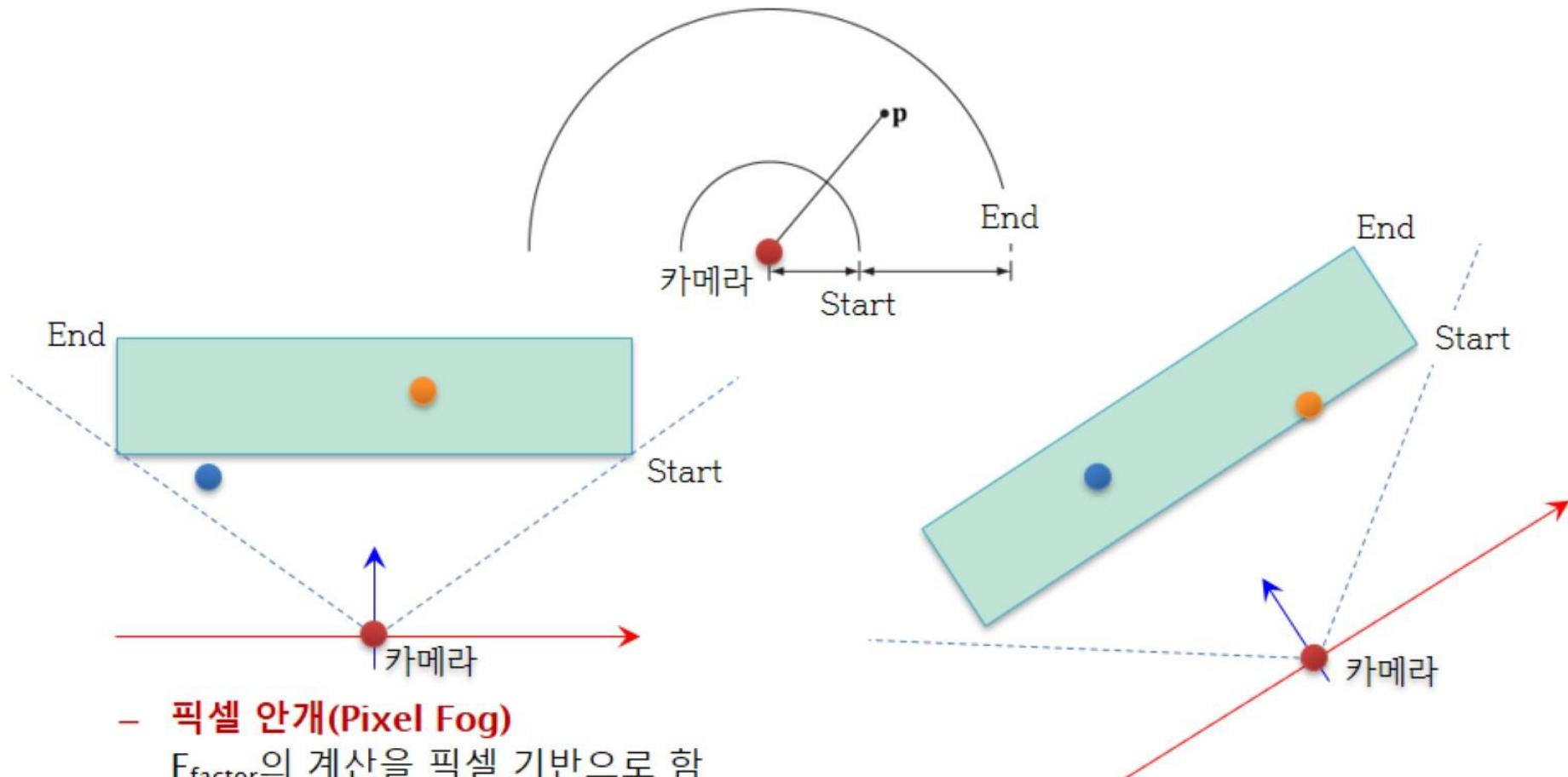
$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & e \\ 0 & 0 & d & 0 \end{pmatrix} \rightarrow \begin{pmatrix} a/e & 0 & 0 & 0 \\ 0 & b/e & 0 & 0 \\ 0 & 0 & c/e & 1 \\ 0 & 0 & d/e & 0 \end{pmatrix}$$

블렌딩(Blending)

- **안개(Fog)**

- 정점 안개(Vertex Fog)

카메라 좌표계의 Z-값을 사용하면 카메라가 회전할 때 문제가 발생할 수 있음



- **픽셀 안개(Pixel Fog)**

F_{factor} 의 계산을 픽셀 기반으로 함

고정된 거리에 대하여 F_{factor} 를 미리 계산하고 테이블을 생성

픽셀의 카메라 좌표계 z-값을 사용하여 F_{factor} 를 읽음

블렌딩(Blending)

- 안개(Fog)

```
#define LINEAR_FOG 1.0f
#define EXP_FOG      2.0f
#define EXP2_FOG     3.0f

cbuffer cbCamera : register(cb2) {
    float4 gvCameraPosition;
};
```

```
cbuffer cbFog : register(cb3) {
    float4 gcFogColor;
    float4 gvFogParameter; // (Mode, Start, End, Density)
};
```

```
float4 Fog(float4 cColor, float3 vPosition) {
    float3 vCameraPosition = gvCameraPosition.xyz;
    float3 vPositionToCamera = vCameraPosition - vPosition;
    float fDistanceToCamera = length(vPositionToCamera);
    float fFogFactor = 0.0f;
    if (gvFogParameter.x == LINEAR_FOG) {
        float fFogRange = gvFogParameter.z - gvFogParameter.y;
        fFogFactor = saturate((gvFogParameter.z - fDistanceToCamera) / fFogRange);
    }
    float4 cColorByFog = lerp(cColor, gcFogColor, fFogFactor);
    return(cColorByFog);
}
```

End 대신에 Range = (End - Start)를 사용하면 계산 효율

$$F_{factor} = \frac{end - d}{end - start}$$

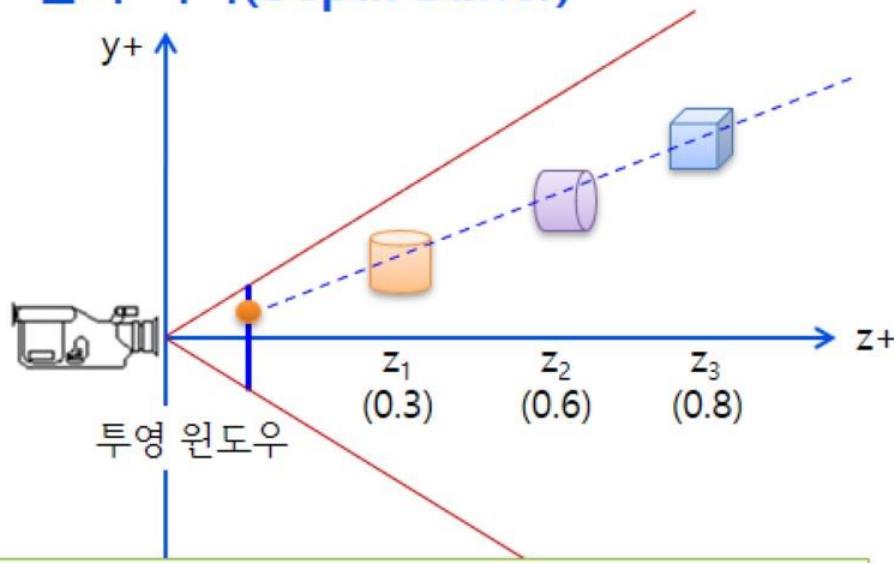
$$F_{factor} = \frac{1}{e^{d * density}}$$

$$F_{factor} = \frac{1}{e^{(d * density)^2}}$$

$$C_{final} = F_{factor} \times C_{original} + (1 - F_{factor}) \times C_{fog}$$

깊이 검사(Depth Test)

- 깊이 버퍼(Depth Buffer)

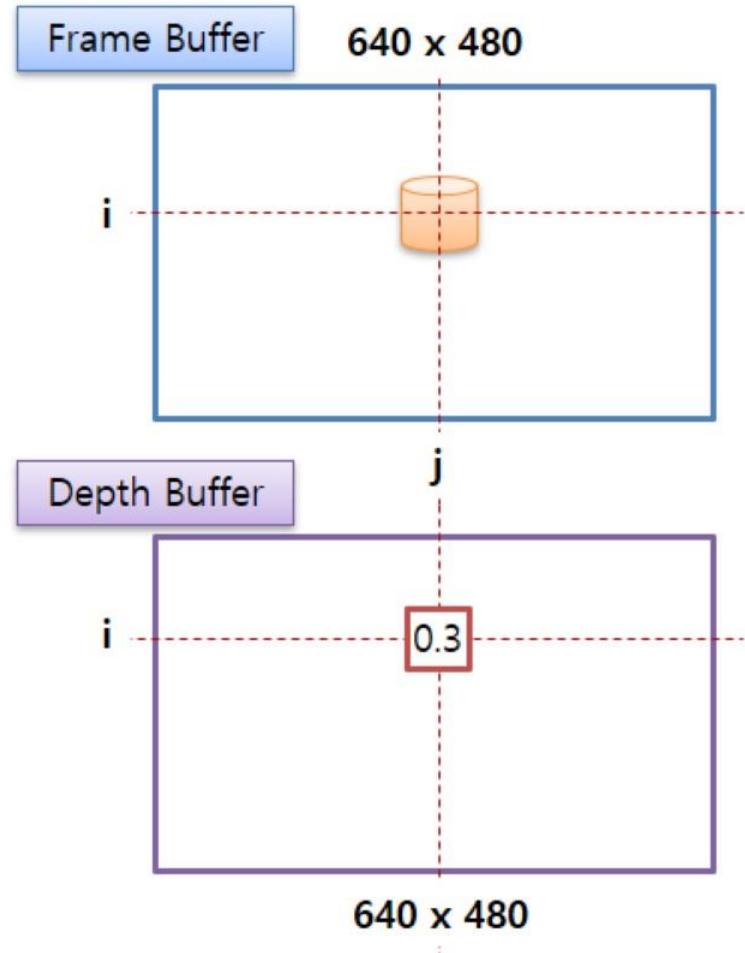


```
for (i = 0; i < 480; i++)  
{  
    for (j = 0; j < 640; j++) DepthBuffer[i][j] = 1.0f;  
}
```

메쉬의 렌더링으로 T&L 과정이 수행됨

화면 픽셀 좌표 (i, j) 로 매핑되는 점의 색상: $fColor$
화면 픽셀 좌표 (i, j) 로 매핑되는 점의 깊이값: $zDepth$

```
if (zDepth < DepthBuffer[i][j]) // 깊이 값 비교  
{  
    DepthBuffer[i][j] = zDepth;  
    FrameBuffer[i][j] = fColor;  
}
```

