9. Output-Merging

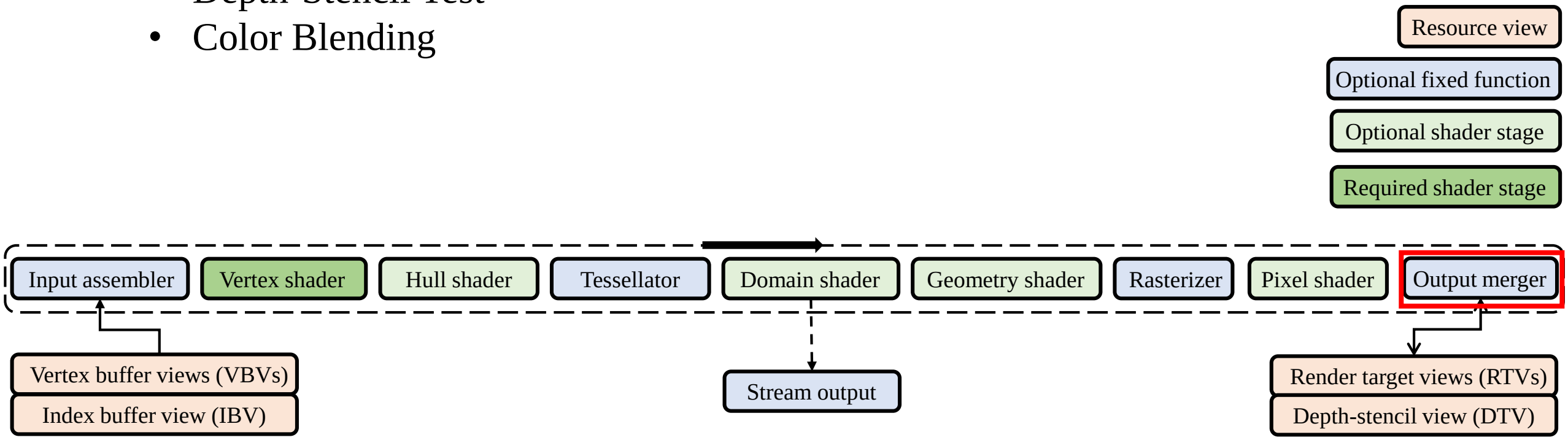
Prof. HyeongYeop Kang
siamiz@khu.ac.kr
YouTube: HKang IIIXR LAB
IIIXR LAB

Output-Merger Stage

Rendering pipeline (revisited)

- The output of the Pixel Shader, pixel, is passed through a sequence of operations.

- We call these as output-merger stage:
  - Depth-Stencil Test
  - Color Blending



Output-Merger Stage

Introduction to output-merger stage

- The output-merger (OM) stage generates the final rendered pixel color using
  1) a combination of pipeline state,
  2) the pixel data generated by the pixel shaders,
  3) the contents of the render targets,
  4) and the contents of the depth/stencil buffers.

Pipeline State

Pipeline state

- Pipeline state is a hardware setting (or description) that determines how to interpret and process the GPU input data for rendering.
- This includes common settings such as the rasterizer state, blend state, and depth-stencil state, as well as the primitive topology type of the given geometry information and the shader states.
- In Direct3D 12, most graphics pipeline state is set by using pipeline state objects (PSO).
- Multiple number of PSOs are typically created at initialization time.
- Then, they are quickly switched at rendering time to use different pipeline states for rendering.

Pipeline State

Pipeline state

- In the reference topic for the D3D12_GRAPHICS_PIPELINE_STATE_DESC presents all of the different pipeline states.

```
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;
  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;
```

Render Target

What is a render target?

- A render target is a resource or object that can receive drawing commands.
- Render targets enable a scene to be rendered to a temporary intermediate buffer, rather than to the back buffer to be rendered to the screen.
- Render target drawing methods allow users to render content on the render target.

What is framebuffer?

- The term ‘frame buffer’ traditionally refers to a portion of random-access memory (RAM) that stores the color data.

What is back buffer?

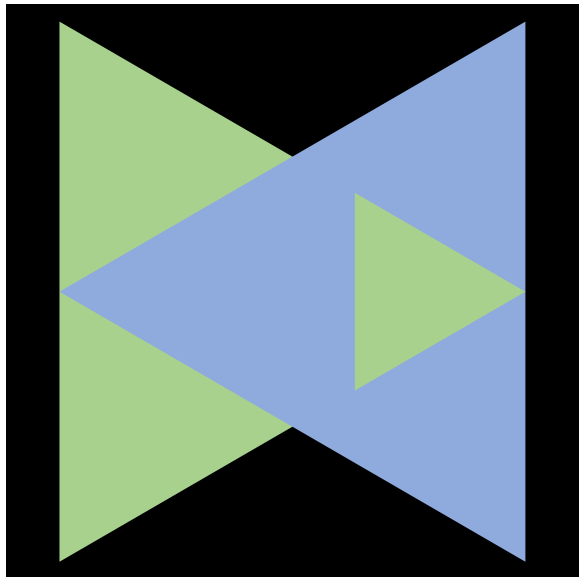
- The framebuffer that is currently being displayed is called the front buffer or primary buffer, and the framebuffer that we are drawing to is called the back buffer or secondary buffer.

