GRAPHICS PROGRAMMING I

HARDWARE RASTERIZATION PART III

DirectX: Shading

- Last week we rendered our mesh with just a diffuse texture. This week we'll do our typical shading:
  - Diffuse Color
  - Normal Mapping
  - Specular Color (Phong)
- Enable the parsing of normals and tangents in your .obj parser again. This also means you must adjust your vertex struct on both the CPU and GPU side, including adjusting the input layout! Hint: use the semantics NORMAL and TANGENT.

DirectX: Shading

- Let's start the shading process!
- Create a pixel shader that has the same functionality as the PixelShading function of your software rasterizer, including specular using the Phong method.
- Just as previous week, you'll have to provide some extra information to your shader in order to be able to do the proper rendering. The following additional information is needed:
    - Normal Map : Texture2D
    - Specular Map : Texture2D
    - Glossiness Map : Texture2D
    - Light Direction : float3 → for the moment you can hardcode this using: { 0.577f, -0.577f, 0.577f }
    - World Matrix : float4x4
    - View Inverse Matrix : float4x4
- There are also some other pieces of information you might want to define as a variable:
    - PI
    - Light Intensity (7.0f)
    - Shininess (25.0f)

DirectX: Shading

- Hold up, why do we need the World matrix?
  - We need to transform our normal and tangents with the World matrix, NOT the WorldViewProjection! We also transform them in the vertex shader.
  - We pass it as a float4x4, but we only need the rotation part + our normals and tangents are of the type float3. We can extract the rotation part of a float4x4 as follows:

```
mul(normalize(someFloat3), (float3x3)someFloat4x4Matrix)
```

- So why do we need the ViewInverse matrix? Can you think of another name for the ViewInverse?
  - The ViewInverse is just the ONB/World of our camera. Remember, we use the inverse of the ONB to create the view matrix!
  - The ONB of the camera holds the position of the camera in the last row. We also have the world matrix, which defines the position of the model in the world. If we transform our position with the world matrix in the vertex shader and we store the result in an extra parameter that will be interpolated in the rasterizer, we can calculate the view direction for every pixel! ☺

DirectX: Shading

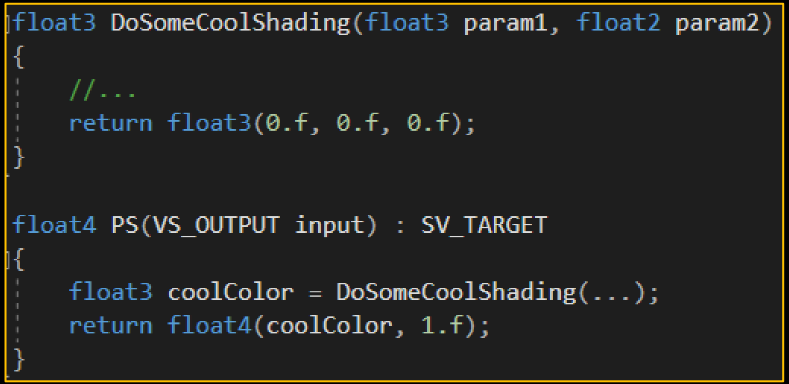
- Add the two matrices in the shader, using the semantics WORLD and VIEWINVERSE.
- Capture the variables on the CPU side and before rendering, update the GPU variables using the appropriate ID3DX11EffectMatrixVariable.
- Transform the incoming position (defined in model space) with the world matrix and store it in an additional variable in the vertex output struct.
- In the pixel shader, use the interpolated world position of the pixel and the ONB of the camera to calculate the view direction.

```
struct VS_OUTPUT
{
    float4 Position         : SV_POSITION;
    float4 WorldPosition    : COLOR;
    float3 Normal           : NORMAL;
    float3 Tangent          : TANGENT;
    float2 TexCoord         : TEXCOORD;
};
```

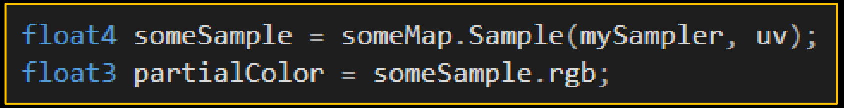
```
float3 viewDirection = normalize(input.WorldPosition.xyz - gViewInverse[3].xyz);
```

DirectX: Shading

- Implement the shading effects as you did in your software rasterizer. Below some useful functions and tips:
  - You can write separate functions that you can call in the shader functions.
  - Use the intrinsic functions to do the necessary math:
    - cross → beware **handedness**!
    - dot
    - saturate → clamp to range [0, 1]
    - reflect
    - normalize
    - https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-intrinsic-functions
  - Use the swizzling functionality of HLSL
    - https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx9-graphics-reference-asm-ps-registers-modifiers-source-register-swizzling



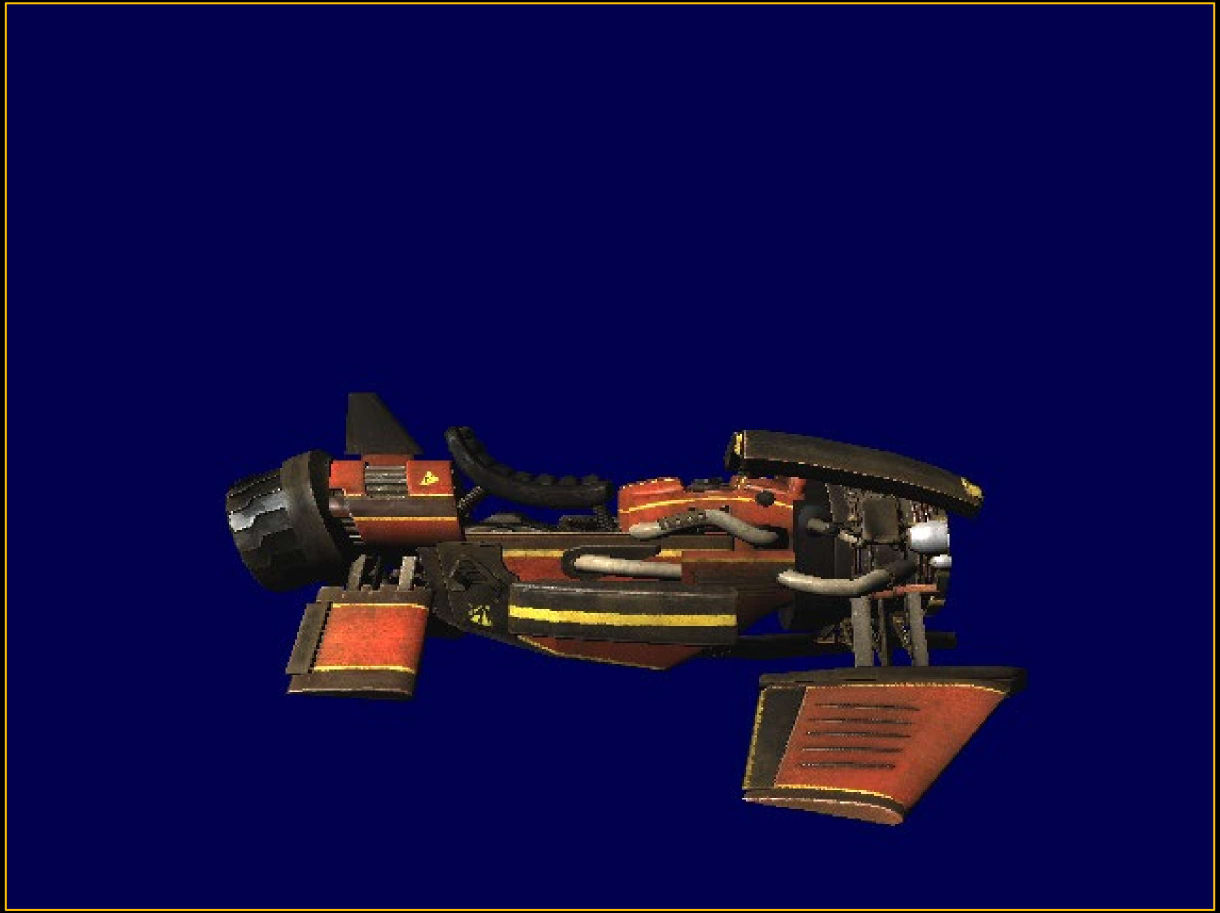
```
float3 DoSomeCoolShading(float3 param1, float2 param2)
{
    //...
    return float3(0.f, 0.f, 0.f);
}

float4 PS(VS_OUTPUT input) : SV_TARGET
{
    float3 coolColor = DoSomeCoolShading(...);
    return float4(coolColor, 1.f);
}
```



```
float4 someSample = someMap.Sample(mySampler, uv);
float3 partialColor = someSample.rgb;
```

DirectX: Shading

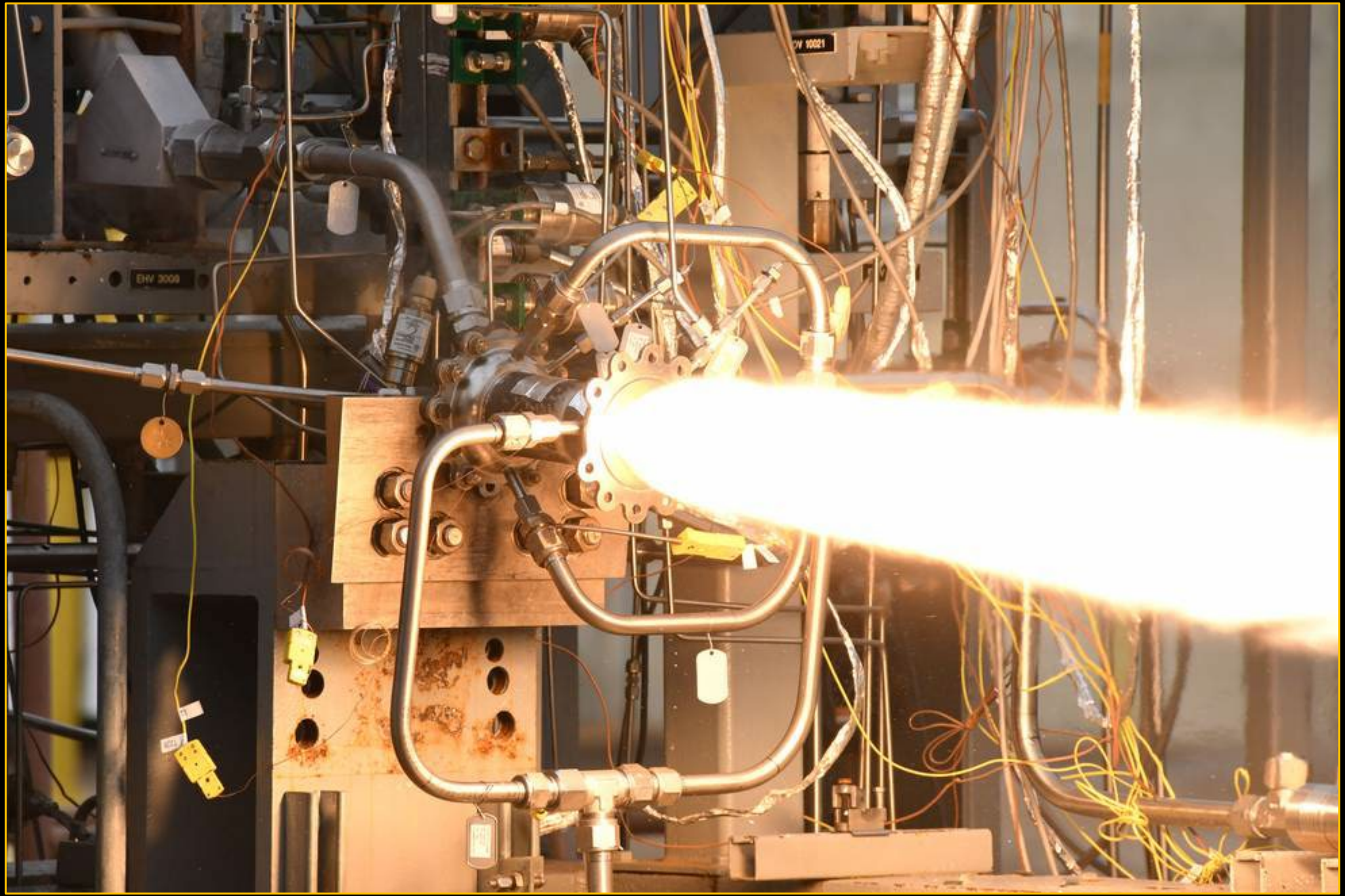
- With the correct pixel shader and rasterizer state, you should get a similar result.