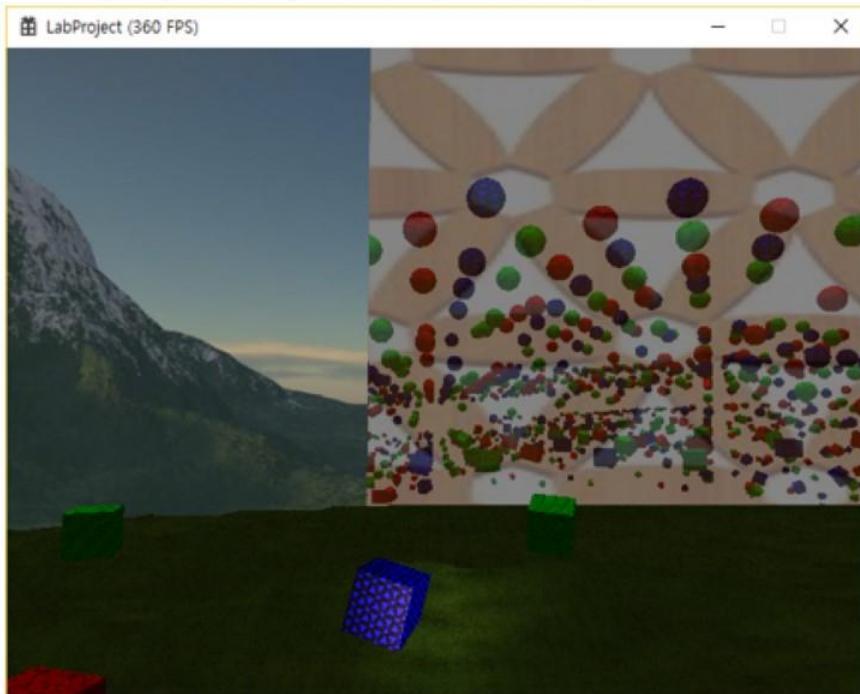


Painter's Algorithm

스텐실 검사(Stencil Test)

- **스텐실 검사(Stencil Test)**

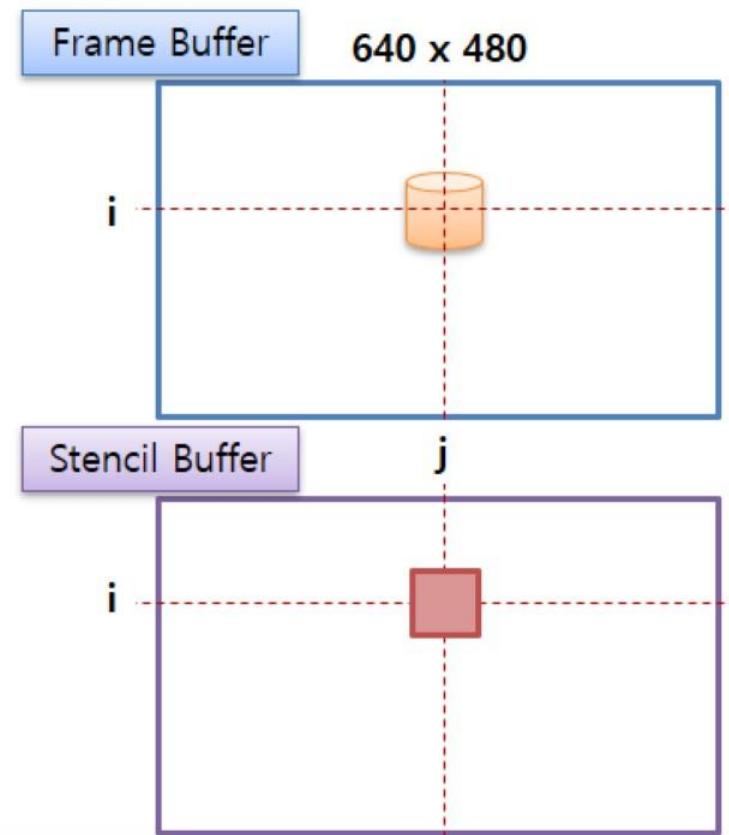
- 렌더 타겟의 일부 영역을 렌더링하지 않도록 설정(Block, Mask)
물감으로 색칠을 할 때 양초로 그린 부분이 색칠이 되지 않는 것과 유사
- 거울(Mirror) 반사의 표현 예



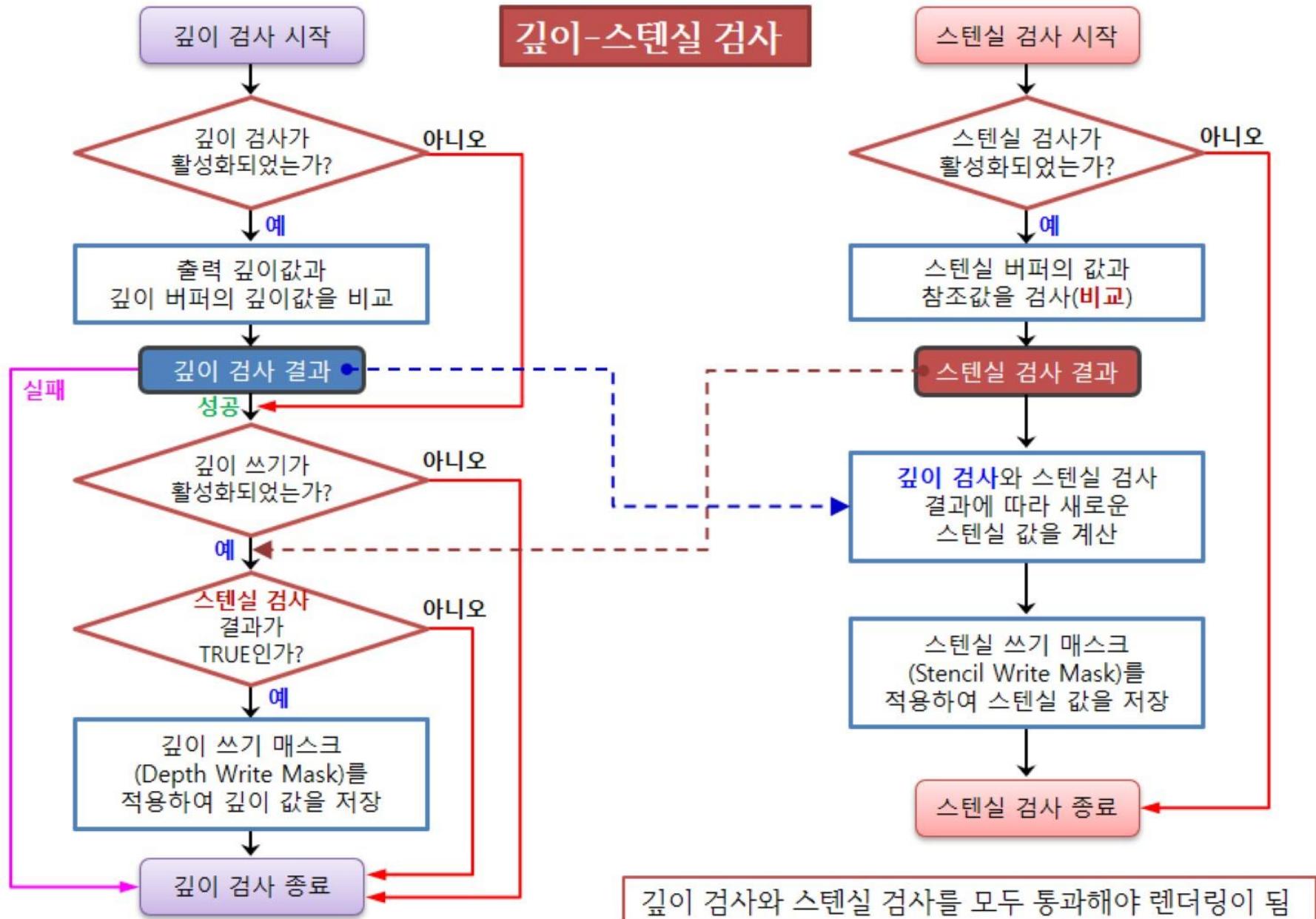
스텐실 검사(Stencil Test)

```
if ((StencilReferenceValue & StencilReadMask) □ (StencilBuffer[i][j] & StencilReadMask))  
{  
    StencilBuffer[i][j] = nStencil;  
    FrameBuffer[i][j] = fColor;  
}
```

□ : 비교 연산자(<, =, ≤, >, ≥, ≠)



깊이-스텐실 검사(Depth-Stencil Test)



깊이-스텐실 상태(Depth-Stencil State)

- 그래픽 파이프라인 상태

```
typedef struct D3D12_GRAPHICS_PIPELINE_STATE_DESC {  
    ID3D12RootSignature *pRootSignature;  
    D3D12_SHADER_BYTECODE VS;  
    D3D12_SHADER_BYTECODE PS;  
    D3D12_SHADER_BYTECODE DS;  
    D3D12_SHADER_BYTECODE HS;  
    D3D12_SHADER_BYTECODE GS;  
    D3D12_STREAM_OUTPUT_DESC StreamOutput;  
    D3D12_BLEND_DESC BlendState;  
    UINT SampleMask;  
    D3D12_RASTERIZER_DESC RasterizerState;  
    D3D12_DEPTH_STENCIL_DESC DepthStencilState;  
    D3D12_INPUT_LAYOUT_DESC InputLayout;  
    D3D12_INDEX_BUFFER_STRIP_CUT_VALUE IBStripCutValue;  
    D3D12_PRIMITIVE_TOPOLOGY_TYPE PrimitiveTopologyType; //기하 쉐이더와 헐 쉐이더  
    UINT NumRenderTargets;  
    DXGI_FORMAT RTVFormats[8];  
    DXGI_FORMAT DSVFormat;  
    DXGI_SAMPLE_DESC SampleDesc;  
    UINT NodeMask;  
    D3D12_CACHED_PIPELINE_STATE CachedPSO;  
    D3D12_PIPELINE_STATE_FLAGS Flags;  
} D3D12_GRAPHICS_PIPELINE_STATE_DESC;
```

```
typedef struct D3D12_DEPTH_STENCIL_DESC {  
    BOOL DepthEnable;  
    D3D12_DEPTH_WRITE_MASK DepthWriteMask;  
    D3D12_COMPARISON_FUNC DepthFunc;  
    BOOL StencilEnable;  
    UINT8 StencilReadMask;  
    UINT8 StencilWriteMask;  
    D3D12_DEPTH_STENCILOP_DESC FrontFace;  
    D3D12_DEPTH_STENCILOP_DESC BackFace;  
} D3D12_DEPTH_STENCIL_DESC;
```

ID3D12Device::CreateGraphicsPipelineState(D3D12_GRAPHICS_PIPELINE_STATE_DESC *pDesc, ...);

