II. Direct3D Foundations
10. Blending

Game Graphic Programming

Kyung Hee University

Software Convergence

Prof. Daeho Lee

The Blending Equation

- Blending equation
  - Let $\mathbf{C}_{src}$ be the color output from the pixel shader for the ijth pixel we are currently rasterizing (source pixel), and let $\mathbf{C}_{dst}$ be the color of the ijth pixel currently on the back buffer (destination pixel).
  - Without blending, $\mathbf{C}_{src}$ would overwrite $\mathbf{C}_{dst}$ (assuming it passes the depth/stencil test) and become the new color of the ijth back buffer pixel. But with blending, $\mathbf{C}_{src}$ and $\mathbf{C}_{dst}$ are blended together to get the new color $\mathbf{C}$ that will overwrite $\mathbf{C}_{dst}$ (i.e., the blended color $\mathbf{C}$ will be written to the ijth pixel of the back buffer).

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \bigcirc \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$ ○: binary blend operator

    - $\mathbf{F}_{src}$ and F_{dst} are the source blend factor and the destination blend factor, respectively.
    - The alpha component is actually handled by a separate similar equation:

$$A = A_{src} \otimes F_{src} \bigcirc A_{dst} \otimes F_{dst}$$

Blend Operations (1)

- The binary ◯ operator used in the blending equation may be one of the following:

```
typedef enum D3D12_BLEND_OP {
  D3D12_BLEND_OP_ADD = 1,              // C_src ⊗ F_src + C_dst ⊗ F_dst
  D3D12_BLEND_OP_SUBTRACT = 2,         // C_dst ⊗ F_dst − C_src ⊗ F_src
  D3D12_BLEND_OP_REV_SUBTRACT = 3,     // C_src ⊗ F_src − C_dst ⊗ F_dst
  D3D12_BLEND_OP_MIN = 4,              // min(C_src, C_dst)
  D3D12_BLEND_OP_MAX = 5               // max(C_src, C_dst)
} ;
```

Blend Operations (1)

- Logic operations

```
typedef enum D3D12_LOGIC_OP {
  D3D12_LOGIC_OP_CLEAR = 0,
  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
} ;
```

You cannot use traditional blending and logic operator blending at the same time; you pick one or the other.

Blend Factors (1)

- **D3D12_BLEND** specifies blend factors, which modulate values for the pixel shader and render target.

```
typedef enum D3D12_BLEND {
  D3D12_BLEND_ZERO = 1,              // F=(0,0,0), F = 0
  D3D12_BLEND_ONE = 2,               // F=(1,1,1), F = 1
  D3D12_BLEND_SRC_COLOR = 3,         // F=(rs,gs,bs)
  D3D12_BLEND_INV_SRC_COLOR = 4,     // F=(1−rs,1−gs,1−bs)
  D3D12_BLEND_SRC_ALPHA = 5,         // F=(as,as,as), F=as
  D3D12_BLEND_INV_SRC_ALPHA = 6,     // F=(1−as,1−as,1−as), F= 1−as
  D3D12_BLEND_DEST_ALPHA = 7,        // F=(ad,ad,ad), F=ad
  D3D12_BLEND_INV_DEST_ALPHA = 8,    // F=(1−ad,1−ad,1−ad), F= 1−ad
```

Blend Factors (1)

```
    D3D12_BLEND_DEST_COLOR = 9,                 // F=(rd,gd,bd)
    D3D12_BLEND_INV_DEST_COLOR = 10,            // F=(1−rd,1−gd,1−bd)
    D3D12_BLEND_SRC_ALPHA_SAT = 11,             // F=(a's,a's,a's), F=a's
      // a's=clamp(as,0,1)
    D3D12_BLEND_BLEND_FACTOR = 14,              // F=(r,g,b), F=a
      // (r,g,b,a) is supplied to the second parameter of the
      // ID3D12GraphicsCommandList::OMSetBlendFactor method.
    D3D12_BLEND_INV_BLEND_FACTOR = 15,          // F=(1−r,1−g,1−b), F=1−a
    // ...
} ;
```

$$\text{clamp}(x,a,b)=\begin{cases} x & a \le x \le b \\ a & x < a \\ b & b < x \end{cases}$$

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

Blend State (1)

- Blend state
  - As with other Direct3D states, the blend state is part of the PSO. Thus far we have been using the default blend state, which disables blending.
  - The graphics pipeline state object (PSO) is described by:

```
typedef struct D3D12_GRAPHICS_PIPELINE_STATE_DESC {
  // …
  D3D12_BLEND_DESC                    BlendState;
  // …
} D3D12_GRAPHICS_PIPELINE_STATE_DESC;

D3D12_GRAPHICS_PIPELINE_STATE_DESC PsoDesc;
PsoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
```