Depth-Stencil Testing

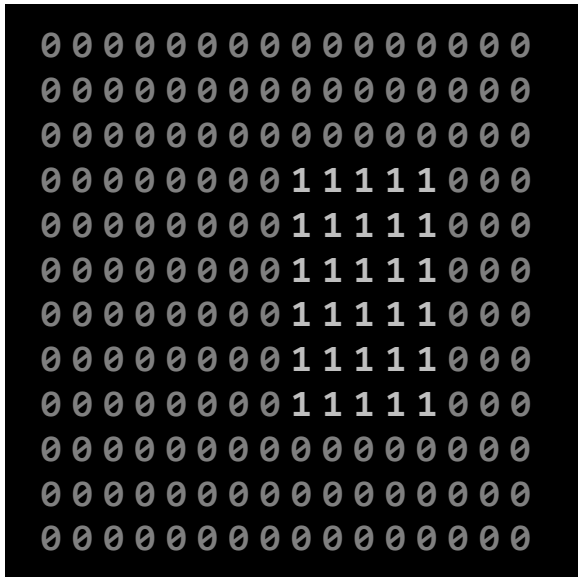
Depth Test and Stencil Test

- A depth-stencil buffer (texture resource), can contain both depth data and stencil data.
- The depth data is used to determine which pixels lie closest to the camera, and the stencil data is used to mask which pixels can be updated.
- Both the depth and stencil values data are used by the output-merger stage to determine if a pixel should be drawn or not.

only the pixels allowed by the stencil buffer are rendered.



depth test example



stencil test example

Depth-Stencil Testing

In DX12, the creation of a depth-stencil resource uses a texture resource.

depth-stencil resource

```
typedef struct D3D12_DEPTH_STENCIL_VIEW_DESC {
  DXGI_FORMAT           Format;
  D3D12_DSV_DIMENSION ViewDimension;
  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 DXGI_FORMAT {
  DXGI_FORMAT_UNKNOWN = 0,
  DXGI_FORMAT_R32G32B32A32_TYPELESS = 1,
  DXGI_FORMAT_R32G32B32A32_FLOAT = 2,
  DXGI_FORMAT_R32G32B32A32_UINT = 3,
```

```
typedef enum D3D12_DSV_DIMENSION {
  D3D12_DSV_DIMENSION_UNKNOWN = 0,
  D3D12_DSV_DIMENSION_TEXTURE1D = 1,
  D3D12_DSV_DIMENSION_TEXTURE1DARRAY = 2,
  D3D12_DSV_DIMENSION_TEXTURE2D = 3,
  D3D12_DSV_DIMENSION_TEXTURE2DARRAY = 4,
  D3D12_DSV_DIMENSION_TEXTURE2DMS = 5,
  D3D12_DSV_DIMENSION_TEXTURE2DMSARRAY = 6
} ;
```

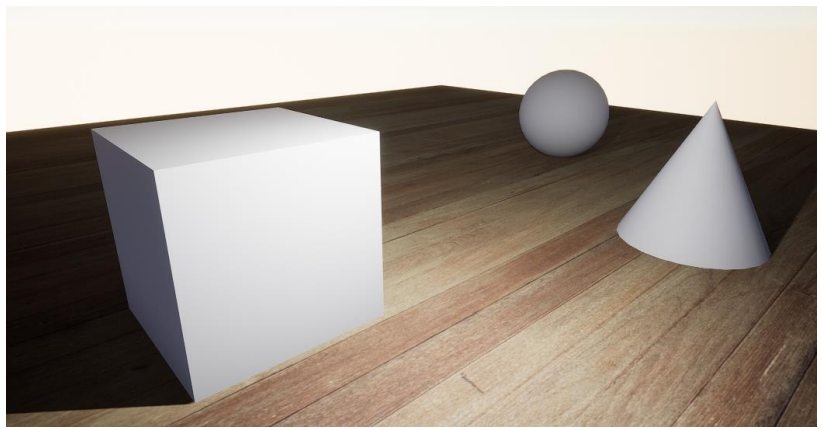
Depth-Stencil Testing

Depth Test

- The process of using the depth buffer to determine which pixel should be drawn is called depth buffering, also sometimes called z-buffering.

- Once depth values reach the output-merger stage (whether coming from interpolation or from a pixel shader) they are always clamped by
  - z = *min*(Viewport.MaxDepth, *max*(Viewport.MinDepth, z))

- After clamping, the depth value is compared against existing depth-buffer value.