DirectX: Shading

DirectX: Transparency

- Let’s make our demo even better by adding a cool combustion effect!



DirectX: Transparency

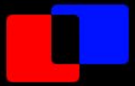
- If you take a closer look to the previous image, you'll see that at the edges of the flames, we can see the background. Unlike our actual vehicle, there are no "hard edges".
- In order to create a similar effect and also have a shape that is not the same as our actual mesh, we need to use transparency.
- Transparency is the root of a lot of evil in the rendering pipelines of game engines. A lot of techniques require special passes in order to render meshes correctly. There are a lot of potential issues, so I won't sum them up. ☺
- Have a look at the following video by Morgan McGuire: https://www.youtube.com/watch?v=rVh-tnsJv54

DirectX: Transparency

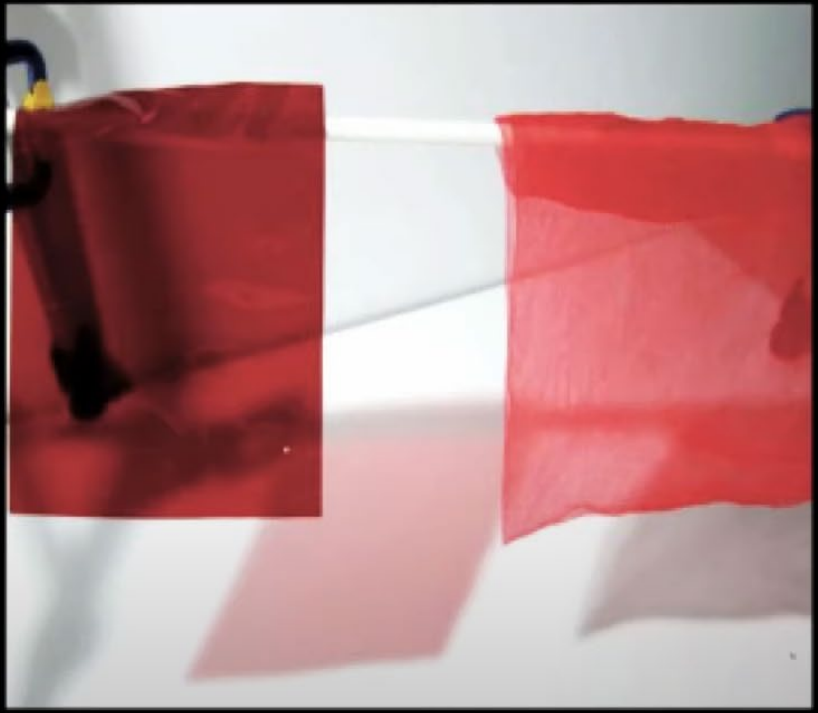
Transmission

(Light passing through medium)

Refraction, diffusion, extinction, chromatic abberation, colored shadows, colored background, etc.



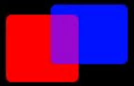
Red over Blue = Black



Partial Coverage

(Varying binary coverage within a pixel)

Gray shadow. No diffusion or refraction. No coloring of background. Includes depth of field and motion blur.



Red over Blue = Purple

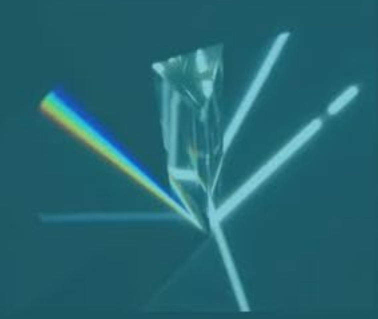
Transmission

Partial Coverage

Transparent Shadows

Multiple Scattering

Colored Transmission


Chromatic Abberation

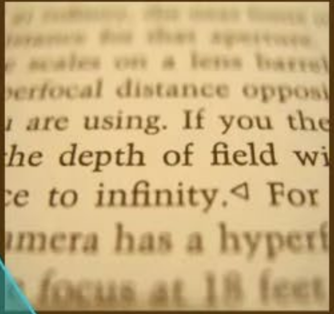
Caustics

Rainbows

Refraction

Transparent Emissives

Volumetric Light & Shadow


Depth of Field

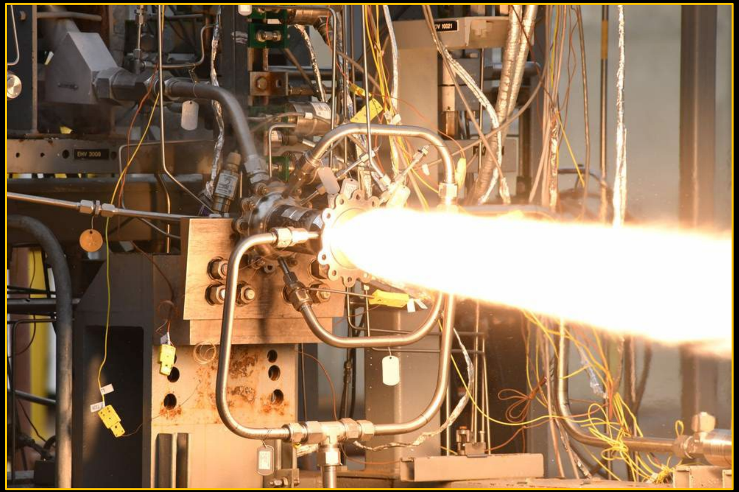
Motion Blur

Thin &Perforated Objects

*All images from Wikimedia Commons except * marketing image from http://www.dachauer-spezialglas.de/praezisionsoptik/optische-filt*

DirectX: ~~Transparency~~ Partial Coverage

- If you take a closer look to the previous image, you'll see that at the edges of the flames, we can see the background. Unlike our actual vehicle, there are no "hard edges".
- In order to create a similar effect and have a shape that is not the same as our actual mesh, we need to use transparency.
- Transparency is the root of a lot of evil in the rendering pipelines of game engines. A lot of techniques require special passes in order to render meshes correctly. There are a lot of potential issues, so I won't sum them up. ☺
- Have a look at the following video by Morgan McGuire: https://www.youtube.com/watch?v=rVh-tnsJv54

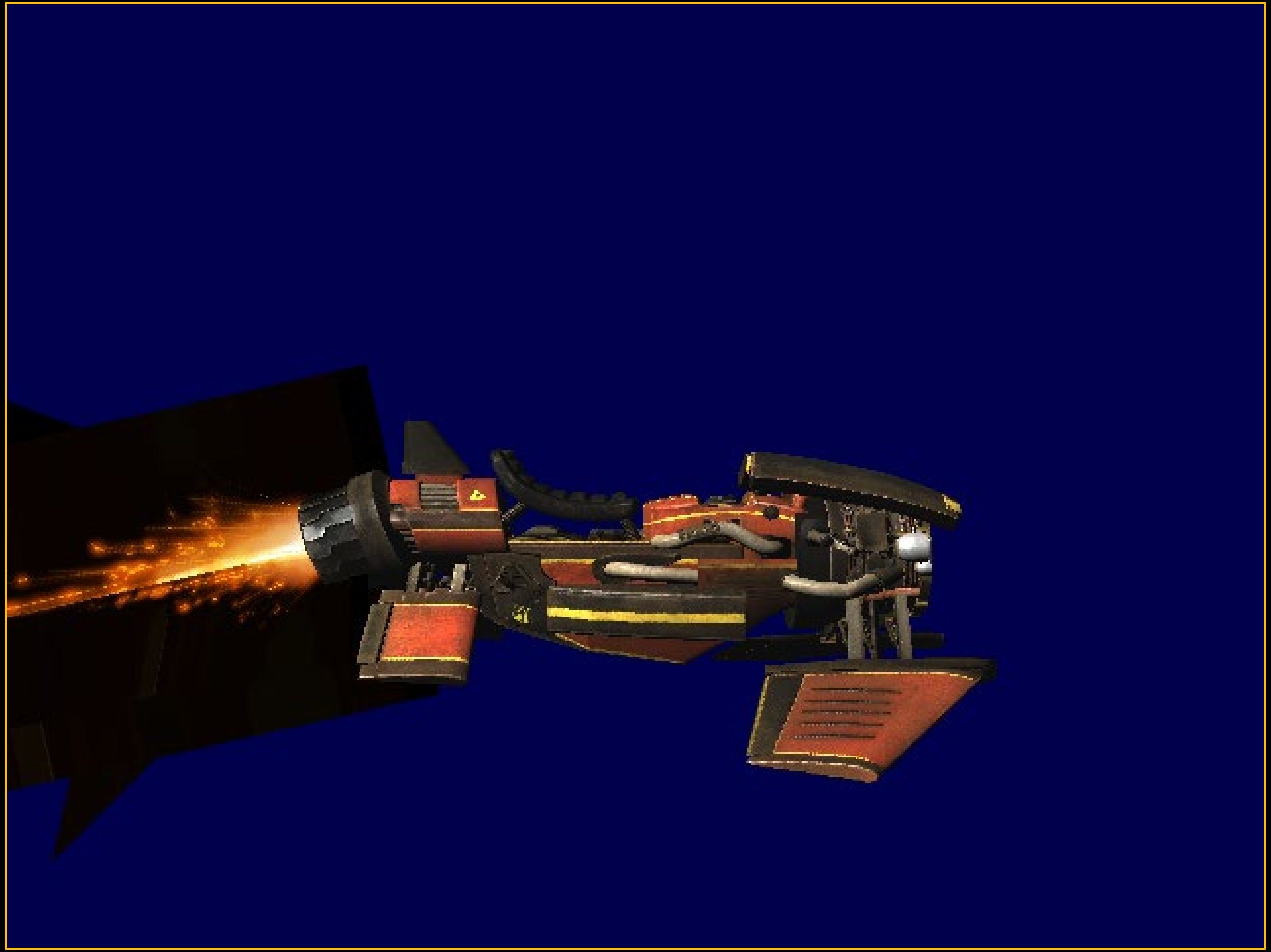


- Thus, transparency consist out of two categories:
  - Transmission
  - Partial Coverage
- So, we are looking at partial coverage!!!