Blend State (2)

```
CD3DX12_BLEND_DESC(D3D12_DEFAULT);
    AlphaToCoverageEnable                 FALSE
    IndependentBlendEnable                        FALSE
    RenderTarget[0].BlendEnable           FALSE
    RenderTarget[0].LogicOpEnable FALSE
    RenderTarget[0].SrcBlend                      D3D12_BLEND_ONE
    RenderTarget[0].DestBlend             D3D12_BLEND_ZERO
    RenderTarget[0].BlendOp               D3D12_BLEND_OP_ADD
    RenderTarget[0].SrcBlendAlpha D3D12_BLEND_ONE
    RenderTarget[0].DestBlendAlphaD3D12_BLEND_ZERO
    RenderTarget[0].BlendOpAlpha  D3D12_BLEND_OP_ADD
    RenderTarget[0].LogicOp               D3D12_LOGIC_OP_NOOP
    RenderTarget[0].RenderTargetWriteMask
                                  D3D12_COLOR_WRITE_ENABLE_ALL
```

Blend State (3)

- The blend state is described by:

```
typedef struct D3D12_BLEND_DESC {
 BOOL AlphaToCoverageEnable;          // Default: False
 BOOL IndependentBlendEnable;         // Default: False
 D3D11_RENDER_TARGET_BLEND_DESC RenderTarget[8];
} D3D11_BLEND_DESC
```

  - **`AlphaToCoverageEnable`**: Specify whether to enable alpha-to-coverage. Alpha-to-coverage requires multisampling to be enabled (i.e., the back and depth buffers were created with multisampling).
  - **`IndependentBlendEnable`**: Direct3D supports rendering to up to eight render targets simultaneously. When this flag is set to **`true`**, it means blending can be performed for each render target differently (different blend factors, different blend operations, blending disabled/enabled, etc.). If this flag is set to **`false`**, it means all the render targets will be blended the same way as described by the first element in the **`D3D12_BLEND_DESC::RenderTarget`** array.
  - **`RenderTarget`**: An array of 8 **`D3D12_RENDER_TARGET_BLEND_DESC`** elements. These correspond to the eight render targets that can be bound to the output-merger stage at one time.

Blend State (3)

- D3D12_RENDER_TARGET_BLEND_DESC
  - This describes the blend state for a render target.

```
typedef struct D3D12_RENDER_TARGET_BLEND_DESC {
  BOOL           BlendEnable;
  BOOL           LogicOpEnable;
  D3D12_BLEND    SrcBlend;
  D3D12_BLEND    DestBlend;
  D3D12_BLEND_OP BlendOp;
  D3D12_BLEND    SrcBlendAlpha;
  D3D12_BLEND    DestBlendAlpha;
  D3D12_BLEND_OP BlendOpAlpha;
  D3D12_LOGIC_OP LogicOp;
  UINT8          RenderTargetWriteMask;
} D3D12_RENDER_TARGET_BLEND_DESC;
```

Examples

- No Color Write

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

- Adding

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

- Multiplying

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

- Transparency

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

Blending and the Depth Buffer (1)

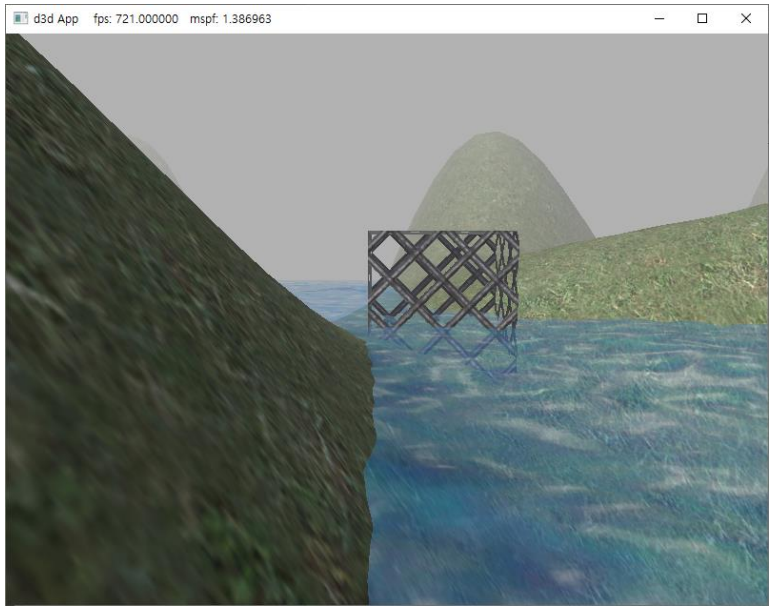
- If we are rendering a set S of objects with additive blending, their colors are meant to simply accumulate.