깊이-스텐실 상태(Depth-Stencil State)

- 깊이-스텐실 상태(Depth-Stencil State)

```
typedef struct D3D12_DEPTH_STENCIL_DESC {  
    BOOL DepthEnable;  
    D3D12_DEPTH_WRITE_MASK DepthWriteMask;  
    D3D12_COMPARISON_FUNC DepthFunc;  
    BOOL StencilEnable;  
    UINT8 StencilReadMask;  
    UINT8 StencilWriteMask;  
    D3D12_DEPTH_STENCIL_DESC FrontFace;  
    D3D12_DEPTH_STENCIL_DESC BackFace;  
} D3D12_DEPTH_STENCIL_DESC;
```

DepthEnable	깊이 검사를 활성화
DepthWriteMask	깊이 버퍼 쓰기 마스크
StencilEnable	스텐실 검사를 활성화
StencilWriteMask	스텐실 버퍼에 쓰기 위한 마스크

깊이 검사와 스템실 검사가 모두 성공일 때 렌더 타겟 갱신
깊이 검사와 스템실 검사가 모두 성공일 때 깊이 버퍼 갱신
스텐실 검사가 활성화되면 스템실 버퍼 갱신

깊이 검사		스텐실 검사		깊이 버퍼 갱신	스텐실 버퍼 갱신
활성화	검사 결과	활성화	검사 결과		
x	true	x	true	Write Mask	x
x	true	o	true	Write Mask	Write Mask
x	true	o	false	x	Write Mask
o	false	x	true	x	x
o	false	o	true	x	Write Mask
o	false	o	false	x	Write Mask
o	true	x	true	Write Mask	x
o	true	o	true	Write Mask	Write Mask
o	true	o	false	x	Write Mask

깊이-스텐실 상태(Depth-Stencil State)

- 깊이-스텐실 상태(Depth-Stencil State)

```
typedef struct D3D12_DEPTH_STENCIL_DESC {  
    BOOL DepthEnable; //깊이 검사를 활성화  
    D3D12_DEPTH_WRITE_MASK DepthWriteMask; //깊이 버퍼 쓰기 매스크(깊이 검사가 성공할 때 적용)  
    D3D12_COMPARISON_FUNC DepthFunc; //깊이 값을 비교하는 함수  
    BOOL StencilEnable; //스텐실 검사를 활성화  
    UINT8 StencilReadMask; //스텐실 버퍼 읽기 매스크  
    UINT8 StencilWriteMask; //스텐실 버퍼 쓰기 매스크  
    D3D12_DEPTH_STENCILOP_DESC FrontFace; //전면 다각형에 대하여 스템실 버퍼를 갱신하는 방법  
    D3D12_DEPTH_STENCILOP_DESC BackFace; //은면 다각형에 대하여 스템실 버퍼를 갱신하는 방법  
} D3D12_DEPTH_STENCIL_DESC;
```

```
typedef enum D3D12_DEPTH_WRITE_MASK {  
    D3D12_DEPTH_WRITE_MASK_ZERO, //깊이 버퍼에 쓰지 않음  
    D3D12_DEPTH_WRITE_MASK_ALL //깊이 버퍼에 쓰기를 허용  
} D3D12_DEPTH_WRITE_MASK;
```

```
typedef enum D3D12_COMPARISON_FUNC {  
    D3D12_COMPARISON_FUNC_NEVER, //항상 비교가 실패함  
    D3D12_COMPARISON_FUNC_LESS, //소스(새로운) 데이터가 목표(기존) 데이터보다 작으면 비교에 성공  
    D3D12_COMPARISON_FUNC_EQUAL, //소스 데이터가 목표 데이터와 같으면 비교에 성공  
    D3D12_COMPARISON_FUNC_LESS_EQUAL, //소스 데이터가 목표 데이터보다 작거나 같으면 비교에 성공  
    D3D12_COMPARISON_FUNC_GREATER, //소스 데이터가 목표 데이터보다 크면 비교에 성공  
    D3D12_COMPARISON_FUNC_NOT_EQUAL, //소스 데이터가 목표 데이터와 같지 않으면 비교에 성공  
    D3D12_COMPARISON_FUNC_GREATER_EQUAL, //소스 데이터가 목표 데이터보다 크거나 같으면 비교에 성공  
    D3D12_COMPARISON_FUNC_ALWAYS //항상 비교가 성공함  
} D3D12_COMPARISON_FUNC;
```

깊이-스텐실 상태(Depth-Stencil State)

- 깊이-스텐실 상태(Depth-Stencil State)

```
typedef struct D3D12_DEPTH_STENCIL_DESC {  
    ...  
    BOOL StencilEnable; //스텐실 검사를 활성화  
    UINT8 StencilReadMask; //스텐실 버퍼 읽기 마스크(스텐실 버퍼의 값과 & 연산)  
    UINT8 StencilWriteMask; //스텐실 버퍼 쓰기 마스크(스텐실 버퍼 |= 스템실 값 & 마스크)  
    D3D12_DEPTH_STENCIL_DESC FrontFace; //전면 다각형에 대하여 스템실 버퍼를 갱신하는 방법  
    D3D12_DEPTH_STENCIL_DESC BackFace; //은면 다각형을 렌더링하지 않으면 설정할 필요없음  
} D3D12_DEPTH_STENCIL_DESC;
```

```
typedef struct D3D12_DEPTH_STENCILOP_DESC {  
    D3D12_STENCIL_OP StencilFailOp; //스텐실 검사가 실패할 때(깊이 검사의 결과에 상관없이) 실행할 연산  
    D3D12_STENCIL_OP StencilDepthFailOp; //스텐실 검사가 성공하고 깊이 검사가 실패일 때 실행할 연산  
    D3D12_STENCIL_OP StencilPassOp; //스텐실 검사와 깊이 검사가 모두 성공할 때 실행할 연산  
    D3D12_COMPARISON_FUNC StencilFunc; //스텐실 값을 비교하는 함수  
} D3D12_DEPTH_STENCILOP_DESC;
```

```
typedef enum D3D12_STENCIL_OP {  
    D3D12_STENCIL_OP_KEEP, //기존 스템실 버퍼의 값을 유지  
    D3D12_STENCIL_OP_ZERO, //스텐실 버퍼의 값을 0으로 설정  
    D3D12_STENCIL_OP_REPLACE, //스텐실 버퍼의 값을 참조 값으로 설정(OMSetStencilRef())  
    D3D12_STENCIL_OP_INCR_SAT, //스텐실 버퍼의 값을 1 증가, 클램핑(8-비트: 0~255)  
    D3D12_STENCIL_OP_DECR_SAT, //스텐실 버퍼의 값을 1 감소, 클램핑(8-비트: 0~255)  
    D3D12_STENCIL_OP_INVERT, //스텐실 버퍼의 값(비트)을 인버트(Invert)  
    D3D12_STENCIL_OP_INCR, //스텐실 버퍼의 값을 1 증가, 랩(Wrap: 256→0)  
    D3D12_STENCIL_OP_DECR //스텐실 버퍼의 값을 1 감소, 랩(Wrap: 0→255)  
} D3D12_STENCIL_OP;
```

깊이-스텐실 상태(Depth-Stencil State)

- 깊이-스텐실 상태(Depth-Stencil State) 상태

- 스텐실 참조 값 설정

```
void ID3D12GraphicsCommandList::OMSetStencilRef(  
    UINT StencilRef //스텐실 검사에서 사용할 참조 값  
)
```

픽셀 쉐이더에서 SV_StencilRef를 사용하여 출력 가능

기본 깊이-스텐실 상태

```
typedef struct D3D12_DEPTH_STENCIL_DESC {  
    BOOL DepthEnable;  
    D3D12_DEPTH_WRITE_MASK DepthWriteMask;  
    D3D12_COMPARISON_FUNC DepthFunc;  
    BOOL StencilEnable;  
    UINT8 StencilReadMask;  
    UINT8 StencilWriteMask;  
    D3D12_DEPTH_STENCILOP_DESC FrontFace;  
    D3D12_DEPTH_STENCILOP_DESC BackFace;  
} D3D12_DEPTH_STENCIL_DESC;
```

DepthEnable	TRUE
DepthWriteMask	D3D12_DEPTH_WRITE_MASK_ALL
DepthFunc	D3D12_COMPARISON_LESS
StencilEnable	FALSE
StencilReadMask	D3D12_DEFAULT_STENCIL_READ_MASK(0xff)
StencilWriteMask	D3D12_DEFAULT_STENCIL_WRITE_MASK(0xff)
FrontFace.StencilFunc	D3D12_COMPARISON_ALWAYS
BackFace.StencilFunc	
FrontFace.StencilDepthFailOp	D3D12_STENCIL_OP_KEEP
BackFace.StencilDepthFailOp	
FrontFace.StencilPassOp	D3D12_STENCIL_OP_KEEP
BackFace.StencilPassOp	
FrontFace.StencilFailOp	D3D12_STENCIL_OP_KEEP
BackFace.StencilFailOp	

깊이-스텐실 뷰(Depth-Stencil View)

- 깊이-스텐실 뷰(Depth-Stencil View) 생성

```
void ID3D12Device::CreateDepthStencilView(
```

```
    ID3D12Resource *pResource, //깊이-스텐실 버퍼 리소스
```

```
    D3D12_DEPTH_STENCIL_VIEW_DESC *pDesc, //NULL: 리소스의 형식과 차원을 사용
```

```
    D3D12_CPU_DESCRIPTOR_HANDLE DestDescriptor //생성된 뷰(서술자)의 핸들(주소)
```

```
);
```

```
typedef enum D3D12_DSV_DIMENSION {  
    D3D12_DSV_DIMENSION_UNKNOWN,  
    D3D12_DSV_DIMENSION_TEXTURE1D,  
    D3D12_DSV_DIMENSION_TEXTURE1DARRAY,  
    D3D12_DSV_DIMENSION_TEXTURE2D,  
    D3D12_DSV_DIMENSION_TEXTURE2DARRAY,  
    D3D12_DSV_DIMENSION_TEXTURE2DMS,  
    D3D12_DSV_DIMENSION_TEXTURE2DMSARRAY  
} D3D12_DSV_DIMENSION;
```

```
typedef struct D3D12_TEX2D_DSV {  
    UINT MipSlice; // mipSlice 레벨 인덱스  
} D3D12_TEX2D_DSV;
```

```
DXGI_FORMAT_D16_UNORM  
DXGI_FORMAT_D24_UNORM_S8_UINT  
DXGI_FORMAT_D32_FLOAT  
DXGI_FORMAT_D32_FLOAT_S8X24_UINT  
DXGI_FORMAT_UNKNOWN
```

```
typedef struct D3D12_DEPTH_STENCIL_VIEW_DESC {  
    DXGI_FORMAT Format; //리소스 뷰의 형식  
    D3D12_DSV_DIMENSION ViewDimension; //차원  
    D3D12_DSV_FLAGS Flags;  
    union {  
        D3D12_TEX1D_DSV Texture1D;  
        D3D12_TEX1D_ARRAY_DSV Texture1DArray;  
        D3D12_TEX2D_DSV Texture2D;  
        D3D12_TEX2D_ARRAY_DSV Texture2DArray;  
        D3D12_TEX2DMS_DSV Texture2DMS;  
        D3D12_TEX2DMS_ARRAY_DSV Texture2DMSArray;  
    };  
} D3D12_DEPTH_STENCIL_VIEW_DESC;
```

```
typedef enum D3D12_DSV_FLAGS {  
    D3D12_DSV_FLAG_NONE,  
    D3D12_DSV_FLAG_READ_ONLY_DEPTH,  
    D3D12_DSV_FLAG_READ_ONLY_STENCIL  
} D3D12_DSV_FLAGS;
```