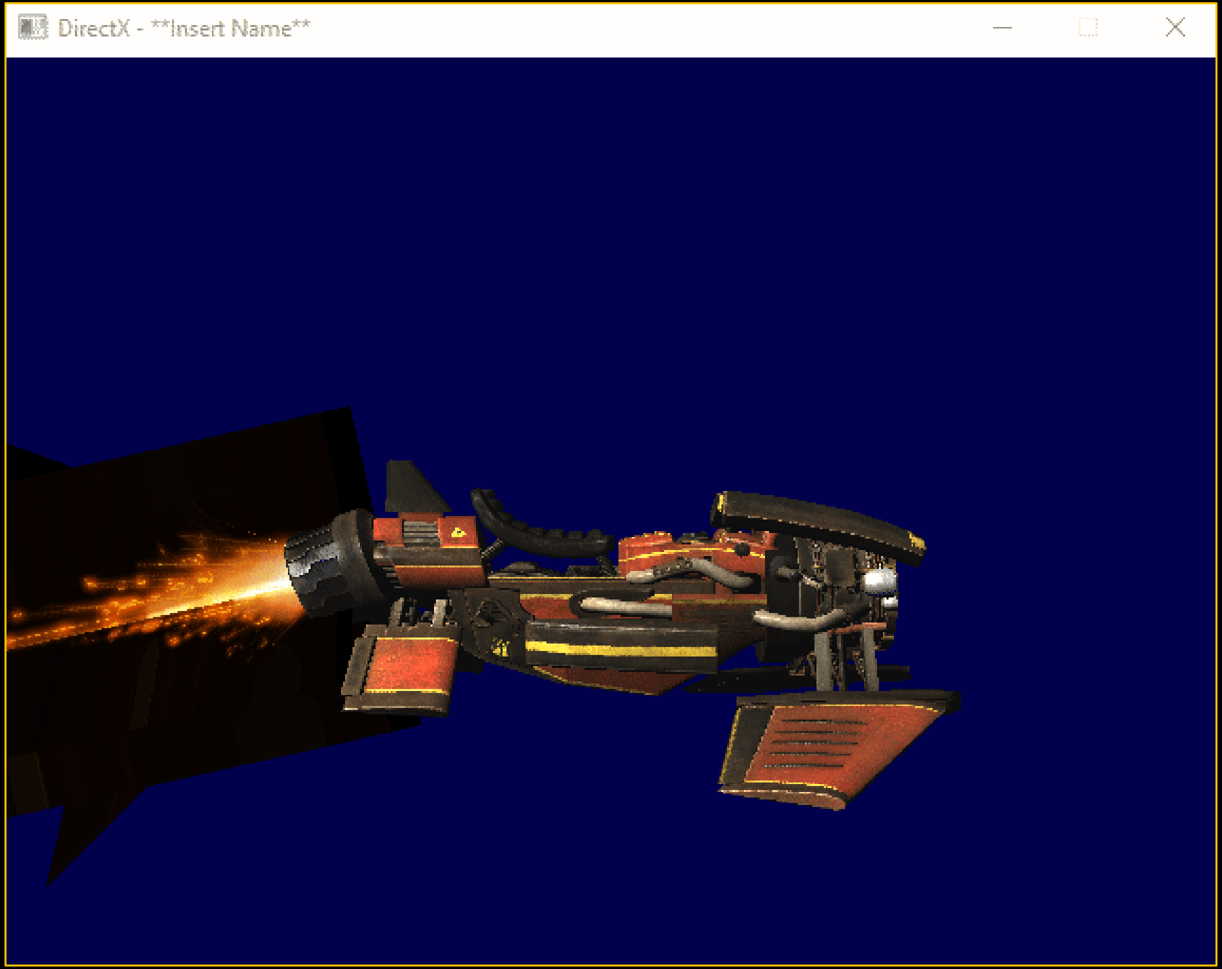
DirectX: Partial Coverage

- Let's start by making a new shader and effect class. Why?
  - We don't want any fancy shading for the fire effect. We want flat shading. Creating a new shader and effect class allows us to skip the part where we push texture data to the GPU and it allows us to write a very simple vertex and pixel shader.
  - It's smart to write a base effect class that loads and compiles the shader. It also updates the WorldViewProjection matrix, because "all" shaders need this!
  - You can then inherit from this effect and add more variables to it.
- Our flat shading effect will just read a value of a diffuse map and return that value, including the alpha channel value!
- Once you've made the new shader and effect class, load in the new .obj and the associated diffuse map.
- Some terminology that might cause confusion:
  - Shader → sometimes called effect. It's the code that runs on the GPU.
  - Effect → often called material. It holds the data for one particular shader, as well as the associated resource variables and/or views. It is also responsible for updating the shader variables.

DirectX: Partial Coverage



DirectX: Partial Coverage

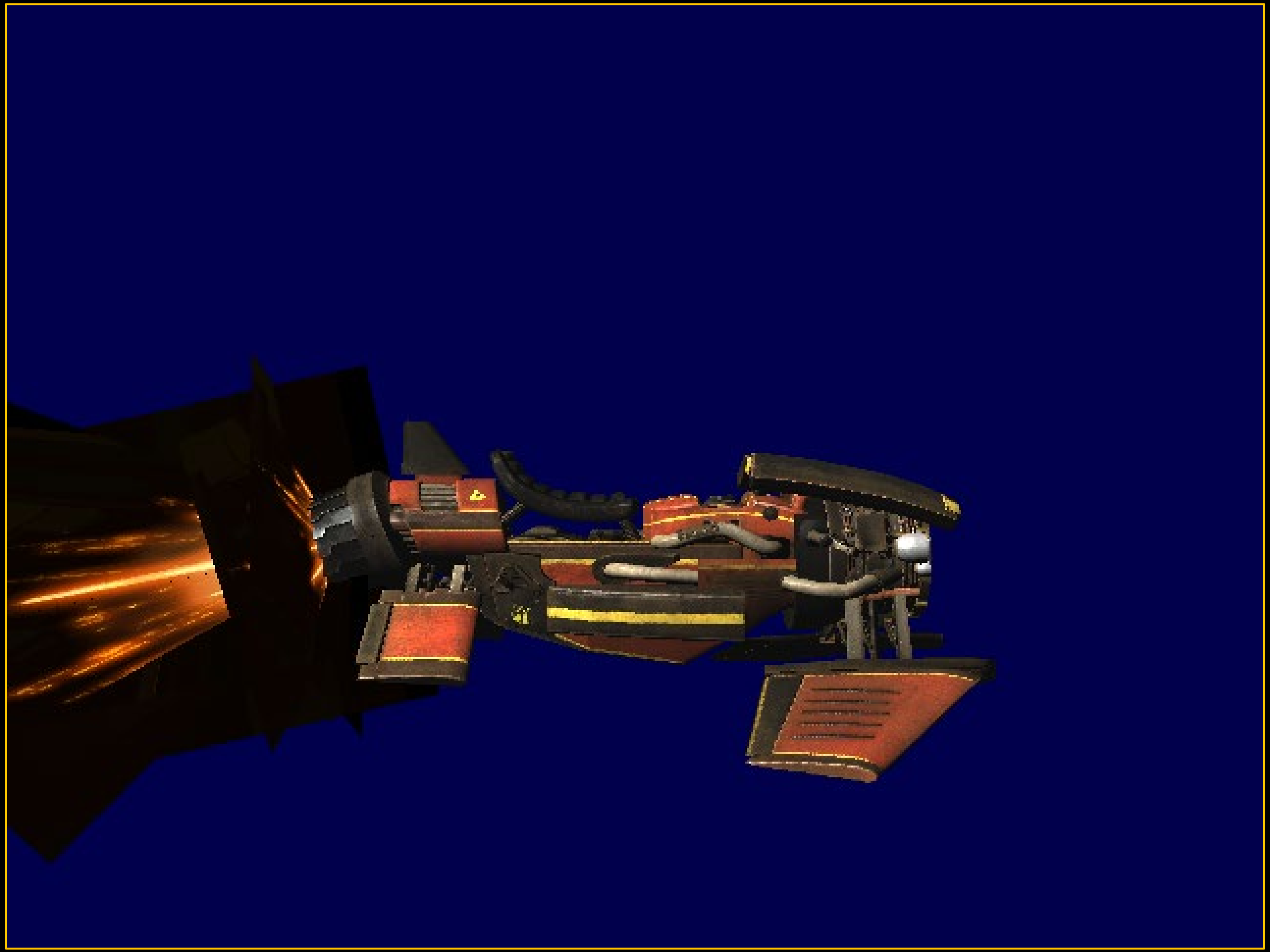


DirectX: Partial Coverage

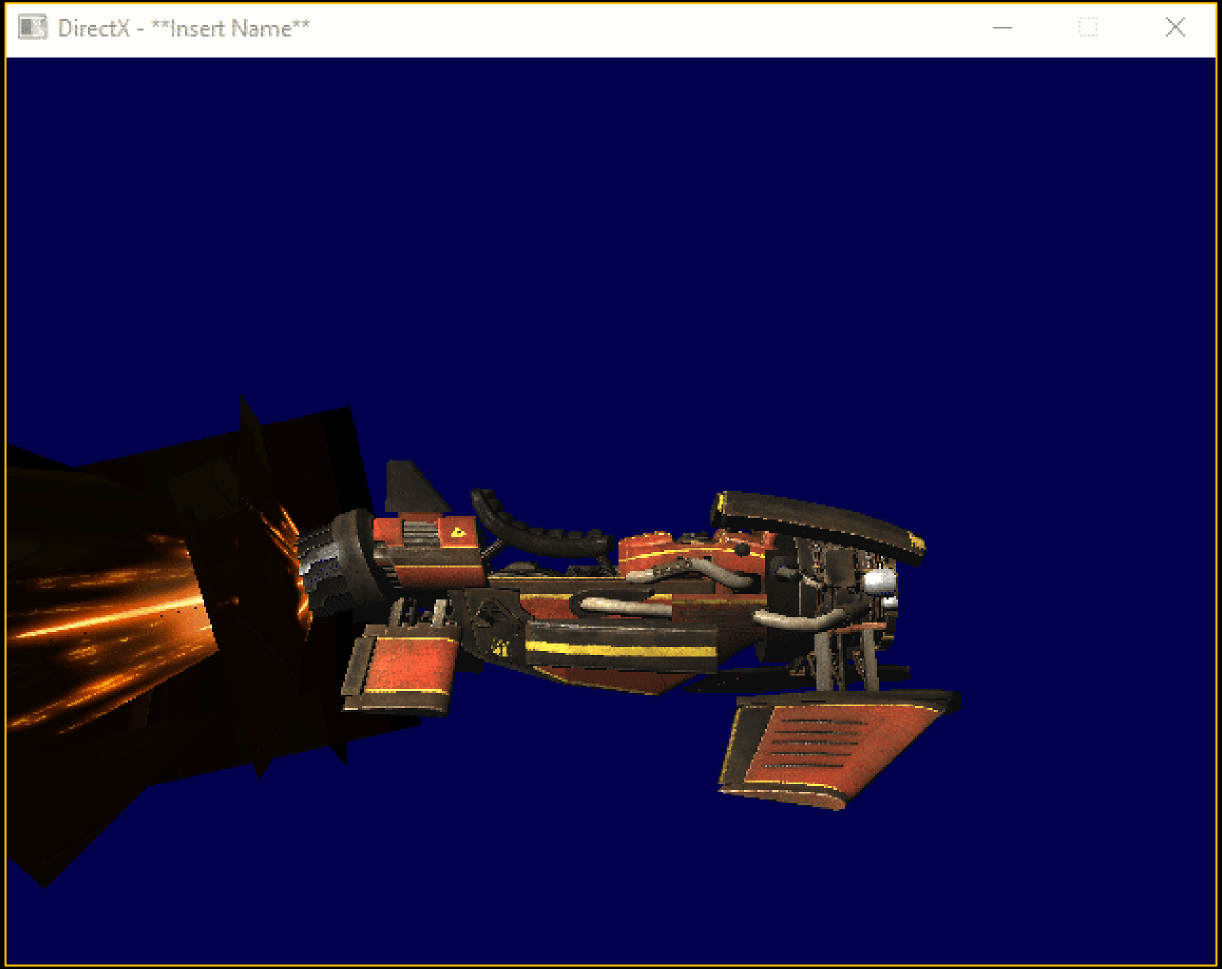
- Ok there are a few problems here. Let's start with the first one.
- As you can maybe notice, there are bits and pieces of information missing. There are planes that are there, but that are not visible. Why?
  - Back-face culling! In this case, we want a double-sided effect, which means we want to be able to see the planes from all angles.
  - Fix this by adding a rasterizer state that disables back-face culling.

```
RasterizerState gRasterizerState
{
    CullMode = none;
    FrontCounterClockwise = false; //default
};
```