3D scene and its depth buffer

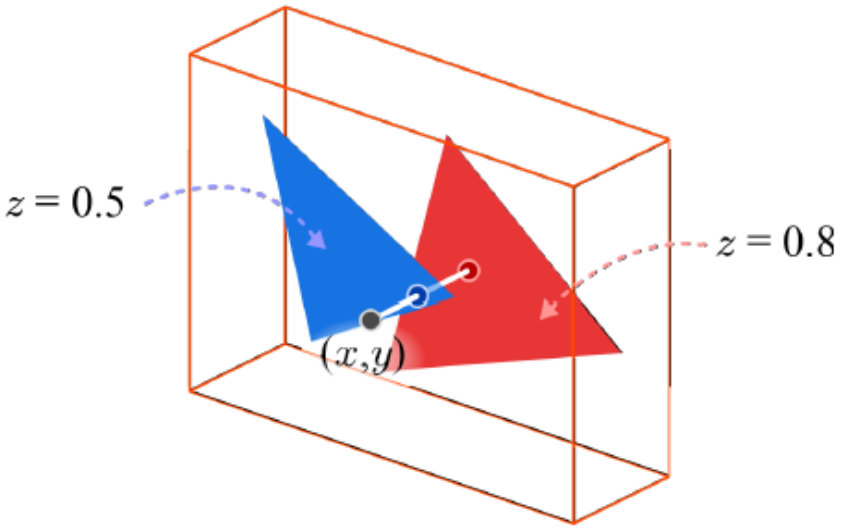
Depth-Stencil Testing

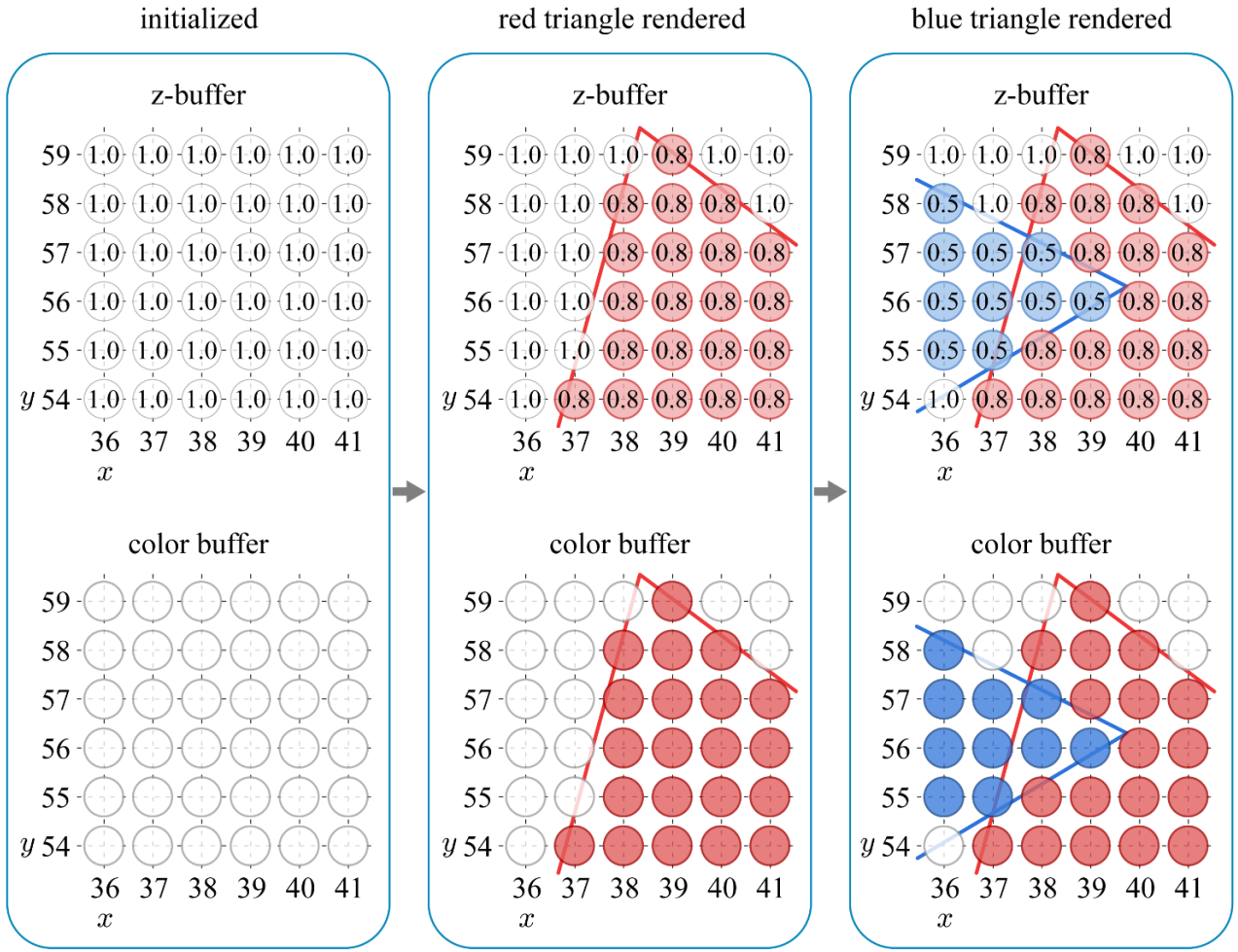
Depth Test

- The depth test is a per-sample operation.
- The depth test determines which pixel are visible, and which are hidden.
- When an object is projected on the screen, the depth of a generated pixel is compared with the current depth buffer value.
- If the pixel's depth value is smaller than stored depth value, the pixel is judged to be in front of the pixel. Therefore, pixel's color and depth value update the color buffer and depth buffer, respectively.
- Otherwise, the pixel is judged to lie behind the pixel and thus be invisible.

