



DirectX 10 for Techies

Nick Thibieroz

European Developer Relations

MrT@ati.com



11-13 JULY 2006

DEVELOP
IN BRIGHTON

GAME DEVELOPERS
CONFERENCE



Introduction

- DirectX 10 Basics
- DXGI Formats
- Shader Model 4.0
- Geometry Shader
- Stream out
- Multisampling



DirectX10 Basics



DirectX 10 Overview – Timeframe

- DirectX 10 requires Vista OS
 - ...and DX10-capable graphics card!
- DirectX 10 release alongside Vista
 - Currently planned for Q1 2007
- Beta program available for Vista and DX10
 - Register with Microsoft (connect.microsoft.com)
- DX10 SDK publicly available
 - With full REF implementation



DirectX 10 Overview – Design goals

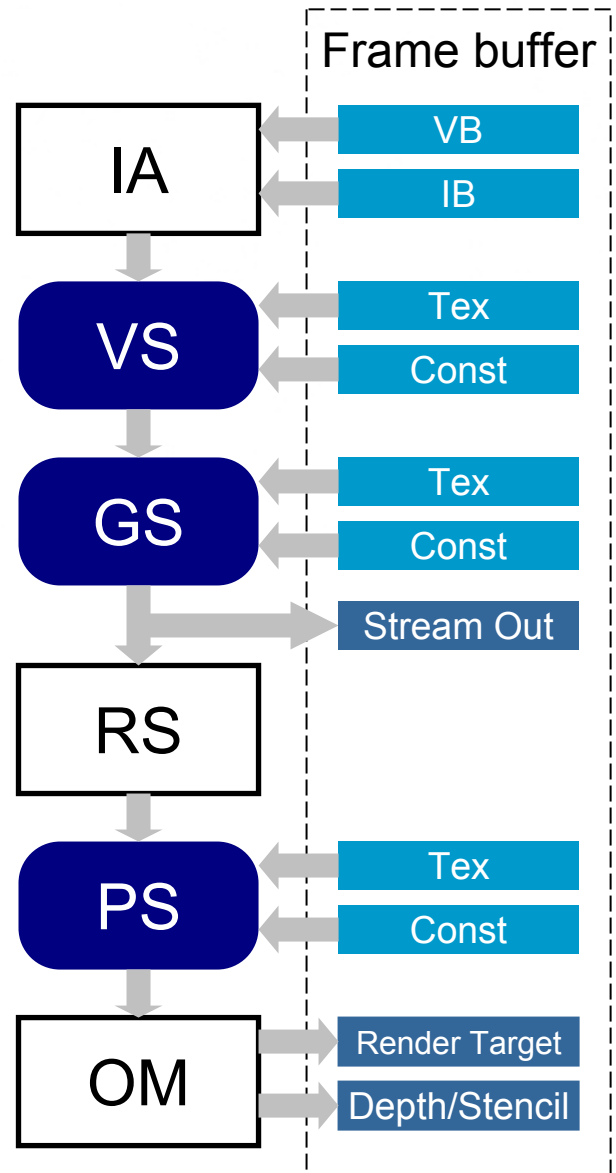
- Thin runtime
 - No legacy code
 - No fixed function
- New driver model
 - User-mode driver
 - Resource virtualization
- Less CPU dependency on API calls
 - Validation done at creation time
 - State objects
 - Constant buffers
- Common feature set
 - No more caps! (well...)
 - Main difference between DX10 implementations will be performance