DirectX: Partial Coverage

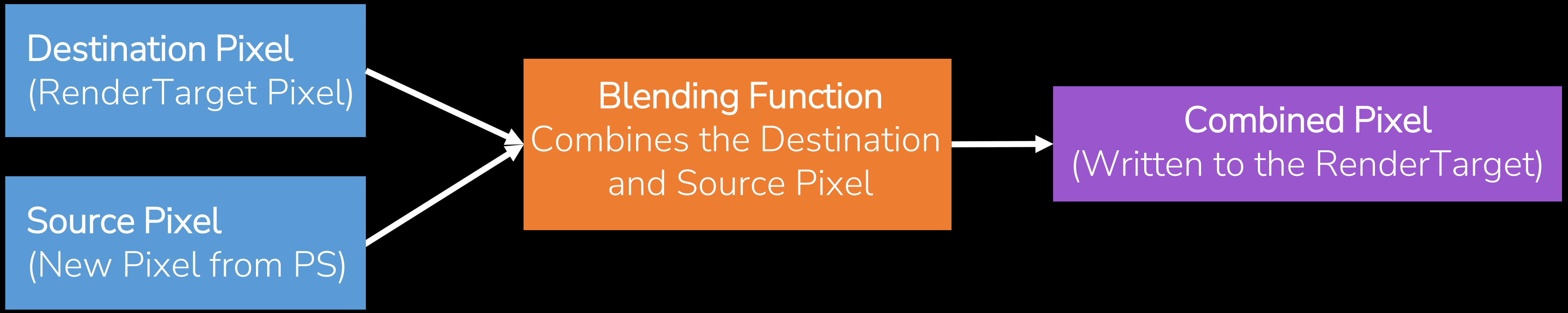


DirectX: Partial Coverage

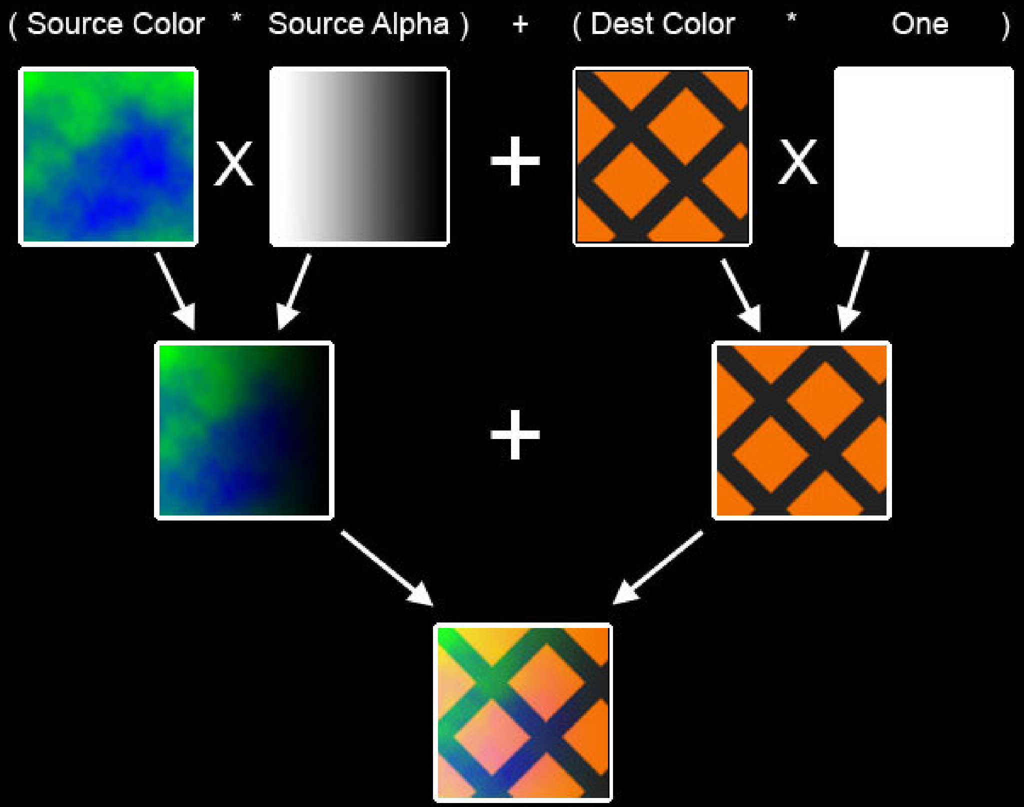


DirectX: Partial Coverage

- That's already better, but even though we are returning our alpha value, there is no "transparency". Why?
- We didn't tell the pipeline it has to blend the returned value with the one that is already in the backbuffer!
- So how does this blending work?
  - Every pixel that survives the stencil and depth test, is passed to the blending function in the output merger stage.



DirectX: Partial Coverage



Result = Color_{src} x BlendFactor_{src} + Color_{dst} x BlendFactor_{dst}

DirectX: Partial Coverage

$$\boxed{Result = Color_{src} \times BlendFactor_{src} + Color_{dst} \times BlendFactor_{dst}}$$

- The blend function consist out of three important parts:
  - Colors:
    - Source → Pixel value from the Pixel Shader
    - Destination → Pixel value in the Backbuffer (RenderTarget)
  - BlendFactors for both the source and destination value (always multiplied).

```
typedef enum D3D11_BLEND {
  D3D11_BLEND_ZERO             = 1,
  D3D11_BLEND_ONE              = 2,
  D3D11_BLEND_SRC_COLOR        = 3,
  D3D11_BLEND_INV_SRC_COLOR    = 4,
  D3D11_BLEND_SRC_ALPHA        = 5,
  D3D11_BLEND_INV_SRC_ALPHA    = 6,
  D3D11_BLEND_DEST_ALPHA       = 7,
  D3D11_BLEND_INV_DEST_ALPHA   = 8,
  D3D11_BLEND_DEST_COLOR       = 9,
  D3D11_BLEND_INV_DEST_COLOR   = 10,
  D3D11_BLEND_SRC_ALPHA_SAT    = 11,
  D3D11_BLEND_BLEND_FACTOR     = 14,
  D3D11_BLEND_INV_BLEND_FACTOR = 15,
  D3D11_BLEND_SRC1_COLOR       = 16,
  D3D11_BLEND_INV_SRC1_COLOR   = 17,
  D3D11_BLEND_SRC1_ALPHA       = 18,
  D3D11_BLEND_INV_SRC1_ALPHA   = 19
} D3D11_BLEND;
```

DirectX: Partial Coverage

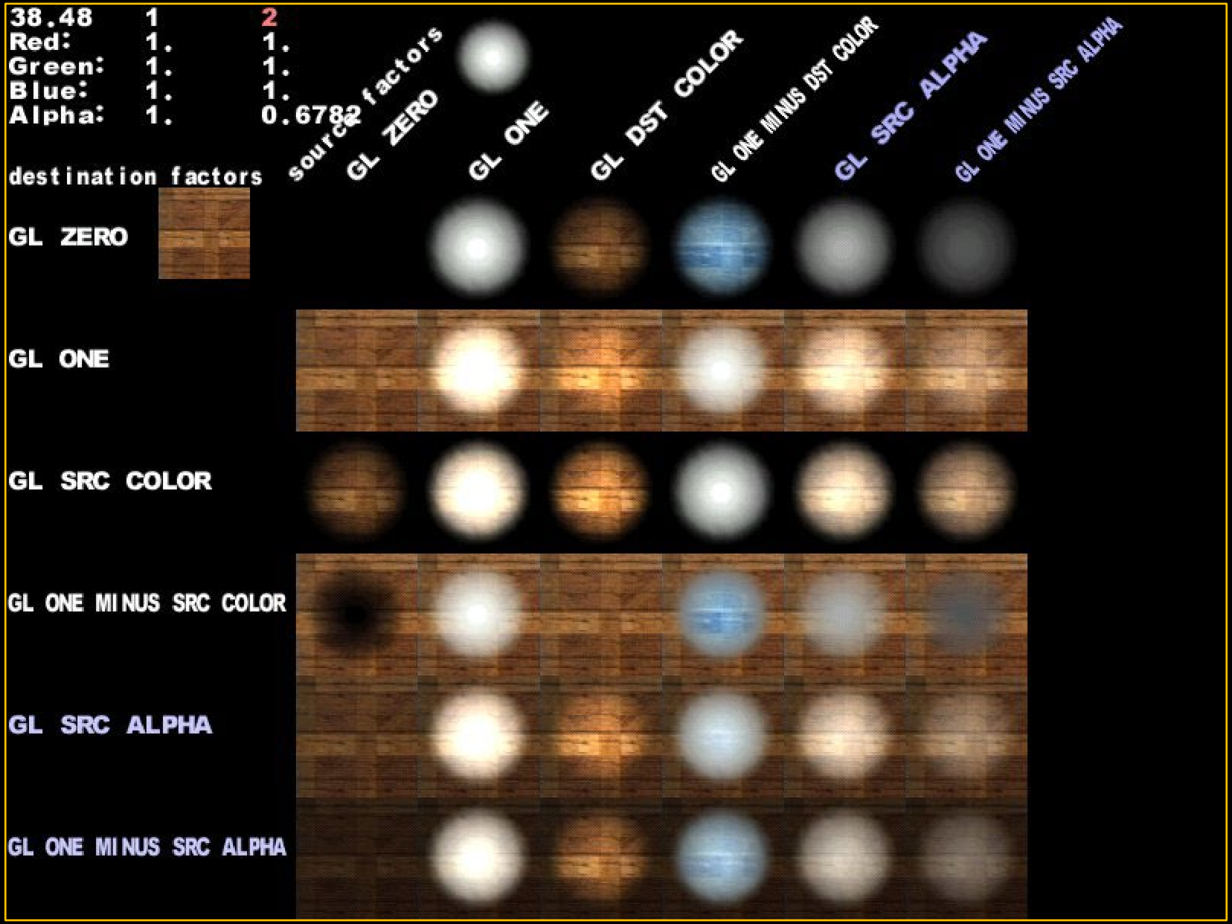
$$\boxed{\text{Result} = \text{Color}_{src} \times \text{BlendFactor}_{src} + \text{Color}_{dst} \times \text{BlendFactor}_{dst}}$$

- The blend function consist out of three important parts:
  - Colors:
    - Source → Pixel value from the Pixel Shader
    - Destination → Pixel value in the Backbuffer (RenderTarget)
  - BlendFactors for both the source and destination value (always multiplied).
  - Blend Operation

```
typedef enum D3D11_BLEND_OP {
  D3D11_BLEND_OP_ADD          = 1,
  D3D11_BLEND_OP_SUBTRACT     = 2,
  D3D11_BLEND_OP_REV_SUBTRACT = 3,
  D3D11_BLEND_OP_MIN          = 4,
  D3D11_BLEND_OP_MAX          = 5
} D3D11_BLEND_OP;
```

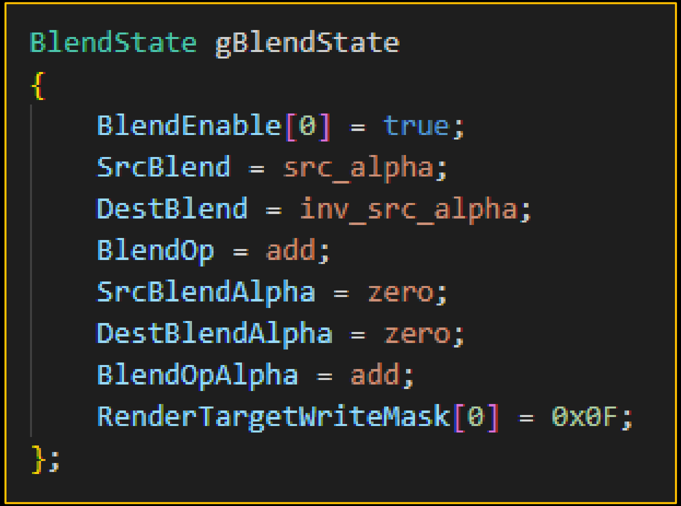
DirectX: Partial Coverage

- Using those adjustable parameters, you can create a lot of blending effects!