Depth Test

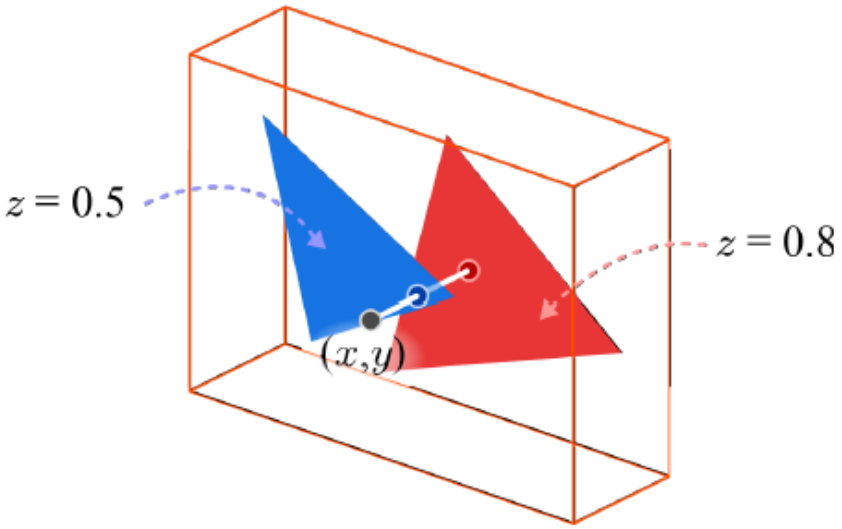
Example 1. Assume min_depth (min_z) is 0.0, max_depth (max z) is 1.0, the red triangle's depth is 0.8, and the blue one's is 0.5. In the following example, red triangle was processed first, and then the blue triangle is processed next.

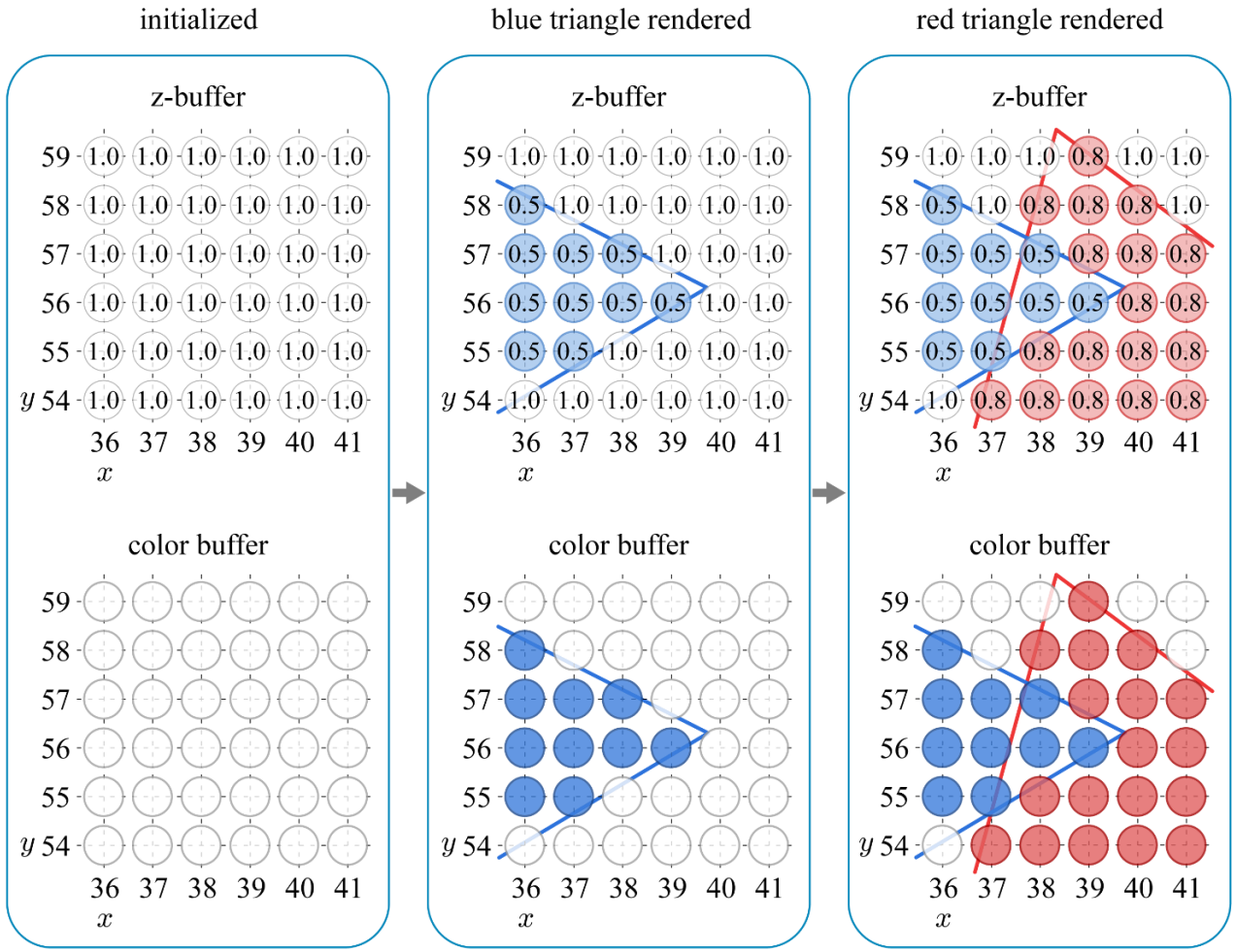




Depth Test

Example 2. In the same condition as example 1, blue triangle was processed first, and then the red triangle is processed next.





Stencil Test

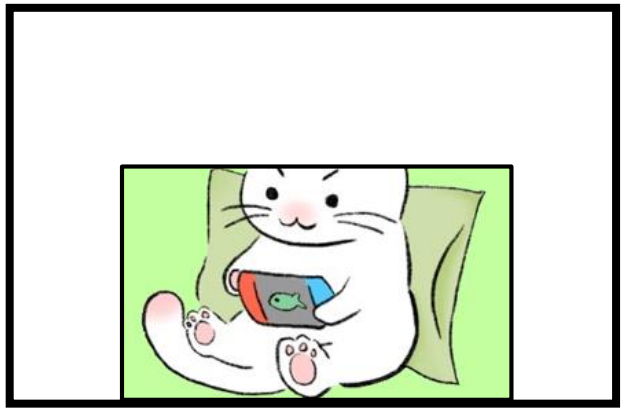
Stencil Testing

- A simple example of a stencil buffer is shown below.