Microsoft®
DIRECTX®



Direct3D 10 Pipeline

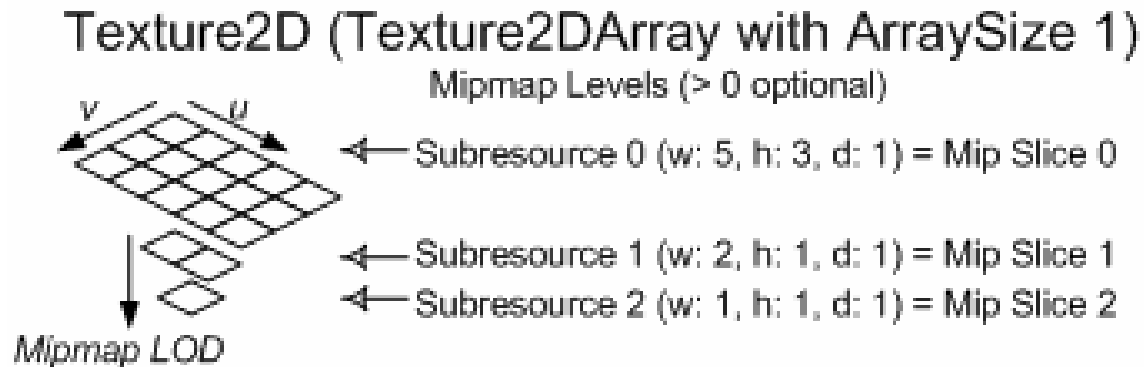
- New pipeline in Direct3D 10
- IA = Input Assembler
- RS = Rasterizer Stage
- OM = Output Merger





Resource Views

- All resources are memory arrays
- A resource view defines how a resource is accessed
 - Resources are not bound directly to the pipeline; views are bound instead
- Allows reinterpretation of video memory through “views”
 - Example 1: a specific MIP level can be selected as a render target
 - Example 2: 3D texture viewed as an array of 2D render target textures
 - Example 3: texture viewed as depth buffer





DXGI





DXGI Formats

- Some formats come with different suffices

`_UNORM` `_SNORM` `_UINT` `_SINT`
`_FLOAT` `_SRGB` `_TYPELESS`

- Examples:

- `DXGI_FORMAT_R8G8B8A8_UNORM`: each channel maps to [0.0, 1.0]
- `DXGI_FORMAT_R8G8B8A8_SNORM`: each channel maps to [-1.0, 1.0]
- `DXGI_FORMAT_R8G8B8A8_UINT`: each channel maps to [0, 255]
- `DXGI_FORMAT_R8G8B8A8_SINT`: each channel maps to [-128, 127]
- `DXGI_FORMAT_R8G8B8A8_UNORM_SRGB`: gamma-corrected read
- `DXGI_FORMAT_R8G8B8A8_TYPELESS`: ?

- Cheap HDR formats

- `DXGI_FORMAT_R9G9B9E5_SHAREDEXP` &
`DXGI_FORMAT_R11G11B10_FLOAT`



DXGI Compressed Formats

- DXGI_FORMAT_BC1_TYPELESS / _UNORM / _SRGB
 - For opaque or 1-bit alpha textures
 - 4 bits per pixel compression (DXT1)
- DXGI_FORMAT_BC2_TYPELESS / _UNORM / _SRGB
 - For translucent textures – explicit alpha
 - 8 bits per pixel compression (DXT3)
- DXGI_FORMAT_BC3_TYPELESS / _UNORM / _SRGB
 - For translucent textures – interpolated alpha
 - 8 bits per pixel compression (DXT5)
- DXGI_FORMAT_BC4_TYPELESS / _UNORM / _SNORM
 - Single channel textures (signed or unsigned)
 - 4 bits per pixel compression (ATI1N)
- DXGI_FORMAT_BC5_TYPELESS / _UNORM / _SNORM
 - Dual channel textures (signed or unsigned)
 - 8 bits per pixel compression (3DC/ATI2N)