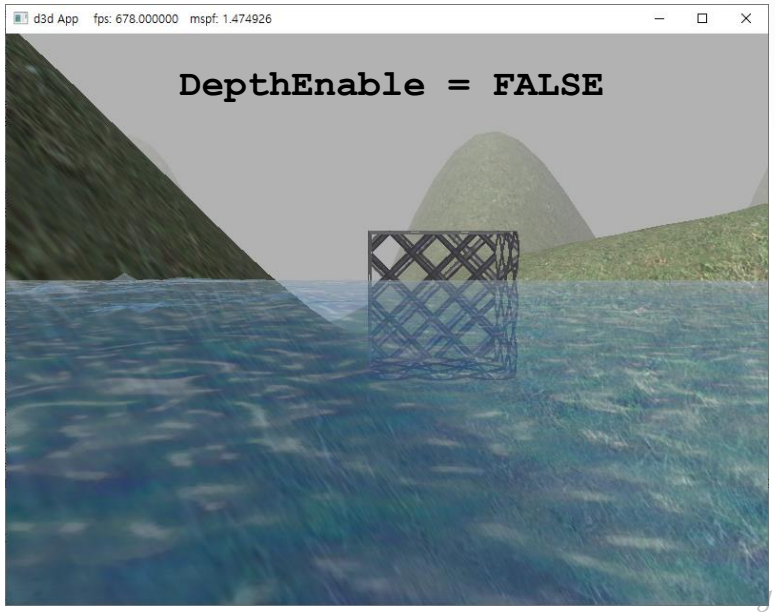
- The depth test compares the depths of pixels competing to be written to a particular pixel location on the back buffer.

- We do not want to perform the depth test between objects in S, without a back-to-front draw ordering, one of the objects in S would obscure another object in S, thus the object's pixel colors would not be accumulated into the blend sum.

Blending and the Depth Buffer (2)

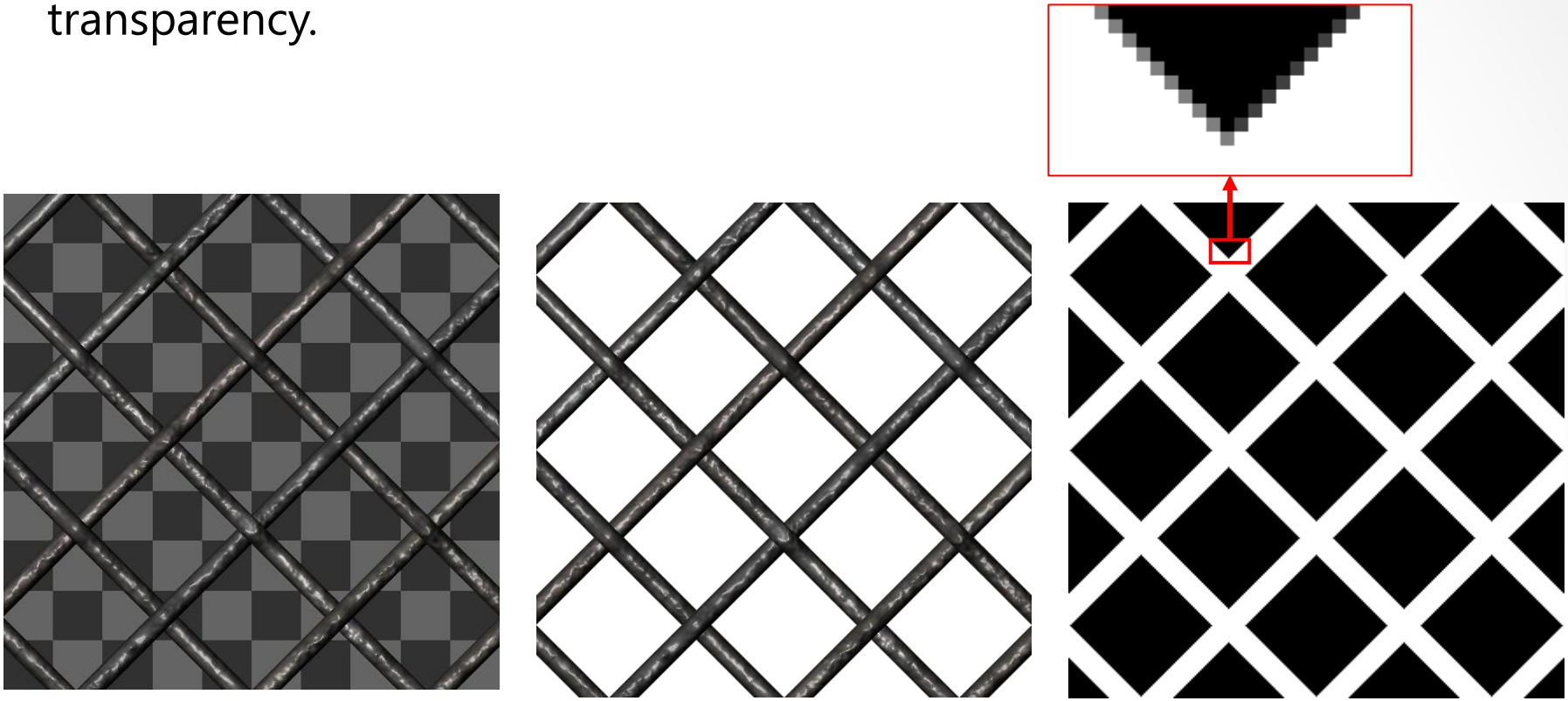
```
void Application::BuildPSOs() {
// …
D3D12_GRAPHICS_PIPELINE_STATE_DESC PsoDesc;
PsoDesc.DepthStencilState = CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT);
→
D3D12_DEPTH_STENCIL_DESC ds_desc =
    CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT);
ds_desc.DepthEnable = FALSE;
PsoDesc.DepthStencilState = ds_desc;
//…
}
```