Back buffer

0	0	0	0	0	0
0	0	0	0	0	0
0	1	1	1	1	0
0	1	1	1	1	0
0	1	1	1	1	0

Stencil buffer

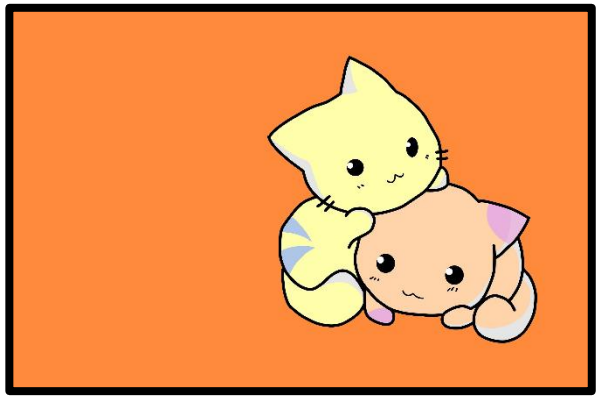


After stencil test

Practical use of stencil test

1) Transition effect

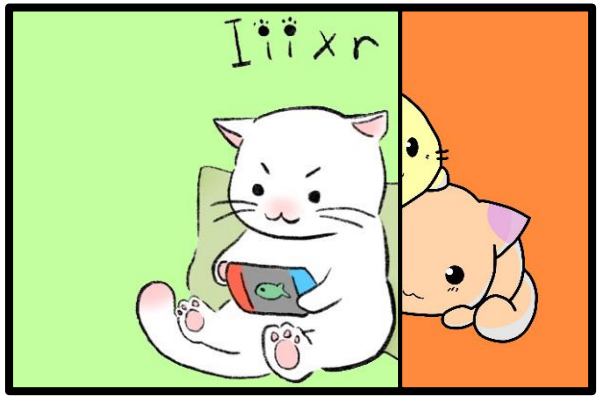
Previous Scene



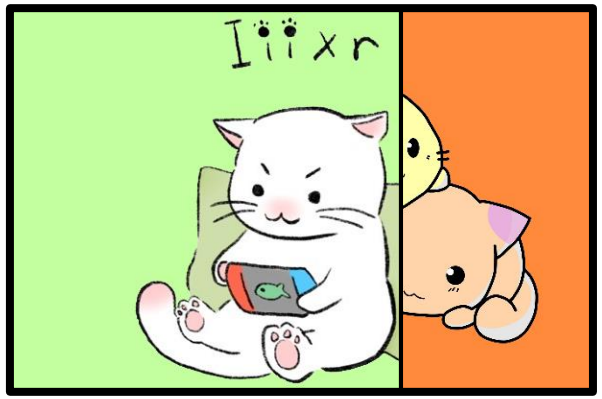
Back buffer

0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	0	1	1

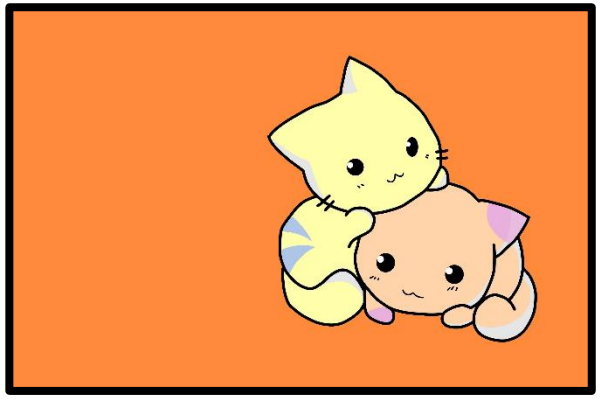
Stencil buffer



After stencil test



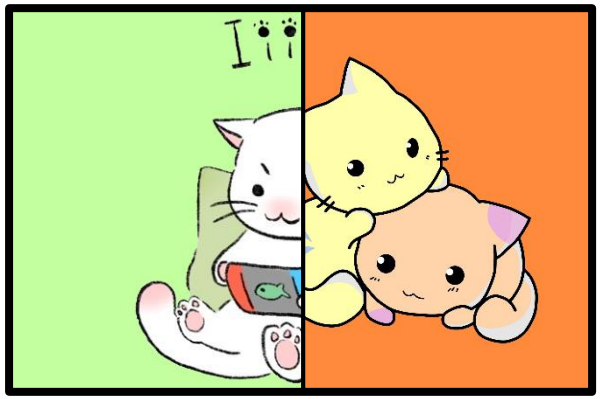
Previous Scene



Back buffer

0	0	0	1	0	0
0	0	0	1	0	0
0	0	0	1	0	0
0	0	0	1	0	0
0	0	0	1	0	0

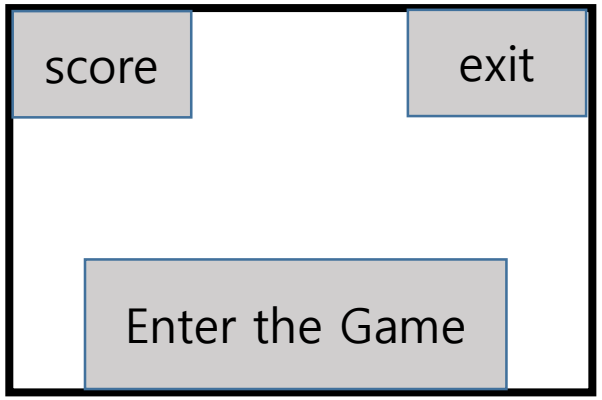
Stencil buffer



After stencil test

Practical use of stencil test

2) Discarding UI area (reducing the number of fragments to be processed)