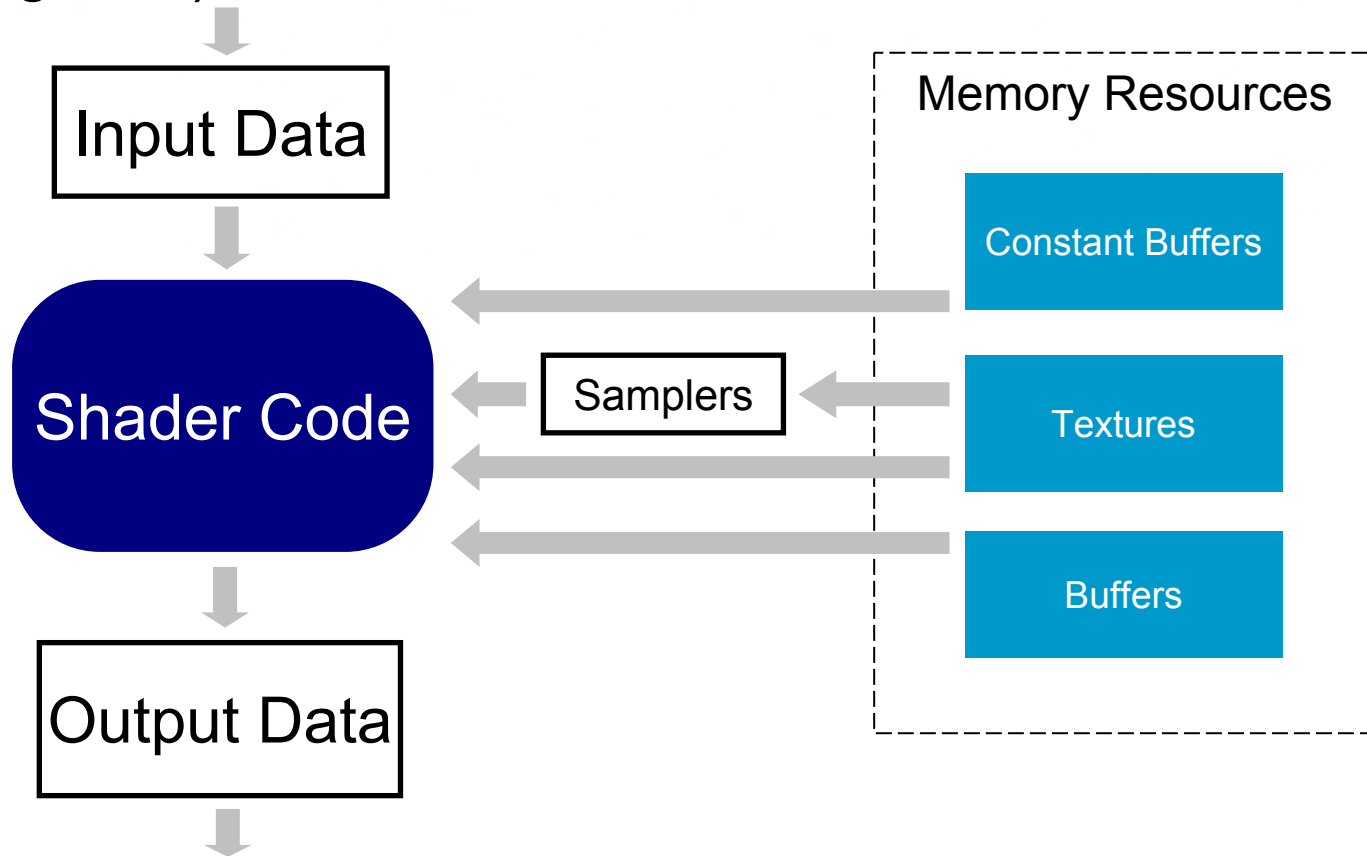
Shader Model 4.0





Shader Model 4.0

- Unified API shader code for VS, GS and PS
- Only HLSL supported (no assembly shaders, but can debug asm)





Shader Model 4.0 HLSL – Basics

- Load-time or offline compilation supported
 - Offline compilation recommended! ...especially for longer shaders
- No partial precision (“half” exists for legacy purposes only)
- Integer registers
- Full (x,y,z,w) screen coordinates can be input to pixel shader
- Textures and samplers now independent objects
- System Values
- 16 VS inputs, 16 PS inputs (32 with GS)
- Effect file system now part of core D3D10 API



Constant Buffers

- Constant updates are done per “block”
 - Allows reduced API overhead compared to DX9-style constant setting
 - One API call to update all constants within a buffer
- Constant buffers declared within a namespace

```
Cbuffer MyConstantBlock  
{  
    float4x4 matMVP;  
    float3 fLightPosition;  
}
```

- Group constants into buffers according to update frequency
- Constant buffers updated with resource loading operations
 - Map()/Unmap() or UpdateSubResource().



Shader Model 4.0 HLSL – Resource Functions

- Texture functions now templated

<T>.GetDimensions(...) <T>.Sample(...)