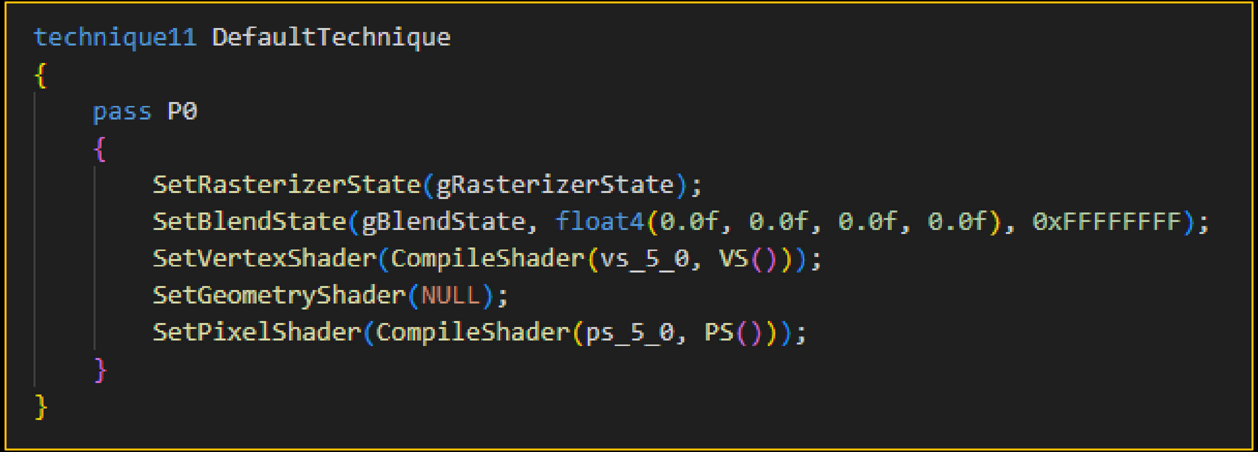


DirectX: Partial Coverage

- So how do we now set the rendering pipeline to blend our returned value?
- Just as with the rasterizer state you can set it on the CPU side and push it to the GPU, or you define it in the shader itself. But instead of a rasterizer state, we use a **Blend State**.



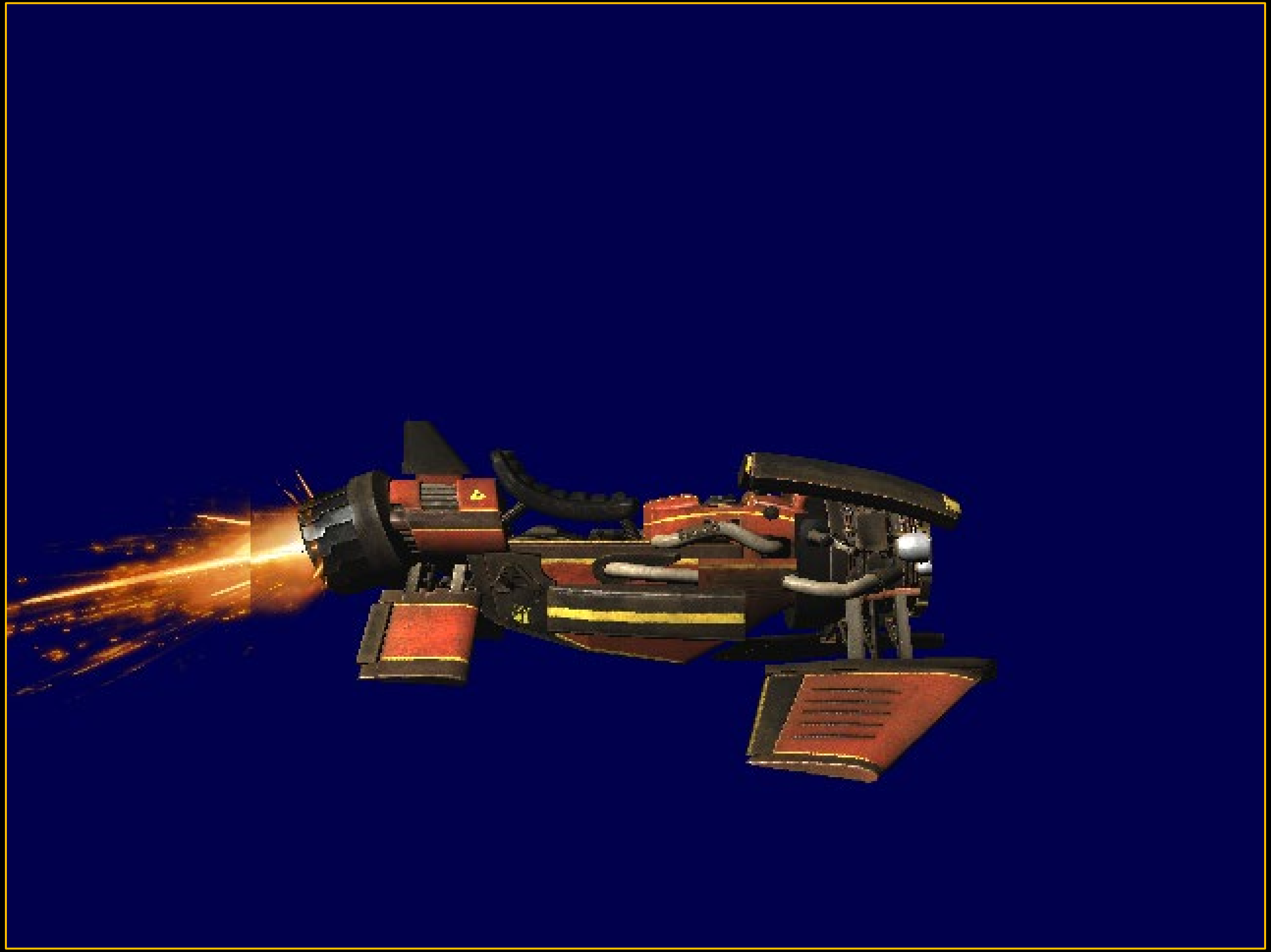
```
BlendState gBlendState
{
    BlendEnable[0] = true;
    SrcBlend = src_alpha;
    DestBlend = inv_src_alpha;
    BlendOp = add;
    SrcBlendAlpha = zero;
    DestBlendAlpha = zero;
    BlendOpAlpha = add;
    RenderTargetWriteMask[0] = 0x0F;
};
```



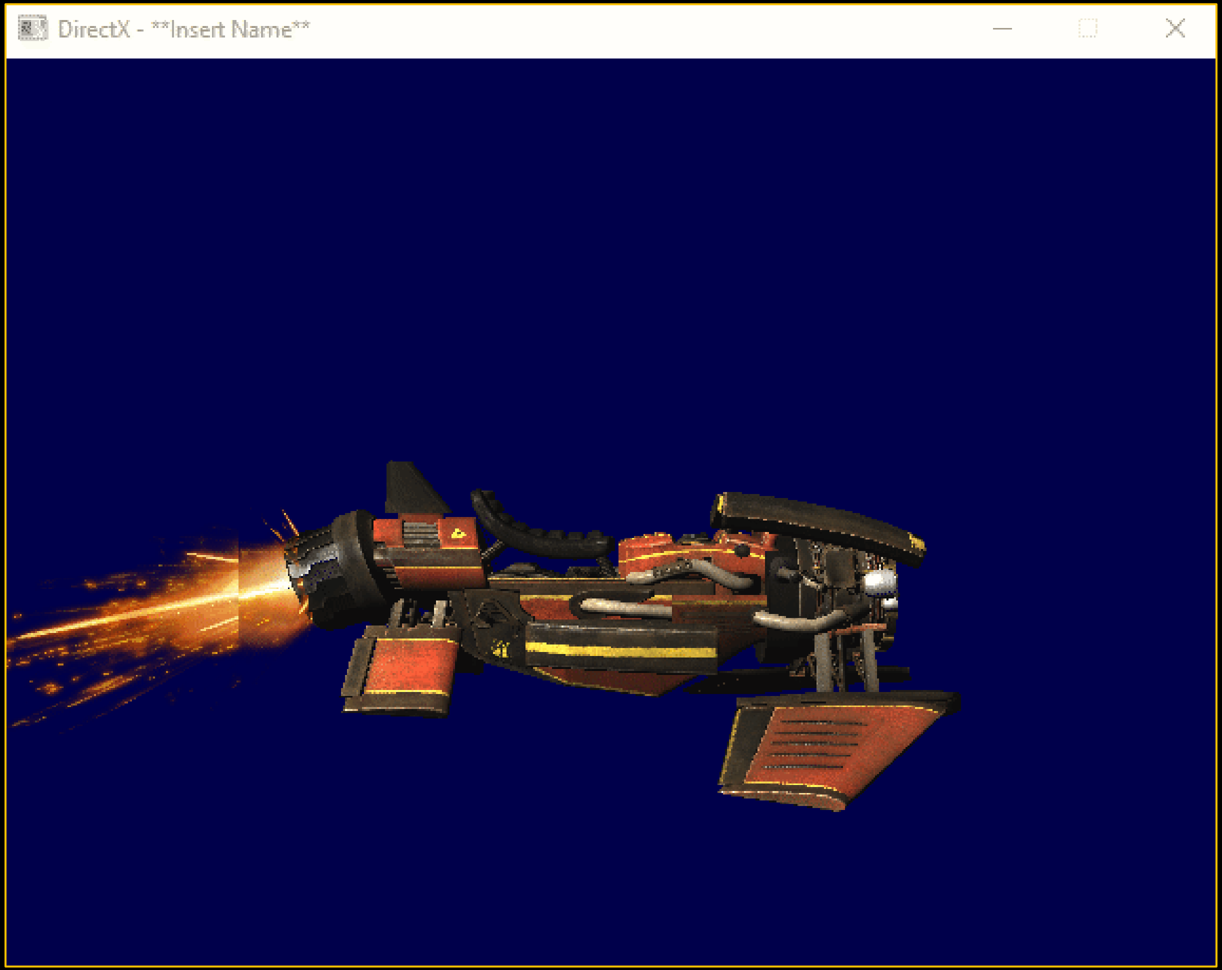
```
technique11 DefaultTechnique
{
    pass P0
    {
        SetRasterizerState(gRasterizerState);
        SetBlendState(gBlendState, float4(0.0f, 0.0f, 0.0f, 0.0f), 0xFFFFFFFF);
        SetVertexShader(CompileShader(vs_5_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, PS()));
    }
}
```

- To check what each option does, use MSDN!
  https://docs.microsoft.com/en-us/windows/win32/api/d3d11/ns-d3d11-d3d11_render_target_blend_desc