깊이-스텐실 뷰를 생성하려면 깊이-스텐실 버퍼와 깊이-스텐실 서술자 힙이 필요

깊이-스텐실 뷰(Depth-Stencil View)

- **서술자 힙(ID3D12DescriptorHeap)**

- 리소스를 서술하는 서술자들을 저장하는 연속적인 메모리 영역(배열)

```
HRESULT ID3D12Device::CreateDescriptorHeap(  
    D3D12_DESCRIPTOR_HEAP_DESC *pDescriptorHeapDesc,  
    REFIID iid, //__uuidof(ID3D12DescriptorHeap)  
    void **ppvHeap //ID3D12DescriptorHeap  
)
```

```
typedef enum D3D12_DESCRIPTOR_HEAP_TYPE {  
    D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV,  
    D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER,  
    D3D12_DESCRIPTOR_HEAP_TYPE_RTV,  
    D3D12_DESCRIPTOR_HEAP_TYPE_DSV  
} D3D12_DESCRIPTOR_HEAP_TYPE;
```

```
typedef struct D3D12_DESCRIPTOR_HEAP_DESC {  
    D3D12_DESCRIPTOR_HEAP_TYPE Type;  
    UINT NumDescriptors; //서술자의 개수  
    D3D12_DESCRIPTOR_HEAP_FLAGS Flags;  
    UINT NodeMask; //단일 GPU: 0  
} D3D12_DESCRIPTOR_HEAP_DESC;
```

```
typedef enum D3D12_DESCRIPTOR_HEAP_FLAGS {  
    D3D12_DESCRIPTOR_HEAP_FLAG_NONE,  
    D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE  
} D3D12_DESCRIPTOR_HEAP_FLAGS;
```

```
D3D12_DESCRIPTOR_HEAP_DESC ID3D12DescriptorHeap::GetDesc();
```

- 디바이스(어댑터)마다 서술자 유형별 메모리 크기가 다름(32~64 바이트)
서술자 힙을 사용하려면 서술자 배열의 원소의 크기를 알아야 함

```
UINT ID3D12Device::GetDescriptorHandleIncrementSize(  
    [in] D3D12_DESCRIPTOR_HEAP_TYPE DescriptorHeapType  
)
```

깊이-스텐실 뷰(Depth-Stencil View)

- 렌더 타겟 서술자 힙과 깊이-스텐실 서술자 힙 생성

```
void CGameFramework::CreateRtvAndDsvDescriptorHeaps()
{
    D3D12_DESCRIPTOR_HEAP_TYPE d3dDescriptorHeapType = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
    D3D12_DESCRIPTOR_HEAP_DESC d3dDescriptorHeapDesc;
    ::ZeroMemory(&d3dDescriptorHeapDesc, sizeof(D3D12_DESCRIPTOR_HEAP_DESC));
    d3dDescriptorHeapDesc.NumDescriptors = m_nSwapChainBuffers; //뷰(서술자)의 개수
    d3dDescriptorHeapDesc.Type = d3dDescriptorHeapType; //렌더 타겟 뷰
    d3dDescriptorHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
    d3dDescriptorHeapDesc.NodeMask = 0; //단일 CPU
    m_pd3dDevice->CreateDescriptorHeap(&d3dDescriptorHeapDesc, __uuidof(ID3D12DescriptorHeap), (void**)(&m_pd3dRtvDescriptorHeap));

    m_nRtvDescriptorSize = m_pd3dDevice->GetDescriptorHandleIncrementSize(d3dDescriptorHeapType);

    d3dDescriptorHeapType = D3D12_DESCRIPTOR_HEAP_TYPE_DSV; //깊이-스텐실 뷰
    d3dDescriptorHeapDesc.NumDescriptors = 1;
    d3dDescriptorHeapDesc.Type = d3dDescriptorHeapType;
    m_pd3dDevice->CreateDescriptorHeap(&d3dDescriptorHeapDesc, __uuidof(ID3D12DescriptorHeap), (void**)(&m_pd3dDsvDescriptorHeap));

    m_nDsvDescriptorSize = m_pd3dDevice->GetDescriptorHandleIncrementSize(d3dDescriptorHeapType);
}

ID3D12DescriptorHeap *m_pd3dRtvDescriptorHeap; //렌더 타겟 뷰(서술자) 힙
UINT m_nRtvDescriptorSize; //렌더 타겟 뷰(서술자) 힙 원소 하나의 크기
ID3D12DescriptorHeap *m_pd3dDsvDescriptorHeap; //깊이-스텐실 뷰(서술자) 힙
UINT m_nDsvDescriptorSize; //깊이-스텐실 뷰(서술자) 힙 원소 하나의 크기
```

깊이-스텐실 뷰(Depth-Stencil View)

- 깊이-스텐실 버퍼의 생성

```
HRESULT ID3D12Device::CreateCommittedResource(
```

```
    D3D12_HEAP_PROPERTIES *pHeapProperties,
```

```
    D3D12_HEAP_FLAGS HeapFlags,
```

```
    D3D12_RESOURCE_DESC *pResourceDesc,
```

```
    D3D12_RESOURCE_STATES InitialResourceState, //리소스가 사용되는 방법에 따른 리소스의 초기 상태
```

```
    D3D12_CLEAR_VALUE *pOptimizedClearValue, //NULL: D3D12_RESOURCE_DIMENSION_BUFFER
```

```
    REFIID riidResource, //__uuidof(ID3D12Resource)
```

```
    void **ppvResource
```

```
);
```

```
typedef enum D3D12_HE
```

```
    D3D12_HEAP_FLAG_NOC
```

```
    D3D12_HEAP_FLAG_SH
```

```
    D3D12_HEAP_FLAG_DE
```

```
    D3D12_HEAP_FLAG_AL
```

```
    D3D12_HEAP_FLAG_SH
```

```
    D3D12_HEAP_FLAG_DE
```

```
    D3D12_HEAP_FLAG_DE
```

```
    D3D12_HEAP_FLAG_AL
```

```
    typedef enum D3D12
```

```
        D3D12_RESOURCE
```

```
    typedef struct D3D12_HEAP_PROPERTIES {
```

```
        D3D12_HEAP_TYPE Type;
```

```
        D3D12_CPU_PAGE_PROPERTY CPUPageProperty;
```

```
        D3D12_MEMORY_POOL MemoryPoolPreference;
```

```
        UINT CreationNodeMask;
```

```
        UINT VisibleNodeMask;
```

```
    } D3D12_HEAP_PROPERTIES;
```

```
    typedef struct D3D12_CLEAR_VALUE {
```

```
        DXGI_FORMAT Format;
```

```
        union {
```

```
            FLOAT Color[4];
```

```
            D3D12_DEPTH_STENCIL_VALUE DepthStencil;
```

```
        };
```

```
    } D3D12_CLEAR_VALUE;
```

```
    D3D12_RESOURCE_STATE_PRESENT,
```

```
    D3D12_RESOURCE_STATE_PREDICATION
```

```
    D3D12_RESOURCE_DESC;
```

```
} D3D12_RESOURCE_DESC;
```

깊이-스텐실 뷰(Depth-Stencil View)

- 깊이-스텐실 버퍼의 생성

- ID3D12Device::CreateCommittedResource()

```
typedef struct D3D12_RESOURCE_DESC {  
    D3D12_RESOURCE_DIMENSION Dimension;  
    UINT64 Alignment; // 0(64KB), 4KB, 64KB, 4MB  
    UINT64 Width; // 가로 크기  
    UINT Height; // 세로 크기  
    UINT16 DepthOrArraySize; // 깊이 또는 배열의 크기  
    UINT16 MipLevels; // mip맵 레벨, 0(자동 계산)  
    DXGI_FORMAT Format; // 리소스 형식  
    DXGI_SAMPLE_DESC SampleDesc; // 다중 샘플링  
    D3D12_TEXTURE_LAYOUT Layout; // 다차원 리소스를 1차원 리소스로 매핑하기 위한 방법  
    D3D12_RESOURCE_FLAGS Flags;  
} D3D12_RESOURCE_DESC;
```

```
DXGI_FORMAT_D16_UNORM  
DXGI_FORMAT_D24_UNORM_S8_UINT  
DXGI_FORMAT_D32_FLOAT  
DXGI_FORMAT_D32_FLOAT_S8X24_UINT  
DXGI_FORMAT_UNKNOWN
```

```
typedef enum D3D12_TEXTURE_LAYOUT {  
    D3D12_TEXTURE_LAYOUT_UNKNOWN,  
    D3D12_TEXTURE_LAYOUT_ROW_MAJOR, // 행 우선 순서  
    D3D12_TEXTURE_LAYOUT_64KB_UNDEFINED_SWIZZLE,  
    D3D12_TEXTURE_LAYOUT_64KB_STANDARD_SWIZZLE  
} D3D12_TEXTURE_LAYOUT;
```

```
typedef enum D3D12_RESOURCE_DIMENSION {  
    D3D12_RESOURCE_DIMENSION_UNKNOWN,  
    D3D12_RESOURCE_DIMENSION_BUFFER,  
    D3D12_RESOURCE_DIMENSION_TEXTURE1D,  
    D3D12_RESOURCE_DIMENSION_TEXTURE2D,  
    D3D12_RESOURCE_DIMENSION_TEXTURE3D  
} D3D12_RESOURCE_DIMENSION;
```

```
typedef enum D3D12_RESOURCE_FLAGS {  
    D3D12_RESOURCE_FLAG_NONE,  
    D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET,  
    D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL,  
    D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS,  
    D3D12_RESOURCE_FLAG_DENY_SHADER_RESOURCE,  
    D3D12_RESOURCE_FLAG_ALLOW_CROSS_ADAPTER,  
    D3D12_RESOURCE_FLAG_ALLOW_SIMULTANEOUS_ACCESS  
} D3D12_RESOURCE_FLAGS;
```

깊이-스텐실 뷰(Depth-Stencil View)