<T>.SampleCmp(...) <T>.SampleCmpLevelZero(...)

<T>.SampleGrad(...) <T>.SampleLevel(...)

- New function to load data from resources

<T>.Load(...)

- Template types:

Buffer

Texture1D

Texture1DArray

Texture2D

Texture2DArray

Texture2DMS

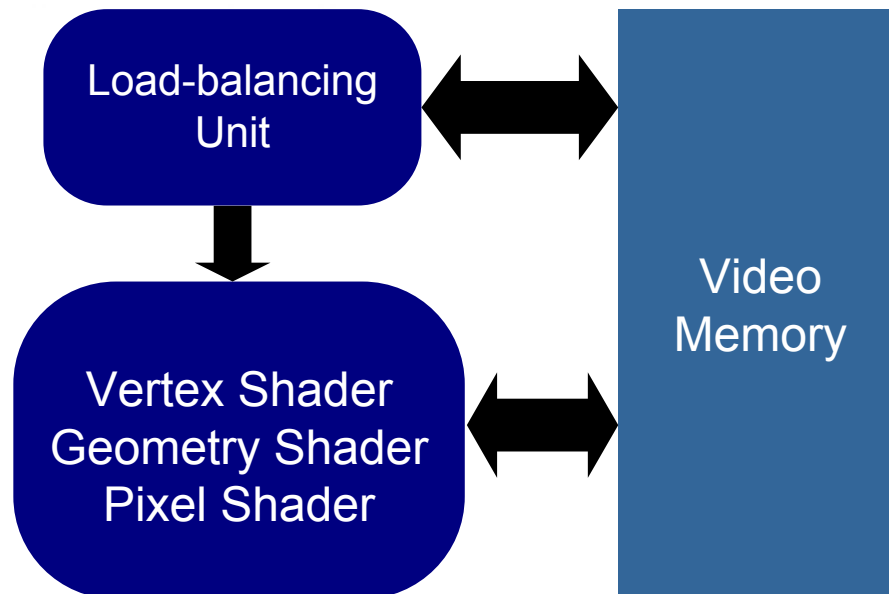
Texture2DMSArray

Texture3D



GPU Considerations: Unified Shader Architecture

- Common core for vertex/geometry/pixel shader
- A dedicated hardware unit handles load-balancing
 - 100% usage of shader units = best efficiency possible
- The concept of “free” stages is *not true anymore*





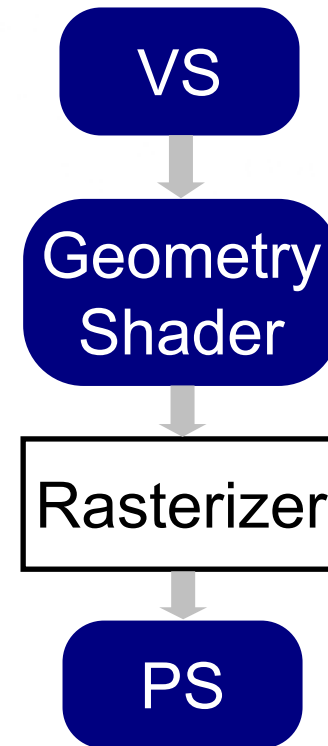
Geometry Shader





Geometry Shader

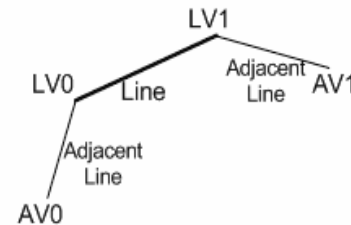
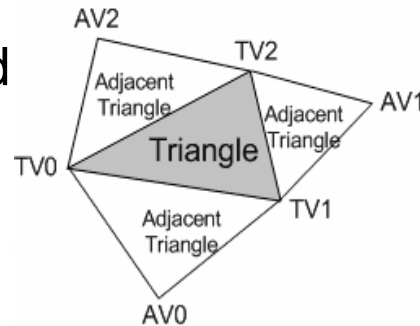
- Geometry shader has the ability to generate and kill primitives
 - Can be disabled by passing a NULL Geometry Shader
- Also allow render target index output
- Shadow volumes
- Fur/Fins generation
- Point sprites
- Render to cube map