DirectX: Partial Coverage

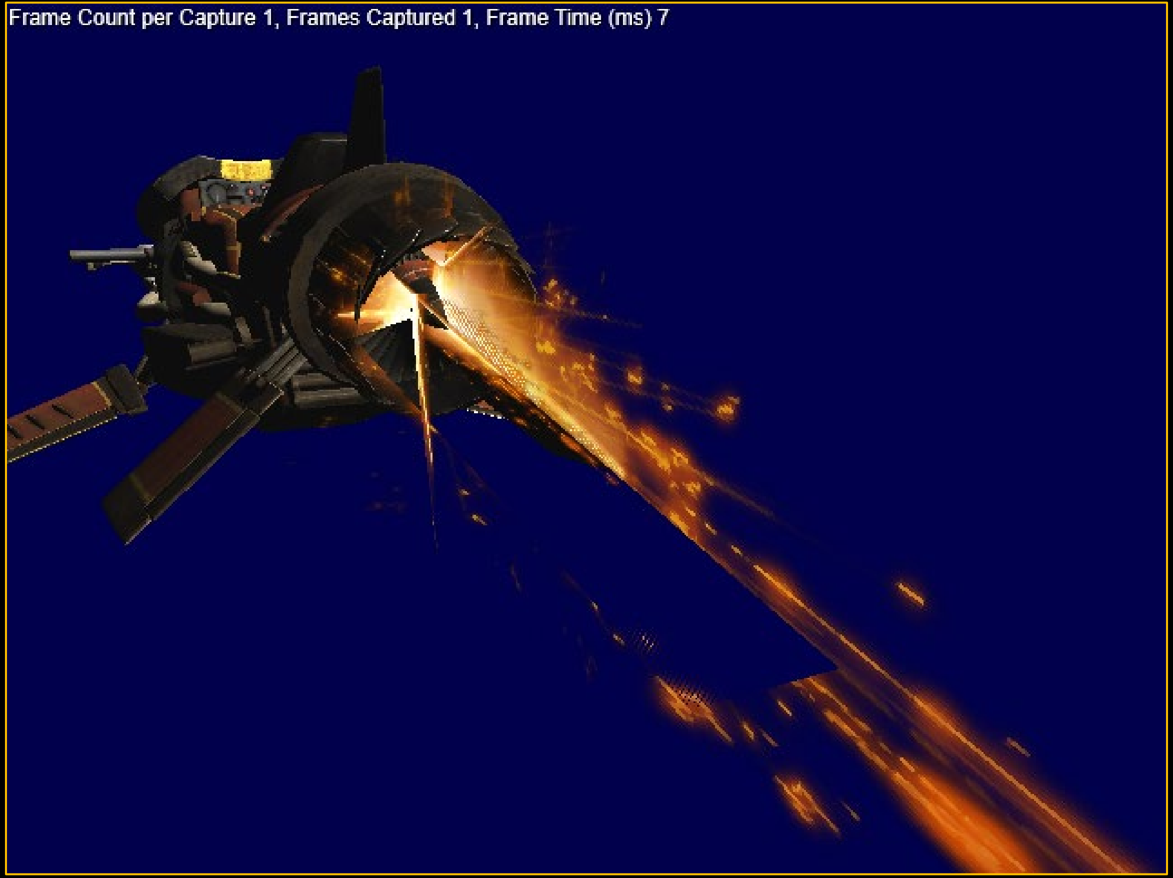


DirectX: Partial Coverage



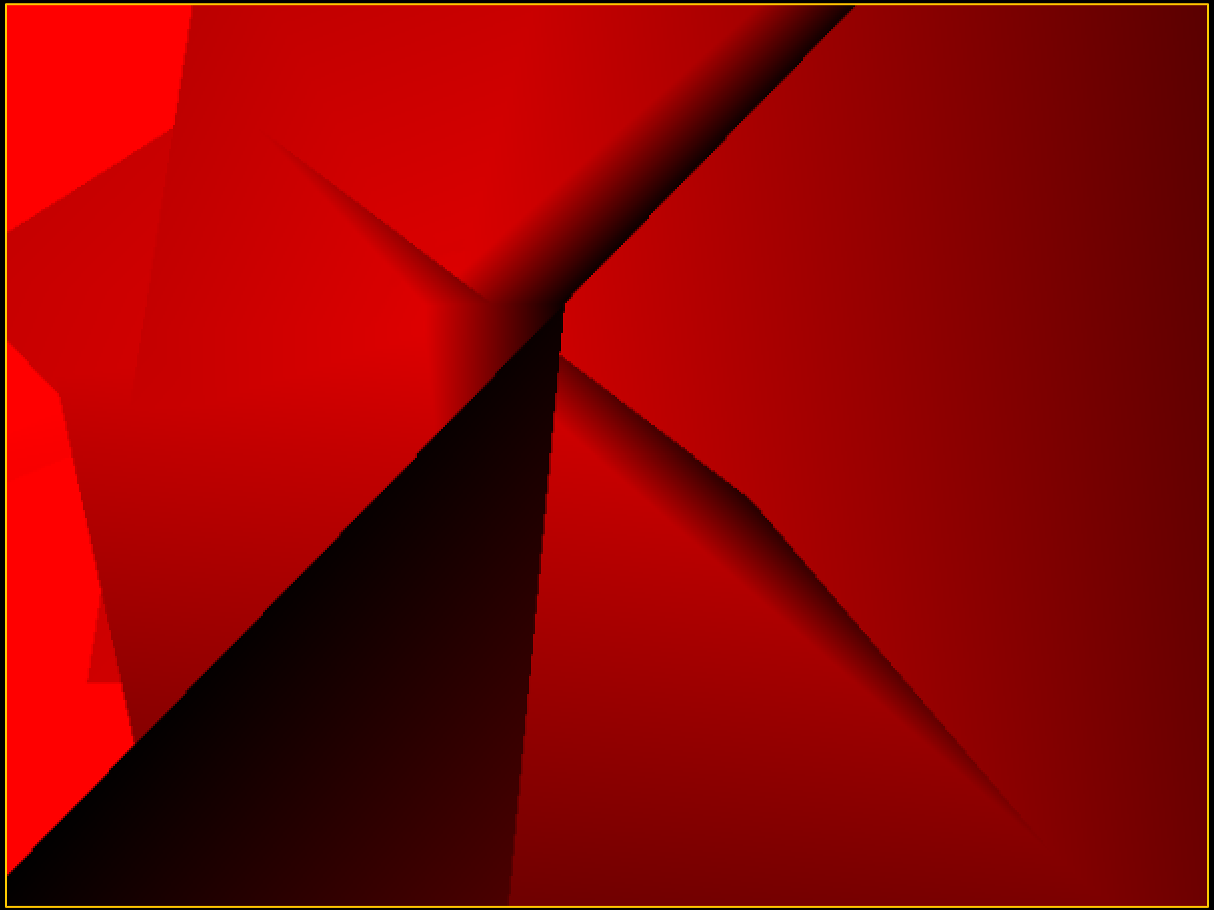
DirectX: Partial Coverage

- We still have one problem. Can you guess what causes this issue? ☺


Frame Count per Capture 1, Frames Captured 1, Frame Time (ms) 7

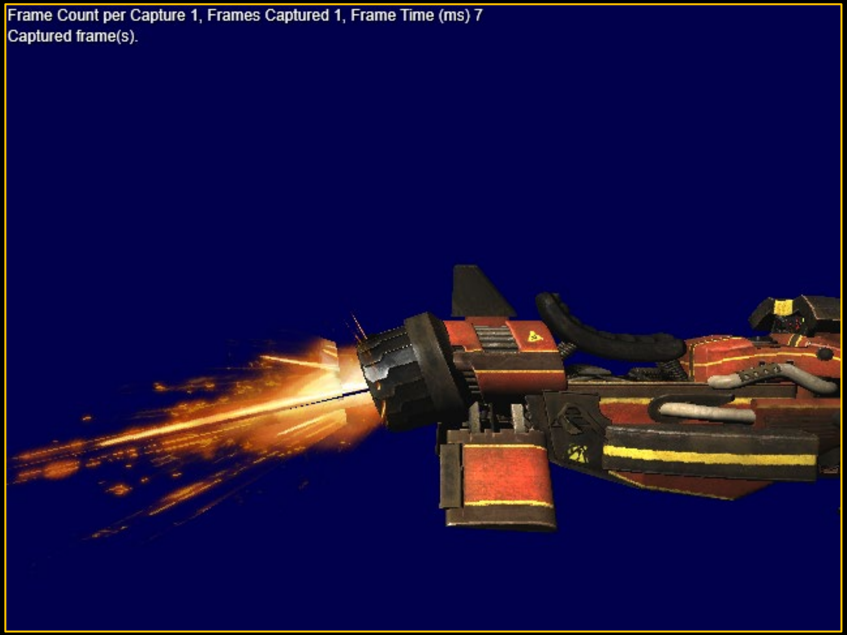
DirectX: Partial Coverage

- When we look at the depth buffer, we see this ☹

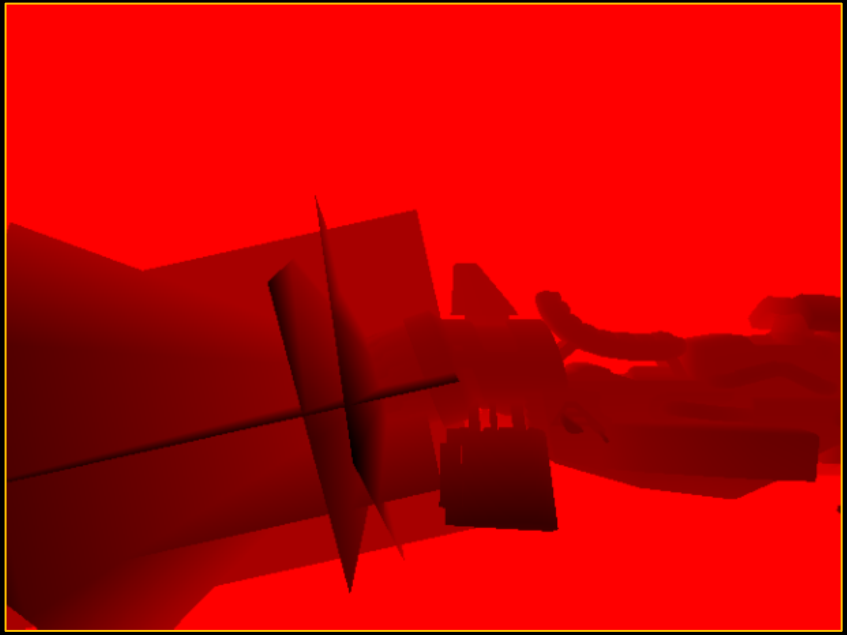


DirectX: Partial Coverage

- The weird transparent gaps are caused by the pixel failing the depth test! Because we are not in control of the order of rendering in the draw call itself (well we can, if we sort the planes in 3DS Max for <u>one</u> particular view), some planes occlude pixels that should be visible.
- The depth buffer doesn't store alpha values. So, it is not aware of the transparent pixels of the planes. How can we fix this?

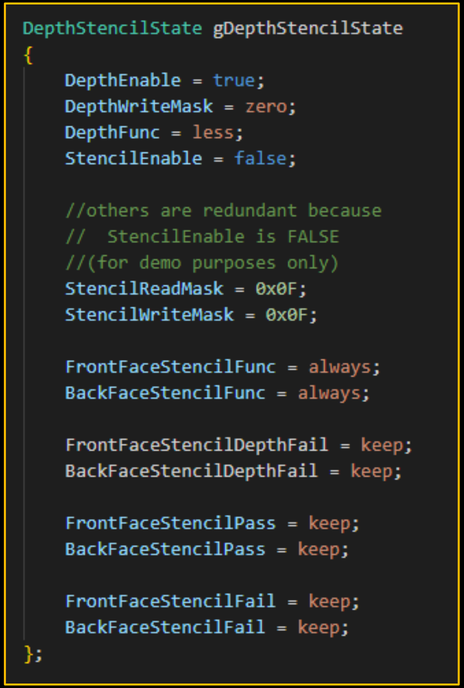


Frame Count per Capture 1, Frames Captured 1, Frame Time (ms) 7
Captured frame(s).