Alpha Channels (1)

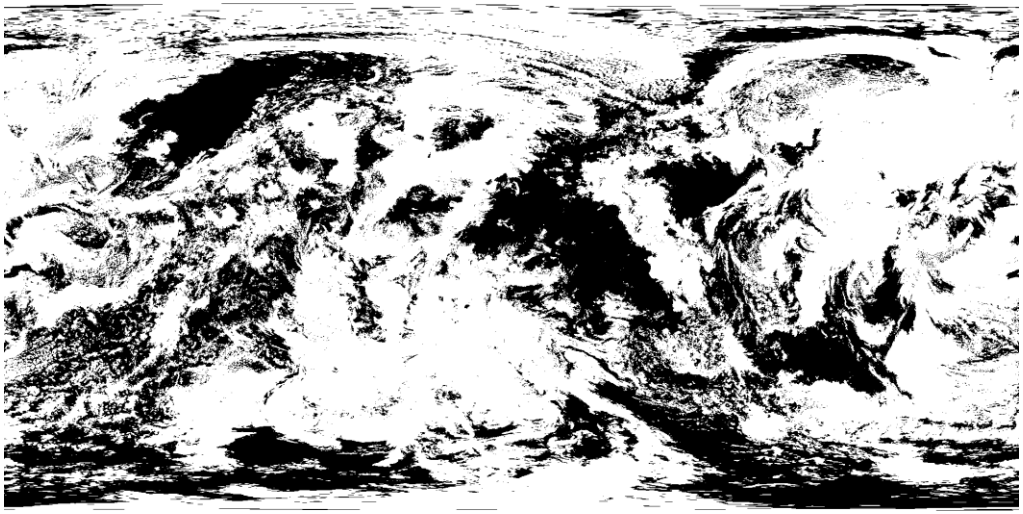
- The source alpha components can be used in RGB blending to control transparency.

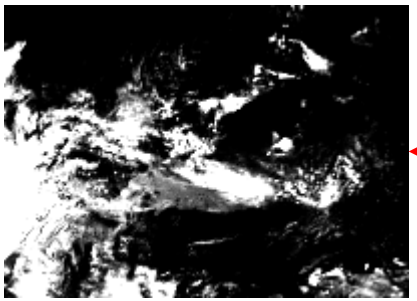


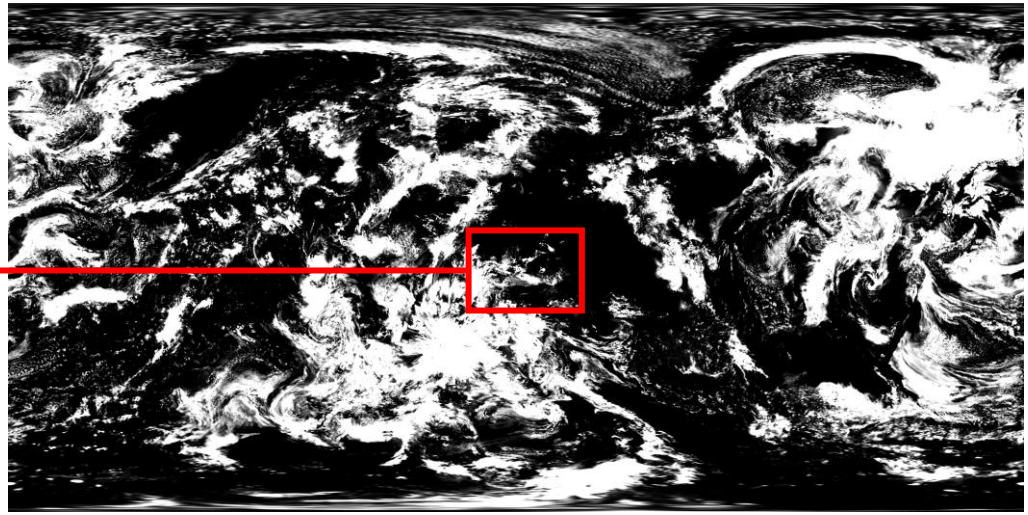
The alpha channel determines the opacity (transparency) of a pixel, ranging from fully transparent (0) to fully opaque (1).