Geometry Shader: Inputs & Outputs

- GS inputs are geometric primitives: Triangle, line, point
 - No index data
 - Adjacency data can also be requested
- Supported primitive output types
 - Triangle strip
 - Line strip
 - Point list
- Number of primitives emitted can vary
 - Maximum number of primitives must be declared
- `<T>.Append()` adds a new vertex to the output stream
- `<T>.RestartStrip()` resets strip





Geometry Shader: Syntax

```
struct GS_INPUT_VERTEX
{
    float4 Pos : SV_POSITION;
    float Radius : TEXCOORD0;
}

struct GS_OUTPUT_VERTEX
{
    float4 Pos : SV_POSITION;
    float2 Tex : TEXCOORD0;
}

[maxvertexcount(4)]
void GSmain( point GS_INPUT_VERTEX input,
            inout TriangleStream< GS_OUTPUT_VERTEX> Stream )
{
    GS_OUTPUT_VERTEX output;

    // Expand data to turn point to quad...
    for (i=0; i<4; i++)
    {
        float3 position = g_positions[i]*input.Radius;
        position = mul(position, g_mInvView) + input.Pos;
        output.Pos = mul( float4(position, 1.0), g_mWorldViewProj);
        output.Tex = g_texcoords[i];
        Stream.Append(output);
    }

    // Start a new strip
    Stream.RestartStrip();
}
```



Geometry Shader: GPU Considerations

- Geometry Shader can require *a lot* of memory accesses
 - Especially when doing amplification or decimation
- Each input element can generate up to 1024 float values
 - This may require the output of results to video memory
- Use ALU instructions to cover latency
 - This means longer programs
- GPU process elements in parallel
 - Varying number of outputs can cause efficiency issues

- Talk to us about your GS intentions



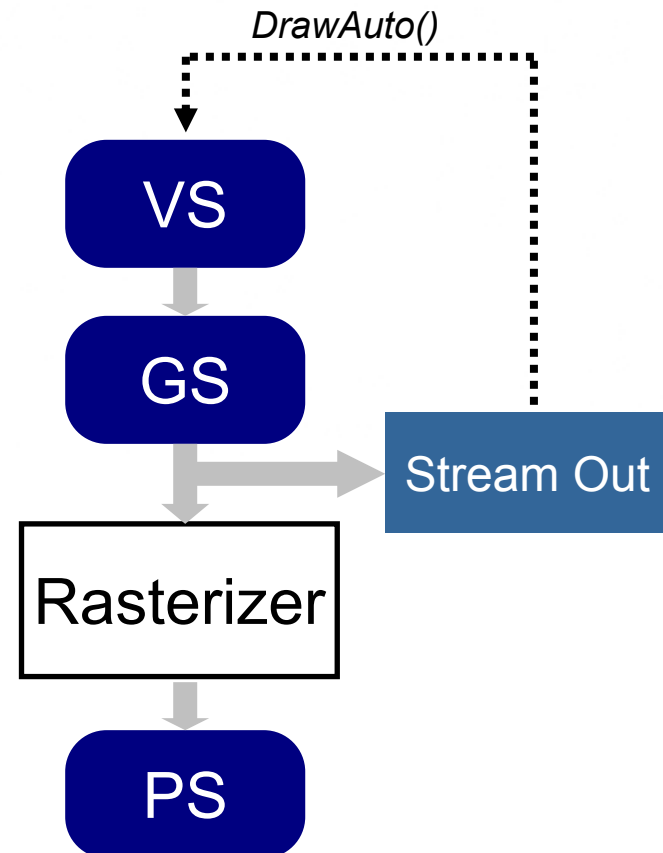
Stream Out





Stream Out

- Ability to store primitives into a buffer
- Data output from Geometry Shader
 - Or Vertex Shader if GS is NULL
- Rest of the pipeline will still execute
 - Unless disabled e.g. color writes off
- Topologies converted to lists on output
- `ID3DDevice::DrawAuto(...)` to recirculate streamed out buffer
 - No need to know how much data was written
 - Only works on a single vertex stream





Stream Out (2)