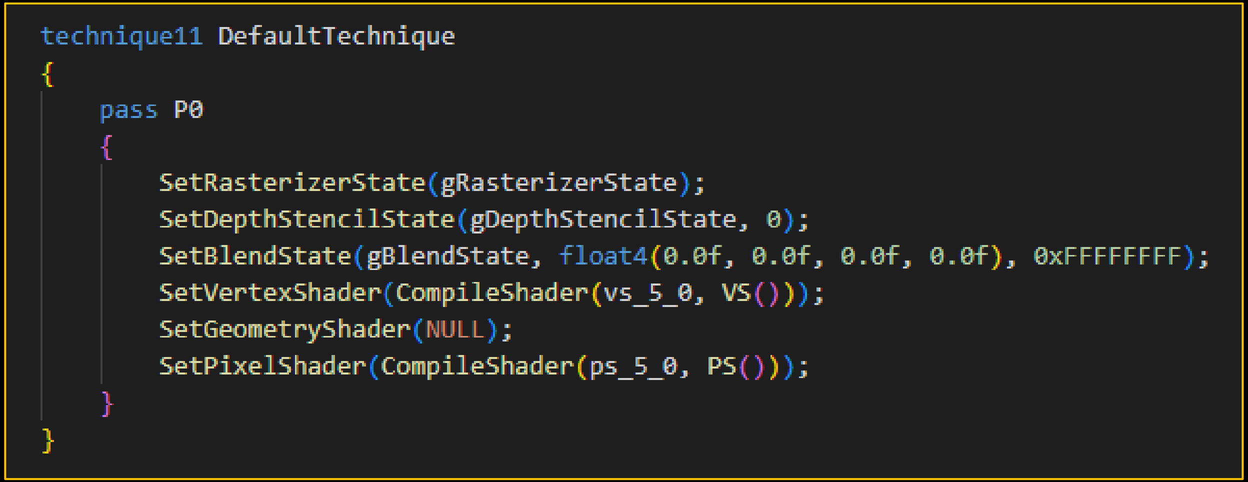


DirectX: Partial Coverage

- Again, just as with the rasterizer state and blend state, there is another state called the **depth stencil state**. Using a depth stencil state, we can say what needs or doesn't need to happen with the depth and stencil buffer.
- In our case we want to do an actual **depth test** (checking against all the other geometry in our scene) but we **don't want to write to the depth buffer**!



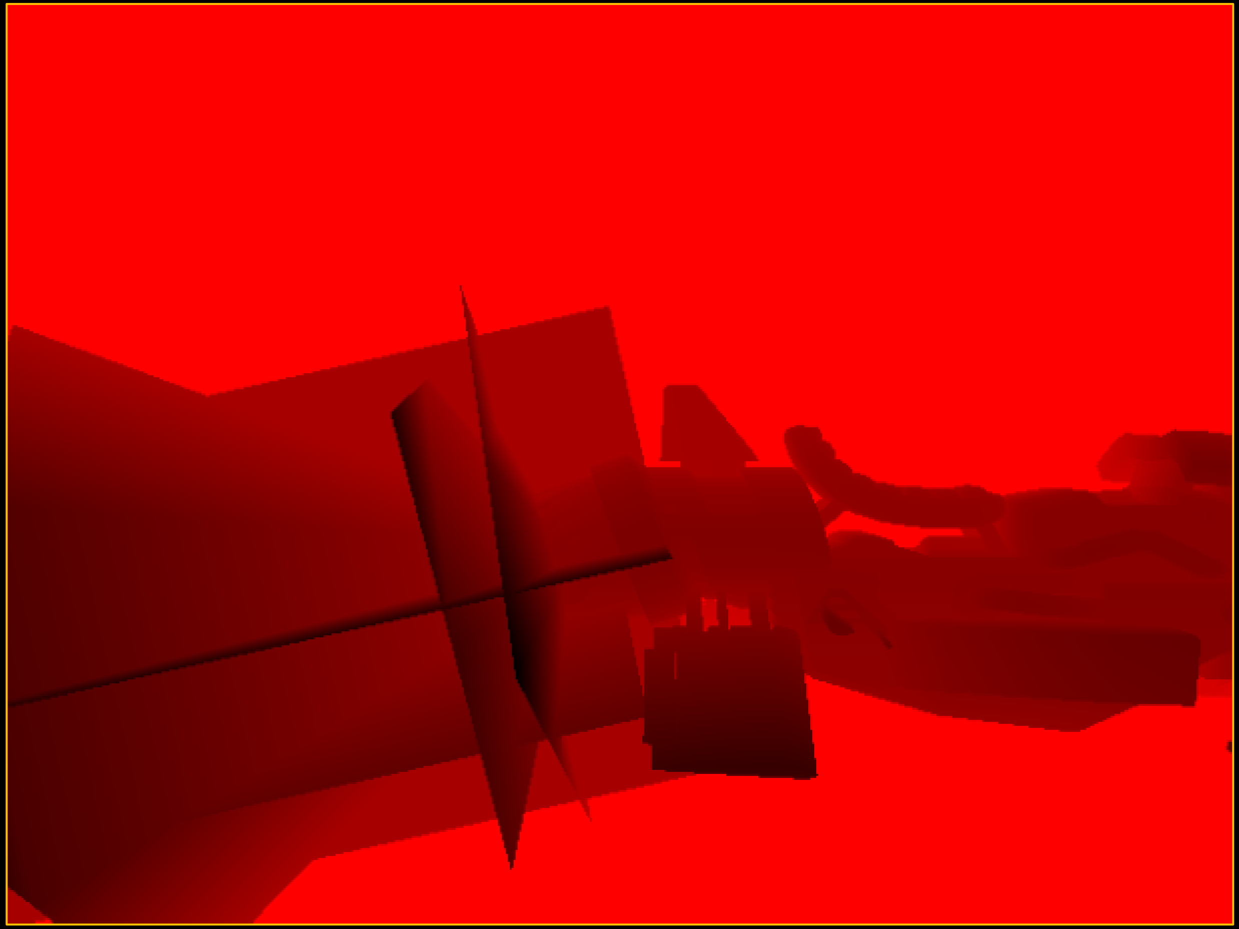
```
DepthStencilState gDepthStencilState
{
    DepthEnable = true;
    DepthWriteMask = zero;
    DepthFunc = less;
    StencilEnable = false;

    //others are redundant because
    //  StencilEnable is FALSE
    //(for demo purposes only)
    StencilReadMask = 0x0F;
    StencilWriteMask = 0x0F;

    FrontFaceStencilFunc = always;
    BackFaceStencilFunc = always;

    FrontFaceStencilDepthFail = keep;
    BackFaceStencilDepthFail = keep;

    FrontFaceStencilPass = keep;
    BackFaceStencilPass = keep;

    FrontFaceStencilFail = keep;
    BackFaceStencilFail = keep;
};
```

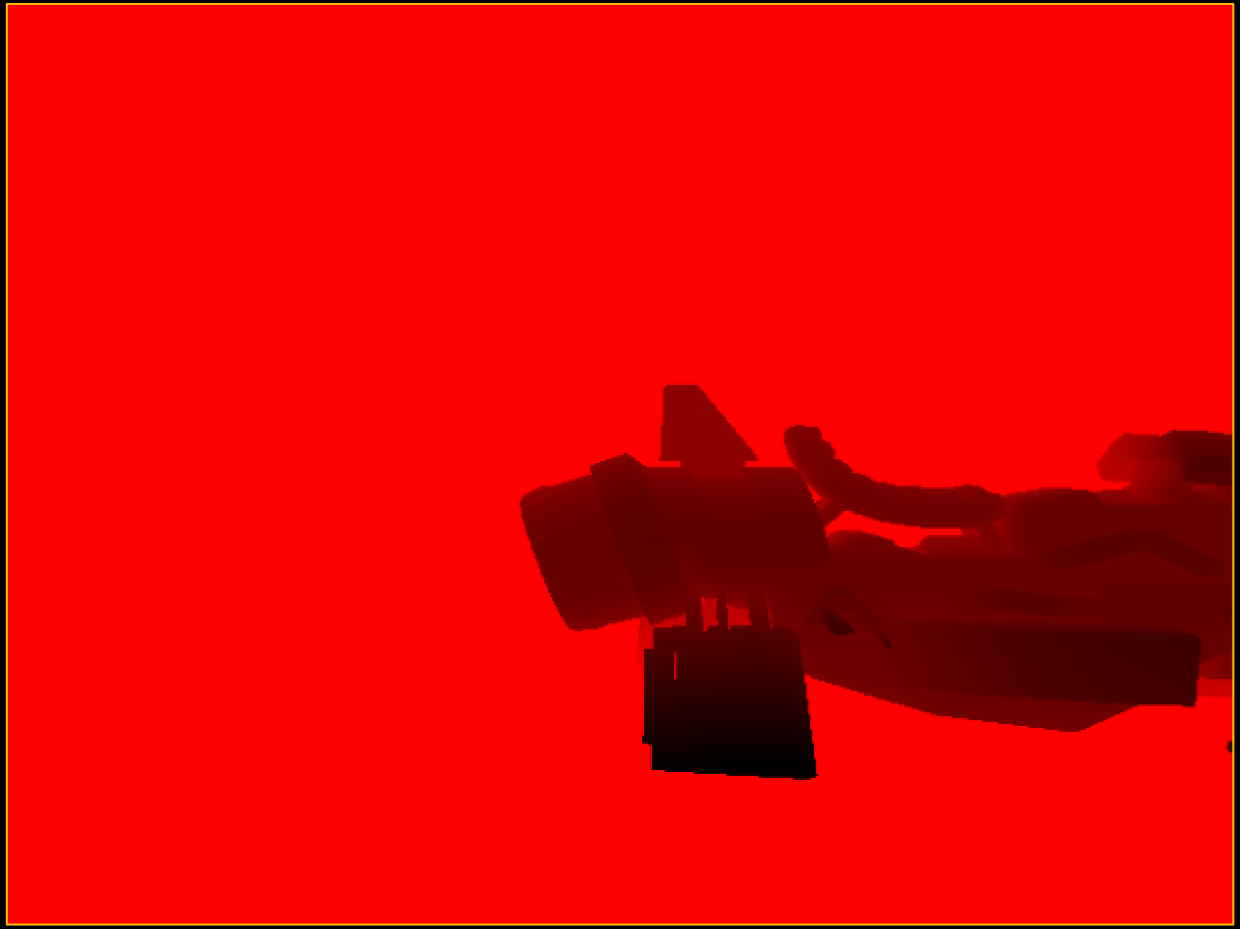


```
technique11 DefaultTechnique
{
    pass P0
    {
        SetRasterizerState(gRasterizerState);
        SetDepthStencilState(gDepthStencilState, 0);
        SetBlendState(gBlendState, float4(0.0f, 0.0f, 0.0f, 0.0f), 0xFFFFFFFF);
        SetVertexShader(CompileShader(vs_5_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, PS()));
    }
}
```

https://docs.microsoft.com/en-us/windows/win32/api/d3d11/ns-d3d11-d3d11_depth_stencil_desc