- 깊이-스텐실 버퍼와 뷰(서술자) 생성

```
CGameFramework::CreateDepthStencilView() {  
    D3D12_RESOURCE_DESC d3dResourceDesc = { };  
    d3dResourceDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;  
    d3dResourceDesc.Width = m_nWndClientWidth; d3dResourceDesc.Height = m_nWndClientHeight;  
    d3dResourceDesc.DepthOrArraySize = d3dResourceDesc.MipLevels = 1;  
    d3dResourceDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;  
    d3dResourceDesc.SampleDesc.Count = (m_bMsaa4xEnable) ? 4 : 1;  
    d3dResourceDesc.SampleDesc.Quality = (m_bMsaa4xEnable) ? (m_nMsaa4xQualityLevels - 1) : 0;  
    d3dResourceDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;  
    d3dResourceDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;  
    D3D12_HEAP_PROPERTIES d3dHeapProperties;  
    d3dHeapProperties.Type = D3D12_HEAP_TYPE_DEFAULT;  
    d3dHeapProperties.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_UNKNOWN;  
    d3dHeapProperties.MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;  
    d3dHeapProperties.CreationNodeMask = d3dHeapProperties.VisibleNodeMask = 1;  
    D3D12_CLEAR_VALUE d3dClearValue;  
    d3dClearValue.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;  
    d3dClearValue.DepthStencil.Depth = 1.0f;  
    d3dClearValue.DepthStencil.Stencil = 0;  
    m_pd3dDevice->CreateCommittedResource(&d3dHeapProperties, D3D12_HEAP_FLAG_NONE,  
    &d3dResourceDesc, D3D12_RESOURCE_STATE_DEPTH_WRITE, &d3dClearValue, __uuidof(ID3D12Resource),  
    (void **)&m_pd3dDepthStencilBuffer);  
    D3D12_CPU_DESCRIPTOR_HANDLE d3dDsvCPUDescriptorHandle = m_pd3dDsvDescriptorHeap-  
    >GetCPUDescriptorHandleForHeapStart();  
    m_pd3dDevice->CreateDepthStencilView(m_pd3dDepthStencilBuffer, NULL, d3dDsvCPUDescriptorHandle);  
}
```

깊이-스텐실 뷰(Depth-Stencil View)

- 렌더 타겟 버퍼와 깊이-스텐실 버퍼 초기화

- 렌더 타겟 버퍼 초기화

D3D12_RESOURCE_STATES::D3D12_RESOURCE_STATE_RENDER_TARGET

```
void ID3D12GraphicsCommandList::ClearRenderTargetView(  
    D3D12_CPU_DESCRIPTOR_HANDLE RenderTargetView, //렌더 타겟 서출자  
    FLOAT ColorRGBA[4], //초기화할 색상(4-요소 배열), Colors::Blue  
    UINT NumRects, //사각형의 개수  
    D3D12_RECT *pRects //사각형 배열(NULL: 렌더 타겟 전체 영역)  
);
```

typedef RECT D3D12_RECT;

```
typedef struct _RECT {  
    LONG left;  
    LONG top;  
    LONG right;  
    LONG bottom;  
} RECT;
```

```
float pfClearColor[4] = { 0.0f, 0.125f, 0.3f, 1.0f }; //Colors::Azure  
m_pd3dCommandList->ClearRenderTargetView(d3dRtvCPUDescriptorHandle, pfClearColor, 0, NULL);
```

- 깊이-스텐실 버퍼 초기화

D3D12_RESOURCE_STATES::D3D12_RESOURCE_STATE_DEPTH_WRITE

```
void ID3D12GraphicsCommandList::ClearDepthStencilView(  
    D3D12_CPU_DESCRIPTOR_HANDLE DepthStencilView, //깊이-스텐실 서출자  
    D3D12_CLEAR_FLAGS ClearFlags, //초기화할 데이터의 유형  
    FLOAT Depth, //초기화할 깊이 값(0~1.0)  
    UINT8 Stencil, //초기화할 스텐실 값(0~255)  
    UINT NumRects, //사각형의 개수  
    D3D12_RECT *pRects //사각형 배열(NULL: 전체 영역)  
);
```

```
typedef enum D3D12_CLEAR_FLAGS {  
    D3D12_CLEAR_FLAG_DEPTH,  
    D3D12_CLEAR_FLAG_STENCIL  
} D3D12_CLEAR_FLAGS;
```

```
m_pd3dCommandList->ClearDepthStencilView(d3dDsvCPUDescriptorHandle, D3D12_CLEAR_FLAG_DEPTH |  
D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, NULL);
```

깊이-스텐실 뷰(Depth-Stencil View)

- 렌더 타겟과 깊이-스텐실 버퍼를 파이프라인에 연결

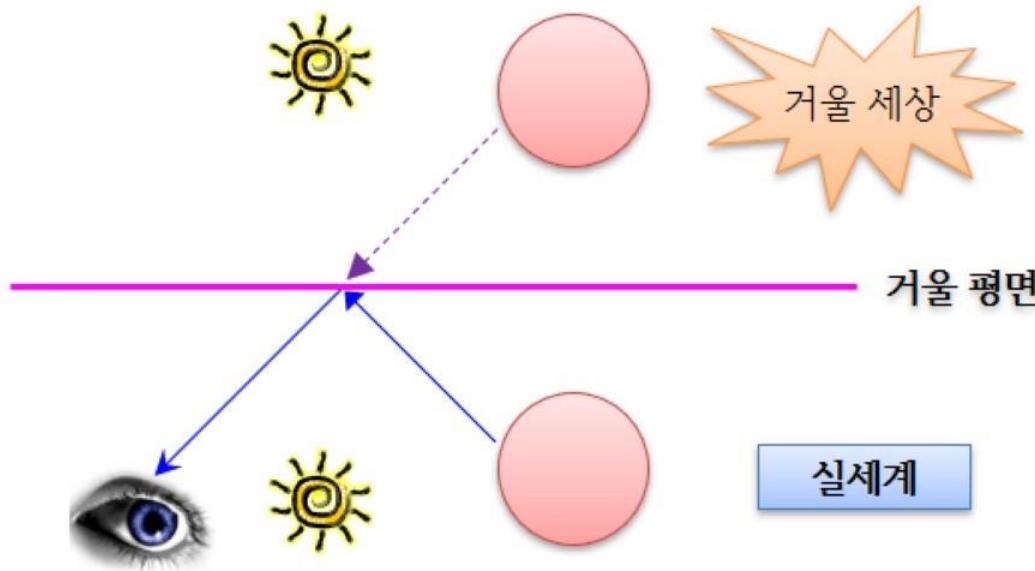
- 렌더 타겟과 깊이-스텐실 버퍼를 위한 CPU 서술자를 설정

```
void ID3D12GraphicsCommandList::OMSetRenderTargets(  
    UINT NumRenderTargetDescriptors, //렌더 타겟 서술자의 개수  
    D3D12_CPU_DESCRIPTOR_HANDLE *pRenderTargetDescriptors, //렌더 타겟 서술자(핸들)들의 배열  
    BOOL RTsSingleHandleToDescriptorRange, //true: 렌더 타겟 서술자들이 연속적인 범위(첫번째 서술자만 필요)  
    D3D12_CPU_DESCRIPTOR_HANDLE *pDepthStencilDescriptor //깊이-스텐실 서술자(핸들)  
);
```

```
D3D12_RESOURCE_BARRIER d3dResourceBarrier;  
d3dResourceBarrier.Type = D3D12_RESOURCE_BARRIER_TYPE_TRANSITION;  
d3dResourceBarrier.Flags = D3D12_RESOURCE_BARRIER_FLAG_NONE;  
d3dResourceBarrier.Transition.pResource = m_ppd3dRenderTargetBuffers[m_nSwapChainBufferIndex];  
d3dResourceBarrier.Transition.StateBefore = D3D12_RESOURCE_STATE_PRESENT;  
d3dResourceBarrier.Transition.StateAfter = D3D12_RESOURCE_STATE_RENDER_TARGET;  
d3dResourceBarrier.Transition.Subresource = D3D12_RESOURCE_BARRIER_ALL_SUBRESOURCES;  
m_pd3dCommandList->ResourceBarrier(1, &d3dResourceBarrier);  
D3D12_CPU_DESCRIPTOR_HANDLE d3dRtvCPUDescriptorHandle = m_pd3dRtvDescriptorHeap-  
>GetCPUDescriptorHandleForHeapStart();  
d3dRtvCPUDescriptorHandle.ptr += (m_nSwapChainBufferIndex * m_nRtvDescriptorSize);  
m_pd3dCommandList->ClearRenderTargetView(d3dRtvCPUDescriptorHandle, Colors::Azure, 0, NULL);  
D3D12_CPU_DESCRIPTOR_HANDLE d3dDsvCPUDescriptorHandle = m_pd3dDsvDescriptorHeap-  
>GetCPUDescriptorHandleForHeapStart();  
m_pd3dCommandList->ClearDepthStencilView(d3dDsvCPUDescriptorHandle, D3D12_CLEAR_FLAG_DEPTH |  
D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, NULL);  
m_pd3dCommandList->OMSetRenderTargets(1, &d3dRtvCPUDescriptorHandle, TRUE,  
&d3dDsvCPUDescriptorHandle);
```

스텐실(Stencil) 응용

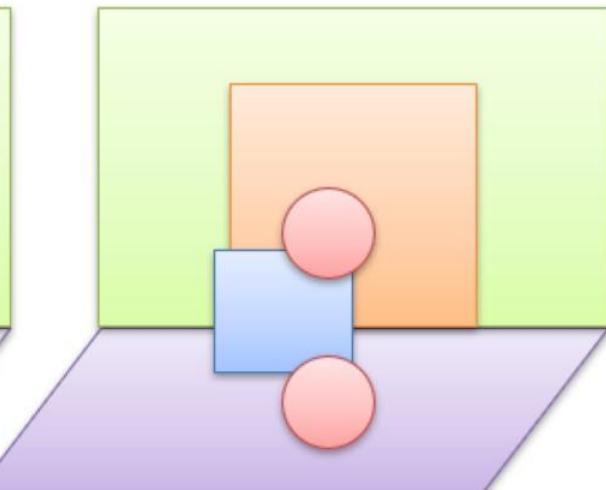
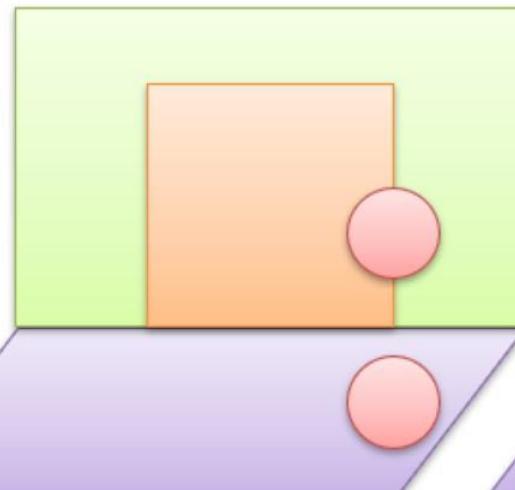
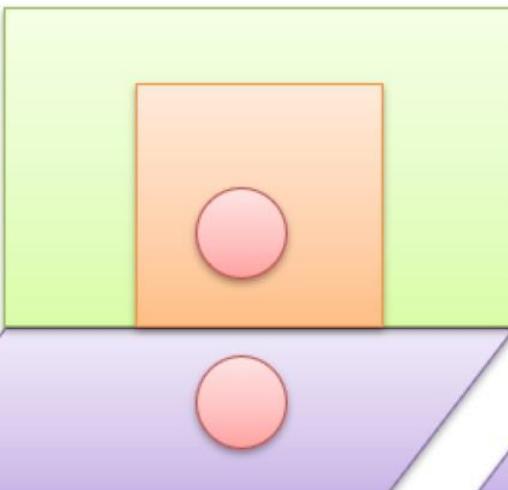
- 평면 거울(Planar Mirror) 구현