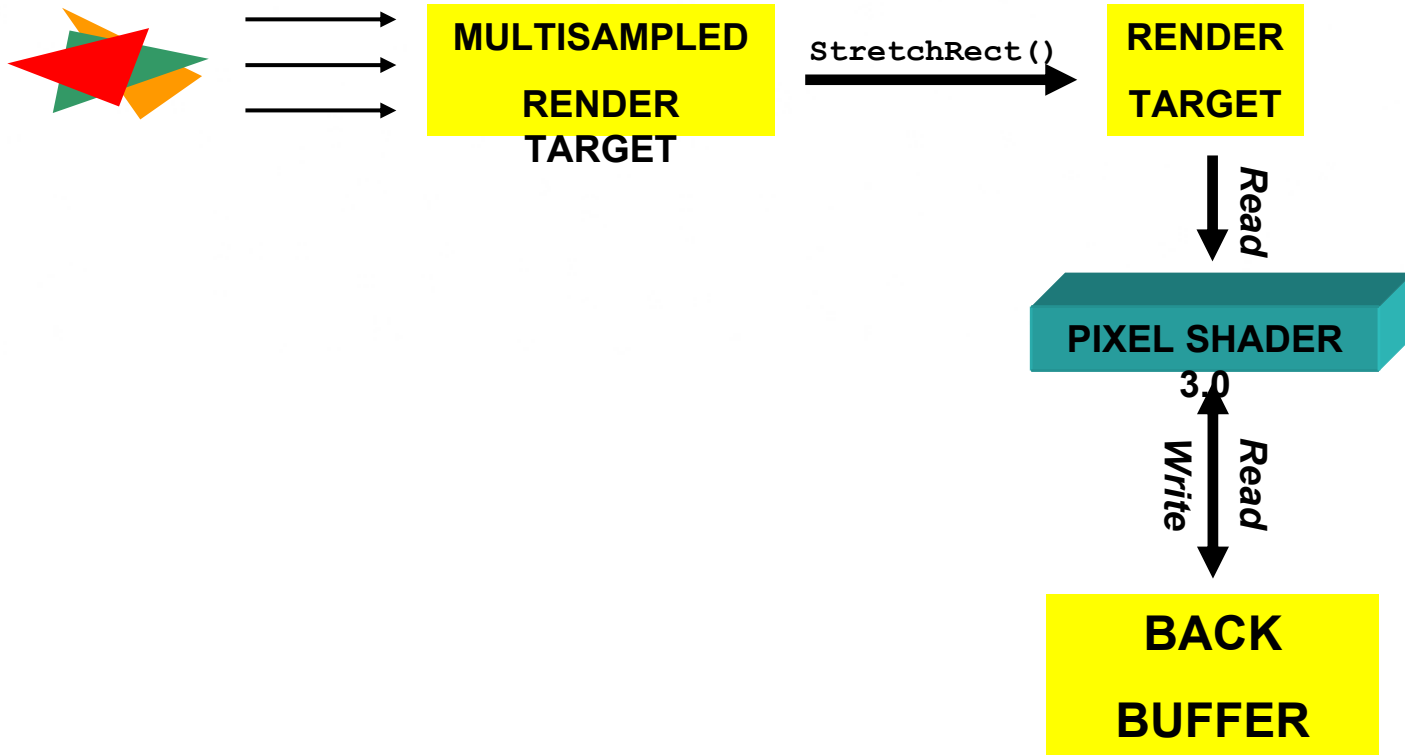
- Data can be copied to staging resource for CPU readback
 - Leverage GPU for (complex) transformations
 - Watch out for stalls!
 - Try to make the copy as late as possible...
- Multipass triangles with index buffer:
 - Render the vertex buffer as a point list
 - Index buffer can then be used in subsequent passes

A large, semi-transparent 3D wireframe model of a human eye is positioned on the left side of the slide. The eye is rendered with a grid of lines, giving it a technical or digital appearance. It is looking towards the right.