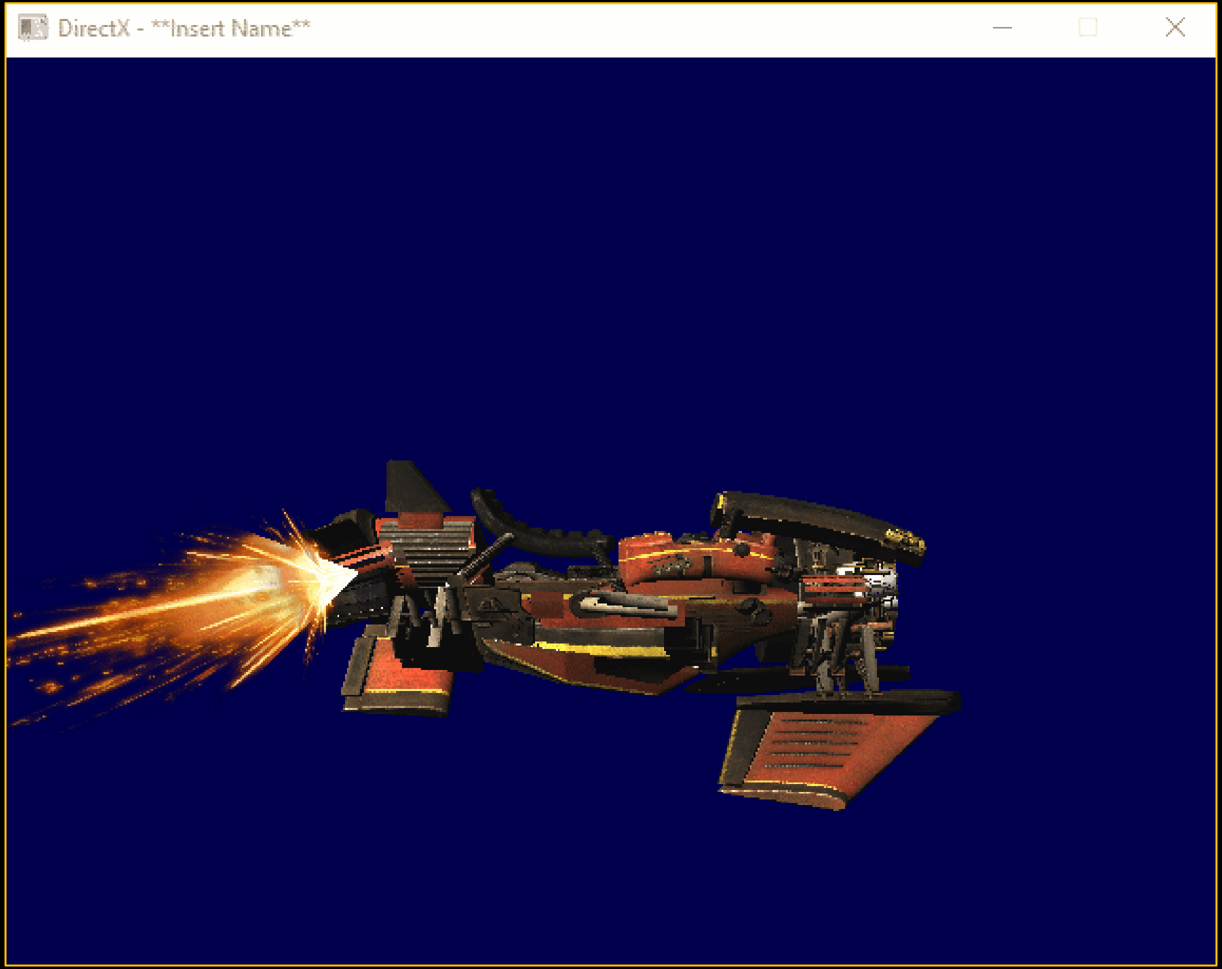
DirectX: Partial Coverage





DirectX: Partial Coverage

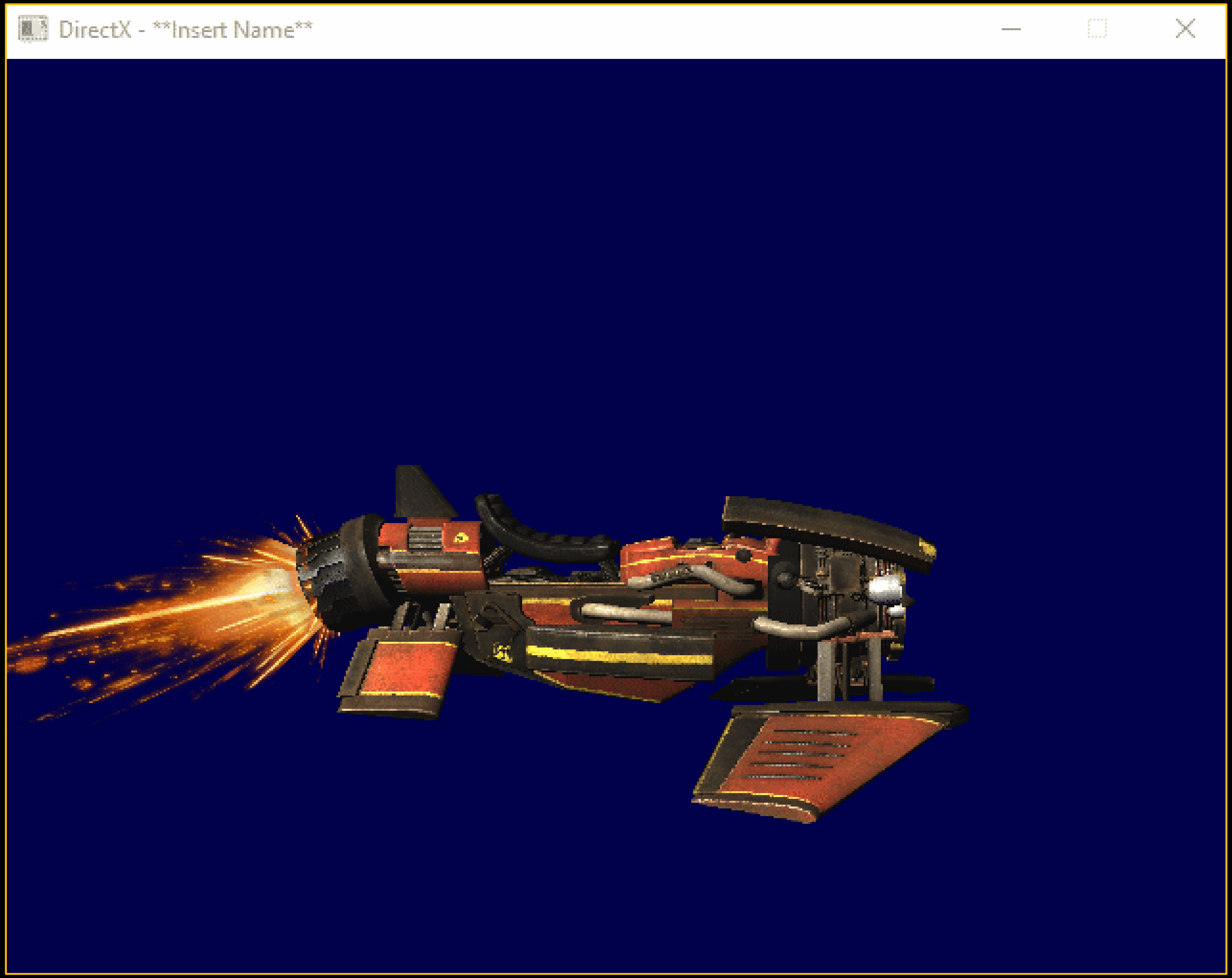
DirectX: Partial Coverage



DirectX: Partial Coverage

- Ok, now our mesh is broken again. Why?
- If you remember from GameTech, the memory on a GPU is persistent, by which we mean, if you don't overwrite a value, the previous value stays in memory. In this case, if we don't change one of the states, our next drawcall will use the same states.
- To fix this, make sure your other shader has a correct rasterizer, blend and depth stencil state in its technique! ☺
- "Fixing" our vehicle shader will result in the following correct end result!

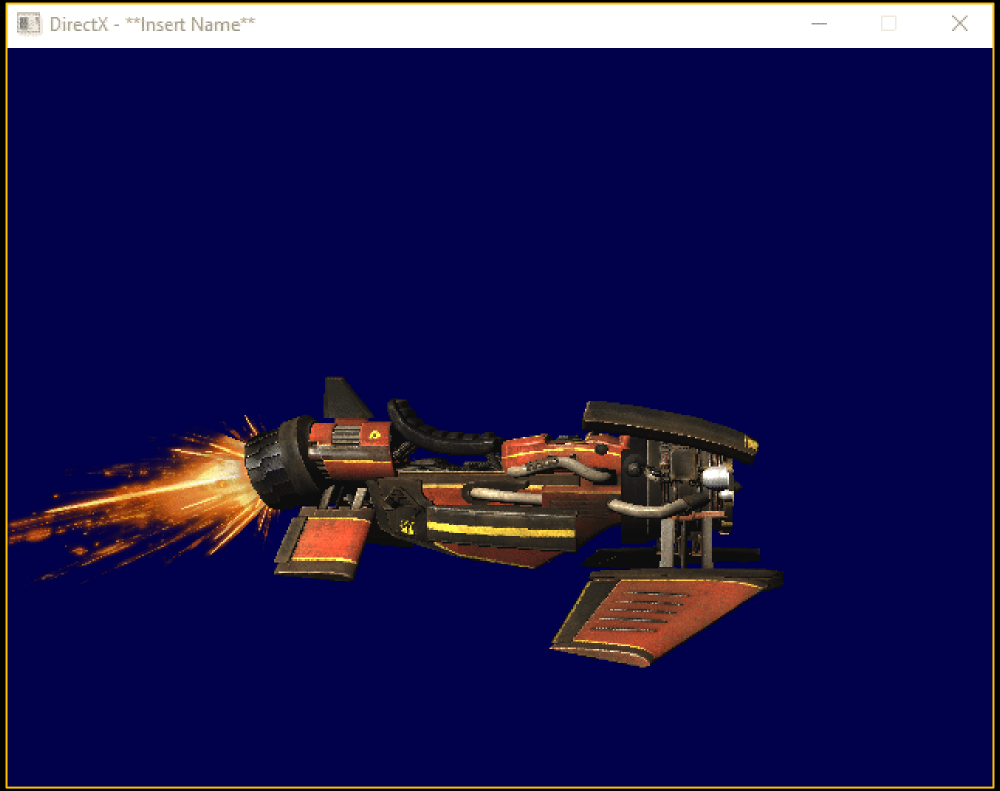
DirectX: Partial Coverage



DirectX: Partial Coverage

- Interestingly enough, I can still break it if I first render the combustion effect and then the vehicle. Why?

→ We still are order dependent! But there is no easy way to fix this ☹



The End

- That’s it folks! It’s time to complete your DirectX project and start your exam project.
- For the exam:
  - Finish all your labs first, then make a new clean project on the perforce repository.
  - Stay calm and use your brains. You are all smart enough! ☺
  - Good night rest is important! Make sure you also sleep before the theory exam. Oh yes, learn for the theory exam!

- Also, twelve weeks ago you’ve probably never done anything related to graphics programming. And now, you’ve written your own:
  - Software Ray Tracer with Direct Lighting and PBR.
  - Software Rasterizer that mimics DirectX, supporting 3D meshes and different shading techniques.
  - Used DirectX 11 to hardware accelerate the rasterization progress. By doing this you’ve also written your first shaders!
- For this…

I SALUTE YOU!

GOOD LUCK!