- ① 거울 평면에 반사된 객체를 구함
- ② 조명이 있다면 반사된 조명을 구함
- ③ 반사된 객체를 그림

실세계

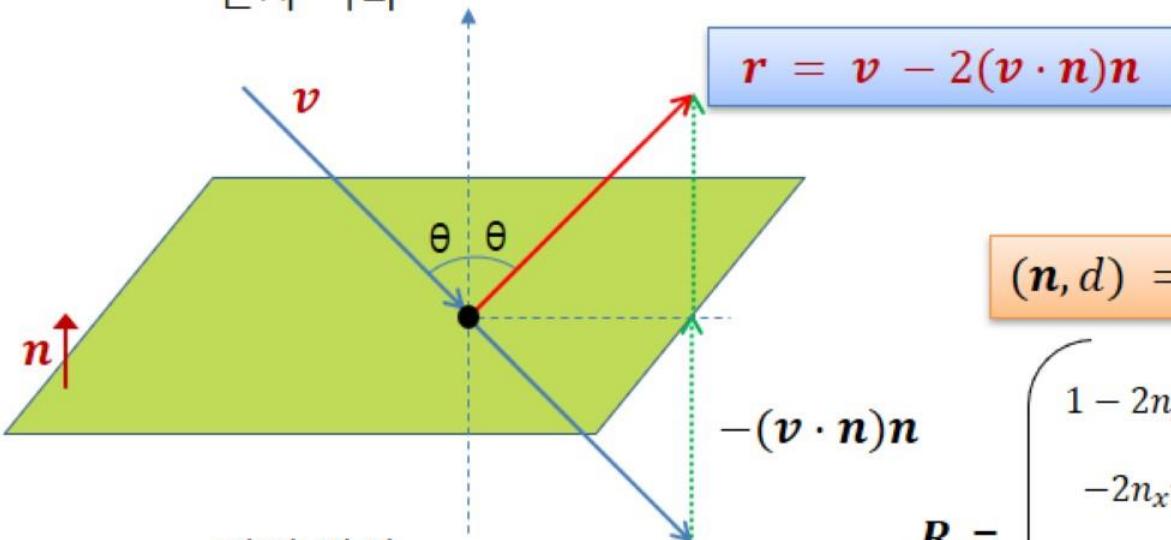
- ① 거울 평면에 반사된 객체를 구함
- ② 조명이 있다면 반사된 조명을 구함
- ③ 거울에만 그려지도록 스텐실을 설정
- ④ 반사된 객체를 그림



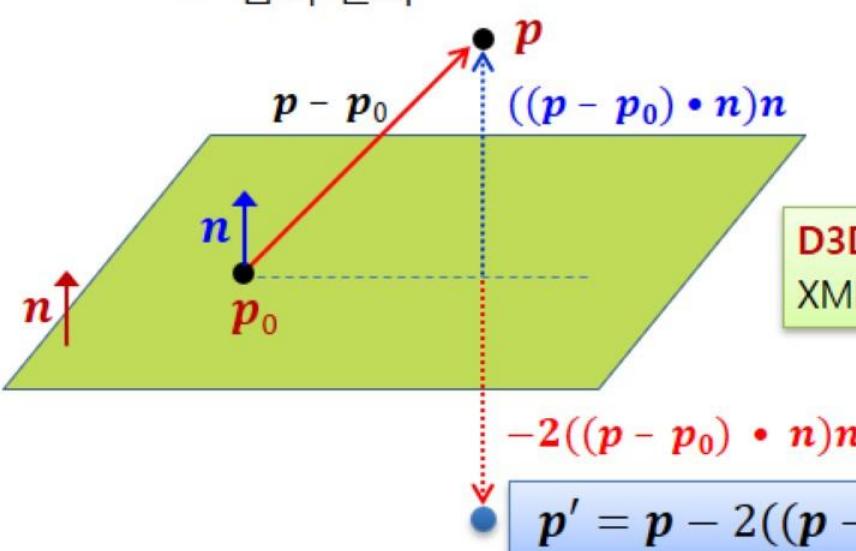
스텐실(Stencil) 응용

- 평면 거울(Planar Mirror) 구현

- 반사 벡터



- 점의 반사



$$(n, d) = (n_x, n_y, n_z, d), d = -n \cdot p_0$$

$$R = \begin{pmatrix} 1 - 2n_x n_x & -2n_x n_y & -2n_x n_z & 0 \\ -2n_x n_y & 1 - 2n_y n_y & -2n_y n_z & 0 \\ -2n_x n_z & -2n_y n_z & 1 - 2n_z n_z & 0 \\ -2dn_x & -2dn_y & -2dn_z & 1 \end{pmatrix}$$

D3DXMatrixReflect(D3DXMATRIX *pOut, D3DXPLANE *pPlane);
XMMATRIX XMMatrixReflect(XMVECTOR ReflectionPlane);

스텐실(Stencil) 응용

- 반사 행렬(Reflection Matrix)

$$p * R = p - 2((p - p_0) \bullet n)n$$

$$\left(\begin{array}{c|c|c|c|c} p_x & p_y & p_z & 1 \end{array} \right) \left(\begin{array}{cccc} 1 - 2n_x n_x & -2n_x n_y & -2n_x n_z & 0 \\ -2n_x n_y & 1 - 2n_y n_y & -2n_y n_z & 0 \\ -2n_x n_z & -2n_y n_z & 1 - 2n_z n_z & 0 \\ -2dn_x & -2dn_y & -2dn_z & 1 \end{array} \right) \equiv \left(\begin{array}{c|c|c|c|c} p'_x & p'_y & p'_z & 1 \end{array} \right)$$

$$p'_x = p_x - 2p_x n_x n_x - 2p_y n_x n_y - 2p_z n_x n_z - 2dn_x = p_x - 2n_x(p_x n_x + p_y n_y + p_z n_z + d)$$

$$= p_x - 2n_x(n \bullet p + d)$$

$$p'_y = -2p_x n_x n_y + p_y - 2p_y n_y n_y - 2p_z n_y n_z - 2dn_y = p_y - 2n_y(p_x n_x + p_y n_y + p_z n_z + d)$$

$$= p_y - 2n_y(n \bullet p + d)$$

$$p'_z = -2p_x n_x n_z - 2p_y n_y n_z + p_z - 2p_z n_z n_z - 2dn_z = p_z - 2n_z(p_x n_x + p_y n_y + p_z n_z + d)$$

$$= p_z - 2n_z(n \bullet p + d)$$

$$p' = p * R = p - 2(n \bullet p + d)n$$

$$= p - 2(n \bullet p - n \bullet p_0)n = p - 2(n \bullet (p - p_0))n$$

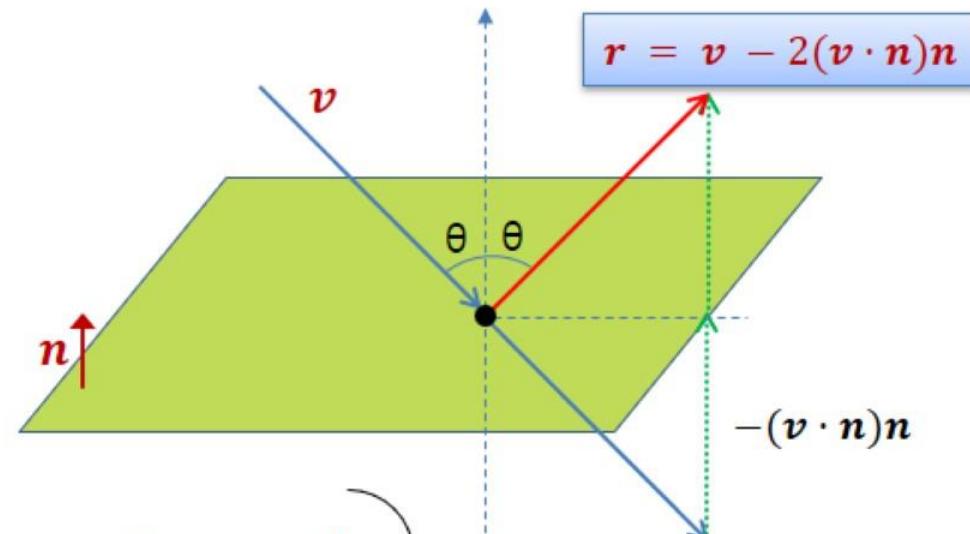
$$= p - 2((p - p_0) \bullet n)n$$

$$d = -n \bullet p_0$$

스텐실(Stencil) 응용

- 반사 행렬(Reflection Matrix)

$$v * R = v - 2(v \cdot n)n$$



$$\left(\begin{array}{c|c|c|c} v_x & v_y & v_z & \mathbf{0} \end{array} \right) \left(\begin{array}{cccc} 1 - 2n_x n_x & -2n_x n_y & -2n_x n_z & 0 \\ -2n_x n_y & 1 - 2n_y n_y & -2n_y n_z & 0 \\ -2n_x n_z & -2n_y n_z & 1 - 2n_z n_z & 0 \\ -2dn_x & -2dn_y & -2dn_z & 1 \end{array} \right) \equiv \left(\begin{array}{c|c|c|c} v'_x & v'_y & v'_z & \mathbf{0} \end{array} \right)$$

$$v'_x = v_x - 2v_x n_x n_x - 2v_y n_x n_y - 2v_z n_x n_z = v_x - 2n_x(v_x n_x + v_y n_y + v_z n_z) = v_x - 2n_x(\mathbf{n} \cdot \mathbf{v})$$

$$v'_y = -2v_x n_x n_y + v_y - 2v_y n_y n_y - 2v_z n_y n_z = v_y - 2n_y(v_x n_x + v_y n_y + v_z n_z) = v_y - 2n_y(\mathbf{n} \cdot \mathbf{v})$$