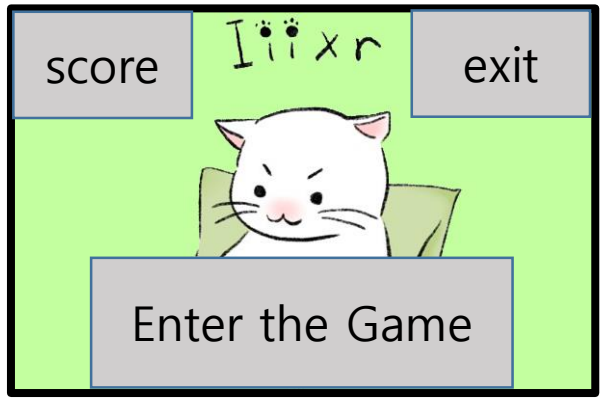


UI area

Back buffer

0	0	1	1	0	0
0	0	1	1	0	0
1	1	1	1	1	1
1	0	0	0	0	1
1	0	0	0	0	1

Stencil buffer



After stencil test

Color Blending

An object behind a translucent object should be visible.

- In the depth test, the pixel either replaces the pixel or is discarded.
- However, some surfaces may be translucent (partially transparent).
- Blending combines one or more pixel values to create a final pixel color.
- It is noteworthy that the blending operations are performed on every pixel shader output before the output value is written to a render target.

Color Blending

The blending process uses the alpha (α) value of the fragment.

- The alpha value represents 256 degrees of opacity: 0 denotes “fully transparent” and 255 denotes “fully opaque”.
- For the opacity representation, the normalized range [0, 1] is preferred to the integer range [0, 255].

The blending equation

- Let define c be the blended color, α be the pixel’s opacity, c_{src} be the color output from the pixel shader, and c_{dst} be the color of the back buffer.
- Without blending, c_{src} would overwrite c_{dst} and become the new color of the back buffer.
- With blending, c_{src} and c_{dst} are blended to get the combined color c.
- The blending equation is defined with source blend factor, f_{src}, and destination blend factor f_{dst}: $c = f_{src}c_{src} \blacksquare f_{dst}c_{dst}$.
- The $\blacksquare$ operator is defined by the blend operation setting.

Color Blending

The blend operations

- The ▣ operator is defined by D3D12_BLEND_OP.
  - ADD: c = $f_{src}c_{src} + f_{dst}c_{dst}$
  - SUBTRACT : c = $f_{dst}c_{dst} - f_{src}c_{src}$
  - REV_SUBTRACT: c = $f_{src}c_{src} - f_{dst}c_{dst}$
  - MIN: c = min (c_{src}, c_{dst})
  - MAX: c = max (c_{src}, c_{dst})

```
typedef enum D3D12_BLEND_OP {
  D3D12_BLEND_OP_ADD = 1,
  D3D12_BLEND_OP_SUBTRACT = 2,
  D3D12_BLEND_OP_REV_SUBTRACT = 3,
  D3D12_BLEND_OP_MIN = 4,
  D3D12_BLEND_OP_MAX = 5
} ;
```

Color Blending

The blend factors

- The blend factor is defined by D3D12_BLEND.
  - ZERO: $f = 0$
  - ONE: $f = 1$
  - SRC_ALPHA: $f = \alpha$ (alpha value from the pixel shader)
  - INV_SRC_ALPHA: $f = 1 - \alpha$

```
typedef enum D3D12_BLEND {
  D3D12_BLEND_ZERO = 1,
  D3D12_BLEND_ONE = 2,
  D3D12_BLEND_SRC_COLOR = 3,
  D3D12_BLEND_INV_SRC_COLOR = 4,
  D3D12_BLEND_SRC_ALPHA = 5,
  D3D12_BLEND_INV_SRC_ALPHA = 6,
  D3D12_BLEND_DEST_ALPHA = 7,
  D3D12_BLEND_INV_DEST_ALPHA = 8,
  D3D12_BLEND_DEST_COLOR = 9,
  D3D12_BLEND_INV_DEST_COLOR = 10,
  D3D12_BLEND_SRC_ALPHA_SAT = 11,
  D3D12_BLEND_BLEND_FACTOR = 14,
  D3D12_BLEND_INV_BLEND_FACTOR = 15,
  D3D12_BLEND_SRC1_COLOR = 16,
  D3D12_BLEND_INV_SRC1_COLOR = 17,
  D3D12_BLEND_SRC1_ALPHA = 18,
  D3D12_BLEND_INV_SRC1_ALPHA = 19,
  D3D12_BLEND_ALPHA_FACTOR,
  D3D12_BLEND_INV_ALPHA_FACTOR
} ;
```

Color Blending

Blend State

- In DX12, the blend state is set by pipeline state object (PSO).

```
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;
  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;
```

Color Blending

Blend State

- The blending is disabled in default.
- To enable blending, we must configure D3D12_BLEND_DESC correctly.