Alpha Channels (2)

RGB



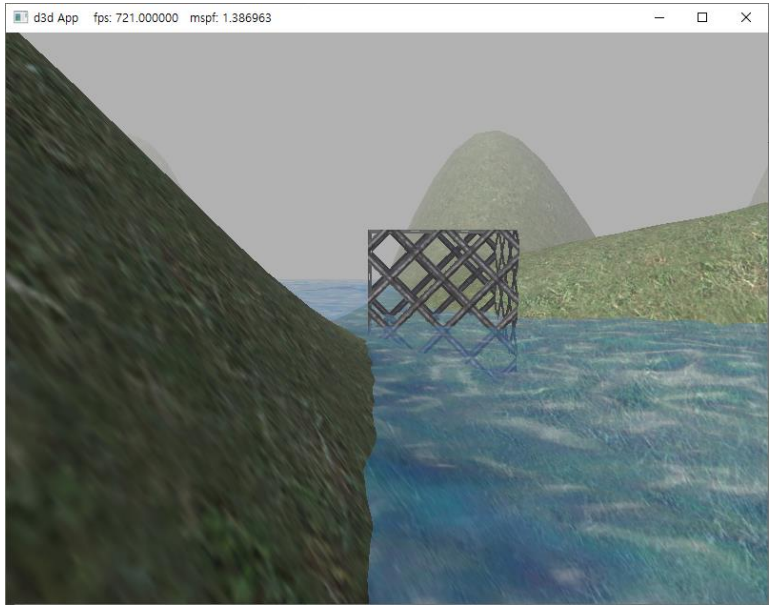


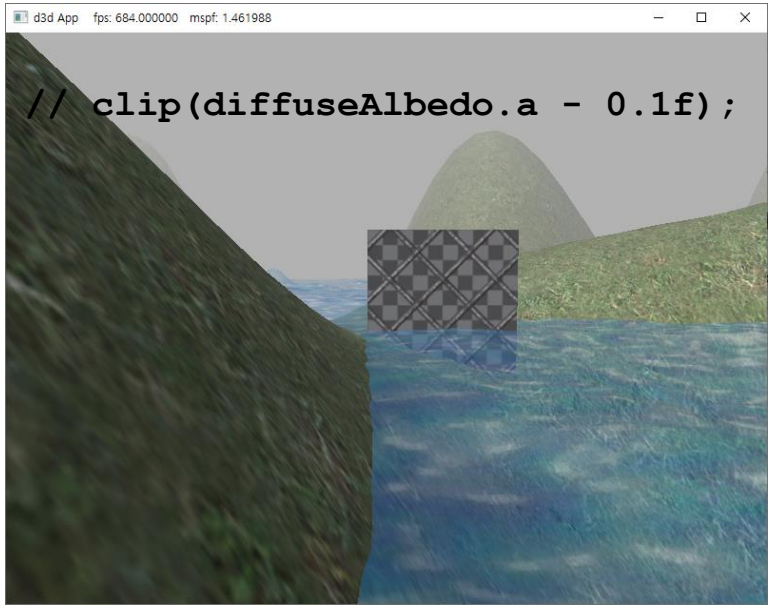


A

Clipping Pixels

- The transparent pixels will be rejected by the **`clip`** function and not drawn.
  - **`clip(x)`**: It discards the current pixel if the specified value (**`x`**) is less than zero. [HLSL, PS]
    - **`clip(diffuseAlbedo.a - 0.1f);`**
      - It discards pixels if the texture alpha < 0.1.





Fog (1)

- To simulate certain types of weather conditions in our games, we need to be able to implement a fog effect.
- In addition to the obvious purposes of fog, fog provides some fringe benefits. For example, it can mask distant rendering artifacts and prevent popping.
  - Popping refers to when an object that was previously behind the far plane suddenly comes in front of the frustum, due to camera movement, and thus becomes visible; so it seems to "pop" into the scene abruptly.

$$C^* = C + s(C_f - C) = (1-s)C + sC_f \qquad 0 \le s \le 1$$

    - C_f: fog color
  - As the distance between a surface point and the eye increases, the point becomes more and more obscured by the fog.