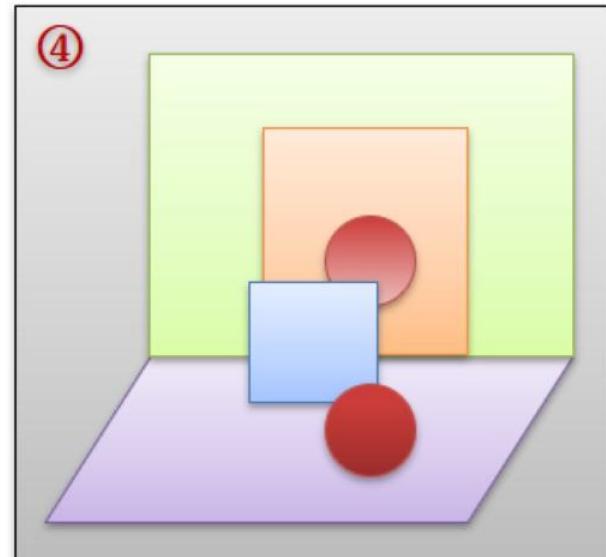
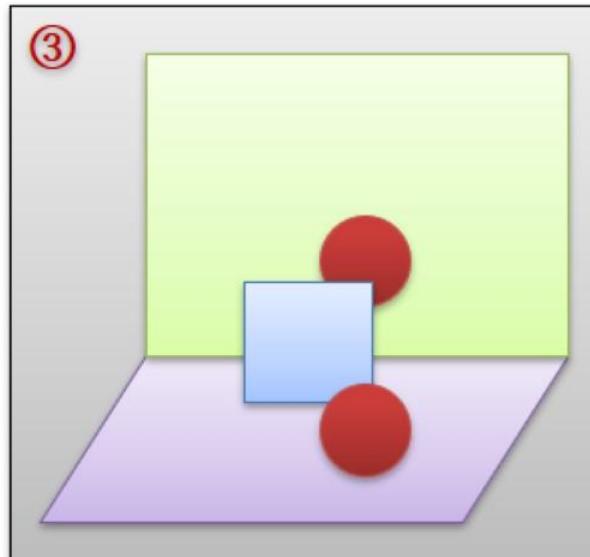
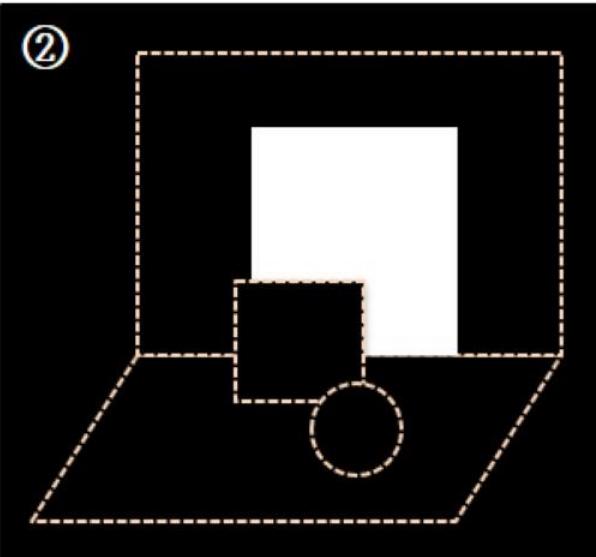
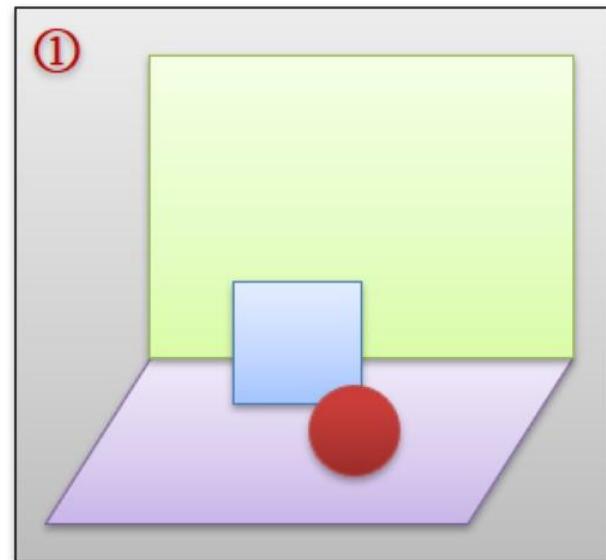
$$v'_z = -2v_x n_x n_z - 2v_y n_y n_z + v_z - 2v_z n_z n_z = v_z - 2n_z(v_x n_x + v_y n_y + v_z n_z) = v_z - 2n_z(\mathbf{n} \cdot \mathbf{v})$$

$$\mathbf{v}' = \mathbf{v} * \mathbf{R} = \mathbf{v} - 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n}$$

스텐실(Stencil) 응용

• 평면 거울(Planar Mirror) 구현

- ① 거울을 제외한 실세계 객체들을 렌더 타겟에 렌더링
깊이 검사를 활성화하고 깊이 버퍼를 변경하도록 설정
스텐실 검사를 수행하지 않음
- ② 스텐실 버퍼 초기화(0)하고 거울을 스텐실 버퍼에 렌더링
깊이 검사와 스텐실 검사(항상 성공)를 수행
깊이 버퍼와 렌더 타겟은 변경하지 않고 스텐실 버퍼만 변경
(거울에 해당하는 스텐실 버퍼 영역의 스텐실 값을 1로 설정)
- ③ 반사된 객체를 렌더 타겟에 렌더링
깊이 검사와 스텐실 검사(스텐실 값이 1일 때 성공)를 수행
거울에 반사된 객체는 은면이 보임에 유의
- ④ 거울을 렌더링(블렌딩)
반사된 객체는 거울 뒤에 존재하므로 거울을 투명하게 그려야 함



스텐실(Stencil) 응용

• 평면 거울(Planar Mirror) 구현

① 거울을 제외한 모든 객체들을 렌더링

거울을 렌더링하기 위하여 파이프라인 상태가 4개 필요함

```
class CMirrorShader : public CShader {  
    CScene *m_pScene;  
    CGameObject *m_pMirrorObject;  
};
```

```
int m_nPipelineStates = 4; //거울을 스텐실로 렌더링, 반사된 객체 렌더링, 거울을 렌더링, 거울 뒤면  
m_ppd3dPipelineStates = new ID3D12PipelineState*[m_nPipelineStates];
```

②-① 스텐실 버퍼 초기화(0)

```
pd3dCommandList->ClearDepthStencilView(d3dDsvCPUHandle, D3D12_CLEAR_STENCIL, 1.0f, 0, 0, NULL);
```

②-② 거울을 스텐실 버퍼에 렌더링(렌더 타겟에 출력하지 않음)

```
D3D12_DEPTH_STENCIL_DESC d3dMirrorToStencilDepthStencilDesc;  
d3dMirrorToStencilDepthStencilDesc.DepthEnable = true;  
d3dMirrorToStencilDepthStencilDesc.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ZERO;  
d3dMirrorToStencilDepthStencilDesc.DepthFunc = D3D12_COMPARISON_FUNC_LESS;  
d3dMirrorToStencilDepthStencilDesc.StencilEnable = true;  
d3dMirrorToStencilDepthStencilDesc.StencilReadMask = 0xff;  
d3dMirrorToStencilDepthStencilDesc.StencilWriteMask = 0xff;  
d3dMirrorToStencilDepthStencilDesc.FrontFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;  
d3dMirrorToStencilDepthStencilDesc.FrontFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;  
d3dMirrorToStencilDepthStencilDesc.FrontFace.StencilPassOp = D3D12_STENCIL_OP_REPLACE;  
d3dMirrorToStencilDepthStencilDesc.FrontFace.StencilFunc = D3D12_COMPARISON_FUNC_ALWAYS;  
d3dMirrorToStencilDepthStencilDesc.BackFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;  
d3dMirrorToStencilDepthStencilDesc.BackFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;  
d3dMirrorToStencilDepthStencilDesc.BackFace.StencilPassOp = D3D12_STENCIL_OP_REPLACE;  
d3dMirrorToStencilDepthStencilDesc.BackFace.StencilFunc = D3D12_COMPARISON_FUNC_ALWAYS;
```

스텐실(Stencil) 응용

- 평면 거울(Planar Mirror) 구현

②-② 거울을 스텐실 버퍼에 렌더링(렌더 타겟에 출력하지 않음)

```
D3D12_BLEND_DESC d3dTurnOffRenderTargetBlendDesc;  
d3dTurnOffRenderTargetBlendDesc.AlphaToCoverageEnable = false;  
d3dTurnOffRenderTargetBlendDesc.IndependentBlendEnable = false;  
d3dTurnOffRenderTargetBlendDesc.RenderTarget[0].BlendEnable = false;  
d3dTurnOffRenderTargetBlendDesc.RenderTarget[0].LogicOpEnable = false;  
d3dTurnOffRenderTargetBlendDesc.RenderTarget[0].SrcBlend = D3D12_BLEND_ONE;  
d3dTurnOffRenderTargetBlendDesc.RenderTarget[0].DestBlend = D3D12_BLEND_ZERO;  
d3dTurnOffRenderTargetBlendDesc.RenderTarget[0].BlendOp = D3D12_BLEND_OP_ADD;  
d3dTurnOffRenderTargetBlendDesc.RenderTarget[0].SrcBlendAlpha = D3D12_BLEND_ONE;  
d3dTurnOffRenderTargetBlendDesc.RenderTarget[0].DestBlendAlpha = D3D12_BLEND_ZERO;  
d3dTurnOffRenderTargetBlendDesc.RenderTarget[0].BlendOpAlpha = D3D12_BLEND_OP_ADD;  
d3dTurnOffRenderTargetBlendDesc.RenderTarget[0].LogicOp = D3D12_LOGIC_OP_NOOP;  
d3dTurnOffRenderTargetBlendDesc.RenderTarget[0].RenderTargetWriteMask = 0; //렌더 타겟 변경하지 않음
```

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC d3dPipelineStateDesc;
```

```
...
```

```
d3dPipelineStateDesc.DepthStencilState = d3dMirrorToStencilDepthStencilDesc;
```

```
d3dPipelineStateDesc.BlendState = d3dTurnOffRenderTargetBlendDesc;
```

```
...
```

```
pd3dDevice->CreateGraphicsPipelineState(&d3dPipelineStateDesc, ..., (void **)&m_ppd3dPipelineStates[0]);
```

```
pd3dCommandList->OMSetStencilRef(1);
```

```
pd3dCommandList->SetPipelineState(m_ppd3dPipelineStates[0]);
```

```
m_pMirrorObject->Render(pd3dCommandList, pCamera);
```

스텐실(Stencil) 응용

- 평면 거울(Planar Mirror) 구현

③-① 반사된 객체들을 거울에 렌더링