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

```
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;
```

Color Blending

Blend State

- The following is the sample code of creating a blend state:

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC samplePsoDesc;
ZeroMemory(& samplePsoDesc, sizeof(D3D12_GRAPHICS_PIPELINE_STATE_DESC));

D3D12_RENDER_TARGET_BLEND_DESC sampleBlendDesc;
sampleBlendDesc.BlendEnable = true;
sampleBlendDesc.LogicOpEnable = false;
sampleBlendDesc.SrcBlend = D3D12_BLEND_SRC_ALPHA;
sampleBlendDesc.DestBlend = D3D12_BLEND_INV_SRC_ALPHA;
sampleBlendDesc.BlendOp = D3D12_BLEND_OP_ADD;
sampleBlendDesc.SrcBlendAlpha = D3D12_BLEND_ONE;
sampleBlendDesc.DestBlendAlpha = D3D12_BLEND_ZERO;
sampleBlendDesc.BlendOpAlpha = D3D12_BLEND_OP_ADD;
sampleBlendDesc.LogicOp = D3D12_LOGIC_OP_NOOP;
sampleBlendDesc.RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;
samplePsoDesc.BlendState.RenderTarget[0] = sampleBlendDesc;
```

Color Blending

Keeping Destination Pixel Example

- Suppose that you want to keep the destination pixel and not overwrite it.
- Then, you would set the source pixel blend factor to D3D12_BLEND_ZERO, the destination blend factor to D3D12_BLEND_ONE, and the blend operator to D3D12_BLEND_OP_ADD.
- The blending equation would be as follows:

  $\rightarrow c = f_{src}c_{src} + f_{dst}c_{dst}$

  $\rightarrow c = 0 * c_{src} + 1 * c_{dst}$

  $\rightarrow c = c_{dst}$

Color Blending

Alpha Blending (Transparency)

- For alpha blending, the source alpha component would be thought as a percent that controls the opacity of the source pixel.
- The most widely setting for alpha blending is as follows:
  - Source blend factor = D3D12_BLEND_SRC_ALPHA
  - Destination blend factor = D3D12_BLEND_INV_SRC_ALPHA
  - Blend Operator = D3D12_BLEND_OP_ADD
- With this setting, the blending equation would be as follows :

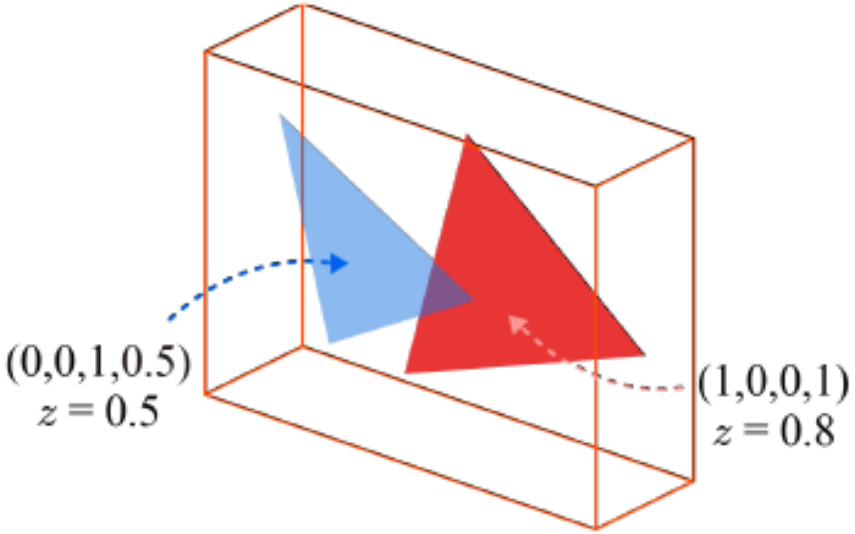