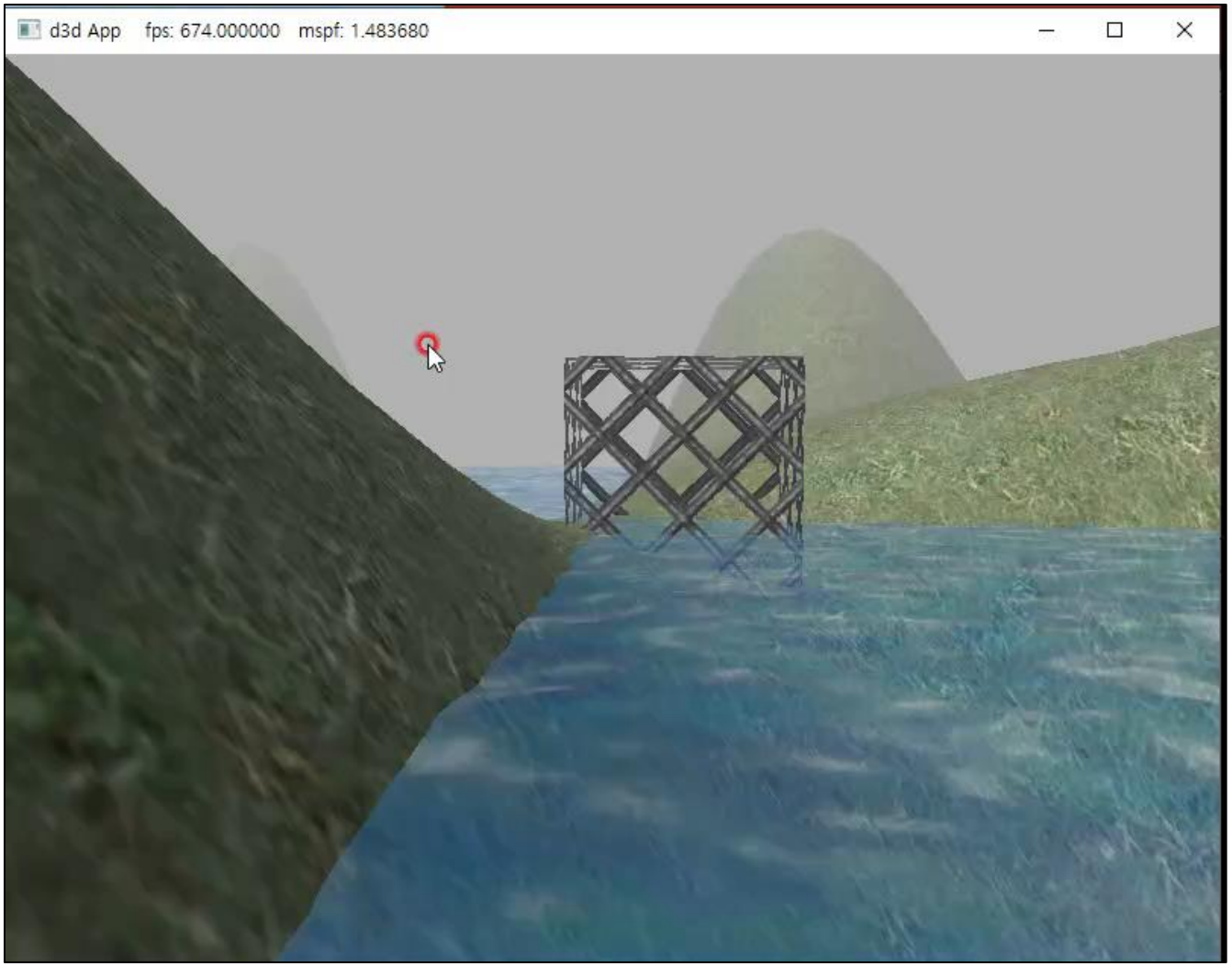
$$s = \text{saturate}\left(\frac{\text{dist}(\mathbf{p}, \mathbf{E}) - f_s}{f_r}\right)$$

    - f_s: distance at which the fog starts affecting the color, f_r: fog range
    - $\mathbf{p}$: surface point, $\mathbf{E}$: camera position

Fog (2)

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

#ifdef ALPHA_TEST
    clip(diffuseAlbedo.a - 0.1f);
#endif
    pin.NormalW = normalize(pin.NormalW);
    float3 toEyeW = gEyePosW - pin.PosW;
    float distToEye = length(toEyeW);
// …
     float4 litColor = ambient + directLight;
#ifdef FOG
    float fogAmount = saturate((distToEye - gFogStart) / gFogRange);
    litColor = lerp(litColor, gFogColor, fogAmount);
#endif
    litColor.a = diffuseAlbedo.a;
    return litColor;
}
// ret lerp(x, y, s)
// x and y: the first and the second floating point values, respectively.
// s: a value for linear interpolation.
```

Blend Demo



Items