Multisampling

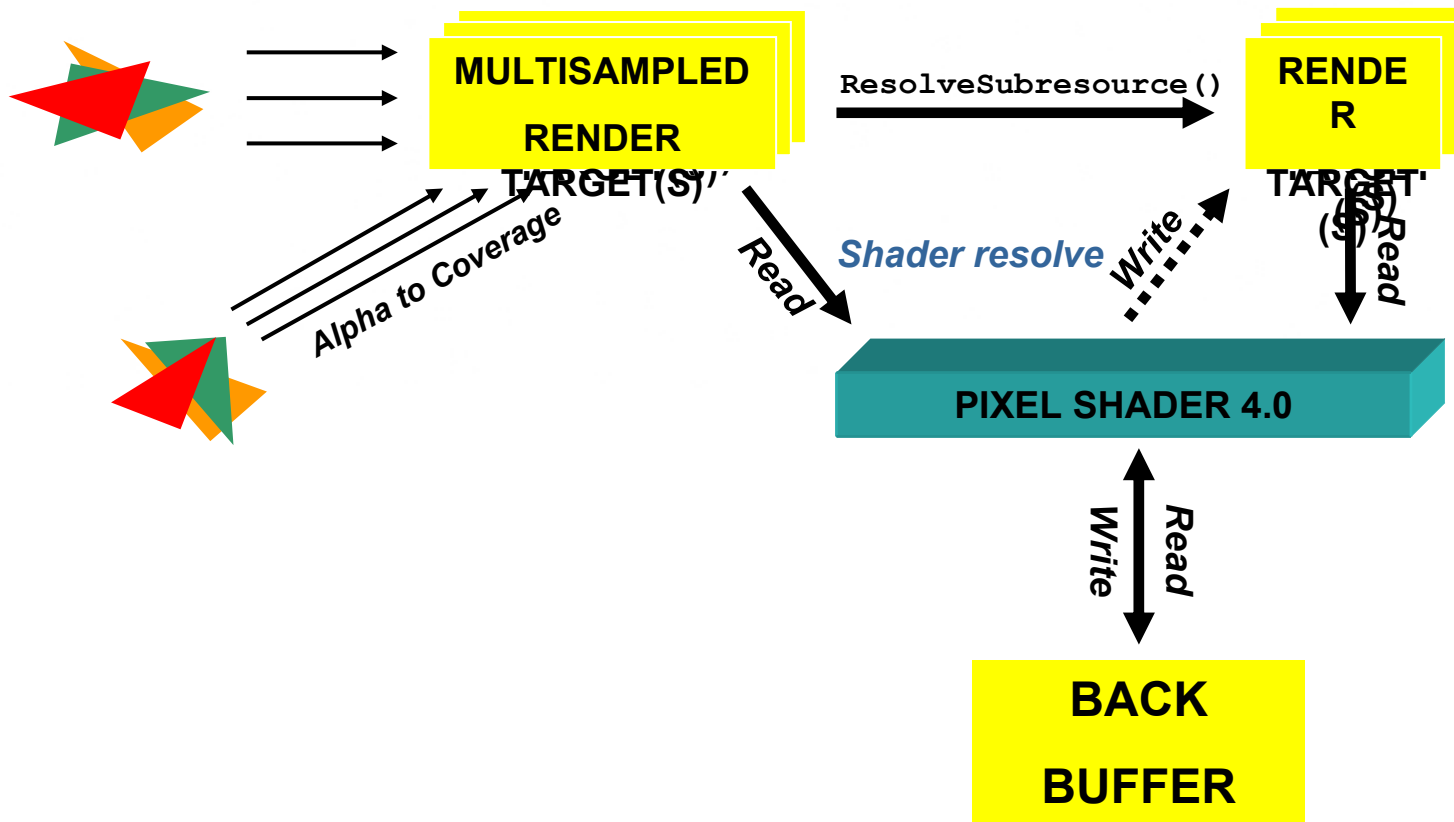


Multisampling: DX9





Multisampling: DX10





Conclusion

- Next iteration of DirectX is more than just evolution this time around
 - Awesome API design
 - Better CPU performance
 - Ubiquitous access to resources
 - Unified shader API
 - Geometry Shader
 - Data recirculation techniques
 - New multisampling features
- Learning curve may be higher than with previous iterations
 - But results will be worth it
- Talk to us about how you want to use the API



Nick Thibieroz
MrT@ati.com