  $\rightarrow c = f_{src}c_{src} + f_{dst}c_{dst}$

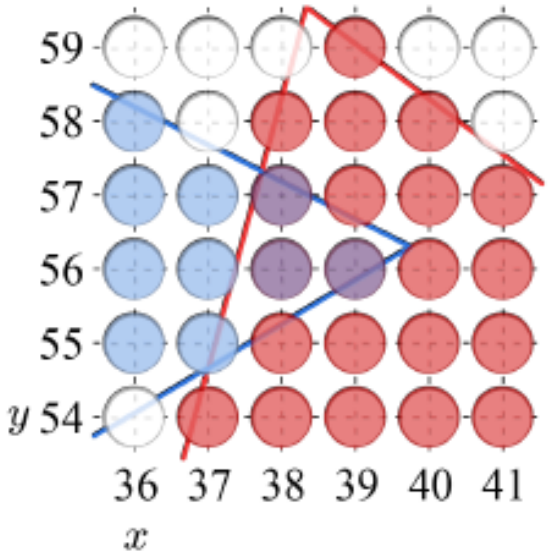
  $\rightarrow c = \alpha_s * c_{src} + (1 - \alpha_s) * c_{dst}$

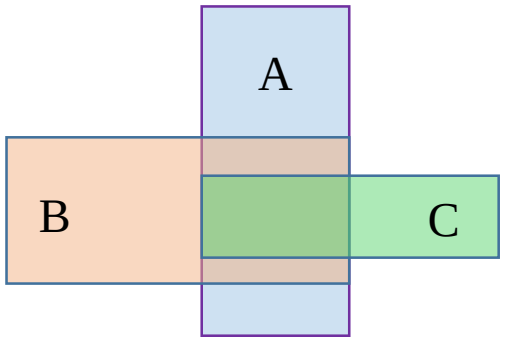
Color Blending

For alpha blending (or color blending), the primitives cannot be rendered in an arbitrary order. They must be rendered *after* all opaque primitives, and in *back-to-front* order. Therefore, the partially transparent objects should be *sorted*.

$$c = \alpha_s * c_{src} + (1 - \alpha_s) * c_{dst}$$







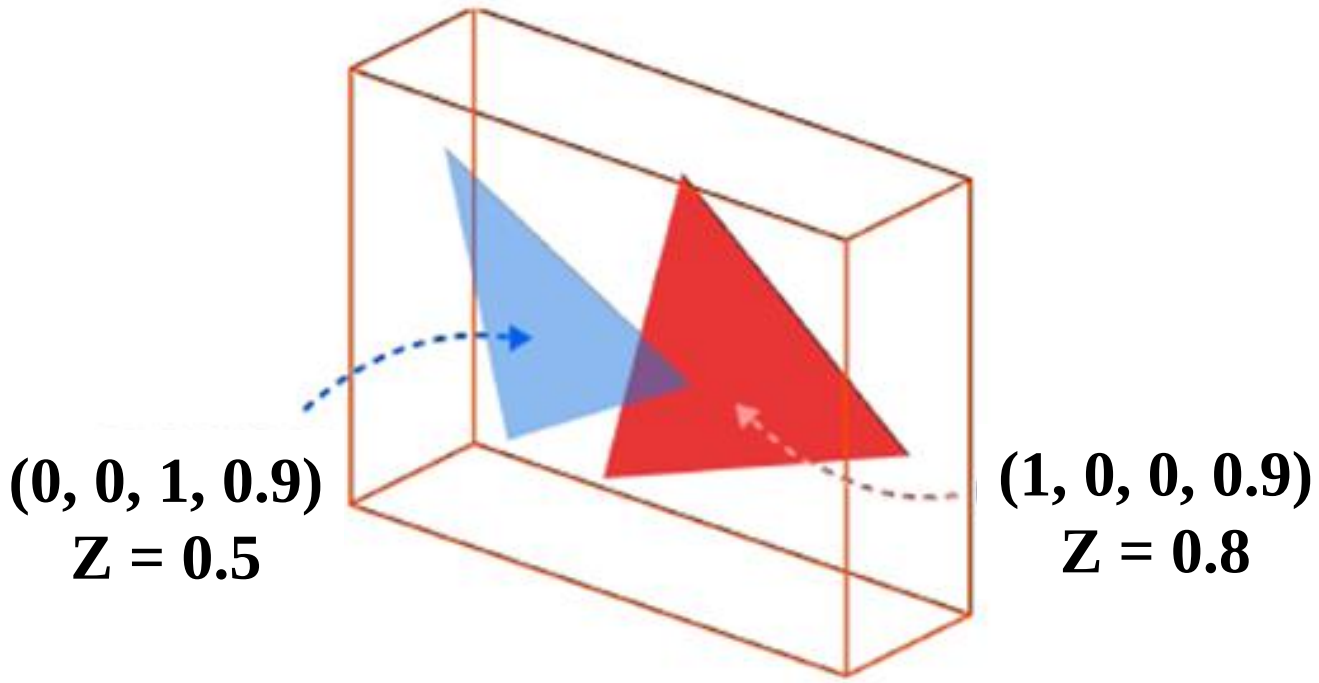
A < B < C
back *front*

Color Blending

Back-to-front vs. front-to-back.

- (0, 0, 1) ∗ 0.9 + 0.1 ∗ (1, 0, 0) = (0.1, 0, 0.9). (back-to-front)
- (1, 0, 0) ∗ 0.9 + 0.1 ∗ (0, 0, 1) = (0.9, 0, 0.1). (front-to-back)

$$c = \alpha_s * c_{src} + (1 - \alpha_s) * c_{dst}$$



Practice

Consider five pixels competing for a pixel location. Their RGBA colors and z-coordinates are given as follows:

- f_1 = {(1, 0, 0, 0.5), 0.2}
- f_2 = {(0, 1, 1 , 0.5), 0.4}
- f_3 = {(0, 0, 1, 1), 0.6}
- f_4 = {(1, 0, 1, 0.5), 0.8}

1. What is the correct order of processing the pixels?

2. Compute the final color of the pixel using the equation $c = \alpha_s * c_{src} + (1 - \alpha_s) * c_{dst}$.

Practice - Solution

Consider five pixels competing for a pixel location. Their RGBA colors and z-coordinates are given as follows:

- f_1 = {(1, 0, 0, 0.5), 0.2}
- f_2 = {(0, 1, 1 , 0.5), 0.4}
- f_3 = {(0, 0, 1, 1), 0.6}
- f_4 = {(1, 0, 1, 0.5), 0.8}

1. What is the correct order of processing the pixels?
   - $f_3 \rightarrow f_4 \rightarrow f_2 \rightarrow f_1$

2. Compute the final color of the pixel using the equation $c = \alpha_s * c_{src} + (1 - \alpha_s) * c_{dst}$.
   - f_4 can be skipped.
   - f_3 and f_2: 0.5 * (0, 1, 1) + 0.5 * (0, 0, 1) = (0, 0.5 , 1)
   - then, f_1: 0.5 * (1, 0 ,0) + 0.5 * (0, 0.5, 1) = (0.5, 0.25, 0.5)