```
enum class RenderLayer : int {Opaque = 0, Transparent, AlphaTested, Count};
std::vector<RenderItem*> mRitemLayer[(int)RenderLayer::Count];

void BlendApp::BuildRenderItems() {
    auto wavesRitem = std::make_unique<RenderItem>();
// …
    mRitemLayer[(int)RenderLayer::Transparent].push_back(wavesRitem.get());

    auto gridRitem = std::make_unique<RenderItem>();
// …
    mRitemLayer[(int)RenderLayer::Opaque].push_back(gridRitem.get());

    auto boxRitem = std::make_unique<RenderItem>();
// …
    mRitemLayer[(int)RenderLayer::AlphaTested].push_back(boxRitem.get());

    mAllRitems.push_back(std::move(wavesRitem));
    mAllRitems.push_back(std::move(gridRitem));
    mAllRitems.push_back(std::move(boxRitem));
}
```

HLSL (1)

```
cbuffer cbPass : register(b1) {
   // …
   float4 gFogColor;
   float gFogStart;
   float gFogRange;
   float2 cbPerObjectPad2;

   Light gLights[MaxLights];
};

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

#ifdef ALPHA_TEST
   clip(diffuseAlbedo.a - 0.1f);
#endif
```

HLSL (2)

```
    pin.NormalW = normalize(pin.NormalW);
    float3 toEyeW = gEyePosW - pin.PosW;
    float distToEye = length(toEyeW);
    toEyeW /= distToEye;
    float4 ambient = gAmbientLight*diffuseAlbedo;
    const float shininess = 1.0f - gRoughness;
    Material mat = { diffuseAlbedo, gFresnelR0, shininess };
    float3 shadowFactor = 1.0f;
    float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
        pin.NormalW, toEyeW, shadowFactor);

    float4 litColor = ambient + directLight;
#ifdef FOG
    float fogAmount = saturate((distToEye - gFogStart) / gFogRange);
    litColor = lerp(litColor, gFogColor, fogAmount);
#endif
    litColor.a = diffuseAlbedo.a;
    return litColor;
}
```

Shaders

```
std::unordered_map<std::string, ComPtr<ID3DBlob>> mShaders;

void BlendApp::BuildShadersAndInputLayout() {
    const D3D_SHADER_MACRO defines[] = {"FOG", "1", NULL, NULL};
    const D3D_SHADER_MACRO alphaTestDefines[] =
        {"FOG", "1",  "ALPHA_TEST", "1", NULL, NULL};

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

```
typedef struct _D3D_SHADER_MACRO {
  LPCSTR Name;
  LPCSTR Definition;
} D3D_SHADER_MACRO, *LPD3D_SHADER_MACRO;
```

PSOs (1)

```
std::unordered_map<std::string, ComPtr<ID3D12PipelineState>> mPSOs;

void BlendApp::BuildPSOs() {
    D3D12_GRAPHICS_PIPELINE_STATE_DESC opaquePsoDesc;

    ZeroMemory(&opaquePsoDesc, sizeof(D3D12_GRAPHICS_PIPELINE_STATE_DESC));
    opaquePsoDesc.InputLayout = { mInputLayout.data(),
        (UINT)mInputLayout.size() };
    opaquePsoDesc.pRootSignature = mRootSignature.Get();

    opaquePsoDesc.VS = {
        reinterpret_cast<BYTE*>(mShaders["standardVS"]->GetBufferPointer()),
        mShaders["standardVS"]->GetBufferSize()
    };

    opaquePsoDesc.PS = {
        reinterpret_cast<BYTE*>(mShaders["opaquePS"]->GetBufferPointer()),
        mShaders["opaquePS"]->GetBufferSize()
    };
```

PSOs (2)