```
D3D12_DEPTH_STENCIL_DESC d3dReflectDepthStencilDesc;  
::ZeroMemory(&d3dReflectDepthStencilDesc, sizeof(D3D12_DEPTH_STENCIL_DESC));  
d3dReflectDepthStencilDesc.DepthEnable = true;  
d3dReflectDepthStencilDesc.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ALL;  
d3dReflectDepthStencilDesc.DepthFunc = D3D12_COMPARISON_FUNC_LESS;  
d3dReflectDepthStencilDesc.StencilEnable = true;  
d3dReflectDepthStencilDesc.StencilReadMask = d3dReflectDepthStencilDesc.StencilWriteMask = 0xff;  
d3dReflectDepthStencilDesc.FrontFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;  
d3dReflectDepthStencilDesc.FrontFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;  
d3dReflectDepthStencilDesc.FrontFace.StencilPassOp = D3D12_STENCIL_OP_KEEP;  
d3dReflectDepthStencilDesc.FrontFace.StencilFunc = D3D12_COMPARISON_FUNC_EQUAL;
```

```
D3D12_RASTERIZER_DESC d3dCWCullRasterizerDesc;  
::ZeroMemory(&d3dCWCullRasterizerDesc, sizeof(D3D12_RASTERIZER_DESC));  
d3dCWCullRasterizerDesc.FillMode = D3D12_FILL_MODE_SOLID;  
d3dCWCullRasterizerDesc.CullMode = D3D12_CULL_MODE_BACK;  
d3dCWCullRasterizerDesc.FrontCounterClockwise = true;  
d3dCWCullRasterizerDesc.DepthClipEnable = true;
```

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC d3dPipelineStateDesc;  
...  
d3dPipelineStateDesc.DepthStencilState = d3dReflectDepthStencilDesc;  
d3dPipelineStateDesc.RasterizerState = d3dCWCullRasterizerDesc;  
...  
pd3dDevice->CreateGraphicsPipelineState(&d3dPipelineStateDesc, ..., (void **)&m_ppd3dPipelineStates[1]);
```

조명(Lighting)

- 조명 자료 구조

```
#define POINT_LIGHT 1
#define SPOT_LIGHT 2
#define DIRECTIONAL_LIGHT 3
```

```
#define MAX_LIGHTS 4
#define MAX_MATERIALS 10
```

```
struct LIGHT {
    float4 m_cAmbient;
    float4 m_cDiffuse;
    float4 m_cSpecular;
    float3 m_vPosition;
    float m_fFalloff;
    float3 m_vDirection;
    float m_fTheta; //cos(m_fTheta)
    float3 m_vAttenuation;
    float m_fPhi; //cos(m_fPhi)
    bool m_bEnable;
    int m_nType;
    float m_fRange;
    float padding;
};
```

```
cbuffer cbLights : register(b4) {
    LIGHT gLights[MAX_LIGHTS];
    float4 gcGlobalAmbientLight;
};
```

```
cbuffer cbMaterials : register(b3) {
    MATERIAL gMaterials[MAX_MATERIALS];
};
```

```
struct MATERIAL {
    float4 m_cAmbient;
    float4 m_cDiffuse;
    float4 m_cSpecular;
    float4 m_cEmissive;
};
```

```
struct LIGHT {
    XMFLOAT4 m_xmf4Ambient;
    XMFLOAT4 m_xmf4Diffuse;
    XMFLOAT4 m_xmf4Specular;
    XMFLOAT3 m_xmf3Position;
    float m_fFalloff;
    XMFLOAT3 m_xmf3Direction;
    float m_fTheta;
    XMFLOAT3 m_xmf3Attenuation;
    float m_fPhi;
    bool m_bEnable;
    int m_nType;
    float m_fRange;
    float unused;
};
```

```
struct MATERIAL {
    XMFLOAT4 m_xmf4Ambient;
    XMFLOAT4 m_xmf4Diffuse;
    XMFLOAT4 m_xmf4Specular;
    XMFLOAT4 m_xmf4Emissive;
};
```

```
struct LIGHTS {
    LIGHT m_pLights[MAX_LIGHTS];
    XMFLOAT4 m_xmf4GlobalAmbient;
};
```

```
struct MATERIALS {
    MATERIAL m_pReflections[MAX_MATERIALS];
};
```

스텐실(Stencil) 응용

- 평면 거울(Planar Mirror) 구현

③-② 거울에 반사된 조명의 위치와 방향을 반영

```
XMVECTOR xmvpMirrorPlane = ...; //거울 평면 방정식
XMMATRIX xmmtxReflect = XMMatrixReflect(xmvpMirrorPlane);
for (int i = 0; i < MAX_LIGHTS; i++)
{
    XMVECTOR xmvpLightPos = XMLoadFloat3(&m_pScene->m_pLights->m_pLights[i].m_xmf3Position);
    XMVECTOR xmvpReflectedLightPos = XMVector3TransformCoord(xmvpLightPos, xmmtxReflect);
    XMStoreFloat3(&m_pScene->m_pcbMappedLights[i].m_xmf3Position, xmvpReflectedLightPos);
    XMVECTOR xmvpLightDir = XMLoadFloat3(&m_pScene->m_pLights->m_pLights[i].m_xmf3Direction);
    XMVECTOR xmvpReflectedLightDir = XMVector3TransformNormal(xmvpLightDir, xmmtxReflect);
    XMStoreFloat3(&m_pScene->m_pcbMappedLights[i].m_xmf3Direction, xmvpReflectedLightDir);
}
```

③-③ 반사된 객체를 렌더링(반사된 조명을 사용)

```
pd3dCommandList->OMSetStencilRef(1);
pd3dCommandList->SetPipelineState(m_ppd3dPipelineStates[1]);
for (int i = 0; i < m_pScene->m_nObjects; i++)
{
    XMFLOAT4X4 xmf4x4World = m_pScene->m_pObjects[i]->m_mtxWorld;
    XMMATRIX xmmtxWorld = XMLoadFloat4x4(&m_pScene->m_pObjects[i]->m_mtxWorld);
    XMMATRIX xmmtxReflected = XMMatrixMultiply(xmmtxWorld, xmmtxReflect);
    XMStoreFloat4x4(&m_pScene->m_pObjects[i]->m_mtxWorld, xmmtxReflected);
    m_pScene->m_pObjects[i]->Render(pd3dCommandList, pCamera);
    m_pScene->m_pObjects[i]->m_mtxWorld = xmf4x4World;
}
```

스텐실(Stencil) 응용

- 평면 거울(Planar Mirror) 구현

④ 거울을 렌더링(블렌딩)

```
D3D12_BLEND_DESC d3dTransparentBlendDesc;  
::ZeroMemory(&d3dTransparentBlendDesc, sizeof(D3D12_BLEND_DESC));  
d3dTransparentBlendDesc.AlphaToCoverageEnable = false;  
d3dTransparentBlendDesc.IndependentBlendEnable = false;  
d3dTransparentBlendDesc.RenderTarget[0].BlendEnable = true;  
d3dTransparentBlendDesc.RenderTarget[0].LogicOpEnable = false;  
d3dTransparentBlendDesc.RenderTarget[0].SrcBlend = D3D12_BLEND_SRC_ALPHA;  
d3dTransparentBlendDesc.RenderTarget[0].DestBlend = D3D12_BLEND_INV_SRC_ALPHA;  
d3dTransparentBlendDesc.RenderTarget[0].BlendOp = D3D12_BLEND_OP_ADD;  
d3dTransparentBlendDesc.RenderTarget[0].SrcBlendAlpha = D3D12_BLEND_ONE;  
d3dTransparentBlendDesc.RenderTarget[0].DestBlendAlpha = D3D12_BLEND_ZERO;  
d3dTransparentBlendDesc.RenderTarget[0].BlendOpAlpha = D3D12_BLEND_OP_ADD;  
d3dTransparentBlendDesc.RenderTarget[0].LogicOp = D3D12_LOGIC_OP_NOOP;  
d3dTransparentBlendDesc.RenderTarget[0].RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;
```

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC d3dPipelineStateDesc;
```

```
...
```

```
d3dPipelineStateDesc.BlendState = d3dTransparentBlendDesc;
```

```
...
```

```
pd3dDevice->CreateGraphicsPipelineState(&d3dPipelineStateDesc, ..., (void **)&m_ppd3dPipelineStates[2]);
```

```
pd3dCommandList->OMSetStencilRef(0);
```

```
pd3dCommandList->SetPipelineState(m_ppd3dPipelineStates[2]);
```

```
m_pMirrorObject->Render(pd3dCommandList, pCamera);
```

스텐실(Stencil) 응용

- 평면 거울(Planar Mirror) 구현

* 거울 뒷면을 렌더링(깊이 검사를 하지만 깊이 값을 깊이 버퍼에 쓰지 않음)

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC d3dPipelineStateDesc;  
::ZeroMemory(&d3dPipelineStateDesc.DepthStencilState, sizeof(D3D12_DEPTH_STENCIL_DESC));  
d3dPipelineStateDesc.DepthStencilState.DepthEnable = true;  
d3dPipelineStateDesc.DepthStencilState.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ZERO;  
d3dPipelineStateDesc.DepthStencilState.DepthFunc = D3D12_COMPARISON_FUNC_LESS;  
d3dPipelineStateDesc.DepthStencilState.StencilEnable = false;  
pd3dDevice->CreateGraphicsPipelineState(&d3dPipelineStateDesc, ..., (void **)&m_ppd3dPipelineStates[3]);
```

```
CMirrorShader::Render(ID3D12GraphicsCommandList *pd3dCommandList, CCamera *pCamera)  
{  
    pd3dCommandList->SetPipelineState(m_ppd3dPipelineStates[3]); //거울 뒷면을 렌더링  
    m_pMirrorBackObject->Render(pd3dCommandList, pCamera);  
    pd3dCommandList->ClearDepthStencilView(d3dDsvCPUHandle, D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, ...);  
    pd3dCommandList->OMSetStencilRef(1);  
    pd3dCommandList->SetPipelineState(m_ppd3dPipelineStates[0]);  
    m_pMirrorObject->Render(pd3dCommandList, pCamera); //거울을 스텐실 버퍼에 렌더링  
    pd3dCommandList->OMSetStencilRef(1);  
    pd3dCommandList->SetPipelineState(m_ppd3dPipelineStates[1]);  
    for (int i = 0; i < m_pScene->m_nObjects; i++) { ... } //거울에 반사된 객체들을 렌더링  
    pd3dCommandList->OMSetStencilRef(0);  
    pd3dCommandList->SetPipelineState(m_ppd3dPipelineStates[2]); //거울을 렌더링(블렌딩)  
    m_pMirrorObject->Render(pd3dCommandList, pCamera);  
}
```