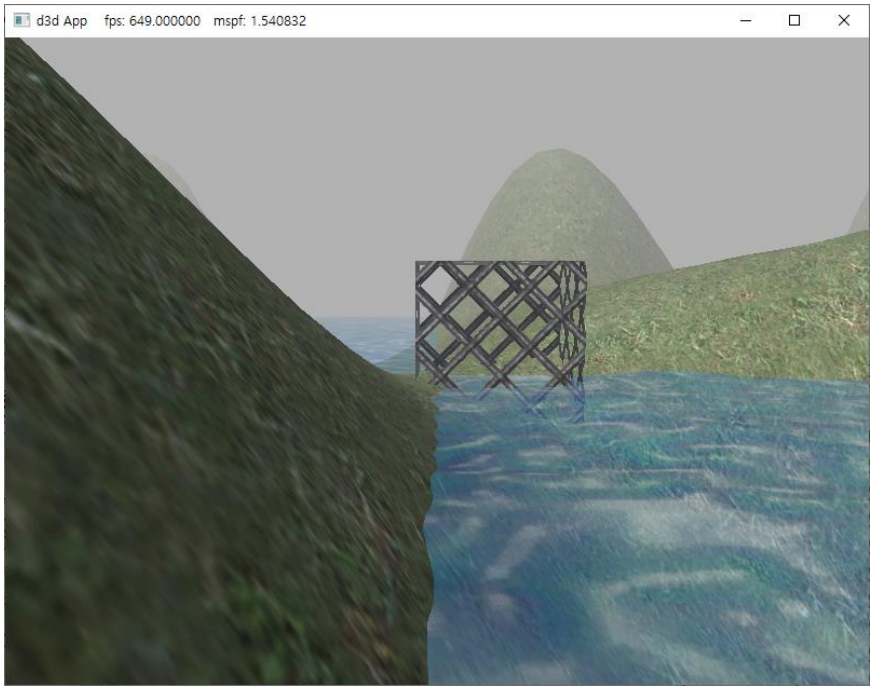
```
opaquePsoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
opaquePsoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
opaquePsoDesc.DepthStencilState = CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT);
opaquePsoDesc.SampleMask = UINT_MAX;
opaquePsoDesc.PrimitiveTopologyType= D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
opaquePsoDesc.NumRenderTargets = 1;
opaquePsoDesc.RTVFormats[0] = mBackBufferFormat;
opaquePsoDesc.SampleDesc.Count = m4xMsaaState ? 4 : 1;
opaquePsoDesc.SampleDesc.Quality = m4xMsaaState ? (m4xMsaaQuality - 1) : 0;
opaquePsoDesc.DSVFormat = mDepthStencilFormat;

ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&opaquePsoDesc,
    IID_PPV_ARGS(&mPSOs["opaque"])));
```

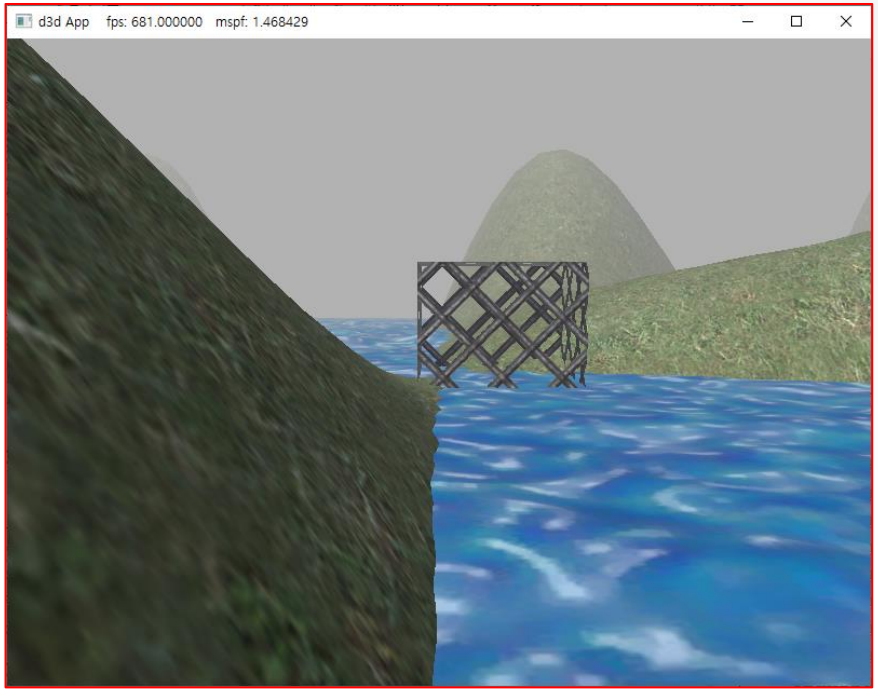
PSOs (3)

```
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"])));

                water->DiffuseAlbedo = XMFLOAT4(1.0f, 1.0f, 1.0f, 0.5f);
```

PSOs (4)

```
// transparentPsoDesc.BlendState.RenderTarget[0]
//  = transparencyBlendDesc;
```





PSOs (5)

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

```
// alphaTestedPsoDesc.RasterizerState.CullMode = D3D12_CULL_MODE_NONE;
```

Draw

```
void BlendApp::Draw(const GameTimer& gt) {
    auto cmdListAlloc = mCurrFrameResource->CmdListAlloc;

    ThrowIfFailed(cmdListAlloc->Reset());
    mCommandList->Reset(cmdListAlloc.Get(), mPSOs["opaque"].Get());

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

    mCommandList->SetPipelineState(mPSOs["alphaTested"].Get());
    DrawRenderItems(mCommandList.Get(),
        mRitemLayer[(int)RenderLayer::AlphaTested]);

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